15-442/15-642: Machine Learning Systems

Advanced Topics: Mega-Kernel

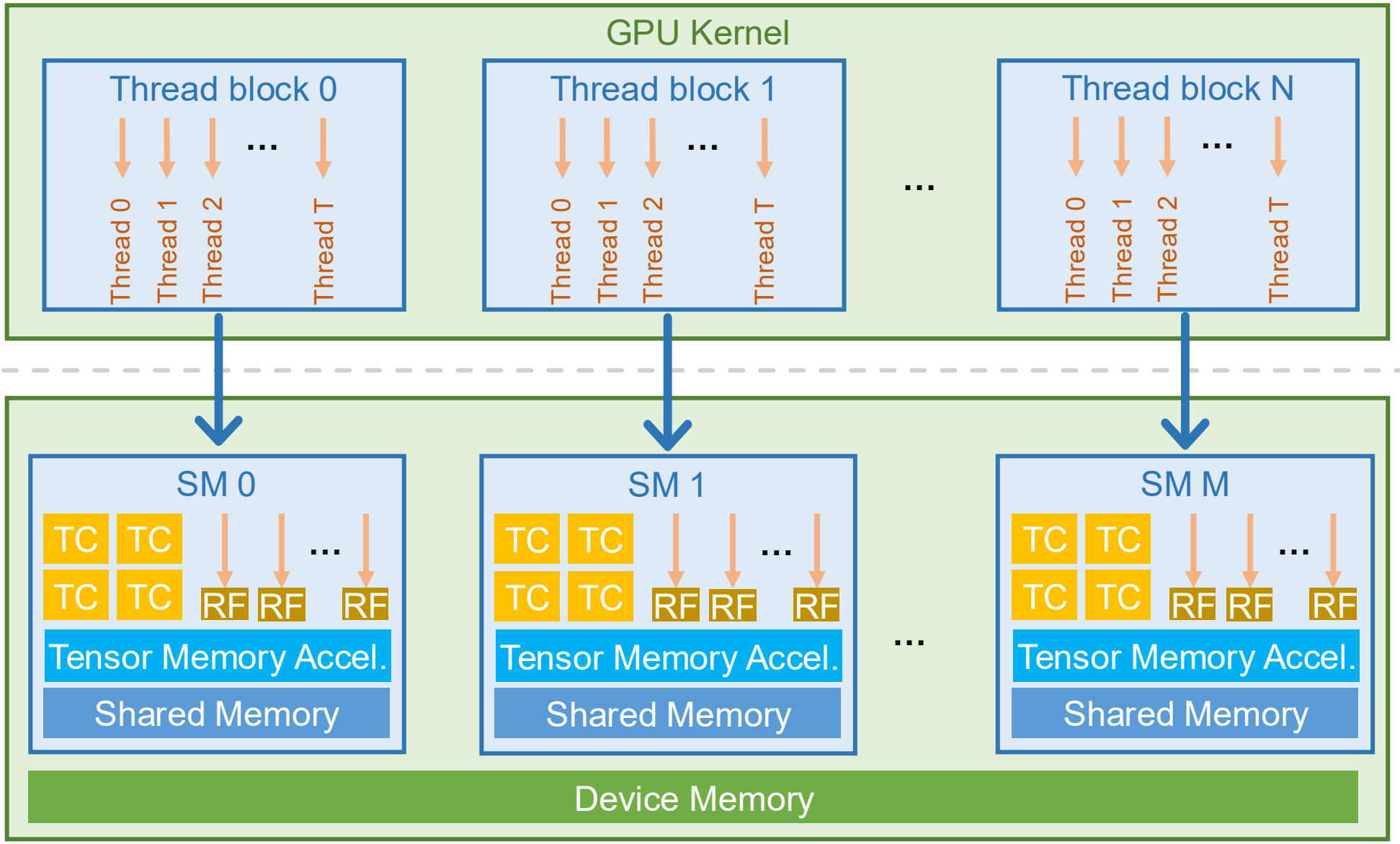
Tianqi Chen and Zhihao Jia

Carnegie Mellon University

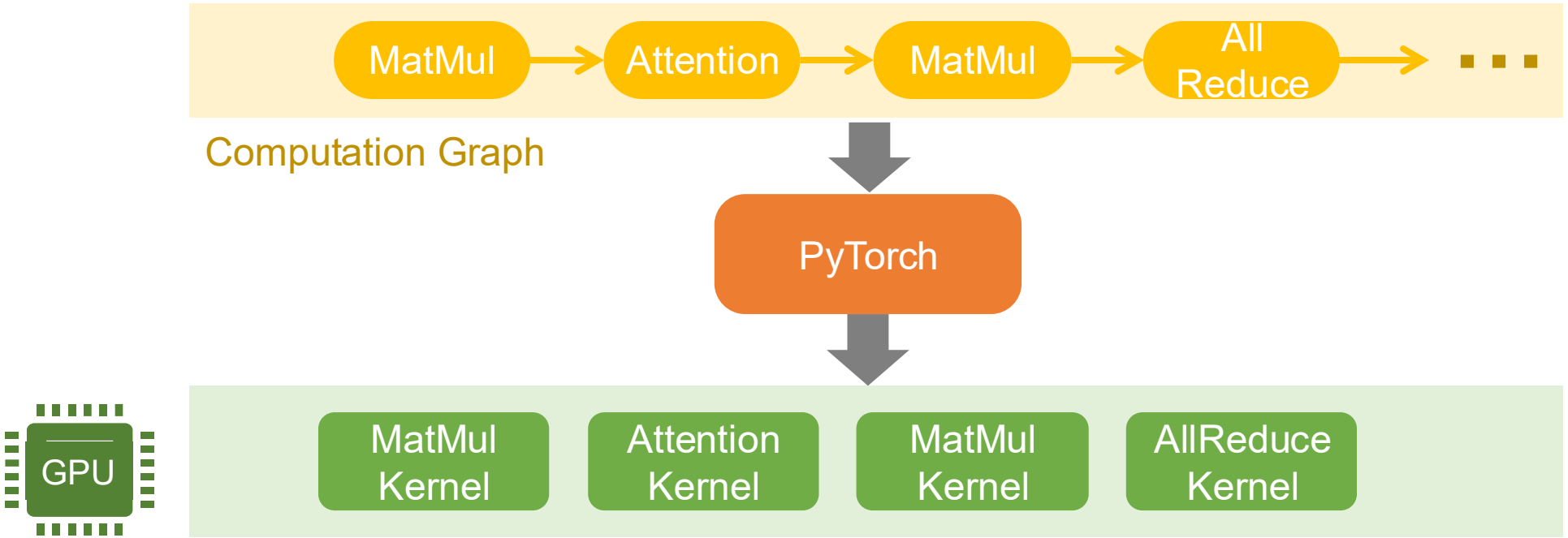
What is a (Mega-)Kernel?

