FedCM: Fine-grained Kernel Scheduling and Management to Improve GPU Sharing

Xianwei Zhang[+]
CSE, Sun Yat-sen University
Guangzhou, China
zhangxw79@mail.sysu.edu.cn

Xuanteng Huang[+], Tianyu Guo
CSE, Sun Yat-sen University
Guangzhou, China
{huangxt57, guoty9}@mail2.sysu.edu.cn

Nong Xiao[*]
CSE, Sun Yat-sen University
Guangzhou, China
xiaon6@mail.sysu.edu.cn

***Abstract*—GPU has become the *de facto* device to accelerate widespread machine learning and general purpose computing applications. Sharing a GPU is increasingly important to achieve higher throughput and better resource utilization. However, existing GPU sharing adopts either coarse-grained collocation approaches or interference-unaware spatial partition strategies that produce suboptimal results. In this paper, we propose FedCM, a kernel-level, collocation-based GPU sharing scheme to establish a federated use of compute and on-chip memory resources. FedCM evaluates the collocation potential of ready kernels and dispatches them in a way to maximize system throughput. During collocated execution, FedCM adopts kernel-wise management to arbitrate cache usage via customizing cache policies. The evaluation of our implementation on the off-the-shelf GPUs demonstrates that FedCM improves the overall throughput by 48.3% and 17.4%, compared to standard sharing baseline and prior state-of-the-art, respectively.**

***Index Terms*—GPU, Kernel Sharing, Cache, ROCm**

I. Introduction

Featuring massive parallelism and high energy efficiency, GPUs are now widely deployed to accelerate a broad spectrum of applications, including machine learning and general computing in data centers, edge and embedding circumstances [1]. To meet the ever-increasing demands, GPUs keep adding richful computation and memory resources, which can be extremely challenging to be fully exploited by monolithic tasks [2]–[4]. For better utilizing the ample resources, GPU vendors and research communities have introduced various concurrent kernel execution mechanisms.

Figure 1 depicts different GPU sharing schemes, where vendors provide (a) *hardware isolation* for interference-avoid execution environment for each concurrent task. However, we can observe GPU under-utilization inside each isolated instance as the kernel-level resource fluctuation cannot be satisfied by task-level partition. (b) *temporal sharing* scheme [5] manages concurrent tasks by assigning them to time slices where each task owns the exclusive GPU access. Similarly, one single task may not occupy the entire GPU within one time slice, hence Orion [6] and REEF [7] attempt to schedule multiple tasks simultaneously, i.e., *Task1* + *Task2* in Fig. 1(b). Nevertheless, it still treats the GPU as an indispensable device and only achieves sub-optimal utilization without dedicated task consolidation in the spatial dimension. Recently, CUDA/HIP releases support the fine-grained (c) *spatial partition* across SMs (Streaming Multiprocessor), which allows users to specify the execution units for each GPU context or stream [8]. Based on this, KRISP [9] and BLESS [10] propose fine-grained resource allocation strategy to dynamically update the SM affinity for each kernel. But it ignores the potential compute and cache resource contention between consolidated tasks (competing overlap in Fig. 1(c)), thus leading to performance downgrade.

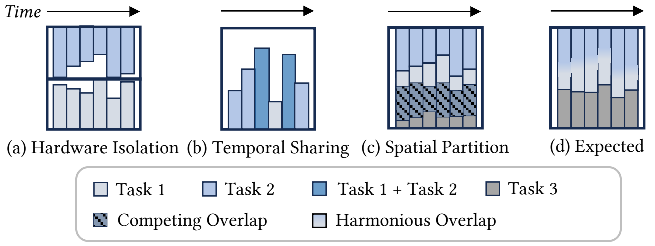


Fig. 1. Comparison between FedCM and other GPU sharing schemes, where Task1 and Task2 are harmonious with complementary resource demands while Task1 and Task3 are competitive.

Theoretically, GPU sharing could further benefit from **fine-grained collocation-based strategy**, i.e., having multiple harmonious kernels allocated to a subset of GPU resources, which achieves better performance as each of them demands complementary hardware resources such that there is no severe interference. Fig. 1(d) shows the expected harmonious sharing where *Task 1* and *Task 2* are overlapped, resulting in less on-chip resource contention and thus improved performance. To achieve such resource-aware kernel sharing, the challenge lies in identifying and consolidating harmonious kernels at shared compute units to accomplish flexible scheduling and avoid large overhead. In addition to compute resources, the memory resources, in particular, on-chip caches, become increasingly important [11]–[13] and thus should be carefully managed for collocated kernels. For optimized GPU sharing, collocated kernels prefer customized on-chip cache management, enabled by architectural support of modern GPUs, to mitigate contentions. Thus, throughput-oriented GPU sharing demands dedicated kernel-level management for both compute and cache resources.

In this work, we propose FedCM, a fine-grained kernel-

[+] These authors contributed equally to this work.

[*] Corresponding author.

level collocation-based GPU resource scheduler, to improve the system throughput of GPU sharing. To accomplish harmonious sharing, we consider both the compute and cache resources among concurrent kernels to avoid severe contention. FedCM leverages the resource usage profiles at the kernel level and dynamically adjusts the compute resource allocation, and selectively bypasses on-chip cache resources for kernel memory access instructions based on its cache usage sensitivity. We summarize our contributions as follows.

