Optimizing Deep Learning Models on GPUs: A Cross-Stack Survey from Sampling Kernels to Auto-Parallelized Clusters

Chi Doan
Independent Researcher, Hanoi, Vietnam
chi.doan.research@gmail.com

July 1, 2026 • Preprint (survey / technical report)

Abstract

Modern deep learning workloads, especially large language model (LLM) training and inference, are fundamentally bottlenecked by how efficiently they use graphics processing units (GPUs). The gap between a GPU's theoretical peak throughput and the throughput actually achieved by a naive implementation routinely spans an order of magnitude, and closing that gap has become a distinct field of systems research. This survey organizes that field into a coherent, six-layer taxonomy that spans the entire software stack: (1) operator- and kernel-level optimization, (2) graph- and compiler-level optimization, (3) parallelization and distributed execution, (4) memory optimization, (5) numerical precision and sparsity, and (6) inference-serving and runtime optimization. We use three recent, complementary systems as running exemplars. **Qrita** [1] is a pivot-based Top-k/Top-p sampling kernel that avoids full-vocabulary sorting and delivers up to 2× higher throughput at half the memory of sorting-based samplers; it represents the operator layer. **Mirage** [2] and its successor **MPK** [3] are multi-level superoptimizers that search for and fuse tensor programs, ultimately compiling an entire LLM into a single *megakernel* for 1.2×–6.7× lower latency; they represent the graph and runtime layers. **Alpa** [4] automatically discovers hybrid data, operator, and pipeline parallel strategies for distributed clusters; it represents the parallelization layer. Around these anchors we survey more than fifty additional techniques and systems, including FlashAttention, TVM/Ansor, Triton, TASO, Megatron-LM, ZeRO/DeepSpeed, FSDP, GSPMD, PagedAttention/vLLM, RadixAttention/SGLang, FlashInfer, and modern quantization methods (GPTQ, AWQ, SmoothQuant, FP8). For each layer we identify the underlying performance principle, the search and cost-modeling machinery that powers automation, and the trade-offs that arise when layers are composed. We close with the phase-ordering, determinism, and portability challenges that define the frontier of GPU optimization for deep learning.

Keywords: GPU optimization, deep learning systems, LLM inference, kernel fusion, operator scheduling, superoptimization, automatic parallelization, quantization, memory management, compilers.

1 Introduction

The dominant cost of contemporary machine learning is no longer algorithmic novelty but *execution efficiency on accelerators*. Training a frontier LLM consumes tens of thousands of GPU-years, and serving one at scale is a continuous, latency-sensitive workload measured in millions of tokens per second. In both regimes the GPU is the unit of account, and the central engineering question is the same: *given a fixed model and fixed hardware, how close can we get to the hardware's roofline?*

The answer, for a naive implementation, is "not very close." A straightforward PyTorch model expressed as a sequence of framework operators leaves large amounts of performance on the table. Each operator launches a separate GPU kernel, round-trips its inputs and outputs through high-bandwidth memory (HBM), and is tuned—if at all—for an average case rather than the specific shapes at hand. Meanwhile the arithmetic units sit idle waiting on memory, occupancy is low, and inter-device communication is scheduled pessimistically. The discipline of *GPU optimization for deep learning* exists to recover this lost performance, and over the past decade it has matured from a collection of hand-written CUDA kernels into a layered stack of automated compilers, schedulers, and runtime systems.

1.1 Three exemplars, three layers

This survey is organized around three recent systems that, taken together, illustrate the breadth of the field. They were chosen because each attacks a different layer of the stack and each is state-of-the-art in its niche.

- **Qrita** [1] operates at the *operator/kernel* layer. It replaces the sort-and-filter implementation of Top-k and Top-p truncation—the sampling step executed on every decoded token of every LLM request—with a pivot-based selection algorithm. Two ideas do the work: a Gaussian σ-truncation that shrinks the candidate set using the statistical distribution of the logits, and a quaternary pivot search with duplicate bookkeeping that halves the number of search iterations while guaranteeing *deterministic* output. Implemented in Triton, Qrita reaches up to 2× the throughput of the samplers in vLLM, SGLang, and FlashInfer at half the memory, and is now the default GPU Top-p sampler in vLLM.
- **Mirage** [2] and **MPK** [3] operate at the *graph/compiler and runtime* layers. Mirage is a multi-level superoptimizer that searches a space of μGraphs—representations that jointly describe computation at the kernel, thread-block, and thread levels of the GPU hierarchy—and verifies candidate rewrites for equivalence probabilistically. MPK pushes this to its logical conclusion by fusing an entire LLM's forward pass, including cross-GPU communication, into a single persistent *megakernel*, cutting inference latency by 1.2× to 6.7×.
- **Alpa** [4] operates at the *parallelization* layer. It automates the notoriously manual task of sharding a large model across a cluster by decomposing the search into two levels—an integer-linear-program (ILP) formulation for intra-operator (data and tensor) parallelism nested inside a dynamic program for inter-operator (pipeline) parallelism—so that a single decorator turns single-device code into a distributed training job. Alpa's core auto-sharding algorithm has since been upstreamed into XLA.

These three systems share a design philosophy—*replace expert hand-tuning with automated search over a well-defined space*—but they never touch the same code. Qrita optimizes one operator; Mirage rewrites subgraphs and fuses kernels; Alpa decides how the whole graph is distributed. A production LLM deployment uses all three kinds of optimization simultaneously, which is precisely why a stack-wide taxonomy is useful: it lets us reason about which techniques compose, which conflict, and where the remaining opportunities lie.

1.2 Contributions

1. A **six-layer taxonomy** (Table 2, Figure 1) that classifies GPU optimization techniques for deep learning by the layer of the stack they operate on and the performance principle they exploit.
2. A **unified treatment of more than fifty techniques and systems** beyond the three exemplars, from vendor libraries (cuDNN, CUTLASS) through auto-schedulers (TVM, Ansor), IO-aware kernels (FlashAttention, FlashInfer), automatic parallelizers (Mesh-TensorFlow, GSPMD, Alpa, Unity, Galvatron), memory systems (ZeRO, FSDP, PagedAttention), and quantization methods (GPTQ, AWQ, SmoothQuant, FP8).
3. A **cross-cutting analysis of search and cost modeling** (Section 10), showing that the ILP/DP decomposition of Alpa, the probabilistic-equivalence search of Mirage, and the learned cost models of Ansor are three points in one design space.
4. A discussion of **composition, phase ordering, determinism, and portability** (Section 11) that frames the open problems of the field.