Programming Abstraction

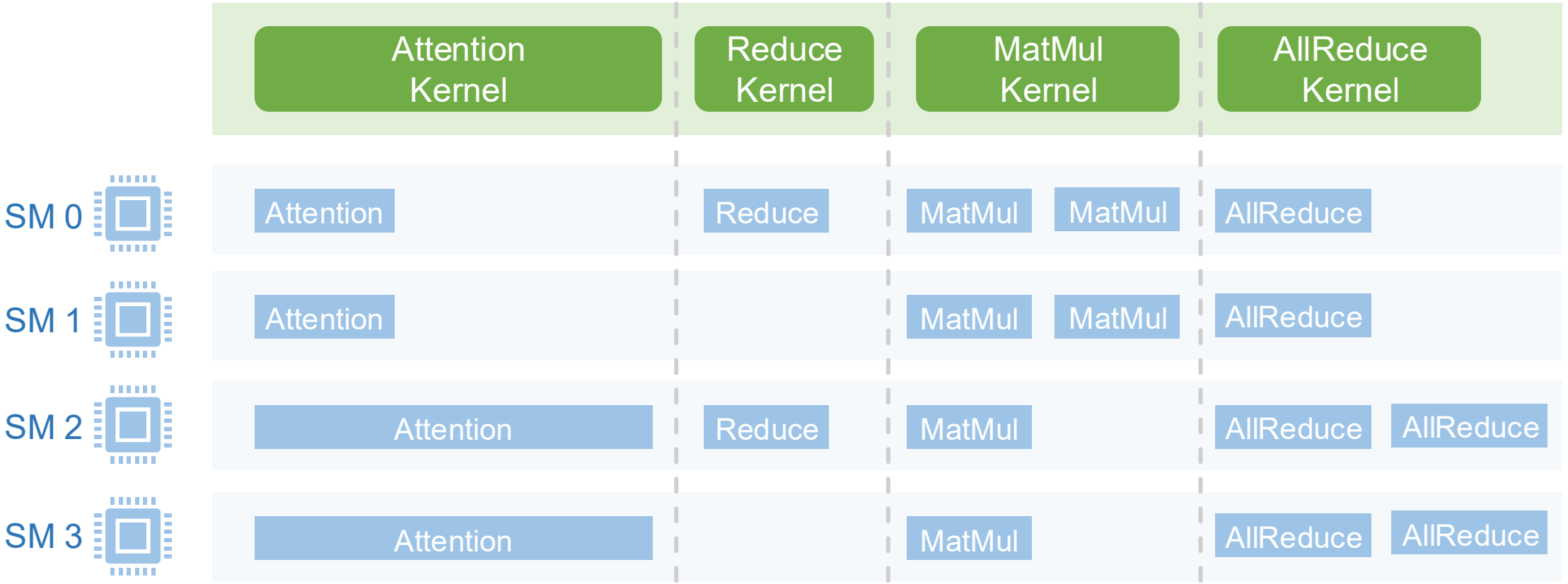
Hardware Architecture



Existing Kernel-Per-Operator Approach



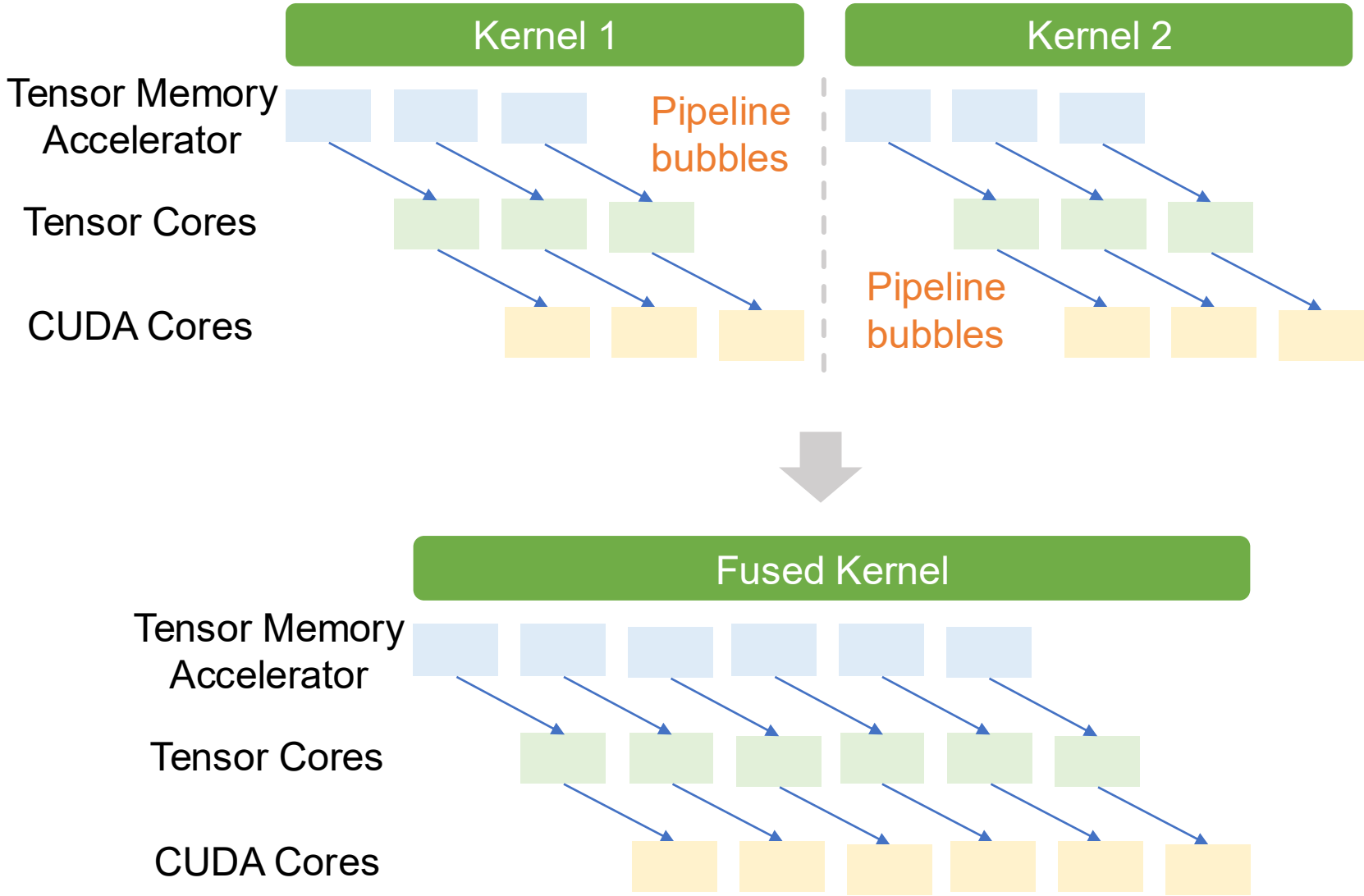
Existing Kernel-Per-Operator Approach



Limitations

No Inter-Layer Pipelining

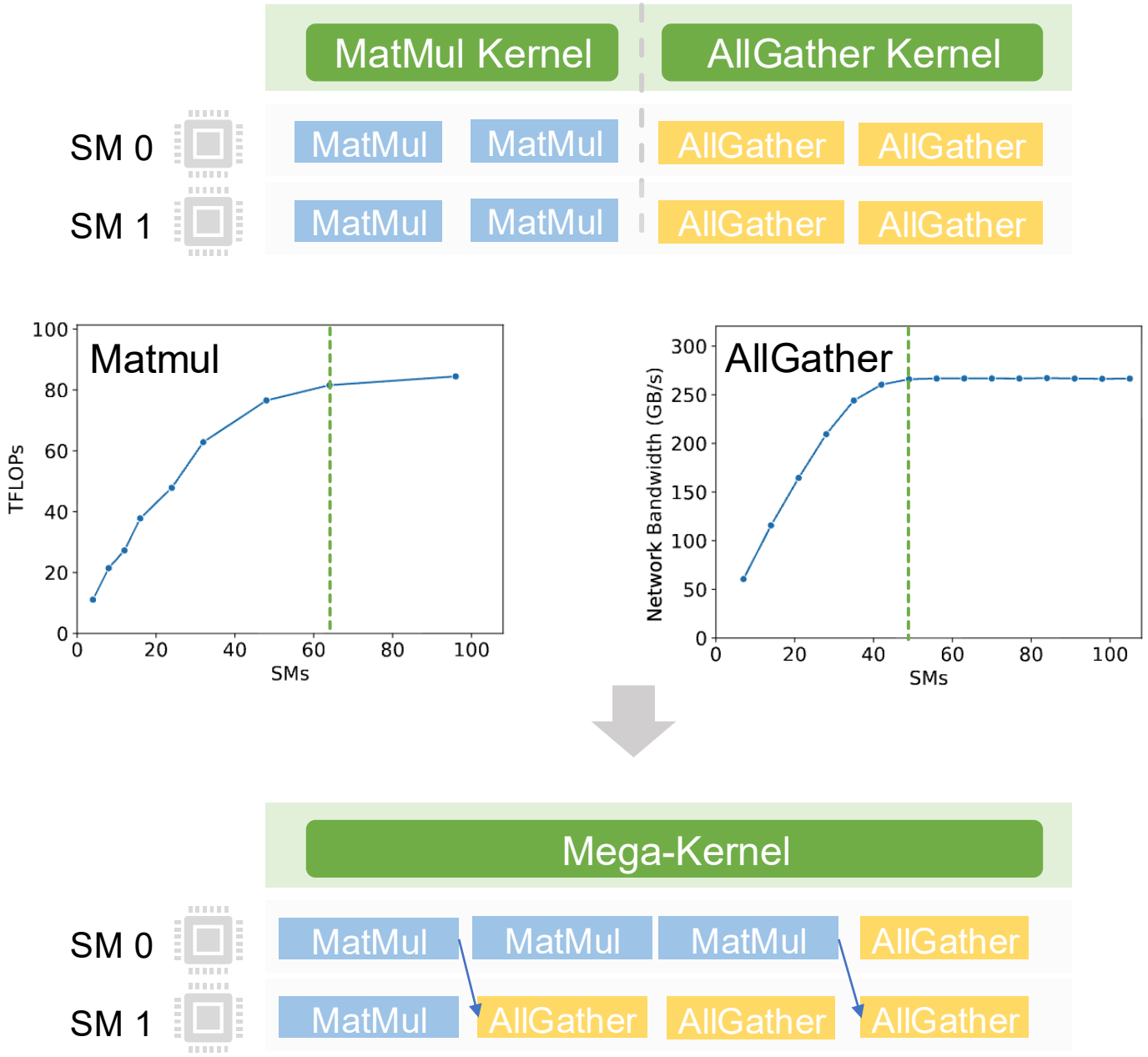
Kernel barriers prevent inter-layer pipelining



Limitations

No Inter-Layer Pipelining
Kernel barriers prevent inter-layer pipelining

No Fine-Grained Overlap
Coarse-grained dependency prevents comp. & comm. overlap



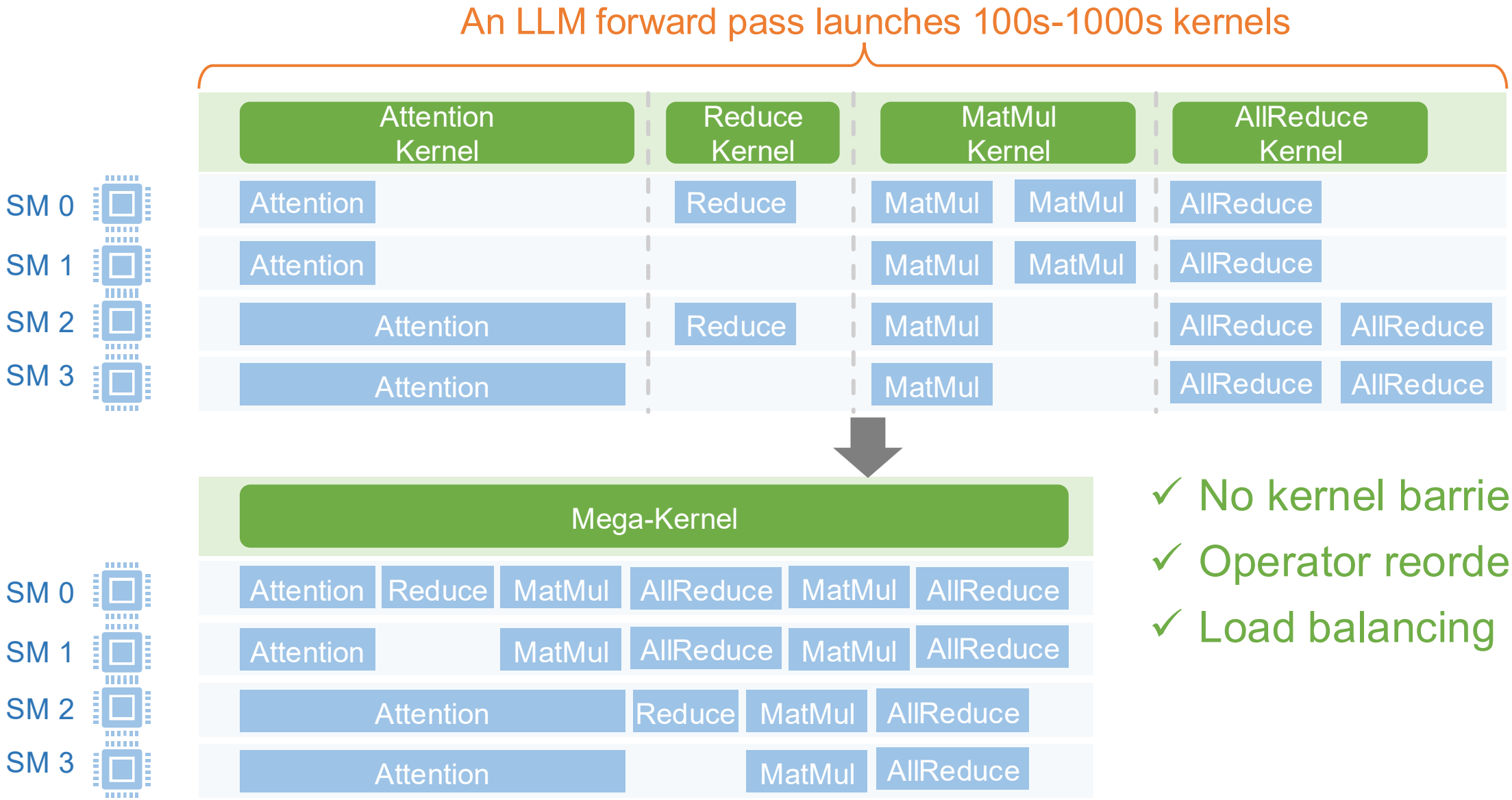
Limitations

No Inter-Layer Pipelining
Kernel barriers prevent inter-layer pipelining

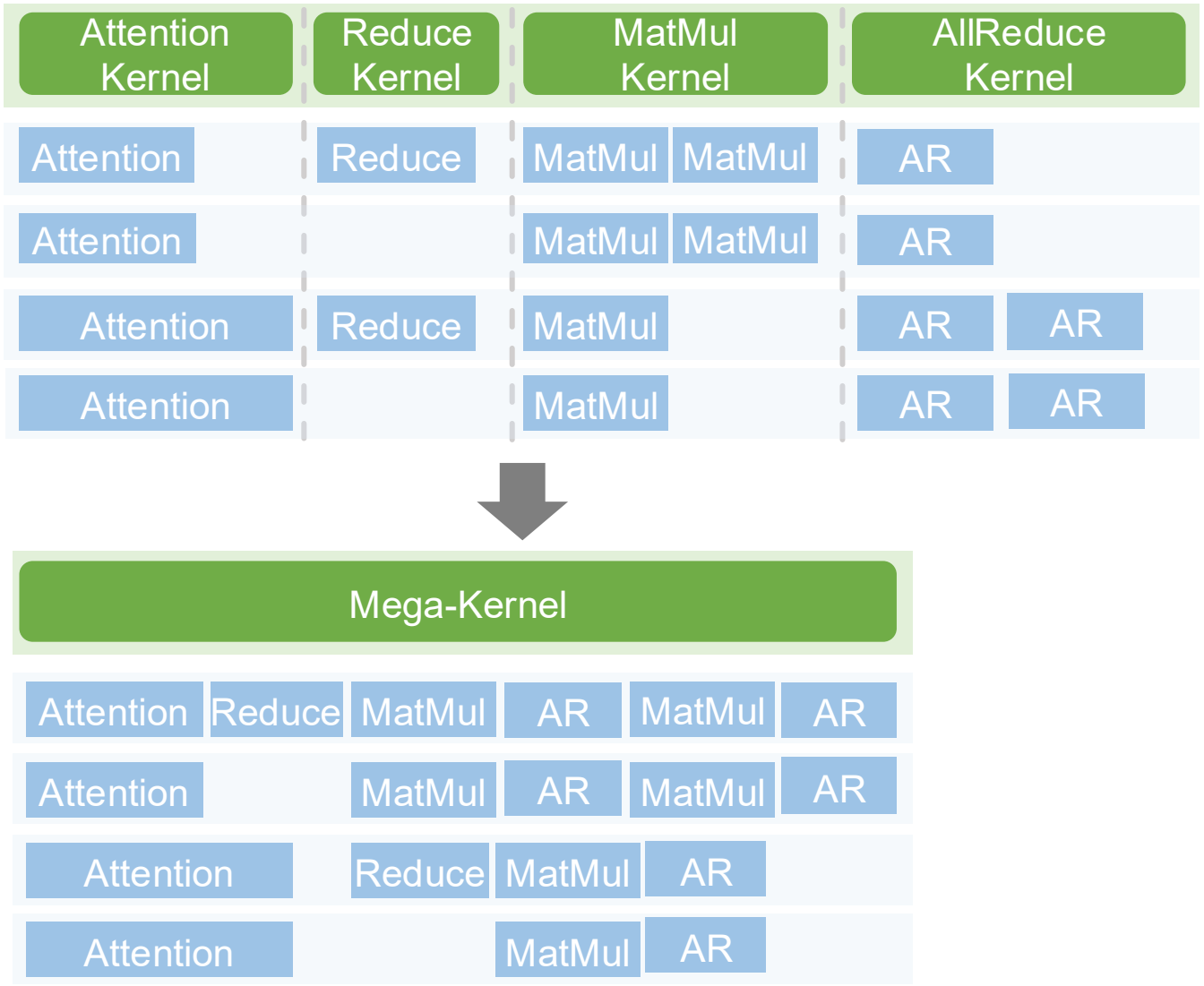
No Overlapping
Coarse-grained dependency prevents comp. & comm. overlap

