Interview Project Notes (45-min Ready)

Minimal talk-tracks + core concepts + expected questions

Key Idea

Need-to-knows (Performance + Scaling Cheat Sheet)

CPU architecture:

- A small number of cores, each with its own L1 cache and control logic attached.
- L2/L3 caches sit outside / above the cores (shared at higher levels).

GPU architecture:

- Many cores (lots of parallel lanes).
- L1 cache + shared memory close to execution units; L2 cache shared; DRAM = high bandwidth memory (HBM).
- Optimized for moving data fast and doing lots of arithmetic per unit time.

- **Operational Intensity (OI)** (a.k.a. arithmetic intensity):

$$\text{OI} = \frac{\text{FLOPs}}{\text{Bytes moved to/from memory}}$$

Key Idea

Roofline intuition: achievable performance is limited by either compute or memory bandwidth.

$$\text{Perf} \leq \min\Big(\text{Peak FLOPs/s},\ \text{OI} \times \text{Peak BW}\Big)$$

How to interpret:

- Low OI → **memory-bound** (limited by bandwidth). Fixes: reuse data (tiling/fusion), reduce reads/writes, improve locality, mixed precision.
- High OI → **compute-bound**. Fixes: use vectorization/tensor cores, improve occupancy, reduce divergence, improve instruction mix.

Regime	What you see	What you do
Compute-bound	high SM active, BW not maxed, kernels dominate runtime	increase tensor core use, fuse kernels, improve occupancy, reduce divergence
Memory-bound	BW near peak, moderate SM active, lots of stalls	tiling, shared memory/cache reuse, coalescing, fewer global reads/writes
I/O-bound	GPU idle, data loader maxed, disk/network busy	caching, preprocessing offline, parallel loading, faster storage/network

- **In HPC/AI infra, performance issues usually fall into 3 buckets:**

1. **Compute-bound**
   - **Signs:** GPU utilization is high (often 90–100%), memory bandwidth is **not** maxed, kernels take most of runtime, pipeline is keeping up.
   - **Tools:**
     * **Nsight Compute (NCU):** `ncu --set full ./your_app` (**INSERT complete**).
     * **Nsight Systems (NSYS):** `nsys profile ./your_app` (timeline + CPU/GPU overlap).
     * **PyTorch Profiler:** `torch.profiler` for operator-level breakdown.
     * **CPU:** `top`, `htop`.
     * **Cluster:** Slurm job stats (e.g., `squeue`, `sstat`, `sacct`).
   - **Solutions (systems):** kernel fusion, better parallelism, vectorization (warps/TC), reduce divergence, tune block sizes/occupancy.
   - **Solutions (ML):** mixed precision (FP16/BF16/INT8), quantization, pruning, and batch tuning (bigger batch → faster but more memory; smaller batch → slower but safer memory).
2. **Memory-bound**
   - **Signs:** GPU utilization is moderate, memory bandwidth is near peak, kernels stall on memory; profilers show memory bottlenecks (cache misses, uncoalesced access, large tensors moving around).
   - **Where common:** CNNs and Transformers can be memory-heavy; runtime is often dominated by tensor movement.
   - **Solutions:**
     * **Coalescing:** make neighboring threads access neighboring memory (e.g., column-wise indexing like `col = baseCol + threadIdx.x`).
     * **Reuse:** shared memory / cache tiles, blocking/tiling.
     * **Reduce traffic:** avoid unnecessary copies, fuse ops, reduce precision.
     * **ML tricks:** gradient checkpointing, activation recomputation, better batching.
3. **I/O-bound**
   - **Signs:** GPU is idle waiting for data, disk/network activity is high, training pauses intermittently, CPU data loader is maxed.

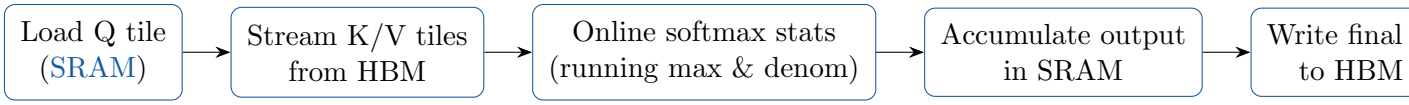
  - **Tools:** `iostat`, `iftop/nload` (network), `nvidia-smi` utilization drops, Slurm job stats.
  - **Solutions:**
    * dataset caching, preprocessing offline, parallel data loaders
    * faster storage (SSD/NVMe), reduce contention
    * distributed: InfiniBand/RDMA, dataset sharding, node-local caching/sharing

Important

Scaling warning: adding more nodes can make training *slower* if communication dominates (gradient sync/all-reduce), network latency/bandwidth limits, suboptimal collective algorithms, synchronization costs, or input-pipeline bottlenecks.