1.3 Scope and method

We focus on techniques that improve wall-clock throughput, latency, or memory footprint of deep learning workloads on NVIDIA-class GPUs, with an emphasis on the transformer and LLM workloads that dominate current practice. We deliberately treat *algorithmic* model compression (pruning, distillation) only where it interacts with GPU execution, and we do not cover accelerator microarchitecture or training-data efficiency.

The three exemplar systems are described from their primary sources [1–4]; the surveyed systems are cited to their originating publications throughout.

2 Background: The GPU Execution Model

To understand why the techniques in this survey work, one must first understand where GPU performance is lost. This section is a compact refresher.

2.1 Hardware hierarchy

A modern data-center GPU is a throughput machine built from dozens to hundreds of *streaming multiprocessors* (SMs). Work is issued as a *grid* of thread blocks; each block is resident on one SM and executes as *warps* of 32 threads in lockstep (SIMT). Performance is governed by a steep memory hierarchy: per-thread registers (fastest, tiny), per-block shared memory / L1 (fast, kilobytes), a chip-wide L2 cache, and off-chip HBM (large, but with latency and bandwidth orders of magnitude worse than registers). Dedicated matrix units—*Tensor Cores*—provide the bulk of arithmetic throughput for the low-precision matrix multiplications at the heart of deep learning, but only if data can be fed to them fast enough. Across GPUs, high-speed interconnects (NVLink within a node, InfiniBand across nodes) carry the collective communication required by distributed execution.

2.2 The roofline and the two regimes

The roofline model [5] captures the first-order constraint: a kernel's achievable performance is bounded by $\min(\pi,\ \beta \cdot I)$, where π is peak compute, β is memory bandwidth, and I is *arithmetic intensity* (FLOPs per byte of memory traffic). Kernels split into two regimes:

- **Compute-bound** (high I): large matrix multiplications, convolutions. Here the win comes from better tiling, higher Tensor-Core utilization, and lower precision. Vendor libraries and template kernels (Section 4) dominate.
- **Memory-bound** (low I): element-wise operations, normalizations, activations, attention with short sequences, and—critically—the token-by-token *decode* phase of LLM inference. Here the win comes from *avoiding memory traffic*: fusing operators so intermediates never reach HBM, and redesigning algorithms (as Qrita does for sampling) so they touch less data.

2.3 Sources of inefficiency

Five recurring inefficiencies motivate essentially every technique in this survey, and it is worth naming them explicitly (Table 1). Each layer of the taxonomy in Section 3 can be read as a family of responses to one or more of them.

3 A Taxonomy of GPU Optimization Techniques

We organize the field into six layers, ordered roughly from the narrowest scope (a single GPU kernel) to the broadest (an entire cluster and serving system). Figure 1 places the three exemplar systems and the major surveyed systems within this stack, and Table 2 summarizes each layer's goal, principle, and representatives. The remaining sections (4–9) treat the layers in turn.

4 Layer 1: Operator- and Kernel-Level Optimization

The lowest layer optimizes individual GPU kernels: the compiled routines that execute a single operation such as a matrix multiply, a normalization, an attention block, or a sampling step. Because a kernel is where FLOPs and bytes are actually spent, this layer sets the performance ceiling for everything above it.

Table 1: Five recurring sources of GPU inefficiency in deep learning and the optimization layers that address them (layer numbers refer to Section 3).

Inefficiency	Description	Layers
Excess memory traffic	Intermediates round-trip through HBM between kernels; low arithmetic intensity starves compute units.	1, 2, 4
Kernel-launch overhead	Thousands of tiny kernels per step; per-launch fixed cost and lost pipelining dominate the decode phase.	1, 2, 6
Low occupancy / poor tiling	Suboptimal block/tile sizes leave SMs and Tensor Cores underutilized for the shapes at hand.	1, 2
Communication stalls	Gradient all-reduce or activation exchange serializes with compute across GPUs.	3
Precision & capacity waste	FP32 where FP16/FP8 would do; dense compute on sparse or low-rank structure; redundant KV storage.	4, 5, 6

4.1 Vendor libraries and template kernels

The historical baseline is a hand-tuned vendor library. **cuDNN** [6] and cuBLAS provide expertly optimized primitives for convolution and matrix multiplication, selected at runtime from a set of pre-tuned implementations. **CUTLASS** [7] exposes the underlying building blocks as C++ templates, letting developers assemble custom high-performance GEMM and convolution kernels with control over tiling, data movement, and Tensor-Core instruction selection. These libraries are extremely fast on the shapes they were tuned for, but they are closed to fusion (a library call is an opaque kernel boundary) and cannot anticipate every shape or operator a new model introduces.

4.2 Kernel programming models: Triton

Triton [8] is a Python-embedded language and compiler that lets developers write GPU kernels at the granularity of *thread blocks* operating on tiles, while the compiler handles intra-block scheduling, memory coalescing, and shared-memory allocation. Triton hit a practical sweet spot: kernels are far easier to write than raw CUDA yet reach a large fraction of hand-tuned performance, which is why it has become the implementation substrate for much recent work—including Qrita, whose entire Top-k/Top-p pipeline is written in Triton.

4.3 IO-aware kernels: FlashAttention

Attention is the canonical example of a memory-bound operator made fast by algorithmic redesign. **FlashAttention** [9] observes that the standard implementation materializes the full $N \times N$ attention matrix in HBM, dominating both memory and time for long sequences. By *tiling* the computation and fusing softmax with the two matrix multiplies—keeping partial results in shared memory and recomputing rather than storing the attention matrix—FlashAttention makes attention IO-aware and sub-quadratic in memory. **FlashAttention-2** [10] improves work partitioning and parallelism, and **FlashAttention-3** [11] exploits asynchrony and low-precision (FP8) on Hopper-class GPUs. The lesson—*fuse to avoid HBM round-trips, recompute cheap quantities rather than store them*—recurs at every layer of this survey.

