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

Zejia Lin[§†], Zewei Mo[§‡], Xuanteng Huang[†],
Xianwei Zhang[#†], Yutong Lu[†]

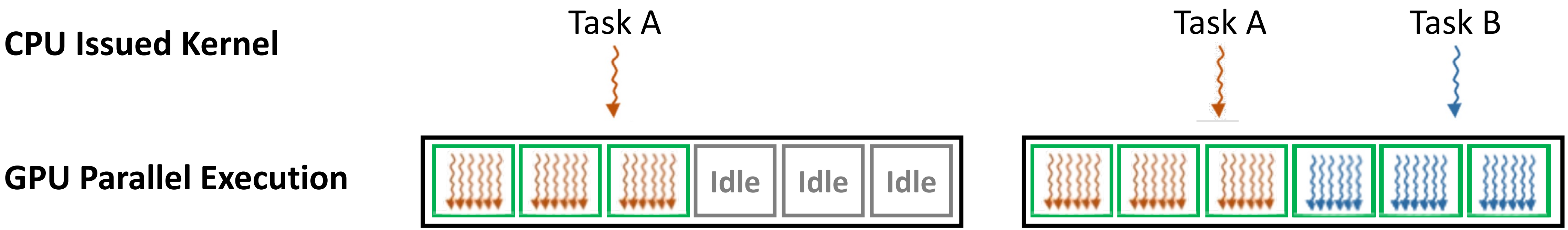
†Sun Yat-sen University, ‡University of Pittsburgh
Email: linzj39@mail2.sysu.edu.cn

§ Equal contribution
† Work done when studying at Sun Yat-sen University
# Corresponding author

Background

- **GPU is mainly known for its data-level parallelism**
  - Thousands of cores, with thousands of outstanding threads
  - Massively parallel computation

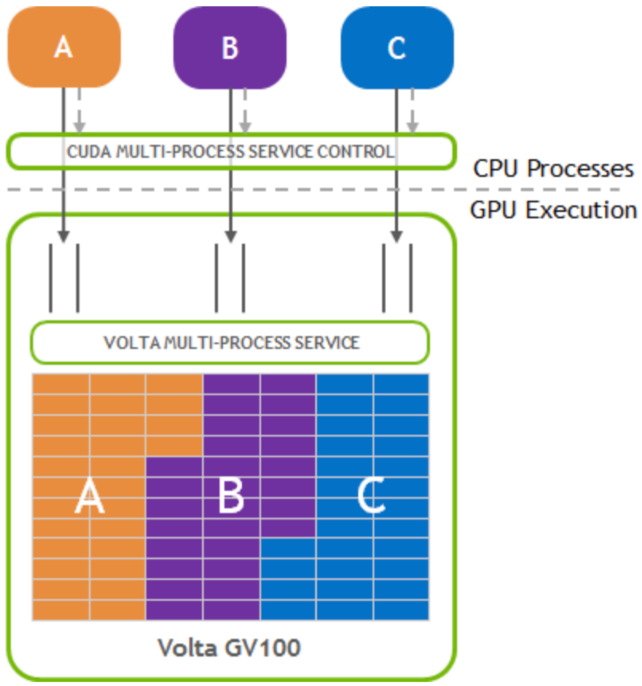
- **Still need kernel-level parallelism**
  - GPU is underutilized by a single application process
  - Executing independent kernels in parallel ⟹ Improve utilization



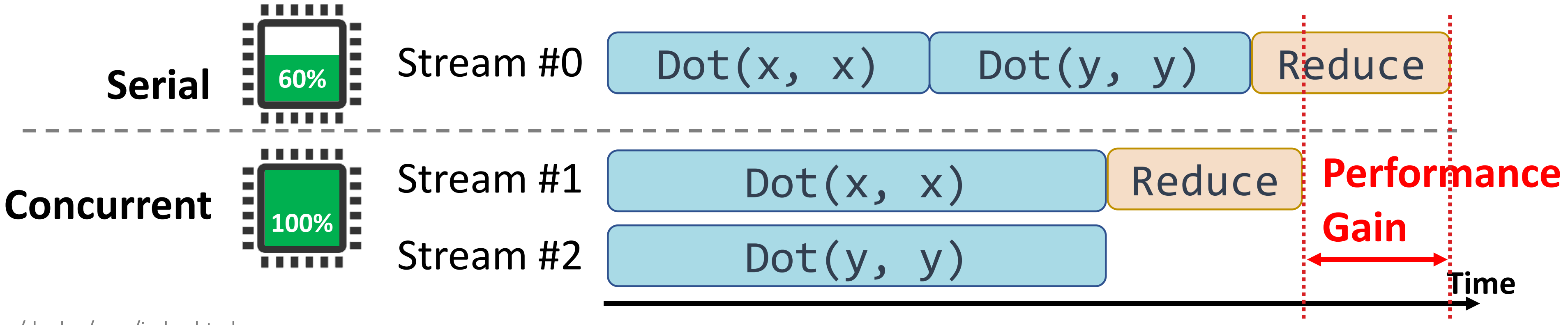
Concurrent Kernel Execution (CKE)

- **Techniques**
  - Vendor provided multi-process service (MPS)[1]
  - Stream / Task queue in programming models

- **Asynchronous queues in GPU programming models**
  - CUDA stream / graph[1]
  - HIP stream / graph[2]
  - SYCL command queue[3]



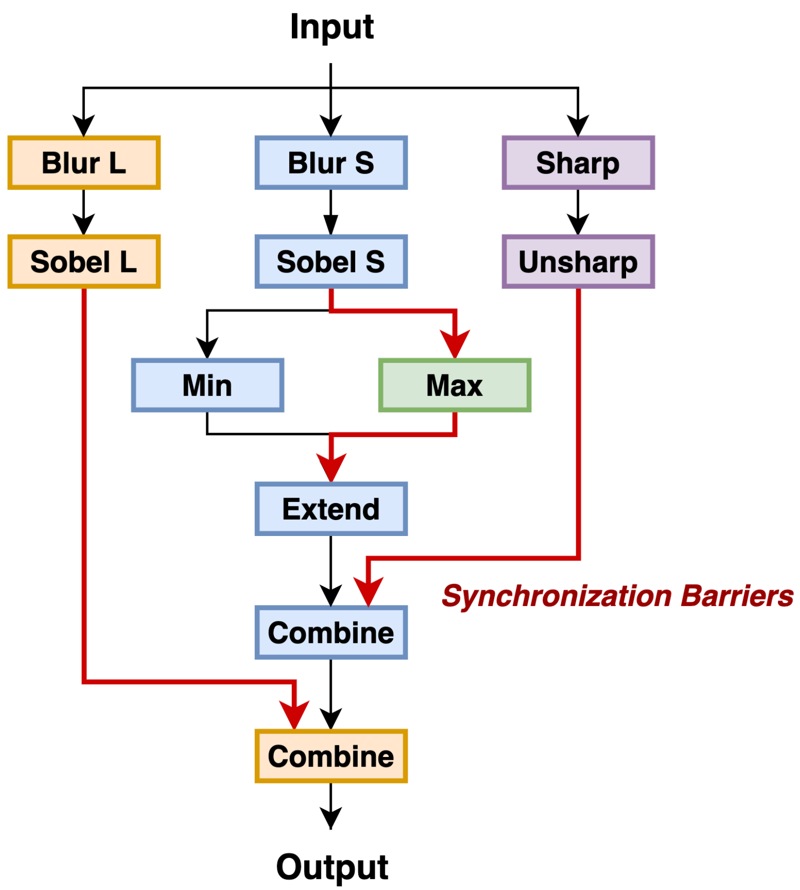
Credits: [1]



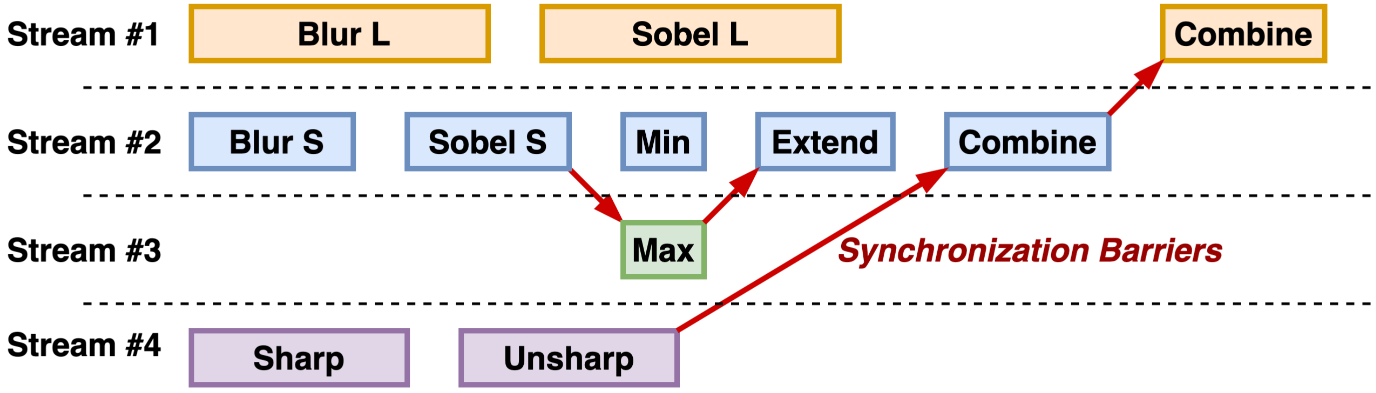
[1] https://docs.nvidia.com/deploy/mps/index.html

Example: Transforming Serial Code into CKE

Image process pipeline

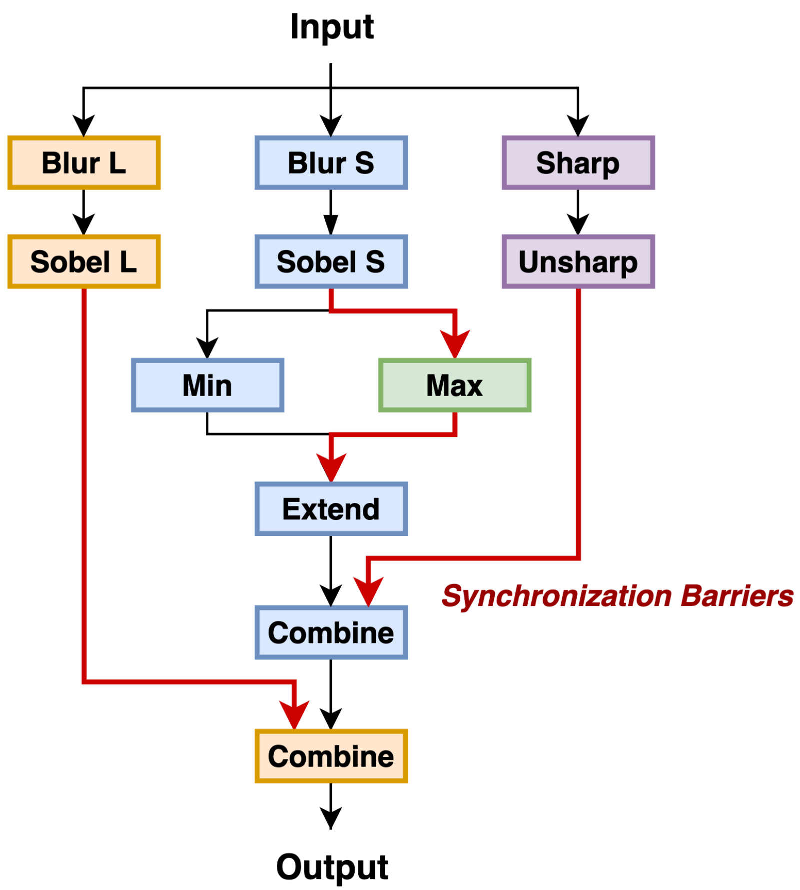


Assign kernels to multiple streams (software task queue)



Example: Transforming Serial Code into CKE (cont.)

