

Taming Dynamism on GPUs: Cross-SM Kernel Fusion via SM Cooperation and Just-in-Time Reduction

Jingkai He, Guangda Sun, Tianjian Li, Dong Du, Yubin Xia, Haibo Chen

Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

Institute of Artificial Intelligence Operating Systems, Shanghai Jiao Tong University

Abstract

Modern LLM architectures and systems render GPU kernel input shapes increasingly dynamic, e.g., conditional expert routing in MoE and batches with variable-length sequences. This dynamism degrades performance in both expert-tuned kernels and compiler frameworks. We identify the root cause as *dynamic cross-SM data dependencies* across execution phases, which force distinct phases into separate kernels for correctness. The resulting kernel boundaries amplify intra-phase workload imbalance across SMs, and compel intermediate states to expensive round trips through off-chip memory.

We propose *cross-SM kernel fusion*, which pushes kernel fusion boundary from traditional intra-SM scope to the entire GPU. Instead of enforcing dependencies via kernel boundaries, it utilizes *dynamic SM cooperation* and *just-in-time reduction*. We present MeldKernel (Meld for short), a compiler framework realizing cross-SM fusion for dynamic workloads. First, utilizing GPU on-chip interconnects, Meld enables fine-grained SM cooperation, which decomposes coarse-grained, phase-level dependencies into sub-task-level ones, transforming sequential kernels into a fluid, cross-SM pipeline. This further enables just-in-time reduction, facilitating waiting-free and localized state reduction via asynchronous communication. Second, to handle workload dynamism, we propose a decoupled execution model. Developers declaratively define computation logic, while the hardware mapping and inter-SM cooperation are deferred to runtime and adapted to input shapes. Meld accelerates critical workloads like decoding attention by 1.3× on average over the best solutions. Meld shifts GPU execution from a BSP-centric model toward fine-grained, dataflow-driven execution.

CCS Concepts: • Computing methodologies → Parallel computing methodologies; • Computer systems organization → Parallel architectures; • Software and its engineering → Compilers.

Keywords: GPU Kernel, GPU Architecture, Kernel Fusion



This work is licensed under a Creative Commons Attribution 4.0 International License.

SOSP '26, Prague, Czech Republic

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2585-2/2026/09

<https://doi.org/10.1145/3830418.3843857>

ACM Reference Format:

Jingkai He, Guangda Sun, Tianjian Li, Dong Du, Yubin Xia, Haibo Chen. 2026. Taming Dynamism on GPUs: Cross-SM Kernel Fusion via SM Cooperation and Just-in-Time Reduction. In *ACM SIGOPS 32nd Symposium on Operating Systems Principles (SOSP '26)*, September 29–October 02, 2026, Prague, Czech Republic. ACM, New York, NY, USA, 21 pages. <https://doi.org/10.1145/3830418.3843857>

1 Introduction

The explosive growth of GPU workloads, particularly LLMs [1–5], imposes immense pressure on the efficiency of GPU kernels. Consequently, substantial efforts have been devoted to hand-crafted kernels [6–15] (e.g., FlashAttention (FA) [16–20]) and compiler frameworks [21–39] (e.g., Triton [40]).

Traditionally, with the premise of uniform and relatively static inputs (e.g., pre-training), efforts have focused on intra-Streaming Multiprocessor (SM) (i.e., intra-thread-block) optimizations, e.g., pipelining [18, 22, 26], parallelization [41, 42], and fusion [17, 23–25], while treating SMs as independent parallel instances. For example, early FlashAttention [16] dedicates a thread block to each head of each sequence. However, today’s shift in models and high-level systems renders *kernel inputs highly dynamic* (and irregular). For instance, continuous batching [43] forces attention kernels to process sequences of wildly varying lengths and batch sizes, with reasoning models further exacerbating intra-batch length skewness; and Mixture-of-Experts (MoE) [44] introduces skewed, dynamic expert routing. Under rigid mapping, dynamic workloads translate into a massive long tail, where a few SMs toil on heavy tasks (Fig. 1 (a)).

To mitigate the challenges of dynamic inputs, the latest efforts unlock inter-SM parallelism by partitioning the tasks¹ into *multiple execution phases*, with each phase comprising multiple sub-tasks executed in parallel across SMs (Fig. 1 (b)). For example, FlashDecoding [6] divides each sequence into multiple sub-sequences to compute local attention results (“*split-KV*” phase), then merges them into final output (“*combine*” phase). This multi-phase execution enhances parallelism and mitigates the long tail. It is widely adopted across programs with dynamic inputs (Table.1), e.g., NSA [45], MoE forward, etc. Some expert-tuned implementations [7–9] further introduce specialized schedulers to balance “Split-KV”

¹In this paper, a task is a unit within a batch, e.g., a sequence for attention; a sub-task is a partition of a task, e.g., attention of a sub-sequence.

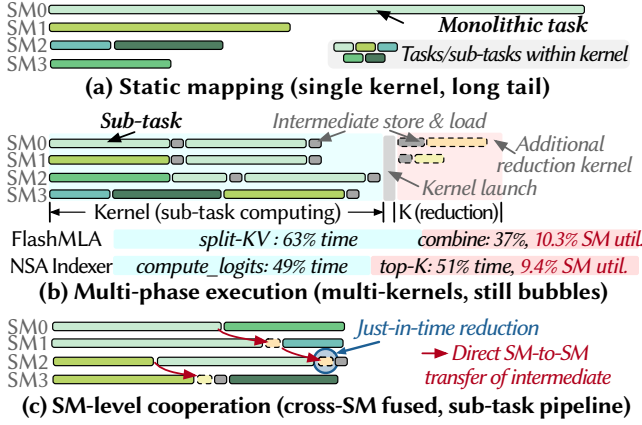


Figure 1. Execution paradigms for dynamic workloads.

workload across SMs. However, such heavily optimized operators still exhibit striking SM underutilization, e.g., in real-world workloads, FlashMLA yields only 67% SM occupancy.²

The inefficiency is rooted in the **dynamic, cross-SM data dependencies** across phases, e.g., a combine subtask relies on the results of various split-KV subtasks. Traditionally, the only mechanism to enforce these dependencies is kernel-level synchronization: by executing phases as distinct kernels, kernel boundaries ensure all thread blocks of a preceding phase complete before the next begins (BSP-style [47]). Besides the kernel launch and termination costs, this also leads to three issues. (1) *Intra-phase imbalance*. Current coupled programming of computation logic and hardware mapping restricts general frameworks (e.g., Triton [48, 49]) and most hand-written kernels [6, 50] to *pre-defined, static* parallelization schemes (e.g., fixed split-K), causing load imbalance across SMs. (2) *Low-utilization reduction phase*. Hampered by low parallelism and the long tail, it emerges as a non-negligible bottleneck (Fig. 1 (b)). (3) *Memory traffic*. Kernel boundaries force massive intermediate results to be repeatedly stored to and reloaded from the slow off-chip memory. While existing kernel fusion fuses operators within intra-SM scope, it cannot address dependencies that span across SMs.

We advocate **cross-SM kernel fusion via dynamic SM-level cooperation**. By fusing distinct kernels, it eliminates workload imbalance and mitigates off-chip memory traffic.

There are two insights. First, *disentangling dependency via SM cooperation*. Modern GPUs introduce SM-to-SM interconnects, enabling direct shared memory access between SMs. Their sufficient bandwidth and fine-grained synchronization unlock fine-grained SM cooperation, which turns isolated SMs into collaborators that flexibly exchange states. With SM cooperation, we can decompose the coarse-grained, kernel-level dependencies into fine-grained, sub-task-level ones, transforming traditional sequential kernels into a fluid, sub-task-level pipeline. Furthermore, we propose *just-in-time reduction*. Leveraging hardware-offloaded asynchronous trans-

mission, a remote SM’s intermediate states arrive at the local shared memory before they are needed for reduction (i.e., just-in-time). This enables waiting-free reduction to be performed directly within local SRAM, as Fig. 1 (c) shows.

Second, *dynamic coordination via decoupled execution*. Current programming models fail to adapt to the dynamic workloads due to the coupling of computation logic and hardware mapping. With decoupled programming and execution, developers only define the logic declaratively, whereas the actual hardware mapping and cooperation are deferred to runtime and adapted to the dynamic input shapes.

Guided by these insights, we present MeldKernel (Meld for short), a compiler for dynamic workloads. We address the following challenges.

Challenge-1. Programming cooperation: coupled abstraction buries cooperation semantics. Existing programming abstractions treat thread blocks as self-contained units. Their expressiveness is confined to intra-block computation, data movement, and work partitioning, which buries cross-SM parallelization logic and data dependencies beneath fragmented, low-level details (e.g., hardcoded global memory accesses). Consequently, the system is deprived of the semantic visibility to identify SM cooperation opportunities.

Challenge-2. Coordinating cooperation: non-uniform interconnect impedes coordination. Our characterization reveals that GPUs’ interconnect topology is highly *heterogeneous* and *localized*. Therefore, existing hardware-oblivious primitives (e.g., CUDA cluster³) result in severe SM fragmentation and idling; and the SM cooperation has to rely on hybrid communication paths (HBM-proxied and on-chip links). Furthermore, it requires meticulous scheduling to reduce off-chip memory traffic while maintaining global balance.

Challenge-3. Executing cooperation: decoupled execution challenges efficiency. Decoupled execution shatters monolithic kernels into fine-grained, dynamically scheduled units, which clashes with traditional GPU execution models. The GPU’s opaque scheduler hinders precise mapping of execution plans onto the on-chip topology. Besides, fine-grained units fragment continuous intra-SM hardware pipelines, introducing stalls and bubbles. And dynamic scheduling incurs GPU idling, negating the benefits of dynamic coordination.

To address the challenges, we propose the following designs. First, we propose the *operator-based declarative abstractions*. Unlike prior abstractions centered on intra-thread-block computation/pipeline [21–24] (e.g., Triton), Meld elevates *inter-SM cooperation* to first-class abstractions. Meld declaratively expresses DNN computation logic and inter-SM cooperation semantics by defining and composing a set of operators (e.g., Map, Iter, Reduce). Furthermore, Meld establishes batch-level global views to enable holistic optimizations. It proposes *cooperative loading* to absorb redundant memory traffic across SMs (e.g., shared KV cache prefix).

²DeepSeek-V3.2, ShareGPT [46] trace, H100 GPU, batch size = 16 (Table. 4).

³A cluster is a group of localized SMs with SM-to-SM interconnects.

Second, we propose the *hybrid cluster primitives* and *hierarchical scheduling*. To tame the heterogeneous interconnect, instead of relying on a uniform cluster abstraction, Meld transparently mixes diverse cluster settings based on the specific topology. We also design the communication primitives leveraging hybrid communication paths, enabling whole-GPU SM cooperation. We propose hierarchical scheduling. It first dispatches tasks across clusters with a hybrid cut of batch- and task-level traffic graphs, minimizing off-chip memory traffic, and then, within each cluster, it utilizes *stage fusion* and *just-in-time reduction* to orchestrate cross-SM fused pipelines.

Third, we propose *Meld runtime*. Instead of generating static kernels, Meld compiles operators into fine-grained *handlers*, paired with a *wrapper* kernel, which invokes handlers and communication primitives based on schedule plans. For topology-aware mapping, Meld bypasses the GPU scheduler through software-managed per-SM instruction queues. Leveraging the shared warp specializations across handlers, it fuses intra-SM pipelines across sub-tasks. Meld removes scheduling overhead from the critical path with async scheduling and plan cache, and coordinates with tiling strategies.

We implement a wide range of DNN programs with Meld, including MLA, GQA, MoE Forward, NSA Indexer, RMSNorm, etc., and evaluate them across various GPU architectures. Under real-world workloads, Meld outperforms the best baseline by 1.3 $\times$ on average (up to 2.04 $\times$). These improvements result in end-to-end inference acceleration by up to 1.4 $\times$.

In short, **Meld brings the dataflow programming and execution model to SIMT GPUs**, shifting GPU execution from a BSP-centric model toward fine-grained, dataflow-driven execution. Meld is open-sourced at <https://github.com/he-jingkai/tilelang.dataflow>.

2 Revisiting GPU Execution for Dynamism

2.1 GPU Primer: Computing and Memory Hierarchy

As a SIMT processor, the GPU comprises numerous Streaming Multiprocessors (SMs), each containing a cluster of CUDA cores and Tensor cores. From a computing perspective, as Fig. 2 (a) shows, the GPU presents a hierarchical abstraction of *thread - warp - thread block - grid* [51]. A thread is a logical execution unit on a CUDA core. A warp comprises 32 consecutive threads executing in lockstep. A thread block consists of multiple warps scheduled onto the same SM. Multiple blocks form a grid for parallel computation across SMs.

From a memory perspective, GPU features a four-level structure, including *register files* (thread-private), *L1 cache* and *shared memory* (shared within a thread block), *L2 cache* (shared across SMs), and off-chip *global memory* (HBM or GDDR). The L1/L2 caches accelerate access transparently (hardware-managed), while the register file, shared memory, and global memory are explicitly managed. Starting from Hopper [52] architecture, Tensor Memory Accelerator

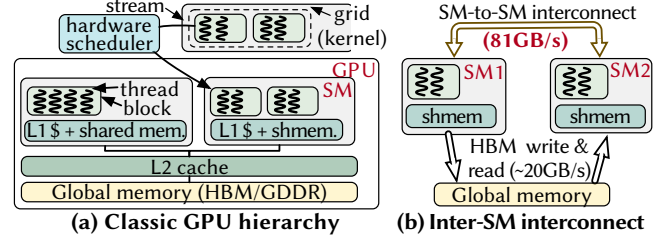


Figure 2. GPU hierarchy of memory, computation and link. The bandwidths are from H100 (SXM5).

(TMA) [53] facilitates high-bandwidth, asynchronous data transfer between global memory and shared memory.

2.2 Dynamic Tasks Challenge GPU Kernel Efficiency

Traditional hand-optimized kernels [6–14] and compilers [21–34] have extensively explored intra-block (i.e., intra-SM) optimizations. In these approaches, blocks (SMs) are treated as independent parallel instances for horizontal scaling, and tasks are linearly mapped to blocks. For example, FlashAttention [16–19] dedicates each block to a single attention head of one sequence. This paradigm performs well when workloads are uniform and static (e.g., pre-training).

However, today’s models and frameworks make a wide range of kernels’ input shapes highly dynamic (and irregular), as Table. 1 lists. For instance, MoE [44] dynamically activates experts, continuous batching [43] dynamically composes sequences of varying lengths, and schedulers [54–57] vary the batch sizes. The dynamic inputs and scaling GPU SM counts make traditional linear mapping suffer from long-tail effects (e.g., long sequences for attention, hot experts for MoE forward) and SM underutilization, as Fig. 3 (a) shows.