- **Transformer attention (Q, K, V) — in plain words (preserved + cleaned):**
  When the model reads each token, it creates Query (Q), Key (K), and Value (V) vectors. Keys and values help the model store what this token means. During attention, the current token's query is compared against other tokens' keys to decide what context matters. The model then uses the matched values to pass forward the information that matters. This is how the model tracks context and relationships across a sentence or conversation.

- **FlashAttention (tile-wise outline):** Process keys/values in tiles, maintain running softmax statistics, and accumulate the output in SRAM.



Key Idea

Why it works: keep tiles in fast memory (SRAM), do more compute per byte loaded, and avoid materializing the full attention matrix in HBM.

- **FP32 fallback problem (why quantization/low-precision is tricky):**
  Some operations are sensitive and often require higher precision to stay numerically stable:
  - softmax
  - normalization layers
  - attention score calculations
  - reductions

- **Memory layout changes (example: INT4):**
  INT4 means two weights per byte. That forces you to:
  - pack/unpack bits
  - align memory correctly

  - avoid warp divergence (branchy unpack logic can hurt)

- **Warp divergence (clean definition):**
  Warp divergence occurs when threads within the same warp (typically 32 threads) follow different execution paths due to branches (`if/else`, loops). The GPU serializes the paths (masking off inactive threads), which reduces performance.

- **If code is slow on a cluster, check:**
  - network filesystem latency
  - improper CPU/GPU allocation (pinning, affinity, data loader threads)
  - environment differences (drivers, CUDA version, library builds)
  - container overhead / misconfiguration
  - I/O contention (shared storage hot spots)

- **GPU memory hierarchy (fast recall):**
  GPUs have fast on-chip memory (registers, shared memory) and slower global memory (HBM/DRAM). Efficient kernels maximize locality and minimize global memory access.

- **Scheduler vs dispatcher:**
  Scheduler picks which warp runs; dispatcher issues the instruction to the pipelines/cores.

- **Large blocks vs small blocks (tradeoffs):**

Aspect	**Large blocks**	**Small blocks**
Reuse / shared memory	Often higher (good tiling)	Often lower
Latency hiding	Can be worse if fewer warps fit	Often better (more runnable warps)
Global memory traffic	Often lower (reuse)	Often higher
Occupancy	May drop (regs/shared mem limit blocks)	May rise (more blocks fit)

- **GPU terms (in your words, clarified):**

Term	**Meaning (your wording, clarified)**
Block	A block of threads/tasks scheduled onto one SM (like a "work unit").
Warp	Fixed group of 32 threads (SIMT unit) scheduled together.
Warp scheduler	Chooses which warp issues instructions each cycle.
Dispatcher	Sends the chosen warp's instruction into the pipelines/cores.
GigaThread engine	Higher-level manager coordinating blocks across SMs (global scheduling).

- **Zooming out: whole GPU picture (Ampere/Hopper-class systems):**
  - **PCIe:** host interface (bus moving data between host and device).

– **NVLink:** high-bandwidth interconnect between GPUs.

Latency comparison (slow → fast):
Global memory < L2 < L1 / Shared < Registers < FMA < Tensor Cores

> **Key Idea**
>
> **Practical takeaway:** hide memory latency and network communication (often the slow parts).

- **Arithmetic intensity (repeat, because it's that important):**
  Arithmetic intensity = FLOPs / bytes moved over HBM (device memory). Compute via profiling and compare to hardware:
  - If workload ratio < hardware ratio ⇒ not compute-bound (likely memory or network-bound).
  - If workload ratio ≈ hardware ratio ⇒ likely compute-bound (need better kernels or more GPUs).
- **MFU (Model FLOPs Utilization):** value in $[0, 1]$ = actual FLOPs achieved / achievable FLOPs on hardware.
- **Interconnect cheat sheet:**

Tech	Connects	Why it matters
PCIe	CPU/host ↔ GPU	host-device bandwidth/latency bottleneck
NVLink	GPU ↔ GPU	high bandwidth for multi-GPU on-node
InfiniBand/RDMA	node ↔ node	faster distributed training communication
GPUDirect RDMA	GPU ↔ NIC	avoids host copies (lower latency, less CPU overhead)

- **CUDA mapping (fast recall):**
  Grid → many blocks across the GPU Block → runs on one SM Warp → 32 threads scheduled together
- **NCCL (collective communication library):**
  Rank is a unique identifier for a GPU (e.g., one node broadcasts weights to all others).

Collective	What it does (your wording, clarified)	Common use
Broadcast	One rank sends data to all other ranks	weights/config
AllReduce	Reduce across all ranks and store result in every rank's buffer	gradient sync (DP)
Reduce	Reduce across ranks; result stored in one rank	aggregation
AllGather	Gather shards so every rank ends with the full data	unshard (FSDP)
ReduceScatter	Reduce, then scatter shards so each rank keeps only a shard	FSDP grads

Used for gradient sync (data-parallel training): each GPU has a model replica, runs a separate mini-batch, does forward + backward, produces partial gradients, then synchronizes gradients and applies the weight update.

