Programming Techniques for Optimizing ML on GPUs

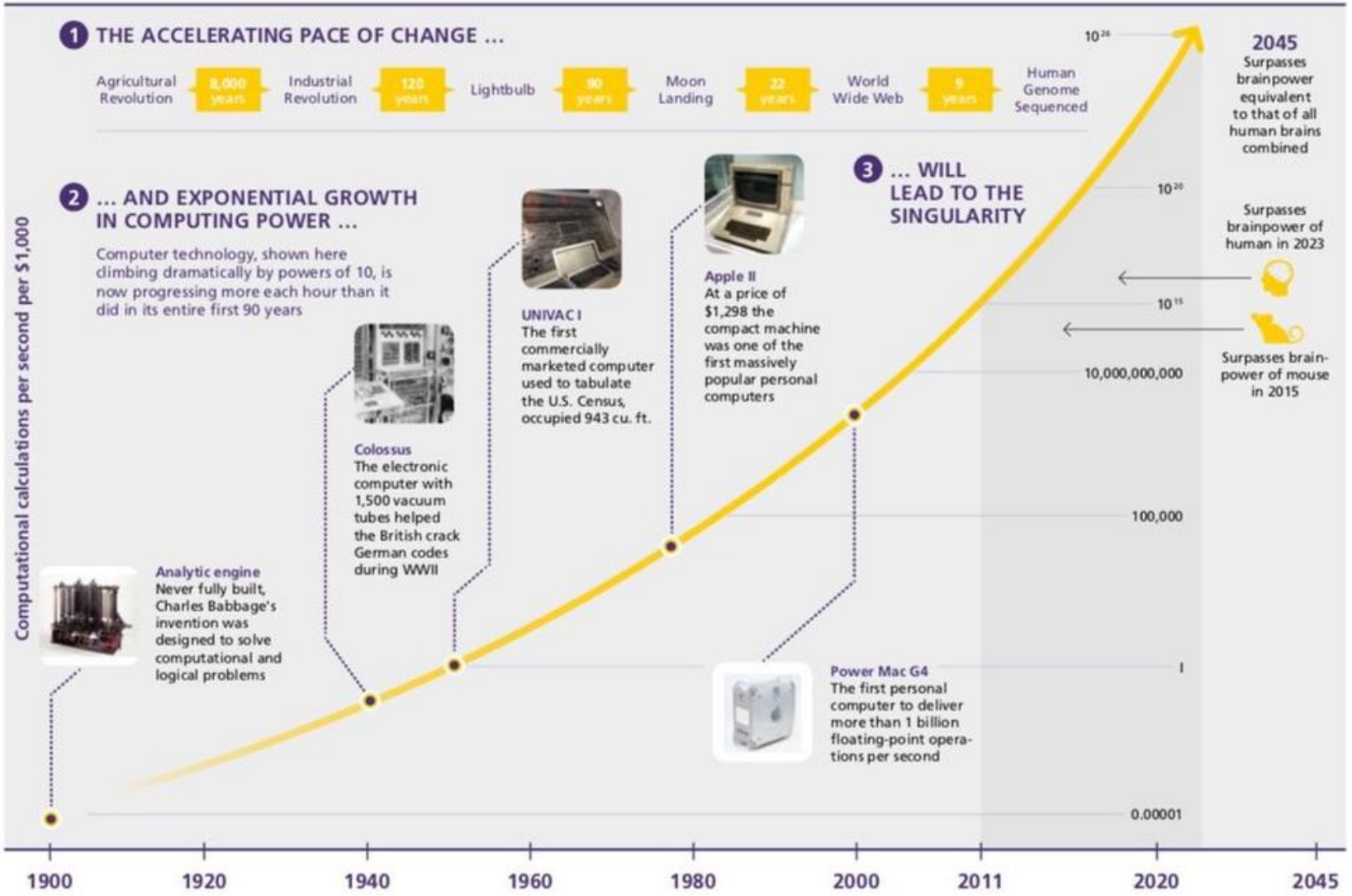
Zhihao Jia

Computer Science Department

Carnegie Mellon University

What are the fundamental driving forces behind the success of ML?

Compute Per Second Per Dollar

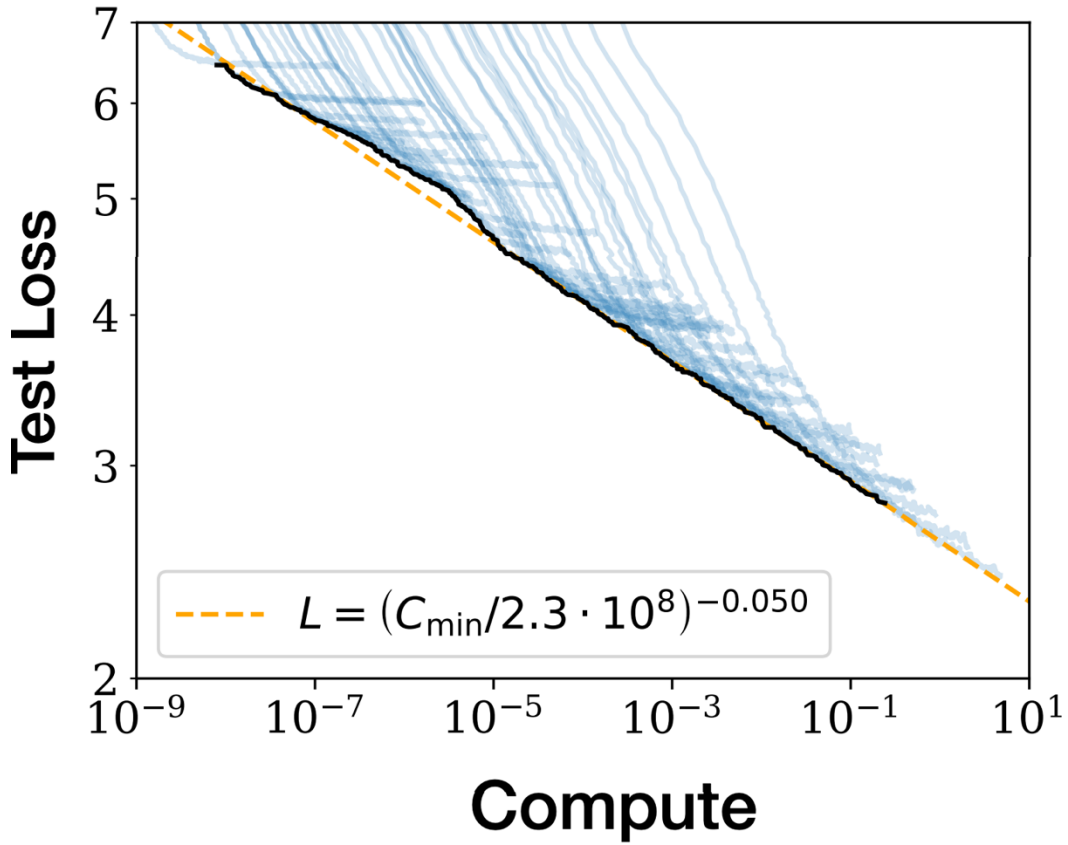


Surpass human brainpower in 2023

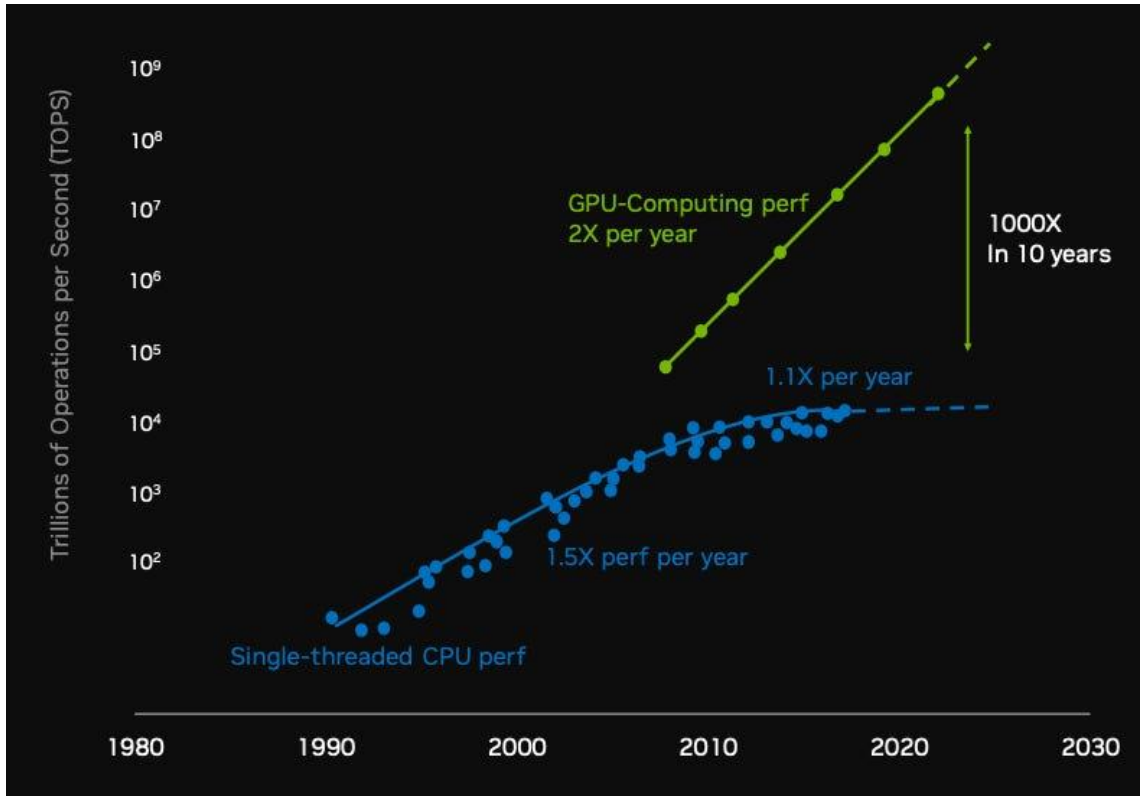
* Ray Kurzweil. *The Singularity Is Near: When Humans Transcend Biology.* 2005

Scaling Law in ML

Improving model accuracy & capability ← Scaling training & inference compute ← Hardware parallelization and specialization

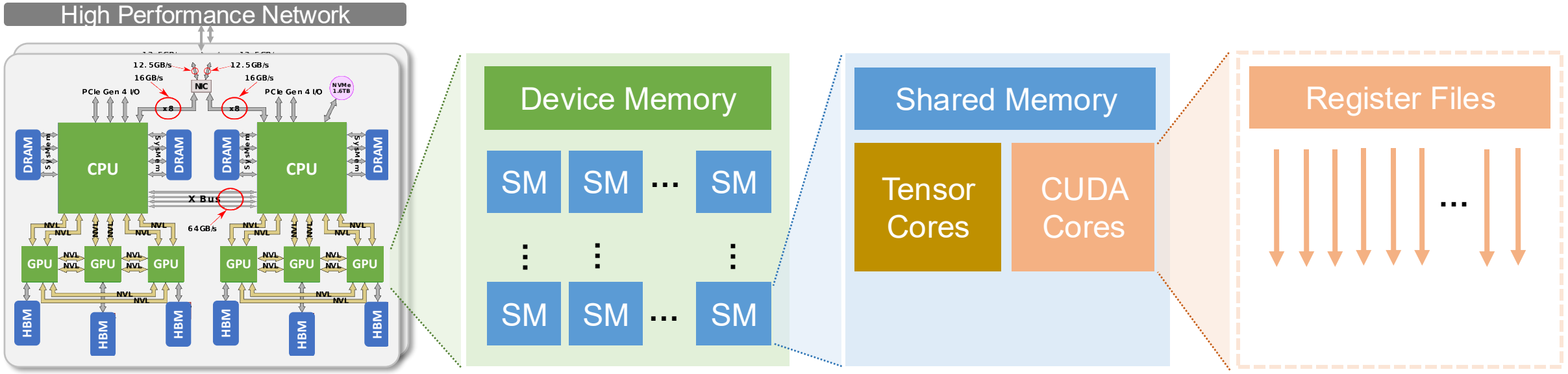


Source: OpenAI



Source: NVIDIA

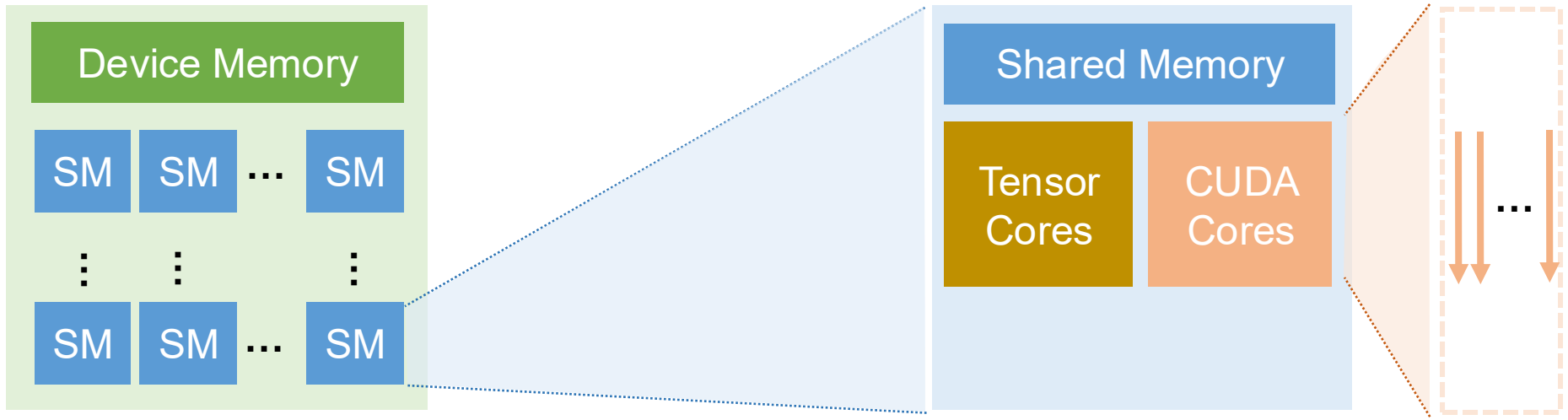
ML Hardware is Massively Parallel, Highly Heterogeneous



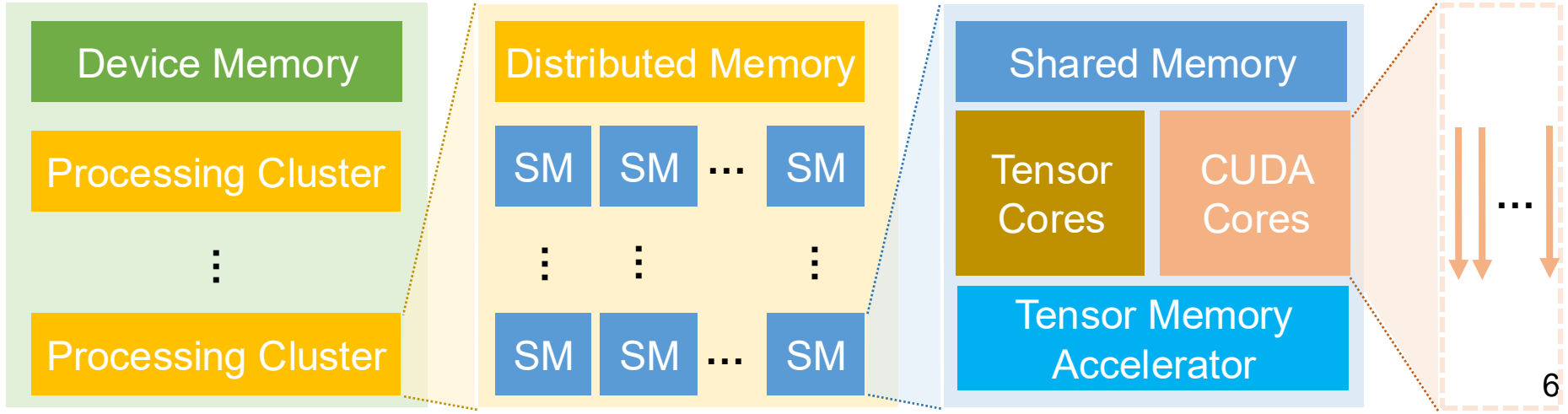
1K-100K GPUs × 100s streaming multiprocessors (SMs) × 128 CUDA cores 4 tensor cores = 10M-1B CUDA cores 400K-40M tensor cores