Image process pipeline



Pseudo serial code

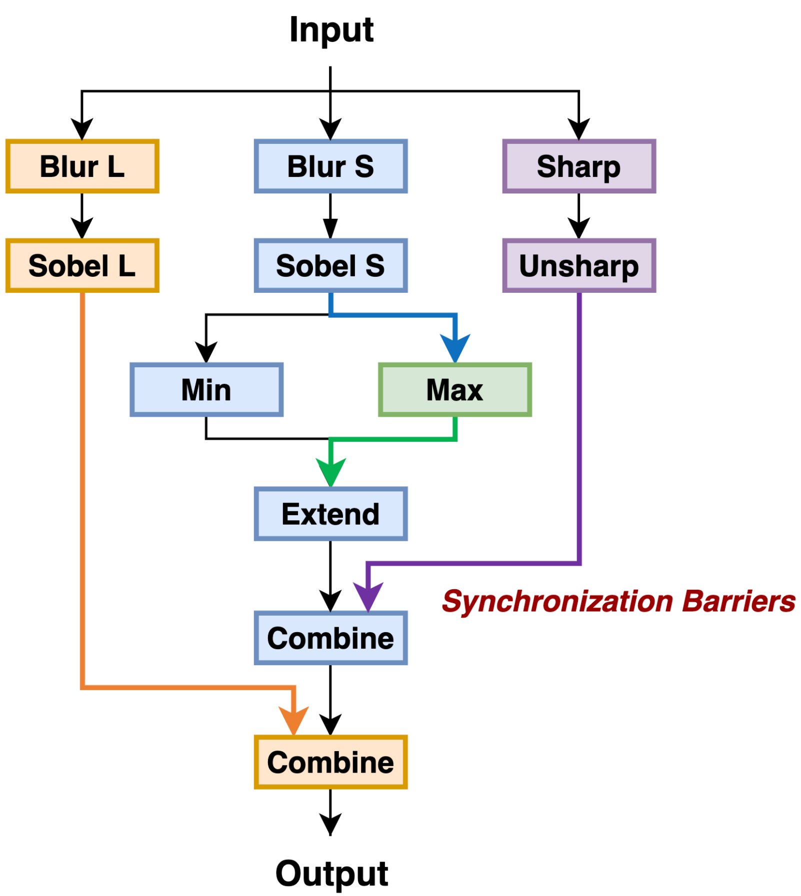
```
void Sync_IMG( … ) {
    blur( … );
    blur( … );
    sharp( … );
    sobel( … );
    sobel( … );
    unsharpen( … );
    max( … );
    min( … );
    extend( … );
    combine( … );
    combine( … );
}
```

First glance

- 11 kernels
- Massive dependency
- Error-prone refactoring
- ……

Example: Transforming Serial Code into CKE (cont.)

Image process pipeline



Pseudo serial code

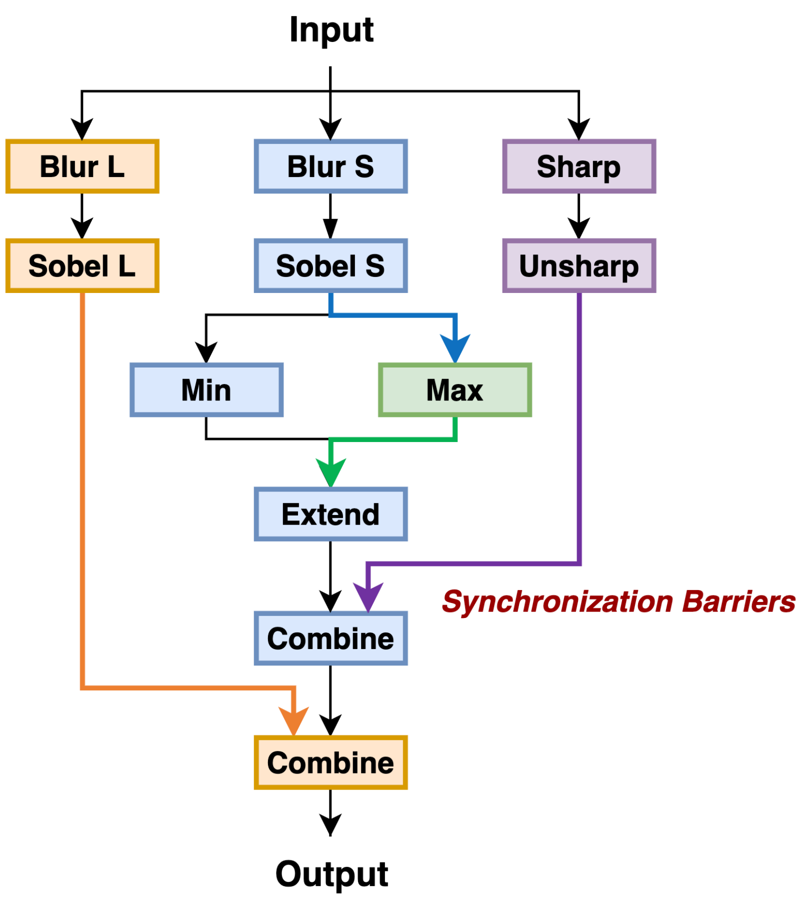
```
void Sync_IMG( … ) {
    blur( … );
    blur( … );
    sharp( … );
    sobel( … );
    sobel( … );
    unsharpen( … );
    max( … );
    min( … );
    extend( … );
    combine( … );
    combine( … );
}
```

Non-trivial Efforts

- Dependence analysis

Example: Transforming Serial Code into CKE (cont.)

Image process pipeline



Pseudo async code

```
void Async_IMG( … ) {
    // create streams and events
    blur( 1, … );
    blur( 2, … );
    sharp ( 3, … );

    sobel ( 1, … );
    sobel ( 2, … );

    max ( 4, … );
    min ( 2, … );

    extend ( 2, … );
    unsharpen ( 3, … );

    combine ( 2, … );
    combine ( 1, … );
}
```

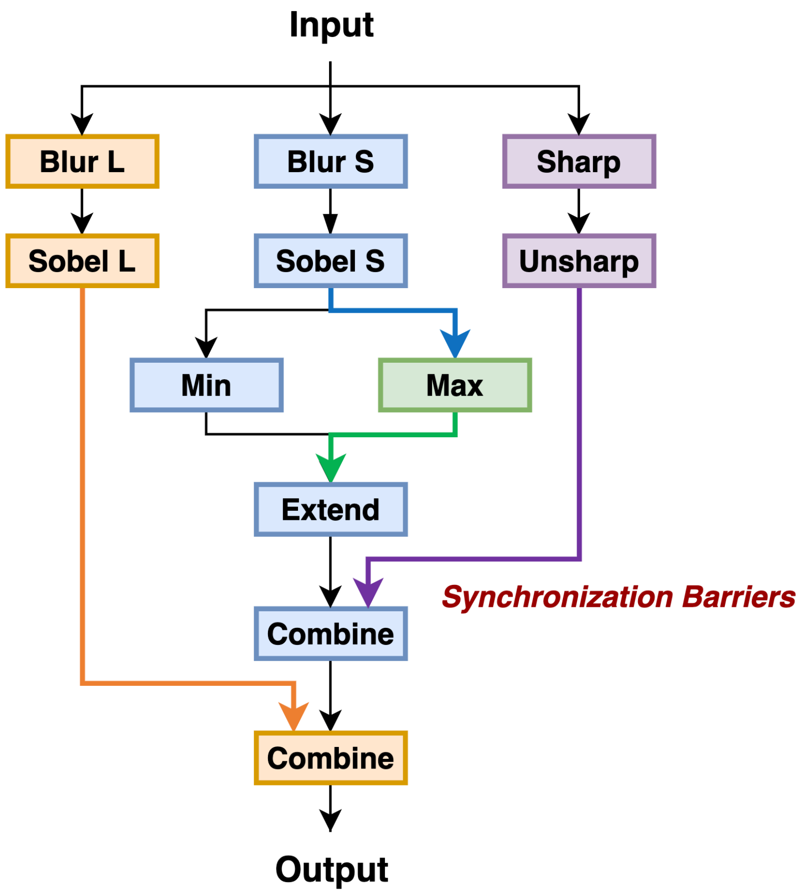
Stream ID

Non-trivial Efforts

- Dependence analysis
- Scheduling
- Stream assignment

Example: Transforming Serial Code into CKE (cont.)

Image process pipeline



Pseudo async code

```
void Async_IMG( … ) {
    // create streams and events
    blur( 1, … );
    blur( 2, … );
    sharp ( 3, … );
    ......
    cudaEventRecord(e1, 2);
    cudaStreamWaitEvent(4, e1);
    ......
    cudaEventRecord(e2, 4);
    cudaStreamWaitEvent(2, e2);
    ......
    cudaEventRecord(e3, 3);
    cudaStreamWaitEvent(2, e3);
    ......
    cudaEventRecord(e4, 2);
    cudaStreamWaitEvent(1, e4);
    combine ( 1, … );
}
```

Synchronization Events & Barriers

Non-trivial Efforts

- Dependence analysis
- Scheduling
- Stream assignment
- Synchronization
- ……

Tremendous Programming Burden

Hard to obtain **bug-free** and **performant** code

```
void Sync_IMG( … ) {
    blur( … );
    blur( … );
    sharp( … );
    sobel( … );
    sobel( … );
    unsharpen( … );
    max( … );
    min( … );
    extend( … );
    combine( … );
    combine( … );
}
```

2.8× LoC →

```
void Async_IMG( … ) {
    // create streams and events
    blur( 1, … );
    blur( 2, … );
    sharp ( 3, … );
    ......
    cudaEventRecord(e1, 2);
    cudaStreamWaitEvent(4, e1);
    ......
    cudaEventRecord(e2, 4);
    cudaStreamWaitEvent(2, e2);
    ......
    cudaEventRecord(e3, 3);
    cudaStreamWaitEvent(2, e3);
    ......
    cudaEventRecord(e4, 2);
    cudaStreamWaitEvent(1, e4);
    combine ( 1, … );
}
```

Non-trivial Efforts

- Dependence analysis
- Scheduling
- Stream assignment
- Synchronization
- ……

Tremendous Programming Burden (cont.)

- **Optimization**
  - When and where to issue kernel
  - Efficient overlap with computation and data transfer
  - ......

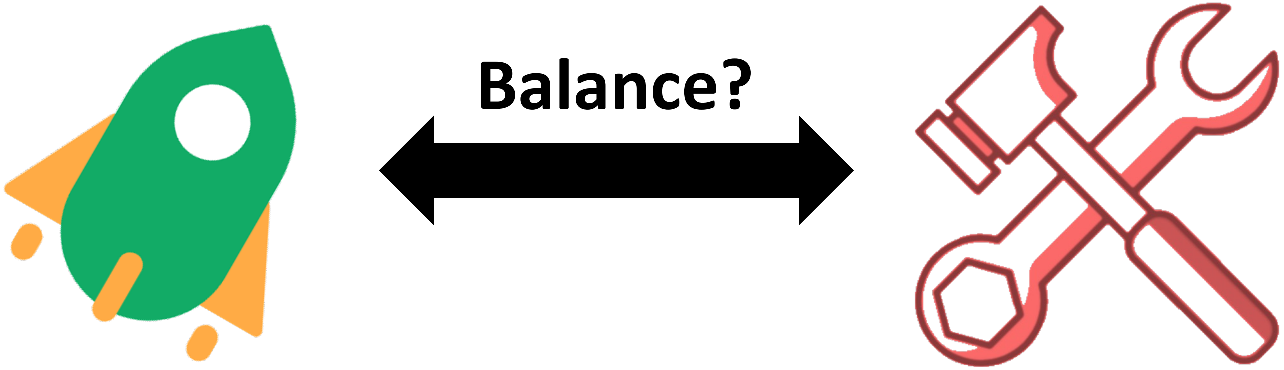
- **Optimal scheduling improves performance, comes with cumbersome manual efforts**
  - Understanding the code
  - Identifying optimization opportunities
  - Refactoring the code
  - ......

[1] Nvidia. CUDA C++ Programming Guide
[2] AMD. HIP Runtime API Reference
[3] Khronos. SYCL 2020 Specification

Tremendous Programming Burden (cont.)