- We characterize the compute and cache memory contention of GPU sharing, and observe significant improvement opportunities for kernel-level collocation.
- We propose FedCM, a fine grained kernel-level collocation-based resource scheduler, to maximize the system throughput of shared GPUs.
- We implement FedCM on off-the-shelf GPUs, with only moderate changes to the runtime. Evaluations show FedCM achieves considerable throughput improvement.

II. Background and Motivation

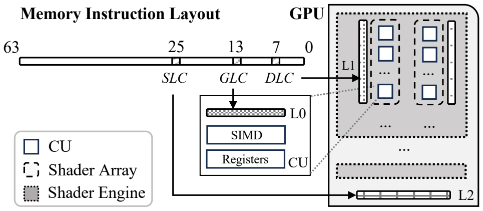
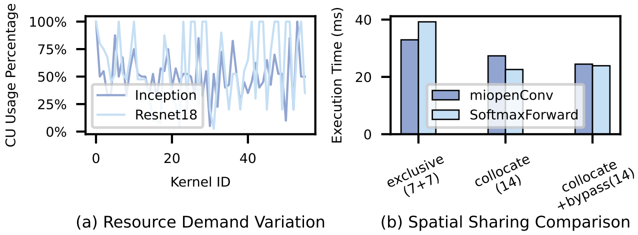


Fig. 2. GPU compute components and cache levels organization.

A. Fine-grained Collocation for Utilization

Figure 2 illustrates the high-level architecture of a general-purpose GPU, based on the recent AMD RDNA design [1] [12], [14]. A GPU is composed of multiple compute units (CUs), also known as streaming multiprocessors (SMs). A cluster of CUs are grouped as shader arrays (SAs), and a bundle of SAs are housed by an shader engine (SE). As the basic computing unit, each CU consists of dozens of SIMD units or computing cores, on-chip memory, and other dedicated components. For the memory hierarchy, each CU contains private scalar and vector general-purpose registers (SGPRs and VGPRs), L0 data cache, and shared memory (SMEM or LDS). CUs within an SA share an L1, and all CUs of the GPU share the globally accessed L2 cache, which in turn connects to main memory, which can be GDDR or HBM.

GPU application is usually composed of multiple kernels that exhibit different resource demands and execution behaviors. We run two benchmark workloads `Inception` and `ResNet18` on an AMD Radeon 6900XT GPU and summarize their resource usages in Fig. 3(a). The numbers of CUs that each kernel demands to maximize the performance vary dramatically across different programs and across different kernels within one program. About 40% of kernels in both programs desire fewer than half of GPU compute resources. In addition, the demands fluctuate significantly, e.g., from 100% plunged to 20% for two consecutive kernels. Existing coarse-grained GPU sharing schemes [6], [7], [15]–[17] determine the resource provision for a task based on the maximal resource demand of all its kernels, failing to capture the intra-task, i.e., the kernel level, demand fluctuations, and therefore under-utilize the GPUs. As such, fine-grained resource partition and collocation strategies (Fig. 1) can be exploited to provision optimal resources for tasks and adjust as they progress to improve the system performance and utilization.



(a) Resource Demand Variation (b) Spatial Sharing Comparison

Fig. 3. (a) Varietal CU demands fluctuate across kernels in `Inception` and `ResNet18`. (b) Performance for different compute and cache resource allocations for `miopenConv` and `SoftmaxForward` from DL workloads.

Figure 3(b) exemplifies the collocation benefits for two kernels `miopenConv` and `SoftmaxForward` under different scheduling policies: *exclusive(7+7)* dispatches the two kernels onto two disjoint CU groups with each group containing 7 CUs, while *collocate(14)* collocates both kernels into a group of 14 CUs. Although with same amount of CUs, *collocate(14)* the achieves better performance as 38.7% shorten time is observed for `SoftmaxForward`. Therefore, the concurrent execution and resource utilization can be further improved by identifying and dispatching *harmonious* GPU kernels (Fig. 1(d)) compared to isolated spatial partition (Fig. 1(c)).

B. Managing Cache to Avoid Contention

With the increasing demands for high performance, modern GPUs continue to add CUs and memory resources. In particular, on-chip caches are significantly enriched to support massive thread parallelism. For example, NVIDIA has aggressively expanded the L2 cache from less than 6MB to more than 40MB since Ampere [18]. Meanwhile, AMD GPUs have also seen significant enhancements in the cache architecture, including more levels and much larger capacity [19]. To effectively manage the cache space for divergent workloads, software controlled mechanisms have also been proposed, e.g., residency control [18] and flexible policy tuning [20]–[22]. For the latter, recent RDNA architectures integrate architectural support to add three optional annotation bits to memory instructions (SLC, GLC and DLC in Figure 2) to control the data coherency and cache access behavior [14]. The annotation bits can be customized for each GPU kernel to specify the desired management policy at each cache level.

For collocation-based GPU sharing, it may exhibit natural interference and contention on both compute and memory resources between concurrent kernels compared to the isolated spatial partition. Higher resource utilization may lead to potentially severer resource contention of GPU resources,

[1] The paper elaborates our design using AMD GPU architectures. The design principles are applicable to architectures from other vendors.