4.4 Auto-scheduling: TVM and Ansor

Rather than hand-tune each kernel, *auto-schedulers* search the space of loop transformations (tiling, reordering, unrolling, vectorization) automatically. **TVM** [12] pioneered a separation between the mathematical definition of an operator and its *schedule*, then searched schedules with a learned cost model (AutoTVM). **Ansor** [13] removed the need for hand-written schedule templates by sampling from a hierarchical space of

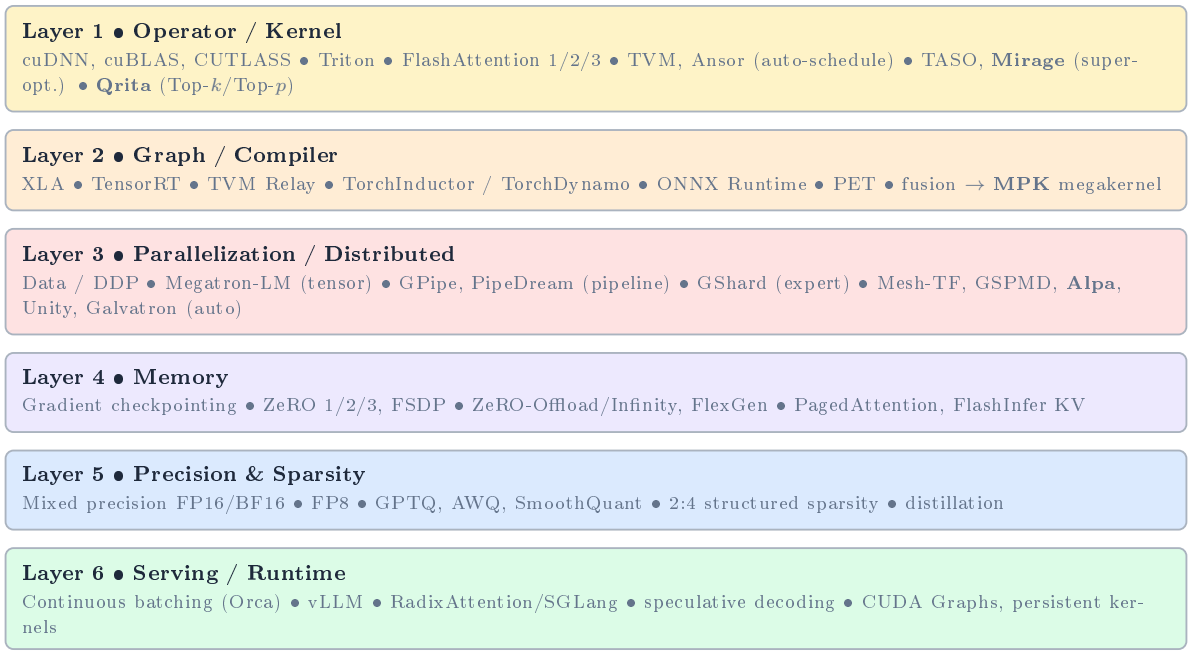


Figure 1: The six-layer taxonomy of GPU optimization for deep learning. Bold entries are the three exemplar systems used throughout this survey; each occupies a different layer (Qrita at the operator layer, Mirage/MPK spanning the graph and runtime layers, and Alpa at the parallelization layer). A production LLM system stacks techniques from every layer.

complete programs and prioritizing promising candidates with a learned model, matching or beating vendor libraries on many operators. Auto-scheduling established the template that later superoptimizers refine: *define a search space, define a cost model, search.*

4.5 Superoptimization: TASO and Mirage

Superoptimization goes beyond scheduling a fixed computation—it *rewrites the computation itself* into an equivalent but faster form, verifying equivalence rather than trusting a fixed rule set. **TASO** [14] automatically generated and verified graph substitutions (*e.g.*, fusing a convolution and a following operation, or reassociating matrix products) for tensor computation graphs, replacing years of manual rewrite-rule engineering. **PET** [15] extended this to *partially equivalent* transformations with automatic correction.

Mirage [2], our second exemplar, is the current state of the art in this line. Its key idea is the μGraph: a representation that describes a tensor program *simultaneously at three levels of the GPU hierarchy*—the kernel level (grids of blocks), the thread-block level (tiles in shared memory), and the thread level (register computation). By searching over μGraphs, Mirage discovers optimizations that cut across these levels—for example, a novel fusion of attention and subsequent operations, or a reorganization of a normalization—that neither a graph-level rewriter nor a single-kernel scheduler could find in isolation. Because the search generates candidate programs aggressively, Mirage verifies each candidate's equivalence to the original *probabilistically*, evaluating both on random inputs in a finite field to make the check tractable while keeping the false-acceptance probability negligible. The result is a superoptimizer that regularly outperforms expert-written and vendor kernels on LLM building blocks.

4.6 Specialized operator algorithms: Qrita

Some operators are best served not by generic search but by a better algorithm. Qrita, our first exemplar, is such a case. The operator is LLM *sampling truncation*: on every decoded token, the model's output logits

Table 2: Overview of the six-layer taxonomy: the goal, the core performance principle, the dominant automation strategy, and representative systems for each layer.

