Lecture 1:

Computer Structure and Networks for ML

Ho-min Park

homin.park@ghent.ac.kr

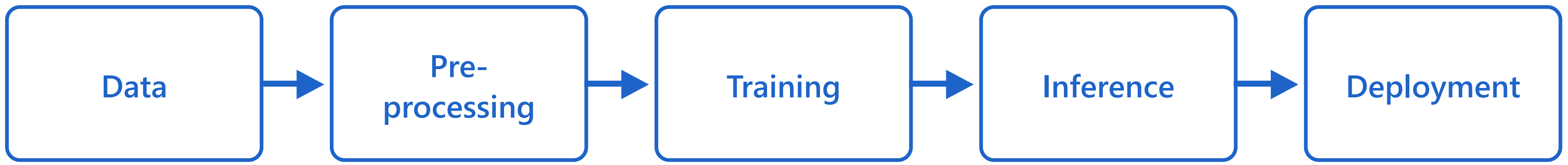
powersimmani@gmail.com

Lecture Contents

Part 1/3:

Data Representation and ML Hardware Fundamentals

ML Workflow and Computer Architecture



Key Impact Areas

- Computer architecture impacts every stage
- Understanding hardware optimizes performance

Critical Bottlenecks

- Data loading efficiency
- Computation speed
- Memory bandwidth limits

Bits and Bytes - Understanding ML Data Types

Fundamental Unit: **1 Byte = 8 Bits**

FP32

Float 32-bit

4 Bytes

PyTorch/TensorFlow default for training

FP16

Float 16-bit

2 Bytes

Half precision, 50% memory

INT8

Integer 8-bit

1 Byte

Quantized, 75% memory savings

Data Type Impact

- Memory usage
- Processing speed
- Model accuracy

Example Calculation:
1B parameters × FP32
= 1B × 4 bytes = 4GB

Byte Representation: Signed vs Unsigned

Unsigned Byte (8-bit)

Signed Byte (8-bit)

Range: 0 to 255

Binary: 00000000 to 11111111

Only positive values

Range: -128 to 127

Uses **two's complement**

First bit = sign bit

Hexadecimal (Base-16) Representation

1 Byte = 2 Hex Digits (0-9, A-F)

`0x00` Decimal: 0	`0x0F` Decimal: 15	`0x10` Decimal: 16	`0xFF` Decimal: 255

ASCII Code Examples

Each character = 1 Byte (8 bits)

A 65 (0x41)	**B** 66 (0x42)	**a** 97 (0x61)	**0** 48 (0x30)	**Space** 32 (0x20)

Bit-Level Representation

8-bit Integer (INT8)

32-bit Float (FP32)

Decimal: 42
`00101010`

Decimal: -42 (signed)
`11010110`

Decimal: 127 (max)
`01111111`

Structure:
`S EEEEEEEE MMMMMMMMMMMMMMMMMMMMMMM`

1 Sign bit

8 Exponent bits

23 Mantissa bits

RGB Color Codes and Bytes

Each color channel = 1 Byte (0-255) • Total = 3 Bytes per pixel

🟥	🟩	🟦
Red	**Green**	**Blue**
`#FF0000`	`#00FF00`	`#0000FF`
RGB(255, 0, 0)	RGB(0, 255, 0)	RGB(0, 0, 255)
⬜	▨	🟦
White	**Gray**	**Theme Blue**
`#FFFFFF`	`#808080`	`#1E64C8`

RGB(255, 255, 255)	RGB(128, 128, 128)	RGB(30, 100, 200)

Number Representation Methods - Fixed Point vs. Floating Point

Fixed Point

Integer with implicit decimal position

- ✓ Fast computation
- ✓ Limited range
- ✓ Fixed precision

Floating Point

Sign + Exponent + Mantissa

- ✓ Flexible representation
- ✓ Wide range
- ✓ Slower processing

Floating Point Formats

FP32 (32 bits)

Sign:	1 bit
Exponent:	8 bits
Mantissa:	23 bits

Range: $\pm 3.4 \times 10^{38}$

FP16 (16 bits)

Sign:	1 bit
Exponent:	5 bits
Mantissa:	10 bits

Range: $\pm 6.5 \times 10^{4}$

Key Trade-off

Precision ⇄ Memory ⇄ Speed

Basic Logic Gates

⊼

AND

$Y = A \cdot B$

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

⊽

OR

$Y = A + B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

⊕

XOR

$Y = A \oplus B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

¬

NOT

$Y = \bar{A}$

A	Y
0	1
1	0

⊼

NAND

$Y = (A \cdot B)^{-}$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

⊽

NOR

$Y = (A + B)^{-}$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Adder Circuits (Building Blocks)

Half Adder

A B

↓

Full Adder

A B C_{in}

↓

Half Adder

↓

Sum Carry

```
Sum = A XOR B
Carry = A AND B
```

Full Adder

↓

Sum C_{out}

Sum = A XOR B XOR C_{in}
C_{out} = (A AND B) OR (C_{in} AND (A XOR B))

Calculation Examples Using Logic Gates

Integer Addition (8-bit): 5 + 3

Step 1: Binary Conversion

5 =	0	0	0	0	0	1	0	1
3 =	0	0	0	0	0	0	1	1

Bit 0: XOR→0, AND→1(C) →

Bit 1: Full Add→0, C=1 →

Bit 2: Full Add→0, C=1 → Bit 3: 1

Result

8 =	0	0	0	0	1	0	0	0

5 + 3 = 8 ✓

FP16 Representation: 3.5

FP16 Bit Layout

3.5 =	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0

🟥 Sign (1 bit) 🟩 Exponent (5 bits) 🟩 Mantissa (10 bits)

Decoding:

- Sign: 0 → Positive
- Exponent: 10000_2 = 16 → 16 - 15 = 1
- Mantissa: 1.11_2 (implicit 1)

$$(-1)^0 \times 1.11_2 \times 2^1 = 1.75 \times 2 = 3.5$$

Floating Point Addition: 2.5 + 1.25

1. Compare Exponents → 2. Align Mantissas → 3. Add Mantissas (XOR/AND chains) → 4. Normalize → 5. Round

$2.5 = 1.01_2 \times 2^1$

Exp: 16 (bias 15 + 1)