ML Hardware is Quickly Evolving

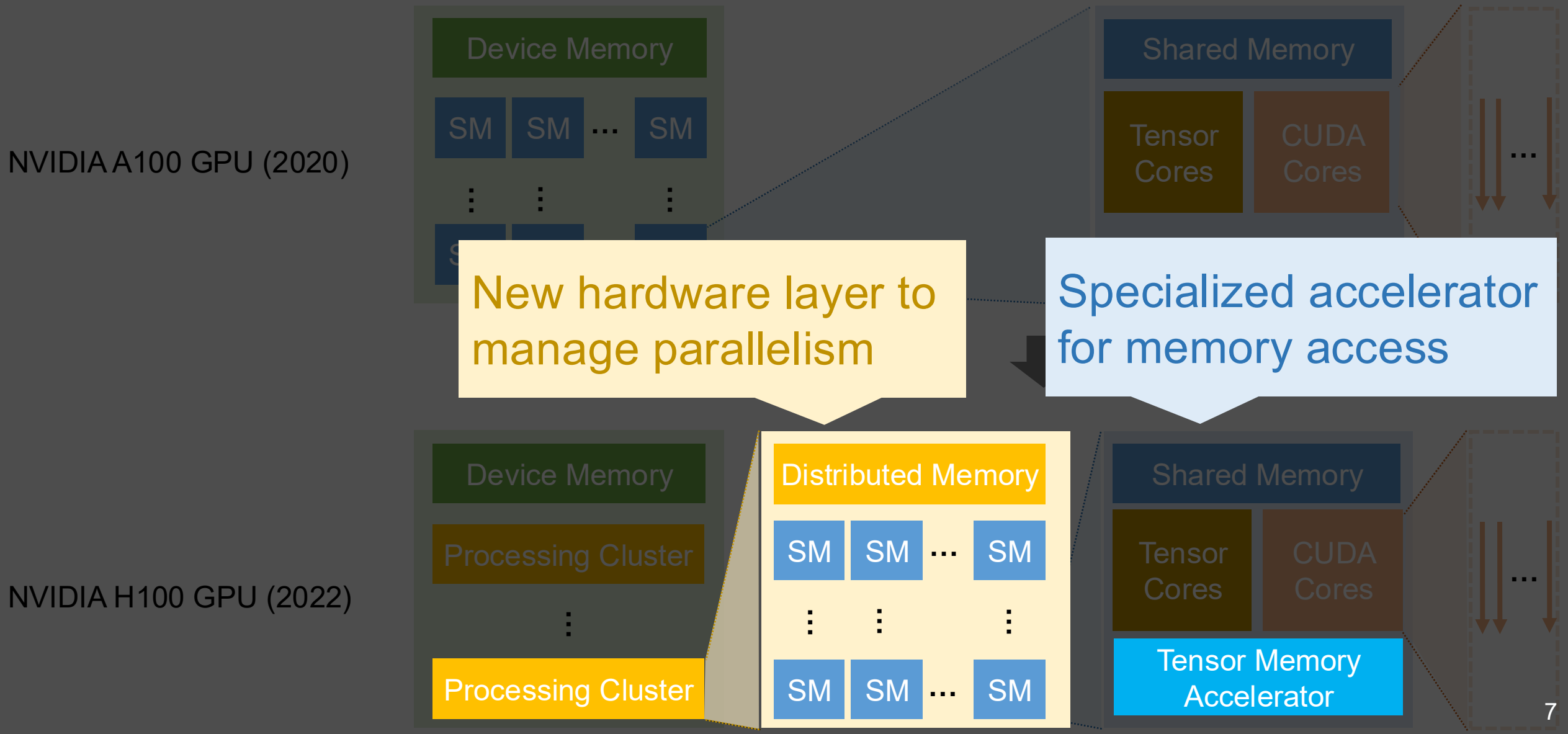
NVIDIA A100 GPU (2020)



NVIDIA H100 GPU (2022)

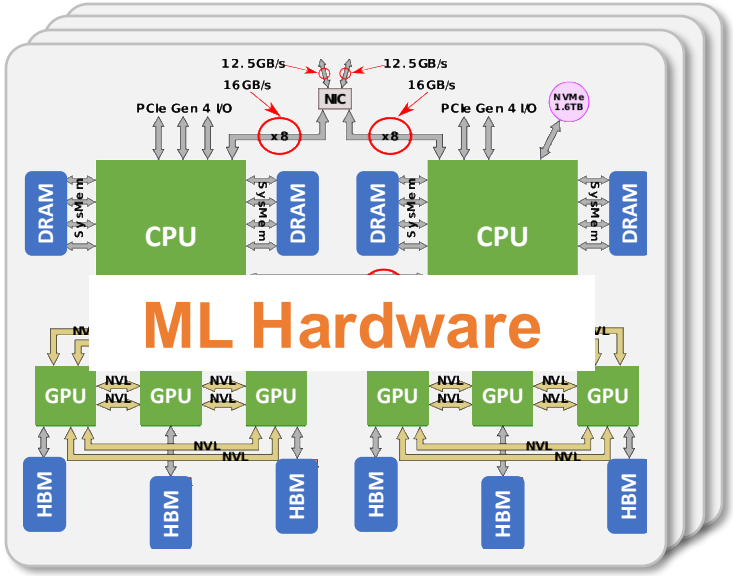


ML Hardware is Quickly Evolving



CMU Automated Learning Systems Lab

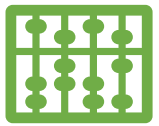
ML Systems



Goal:
Efficiently deploying ML applications on massively parallel, increasingly heterogeneous, rapidly evolving hardware platforms

catalyst https://catalyst.cs.cmu.edu/

Key Challenges for Optimizing ML on GPUs



Massively Parallel

Billions of compute units on modern ML hardware

How can we find the best way to parallelize ML computation?

Increasingly Heterogeneous

CUDA cores, tensor cores, tensor memory accelerators, processing clusters, and more

How can we handle different accelerator types and complex memory hierarchy?

Rapidly Evolving

New generation every 2-3 years; but building high-quality systems & compilers takes much longer

How can we deal with the rapid evolution of ML hardware?

Two Key Techniques for ML Training/Inference

Parallelization: optimize inter-device performance

Kernel Fusion: optimize intra-device performance

Two Key Techniques for ML Training/Inference

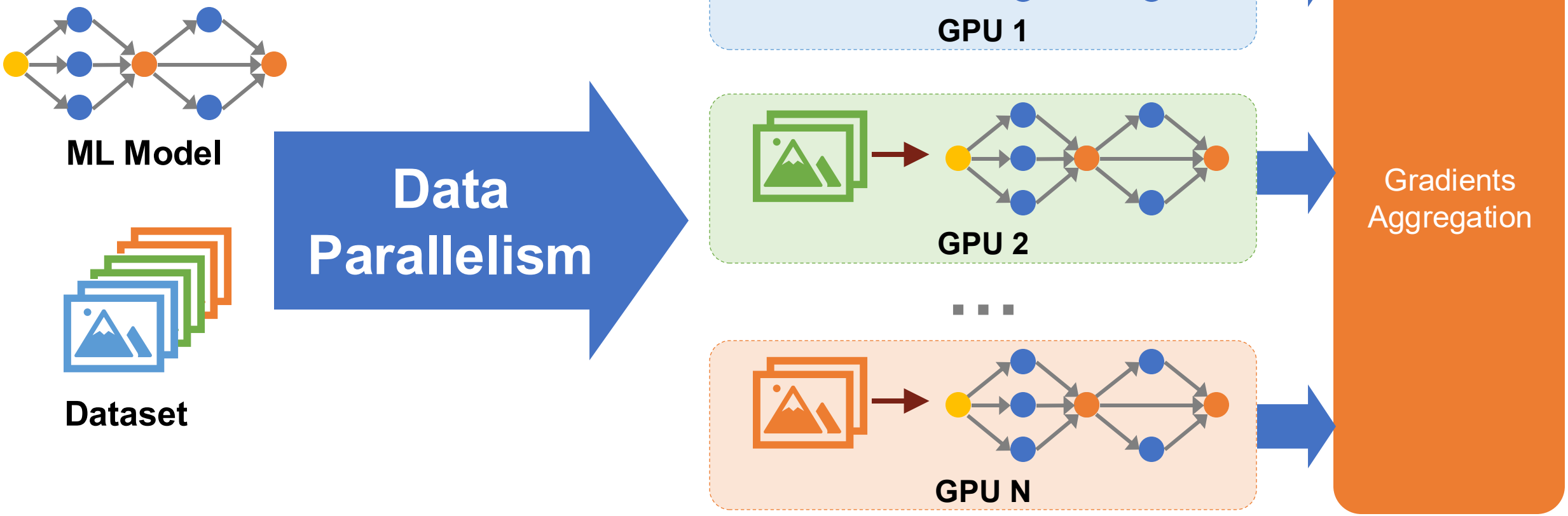
Parallelization: optimize inter-device performance

Many ways to parallelize ML across devices

- Data Parallelism
- Model Parallelism
- Pipeline Parallelism
- Sequence Parallelism
- Attribute Parallelism
- Reduction Parallelism
- Expert Parallelism
- …

