FluidGPU: Fine-Grained Kernel Disaggregation for Large Model Inference on Heterogeneous GPUs

Tiancheng Hu[1*], Jin Qin[23], Lei Chen[23], Junhao Hu[1], Yuzheng Wang[1*], Mingxing Zhang[4], Chunwei Xia[5], Yizhou Shan[‡], Huimin Cui[23], Ting Cao[4], Zheng Wang[5], Tao Xie[1], Chenxi Wang[23]
[1]SCS, Peking University, Beijing, China
[2]SKLP, Institute of Computing Technology, CAS
[3]University of Chinese Academy of Sciences, Beijing, China
[4]Tsinghua University, Beijing, China
[5]University of Leeds, West Yorkshire, England

***Abstract*—Disaggregation assigns components of an AI workload to different GPU types, providing a way to utilize modern heterogeneous GPU clusters. However, existing solutions operate at a coarse granularity and are tightly coupled to specific model architectures, leaving substantial performance untapped. This paper presents FluidGPU, the first kernel disaggregation system designed to improve the performance and cost efficiency of large-model inference on heterogeneous GPUs. Our key insight is that kernels within a single application exhibit diverse resource demands, making kernel-level disaggregation the most suitable granularity for aligning computation with hardware capabilities. FluidGPU combines offline analysis with online adaptation. It extracts precise inter-kernel dependencies from PTX to ensure correctness, overlaps communication with computation through pipelined execution, and uses workload-aware scheduling with lightweight runtime adaptation. Our evaluation across five heterogeneous GPUs and four model architectures, scaling up to 16 GPUs, shows that FluidGPU achieves up to 2.3× the serving throughput of existing disaggregation methods while generalizing to model architectures where prior approaches do not apply. Its cost efficiency, measured as throughput per dollar, is up to 1.6× that of these methods. Our evaluation further shows that FluidGPU on a heterogeneous GPU pair can, in some cases, deliver more than twice the throughput of the high-end GPU alone while costing less than two high-end GPUs.**

I. Introduction

Modern GPU data centers are increasingly heterogeneous, integrating a wide range of GPU types and generations. This heterogeneity stems from the mismatch between GPU release cycles and retirement schedules, driven by high costs and limited supply [1], [2]. For example, Google Cloud offers both high-end GPUs (*e.g.*, A100 and H100) and general-purpose GPUs (*e.g.*, L4 and RTX Pro 6000) [3], and AWS shows a similar trend [4]. At the same time, the rapid growth of AI has dramatically increased demand for GPU resources [5]. This demand is further amplified by the emergence of agentic AI applications, where a single user request may trigger multiple sequential model invocations, significantly increasing inference workload and cost [6], [7].

In this context, a primary challenge for cloud providers is maximizing the serving performance of heterogeneous GPU clusters while improving cost efficiency (Perf/$) [8], [9]. However, existing scheduling mechanisms fail to fully exploit the architectural diversity of heterogeneous GPUs, leaving substantial performance and cost-efficiency gains untapped.

Existing approaches. Recent studies [10], [11], [12], [13], [9] use disaggregation to exploit GPU heterogeneity. These methods partition AI workloads according to their computational characteristics and route them to the most suitable GPU. This strategy is particularly common in Transformer-based LLM inference, where distinct phases or blocks have different resource demands. For example, prefill-decode (PD) disaggregation assigns the compute-intensive prefill phase to high-TFLOPS GPUs, while offloading the memory-bound decode phase to GPUs with higher memory bandwidth [14]. Similarly, block-level disaggregation schemes, such as Attention-FFN disaggregation [12], aim to better align computation with hardware specialization.

Despite their effectiveness, existing disaggregation methods suffer from two fundamental limitations. First, they operate at a coarse granularity (*e.g.*, phases or computation blocks), leaving substantial performance untapped. For example, even within a single FFN block, different kernels may exhibit markedly different resource demands, which coarse-grained disaggregation fails to capture. Second, these approaches are often application-specific and tightly coupled to particular model architectures, limiting their generality across model types.

Key insight. We observe that the effective utilization of heterogeneous GPUs requires aligning *hardware heterogeneity with application heterogeneity* at *the right granularity*. On the hardware side, modern clusters consist of diverse GPUs with complementary strengths, including differences in compute capacity, memory bandwidth, and cost efficiency (§II-A). On the application side, the GPU kernel is the basic user-level scheduling unit and therefore a natural granularity for disaggregation. Within a kernel, finer granularities, such as thread blocks, are largely homogeneous and do not benefit from further partitioning. In contrast, coarser granularities,

[*] Part of the work was done during the internship at SKLP, Institute of Computing Technology, CAS.
‡ Independent Researcher

SC26, November 15-20, 2026, Chicago, Illinois, USA
979-8-3195-4789-7/26/$31.00 ©2026 IEEE

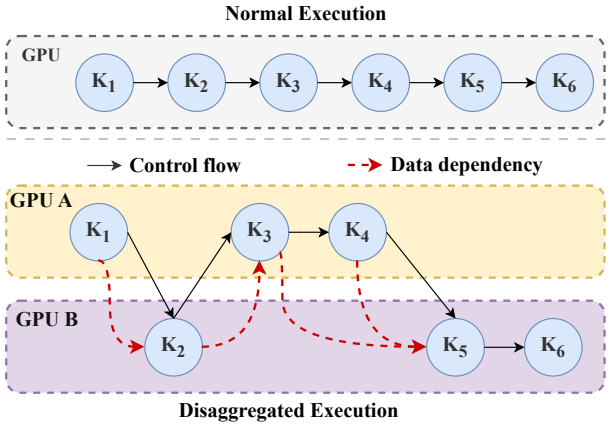

Fig. 1: Simplified example of kernel disaggregation.

	A100	H100	B200	L40s	RTX Pro 6000
CUDA core (TFLOPS)	19.5	67	80	91.6	120
Tensor core (TFLOPS)	312	989	∼2500	366.5	∼500
L2 cache (MB)	40	50	126	96	126
Memory (GB)	80	80	192	48	96
Bw. (GB/s)	1935	3350	∼8000	864	1597
Price	1.5	2.9	5.0	1	1.2

TABLE I: Performance metrics and relative cost of different GPU architectures. CUDA-core and Tensor-core performance are reported in FP32 and BF16 TFLOPS, respectively. The prices are derived mainly from Google Cloud's standard instance rates, normalized by the L40s [18].

such as execution phases, fail to capture important variations in resource demand (§II-C). Moreover, kernels within a single application exhibit diverse resource behavior, ranging from compute-bound to memory-bound, which indicates strong heterogeneity at the kernel level (§II-B).

Our approach. We present FluidGPU, the first system to exploit heterogeneous GPUs through fine-grained kernel disaggregation. Given a set of heterogeneous GPU resources, FluidGPU aligns GPU heterogeneity with kernel heterogeneity to maximize overall performance and cost efficiency. Figure 1 shows a simplified example of kernel disaggregation. Realizing this approach requires addressing three key challenges:

The *first challenge* is preserving functional correctness under kernel disaggregation. Distributing kernel execution across GPUs makes data dependencies that were implicit in local execution explicit. A kernel scheduled on one GPU may depend on data produced by a preceding kernel on another GPU. To prevent data hazards, the system must analyze the memory addresses and sizes of buffers accessed by each kernel. This is difficult in modern AI frameworks, which rely on opaque kernels such as user-defined or JIT-compiled kernels whose semantics are unavailable at compile time. We therefore design a PTX-level kernel analyzer [15]. By instrumenting memory-access instructions, the analyzer captures precise buffer boundaries and extracts read-after-write (RAW) dependencies, thereby preserving correctness (§III-A).

The *second challenge* is mitigating communication overhead and GPU underutilization. Kernel disaggregation introduces inter-GPU transfers for cross-device data dependencies. When dependent kernels are placed on different GPUs, downstream kernels must wait for upstream completion, leaving GPUs idle. To address this issue, FluidGPU employs a pipelined request-processing mechanism that executes multiple requests concurrently across heterogeneous GPUs. When a GPU stalls waiting for remote data, the GPU worker schedules ready kernels from other requests, overlapping communication with computation and maintaining high utilization (§III-C).

The *third challenge* is designing an optimized scheduling policy. An unsuitable scheduling policy may incur excessive inter-GPU communication and poor load balance, causing system-level overheads to outweigh computation gains. To address this issue, FluidGPU derives workload-aware scheduling policies for different optimization objectives. For offline workloads, which prioritize system throughput, FluidGPU formulates scheduling as a mixed-integer linear programming (MILP) problem that jointly models kernel characteristics, compute throughput, communication overhead, and GPU load balance. For latency-sensitive online workloads (*e.g.*, LLM serving), FluidGPU employs a latency-oriented scheduling policy (§III-B). A lightweight online monitor dynamically adjusts these policies as request loads change (§III-D).

We integrate FluidGPU into popular vLLM [16] and PyTorch [17] frameworks, and evaluate FluidGPU across five heterogeneous GPUs and four model architectures, scaling up to 16 GPUs. Compared with existing disaggregation methods, FluidGPU achieves up to 2.3× their serving throughput. Its cost efficiency, measured as throughput per dollar, reaches up to 1.6× that of these methods. Furthermore, FluidGPU sustains a 1.3× higher request rate under the same SLO. Our evaluation further shows that FluidGPU on a heterogeneous GPU pair can, in some cases, deliver more than twice the throughput of the high-end GPU alone while costing less than two high-end GPUs. In summary, this paper presents the following contributions:

- We survey GPU architectural heterogeneity across modern data center GPUs and conduct a systematic study of kernel-level heterogeneity across diverse AI workloads.
- We show that existing disaggregation methods are too coarse-grained to capture kernel-level heterogeneity, fundamentally limiting their ability to exploit heterogeneous GPUs.
- We design and implement FluidGPU, the first kernel disaggregation system for heterogeneous GPUs.
- We conduct a comprehensive evaluation demonstrating that FluidGPU consistently outperforms state-of-the-art disaggregation methods in both throughput and cost efficiency, while generalizing to model architectures where prior approaches are inapplicable.