- **Optimization**
  - When and where to issue kernel
  - Efficient overlap with computation and data transfer
  - ……

- **Optimal scheduling improves performance, comes with cumbersome manual efforts**



[1] Nvidia. CUDA C++ Programming Guide
[2] AMD. HIP Runtime API Reference
[3] Khronos. SYCL 2020 Specification

Observation Ⅰ: Regular Workflow Patterns

Wrap up vendor's API to ease multi-tasking

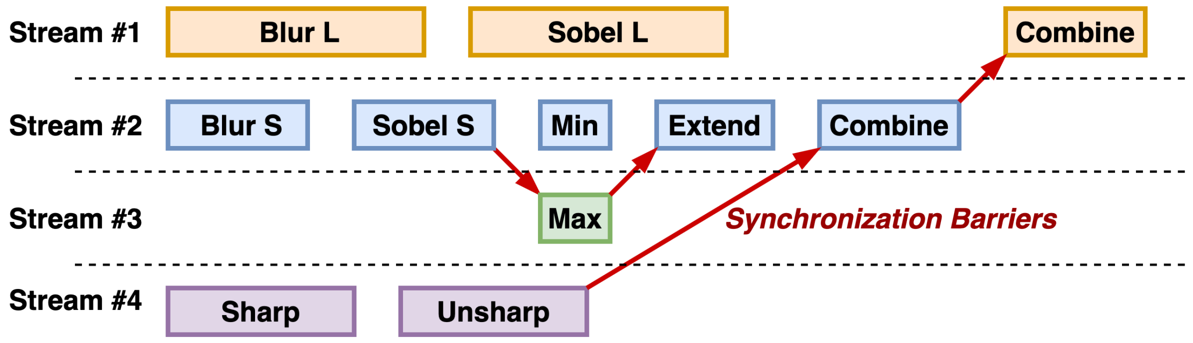
- Taskflow[1] ⟹ cudaGraph + scheduler implemented in C++ wrapper API
- GrSched[2] ⟹ cudaStream + scheduler implemented in language VM

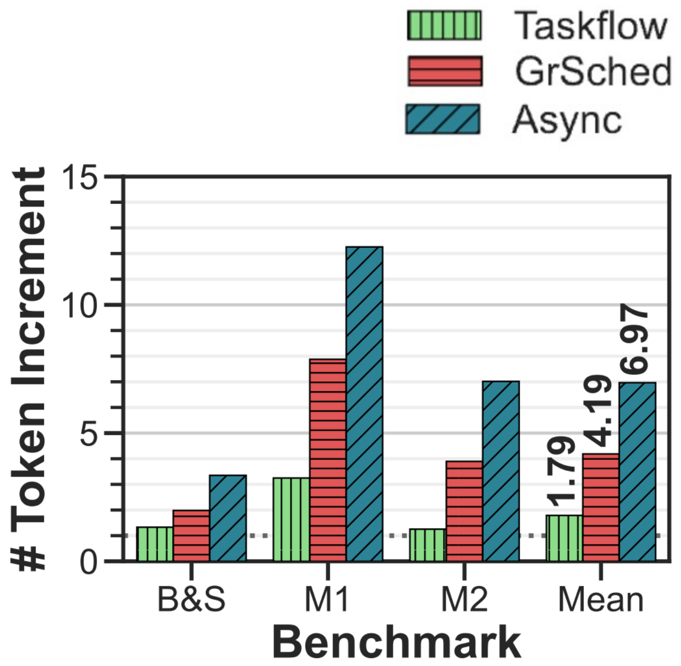
Similar workflow in implementing CKE

❶ *Dependence analysis*

❷ *Assign kernel to stream*

❸ *Create synchronization barrier*





[1] Tsung-Wei Huang et al. Taskflow: A lightweight parallel and heterogeneous task graph computing system. IEEE Transactions on Parallel and Distributed Systems

[2] Alberto Parravicini et al. Dag-based scheduling with resource sharing for multi-task applications in a polyglot GPU runtime. IPDPS 2021

Observation Ⅱ: Performance Downgrade

Wrap up vendor's API to ease multi-tasking

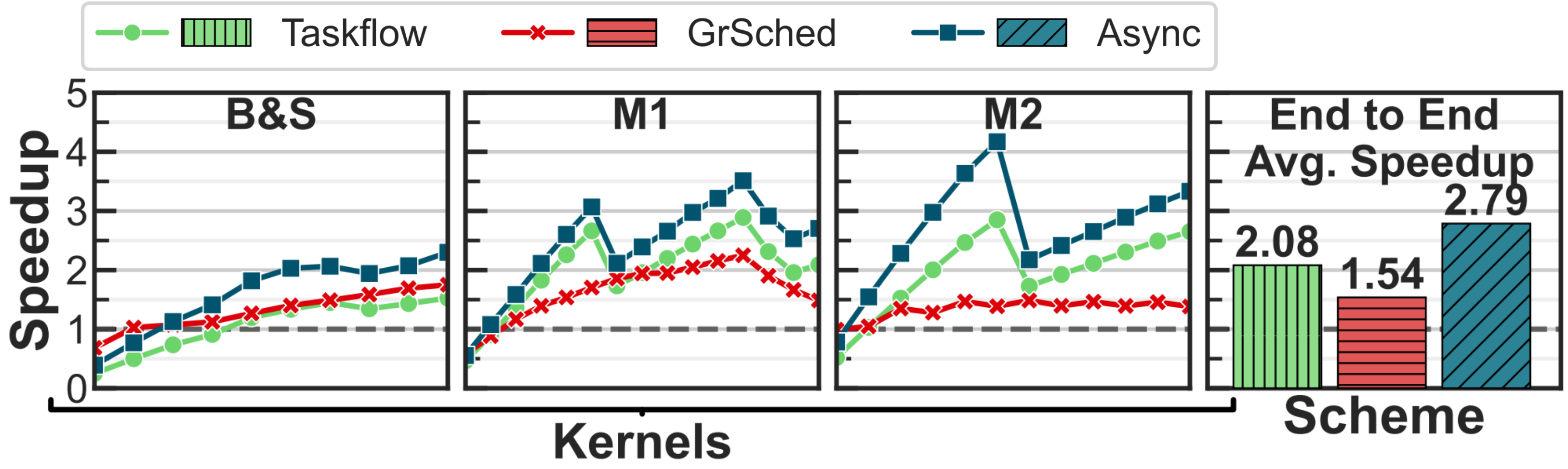
- Taskflow[1] ⟹ cudaGraph + scheduler implemented in C++ wrapper API
- GrSched[2] ⟹ cudaStream + scheduler implemented in language VM

Runtime scheduling brings overhead

❶ *Dependence analysis* ⟹ Runtime task graph construction

❷ *Assign kernel to stream* ⟹ Runtime schedule decision

❸ *Create synchronization barrier* ⟹ Also a part of task graph construction



[1] Tsung-Wei Huang et al. Taskflow: A lightweight parallel and heterogeneous task graph computing system. IEEE Transactions on Parallel and Distributed Systems

[2] Alberto Parravicini et al. Dag-based scheduling with resource sharing for multi-task applications in a polyglot GPU runtime. IPDPS 2021

Opportunity: Compiler for Automation

Schedule the execution at compile-time

- Automatic dependence analysis
- Compile-time scheduling
- Stream and synchronization management

❶ *Dependence analysis* ⟹ Runtime task graph construction

❷ *Assign kernel to stream* ⟹ Runtime schedule decision

❸ *Create synchronization barrier* ⟹ Also a part of task graph construction

Laborious work **Runtime overhead**

Use compiler to automate the workflow with no runtime overhead

Challenges

Sheduling machanism

- How to acheive competent **performance** against manual-optimized code?

Extensibility

- How to co-schedule **independent** tasks to share GPU?

Code transformation

- How is the design **seamlessly integrated** into existing compilation workflow?

KeSCo Overview

Code

Compiler Frontend

Optimizations **KeSCo**

Compiler Backend

Binary

Kernel-level ***Scheduler***

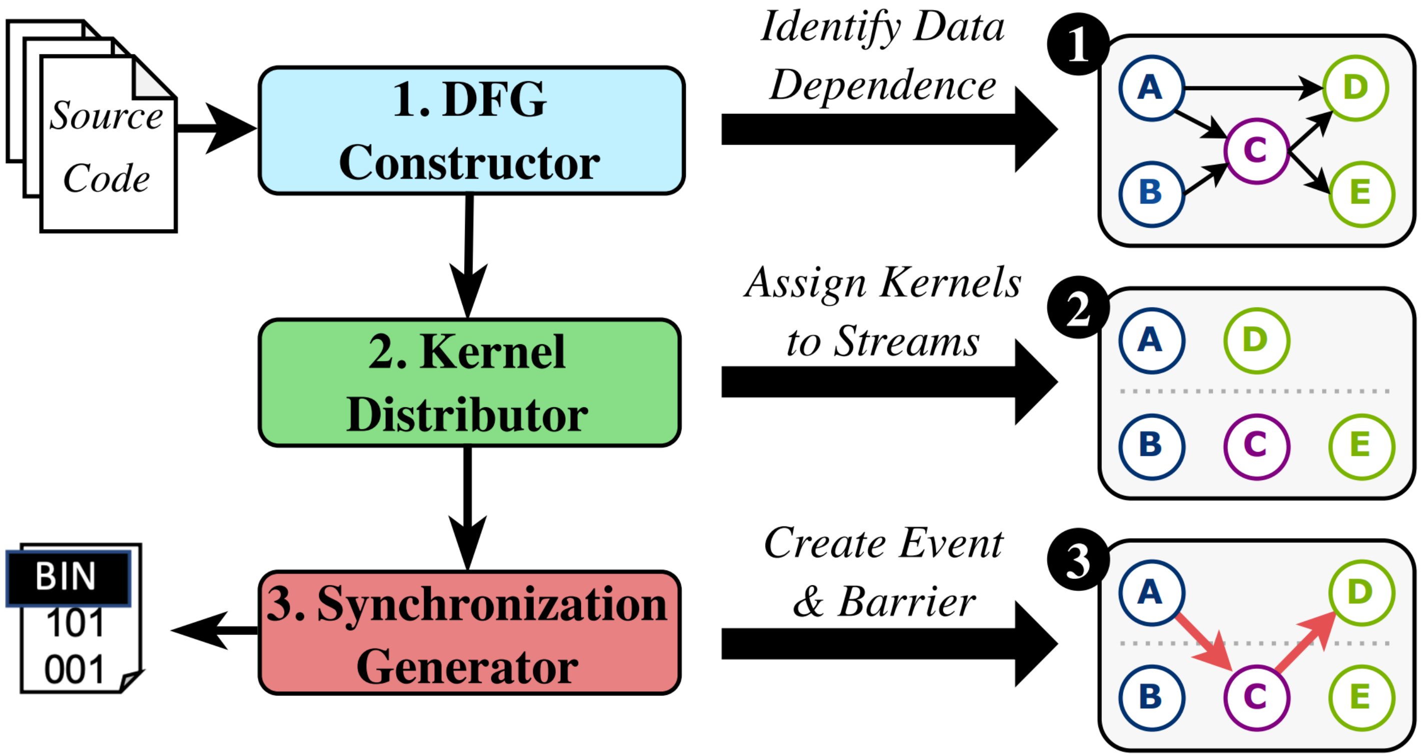
Automatically analyze dependency, rearrange kernels for higher overlap and less synchronization

Task-level ***Scheduler***

Coordinates independent prioritized tasks, extends the kernel-level scheduler to broader usage

KeSCo Overview (cont.)

- **DFG (Data Flow Graph) Constructor:** *analyze kernel dependence*
- **Kernel Distributor:** *where the scheduling happens*
- **Synchronization Generator:** *guarantees correctness of the asynchronous execution*

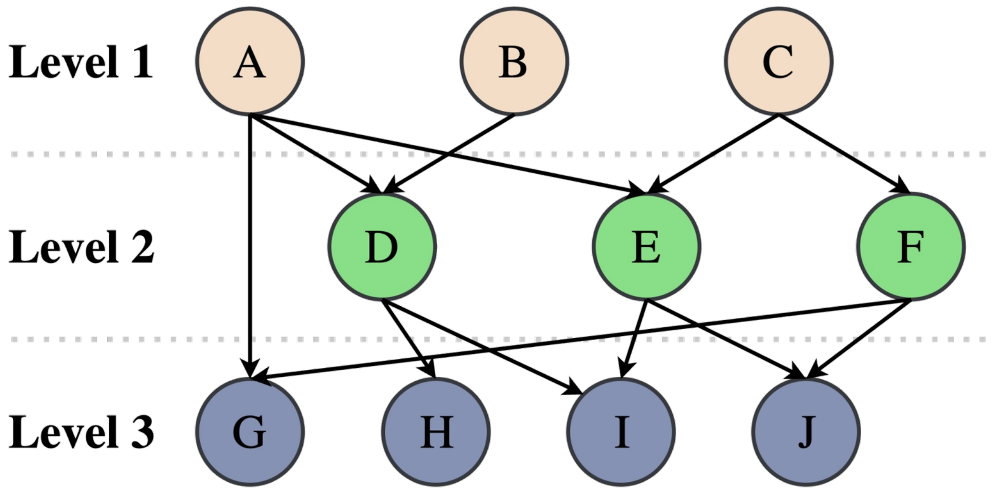


Kernel-level Scheduling

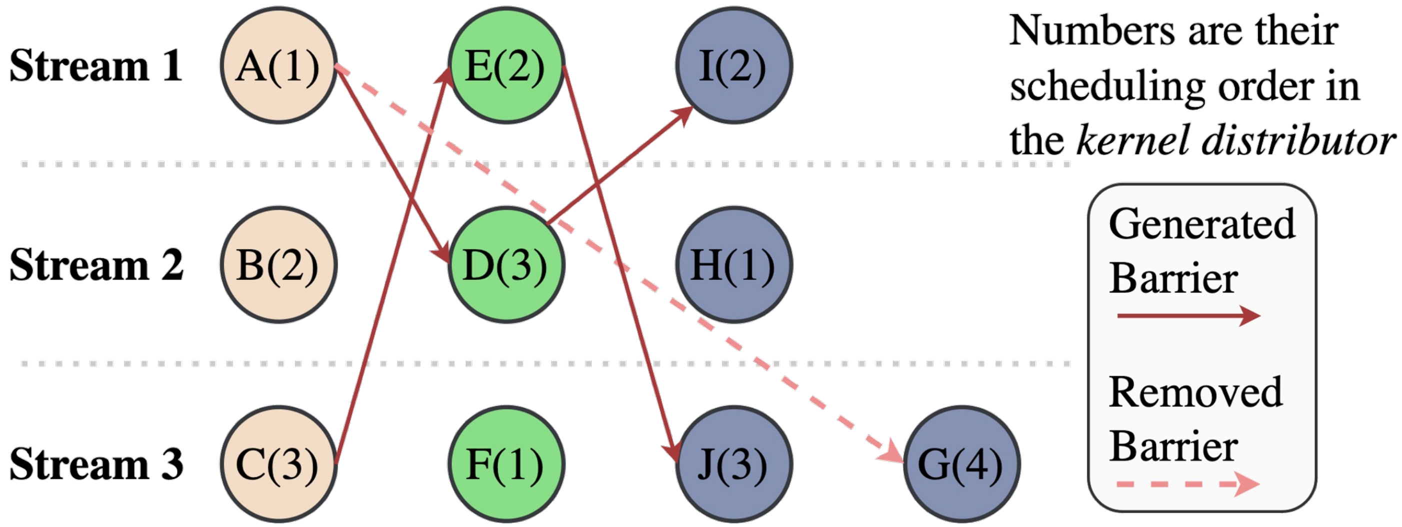
Goal: ❶ Increase overlap ❷ Minimize synchronization ❸ Load balance