$1.25 = 1.01_2 \times 2^0$

Exp: 15 (bias 15 + 0)

Aligned & Added

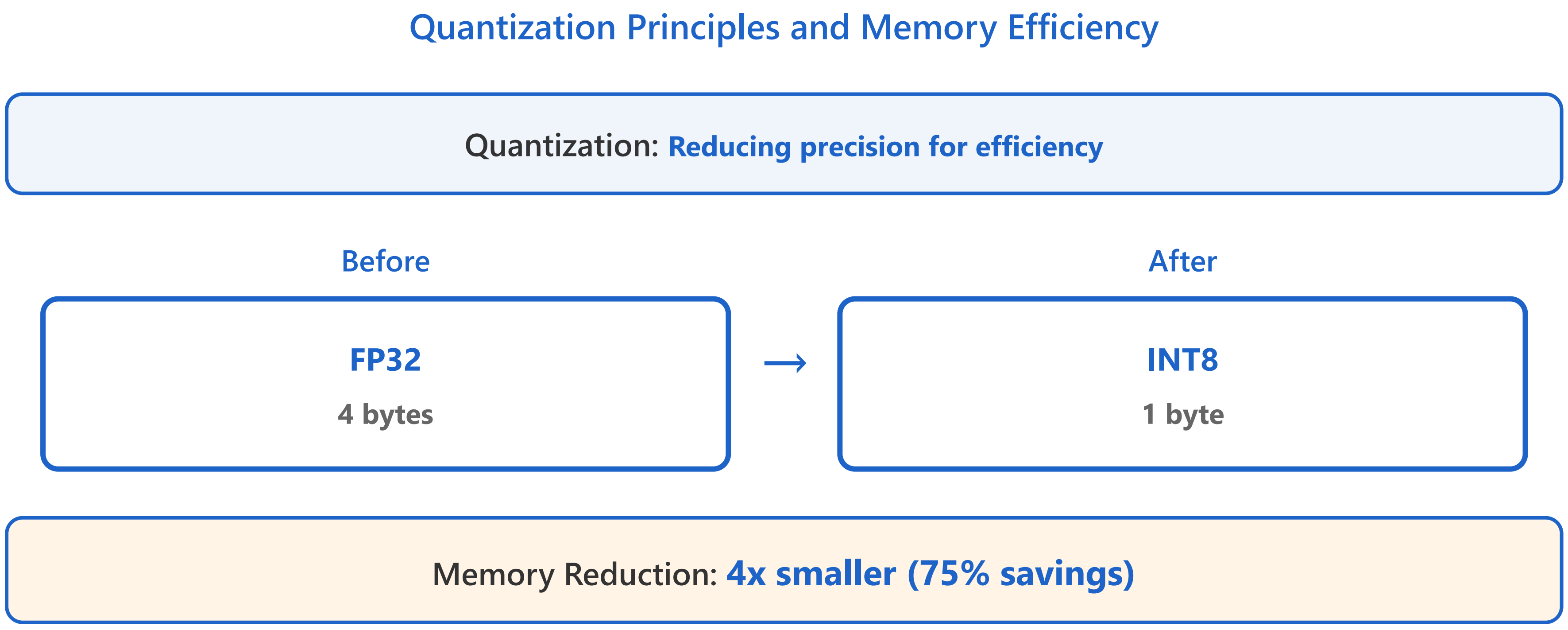
$$
\begin{array}{rll}
 & 1.010_2 & \times 2^1 \ (2.5) \\
+ & 0.101_2 & \times 2^1 \ (1.25 \text{ shifted}) \\
\hline
 & 1.111_2 & \times 2^1 = 3.75
\end{array}
$$

2.5 + 1.25 = 3.75 ✓

Hardware Implementation Summary

Integer: Direct Full Adder chains → Fast, simple circuits

Float: Exponent compare + Shift + Add + Normalize → More gates, slower


Quantization Principles and Memory Efficiency
Quantization: Reducing precision for efficiency
Before
FP32
4 bytes
After
INT8
1 byte
Memory Reduction: 4x smaller (75% savings)

Quantization Methods

- Post-training quantization
- Quantization-aware training

Accuracy Impact

- Minimal accuracy loss
- Typically <1-2% degradation
- With proper quantization

Real-World Applications

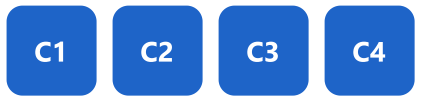
Mobile Deployment

Edge Devices

Faster Inference

CPU vs. GPU - Architectural Comparison

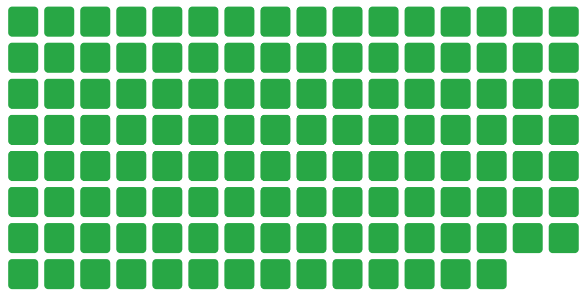
CPU



Few powerful cores (4-64)

- Optimized for sequential tasks
- Better for control flow & branching
- General computing excellence

GPU



Thousands of simple cores (1000s-10000s)

- Massive parallel processing
- 10-100x faster for matrix ops
- Ideal for ML workloads