AllGather (preserved + clarified): used when parameters/activations are sharded across devices; you may need to unshard so everything is available for a step.

ReduceScatter (preserved + clarified): instead of every rank having a full copy, each rank keeps only a shard of the reduced result (common in FSDP). After the update, you often AllGather again.

- **Networks in a data center (quick table):**

Network	Purpose	Key requirements
Compute	GPU↔GPU traffic (training/inference)	low latency, high throughput, scalable
In-band mgmt	SSH, scheduling, repos, config	reliable, secure isolation, moderate BW
OOB mgmt	manage server if OS down (BMC)	always on, strong access control

- **Tiered storage (quick table):**

Tier	Used for	Tradeoff
Local NVMe	hot data + fast scratch I/O	fastest, limited capacity
Parallel FS	shared high-speed access	scalable, more complex
NFS	configs/scripts/small data	simple, not for heavy throughput
Object store	checkpoints/logs/big datasets	durable + cheap, higher latency

- **Layers in NVIDIA infrastructure (clean list):**

1. Physical layer: GPU, data center, NICs, DPU
2. Data movement & I/O acceleration: NVLink, RDMA, storage, HPC fabrics (InfiniBand)

3. OS / drivers / virtualization: GPU drivers
4. Core libraries: CUDA, NCCL
5. Monitoring & management: NVIDIA-SMI, etc.
6. Applications / vertical solutions: NVIDIA NIMs, enterprise stacks

- **GPUDirect summary (quick table):**

Feature	Connects	Goal
GPUDirect RDMA	GPU mem ↔ NIC ↔ network ↔ NIC ↔ GPU mem	low latency, avoid CPU copies
GPUDirect Storage	GPU mem ↔ NVMe / storage fabric	feed GPUs faster (I/O)

- **vGPU vs MIG (quick table):**

Tech	Isolation	Typical usage
vGPU	software-enforced (hypervisor)	VM environments, cloud multi-tenant VMs
MIG	hardware-enforced	containers, HPC/AI clusters, strong partitioning

- **BCM (Base Command Manager):**
  Manage, monitor, and orchestrate GPU resources and workloads across an entire AI data center (cluster utilization, infra health, job status, quotas, workload performance).
  - Management/config: firmware configs, power policies, provisioning, scheduling, allocation, deployments.
  - Monitoring: CPU, GPUs, networking, storage, workloads (AI/HPC).
  - Interfaces: CLI, web UI, REST API.

1 Code Snippets I Should Know

Key Idea

Goal: be able to name the tool, what it tells you, and when you'd use it in an interview.

1.1 Profiling / Debugging Cheat Sheet

Tool / Snippet	Meaning (clean + interview-ready)
Nsight Systems (nsys)	End-to-end timeline: CPU↔GPU overlap, kernel launches, synchronization, memcpy, data loader stalls.
Nsight Compute (ncu)	Deep kernel analysis: occupancy, memory throughput, instruction mix, warp stalls, cache behavior.
cuda-gdb	Debug CUDA kernels (breakpoints, stepping, inspecting variables).
cuda-memcheck	Memory correctness: out-of-bounds, invalid accesses, (some) race-like issues.
CUDA runtime + driver APIs	Managing devices, memory, and program execution on the GPU.
cudaMalloc(...)	Allocates memory on the **GPU**. (`malloc` allocates on **CPU**.)
cudaMemcpy(...)	Copies data **CPU↔GPU** (host-device transfers).

Important

Fast rule: nsys tells you *where time goes overall.* ncu tells you *why a specific kernel is slow.*

1.2 NVIDIA-SMI (Node Monitoring)

What you want	Command
Show summary (default view)	`nvidia-smi`
Refresh every 1 second	`nvidia-smi -l 1`
Show specific fields (custom query)	`nvidia-smi --query-gpu=name,uuid,utilization.gpu,utilization.m --format=csv`
Save full report to a file	`nvidia-smi -q > smi_report.txt`
Check clocks in use (graphics/SM/memory)	`nvidia-smi -q -d CLOCK`
Check all supported clocks	`nvidia-smi -q -d SUPPORTED_CLOCKS`
Set power limit (requires permission)	`sudo nvidia-smi -pl 200`
Set application clocks (if supported + permitted)	`sudo nvidia-smi -ac <memClockMHz>,<smClockMHz>`
Why it might fail (permission note)	If you see **Insufficient Permissions**, you need admin/sudo access or the cluster must allow user-level power/clock controls.

1.3 Nsight Compute (NCU) Quick Commands

Key Idea

Use NCU to answer: am I compute-bound or memory-bound, and what limits occupancy/stalls?