Therefore, the latest solutions extend parallelization strategies. They *partition the tasks* into multiple execution phases, with each phase comprising multiple sub-tasks executed in parallel. For example, FlashDecoding [6] splits each sequence into several sub-sequences to compute local attention in parallel (Fig. 3 (b), *split-KV phase*), then merges the intermediate results to produce final output (*combine phase*). MoE forward comprises three phases. Phase-1 performs GEMM-1 and activation on input hidden states. Phase-2 uses phase-1’s outputs for GEMM-2, while phase-3 (optional) performs a weighted sum of the expert outputs for each token. Sub-tasks in phase-1/2 are partitioned by token chunks.

Dynamic cross-SM data dependencies. Although the multi-phase strategies enhance parallelism and mitigate the long tail by splitting monolithic tasks into finer-grained sub-tasks, we find they still exhibit significant inefficiencies, e.g., FlashMLA yields only 67% SM occupancy⁴, and vLLM’s PagedAttention suffers from 39% SM idleness (Fig. 4). The root cause is the *dynamic, cross-SM data dependencies* across phases, e.g., in attention, a thread block in the “combine” phase depends on the results of multiple blocks in the “split-KV” phase.

⁴These metrics use ShareGPT [46] trace, batch size = 16, H100 GPU (Table.4).

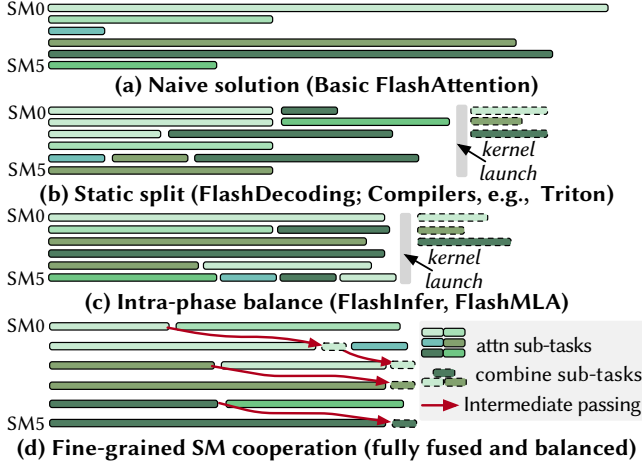


Figure 3. Execution paradigms for decoding attention.

Traditionally, the only mechanism to enforce the dynamic cross-SM ordering is kernel-level synchronization: by executing phases as sequential kernels, kernel boundaries ensure all thread blocks of a preceding phase complete before the next begins. Specifically, besides kernel launch/termination overhead, this paradigm suffers from the following issues.

I-1: Intra-phase imbalance. This stems from the *static parallelization strategies* and *uniform kernel implementations*. First, despite the dynamic inputs, programs usually can only pre-define sub-task partitioning schemes statically. For example, FlashDecoding and Triton kernels in vLLM/SGLang [48, 49] rely on *static split-K*, splitting sequences into a fixed number of chunks; vLLM’s PagedAttention [50] uses fixed 512-token chunks. Under dynamic workloads, this rigid partitioning usually results in severe workload imbalance across SMs. Due to the kernel barriers between phases, fast-finishing SMs must sit idle until the last block completes. GPU’s run-to-completion scheduling further amplifies this tail, as the final dispatch wave often occupies only a fraction of SMs.

Second, while some expert-optimized implementations, e.g., FlashMLA/FlashInfer, integrate customized schedulers (Fig. 3 (c)), their tile size, which determines the minimum unit of work, is statically fixed at implementation time, leading to padding and SM underutilization under varying sub-task shapes. FA-3 [18] (the basis of FlashInfer) suffers performance degradation for “short” sequences, e.g., 2× degradation at length 512 (vs. a compiler-tuned tile [21]).

I-2: Additional “combine” phase with low utilization. The split-based parallelization introduces “combine” phases (kernels) to merge intermediate results. As SM counts scale, their overhead becomes non-negligible. Under real-world workloads, vLLM GQA’s combine phase accounts for 25% of the time (Fig. 4). Similarly, the combine phases in FlashMLA and NSA [45] occupy 37% and 51% of the time. This phase suffers from low parallelism and a long tail. Its parallelism degree is bounded by batch size (much less than SM counts), and each sub-task’s time depends on the number of partitions

(which is also skewed). E.g., the combine phases in FlashMLA and NSA only have an SM occupancy of 10.3% and 9.4%.

I-3: Memory traffic overhead. Restricted by kernel barriers, intermediate results generated in each phase must be written to and subsequently read from global memory, which incurs significant traffic. The intermediate state in FlashMLA amounts to 32 KB (TP = 4) per sub-task; in MoE forward, intermediate result traffic accounts for 20% of the total traffic.⁵

This multi-kernel execution paradigm and resulting issues are prevalent across programs with dynamic inputs (Table. 1).

2.3 Dynamic Workloads Need Cross-SM Fusion

To address the issues, we propose **cross-SM kernel fusion**, which fuses distinct phases into a single kernel. Unlike prior methods constrained by kernel-level dependencies, it decomposes dependencies into sub-task-level ones, and balances loads across SMs through flexible sub-task partitioning.

Taking decoding attention as an example, cross-SM kernel fusion fuses distinct stages into a sub-task-level pipeline, as Fig. 3 (d) shows. Leveraging the variability in sequence lengths within a batch, it overlaps the stages across sequences to achieve global balance. Specifically, upon finishing a split-KV sub-task, the producer SM “persists” the intermediate result and proceeds to the next sub-task. The consumer SM, after finishing its split-KV, executes the “combine” sub-task, and may further move on to other subtasks. In this pipeline, long tasks are collaboratively processed by multiple SMs, and combine operations are distributed via cascaded/tree-based reduction, while short tasks fill the pipeline bubbles. Enabled by the flexible partitioning of sub-tasks, it can achieve fine-grained load balancing and bubble-free execution.

However, actualizing cross-SM fusion presents significant challenges for both hand-written kernels and compilers. First, from a *hardware* perspective, cross-SM fusion necessitates fine-grained SM synchronization, which is not natively supported by classic GPUs. Second, from a *programming* perspective, fine-grained pipelining and balancing for dynamic inputs need dynamic partitioning and scheduling. Yet, as §2.2 proves, all compilers and most hand-written kernels can only rely on pre-defined static (and coarse-grained) hardware mapping strategies. Third, from a *cost* perspective, fine-grained pipelining demands fine-grained partitioning, which significantly increases memory traffic for intermediate results, diminishing the benefits or even leading to degradation.

2.4 Hardware Opportunity of On-Chip Interconnect

Starting with Hopper GPUs, SM-to-SM links are introduced, enabling one SM to directly access another’s shared memory [63–65]. As Fig. 2 (b) shows, compared to HBM-proxied transfer, the links reduce latency by 7× and provide 4× bandwidth [66, 67]. CUDA introduces mbarrier-based hardware synchronization, where one SM can signal another’s barrier

⁵Qwen3-30B-A3B, FlashInfer CUTLASS FusedMoE [62], 64 tokens/expert.

GPU kernel	Scenario	Description	Dynamism	Execution phases
Attention (MHA, GQA, MLA [58])	Inference	Batched attention in inference (decoding)	Batch size and sequence lengths	Partial: attention of subsequences; Reduce: merge intermediate results (rescale, accumulate, normalize).
MoE forward	Infer. & train	Batched expert activation, local reduction to save bandwidth	Batch size and each expert's load	Partial-1/2: each expert's FFN computations (grouped execution of multiple tokens); Partial-1: GEMM-1 and activation; Partial-2: GEMM-2; Reduce: (optional) weighted accumulation of each token's expert outputs.
DeepSeek NSA [45] lightning indexer	Infer. & train	Choose top-K relative tokens from the whole sequence	Batch size and sequence lengths	Partial: score and sort within the subsequence; Reduce: sort the intermediate top-K results to get the final top-K.
Cross-entropy loss	Training (SFT/RL)	Calculate per-sequence loss using per-token loss	Batch size and sequence lengths	Partial: calculate the mean loss of each subsequence; Reduce: merging intermediate results with normalization.
Top-K and top-P sampling	Infer. & train	Sample top-K/top-P tokens from vocabulary	Batch size	Partial: sort and select top-K/top-P tokens from local subsequence; Reduce: merge intermediate results to get final top-K/top-P.
LayerNorm & RMSNorm [59]	Infer. & train	Normalize the input before each layer	Batch size and sequence lengths	Partial-1: subsequence processing; Reduce: accumulated inverse RMS calculation or Welford [60, 61] merge; Partial-2: normalizing subsequences using reduction results in parallel.
Cluster probing	Vector search	Each query probes several clusters and traverses the points	Batch size and each cluster's size	Partial: traverse the data points in each cluster; Reduce: sort and merge local top-K results to get the k-nearest neighbors.

Table 1. DNN programs with dynamic workloads, and their execution stages.

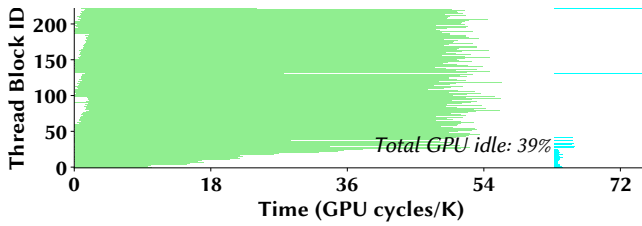


Figure 4. GPU utilization for decoding attention under real workloads (vLLM). Each green band shows a *split-KV attention* block, with each blue band indicating the *combine* of a sequence.

in shared memory, enabling *fine-grained cross-SM synchronization*. Furthermore, with TMA (§2.1), the interconnect-based data transfer and status signaling can be *made asynchronous and offloaded to hardware*. To expose this capability, the thread block cluster abstraction is introduced. Each kernel specifies a *single, uniform* cluster size⁶, and blocks within each cluster can utilize the interconnect.

However, the underlying interconnect topology remains undisclosed, precluding its effective utilization by software systems. Its practical adoption is hindered by two issues. First, *strict locality*. The maximum cluster size is limited to 16, restricting the interconnect to only a small group of adjacent SMs. Second, *SM underutilization and fragmentation*. For instance, on H100 GPUs, configuring a cluster size of 10–16 renders 62–20 SMs incapable of hosting thread blocks, leaving them idle. Full SM utilization is achievable only when cluster size is restricted to 2, leaving the interconnect fragmented. Consequently, it remains overlooked by mainstream hand-optimized kernels and all compilers.

3 Overview

3.1 Observations and Insights

Meld is motivated by two insights. First, *disentangling dependency via SM-level cooperation*. Modern GPUs’ on-chip interconnects provide sufficient bandwidth and fine-grained synchronization, transforming the GPU into a cooperative system with highly flexible state exchange across the SMs.

⁶Blackwell relaxes this with a fallback size, but constraints persist (§4.2).

This unlocks *fine-grained SM cooperation*: by decomposing coarse-grained, kernel-level dependencies into sub-task-level ones, it transforms traditional multi-kernel execution into flexible, sub-task-level pipelines. It also enables absorption of redundant memory traffic with cooperative loading.

This SM cooperation further unlocks *just-in-time reduction*. Leveraging hardware-offloaded asynchronous transmission and signaling (§2.4), the intermediate states can be asynchronously delivered to the target SM’s shared memory, ready before they are needed for reduction (i.e., just-in-time). Once the target SM finishes its sub-task, it performs the reduction directly within its local SRAM in a wait-free manner, thereby further enhancing pipeline efficiency.

Second, *dynamic coordination via decoupled execution*. Existing programming frameworks fail to adapt to dynamic workloads due to the coupling of *computation logic* and *hardware mapping*. This inherent coupling constrains general-purpose frameworks (e.g., Triton) to static parallelism (e.g., split-K), while restricting expert-optimized kernels to uniform tiling strategies. With decoupled programming and execution, developers declaratively define the computation logic, while the runtime dynamically determines parallelization and cooperation strategies according to input shapes.

Together, these two insights enable *cross-SM kernel fusion* for GPU programs with dynamic workloads. As Fig. 3 (d) shows, after finishing a split-KV sub-task, instead of materializing the intermediate results, the producer SM (asynchronously) forwards them to the consumer SM’s shared memory and proceeds to the next sub-task, while the consumer SM performs reduction directly. The hardware barriers (mbarrier) ensure pipeline correctness. With decoupled execution, the runtime dynamically adapts the parallelization and cooperation plan to input shapes. This dynamic cooperation further enables adaptive *copy absorption*. In many cases, multiple SMs load identical data, e.g., many sequences share a prefix in attention [68–71], and multiple SMs load the same hidden states (GEMM-1) and intermediate states (GEMM-2) in MoE forward. Via on-chip interconnects, we use one SM to load data and broadcast it to others.

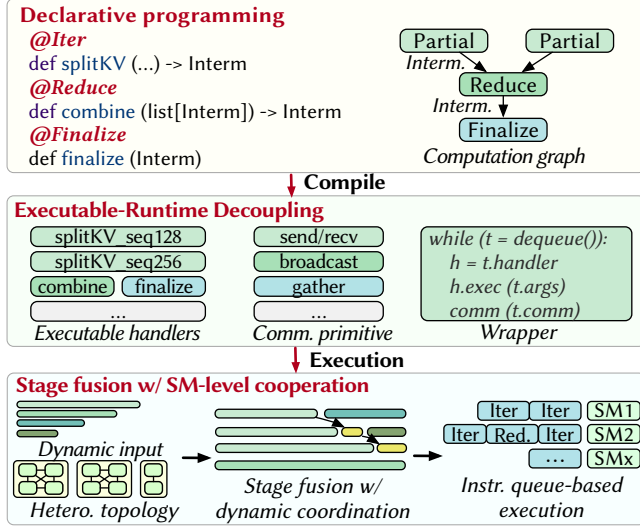


Figure 5. Overview of the Meld compiler system.

3.2 Design Overview

We design Meld, a compiler framework featuring dynamic SM-level cooperation. Fig.5 shows the overview of Meld.

Programming cooperation (§4.1). We propose operator-based declarative programming, which abstracts DNNs through the composition of operators. Developers only need to express computation logic by defining and composing the operators, while leaving the hardware mapping and adaptive coordination strategies dynamically adapted to input shapes at runtime. To expose semantics for SM-level cooperation, Meld elevates cross-SM cooperation to first-class abstractions, chaining the workflow through the transfer and transformation of intermediate states, as Fig.5 shows.

Scheduling cooperation (§4.2, §4.3). Leveraging the cooperation semantics exposed by operator-based programming, Meld dynamically coordinates SM cooperation. We reverse-engineer the on-chip interconnect topology of mainstream GPUs. To tame the heterogeneous and localized interconnects, we design communication primitives over hybrid paths (on-chip links and HBM-proxied), and the hybrid cluster dispatching mechanism to resolve SM fragmentation and idling. We propose hierarchical scheduling: inter-cluster scheduling uses hybrid cuts of batch-level and task-level traffic graphs to minimize off-chip traffic, while intra-cluster scheduling employs stage fusion and just-in-time reduction to arrange the fine-grained sub-task-level pipelines.