GPU Performance: **10-100x faster for matrix operations**

Modern ML Workflow

GPU: Training

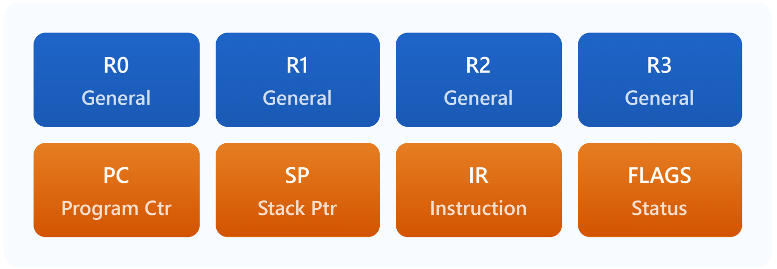
CPU: Preprocessing / Serving

CPU & GPU Common Architecture Components

Registers

Ultra-fast temporary storage

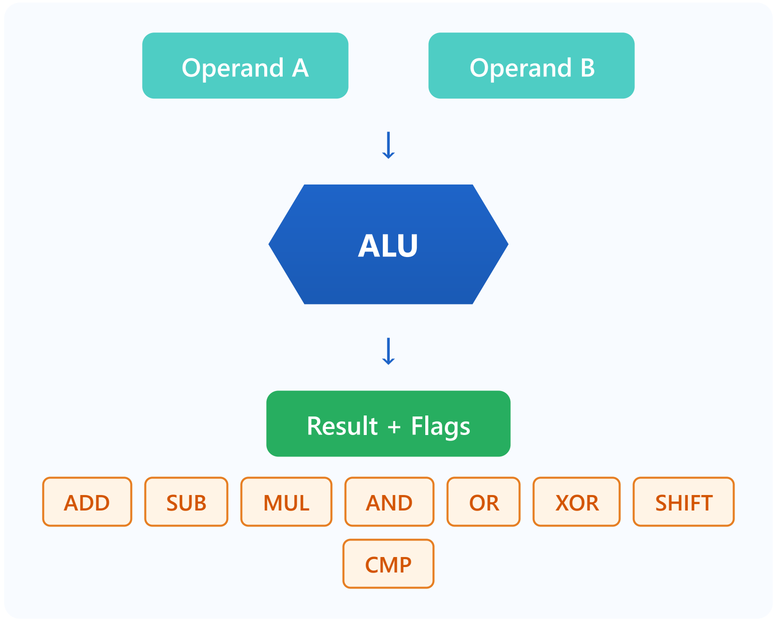
Small, high-speed storage locations within the processor. Hold data being actively processed, instruction addresses, and status flags.



ALU (Arithmetic Logic Unit)

The computational engine

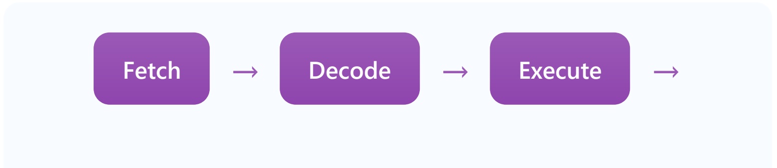
Performs all arithmetic and logical operations. Built from logic gates (AND, OR, XOR, etc.).



Control Unit (CU)

The orchestrator

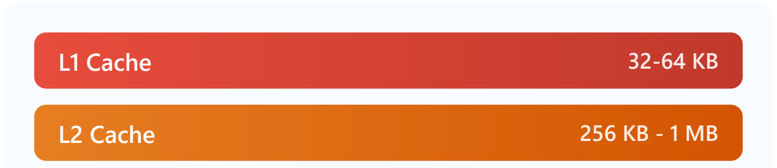
Directs all processor operations by generating control signals. Manages the instruction cycle and coordinates data flow.



Cache Memory

Speed bridge to main memory

High-speed memory hierarchy that reduces access time. Closer to core = faster but smaller capacity.



Write Back

L3 Cache	4-64 MB
Main Memory (RAM)	8-128 GB

CPU vs GPU: Shared Components, Different Scale

CPU
Central Processing Unit

Cores	**4 - 64 cores**
ALUs per Core	**Few (complex)**
Registers per Core	**~100s**
Cache Size	**Large (MB)**
Control Logic	**Sophisticated**
Optimized For	**Sequential tasks**

GPU
Graphics Processing Unit

Cores	**1000s - 10000s**
ALUs per Core	**Many (simple)**
Registers per Core	**~1000s**
Cache Size	**Smaller (KB)**
Control Logic	**Minimal**
Optimized For	**Parallel tasks**

Data Path: How Data Flows Through the Processor

Memory → Registers → ALU → Control → Output

GPU Cores and CUDA - Understanding Parallel Processing

CUDA: NVIDIA's parallel computing platform (launched 2007)

GPU Architecture Hierarchy

GPU — Graphics Processing Unit

↓

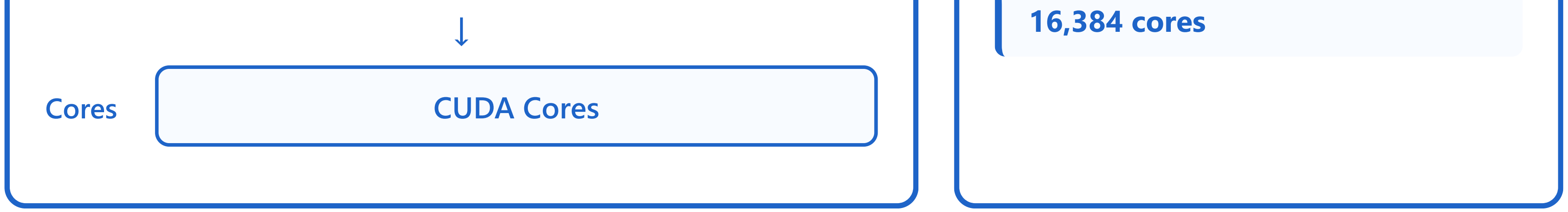
SMs — Streaming Multiprocessors

Example

RTX 4090

Streaming Multiprocessors
128 SMs

CUDA Cores



SIMT Execution

Single Instruction, Multiple Threads model for parallel processing

Automatic Integration

PyTorch/TensorFlow automatically leverage CUDA kernels

PyTorch & TensorFlow automatically leverage CUDA for GPU acceleration

FLOPS and ML Model Performance Metrics

FLOPS

Floating Point Operations Per Second

Hardware capability metric

FLOPs

Floating Point Operations

Model complexity metric (total operations required)