Key idea: Issue a kernel immediately after its predecessor whenever feasible

Data Flow Graph



Scheduled Kernels

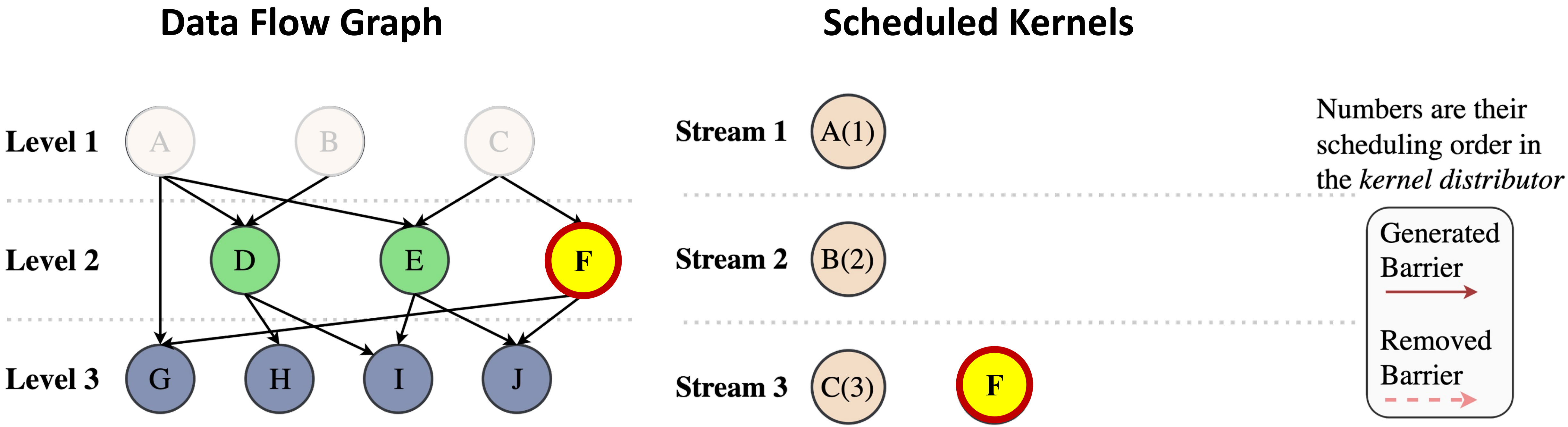


Kernel-level Scheduling (cont.)

Goal: ❶ Increase overlap ❷ Minimize synchronization ❸ Load balance

Key idea: Issue a kernel immediately after its predecessor whenever feasible

Procedure: Kernel F has the least number of predecessors

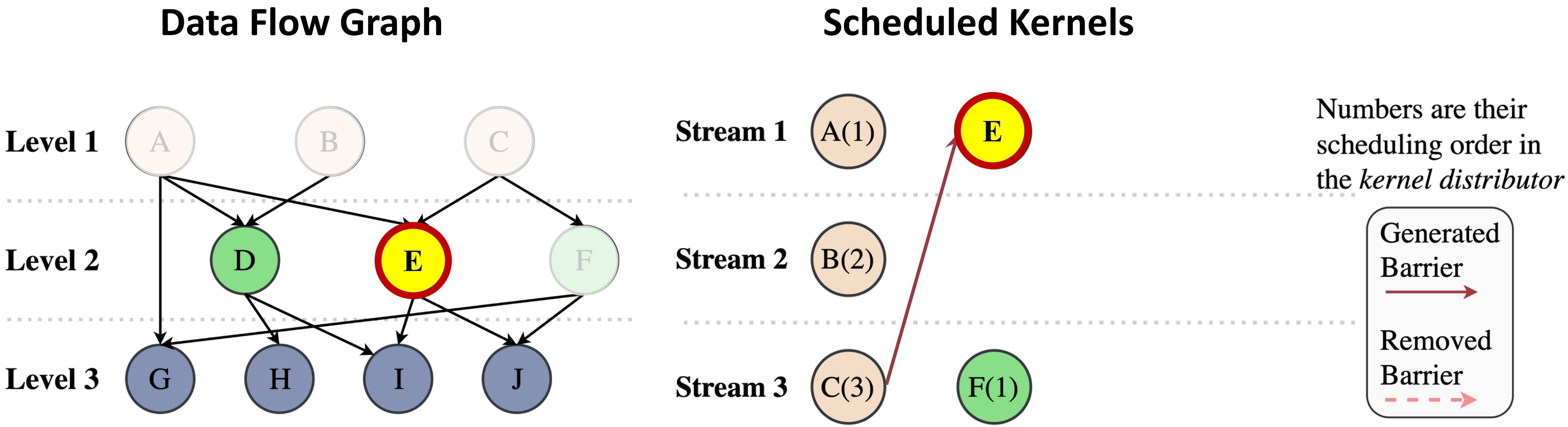


Kernel-level Scheduling (cont.)

Goal: ❶ Increase overlap ❷ Minimize synchronization ❸ Load balance

Key idea: Issue a kernel immediately after its predecessor whenever feasible

Procedure: Kernel E can only be placed after kernel A



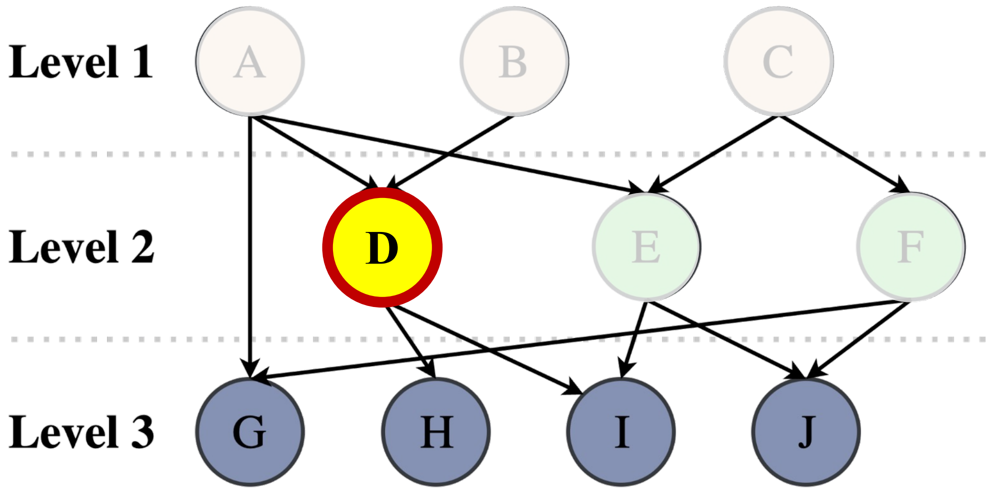
Kernel-level Scheduling (cont.)

Goal: ❶ Increase overlap ❷ Minimize synchronization ❸ Load balance

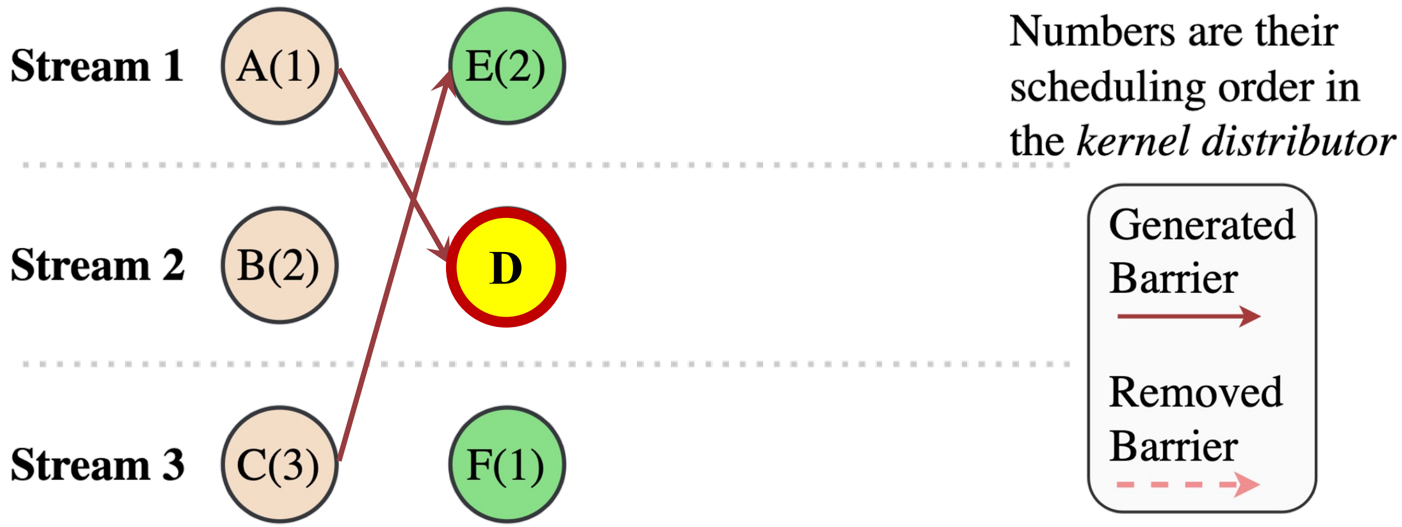
Key idea: Issue a kernel immediately after its predecessor whenever feasible

Procedure: Kernel D positioned in Stream 2 to overlaps with kernel E and F

Data Flow Graph



Scheduled Kernels



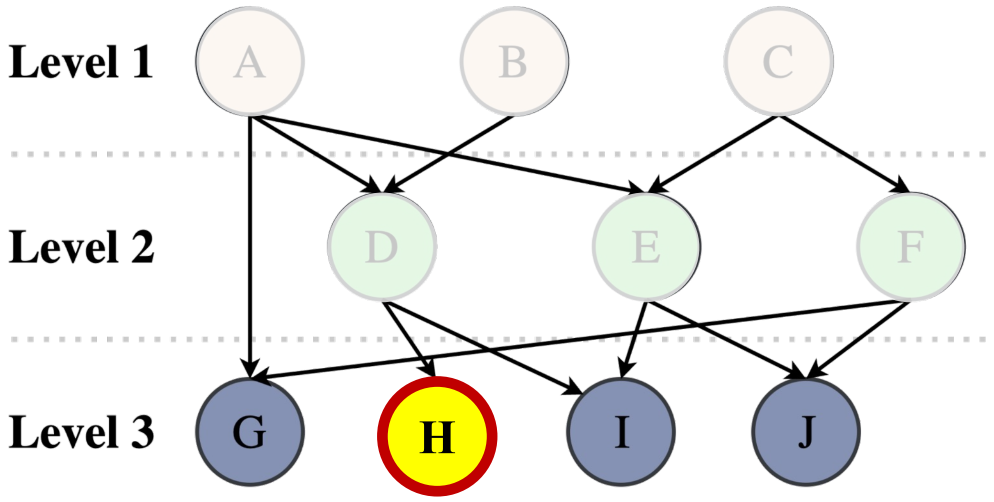
Kernel-level Scheduling (cont.)