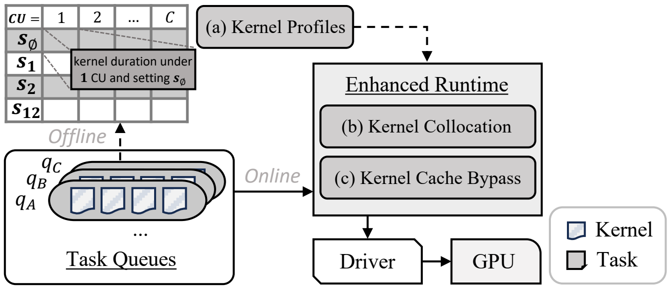


Fig. 4. FEDCM workflow overview. Dashed arrows depict offline workflows, while solid arrows present procedures in the online critical path.

in particular, the on-chip caches with limited capacity. By exploiting the recent architectural support for cache accesses, we may manage the on-chip cache sharing at fine granularity as well, which provides the flexibility for GPU sharing. In Fig. 3, the *collocate(14)+bypass* scheme indicates the design in which we command `SoftmaxForward` to bypass L1 cache. The results show that `SoftmaxForward` has no observable performance degradation even though its accesses skip L1 cache; on the other hand, `miopenConv` achieves an significant improvement due to eliminated L1 cache contention. Consequently, the overall execution time of two kernels are effectively shortened from fine-grained on-cache control. In summary, we may exploit fine-granularity on-chip cache control to achieve better GPU resource utilization when adopting the collocation based GPU sharing.

III. DESIGN

In this section, we present FEDCM, a kernel-level collocation based GPU sharing strategy for fine-grained compute units and on-chip cache regulation. Fig. 4 illustrates the overall workflow of FEDCM, which consists of three primary phases as follows. (a) FEDCM collects the task kernel performance profiles through offline profiling (Section III-A). (b) FEDCM performs interference estimation and decides the collocation for concurrent kernels (Section III-B). (c) Then FEDCM applies adaptive cache bypass on launch-ready kernels to alleviate the potentially magnified contentions caused by consolidation.

A. Kernel Profiling

Targeting the optimization of the system throughput of concurrent GPU tasks, FEDCM profiles the applications when they are scheduled to run for the first time on these servers. The kernel profiles (upper left in Fig. 4) elaborates the collected kernel statistics, i.e., the execution time for each kernel when the application uses different numbers of CUs (from 1 to the total number of CUs C) and the following four settings for controlling cache skips:

$s_\varnothing$: the baseline execution mode;
s_1: the execution that skips L1 cache;
s_2: the execution that skips L2 cache;
s_{12}: the execution that skips both L1 and L2 caches.

We also collect the kernel resource demand $d \in (0, 1]$, which is the minimal percentage of CUs such that, when running with these many CUs, its performance is within 10% degradation of the peak performance, i.e., that with all CUs in the GPU used. That's say, we can allocate $d \times C$ CUs for the kernel to achieve desired performance instead of using all the CUs. In addition, the following static on-chip resource usage for a CTA can be collected during a kernel launch from the data packet sent to underlying driver:

- *lds*: the usage of shared memory;
- *cntS*/*cntV*: the number of used scalar/vector registers;
- *thrds*: the number of threads in a CTA.

While we profile an application when running it for the first time, the profiled execution is applied only to the *critical* kernels. A kernel is considered critical if its execution time is ranked among the top 10% of the kernels when sorted by their execution time, and we investigate the impact of critical ratio in section V-D. Such a strategy helps to minimize the configuration overhead at runtime. The kernel profile, including the execution time at different settings, and the program statistics, are fed to the runtime to make kernel collocation decisions.

B. Kernel Collocation and Cache Bypass

Assume we schedule N concurrent applications on the GPU and keep ready kernels from each application in a separate ready queue. We list the notations used in our description as follows in Table I.

TABLE I
NOTATIONS FOR ELABORATION

Notation	Meaning
N	the number of concurrent tasks to collocate;
C	the number of CUs in the GPU;
$U[i, c]$	the resource usage for the kernel from queue i ($i \in [1, N]$) on the c-th CU ($c \in [1, C]$);
For each queue i, $i \in [1, N]$	
$Q[i].k$	the ready kernel from queue i
$Q[i]$.mask	the CU mask indicating what CUs are bound to its kernel
For each kernel	
$k.T[s, c]$	the performance profile of kernel k, indicating the duration of k on cache setting s and c CUs (Figure 4)
$k.d$	the percentage of CUs with desired trade-off for kernel k

The estimation of system resource usage. To collocate ready kernels at runtime, we need to dynamically estimate the system resource usage of each kernel, in particular, under collocation. In this work, the collocation decisions are made at the CU granularity, that is, two CUs may execute thread blocks from different but overlapped subsets of kernels. The resource usage of a kernel may not be the same on all of its assigned CUs. Therefore, we estimate the resource usage on each CU and get the total usage by summarizing the usage from each individual CU.

