Deep Learning Compiler

From the golden days to the era of LLM

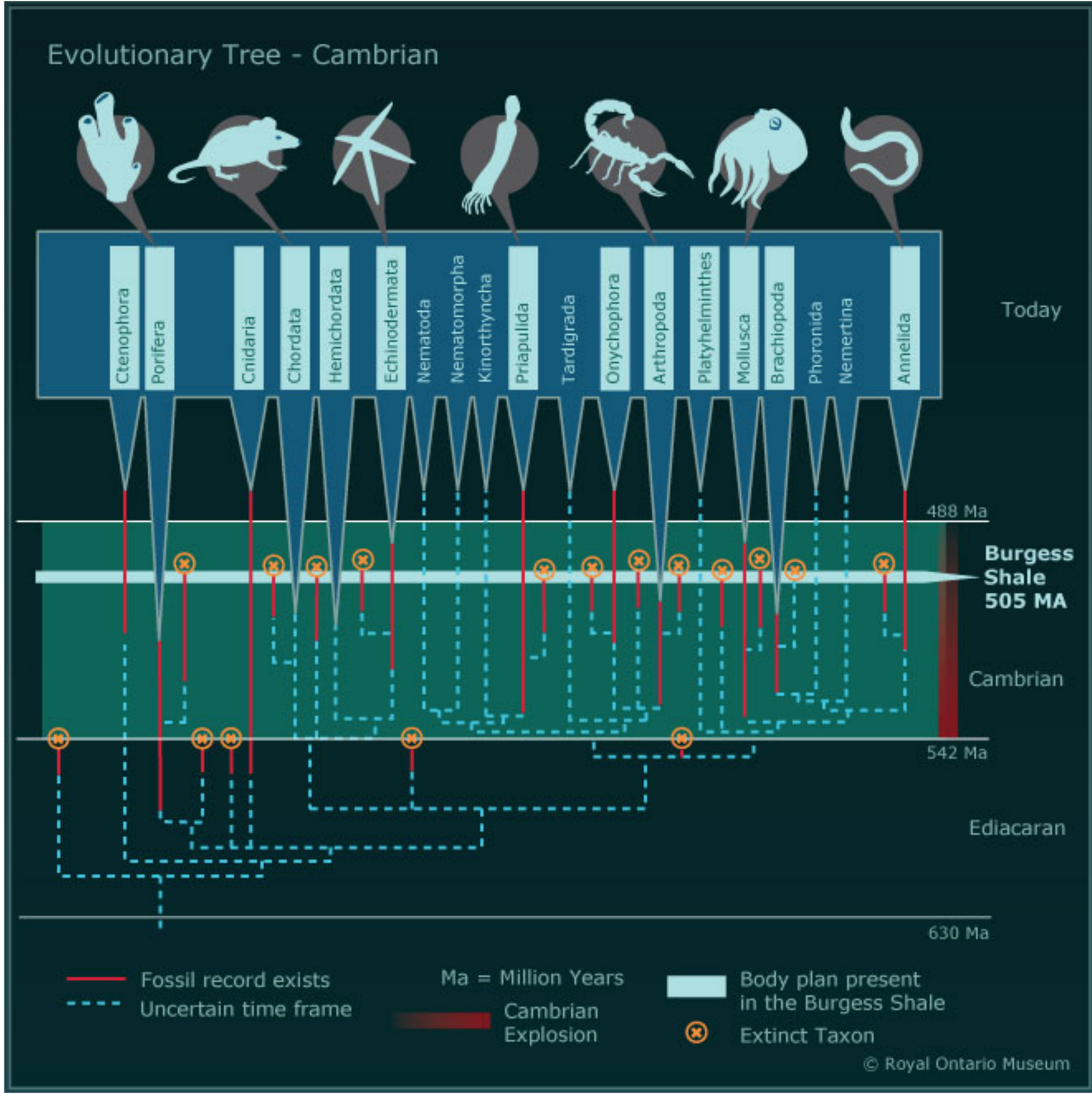
Li Shang
lishang@slai.edu.cn

Acknowledgement

Machine learning systems is a broad and rapidly evolving field. The course material has been developed using a broad spectrum of resources, including research papers, lecture slides, blog posts, research talks, tutorial videos, and other materials shared by the research community. Part of the course material was created by LLM itself.

Cambrian Explosion

The Cambrian explosion was a period of rapid evolutionary change approximately 541 million years ago, resulting in the appearance of most major animal phyla.



Cambrian Explosion vs. Deep Learning

Feature	Cambrian Explosion (Biology)	Deep Learning (AI)
Preconditions	Oxygen buildup and genetic complexity primed evolution.	Decades of AI research, data growth, and GPUs set the stage.
Catalyst	Oxygen surge and vision sparked complex life.	AlexNet (2012) + GPUs proved deep nets could dominate.
Rapid Diversification	Major animal phyla emerged in a short time.	CNNs, RNNs, GANs, Transformers rapidly appeared.
Niche Filling	Organisms filled ecological roles across oceans.	Models specialized for vision, language, robotics, medicine.
Rising Complexity	Shells, limbs, and nervous systems evolved.	Models scaled to billions of parameters and emergent skills.
Unpredictable Outcomes	Evolution produced unforeseen lifeforms.	AI evolves in unexpected, creative directions.