L	Goal	Core principle	Automation	Representatives
1	Fast single kernels	Maximize intensity; exploit memory hierarchy & Tensor Cores	Auto-scheduling, superoptimization	cuDNN, CUTLASS, Triton, FlashAttention, TVM/Ansor, TASO, **Mirage**, **Qrita**
2	Efficient whole graphs	Fuse ops; eliminate HBM round-trips; algebraic rewrites	Pattern rules, e-graphs, search	XLA, TensorRT, TorchInductor, PET, **MPK**
3	Scale across devices	Overlap compute & communication; partition tensors & stages	ILP + DP, cost models	Megatron-LM, GPipe, GShard, GSPMD, **Alpa**, Galvatron
4	Fit in memory	Trade compute/bandwidth for capacity; page & offload state	Static planning, paging	Checkpointing, ZeRO, FSDP, FlexGen, PagedAttention
5	Cheaper arithmetic	Fewer bits; skip zeros; low-rank structure	Calibration, QAT, search	FP16/BF16/FP8, GPTQ, AWQ, SmoothQuant, 2:4 sparsity
6	High-throughput serving	Batch dynamically; cache prefixes; hide launch cost	Schedulers, caches	Orca, vLLM, SGLang, FlashInfer, speculative decoding

over a vocabulary of 10^5 or more tokens must be truncated to the Top-k highest-probability tokens (keep the k largest) and/or to the Top-p nucleus (the smallest set whose cumulative probability exceeds p) [16] before a token is drawn. Existing LLM engines implement this by *sorting the entire vocabulary*, which is exact but wasteful: sorting is $O(V \log V)$, memory-heavy, and a poor fit for the GPU, and it runs in the innermost loop of decoding.

Qrita replaces the sort with *pivot-based selection*, extending an idea (RTop-k) originally developed for node selection in graph neural networks. Two techniques make it both fast and correct:

- **Gaussian σ-truncation.** Logit values across a large vocabulary are approximately normally distributed. Qrita exploits this to compute, from the mean and standard deviation of the logits, a threshold that discards the vast majority of tokens that cannot possibly be in the target set—*before* any search begins—shrinking the candidate space dramatically at negligible cost.
- **Quaternary pivot search with duplication handling.** Rather than a binary search for the cutoff value, Qrita evaluates *four* pivots per iteration, halving the number of iterations needed to converge on the exact threshold. Careful bookkeeping of *duplicate* logit values (ties at the boundary) guarantees that the output is *deterministic*—the same input always yields the same truncated set—which distinguishes Qrita from prior stochastic approximations that trade correctness for speed.

Algorithm 1 sketches the resulting selection procedure. Evaluated against the Top-k/Top-p kernels of vLLM, SGLang, and FlashInfer, Qrita delivers up to 2× the throughput and half the memory of sorting-based samplers while producing *identical* output, and improves end-to-end serving throughput by up to 1.4×. It is now the default GPU Top-p sampler in vLLM. Qrita is a reminder that the operator layer is not only about scheduling known computations faster—sometimes the biggest win is a better algorithm for a step everyone else treated as fixed.

A minimal Triton kernel skeleton for such a fused reduction is shown in Listing 1; the point is that the entire candidate-reduction runs inside one kernel, so the logits are read from HBM once and the truncated set is produced without materializing a sorted copy.

Listing 1: A Triton kernel skeleton: one block reduces a tile of logits, threshold and counts stay in shared memory / registers—no full-vocabulary sort, one HBM read.

Algorithm 1 Pivot-based Top-k selection with Gaussian pre-truncation (Qrita-style sketch)

Require: logits L over vocabulary V; target k; tolerance handling for ties
Ensure: exact set of the k largest logits (deterministic)
$\mu \leftarrow \mathrm{mean}(L)$; $\quad s \leftarrow \mathrm{std}(L)$
$\tau_0 \leftarrow \mu + c\,s$ ▷ Gaussian σ-truncation: coarse threshold
$C \leftarrow \{\, i \in V : L_i \geq \tau_0 \,\}$ ▷ candidate set, $|C| \ll |V|$
while $|C| \neq k$ **do** ▷ converge on exact cutoff
    choose 4 pivots $\pi_1 < \pi_2 < \pi_3 < \pi_4$ spanning the value range of C
    partition C by the pivots; count elements in each bucket (one pass)
    select the bucket whose running count brackets k; $\quad C \leftarrow$ that bucket
    record duplicate values at bucket boundaries ▷ ensures determinism
end while
return C (break boundary ties by recorded order)

```python
import triton
import triton.language as tl

@triton.jit
def topk_reduce(logits_ptr, out_ptr, V, k, BLOCK: tl.constexpr):
    pid = tl.program_id(0)
    offs = pid * BLOCK + tl.arange(0, BLOCK)
    mask = offs < V
    x = tl.load(logits_ptr + offs, mask=mask, other=-float("inf"))

    # Gaussian sigma-truncation: cheap statistics over the tile
    mu = tl.sum(x, axis=0) / V
    var = tl.sum(tl.where(mask, (x - mu) * (x - mu), 0.0), axis=0) / V
    tau = mu + 2.0 * tl.sqrt(var)          # coarse candidate threshold

    keep = x >= tau                        # prune the tile in registers
    # ... pivot-based count/select loop over survivors (deterministic) ...
    tl.store(out_ptr + offs, tl.where(keep, x, -float("inf")), mask=mask)
```

5 Layer 2: Graph- and Compiler-Level Optimization

Where Layer 1 makes a single kernel fast, Layer 2 optimizes the *computation graph* as a whole: the directed acyclic graph of operators that constitutes a model. Its central weapon is **operator fusion**—merging adjacent operators into a single kernel so that intermediate tensors never leave on-chip memory—together with classical compiler techniques: layout assignment, constant folding, common-subexpression elimination, and algebraic simplification.

5.1 Production graph compilers

XLA [17] (the compiler behind JAX and, historically, TensorFlow) performs aggressive fusion of element-wise and reduction operators and is the backend on which Alpa builds. **TensorRT** [18] is NVIDIA's inference compiler, fusing layers, selecting precision, and picking tuned kernels for a target GPU; **TensorRT-LLM** specializes this for transformer inference. In the PyTorch ecosystem, **TorchDynamo** captures Python execution into a graph and **TorchInductor** [19] lowers it—emitting Triton kernels for GPUs—bringing compilation to eager-mode users with minimal code change. **ONNX Runtime** [20] provides a portable graph optimizer across hardware backends. These systems rely largely on *curated rewrite rules*: fast, predictable, but limited to transformations their authors anticipated.

5.2 Search-based graph rewriting

The superoptimizers of Section 4—TASO [14], PET [15], and Mirage [2]—also live partly at this layer, because the substitutions they discover are graph rewrites. The trend they embody is the replacement of hand-curated rule sets with *automatically generated and verified* rewrites, and the erosion of the boundary between "graph optimization" and "kernel optimization": Mirage's μGraphs deliberately span both, because the best fusion often depends on how the fused kernel will be scheduled internally.

