High-Performance Machine Learning (HPML)

Fall 2026 Syllabus: Columbia University, Department of Computer Science

Instructor

Dr. Kaoutar El Maghraoui, Adjunct Professor of Computer Science, Principal Research Scientist, IBM T.J. Watson Research Center, NY

Course Description

Modern AI runs on the ability to train and serve very large models efficiently. That ability is now the difference between a research idea and a deployable product. High-Performance Machine Learning (HPML) sits at the intersection of AI and High-Performance Computing (HPC), where algorithms, software, and hardware are co-designed for speed, scale, and efficiency.

This course covers the foundational and advanced techniques behind efficient AI systems, with an emphasis on modern foundation-model systems and the computational demands of training, post-training, inference, and serving. You will learn to optimize AI performance across both training and inference through topics including:

- **Performance engineering and profiling:** reproducible benchmarking; GPU-aware timing; bottleneck analysis with Amdahl's Law and the Roofline model; profiling and tracing tools.
- **GPU programming:** CUDA and Triton, memory hierarchy, and kernel optimization.
- **Distributed and large-scale training:** DDP; FSDP2 and ZeRO; tensor, pipeline, sequence, context, and expert parallelism; DeviceMesh/DTensor concepts; activation and distributed checkpointing.
- **Model compression and reduced precision:** numeric formats and quantization algorithms; FP8/FP4; weight-only, weight-and-activation, and KV-cache quantization; pruning, sparsity, and knowledge distillation.
- **Efficient post-training:** LoRA/QLoRA and adapters; DPO/GRPO/RLHF from a systems perspective, including rollout throughput, memory footprint, and time-to-target quality.
- **Efficient transformer execution:** FlashAttention, GQA/MQA, KV-cache optimization, Mixture of Experts, and speculative decoding.
- **Production serving:** vLLM, PagedAttention, continuous batching, chunked prefill, prefix caching, KV-cache management, prefill/decode disaggregation, speculative decoding, multi-tenancy, latency SLOs, and cost/energy efficiency.

By the end of the course you will have hands-on experience with these techniques and be able to design, implement, measure, and optimize scalable AI systems that balance performance, energy efficiency, cost, and model quality. The course carries an explicit industrial lens: where each technique appears in real training and serving stacks, what bottleneck it addresses, and what trade-offs it introduces.

The course is based on PyTorch, CUDA, and Triton.

Objectives

1. Identify and resolve performance bottlenecks in AI software using HPC techniques.
2. Perform rigorous and reproducible performance measurement and profiling, including correct GPU timing, warm-up, variability, and hardware/software configuration reporting.
3. Critically compare AI software stacks and hardware platforms to find efficient configurations.
4. Use distributed training and sharding abstractions to scale models across many GPUs and nodes, and reason about memory, communication, and topology trade-offs.
5. Apply model compression (quantization, pruning, knowledge distillation) to improve efficiency with minimal accuracy loss.
6. Implement and optimize efficient LLM training/post-training and inference using compiler, kernel, parallelism, quantization, attention, KV-cache, and serving techniques.
7. Analyze emerging AI hardware and systems trends using workload-aware metrics such as utilization, MFU, latency, throughput/goodput, tokens per joule, and cost per token.
8. Validate that performance optimizations preserve numerical correctness or task/model quality, and explain why an optimization succeeds or fails.

Prerequisites

- General knowledge of computer architecture and operating systems.
- Intermediate C/C++ programming.
- Intermediate Python programming.
- A good understanding of neural network algorithms.

The course focuses on model performance rather than model algorithms. A high-level review of the algorithms is included, but you should arrive comfortable with logistic regression, feed-forward networks, CNNs, RNNs, and transformer architectures.

Course Materials

There is no required textbook. Pointers to papers, current systems documentation, and web resources are provided in class. Because the ML systems stack evolves quickly, readings may be updated during the semester while the core learning objectives remain fixed. Useful references include:

- *Introduction to High-Performance Computing for Scientists and Engineers*, Hager and Wellein, CRC Press.
- *Introduction to High-Performance Scientific Computing* (online), Eijkhout, Chow, van de Geijn.
- *Computer Architecture: A Quantitative Approach*, 6th ed., Hennessy and Patterson, Morgan Kaufmann.