Kernel Fusion: optimize intra-device performance

Data Parallelism

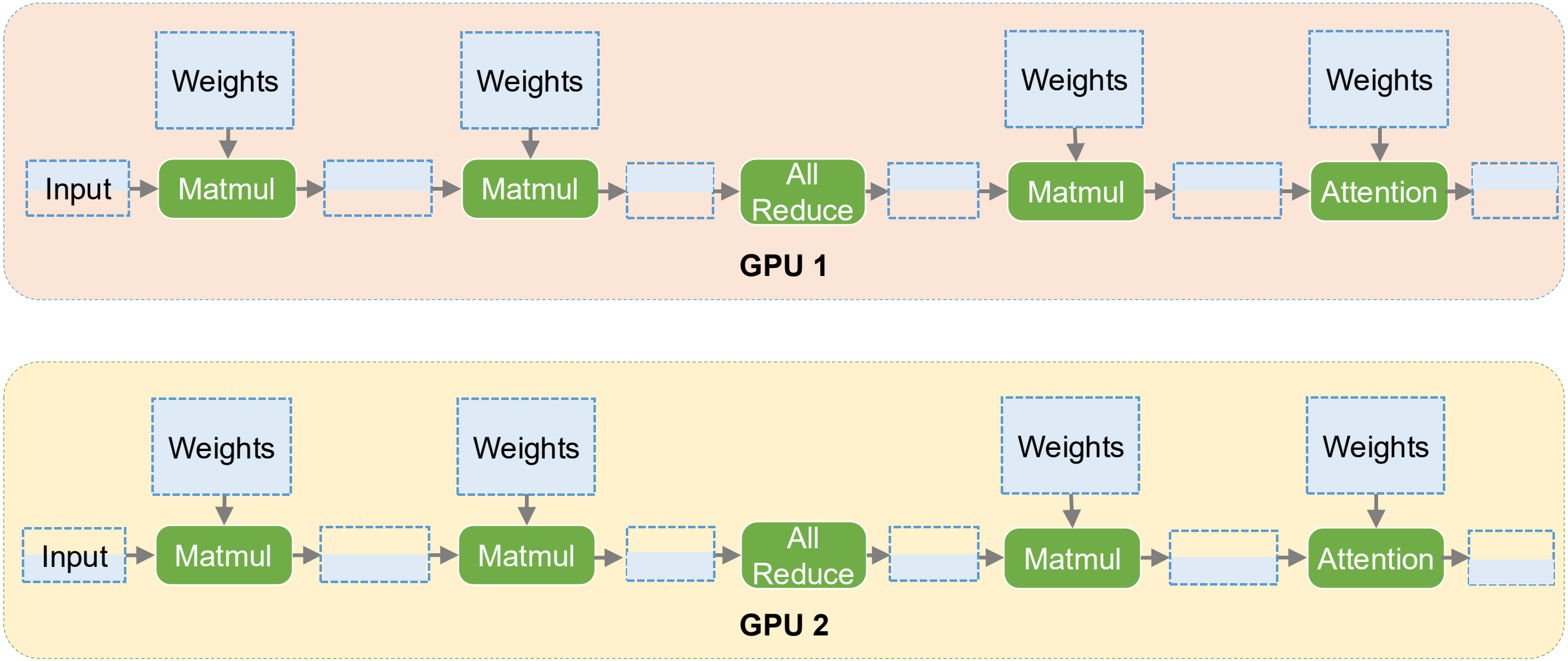


1. Partition dataset into batches

2. Forward/backward of each batch on a GPU

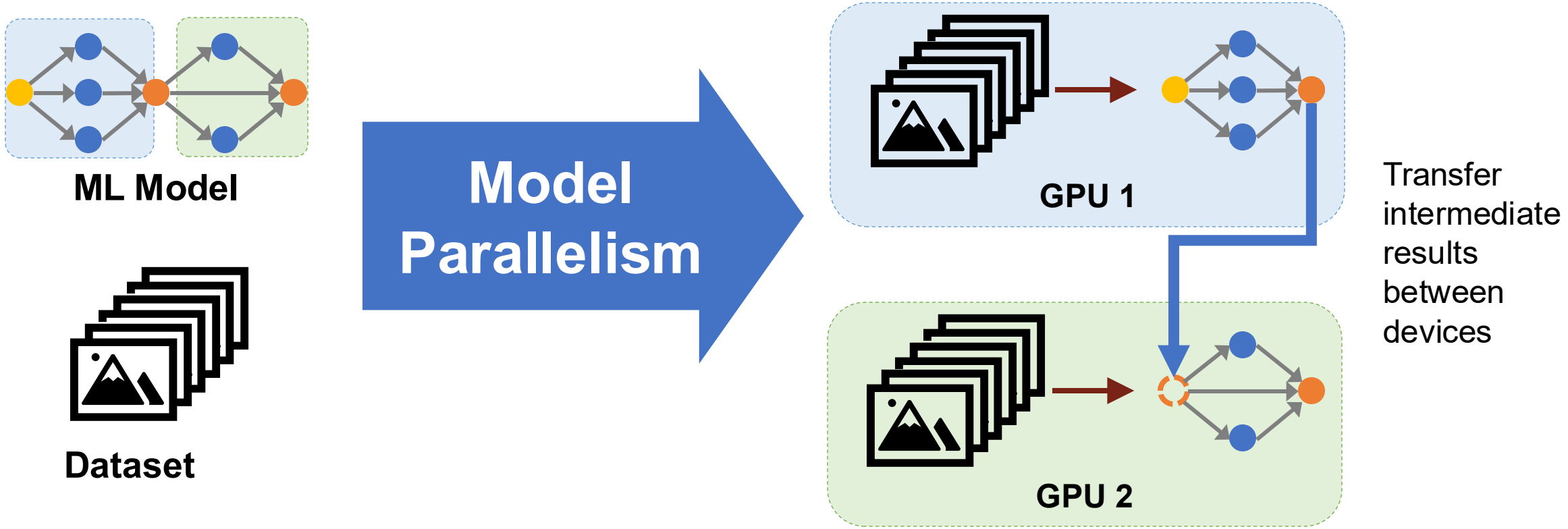
3. Aggregate gradients across GPUs

Data Parallelism for Transformer

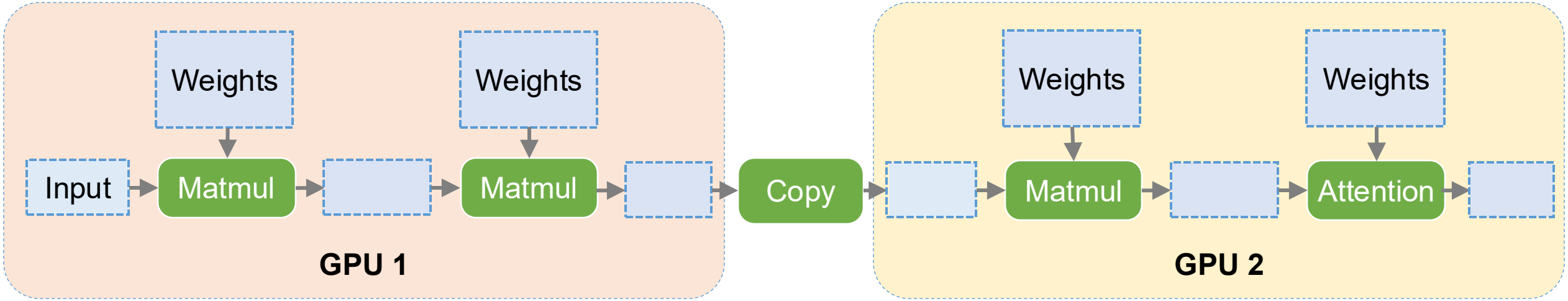


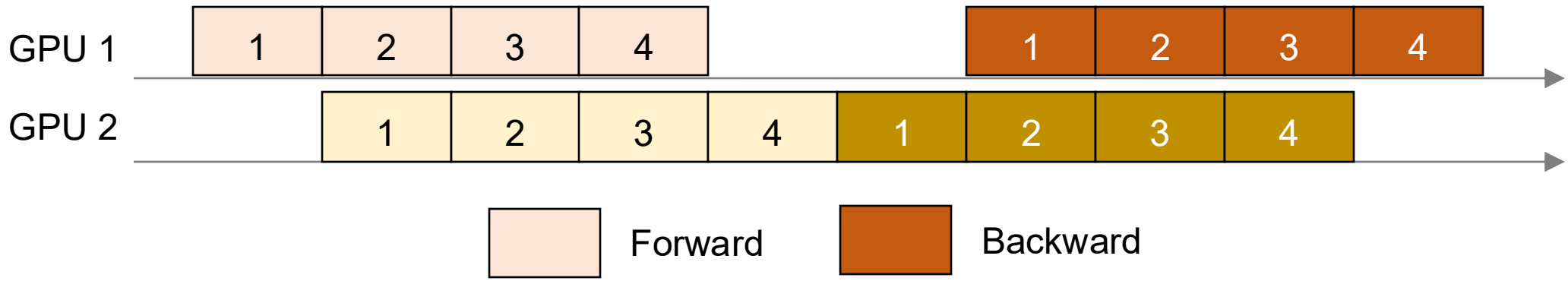
Model Parallelism

- Split a model into multiple subgraphs and assign them to different devices

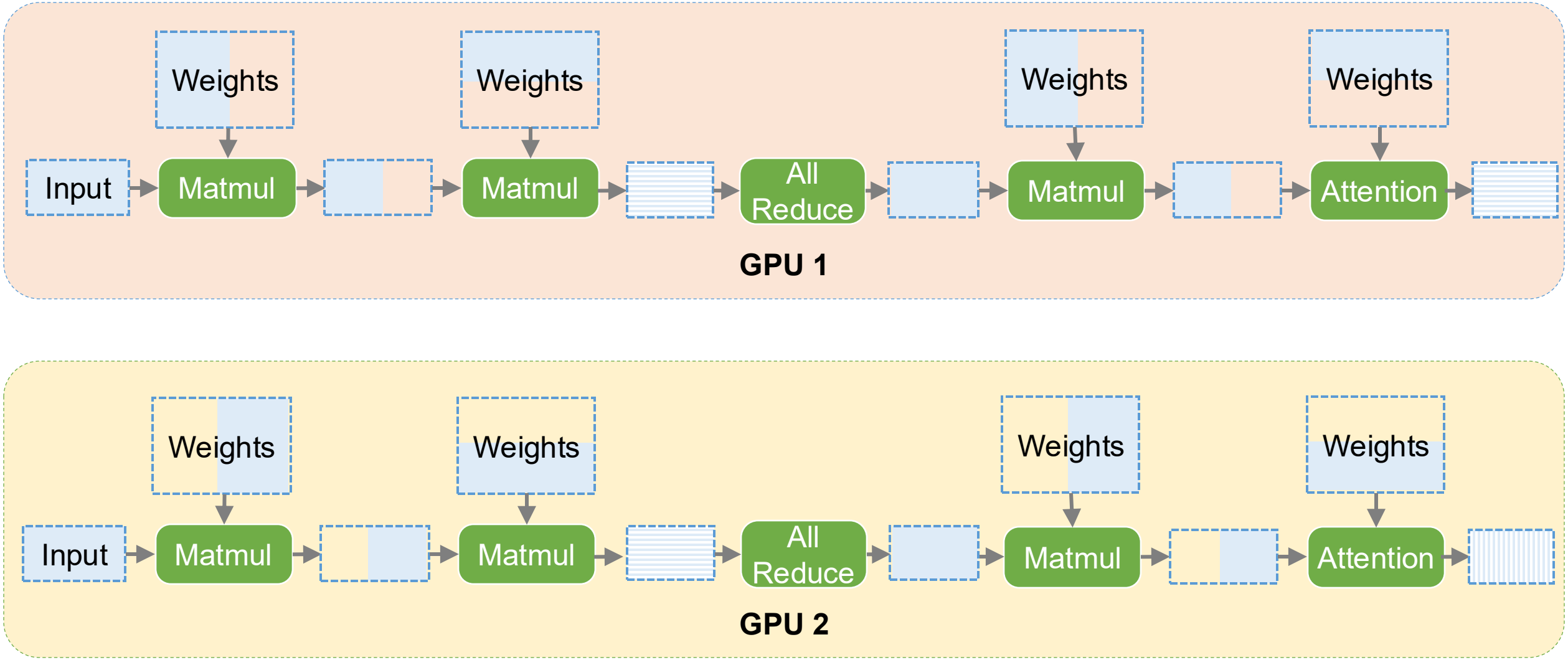


Pipeline Model Parallelism for Transformer

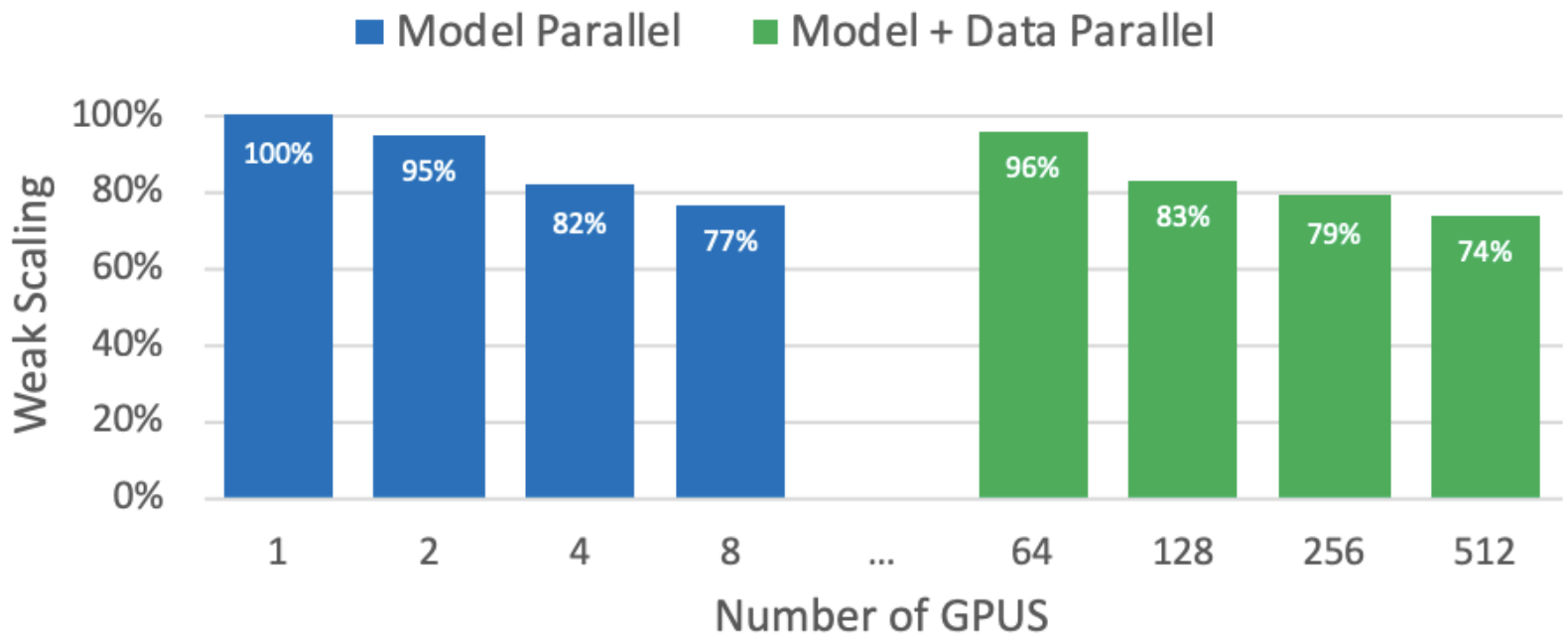




Tensor Model Parallelism for Transformer



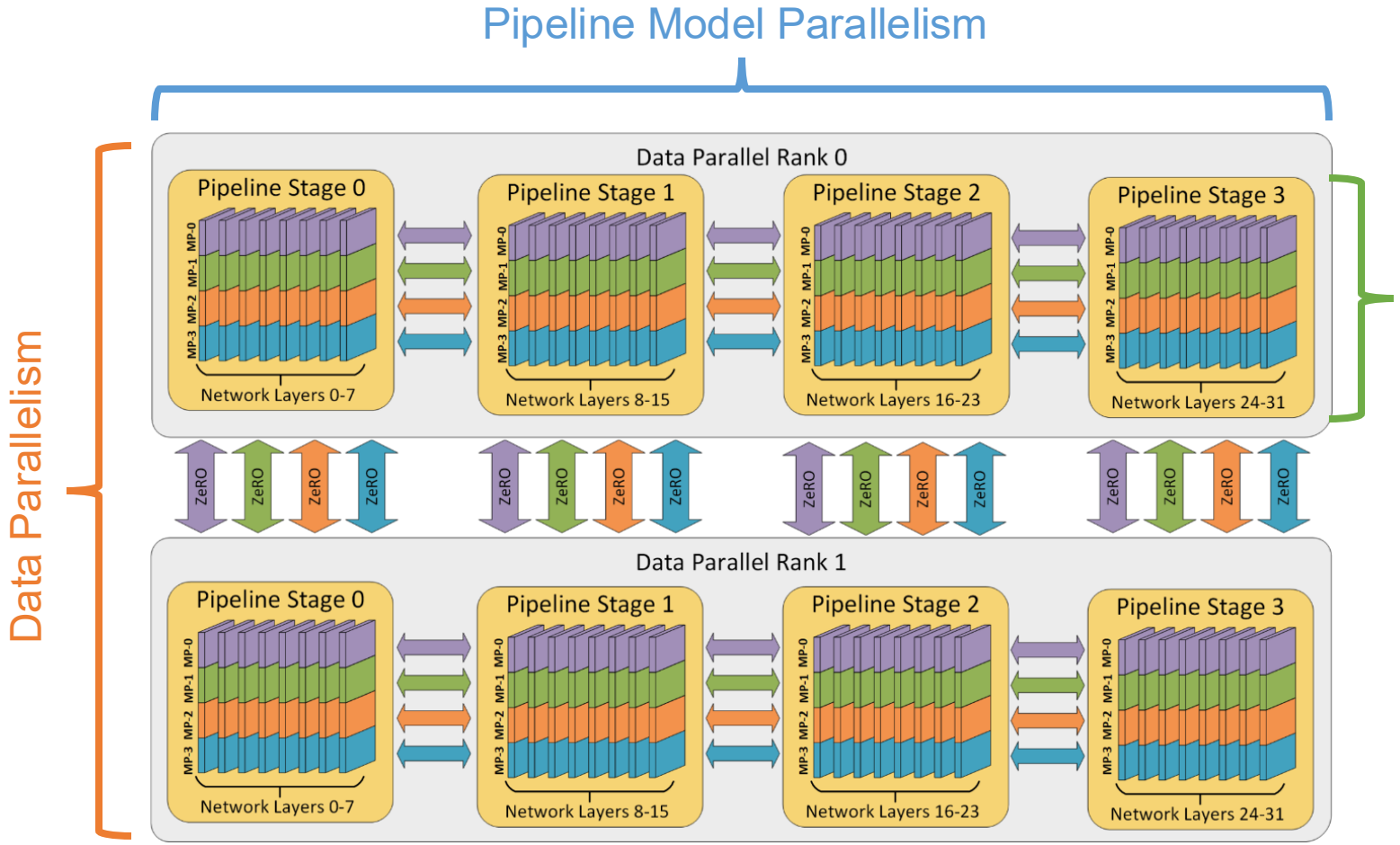
Important to Combine Different Parallelization Strategies

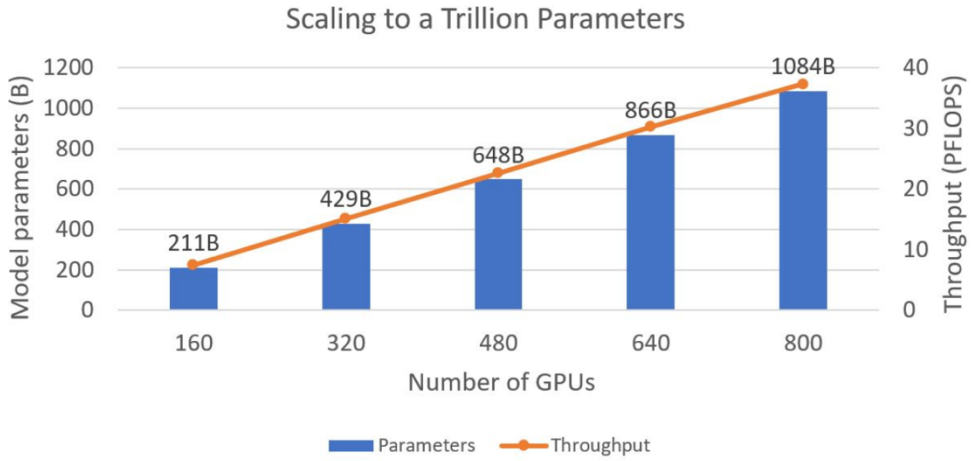


Scale to 512 GPUs by combining data and model parallelism

Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism.

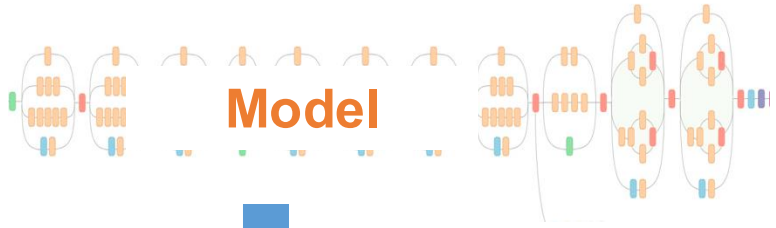
3D Parallelism for LLM Training





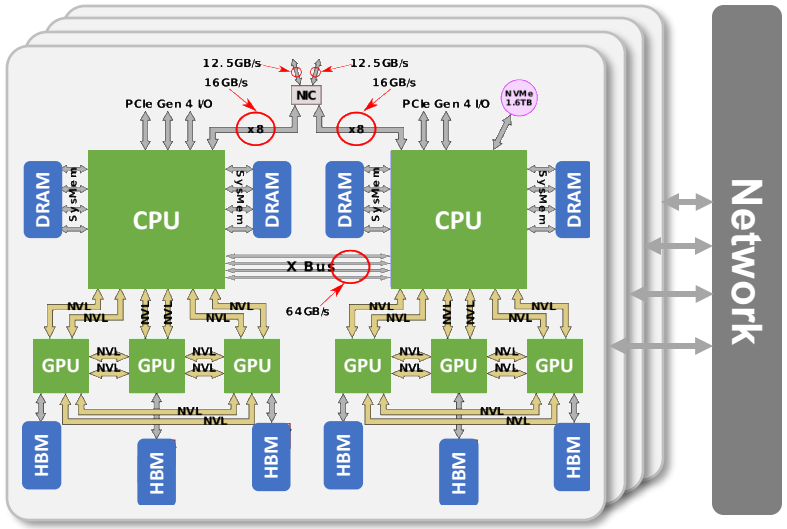
https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone/

Best Way to Parallelize ML Training/Inference?



ML Systems

Parallelization Strategy

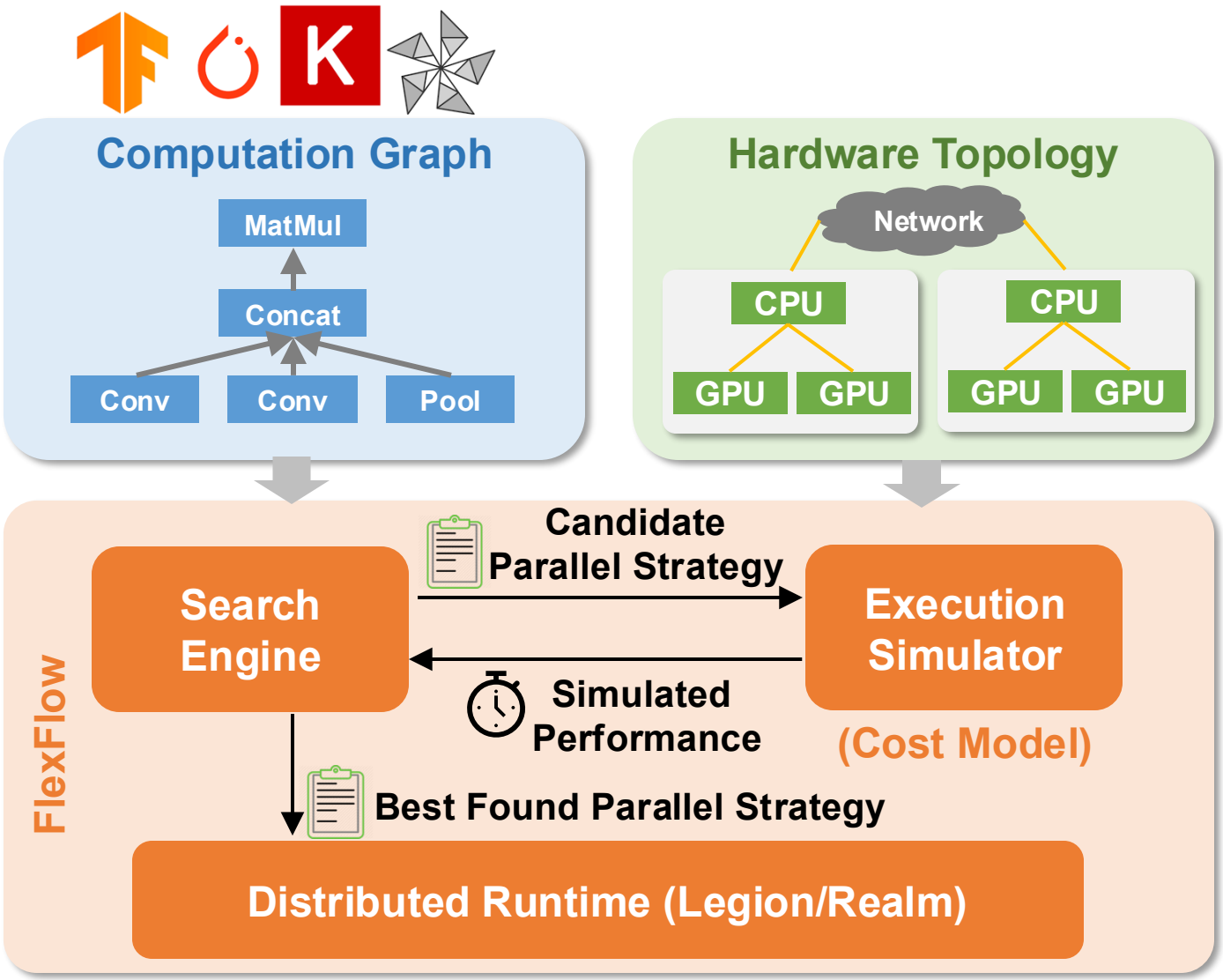


- Hard to manually design
- Suboptimal performance
- Limited portability

- Data Parallelism
- Tensor Parallelism
- Pipeline Parallelism
- Sequence Parallelism
- Attribute Parallelism
- Reduction Parallelism
- Expert Parallelism
- …

