Machine Learning Systems and Hardware

L2: Compilation and Mapping

Hongxiang Fan

IMPERIAL

Transition from Algorithms to Systems and Hardware

- Recap (Last Lecture): Explored a variety of neural network architectures
  - Convolutional Neural Networks (CNNs)
  - Transformers (Attention-based Neural Networks)
  - *Emerging architectures: Diffusion models, Mamba, etc.
- Design Decisions Impact Both Algorithm & Hardware:
  - Floating point operation (FLOP) count
  - Parameter counts /model size
- Key insight: Regardless of architecture, the core computation focuses on:
  - Matrix Multiplication (MatMul)
  - Activation Functions and Normalization

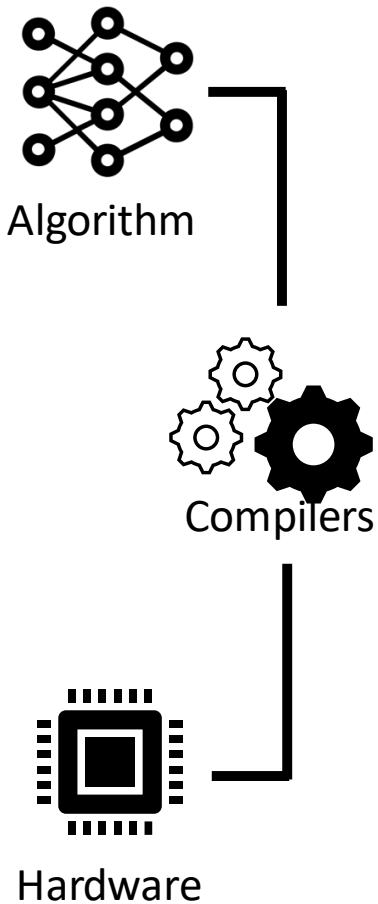
CATALOG

01
ML Compilation

02
ML Hardware Basic

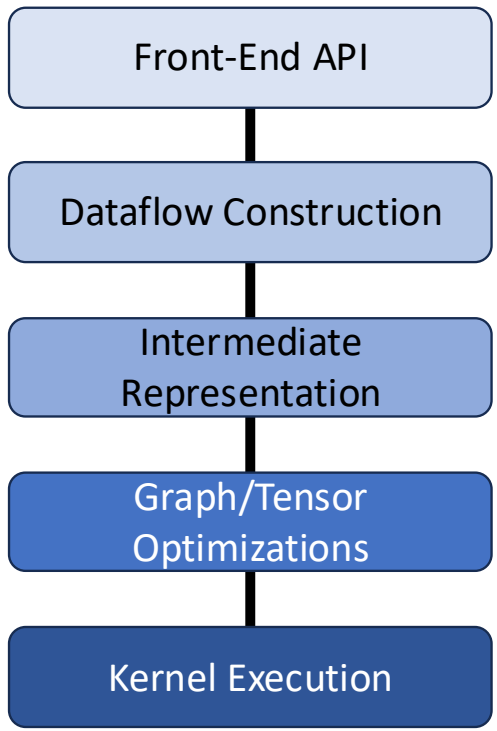
Why ML Compilation

- Algorithmic Complexity (FLOPS) $\neq$ Hardware Performance
  - Any example we visited before?
  - Depth-wise convolution: fewer FLOPs but limited parallelism, high memory bandwidth cost
  - End-to-end performance depends on:
    $e2e\_Perf = F(\boldsymbol{FLOPs}, \quad \boldsymbol{Infrastucture}, \quad \boldsymbol{Hardware})$
- Role of ML compilers
  - Bridge between algorithm and hardware
  - Decide how computation is scheduled, fused, and mapped to specific hardware units (e.g., GPU tensor cores)
  - Translate algorithmic improvements into real speed-up



Overview of ML Compilers

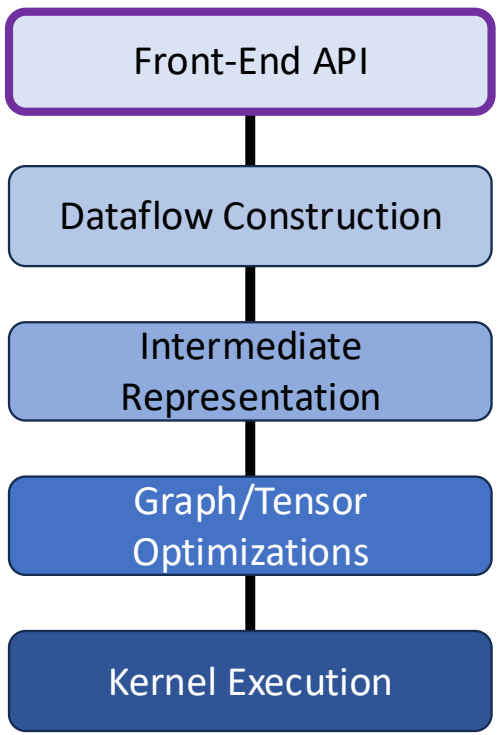
- Front-end API for model construction
  - Imperative/declarative(symbolic)
- Dataflow graph construction
  - Capture computation as nodes (operations) and edges (flows)
- Intermediate Representation (IR)
  - From high-level framework-specific IR to hardware-agnostic IR
- Graph/Tensor optimizations
  - Operator fusion, constant folding, algebraic simplification
  - Layout transformations, memory planning, op reordering
- Kernel execution
  - Vendor libraries (cuBLAS, cuDNN, MKL), custom kernel design
- Hardware
  - GPU, TPU, CPU, ASIC



Front-End: Imperative Programming – PyTorch

- Philosophy: Build-**and**-Execute
  - Operations are executed immediately as Python code runs
  - No explicit computation graph needs to be defined beforehand
- Advantages:
  - Easier for debugging: inspect tensors at any point
  - Beginner-friendly: familiar Python control flow
- Example: Fully Connected (Linear) Layer in PyTorch

```
import torch
import torch.nn as nn

model = nn.Linear(10, 1)
x = torch.randn(1, 10)
y = model(x)  # executes immediately
loss = y.sum()
loss.backward()
```



Front-End: Declarative Programming – Tensorflow

- Philosophy: Build-**then**-Execute
  - Define a computation graph first, then execute it in a session
- Advantages:
  - Easier for compilers to optimize
  - Well-suited for distributed execution and batching
- Example: Fully Connected (Linear) Layer in early version of Tensorflow

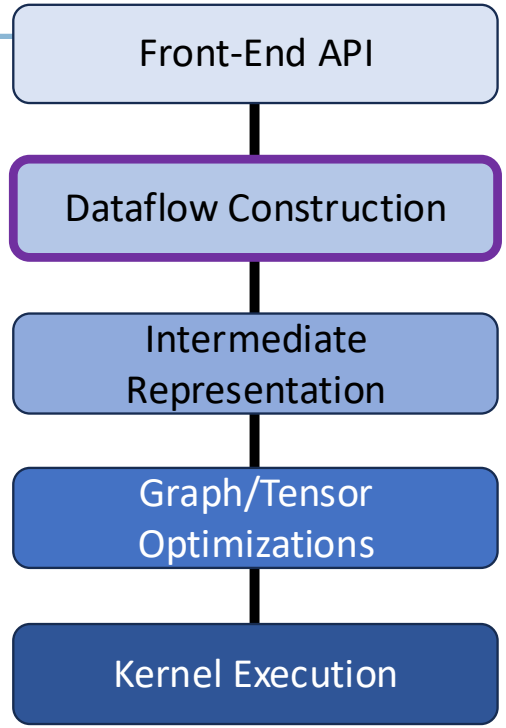
– it is easier to optimize (target certain dev.) in tensorflow

```
import tensorflow as tf

# define computational graph
x = tf.placeholder(tf.float32, shape=(1, 10))
W = tf.Variable(tf.random.normal((10, 1)))
y = tf.matmul(x, W)
loss = tf.reduce_sum(y)
```

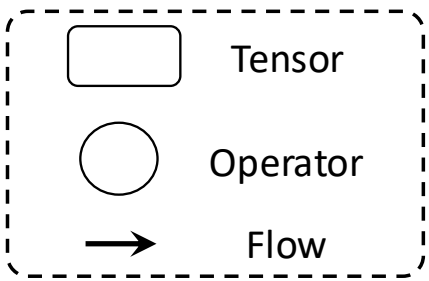
```
# launch run
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    sess.run(loss, feed_dict={x: [[0.5]*10]})
```

Dataflow Construction

- Why dataflow graphs?
  - Makes data dependencies explicit
  - Facilitate graph/tensor optimizations (e.g., reordering)
- Dataflow representation:
  - Directed acyclic graph (DAG)
  - Rectangle node: tensor with shape, dtype, layout, etc.
  - Square node: operation with attributes (e.g., stride, padding)
  - Edges: data dependencies and control flow
- Inference: forward DAG only
- Training: forward + gradient update DAG
  - Gradient update DAG is constructed by Autodiff engine
  - Autodiff engine: Forward Mode Autodiff* & Backward Mode Autodiff (Link)
  - As **Backward Mode** is mainstreaming, this module only focus on this



creates backward pass
paths with derivatives included



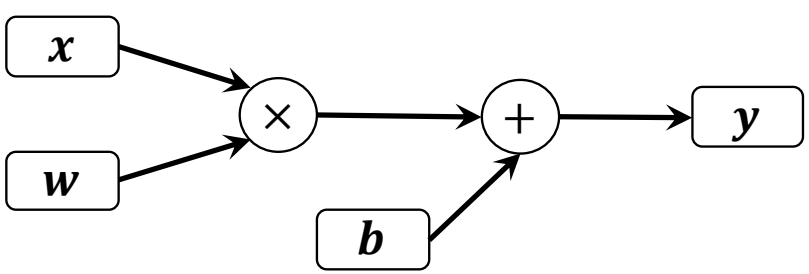
Dataflow Construction: Forward DAG

- Example of linear transformation

$$\boldsymbol{y} = \boldsymbol{x} \cdot \boldsymbol{w} + \boldsymbol{b}$$

- Forward DAG construction
  - Encodes pure tensor computation: Matmul ($\boldsymbol{x} \cdot \boldsymbol{w}$) and addition ($+b$)
  - Tensor nodes: $\boldsymbol{w},\ \boldsymbol{x},\ \boldsymbol{b},\ \boldsymbol{y}$
  - Op nodes: $Mul\ (\times),\ Add(+)$
  - No gradient-related nodes.