Goal: ❶ Increase overlap ❷ Minimize synchronization ❸ Load balance

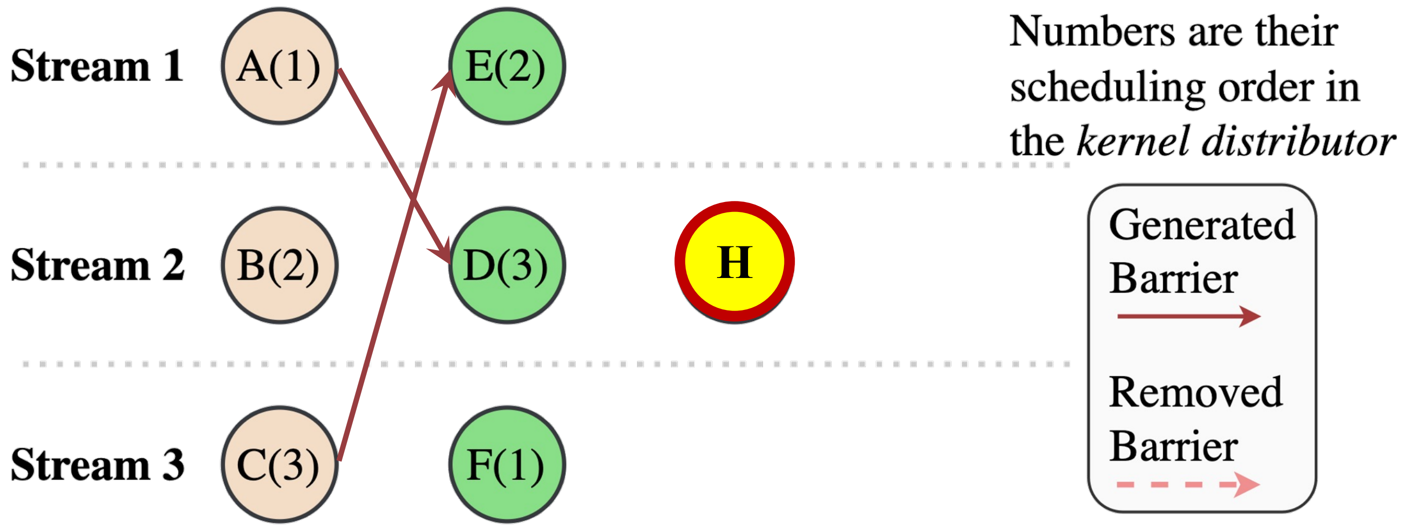
Key idea: Issue a kernel immediately after its predecessor whenever feasible

Procedure: Kernel H has the least number of predecessors

Data Flow Graph



Scheduled Kernels

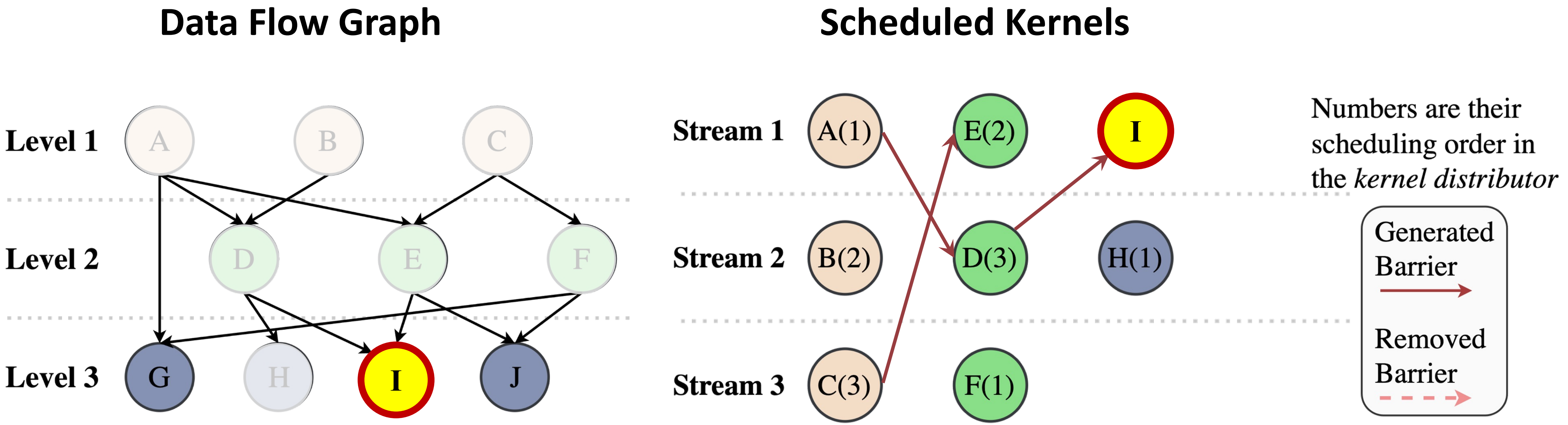


Kernel-level Scheduling (cont.)

Goal: ❶ Increase overlap ❷ Minimize synchronization ❸ Load balance

Key idea: Issue a kernel immediately after its predecessor whenever feasible

Procedure: Rule applied similar to E



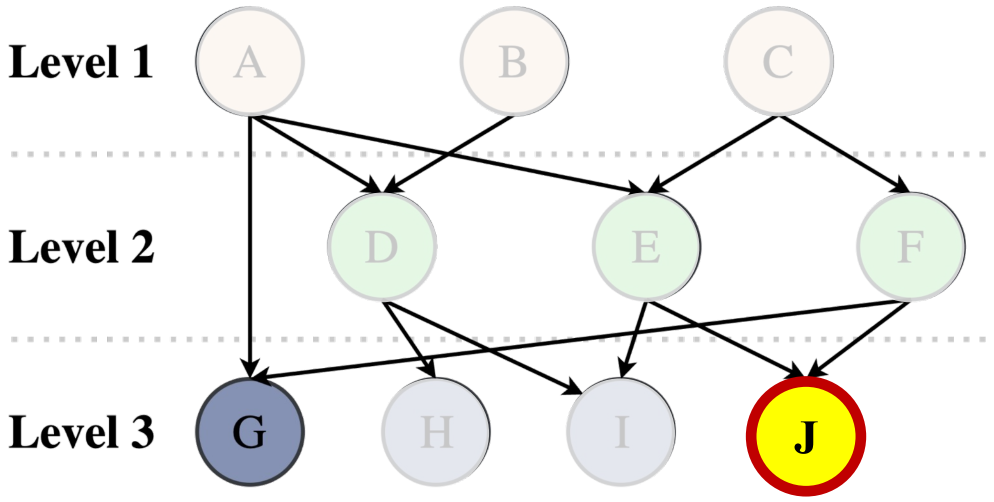
Kernel-level Scheduling (cont.)

Goal: ❶ Increase overlap ❷ Minimize synchronization ❸ Load balance

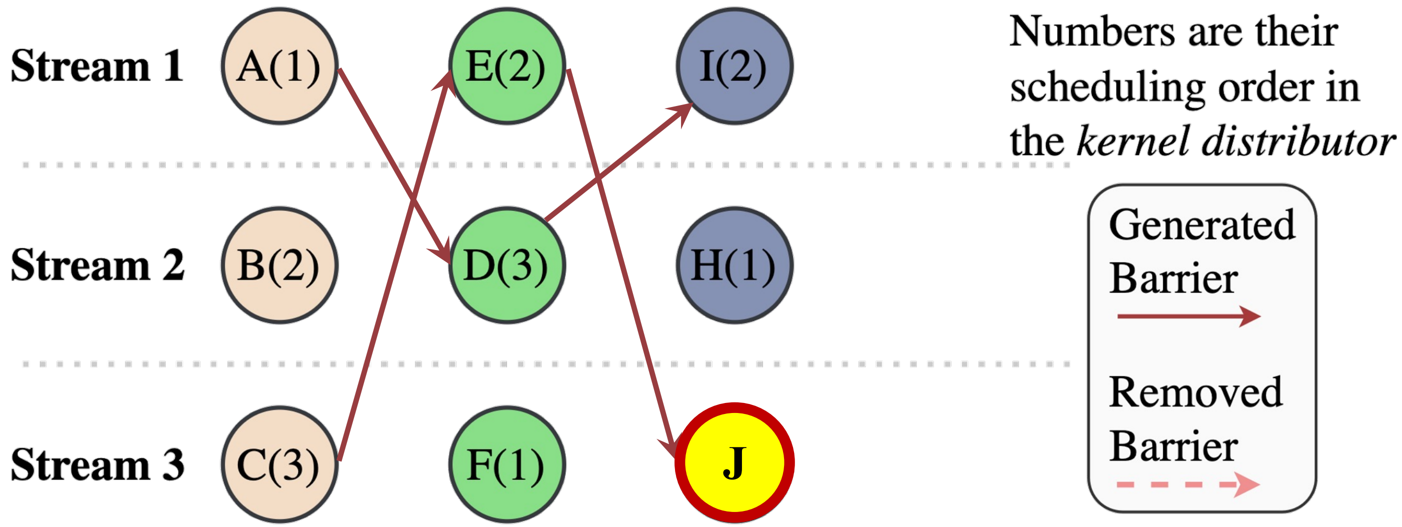
Key idea: Issue a kernel immediately after its predecessor whenever feasible

Procedure: Rule applied similar to E

Data Flow Graph



Scheduled Kernels



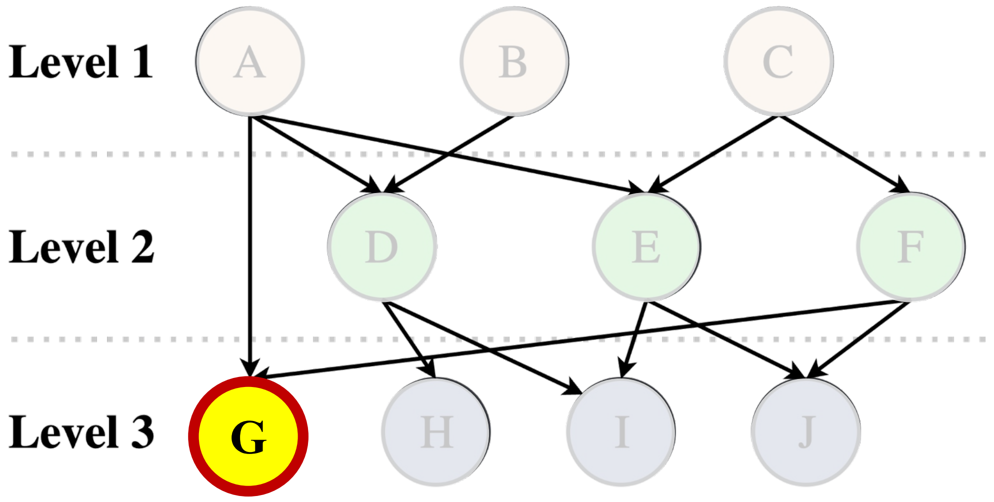
Kernel-level Scheduling (cont.)

Goal: ❶ Increase overlap ❷ Minimize synchronization ❸ Load balance

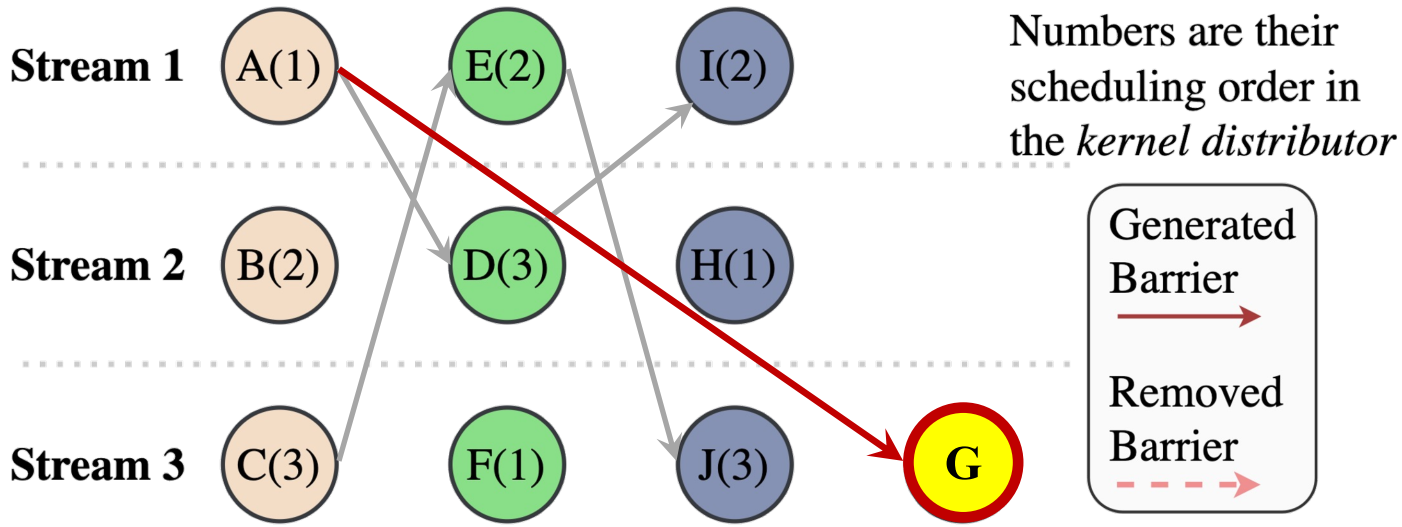
Key idea: Issue a kernel immediately after its predecessor whenever feasible