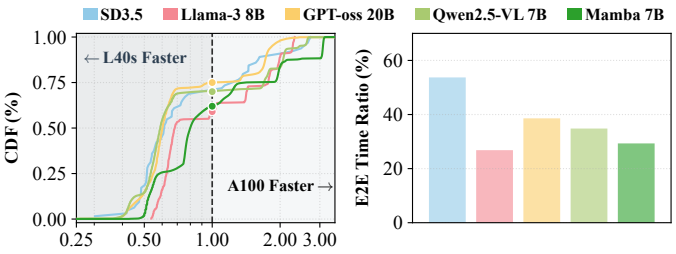

Fig. 2: Kernel heterogeneity between A100 and L40s. (a) CDF of kernel execution-time ratios (L40s/A100) based on kernel count, and (b) share of A100 end-to-end (E2E) execution time attributable to L40s-faster kernels.

II. Background & Motivation

A. GPU Heterogeneity

Modern GPU clusters are inherently heterogeneous. GPU models make different trade-offs across key architectural dimensions, including compute throughput, memory bandwidth, memory capacity, and cost efficiency. As shown in Table I, no single GPU dominates across all dimensions. For example, compared to the A100, the L40s delivers 4.7× higher CUDA-core throughput and 1.2× higher Tensor-core throughput, but provides only 45% of the A100's memory bandwidth. Similarly, while the B200 offers near-leading performance across most hardware metrics, GPUs such as the L40s and RTX Pro 6000 provide up to 7.4× the compute throughput per dollar of high-end GPUs. These complementary strengths make heterogeneous GPU deployments both practical and economically attractive.

This heterogeneity is a sustained production trend rather than a transient condition. GPU vendors continuously release devices that make different trade-offs among multiple dimensions, so future deployments are also likely to remain heterogeneous rather than converging to a single dominant GPU type. This trend is further reinforced by platforms such as NVIDIA MGX [19], which have been adopted by multiple vendors, including Supermicro [20] and Gigabyte [21], and explicitly support heterogeneous GPU configurations within a single server. Together, these trends motivate systems that can exploit heterogeneous GPUs effectively.

Comparable architectural diversity exists in other ecosystems, including AMD. Although broader heterogeneity across vendors or accelerator types (*e.g.*, GPUs and NPUs) is also possible, it introduces additional challenges, including fragmented software stacks [22] and the lack of a unified high-speed interconnect [23]. Given NVIDIA's dominant market position and mature software ecosystem, this paper focuses on heterogeneous GPUs within the NVIDIA platform.

B. Intrinsic Kernel Heterogeneity

FluidGPU is designed to optimize GPU kernels that exhibit intrinsic heterogeneity in modern AI workloads. To quantify this heterogeneity, we profile five representative models spanning LLMs (Llama-3 8B and GPT-oss 20B), SSMs (Mamba 7B), MLLMs (Qwen2.5-VL 7B), and diffusion models (Stable Diffusion 3.5 (SD3.5)). We measure individual-kernel execution time on two heterogeneous GPUs, A100 and L40s.

Figure 2 illustrates kernel heterogeneity from two perspectives. Figure 2(a) shows the count-based CDF of kernel execution-time ratios (L40s/A100). On average, 67% of kernels execute faster on L40s across models and tasks. However, kernel counts alone do not reflect their end-to-end impact. Therefore, Figure 2(b) presents a time-weighted analysis, measuring the fraction of total A100 execution time contributed by kernels that run faster on L40s. These kernels account for 36% of total execution time on average and up to 53% for diffusion models. These results show substantial opportunities to accelerate large-model inference on heterogeneous GPUs through fine-grained kernel disaggregation.

C. Limitations of Coarse-Grained Disaggregation

Existing disaggregation approaches operate at either the phase level [10], [9] or the block level [12]. Both granularities leave substantial performance untapped.

Prefill-decode disaggregation [10], [9] assigns the entire prefill phase to one GPU and the decode phase to another. However, as shown in Figure 3(a), kernels within the same phase exhibit diverse GPU preferences: 45% of prefill kernels and 57% of decode kernels run faster on L40s than on A100. As a result, any phase-level disaggregation inevitably misplaces a large fraction of kernels.

To understand the root causes of kernel heterogeneity, we examine two representative prefill kernels. `cublasGemv` [24] is a widely used matrix–vector multiplication kernel that is strongly memory-bandwidth-bound, with an operational intensity of approximately 1 FLOP/byte. According to the roofline model [25], its performance is primarily determined by memory bandwidth. A100's HBM provides approximately twice the bandwidth of L40s, yielding a 1.9× speedup. In contrast, FlashAttention [26] is compute-bound with high operational intensity that scales with sequence length. Its tile-based design maximizes data reuse in shared memory and the L2 cache, substantially reducing HBM traffic. As a result, performance is dominated by SM-level execution efficiency rather than memory bandwidth. Benefiting from a larger L2 cache, higher core frequency, and more efficient tensor cores, L40s achieves a speedup of up to 2.1× over A100.

Attention-FFN disaggregation [12] operates at the block level, mapping attention and FFN to separate GPUs. However, as shown in Figure 3(b), kernels within the same block still exhibit divergent GPU preferences: 71% of attention kernels run faster on L40s, and 33% of FFN kernels run faster on L40s. Therefore, block-level disaggregation still misplaces a significant fraction of kernels, leading to suboptimal performance. These limitations motivate FluidGPU, which disaggregates at the kernel level to better exploit GPU heterogeneity.

III. Design

FluidGPU is the first system that disaggregates kernel execution across heterogeneous GPUs. Its primary design goal is correct and efficient kernel disaggregation.

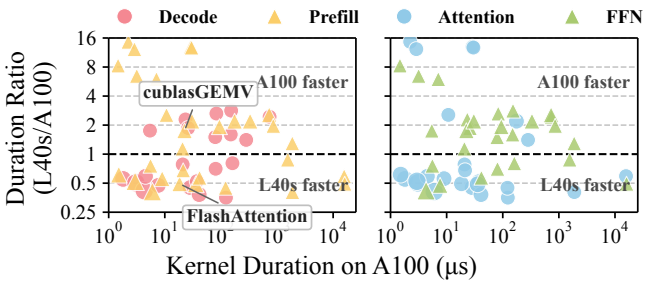

Fig. 3: Kernel duration ratio (L40s / A100) for GPT-oss 20B inference, grouped by (a) prefill and decode phases and (b) attention and FFN blocks. Kernels above the dashed line run faster on A100, while those below run faster on L40s.

```
.visible .entry opaque_kernel(
    ... // original parameters
    .param .u64 inst_buf ){
    .reg .b64   %rd<15>; // Declare registers
    .reg .b64   %unused
    ... // Load parameters
    ld.param.u64 %rd10, [inst_buf];
    ... // Original kernel body
    mad.wide.u32 %rd11, k, 16, %rd10;
    atom.global.min.u64 %unused, [%rd11], %rd1;
    atom.global.max.u64 %unused, [%rd11+8], %rd1;
    // Original k-th memory access instruction
    ld.global.f32 %f1, [%rd1];
    ... // Remaining kernel body
    }
```

Listing 1: PTX instrumentation logic for an opaque kernel.

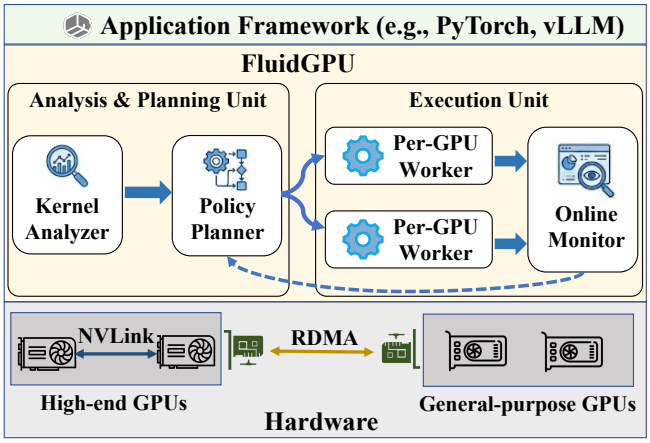

Fig. 4: Design overview of FluidGPU.

As shown in Figure 4, FluidGPU consists of four main components. The kernel analyzer profiles kernels offline to capture memory access patterns, extract inter-kernel data dependencies, and measure kernel latency on heterogeneous GPUs. Using these results, the policy planner derives workload-aware kernel scheduling policies that jointly consider kernel characteristics, communication overhead, and load balance. At runtime, the GPU worker executes the scheduling policy for disaggregated kernel execution. Each worker executes only its assigned kernels and orchestrates the required inter-GPU communication. Meanwhile, the online monitor continuously collects runtime statistics (*e.g.*, request and kernel latencies) to support dynamic policy adaptation under changing workloads.

A. Kernel Analyzer

The kernel analyzer identifies accessed buffer addresses and sizes from PTX code to resolve inter-kernel data dependencies. In practice, GPU kernels fall into two categories. The first includes standard library kernels, such as `cublasSgemm` [24], whose semantics and memory access patterns are well defined. The second comprises opaque computation kernels. Analyzing such kernels is challenging, as they are often developer-defined or JIT-compiled, making their memory access patterns difficult to observe.