5.3 The limit case: whole-model megakernels (MPK)

Fusion has a natural endpoint. If fusing two kernels avoids one HBM round-trip, why not fuse *all* of them? **MPK** (Mirage Persistent Kernel) [3], the successor to Mirage and our runtime-layer exemplar, does exactly this: it compiles an entire LLM's forward pass into a *single persistent megakernel*. Rather than launching thousands of small kernels per decoded token—each with launch overhead and each synchronizing through HBM—MPK expresses the whole model as a task graph executed by a fixed pool of persistent workers and schedulers resident on the SMs, so that computation *and* inter-GPU communication (all-reduce, all-gather) happen inside one kernel launch. The developer defines the model's computation graph in a few dozen lines of Python, chaining fused layers such as a fused RMSNorm-plus-Linear block, and MPK generates the CUDA megakernel. Because the decode phase is dominated by launch overhead and memory-bound element-wise work, this end-to-end fusion cuts inference latency by 1.2×–6.7×. MPK is the clearest demonstration of a theme this survey returns to: the layers are not truly separate, and the most aggressive systems collapse operator, graph, and runtime optimization into one.

Listing 2 illustrates the megakernel programming model at a high level.

Listing 2: Defining an LLM forward pass as a persistent megakernel (MPK-style API). The whole graph is fused; `compile()` emits one CUDA kernel.

```
import mirage as mi

mpk = mi.PersistentKernel(world_size=W, mpi_rank=r,
                          num_workers=96, num_local_schedulers=48,
                          num_remote_schedulers=0,
                          meta_tensors=[step, tokens])

x = mpk.attach_input(torch_tensor=hidden, name="hidden")
mpk.rmsnorm_linear_layer(input=x, weight_norm=w_norm,
                         weight_linear=w_qkv, output=attn_in,
                         grid_dim=(96,1,1), block_dim=(128,1,1))
# ... chain the remaining fused layers of the model ...
mpk.compile()    # generate a single fused CUDA megakernel
mpk()            # run: all compute + communication in one launch
```

6 Layer 3: Parallelization and Distributed Execution

When a model or its activations exceed a single GPU's memory, or when training must finish in reasonable time, work must be split across many GPUs. This layer decides *how*, and its central difficulty is that partitioning introduces communication that can easily erase the benefit of added hardware. Table 3 summarizes the major strategies.

6.1 Manual and semi-automatic parallelism

The workhorses of large-model training are hand-designed. **Megatron-LM** [21] introduced the tensor-parallel partitioning of transformer layers that most large models still use. **GPipe** [22] established synchronous pipeline parallelism with micro-batching, and **PipeDream** [23] added asynchronous and 1F1B scheduling to keep pipeline bubbles small. **GShard** [24] scaled mixture-of-experts models via all-to-all ex-

Table 3: Parallelism strategies for distributed deep learning. Real systems combine several of these; automatic parallelizers such as Alpa search the combined space.

Strategy	What is partitioned	Communication	Representative systems
Data parallel	The batch; each GPU holds a full model replica	All-reduce of gradients per step	PyTorch DDP, Horovod
Tensor / operator (intra-op)	Individual operators (*e.g.* a matmul split across GPUs)	All-reduce / all-gather within a layer	Megatron-LM [21]
Pipeline (inter-op)	The layer stack into stages across GPUs	Point-to-point activations between stages	GPipe [22], PipeDream [23]
Sequence / context	The sequence dimension of activations	Exchange along sequence	Ring/context-parallel attention
Expert (MoE)	Experts of a mixture across GPUs	All-to-all token routing	GShard [24]
Sharded data (ZeRO/FSDP)	Optimizer/gradient/parameter state (Layer 4)	Gather params just-in-time	DeepSpeed ZeRO [30], FSDP [31]

pert routing. Combining these—so-called 3D parallelism—is powerful but requires expert effort to tune for each model and cluster.

6.2 Toward automatic parallelization

A line of work seeks to *automate* the choice of parallel strategy. **Mesh-TensorFlow** [25] let users annotate tensor dimensions for partitioning; **GSPMD** [26] generalized this into an XLA-level sharding system driven by lightweight annotations and propagation. These reduce effort but still lean on human guidance for the overall plan.

6.3 Alpa: two-level automatic parallelization

Alpa [4], our third exemplar, removes the human from the loop by observing that the space of parallel strategies has a natural *hierarchy* that mirrors the hardware. Alpa distinguishes:

- **Intra-operator parallelism**—data and tensor parallelism that partition the axes of individual operators. This maps to tightly-coupled, high-bandwidth device groups (*e.g.* GPUs within a node connected by NVLink), where the frequent collective communication it requires is cheap.
- **Inter-operator parallelism**—pipeline parallelism that assigns disjoint *stages* of the graph to different device groups. This maps to the looser, lower-bandwidth connections *between* nodes, where only stage-boundary activations cross the wire.

The two levels are solved with different tools (Algorithm 2). For a given graph partition into stages and a device assignment, Alpa formulates the best intra-op plan for each stage as an **integer linear program** (ILP) that minimizes compute plus communication cost subject to consistency constraints on tensor shardings. It then wraps this in a **dynamic program** over the space of graph-into-stage partitions and device-group assignments (inter-op), choosing the slicing that minimizes the pipelined end-to-end cost. This decomposition tames a combinatorial space that is intractable to search jointly, and it lets a single `@parallelize` decorator (Listing 3) turn single-device JAX code into a distributed job. Alpa achieves strong scaling on models with billions of parameters, and its auto-sharding core has been merged into XLA.