GPU-3 Training Example

Total FLOPs: **~3.14×10^{23}** (314 zettaFLOPs)

GPU Performance Comparison

NVIDIA A100

312

TFLOPS

3.2x faster

NVIDIA H100

1000

TFLOPS

Training Time Calculation

Training Time = **Model FLOPs** / **Hardware FLOPS** / **GPU Count**

Tensor Operations and Hardware Optimization

Tensor: Multi-dimensional array (Foundation of deep learning)

Common Tensor Operations

matmul
Matrix multiplication
Parallelizable

conv2d
2D convolution
Parallelizable

attention
Self-attention
Parallelizable

Coalesced Access

Sequential memory reads
⚡ Fast & efficient

Uncoalesced Access

Random memory reads
⚠️ Slow & inefficient

Optimization Strategies

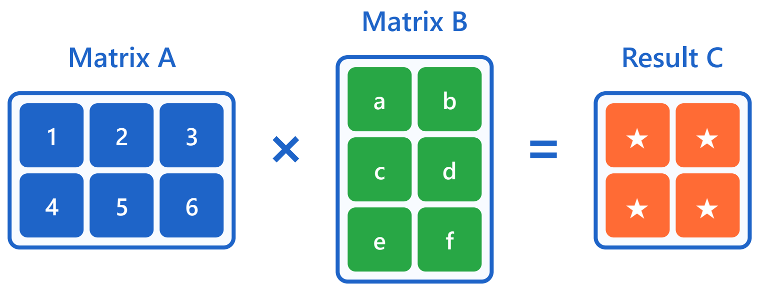
- ✓ Batch operations together
- ✓ Maintain contiguous memory
- ✓ Minimize CPU-GPU transfers

Tensor Cores

Specialized hardware for mixed-precision matmul

How Parallelization Works

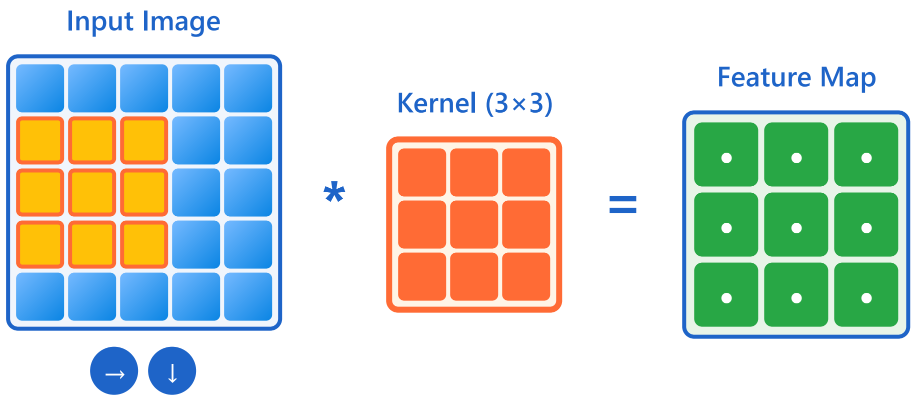
① Matrix Multiplication (matmul)



GPU Parallel Execution

Each output element computed by separate thread → All at once!

② 2D Convolution (conv2d)



GPU Parallel Execution

All positions processed simultaneously
224×224 image × 64 filters = 3.2M parallel ops

③ Self-Attention (Transformer)

$$\boxed{\text{Attention}(Q, K, V) = \text{softmax}(\, QK^T / \sqrt{d_k}\,) \cdot V}$$

🟥 Q (Query) 🟦 K (Key) 🟩 V (Value)

Step 1

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad QK^T$$

Q × Kᵀ → Attention scores (parallel matmul)

Step 2

$$\text{softmax}\left(\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \right) = \begin{bmatrix} .5 & .5 \\ .27 & .73 \end{bmatrix}$$

Softmax → Probability distribution (parallel per row)

Step 3

$$\begin{bmatrix} .5 & .5 \\ .27 & .73 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2.0 & 3.0 \\ 2.5 & 3.5 \end{bmatrix}$$

Attn × V → Weighted sum of values (parallel matmul)