Pre-LLM Era (2017-2020)

- During this period, deep learning expanded rapidly after breakthroughs such as ResNet, Inception, and BERT.
- Researchers explored a wide variety of architectures — CNNs, RNNs, GANs, and attention-based hybrids — creating enormous diversity in operators and dataflows.
- Meanwhile, the hardware landscape fragmented. NVIDIA GPUs, Google TPUs, Intel Movidius, Huawei Ascend, Graphcore IPUs, and other custom ASICs each introduced proprietary instruction sets, memory hierarchies, and parallelization models, making portability extremely difficult.
- Developers of frameworks such as TensorFlow and PyTorch struggled to deploy models efficiently across these heterogeneous systems, raising a key question:
- **“AI everywhere”: How can we write a model once and run it efficiently everywhere?**

The LLM Era (After 2020)

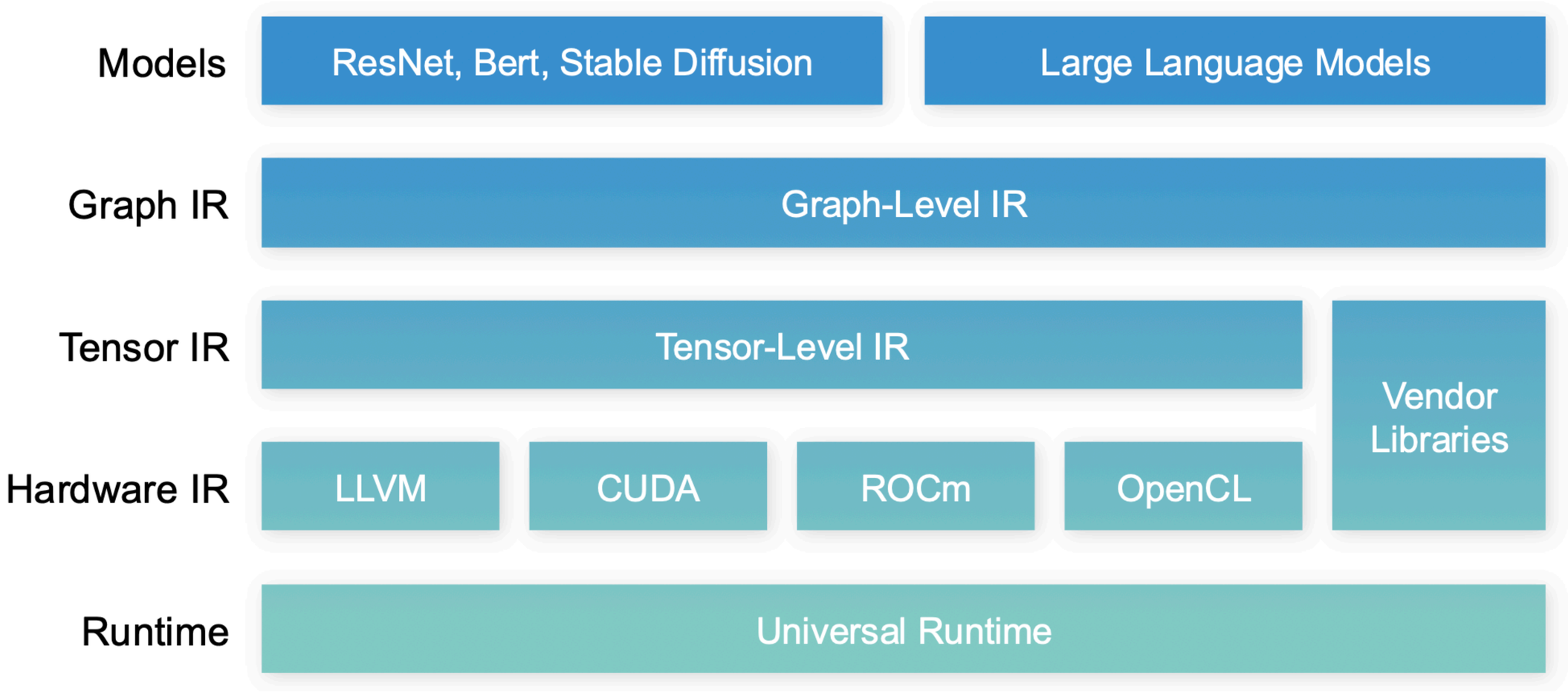
Attention indeed is all you need — so far.

- After 2020, the landscape shifted with the rise of Large Language Models (LLMs) such as GPT-3, PaLM, and Claude. Model architectures converged around the Transformer, replacing the earlier diversity of CNNs and RNNs.

- At the same time, NVIDIA's ecosystem—CUDA, cuDNN, Tensor Cores, NVLink, and A100/H100 GPUs—became the industry standard. Since LLM developers prioritized accuracy and scalability, cross-hardware portability was no longer a first-order concern.