What you want	Command
Profile app (basic)	`ncu ./vecAdd`
Fuller kernel metrics (heavier)	`ncu --set full ./vecAdd`
Profile only one kernel by name	`ncu -k vecAdd ./vecAdd`
Save report file	`ncu -o report_vecAdd ./vecAdd`
Reduce overhead (often faster)	`ncu --replay-mode application ./vecAdd`

1.4 CUDA Indexing (Fast Recall)

Name	Meaning
`gridDim.x`, `gridDim.y`	How many blocks wide/tall in the launched grid.
`blockDim.x`, `blockDim.y`	Threads per block in x/y.
`threadIdx.x`, `threadIdx.y`	Thread's local coordinates inside its block.
`blockIdx.x`, `blockIdx.y`	Block's coordinates inside the grid.

Key Idea

2D output indexing (matmul / conv-style):

$i =$ `blockIdx.y`$\cdot$`blockDim.y`$+$`threadIdx.y`, $\quad j =$ `blockIdx.x`$\cdot$`blockDim.x`$+$`threadIdx.x`

1.5 Cluster Launch (Slurm) — `srun` Line Breakdown

Key Idea

Command (preserved):

```
srun --partition=gpu --gres=gpu:1 --mem=16G --time=01:00:00 --pty bash
```

Flag	What it does
`--partition=gpu`	Run on the GPU partition/queue.
`--gres=gpu:1`	Request 1 GPU resource.
`--mem=16G`	Request 16 GB of system RAM.
`--time=01:00:00`	Time limit of 1 hour.
`--pty bash`	Start an interactive pseudo-terminal (a shell) on the allocated node.

Important

Interview phrasing: "`srun --pty` is how I grab an interactive GPU node to debug quickly before running a long `sbatch` job."

1.6 Naive Matrix Multiply (CPU + CUDA) — Interview Baseline

Key Idea

Purpose: show you understand indexing, memory layout, and where the bottleneck comes from. This is naive (no tiling/shared memory). Optimizations come after.

CPU Naive Matmul (row-major)

```
void matmul_cpu(const float* A, const float* B, float* C,
                int M, int K, int N) {
  // A: MxK, B: KxN, C: MxN (row-major)
  for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
      float sum = 0.0f;
      for (int k = 0; k < K; k++) {
        sum += A[i*K + k] * B[k*N + j];
      }
      C[i*N + j] = sum;
    }
  }
}
```

CUDA Naive Matmul Kernel

```
__global__ void matmul_cuda(const float* A, const float* B, float* C,
                            int M, int K, int N) {
  int i = blockIdx.y * blockDim.y + threadIdx.y; // row
  int j = blockIdx.x * blockDim.x + threadIdx.x; // col

  if (i < M && j < N) {
    float sum = 0.0f;
    for (int k = 0; k < K; k++) {
      sum += A[i*K + k] * B[k*N + j];
    }
    C[i*N + j] = sum;
  }
}
```

Launch Configuration (simple)

```
// Example: 16x16 threads per block
```

```
dim3 block(16, 16);
dim3 grid((N + block.x - 1) / block.x,
          (M + block.y - 1) / block.y);

matmul_cuda<<<grid, block>>>(A, B, C, M, K, N);
```

Important

What makes this naive kernel slow:

- Each thread re-loads A/B from global memory many times (low data reuse).
- No shared-memory tiling, so memory traffic is high.
- Typically becomes memory-bound at scale (unless extremely small sizes).

1.7 Interview Memory Table: NN Concepts + How They Connect

Key Idea

Goal: quick recall for interviews — definition + how it connects in training/inference.

Concept	What it is (1-liner)	How it relates to everything else
Weights (W)	Learned parameters that scale/combine input features.	Used in the **forward pass** to compute $z = Wx + b$; updated in **backprop** via gradients $\nabla_W \mathcal{L}$; influenced by **initialization** and **regularization** (e.g., weight decay); **batch norm** can change effective scaling.
Biases (b)	Learned offsets that shift activations.	Also used in forward pass $z = Wx + b$; gradients $\nabla_b \mathcal{L}$ computed in backprop; often less regularized than W; interacts with **batch norm** (BN can reduce the need for explicit biases).
Activation (σ)	Nonlinearity applied to z (e.g., ReLU, sigmoid).	Makes networks expressive; in **backprop** you multiply by $\sigma'(z)$ (chain rule); affects gradient flow (vanishing/exploding) and drives good **initialization** choices.
Forward pass	Compute predictions from inputs.	Pipeline: input $\rightarrow$ (optional **BN**) $\rightarrow$ linear (W, b) $\rightarrow$ **activation** $\rightarrow$ (optional **pooling**) $\rightarrow \hat{y}$; stores activations needed for the **backward pass**.
Loss ($\mathcal{L}$)	Measures prediction error (e.g., cross-entropy, MSE).	Backprop computes gradients of $\mathcal{L}$ w.r.t. parameters; may include **regularization**: $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{data}} + \lambda R(W)$; drives **gradient descent** updates.