⚡ 3-Level Parallelism

📦 Batch × 🔀 Heads × 📝 Tokens = 🚀 Millions parallel

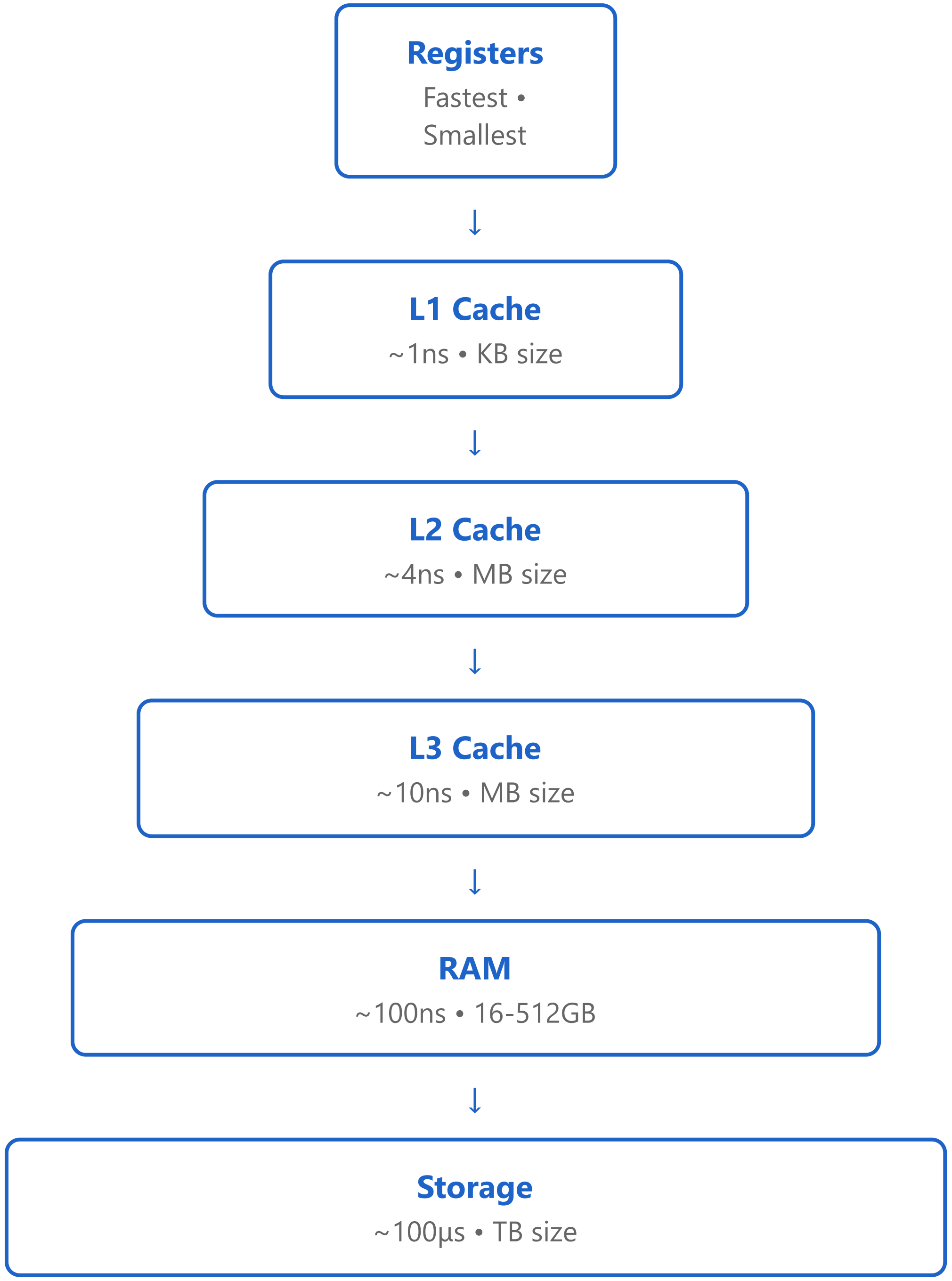
💡

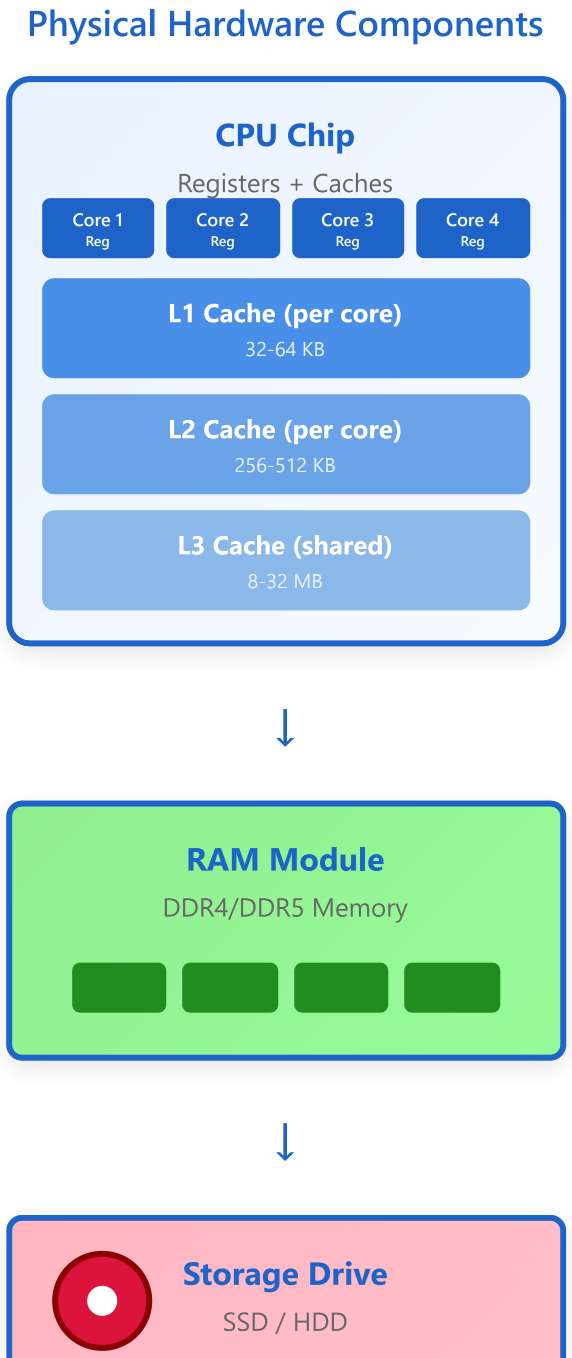
Key Insight: GPU runs thousands of threads simultaneously — turning sequential math into parallel power

Part 2/3:

Memory and ML Model Execution

Memory Hierarchy - RAM, VRAM, Cache





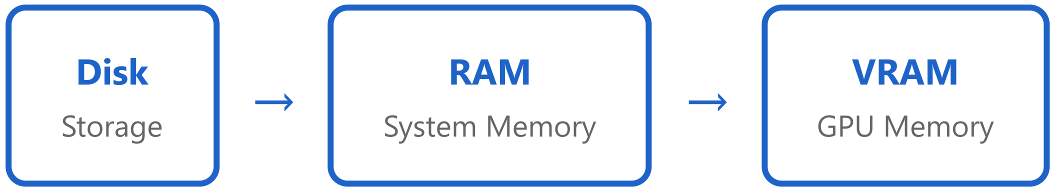
Trade-off: **Faster = Smaller + Higher Cost**

