2023 IEEE 41st International Conference on Computer Design (ICCD) | 979-8-3503-4291-8/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICCD58817.2023.00046

KeSCo: Compiler-based Kernel Scheduling for Multi-task GPU Applications

Zejia Lin[§]
Sun Yat-sen University
Guangzhou, China
linzj39@mail2.sysu.edu.cn

Zewei Mo[§+]
University of Pittsburgh
Pittsburgh, Pennsylvania, USA
zewei.mo@pitt.edu

Xuanteng Huang
Sun Yat-sen University
Guangzhou, China
huangxt57@mail2.sysu.edu.cn

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

Yutong Lu
Sun Yat-sen University
Guangzhou, China
yutong.lu@nscc-gz.cn

***Abstract*—Nowadays, Graphics Processing Units (GPUs) dominate in a wide spectrum of computing realms and multi-task is increasingly applied in various complicated applications. To gain higher performance, multi-task programs require cumbersome programming efforts to take advantage of inter-kernel concurrency at source-code level. Although there exist works automatically scheduling kernels to enable inter-kernel concurrency, they all inevitably introduce new programming frameworks and some even bring significant performance downgrade compared to the expertise-based optimizations. To address this issue, we propose `KeSCo`, a compiler-based scheduler to expose kernel level concurrency in multi-task programs with trivial code modification. In compilation, `KeSCo` applies a strategy to schedule kernels in task queues, accounting for both load balance and synchronization cost. Also, `KeSCo` utilizes a customized algorithm designed for computational flow to remove redundant synchronizations. The design is further extended to support multi-process scenario, where multiple GPU processes are sharing a single context. Evaluations on representative benchmarks show that the proposed approach gains a 1.28× average speedup for multi-task scenario (1.22× for multi-process). Even with lessened programming efforts, our proposed design outperforms two state-of-the-arts GrSched and Taskflow by 1.31× and 1.16× on average, respectively.**

***Index Terms*—GPU, Compiler, Multi-Task, Kernel Scheduling**

I. INTRODUCTION

In the last decade, Graphics Processing Units (GPUs) have been widely applied in a myriad of domains, owing to their excessive computation capability and high memory throughput. Advanced GPUs incorporate ample resources than what a typical monolithic GPU task or kernel necessitates and are thus frequently being underutilized, especially when executing single-task programs, which launch just one kernel at a time. To alleviate the under-utilization issue, a plethora of approaches have been proposed, like concurrently executing sliced kernels [1] and resource virtualization [2].

However, as GPU applications getting more complex, multi-task programs, originally consisting of concurrently executable kernels, show up in diverse domains. Compared to single-task programs with constrained inter-kernel concurrency, multi-task programs can leverage various GPU streams and synchronization events to parallelize serial kernel executions to efficiently shorten run time. Such an optimization requires developers to correctly analyze dependency between kernels and then re-arrange kernels in task queues to strike load balance and minimize synchronization cost. With no doubt, considerable programming efforts should be paid to obtain bug-free and highly performant codes, particularly for increasingly complicated programs. To address the issue, a bunch of designs have been recently presented to automate inter-kernel concurrency of GPU applications, especially for general high performance computing (HPC). RAMMER [3] focuses on inter- and intra-kernel concurrency in Recursive Neural Network (RNN), but lacks scalability to handle general programs. Taskflow [4] proposes a new heterogeneous programming framework for automatic optimization of inter-kernel concurrency. It harnesses cudaGraph [5] to reduce overheads of fragmented kernel launches. Nevertheless, such method requires developers to grasp a new programming model and manually specify kernel dependencies, inevitably raising coding difficulty. A GrCUDA-based [6] runtime approach [7] applies a virtual machine, exempting developers from the need to explicitly claim kernel dependencies. But compared to expertise-based optimizations, it introduces serious performance downgrade due to the overheads of run-time scheduling.

To automatically achieve kernel concurrency in multi-task programs at source-code level with petty programming effort, we propose `KeSCo`, a compiler-based static kernel scheduler requiring trivial code modification. It automatically identifies data dependencies and then places kernels into different streams concerning load balance and synchronization cost. The transformed code is a highly performant executable with kernels being ready to run concurrently. The design is further extended to schedule a collective of prioritized multi-task

[§]Equal contribution.
[+]Work done when studying at Sun Yat-sen University.
[#]Corresponding author.

processes, which is common in today's GPU space-sharing scenario. The scheduler maintains a stream zone for each sub-program, actively issues kernels of high priority, and meanwhile demotes leftover kernels via adding barriers across zones. This promotes the early completion time of highly prioritized tasks while saturating GPU resources with the low-priority ones, thus effectively reducing the makespan of the whole program.

In summary, the contributions of this paper are:

- We highlight the inadequate performance enhancement and programming weakness of prior arts in automatically achieving inter-kernel concurrency for multi-task programs.
- We propose a static scheduler for inter-kernel concurrency in multi-task programs, well accounting for both load balance and synchronization cost.
- We design a priority-based scheduling strategy for kernels across multi-task programs, with lowered programming burden to enable high kernel concurrency and facilitate prioritized kernels to speed up executions.
- The evaluations show that our design can effectively raise kernel-level parallelism to boost GPU performance, outperforming the state-of-the-arts with obviously less programming efforts.