By hypothetically collocating a kernel to one CU, we calculate the resources that this kernel can get on this CU using the static resource demands of kernels that were already collocated to that CU. For example, if a kernel k_i from $Q[i]$ demands 3MB shared memory, while the two kernels running

Algorithm 1 Kernel Collocation Algorithm

```
Input:
    $Q[i], i \in [1, N]$: concurrent task queues
    $r$: the ready kernel is from $Q[r]$
Output:
    $Q[t]$: the target task queue that will be collocated with
    task queue $q_r$

function TaskCollocation($Q, r$)
    for $i \in [1, N]$ do
        if $i = r$ then
            $m_i \leftarrow Q[r]$.mask                          ▷ Reuse
        else
            $m_i \leftarrow Q[i]$.mask $\vee$ $Q[r]$.mask          ▷ Merge
        if $u(r, m_i) \notin [0.9, 1.1] \times Q[r].k.d$ then
            add/remove CUs accordingly
        $est[i] \leftarrow \frac{1}{u(r,m_i)} \times Q[r].k.T[s_\varnothing, c_{m_i}]$
    $t \leftarrow \mathrm{argmin}_i(est[i])$                  ▷ Shortest est. time
    return $Q[t]$
```

on the CU j ask for 4MB and 1MB, respectively. The resource regarding the shared memory for k_i can be computed as $3/(3 + 4 + 1) = 37.5\%$, i.e., the kernel may get 37.5% share of shared memory from collocating the kernel on CU j. We leverage the shares for on-chip static resources including shared memory, S/V registers, and CTA size (section III-A), respectively and then compute $U[i, j]$ as their average on the share of the system resource that a kernel from $Q[i]$ may get from collocation on CU j.

The estimation of interference. For effective scheduling, we also need to estimate the interference from collocation. Thus, the interference from collocation comes mainly from how the system resources are distributed on different CUs.

Assume a kernel from $Q[i]$ is launched with a mask m to run on c_m CUs (number of 1s in m), and the system resources that this kernel may get from collocation on these CUs are $U[i, j]$ (j is the positions of 1s in m). The *utilization factor* $u(i, m)$ for queue i is calculated as

$$u(i, m) = \frac{\sum_{j \in m} U[i, j]}{c_m} \tag{1}$$

Intuitively, the utilization factor represents the percentage of resource allocated for $Q[i]$, e.g., $u(i, m) = 50\%$ means half of GPU resources are used by kernels from $Q[i]$. The utilization factor provides the estimation of system resource distribution from CU level for kernel collocation decision.

Collocation algorithm. With kernel resource demands and mutual interference estimation, Algorithm 1 presents the allocation algorithm that determines how to assign a ready kernel to a subset of CUs. The algorithm is invoked once a queue finishes its execution of the current kernel and the next ready kernel is critical. FEDCM simply adopts the collocation decision from the preceding round for non-critical kernels.

The inputs to the algorithm include the queue and its ready kernel to be scheduled to the GPU, i.e., ($Q[r]$, $Q[r].k$), together with offline profiles and the statistics fed into runtime. We first scan all task queues (including the queue $Q[r]$). For $Q[r]$, we evaluate if the collocation decision from the preceding round is sufficient for the current ready kernel. We first collect the usage statistics from each CU in $Q[r]$.mask, and evaluate the current interference factor.

For other queues, we scan one at a time assuming that $Q[r].k$ is collocated with that queue, and then evaluate the usage and the interference from the collocation. Specifically, for each iterated queue $Q[i]$, we attempt to collocate the ready kernel with it by conducting the bitwise `OR` operations on masks of both queues to produce a new mask m_i. If the resource usage share for $Q[r].k$ under m_i deviates significantly from its resource demand, we randomly allocate/eliminate CUs until it is within a reasonable range (e.g., 10% more or less). FEDCM calculates the utilization factor (Equation 1) and estimates the possible duration of the ready kernel when collocating with $Q[i]$ specified by m_i. For all collocation possibilities, FEDCM chooses the one with shortest estimated time for $Q[r]$ and updates the queue mask accordingly.

Kernel cache bypass. Although helpful in boosting GPU throughput, sharing compute units with multiple kernels tends to worsen the *memory contention* due to intensified accesses. To mitigate potential throughput loss, we propose a kernel-level cache bypass mechanism to cooperatively utilize the limited on-chip memory space. Based on the cache hierarchy of modern GPUs, we develop three instrumentation policies s_1, s_2 and s_{12} that bypass the L1 cache, the L2 cache, and both caches, respectively. Together with the default policy $s_\varnothing$ that has no cache bypass, these four policies serve as the cache management alternatives in FEDCM.

As shown in Section II, two kernels fully share the L1 cache if they exploit exactly the same set of SAs, leading to potential on-chip cache contention. If one kernel occupies all available SAs in the GPU, it aggravates the off-chip interconnect congestion [23]. As such, we leverage the shader array usages for the two kernels that we decide to collocate and determine the cache bypass policy for the kernel to be dispatched.

When the collocation algorithm selects $Q[r].k$ and $Q[t].k$ to collocate where $Q[r].k$ is the ready kernel to be assigned, we determine the bypass setting s based on the following rules:

- If $Q[r]$.mask.SA = $Q[t]$.mask.SA, then bypass L1
- If $Q[r]$ exploits all SAs in GPU, then bypass L2

where SA stands for the occupied SAs for a specific mask.

In the real setting, the bypass decision might hurt the performance of cache-sensitive kernels. To avoid this, we set one extra condition as the filter to prevent $Q[r].k$ suffering from significant performance downgrade due to cache absence. The bypass operation takes effect only if $Q[r].k.T[s, c_{m_r}] \leq \gamma \times Q[r].k.T[s_\varnothing, c_{m_r}]$ holds, where s is one of the cache settings determined by the heuristic rules and c_{m_r} is the number of active CUs from collocation. γ (e.g., 1.1) is the factor controls the slowdowns due to the cache bypass [24] locate in a tolerable range.