Memory access analysis. For standard library kernels, identifying memory access patterns is straightforward because of their deterministic semantics. For example, the semantics of `cublasSgemm(.,A,B,.,C,.)` specify reads from `bufferA` and `bufferB` and a write to `bufferC`, while the API parameters provide the buffer sizes. In contrast, opaque kernels lack exposed semantics, making their memory access patterns difficult to infer.

PhoenixOS [27] represents prior work that identifies accessed buffers through GPU API argument speculation and runtime validation. Although this technique recovers buffer base addresses, it has a fundamental limitation. Modern AI frameworks such as PyTorch [17] employ internal memory managers (*e.g.*, the caching allocator [28]) that virtualize GPU memory independently of the CUDA runtime. Because CUDA allocation APIs are not invoked for every tensor creation, this technique often overestimates the actual buffer size and causes unnecessary communication in FluidGPU.

FluidGPU addresses this limitation with a kernel analyzer implemented at the PTX level [15]. Following an approach similar to eBPF [29], [30], FluidGPU instruments kernel code through PTX injection to track memory accesses. Listing 1 illustrates how instrumentation is performed on an opaque kernel. Specifically, each global memory instruction (*e.g.*, `st.global`, `ld.global`) is instrumented to record the virtual addresses it accesses. Since the virtual addresses of buffers are contiguous, FluidGPU maintains only the minimum and maximum accessed addresses per instruction. These values are safely aggregated across concurrent threads using atomic operations (*e.g.*, `atom.global`) to avoid data races. After kernel execution, FluidGPU retrieves the instruction-level $[min_k, max_k]$ intervals and aligns these aggregated intervals with the buffer base addresses to compute precise buffer sizes. The instruction opcode (*e.g.*, `ld.global` or `st.global`) also indicates whether the identified buffer is read or written. Implementation details of the PTX analyzer are provided in §IV.

Data dependency analysis. After determining the precise

read and write semantics of each accessed buffer, FluidGPU constructs a *data dependency graph* (DDG) that captures inter-kernel dependencies. Since kernel disaggregation preserves the original execution order, the analyzer only needs to identify true data dependencies, *i.e.*, read-after-write (RAW) dependencies. Specifically, the analyzer maintains a global buffer registry that tracks the last writer of each buffer. When a kernel reads a buffer, the analyzer queries the registry to identify the most recent writer. If the writer differs from the current kernel, a dependency edge is added from the writer to the reader. By iterating over kernels in execution order, FluidGPU constructs a DDG where nodes represent kernels and directed edges encode RAW dependencies annotated with concrete buffer sizes. Asynchronous memory copies are treated uniformly as kernel nodes in the DDG, since their source, destination, and transfer size are explicitly specified in the API parameters.

The DDG captures intra-iteration dependencies by default. For buffers with cross-iteration RAW dependencies (*e.g.*, KV caches in Transformer-based models [31]), the analyzer detects these dependencies by profiling multiple iterations and identifying buffers written in one iteration and read in subsequent iterations. The GPU workers handle these cross-iteration dependencies through per-GPU buffer replication and asynchronous delta transfers, as described in §III-C.

Inference workloads may exhibit dynamic execution patterns, for example, due to varying tensor shapes or changes in computation driven by input sequence length. In practice, this is less problematic for inference, as modern inference frameworks (*e.g.*, vLLM [16], TensorRT [32]) increasingly use CUDA Graph [33] to reduce kernel launch overhead. Since CUDA Graph naturally requires a static execution pattern [34], FluidGPU can construct a separate DDG for each captured graph, capturing dynamic behaviors across different executions. For frameworks that do not yet support CUDA Graph capture, FluidGPU provides a preprocessing tool to automatically record the kernel execution trace and replay it under CUDA's stream capture API to produce an equivalent CUDA Graph.

Therefore, FluidGPU does not require the original application to use CUDA Graphs, but requires only a repeatable kernel sequence for each supported input shape. The profiling pipeline is executed once for each combination of hardware configuration and CUDA Graph or input-shape pattern. Subsequent requests matching the same pattern reuse the cached DDG and scheduling policy without repeating PTX instrumentation or dependency analysis. For data-dependent control flow, FluidGPU pre-generates a graph, DDG, and policy for each profiled path and falls back to single-GPU execution for unseen or non-capturable paths. Different requests therefore do not trigger continuous element-level dependency tracking. The offline preprocessing overhead is evaluated in §V-D.

In addition, the analyzer profiles the execution latency of each kernel on heterogeneous GPUs to inform scheduling-policy generation. The overall analysis is a one-time preprocessing step that can usually be completed during application warm-up.

B. Policy Planner

The policy planner generates a kernel-to-GPU scheduling policy for each DDG produced by the kernel analyzer. Because performance goals vary across scenarios (*e.g.*, latency-critical online serving vs. throughput-driven offline processing), FluidGPU supports both throughput- and latency-oriented policies.

Throughput-oriented policy. This policy maximizes system throughput and targets offline inference workloads that seek to process as many requests as possible within a given time window (*e.g.*, batched document processing). We formulate this mapping strategy as a mixed-integer linear programming (MILP) problem, with key terminology summarized in Table II.

The solver first introduces a set of binary placement variables $x_{k,g}$, where $x_{k,g} = 1$ indicates that kernel k is assigned to GPU g. Each kernel must be assigned to exactly one GPU:

$$\sum_{g \in G} x_{k,g} = 1, \quad \forall k \in K.$$

For each GPU $g \in G$, the computation time T_g is the aggregated execution latency of all kernels assigned to it:

$$T_g = \sum_{k \in K} t_{k,g}\, x_{k,g}$$

The communication overhead M_g captures the total transfer cost incurred by all incoming cross-GPU dependency edges targeting GPU g:

$$M_g = \sum_{(i,j) \in E} \sum_{u \in G \setminus \{g\}} c_{ij}^{u,g} y_{ij}^{u,g},$$

$$c_{ij}^{u,g} = \ell_{u,g} + \frac{d_{ij}}{bw_{u,g}}$$

Here, $y_{ij}^{u,g} \in \{0,1\}$ encodes the joint placement of edge (i,j), *i.e.*, $y_{ij}^{u,g} = x_{i,u} x_{j,g}$. To keep the formulation strictly linear, FluidGPU applies standard boolean linearization techniques [35] to replace this quadratic term.

Under pipelined asynchronous execution, each GPU alternates between computation and communication. The per-request stage time on GPU g is therefore bounded by its slower component:

$$W_g = \max(T_g, M_g)$$

Following prior pipeline analysis [36], the steady-state throughput of the entire system is determined by the slowest stage. Maximizing overall throughput is thus equivalent to minimizing the maximum stage time. The final objective is:

$$\min \max_{g \in G} W_g$$

Latency-oriented policy. The latency-oriented policy targets online serving scenarios where the objective is to minimize per-request end-to-end latency. At low load, limited request concurrency prevents FluidGPU from effectively overlapping computation and communication through pipelined execution. As a result, the per-request latency can be decomposed into

TABLE II: Terminology used in the problem formulation.

Symbol	Description
G	Set of available heterogeneous GPUs.
K	Set of kernels.
E	Set of data-dependency edges in the DDG.
$t_{k,g}$	Profiled latency of kernel k on GPU g.
d_{ij}	Size of the buffer transferred from kernel i to j.
$bw_{u,g}$	Communication bandwidth between GPU u and g.
$\ell_{u,g}$	Base communication latency between GPU u and g.
$c_{ij}^{u,g}$	Data transfer overhead for edge (i,j) crossing from GPU u to g.
$x_{k,g}$	Binary variable equal to 1 if kernel k is assigned to GPU g and 0 otherwise.

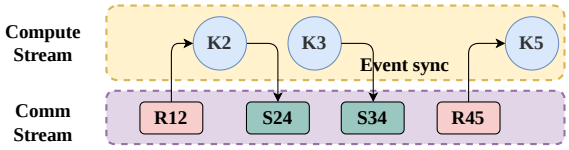

Fig. 5: An execution example of the GPU worker. Compute kernels and communication operations run on separate streams and synchronize through CUDA events. R_{ij} and S_{ij} label receive and send operations using their corresponding DDG edge identifiers.

two additive components: the total kernel execution time and the cumulative cross-GPU transfer overhead incurred along cut dependency edges. Under such conditions, the objective of the MILP solver is:

$$\min \left(\sum_{k \in K} \sum_{g \in G} t_{k,g} \cdot x_{k,g} + \sum_{(i,j) \in E} \sum_{u \in G} \sum_{g \in G \setminus \{u\}} c_{ij}^{u,g} y_{ij}^{u,g} \right)$$

The resulting MILPs can be solved efficiently with standard optimizers such as Gurobi [37]. The number of decision variables scales with $|K| \times |G|$, where $|K|$ denotes the number of kernels per DDG and $|G|$ the number of GPUs. In a typical configuration from our evaluation, GPT-oss 20B on an A100+L40s node has $|G| = 2$ and $|K| \approx 500$. Gurobi solves each DDG in about 20 ms, and enumerating all execution patterns takes roughly 5 seconds in total. This overhead is negligible because MILP optimization is performed offline and incurs no runtime overhead. The scalability of the MILP solver is discussed in §V-D.

C. GPU Workers

At runtime, FluidGPU launches one GPU worker on each GPU, and each worker executes its assigned portion of the computation.

Disaggregated kernel execution. Based on the scheduling policy, each GPU worker executes only its assigned kernels. Because disaggregated execution requires explicit cross-GPU data transfers along cut dependency edges [38], FluidGPU adopts GPU-initiated communication via InfiniBand GPUDirect Async (IBGDA) [39], as supported by NCCL [40], to eliminate control-path overhead from CPU–GPU synchronization. This CPU-bypass mechanism allows `send` and `recv` operations to be issued as GPU-side kernels.