Backprop	Compute gradients using the chain rule.	Uses saved forward activations; outputs $\nabla_W \mathcal{L}$, $\nabla_b \mathcal{L}$, etc.; gradients are consumed by the **optimizer**; stability depends on **activation**, **initialization**, and **BN**.
Gradient descent	Update rule: move parameters opposite the gradient.	Core update: $W \leftarrow W - \eta \nabla_W \mathcal{L}$; η is the **learning rate**; most **optimizers** are improved variants (momentum/adaptive steps).
Learning rate (η)	Step size for parameter updates.	Too big $\Rightarrow$ divergence; too small $\Rightarrow$ slow training; interacts with **optimizer** (Adam often tolerates larger η than SGD), **BN** (stabilizes), and **regularization** (effective update size changes).
Optimizer	Algorithm that uses gradients to update parameters (SGD, Adam, etc.).	Consumes gradients from backprop; may add momentum/adaptive scaling/weight decay; strongly interacts with **learning rate** and **regularization**; determines training dynamics and convergence speed.
Regularization	Techniques to reduce overfitting (L2, dropout, etc.).	Often added to loss (e.g., L2: $\lambda \|W\|^2$), changing gradients; constrains weight growth (helps stability); interacts with optimizer (e.g., AdamW decouples weight decay).
Initialization	How you set starting values of W, b.	Sets early signal/gradient scales; poor init $\Rightarrow$ vanishing/exploding gradients; chosen based on **activation** (He for ReLU, Xavier for tanh/sigmoidish).
Batch Norm (BN)	Normalizes activations using batch statistics.	Inserted in forward pass; changes gradient flow in backprop; stabilizes training, allows higher **learning rate**, reduces sensitivity to **initialization**; adds learnable scale/shift (γ, β).
Pooling	Downsamples spatial features (CNNs).	Part of forward pass after conv/activation; reduces compute and adds invariance; in backprop routes gradients (max $\rightarrow$ argmax only, avg $\rightarrow$ distributed).

Important

Interview pattern: Forward pass creates activations $\rightarrow$ loss measures error $\rightarrow$ backprop computes gradients $\rightarrow$ optimizer updates W, b (controlled by η and regularization).

2 Serving an LLM (Table Summary)

Key Idea

Big idea: Serving an LLM means turning a trained model into a fast, reliable API that real applications can call.

Stage	What Happens	Key Points / Interview Memory
Freeze Model	Save weights, tokenizer, configs; lock versions.	Start with FP16/BF16; use quantization later if memory/latency matters. Ensures reproducibility and stable deployment.
Inference Engine	Software that loads model onto GPUs and serves requests.	Example: **vLLM** — efficient batching, KV-cache optimization, high throughput. Provides API endpoint for apps.
Chain / App Layer	Business logic between app and model.	Build prompts, call tools (DB/search/code), format outputs, retries, moderation. Often FastAPI/Node.
RAG (Optional)	External knowledge retrieval before generation.	Chunk docs → embeddings → vector DB → retrieve relevant text → add to prompt. **Memory idea:** Search first, then generate.
Production Architecture	Typical serving pipeline.	App → API Gateway → Chain Service → LLM Server → GPU. Separates concerns for scalability.
Production Concerns	Operational considerations.	Latency: TTFT, tokens/sec. GPU usage: memory/utilization. Scaling: batching, multi-GPU. Safety: moderation, injection defense. Reliability: monitoring, retries.
Deployment Options	Infrastructure scale choices.	Single GPU (simple dev), multi-GPU node (higher throughput), distributed multi-node serving (large scale).

Key Idea

Interview one-liner:
"Serving an LLM means freezing the model, running it behind an optimized inference server, adding business logic or RAG in front, and monitoring latency, GPU usage, and reliability in production."

3 Research: Evaluation of Optimization Techniques for Large Language Model Inference

Key Idea

Stack: LLM inference frameworks, profiling, quantization, batching, GPU performance

3.1 Short Pitch (GPU / HPC Focus)

Studied performance bottlenecks in LLM inference pipelines across different runtimes, focusing on GPU utilization, memory bandwidth, batching strategies, and quantization. Goal was understanding how to maximize throughput on modern GPU clusters.

3.2 Why I Did It + What I Learned

- Wanted deeper intuition for inference bottlenecks beyond training.
- Learned how KV-cache behavior, batching, and memory bandwidth dominate LLM serving.
- Built strong understanding of arithmetic intensity and GPU utilization tradeoffs.

3.3 Challenges

- Separating compute-bound vs memory-bound inference stages.
- Interpreting profiler output across different frameworks.
- Understanding scaling effects in multi-GPU serving setups.

3.4 Key Interview Takeaways

- GPU inference is often memory-bound, not compute-bound.
- Batching + KV cache efficiency heavily impact throughput.
- Expect questions on quantization, kernel efficiency, and GPU memory layout.