II. Background And Motivation

A. Concurrent Kernel Execution (CKE)

Designed for massively parallel computation, modern GPUs are typically equipped with many *streaming multiprocessors* (SMs), each of which has hundreds of computing cores and can simultaneously execute up to thousands of threads. In most cases, one single kernel cannot fully utilize all resources, thus causing a great waste of computation power and low performance. To alleviate such a problem, CKE parallelizes inter-kernel execution on available hardware components. It issues operations in multiple software task queues (called *streams* in CUDA [5]), which are mapped onto different hardware queues and processed concurrently if the demanded resources, typically SMs, are sufficient.

The multi-task workloads provide a perfect scenario to implement CKE for acceleration, as they have independent kernels ready to execute concurrently. Developers need to properly scrutinize the complex dependency, schedule kernels in streams, and generate synchronization barriers. Such code re-organization incurs tremendous manual efforts and is thus also error-prone. This laborious process can be automated at different levels of granularity. Many DL compilers like TVM [8] or XLA [9] leverage dedicated collaboration with domain-specific language (DSL) embedded in Python for such task-level parallelization. In HPC where applications are majorly implemented in C++ with GPU programming model, parallelization is exploited at finer sub-task level. Representative techniques include slicing kernels into sub-kernels to saturate GPU resources [10][11], and employing preemption for high-priority tasks [12][13]. They involve coupled compiler-runtime

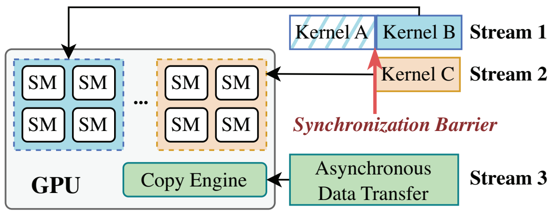


Fig. 1: Execution of concurrent tasks on GPU.

systems for thorough optimizations and have shown great improvement in performance. Nevertheless, these intra-kernel approaches necessitate fine-grained code transformation inside kernels, which becomes infeasible when dealing with hand-tuned kernels or hardware vendors' highly optimized closed-source libraries having no available codes.

B. CKE Programming in CUDA

Many popular GPU programming models offer a series of concurrency APIs for CKE, here we take CUDA as an example. A data flow graph (DFG) needs to be constructed correctly first to help schedule the executions. The DFG is further divided into multiple levels such that kernels from the same level have no data dependence. Then developers need to create multiple *CUDA streams*, and issue kernels on different streams to co-execute on GPUs. To ensure the execution order of data dependent kernels across streams, *CUDA events* are inserted after a kernel's predecessors as trackers of the completion state, which are awaited by the synchronization barriers before the kernel.

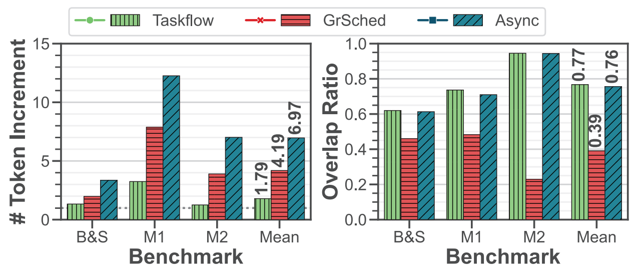
Figure 1 shows an example of three concurrent tasks sharing a GPU. Kernels B and C, are mutually independent, both depend on kernel A. After kernel A finishes, kernels B and C are issued on different streams and executed simultaneously on different SMs. At the same time, an asynchronous copy is proceeding on the copy engine, which is a complementary hardware resource with respect to SMs. Therefore computation of the two kernels and data transfer are overlapped, helping utilize the abundant resources of GPUs.

C. Motivation

1) Programming Efforts: Aiming to automate inter-kernel CKE for GPU programs, approaches like Taskflow [4] and a GrCUDA-based [6] scheduler (aliased as GrSched for ease of reference)[7] have been proposed to craft new programming frameworks by extending CUDA's API for stream management and synchronization. Taskflow demands explicitly specifying dependencies through its APIs, while GrSched introduces DSL embedded in Python to support automatic analysis and scheduling. In Figure 2a, we evaluate the programming efforts required by implementing CKE at source-code level in three benchmarks elaborated in Section IV-A. Compared with serial execution, manual optimization with CUDA's streams and events APIs (named *Async* in figures) costs $6.97\times$ extra tokens to fully expose the kernel concurrency. Taskflow and GrSched yet involve $1.79\times$ and $4.19\times$ additional tokens respectively, to transplant the serial implementation into their programming models. In contrast with *Async*, they provide straightforward view of dependencies and facilitate maintainability, but yield thorough refactoring of source code.

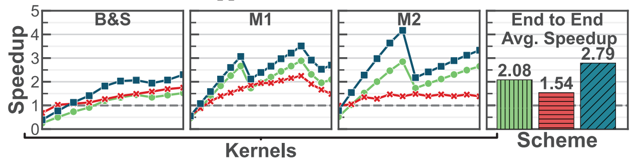
2) Scheduling Policy: Another preliminary study finds that the aforementioned frameworks would bring serious performance penalties compared to the expertise optimization (the *Async* scheme). Figure 2b illustrates the actual overlapped time with respect to the theoretical peak. Figure 2c explores the speedup of each kernel normalized to the serial execution. The static scheduler Taskflow's overlap ratio is slightly higher than *Async*, but the makespan is 34.1% longer. This performance degradation is caused by prolonged kernel execution time, indicating that the scheduler issues excessive kernels and thus poses resource competition. As a dynamic scheduler, GrSched insufficiently overlaps the computation, with only 39% to the theoretical maximum and 81.1% slower than *Async*. The major reason is the overhead brought by runtime dependency analysis, blocking CPU from launching GPU kernels concurrently. We observe from the above cases that performance gain is two-folded as contributed by both the overlap ratio and the number of issued kernels. A balance between them is demanded to bring an optimal scheduling policy.