- *Efficient Processing of Deep Neural Networks*, Sze, Chen, Yang, Emer, Morgan & Claypool.
- Living references: current PyTorch compiler/distributed documentation, NVIDIA CUDA programming and performance guides, Triton documentation, and current LLM serving-system documentation (for example vLLM/SGLang).

Topics Covered

- ML/DL and PyTorch basics
- Performance engineering methodology, reproducible benchmarking, profiling/tracing, Amdahl's Law, and the Roofline model
- Python and PyTorch performance internals; torch.compile (Dynamo, AOTAutograd, Inductor), graph breaks, guards, recompilation, dynamic shapes, and compile-time vs steady-state trade-offs
- GPU programming with CUDA and Triton
- GPU memory hierarchy; memory coalescing, tiling, fusion and occupancy; math/kernel libraries (cuBLAS/cuDNN and related libraries)
- Distributed training: data parallelism, DDP, NCCL, all-reduce
- LLM-scale parallelism: FSDP2/ZeRO; DeviceMesh/DTensor concepts; tensor/pipeline/sequence/context/expert parallelism; activation checkpointing and distributed checkpointing
- Quantization and reduced precision: FP16/BF16/FP8/FP4 and microscaling formats; PTQ/QAT; GPTQ/AWQ/SmoothQuant; weight-only, W+A, and KV-cache quantization; hardware/kernel support
- Sparsity, pruning, and knowledge distillation, with emphasis on when algorithmic sparsity produces real hardware speedup
- Efficient post-training: LoRA/QLoRA, adapters, DPO, GRPO/RLHF; systems cost of rollouts, optimizer/adapter state, and time-to-target quality
- Efficient transformer execution: FlashAttention, GQA/MQA, KV-cache optimization and quantization, and long-context memory behavior
- Mixture of Experts: routing, load balance, expert parallelism, communication, and serving implications
- LLM serving: vLLM/PagedAttention, continuous batching, chunked prefill, prefix caching, KV-cache eviction/offload/quantization, disaggregated serving, speculative decoding, multi-tenancy, TTFT/ITL/tail latency/goodput, and cost/energy per token
- Hardware acceleration for AI and emerging accelerators

Deliverables and Grading

Course-wide performance workflow: Baseline → Measure → Identify the bottleneck → Form a hypothesis → Optimize → Re-measure → Validate correctness/quality → Explain the result.

Homework and projects are expected to follow this workflow.

Final Project 40% + Homework 40% + Quizzes 15% + Participation 5%

- **Final Project:** teams of 3 to 4, and the course's summative assessment. Proposal (Week 4), mid-point report (Week 9), final report and presentation (finals period). Evaluation emphasizes a reproducible baseline, profiling/bottleneck diagnosis, technically justified optimization, correctness or quality validation, quantitative results, and explanation of trade-offs. Includes an individual-accountability component: a per-student contribution statement and individual technical Q&A at the presentation.
- **Homework:** 4 graded assignments, mostly programming and GPU experiments in Python, PyTorch, CUDA, and Triton, plus an ungraded HW0 environment/GPU setup check.
- **Quizzes:** short individual checks distributed across the semester and spanning every module. The optional final quiz can replace the lowest required quiz score.
- **No sit-down final exam.** The finals period is used for project presentations (see below).

Late Homework Policy

Quizzes and project submissions must be on time; late submissions receive zero credit unless an approved accommodation applies. Each student has 6 "late days" usable only for graded homework. After the 6 late days are used, 20% of the total marks are deducted per additional late day up to 5 days; beyond that the assignment receives zero. Late days are counted daily, with a new day starting at 11:59 PM ET.

Course Project

- Proposals due Week 3-4.
- Mid-point report due Week 9.
- Final report due at the end of the Week 13 (Dec 14) week.
- Presentations during the finals period (Dec 17-23): 3-4 days of parallel sessions, with 15-minute slots per team (approximately 10 minutes presentation + 5 minutes technical Q&A).

Project performance claims must report the workload, hardware/software configuration, dtype/precision, batch and sequence/context sizes when relevant, warm-up methodology, repetitions/variability, baseline, and correctness or model-quality criterion.

Weekly Lesson Plan

Module I: Foundations and Performance Engineering

Week 1: Why High-Performance ML? From Models to Systems
Course organization; why efficiency now (scale, cost, energy and latency); the four performance walls (compute, memory/data movement, communication, power/cost); modern AI system anatomy (accelerator, HBM, host, network, storage, software stack); training vs inference; prefill/decode and a first KV-cache sizing example; HPC principles reinterpreted for modern AI.