IV. Evaluation Methodology

To evaluate the effectiveness of our proposed kernel-level collocation-based GPU sharing, we implement FEDCM in

ROCm 5.5.0 atop of the commodity AMD Radeon 6900XT GPU which uses RDNA2 architecture with GFX1030 ISA. It contains 40 CUs (or WGPs) with 32KB L0 cache per CU, 128KB L1 cache per SA, and 4MB global L2 cache. FEDCM is implemented as a transparent middleware between the driver/hardware and the user program, thus requiring no modification to the application code.

Workloads. First, we compose benchmarks from representative ML models, including AlexNet (A), MobileNet (M), ResNext (R), ResNet18 (R1), Inception (I), ShuffleNet (S), SqueezeNet (S1) and VGG19 (V). We also include five general computing benchmarks for the evaluation of the applicability of FEDCM, including CAR (C), FSM (F), FFT (F1), EPISTASIS (E) and HAUSDORFF (H). We combine multiple [2] ($N = 2, 3, 4$) tasks as one *concurrent workload* and permute the combinations for evaluation. For fair evaluation, we only retain the workloads whose comprised tasks have similar execution time, i.e., within 10× range. The programs use vendor libraries (e.g., MIOpen [25] and rocBLAS [26]) with invariant kernels such that FEDCM can profile the execution prior to collocation.

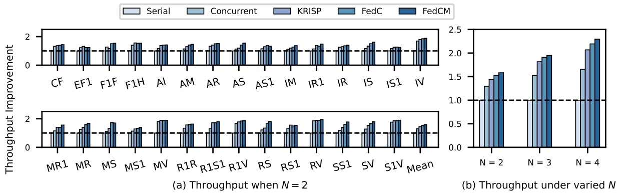
Evaluated schemes. We evaluate and compare the following schemes in our experiments:

- Serial: it launches tasks sequentially one by one. One task cannot start before the previous finishes its execution.
- Concurrent: it is the default task sharing scheme in both AMD's GPU and MPS-enabled NVIDIA's GPU where concurrent tasks are launched onto the GPU simultaneously.
- KRISP [9]: it is the recent kernel-level spatial-partition-based GPU sharing scheme that re-allocates the compute resource of a task at every kernel launch. There is no cache memory management.
- FedC: it is the incomplete FEDCM design that enables only the collocation, i.e., there is no on-chip cache bypass.
- FedCM: it is the full design of FEDCM, which consists of both task collocation and kernel cache bypass management.

V. RESULTS AND ANALYSIS

A. Throughput

We first evaluate *throughput* for different schemes, which is calculated as the sum of relative performance compared to standalone execution [27]. It can be observed from Fig. 5 that FedCM brings an averaged 58.2% throughput improvement compared to Serial when $N = 2$, and 94.8% and 129.4% for $N = 3, 4$. In particular, by enabling cache bypass, FedCM achieves an extra 5.7% improvement over FedC. Note that the number of kernels in each task varies greatly.For workloads stemming from tasks with more kernels (ShuffleNet, MobileNet), FedCM outperforms KRISP more significantly (MS, RS, SV). Recall that FEDCM only schedules on *critical* kernels (Section III-A) while KRISP adjusts the resource allocation on every kernel launch. The latter tends to introduce large scheduling and queue-level mask updating overheads.



(a) Throughput when $N = 2$

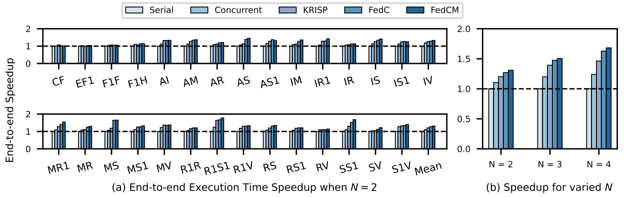
(b) Throughput under varied N

Fig. 5. Throughput improvement for different schemes.

Fig. 5(b) reports the averaged throughput improvement when the number of concurrent tasks (N) changes, where FedCM achieves 13.3% for $N = 3$ and 22.7% for $N = 4$ compared to state of the art KRISP. Injecting every kernel launch leads to severer contentions to access critically shared data structures (e.g., the resource usage statistics U) and thus higher scheduling overhead, making KRISP less competitive than our proposed design. Meanwhile, the disparity between FedC and FedCM also increases (from 6.4% when $N = 2$ to 10.0% when $N = 4$) as the selective cache bypass design involved in FedCM is able to reduce unnecessary data residencies and evictions, leaving the effective cache capacity larger for concurrent kernels.

B. Execution Time

We next study the end-to-end execution time, a critical metric to evaluate the perceived experience for the end-users. Fig. 6(a) demonstrates the *end-to-end execution time* for each composed workload, which represents the elapsed time between synchronized beginning and end for the last finished task. With FedCM, 30.8% end-to-end speedup can be captured compared to the serial execution, and kernel cache management can deliver additional 3.8% benefits for $N = 2$. Compared to KRISP, FedCM also brings 10.8% end-to-end performance elevation. Similar to the overall throughput, FedCM also outperforms more significantly than other schemes when the encompassed task has more kernels (SV, MS).



(a) End-to-end Execution Time Speedup when $N = 2$

(b) Speedup for varied N

Fig. 6. End-to-end task execution time speedup under evaluated schemes.