3) Multi-process Scenario: It is prevalent to serve multiple programs with different priorities on a single GPU since one application might not fully utilize all GPU resources [14]. To improve the performance of multi-program on a shared GPU, Nvidia MPS [15] transparently co-operates multiple CUDA processes at runtime. This code-free tool eliminates manual labor but has no guarantee for priority and suffers from overheads of dynamic scheduling. Other prior arts including GrSched and Taskflow are designed for scheduling one single program and lack prioritized scheduling mechanisms to ensure the early completion of high-priority programs, hindering their extension in the multi-process scenario. Therefore, there remains the urgency for a source code-level approach capable of priority-based scheduling without involving much programming effort in these complicated scenarios.



(a) Programming effort required to transform the serial application.

(b) Overlap ratio (actual *vs.* theoretical).



(c) Speedup of each kernel's completion time from the beginning of the application. The i-th point in each sub-figure indicates the i-th kernel, the rightmost point (last kernel) indicates the speedup of the whole application's execution.

Fig. 2: Analysis on three applications with 12 kernels and 10 dependencies on average. Over 5 of the kernels in each application can be executed concurrently.

The aforementioned observations suggest that the schedulers wrapped as new programming frameworks incur inefficient scheduling strategies and strenuous programming efforts. Instead, a compiler-based approach naturally grasps global information about the application and optimizes in-depth without refactoring the source code. To this end, we propose `KeSCo` to automate the scheduling at compile-time and achieve competent performance compared with manual-optimized approach.

III. Design

We introduce `KeSCo`, a compiler-based static scheduler for concurrent kernel execution in multi-task programs, to automatically enable inter-kernel concurrency[1]. It leverages lightweight code modifications to help construct data flow graph (DFG) of kernels and then schedules kernels to multiple streams as well as generates synchronization barriers to guarantee correct execution order with low cost. In addition to supporting inter-kernel concurrency in a single program, we extend `KeSCo` to support multi-process scheduling while meeting the requirements of early completion of high-priority sub-programs[2] and saturating hardware resources with low-priority ones.

A. Overview

Figure 3 shows the overall workflow of our proposed design. The input is the source code of an application with serial execution and the output is a high-performance executable with concurrent kernel execution. The optimizing procedure consists of three parts: *DFG constructor*, *kernel distributor*, and *synchronization generator*. As the forefront phase, *DFG constructor* analyzes the input and output of each GPU kernel and builds a data flow graph according to their execution order.

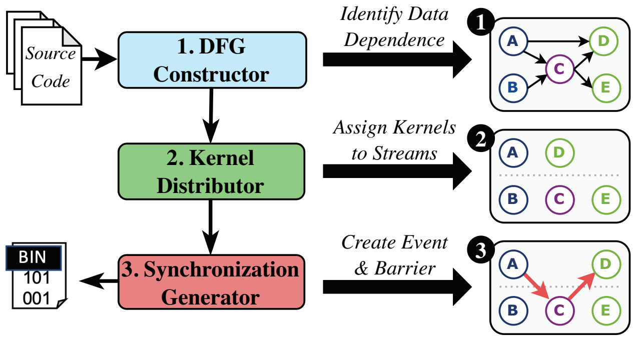


Fig. 3: General workflow of `KeSCo` to transform serial source code into the executable with efficient parallelism.

[1]The proposed design is a kernel-level scheduler. Therefore we refer to kernel and task interchangeably, unless otherwise specifically stated.

[2]We indicate sub-program as a distinct multi-task application, compounded in the multi-process application.

Then the *kernel distributor* leverages the graph and schedules kernels into different streams. Last, the *synchronization generator* creates barriers for dependent kernels .

B. DFG Constructor

DFG constructor analyzes dependencies among kernels based on their serial execution order and the relations of Write-After-Read (WAR), Write-After-Write (WAW) and Read-After-Write (RAW). As massive dependencies for variables are ubiquitous in complex HPC programs, it can be expensive to construct the kernels' dependency graph from the complicated data flow. To reduce repeated search operations, we construct the DFG by finding all predecessors of a kernel. Kernels are iterated by their reversed execution order. For every kernel, breadth-first search is adopted to find its direct predecessors and terminated once all of them are found. Additionally, to distinguish read-only from writable parameters of kernels, which are passed as pointers and potentially have identical memory addresses, hindering compiler-based approach from analyzing them statically, developers are necessitated to add a light wrapper to the kernel. The mechanism is detailed in Section III-F.