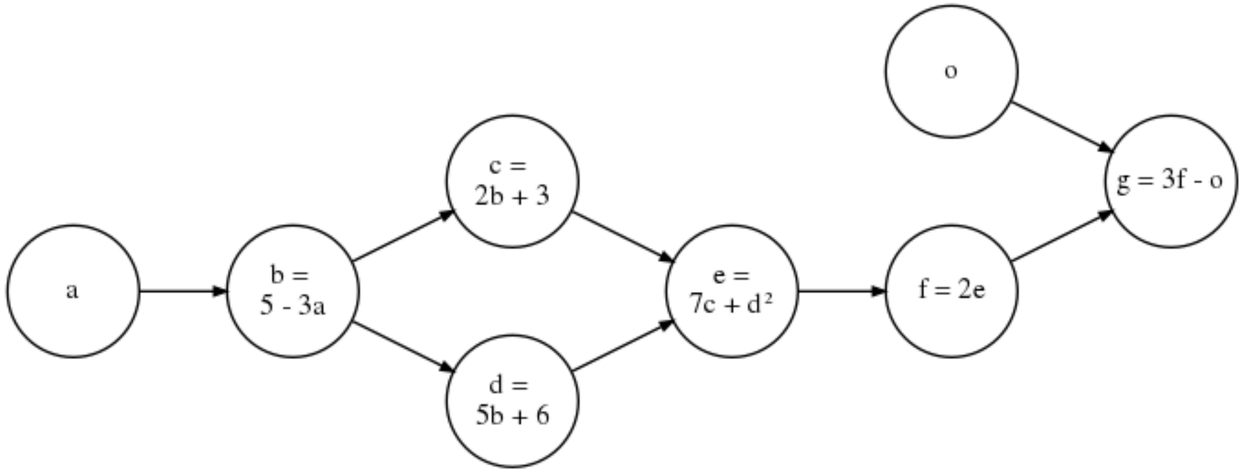
- As a result, the focus of deep-learning compilers moved from hardware generality to maximizing performance on a single dominant platform. In this context, Triton—an open-source project from OpenAI—emerged as a new focal point. Often described as "CUDA for Python," Triton enables developers to write efficient custom GPU kernels with automatic memory and vectorization management, serving as a specialized compiler tailored for LLM workloads.

Machine Learning Compilation Abstractions



Background

As system researchers, we may first ask what's the pattern



Understand the workload: the computational patterns that dominate performance.

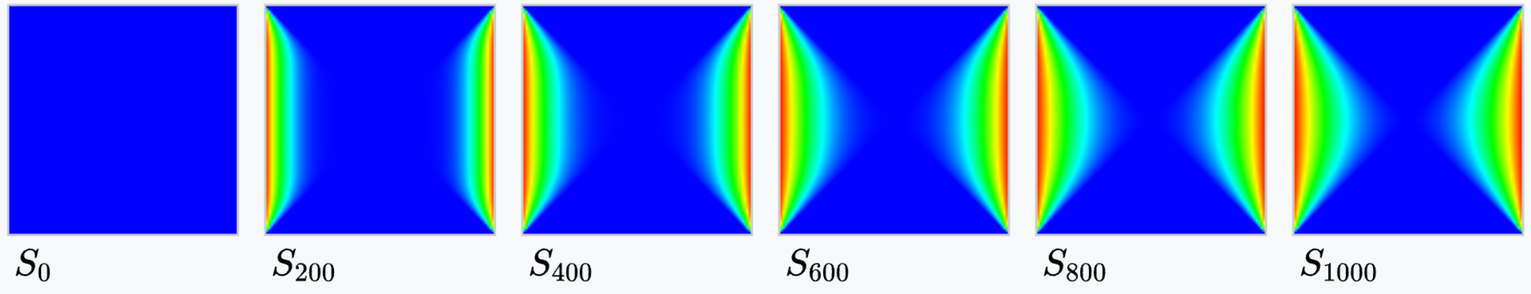
Background

As system researchers, we may first ask what's the pattern

- DNN compute is composed of operators/kernels

```
for t in range(T):
    apply_stencil(u_old, u_new)
    swap(u_old, u_new)
```

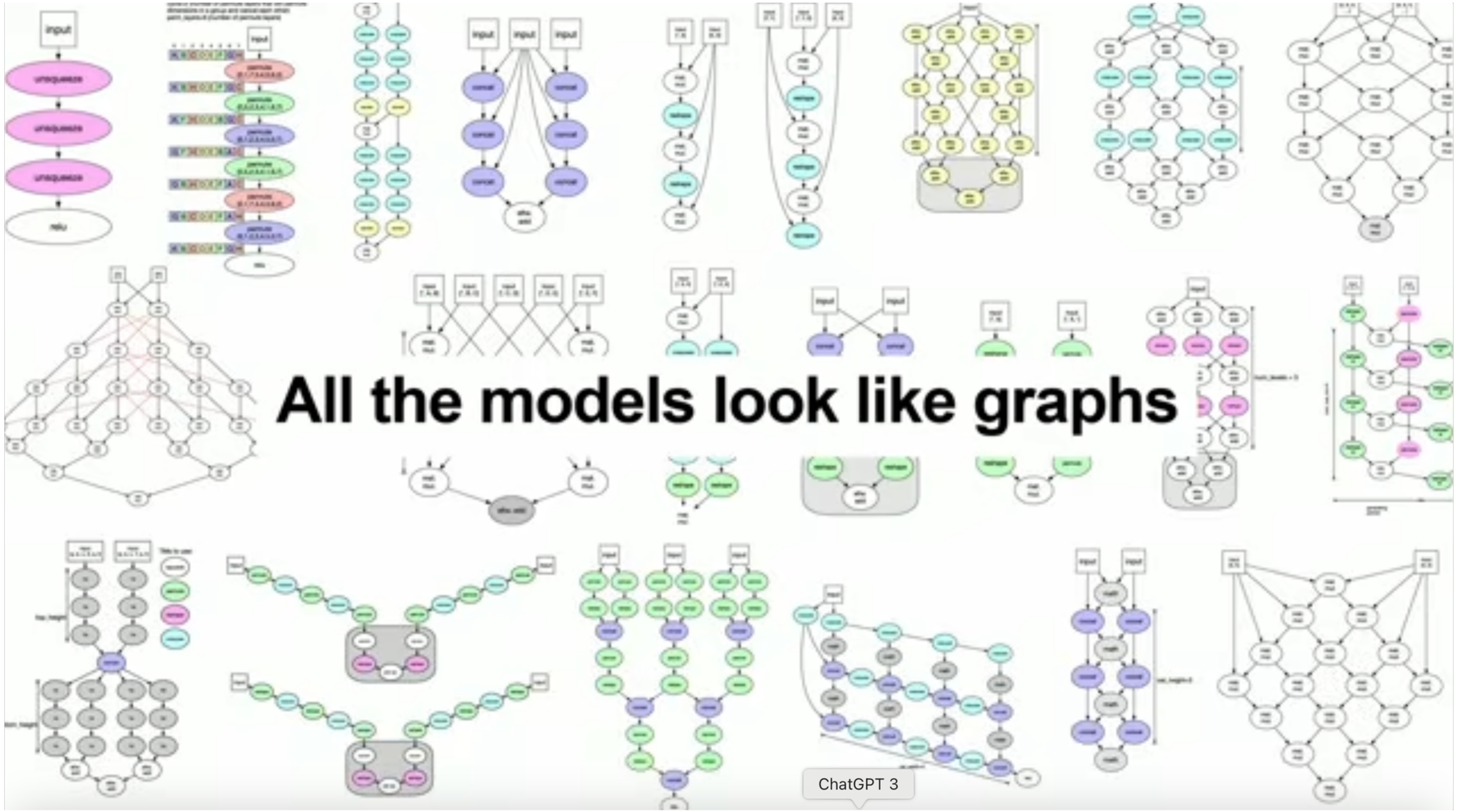
```
for i in range(1, N-1):
    for j in range(1, N-1):
        u_new[i][j] = 0.25 * (
            u_old[i-1][j] + u_old[i+1][j] + u_old[i][j-1] + u_old[i][j+1]
        )
```