Forward DAG

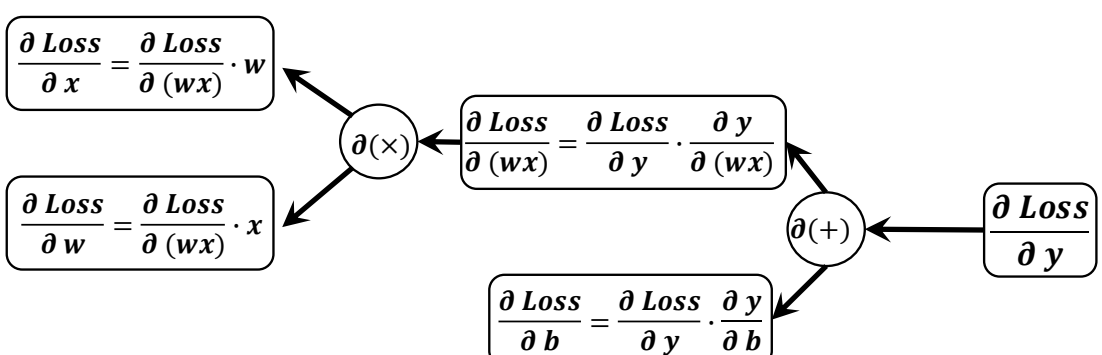


Dataflow Construction: Backward DAG

- Goal: Compute gradients based on loss function: $\partial\, Loss(.)$
  - Adds gradient operations: Handle by Autodiff engine
  - Tensor nodes: $\frac{\partial\, Loss}{\partial\, y}, \frac{\partial\, Loss}{\partial\, x}, \frac{\partial\, Loss}{\partial\, w}, \frac{\partial\, Loss}{\partial\, b}$
  - Addition → ∂(+): Sum rule (branching)
  - Multiplication → ∂(x): Product rule

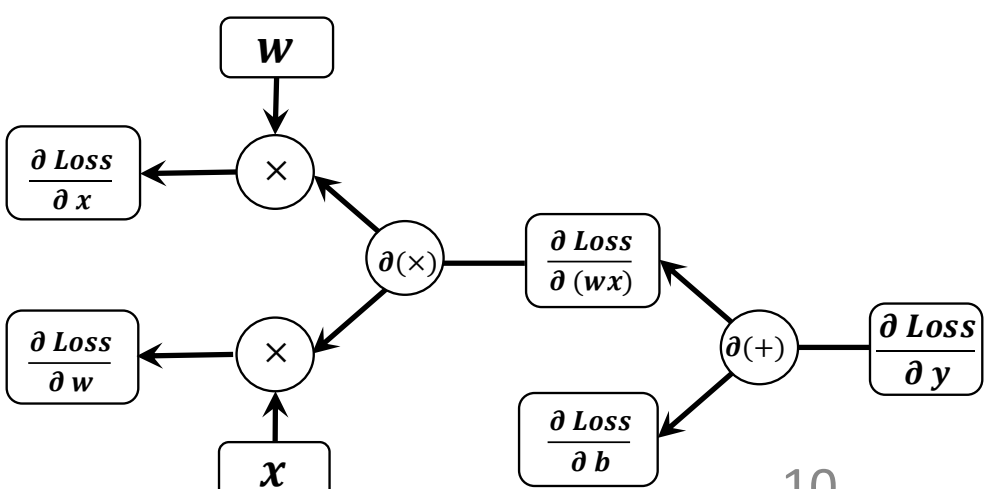
↳ this is what pytorch does automatically

$y = w \cdot x + b$



– always applying chain-rule for derivatives

Backward DAG

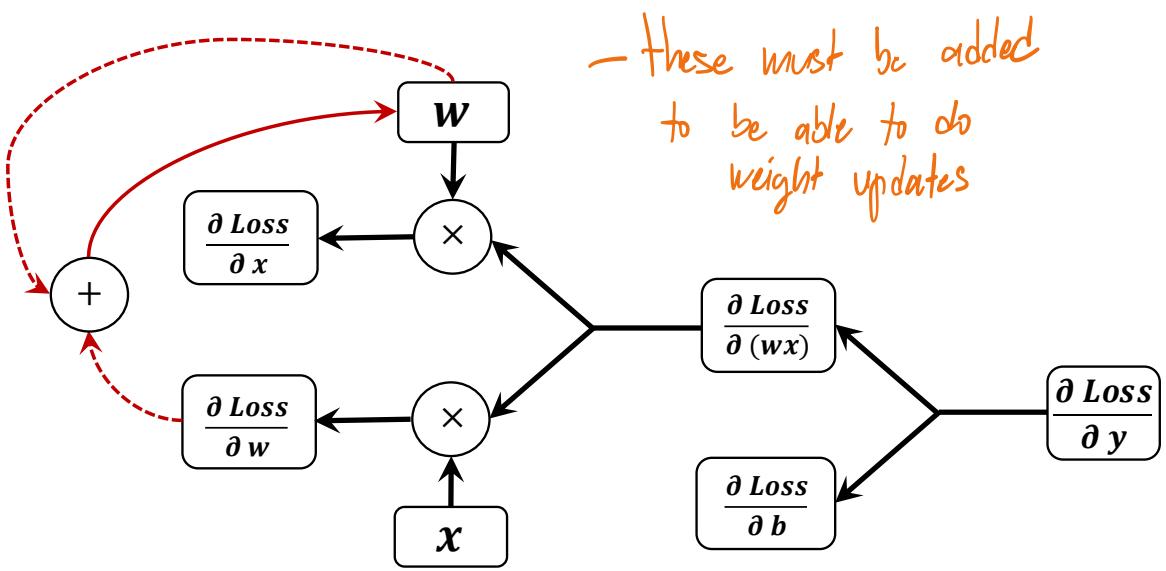


Dataflow Construction: Weights Update

- Update learnable parameters using gradients
  - Gradient Decent:

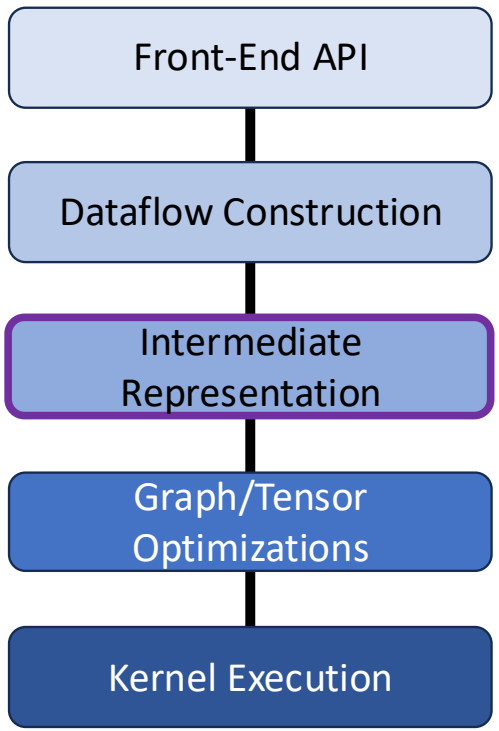
$$w^{t+1} = w^t - \eta \frac{\partial\, Loss\,(.)}{\partial\, w}$$

  - Different variants to fit real scenarios constraints



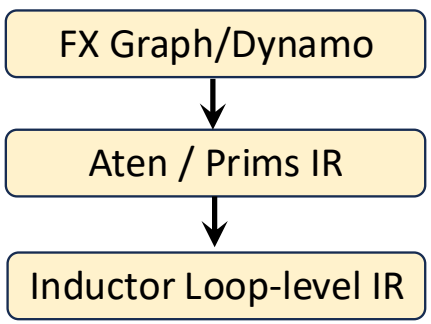
Intermediate Representation

- Why Dataflow Graph Is Not Enough?
  - Captures only functional dependencies
  - Lacks optimization metadata (e.g., layout, loop structure)
- Why Intermediate Representation (IR):
  - Bridge between high-level models and low-level kernel execution
  - Provides richer, structured abstraction for advanced graph & tensor optimizations
- IR designs and stacks:
  - Pytorch: FX Graph → Aten/Prims IR → Inductor Loop-level IR
  - Tensorflow: MLIR (TF Dialect) → HLO
  - Triton-Python: Triton IR → LLVM IR
  - Torch-MLIR: Unified bridge to MLIR ecosystem [GitHub]