Limited Dynamism
Rely on CUDA graphs to reduce kernel launch overhead

Kernel-Per-Operator v.s. Mega-Kernel



Advantages of Mega-Kernel



Inter-Layer Pipelining
All layers are fused in the same mega-kernel

Overlapping Comp/Comm
Fine-grained dependency + operator reordering

Dynamic Workloads
No need for CUDA graphs + load balancing

Key Challenges

1. How to manage dependency?

No kernel barriers in mega-kernel

Task Graph

2. How to handle dynamism?

Continuous batching, prefill/decode,
paged/radix attention, speculative decoding

In-Kernel
Parallel Runtime

3. How to optimize performance?

Existing compilers target individual kernels

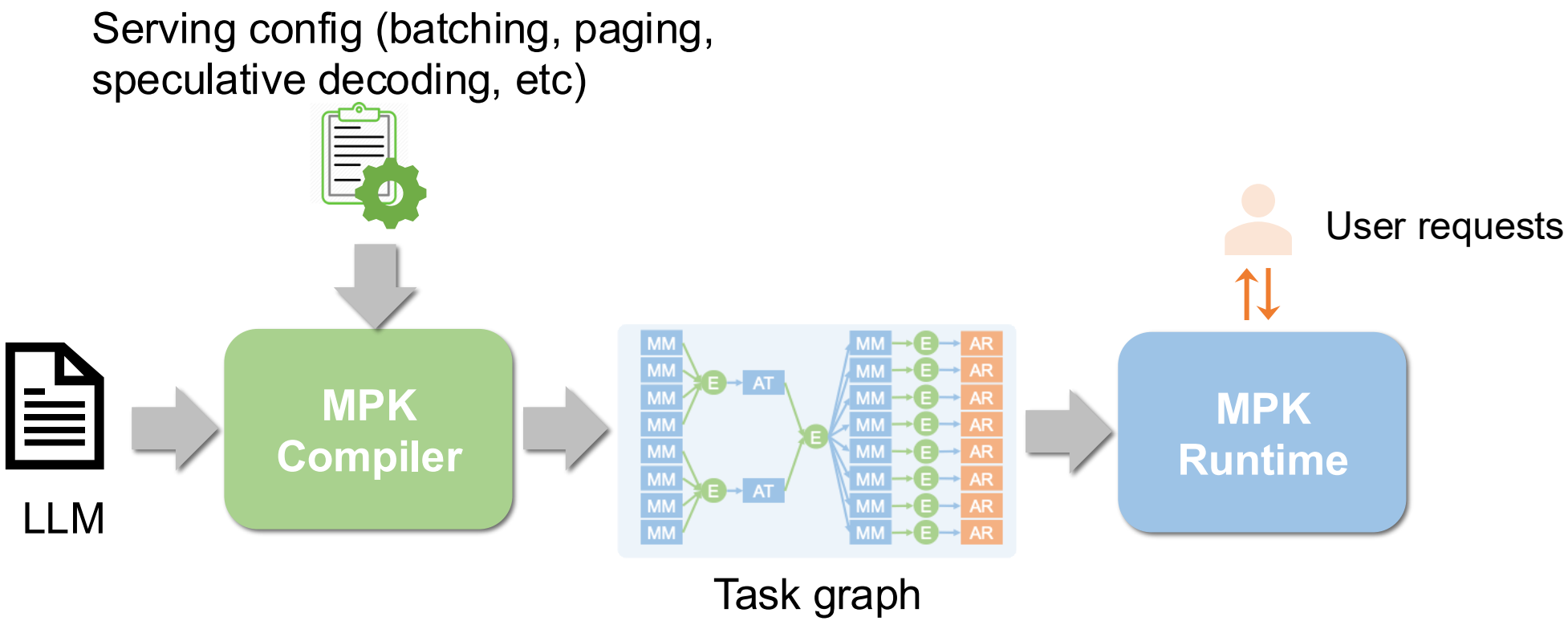
Mirage Superoptimizer*

* Mirage: A Multi-Level Superoptimizer for Tensor Programs. OSDI'25.

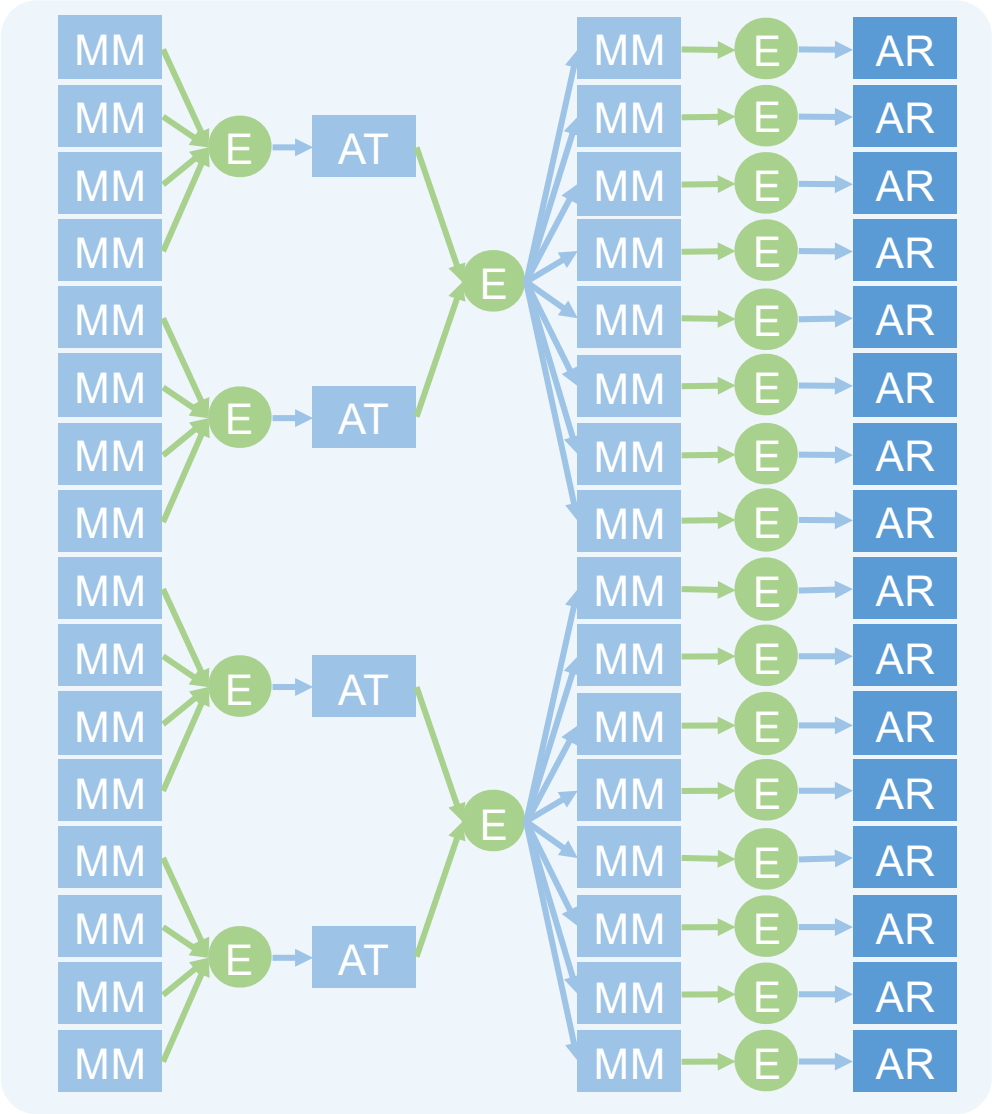
Mirage Persistent Kernel: Compiling LLM Serving into a Mega-Kernel

- **Low engineering effort**: a few dozen lines of Python code to mega-kernelize an LLM
- **Better performance**: outperform existing systems by 1.2-6.7x
- **Day-0 support for new models:** do not rely on manual implementation

MPK Overview

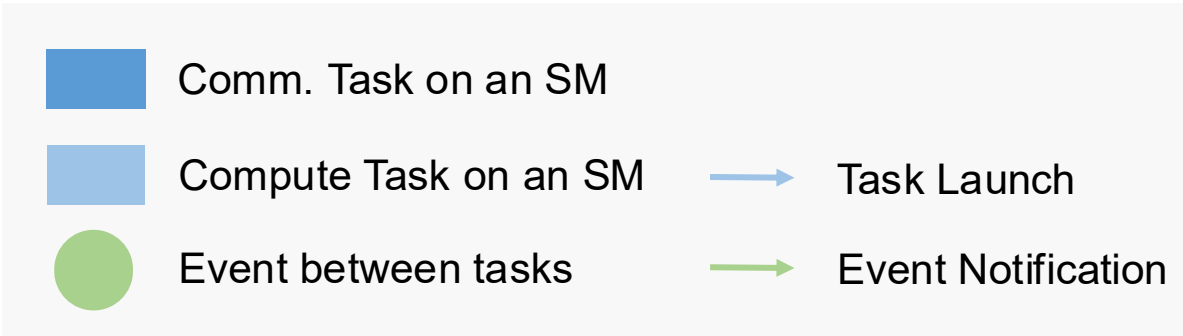


Task Graph



Interleave tasks and events

- task: a unit of workload on one SM
- event: synchronization among tasks
- task→event: notify event once task is done
- event→task: launch task once event is triggered

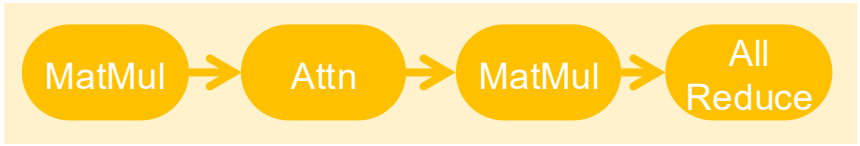


Task Graph vs. CUDA Graph

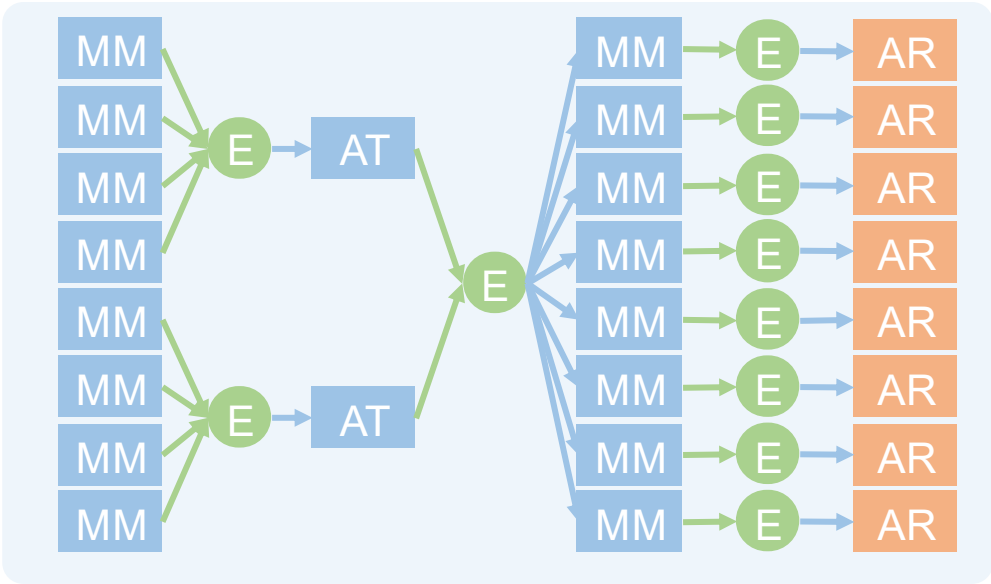
Task graph is a “lower-level” CUDA graph

- Capture sub-kernel dependency
- Static, immutable
- Constructed once and replayed many times