4 Convolutional Layer Optimization with Triton

Key Idea

Stack: Triton, CUDA, PyTorch, GPU kernels

4.1 Short Pitch

Implemented custom Triton kernels to optimize 2D convolution layers, reducing kernel overhead and improving GPU utilization compared to standard PyTorch ops.

4.2 Why I Did It + What I Learned

- Wanted hands-on experience writing GPU kernels for deep learning workloads.
- Learned tiling strategies, memory coalescing, and warp-level execution effects.
- Gained intuition for tensor core usage and instruction scheduling.

4.3 Challenges

- Achieving good occupancy without exceeding register/shared memory limits.
- Debugging kernel correctness and performance simultaneously.
- Balancing compute intensity with memory throughput.

4.4 Key Interview Takeaways

- Kernel optimization = memory locality + warp efficiency.
- Profiling is essential (Nsight Compute).
- Expect questions on tiling, occupancy, and tensor core usage.

5 LABUBU AI Shopper

Key Idea

Stack: GPT-4, vector search, FastAPI, React, Faiss

5.1 Short Pitch

Built a real-time AI shopping assistant using vector search + LLM prompting, handling 1000 SKUs with sub-second response time.

5.2 Why I Did It + What I Learned

- Wanted production-style LLM system experience.
- Learned prompt engineering, retrieval augmentation, and latency optimization.
- Understood tradeoffs between embedding search cost and response speed.

5.3 Challenges

- Managing latency across retrieval + LLM inference pipeline.
- Scaling vector search while maintaining response quality.
- Designing reliable API integration.

5.4 Key Interview Takeaways

- Retrieval latency often dominates inference latency.
- RAG pipelines require careful caching and batching.
- Expect questions on LLM serving architecture.

6 GPU-Accelerated Image Processing Pipeline

Key Idea

Stack: CUDA, PyTorch, V100 GPUs

6.1 Short Pitch

Built GPU-accelerated image filters (blur/sharpen) achieving 8× speedup over CPU baselines through CUDA parallelization.

6.2 Why I Did It + What I Learned

- Wanted strong GPU fundamentals before deeper ML systems work.
- Learned kernel launch configuration, memory hierarchy, and profiling.
- Developed intuition for when GPU acceleration is worthwhile.

6.3 Challenges

- Optimizing memory access patterns.
- Avoiding unnecessary host-device transfers.
- Achieving stable benchmarking.

6.4 Key Interview Takeaways

- Memory movement often dominates GPU pipelines.
- Data locality strongly affects performance.
- Expect questions on CUDA memory hierarchy.

7 High-Performance Matrix Inversion via GPU Kernels

Key Idea

Stack: Triton, OpenCL, C++

7.1 Short Pitch

Implemented custom GPU kernels for large matrix inversion (5000×5000), achieving 50% runtime reduction versus CPU implementations.

7.2 Why I Did It + What I Learned

- Interest in HPC linear algebra performance optimization.
- Learned parallel decomposition, kernel scheduling, and memory tiling.
- Improved understanding of GPU numerical stability issues.

7.3 Challenges

- Managing numerical precision on GPU.
- Ensuring stable performance scaling.
- Avoiding excessive memory transfers.

7.4 Key Interview Takeaways

- Linear algebra dominates many ML workloads.
- GPU kernels require careful tiling.
- Expect HPC-style performance questions.

8 Parking Ticket Prevention Streaming Platform

Key Idea

Stack: Spark, Kafka, Lambda

8.1 Short Pitch

Built a real-time streaming system tracking enforcement vehicles and delivering proactive alerts using distributed event pipelines.

8.2 Why I Did It + What I Learned

- Wanted distributed systems experience.
- Learned event streaming, fault tolerance, and scaling pipelines.
- Improved understanding of latency vs throughput tradeoffs.

8.3 Challenges

- Managing streaming latency.
- Coordinating multiple distributed services.
- Ensuring reliability under load.

8.4 Key Interview Takeaways

- Distributed systems bottlenecks often shift to networking.
- Monitoring is critical.
- Expect system design questions.

9 HPC Workflows: Multi-GPU Training + Benchmarking

Key Idea

Stack: Slurm, Kubernetes, profiling

9.1 Short Pitch

Designed HPC workflows supporting multi-GPU training, cluster utilization optimization, and performance benchmarking.

9.2 Why I Did It + What I Learned

- Direct exposure to GPU cluster infrastructure.
- Learned job scheduling, resource allocation, and profiling.
- Developed intuition for distributed ML bottlenecks.

9.3 Challenges

- Load balancing across GPUs.
- Debugging distributed training issues.
- Maintaining reproducibility across environments.

9.4 Key Interview Takeaways

- Cluster utilization is often the biggest efficiency lever.
- Communication overhead limits scaling.
- Expect Slurm/Kubernetes/GPU scheduling questions.