SSD: Flash Memory
HDD: Magnetic Disk

RAM	VRAM (GPU)	Cache
16-512GB	8-80GB	L1: KB
~20GB/s bandwidth	~1-2TB/s bandwidth	L2: MB
System memory	HBM2/HBM3	L3: MB

ML Bottleneck: RAM ↔ VRAM *Data transfer overhead*

ML Models Loading and Memory Management



Challenge: **LLaMA-70B = 140GB (FP16)** exceeds single GPU VRAM

Solutions for Large Models

1

Model Sharding

Distribute across multiple GPUs (tensor parallelism)

2

Offloading

Keep parameters in RAM, load when needed

3

Quantization

Reduce memory footprint (8-bit, 4-bit)

Key Insights

- Memory bandwidth limits loading speed
- Disk I/O is not the bottleneck
- Use memory-mapped files (mmap)

Popular Tools

- Hugging Face Accelerate
- DeepSpeed ZeRO

Batch Size and Memory Usage Calculation

Memory Usage Components

Model Params + Optimizer States + Gradients + Activations

Activations Memory ∝ **Batch Size × Sequence Length**

Example: BERT-base

batch = 32 • **seq = 512** → **~8GB** activation memory

Gradient Accumulation

Split batch into micro-batches to save memory

Gradient Checkpointing

Recompute activations (trade compute for memory)

💡 **Rule of Thumb**

OOM Error?

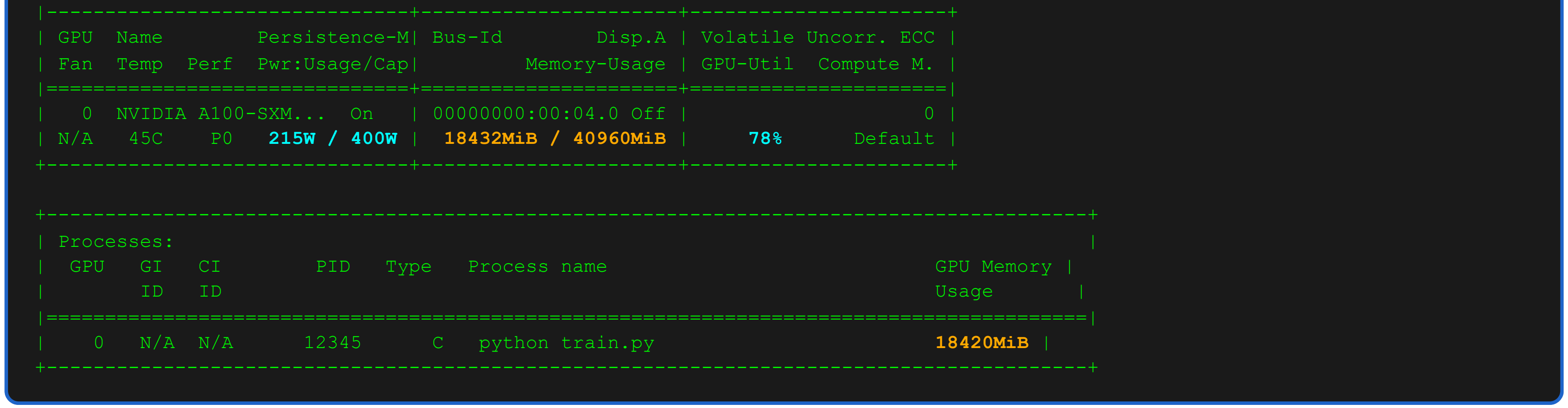
Reduce batch size by 50%

🔍 **Monitoring Tools**

`nvidia-smi`

`torch.cuda.memory_allocated()`

```
$ nvidia-smi
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05   Driver Version: 535.104.05   CUDA Version: 12.2     |
```



```
|-----------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  NVIDIA A100-SXM...  On   | 00000000:00:04.0 Off |                    0 |
| N/A   45C    P0   215W / 400W |  18432MiB / 40960MiB |     78%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      12345      C   python train.py                18420MiB |
+-----------------------------------------------------------------------------+
```