Intermediate Representation

- Key Design Principles
  - Abstraction: Remove unnecessary details to simplify transformations
  - Layered Representation: Different IRs for different optimization scopes
  - Lowering Pathway: IRs gradually transform computations into backend-executable
- Example: PyTorch IR Stack
  - FX Graph: operator fusion, constant folding …
  - Aten / Prims IR: type promotion, broadcasting ops …
  - Inductor Loop-level IR: loop fusion, loop unrolling, memory layout optimization …

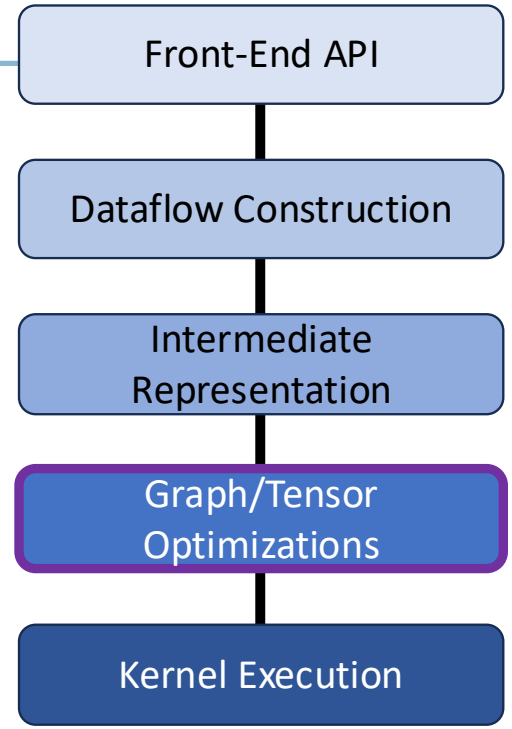


Multi-Level Intermediate Representation (MLIR)

- A compiler infrastructure that facilitate developers to design custom IR
  - Introduced by Google in 2019
  - MLIR: A Compiler Infrastructure for the End of Moore's Law
- Key design philosophy:
  - Multi-Level IRs: Capture computation at different levels of abstraction
  - Progressive lowering: Gradually refine high-level IRs into lower-level ones
  - Extensibility via Dialects: Easily define domain-specific ops/types with custom semantics
    - Dialects: a logical grouping of Ops, attributes and types under a unique namespace
- Trade-offs and Limitations:
  - Increased IR Complexity: Complexity shifts to managing dialects and lowering passes
  - Steep Learning Curve: Requires understanding of dialect design, and transformation rules

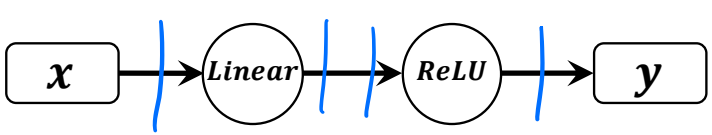
Optimization-1: Operator Fusion — big factor for optim.

- Combine adjacent ops into a single fused op to reduce overhead
- Benefits:
  - Reduced kernel launches
  - Lower memory access & transfer cost
  - Better backend-specific fused kernels
- PyTorch Example: Linear and ReLU
- Note: Operator fusion is enabled by FX and TorchInductor together in PyTorch



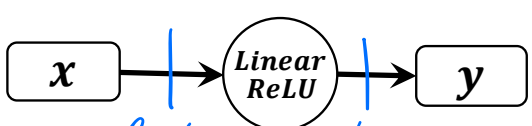

```
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(4, 4),
    nn.ReLU()
)
```

Unoptimized → this is inefficient



I need memory operations here

Optimized

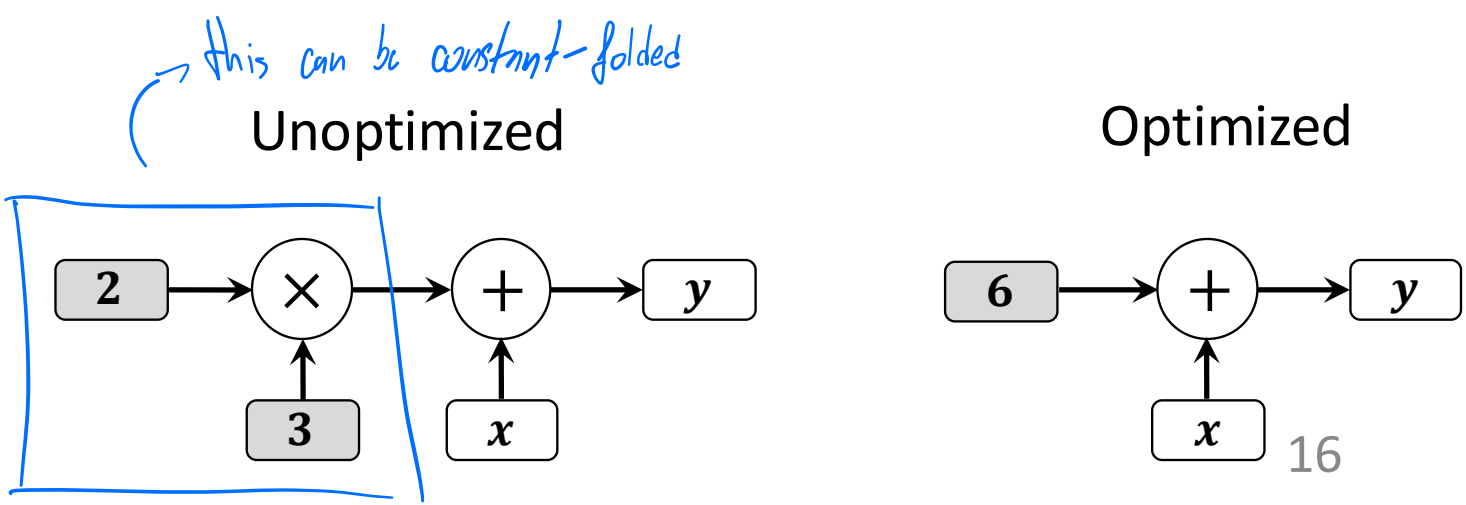


load store only

Optimization-2: Constant Folding

- Fold constant subexpressions at compile time to reduce runtime computation
- Benefits:
  - Eliminates redundant calculations
  - Reduces runtime overhead
  - Simplifies computation graph
- Can be applied in high-level IRs in early stage

```
import torch
x = torch.tensor([1.0])
# constant subexpression
y = x + 3 * 2
```



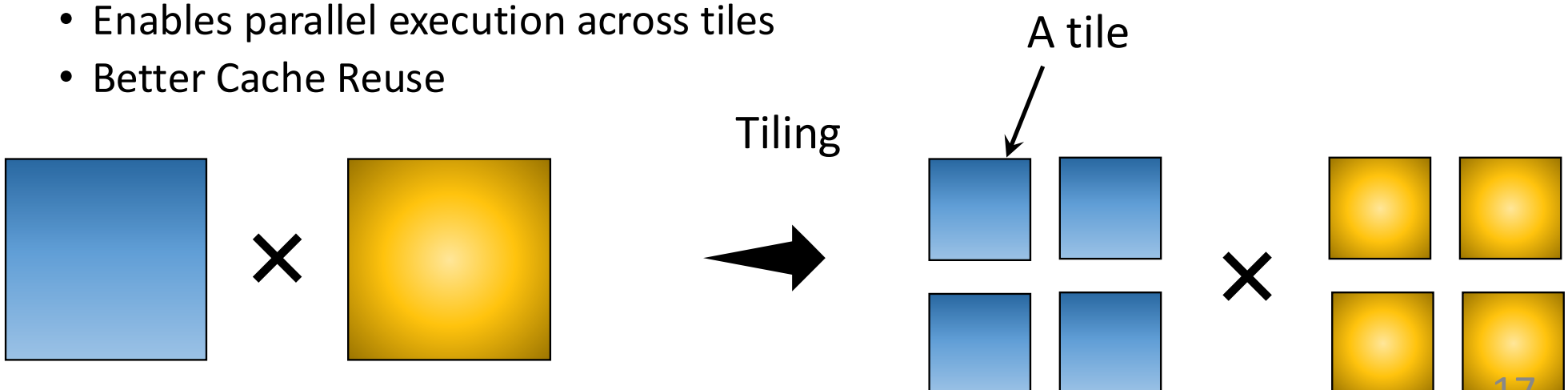
Optimization-3: Tiling

– Datové struktury, Fish

- Key operations in deep learning: matrix multiplication
  - Input matrices: $\boldsymbol{A}, \boldsymbol{B}$ Output matrix: $\boldsymbol{C}$

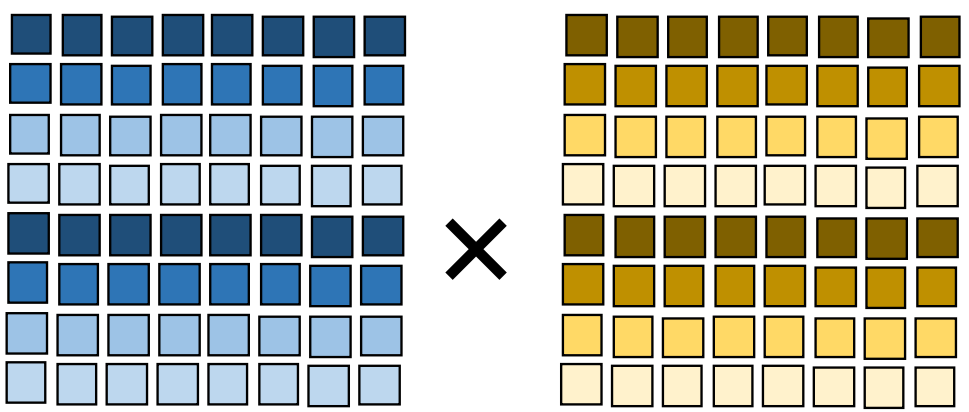
$$\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$$