Procedure: Kernel G has a redundant barrier

Data Flow Graph



Scheduled Kernels

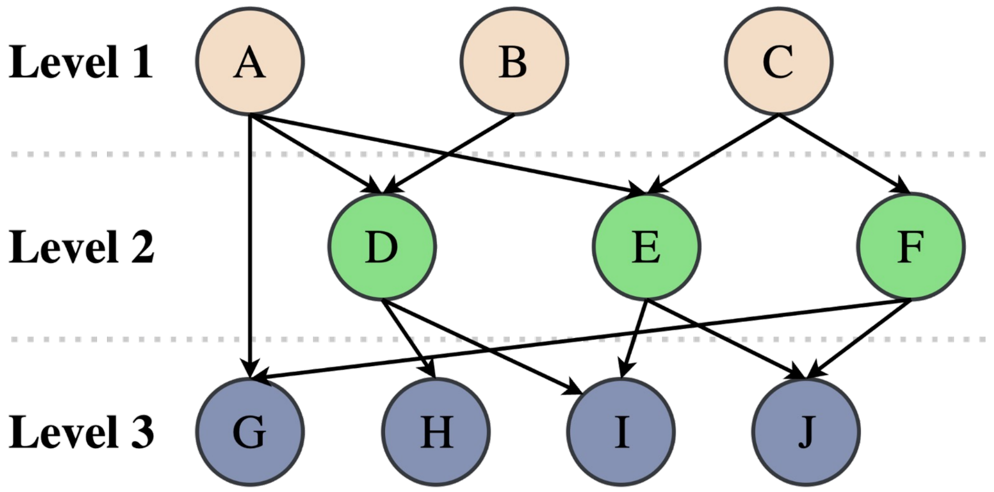


Kernel-level Scheduling (cont.)

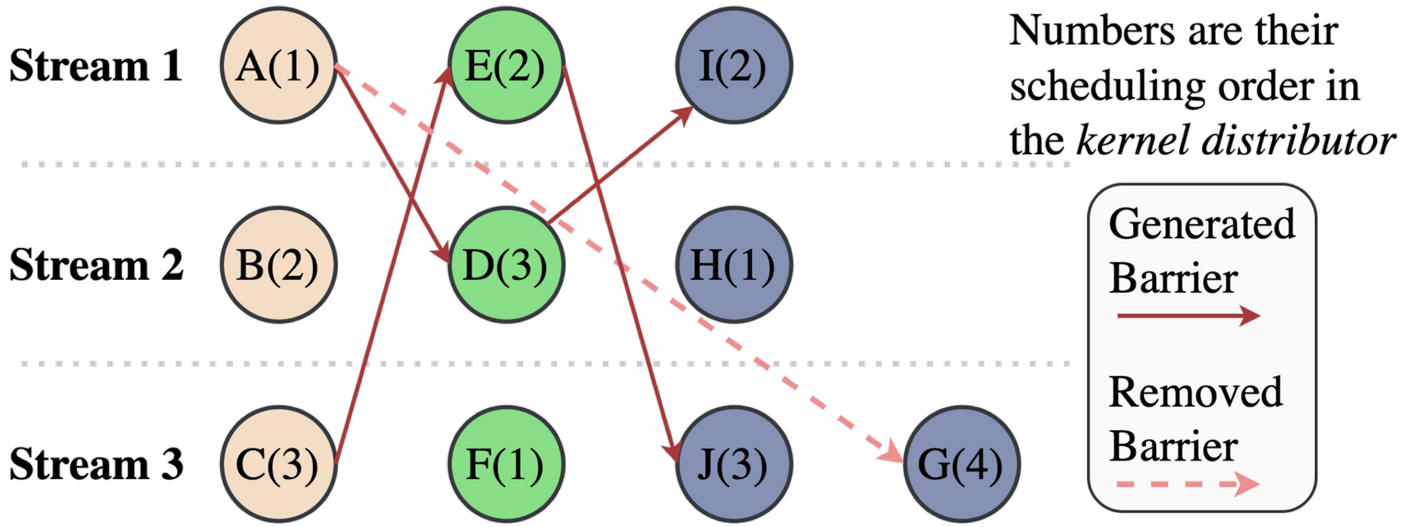
Goal: ❶ Increase overlap ❷ Minimize synchronization ❸ Load balance

Key idea: Issue a kernel immediately after its predecessor whenever feasible

Data Flow Graph



Scheduled Kernels



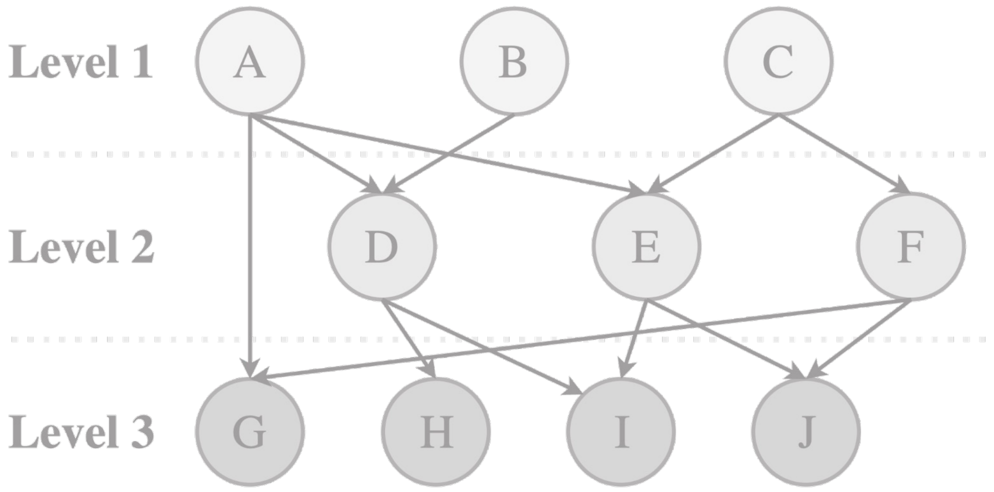
Kernel-level Scheduling (cont.)

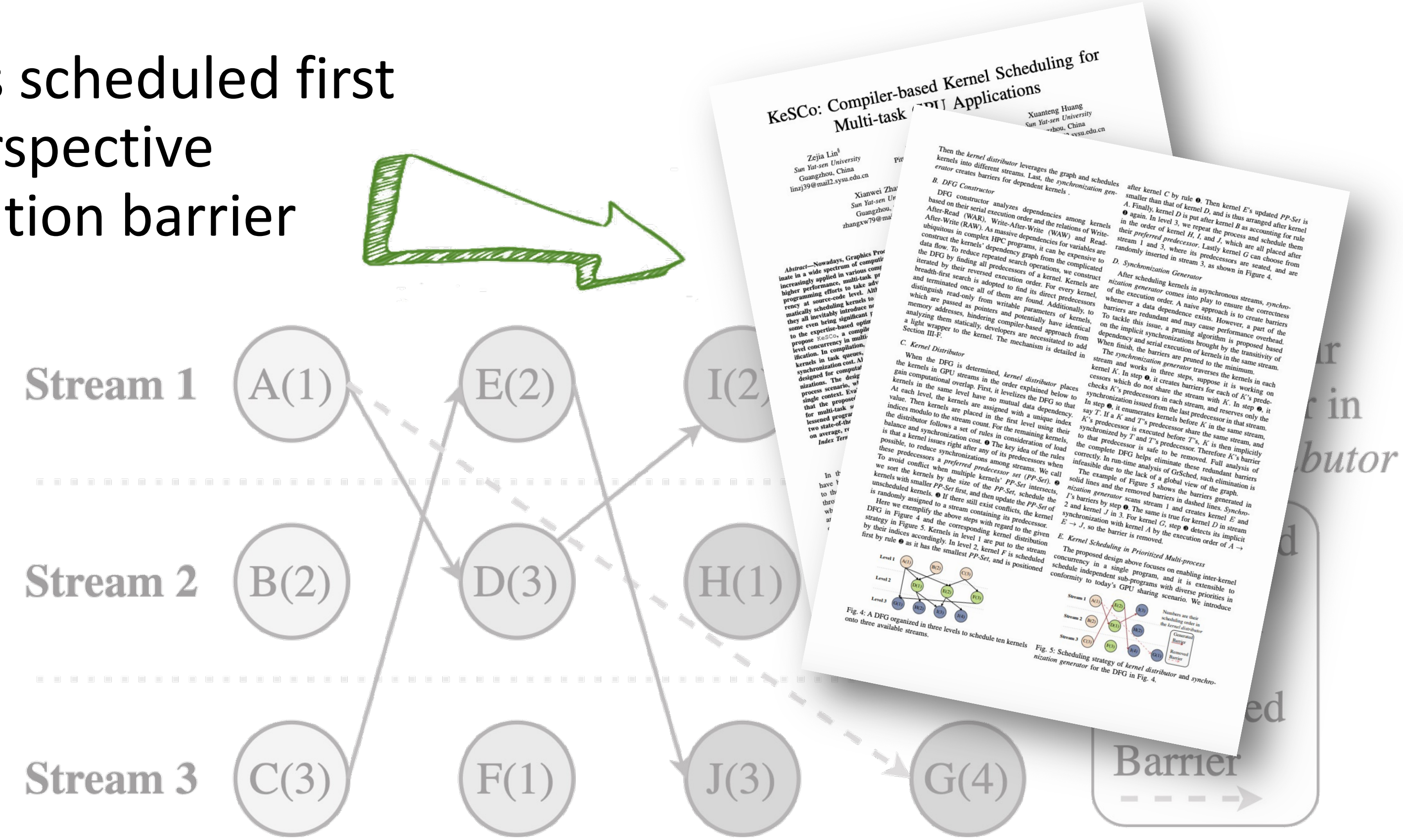
Goal: ❶ Increase overlap ❷ Minimize synchronization ❸ Load balance

Key idea: Issue a kernel immediately after its predecessor whenever feasible

Details

- Kernel with less predecessors is scheduled first
- Rearrangement from global perspective
- Remove redundant synchronization barrier
- ……





KeSCo Overview

Code

Compiler Frontend

Optimizations **KeSCo**

Compiler Backend

Binary

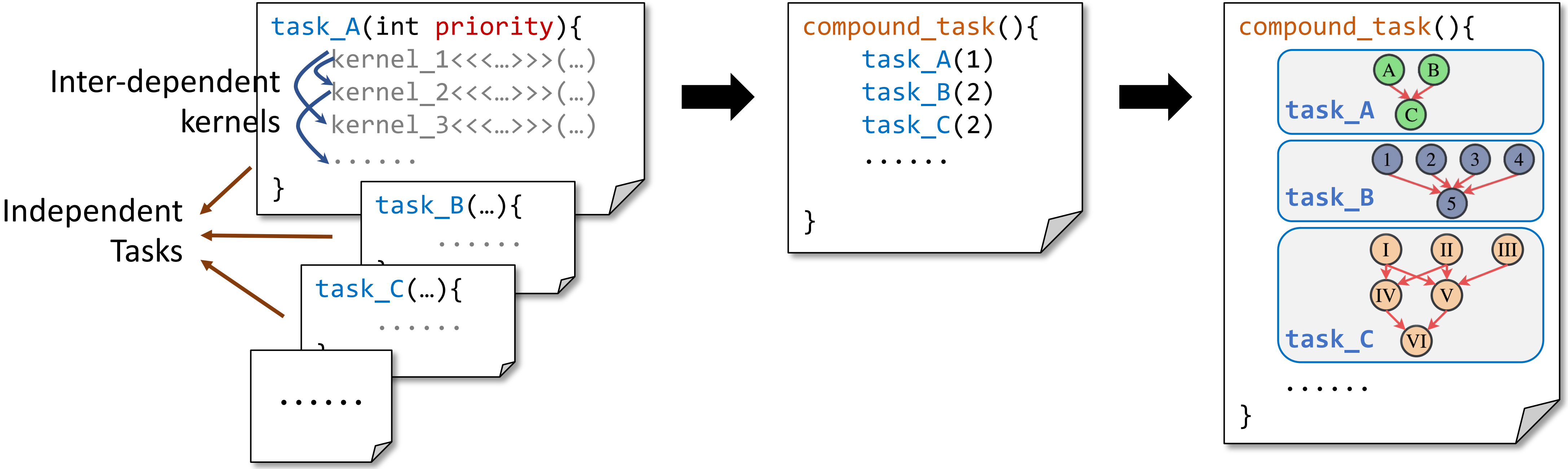
***Kernel-level* Scheduler**

Automatically analyze dependency, rearrange kernels for higher overlap and less synchronization