Listing 3: Alpa turns single-device training into a distributed job with one decorator; the two-level search runs under the hood.

```
import alpa
```

Algorithm 2 Alpa's two-level automatic parallelization (conceptual sketch)

Require: computation graph G; cluster of device groups; cost model $\mathcal{C}$
Ensure: a hybrid inter-/intra-operator parallel execution plan
Inter-op (dynamic program):
for each way of slicing G into pipeline stages $S_1, \ldots, S_m$ **do**
  for each assignment of stages to device groups **do**
    Intra-op (ILP): for each stage S_j, solve

$$\min_{\text{shardings}} \sum_{o \in S_j} \big(\text{compute}(o) + \text{comm}(o)\big) \text{ s.t. sharding consistency}$$

    $\text{cost}(S_j) \leftarrow$ ILP optimum
  end for
  evaluate pipelined cost of this slicing via $\mathcal{C}$ (stage costs + bubbles)
end for
return the slicing + assignment + per-stage shardings of minimum total cost

```
@alpa.parallelize          # auto data + operator + pipeline parallelism
def train_step(state, batch):
    def loss_fn(params):
        out = state.forward(params, batch["x"])
        return jnp.mean((out - batch["y"]) ** 2)
    grads = grad(loss_fn)(state.params)
    return state.apply_gradient(grads)

state = create_train_state()
for batch in data_loader:      # now runs across the whole cluster
    state = train_step(state, batch)
```

6.4 Beyond Alpa

Contemporary and follow-on systems broaden the automatic-parallelization search. **Unity** [27] jointly optimizes algebraic transformations and parallelization in one substitution-based search; **Galvatron** [28] automatically selects among multiple parallelism dimensions for transformer training with a decision-tree/dynamic-programming search. The common thread is the same as at Layers 1–2: a well-defined space, a cost model, and an efficient search—applied here to the placement of computation across a cluster.

7 Layer 4: Memory Optimization

GPU memory is scarce and often the binding constraint: a model may be fast enough but simply not fit. This layer trades compute or bandwidth for capacity.

7.1 Recomputation and state sharding

Gradient checkpointing (activation recomputation) [29] stores only a subset of activations during the forward pass and recomputes the rest during the backward pass, cutting activation memory at the cost of extra FLOPs—a direct compute-for-memory trade. **ZeRO** [30] attacks the memory of the *training state* itself: its three stages progressively shard optimizer state (ZeRO-1), gradients (ZeRO-2), and parameters (ZeRO-3) across data-parallel workers, so that no single GPU holds the full state, while gathering parameters just-in-time for each layer. PyTorch's **FSDP** [31] brings the same fully-sharded data-parallel approach into the core framework. Because these techniques shard *state* rather than *computation*, they compose cleanly with the parallelism of Layer 3.

7.2 Offloading

When even sharding is insufficient, state can be pushed to slower, larger memory. **ZeRO-Offload** and **ZeRO-Infinity** [32] move optimizer state and parameters to CPU RAM and NVMe, enabling models far larger than aggregate GPU memory. For inference, **FlexGen** [33] schedules weights and KV cache across GPU, CPU, and disk to run large models on a single commodity GPU at high throughput for latency-tolerant, batch-oriented workloads. Offloading trades bandwidth (and latency) for capacity, and is most effective when computation can be overlapped with the slower transfers.

7.3 KV-cache management

In autoregressive inference, the dominant memory consumer is the *KV cache*: the keys and values of all past tokens, kept to avoid recomputation. **PagedAttention**, the core of **vLLM** [34], applies the operating-system idea of paging to this cache: it stores KV in fixed-size, non-contiguous *blocks* referenced through a block table, eliminating the fragmentation and over-reservation of contiguous allocation and allowing near-zero-waste memory use and prefix sharing across requests. **FlashInfer** [35] generalizes KV storage into composable block-sparse formats with JIT-compiled attention templates and load-balanced scheduling, and has become a shared attention backend across vLLM, SGLang, and MLC. Since KV-cache management sits at the boundary of memory and serving, it recurs in Layer 6.

8 Layer 5: Numerical Precision and Sparsity

The cheapest FLOP is the one performed in fewer bits—or skipped entirely. This layer reduces the cost of arithmetic by lowering precision or exploiting structural zeros.

8.1 Reduced-precision training and inference

Mixed-precision training [36] performs most computation in FP16 (or BF16) while keeping a master copy of weights and select reductions in FP32, with loss scaling to preserve small gradients—roughly doubling throughput and halving memory on Tensor-Core hardware at no accuracy loss. Newer hardware adds **FP8** formats, exploited for both training and inference (*e.g.* in FlashAttention-3 [11]), pushing the trade further.

8.2 Quantization

Post-training quantization compresses trained weights (and sometimes activations) to 8-bit or 4-bit integers. **GPTQ** [37] uses second-order information to quantize LLM weights to 3–4 bits with minimal accuracy loss. **SmoothQuant** [38] makes *activation* quantization tractable by migrating quantization difficulty from activations to weights, enabling efficient INT8 weight-and-activation inference. **AWQ** [39] protects the salient weight channels that matter most for accuracy, achieving strong 4-bit weight-only quantization. These methods are largely orthogonal to the other layers: a quantized model still benefits from fused kernels, paging, and parallelism.

8.3 Sparsity and structural compression

Pruning removes weights; when the remaining pattern matches hardware support—notably the **2:4 structured sparsity** accelerated by Ampere-and-later Tensor Cores [40]—it yields real speedups rather than merely smaller storage. Knowledge distillation and low-rank adaptation reduce effective model size, and interact with GPU execution by changing the shapes and intensities of the underlying kernels. Precision and sparsity are the layer where *model* and *system* optimization most visibly meet.

9 Layer 6: Inference-Serving and Runtime Optimization

The top layer optimizes the serving system: how many requests share the GPU, how their state is reused, and how the per-token runtime overhead is hidden. This is where several lower-layer techniques—Qrita's sampler, FlashInfer's attention, PagedAttention's KV cache, MPK's megakernel—are actually deployed.

9.1 Batching and scheduling

Static batching wastes capacity when requests have different lengths. **Orca** [41] introduced *continuous* (iteration-level) batching, in which finished sequences leave the batch and new ones join at each decoding step, keeping the GPU saturated. **vLLM** [34] combines continuous batching with PagedAttention to raise serving throughput by an order of magnitude over earlier systems.

9.2 Prefix and prompt reuse

Many requests share prefixes—system prompts, few-shot examples, chat history. **SGLang** [42] introduces **RadixAttention**, which stores the KV cache in a radix tree keyed by token prefix so that shared prefixes are computed once and reused across requests, cutting redundant prefill work substantially for structured and multi-turn workloads.