C. Kernel Distributor

When the DFG is determined, *kernel distributor* places the kernels in GPU streams in the order explained below to gain computational overlap. First, it levelizes the DFG so that kernels in the same level have no mutual data dependency. At each level, the kernels are assigned with a unique index value. Then kernels are placed in the first level using their indices modulo to the stream count. For the remaining kernels, the distributor follows a set of rules in consideration of load balance and synchronization cost. ❶ The key idea of the rules is that a kernel issues right after any of its predecessors when possible, to reduce synchronizations among streams. We call these predecessors a *preferred predecessor set* (*PP-Set*). ❷ To avoid conflict when multiple kernels' *PP-Set* intersects, we sort the kernels by the size of the *PP-Set*, schedule the kernels with smaller *PP-Set* first, and then update the *PP-Set* of unscheduled kernels. ❸ If there still exist conflicts, the kernel is randomly assigned to a stream containing its predecessor.

Here we exemplify the above steps with regard to the given DFG in Figure 4 and the corresponding kernel distribution strategy in Figure 5. Kernels in level 1 are put to the stream by their indices accordingly. In level 2, kernel *F* is scheduled first by rule ❷ as it has the smallest *PP-Set*, and is positioned after kernel *C* by rule ❶. Then kernel *E*'s updated *PP-Set* is smaller than that of kernel *D*, and is thus arranged after kernel *A*. Finally, kernel *D* is put after kernel *B* as accounting for rule ❶ again. In level 3, we repeat the process and schedule them in the order of kernel *H*, *I*, and *J*, which are all placed after their *preferred predecessor*. Lastly kernel *G* can choose from stream 1 and 3, where its predecessors are seated, and are randomly inserted in stream 3, as shown in Figure 4.

D. Synchronization Generator

After scheduling kernels in asynchronous streams, *synchronization generator* comes into play to ensure the correctness of the execution order. A naive approach is to create barriers whenever a data dependence exists. However, a part of the barriers are redundant and may cause performance overhead. To tackle this issue, a pruning algorithm is proposed based on the implicit synchronizations brought by the transitivity of dependency and serial execution of kernels in the same stream. When finish, the barriers are pruned to the minimum.

The *synchronization generator* traverses the kernels in each stream and works in three steps, suppose it is working on kernel K. In step ❶, it creates barriers for each of K's predecessors which do not share the stream with K. In step ❷, it checks K's predecessors in each stream, and reserves only the synchronization issued from the last predecessor in that stream. In step ❸, it enumerates kernels before K in the same stream, say T. If a K and T's predecessor share the same stream, and K's predecessor is executed before T's, K is then implicitly synchronized by T and T's predecessor. Therefore K's barrier to that predecessor is safe to be removed. Full analysis of the complete DFG helps eliminate these redundant barriers correctly. In run-time analysis of GrSched, such elimination is infeasible due to the lack of a global view of the graph.

The example of Figure 5 shows the barriers generated in solid lines and the removed barriers in dashed lines. *Synchronization generator* scans stream 1 and creates kernel E and I's barriers by step ❶. The same is true for kernel D in stream 2 and kernel J in 3. For kernel G, step ❸ detects its implicit synchronization with kernel A by the execution order of $A \rightarrow E \rightarrow J$, so the barrier is removed.

E. Kernel Scheduling in Prioritized Multi-process

The proposed design above focuses on enabling inter-kernel concurrency in a single program, and it is extensible to schedule independent sub-programs with diverse priorities in conformity to today's GPU sharing scenario. We introduce

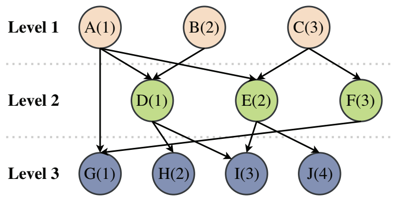


Fig. 4: A DFG organized in three levels to schedule ten kernels onto three available streams.

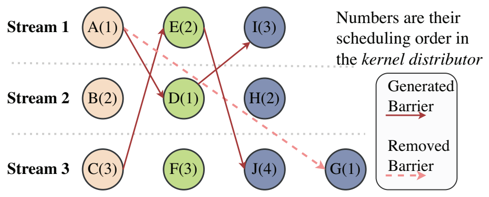


Fig. 5: Scheduling strategy of *kernel distributor* and *synchronization generator* for the DFG in Fig. 4.

stream zone manager as an orthogonal module with *kernel distributor* and *synchronization generator* to coordinate among sub-programs and limit the number of issued kernels in avoidance of resource competition.

In the design extension, developers first manually wrap independent applications as distinct functions, and *stream zone manager* provides each sub-program a separated set of streams, in which the *kernel distributor* and *synchronization generator* schedules the sub-program's tasks. Policies on scheduling across stream zones are straightforward: ❶ The foremost is creating barriers for low-priority tasks to block them until the high-priority ones are finished; ❷ Then for tasks of the same priority, the number of issued kernels is limited by a hyper-parameter[3]; ❸ If there is only one kernel left to execute a high-priority task, an additional kernel from a lower-priority task is issued beforehand. This is because a single kernel can rarely saturate all GPU resources. The above steps introduce a significant number of synchronization barriers and are pruned by *synchronization generator*. Figure 6 shows an example of scheduling two sub-programs. Kernels A, B, and C are issued first as they are of high priorities. By the additional rule ❸, kernel 1 is promoted to launch in interleave with kernel C, and barriers are generated for kernels 2, 3, and 4.