- Core idea: divide large computations into smaller blocks (tiles) that fit in memory/cache
- Benefits of Tiling:
  - Improved Data Locality
  - Enables parallel execution across tiles
  - Better Cache Reuse

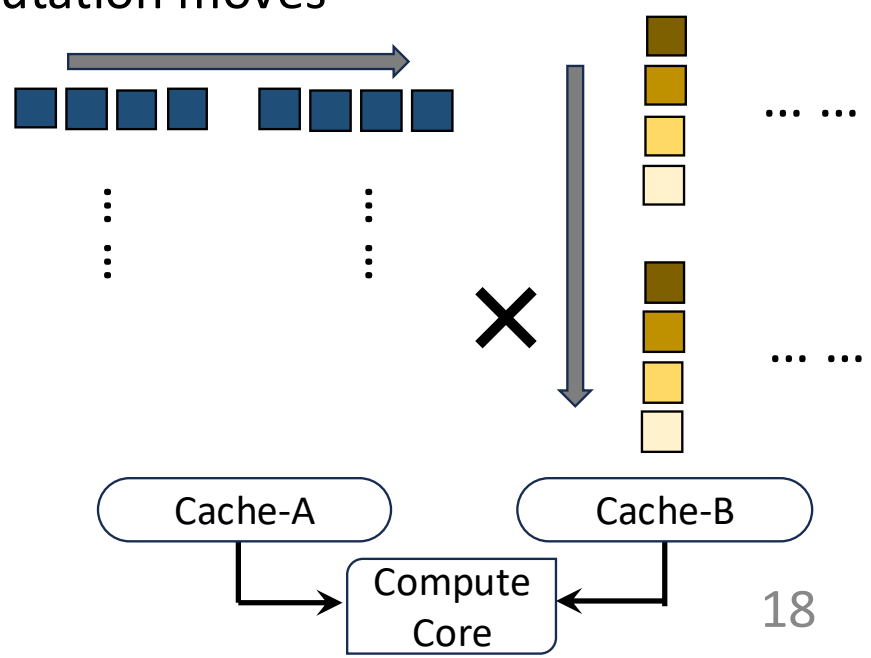


Optimization-3: Tiling

- Assumption: full matrices cannot be fitted into cache:
  - Frequent data eviction → poor data locality
- A concrete example: 8×8 matrix multiplication (naïve implementation)
  - One compute core with two small input caches (each holds 4 elements)
  - Entire row from A and column from B needed for one output element
  - Old cache contents are evicted as the computation moves

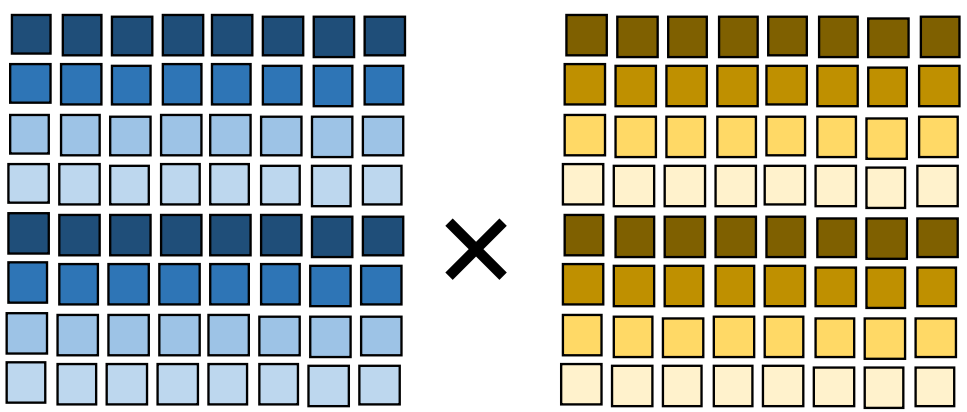


```
for i in range(8):
    for j in range(8):
        for k in range(8):
            C[i][j] += A[i][k] * B[k][j]
```

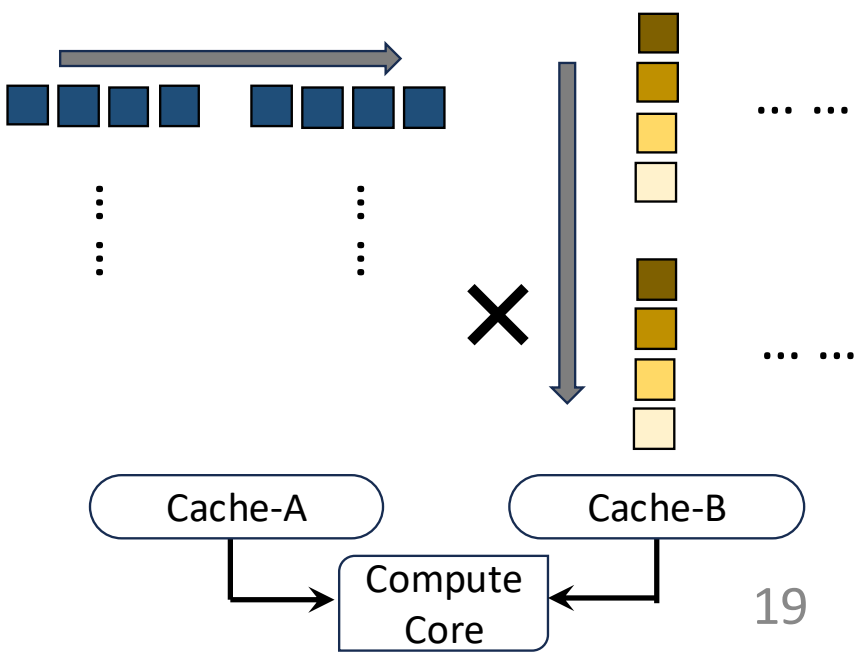


Optimization-3: Tiling

- Data locality: reuse of data once it has been loaded into cache
  - **Temporal locality**: reusing the same data soon after it was accessed
  - Spatial locality: accessing data that is close in memory to recently accessed data
- Poor data locality in naïve implementation
  - No data reuse: cache eviction
  - 4 multiplication per data load ([1,4] * [4,1])

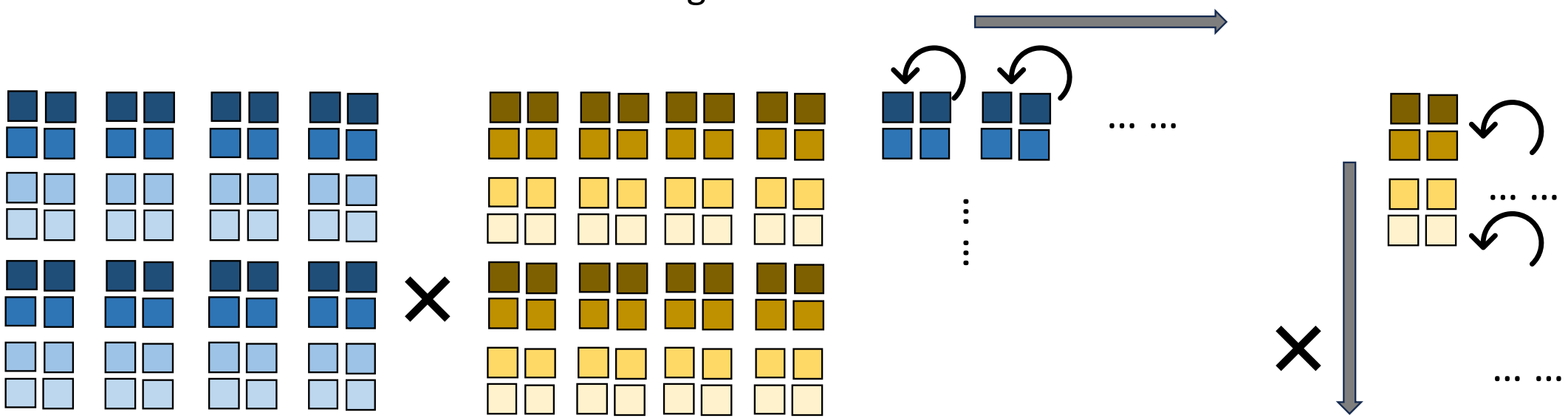


```
for i in range(8):
    for j in range(8):
        for k in range(8):
            C[i][j] += A[i][k] * B[k][j]
```

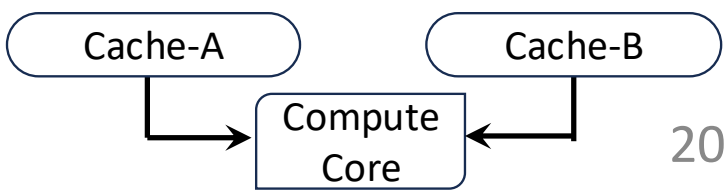


Optimization-3: Tiling

- Tiling: multiple 2×2 tile blocks
  - Splitting along i, j, k dimensions, creating three more loops for tile control ii, jj, kk
  - 8 multiplication per data load ([2,2]*[2,2])
  - Better data reuse with caching