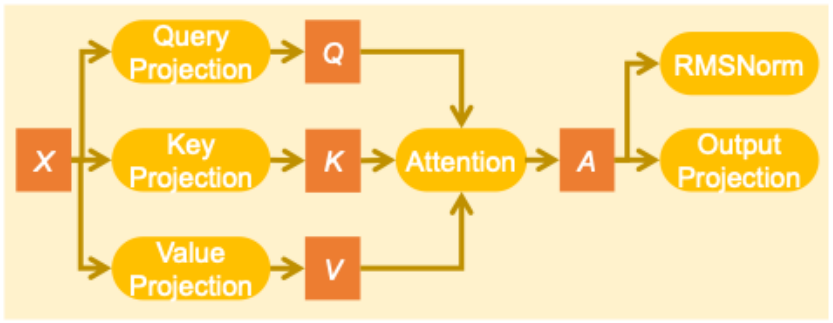
CUDA Graph: nodes are kernels on GPUs



Task Graph: nodes are tasks on SMs

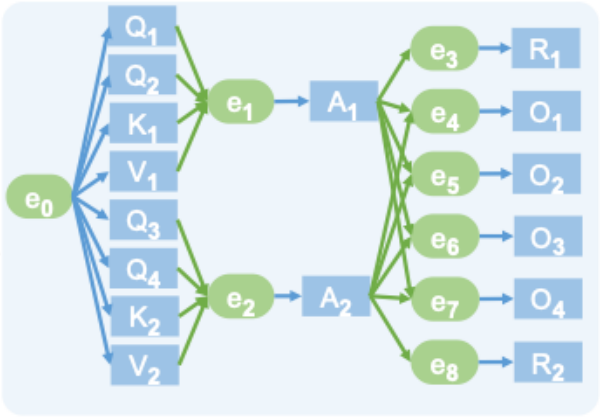


Compiler Workflow

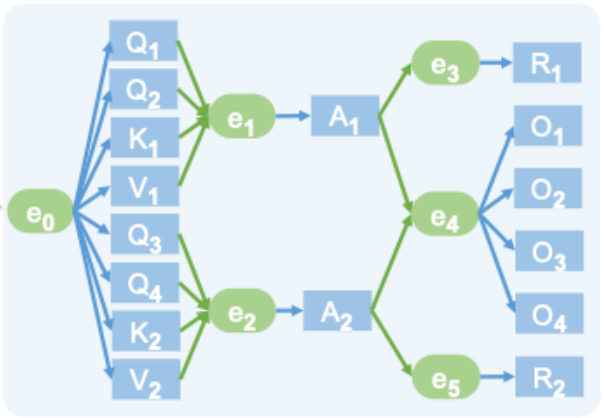


(a) Computation Graph

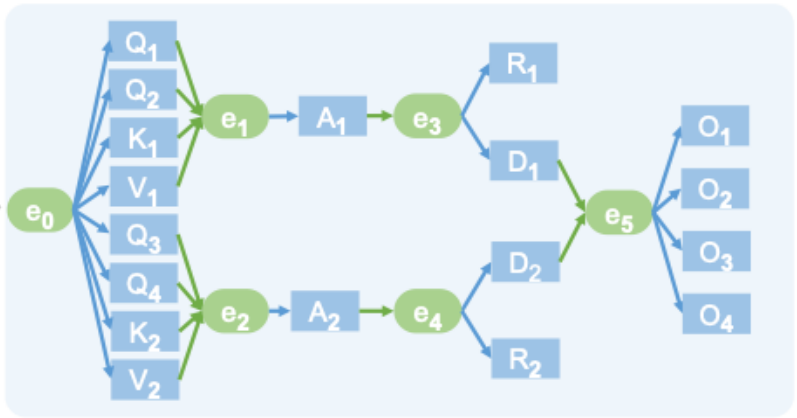
(b) Operator decomposition & dependency analysis



(c) Successor-set fusion



(d) Predecessor-set fusion



(e) *t*Graph normalization

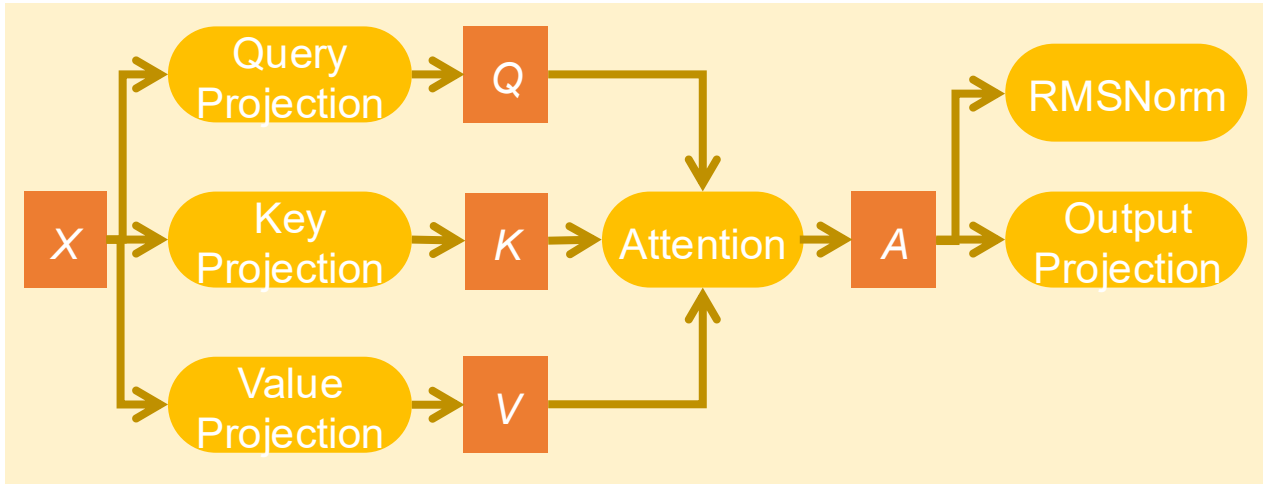
IDs	1	2	3	4	5	6	7	8	9
Tasks	Q_1	Q_2	K_1	V_1	Q_3	Q_4	K_2	V_2	A_1
Depend	e_0	e_0	e_0	e_0	e_0	e_0	e_0	e_0	e_1
Trigger	e_1	e_1	e_1	e_1	e_2	e_2	e_2	e_2	e_3
IDs	10	11	12	13	14	15	16	17	18
Tasks	A_2	R_1	D_1	R_2	D_2	O_1	O_2	O_3	O_4
Depend	e_2	e_3	e_3	e_4	e_4	e_5	e_5	e_5	e_5
Trigger	e_4	-	e_5	-	e_5	-	-	-	-

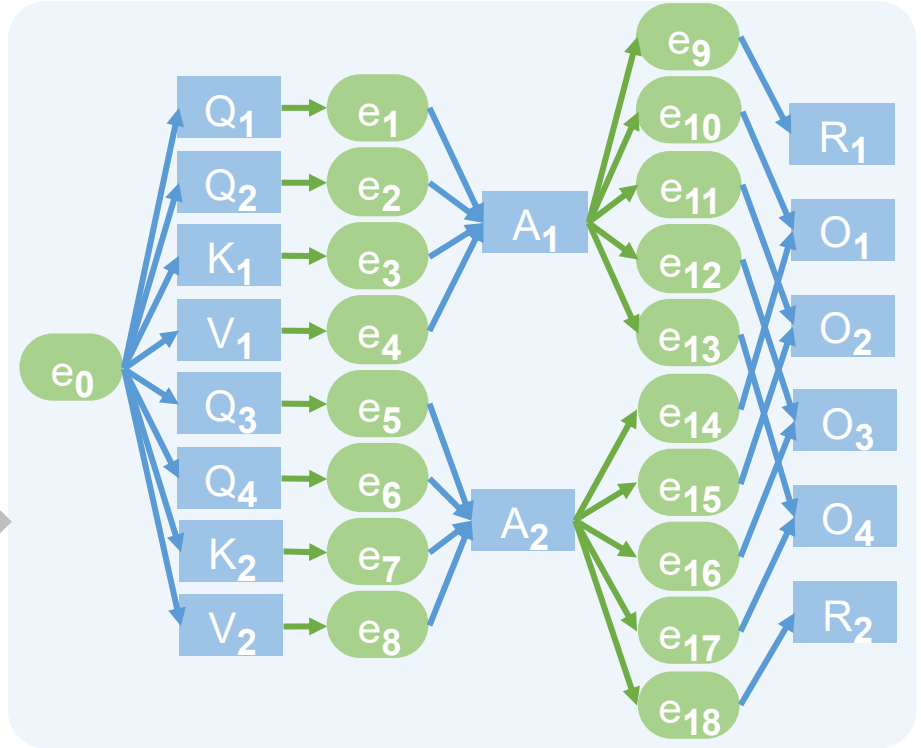
Events	e_0	e_1	e_2	e_3	e_4	e_5
Counts	0	4	4	1	2	1
First task ID	1	9	10	11	13	15
Last task ID	8	9	10	12	14	18

(f) *t*Graph linearization

Compiler Workflow #1: Operator Decomposition & Dependency Analysis

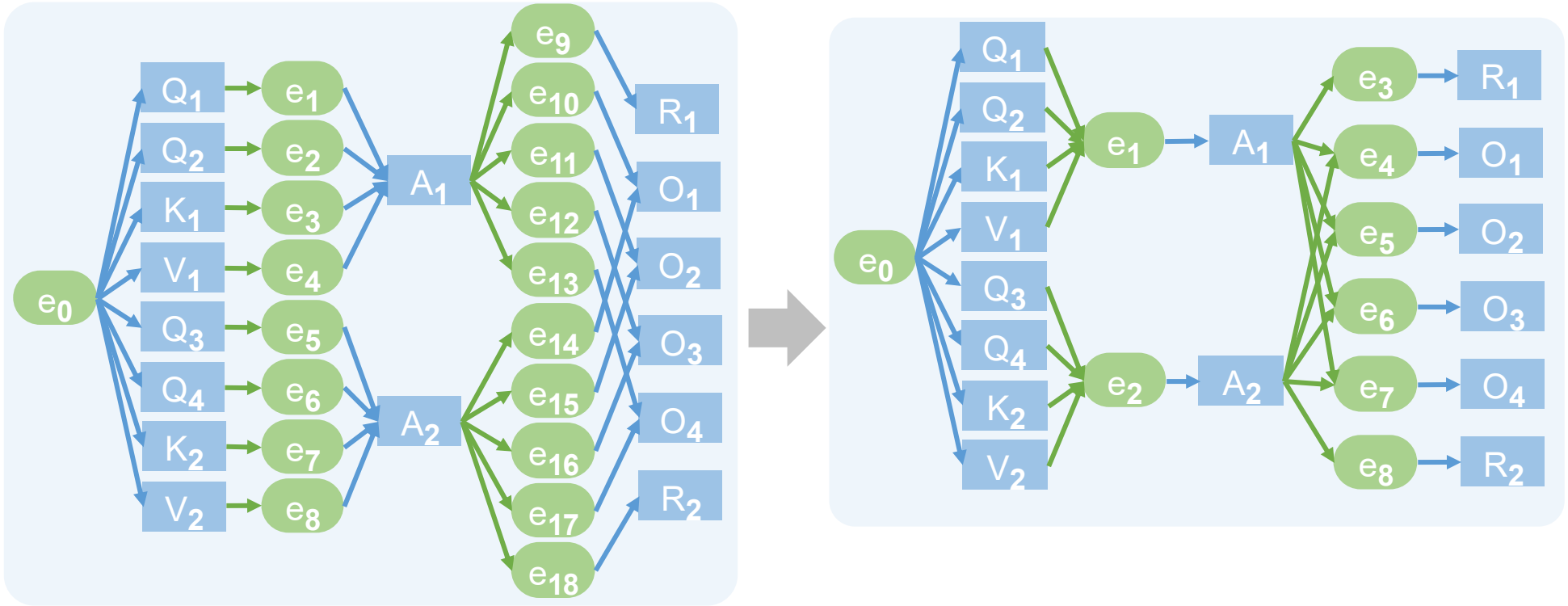
- Optimize operator-to-task decomposition based on available SMs
- Add synchronization events to capture precise task dependencies
- Generate high-performance CUDA implementation for each task





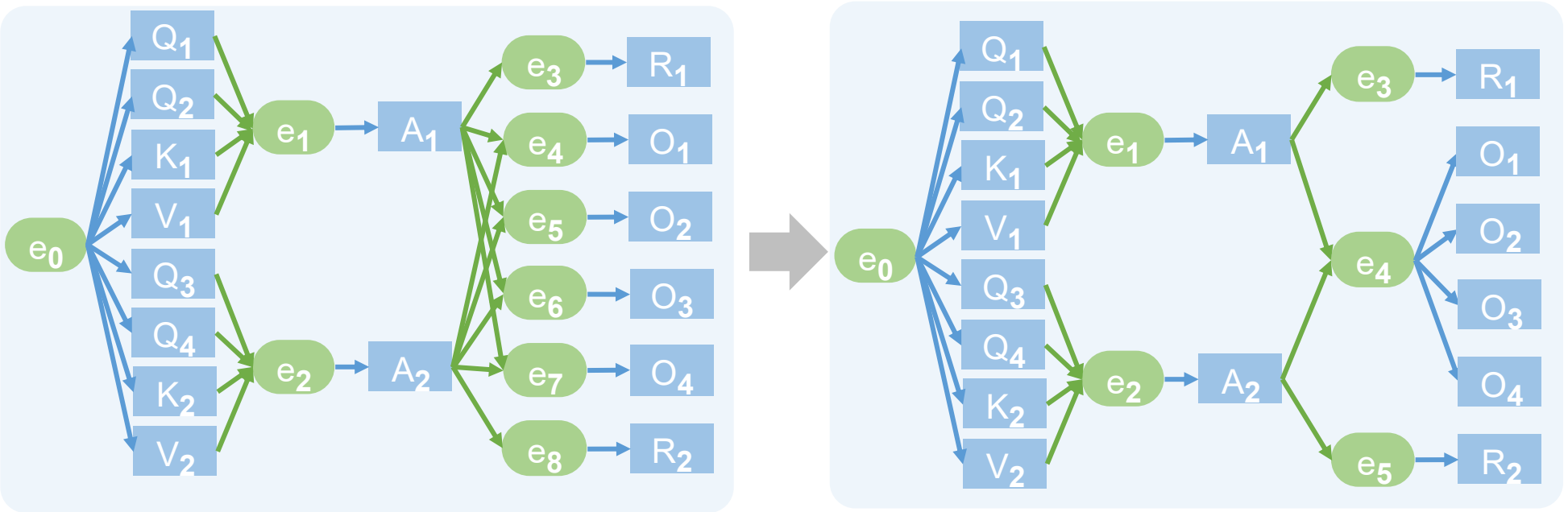
Compiler Workflow #2: Successor-Set Fusion

