HPML

* Scaling Efficiency ⇒ $E = \frac{t_{serial}}{t_{parallel} * p} \leq 1$

* Strong Scaling ⇒ same problem size, increasing p.
  Weak Scaling ⇒ ↑ problem size with ↑ p.

* Averages ⇒ Throughput
  t constant → Arithmetic mean
  # operations constant → Harmonic mean
  Speedup / any ratios → Geometric mean

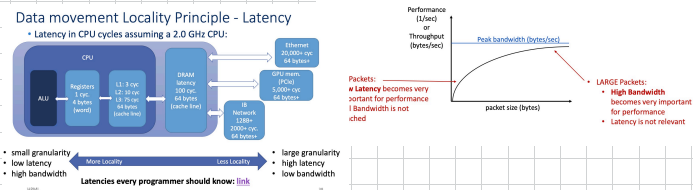
* Benchmarks
  Micro kernels, micro-benchmarks, kernels, synthetic benchmark, real applications, real workflow
  ⟵ complexity ⟶

* Compute time → real time, CPU time, non-CPU time
  real ≥ sys + user

* Profiling → Sampling → estimate
  → Counting → actual → software counters / hardware counters

* Amdahl's law → $S(p,s) = \frac{1}{(1-p)+p/s}$

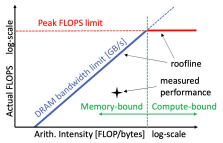
* Critical path → section of program that accounts for the most time.
  Bottleneck → system resource that affects critical path.



* Peak FLOPS → #cores × $\frac{cycles}{s}$ × $\frac{FLOPs}{cycle}$

* Arithmetic Intensity → $\frac{FLOP}{bytes}$ ⇒ $\frac{\#\text{Arithmetic operations}}{\text{DRAM data}}$ ⇒ program property

* There are more complex roofline models that involves cache as well as DRAM.



Cache blocking (↗) → reduces DRAM traffic → increase arithmetic intensity
Vectorization (↑) → increases achieved FLOPs → AI remains the same

Measure → Analyse → optimize

* $\theta = \theta - \alpha \nabla J(\theta) \longrightarrow v_t = \gamma v_{t-1} + \alpha \nabla J(\theta)$
  $\theta -= v_t$

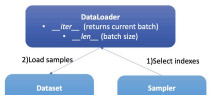
Nesterov → $v_t = \gamma v_{t-1} + \alpha \nabla J(\theta - \gamma v_{t-1})$
↳ momentum (usually 0.9)

Adagrad → separate learning rate for each parameter ⇒ $\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{diag(G_t)+\epsilon}} \odot \nabla J(\theta_t)$
↳ good for sparse data but gets smaller over time
Fixed using adadelta → only consider recent time window history.

Adam is generalisation of Adagrad.

* Python threading is concurrency, not parallelism.

* Torch.utils.data.Datasets → __len__, __getitem__



* Useful Parameters:
  * *batch_size:* batch size
  * *sampler:* define which sampler to use
  * *pin_memory:* copy into CUDA pinned memory before returning the data
  * *shuffle:* reshuffle data at each epoch
* Available *samplers:*
  * *SequentialSampler*
  * *RandomSampler*
  * *SubsetRandomSampler*

* Disk performance ⇒ I/O Ops/s
  For throughput multiply this by block size.

* Computational graph → exposes parallelism, dynamic in Pytorch, static in TF.
  ↗ Python
  Declarative → declare all, compile, compute
  Imperitive → start declaring, keep declaring and computing.
  ↳ SQL
  Eager vs graph ⇒ when/how program is executed.

	Declarative	Imperative
Productivity	-	+
Debugging	-	+
Static analysis/optimization	+	-

* Python optimizers - Pyfy, Numba, cython

* Concrete Syntax tree (CST)
  ↓ optimize
  Abstract syntax tree (AST)

Language
↓
Bytecode → JIT cache (Ahead of Time compile)
↓ JIT compile
Interpreter (VM) → Binary
↓
CPU (Hardware)

* Cython is general purpose, easier to distribute than Numba.