Executing cooperation (§4.4). To carry out scheduling plans, we propose Meld runtime, bridging static compilation and dynamic coordination. Meld compiles a program into two components: the *handlers* derived from operators, and a *wrapper* kernel that serves as the on-GPU executor. To bypass GPU’s black-box scheduler, Meld runtime employs per-SM instruction queues for topology-aware task mapping. Guided by the instructions, the wrapper block on each

SM invokes the handlers and communication primitives accordingly, forming fine-grained, whole-GPU SM cooperation. Meld runtime further fuses intra-SM pipelines across sub-tasks, proactively generates handlers tailored to different sub-task shapes for adaptive dispatch, and employs async scheduling with plan cache to maximize execution efficiency.

Putting these pieces together, a Meld program proceeds through three stages: compilation, scheduling, and execution. Compilation is performed only once. It lowers the program’s operators into handlers and generates a wrapper kernel. Each input then undergoes scheduling and execution. On the CPU, the scheduler partitions the input workload into sub-tasks and maps them onto the SMs. It writes the resulting per-SM execution plans into the per-SM instruction queues in GPU memory. Scheduling is typically performed asynchronously, and cached plans are reused whenever possible. Kernel execution takes place on the GPU, where the wrapper thread block on each SM drains its instruction queue and executes the sequence of sub-tasks by calling the corresponding handlers.

Meld builds its intra-SM programming and handler code generation on its intra-SM compiler backend (TileLang [21]).

4 Meld Design

4.1 Programming with Declarative Abstraction

Traditional compilers [21–24, 26–30, 39] treat computation as $Y = F(X)$, where the shape of X is known at compilation time or deduced through symbolic shape. This paradigm suffers from fundamental challenges for dynamic workloads.

Challenges. The traditional *coupled programming paradigm* of computation logic and hardware mapping constrains dynamic and flexible coordination (§2.2). Besides, the *intra-block-centric model* results in the SM cooperation semantics being obscured within fragmented, implementation-specific details (e.g., hardcoded global memory access). And the *single-task-centric model* hinders holistic optimization opportunities, e.g., absorbing redundant loading with interconnects.

Therefore, we propose the declarative programming paradigm to take *cross-SM cooperation as first-class abstractions*.

Observation. Despite the vast diversity in the computational semantics of DNN programs, we observe that those necessitating SM-level cooperation share a highly stylized workflow. We formalize it as the **partial-reduce execution model**, where the *partial* stage generates intermediate states in parallel, and the *reduce* stage aggregates these intermediate states. For instance, in decoding attention (§2.2), the Split-KV operations correspond to the partial phase, whereas the combine operations act as the reduce phase. Furthermore, the partial and reduce operations can be flexibly composed into various execution patterns. As Table.1 shows, the MoE forward conforms to a *partial-partial-reduce* topology, while the RMSNorm follows a *partial-reduce-partial* pattern.

Operator-based declarative programming. Inspired by classic systems such as Spark [72, 73], we propose a declara-

```

1 def decoding_attn (Q, K, V, NUM_HEAD, batch):
2   @Intermediate
3   class attnInter: lse: float; o: Tensor(Q_shape)
4   @Iter(range=(start, end), affinity = shared_prefix_range)
5   def attnSplitKV(seq, head, Q, K, V) -> attnInter:
6     acc_o, lse = 0, -inf # The implementation is simplified
7     Copy(Q[seq, head], Q_shared)
8     for idx in Pipelined(start, end): #load&compute pipeline
9       Copy(K[seq, idx, head], K_shared, absorbable = True)
10      Copy(V[seq, idx, head], V_shared, absorbable = True)
11      GEMM(Q_shared, K_shared, acc_s)
12      Softmax(acc_s, acc_s_cast, lse, scores_scale, ...)
13      GEMM(acc_s_cast, V_shared, acc_o)
14      return attnInter(lse=lse, o=acc_o)
15   @Reduce
16   def attnCombine(iList: [attnInter]) -> attnInter:
17     l_max, l_norm = -inf, 0.0
18     acc = alloc_shared(init=0.0, shape=Q_shape)
19     for i in iList:
20       weightedAccumAndNorm(i.lse, i.o, l_max, l_norm, acc)
21     return attnInter(lse=l_max + log2(l_norm), o=acc)
22   @Finalize
23   def attnFinalize(inter: attnInter, seq, head, Output):
24     Copy(inter.o / exp2(inter.lse), Output[seq, head])
25
26   with ParallelTasks(NUM_HEAD, batch) as (head, seq):
27     partial(attnSplitKV(seq, head, Q, K, V))
28     .reduce(attnCombine())
29     .finalize(attnFinalize(seq, head, Output))
30
31   ▶MoE forward, with the function implementations omitted
32   with ParallelTasks(NUM_EXPERT) as (expert):
33     map(FFN1(expert, tokens)).reshard(reshardInterm())
34     .map(FFN2(expert)).gather(by=tokenID)
35     .reduce(weightedSum())
36     .finalize(MoEFinalize(Output))

```

Figure 6. Decoding attention and MoE forward with Meld. Notably, the chaining syntax (e.g., `partial().reduce()`) only declares **dataflow semantics**; the actual sub-task partitioning and reduction orchestration are determined at runtime.

tive programming model. In contrast to traditional coupled programming paradigms, Meld enables developers to define the generic computation logic through operators, while leaving the specific parallelization/cooperation strategies dynamically tailored by the runtime according to input shapes.

Meld’s programming model is *operator-centric*. As Table 2 shows, developers express computation logic by defining and composing operators. The `iter` and `map` operators are both partial operators that compute over an arbitrary range along an input dimension. But `iter` produces fixed-size intermediate results regardless of range length (e.g., attention), while `map`’s output size scales with the range (e.g., MoE forward). The `reduce` operator specifies the logic for merging intermediate results, and `finalize` dictates how the final reduction output is handled. The operators support parallel execution across multiple SMs via the `Parallel` primitive. Fig. 6 shows the code for decoding attention and MoE forward.

Meld specifies the intermediate states with the intermediate abstraction. It serves as the connection between the operators. Through this abstraction, Meld decouples the management of intermediate results from the computation logic. Programmers are relieved from managing the placement and

Op.	Description
Iter	▶ A partial operator. An <code>Iter</code> subtask generates an intermediate result from inputs within a designated range.
	▶ The affinity accepts a user-defined function to specify the range eligible for copy absorption between two tasks. <code>absorbable</code> flag of <code>.copy</code> annotates data loads eligible for absorption.
	▶ It can exploit multiple blocks with Parallel primitive , e.g., with <code>Parallel(ceildiv(dexpert, block_dexpert), 1)</code> as <code>(bx, by)</code> :
Map	A partial operator, generates an interm. result for each element.
Reduce	It aggregates two or more intermediate results. The output is also an interm. result to facilitate cascaded/tree-based reduction.
Reshard	Redistribute interm. results between two partial operators; e.g., between MoE forward’s two partial stages, the interm. results are re-sharded across SMs (facilitated by on-chip interconnect).
Gather	It collects per-element interm. results produced by <code>Map</code> across tasks and regroups them by element identity, enabling cross-task <code>Reduce</code> ; e.g., in MoE, it gathers each token’s expert outputs.
Finalize	Execute following the final reduce (e.g., materializing results).

Table 2. Operators in Meld. Meld operator set is extensible.

transmission details, which are coupled to specific scheduling strategies, and dynamically orchestrated by the runtime.

Holistic optimization with copy absorption. SM cooperation unlocks a new optimization we call *copy absorption* [74], which eliminates redundant data fetches from global memory by broadcasting the shared data across SMs with on-chip interconnects. Meld introduces the affinity abstraction, which accepts a user-defined function that identifies the ranges eligible for copy absorption between two partial tasks (e.g., with shared prefix KV cache). Within Meld, data copies are facilitated by the `Copy` primitive, where developers can annotate the copies that are eligible for absorption. With affinity abstraction, Meld constructs a global view for batch-level optimization, which guides runtime’s orchestration (§4.3).

4.2 Taming On-Chip Links with Hybrid Primitives

Although the on-chip interconnects have been introduced in mainstream (NVIDIA Hopper/Blackwell) GPUs, their underlying topology remains opaque. Furthermore, relying on the high-level CUDA interfaces causes severe SM underutilization (§2.4), which often negates the performance benefits.

Dissecting the on-chip heterogeneity. Our reverse engineering⁷ finds that, distinct from chip-wide interconnects in wafer-scale chips [75–88], GPU on-chip interconnects adopt a group-based topology. Specifically, SMs are (physically) partitioned into multiple groups (GPCs, Graphics Processing Clusters). As Fig. 7 (b) shows, SMs within a group enjoy all-to-all connectivity, but no direct interconnect exists between SMs across groups. The interconnect provides highly efficient data transfer, with $4\times$ the bandwidth and $1/7$ the latency of HBM-proxied transfer. However, as Fig. 7 (a, c) illustrates, it is spatially localized and highly heterogeneous; on a single GPU, the GPC sizes range from 2 to 20.

⁷Method detailed in Appendix A. The combination of some SMs (e.g., the 1×8 -SM and 4×2 -SM in H100) is considered a special logical GPC (without full on-chip interconnect) in some documents.

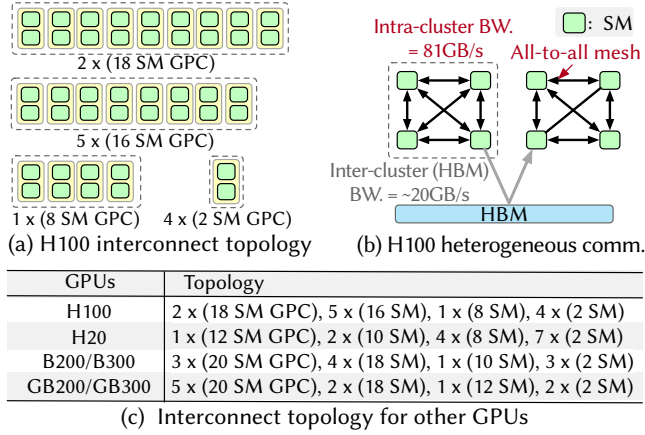


Figure 7. GPU on-chip topology. We report bidirectional bandwidth. The SM-to-SM bandwidth drops to 84% for clusters with > 8 SMs, indicating more complex hardware mechanisms. As far as we know, this is the first work to characterize GPUs’ on-chip cluster topology.

Challenges. The unique topology of GPU interconnects introduces two dimensions of heterogeneity, which impede efficient SM-level cooperation. First, the locality of interconnects dictates that SM-level cooperation has to rely on *heterogeneous communication paths* (i.e., HBM-proxied and link-based), necessitating specialized communication primitives. Second, *cluster heterogeneity* dictates that relying solely on CUDA’s native cluster abstraction leads to SM fragmentation, demanding novel cluster dispatching mechanisms.

Hybrid primitives for heterogeneous links. Due to the spatial locality of on-chip interconnects, practical SM-level cooperation must rely on hybrid communication paths, combining high-speed on-chip links and global HBM. Meld designs a set of primitives based on the hybrid model.

As Table.3 shows, the send/recv primitives leverage TMA-based async copy (between SMs’ shared memory) and double buffering (async mode) to achieve high-performance transfer, while employing hardware mbarriers for synchronization. When cross-cluster communication is required, they fall back to HBM-proxied transfer, with polling-based synchronization. The broadcast facilitates copy absorption (i.e., cooperative loading). The allGather and allReduce primitives adopt a two-phase strategy. They first communicate (in parallel) within each cluster, followed by a data exchange via global memory. To maximize efficiency, Meld scheduler (§4.3) prioritizes the use of async send/recv for communication, and attempts to use allGather/allReduce within a single cluster.

Hybrid dispatching for heterogeneous clusters. CUDA introduces the cluster abstraction, enabling a kernel to specify its cluster size. But the mismatch between the hardware-oblivious abstraction and heterogeneous hardware clusters leads to severe SM fragmentation (e.g., a cluster size of 10–16 on H100 leaves 20–62 SMs idle). While Blackwell introduces a fallback cluster size⁸, the alignment constraints restrict

⁸cudaLaunchAttributePreferredClusterDimension.

Primitives	Design and implementation
send/recv	Intra-cluster: Async transfer with TMA, and mbarrier for synchronization.
	Inter-cluster: TMA-based store and load to/from HBM, polling an HBM synchronization variable for synchronization. Device-scope fences enforce memory ordering.
broadcast (load)	The program specifies a group of SMs within a cluster with a mask. A designated SM issues (TMA) copies and multicasts to the others, while the others synchronize via mbarriers.
allGather allReduce	Phase 1: Intra-cluster. Ring allGather/allReduce.
	Phase 2: Inter-cluster. Global memory-based data exchange and flag-based synchronization.

Table 3. Hybrid communication primitives. Meld can adopt other inter-cluster allGather/allReduce algorithms [89, 90].

this fallback to 2 for full SM utilization, underutilizing on-chip interconnects; e.g., on B40 GPUs, optimal interconnect efficiency requires a layout with 10 16-SM and two 14-SM clusters, but currently only a 2-SM fallback is supported.

Therefore, Meld proposes heterogeneity-aware cluster dispatching. For each GPU, guided by its topology, Meld determines a mix of 2–3 cluster sizes, to maximize on-chip interconnect utilization while ensuring full SM occupancy. Specifically, starting from the largest feasible cluster size, it progressively matches GPCs and falls back to smaller sizes for remaining SMs. To execute a program with different cluster sizes, Meld transparently launches *concurrent kernels*⁹ with distinct cluster configurations (via multiple streams). GPU’s unified global memory address space enables the clusters in concurrent kernels to still synchronize and communicate via the HBM-based primitives. Meld virtualizes cluster/block ranks across the concurrent kernels. On Blackwell, it hybridizes concurrent kernel dispatching with native fallback cluster size. Meld scheduler (§4.3) orchestrates tasks across the clusters, which remain transparent to developers.

4.3 Hierarchical Coordination with Stage Fusion

Empowered by decoupled execution and Meld abstraction, Meld dynamically dictates hardware mapping and collaboration strategies based on input shapes with Meld scheduler.

Challenges. Unlike wafer-scale chips [77, 78] that support fabric-wide data flows, a GPU’s on-chip interconnects are strictly confined within SM clusters. This architectural constraint introduces a fundamental tension: coarse-grained task allocation fails to achieve GPU-wide load balance, whereas fine-grained assignment incurs prohibitive transmission and synchronization overheads via off-chip memory.

Hierarchical scheduling. To align with this hardware topology, Meld introduces a two-tiered scheduler. Tier-1 governs inter-cluster task allocation to minimize cross-cluster data traffic, while Tier-2 manages intra-cluster orchestration to establish a bubble-free, sub-task-level execution pipeline.

⁹Kernels on different streams run concurrently when their SM/cluster requirements do not conflict. Meld chooses non-overlapping cluster/grid configurations to enable co-residency, and assigns different priorities to the streams (so that kernels with larger cluster sizes are launched first).