For a kernel k with incoming cut dependency edges, the worker posts the corresponding `recv` kernels before launching k. After k completes, if it has outgoing cut edges, the worker posts the corresponding `send` kernels. The two streams enable communication to overlap with independent computation, while lightweight CUDA events synchronize only direct dependencies.

Figure 5 illustrates dependency-local synchronization within one GPU worker. The receive operation R_{12} must complete before K_2 starts. After K_2 completes, S_{24} proceeds on the communication stream while the independent K_3 continues on the compute stream. The send operation S_{34} waits for K_3 to complete, while K_5 waits for R_{45}. Only operations connected by an event synchronize, allowing unrelated computation and communication to overlap within the same request.

Dynamic execution patterns. The GPU worker handles dynamic execution patterns through the CUDA Graphs described in §III-A. Since the kernel analyzer and policy planner have already constructed a separate DDG and scheduling policy for each CUDA Graph offline, the GPU worker only needs to select the correct policy at runtime. Each worker maintains a dispatch table that maps each CUDA Graph handle to its prebuilt scheduling policy. When a worker intercepts a CUDA Graph launch, it looks up and executes the corresponding policy. For low-overhead replay, FluidGPU decomposes each CUDA Graph into per-GPU subgraphs with `send` and `recv` nodes embedded alongside compute nodes (§IV).

Pipelined request processing. Disaggregated kernel execution without pipelining leads to substantial GPU idle time during communication. To address this issue, FluidGPU employs a pipelined request-processing mechanism that executes multiple independent requests concurrently across heterogeneous GPUs using multiple CUDA streams [41]. Each request is assigned to a dedicated compute stream. When one stream is blocked by a data transfer required by a dependency, the GPU hardware scheduler dispatches ready kernels from other streams. This asynchronous execution overlaps computation with communication and reduces GPU idle time. However, when multiple requests execute concurrently without differentiation, they compete equally for Streaming Multiprocessor (SM) resources and tend to progress through the pipeline at similar rates. As a result, their communication phases may align, causing all streams to stall on data transfers simultaneously and leaving the GPU idle.

To address this, FluidGPU maintains a fixed pool of priority-ordered CUDA compute streams whose size is bounded by the configured pipeline depth. A priority encodes only the relative arrival order of requests currently in flight. Earlier requests use higher-priority streams, biasing the GPU scheduler toward their ready kernels and staggering communication phases [42]. When a request completes, its stream and priority slot return to the pool. The runtime then recomputes the relative order of the remaining requests and updates their live streams through `cudaStreamSetAttribute` with

`cudaStreamAttributePriority`. The updated priority applies only to work submitted after the call and does not affect previously submitted or executing kernels [41]. Later requests reuse the available streams and priority slots. The effectiveness of pipelined request processing is evaluated in §V-D.

KV-cache delta synchronization. FluidGPU does not transfer the entire KV cache between GPUs. Instead, each GPU maintains a KV-cache replica. After every decoding step, the GPU worker uses the paged-KV block table and slot mapping maintained by vLLM [16] to determine only the addresses and lengths of the newly generated K and V slices. It asynchronously propagates these KV-cache deltas to peer GPUs through a dedicated communication stream and a separate NCCL communicator [40], while latency-critical intra-iteration transfers use a higher-priority traffic class [43]. The DDG records the KV cache's cross-iteration RAW dependencies and the corresponding modified byte ranges. Before a kernel reads an updated KV-cache range in the next decoding step, the GPU worker waits for the associated delta-transfer completion event. In our measurements, asynchronously transferring this KV-cache delta adds less than 0.1% to end-to-end decoding latency.

Composability with model parallelism. FluidGPU is orthogonal to model parallelism and composes naturally with existing parallelism strategies. It operates below the model-parallelism layer and leaves existing communication operations and topology unchanged. FluidGPU schedules only the compute kernels between consecutive communication steps within each heterogeneous GPU group. For tensor parallelism (TP), collectives remain on the original homogeneous TP group, and FluidGPU optimizes compute kernels within each TP rank's heterogeneous GPU group, as evaluated in Figure 9(b). For sequence parallelism (SP) and Ring Attention [44], sequence-shard exchanges and ring send and receive operations remain unchanged, while local attention and GEMM kernels between two communication steps can be disaggregated. For expert parallelism (EP) in MoE models [12], expert dispatch and combine all-to-alls remain fixed, while expert GEMMs are scheduled across the heterogeneous GPU group. Because input-dependent expert routing can produce different kernel graphs, each profiled routing pattern requires a separate DDG and policy. An unprofiled pattern conservatively retains the original expert placement.

D. Online Monitor

The online monitor targets serving scenarios where request arrival rates fluctuate unpredictably over time. Unlike offline workloads with fixed load, online serving must handle dynamic queueing pressure, making any static policy suboptimal. A latency-oriented policy processes requests without pipelining, which underutilizes GPU resources and incurs high queueing delay under heavy load. In contrast, a throughput-oriented policy becomes less effective under light load due to limited concurrency and may introduce additional latency from suboptimal scheduling.

To adapt to workload dynamics, the monitor tracks per-request latency, per-kernel-group latency, and communication time. Instead of profiling individual kernels, it measures the interval between consecutive communication operations. This interval contains a sequence of kernels, which we define as a *kernel group*. This design reduces monitoring overhead while retaining sufficient information for effective policy selection. Based on these signals, FluidGPU performs queueing-aware policy switching at fixed intervals of length W.

At each window boundary, the monitor computes the average request latency $\bar{L}_{\text{req}}$ and the execution-only latency $\bar{L}_{\text{exec}}$, which aggregates computation and communication time while excluding queueing delay. The ratio $\bar{L}_{\text{req}}/\bar{L}_{\text{exec}}$ serves as an indicator of queueing pressure. A low ratio implies negligible queueing delay and favors the latency-oriented policy, whereas a high ratio indicates that queueing dominates end-to-end latency, triggering a switch to the throughput-oriented policy to improve system processing capacity.

The monitor introduces two key hyperparameters, the time window W and the queueing threshold β, which can be tuned according to workload characteristics. We further analyze their sensitivity and the runtime overhead of the monitor in §V-D.

IV. Implementation

The current implementation of FluidGPU targets NVIDIA GPUs. It comprises approximately 3K lines of C++/CUDA code for the GPU worker runtime and 2K lines of Python code for the kernel analyzer, including PTX instrumentation and data dependency analysis. FluidGPU is integrated with vLLM [16] and PyTorch [17].

PTX analyzer. Building on the probe engine from NEUTRINO [29], we implement a PTX-level kernel analyzer. To track memory accesses, we extend the kernel parameter list with an instrumentation buffer that is loaded through inserted `ld.param` instructions. We then inject `atom.global` instructions around each global memory access (*e.g.*, `ld.global`, `st.global`) to record the minimum and maximum accessed addresses per instruction. The modified PTX is recompiled into machine code with `ptxas` [45].

A known limitation is that the analyzer cannot handle indirect buffer accesses. For example, if a kernel accesses GPU buffers via global variables not listed in the kernel launch arguments, the analyzer cannot obtain the base buffer address to calculate the buffer size. However, a comprehensive study [27] shows that the most popular ML frameworks (*e.g.*, PyTorch, vLLM) do not contain such kernels. Only one kernel in the legacy Rodinia benchmark [46] exhibits this pattern. In these rare cases, FluidGPU conservatively executes the kernel on the same GPU without disaggregation.

CUDA Graph adaptation. For an application-provided CUDA Graph, FluidGPU partitions the compute nodes into cached per-GPU subgraphs in topological order according to the scheduling policy. It inserts GPU-initiated `send` and `recv` operations as kernel nodes at cross-GPU dependency edges and represents CUDA event record and wait operations as event nodes. Each GPU worker replays its complete cached

subgraph with a single `cudaGraphLaunch` rather than issuing individual kernels from the host. Compute kernels therefore remain back-to-back wherever no cross-GPU dependency intervenes, preserving the amortization of CPU-side launch and setup overhead provided by CUDA Graphs [47]. Only communication required by a cut dependency separates compute regions, and the policy planner explicitly models this cost. A graph generated from a conventional execution trace follows the same partitioning, caching, and replay procedure.

Memory management. The current MILP formulation does not explicitly model per-GPU memory-capacity constraints and replicates the full model weights on every GPU during initialization, similar to PD disaggregation [10]. In practice, full replication is conservative because each GPU only accesses a subset of weight buffers, and the remaining unused buffers could be reclaimed. For offline workloads whose scheduling policy remains fixed, selective buffer reclamation can safely free memory for larger batch sizes.

V. Evaluation

A. Experimental Setup

Testbeds. We evaluate FluidGPU in two testbed environments. Our main evaluation platform is a *local setup* with three heterogeneous GPU pairs: 1 A100 + 1 L40s, 1 H100 + 1 RTX Pro 6000, and 1 B200 + 1 H100. Each GPU is equipped with a 200 Gbps RDMA NIC. We further evaluate FluidGPU's scalability in a *cluster setup* with two distributed configurations. The first combines a node with 2 A100 GPUs and a node with 1 L40s GPU. The second combines a node with 8 B200 GPUs and a node with 8 H100 GPUs. Within each homogeneous high-end GPU group, GPUs are interconnected via NVLink, while cross-node communication is conducted over 200 and 400 Gbps RDMA NICs, respectively.