2|Parallelization|

FlexFlow: Automatically Optimizing DNN Parallelization



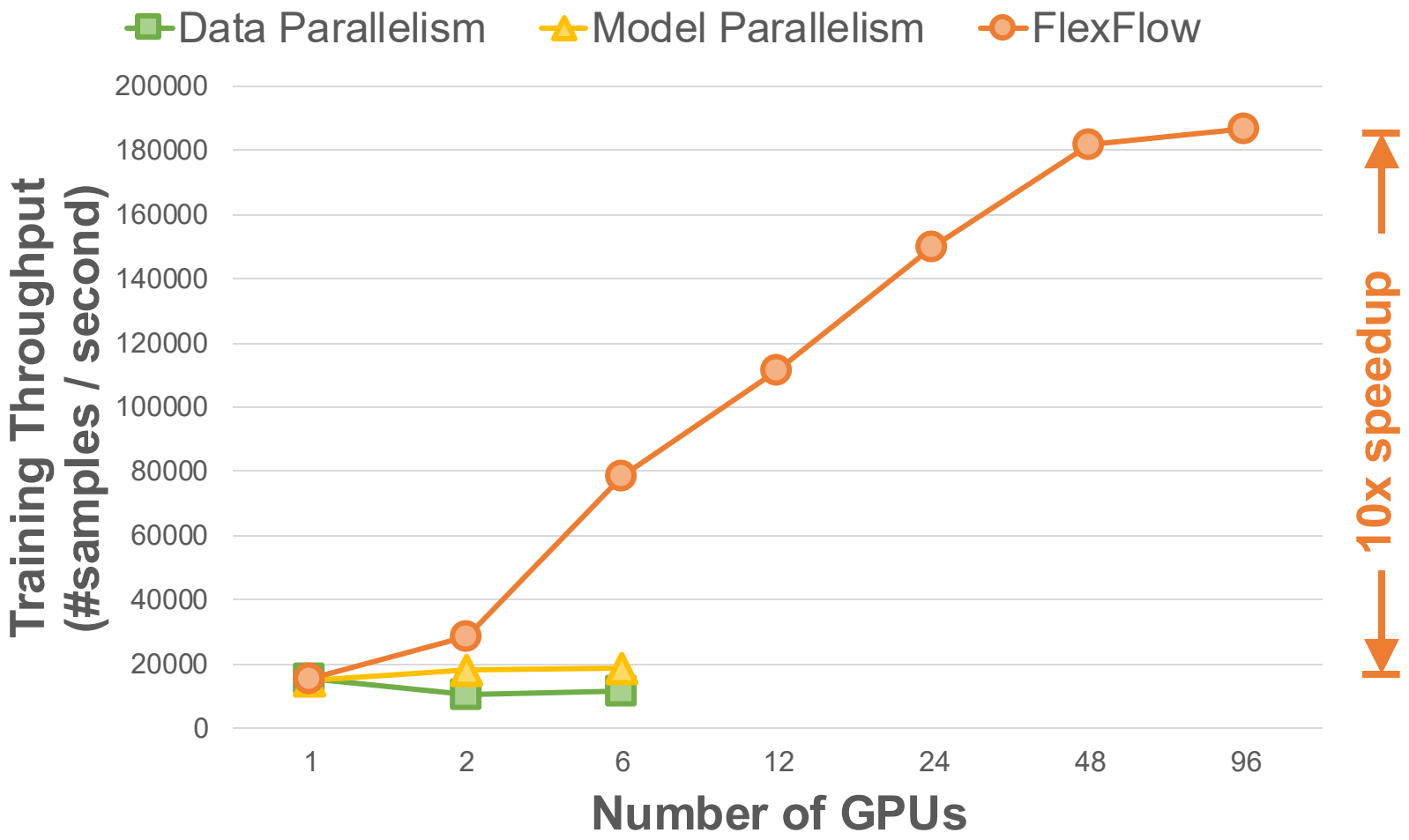
Better Performance
Up to 10x faster than manually designed strategies

Fast Deployment
Minutes of automated search to discover performant strategies

No Manual Effort
Automatically find strategies for new models or hardware platforms

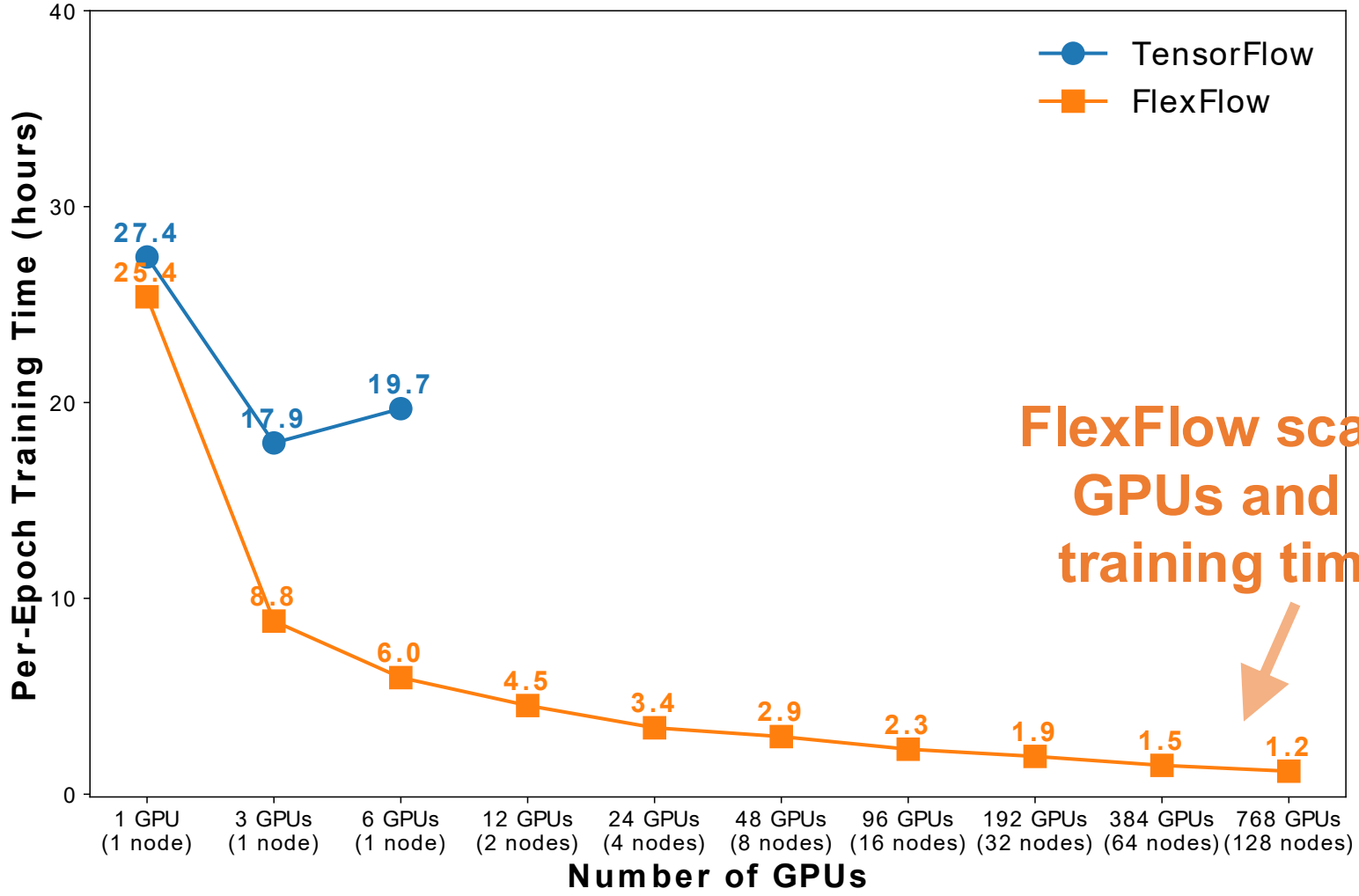
1. FlexLLM: Token-Level Co-Serving of LLM Inference and Fine-Tuning with SLO Guarantees. NSDI'26
2. GraphPipe: Improving Performance and Scalability of DNN Training with Graph Pipeline Parallelism. ASPLOS'25
3. TopoOpt: Optimizing the Network Topology for Distributed DNN Training. NSDI'23
4. Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization. OSDI'22
5. Beyond Data and Model Parallelism for Deep Neural Networks. MLSys'19

Train Large-Scale Recommendation Models in Production



1. Software-Hardware Co-design for Fast and Scalable Training of Deep Learning Recommendation Models. ISCA'22

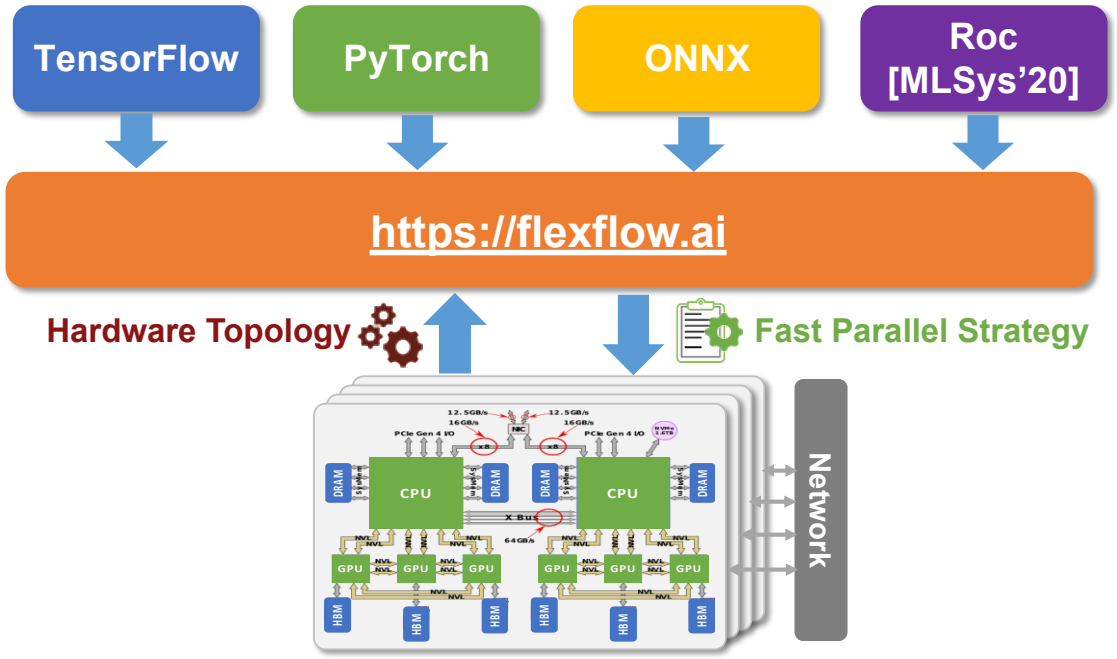
Reduce AI4Science Model Training from Days to Hours



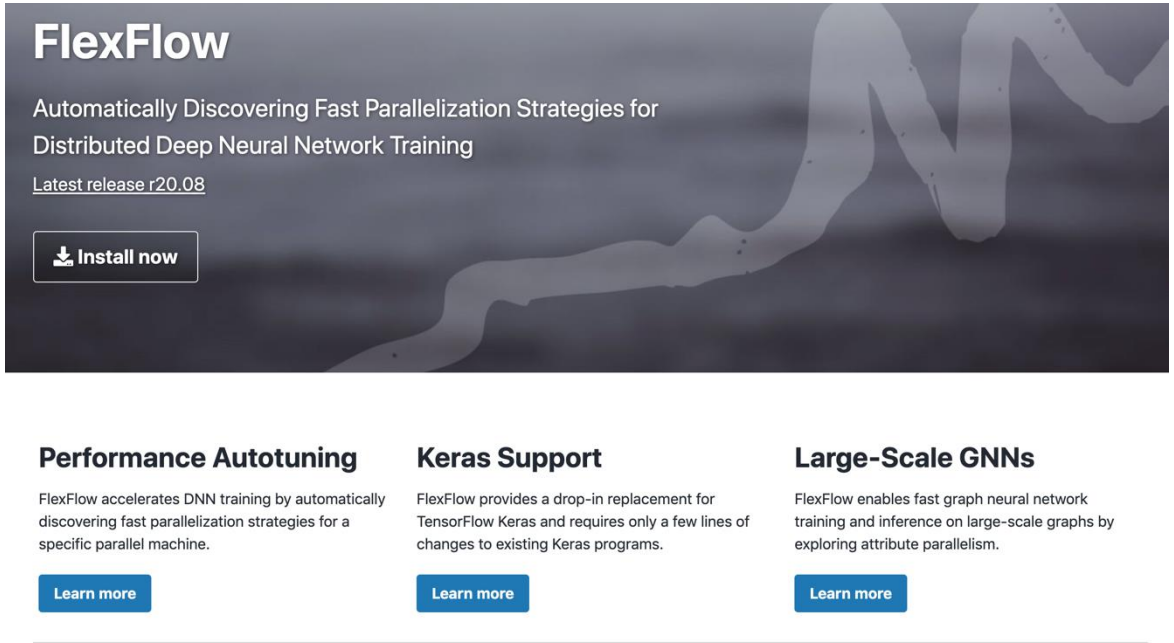
FlexFlow scales to 768 GPUs and reduces training time by 15x

1. Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization. OSDI'22

FlexFlow: Joint Research Effort Across Institutes



https://flexflow.ai



Carnegie Mellon University · Stanford · NVIDIA · SLAC National Accelerator Laboratory · Meta · Los Alamos National Laboratory EST.1943 · MIT · perspecta LABS · catalyst

Two Key Techniques for ML Training/Inference

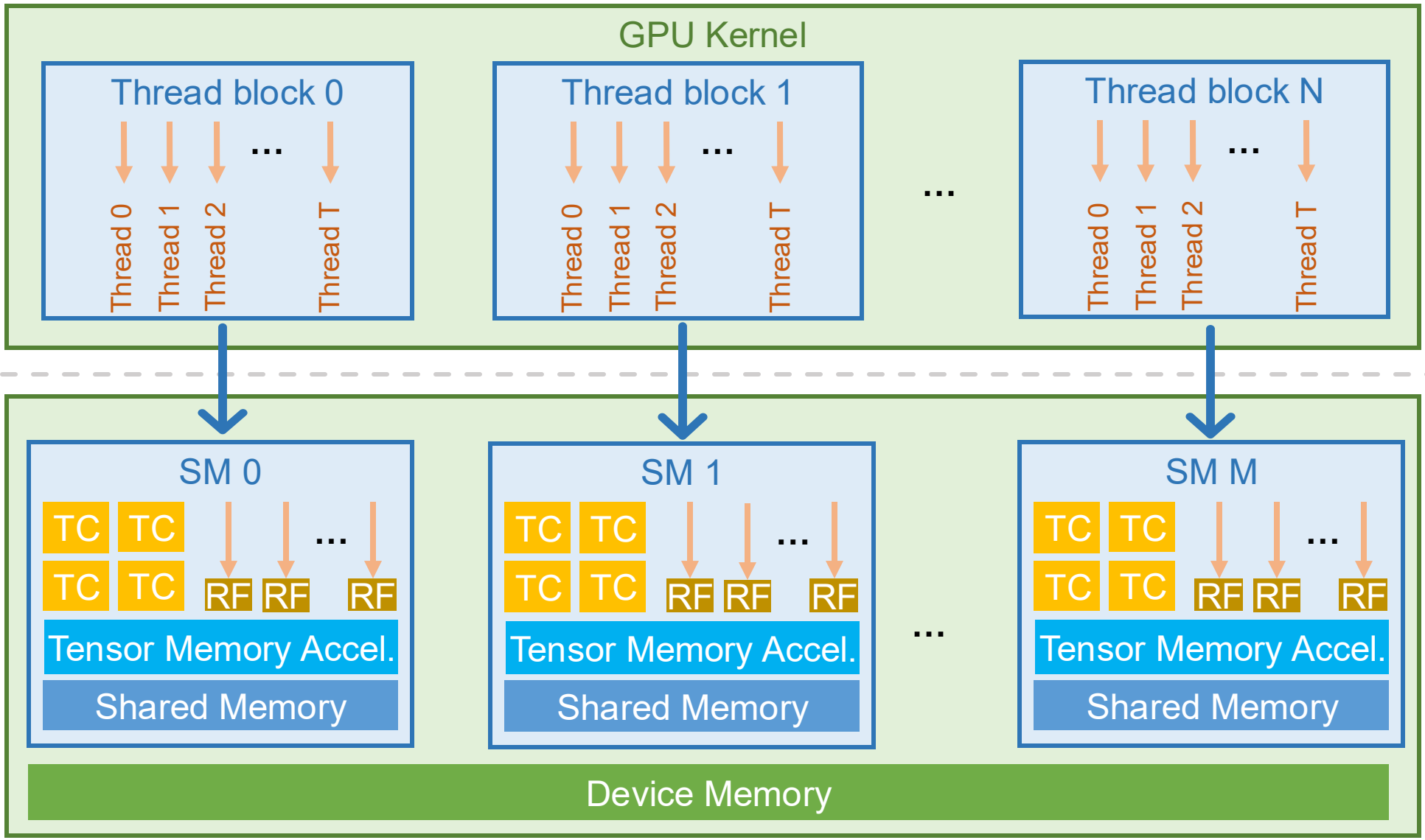
Parallelization: optimize inter-device performance

Kernel Fusion: optimize intra-device performance

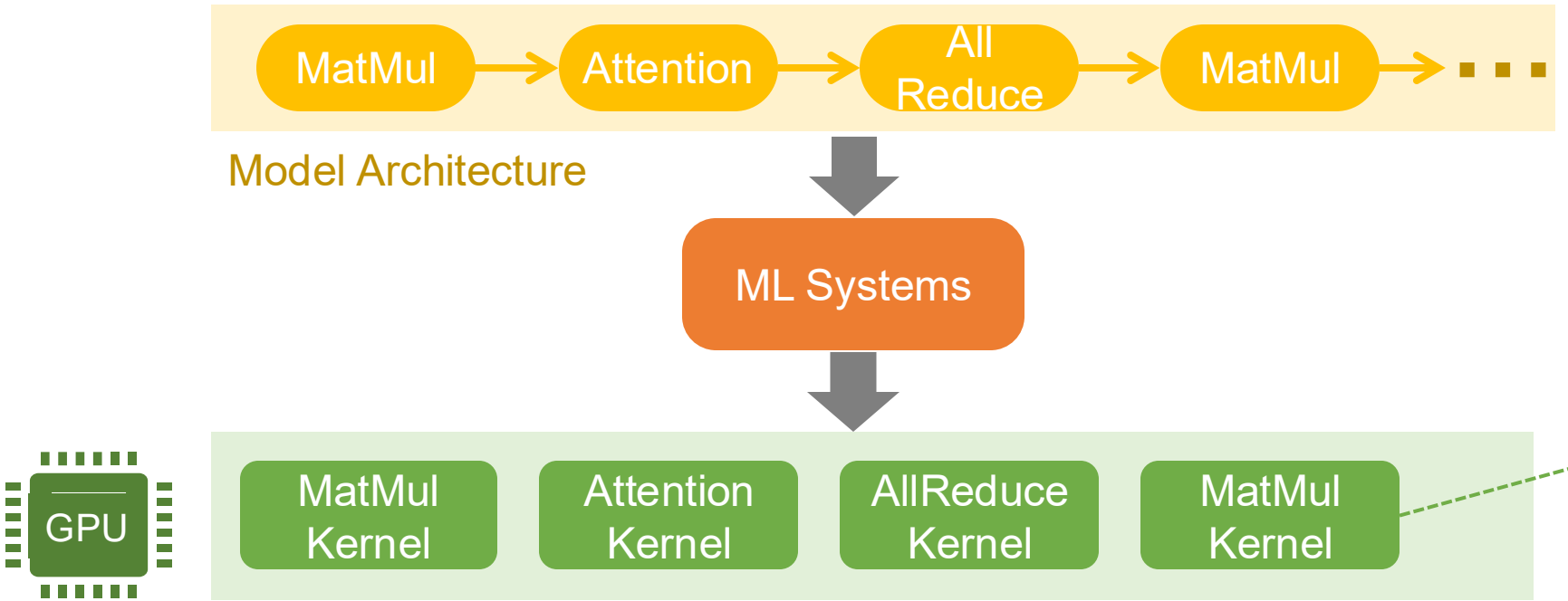
What is a Kernel?