F. Implementation

Figure 7 shows the implementation pipeline of `KeSCo`, on basis of LLVM Compiler Infrastructure [16]. Although targeting at the CUDA platform, our design can be easily applied to other frameworks that support concurrent task queues (e.g. HIP [17] and SYCL [18]). The GPU kernel code is separated from the host (CPU) code and compiled individually to binary, while host code is compiled to intermediate representation (IR). The binary file of GPU kernels is embedded into host IR file and kernel functions are called by `cudaPushCallConfiguration`. We pinpoint this pattern to find the serial-issued kernels in host IR and apply our optimizations to their caller functions.

Identifying writable parameters at the compiler level is a challenging task, as computational data is often passed as pointers. These pointers may have identical addresses, making it impossible to discern between read-only and writable memory locations at compile time. As a workaround, developers necessitate adding a lightweight wrapper to the kernel,

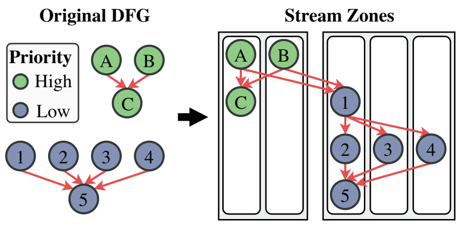


Fig. 6: Two sub-programs in a multi-process workload are scheduled based on priority.

[3]Tuning of this parameter is left to future work. It is set to six in our experiments as it saturates our device in most cases.

which specifies the number of writable parameters N_{out} and rearranges them to the first N_{out} parameters. The wrapper additionally takes a parameter specifying the kernel priority, to work with *stream zone manager*. This technique enables *DFG constructor* to analyze dependencies automatically, without involving any new programming framework.

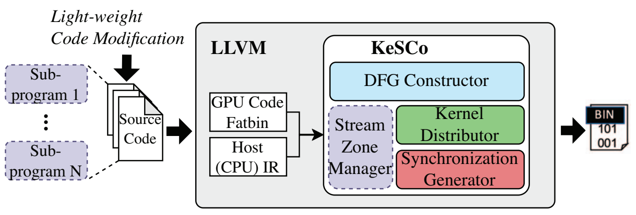


Fig. 7: The processing framework of `KeSCo` implementation. Stream zone manager will be activated if sub-programs with various priorities are contained.

IV. Experimental Evaluation & Results

A. Experimental Setup

1) Platforms: We conduct experiments on a server equipped with Nvidia A100-PCIe-40GB GPUs, an AMD EPYC 7742 64-Core CPU and 256GB DRAM. The operating system is Debian 5.10.179 and the version of Nvidia driver is 470.182.03. We compile GPU programs using LLVM 14.0.0 and CUDA 11.4.4, with the compiler option `-O3` switched on to optimize performance.

2) Benchmarks: We use eight representative workloads listed in Table I. The two in-house micro-benchmark are drawn from the kernels in Nvidia FasterTransformer[19], the rest benchmarks are introduced in [7].

TABLE I: Evaluated benchmarks.

Name	Notation	Domain	Max DFG Width
Micro-1	M1	AI	6
Micro-2	M2	AI	12
Vector Square	VEC	HPC	2
Black & Scholes	B&S	HPC	10
Image Processing	IMG	HPC	3
Machine Learning	ML	AI	2
HITS	HITS	HPC	2
Deep Learning	DL	AI	2

3) Evaluated Schemes: We compare `KeSCo` against the baseline (serial execution, named *Serial* below), the one with expertise concurrency optimization (using CUDA's APIs, named *Async*) and two prior arts including a static scheduler *Taskflow* [4] and a dynamic scheduler [7] based on GrCUDA [6] (denoted as *GrSched*).

4) Metrics: To quantify programming efforts, we consider the Line of Code (LoC) and the number of tokens needed in the manual modification. And, for performance, we average the execution time of all GPU kernels in 10 repeated runs.

B. Programming Effort

Table II lists the programming efforts required by different schemes, in terms of the average LoC and token count. Compared with *Serial*, `KeSCo` costs only 2.3% LoC and 6.1%

tokens in extra to enable CKE and automates both dependency analysis and concurrency management. The extra code is sourced from the kernels' light wrapper for writable parameter identification, and our compiler-based approach encompasses the rest of transformation and optimization. In contrast, significant code modification is involved in other schemes. *Async* necessitates manually managing CUDA's asynchronous APIs, while *Taskflow* lessened this burden and still requires explicit dependence specification. *GrSched* has the merit of automation but is limited to dynamic programming language for runtime analysis. `KeSCo` covers both automation and adaptation in enabling inter-kernel concurrency, and has the benefit of fulfillment in complex scenarios.

C. Speedup in Multi-task Programs

We conduct the evaluations with input consuming around 5GB of memory for each benchmark. The kernel launch configurations vary in *block sizes* (the number of threads per block) under a fixed number of *grid size* (the number of blocks) where *Serial* achieves the highest performance. For each benchmark using the 1D *thread block*, we average results for the *block sizes* from 32 to 1024. For those using the 2D or 3D *thread block* (e.g. HITS, IMG, and DL), the *block sizes* range from 8 or 16 to the maximum acceptable size. We report the execution time in Figure 8. `KeSCo` achieves $1.28\times$ average speedup for all benchmarks, comparable to *Async*. On the other hand, *GrSched* and *Taskflow* achieve $0.98\times$ and $1.10\times$ speedup on average, respectively. For *GrSched*, the most significant performance penalty comes from the high cost of dynamic scheduling, including dependency analysis as well as runtime capture and issue kernels. Furthermore, such overhead prevents the CPU from launching kernels simultaneously. When the kernels complete shortly (e.g. the input size decreases or launched threads increase), the overhead takes a higher proportion of execution time. This worsens the insufficient overlaps among kernels and thus lowers the performance.