Workloads. To assess the generality of FluidGPU, we evaluate four model families: (1) *LLMs*: Llama-3 8B (LM) [48] and GPT-oss 20B (GT) [49], Transformer-based autoregressive models with distinct prefill and decode phases; (2) *SSMs*: Mamba-Codestral 7B (MB) [50], which replaces attention with selective state-space layers for linear-time sequence generation; (3) *MLLMs*: Qwen2.5-VL 7B (QW) [51], which augments an LLM decoder with a vision encoder for joint image-text reasoning; and (4) *Diffusion models*: Stable Diffusion 3.5 (SD) [52], [53], a text-to-image model based on the Multimodal Diffusion Transformer architecture, which synthesizes images through iterative denoising rather than autoregressive token generation. For *offline evaluation*, the LLM and SSM workloads use the Splitwise conversational dataset [9], with a median input length of 1020 tokens and a median output length of 129 tokens. The MLLM workload uses images from the COCO captioning dataset [54], resized to 512×512 pixels. The diffusion workload uses text prompts from the PartiPrompts dataset [55] at 1024×1024 resolution with 28 denoising steps. We measure the maximum sustainable throughput following common practice [56], [57]. For *online evaluation*, we focus on GPT-oss 20B inference under both Poisson arrivals at varying request rates and a real-world trace from the Azure Conversation dataset [9], following previous work [58].

Baselines. We compare FluidGPU with two state-of-the-art disaggregation methods:

- **Prefill–decode disaggregation (PD Dis.):** a phase-level approach following DistServe [10], which assigns the compute-intensive prefill phase and the memory-intensive decode phase to separate GPUs. This baseline does not apply because diffusion models lack a prefill–decode separation.
- **Attention–FFN disaggregation (AF Dis.):** a block-level approach following MegaScale-Infer [12], which maps memory-intensive attention and compute-intensive FFN operations to different GPUs. This baseline does not apply to SSM and diffusion models because these models do not expose the standard attention–FFN block structure assumed by AF disaggregation.

We also include request distribution (Request Dist.) for offline throughput. Each GPU in the heterogeneous pair runs an independent vLLM replica, and incoming requests use the best-performing vLLM load-balancing policy among those evaluated. Unlike the disaggregation baselines, this scheme assigns each complete request to one GPU and performs no cross-GPU kernel execution.

We also include homogeneous-GPU baselines without disaggregation. The inference engine is vLLM v0.18.0 [16], with NCCL [40] as the communication backend. We enable standard scheduling techniques (*e.g.*, continuous batching) and tune each disaggregation baseline to its best-performing GPU configuration for each hardware setting. To reduce latency jitter, we disable GPU dynamic voltage and frequency scaling (DVFS) [59]. Each experiment is repeated multiple times, with variance below 1%, and we report the average results.

B. End-to-end Performance

Offline throughput. Figure 6 reports the maximum sustainable throughput in offline serving across five workloads and three heterogeneous GPU pairs. We measure throughput as output tokens per second for LLMs, SSMs, and MLLMs, and as requests per minute for diffusion models. Across applicable configurations, FluidGPU outperforms PD disaggregation by 1.5× on average (up to 2.3×) and AF disaggregation by 1.35× on average (up to 1.7×).

Across the evaluated workloads and GPU pairs, Request Dist. falls below the ideal sum of the two single-GPU throughputs because of request-level load imbalance and routing overhead. For GPT-oss 20B on A100+L40s, its measured throughput is 2,052 tokens/s, whereas FluidGPU reaches 2,641 tokens/s, a 28.7% improvement. Both configurations use the same request workload and both GPUs. This result shows that assigning whole requests to independent replicas cannot capture the within-request kernel preferences exploited by FluidGPU.

The performance gains arise because FluidGPU captures kernel-level heterogeneity that coarser-grained approaches miss. PD disaggregation and AF disaggregation treat an entire

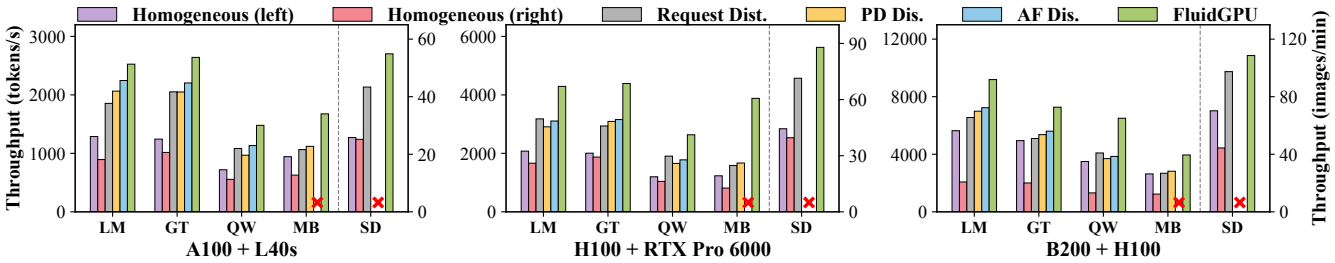

Fig. 6: Offline throughput across five workloads and three heterogeneous GPU pairs. Request Dist. uses two independent replicas for every workload and GPU pair. The left y-axis shows tokens/s for LLM, MLLM, and SSM workloads, and the right y-axis shows images/min for SD. Red X marks indicate that a disaggregation method is inapplicable.

phase or block as a single scheduling unit, forcing kernels with diverse resource demands onto the same GPU. In contrast, FluidGPU schedules kernels independently, aligning fine-grained kernel heterogeneity with hardware characteristics.

Existing disaggregation methods also have limited applicability. AF disaggregation does not apply to SSM and diffusion models because these models do not expose its assumed standard attention–FFN block structure. PD disaggregation does not apply because diffusion models lack a prefill–decode separation. In contrast, FluidGPU is model-agnostic and applies uniformly across diverse model architectures, providing strong generality.

Cost efficiency. Table III reports cost efficiency (Perf/$), computed as the average throughput across five workloads divided by GPU rental cost and normalized to the left-hand homogeneous-GPU baseline. GPU rental prices are listed in Table I, and for disaggregation methods, the total cost is the sum of both GPUs. Across all three heterogeneous GPU pairs, FluidGPU consistently achieves the highest cost efficiency, outperforming PD disaggregation by 1.5× on average (up to 1.6×) and AF disaggregation by 1.4× on average (up to 1.5×).

The H100+RTX Pro 6000 pair provides a concrete example of the benefits of heterogeneous execution. Across the five offline workloads, FluidGPU averages a 2.3× throughput speedup over a single H100 and exceeds twice the measured single-H100 throughput on four workloads. Under the rental-cost model in Table I, the heterogeneous pair has an aggregate rental cost 29.3% lower than that of two H100s and achieves 1.64× the Perf/$ of a single-H100 deployment. The 2.3× result compares throughput, whereas the 1.64× result compares throughput per dollar after accounting for aggregate GPU rental cost. The H100 favors bandwidth-sensitive kernels because of its higher memory bandwidth, whereas the RTX Pro 6000 benefits CUDA-core- or cache-sensitive kernels through its higher FP32 CUDA-core throughput and larger L2 cache. FluidGPU exploits these complementary strengths through kernel-level placement.

These results suggest that cloud providers can improve cost efficiency by pairing high-end GPUs with more affordable counterparts, rather than provisioning additional expensive GPUs.

Online latency. We evaluate GPT-oss 20B serving with Poisson arrivals at varying request rates. We measure normalized latency as the end-to-end request latency divided by output length in tokens. Figure 7 reports normalized latency across different request rates on three GPU pairs. At low request rates, PD disaggregation and AF disaggregation have normalized latencies 1.3× and 1.2× higher than FluidGPU, respectively, because FluidGPU's latency-oriented policy minimizes per-request critical-path delay. Moreover, under an SLO target of 50 ms/token, FluidGPU sustains up to 1.3× higher request rate than the best baseline before violating the constraint.

TABLE III: Cost efficiency (Perf/$) on three GPU pairs, normalized to the homogeneous left GPU baseline.

	A100+L40s	H100+RTX Pro 6000	B200+H100
Homo. (left)	1.00	1.00	1.00
Homo. (right)	1.18	2.01	0.78
PD Dis.	0.87	1.00	0.70
AF Dis.	1.02	1.07	0.74
FluidGPU	**1.21**	**1.64**	**1.01**

Figure 8 reports the P99 time-to-first-token (TTFT) and P99 time-per-output-token (TPOT) for GPT-oss 20B under 16 req/s on A100+L40s. FluidGPU achieves the lowest tail latency for both metrics. Its higher throughput lets the system process queued requests more quickly, reducing queueing delay and therefore P99 TTFT. Its kernel-level disaggregation places each kernel on the GPU best suited to its resource characteristics, shortening kernel execution along each request's critical path and therefore P99 TPOT. PD disaggregation has 1.28× higher P99 TTFT and 1.11× higher P99 TPOT than FluidGPU. AF disaggregation has 1.20× higher P99 TTFT and 1.26× higher P99 TPOT than FluidGPU.

Compared to the offline setting, kernel disaggregation faces additional challenges in online serving. Inter-GPU transfer latency lies on the critical path of individual requests, and at low request rates, the pipeline cannot be fully saturated, limiting opportunities to overlap communication with computation. Nevertheless, FluidGPU still achieves the lowest latency among all disaggregation baselines, enabled by its latency-oriented policy and the queueing-aware policy switching of the online monitor.

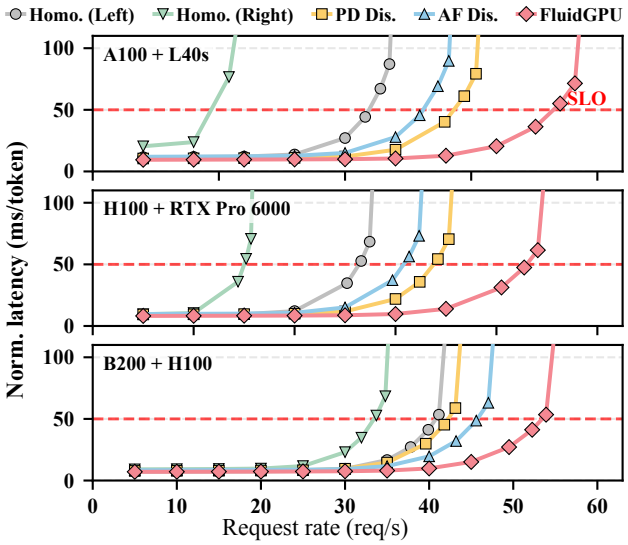

Fig. 7: Online normalized latency on GPT-oss 20B across three GPU pairs. The red dashed line marks the SLO target (50 ms/token).