Programming Abstraction

Hardware Architecture

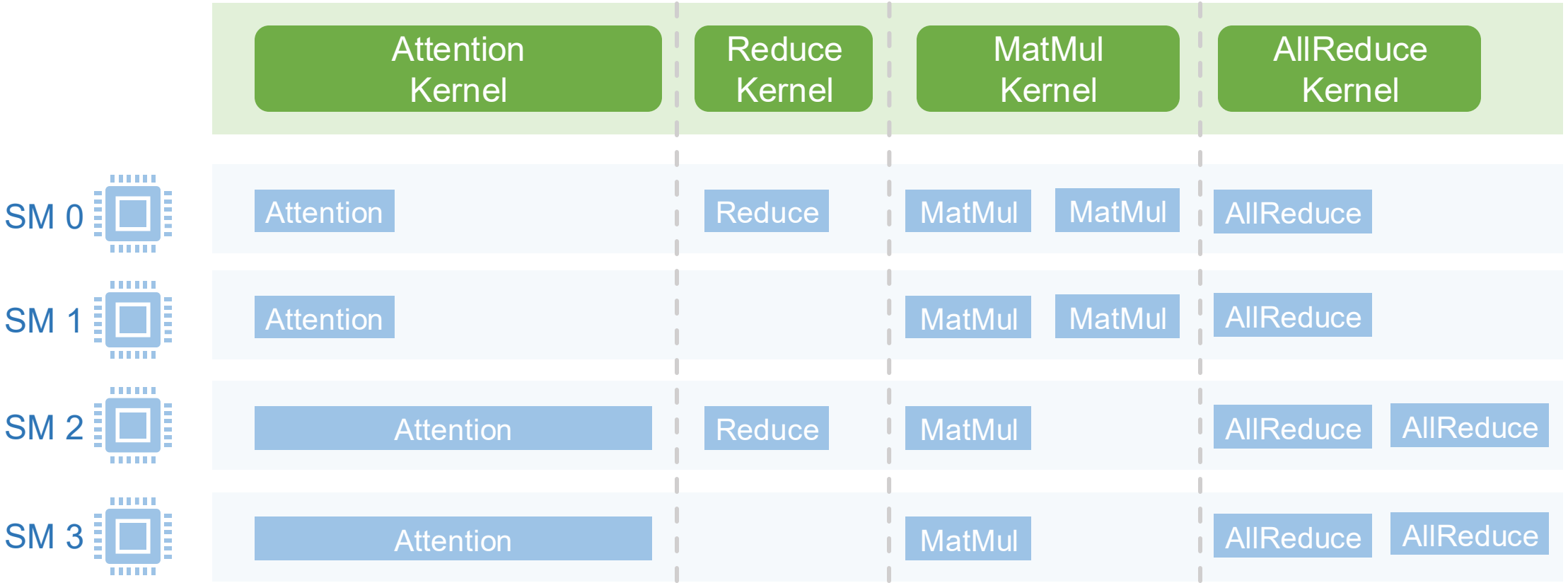


Existing Kernel-Per-Layer Approach

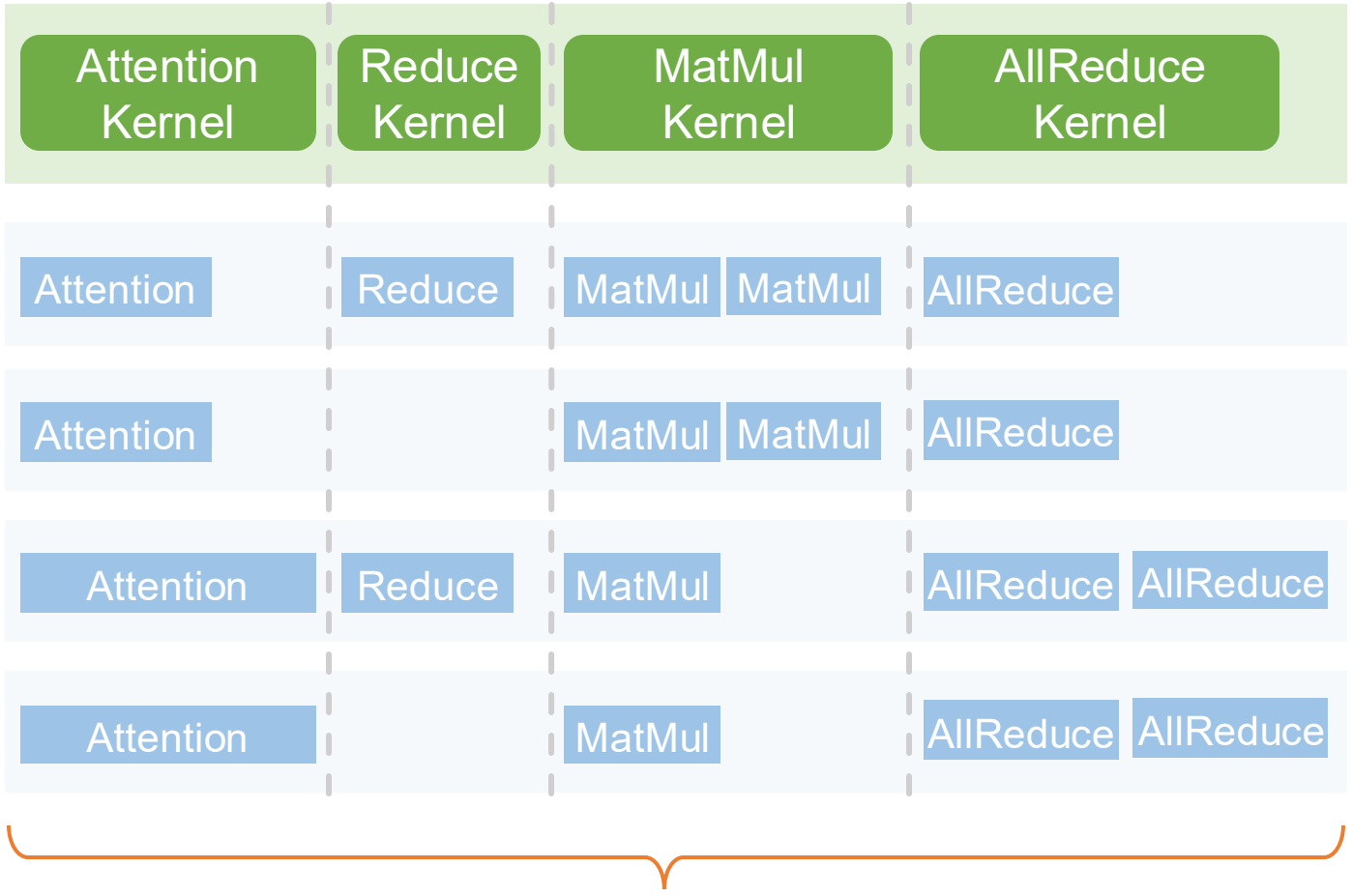



```python
@triton.jit
def matmul_kernel(
        a_ptr, b_ptr, c_ptr,
        M, N, K,
        stride_am, stride_ak,  #
        stride_bk, stride_bn,  #
        stride_cm, stride_cn,
        BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.con
        GROUP_SIZE_M: tl.constexpr,  #
        ACTIVATION: tl.constexpr  #
):
    """Kernel for computing the matmul C = A x B.
    A has shape (M, K), B has shape (K, N) and C has shape (M, N)
    """
    pid = tl.program_id(axis=0)
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
    num_pid_in_group = GROUP_SIZE_M * num_pid_n
    group_id = pid // num_pid_in_group
    first_pid_m = group_id * GROUP_SIZE_M
    group_size_m = min(num_pid_m - first_pid_m, GROUP_SIZE_M)
    pid_m = first_pid_m + ((pid % num_pid_in_group) % group_size_m)
    pid_n = (pid % num_pid_in_group) // group_size_m

    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)) % M
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)) % N
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)

    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
        accumulator = tl.dot(a, b, accumulator)
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk
    if ACTIVATION == "leaky_relu":
        accumulator = leaky_relu(accumulator)
    c = accumulator.to(tl.float16)

    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
    tl.store(c_ptrs, c, mask=c_mask)
```

Existing Kernel-Per-Layer Approach



Limitations of Kernels



Each LLM forward pass launches 100s-1000s kernels

No Inter-Layer Pipelining
Kernel barriers prevent software pipelining across kernels

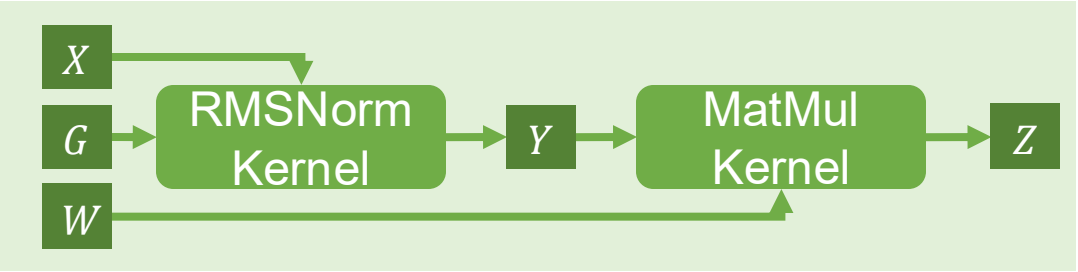
No Overlapping
Coarse-grained dependency prevents comp. & comm. overlap

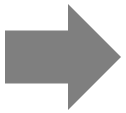
Limited Dynamism
Rely on CUDA graphs to reduce kernel launch overhead

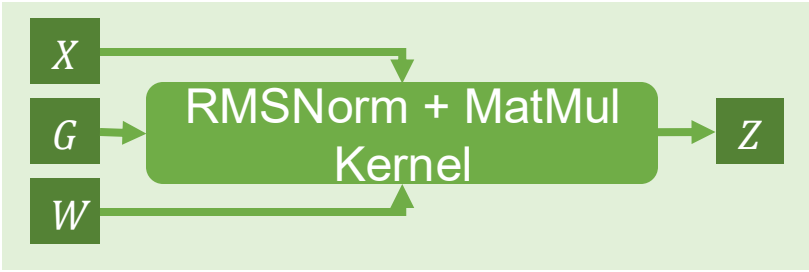
Kernel Fusion

Fuse multiple kernels into one

- Reduce kernel launch overhead
- Pipelining across layers
- Reduce device memory access



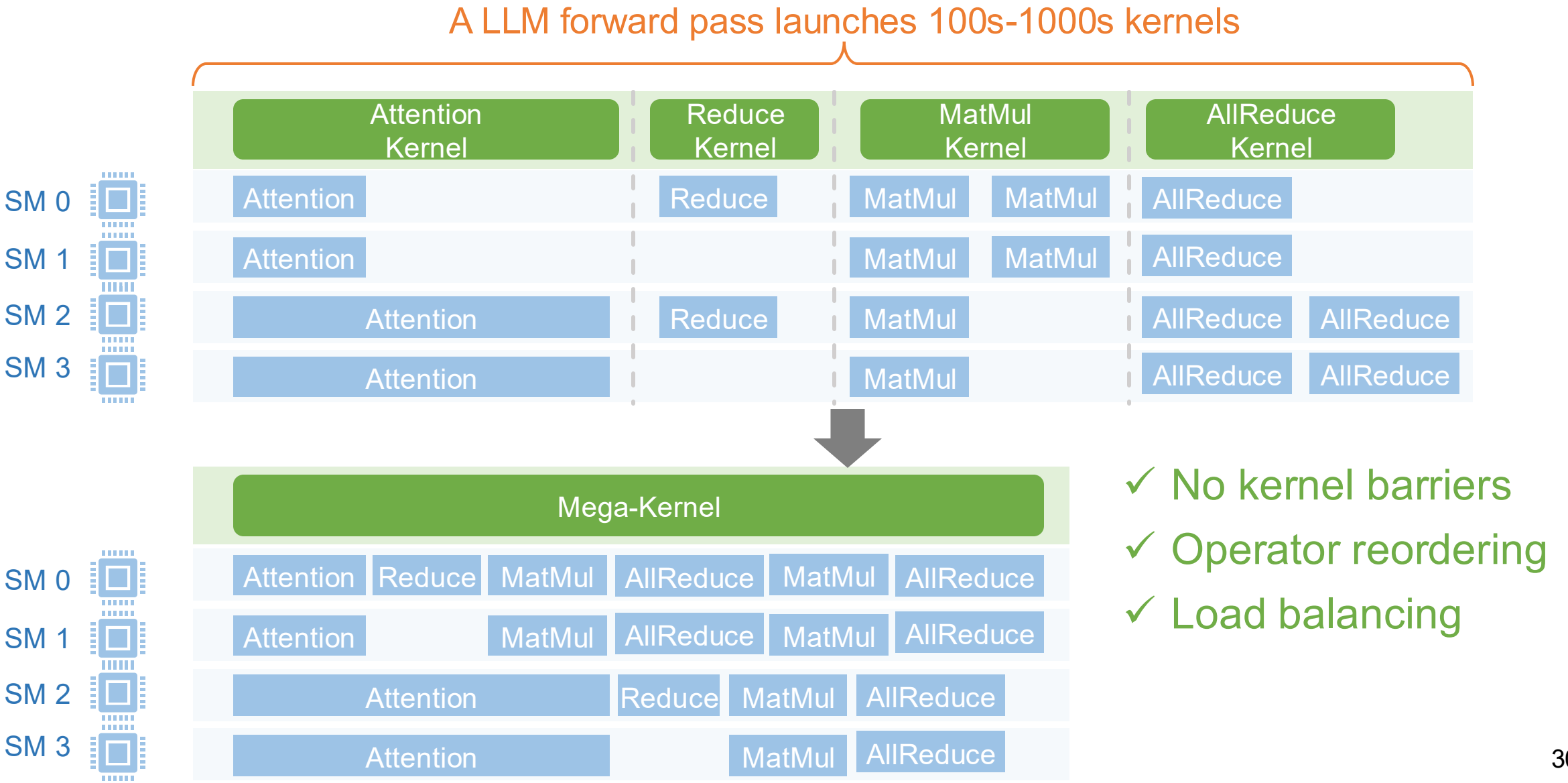




$$y_i = \frac{x_i g_i}{\sqrt{\frac{1}{N}\sum_j x_j^2}} \qquad z_i = \sum_k w_{ik} y_k$$