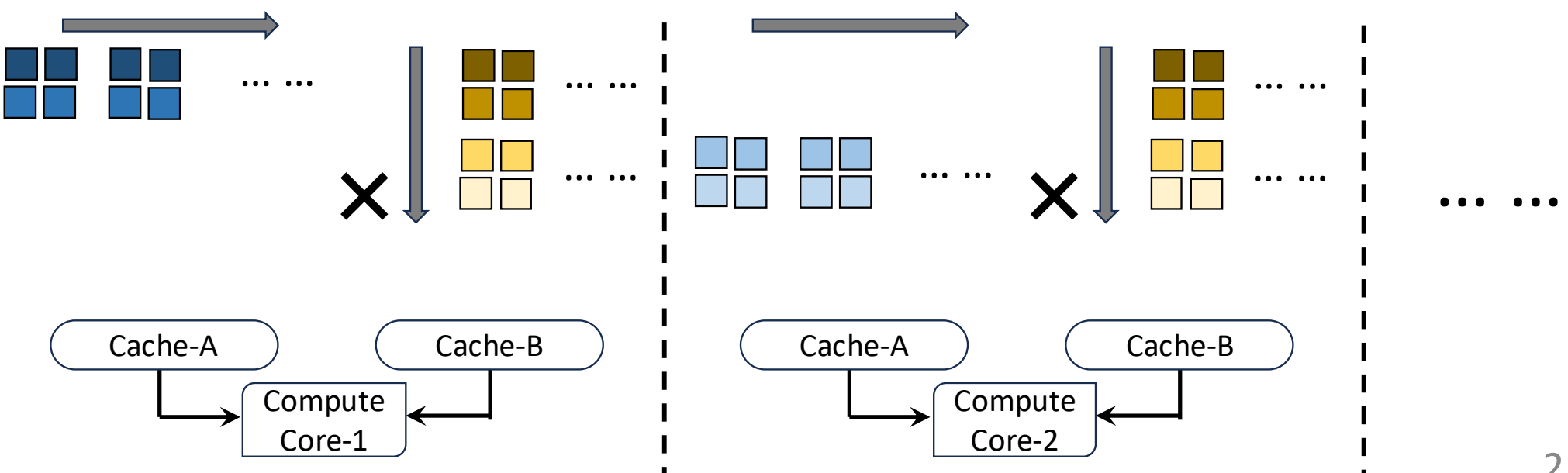


```
for ii in range(0, 8, 2):       # tile row
    for jj in range(0, 8, 2):   # tile col
        for kk in range(0, 8, 2): # tile k
            for i in range(ii, ii+2):
                for j in range(jj, jj+2):
                    for k in range(kk, kk+2):
                        C[i][j] += A[i][k] * B[k][j]
```



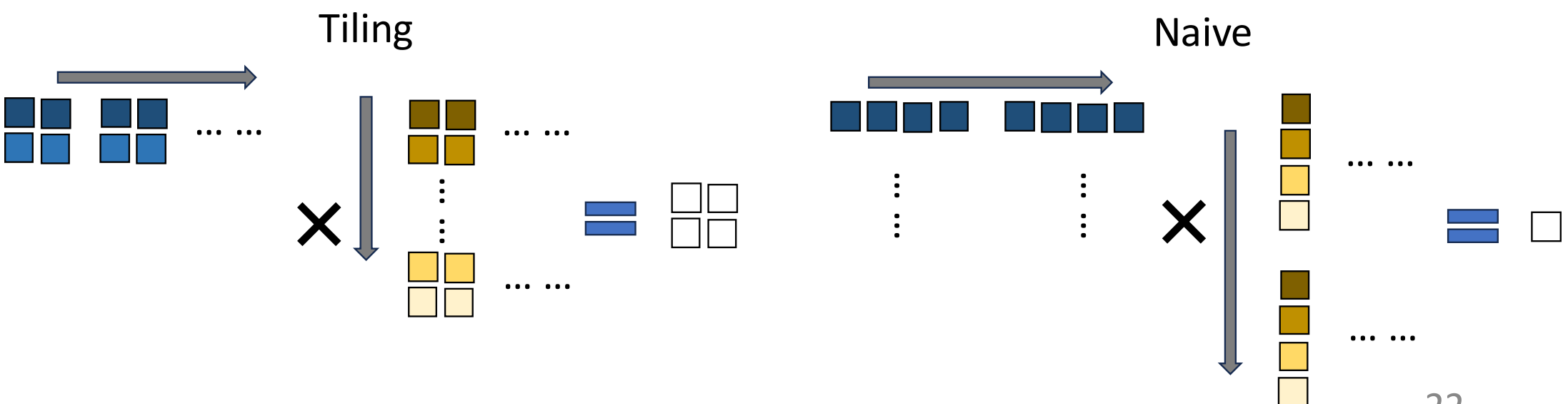
Optimization-3: Tiling

- Enable parallel execution across tiles
  - Each core independently computes a subset (tile)
  - Input tiles are loaded into local caches (Cache-A and Cache-B)
  - Efficient workload distribution
  - Reduced contention for memory bandwidth



Optimization-3: Tiling

- No free lunch: Extra memory to cache intermediate results
- Key trade-off:
  - Hardware constraints: Cache size, memory hierarchy, and bandwidth
  - Memory manipulation: Requires careful management of on-chip shared memory
  - Edge case handling: Irregular shapes or incomplete tiles must be handled separately

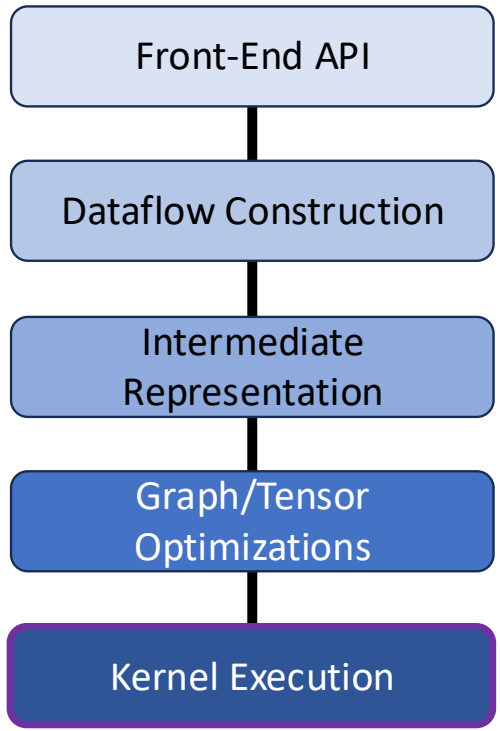


Optimization and Beyond

- Additional optimization techniques not covered yet:
  - Loop-level optimizations: unrolling, reordering, etc.
  - Memory& layout optimization: memory coalescing, layout transformation, etc.
  - Hardware-specific Optimizations: instruction scheduling, tuning approaches, etc.
- Why not cover them now?
  - Many require knowledge on computer architecture of AI hardware:
    - Compute core and pipeline design
    - Memory hierarchy (e.g., registers, caches)
    - Interconnects and communication
- These topics will be revisited later in the module:
  - Concrete examples
  - Hands-on tutorials

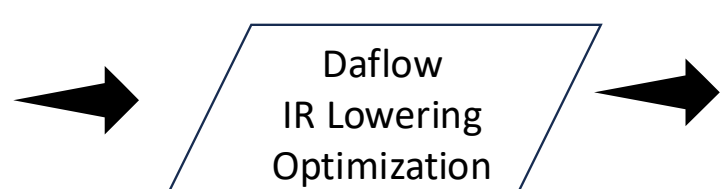
Kernel Execution

- Final stage after graph-level and IR-level optimizations
- Two main backend strategies:
  - Vendor-optimized libraries
    - Call precompiled kernels (e.g., cuBLAS, cuDNN, MKL, MIOpen)
    - High performance with minimal codegen overhead
    - Limited flexibility and harder to fuse or schedule across ops.
  - Custom kernel generation
    - Generate hardware-specific code (e.g., CUDA, Triton, Metal, or C++)
    - Enables kernel fusion, layout tuning, tiling, etc.
    - More flexible and tunable, but codegen is more complex.



Example of Calling vendor-optimized library

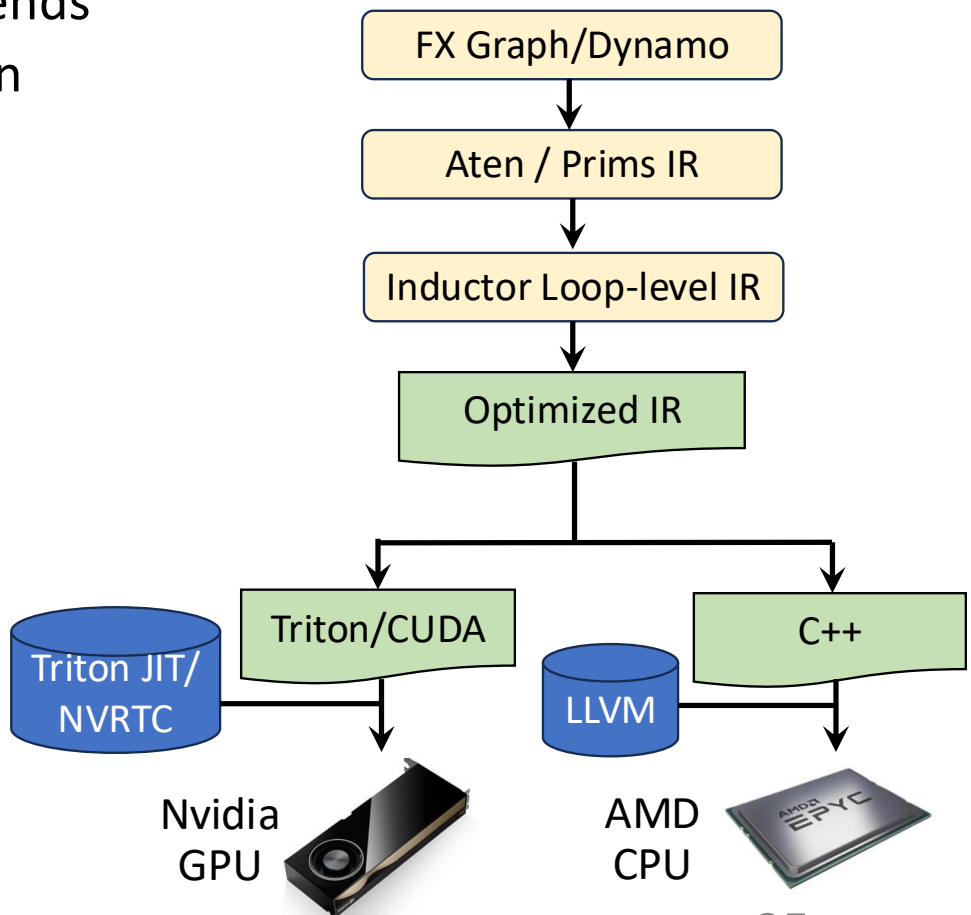
```
# High-level PyTorch
C = torch.matmul(A, B)
```




```
# After lowering, backend call with cublas:
call_cublasGemmEx(handle, A, B, C, ...)
```