9.3 Reducing the length of the decode loop

Speculative decoding [43, 44] runs a small "draft" model to propose several tokens that the large model then *verifies* in parallel, accepting a run of tokens per expensive forward pass and thereby amortizing the memory-bound decode step. **Medusa** [45] attaches multiple decoding heads to the model itself to generate and verify candidate continuations without a separate draft model. These methods reduce the *number* of sequential large-model steps.

9.4 Hiding launch and dispatch overhead

Because the decode phase issues many tiny kernels, per-launch overhead is significant. **CUDA Graphs** capture a fixed sequence of kernel launches and replay them as a single unit, removing per-launch CPU cost; **persistent kernels** keep a worker pool resident on the SMs to process a stream of tasks without relaunching. **MPK**'s megakernel [3] (Section 5.3) is the extreme form of this idea, and Qrita's fused sampler [1] removes one of the small, memory-bound kernels that would otherwise punctuate every decode step. The serving layer is thus the integration point where the whole stack's optimizations are composed into a single deployed system.

10 Cross-Cutting Concern: Search, Cost Models, and Autotuning

A striking commonality runs through the exemplars and much of the surveyed work: modern GPU optimization is increasingly *search over a defined space guided by a cost model*, rather than fixed heuristics. The differences between systems are largely differences in (i) what the search space is, (ii) how correctness of a candidate is guaranteed, and (iii) how cost is estimated. Table 4 places the three exemplars and several surveyed systems in this design space.

Three observations follow. First, *correctness guarantees trade against search-space size*: Alpa's plans are semantics-preserving by construction so it can search freely; Mirage generates such aggressive rewrites that it must fall back to probabilistic verification; Qrita, operating on a single well-understood operator, can afford an *exact and deterministic* guarantee, which is precisely its selling point over stochastic samplers. Second, *cost models range from closed-form to learned*: Qrita's Gaussian model of the logit distribution is analytic; Alpa's cost model sums analytical communication and compute terms; Ansor learns its model from measurements. Third, *decomposition is what makes search tractable*: Alpa's nesting of ILP inside DP, and

Table 4: GPU optimization as search: the space explored, the search strategy, the correctness guarantee, and the cost model, across representative systems. The three exemplars occupy distinct cells of one design space.

System	Search space	Strategy	Correctness	Cost model
Ansor [13]	Loop schedules of one operator	Evolutionary + sampling	Exact (same computation)	Learned
TASO [14]	Graph substitutions	Enumerate + verify	Formal equivalence	Analytical
Mirage [2]	μGraphs across kernel/block/thread levels	Guided generation	*Probabilistic* equivalence	Analytical + profiled
Qrita [1]	Pivot thresholds for Top-k/Top-p	Multi-pivot (quaternary) search	*Exact*, deterministic	Closed-form (Gaussian)
Alpa [4]	Inter-op stages $\times$ intra-op shardings	DP over ILP	Semantics-preserving	Analytical (comm. + compute)
Galvatron [28]	Parallelism-dimension combinations	Dynamic programming	Semantics-preserving	Analytical

Qrita's pre-truncation before pivot search, are both moves that shrink an intractable space into a searchable one. Recognizing GPU optimization as a search problem clarifies why advances often come from *better spaces and cheaper verification*, not just faster kernels.

11 Discussion: Composition, Trade-offs, and Open Challenges

11.1 The layers compose—but not freely

The practical power of the taxonomy is that a real deployment stacks techniques from every layer: a model is quantized (Layer 5), fused and compiled (Layers 1–2), parallelized across a cluster (Layer 3) with sharded and paged memory (Layer 4), and served with continuous batching and an optimized sampler (Layer 6). Yet composition is not free. A choice at one layer constrains another: an aggressive Layer-3 sharding changes the operator shapes that Layer-1 auto-scheduling must tune; a Layer-2 megakernel that fuses everything (MPK) leaves little for a separate Layer-6 batching scheduler to do and must therefore absorb that responsibility. This is the classic **phase-ordering problem** of compilers, now spread across an entire systems stack, and it is largely unsolved: today's systems optimize each layer in sequence with hand-designed interfaces between them, and joint optimization across layers—as Unity [27] attempts for algebra and parallelism, or as Mirage attempts across kernel and block levels—remains an active frontier.

11.2 Determinism, reproducibility, and correctness

Performance optimizations frequently perturb numerics or ordering, and the field is increasingly aware that *correctness is a feature*. Qrita's explicit guarantee of deterministic, sort-identical output [1] is a direct response to earlier samplers that bought speed with stochastic approximation; Mirage's probabilistic equivalence check [2] is an engineered bound on the risk that aggressive rewriting introduces. As optimizations grow more aggressive, principled correctness guarantees—formal, probabilistic, or exact-by-construction—become part of the value proposition rather than an afterthought.

11.3 Portability and the cost of automation

Hand-tuned kernels are fast but brittle: a new GPU generation, a new attention variant, or a new sparsity pattern can invalidate months of work. The automated systems surveyed here—auto-schedulers, superoptimizers, auto-parallelizers—exist largely to make performance *portable* across hardware and models by re-searching rather than re-engineering. The cost is search time and the difficulty of building accurate cost models; the benefit is that a single system such as Alpa or Mirage adapts to new targets with far less human

effort. The trajectory of the field is clearly toward more automation, with expert kernels reserved for the few hottest operators.

11.4 Open challenges

We highlight four. (1) **Cross-layer joint optimization**: principled co-optimization of parallelism, fusion, memory, and precision, rather than sequential per-layer passes. (2) **Search cost**: making superoptimization and auto-parallelization fast enough to run routinely, perhaps with transferable or learned cost models. (3) **Correctness at scale**: extending exact or probabilistic guarantees to the whole optimized stack. (4) **Hardware co-evolution**: FP8, structured sparsity, and megakernel-friendly scheduling change the target underneath every technique, so the taxonomy's contents will keep shifting even as its structure endures.