***Task-level* Scheduler**

Coordinates independent prioritized tasks, extends the kernel-level scheduler to broader usage

Multiple Workload Scheduling



A task is composed of inter-dependent kernels

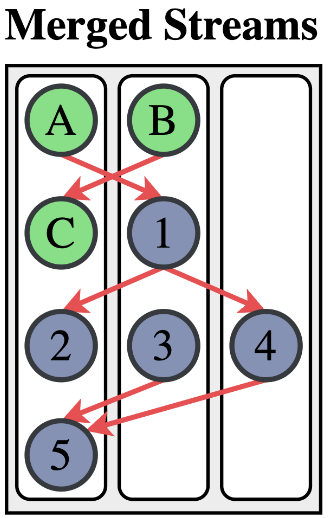
Independent tasks are compounded

Essentially a larger task graph

Extending the kernel-level scheduler to support multiple independent workloads

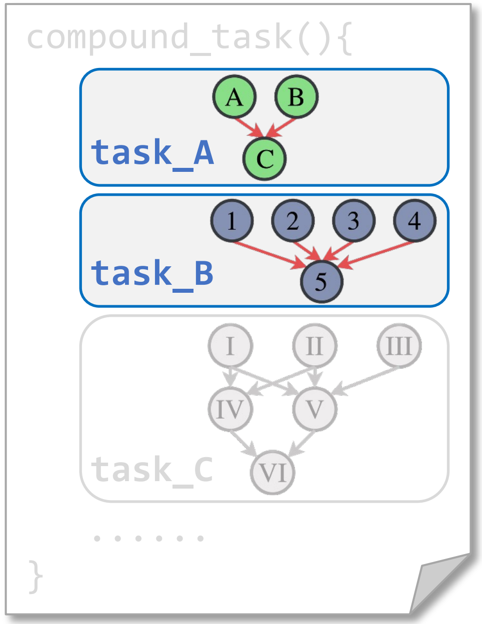
Key idea: Schedules hierarchically, postpone low-priority tasks

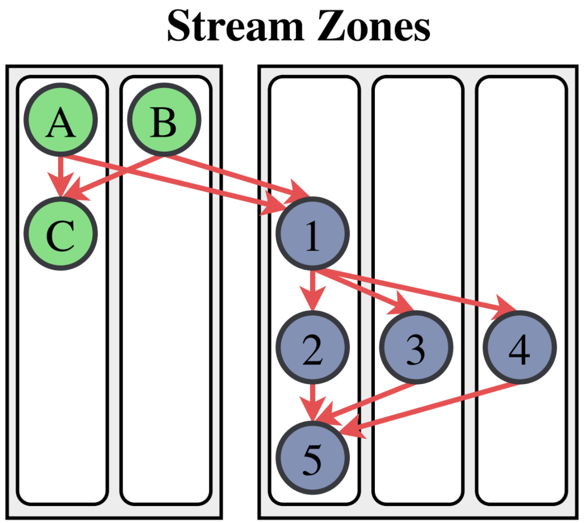
Multiple Workload Scheduling

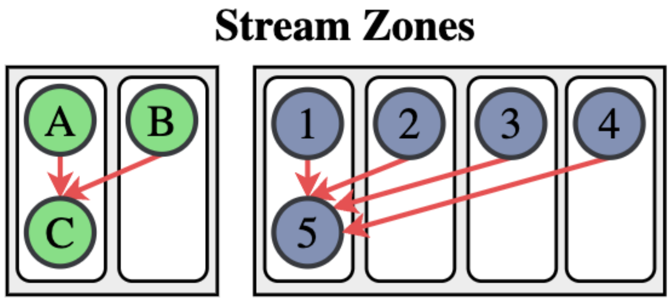


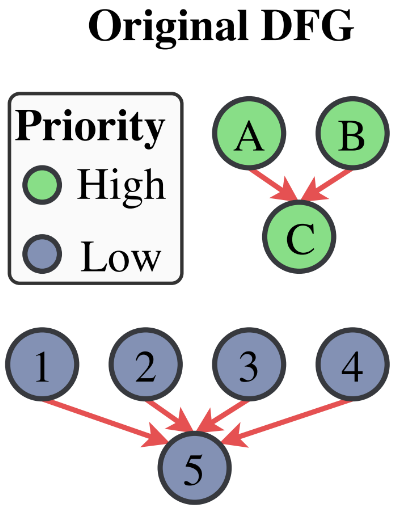
Hierarchical scheduling

1. Adopt **kernel-level scheduling** approach independently for each zone
2. Demotes low-priority task
3. **Remove** redundant barriers and **merge** streams









KeSCo Overview

Code

Clang Frontend

Optimization Middle-end

KeSCo

Code Gen Backend

Binary

Kernel-level Scheduler

Automatically analyze dependency, rearrange kernels for higher overlap and less synchronization

Task-level Scheduler

Coordinates independent prioritized tasks, extends the kernel-level scheduler to broader usage

Compilation Pipeline Integration

Code

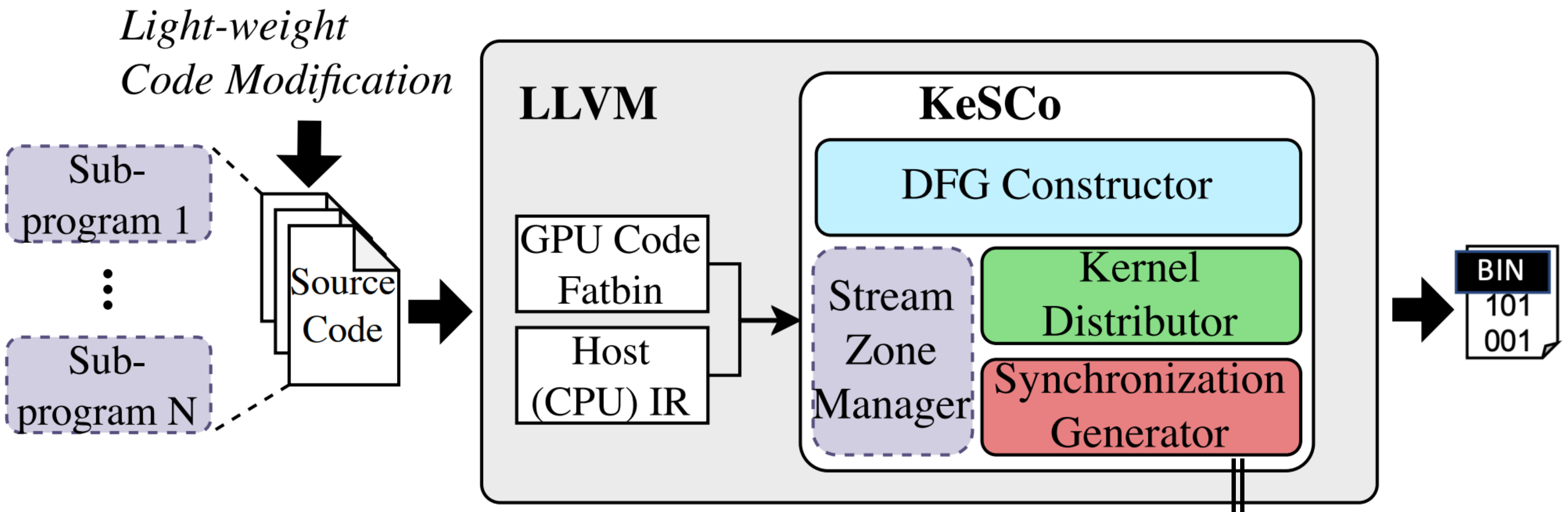
Clang Frontend

Optimization Middle-end **KeSCo**

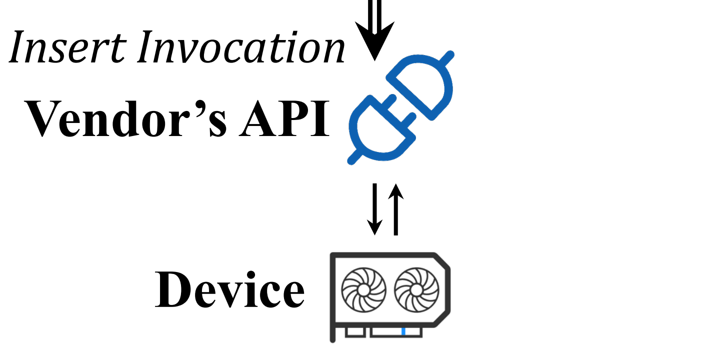
Code Gen Backend

Binary

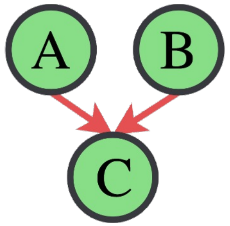
Impelemented as a set of compiler plugins for code transformation



Portability: *KeSCo targets CUDA, but can easily port to other concurrent task queue-supported frameworks*



Compilation Pipeline Integration (cont.)



Serial Code

```
kernel_A<0>(...);
kernel_B<0>(...);
kernel_C<0>(...);
```

Denotes stream ID (pseudo code for simplicity)

KeSCo

```
kernel_A<0>(..., 1);
kernel_B<0>(..., 1);
kernel_C<0>(..., 1);
```

CUDA Stream

```
kernel_A<1>(...);
kernel_B<2>(...);
cudaEventRecord(e1, 2);
cudaStreamWaitEvent(2, e1);
kernel_C<1>(...);
```

```
__global__ void axpby(float *Y, int n, float alpha, float *X, float beta,
                      int outputs = 1, int priority = 1);
```

# of writable parameters (outputs)

priority of the kernel (optional) (priority)

Experimental Setup

- Platform
  - GPU: Nvidia A100
  - CPU: AMD EPYC 7742
  - CUDA: 11.4.4
  - LLVM: 14.0.0

- Single process schemes
  - Sync: Serial execution
  - Async: Manual-opt. CUDA stream execution
  - Taskflow[1]: Programming model in C++
  - GrSched[2]: Dynamic scheduler in Python
  - **KeSCo**: Our compiler-based optimization

- Workload[2]

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

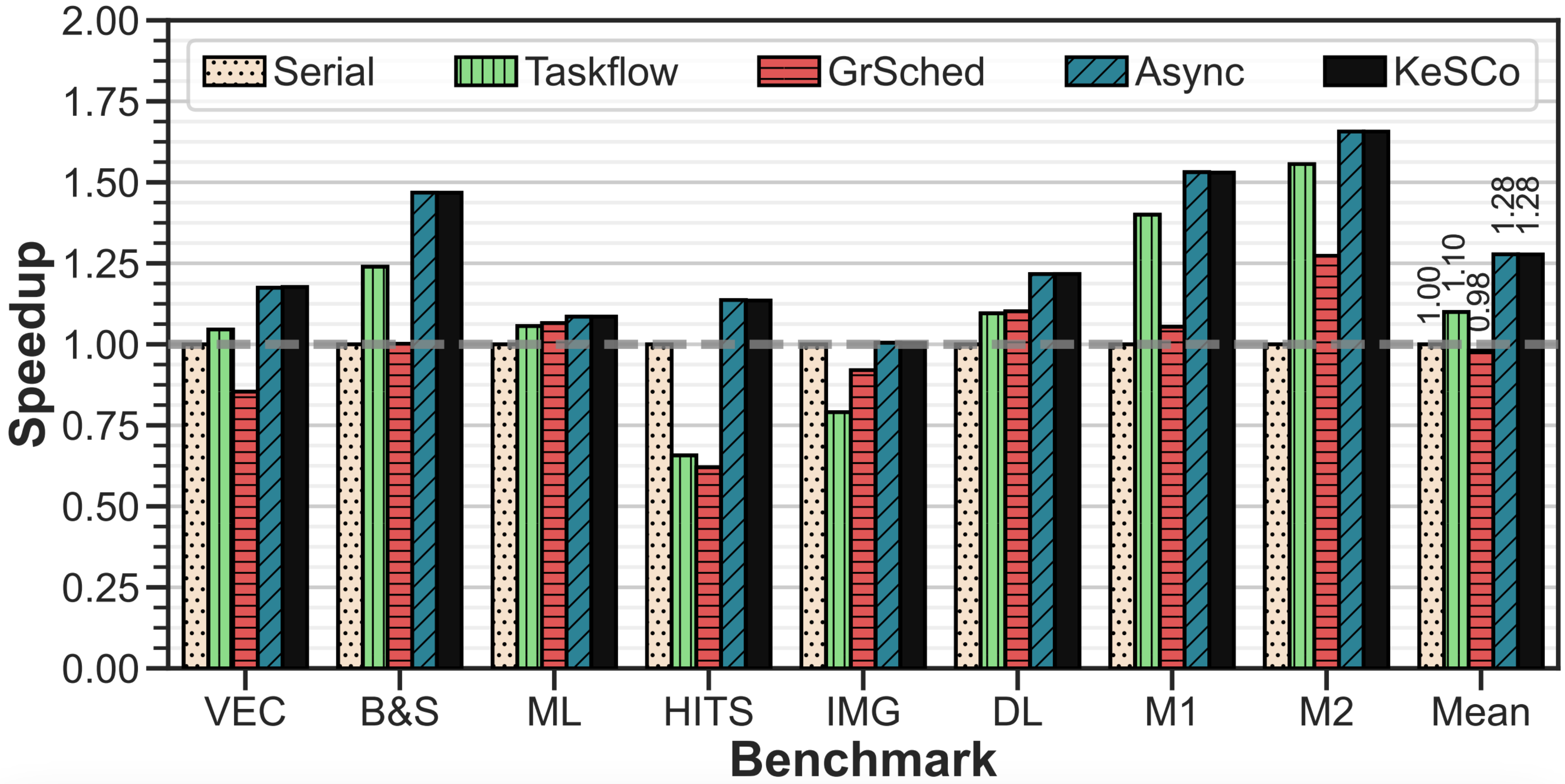
- Multi process schemes
  - Baseline: Launching all tasks simultaneously
  - Nvidia MPS[3]: Multi-process service
  - **KeSCo**: Our compiler-based optimization