Kernel Execution

- Code Generation Path
  - Different paths for different hardware backends
  - Enables hardware-aware kernel optimization
- GPU backend:
  - CUDA C++ (via NVTRC to PTX)
  - Triton Kernel (via Triton JIT)
  - AMD GPUs take different routes (ROCm IR)
- CPU backend:
  - C++ code (via LLVM)
- Other Architectures (e.g., XPU, IPUs)
  - Requires custom backend support



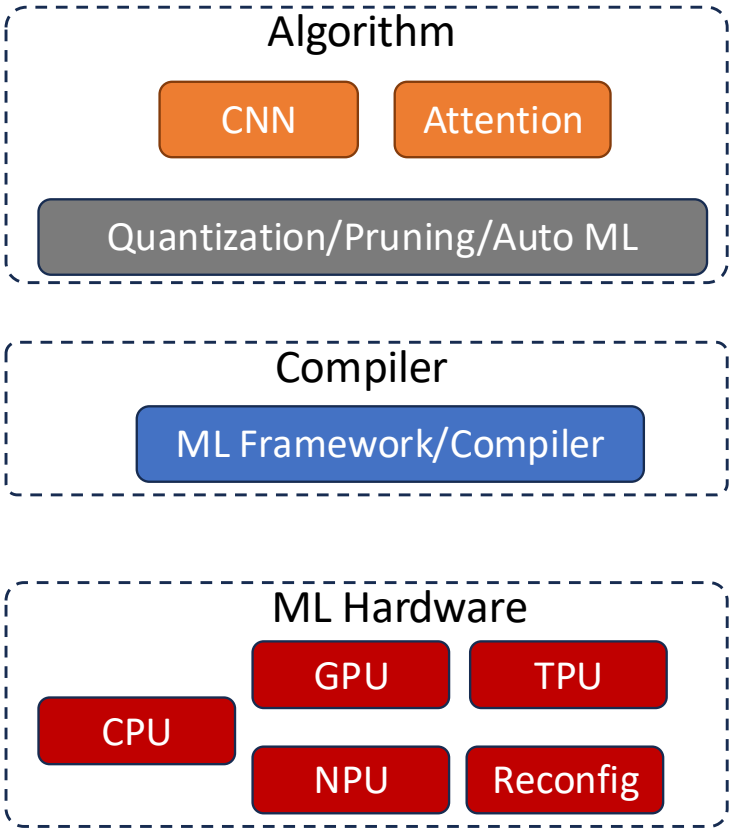
CATALOG

01
ML
Compilation

02
ML Hardware
Basic

ML Hardware

- Until now:
  - Basic DNN algorithms
    - CNN, RNN, Attention-based NN
    - Emerging NN: MAMBA, Diffusion LLM
  - ML Compiler
    - Front-end framework: Imperative & declarative
    - IR stacks: different abstraction and optimizations
    - Kernel execution
- Next to cover:
  - ML Hardware
    - CPU, GPU, FPGA, etc.
    - NPU, Processing in/near memory/sensor
    - Hardware-level optimizations

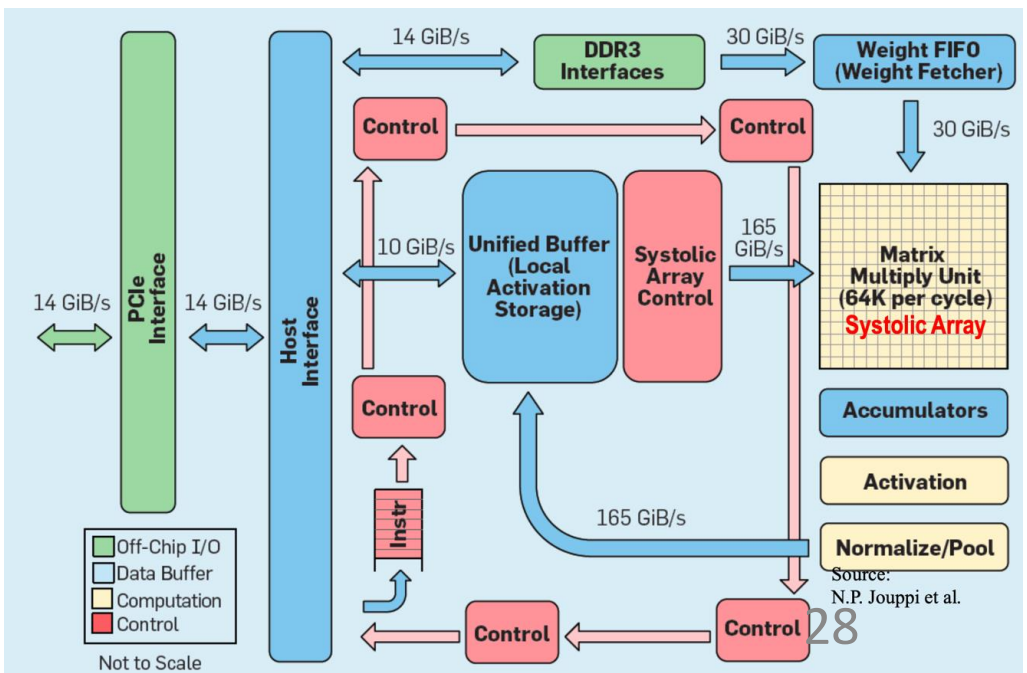


From Basic Computer Architecture

- Art of Computer Architecture
- Structure of a hardware system
  - Main hardware components
  - Interconnects
  - Hardware/software interface

Analogy

Google TPU



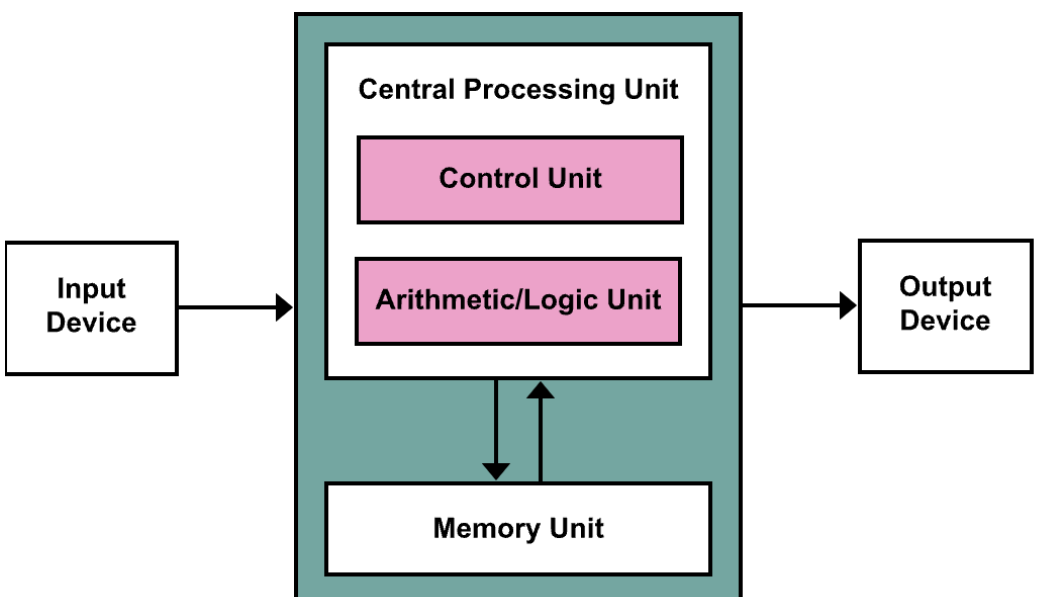
Source:
N.P. Jouppi et al.