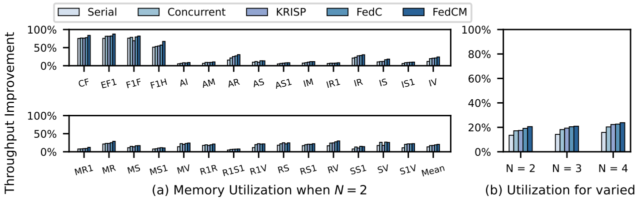
Fig. 6(b) exhibits the end-to-end duration improvements for varied N, where FedCM achieves 11.2% better than KRISP when $N = 3$ and 22.0% when $N = 4$, indicating that FedCM scales well when there are more concurrent tasks.

C. Memory Utilization

We leverage rocm-smi [28] to study GPU on-chip memory utilization under different schemes and summarize the results in Fig. 7. A higher on-chip memory utilization indicates fewer cache evictions and reduced memory stalls, which further result in improved performance. Figure 7(a) presents the memory utilization (in percentage) for the workloads when

[2] When $N \geq 5$, some tasks have to be serialized since the memory demands of parallel execution exceed the capacity of the testbed GPU. The N can be correspondingly adjusted when adopting other GPUs with varying resources.

$N = 2$. KRISP achieves a better averaged memory utilization over Concurrent and Serial. However, since it does not consider memory contention during scheduling, the memory utilization of KRISP can be worse than Concurrent for some workloads, e.g., SS1 and SV.



(a) Memory Utilization when $N = 2$ (b) Utilization for varied N

Fig. 7. Memory utilization for manifold schemes.

FedCM, due to its proactive on-chip cache management, achieves the best memory utilization of all schemes. On average, the memory utilization in FedCM is 17.8% and 6.9% higher than those in KRISP and FedC, respectively. As we can see, general computing workloads (CF, EF1, F1F and F1H) posses higher memory utilization for they usually consist of compute-intensive kernels. Regarding ML workloads, we also observe that the workloads that contains ResNext (R) show higher memory bandwidth than others. This is because, ResNext implements most convolutions via gridwise_convolution* kernels. These kernels demand all CUs in the GPU but are cache insensitive. FedCM decides to bypass the global L2 cache for these kernels during execution such that other collocated kernels can leverage the on-chip cache space and achieve higher memory utilization.

Figure 7(b) summarizes the memory utilization when we vary N from 2 to 4. FedCM achieves the best memory utilization in all cases: 19.9% higher over Concurrent when $N = 3$ and 17.1% higher when $N = 4$.

D. Sensitivity Studies

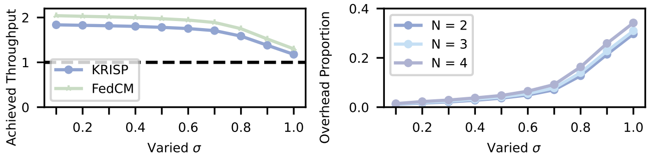


Fig. 8. Sensitivity studies for (a) throughput (b) FedCM scheduling overhead proportion relative to whole execution with varying critical kernel ratio (σ).

We use σ as the ratio of kernels identified as critical (Section III-A) and get injected based on their execution time ranks within the task. Figure 8(a) exhibits the achieved throughput for ML workloads when σ varies from 0.1 (top 10% long running kernels are critical) to 1.0 (all kernels are injected). As σ increases, the achieved throughput drops due to more frequent mask updates as more kernels are identified as critical. Besides, under the same σ, FedCM still outperforms KRISP since it has the holistic consideration of compatible kernel consolidation and adaptive memory customization. Figure 8(b) shows the averaged overhead proportion relative to the total duration of each task. We can draw that the overhead of FedCM elevates as the number of concurrent tasks N increases (from $N = 2$ to 4) under the same σ.

VI. Related Work

GPU task sharing. Hyper-Q [5] and multi-programming [8] are vendor provided techniques to enable concurrent kernel execution without service level guarantees. To strengthen QoS, isolation mechanism like MPS [15], MIG [16] and MxGPU [17] have been introduced to assign disjoint system resources to GPU tasks. To improve GPU sharing, studies have been performed to enhance task scheduling and management [29]–[37]. DASE [38], Prophet [39], C-Laius [40], Themis [41] and HSM [42] predict the co-execution slowdowns so as to improve sharing strategies. Recent works of REEF [7], Paella [43], Orion [6] and KRISP [9] target at real-time scheduling of batched inferences. KRISP [9], the closest work to FedCM, exploits kernel-wise resource allocation to speed up ML inferences. Compared to FedCM, recent designs [7], [43] share GPUs at coarse granularity, e.g., process or task levels. They tend to miss the kernel-kernel scheduling opportunities, which differs from the collocation strategy in FedCM. FedCM also manages on-chip caches to alleviate memory resource contention in kernel collocation.

GPU memory management. Tremendous efforts have been devoted into the research of GPU memory management [24], [44]–[57]. Given the criticality of on-chip caches, a myriad of schemes have been presented to optimize cache management, including line replication [48]–[50], bypassing [24], [44], space unification [51]–[53], and capacity extension [56], etc. For the bypassing designs, Xie *et al.* proposes a mixture of static and dynamic mechanism [44] to selectively bypass cache at instruction level. The revised design [24] combines task-aware cache partition and cache bypassing at a per-thread level. Those cache optimization designs, including the bypass ones, are all requiring dedicated hardware changes, thus making them infeasible on existing commodity GPUs. Recently, software control has been introduced to GPUs, e.g., NVIDIA residency control [18], [55] and AMD policy tuning [20], which generally lays the foundation of FedCM's fine-grained cache management.