10 Malware Detection Research

Key Idea

Stack: PyTorch, ETL pipelines, synthetic data generation

10.1 Short Pitch

Developed ML-based malware detection workflows with synthetic data generation and large-scale preprocessing pipelines.

10.2 Why I Did It + What I Learned

- Early ML research exposure.
- Learned dataset engineering, imbalance handling, and ML pipelines.
- Strengthened debugging and experimental design skills.

10.3 Challenges

- Class imbalance.
- Data preprocessing at scale.
- Model generalization.

10.4 Key Interview Takeaways

- Data quality often matters more than model complexity.
- Scaling preprocessing is non-trivial.
- Expect ML pipeline design questions.

11 Research: Evaluation of Optimization Techniques for Large Language Model Inference

Key Idea

Stack: LLMCompass, hardware benchmarking, inference optimization

11.1 Short Pitch

This research focused on understanding how large language model inference performance varies across different GPU hardware using the LLMCompass computational graph framework. LLMCompass models the inference pipeline as a computational graph and estimates how different GPUs handle memory bandwidth, compute throughput, and communication overhead. I analyzed how architectural factors like tensor core throughput, memory hierarchy, and interconnect bandwidth influence real inference performance rather than just theoretical FLOPs.

11.2 Why I Did It + What I Learned

I was interested in how hardware architecture impacts deep learning inference performance, especially for large language models deployed on GPU clusters. By studying the LLMCompass codebase, I learned how computational graphs can simulate inference workloads across different hardware configurations. I focused on understanding how memory movement, kernel scheduling, and arithmetic intensity determine bottlenecks. This helped me build intuition about why inference often becomes memory-bound and how batching, quantization, and kernel fusion improve throughput.

11.3 Challenges

The main challenge was interpreting the modeling assumptions in the LLMCompass framework and connecting them to real GPU behavior. I spent time tracing how the computational graph represented operations like attention, matrix multiplication, and KV-cache updates, then compared predicted performance with expected hardware characteristics. Understanding how communication overhead scales across GPUs was also non-trivial.

11.4 Key Interview Takeaways

This work reinforced that theoretical FLOPs rarely translate directly to real inference speed; memory bandwidth and data movement dominate. It also strengthened my understanding of hardware-aware model optimization and how profiling tools and simulation frameworks can guide deployment decisions.

12 Convolutional Layer Optimization with Triton

Key Idea

Stack: Triton, CUDA concepts, PyTorch kernels

12.1 Short Pitch

In this project, I implemented optimized convolution kernels using Triton to accelerate 2D convolution layers commonly used in deep learning workloads. The goal was to improve GPU utilization by controlling memory tiling, thread mapping, and kernel execution rather than relying solely on high-level PyTorch primitives in CuDNN.

12.2 Why I Did It + What I Learned

I wanted hands-on experience writing GPU kernels to better understand performance bottlenecks in deep learning workloads. Writing Triton kernels helped me understand how threads map to data tiles, how shared memory and registers reduce global memory access, and how warp-level execution impacts efficiency. I also gained experience profiling GPU kernels and interpreting occupancy and throughput metrics.

12.3 Technical Implementation (Code Logic)

The kernel divides convolution output tensors into tiles so each GPU thread block computes a small portion of the output. Threads collaboratively load input feature maps and filters into faster on-chip memory, perform multiply–accumulate operations, and write results back to global memory. This tiling strategy increases arithmetic intensity by reusing data already loaded into shared memory. The kernel also carefully aligns memory accesses so adjacent threads access adjacent memory locations, improving coalescing and reducing bandwidth waste.

12.4 Challenges

Balancing occupancy with register and shared memory usage was the main challenge. Larger tiles improve data reuse but can reduce the number of concurrent warps. Debugging correctness while optimizing performance also required iterative profiling.

12.5 Key Interview Takeaways

GPU performance optimization often comes down to memory locality, tiling strategy, and efficient warp scheduling. Understanding how kernels map computation onto hardware is essential for deep learning inference optimization.

13 GPU-Accelerated Image Processing Pipeline

Key Idea

Stack: CUDA, PyTorch, C++, NVIDIA GPUs

13.1 Short Pitch

This project implemented GPU-accelerated image filters such as blur and sharpen operations using CUDA, achieving significant speedups compared to CPU implementations by parallelizing pixel-level operations across thousands of GPU threads.

13.2 Why I Did It + What I Learned

I wanted foundational GPU programming experience before working on larger ML infrastructure problems. This project helped me understand CUDA kernel launches, thread indexing, memory hierarchies, and profiling workflows.

13.3 Technical Implementation (Code Logic)

Each CUDA thread processes a small region of the image, computing convolution-style filter operations independently. The kernel calculates pixel indices based on block and thread IDs, loads neighboring pixels needed for filtering, performs weighted averaging or sharpening computations, and writes results back to global memory. Performance improvements mainly came from parallelism and minimizing host–device memory transfers.