- Fuse events that share identical consumer (successor) sets



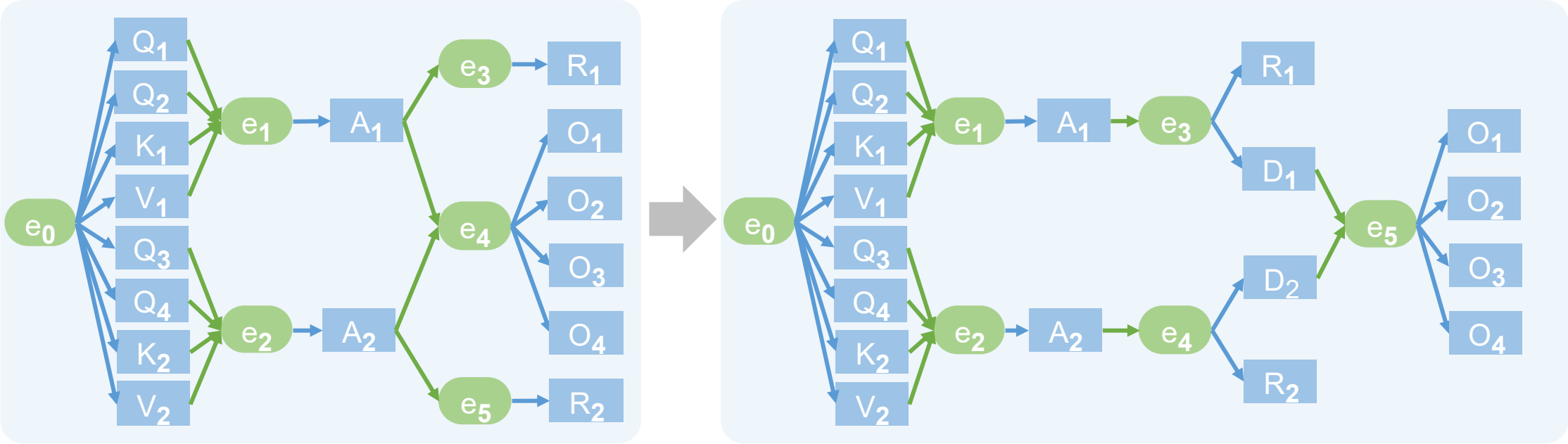
Compiler Workflow #3: Predecessor-Set Fusion

- Fuse events with identical input dependencies



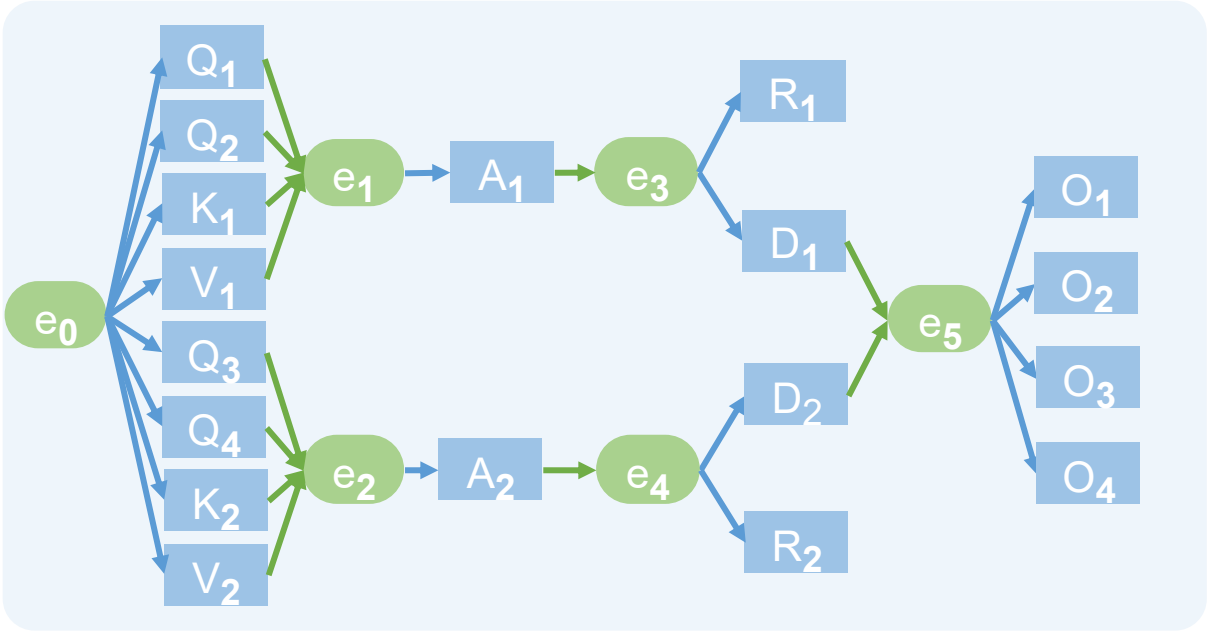
Compiler Workflow #4: Graph Normalization

- **Challenge**: a task may depend on and trigger multiple events
- **Normalization**: transform graph so each task has at most one dependent event and one triggering event



Compiler Workflow #5: Graph Linearization

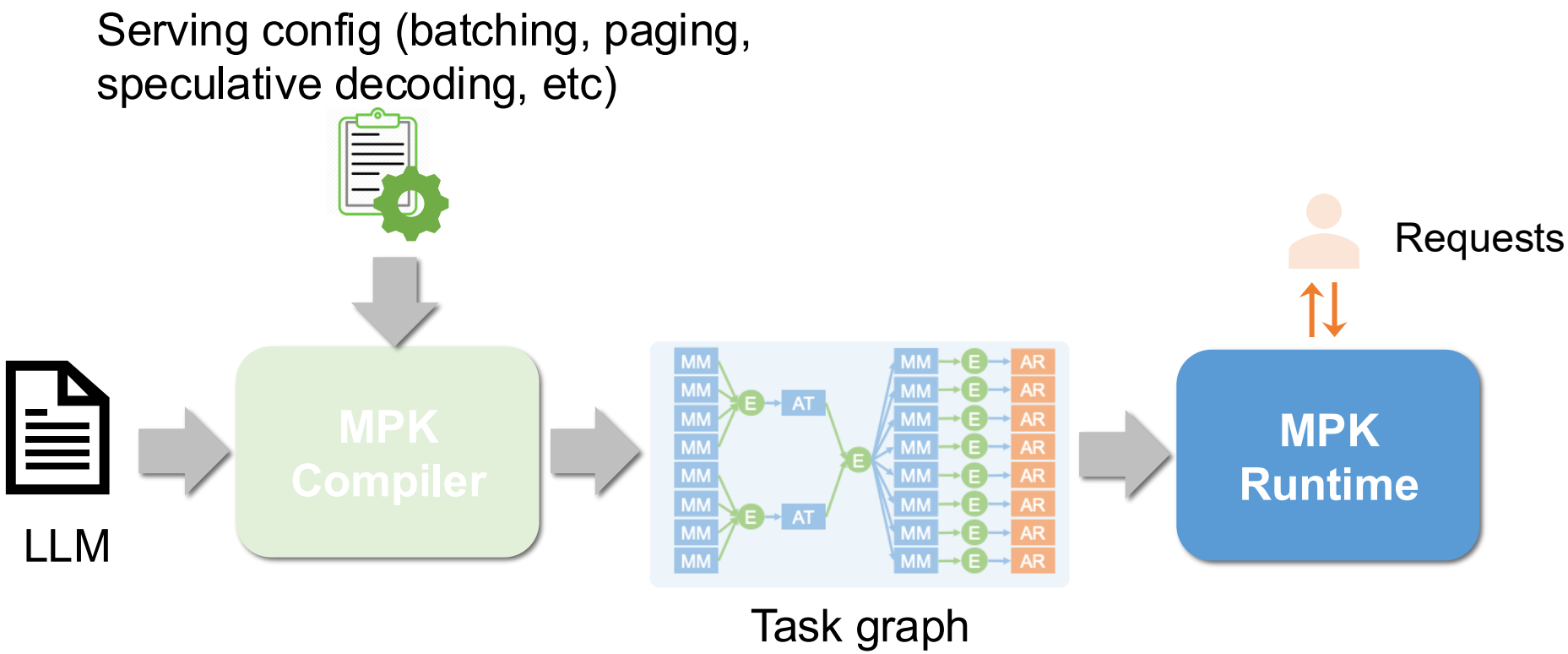
- **Challenge**: an event may trigger many tasks, need to store task indices
- **Linearization**: assign contiguous indices to tasks triggered by the same event
  - compact representation (first task and last task)



IDs	1	2	3	4	5	6	7	8	9
Tasks	Q_1	Q_2	K_1	V_1	Q_3	Q_4	K_2	V_2	A_1
Depend	e_0	e_0	e_0	e_0	e_0	e_0	e_0	e_0	e_1
Trigger	e_1	e_1	e_1	e_1	e_2	e_2	e_2	e_2	e_3
IDs	10	11	12	13	14	15	16	17	18
Tasks	A_2	R_1	D_1	R_2	D_2	O_1	O_2	O_3	O_4
Depend	e_2	e_3	e_3	e_4	e_4	e_5	e_5	e_5	e_5
Trigger	e_4	-	e_5	-	e_5	-	-	-	-

Events	e_0	e_1	e_2	e_3	e_4	e_5
Counts	0	4	4	1	2	1
First task ID	1	9	10	11	13	15
Last task ID	8	9	10	12	14	18

MPK Overview

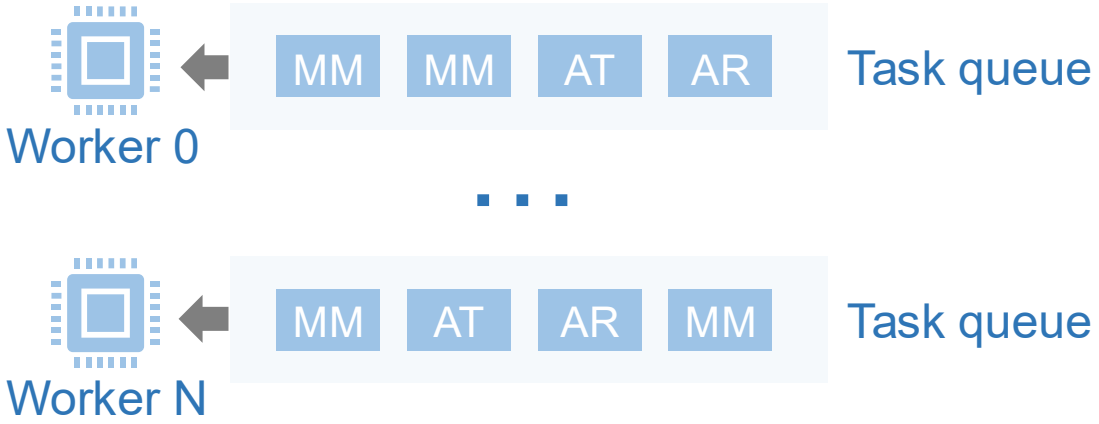


The MPK Runtime

Each scheduler runs on one warp

Each worker runs on one SM

Task-Based Parallel Runtime



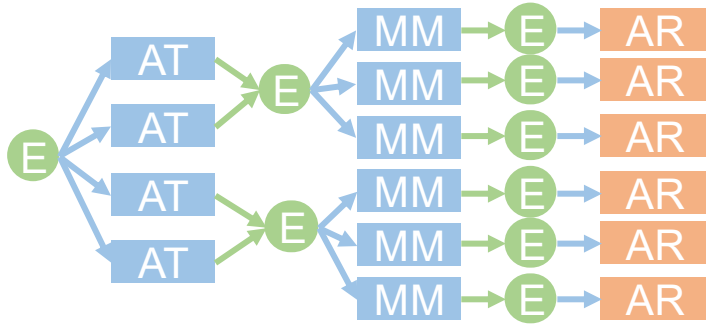
Repeatedly

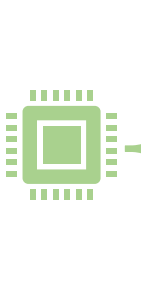
1. Fetch a task from its queue
2. Execute the task
3. Trigger the completion event