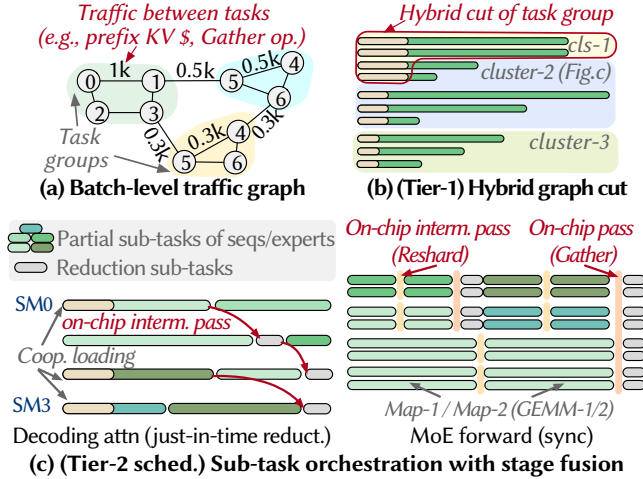


Figure 8. Hierarchical scheduling in Meld scheduler.

Inter-cluster scheduling with hybrid graph. To preserve global load balance without exacerbating communication costs, Meld first calculates a workload budget for each cluster. It then constructs a *two-level traffic graph* to guide task allocation (Fig. 8 (a)). Specifically, at batch level, the graph captures inter-task traffic, e.g., for attention, its edges denote the reducible data loading from copy absorption across sequences; for MoE forward, its edges denote the intermediate state transfers incurred by the gather operator across experts. At task level, the graph captures the dataflow between operators (e.g., partial $\rightarrow$ reduce), where the edge weights, representing intermediate state transfer costs, are parameterized by the degree of partitioning of partial operators.

Based on this hybrid graph, Meld clusters nodes into task groups and maps them to hardware clusters using a largest-first heuristic. If a group exceeds a cluster’s budget, Meld applies a *hybrid cutting* strategy that blends group-level and task-level partitioning to minimize the overall increase in off-chip memory traffic, as Fig. 8 (b) shows.

Intra-cluster orchestration with stage fusion. Within each cluster, Meld schedules sub-tasks using *stage fusion*. By leveraging fine-grained synchronization and high-speed on-chip data transfers, stage fusion transforms traditional phase-by-phase execution into a fluid, sub-task-level pipeline.

Specifically, as Fig. 8 (c) shows, when the available shared memory suffices for a communication buffer (e.g., most iter tasks like attention), Meld prioritizes asynchronous transmission with **just-in-time reduction**. It first schedules cooperatively loaded segments, then arranges the tasks (e.g., sequences) in longest-first order, and finally adjusts the plan by interleaving short tasks into the SMs filled by long ones, forming a fluid asynchronous pipeline. Conversely, when shared memory is constrained (e.g., Map tasks like MoE forward), Meld falls back to synchronous communication while still sustaining the sub-task-level pipeline. The scheduling is constrained by the per-SM workload budget to ensure

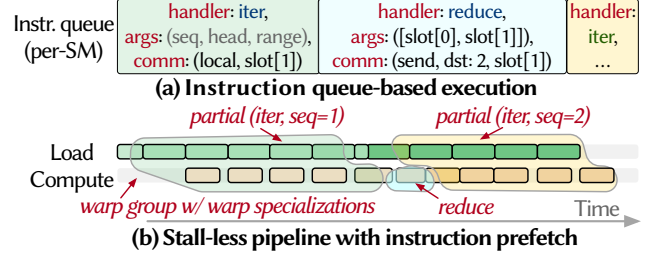


Figure 9. Execution paradigm of Meld. When multiple thread blocks are needed to saturate an SM, the instruction queue is per-block.

workload balance across SMs. After intra-cluster orchestration, Meld inserts cross-cluster communication primitives to integrate intra-cluster pipelines into GPU-wide cooperation. Developers can also integrate custom schedulers.

4.4 Execution with Executable-Runtime Decoupling

Traditional compilers [22–24, 26] conceptualize compilation as lowering high-level abstractions into fixed programs. However, dynamic inputs necessitate adaptive scheduling, and the resulting schedules must be adaptively mapped onto the non-uniform topology, which challenges this static model.

Execution with decoupling. Meld abandons the static model, and decouples the output program into two distinct components: static *executables* and a dynamic *runtime*. The executables consist of pre-optimized operators wrapped as *handlers*. The runtime comprises a scheduler for orchestration and a GPU kernel (called *wrapper*) that executes the scheduler’s directives. During execution, the scheduler first generates the execution plan based on specific input shapes. Subsequently, the GPU wrapper carries out the plan by invoking the appropriate handlers and communication primitives.

Challenges. While decoupled execution enables adaptability, building an efficient runtime introduces three challenges. First, the opaque, black-box GPU hardware scheduler prevents the precise mapping of our execution plans onto the heterogeneous SM topology. Second, fine-grained, dynamic task partitioning can easily disrupt and stall the highly optimized intra-SM hardware pipelines. Third, the overhead of dynamic scheduling may incur undesirable GPU stalls.

Task mapping with instruction queue. Traditional approaches that rely on the hardware GPU scheduler [21, 22] cannot achieve precise task mapping onto the heterogeneous topology. To overcome this, Meld bypasses the GPU scheduler entirely. At compile time, Meld derives a unified thread block configuration (e.g., shared memory size, thread count) for each DNN program based on its operators. At runtime, it launches a *wrapper* kernel to fully saturate the GPU. Each wrapper block fetches sub-tasks from an SM-specific *instruction queue* populated by the Meld scheduler. Each instruction in the queue specifies a handler, its parameters, and the communication primitives to invoke upon handler completion (Fig. 9 (a)). This OS-like approach decouples hardware threads

from software tasks, enabling topology-aware mapping and highly flexible SM cooperation.

Stall-less intra-SM pipeline. Modern GPUs rely on intra-SM pipelines across hardware units to maximize efficiency [22]. This is typically realized through warp specialization [18, 41], where warp groups are assigned distinct roles (e.g., usually one warp group for async load/store via TMA [53], and several groups for computation). However, in Meld, executing instructions at the granularity of thread blocks introduces pipeline stalls between consecutive instructions, degrading overall efficiency. To resolve this, Meld leverages the consistent role distribution across handlers to construct a stall-free pipeline. It prefetches the instructions and fuses the intra-SM pipelines across handlers. As Fig. 9 (b) shows, data loading for the subsequent instruction is proactively executed while the current instruction’s computation is still in progress.

Coordination with tiling strategies. Traditional approaches rely on uniform kernel implementations, which compromise efficiency of fine-grained SM cooperation. Meld not only dynamically determines the execution plan but also adaptively tailors the tiling strategy for each sub-task. Specifically, at compile time, Meld generates handlers with different tiling strategies for different sub-task shape ranges. At runtime, it dispatches the appropriate handler to each sub-task.

Async scheduling and plan cache. Although Meld scheduler is efficient, it may still lead to GPU idling and impede the full effectiveness of CUDA graphs [91]. We observe that within high-level frameworks [43, 54, 92–96], time windows typically exist between determination of input shapes and actual kernel invocation, creating an async scheduling opportunity. For instance, inference frameworks employ async task scheduling [97, 98], overlapping task scheduling for the next decoding step with the execution of the current one. Similarly, in expert parallelism [99, 100], token-to-expert routing metadata is available before the actual token arrives.

Therefore, Meld schedules asynchronously, allowing frameworks to pre-schedule and persist the results (i.e., instruction queues) in GPU memory. Furthermore, Meld proposes a *plan cache* to reuse the scheduling plans. For example, during inference, the plans for attention can be shared across all the transformer layers within a step. And because Meld schedules at the granularity of KV cache blocks, consecutive decoding steps can usually also share the plans.

5 Implementation

Meld is built on TileLang [21, 27] and TVM [28], with ~60K lines of CUDA/Python modifications.

Tile selection and resource provisioning. At compile time, for each operator, Meld generates multiple handlers for different sub-task size ranges (e.g., sub-sequence-length ranges for attention), potentially using a different tiling strategy for each handler. The wrapper uses the maximum thread count required by the handlers. A handler requiring fewer

threads activates only the corresponding subset of threads or warps. The wrapper’s per-thread register allocation is likewise fixed at compile time based on the handler set. When handlers use different numbers of active warp groups, Meld optionally emits `setmaxnreg.dec` and `setmaxnreg.inc` instructions to redistribute registers among warp groups.

Shared-memory allocation. Within the available shared-memory budget, Meld follows a *compute-first* allocation policy. It first reserves the memory required by the handler’s computation tiles and assigns the remaining capacity to communication slots. Async just-in-time reduction requires one additional intermediate-sized buffer. If the compiler finds the remaining shared memory insufficient for communication slots, the scheduler selects synchronous communication, which reuses the computation tiles for communication.

Cross-SM synchronization. Each communication slot is associated with a hardware-managed `mbarrier`. Before issuing a transfer, the producer ensures that the destination slot is reusable and initializes the barrier with the expected number of bytes (`mbarrier.arrive.expect_tx`). It then issues an asynchronous TMA copy directly into the consumer SM’s slot. TMA hardware updates the barrier as bytes arrive, and the consumer accesses the slot only after the barrier reports completion (`mbarrier.try_wait.parity`). The barrier also guards the lifetime of the source slot: the producer cannot reuse the source slot or exit while the communication is still live. For cross-cluster communication, where direct shared-memory transfer is unavailable, Meld instead uses HBM-resident exchange buffers, atomic synchronization variables, and device-scope fences to enforce visibility and ordering.

Integration with CUDA Graph. Kernels generated by Meld can be integrated into CUDA Graphs as before (i.e., capture once and then replay) because their launch metadata remains fixed across inputs, e.g., kernel arguments, pointer addresses, and grid/block configuration. For dynamic inputs, Meld scheduler only updates the contents of per-SM instruction queues before graph replay via CPU-GPU copy.

6 Evaluation

Kernels and hardware. We evaluate Meld using kernels critical to LLM inference/training with dynamic workloads, as Table.4 details. We mainly evaluate on Hopper (H100) and Blackwell (GB200) GPUs, which represent the mainstream hardware in industrial data centers. We also validate on RTX 5090¹⁰, a representative consumer-grade GPU.

Workloads. For kernel-level performance benchmarks (§6.1), we replay representative real-world traces (Table.4) to cap-

¹⁰Much evidence suggests that, on GPUs based on GB202 chipsets, e.g., RTX 5090 and B40 (SM120), the cluster functionality is emulated in software using synchronization variables and buffers in global memory, rather than implemented through hardware on-chip interconnects. Meld uses cluster functionality for fine-grained inter-SM synchronization, but avoids using its multicast functionality, which would cause severe performance degradation.

Kernels	GPUs	Settings
Attention (GQA)	RTX 5090	Qwen3-1.7B [3], TP=1
Attention (GQA)	H100/GB200	Qwen3-30B-A3B, TP=2
Attention (MLA)	H100/GB200	DeepSeek-V3.2 [2], TP=4
MoE forward	H100/GB200	Qwen3-30B-A3B and DeepSeek-V3.2, EP=4, EPLB [101]
MoE forward	RTX 5090	GPT-OSS-20B, EP=1 [101]
NSA indexer	H100/GB200	DeepSeek-V3.2, FP8
RMSNorm	H100/GB200	Dim of 32K–80K
Cross-entropy loss	H100/GB200	RL/SFT [102–104] for math/code

Scenarios	Traces	Prompt avg. & std.	Response avg. & std.
Chat w/ think	Dolphin-r1 [105]	138 & 325	3259 & 3497
Fast chat	ShareGPT [46]	929 & 1264	466 & 652
Math	OpenR1-Math [106]	87 & 66	11311 & 7401
Code	OpenThoughts-Code [107]	405 & 234	7650 & 5641

Table 4. Kernels and workload traces used for evaluation.

ture batch-level load distributions. For sequence-length-dependent kernels (e.g., attention, NSA), we record per-batch length distributions across batch sizes for every forward pass and apply *quantile normalization* to derive average distributions. For MoE, we record per-token expert routing to compute average activation distributions. For serving benchmarks (§6.4), we use the traces directly as the input requests.

Metrics. We use latency as the primary metric for kernel performance, and report the average latency across 100 iterations following 50 warm-up runs. We also assess the end-to-end performance within a fully integrated serving system, with throughput as the metric, measured in tokens/s.

Baselines. We compare with 6 baselines, including expert-optimized kernels [6–8, 10], i.e., FlashInfer (v0.6.0), FlashMLA (48c6dc4), FlashDecoding (a5856bf), DeepGEMM (477618), and widely adopted compilers, i.e., TileLang (v0.1.8), Triton (v3.5.0). The TileLang/Triton operator implementations are from TileOps [108], official examples [109], and production implementations within vLLM/SGLang [43, 54].

6.1 Performance of GPU Programs

Attention (GQA, MLA). For the GQA kernel, Meld outperforms FlashDecoding, FlashInfer, TileLang, and Triton by 1.24–3.68 $\times$, 1.02–1.53 $\times$, 1.10–2.45 $\times$, and 1.35–3.00 $\times$. For the MLA kernel, Meld outperforms FlashMLA, FlashInfer, TileLang, and Triton by 0.98–1.65 $\times$, 1.01–2.78 $\times$, 1.12–2.03 $\times$, and 1.46–2.59 $\times$. As detailed in §2.2 and Fig. 3, FlashDecoding, TileLang, and the Triton kernel (from vLLM) all employ the Split-KV strategy. Although FlashInfer and FlashMLA incorporate customized scheduling, fundamental inefficiencies persist, e.g., SM underutilization during the “combine” phase, memory traffic for intermediate results, and uniform tiling strategies. Furthermore, FlashDecoding incurs internal memory allocation overhead for intermediate results, which severely degrades its performance at small batch sizes. We utilize randomly generated KV here. With shared prefixes, Meld achieves an additional speedup of 1.04–1.71 $\times$ (Fig. 11).

DeepSeek NSA indexer uses compressed and quantized Queries and Keys to compute scores for each token chunk and select the top-k chunks for sparse attention. Meld outperforms DeepGEMM, TileLang, and Triton by 1.12–2.06 $\times$, 1.12–3.06 $\times$, and 1.39–3.58 $\times$. The baselines execute with two kernels. The first computes logits for every chunk; the second sorts and selects the top-k chunks. In contrast, Meld employs a fused partial (Iter)-reduce execution. An Iter sub-task finds the top-k chunks within its local sub-sequence, and outputs the local top-k chunks, their scores, and the Log-Sum-Exp (LSE) (for normalization) as intermediate states. A Reduce sub-task normalizes the scores and applies a merge sort to yield the merged top-k results. Meld bypasses the global memory traffic incurred by intermediate logits, and avoids the long tail and SM idling of the top-k phase.