The above evaluation mainly takes into account the scheduling among kernels. But in more general scenarios, data transfer is also a significant factor affecting performance. Thus, we evaluate different schemes except *Taskflow*[4] on three representative benchmarks, each of which involves over twenty operations of data transfer in scheduling. Figure 9 demonstrates the result. `KeSCo` achieves $5.01\times$ average speedup, which is only about 7% lower than *Async*, and outperforms *GrSched* by 31.5%. This performance gap between `KeSCo` and *Async* is due to the insufficient overlap of data transfer and kernels, which is caused by the hash-based distributing algorithm for kernels or operations.

TABLE II: Average programming efforts.

Scheme	LoC	#Tokens	D.A.[a]	C.M.[b]	N.P.F[c]	P.L.[d]
Serial	86	378	✗	✗	✓	C++
Async	106	483	✗	✗	✓	C++
Taskflow	173	914	✗	✓	✗	C++
GrSched	366	1832	✓	✓	✗	Python
`KeSCo`	88	401	✓	✓	✓	C++

[a] Automatic Dependency Analysis
[b] Automatic Concurrency Management
[c] No New Programming Framework
[d] Programming Language

[4]The evaluation requires virtual memory mechanism of GPU and asynchronous data prefetch for overlapping computation with data transfer, which is not yet supported in Taskflow

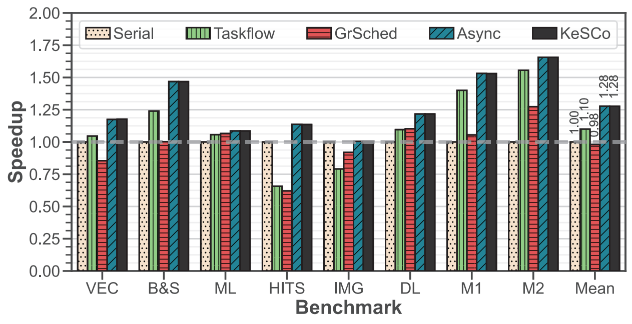


Fig. 8: Average speedup gained by different schemes in eight benchmarks under multiple launching settings.

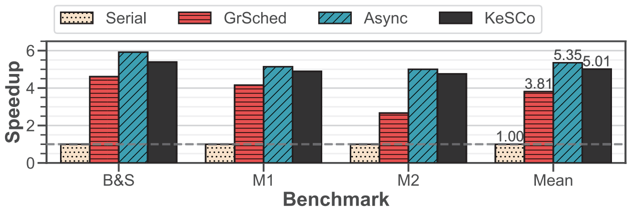


Fig. 9: Speedup from asynchronous data prefetch.

D. Hardware Resource Utilization

In this section, we analyze how `KeSCo` affects hardware metrics such as FP32/64 instruction throughput, memory bandwidth utilization and SM occupancy. We leverage DGCM [20], a tool for monitoring global hardware-level metrics of GPU with low overheads, to collect the average value of each metric with an interval of one millisecond during each execution. Figure 10 details the improvement of these metrics achieved by `KeSCo`. With `KeSCo` being enabled, the average FP32/64 instruction throughput, memory bandwidth utilization, and SM occupancy on all benchmarks increases by $1.78\times$, $1.61\times$ and $2.76\times$, respectively. Particularly, the memory bandwidth throughput of VEC increases by $1.98\times$, indicating that the

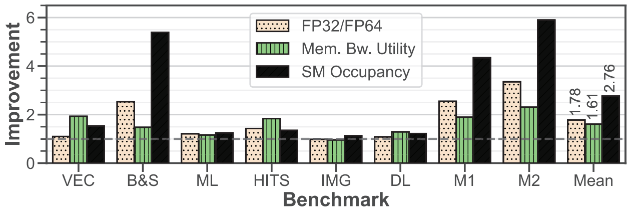


Fig. 10: Hardware metric improvement gained by `KeSCo` in different benchmarks, under the setting where `KeSCo` achieves the highest speedup.

acceleration of VEC mostly benefits from this. For M1, M2, and B&S, KeSCo greatly raises the SM occupancy and the throughput of FP32/64 instructions, thus utilizing more computation resources and improving performance.

E. Sensitivity Studies

In this section, we study how KeSCo's performance is affected by different kernel inputs and stream counts. We evaluate all schemes on each benchmark with input sizes ranging from 1 - 10GB of memory occupancy. Figure 11 illustrates the results. KeSCo achieves an average speedup of $1.28\times$ under all inputs, which is almost identical to *Async*. However, *GrSched*'s performance is unstable under varying input sizes. This is because *GrSched*'s kernel overlap depends on the ratio of dynamic scheduling overhead to the time cost of kernels, which varies greatly with input size.

For the robustness study on CUDA stream counts used in KeSCo, we evaluate three benchmarks whose DFG maximum widths are over five. Figure 12 details the results under the launch setting where KeSCo gains the highest speedup. For B&S, KeSCo achieves the highest speedup $2.76\times$ with six streams and downgrades as the stream increases. We used Nvidia Nsight Systems to profile and observe that up to seven kernels execute simultaneously on GPU, indicating the hardware resource is saturated by seven kernels. Therefore increasing streams aggravates resource competition and reduces performance. For M1 and M2, the max concurrency of kernels is also limited to eight for the same reason.