Week 2: Performance Engineering + PyTorch Fundamentals
Baseline-measure-analyze-optimize methodology; latency, throughput, speedup, scalability and reproducibility; correct GPU timing and warm-up; Amdahl's Law; critical path and bottleneck analysis; latency vs bandwidth; arithmetic intensity and Roofline with both HPC and ML examples; PyTorch tensors, dtype/device/layout, autograd and computation graphs.

Week 3: PyTorch Runtime and Profiling
Python interpreter/runtime overhead and the C++/CUDA core; vectorization and kernel-launch overhead; torch.profiler and GPU timeline interpretation; data pipeline performance (Dataset/DataLoader, workers, pinned memory, prefetching and asynchronous H2D transfer); optimizer systems cost with emphasis on SGD/momentum and AdamW; time-to-target quality vs steps/second.

Week 4: PyTorch Compilation with torch.compile
Eager vs graph execution; Dynamo, AOTAutograd and Inductor; graph capture and fusion; graph breaks, guards, recompilation and dynamic shapes; compile cold-start vs steady-state runtime; debugging and deciding when compilation helps or hurts. Triton is introduced as Inductor's common GPU code-generation target and opened in Week 6.

Module II: GPU and Distributed Computing

Week 5: CUDA Basics
Heterogeneous execution and GPU architecture; CUDA compilation/runtime; grid-block-thread programming model; warps and SIMT intuition; global-memory access and coalescing basics; synchronization; naive matrix multiplication and first-principles performance measurement.

Week 6: Advanced CUDA and Triton
Global/shared/register/cache hierarchy; tiled matrix multiplication; memory coalescing and bank conflicts; warp scheduling and occupancy; streams and asynchronous execution; kernel/math libraries; Triton's program-over-blocks model; writing a tiled kernel; comparing CUDA, Triton and library implementations using measured performance.

Week 7: Distributed Deep Learning I: Data Parallelism
Data/model/hybrid parallelism; synchronous training, stragglers and scaling efficiency; PyTorch DDP; collectives and ring/tree all-reduce intuition; NCCL; gradient bucketing and communication/computation overlap; topology-aware scaling.

Week 8: LLM-Scale Parallelism
Why DDP alone is insufficient for large models; ZeRO and FSDP2; DeviceMesh/DTensor concepts; tensor, pipeline, sequence, context and expert parallelism; activation checkpointing/rematerialization; distributed checkpointing; memory/communication trade-offs and composing multiple parallel dimensions; modern training-at-scale case studies.

Module III: Model Efficiency and Post-Training

Week 9: Quantization and Reduced Precision

Separate numeric formats from quantization algorithms and kernel support; FP16/BF16/FP8/FP4 and microscaling; calibration, PTQ and QAT; weight-only vs weight-and-activation quantization; GPTQ, AWQ and SmoothQuant; KV-cache quantization; hardware-aware evaluation of speed, memory and quality.

Week 10: Sparsity, Pruning and Knowledge Distillation

Weight/activation sparsity; structured vs unstructured pruning; 2:4 sparsity; SparseGPT/Wanda; when sparsity does and does not translate to end-to-end speedup; knowledge distillation for efficient models; NAS retained as optional/reference material.

Week 11: Efficient Post-Training: LoRA/QLoRA and RL

LoRA/QLoRA theory and implementation; adapters and prompt tuning; DPO and GRPO/RLHF overview; systems view of rollout generation, actor/reference/reward-model memory, batching, communication and inference throughput; connecting post-training with compression and serving constraints.

Module IV: Efficient LLM Execution and Serving

Week 12: Efficient Transformer Execution: Attention, KV Cache and MoE

Transformer/attention performance recap; FlashAttention and IO-aware attention; GQA/MQA; KV-cache sizing, precision and memory management; Mixture of Experts routing, load balance and expert parallelism; compute-vs-communication trade-offs with modern case studies.

Week 13: LLM Serving Systems

Serving metrics (TTFT, ITL/TPOT, throughput, tail latency, goodput, tokens/J and cost/token); vLLM/PagedAttention; continuous batching; chunked prefill; prefix caching; KV-cache eviction/offload/quantization; prefill/decode disaggregation; speculative decoding; multi-tenancy; comparison of serving stacks and SLO-driven production deployment.