[1] Tsung-Wei Huang et al. Taskflow: A lightweight parallel and heterogeneous task graph computing system. IEEE Transactions on Parallel and Distributed Systems
[2] Alberto Parravicini et al. Dag-based scheduling with resource sharing for multi-task applications in a polyglot GPU runtime. IPDPS 2021
[3] NVIDIA. Multi-process service. https://docs.nvidia.com/deploy/mps/index.html

Speedup *w/o* Data Prefetch

On average: Competitive performance against manual optimization

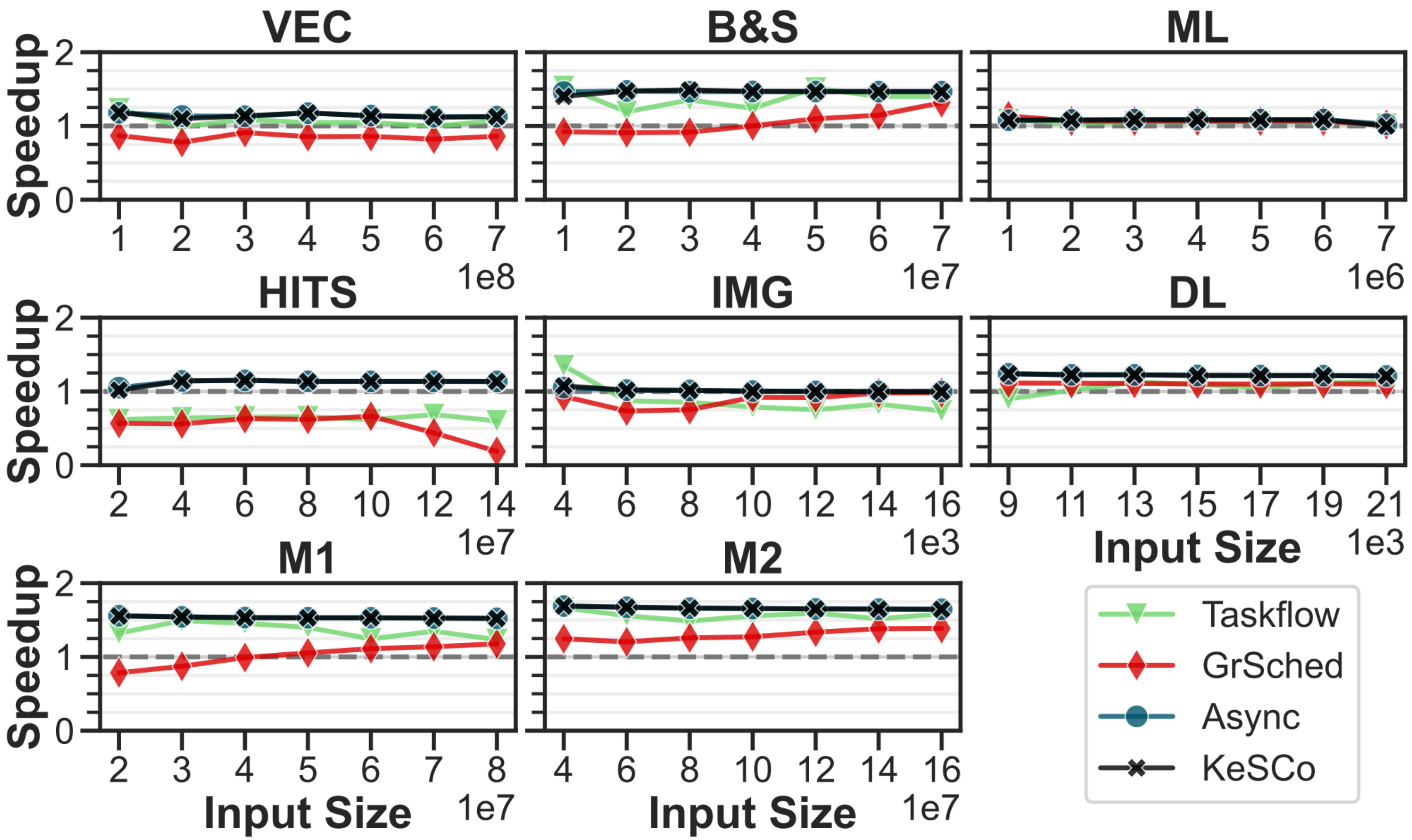
1.28× to *Serial*, **1.16×** to *Taskflow*, **1.31×** to *GrSched*



Speedup *w/o* Data Prefetch (cont.)

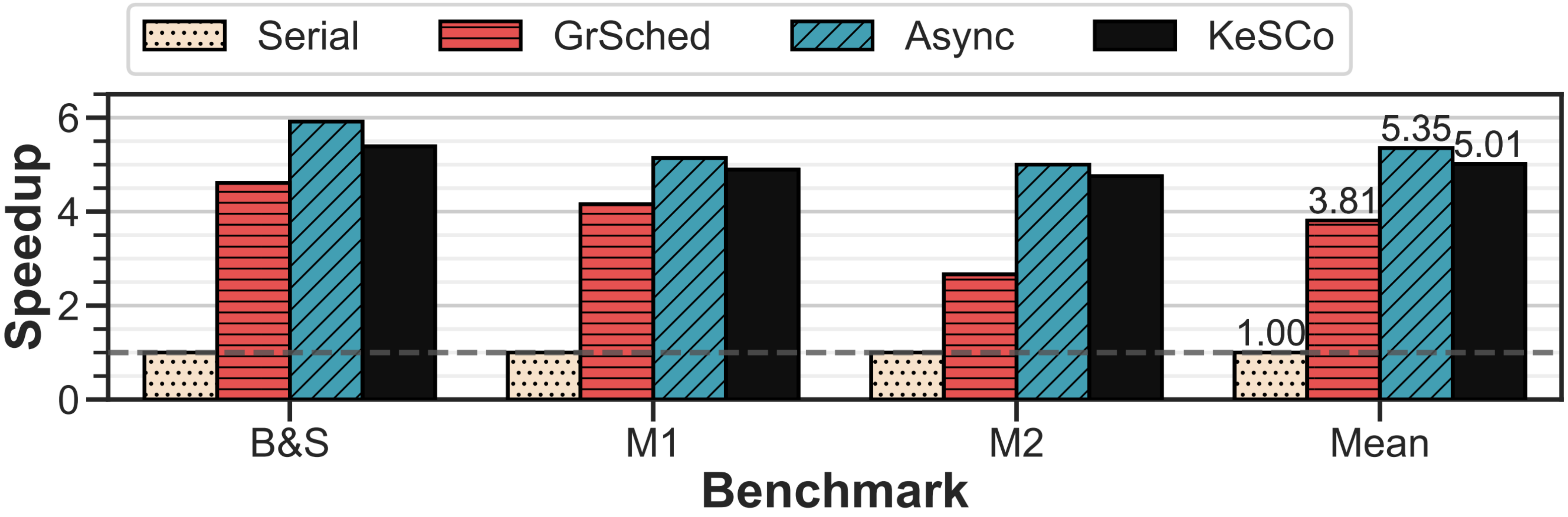
Memory occupation 1GB – 10GB

Robust against varying computational demand



Speedup *w/* Data Prefetch

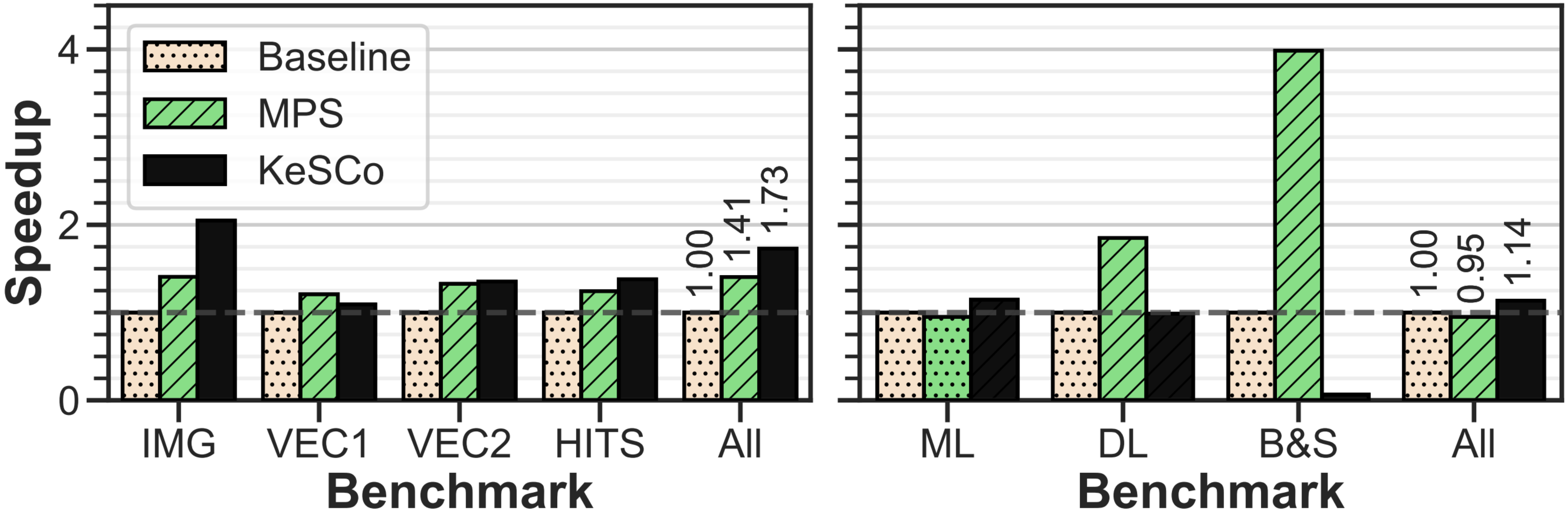
On average: Achieves **93%** performance compared to manual optimization
5.01× to *Serial*, **1.32×** to *GrSched*



Speedup in Multiple Independent Tasks

On average: **1.43×** to *Baseline (uncoordinated execution)*, **1.22×** to *MPS*

- Priority in decreasing order
- **MP-1**: IMG + 2×VEC + HITS (~20GB mem.)
- **MP-2**: ML + DL + B&S (~15GB mem.)



Programming Efforts

- ✓ Automatic dependency analysis
- ✓ Automatic concurrency management
- ✓ No new programming framework

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

Conclusion

- **Engineering burden** and **performance gap** is observed in implementing concurrent kernel execution with existing programming models.

- We propose KeSCo, a **compiler-based scheduler**
  - ➢ Expose kernel-level concurrency with trivial human efforts
  - ➢ Low synchronization, load balance scheduling algorithm
  - ➢ Extensible to multi-process scenario

- KeSCo outperforms the SOTAs with **lessened programming efforts**.

Thank you

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

Zejia Lin[§†], Zewei Mo[§‡], Xuanteng Huang[†], Xianwei Zhang[#†], Yutong Lu[†]

†Sun Yat-sen University, ‡University of Pittsburgh
Email: linzj39@mail2.sysu.edu.cn

中山大學 SUN YAT-SEN UNIVERSITY

ARCSYSU Refines Computing SYStem Uses

University of Pittsburgh

§ Equal contribution
† Work done when studying at Sun Yat-sen University
# Corresponding author