Repeatedly

1. Dequeue fully triggered event
2. Launch all tasks depending on the event

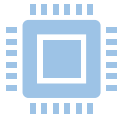
Event-Driven Execution



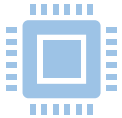


Scheduler
on a warp

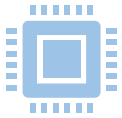
Scheduler
on a warp



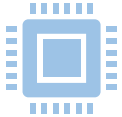
Worker on
an SM



Worker on
an SM

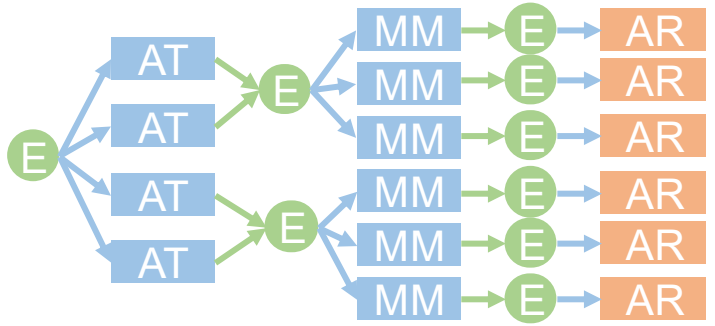


Worker on
an SM



Worker on
an SM

Event-Driven Execution



Scheduler on a warp

Scheduler on a warp

E

Worker on an SM

Worker on an SM

Worker on an SM

Worker on an SM

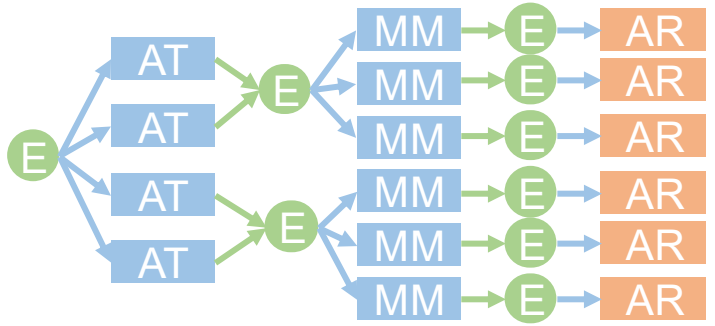
Attention

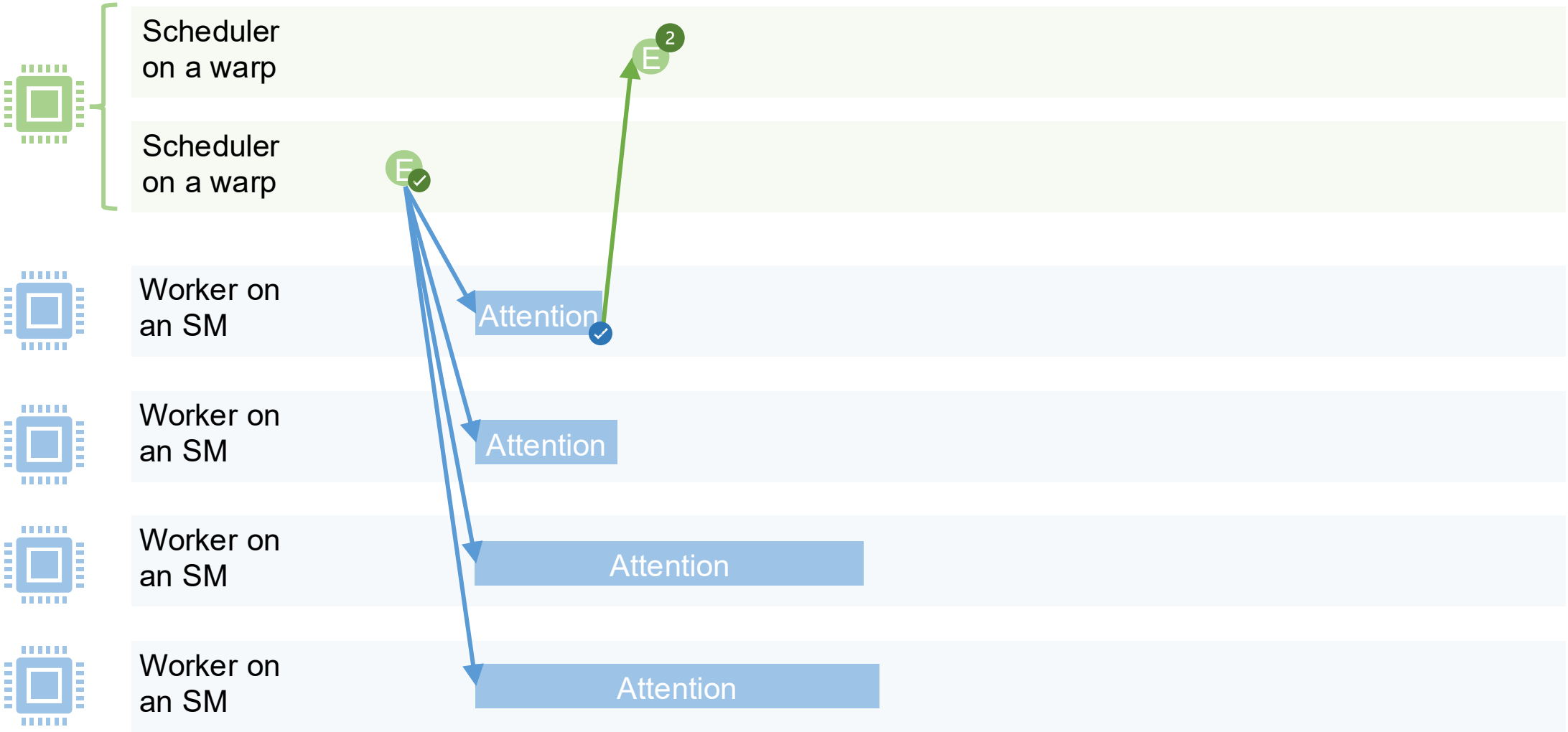
Attention

Attention

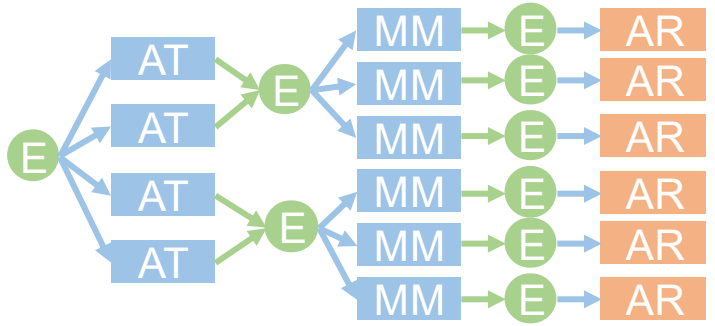
Attention

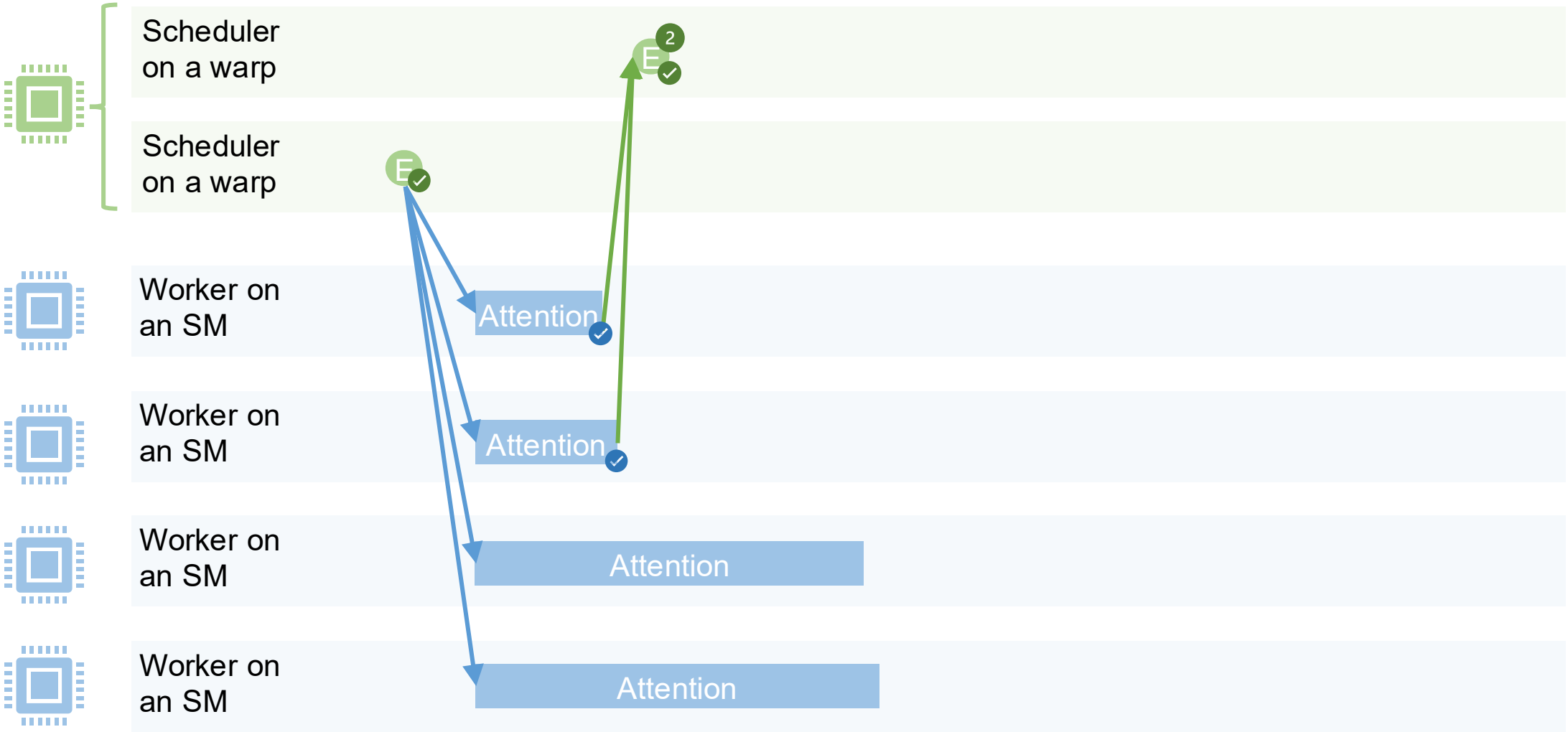
Event-Driven Execution