2D Jacobi Iteration on a 100^2 Array

Key characteristics: iterative, data-parallel, and memory-bandwidth-sensitive

What is new in deep learning is the scale, diversity of operators, and hardware specialization


All the models look like graphs
ChatGPT 3

Operators & Kernels

An **operator** is a high-level, mathematically defined function that takes one or more tensors (multi-dimensional arrays of data) as input and produces one or more tensors as output. They represent the building blocks of a neural network or a larger computation.

- Examples in DNNs: Convolution (Conv2D)，Matrix Multiplication (MatMul)，ReLU，Pooling (MaxPool).

A **kernel** is the low-level, highly optimized implementation of an operator for a specific hardware architecture (like a GPU, CPU, or a specialized DNN accelerator).

- Purpose: To maximize computational efficiency by running thousands of identical tasks simultaneously.

Relationship to Operator: An operator like "Convolution" might be implemented by several different kernels depending on the input size, memory layout, and target hardware (e.g., one kernel for small inputs, another for large inputs on a GPU).

Background

As system researchers, we may first ask what's the pattern

- DNN compute is composed of operators/kernels
  - balance between parallelism, locality and compute redundancy

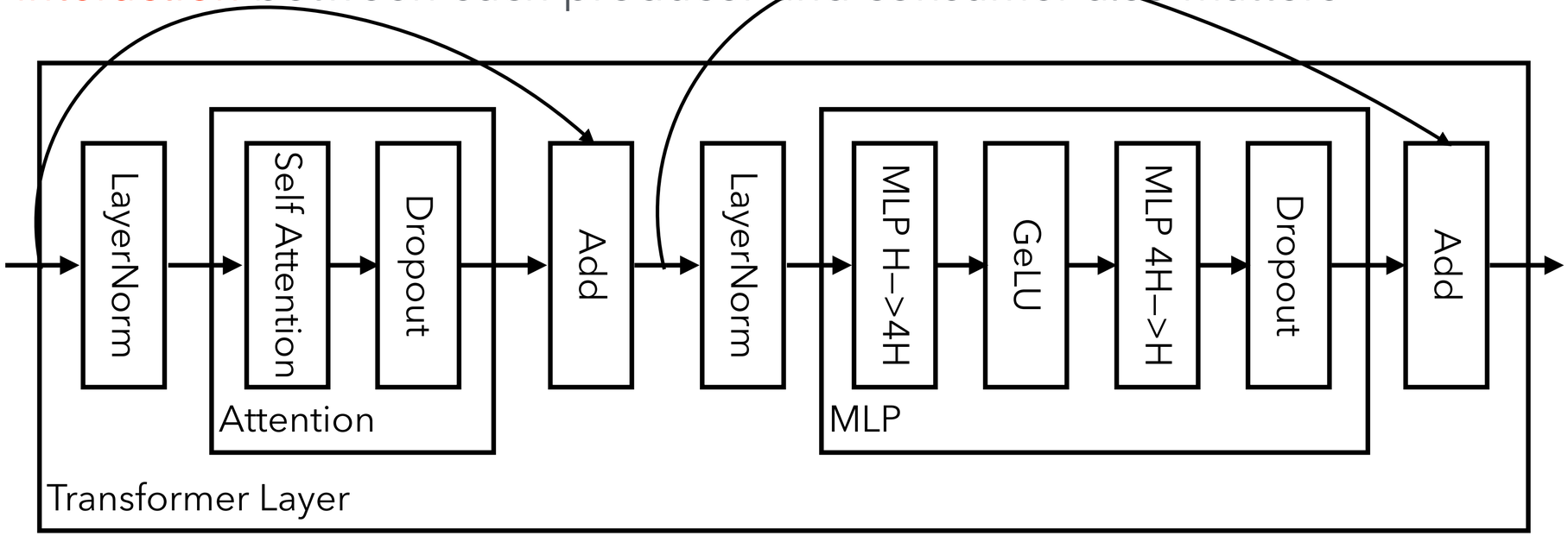
```
for t in range(T):
    apply_stencil(u_old, u_new)
    swap(u_old, u_new)
```

```
for i in range(1, N-1):
    for j in range(1, N-1):
        u_new[i][j] = 0.25 * (
            u_old[i-1][j] + u_old[i+1][j] + u_old[i][j-1] + u_old[i][j+1]
        )
```