MoE forward follows a partial (Map)-partial (Map)-reduce paradigm (§2.2). Meld outperforms the baselines (FlashInfer, TileLang, Triton) by 1.06–2.92 $\times$, 1.34–1.96 $\times$, and 1.53–3.76 $\times$, primarily because: (1) By leveraging fine-grained sub-task-level dependencies, Meld fuses all phases into a single kernel, eliminating the overhead of repeated kernel launches and terminations, as well as the low utilization caused by long-tail execution in each phase (heavy experts or the underfilled tail-wave). (2) Through SM cooperation, Meld enables direct intermediate state transfer across phases without incurring expensive HBM traffic. (3) Copy absorption further reduces the overhead of repeatedly loading hidden states in GEMM-1.

RMSNorm normalizes hidden states via two stages: traversing elements to compute Root Mean Square and applying it to normalize elements. FlashInfer and Triton restrict each hidden state’s RMSNorm to a single SM. For large dimensions, they employ redundant chunked loading. TileLang’s Split-K uses three stages, i.e., parallel local RMS computing, RMS merging, and parallel normalizing, but incurs HBM overhead for intermediate results and redundant hidden state loading. Meld adopts a fused partial (Iter)-reduce-partial (Map) paradigm: multiple SMs compute local RMS in parallel, merge values via the on-chip interconnects, and directly normalize the hidden state that is already resident in shared memory. This cross-SM fused execution avoids HBM traffic and redundant loading. Meld outperforms FlashInfer, TileLang, and Triton by 1.05–1.61 $\times$, 1.02–1.49 $\times$, and 1.09–1.80 $\times$.

Cross-entropy loss (CE) is commonly employed in Reinforcement Learning (RL) and Supervised Fine-Tuning (SFT) to aggregate per-token losses and derive the final per-sequence loss. Meld adopts a partial (Iter)-reduce paradigm. In contrast to baselines that rely on static Split-K strategies, Meld achieves more effective global workload balancing and low-overhead intermediate transmission. It outperforms TileLang and Triton by 1.22–1.52 $\times$ and 1.27–1.56 $\times$.

Copy absorption for attention. During inference, requests often share common prefixes, e.g., system prompts and agent tool/skill descriptions. For instance, Claude prepends a 13K-

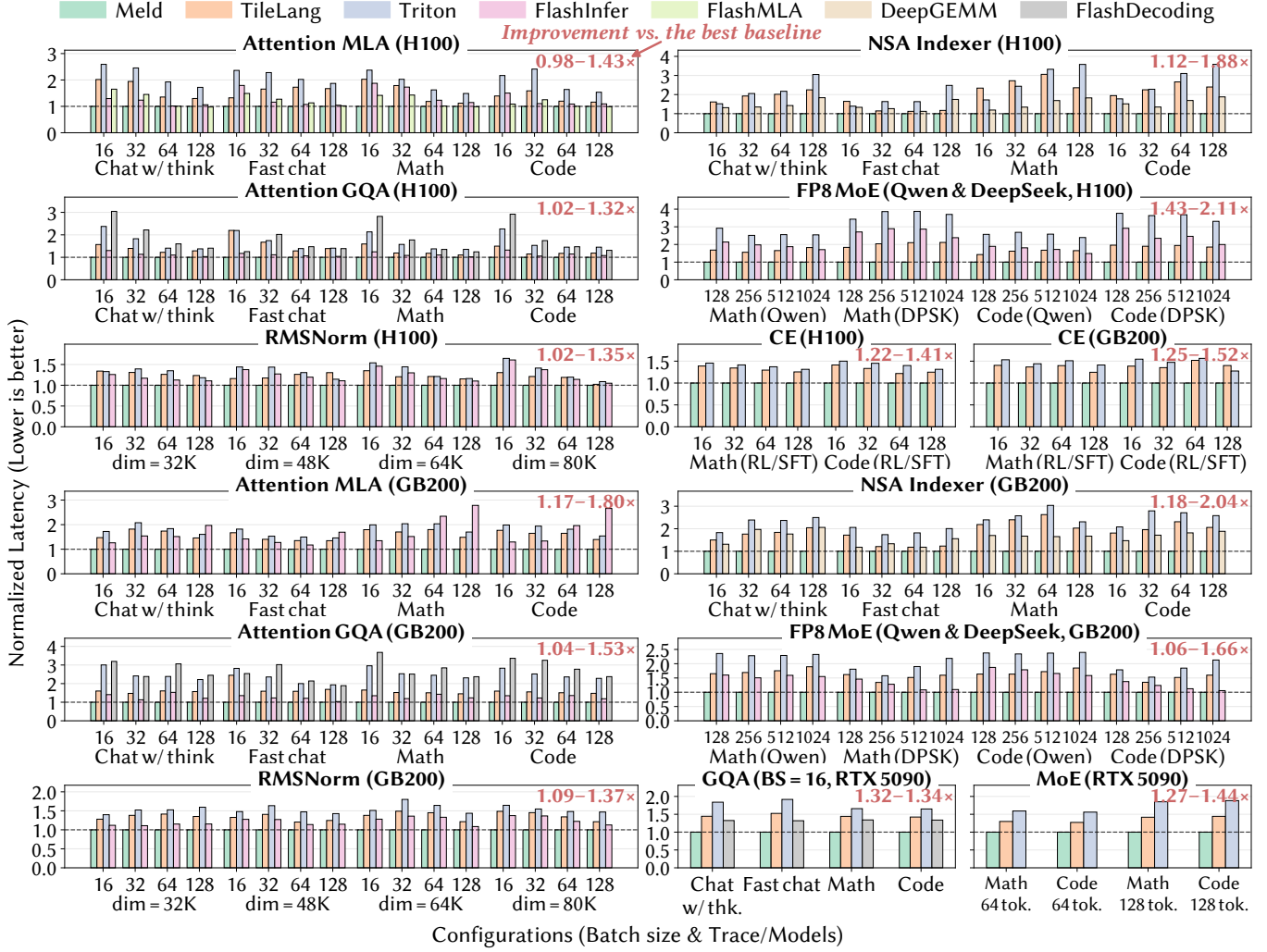


Figure 10. Performance of various kernels, under various traces.

token system prompt [110], while OpenClaw’s skill descriptions can span tens of thousands of tokens [111]. Inference frameworks [54, 55] employ prefix cache [112–114] to avoid redundant KV cache storage; Meld further eliminates redundant loading via copy absorption. As Fig. 11 shows, for prefixes of 1K–8K tokens, Meld achieves 1.04–1.71× additional speedup in MLA over Meld without copy absorption.

Summary. Meld outperforms hand-optimized and compiler-based solutions, because: (1) It *dynamically balances* workloads across SMs, minimizing SM idling. (2) With *stage fusion*, it mitigates HBM traffic for intermediate states, and circumvents SM underutilization in “reduction” phases. (3) It generates *customized handlers* (with customized tiling strategies) tailored to varying sub-task shapes, and uses the optimal variant flexibly at runtime. (4) With *copy absorption*, it reduces redundant data loading, e.g., attention and MoE forward.

While some hand-optimized kernels use customized schedulers for “partial computing” phases, they fail to address the systemic challenges in (2)–(4) without SM-level cooperation.

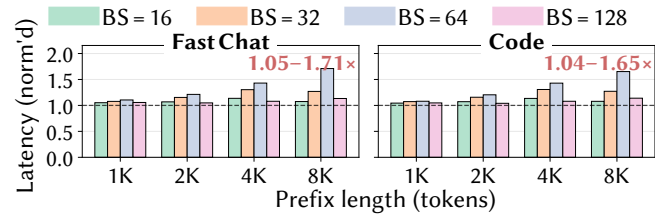


Figure 11. MLA performance with different prefix lengths. H100 (SXM5) GPU, BS = batch size. The relative uplift diminishes at batch size 128 as the proportion of prefix tokens decreases.

6.2 Ablation Study

We further break down the improvements. As Fig. 12 shows, “decouple” means dynamic orchestration with decoupled execution, i.e., with persistent scheduling and fine-grained load balancing, but using HBM-based communication instead of the on-chip interconnects; “+cooperate” means on-chip interconnect-based communication, i.e., just-in-time reduction; and “+absorb” means copy absorption (i.e., full Meld).

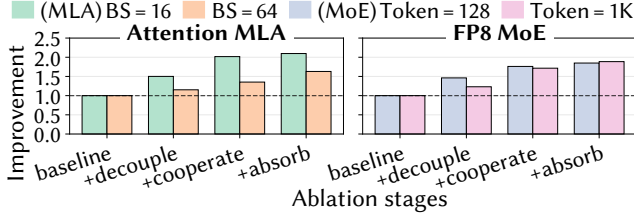


Figure 12. Improvement breakdown. H100 GPU, DeepSeek-V3.2. ShareGPT trace for MLA, shared prefix = 4K. Math trace for MoE.

Take attention and MoE as examples. For fused-MoE kernels, the gains primarily come from *decoupling* and *cooperation*. *Decoupling* introduces fine-grained dependencies, enabling GEMM-1 and GEMM-2 to be fused and eliminating the long tails caused by heavy experts. *Cooperation* enables GEMM-1’s intermediate to be resharded directly among SMs through the SM-to-SM interconnect, avoiding additional HBM traffic and device-scope fences. *Copy absorption* is used only to accelerate hidden-state loading in GEMM-1 and provides a modest improvement (1.04× for DeepSeek).

For attention kernels, fine-grained load balancing enabled by *decoupling* contributes significantly at small batch sizes, where the long tails and workload imbalance in the Split-KV and Combine phases are more severe. *Copy absorption* contributes substantially at large batch sizes but provides modest benefits at small batch sizes. *Cooperation* delivers performance benefits at both large and small batch sizes.

6.3 Comparison with MegaKernels

Meld and recent MegaKernels [115–117] share some sources of speedup, such as reducing kernel-launch overhead through extensive fusion. However, even fully fused MegaKernels can still suffer from low utilization under dynamic and irregular workloads, which usually have complex and dynamic cross-SM dependencies across execution phases. For example, MPK [115] relies on a static partition method for decoding attention (i.e., a pre-defined max chunk length), and the reduction for a long sequence starts only after all of its partial (i.e., split-KV) tasks have completed, which can create tails for the long sequences within the batch (and leave SMs waiting on the event counter before entering MoE Grouped GEMM). Meld’s key contribution is that it proposes a general and efficient solution for dynamic cross-SM dependencies, i.e., *fine-grained dependencies enabled by SM cooperation*, which enables balanced execution and high SM utilization.

In workloads with relatively simple and static cross-SM dependencies, such as MoE forward, recent MegaKernels (e.g., MegaMoE [116, 118]¹¹) achieve load balance using specialized schedulers and implement fine-grained communication through MoE-specific HBM-resident buffers. Compared with them, Meld expresses these mechanisms through a general DSL and runtime, and further leverages on-chip SM cooperation to optimize synchronization and data movement.

¹¹MegaMoE was released after this paper’s submission.

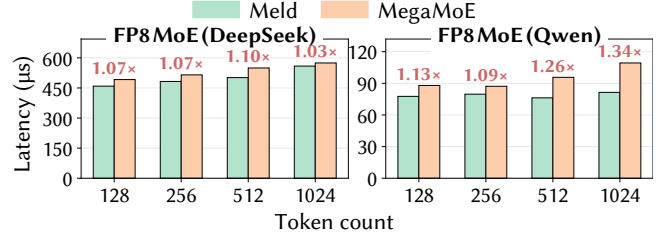


Figure 13. MoE forward performance. H100 GPU, Math trace.

As Fig. 13 shows, compared with MegaMoE, the fused-MoE kernels generated by Meld achieve comparable performance (up to 1.34×) and perform better at small token counts (DeepSeek) or small expert sizes (Qwen). This improvement primarily comes from Meld’s on-chip-interconnect-based SM cooperation, which is more efficient than inter-SM data transfer through global-memory buffers.

Meld can be further integrated with MegaKernel methods [115, 119] and the communication-computation fusion techniques [120, 121] to enable broader fusion while maintaining efficient and load-balanced execution.

6.4 End-to-End Performance

We integrate Meld into Mini-SGLang [122], which is SGLang’s official compact implementation with equivalent performance. Mini-SGLang uses FlashInfer, TensorRT-LLM [123] kernels, and hand-written SGLang kernels [124]. We allow the inference system to automatically adjust batch size and intra-batch composition. As Fig. 14 shows, under real-world workloads, Meld achieves up to a 1.39× throughput improvement.

The improvement of Meld on GB200 is more significant than on H100, because GB200 has more SMs, which amplifies the workload imbalance among SMs, makes SM underutilization during reduce phases more severe, and increases the overhead of repeatedly storing and loading intermediate results (as tasks are partitioned into more sub-tasks).

6.5 Extended Studies

6.5.1 Development Efforts. To use Meld, developers typically only need to rearrange existing computation logic into operators (§4.1). As Table. 5 shows, MLA on H100 with Meld requires 27 lines of additional code.

6.5.2 Hardware utilization. From the perspective of hardware utilization, Meld accelerates kernels for two main reasons. (1) Fine-grained dependencies and balance eliminate the low-utilization tail of the phases, improving compute and memory-bandwidth utilization. (2) SM cooperation avoids expensive round trips through global memory, which reduces end-to-end time. Theoretical effective-bandwidth estimates show that Meld achieves 2,688 GB/s HBM bandwidth (H100, 128 tokens, DeepSeek-Math), which is 2.5× that of FlashInfer CUTLASS fusedMoE (1,070 GB/s). This improvement primarily results from (1). Compared with MegaMoE (2,781 GB/s), Meld achieves 97% of its effective bandwidth. Its lower la-

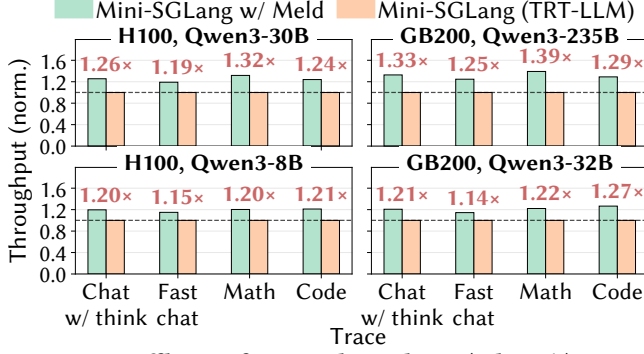


Figure 14. Offline inference throughput (tokens/s). $TP = 4$, $EP = 4$ for MoE models (30B/235B), $TP = 1$ for dense models (8B/32B).

tency relative to MegaMoE is because of (2).

6.5.3 Scheduling and Compiling Time. As Table.5 shows, Meld’s scheduling takes < 2 ms, which can be overlapped via async scheduling. Its compilation takes 1.5–2.2 $\times$ as long as TileLang’s, as it generates handlers for different sub-task shapes. This is acceptable because only an initial compilation is needed for dynamic workloads.