12 Conclusion

Getting deep learning to run efficiently on GPUs is a stack-wide effort, and this survey has organized that effort into six layers—operator, graph, parallelization, memory, precision, and serving—unified by a common methodology of *search over a defined space under a cost model with an explicit correctness guarantee*. Our three exemplars make the breadth concrete: Qrita [1] rethinks a single sampling operator and wins 2× throughput with an exact, deterministic algorithm; Mirage and MPK [2, 3] search and fuse tensor programs until an entire model becomes one megakernel, cutting latency several-fold; and Alpa [4] automates the partitioning of a model across a cluster with a two-level ILP-in-DP search. Around them, dozens of systems—FlashAttention, TVM/Ansor, Megatron-LM, ZeRO/FSDP, PagedAttention, SGLang, FlashInfer, and modern quantization—fill out the stack. The layers compose in every production system yet interact in ways that current tools handle only sequentially, leaving cross-layer joint optimization, cheaper search, end-to-end correctness guarantees, and hardware co-design as the defining challenges of the next several years of GPU optimization for deep learning.

References

[1] J. Park, S. Kim, A. Cheung, and I. Stoica. Qrita: High-performance Top-k and Top-p algorithm for GPUs using pivot-based truncation and selection. *arXiv preprint arXiv:2602.01518*, 2026.

[2] M. Wu, X. Cheng, S. Liu, C. Shi, J. Ji, K. Ao, P. Velliengiri, X. Miao, O. Padon, and Z. Jia. Mirage: A multi-level superoptimizer for tensor programs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2025. arXiv:2405.05751.

[3] X. Cheng, Z. Zhang, Y. Zhou, J. Ji, et al. MPK: A compiler and runtime for mega-kernelizing tensor programs. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2026. arXiv:2512.22219.

[4] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, J. E. Gonzalez, and I. Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[5] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[6] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[7] NVIDIA. CUTLASS: CUDA templates for linear algebra subroutines. https://github.com/NVIDIA/cutlass, 2017–.

[8] P. Tillet, H. T. Kung, and D. Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *ACM SIGPLAN Workshop on Machine Learning and Programming Languages (MAPL)*, 2019.

[9] 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 (NeurIPS)*, 2022. arXiv:2205.14135.

[10] T. Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[11] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao. FlashAttention-3: Fast and accurate attention with asynchrony and low-precision. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

[12] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[13] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[14] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken. TASO: Optimizing deep learning computation with automatic generation of graph substitutions. In *27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[15] H. Wang, J. Zhai, M. Gao, Z. Ma, S. Tang, L. Zheng, Y. Li, K. Rong, Y. Chen, and Z. Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[16] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations (ICLR)*, 2020.

[17] Google. XLA: Optimizing compiler for machine learning. https://openxla.org/xla, 2017–.

[18] NVIDIA. TensorRT: A high-performance deep learning inference optimizer and runtime. https://developer.nvidia.com/tensorrt, 2016–.

[19] J. Ansel, E. Yang, H. He, N. Gimelshein, et al. PyTorch 2: Faster machine learning through dynamic Python bytecode transformation and graph compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[20] ONNX Runtime developers. ONNX Runtime: Cross-platform, high-performance ML inferencing and training accelerator. https://onnxruntime.ai, 2018–.

[21] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[22] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen. GPipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[23] 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 *27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[24] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen. GShard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[25] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. Hechtman. Mesh-TensorFlow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[26] Y. Xu, H. Lee, D. Chen, B. Hechtman, Y. Huang, R. Joshi, M. Krikun, D. Lepikhin, A. Ly, M. Maggioni, et al. GSPMD: General and scalable parallelization for ML computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

[27] C. Unger, Z. Jia, W. Wu, S. Lin, M. Baines, C. E. Q. Narvaez, V. Ramakrishnaiah, et al. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[28] X. Miao, Y. Wang, Y. Jiang, C. Shi, X. Nie, H. Zhang, and B. Cui. Galvatron: Efficient transformer training over multiple GPUs using automatic parallelism. *Proceedings of the VLDB Endowment*, 16(3):470–479, 2022.

[29] T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[30] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. ZeRO: Memory optimizations toward training trillion parameter models. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[31] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer, et al. PyTorch FSDP: Experiences on scaling fully sharded data parallel. *Proceedings of the VLDB Endowment*, 16(12):3848–3860, 2023.

[32] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He. ZeRO-Infinity: Breaking the GPU memory wall for extreme scale deep learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

[33] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, D. Y. Fu, Z. Xie, B. Chen, C. Barrett, J. E. Gonzalez, P. Liang, C. Ré, I. Stoica, and C. Zhang. FlexGen: High-throughput generative inference of large language models with

a single GPU. In *International Conference on Machine Learning (ICML)*, 2023.

[34] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with PagedAttention. In *29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[35] Z. Ye, L. Chen, R. Lai, W. Lin, Y. Zhang, S. Wang, T. Chen, B. Kasikci, V. Grover, A. Krishnamurthy, and L. Ceze. FlashInfer: Efficient and customizable attention engine for LLM inference serving. In *Proceedings of Machine Learning and Systems (MLSys)*, 2025. arXiv:2501.01005.

[36] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. Mixed precision training. In *International Conference on Learning Representations (ICLR)*, 2018.

[37] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh. GPTQ: Accurate post-training quantization for generative pre-trained transformers. In *International Conference on Learning Representations (ICLR)*, 2023.

[38] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning (ICML)*, 2023.

[39] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han. AWQ: Activation-aware weight quantization for LLM compression and acceleration. In *Proceedings of Machine Learning and Systems (MLSys)*, 2024.

[40] A. Mishra, J. A. Latorre, J. Pool, D. Stosic, D. Stosic, G. Venkatesh, C. Yu, and P. Micikevicius. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378*, 2021.

[41] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun. Orca: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[42] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. arXiv:2312.07104.

[43] Y. Leviathan, M. Kalman, and Y. Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning (ICML)*, 2023.

[44] C. Chen, S. Borgeaud, G. Irving, J.-B. Lespiau, L. Sifre, and J. Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

[45] T. Cai, Y. Li, Z. Geng, H. Peng, J. D. Lee, D. Chen, and T. Dao. Medusa: Simple LLM inference acceleration framework with multiple decoding heads. In *International Conference on Machine Learning (ICML)*, 2024.