Background

As system researchers, we may first ask what's the pattern

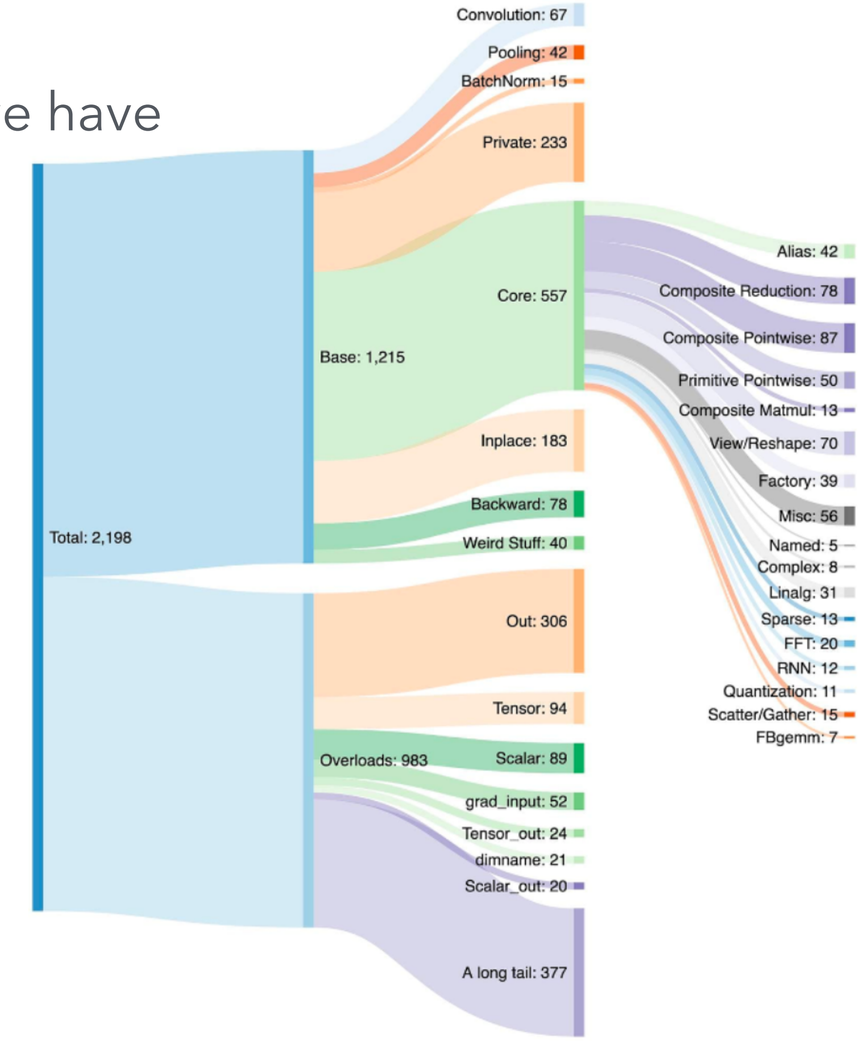
- DNN compute is ~~composed~~ graph of different operators/kernels
  - balance between parallelism, locality and compute redundancy
  - interaction between each producer and consumer also matters



Motivation

Actually, we find traditional ways do not work well

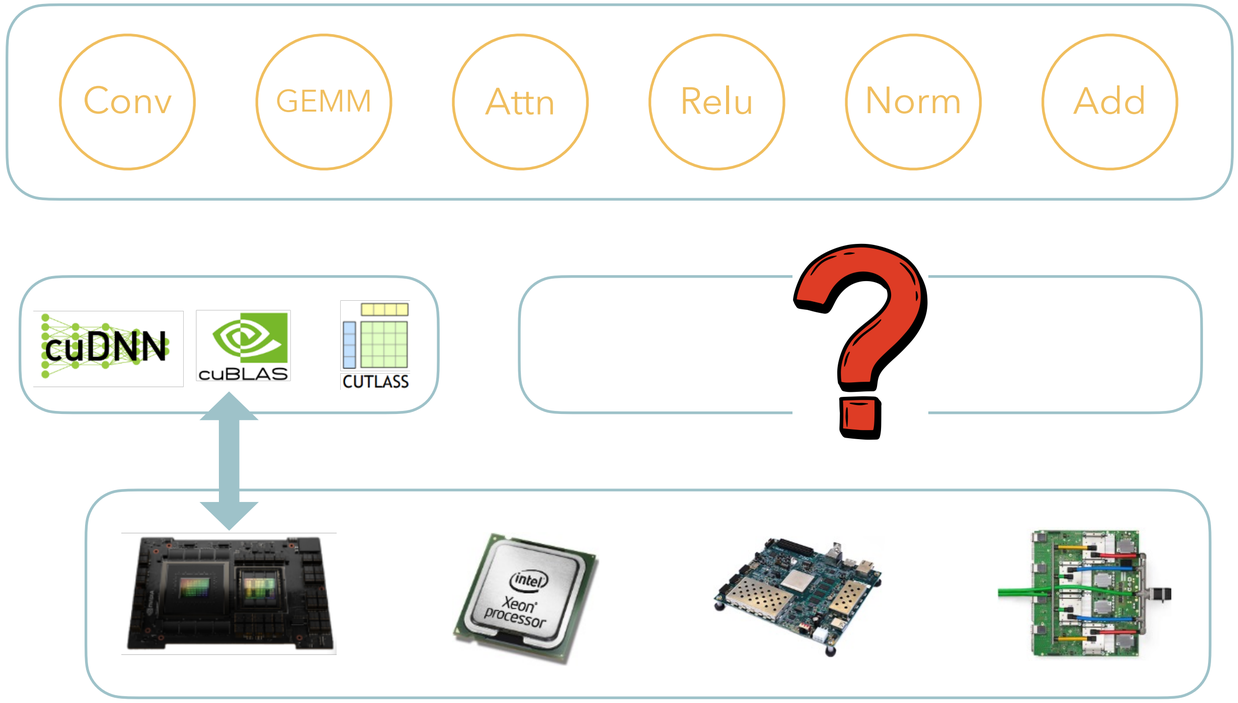
- Deploying any model anywhere is labor-intensive, since we have
  - hundreds of operators