6.6 Discussion

Generality. Meld achieves full functionality on GPUs with on-chip links (Hopper, Blackwell, Rubin). For GPUs without them, it falls back to global memory-based communication and flag-based synchronization, but still yields improvements, as Fig. 12 (+decouple) confirms. Meld is also implemented on Ascend NPUs, proving its generality.

Limitations and scope. Meld runtime is designed for dynamic workloads that require SM cooperation and precise mapping onto the heterogeneous SM topology. For homogeneous, static workloads, e.g., GEMM, relying on the GPU hardware scheduler and CLC [125] to dispatch thread blocks is typically simpler and more efficient.

Deterministic execution. Deterministic, batch-invariant kernel execution is important for training and debugging, but is often undermined by nondeterministic block scheduling and accumulation order. With the runtime that bypasses GPU scheduler and just-in-time reduction, Meld kernels provide deterministic execution. We have verified that Meld achieves bitwise determinism for the evaluated kernels.

7 Related Work

Dataflow processors. While dataflow processors [75–85] have on-chip links, they diverge fundamentally from GPUs. They statically map operators onto PUs, with compile-time-fixed communication patterns; e.g., Graphcore/Groq IPUs enforce chip-wide alternating compute and exchange phases. In contrast, GPU’s SIMT model temporally multiplexes SMs, enabling computation and communication to be flexibly interleaved, which underpins Meld’s flexible SM cooperation. Besides, dataflow processors employ homogeneous meshes,

Kernels	Meld LoC vs. TileLang	sched. time	comp. time
MLA	+27	1.71 ms	21.7 s (1.8 $\times$)
GQA	+31	1.66 ms	23.7 s (2.1 $\times$)
MoE forward	+26	1.04 ms	16.9 s (1.3 $\times$)
NSA indexer	+25	1.56 ms	36.3 s (2.2 $\times$)
RMSNorm	+37	0.44 ms	1.34 s (1.8 $\times$)
CE loss	+19	1.65 ms	7.4 s (1.5 $\times$)

Table 5. Development efforts and compiling/scheduling time. Sched. time reported for the last setting (Fig. 10, H100) per program.

and rely on a flat SRAM hierarchy. GPUs utilize heterogeneous interconnects (§4.2) and a pyramidal hierarchy (§2.1). Kitsune [126] enables the dataflow execution model on GPUs by introducing L2-resident inter-CTA queues and a CUDA spatial pipeline API. Meld exploits commodity GPUs, which dominate data centers.

Specialized utilization of on-chip interconnect. ClusterFusion [127] manually fuses QKV projection, attention, and output projection via interconnects, but adheres to a “phase-by-phase” model. Its rigid static mapping (one cluster per head per sequence) aggravates SM underutilization under dynamic workloads, and fails to address cluster-induced SM idling. FA-4 [19] exploits 2-CTA interconnects with 2-CTA UMMA in backward attention. QuACK [128] exchanges states with interconnects for kernels but does not support dynamism, e.g., batches with various lengths. Unlike the specialized techniques, Meld is a systematic framework for general DNNs. It provides multi-level interconnect abstractions; e.g., DNNs can embed a FA-4-style 2-CTA strategy in lter, while relying on Meld for flexible SM cooperation.

Compiler for dynamic NNs. Brainstorm [129] and PIT [130] optimize systems for dynamic DNNs, which feature dynamic network structures, instead of dynamic workloads. Tempo [131] proposes symbolic dependence graphs for DNNs with temporal dependencies; it is orthogonal to Meld.

8 Conclusion

We present Meld, a DNN compiler framework for dynamic workloads. With dynamic SM-level cooperation, decoupled execution, and just-in-time reduction, Meld fuses kernels across SMs, with critical improvements over SOTA solutions.

Acknowledgments

We are grateful to our shepherd for his detailed suggestions and to the SOSP reviewers for their constructive feedback. We thank Lei Wang and Tong Wu from the TileLang community and Jiaxiao Zheng, Qingyuan Liu, and Haoru Zhao for their support. This work was supported in part by the National Natural Science Foundation of China (No. 62432010 and No. 62472279), the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (JYB2025XDXM122), the Fundamental Research Funds for the Central Universities, and the Advanced Research Center for Agent-Oriented OS (TC20260106010). Corresponding author: Yubin Xia (xiayubin@sjtu.edu.cn).

A Characterization of GPU Topology¹²

We provide the reverse-engineered cluster topologies of mainstream Hopper and Blackwell GPUs to facilitate future research (Fig. 7). We also provide the reverse-engineering methodology, which allows one to obtain the GPU interconnect topology by running the program, without physical access to the hardware. The reverse engineering method is highly portable and automated. It is software-only, and completes in seconds. Therefore, it is integrated into Meld to automatically characterize unknown GPU architectures or updated GPU firmware at compilation time.

Clarification of the “GPC” term. Some documentation mentions that H100/(G)B200/(G)B300 contain 8 GPCs. This is because the SM combination of (1 x 8-SM, 4 x 2-SM) is considered a special logical GPC on H100, and so is the SM combination of (1 x 10-SM, 3 x 2-SM) on B200/B300. For ease of understanding, in this paper, we define a group of interconnected SMs as a “GPC”. Notably, (G)B200/300 GPUs consist of two dies with the same SM count, but they have different GPC topologies. The cluster heterogeneity stems from unavoidable defects during the GPU chip’s manufacturing process. But our extensive testing across GPU clusters reveals that, despite varying manufacturing defects across individual GPUs, NVIDIA uses the firmware to mask the topologies of GPUs of the same model so that they appear identical.

Reverse engineering method. We provide a detailed description of the method to characterize GPC topologies.

Step-1: Probing the architecture. The first step involves a systematic probing mechanism to determine the underlying hardware topology characteristics. We define S as the set of available software cluster sizes:

$$s \in S = \{1, 2, 3, \dots, S_{max}\} \quad (S_{max} = 16) \quad (1)$$

For each cluster size s , we launch a probing kernel with sufficient thread blocks (each thread block occupies the full shared memory capacity of a single SM, i.e., exactly one thread block per SM) to obtain the maximum number of concurrent active thread blocks (by analyzing the timestamps). We denote this observed maximum capacity as N_s .

Through this profiling, we obtain an observation vector, which encapsulates the hardware’s constraints across different (uniform) cluster sizes:

$$\mathbf{N} = [N_1, N_2, \dots, N_{S_{max}}]^T \quad (2)$$

Step-2: Topology reconstruction. Based on the probing results, we can mathematically reconstruct the exact topology of the on-chip interconnect.

Let C be the set of possible hardware cluster sizes (in number of cores). Since hardware clusters are typically composed of even numbers of cores (i.e., hardware clusters are composed of TPCs, with 2 SMs per TPC), we define

$$c \in C = \{2, 4, 6, \dots, C_{max}\} \quad (3)$$

Let x_c represent the unknown number of hardware interconnect groups of size c . Our goal is to determine the vector

$$\mathbf{x} = [x_2, x_4, \dots, x_{C_{max}}]^T \quad (4)$$

For a given hardware group of size c , the number of completely independent software clusters of size s it can accommodate is given by the floor division $\lfloor \frac{c}{s} \rfloor$. Therefore, the total number of active software clusters of size s across the entire system must equal the sum of the contributions from all hardware groups. This relationship yields the following linear constraint for each s :

$$\sum_{c \in C} \left\lfloor \frac{c}{s} \right\rfloor x_c = N_s, \quad \forall s \in S \quad (5)$$

This can be expressed as a system of linear equations in matrix form, $\mathbf{A}\mathbf{x} = \mathbf{N}$, where the coefficient matrix $A_{s,c} = \lfloor \frac{c}{s} \rfloor$:

$$\begin{bmatrix} \lfloor \frac{2}{1} \rfloor & \lfloor \frac{4}{1} \rfloor & \dots & \lfloor \frac{C_{max}}{1} \rfloor \\ \lfloor \frac{2}{2} \rfloor & \lfloor \frac{4}{2} \rfloor & \dots & \lfloor \frac{C_{max}}{2} \rfloor \\ \vdots & \vdots & \ddots & \vdots \\ \lfloor \frac{2}{S_{max}} \rfloor & \lfloor \frac{4}{S_{max}} \rfloor & \dots & \lfloor \frac{C_{max}}{S_{max}} \rfloor \end{bmatrix} \begin{bmatrix} x_2 \\ x_4 \\ \vdots \\ x_{C_{max}} \end{bmatrix} = \begin{bmatrix} N_1 \\ N_2 \\ \vdots \\ N_{S_{max}} \end{bmatrix} \quad (6)$$

Because the hardware configuration must consist of discrete, non-negative quantities of groups, we formulate this as a **Mixed-Integer Linear Programming (MILP)** problem. The objective is to minimize the total number of hardware groups, thereby finding the least fragmented architectural configuration that perfectly satisfies all empirical observations. The optimization model is formulated as follows:

$$\begin{aligned} & \underset{\mathbf{x}}{\text{minimize}} && Z = \sum_{c \in C} x_c \\ & \text{subject to} && \sum_{c \in C} \left\lfloor \frac{c}{s} \right\rfloor x_c = N_s, \quad \forall s \in \{1, \dots, S_{max}\} \\ & && x_c \in \mathbb{Z}_{\geq 0}, \quad \forall c \in C \end{aligned} \quad (7)$$

By solving this MILP using standard optimization solvers (e.g., Gurobi, SciPy), we obtain the exact distribution of hardware cluster sizes (x_c), successfully reverse-engineering the hidden interconnect topology.

References

- [1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran

¹²The appendix was not peer-reviewed.

- Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
- [2] DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Erhang Li, Fangqi Zhou, Fangyun Lin, Fucong Dai, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoran Wei, Haowei Zhang, Haowen Luo, Haozhe Ji, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, Jialiang Huang, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jingchang Chen, Jingting Xi-ang, Jingyang Yuan, Jingyuan Cheng, Jinhua Zhu, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexin Huang, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Liang Zhao, Liangsheng Yin, Lihua Guo, Lingxiao Luo, Linwang Ma, Litong Wang, Liyue Zhang, M. S. Di, M. Y. Xu, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Panpan Huang, Peixin Cong, Peiyi Wang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, S. H. Liu, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoifei Cai, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Lu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Songyang Zhou, Tao Ni, Tao Yun, Tian Pei, Tian Ye, Tianyuan Yue, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjun Gao, Wentao Zhang, Xi Gao, Xiangwen Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xing kai Yu, Xingyou Li, Xinyu Yang, Xinyuan Li, Xu Chen, Xuecheng Su, Xuehai Pan, Xuheng Lin, Xuwei Fu, Y. Q. Wang, Yang Zhang, Yanhong Xu, Yanru Ma, Yao Li, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yiliang Xiong, Ying He, Ying Zhou, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yontong Wu, Yu Wu, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yudian Wang, Yue Gong, Yuhuan Wu, Yuheng Zou, Yukun Li, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehua Zhao, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhiyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Zizheng Pan, Zongqing Yao, Bei Feng, Hui Li, J. L. Cai, Jiaqi Ni, Lei Xu, Meng Li, Ning Tian, R. J. Chen, R. L. Jin, S. S. Li, Shuang Zhou, Tianyu Sun, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xinnan Song, Xinyi Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, Dongjie Ji, Jian Liang, Jianzhong Guo, Jin Chen, Leyi Xia, Miaojun Wang, Mingming Li, Peng Zhang, Ruyi Chen, Shangmian Sun, Shaoqing Wu, Shengfeng Ye, T. Wang, W. L. Xiao, Wei An, Xianzu Wang, Xiaowen Sun, Xiaoxiang Wang, Ying Tang, Yukun Zha, Zekai Zhang, Zhe Ju, Zhen Zhang, and Zihua Qu. 2025. DeepSeek-V3.2: Pushing the Frontier of Open Large Language Models. arXiv:2512.02556 [cs.CL] <https://arxiv.org/abs/2512.02556>
- [3] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chuji Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiayi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] <https://arxiv.org/abs/2505.09388>
- [4] 2026. Introducing GPT-5.4: Designed for professional work. <https://openai.com/index/introducing-gpt-5-4/>.
- [5] 2026. System Card: Claude Opus 4.6. <https://www-cdn.anthropic.com/0dd865075ad3132672ee0ab40b05a53f14cf5288.pdf>.
- [6] Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. 2023. Flash-Decoding for long-context inference. <https://pytorch.org/blog/flash-decoding/>.
- [7] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. 2025. FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving. In *Proceedings of Machine Learning and Systems*, M. Zaharia, G. Joshi, and Y. Lin (Eds.), Vol. 7. MLSys. https://proceedings.mlsys.org/paper_files/paper/2025/file/dbf02b21d77409a2db30e56866a8ab3a-Paper-Conference.pdf
- [8] Jiashi Li and Shengyu Liu. 2025. FlashMLA: Efficient Multi-head Latent Attention Kernels. <https://github.com/deepseek-ai/FlashMLA>.
- [9] Rui Ning, Wei Zhang, and Fan Lai. 2026. PackInfer: Compute- and I/O-Efficient Attention for Batched LLM Inference. arXiv:2602.06072 [cs.DC] <https://arxiv.org/abs/2602.06072>
- [10] DeepSeek.ai. 2026. DeepGEMM: clean and efficient FP8 GEMM kernels with fine-grained scaling. <https://github.com/deepseek-ai/DeepGEMM/tree/main>.
- [11] 2026. NVIDIA cuBLAS Documents. <https://docs.nvidia.com/cuda/cublas/index.html>.
- [12] 2026. NVIDIA cuDNN Documents. <https://docs.nvidia.com/cudnn/index.html>.
- [13] Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. 2024. Atom: Low-Bit Quantization for Efficient and Accurate LLM Serving. In *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. De Sa (Eds.), Vol. 6. 196–209. https://proceedings.mlsys.org/paper_files/paper/2024/file/5edeb57c05c81d04beb716ef1d542fe9e-Paper-Conference.pdf
- [14] Aditya K. Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. 2025. *POD-Attention: Unlocking Full Prefill-Decode Overlap for Faster LLM Inference*. Association for Computing Machinery, New York, NY, USA, 897–912. <https://doi.org/10.1145/3676641.3715996>
- [15] Ping Gong, Jiawei Yi, Shengnan Wang, Juncheng Zhang, Zewen Jin, Ouxiang Zhou, Ruibo Liu, Guanbin Xu, Youhui Bai, Bowen Ye, Kun Yuan, Tong Yang, Gong Zhang, Renhai Chen, Feng Wu, and Cheng Li. 2025. HATA: Trainable and Hardware-Efficient Hash-Aware Top-k Attention for Scalable Large Model Inference. In *Findings of the Association for Computational Linguistics: ACL 2025*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 24856–24871. <https://doi.org/10.18653/v1/2025.findings-acl.1275>
- [16] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 16344–16359. https://proceedings.neurips.cc/paper_files/paper/2022/file/67d57c32e20fd0a7a302cb81d36e40d5-Paper-Conference.pdf
- [17] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *The Twelfth International Conference on Learning Representations*. <https://openreview.net/forum?id=mZn2Xyh9Ec>
- [18] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision. In *Advances in Neural*