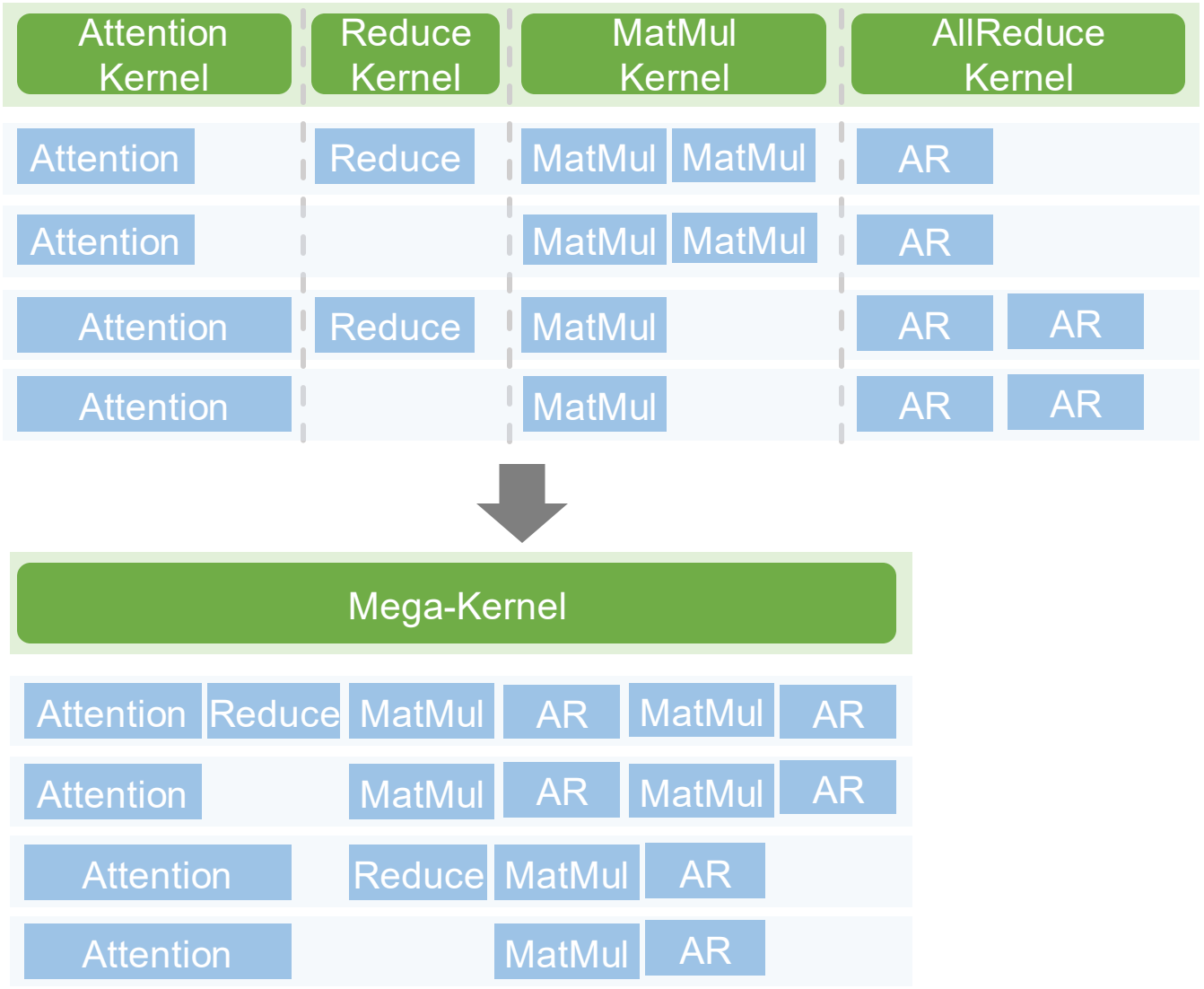
$$z_i = \frac{\sum_k w_{ik}\, x_k g_k}{\sqrt{\frac{1}{N}\sum_j x_j^2}}$$

Can We Fuse Everything in a Single Kernel?



- ✓ No kernel barriers
- ✓ Operator reordering
- ✓ Load balancing

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? → Task Graph

   No kernel barriers in mega-kernel

2. How to handle dynamism? → In-Kernel Parallel Runtime

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

3. How to optimize performance? → Mirage Superoptimizer*

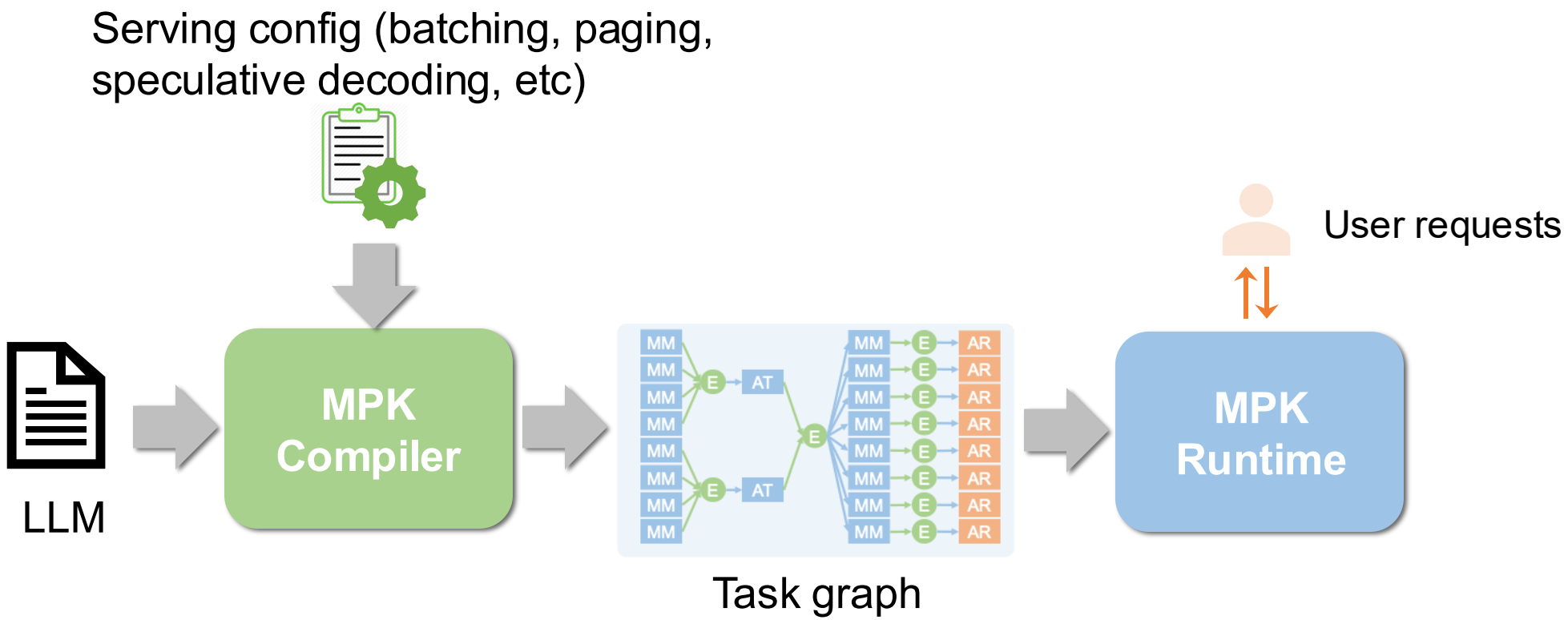
   Existing compilers target individual kernels

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

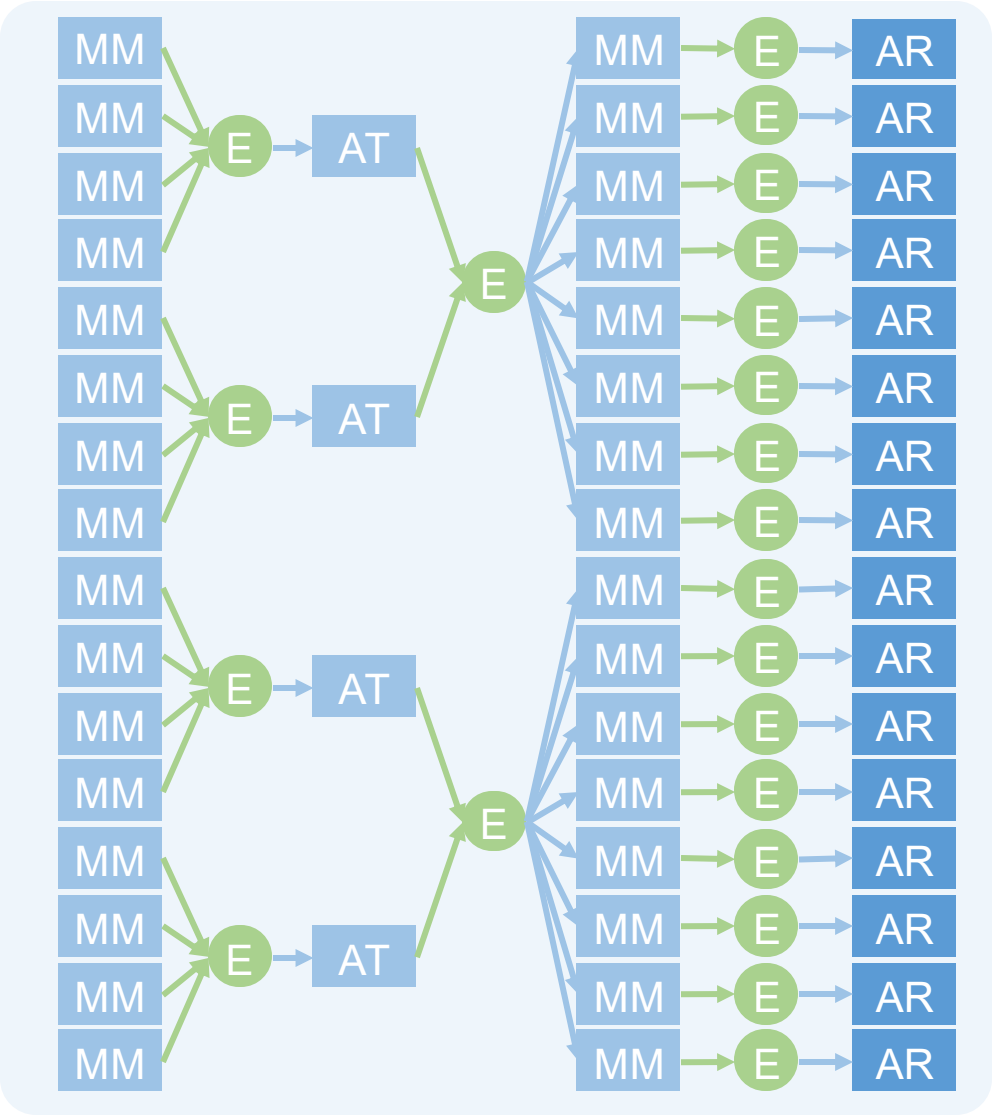
Mirage Persistent Kernel: Compiling LLM Serving into a Mega-Kernel

- **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

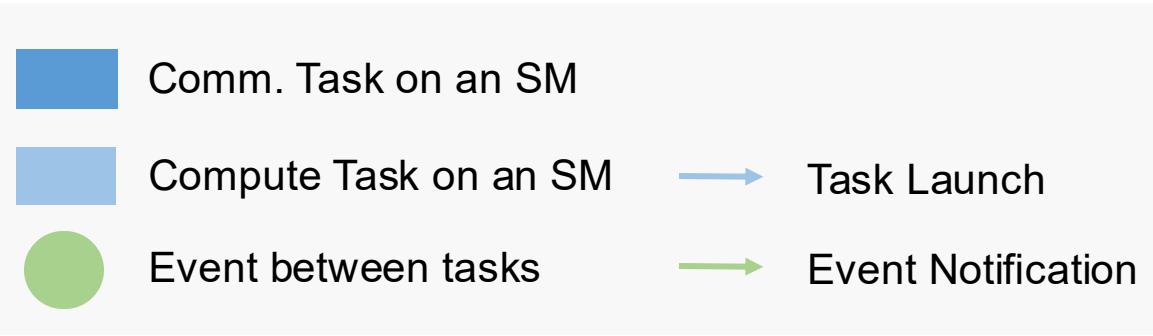


Task Graph



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

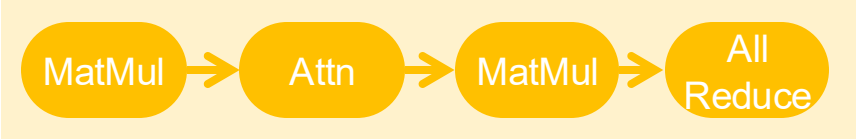


Task Graph vs. CUDA Graph

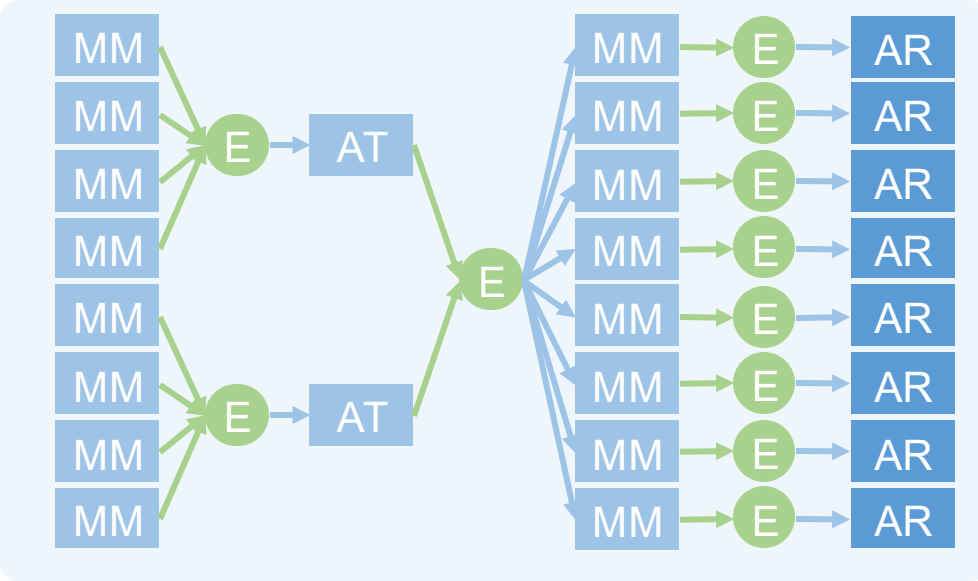
Task graph is a “lower-level” CUDA graph