VII. Conclusion

This paper introduces FedCM, a middleware that enhances GPU runtime to achieve better system throughput. For implementation, FedCM prototype is developed on off-the-shelf AMD GPUs, through enabling kernel-level mask manipulation and cache setting configuration. Evaluation results demonstrate that FedCM can improve the system throughput and execution time more efficiently over state-of-the-arts.

Acknowledgment

We thank the anonymous reviewers for their constructive and insightful feedback. This work was supported in part by the National Natural Science Foundation of China (NSFC) under grant 62472462.

References

[1] M. Park, K. Bhardwaj, and A. Gavrilovska, “Pocket: ML serving from the edge,” in *EuroSys*, 2023.

[2] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, “Analysis of large-scale multi-tenant GPU clusters for DNN training workloads,” in *USENIX ATC*, 2019.

[3] Q. Hu, P. Sun, S. Yan, Y. Wen, and T. Zhang, “Characterization and prediction of deep learning workloads in large-scale GPU datacenters,” in *SC*, 2021.

[4] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, “MLaaS in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters,” in *NSDI*, 2022.

[5] “Tuning CUDA Applications for Kepler,” https://docs.nvidia.com/cuda/archive/11.4.4/kepler-tuning-guide/index.html#hyper-q.

[6] F. Strati, X. Ma, and A. Klimovic, “Orion: Interference-aware, fine-grained GPU sharing for ML applications,” in *EuroSys*, 2024.

[7] M. Han, H. Zhang, R. Chen, and H. Chen, “Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences,” in *OSDI*, 2022.

[8] “HIP Programming Manual — HIP Documentation,” https://rocm.docs.amd.com/projects/HIP/en/latest/user_guide/programming_manual.html.

[9] M. Chow, A. Jahanshahi, and D. Wong, “KRISP: enabling kernel-wise right-sizing for spatial partitioned GPU inference servers,” in *HPCA*, 2023.

[10] W. C. H. Z. C. X. Z. Z. W. L. Shulai Zhang, Quan Chen and M. Guo, “Improving GPU sharing performance through adaptive bubbleless spatial-temporal sharing,” in *EuroSys*, 2025.

[11] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, “Accel-sim: An extensible simulation framework for validated GPU modeling,” in *ISCA*, 2020.

[12] Y. Bao, Y. Sun, Z. Feric, M. T. Shen, M. Weston, J. L. Abellán, T. Baruah, J. Kim, A. Joshi, and D. R. Kaeli, “Navisim: A highly accurate GPU simulator for AMD RDNA GPUs,” in *PACT*, 2022.

[13] T. Guo, X. Huang, K. Wu, X. Zhang, and N. Xiao, “SMILE: LLC-based shared memory expansion to improve GPU thread level parallelism,” in *DAC*, 2024.

[14] “AMD RDNA Whitepaper,” https://www.amd.com/system/files/documents/rdna-whitepaper.pdf.

[15] “NVIDIA Multi-Process Service,” http://docs.nvidia.com/deploy/mps.

[16] “NVIDIA Multi-Instance GPU (MIG),” https://www.nvidia.com/en-us/technologies/multi-instance-gpu.

[17] “AMD Virtual Graphics,” https://www.amd.com/en/graphics/workstation-virtual-graphics.

[18] “NVIDIA Ampere GPU Architecture Tuning Guide,” https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html.

[19] “AMD Instinct Accelerators,” https://www.amd.com/en/graphics/instinct-server-accelerators.

[20] “Syntax of AMD GPU Instruction Modifiers,” https://llvm.org/docs/AMDGPUModifierSyntax.html.

[21] M. Xi, T. Guo, X. Huang, Z. Lin, and X. Zhang, “Mpache: Interaction aware multi-level cache bypassing on GPUs,” in *ASP-DAC*, 2025.

[22] M. Xi, J. He, and X. Zhang, “CacheC: LLM-based GPU cache management to enhance kernel concurrency,” in *Euro-Par*, 2025.

[23] S. Jain, I. Baek, S. Wang, and R. Rajkumar, “Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs,” in *RTAS*, 2019.

[24] Y. Liang, X. Li, and X. Xie, “Exploring cache bypassing and partitioning for multi-tasking on GPUs,” in *ICCAD*, 2017.

[25] “MIOpen Documentation,” https://rocm.docs.amd.com/projects/MIOpen/en/latest/.

[26] “rocBLAS documentation — rocBLAS Documentation,” https://rocm.docs.amd.com/projects/rocBLAS/en/latest/.

[27] C. Zhao, W. Gao, F. Nie, and H. Zhou, “A survey of GPU multitasking methods supported by hardware architecture,” *TPDS*, 2022.

[28] “ROCm System Management Interface Documentation,” https://rocm.docs.amd.com/projects/rocm_smi_lib/en/latest/.

[29] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, “Improving GPGPU concurrency with elastic kernels,” in *ASPLOS*, 2013.