F. Speedup in Multi-process Scenarios

To evaluate how much KeSCo can deliver acceleration in multi-process scenarios where sub-programs have different priorities, we construct two in-house micro-benchmarks with workloads from Table I. We compare KeSCo with ❶ the *Baseline* scheme where sub-programs will be launched independently on GPU at the same time, and ❷ the scheme where Nvidia MPS[15] is enabled to transparently schedule these sub-programs. Figure 13 compares the speedup of each sub-program and the whole application gained by these three schemes. On MP-1, KeSCo outperforms *MPS* by 23.1% overall and 45.41% for IMG, the sub-program with the highest priority. For VEC1 and VEC2, sub-programs with lower priority, KeSCo achieves a speedup of $1.22\times$, which is only 4.01% lower than that of *MPS*. On MP-2, KeSCo achieves a speedup of $1.13\times$ for the whole workload, while *MPS*'s performance drops by 5.21%. This is because *MPS* launches B&S, the sub-program with the lowest priority, earlier than KeSCo does, causing resource conflicts and performance penalties. For ML, the sub-program with the highest priority, KeSCo achieves a speedup of $1.14\times$ while *MPS* causes a 4.95% performance drop. In summary, *MPS* has the merit of transparent scheduling at runtime but it lacks the ability to prioritize tasks from a global perspective. KeSCo, on the other hand, is a compiler-based approach that enables this global optimization, with the help of user hints.

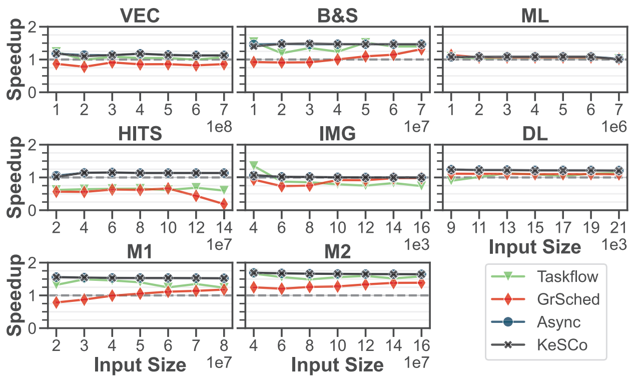


Fig. 11: Average speedup achieved by different schemes under different input sizes and multiple launch settings.

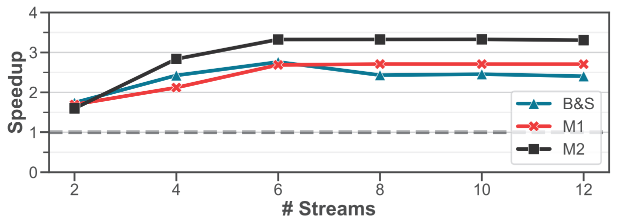
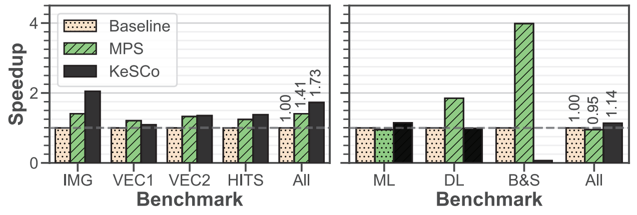


Fig. 12: Speedup achieved by KeSCo with different number of CUDA streams in B&S, M1 and M2.



(a) MP-1 consists of one IMG, two VEC and one HITS. The priority is IMG > VEC > HITS.

(b) MP-2 consists of one ML, one DL and one B&S. The priority is ML > DL > B&S.

Fig. 13: Speedup in two *MPS* programs.

V. Related Work

A. Concurrent Kernel Execution

CKE has been studied in fine granularity widely. Elastic kernel [1] slices kernels into multiple small ones and deploys them on different SM to speed up. Similarly, OpenMP [21] is leveraged to decompose kernels into multiple tasks in Junggler [11] and schedule tasks with dependencies via runtime mechanism. Also, Pagoda [22] concurrently executes narrow tasks at warp-level by virtualizing GPU resources and issues kernels when required resources are available. Targeting at real-world applications, RAMMER [3] proposes a DNN compiler for joint optimization of inter- and intra-kernel concurrency. Taskflow[4] wraps GPU programming model APIs and implements a static scheduler in the framework. A GrCUDA [6]-based runtime scheduler [7] eases prototyping of parallel applications. Distinguishing from those prior arts, our proposed KeSCo aims to statically automate kernel scheduling in multi-task programs, achieving much better performance with reduced programming burden.

B. Task Scheduling

A bunch of task schedulers have been proposed to optimize multi-task applications. On hardware level, new APIs are introduced in [23] to heterogeneous system architecture (HSA) for applications specifying task priority. Chimera [12] extends SM scheduler to estimate the cost of kernel preemption to minimize the overhead. Similarly, command buffer and status table are further embedded in SM scheduler [13] to minimize the overhead for prioritized tasks. On software level, FELP [24] leverages a compiler-runtime system to control task preemption at kernel level. EffiSha [10] schedules kernels at thread-block level dynamically with an online cost model. Pegasus [2] proposes a hypervisor that offers a virtualized accelerator's interface to schedule heterogeneous tasks coordinately in virtual machines. CASE [25] introduces a novel compiler-based approach for scheduling uncooperative tasks, which is not applicable for the dependent task scheduling in our scenario. While prior arts necessitate dedicated systems for scheduling, `KeSCo` seamlessly integrates with compiler and eases the use of task-level concurrency.