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

CUDA Graph: nodes are kernels on GPUs

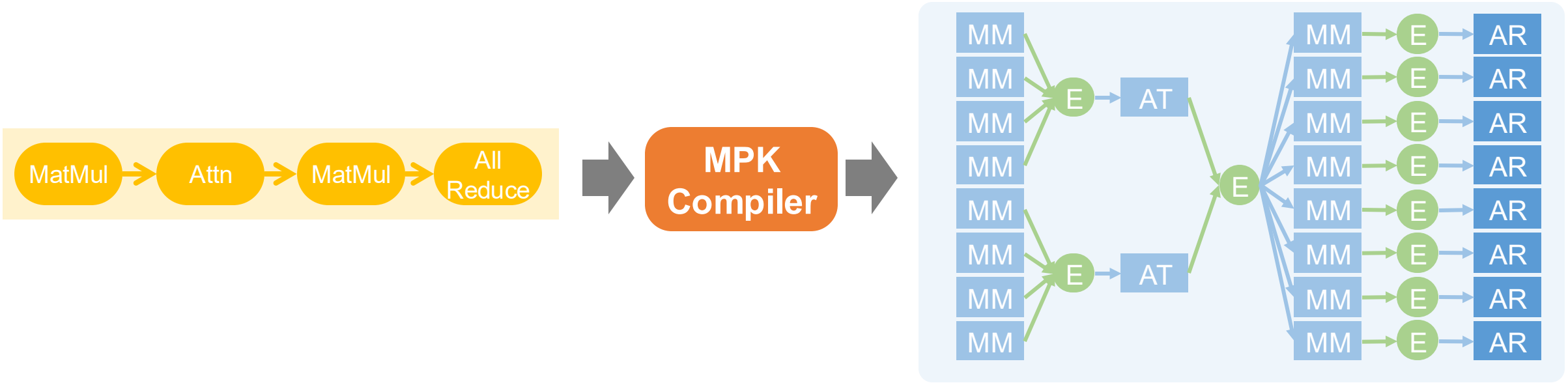


Task Graph: nodes are tasks on SMs

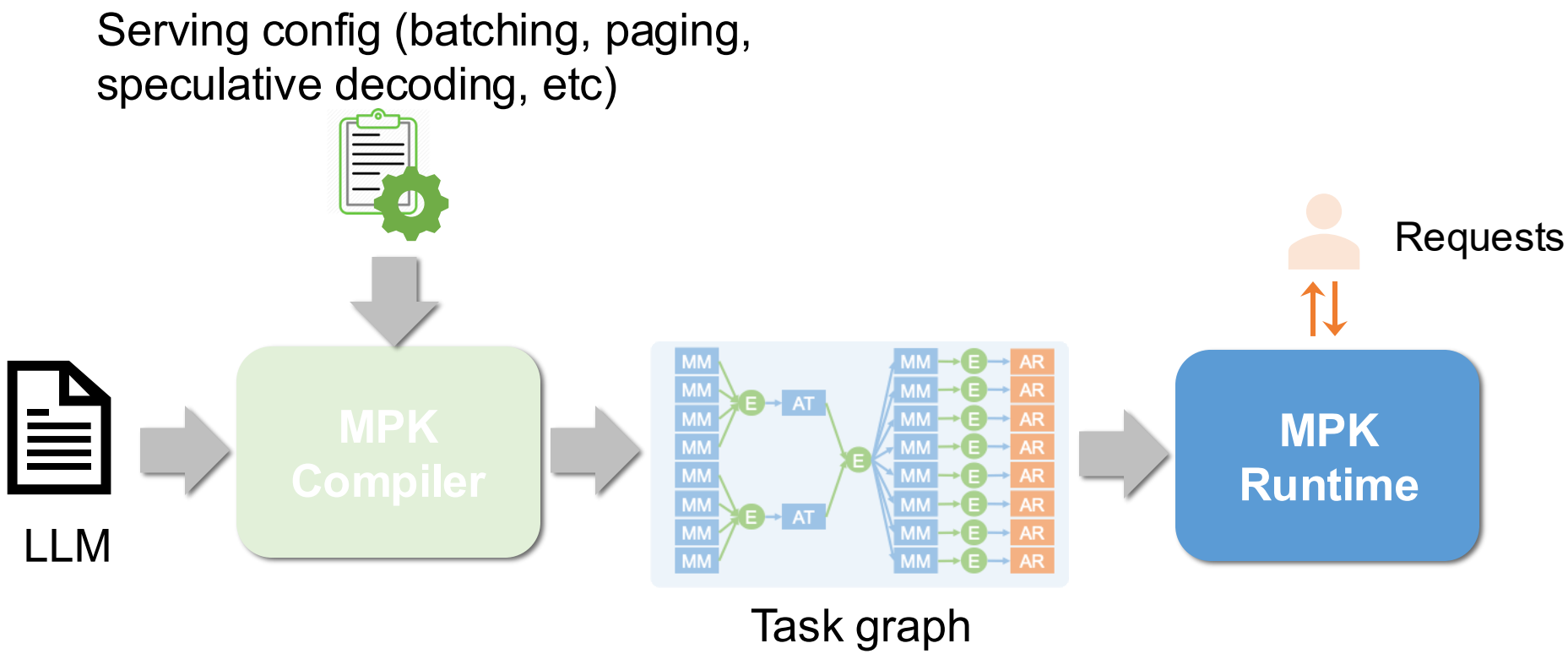


The MPK Compiler

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



MPK Overview

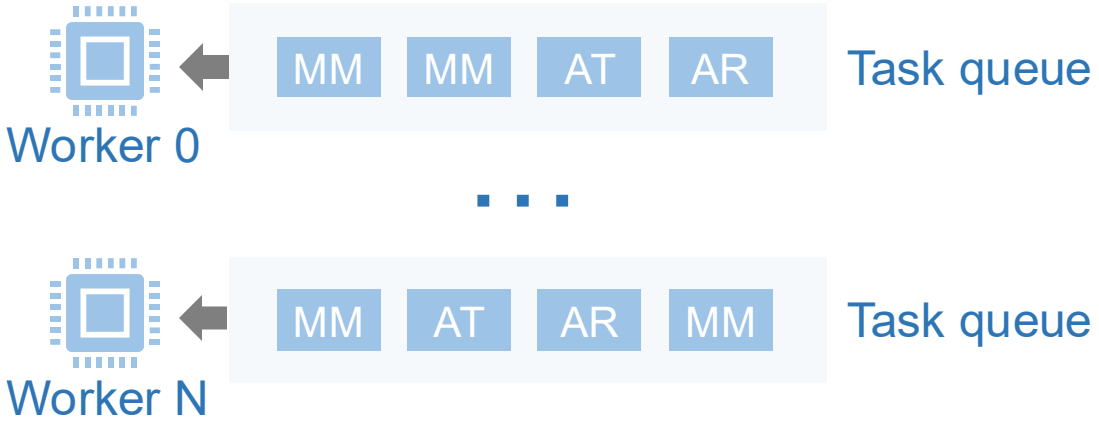


The MPK Runtime

Each scheduler runs on one warp

Each worker runs on one SM

Task-Based Parallel Runtime



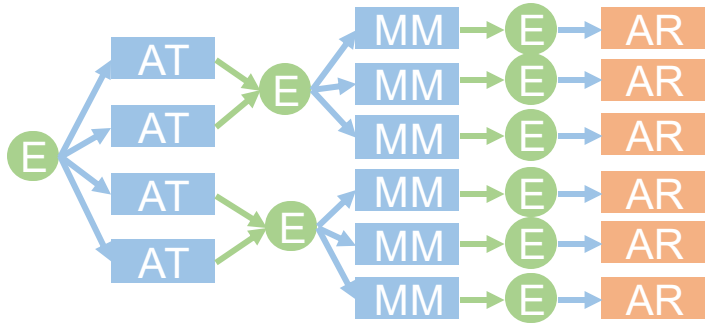
Repeatedly

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

Repeatedly

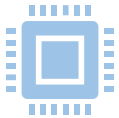
1. Dequeue fully triggered event
2. Launch all tasks depending on the event