13.4 Challenges

Efficient memory access patterns were critical. Uncoalesced memory accesses significantly reduced performance, so restructuring data layout and kernel indexing was necessary. Benchmarking fairly against CPU baselines also required careful measurement.

13.5 Key Interview Takeaways

GPU acceleration is most effective when computation per memory access is high. Understanding memory hierarchy and data locality is essential for both classical GPU workloads and deep learning inference.

14 High-Performance Matrix Inversion via Custom GPU Kernels

Key Idea

Stack: Triton, OpenCL, C++

14.1 Short Pitch

I implemented GPU-based matrix inversion kernels for large matrices, focusing on parallel decomposition strategies to reduce runtime compared to CPU-based numerical linear algebra implementations.

14.2 Why I Did It + What I Learned

This project deepened my understanding of HPC-style numerical workloads and their relevance to deep learning. Many ML operations reduce to large matrix multiplications or inversions, so optimizing these primitives directly improves training and inference performance.

14.3 Technical Implementation (Code Logic)

The implementation partitions matrices into blocks processed in parallel by GPU threads. Each thread block performs partial inversion or elimination steps while leveraging shared memory to reuse intermediate results. Synchronization ensures correctness across thread blocks, while kernel fusion reduces intermediate memory writes. OpenCL experiments helped compare portability versus performance tradeoffs.

14.4 Challenges

Maintaining numerical stability while maximizing performance was challenging. Precision management and synchronization overhead required careful tuning.

14.5 Key Interview Takeaways

Large-scale ML workloads rely heavily on linear algebra performance. Kernel tiling, synchronization strategy, and memory reuse strongly influence GPU efficiency.

15 HPC Workflows: Multi-GPU Training + Benchmarking

Key Idea

Stack: Slurm, Kubernetes, GPU clusters

15.1 Short Pitch

I worked on designing HPC workflows supporting multi-GPU training, benchmarking, and cluster resource optimization. This included job scheduling automation, profiling GPU utilization, and improving overall cluster efficiency.

15.2 Why I Did It + What I Learned

This experience gave me exposure to real GPU cluster infrastructure. I learned how scheduling policies, communication overhead, and data pipelines affect distributed deep learning performance.

15.3 Challenges

Load balancing across GPUs and diagnosing distributed training slowdowns were key challenges. Differences between environments also affected reproducibility.

15.4 Key Interview Takeaways

Scaling ML workloads introduces communication overhead that can offset compute gains. Efficient cluster utilization often requires profiling, scheduling optimization, and pipeline improvements.

16 LABUBU AI Shopper

Key Idea

Stack: GPT-4, vector search, FastAPI, React, Faiss

16.1 Short Pitch

Built a real-time AI shopping assistant using vector search + LLM prompting, handling 1000 SKUs with sub-second response time.

16.2 Why I Did It + What I Learned

- Wanted production-style LLM system experience.
- Learned prompt engineering, retrieval augmentation, and latency optimization.
- Understood tradeoffs between embedding search cost and response speed.

16.3 Challenges

- Managing latency across retrieval + LLM inference pipeline.
- Scaling vector search while maintaining response quality.
- Designing reliable API integration.

16.4 Key Interview Takeaways

- Retrieval latency often dominates inference latency.
- RAG pipelines require careful caching and batching.
- Expect questions on LLM serving architecture.

17 Parking Ticket Prevention Streaming Platform

Key Idea

Stack: Spark, Kafka, Lambda

17.1 Short Pitch

Built a real-time streaming system tracking enforcement vehicles and delivering proactive alerts using distributed event pipelines.

17.2 Why I Did It + What I Learned

- Wanted distributed systems experience.
- Learned event streaming, fault tolerance, and scaling pipelines.
- Improved understanding of latency vs throughput tradeoffs.

17.3 Challenges

- Managing streaming latency.
- Coordinating multiple distributed services.
- Ensuring reliability under load.

17.4 Key Interview Takeaways

- Distributed systems bottlenecks often shift to networking.
- Monitoring is critical.
- Expect system design questions.

18 Malware Detection Research

Key Idea

Stack: PyTorch, ETL pipelines, synthetic data generation

18.1 Short Pitch

Developed ML-based malware detection workflows with synthetic data generation and large-scale preprocessing pipelines.

18.2 Why I Did It + What I Learned

- Early ML research exposure.
- Learned dataset engineering, imbalance handling, and ML pipelines.
- Strengthened debugging and experimental design skills.

18.3 Challenges

- Class imbalance.
- Data preprocessing at scale.
- Model generalization.

18.4 Key Interview Takeaways

- Data quality often matters more than model complexity.
- Scaling preprocessing is non-trivial.
- Expect ML pipeline design questions.