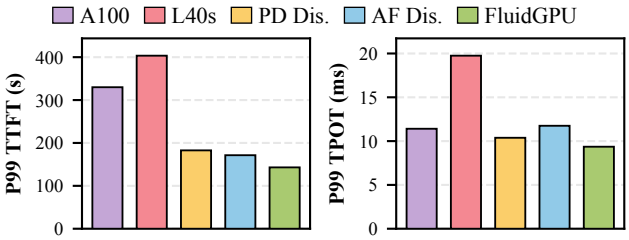

Fig. 8: (a) P99 TTFT and (b) P99 TPOT for GPT-oss 20B under 16 req/s on A100+L40s.

C. Cluster-Scale Results

The preceding experiments evaluate FluidGPU on individual heterogeneous GPU pairs. We now examine its scalability on asymmetric GPU clusters and its composability with model parallelism (*e.g.*, TP) in two offline setups. The first runs GPT-oss 20B on 2 A100 + 1 L40s, and the second runs Qwen3 235B [60] on 8 B200 + 8 H100. For the asymmetric 2 A100 + 1 L40s setup, FluidGPU's policy planner directly formulates a 3-GPU MILP, jointly optimizing kernel placement across all three GPUs. PD disaggregation uses L40s as the prefill instance and each A100 as an independent decode instance, and AF disaggregation runs attention on 2 A100 with TP=2 and FFN on L40s. Both baselines follow the default configurations from their original work [10], [12] and are tuned for maximum throughput. For the 8 B200+8 H100 setup, FluidGPU treats each B200+H100 pair as one TP rank and applies kernel disaggregation within each pair. In each setup, TP runs over NVLink within the homogeneous high-end GPU group, and collectives remain on the original GPU group.

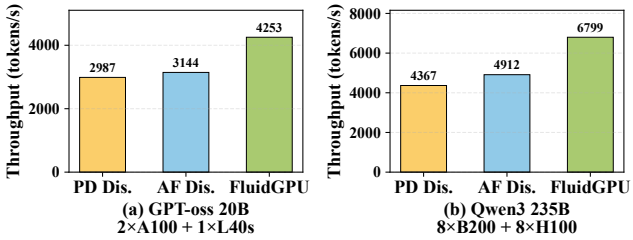

Fig. 9: Cluster-scale offline throughput. (a) GPT-oss 20B on 2 A100 + 1 L40s. (b) Qwen3 235B on 8 B200 + 8 H100.

Figure 9 reports the offline throughput. Across both setups, FluidGPU achieves 1.5× the throughput of PD disaggregation and 1.4× that of AF disaggregation. The fundamental limitations of coarse-grained disaggregation persist at the cluster scale. In contrast, FluidGPU either applies kernel disaggregation independently within each TP rank pair or jointly optimizes kernel placement across the entire cluster while reusing the same policy planner and pipelined execution. These results confirm that FluidGPU composes naturally with model parallelism and scales from a single GPU pair to multi-GPU clusters without requiring changes to the parallelism strategy.

D. Performance Drilldown

Pipeline effectiveness. We evaluate pipelined request processing on GPT-oss 20B in an offline setting. Figure 10(a) reports throughput under three configurations. The red dashed line denotes the optimal throughput, assuming zero communication overhead. Without pipelining, GPUs idle during cross-GPU transfers, achieving only about half of the optimal throughput. Pipelining without priority scheduling improves throughput by 1.47× by overlapping communication with computation from concurrent requests. Incorporating priority-aware scheduling further increases throughput to 96.6% of optimal. Without priority differentiation, concurrent requests may compete for SMs and reach their communication phases simultaneously, leaving the GPU idle while all requests wait for data transfers. By assigning lower stream priorities to later-arriving requests, the GPU hardware scheduler preferentially allocates SMs to earlier requests, staggering their communication phases and keeping the GPU continuously occupied.

Figure 10(b) shows the time breakdown on the bottleneck GPU. With pipelining and priority scheduling, communication idle time falls from nearly 50% to less than 4%, confirming that the pipeline effectively hides transfer latency. This also validates the cost model in §III-B, which assumes communication can be fully overlapped under steady-state pipelining.

Online monitor sensitivity. Figure 11 illustrates the sensitivity of normalized latency and policy-switch frequency to the window size W and queueing threshold β on GPT-oss 20B under a 360-second real-world workload trace [9]. Within each window W, the monitor decides whether to trigger a policy switch based on β. The key trade-off lies between switching overhead and adaptation delay. Each policy switch requires

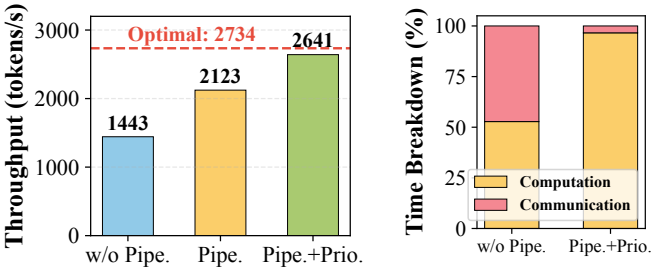

Fig. 10: (a) Throughput under different pipeline configurations. (b) Time breakdown on the bottleneck GPU.

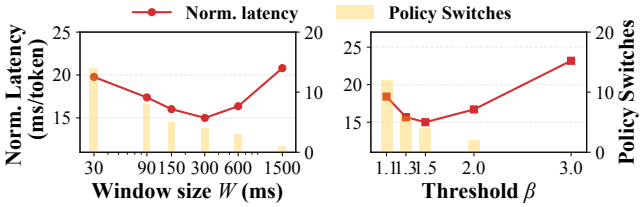

Fig. 11: Sensitivity of normalized latency and policy switch count to window size W ($\beta = 1.5$) and queueing threshold β ($W = 300$ ms) on GPT-oss 20B.

synchronizing all GPU workers at an iteration boundary, incurring a stall of approximately 30 ms. Aggressive configurations (*e.g.*, $W = 30$ ms or $\beta = 1.1$) cause frequent switching, whose accumulated overhead increases latency by a factor of up to 1.32×. In contrast, overly conservative settings (*e.g.*, $W = 1.5$ s or $\beta = 3.0$) delay adaptation to load changes, leaving the system under suboptimal policies during spikes and increasing latency by a factor of up to 1.55×. We therefore use $W = 300$ ms and $\beta = 1.5$ by default, which provide the best trade-off between responsiveness and overhead.

Robustness to reduced bandwidth. We throttle the A100–L40s interconnect from 200 Gbps to 100, 50, and 25 Gbps and evaluate GPT-oss 20B in offline serving and in online serving under light load, as shown in Figure 12(a). For offline serving, even at 25 Gbps, FluidGPU's throughput drops by less than 6% because pipelined request processing hides the increased transfer latency behind computation from concurrent requests. For online serving under a light request load, FluidGPU's latency-oriented policy accounts for the higher communication cost and shifts more kernels to the same GPU to reduce cross-GPU data transfer. Even with a 25 Gbps network, FluidGPU's normalized latency remains lower than the homogeneous A100 baseline, because the latency-oriented policy planner still finds a beneficial kernel placement at this bandwidth. In the worst case, FluidGPU keeps all kernels on A100, avoids cross-GPU transfers, and falls back to homogeneous-GPU execution without a performance cliff.

Offline preprocessing overhead. In our evaluation, PTX instrumentation and dependency extraction take no more than 10 seconds during preprocessing. This measurement covers PTX rewriting, instrumented execution, and RAW-dependency

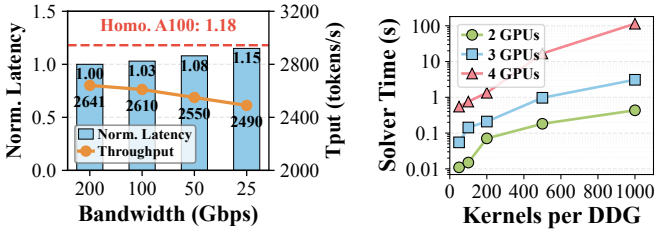

Fig. 12: (a) Impact of network bandwidth on GPT-oss 20B with A100+L40s. (b) MILP solver time.

extraction, but excludes per-GPU kernel latency profiling and policy optimization. By contrast, the approximately 5-second cost reported in §III-B measures MILP optimization over all profiled execution patterns after their DDGs and kernel profiles are available. Cached DDGs and policies are reused across requests, and runtime execution performs no instruction-level or element-level dependency tracking.

MILP scalability. Figure 12(b) reports the MILP solver time as a function of kernel count and GPU count using Gurobi [37]. For the 2-GPU setting, solving a single DDG takes 0.07 s for 200 kernels and 0.43 s for 1000 kernels. Using DeepSeek-V3 671B [61] as a stress test, a forward iteration with one DDG containing nearly 1500 kernels can be solved within 1 s. Although solver time increases with the number of GPUs, most layers in large models share identical kernel compositions, enabling the same placement to be reused across repeated layers. FluidGPU exploits this redundancy to reduce the problem size and further accelerate the MILP solver.