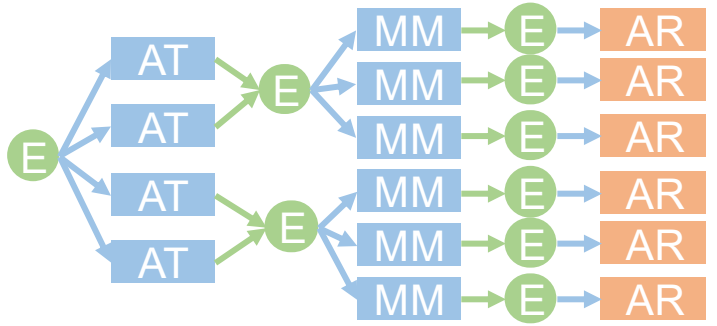


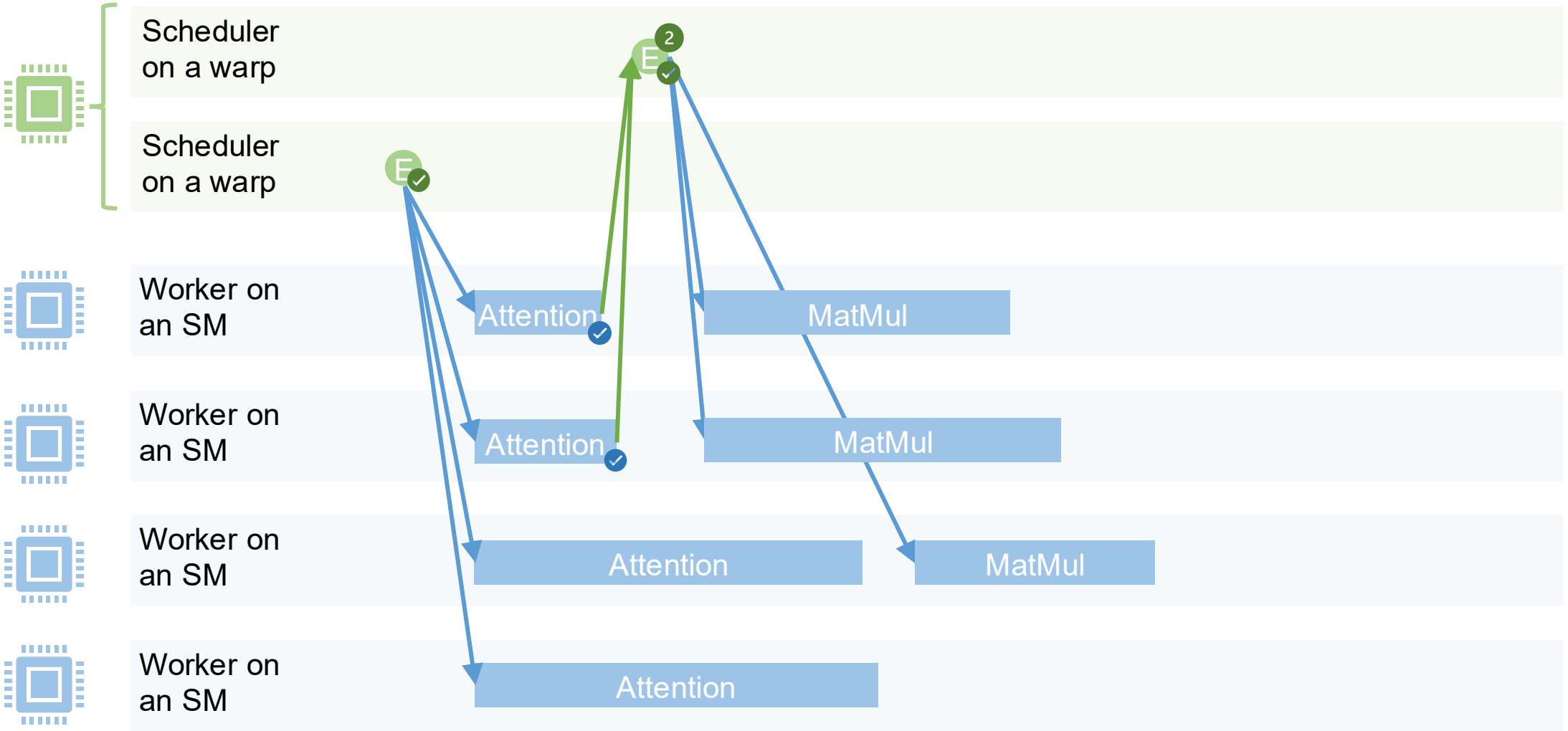
Event-Driven Execution



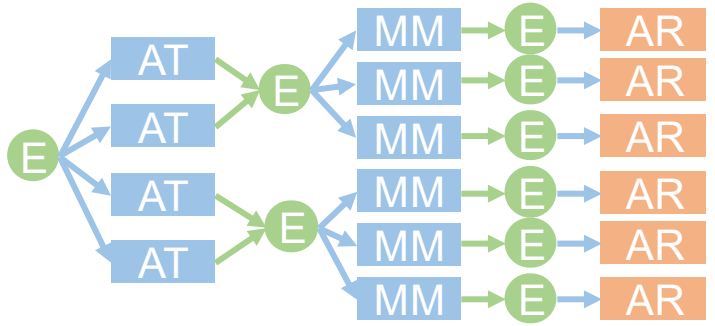


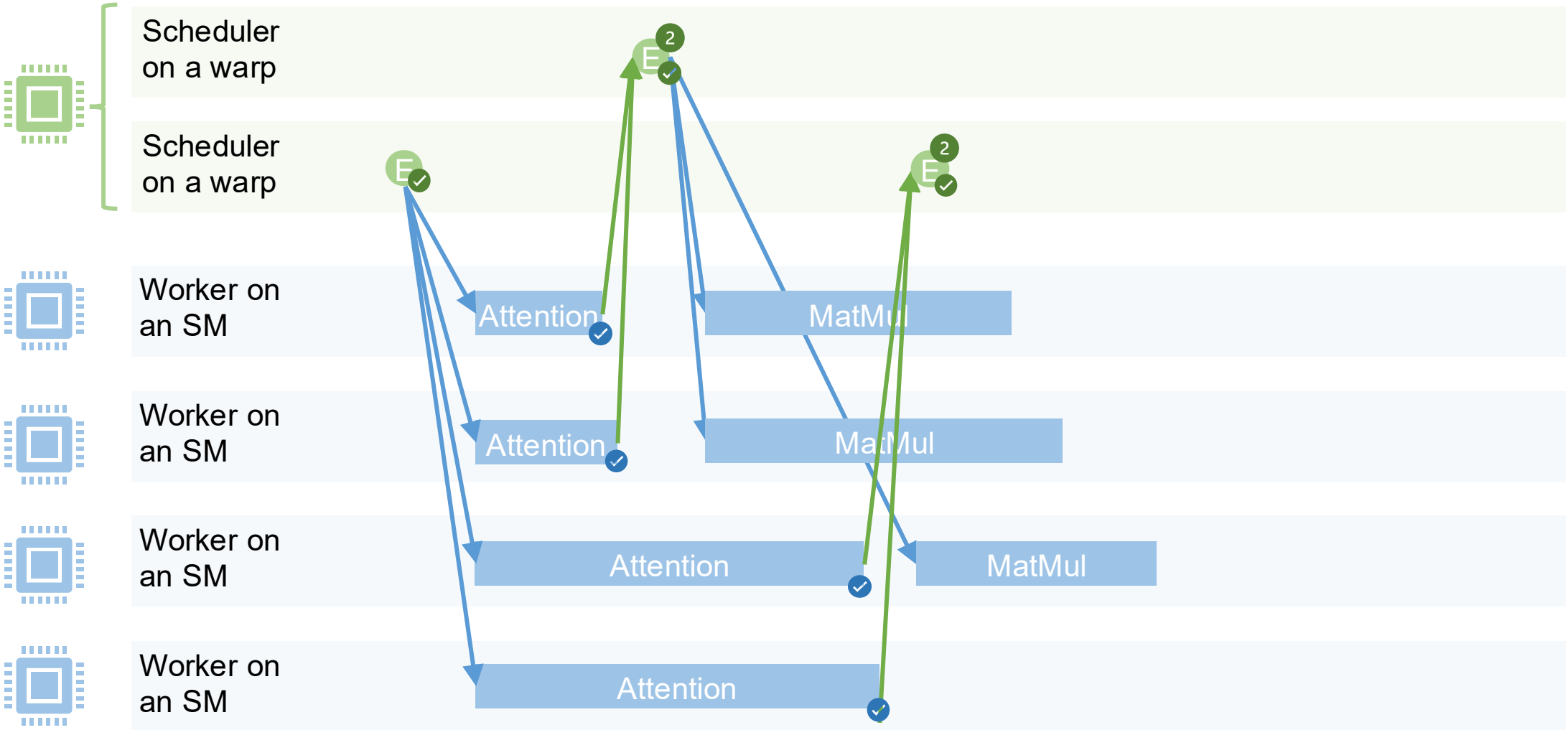
Event-Driven Execution



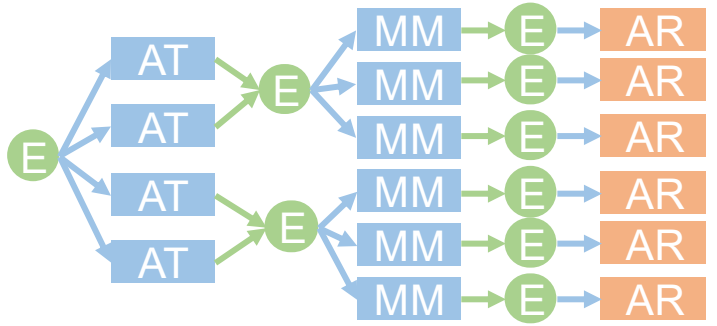


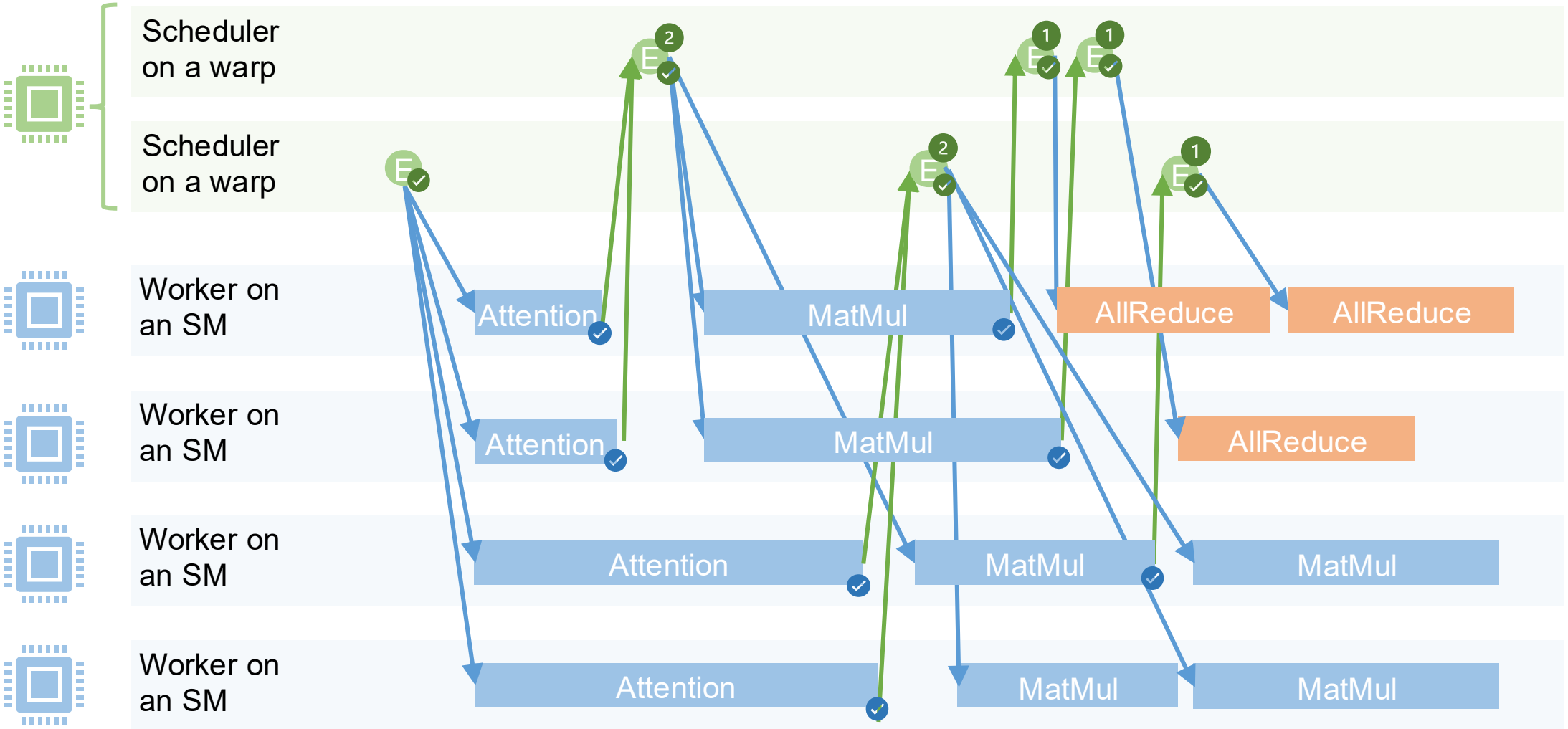
Event-Driven Execution





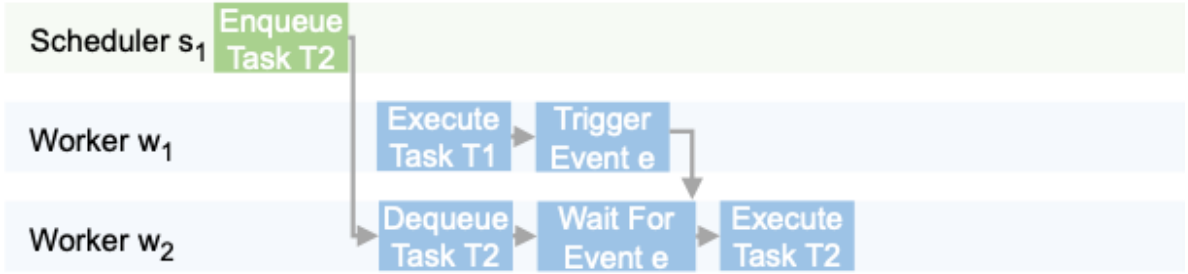
Event-Driven Execution



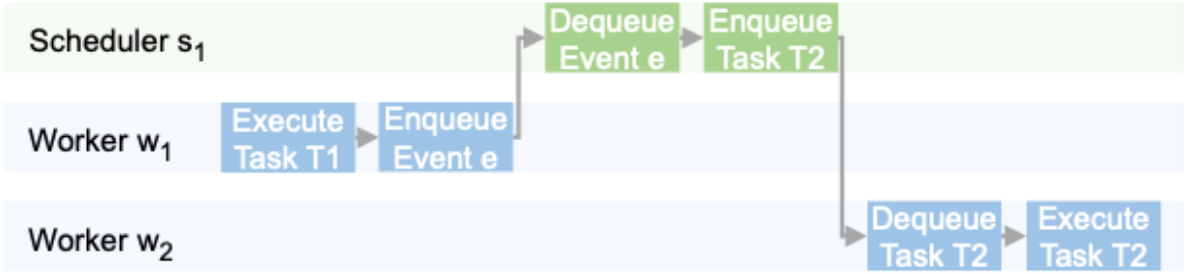


Techniques to Minimize Task Launch Overhead

- **Lightweight workers and schedulers**
  - Task & event queues: circular buffers on device memory
  - Event notification, task enqueue/dequeue: atomic operations
- **Decentralized scheduling**
  - Schedulers assign tasks using only local information
- **Hybrid task launch**
  - Compiler classifies each op as AOT/JIT based on runtime load balance