Motivation

Actually, we find traditional ways do not work well

- Deploying any model anywhere is labor-intensive, since we have
  - hundreds of operators
  - tens of hardware backends



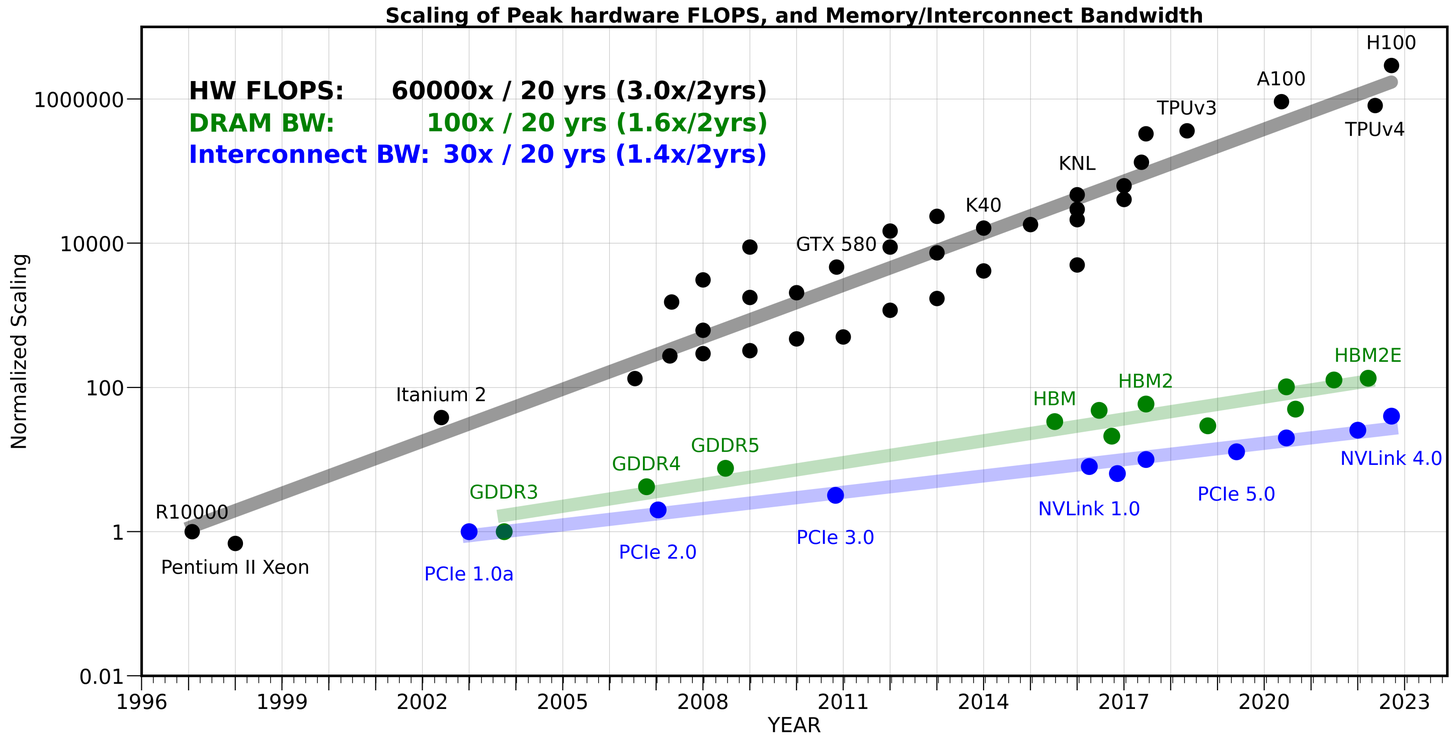
The “AI everywhere” dream is costly

- Implement hundreds of operators,
- Tune them for their unique memory hierarchy and parallelism model, and
- Continuously maintain and update them as models evolve.