VI. Related Work

Disaggregated inference serving. Recent work has explored disaggregation to better align workload characteristics with hardware features. DistServe [10] and Splitwise [9] separate the prefill and decode phases of LLM inference across different GPU pools. HexGen-2 [14] formulates the placement of disaggregated prefill and decode computations on heterogeneous GPUs as a constrained optimization problem, jointly considering resource allocation and parallelization strategies to improve serving throughput. Cronus [11] further enables fine-grained prefill partitioning to mitigate load imbalance arising from heterogeneous GPU capabilities. MegaScale-Infer [12] disaggregates execution at the block level by separating attention and FFN components across GPUs with different hardware strengths. These approaches nevertheless operate at phase or block granularity and remain tightly coupled to specific model architectures. In contrast, FluidGPU disaggregates at the kernel level, providing finer-grained optimization while remaining applicable across diverse model types.

Cluster-level placement and scheduling. Beyond choosing where to disaggregate computation within an inference request, another line of work optimizes model placement, request assignment, and parallelism configurations at the cluster level. Sailor [62] jointly searches data, tensor, and pipeline parallelism configurations for distributed training. Metis [63]

uses workload-aware partitioning to automate training on heterogeneous GPUs. Hetis [58] combines selective operator parallelization with online request dispatching to balance serving load. Helix [57] jointly optimizes model placement and request routing by representing heterogeneous GPU and network capacities through a max-flow abstraction.

Among these systems, Helix is the closest to FluidGPU because both make heterogeneity-aware placement decisions. Their optimization scopes differ. Helix usually assigns contiguous layer groups to GPU nodes and routes requests through the resulting pipelines. FluidGPU uses this cluster-level placement and routing as fixed input, then maps individual CUDA kernels within each heterogeneous GPU group using measured kernel latency and communication cost. Since Transformer layers largely repeat the same kernel composition, layer-level placement cannot exploit different hardware preferences among kernels within a layer. The two systems are therefore complementary rather than fully orthogonal. Helix can first determine cluster-level layer placement and request routes, while its layer-group boundaries become fixed communication boundaries for FluidGPU. Helix requires less fine-grained profiling and creates fewer placement boundaries. FluidGPU incurs one-time dependency-analysis and per-kernel profiling costs and models additional transfers in exchange for finer hardware–kernel alignment.

Heterogeneous accelerator scheduling. General-purpose runtimes including StarPU [64], OmpSs [65], Legion [66], and PaRSEC [67] schedule application tasks across heterogeneous processors and manage the associated data movement. These systems typically require applications to expose work and dependencies through runtime-specific abstractions such as task graphs. FluidGPU targets a different integration point. It starts from CUDA kernels already emitted by existing AI inference frameworks, automatically extracts their memory dependencies, and performs communication-aware kernel placement across heterogeneous GPUs.

VII. Conclusion

This paper presents FluidGPU, the first runtime system that exploits heterogeneous GPUs through fine-grained kernel disaggregation. Our central finding is that kernels within the same inference workload have distinct hardware preferences, making the kernel a natural granularity for aligning workload heterogeneity with GPU architectural diversity. FluidGPU combines automatic PTX-level dependency extraction, communication-aware policy planning, and pipelined execution to improve throughput and cost efficiency across diverse model architectures.

Acknowledgments

We thank the anonymous reviewers for their comments and are particularly grateful to our shepherd for valuable feedback. This work was supported in part by the National Key Research and Development Program of China (No. 2024YFE0204100), the Pioneer Initiative Talent Program B of the Chinese Academy of Sciences (Nos. E545030 and E401430), the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM118), and the National Natural Science Foundation of China under Grant Nos. U25A6023, 92464301, 62302479, U23B2020, and 625B2002. Tao Xie is also affiliated with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China; Beijing Tongming Lake Information Technology Application Innovation Center, China; Institute of Systems for Advanced Computing at Fudan University, China; and Shanghai Research Institute of Systems of Open Computing, China. Tiancheng Hu, Junhao Hu and Yuzheng Wang are also affiliated with the Key Laboratory of High Confidence Software Technologies (Peking University). Ting Cao is affiliated with the Institute for AI Industry Research, Tsinghua University, Beijing, China. Corresponding authors: Tao Xie and Chenxi Wang.

References

[1] Z. J. Kong, Q. Xu, and Y. C. Hu, "PPipe: Efficient video analytics serving on heterogeneous GPU clusters via pool-based pipeline parallelism," in *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. Boston, MA: USENIX Association, Jul. 2025, pp. 679–698. [Online]. Available: https://www.usenix.org/conference/atc25/presentation/kong

[2] J. Stojkovic, C. Zhang, Í. Goiri, and R. Bianchini, "Rearchitecting the datacenter lifecycle for AI," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Computer Architecture (ISCA 26)*, 2026. [Online]. Available: https://www.microsoft.com/en-us/research/publication/rearchitecting-datacenter-lifecycle-for-ai-a-tco-driven-framework/

[3] Google Cloud, "GPU machine types," https://docs.cloud.google.com/compute/docs/gpus, 2026, accessed: Jul. 18, 2026.

[4] Amazon Web Services, "Accelerated computing Amazon EC2 instance types," https://aws.amazon.com/ec2/instance-types/accelerated-computing/, 2026, accessed: Jul. 18, 2026.

[5] M. Lammertyn, "60+ ChatGPT facts and statistics you need to know in 2025," https://blog.invgate.com/chatgpt-statistics, 2024, accessed: Jul. 18, 2026.

[6] J. Kim, B. Shin, J. Chung, and M. Rhu, "The cost of dynamic reasoning: Demystifying AI agents and test-time scaling from an AI infrastructure perspective," in *2026 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2026, pp. 1–16.

[7] M. Luo, X. Shi, C. Cai, T. Zhang, J. Wong, Y. Wang, C. Wang, Y. Huang, Z. Chen, J. E. Gonzalez, and I. Stoica, "Agentix: An efficient serving engine for LLM agents as general programs," in *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. Renton, WA: USENIX Association, May 2026, pp. 2443–2459. [Online]. Available: https://www.usenix.org/conference/nsdi26/presentation/luo

[8] B. Gao, Z. He, P. Sharma, Q. Kang, D. Jevdjic, J. Deng, X. Yang, Z. Yu, and P. Zuo, "Cost-efficient large language model serving for multi-turn conversations with CachedAttention," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 111–126. [Online]. Available: https://www.usenix.org/conference/atc24/presentation/gao-bin-cost

[9] P. Patel, E. Choukse, C. Zhang, A. Shah, I. n. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative LLM inference using phase splitting," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 118–132.

[10] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 193–210. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin

[11] Y. Liu, Q. Xu, and Y. C. Hu, "Cronus: Efficient LLM inference on heterogeneous GPU clusters via partially disaggregated prefill," *arXiv preprint arXiv:2509.17357*, 2025. [Online]. Available: https://arxiv.org/abs/2509.17357

[12] R. Zhu, Z. Jiang, C. Jin, P. Wu, C. A. Stuardo, D. Wang, X. Zhang, H. Zhou, H. Wei, Y. Cheng, J. Xiao, X. Zhang, L. Liu, H. Lin, L.-W. Chang, J. Ye, X. Yu, X. Liu, X. Jin, and X. Liu, "MegaScale-Infer: Efficient mixture-of-experts model serving with disaggregated expert parallelism," in *Proceedings of the ACM SIGCOMM 2025 Conference*. ACM, 2025, pp. 592–608.

[13] StepFun, B. Wang, B. Wang, C. Wan, G. Huang, H. Hu *et al.*, "Step-3 is large yet affordable: Model-system co-design for cost-effective decoding," *arXiv preprint arXiv:2507.19427*, 2025. [Online]. Available: https://arxiv.org/abs/2507.19427

[14] Y. Jiang, R. Yan, and B. Yuan, "HexGen-2: Disaggregated generative inference of LLMs in heterogeneous environment," in *Proceedings of the 13th International Conference on Learning Representations (ICLR 25)*, 2025. [Online]. Available: https://proceedings.iclr.cc/paper_files/paper/2025/hash/0b941a1e5fbce23fe46b049999d04ed0-Abstract-Conference.html

[15] NVIDIA, "Parallel Thread Execution ISA, version 9.0," https://docs.nvidia.com/cuda/archive/13.0.0/parallel-thread-execution/index.html, 2025, accessed: Jul. 18, 2026.

[16] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with PagedAttention," in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP 23)*. ACM, 2023, pp. 611–626.

[17] PyTorch Foundation, "PyTorch documentation," https://pytorch.org/docs/stable/, 2026, accessed: Jul. 18, 2026.

[18] Google Cloud, "Accelerator-optimized machine family pricing," https://cloud.google.com/products/compute/pricing/accelerator-optimized, 2026, accessed: Jul. 18, 2026.

[19] NVIDIA, "NVIDIA MGX modular architecture," https://www.nvidia.com/en-us/data-center/products/mgx/, 2026, accessed: Jul. 18, 2026.

[20] Super Micro Computer, Inc., "Supermicro NVIDIA MGX systems," https://www.supermicro.com/zh_cn/accelerators/nvidia/mgx, 2026, accessed: Jul. 18, 2026.

[21] GIGABYTE Technology, "GIGABYTE MGX servers," https://www.gigabyte.com/Enterprise/MGX-Server, 2026, accessed: Jul. 18, 2026.

[22] R. Carratalá-Sáez, F. J. Andújar, Y. Torres, A. González-Escribano, and D. R. Llanos, "Open SYCL on heterogeneous GPU systems: A case of study," *arXiv preprint arXiv:2310.06947*, 2023. [Online]. Available: https://arxiv.org/abs/2310.06947

[23] J. Liu, "Beyond hardware: Building the software foundation for heterogeneous GPU," https://seclee.com/post/202510_hmc/, 2025, accessed: Jul. 18, 2026.