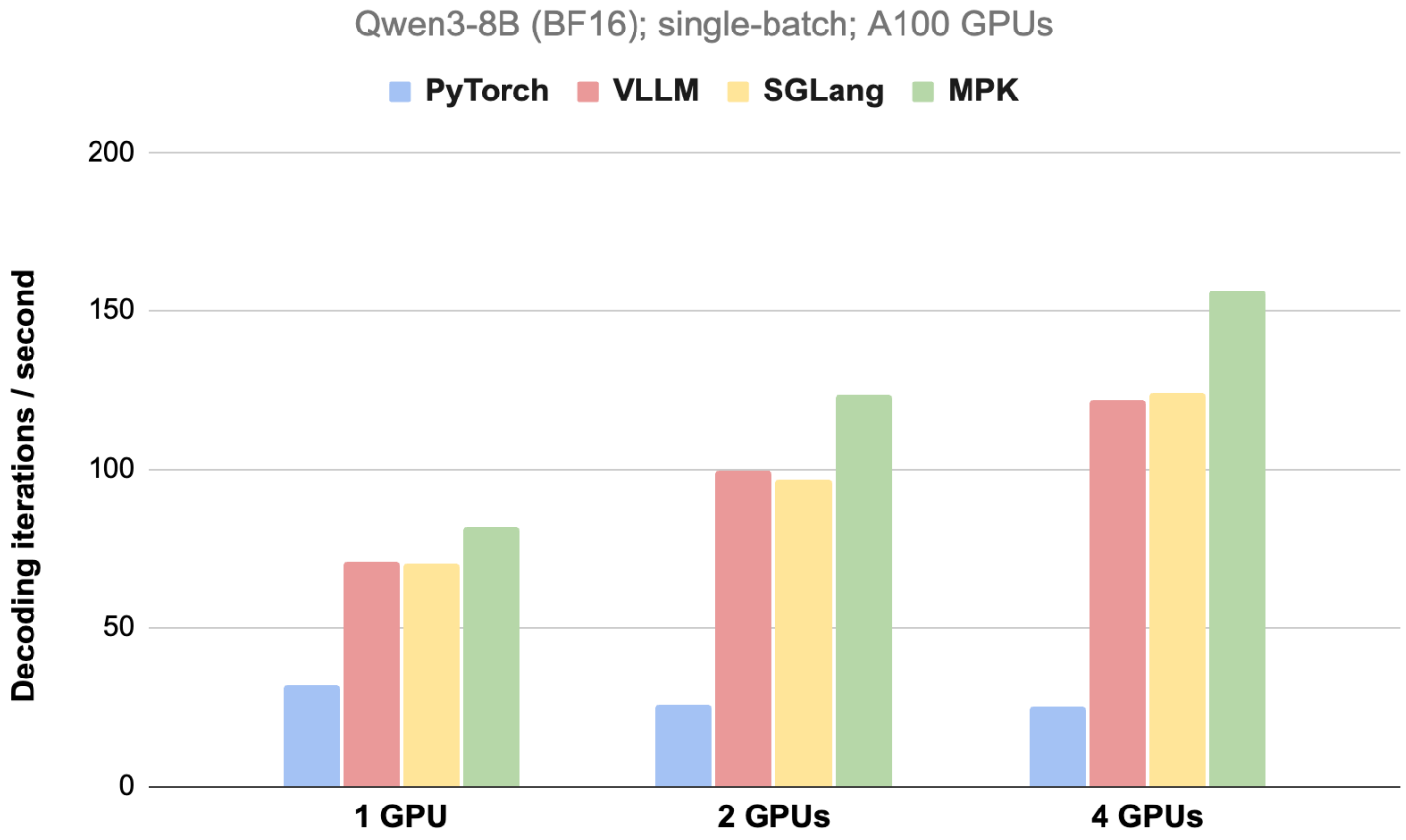
Ahead-of-time: launched before event trigger to reduce latency



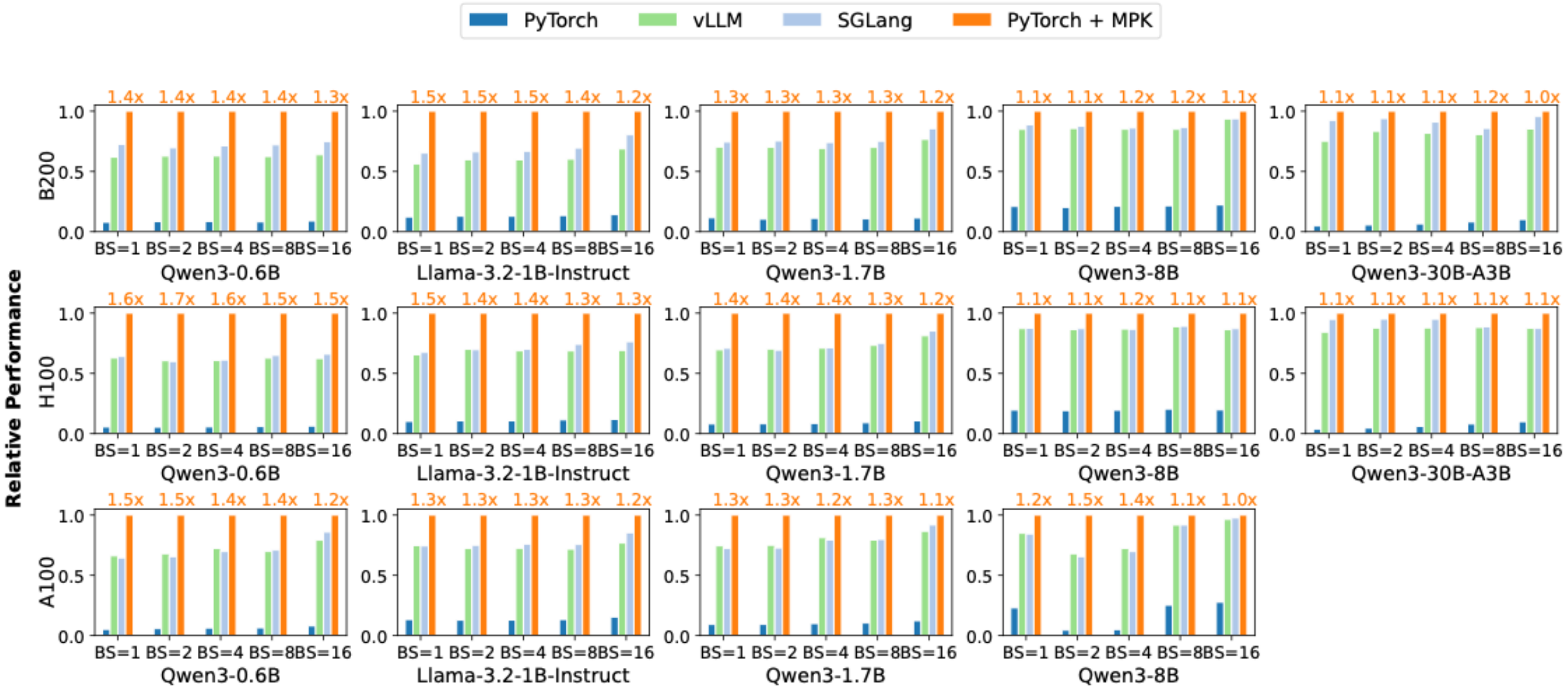
Just-in-time: launched after event trigger to balance load

Pushing LLM Inference Latency Towards HW Limits

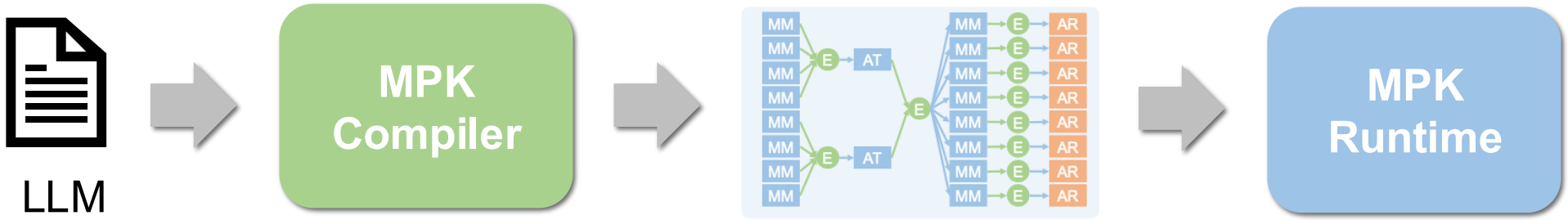
- Reducing Qwen3-8B per-token latency from 14.5ms to 12.5ms
- Approaching theoretical bound of 10ms



Consistently Outperforms Existing Systems Across Models, Batch Sizes, and GPUs



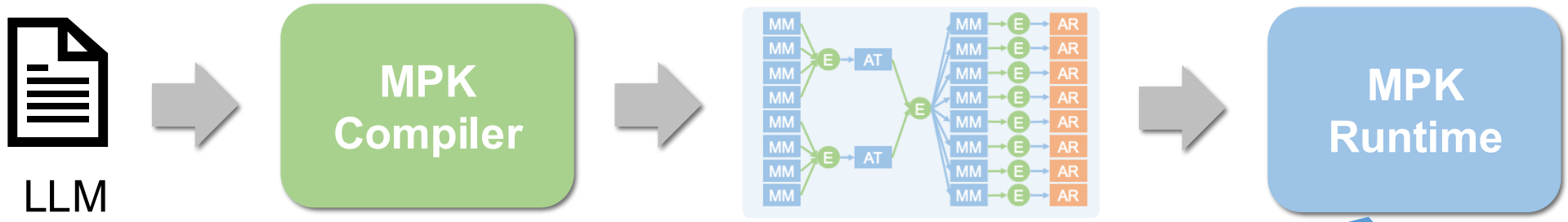
Open Questions



How to decompose layers into tasks?

- Currently rely on heuristics + profiling-based tuning
- Optimized for A100, H100, B200
- Users want portability across diverse GPUs
- Need more automated methods

Open Questions



How to schedule tasks across workers?

Round-Robin

✅ no synchronization with workers

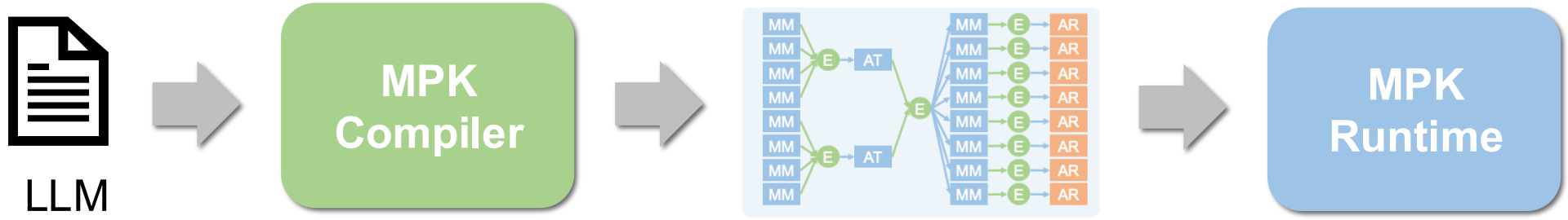
❌ load imbalance

Dynamic Scheduling

✅ adapt to runtime load

❌ coordination overhead

MPK: Compiling LLMs into a Mega-Kernel



https://github.com/mirage-project/mirage/