- Information Processing Systems, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 68658–68685. <https://doi.org/10.52202/079017-2193>
- [19] Ted Zadouri, Markus Hoehnerbach, Jay Shah, Timmy Liu, Vijay Thakkar, and Tri Dao. 2026. FlashAttention-4: Algorithm and Kernel Pipelining Co-Design for Asymmetric Hardware Scaling. arXiv:2603.05451 [cs.CL] <https://arxiv.org/abs/2603.05451>
 - [20] Hongtao Chen, Weiyu Xie, Boxin Zhang, Jingqi Tang, Jiahao Wang, Jianwei Dong, Shaoyuan Chen, Ziwei Yuan, Chen Lin, Chengyu Qiu, Yuening Zhu, Qingliang Ou, Jiaqi Liao, Xianglin Chen, Zhiyuan Ai, Yongwei Wu, and Mingxing Zhang. 2025. KTransformers: Unleashing the Full Potential of CPU/GPU Hybrid Inference for MoE Models. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) (SOSP '25). Association for Computing Machinery, New York, NY, USA, 1014–1029. <https://doi.org/10.1145/3731569.3764843>
 - [21] Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. 2025. TileLang: A Composable Tiled Programming Model for AI Systems. arXiv:2504.17577 [cs.LG] <https://arxiv.org/abs/2504.17577>
 - [22] Yu Cheng, Lei Wang, Yining Shi, Yuqing Xia, Lingxiao Ma, Jilong Xue, Yang Wang, Zhiwen Mo, Feiyang Chen, Fan Yang, Mao Yang, and Zhi Yang. 2025. PipeThreader: Software-Defined Pipelining for Efficient DNN Execution. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 767–783. <https://www.usenix.org/conference/osdi25/presentation/cheng>
 - [23] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. 2023. Welder: Scheduling Deep Learning Memory Access via Tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 701–718. <https://www.usenix.org/conference/osdi23/presentation/shi>
 - [24] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. 2022. ROLLER: Fast and Efficient Tensor Compilation for Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 233–248. <https://www.usenix.org/conference/osdi22/presentation/zhu>
 - [25] Xiao Zhang, Yaoyao Ding, Bolin Sun, Yang Hu, Tatiana Shpeisman, and Gennady Pekhimenko. 2026. Hexcute: A Compiler Framework for Automating Layout Synthesis in GPU Programs. In *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 630–643. <https://doi.org/10.1109/CGO68049.2026.11395194>
 - [26] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Man Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. 2025. Mirage: A Multi-Level Superoptimizer for Tensor Programs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 221–238. <https://www.usenix.org/conference/osdi25/presentation/wu-mengdi>
 - [27] Lei Wang, Yu Cheng, Yining Shi, Zhiwen Mo, Zhengju Tang, Wenhao Xie, Tong Wu, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. 2026. TileLang: Bridge Programmability and Performance in Modern Neural Kernels. In *The Fourteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=Jb1WkNSfUB>
 - [28] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) (OSDI'18). USENIX Association, USA, 579–594.
 - [29] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 863–879. <https://www.usenix.org/conference/osdi20/presentation/zheng>
 - [30] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalam-barkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhres, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) (ASPLOS '24). Association for Computing Machinery, New York, NY, USA, 929–947. <https://doi.org/10.1145/3620665.3640366>
 - [31] Lei Wang, Lingxiao Ma, Shijie Cao, Quanlu Zhang, Jilong Xue, Yining Shi, Ningxin Zheng, Ziming Miao, Fan Yang, Ting Cao, Yuqing Yang, and Mao Yang. 2024. Ladder: Enabling Efficient Low-Precision Deep Learning Computing through Hardware-aware Tensor Transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 307–323. <https://www.usenix.org/conference/osdi24/presentation/wang-lei>
 - [32] Yue Guan, Yuxian Qiu, Jingwen Leng, Fan Yang, Shuo Yu, Yunxin Liu, Yu Feng, Yuhao Zhu, Lidong Zhou, Yun Liang, Chen Zhang, Chao Li, and Minyi Guo. 2024. Amanda: Unified Instrumentation Framework for Deep Neural Networks. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (La Jolla, CA, USA) (ASPLOS '24). Association for Computing Machinery, New York, NY, USA, 1–18. <https://doi.org/10.1145/3617232.3624864>
 - [33] MohammadHossein Olyaiy, Christopher Ng, Alexandra Sasha Fedorova, and Mieszko Lis. 2023. Sunstone: A Scalable and Versatile Scheduler for Mapping Tensor Algebra on Spatial Accelerators. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 259–271. <https://doi.org/10.1109/ISPASS57527.2023.00033>
 - [34] Yaoyao Ding, Bohan Hou, Xiao Zhang, Allan Lin, Tianqi Chen, Cody Hao Yu, Yida Wang, and Gennady Pekhimenko. 2025. Tilus: A Tile-Level GPGPU Programming Language for Low-Precision Computation. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (USA) (ASPLOS '26). Association for Computing Machinery, New York, NY, USA, 281–297. <https://doi.org/10.1145/3760250.3762219>
 - [35] Keren Zhou, Tianle Zhong, Hao Wu, Jihyeong Lee, Yue Guan, Yufei Ding, Corbin Robeck, Yuanwei Fang, Jeff Niu, and Philippe Tillet. 2026. Proton: Towards Multi-level, Adaptive Profiling for Triton. In *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 493–506. <https://doi.org/10.1109/CGO68049.2026.11395207>
 - [36] Ping Gong, Renjie Liu, Zunyao Mao, Zhenkun Cai, Xiao Yan, Cheng Li, Minjie Wang, and Zhuozhao Li. 2023. gSampler: General and Efficient

- GPU-based Graph Sampling for Graph Learning. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) (SOSP '23). Association for Computing Machinery, New York, NY, USA, 562–578. <https://doi.org/10.1145/3600006.3613168>
- [37] Yue Guan, Xinwei Qiang, Zaifeng Pan, Daniels Johnson, Yuanwei Fang, Keren Zhou, Yuke Wang, Wanlu Li, Yufei Ding, and Adnan Aziz. 2025. Mercury: Unlocking Multi-GPU Operator Optimization for LLMs via Remote Memory Scheduling. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) (SOSP '25). Association for Computing Machinery, New York, NY, USA, 1046–1061. <https://doi.org/10.1145/3731569.3764798>
- [38] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) (ASPLOS '20). Association for Computing Machinery, New York, NY, USA, 859–873. <https://doi.org/10.1145/3373376.3378508>
- [39] Siran Liu, Chengxiang Qi, Ying Cao, Chao Yang, Weifang Hu, Xuanhua Shi, Fan Yang, and Mao Yang. 2024. Uncovering Nested Data Parallelism and Data Reuse in DNN Computation with FractalTensor. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) (SOSP '24). Association for Computing Machinery, New York, NY, USA, 160–177. <https://doi.org/10.1145/3694715.3695961>
- [40] OpenAI. 2026. Development repository for the Triton language and compiler. <https://github.com/triton-lang/triton>.
- [41] Hongzheng Chen, Bin Fan, Alexander Collins, Bastian Hagedorn, Evghenii Gaburov, Masahiro Masuda, Matthew Brookhart, Chris Sullivan, Jason Knight, Zhiru Zhang, and Vinod Grover. 2025. Tawa: Automatic Warp Specialization for Modern GPUs with Asynchronous References. arXiv:2510.14719 [cs.LG] <https://arxiv.org/abs/2510.14719>
- [42] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 881–897. <https://www.usenix.org/conference/osdi20/presentation/ma>
- [43] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) (SOSP '23). Association for Computing Machinery, New York, NY, USA, 611–626. <https://doi.org/10.1145/3600006.3613165>
- [44] Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Zhao, Andrew M Dai, zhifeng Chen, Quoc V Le, and James Laudon. 2022. Mixture-of-Experts with Expert Choice Routing. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 7103–7114. https://proceedings.neurips.cc/paper_files/paper/2022/file/2f0e0ecd787b432c1d36f3de9800728eb-Paper-Conference.pdf
- [45] Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Yuxing Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. 2025. Native Sparse Attention: Hardware-Aligned and Natively Trainable Sparse Attention. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 23078–23097. <https://doi.org/10.18653/v1/2025.acl-long.1126>
- [46] 2025. Huggingface ShareGPT90K datasets. <https://huggingface.co/datasets/liyucheng/ShareGPT90K>.
- [47] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. <https://doi.org/10.1145/79173.79181>
- [48] 2026. vLLM Triton Decoding Attention Kernel (v0.17.1). https://github.com/vllm-project/vllm/blob/v0.17.1/vllm/v1/attention/ops/triton_decode_attention.py.
- [49] 2026. SGLang Triton Decoding Attention Kernel (v0.5.9). https://github.com/sgl-project/sglang/blob/v0.5.9/python/sglang/srt/layers/attention/triton_ops/decode_attention.py.
- [50] 2026. vLLM Paged Attention Attention CUDA Kernel (v0.17.1). https://github.com/vllm-project/vllm/blob/v0.17.1/csrc/attention/paged_attention_v2.cu.
- [51] NVIDIA. 2026. CUDA C Programming Guide. <https://docs.nvidia.com/cuda/cuda-c-programming-guide>.
- [52] NVIDIA. 2022. NVIDIA H100 Tensor Core GPU Architecture. <https://resources.nvidia.com/en-us-data-center-overview-mc/en-us-data-center-overview/gtc22-whitepaper-hopper>.
- [53] 2026. Tensor Memory Accelerator (TMA). <https://nvidia.github.io/cccl/unstable/cccl/tma.html>.
- [54] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 62557–62583. <https://doi.org/10.52202/079017-2000>
- [55] 2025. vLLM: A high-throughput and memory-efficient inference and serving engine for LLMs. <https://github.com/vllm-project/vllm>.
- [56] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 117–134. <https://www.usenix.org/conference/osdi24/presentation/agrawal>
- [57] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. 2023. SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills. arXiv:2308.16369 [cs.LG] <https://arxiv.org/abs/2308.16369>
- [58] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaoqun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiusi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhua Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Liu,

- Xin Xie, Xingkai Yu, Xinnan Song, Xinyi Zhou, Xinyu Yang, Xuan Lu, Xuecheng Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Zheng, Yichao Zhang, Yiliang Xiong, Yilong Zhao, Ying He, Ying Tang, Yishi Piao, Yixin Dong, Yixuan Tan, Yiyuan Liu, Yongji Wang, Yongqiang Guo, Yuchen Zhu, Yudian Wang, Yuheng Zou, Yukun Zha, Yunxian Ma, Yuting Yan, Yuxiang You, Yuxuan Liu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhewen Hao, Zhihong Shao, Zhiniu Wen, Zhipeng Xu, Zhongyu Zhang, Zhuoshu Li, Zihan Wang, Zihui Gu, Zilin Li, and Ziwei Xie. 2024. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. arXiv:2405.04434 [cs.CL] <https://arxiv.org/abs/2405.04434>
- [59] Biao Zhang and Rico Sennrich. 2019. Root Mean Square Layer Normalization. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/1e8a19426224ca89e83cef47f1e7f53b-Paper.pdf
- [60] Barry Payne Welford. 1962. Note on a method for calculating corrected sums of squares and products. *Technometrics* 4, 3 (1962), 419–420.
- [61] Erich Schubert and Michael Gertz. 2018. Numerically stable parallel computation of (co-)variance. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management (Bozen-Bolzano, Italy) (SSDBM '18)*. Association for Computing Machinery, New York, NY, USA, Article 10, 12 pages. <https://doi.org/10.1145/3221269.3223036>
- [62] FlashInfer. 2026. FlashInfer CUTLASS fused MoE kernels. https://github.com/flashinfer-ai/flashinfer/blob/4b969c9363e0d32a33f8fccd9ef5a5fec51cd9f/csrc/fused_moe/cutlass_backend/cutlass_fused_moe_kernels.cuh
- [63] NVIDIA. 2026. CUDA C Programming Guide: Distributed Shared Memory. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#distributed-shared-memory>.
- [64] NVIDIA. 2025. Hopper Tuning Guide: Thread Block Clusters. <https://docs.nvidia.com/cuda/hopper-tuning-guide/index.html#thread-block-clusters>.
- [65] Jack Choquette. 2022. Nvidia Hopper GPU: Scaling Performance. In *2022 IEEE Hot Chips 34 Symposium (HCS)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–46. <https://doi.org/10.1109/HCS55958.2022.9895592>
- [66] NVIDIA. 2022. NVIDIA H100 Tensor Core GPU Architecture. <https://resources.nvidia.com/en-us-data-center-overview-mc/en-us-data-center-overview/gtc22-whitepaper-hopper>.
- [67] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. 2024. Benchmarking and Dissecting the Nvidia Hopper GPU Architecture. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 656–667. <https://doi.org/10.1109/IPDPS57955.2024.00064>
- [68] Kuntai Du, Bowen Wang, Chen Zhang, Yiming Cheng, Qing Lan, Hejian Sang, Yihua Cheng, Jiayi Yao, Xiaoxuan Liu, Yifan Qiao, Ion Stoica, and Junchen Jiang. 2025. PrefillOnly: An Inference Engine for Prefill-only Workloads in Large Language Model Applications. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (Lotte Hotel World, Seoul, Republic of Korea) (SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 399–414. <https://doi.org/10.1145/3731569.3764834>
- [69] vLLM team. 2025. Automatic Prefix Caching. https://docs.vllm.ai/en/latest/features/automatic_prefix_caching/.
- [70] Dongsheng Yang, Austin Li, Kai Li, and Wyatt Lloyd. [n. d.]. Learned Prefix Caching for Efficient LLM Inference. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*. <https://openreview.net/pdf?id=Vj48eXaQDM>
- [71] Yifan Yu, Yu Gan, Nikhil Sarda, Lillian Tsai, Jiaming Shen, Yanqi Zhou, Arvind Krishnamurthy, Fan Lai, Hank Levy, and David Culler. 2025. IC-Cache: Efficient Large Language Model Serving via In-context Caching. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (Lotte Hotel World, Seoul, Republic of Korea) (SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 375–398. <https://doi.org/10.1145/3731569.3764829>
- [72] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 15–28. <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia>
- [73] Jeffrey Dean, Sanjay Ghemawat, et al. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI 04)*. USENIX Association, San Francisco, CA. https://www.usenix.org/legacy/publications/library/proceedings/osdi04/tech/full_papers/dean/dean_html/index.html
- [74] Jingkai He, Yunpeng Dong, Dong Du, Mo Zou, Zhitai Yu, Yuxin Ren, Ning Jia, Yubin Xia, and Haibo Chen. 2025. How to Copy Memory? Coordinated Asynchronous Copy as a First-Class OS Service. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (Lotte Hotel World, Seoul, Republic of Korea) (SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 1062–1081. <https://doi.org/10.1145/3731569.3764800>
- [75] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. 2019. Dissecting the Graphcore IPU Architecture via Microbenchmarking. arXiv:1912.03413 [cs.DC] <https://arxiv.org/abs/1912.03413>
- [76] Sean Lie. 2024. Inside the Cerebras Wafer-Scale Cluster. *IEEE Micro* 44, 3 (2024), 49–57. <https://doi.org/10.1109/MM.2024.3386628>
- [77] Cerebras. 2025. The Cerebras Wafer-Scale Engine. <https://www.cerebras.ai/chip>.
- [78] GraphCore. 2025. The GraphCore IPU Processors. <https://www.graphcore.ai/products/ipu>.
- [79] Zheng Xu, Dehao Kong, Jiaxin Liu, Jinxi Li, Jingxiang Hou, Xu Dai, Chao Li, Shaojun Wei, Yang Hu, and Shouyi Yin. 2025. WSC-LLM: Efficient LLM Service and Architecture Co-exploration for Wafer-scale Chips. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, New York, NY, USA, 1–17. <https://doi.org/10.1145/3695053.3731101>
- [80] Congjie He, Yeqi Huang, Pei Mu, Ziming Miao, Jilong Xue, Lingxiao Ma, Fan Yang, and Luo Mai. 2025. WaferLLM: Large Language Model Inference at Wafer Scale. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, USA, 257–273. <https://www.usenix.org/conference/osdi25/presentation/he>
- [81] Tianhao Zhu, Dahu Feng, Erhu Feng, and Yubin Xia. 2025. From Principles to Practice: A Systematic Study of LLM Serving on Multi-core NPUs. arXiv:2510.05632 [cs.AR] <https://arxiv.org/abs/2510.05632>
- [82] Yiqi Liu, Yuqi Xue, Yu Cheng, Lingxiao Ma, Ziming Miao, Jilong Xue, and Jian Huang. 2024. Scaling Deep Learning Computation over the Inter-Core Connected Intelligence Processor with T10. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (Austin, TX, USA) (SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 505–521. <https://doi.org/10.1145/3694715.3695955>
- [83] Yiqi Liu, Yuqi Xue, Noelle Crawford, Jilong Xue, and Jian Huang. 2025. *Elk: Exploring the Efficiency of Inter-core Connected AI Chips with Deep Learning Compiler Techniques*. Association for Computing Machinery, New York, NY, USA, 1284–1299. <https://doi.org/10.1145/3725843.3756064>