Motivation

Actually, we find traditional ways do not work well

- Evolving neural architectures call for an efficient way to generate kernels with
  - higher execution efficiency
  - less memory access



Deep Learning Compiler

Compute-schedule separation

- Crafting a high-performance kernel from scratch involves too many tricks both in algorithm and architecture
- Take a reduce kernel as an example

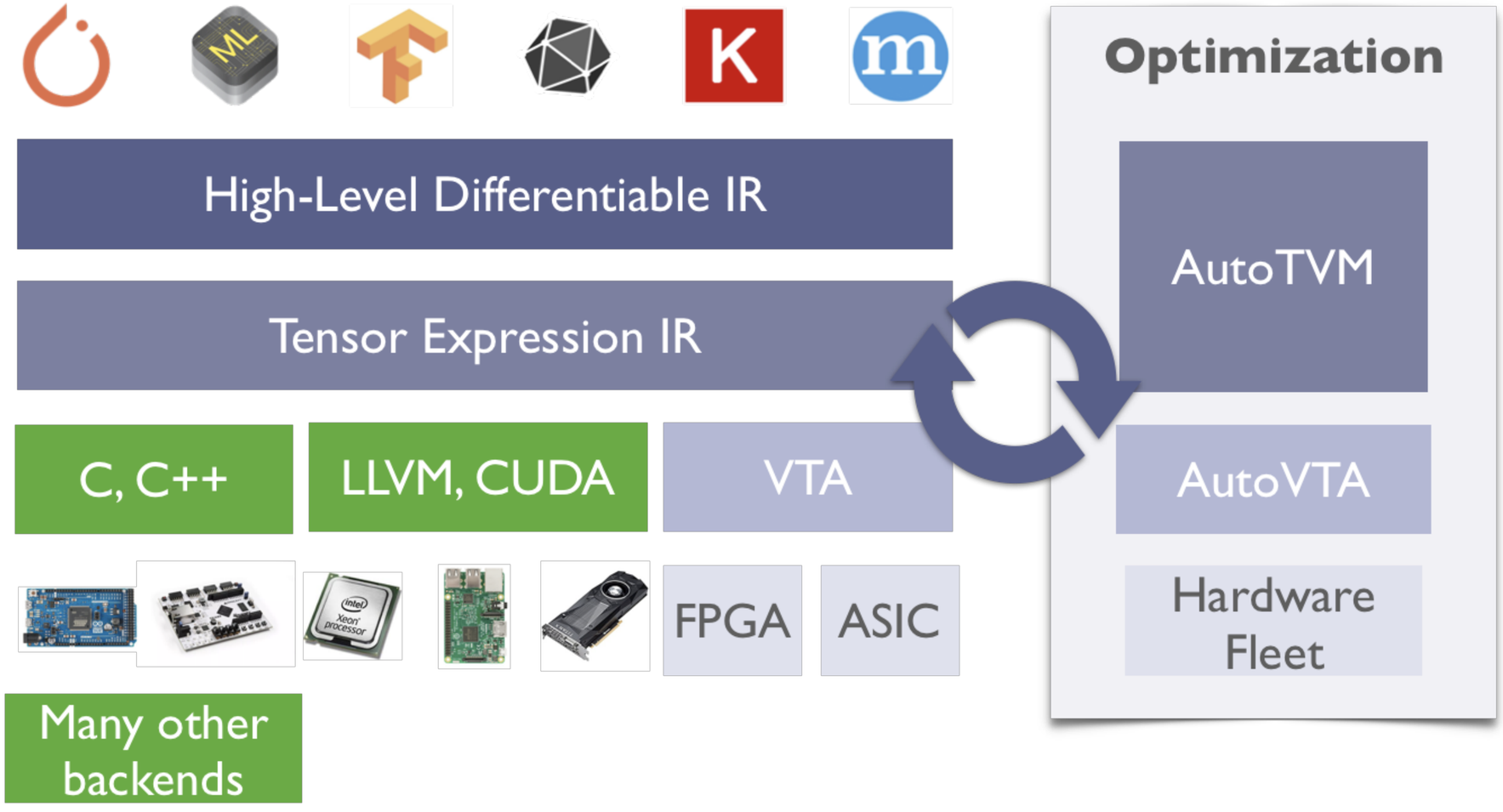
https://zhuanlan.zhihu.com/p/426978026

TVM

TVM (Tensor Virtual Machine) is an open-source, end-to-end deep learning compiler framework that automatically optimizes and generates efficient code for tensor computations on diverse hardware—from CPUs and GPUs to AI accelerators—by combining high-level graph transformations with low-level, machine-learning–driven tensor scheduling.

TVM evolved from an academic prototype to a leading open-source deep learning compiler stack, driving the paradigm shift from framework-specific kernels to compiler-driven, hardware-agnostic optimization.

It represents the transition of AI systems from hand-tuned engineering toward automated performance learning — uniting machine learning, systems, and hardware co-design into a single, extensible ecosystem.


ML
K
m
High-Level Differentiable IR
Tensor Expression IR
C, C++
LLVM, CUDA
VTA
FPGA
ASIC
Many other backends
Optimization
AutoTVM
AutoVTA
Hardware Fleet

TVM

1. Origins (2016–2017): Building a Compiler for Deep Learning

TVM began as an open-source research project led by Tianqi Chen and collaborators at Carnegie Mellon University (CMU) and the University of Washington.

At the time, deep learning frameworks like TensorFlow, Caffe, and PyTorch relied on manually optimized kernels written for specific hardware (e.g., cuDNN for NVIDIA GPUs). This made porting models to new hardware architectures—like ARM CPUs, FPGAs, or emerging AI accelerators—extremely difficult.

TVM's core idea was to create a unified, compiler-based abstraction layer that could automatically optimize and generate high-performance code for diverse hardware backends, starting from high-level deep learning models.

The first public release of Apache TVM appeared in 2017, accompanied by the paper "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning" (OSDI 2018), which became foundational in AI systems research.

TVM

2. Formalization and Core Contributions (2018–2020): Relay IR and AutoTVM

TVM introduced several key innovations that shaped the modern deep learning compilation landscape:

- Relay IR: A high-level, functional intermediate representation (IR) designed for deep learning graphs. Relay replaced the earlier NNVM IR and enabled high-level graph optimizations such as operator fusion, constant folding, and layout transformations.