VI. Conclusion

The paper proposes `KeSCo` to automatically enable CKE in multi-task applications at compile-time with trivial programming effort. `KeSCo` constructs the DFG from source code, distributes kernels across multiple streams, and generates synchronization barriers to achieve load balance and low synchronization cost. Moreover, the design is extended to multi-process scenarios for prioritized sub-program scheduling. Experimental results demonstrate that `KeSCo` effectively boosts performance and eases programming burden.

Acknowledgment

We thank the anonymous reviewers for their constructive comments and suggestions. This research was supported by the National Key R&D Program of China under Grant NO. 2022ZD0115304, the National Natural Science Foundation of China-#62102465, the Major Program of Guangdong Basic and Applied Research NO. 2019B030302002, the Funding by Science and Technology Projects in Guangzhou-#202201011241, Open Project of China Electronic Product Reliability and Environmental Testing Research Institute (CEPREI)-#HK202201334, and CCF-Phytium Fund (CCF-Phytium202204).

References

[1] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving gpgpu concurrency with elastic kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 407–418, New York, NY, USA, 2013.

[2] Vishakha Gupta, Karsten Schwan, and Niraj Tolia et al. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *USENIX Annual Technical Conference*, Portland, OR, USA, 2011.

[3] Lingxiao Ma, Zhiqiang Xie, and Zhi Yang et al. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, USA, 2020.

[4] Tsung-Wei Huang, Dian-Lun Lin, and Chun-Xun Lin et al. Taskflow: A lightweight parallel and heterogeneous task graph computing system. *IEEE Transactions on Parallel and Distributed Systems*, 33(6):1303–1320, 2022.

[5] NVIDIA. Cuda c++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html(Retrieved2023.06), 2023.

[6] NVIDIA. grcuda: Polyglot gpu access in graalvm. https://github.com/NVIDIA/grcuda, 2020 (Retrieved 2023.06).

[7] Alberto Parravicini, Arnaud Delamare, and Marco Arnaboldi et al. Dag-based scheduling with resource sharing for multi-task applications in a polyglot GPU runtime. In *35th IEEE International Parallel and Distributed Processing Symposium*, pages 111–120, Portland, OR, USA, 2021.

[8] Tianqi Chen, Thierry Moreau, and Ziheng Jiang et al. Tvm: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 579–594, USA, 2018.

[9] Google. Xla: Optimizing compiler for machine learning. https://www.tensorflow.org/xla), 2017 (Retrieved 2023.06).

[10] Guoyang Chen, Yue Zhao, and Xipeng Shen et al. EffiSha: A Software Framework for Enabling Effficient Preemptive Scheduling of GPU. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 3–16, Austin Texas USA, 2017.

[11] Mehmet E Belviranli, Seyong Lee, Jeffrey S Vetter, and Laxmi N Bhuyan. Juggler: a dependence-aware task-based execution framework for gpus. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 54–67, 2018.

[12] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 593–606, New York, NY, USA, 2015.

[13] Ivan Tanasic, Isaac Gelado, and Javier Cabezas et al. Enabling preemptive multiprogramming on gpus. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, page 193–204, Minneapolis, Minnesota, USA, 2014.

[14] Jacob T. Adriaens, Katherine Compton, and Nam Sung Kim et al. The case for gpgpu spatial multitasking. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, New Orleans, LA, USA, 2012.

[15] NVIDIA. Multi-process service. https://docs.nvidia.com/deploy/mps/index.html, 2022 (Retrieved 2023.06).

[16] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, pages 75–86, San Jose, CA, USA, 2004.

[17] AMD. Hip:heterogeneous interface for portability. https://github.com/ROCm-Developer-Tools/HIP, 2023 (Retrieved 2023.06).

[18] The Khronos SYCL Working Group. Sycl 2020 specification (revision 7). https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf, 2023 (Retrieved 2023.06).

[19] NVIDIA. Faster transformer. https://github.com/NVIDIA/FasterTransformer, 2023 (Retrieved 2023.06).

[20] NVIDIA. Manage and monitor gpus in cluster environments. https://developer.nvidia.com/dcgm (Retrieved 2023.06).

[21] OpenMP. Openmp. https://www.openmp.org/, 2023 (Retrieved 2023.06).

[22] Tsung Tai Yeh, Amit Sabne, and Putt Sakdhnagool et al. Pagoda: Fine-grained gpu resource virtualization for narrow tasks. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 221–234, New York, NY, USA, 2017.

[23] Sooraj Puthoor, Xulong Tang, and Joseph Gross et al. Oversubscribed Command Queues in GPUs. In *Proceedings of the 11th Workshop on General Purpose GPUs*, pages 50–60, New York, NY, USA, 2018.

[24] Bo Wu, Xu Liu, and Xiaobo Zhou et al. FLEP: Enabling Flexible and Efficient Preemption on GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 483–496, Xi'an China, 2017.

[25] Chao Chen, Chris Porter, and Santosh Pande. Case: A compiler-assisted scheduling framework for multi-gpu systems. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 17–31, 2022.