- [84] 2026. NVIDIA Groq 3 LPX. <https://www.nvidia.com/en-us/data-center/lpx>.
- [85] Raghu Prabhakar, Ram Sivaramakrishnan, Darshan Gandhi, Yun Du, Mingran Wang, Xiangyu Song, Kejie Zhang, Tianren Gao, Angela Wang, Xiaoyan Li, Yongning Sheng, Joshua Brot, Denis Sokolov, Apurv Vivek, Calvin Leung, Arjun Sabnis, Jiayu Bai, Tuowen Zhao, Mark Gottscho, David Jackson, Mark Luttrell, Manish K. Shah, Zhengyu Chen, Kaizhao Liang, Swayambhoo Jain, Urmish Thakker, Dawei Huang, Sumti Jairath, Kevin J. Brown, and Kunle Olukotun. 2024. SambaNova SN40L: Scaling the AI Memory Wall with Dataflow and Composition of Experts. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1353–1366. <https://doi.org/10.1109/MICRO61859.2024.00100>
- [86] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. *SIGARCH Comput. Archit. News* 42, 1 (Feb. 2014), 269–284. <https://doi.org/10.1145/2654822.2541967>
- [87] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 609–622. <https://doi.org/10.1109/MICRO.2014.58>
- [88] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (Orlando, FL, USA) (ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 82, 14 pages. <https://doi.org/10.1145/3579371.3589350>
- [89] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, and Derrick Weathersby. 1994. Efficient algorithms for all-to-all communications in multi-port message-passing systems. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures* (Cape May, New Jersey, USA) (SPAA '94). Association for Computing Machinery, New York, NY, USA, 298–309. <https://doi.org/10.1145/181014.181756>
- [90] Sylvain Jeagey. 2025. PAT: a new algorithm for all-gather and reduce-scatter operations at scale. arXiv:2506.20252 [cs.DC] <https://arxiv.org/abs/2506.20252>
- [91] NVIDIA. 2025. (CUDA Programming Guide) CUDA Graphs. <https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cuda-graphs.html>.
- [92] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) (SC '21). Association for Computing Machinery, New York, NY, USA, Article 58, 15 pages. <https://doi.org/10.1145/3458817.3476209>
- [93] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL] <https://arxiv.org/abs/1909.08053>
- [94] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (Virtual Event, CA, USA) (KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. <https://doi.org/10.1145/3394486.3406703>
- [95] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Mylène Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. arXiv:2304.11277 [cs.DC] <https://arxiv.org/abs/2304.11277>
- [96] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory optimizations Toward Training Trillion Parameter Models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16. <https://doi.org/10.1109/SC41405.2020.00024>
- [97] Alibaba Cloud. 2025. Overview of the LLM inference engine (BladeLLM) (fully asynchronous decoding scheduling architecture). <https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cuda-graphs.html>.
- [98] SGLang. 2025. (SGLang) Overlap Scheduling and Async Execution. <https://deepwiki.com/sgl-project/sglang-jax/6.2-overlap-scheduling-and-async-execution>.
- [99] Chenggang Zhao, Shangyan Zhou, Liyue Zhang, Chengqi Deng, Zhean Xu, Yuxuan Liu, Kuai Yu, Jiashi Li, and Liang Zhao. 2025. DeepEP: an efficient expert-parallel communication library. <https://github.com/deepseek-ai/DeepEP>.
- [100] Zhuoran Zhu, Chunyang Zhu, Hao Lin, Xu Fu, Yiming Zhou, Quanlu Zhang, Zhenhua Li, Feng Qian, Chao Yu, Boxun Li, Guohao Dai, and Yu Wang. 2025. FUSCO: High-Performance Distributed Data Shuffling via Transformation-Communication Fusion. arXiv:2512.22036 [cs.DC] <https://arxiv.org/abs/2512.22036>
- [101] DeepSeek.ai. 2026. Expert Parallelism Load Balancer (EPLB). <https://github.com/deepseek-ai/EPLB>.
- [102] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. arXiv:2402.03300 [cs.CL] <https://arxiv.org/abs/2402.03300>
- [103] Qiyang Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, YuYue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, LingJun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. 2025. DAPO: An Open-Source LLM Reinforcement Learning System at Scale. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*. <https://openreview.net/forum?id=2a36EMSSTp>
- [104] Jingkai He, Tianjian Li, Erhu Feng, Dong Du, Qian Liu, Tao Liu, Yubin Xia, and Haibo Chen. 2026. History Doesn't Repeat Itself but Rollouts Rhyme: Accelerating Reinforcement Learning with RhymeRL. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (USA) (ASPLOS '26)*. Association for Computing Machinery, New York, NY, USA, 929–945. <https://doi.org/10.1145/3779212.3790172>
- [105] 2025. Huggingface Dolphin-r1 datasets. <https://huggingface.co/datasets/cognitivecomputations/dolphin-r1>.
- [106] 2025. Huggingface OpenR1-Math-220k datasets. <https://huggingface.co/datasets/open-r1/OpenR1-Math-220k>.
- [107] 2025. Huggingface OpenThoughts-114k-Code_decontaminated datasets. https://huggingface.co/datasets/open-r1/OpenThoughts-114k-Code_decontaminated.
- [108] Tile.ai. 2026. TileOPs (TOP): Operator Library for LLMs Built on TileLang. <https://github.com/tile-ai/TileOPs>.
- [109] Tile.ai. 2026. TileLang (Repository): Domain-specific language designed to streamline the development of high-performance GPU/CPU/Accelerators kernels. <https://github.com/tile-ai/tilelang>.

- [110] 2026. A collection of GPT system prompts and various prompt injection/leaking knowledge. https://github.com/LouisShark/chatgpt_system_prompt/.
- [111] 2026. System Prompt of OpenClaw. <https://docs.openclaw.ai/concepts/system-prompt>.
- [112] 2026. vLLM Automatic Prefix Caching. https://docs.vllm.ai/en/v0.17.0/design/prefix_caching.
- [113] 2026. SGLang HiCache System Design and Optimization. https://docs.sglang.io/advanced_features/hicache_design.html.
- [114] Chen Zhang, Kuntai Du, Shu Liu, Woosuk Kwon, Xiangxi Mo, Yufeng Wang, Xiaoxuan Liu, Kaichao You, Zhuohan Li, Mingsheng Long, Jidong Zhai, Joseph Gonzalez, and Ion Stoica. 2025. Jenga: Effective Memory Management for Serving LLM with Heterogeneity. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) (SOSP '25). Association for Computing Machinery, New York, NY, USA, 446–461. <https://doi.org/10.1145/3731569.3764823>
- [115] Xinhao Cheng, Zhihao Zhang, Yu Zhou, Jianan Ji, Jincheng Jiang, Zepeng Zhao, Ziruo Xiao, Zihao Ye, Yingyi Huang, Ruihang Lai, Hongyi Jin, Bohan Hou, Mengdi Wu, Yixin Dong, Anthony Yip, Zihao Ye, Songting Wang, Wenqin Yang, Xupeng Miao, Tianqi Chen, and Zhihao Jia. 2025. Mirage Persistent Kernel: A Compiler and Runtime for Mega-Kernelizing Tensor Programs. arXiv:2512.22219 [cs.DC] <https://arxiv.org/abs/2512.22219>
- [116] DeepSeek. 2026. [DeepGEMM Public release 26/04] Introducing Mega MoE, FP4 Indexer and other features/fixes. <https://github.com/deepseek-ai/DeepGEMM/pull/304>
- [117] Benjamin Spector, Jordan Juravsky, Stuart Sul, Owen Dugan, Dylan Lim, Dan Fu, Simran Arora, and Chris Ré. 2025. Designing a Low-Latency Megakernel for Llama-1B. <https://hazyresearch.stanford.edu/blog/2025-05-27-no-bubbles>.
- [118] 2026. [DeepGEMM Pull Request] feat: MegaMOE adaptation for SM90. <https://github.com/deepseek-ai/DeepGEMM/pull/323>
- [119] Hongyi Jin, Bohan Hou, Guanjie Wang, Ruihang Lai, Jinqi Chen, Zihao Ye, Yaxing Cai, Yixin Dong, Xinhao Cheng, Zhihao Zhang, Yilong Zhao, Yingyi Huang, Lijie Yang, Jincheng Jiang, Gabriele Oliaro, Jianan Ji, Xupeng Miao, Vinod Grover, Todd C. Mowry, Zhihao Jia, and Tianqi Chen. 2026. Event Tensor: A Unified Abstraction for Compiling Dynamic Megakernel. In *Proceedings of Machine Learning and Systems*, A. Chowdhery and Z. Jia (Eds.), Vol. 8. MLSys, 1917–1933. https://proceedings.mlsys.org/paper_files/paper/2026/file/53d3f45797970d323bd8a0d379c525aa-Paper-Conference.pdf
- [120] Size Zheng, Wenlei Bao, Qi Hou, Xuegui Zheng, Jin Fang, Chenhui Huang, Tianqi Li, Haojie Duanmu, Renze Chen, Ruifan Xu, Yifan Guo, Ningxin Zheng, Ziheng Jiang, Xinyi Di, Dongyang Wang, Jianxi Ye, Haibin Lin, Li-Wen Chang, Liqiang Lu, Yun Liang, Jidong Zhai, and Xin Liu. 2025. Triton-distributed: Programming Overlapping Kernels on Distributed AI Systems with the Triton Compiler. arXiv:2504.19442 [cs.DC] <https://arxiv.org/abs/2504.19442>
- [121] Size Zheng, Jin Fang, Xuegui Zheng, Qi Hou, Wenlei Bao, Ningxin Zheng, Ziheng Jiang, Dongyang Wang, Jianxi Ye, Haibin Lin, Li-Wen Chang, and Xin Liu. 2025. TileLink: Generating Efficient Compute-Communication Overlapping Kernels using Tile-Centric Primitives. arXiv:2503.20313 [cs.DC] <https://arxiv.org/abs/2503.20313>
- [122] SGLang. 2025. Mini-SGLang: Efficient Inference Engine in a Nutshell. <https://lmsys.org/blog/2025-12-17-minisgl/>
- [123] NVIDIA. 2026. NVIDIA TensorRT-LLM. <https://docs.nvidia.com/tensorrt-llm/index.html>.
- [124] SGLang. 2026. sglang-kernel (prior sgl-kernel). <https://github.com/sgl-project/sglang/tree/main/sgl-kernel>.
- [125] 2026. Blackwell Cluster Launch Control. https://docs.nvidia.com/cutlass/latest/media/docs/cpp/blackwell_cluster_launch_control.html
- [126] Michael Davies, Neal Crago, Karthikeyan Sankaralingam, and Stephen W. Keckler. 2025. Kitsune: Enabling Dataflow Execution on GPUs. arXiv:2502.18403 [cs.AR] <https://arxiv.org/abs/2502.18403>
- [127] Xinhao Luo, Zihan Liu, Yangjie Zhou, Shihan Fang, Ziyu Huang, Yu Feng, Chen Zhang, Shixuan Sun, Zhenzhe Zheng, Jingwen Leng, and Minyi Guo. 2025. ClusterFusion: Expanding Operator Fusion Scope for LLM Inference via Cluster-Level Collective Primitive. arXiv:2508.18850 [cs.DC] <https://arxiv.org/abs/2508.18850>
- [128] 2025. QuACK: A Quirky Assortment of CuTe Kernels. <https://github.com/Dao-AILab/quack>
- [129] Weihao Cui, Zhenhua Han, Lingji Ouyang, Yichuan Wang, Ningxin Zheng, Lingxiao Ma, Yuqing Yang, Fan Yang, Jilong Xue, Lili Qiu, Lidong Zhou, Quan Chen, Haisheng Tan, and Minyi Guo. 2023. Optimizing Dynamic Neural Networks with Brainstorm. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 797–815. <https://www.usenix.org/conference/osdi23/presentation/cui>
- [130] Ningxin Zheng, Huiqiang Jiang, Quanlu Zhang, Zhenhua Han, Lingxiao Ma, Yuqing Yang, Fan Yang, Chengruidong Zhang, Lili Qiu, Mao Yang, and Lidong Zhou. 2023. PIT: Optimization of Dynamic Sparse Deep Learning Models via Permutation Invariant Transformation. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) (SOSP '23). Association for Computing Machinery, New York, NY, USA, 331–347. <https://doi.org/10.1145/3600006.3613139>
- [131] Pedro F. Silvestre and Peter Pietzuch. 2025. Tempo: Compiled Dynamic Deep Learning with Symbolic Dependence Graphs. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) (SOSP '25). Association for Computing Machinery, New York, NY, USA, 572–588. <https://doi.org/10.1145/3731569.3764840>