Python Bytecode and ML Frameworks

Python Execution Flow

Python Code → Bytecode → VM Execution

ML Framework Architecture

Frontend
Python API

+

Backend
C++ / CUDA

Eager Mode
Default

- Flexible execution
- Good for debugging
- Slower performance

Graph Mode
Optimized

- Pre-compiled graph
- Faster execution
- Production ready

JIT Compilation

`torch.jit.script()` speeds up inference via Just-In-Time compilation

TorchScript

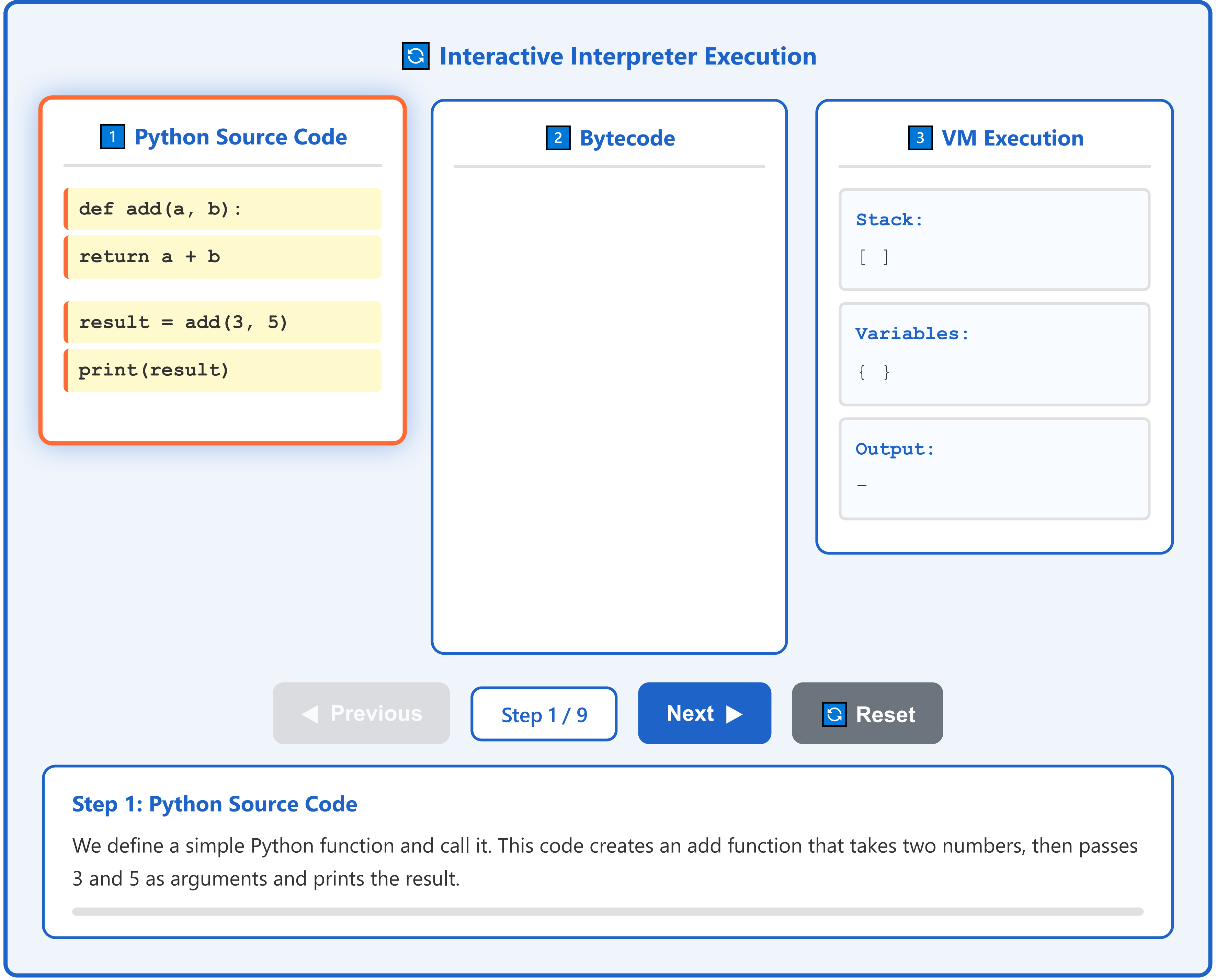
Serialize models for production (remove Python dependency)

Real Performance

CUDA kernels drive performance, not Python code

Python Overhead

Negligible for large tensor operations (batching helps)



```
def add(a, b):
return a + b

result = add(3, 5)
print(result)
```

```
Stack:
[ ]

Variables:
{ }

Output:
—
```

Step 1: Python Source Code

We define a simple Python function and call it. This code creates an add function that takes two numbers, then passes 3 and 5 as arguments and prints the result.

Memory Layout of NumPy/PyTorch Tensors

Tensors: Multi-dimensional arrays stored in **contiguous memory**

Row-Major (C)

Default

Last dimension varies fastest

0,0	0,1	1,0	1,1

Column-Major (Fortran)

First dimension varies fastest

0,0	1,0	0,1	1,1

Stride

Number of bytes to jump to next element in each dimension

Contiguous Tensors ✓

✓ Faster operations

✓ Better cache locality

✓ Optimized GPU performance

Non-Contiguous ⚠

⚠ After transpose(), view()

⚠ May need .contiguous()

⚠ Slower memory access

Memory Layout Impact

- Cache hits efficiency
- Vectorization performance
- GPU operation speed

Check Methods

`.is_contiguous()`

`.stride()`

GPU Memory Management

GPU Memory Allocation

Cached allocator for efficiency (PyTorch)

Free unused cache: `torch.cuda.empty_cache()`

✓ Best Practices

- ✓ Allocate tensors at beginning
- ✓ Reuse tensors when possible
- ✓ Minimize CPU↔GPU transfers

⚠ Avoid

- ✗ Creating tensors in loops
- ✗ Frequent CPU↔GPU transfers
- ✗ Memory fragmentation

Transfer Speed Comparison

Within GPU (device-to-device)	**Fast**
Between GPUs	**Slow**

Unified Memory (CUDA)

Automatic migration between CPU and GPU memory

Profiling Tools

- PyTorch Profiler
- NVIDIA Nsight Systems

Mixed Precision Training - FP16 and FP32

Mixed Precision: Use **FP16 for compute** , **FP32 for stability**

💾

Memory Reduction

2x smaller

⚡

Speed Boost

2-3x faster

(with Tensor Cores)

⚠ **Challenge**

✓ **Solution**

FP16 range limited → gradient underflow/overflow issues

Loss scaling: multiply loss by 1000-10000 to prevent underflow

Master Weights Strategy

FP32 Master Copy → FP16 Forward/Backward → FP32 Updates

AMP Tools

`torch.cuda.amp`

`TF mixed_precision`

Best Hardware

Modern GPUs (Volta+) with Tensor Cores

Accuracy Impact

Minimal loss (<0.1%) for significant speedup

Hands-on: Resource Monitoring Tools

GPU Monitoring

`nvidia-smi`
Real-time GPU utilization, memory, temperature

`watch -n 1 nvidia-smi`
Auto-refresh every second

CPU & System

`htop` / `top`
CPU and system memory monitoring

PyTorch Memory Tools

`torch.cuda.memory_summary()`

`torch.cuda.memory_allocated()`

TensorBoard
Visualize training metrics & memory over time

NVIDIA Nsight
Detailed profiling of GPU operations

W&B / MLflow
Track experiments & resource usage

Practice Workflow

Monitor during training → Identify bottlenecks → Optimize

Part 3/3:

Network and Distributed ML

IP Addresses and Ports - Server Connection Basics

IPv4

32-bit address
4.3 billion addresses

```
192.168.1.100
```

IPv6

128-bit address
Future-proof

```
2001:0db8::1
```

Public IP
Internet-facing address, globally unique

Private IP
Internal network (192.168.x.x, 10.x.x.x)

Common Ports (0-65535)

22	**80**	**443**
SSH	HTTP	HTTPS

ML Server Connection

```
ssh user@192.168.1.100 -p 22
```

Port Forwarding

Firewall

Security

Access remote Jupyter: localhost:8888 → server:8888	Controls port accessibility from outside	Use non-standard ports, close unused ports

SSH and Remote Server Connection Practice

SSH (Secure Shell): Encrypted remote terminal access to servers

Local Machine → **SSH Encrypted** → **Remote Server**

Basic Connection

```
ssh username@hostname
```

SSH Key Authentication Setup (More Secure)

1. **Generate Keys**

```
ssh-keygen -t rsa -b 4096
```

2. **Copy Public Key**

```
ssh-copy-id username@server
```

3. **Connect**

```
ssh username@server
```

No password needed!