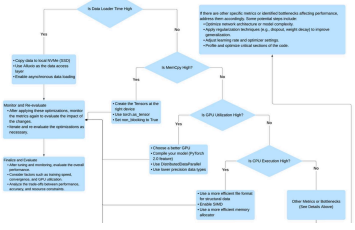
Jit optimizations ⇒ fusion, out of order execution and automatic work scheduling
↳ lazy evaluation

* Possible bottlenecks → CPU, GPU, memory, network, I/O (disk)

* Profiling → Counting, sampling, tracing, analyse
  increasing overhead → high overhead
  ↳ not deterministic

PyTorch profiler, profile, Cprofile, pstats

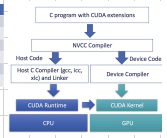
Tool	Description
nvidia-smi	This tool provides information about GPU utilization, memory usage, and other metrics related to the NVIDIA GPU.
htop	It is a command-line tool that hierarchically displays system processes and provides insights into CPU and memory usage
iotop	With this tool, you can monitor I/O usage by displaying I/O statistics of processes running on your system
gpustat	It is a Python-based user-friendly command-line tool for monitoring NVIDIA GPU status.
nvtop	Similar to nvidia-smi, nvtop displays real-time GPU usage and other metrics in a user-friendly interface.
py-spy	It is a sampling profiler for Python that helps identify performance bottlenecks in your code.



• Scripting (torch.jit.script)
  • Analyses the code and produces optimized representation
  • Handles dynamic flow
  • Handles models with complex logic like if and for loops
• Tracing (torch.jit.trace)
  • Runs model with example inputs
  • Records specific operations
  • Simpler but limited - cannot handle data-dependent control flow

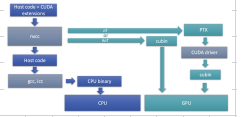
* with record_function ("# forward #") for benchmarking → time it module

Term	Definition
SM	Streaming Multiprocessor
SP	Streaming Processor
TPC	Texture Processing Cluster
GPC	Graphics Processing Cluster
SP	Single-Precision (32-bit)
DP	Double-Precision (64-bit)



* Ct cuda extension → CUDA runtime → CUDA driver

PTX code is considered as assembly for virtual GPU architechture.
x.cu → x.ptx → x.cubin → execute



* CUDA threads execute in SIMD/SIMT. Thread may take different paths.
  CUDA core → streaming → multiprocessor (SM) → GPU
  Threads ∈ blocks ∈ grids → 1 grid/kernel
  3D 3D
  Thread synchronisation → explicit barrier
  generally 32
  ↑
  Warps as scheduling units

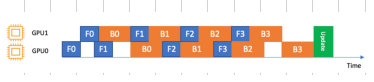
	Executed on:	Callable from:
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

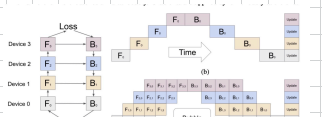
Streaming multi-processors (SM) → maintains thread/block idx, manages thread execution
CUDA context for memory access, host code can be blocking/non-blocking
CUDA streams runs parallelly utilizing hardware
Shared memory is "on-chip" while local/global memory is "off-chip". Local is register spillover
Constant memory is off chip read only from kernel, agressively cached to on chip.

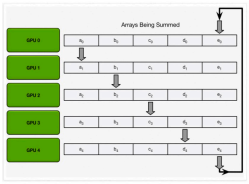
* CUDA UVM - unified virtual memory → if misaligned need 2 mem transactions.
  ↳ coalesce if mem tightly packed.
  Tiling → move tile from global to shared ↳ happens if you go element n for all columns

* PyTorch CUDA ecosystem → Extension and tools (Image and text), distributed training, mixed-precision, profiler

* Parallelism approaches → model ||ism, Pipelining, Data ||ism, Hybrid
  is a form of Parameter Server / All reduce → can be in a ring
  stage time = time without pipelining / number of pipeline stages