Event-Driven Execution

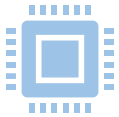


Scheduler
on a warp

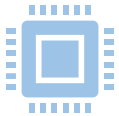
Scheduler
on a warp



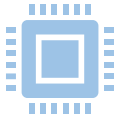
Worker on
an SM



Worker on
an SM

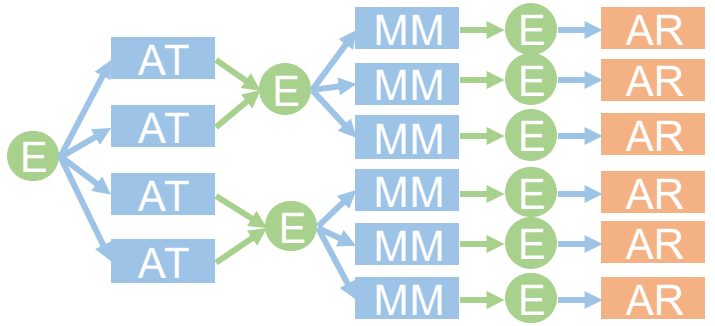


Worker on
an SM



Worker on
an SM

Event-Driven Execution



Scheduler on a warp

Scheduler on a warp

E

Worker on an SM

Attention

Worker on an SM

Attention

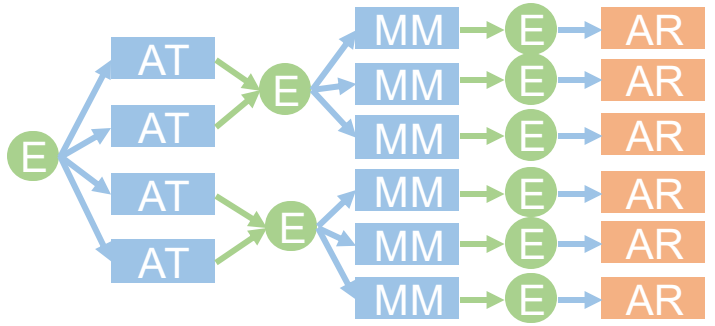
Worker on an SM

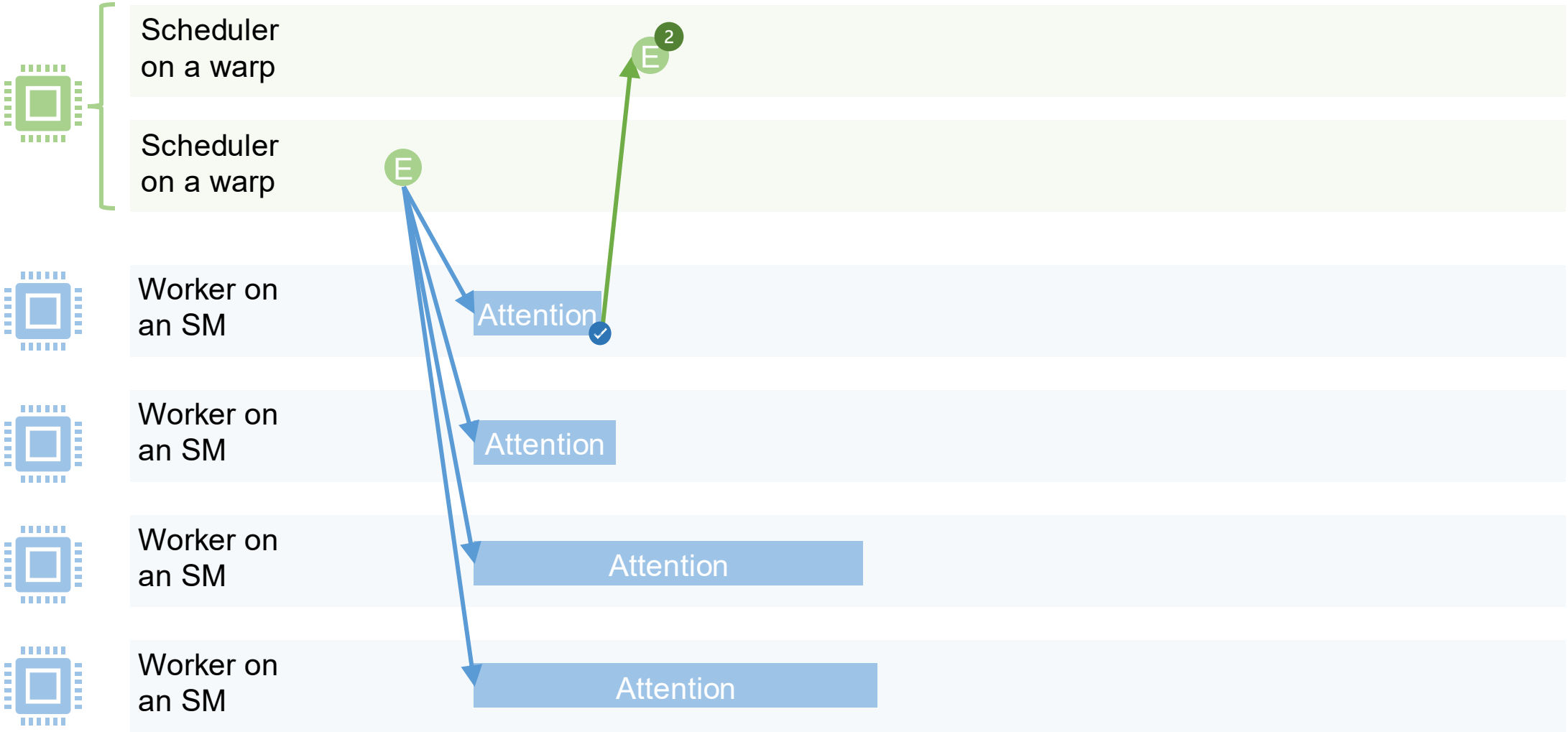
Attention

Worker on an SM

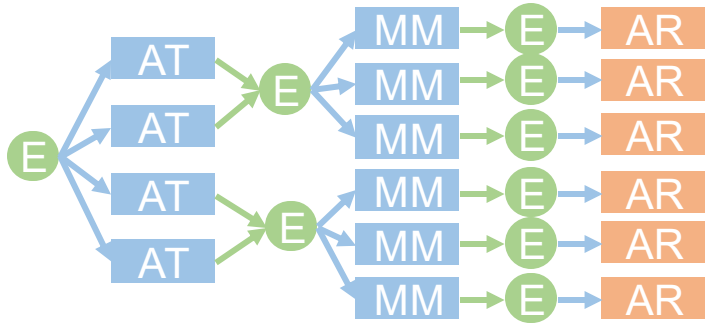
Attention

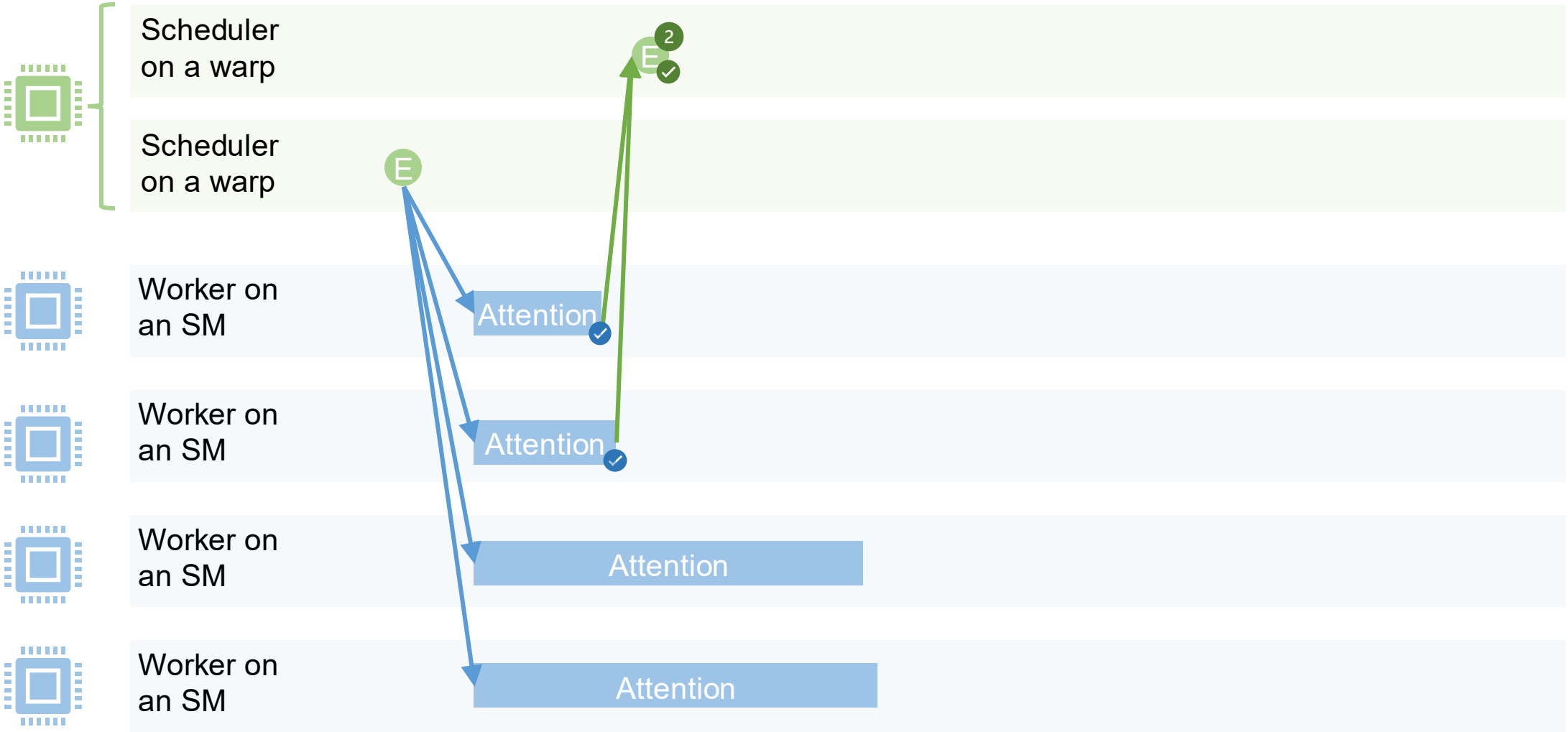
Event-Driven Execution



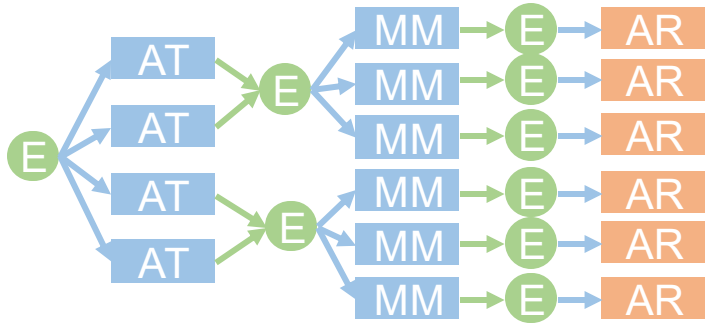


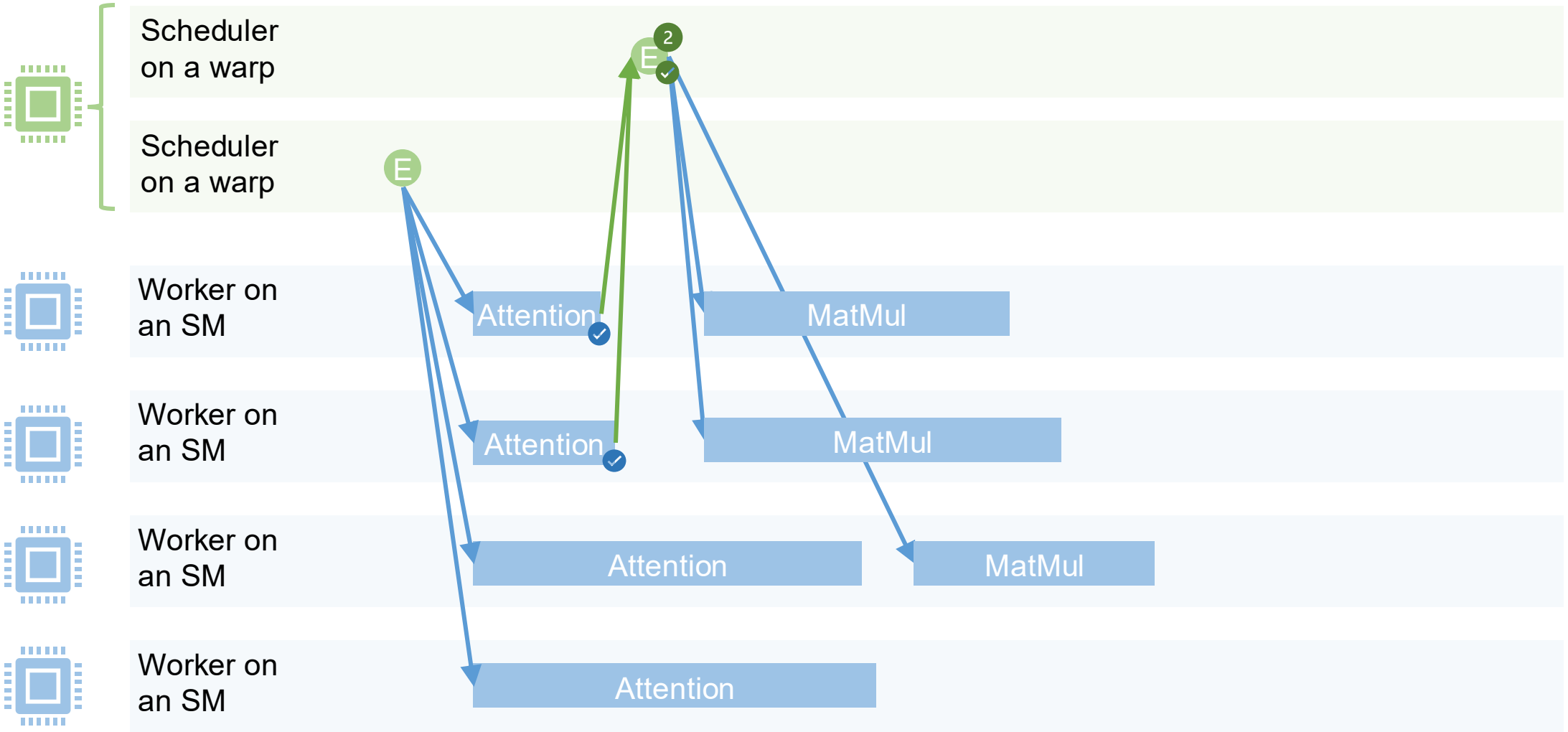
Event-Driven Execution



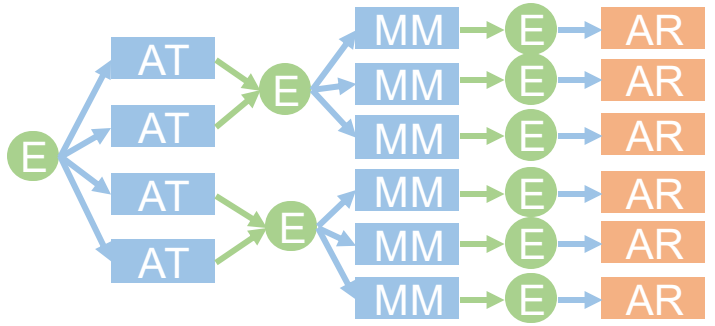


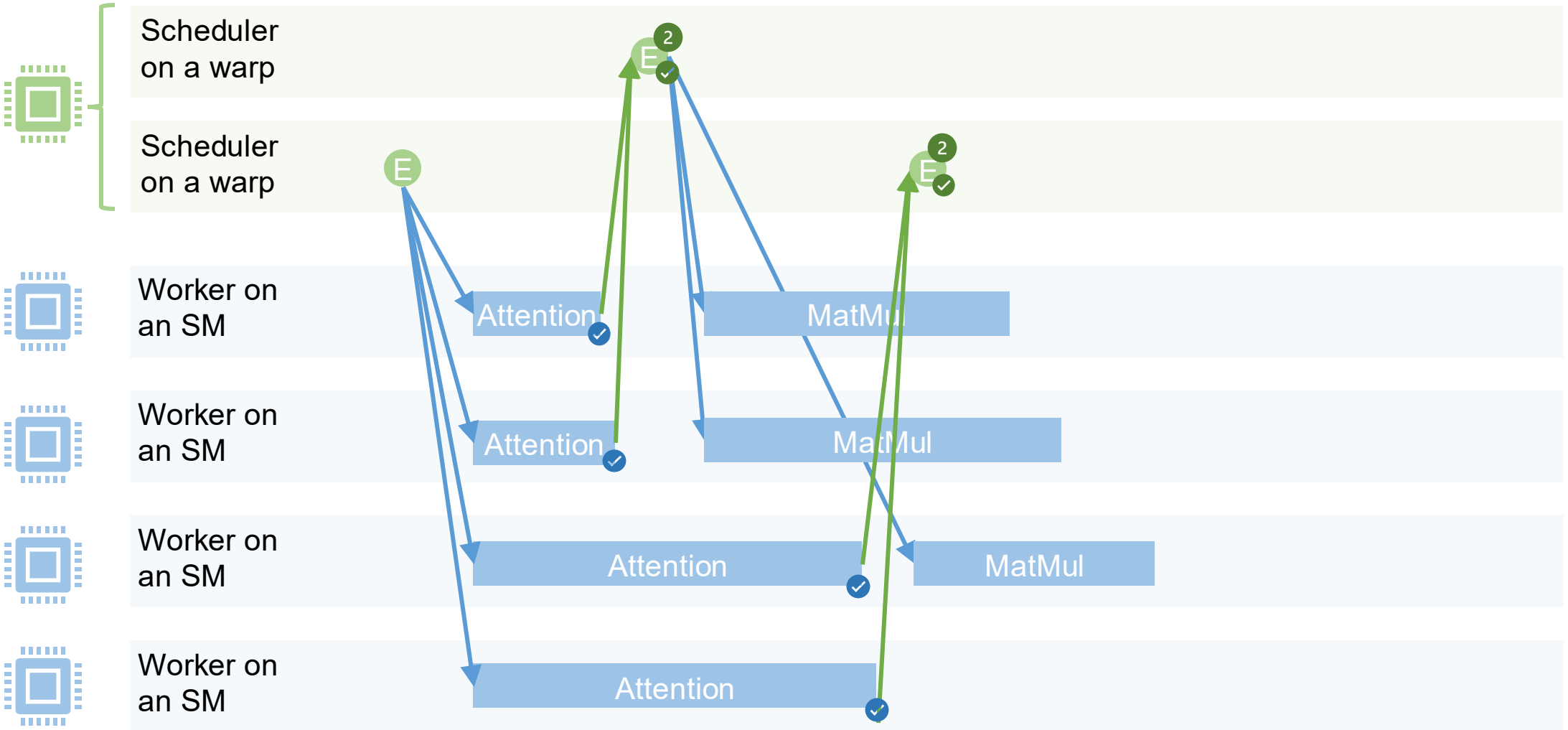
Event-Driven Execution



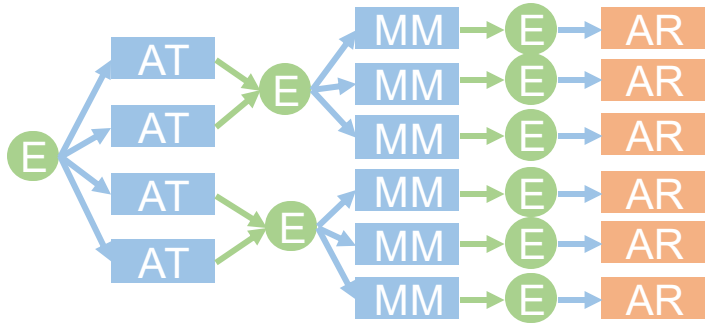


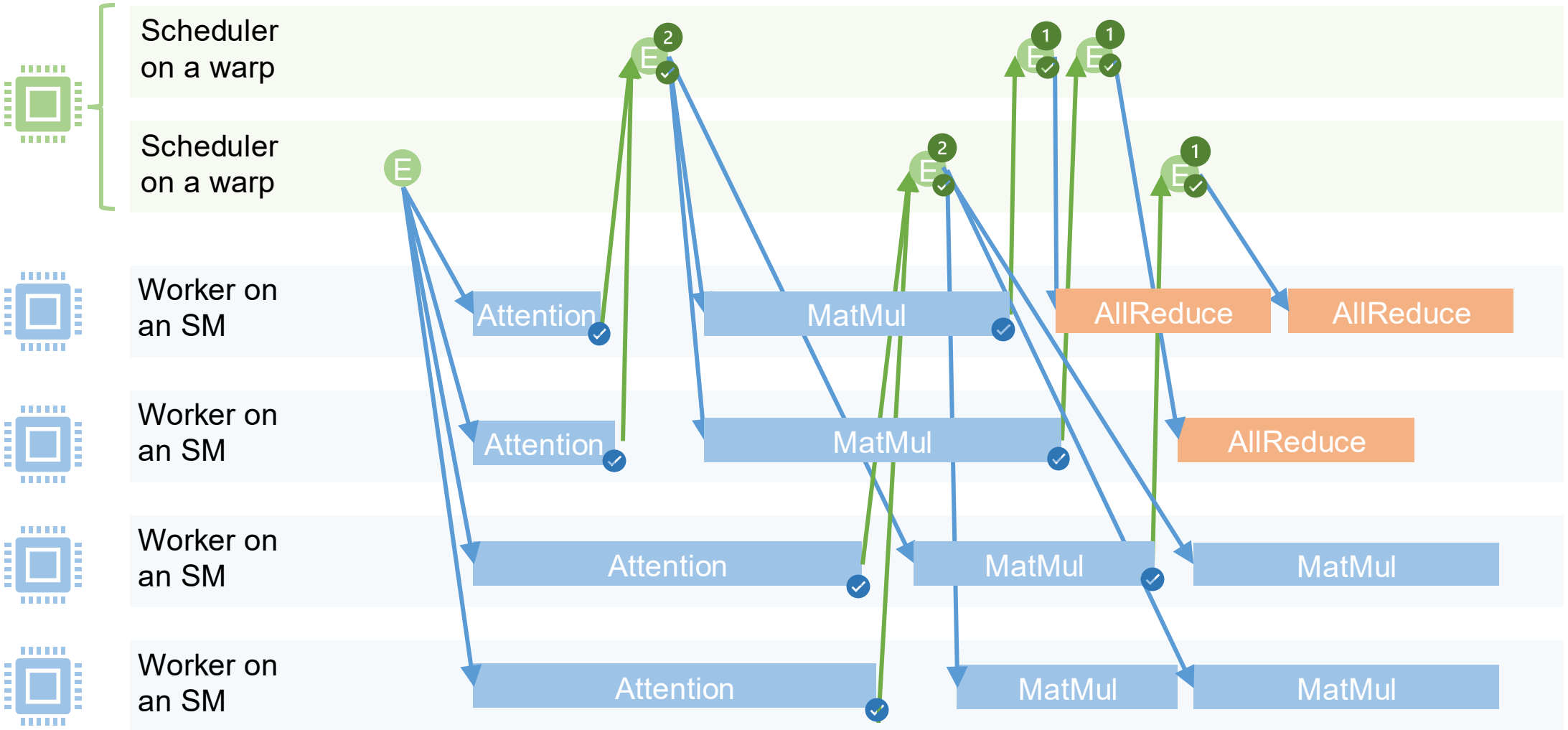
Event-Driven Execution





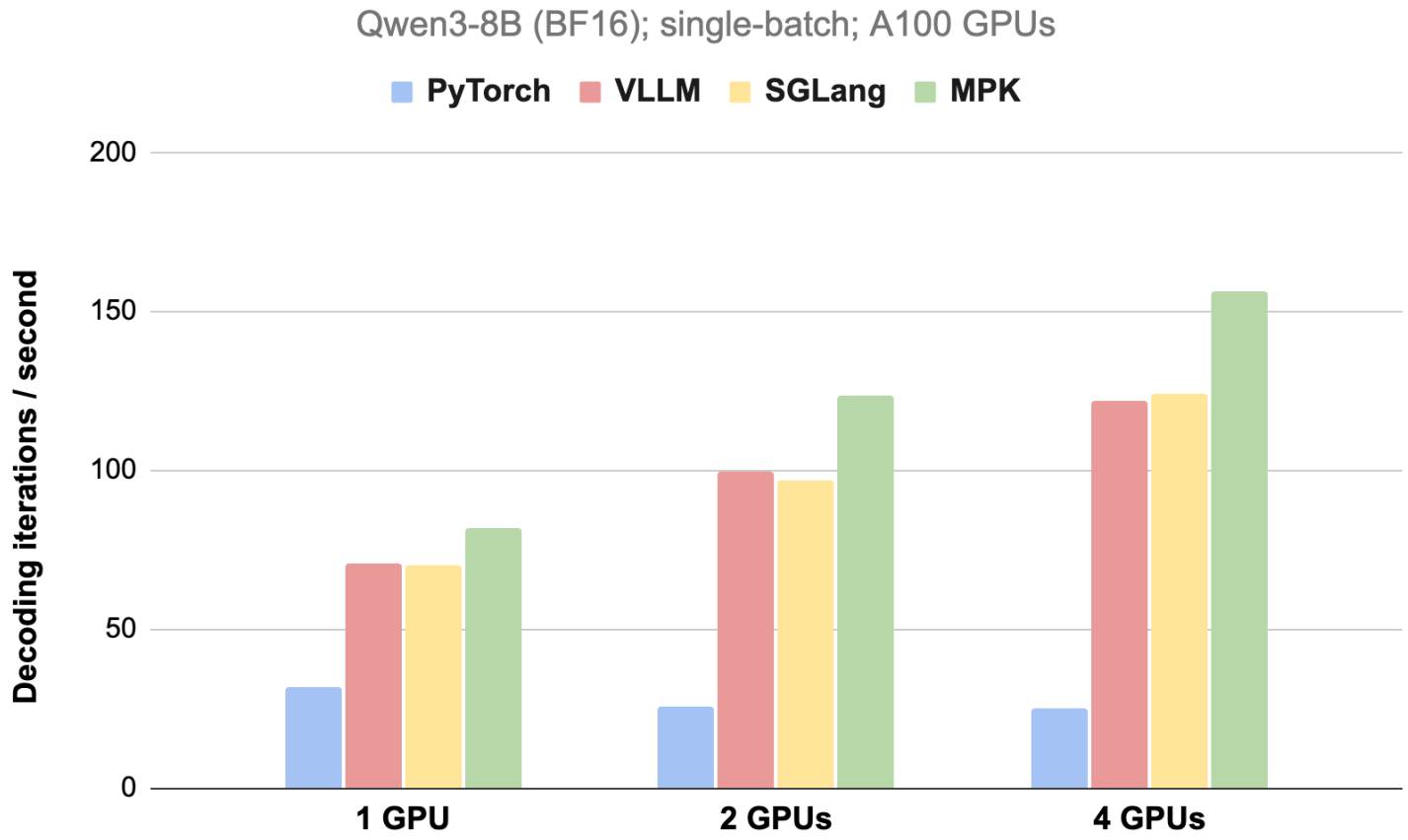
Event-Driven Execution



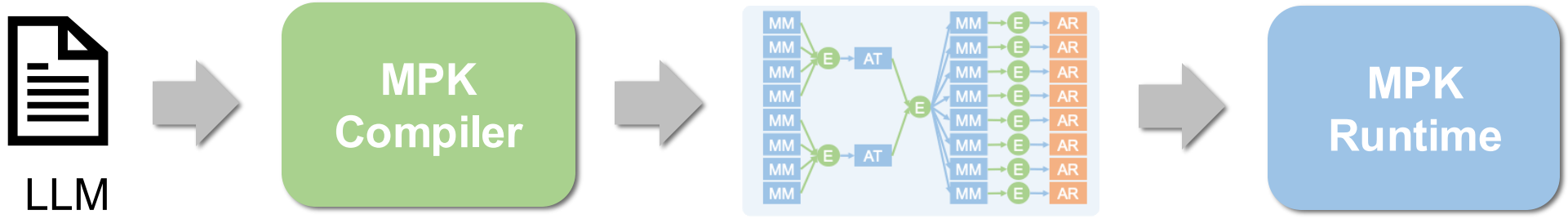


Pushing LLM Inference Latency Towards HW Limits

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



MPK: Compiling LLMs into a Mega-Kernel



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

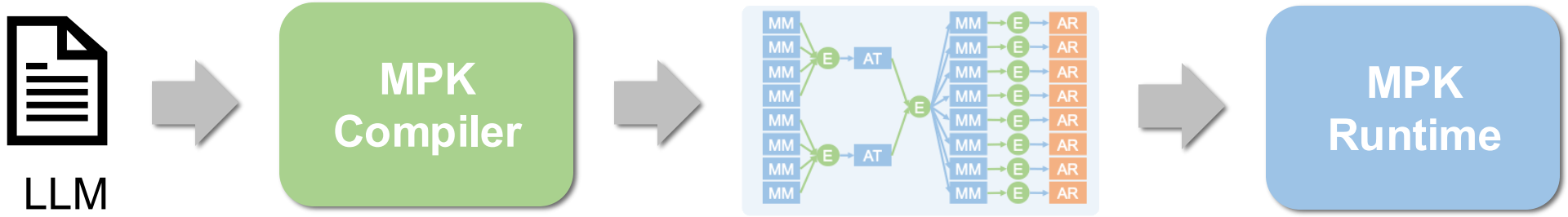
Unwatch 22 | Fork 108 | Starred 1.7k

Hacker News new | past | comments | ask | show | jobs | submit

1. Compiling LLMs into a MegaKernel: A path to low-latency inference (zhihaojia.medium.com)
   134 points by matt_d 4 hours ago | hide | 30 comments
2. Literate programming tool for any language (github.com/zyedidia)
   24 points by LorenDB 1 hour ago | hide | 8 comments
3. Show HN: I wrote a new BitTorrent tracker in Elixir (github.com/dahrkael)
   12 points by dahrkael 1 hour ago | hide | discuss

Backup Slides

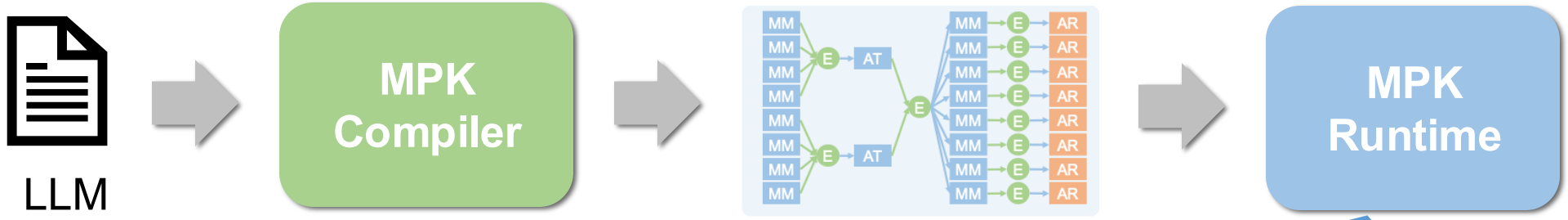
Open Questions



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