[30] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram, “Warped-slicer: Efficient intra-SM slicing through dynamic resource partitioning for GPU multiprogramming,” in *ISCA*, 2016.

[31] Z. Wang, J. Yang, R. G. Melhem, B. R. Childers, Y. Zhang, and M. Guo, “Simultaneous multikernel GPU: multi-tasking throughput processors via fine-grained sharing,” in *HPCA*, 2016.

[32] J. J. K. Park, Y. Park, and S. A. Mahlke, “Dynamic resource management for efficient utilization of multitasking GPUs,” in *ASPLOS*, 2017.

[33] Z. Wang, J. Yang, R. G. Melhem, B. R. Childers, Y. Zhang, and M. Guo, “Quality of service support for fine-grained sharing on GPUs,” in *ISCA*, 2017.

[34] J. Kim, J. Kim, and Y. Park, “Navigator: Dynamic multi-kernel scheduling to improve GPU performance,” in *DAC*, 2020.

[35] H. Zhao, W. Cui, Q. Chen, J. Zhao, J. Leng, and M. Guo, “Exploiting intra-SM parallelism in GPUs via persistent and elastic blocks,” in *ICCD*, 2021.

[36] H. Zhao, W. Cui, Q. Chen, Y. Zhang, Y. Lu, C. Li, J. Leng, and M. Guo, “Tacker: Tensor-CUDA core kernel fusion for improving the GPU utilization while ensuring QoS,” in *HPCA*, 2022.

[37] A. Barnes, F. Shen, and T. G. Rogers, “Mitigating GPU core partitioning performance effects,” in *HPCA*, 2023.

[38] Q. Hu, J. Shu, J. Fan, and Y. Lu, “Run-time performance estimation and fairness-oriented scheduling policy for concurrent GPGPU applications,” in *ICPP*, 2016.

[39] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, “Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers,” in *ASPLOS*, 2017.

[40] W. Zhang, Q. Chen, N. Zheng, W. Cui, K. Fu, and M. Guo, “Toward QoS-awareness and improved utilization of spatial multitasking GPUs,” *IEEE TC*, 2022.

[41] 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*, 2019.

[42] X. Zhao, M. Jahre, and L. Eeckhout, “HSM: A hybrid slowdown model for multitasking GPUs,” in *ASPLOS*, 2020.

[43] K. K. W. Ng, H. M. Demoulin, and V. Liu, “Paella: Low-latency model serving with software-defined GPU scheduling,” in *SOSP*, 2023.

[44] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, “Coordinated static and dynamic cache bypassing for GPUs,” in *HPCA*, 2015.

[45] H. Wang, F. Luo, M. A. Ibrahim, O. Kayiran, and A. Jog, “Efficient and fair multi-programming in GPUs via effective bandwidth management,” in *HPCA*, 2018.

[46] Z. Lin, H. Dai, M. Mantor, and H. Zhou, “Coordinated CTA combination and bandwidth partitioning for GPU concurrent kernel execution,” *ACM TACO*, 2019.

[47] E. C. Marangoz, K. Kang, and S. Shin, “Designing GPU architecture for memory bandwidth reservation,” in *ISPASS*, 2021.

[48] M. A. Ibrahim, O. Kayiran, Y. Eckert, G. H. Loh, and A. Jog, “Analyzing and leveraging decoupled L1 caches in GPUs,” in *HPCA*, 2021.

[49] M. A. Ibrahim, H. Liu, O. Kayiran, and A. Jog, “Analyzing and leveraging remote-core bandwidth for enhanced performance in GPUs,” in *PACT*, 2019.

[50] M. A. Ibrahim, O. Kayiran, Y. Eckert, G. H. Loh, and A. Jog, “Analyzing and leveraging shared L1 caches in gpus,” in *PACT*, 2020.

[51] Y. Oh, G. Koo, M. Annavaram, and W. W. Ro, “Linebacker: preserving victim cache lines in idle register files of GPUs,” in *ISCA*, 2019.

[52] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, “Unifying primary cache, scratch, and register file memories in a throughput processor,” in *MICRO*, 2012.

[53] S. Darabi, M. Sadrosadati, N. Akbarzadeh, J. Lindegger, M. Hosseini, J. Park, J. Gómez-Luna, O. Mutlu, and H. Sarbazi-Azad, “Morpheus: Extending the last level cache capacity in GPU systems using idle GPU core resources,” in *MICRO*, 2022.

[54] S. Pal, S. Venkataramani, V. Srinivasan, and K. Gopalakrishnan, “Efficient management of scratch-pad memories in deep learning accelerators,” in *ISPASS*, 2021.

[55] Y. Fu, E. Bolotin, A. Jaleel, G. Dalal, S. Mannor, J. Subag, N. Korem, M. Behar, and D. W. Nellans, “Autoscratch: ML-optimized cache management for inference-oriented GPUs,” in *MLSys*, 2023.

[56] S. Darabi, E. Yousefzadeh-Asl-Miandoab, N. Akbarzadeh, H. Falahati, P. Lotfi-Kamran, M. Sadrosadati, and H. Sarbazi-Azad, “OSM: off-chip shared memory for GPUs,” *IEEE TPDS*, 2022.

[57] W. Pan, Z. Lin, J. Du, and X. Zhang, “HuntKTm: Hybrid scheduling and automatic management for efficient kernel execution on modern GPUs,” *TACO*, 2025.