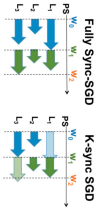




P: number of processes N: total number of model parameters

PS (centralized reduce)
- Amount of data sent to PS by (P-1) learner processes: N(P-1)
- After reduce, PS sends back updated parameters to each learner
- Amount of data sent by PS to learners: N(P-1)
- Total communication cost at PS process is proportional to 2N(P-1)

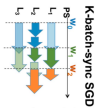
Ring All-Reduce (decentralized reduce)
- Scatter-reduce: Each process sends N/P amount of data to (P-1) learners
  - Total amount sent (per process): N(P-1)/P
- AllGather: Each process again sends N/P amount of data to (P-1) learners
- Total communication cost per process is 2N(P-1)/P



* All reduce got low comm overhead, and PS has to wait for stragglers.

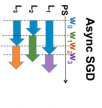
* Downpour SGD → lots of communication, might not converge (momentum issues), no mechanism to remain in the centre.
  → Adagrad within each server shard leads to improvement.
  Decrease α by mean staleness $\alpha = \frac{\alpha_0}{\bar{\tau}}$

$$\eta_j = \min\left\{\frac{C}{\|\mathbf{w}_j - \mathbf{w}_{\tau(j)}\|_2^2}, \eta_{max}\right\}$$



* Increasing power efficiency
  → Compression, quantisation, compilation

Linear Scaling Rule: *When the minibatch size is multiplied by k, multiply the learning rate by k.*



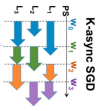
Post training quantisation Quantisation aware training

zero point

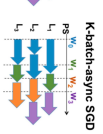
$$\hat{x} = \text{round}\left(\frac{x}{\Delta} - z\right)\cdot\Delta + z \qquad \Delta = \frac{\max x - \min x}{2^n - 1}$$

$$\hat{x} = \text{round}\left(\frac{x}{\Delta}\right)\cdot\Delta \qquad \Delta = \frac{2\max|x|}{2^n - 1}$$

- Linear (uniform)
  - Symmetric/asymmetric
  - Per-tensor/per-channel
- Non-linear (non-uniform)
  - Power-of-two
  - Logarithmic
- Codebook quantization



* Useful when distribution is not normal
  Codewords are selected using clustering or training process.
  Apply kmeans and fine tune centroids
  Just saves storage weight.



Asymmetric $s(\mathbf{x}_{\text{uint8}} - z)$

Symmetric unsigned $s \cdot \mathbf{x}_{\text{uint8}}$

Symmetric signed $s \cdot \mathbf{x}_{\text{int8}}$

* FP32 → INT8 → use scale factor for conversions

Symmetric quantization Asymmetric quantization

$$WX \approx s_W(W_{\text{int}})\, s_X(X_{\text{int}}) \qquad WX \approx s_W(W_{\text{int}} - z_W)\, s_X(X_{\text{int}} - z_X)$$

- Efficient Deep Learning Methods
  - Model compression
    - Pruning
    - Quantization
    - Compact model
  - Code transformation
    - Loop tiling
    - Loop unrolling
    - Operator fusion

* You should simulate quantisation → quantify and de-quantify weights.

$$s = \frac{x_{\max} - x_{\min}}{q_{\max} - q_{\min}} \qquad q = \text{clip}\left(\text{round}\left(\frac{x}{s}\right) + z,\ q_{\min}, q_{\max}\right)$$

$$z = \text{round}\left(q_{\min} - \frac{x_{\min}}{s}\right) \qquad \hat{x} = s\cdot(q - z)$$

$$\text{argmin}_{q_{\min}, q_{\max}}\ \mathcal{E}\left(X, \hat{X}(q_{\min}, q_{\max})\right)$$

MSE Cross-entropy

$$q_{\min} = \min X$$

$$q_{\min} = \min\left(\beta - \alpha\gamma\right) \qquad q_{\max} = \max\left(\beta + \alpha\gamma\right)$$

* Pattern based pruning → N:M, Channel pruning
  Magnitude based pruning → absolute values, L1 norm, L2 norm → $\sqrt{|w_i|^2}$
  element wise row wise row wise

Scaling based pruning → filters / output channels with small scaling would be pruned.