From Basic Computer Architecture

- Processing:
  - control system, functionality of compute unit…
- Communication
  - Bus, Interface, bandwidth…
- Storage
  - memory system, caches…

Von Neumann Architecture

- Stored program computer: general purpose
- Unified Memory for both instructions and data
  - Harvard Architecture: Separate access for instructions and data
- Sequential Instruction Programme



World of Trade-Off: Performance Metrics

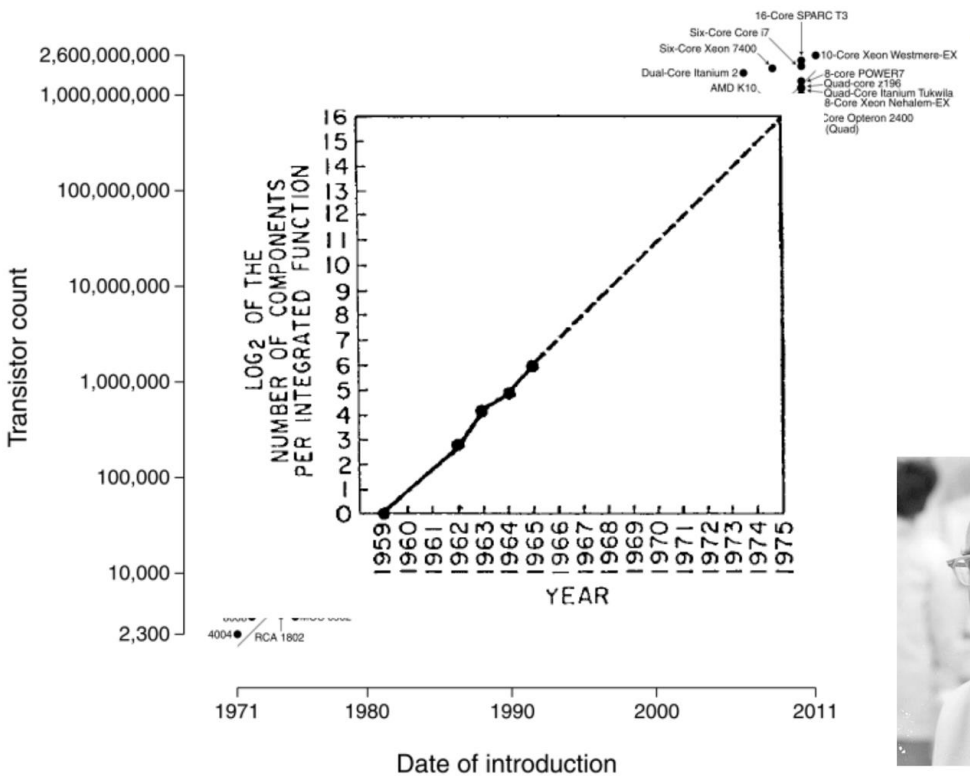
- Core design trade-off: Performance–Power–Area (PPA)
- Performance (P)
  - Latency: Time to complete a task (time to first output)
  - Throughput: Tasks completed per time unit (e.g., frame per second)
- Power (P)
  - Energy consumption over time
  - Affects thermal performance and battery life
- Area (A)
  - Physical silicon footprint (chip size)
  - Influences cost, yield, and scalability
- Impacting Factors:
  - Hardware architecture (e.g., SIMD, pipelining)
  - Technology node (e.g., 5nm, 7nm)

— depends on scenarios (phone vs. data-center)

PPA actually stand for three constraints that must be balanced

Moore's Law

- Number of transistors on an integrated circuit doubles every two years
- Motivation for special-purpose processors



- essentially, if they wait, their non-efficient code will run quickly.

- however the Moore's law does not apply anymore

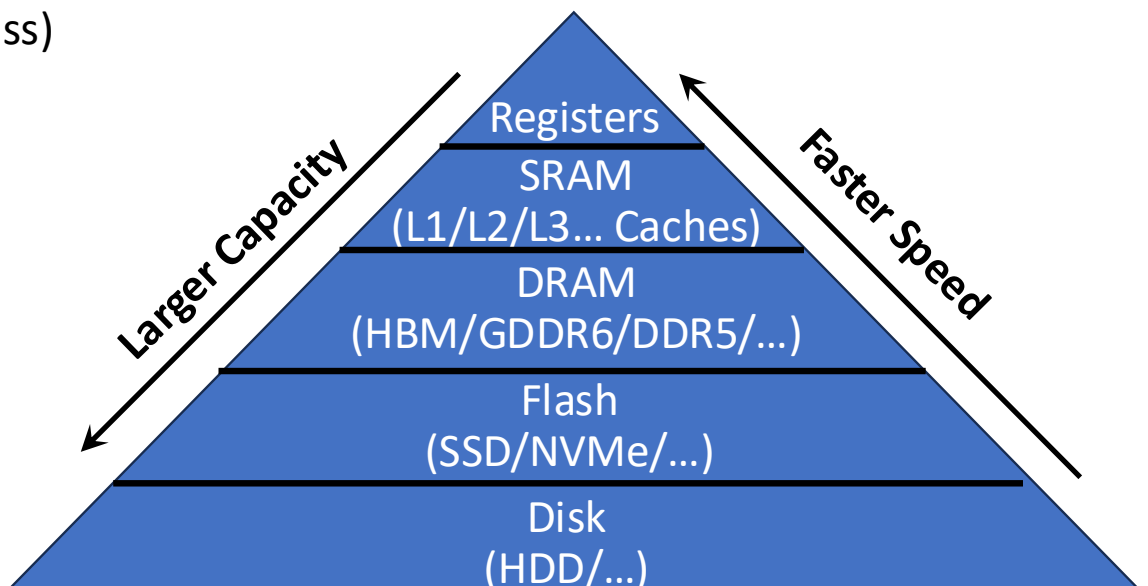
Golden Age of Computer Architecture

- John L. Hennessy, David Patterson, 2017's Turing Award:
  - "A New Golden Age for Computer Architecture", Communications of the ACM, 2019
  - Performance boost brought by technology advanced is slowing down
  - More performance gain from advanced architecture design
- GPU (Graphics Processing Unit)
- Reconfigurable Accelerator
  - FPGA (Field-Programmable Gate Array)
  - CGRA (Coarse-Grained Reconfigurable Array)
- DSA (Domain Specific Architecture)
  - TPU from Google, NPU from Samsung
  - Processing in/near Memory
  - Processing in/near Sensor
  - Processing in/near Network

Different Memory Technologies

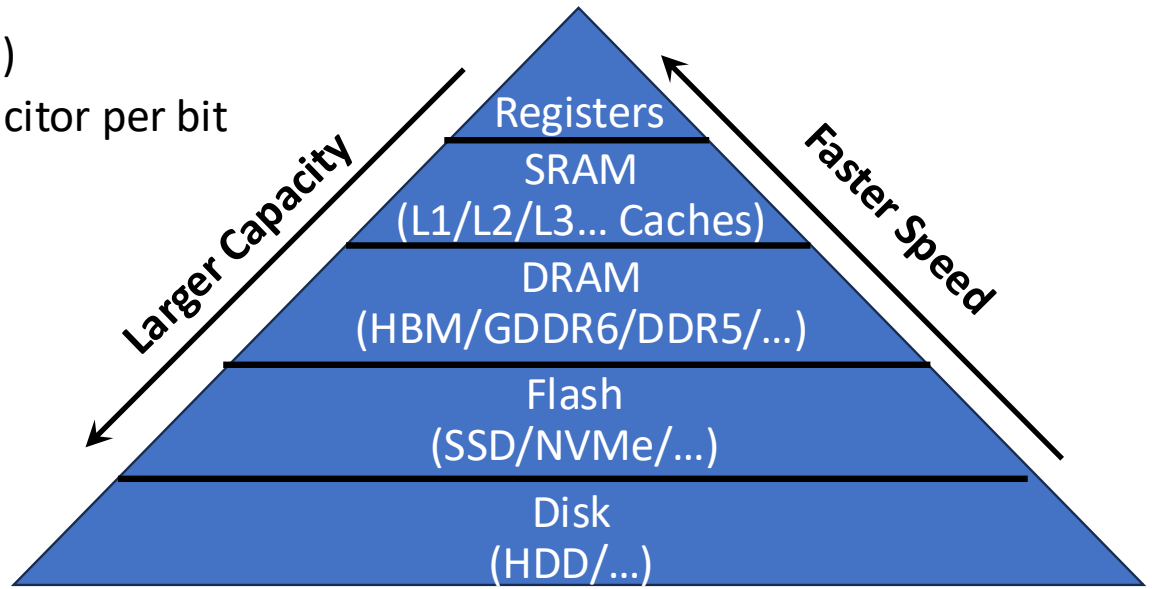
- Gaps in latency, capacity, and bandwidth across different memory tech
- Modern Memory Hierarchy (Top to Bottom):
  - Registers:
    - Fastest, smallest, closest to ALU
    - Transistor counts: >10 (~30-40) per bit
    - Capacity: ~KBs (area constraints and port complexity)
    - Bandwidth: Very high (single-cycle access)
  - SRAM (L1/L2/L3.. Caches):
    - Intermediate latency and capacity
    - Transistor counts: 6 per bit
    - Capacity: Tens of KBs to MBs
    - Bandwidth: High (few cycles)