[24] NVIDIA, "cuBLAS: Basic linear algebra on NVIDIA GPUs," https://developer.nvidia.com/cublas, 2025, accessed: Jul. 18, 2026.

[25] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[26] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and memory-efficient exact attention with IO-awareness," in *Advances in Neural Information Processing Systems 35 (NeurIPS 22)*, 2022, pp. 16 344–16 359. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/hash/67d57c32e20fd0a7a302cb81d36e40d5-Abstract-Conference.html

[27] X. Wei, Z. Huang, T. Sun, Y. Hao, R. Chen, M. Han, J. Gu, and H. Chen, "PhoenixOS: Concurrent OS-level GPU checkpoint and restore with validated speculation," in *Proceedings of the 31st ACM Symposium on Operating Systems Principles (SOSP 25)*. ACM, 2025, pp. 996–1013.

[28] PyTorch Foundation, "CUDA semantics: Memory management," https://pytorch.org/docs/stable/notes/cuda.html#memory-management, 2026, accessed: Jul. 18, 2026.

[29] S. Huang and C. Wu, "Neutrino: Fine-grained GPU kernel profiling via programmable probing," in *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. Boston, MA: USENIX Association, Jul. 2025, pp. 331–355. [Online]. Available: https://www.usenix.org/conference/osdi25/presentation/huang-songlin

[30] Y. Zheng, T. Yu, Y. Yang, Y. Hu, X. Lai, D. Williams, and A. Quinn, "Extending applications safely and efficiently," in *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. Boston, MA: USENIX Association, Jul. 2025, pp. 557–574. [Online]. Available: https://www.usenix.org/conference/osdi25/presentation/zheng-yusheng

[31] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30 (NeurIPS 17)*, 2017, pp. 5998–6008. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[32] NVIDIA, "NVIDIA tensorRT," https://developer.nvidia.com/tensorrt, 2025, accessed: Jul. 18, 2026.

[33] NVIDIA, "CUDA Runtime API: Graph management," https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html, 2025, accessed: Jul. 18, 2026.

[34] NVIDIA, "CUDA graph best practices for PyTorch: Handling dynamic patterns," https://docs.nvidia.com/dl-cuda-graph/latest/torch-cuda-graph/handling-dynamic-patterns.html, 2026, accessed: Jul. 18, 2026.

[35] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*. New York, NY: John Wiley & Sons, 1988.

[36] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: Generalized pipeline parallelism for DNN training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 19)*. ACM, 2019, pp. 1–15.

[37] Gurobi Optimization, LLC, "Gurobi Optimizer reference manual," https://docs.gurobi.com/projects/optimizer/en/current/, 2026, accessed: Jul. 18, 2026.

[38] T. Hu, J. Qin, Y. Wang, K. Liu, T. Li, S. Wang, Z. Hu, T. Hu, W. Wang, L. Li, J. Zhou, X. Bao, H. Sun, J. Zhao, H. Cui, T. Xie, and C. Wang, "StrataCL: Fabric-native communication library for production supernodes," *arXiv preprint arXiv:2607.26444*, 2026. [Online]. Available: https://arxiv.org/abs/2607.26444

[39] NVIDIA, "Improving network performance of HPC systems using NVIDIA Magnum IO NVSHMEM and GPUDirect Async," https://developer.nvidia.com/blog/improving-network-performance-of-hpc-systems-using-nvidia-magnum-io-nvshmem-and-gpudirect-async/, 2022, accessed: Jul. 18, 2026.

[40] NVIDIA, "NVIDIA Collective Communications Library (NCCL) documentation," https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/, 2026, accessed: Jul. 18, 2026.

[41] NVIDIA, "CUDA Runtime API: Stream management," https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html, 2026, accessed: Jul. 18, 2026.

[42] T. Hu, C. Wang, T. Cao, J. Qin, L. Chen, X. Xiao, J. Hu, H. Tian, S. Yan, H. Cui, Q. Chen, and T. Xie, "Hummingbird: SLO-oriented GPU preemption at microsecond-scale," *arXiv preprint arXiv:2601.04071*, 2026. [Online]. Available: https://arxiv.org/abs/2601.04071

[43] NVIDIA, "NCCL communicator quality of service," https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/communicators.html, 2026, accessed: Jul. 18, 2026.

[44] H. Liu, M. Zaharia, and P. Abbeel, "RingAttention with blockwise transformers for near-infinite context," in *The Twelfth International Conference on Learning Representations (ICLR)*, 2024. [Online]. Available: https://openreview.net/forum?id=WsRHpHH4s0

[45] NVIDIA, "PTXAS in the CUDA compiler driver," https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/, 2024, accessed: Jul. 18, 2026.

[46] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2009, pp. 44–54.

[47] NVIDIA, "CUDA Programming Guide: CUDA graphs," https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cuda-graphs.html, 2025, accessed: Jul. 18, 2026.

[48] A. Grattafiori *et al.*, "The Llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024. [Online]. Available: https://arxiv.org/abs/2407.21783

[49] OpenAI, "gpt-oss-120b & gpt-oss-20b Model Card," https://openai.com/index/gpt-oss-model-card/, 2025, accessed: Jul. 18, 2026.

[50] Mistral AI, "Codestral mamba," https://mistral.ai/news/codestral-mamba/, 2024, accessed: Jul. 18, 2026.

[51] S. Bai *et al.*, "Qwen2.5-VL technical report," *arXiv preprint arXiv:2502.13923*, 2025. [Online]. Available: https://arxiv.org/abs/2502.13923

[52] P. Esser, S. Kulal, A. Blattmann, R. Entezari, J. Müller, H. Saini, Y. Levi, D. Lorenz, A. Sauer, F. Boesel, D. Podell, T. Dockhorn, Z. English, and R. Rombach, "Scaling rectified flow transformers for high-resolution

image synthesis," in *Proceedings of the 41st International Conference on Machine Learning (ICML 24)*, ser. Proceedings of Machine Learning Research, vol. 235. PMLR, 2024, pp. 12 606–12 633. [Online]. Available: https://proceedings.mlr.press/v235/esser24a.html

[53] Stability AI, "Stable diffusion 3.5 medium model card," https://huggingface.co/stabilityai/stable-diffusion-3.5-medium, 2024, accessed: Jul. 18, 2026.

[54] X. Chen, H. Fang, T.-Y. Lin, R. Vedantam, S. Gupta, P. Dollár, and C. L. Zitnick, "Microsoft COCO captions: Data collection and evaluation server," *arXiv preprint arXiv:1504.00325*, 2015. [Online]. Available: https://arxiv.org/abs/1504.00325

[55] J. Yu, Y. Xu, J. Y. Koh, T. Luong, G. Baid, Z. Wang, V. Vasudevan, A. Ku, Y. Yang, B. K. Ayan, B. Hutchinson, W. Han, Z. Parekh, X. Li, H. Zhang, J. Baldridge, and Y. Wu, "Scaling autoregressive models for content-rich text-to-image generation," *Transactions on Machine Learning Research*, 2022. [Online]. Available: https://openreview.net/forum?id=AFDcYJKhND

[56] K. Zhu, Y. Gao, Y. Zhao, L. Zhao, G. Zuo, Y. Gu, D. Xie, T. Tang, Q. Xu, Z. Ye, K. Kamahori, C.-Y. Lin, Z. Wang, S. Wang, A. Krishnamurthy, and B. Kasikci, "NanoFlow: Towards optimal large language model serving throughput," in *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. Boston, MA: USENIX Association, Jul. 2025, pp. 749–765. [Online]. Available: https://www.usenix.org/conference/osdi25/presentation/zhu-kan

[57] Y. Mei, Y. Zhuang, X. Miao, J. Yang, Z. Jia, and R. Vinayak, "Helix: Serving large language models over heterogeneous GPUs and network via max-flow," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 25)*. ACM, 2025, pp. 586–602.

[58] Z. Mo, J. Liao, H. Xu, Z. Zhou, and C.-Z. Xu, "Hetis: Serving LLMs in heterogeneous GPU clusters with fine-grained and dynamic parallelism," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 25)*. ACM, 2025, pp. 1710–1724.

[59] Z. Tang, Y. Wang, Q. Wang, and X. Chu, "The impact of GPU DVFS on the energy and performance of deep learning: An empirical study," in *Proceedings of the 10th ACM International Conference on Future Energy Systems (e-Energy 19)*. ACM, 2019, pp. 315–325.

[60] A. Yang *et al.*, "Qwen3 technical report," *arXiv preprint arXiv:2505.09388*, 2025. [Online]. Available: https://arxiv.org/abs/2505.09388

[61] DeepSeek-AI *et al.*, "DeepSeek-V3 technical report," *arXiv preprint arXiv:2412.19437*, 2024. [Online]. Available: https://arxiv.org/abs/2412.19437

[62] F. Strati, Z. Zhang, G. Manos, I. Sánchez Périz, Q. Hu, T. Chen, B. Buzcu, S. Han, P. Delgado, and A. Klimovic, "Sailor: Automating distributed training over dynamic, heterogeneous, and geo-distributed clusters," in *Proceedings of the 31st ACM Symposium on Operating Systems Principles (SOSP 25)*. ACM, 2025, pp. 204–220.

[63] T. Um, B. Oh, M. Kang, W.-Y. Lee, G. Kim, D. Kim, Y. Kim, M. Muzzammil, and M. Jeon, "Metis: Fast automatic distributed training on heterogeneous GPUs," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 563–578. [Online]. Available: https://www.usenix.org/conference/atc24/presentation/um

[64] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[65] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 2, pp. 173–193, 2011.

[66] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC 12)*. IEEE, 2012, pp. 1–11.

[67] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting heterogeneity to enhance scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.