Percentage of zero based pruning => because relu produces zeroes

Regression based pruning → think of channels as co-efficients

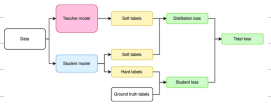
Pruning ratios => sensitivity analysis → might not be optimal because of chanel use iterative ← interactions.

Regularization while fine-tuning - penalise non-zero parameters, encourage smaller parameters

Knowledge → Relation based, feature based, response based.

Soft labels give information about the wrong classes as well.

$$q_i = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}}$$



* FitNets → train deeper and thinner students
  → learn from hint layers to guided layer

* Distillation → online, offline, self

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

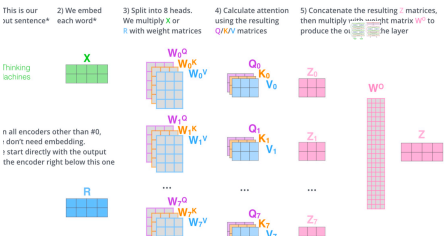
* Feed forward nn is easy to parallelise

KL divergence:
- Non-symmetric metric
- Measures relative entropy or difference in information between distribut
- Indicates the distance between two data distributions
- Reflects dissimilarity between distributions

take average for distillation loss

$$D_{KL}(p(x)\,||\,q(x)) = \sum_{x\in X} p(x)\ln\frac{p(x)}{q(x)}$$

$$D_{KL}(p(x)\,||\,q(x)) = \int_{-\infty}^{\infty} p(x)\ln\frac{p(x)}{q(x)}\,dx$$



$$L_{\text{total}} = \alpha L_{\text{student}} + (1-\alpha)L_{\text{distill}}$$

$$L_{\text{student}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c}\log(p_{i,c})$$

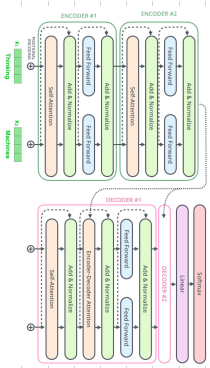
$$L_{\text{distill}} = \frac{1}{N}\sum_i D_{\text{KL}}(P_{\text{soft}}(i)\,\|\,Q_{\text{student}}(i))$$

Relationships can be modeled as:
- Correlation between feature maps
- Graphs
- Similarity matrix
- Feature embeddings

* Flash attention → minimises I/O → read 3 blocks at once

2. FlashAttention: main ideas

FlashAttention (Dao et al., 2022) is an **exact rewrite** of attention that is:
- **IO-aware**: minimizes reads/writes to slow global memory.
- Uses **tiling** so everything fits in **on-chip SRAM (shared memory / registers)**.
- Uses a **streaming softmax** so you never store the full attention matrix.



* Scaling choice → dataset size, model size
  Constraints → hardware

* Parameter efficient fine tuning → only train a subset of weights. (selective)
  → Reparameterization - LoRA
  → Additive - Adapters, prompt tuning

- **Constant Expressions**: Evaluating constant expressions at compile time rather than at runtime. For example, an expression like 2 + 2 will be replaced with 4.
- **Strength Reduction**: Replacing more expensive operations with simpler ones. For example, x * 2 would be optimized to x << 1.
- **Removing Unused Imports**: If an import statement is declared but not used, it can be removed.
- **Optimizing Short Sequences**: Replacing short sequences of instructions with more efficient ones. For example, replacing a sequence of LOAD_CONST instructions with a single BUILD_TUPLE instruction.
- **Variable Lookup Optimization**: Optimizing the lookup of variables by replacing global variable lookups with faster local variable lookups when possible.

- Scaling law form:
  - Variables:
    - $x = N$ (params) or D (tokens) or C (compute).
  - Chinchilla rule-of-thumb:
    - For fixed compute C: optimal when
  - Meaning:
    - Bigger model and more data → lower loss.
    - For given compute, don't oversize the model, train it longer on more tokens.

$$\mathcal{L}(x) \approx \mathcal{L}_\infty + Ax^{-\alpha}$$

$$D \propto N, \quad C \propto ND$$