this applies only to CPU →



Different Memory Technologies

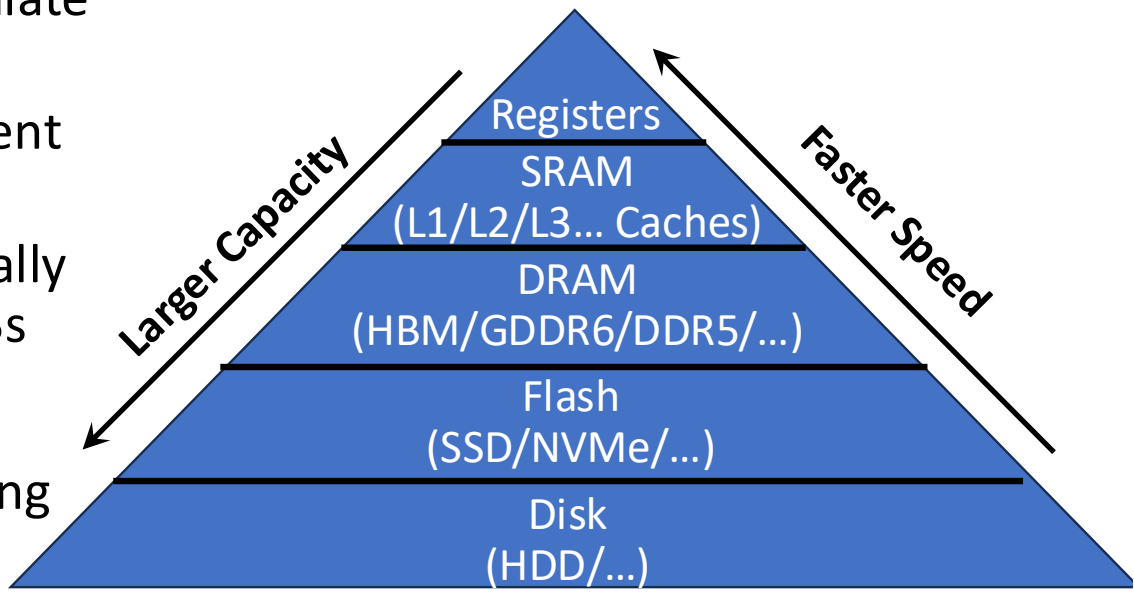
- Modern Memory Hierarchy (Top to Bottom):
  - DRAM:
    - Larger, slower (depends on generation)
    - Transistor counts: 1 transistor + 1 capacitor per bit
    - Capacity: ~ GBs
    - Bandwidth: Lower than caches
  - Flash:
    - Non-volatile storage
    - Capacity: 100s of GB to TB
  - Disk:
    - Mechanical, slowest
    - Capacity: TB+



- Memory hierarchical design varies by different hardware and purposes

Different Memory Technologies

- CPU (e.g., Intel Core i9)
  - Registers: stores operands for immediate use
  - Caches (L1, L2, L3): SRAM with different access speed and capacities
  - Main memory: Off-chip DRAM (typically DDR4/DDR5), tens or hundreds of GBs
- Emerging Hierarchy:
  - Driven by advanced memory packaging technology
  - Example: AMD 3D V-cache, stacking SRAM on top of the compute die to increase cache capacity
- Varying Specifications Across Vendors and Generations: Application driven

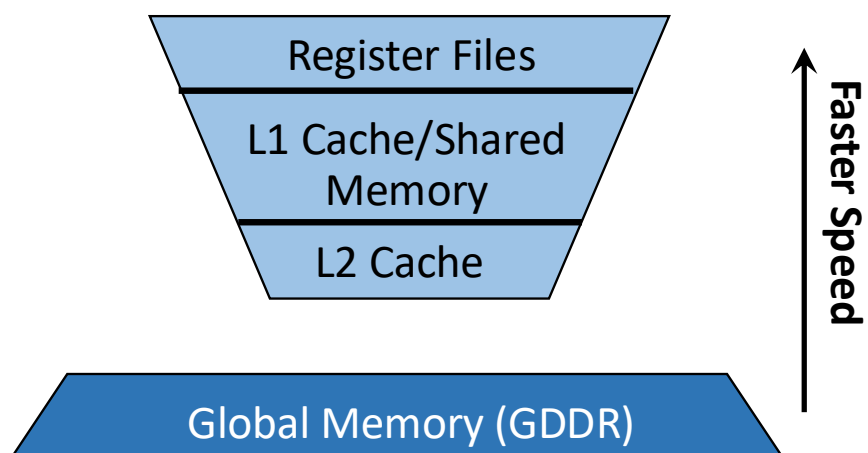


Different Memory Technologies

- Memory hierarchy design also depends on area and power (thermal) limits:
  - SRAM: Fast, but area- and power-hungry — L1, L2, L3...
  - DRAM: Good density and cost, but lower bandwidth and refresh overhead - RAM
  - HBM: High bandwidth and low energy/bit, but complex packaging and thermal limits.
- Scenario Considerations:
  - Edge: Prioritizes low power and small form factor, HBM often unsuitable.
  - Cloud: Can support HBM and large DRAM due to space and cooling.
- Design Space Exploration:
  - Architectural choices demand careful co-design across memory hierarchy:
    - Tapeout and verification costs
    - Physical layout constraints
    - Application-specific access patterns

Different Memory Technologies

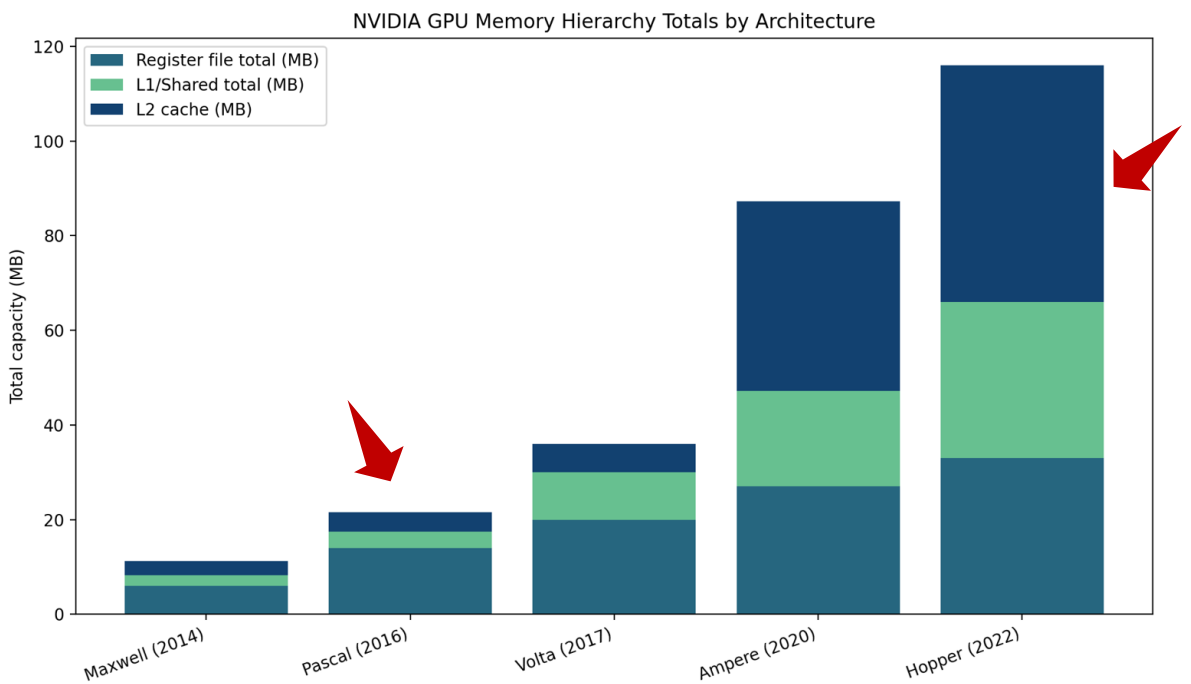
- GPU (e.g., NVIDIA H100):
  - Register file: per-thread storage for operands
  - Caches/Shared Memory: on-chip SRAM for instruction/data caches
  - Global Memory: high-bandwidth DRAM such as HBM3 or GDDR6



SM design of H100, Source: Nvidia

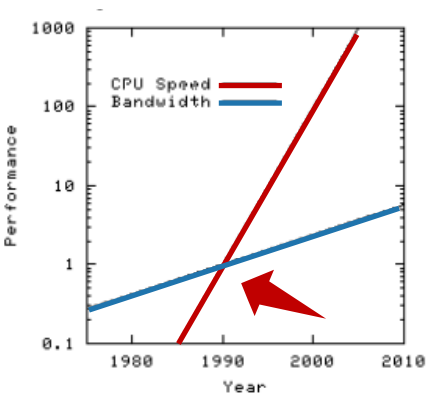
Different Memory Technologies

- Why Different Memory Hierarchy?
  - Physically: More compute cores allows larger register files.
  - Logically: Increasing fast-access memory improves performance.
- Architectures evolve iteratively, driven by application needs
- From NVDIA Maxwell to Hopper Architecture:



Memory Wall

- Origin of the Term:
  - Introduced by Wulf & McKee, 1994
  - "Hitting the Memory Wall: Implications of the Obvious" SIGARCH Computer Architecture News
- Key Concept: Processor growing speed has outpaced memory speed
  - As CPU gets faster, it spends more time waiting for data from memory
  - The latency gap creates a performance bottleneck: "Memory Wall"
- Architectural Facts Behind
  - Cache (SRAM) is fast but small → limited capacity
  - DRAM is large but slow → high latency and energy
  - Data must be fetched from DRAM → leads to pipeline stalls and low compute utilization



Source: Reflections on the Memory Wall

Recap

- ML Compilation Stack
  - Front-end API for model construction
  - Dataflow graph construction (AutoDiff)
  - Intermediate Representation (IR)
  - Graph/Tensor optimizations (Fusion/Tilling)
  - Kernel execution
- ML Hardware Basis:
  - Moore's Law
  - Amdal's Law
  - Memory Wall
    - Registers
    - SRAM
    - DRAM
    - Flash Memory
    - Disk