SSH Config

Persistent Sessions

Practice

Connect to lab server, run training scripts remotely

`~/.ssh/config` for convenient aliases and settings

Use `tmux` or `screen` to keep sessions alive

File Transfer - Using SCP and SFTP

SCP (Secure Copy): Command-line file transfer over SSH

↑ Upload

Local → Remote Server

```
scp local_file.py user@server:/remote/p
ath/
```

↓ Download

Remote Server → Local

```
scp user@server:/remote/model.pth ./loc
al/path/
```

Recursive Transfer (Entire Directories)

Use -r flag to transfer directories

```
scp -r ./dataset/ user@server:/data/
```

SCP
Simple command-line transfer over SSH

SFTP
Interactive file transfer (like FTP but secure)

rsync
Smart sync, only transfers changed files (efficient)

Large Datasets
Consider cloud storage (S3, GCS) or shared filesystems

Compression Tip

Compress before transfer (tar + gzip) for faster upload/download

HTTP API and Model Serving

REST API: Standard way to serve ML models over HTTP

Client Request
POST /predict
JSON data

→

ML Model
Processing
Inference

→

Server Response
JSON
predictions

Popular Frameworks

Flask | **FastAPI** | **TensorFlow Serving**

✓ Benefits

- Language-agnostic
- Easy integration
- Scalable with load balancers

⚠ Considerations

- Latency (~10-100ms)
- Throughput (requests/sec)
- Batching strategies

Production Tools
Docker + Kubernetes for deployment and scaling

Monitoring
Track API response time, error rates, model accuracy drift

Distributed Training Overview

Why Distributed? Single GPU insufficient for large models/datasets

Data Parallelism

M M M

Same model on multiple GPUs, different data batches

Model Parallelism

M_1 M_2 M_3

Split model across GPUs (layers or tensor sharding)

Synchronous Training

Common

All GPUs sync gradients each step. More stable, consistent results.

Asynchronous Training

GPUs update independently. Faster but less stable convergence.

PyTorch Standard Implementation

DistributedDataParallel (DDP) for multi-GPU training

```
torch.nn.parallel.DistributedDataParallel
```

Communication

All-reduce operation for gradient synchronization across GPUs

Efficiency

Linear speedup ideal (2 GPUs = 2x), but overhead exists

Network Bandwidth and Learning Speed

Network Bandwidth: Critical for multi-node distributed training

Single Node
Fast

8 GPUs with NVLink/PCIe

Direct GPU communication

Multi-Node
Moderate

Ethernet: 10-100 Gbps

InfiniBand: 200-400 Gbps

Network dependent

Communication Bottleneck Example

Increases with more nodes and smaller batches

Gradient sync time > Computation time = Bottleneck

Gradient Compression
Reduce data to transfer

Reduce Frequency
Communicate less often

ZeRO (DeepSpeed)
Partition optimizer states

⚡ **Design Rule**

Network bandwidth should match GPU compute capability

NVLink

600 GB/s

GPU-to-GPU direct

High-end servers

PCIe 4.0 x16

32 GB/s

Most common setup

Standard

PCIe 5.0 x16

64 GB/s

Latest generation

Emerging

Docker Basics - ML Environment Containerization

Docker: Package application + dependencies into portable container

✓ Consistent environment • No "works on my machine" issues

📦 Image

Template (read-only)

```
pytorch/pytorch:2.0-cuda11.8-cudnn8-runtime
```

🚀 Container

Running instance of image (isolated process)

📝 Dockerfile

Recipe to build custom images

💾 Volume

Mount local directory for data persistence

Dockerfile Commands

`FROM` `RUN` `COPY` `CMD`

🤔 Without Docker vs With Docker

❌ **Without Docker**

✅ **With Docker**

Computer A

VS

Computer A

see the differ

▶ Start Comparison

🔄 Reset

Leveraging Cloud GPUs

Major Cloud Providers

AWS
EC2 P4/P5

GCP
A2/A3

Azure
NC/ND series

On-Demand

$$$$

Pay per hour - Expensive but flexible

e.g., $32/hr for A100

Spot Instances

70-90% off

Much cheaper, can be interrupted

Good for experiments

Managed Services

- SageMaker (AWS)
- Vertex AI (GCP)
- Azure ML

GPU Marketplaces

- Lambda Labs
- RunPod
- Vast.ai

Cost Optimization
Small GPUs for debug, scale up for training

Free Credits
Research/education credits from providers

Always Shut Down
Stop instances when not in use, monitor spending

Hands-on Project - Training ML Models on Remote Server

Project Goal: Train image classifier on remote GPU server

1 SSH & Setup

SSH into server and setup environment (conda/docker)

2 Transfer Dataset

Use scp or download directly on server

3 Write Training Script

PyTorch/TensorFlow with proper logging

4 Run in tmux

Persistent session even if disconnected

5 Monitor Training

nvidia-smi and TensorBoard (port forwarding)

6 Download & Evaluate

Download trained model and evaluate locally

📦 Project Deliverables

- 🤖 Working Model
- 📊 Training Logs
- 📄 Results Presentation

Thank you

Ho-min Park

homin.park@ghent.ac.kr

powersimmani@gmail.com