- AutoTVM: A machine-learning–based auto-tuning framework that automatically searches the space of low-level schedules (loop tiling, unrolling, vectorization, etc.) to find near-optimal performance configurations for each operator on a given hardware platform.

These innovations established TVM as a general-purpose deep learning compiler stack, bridging the gap between front-end frameworks (TensorFlow, PyTorch, MXNet) and hardware backends (CPU, GPU, FPGA, mobile SoCs).

TVM

3. Ecosystem Growth and Industrial Adoption (2020–2022): From Research to Production

As TVM matured, it was incubated by the Apache Software Foundation (ASF) and became Apache TVM.

During this phase, TVM gained wide industry adoption by major AI companies and hardware vendors:

- Amazon Web Services (AWS) used TVM in its Neuron SDK for Inferentia chips.
- OctoML, a startup founded by TVM's creators, automated model optimization based on TVM.
- ARM, Qualcomm, and NVIDIA contributed backend optimizations and integration layers.
- The community introduced VTA (Versatile Tensor Accelerator) — a fully open-source hardware design integrated with TVM, demonstrating end-to-end hardware–software co-design.

TVM's success inspired a generation of compiler-based ML systems, influencing projects like XLA (TensorFlow), Glow (Meta), and MLIR (LLVM ecosystem).

TVM

4. The Next Generation (2022–Present): MetaSchedule and Unified AI Compilation

Modern TVM efforts focus on automating and unifying every stage of the AI compilation pipeline:

- MetaSchedule (2022): Replaces AutoTVM with a more general, learning-based search framework that can automatically generate, evaluate, and refine scheduling rules for new operators and architectures.

- Relax IR (2023–): Extends Relay to support not only inference but also training workloads, mixed precision, and dynamic shapes—key steps toward end-to-end training-time optimization.

- Unity project (2024–): Aims to merge TVM's compiler, autotuning, and runtime into a single unified infrastructure, integrating meta-learning and symbolic rewriting to make model optimization "self-evolving."

TVM

TVM creates a compiler-driven performance stack that rivals manually optimized libraries across CPUs, GPUs, and specialized accelerators.

1. Graph-level reasoning (operator fusion, memory analysis)
2. Tensor-level scheduling (hardware-aware optimization)
3. Learning-based auto-tuning

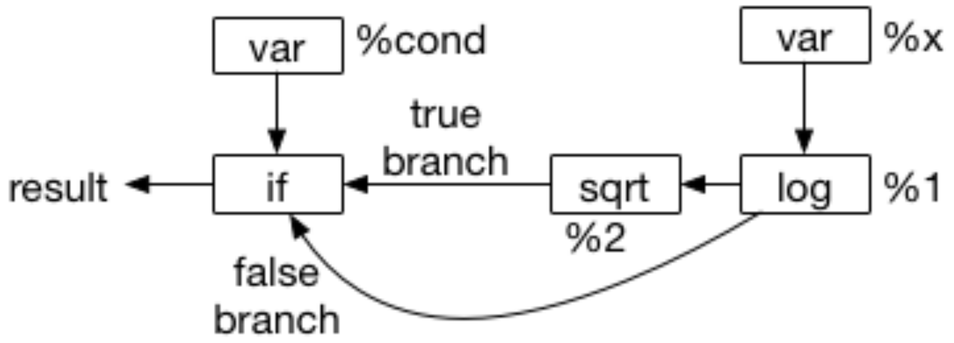
High-level Graph Optimizations

TVM operates on the model as a dataflow graph, rather than individual tensor kernels.

Relay IR is the TVM's unified, functional intermediate representation, it sites between frontend frameworks (like PyTorch, TensorFlow, or ONNX) and backend code generation.

- A Relay graph represents the neural network as a directed acyclic graph (DAG).
- Each node is an operator (e.g., Conv2D, ReLU, BatchNorm, etc.).
- Each edge represents the flow of tensors (outputs feeding into inputs).

```
fn (%cond, %x) {
  %1 = log(%x)
  if (%cond) {
    %2 = sqrt(%1)
    %2
  } else {
    %1
  }
}
```



High-level Graph Optimizations

Relay performs graph-level transformations that are hardware-agnostic, before scheduling and code generation.

(a) Operator Fusion: Minimize memory traffic and kernel invocation overhead.

(b) Layout Transformation: Adapt data format (e.g., NCHW ↔ NHWC) to hardware-specific primitives.

(c) Memory Planning and Reuse: Reduce peak memory usage and allocations.

(d) Kernel Transformation: Convert complex operators into equivalent but hardware-friendly forms.

Operation Fusion

Goal: Minimize memory traffic and kernel invocation overhead.

How: TVM performs pattern-matching over the Relay graph to detect fusible sequences

Example: Conv2D → BiasAdd → ReLU → BatchNorm.

These are merged into a single fused operator, which will later be compiled into a single kernel.

During fusion, the intermediate tensor (e.g., the Conv output) is kept in cache or registers, avoiding write-backs to DRAM.

Benefit: Up to 2–4× speedups on memory-bound workloads (common in CNNs).

Operation Fusion: Why

Each operator (e.g., Conv2D, ReLU) might otherwise:

- Launch a separate kernel on the device.
- Write its intermediate result to global memory (DRAM).
- Then the next operator reads it back, processes, and writes again.

This back-and-forth memory movement dominates runtime cost — especially on GPUs and accelerators — more than the actual arithmetic!

Fusion reduces this by combining operators into one kernel, so data flows directly between computations in registers or cache, not memory.

Operation Fusion: Criteria

Two (or more) operators can be fused if:

1. Their data dependencies are one-directional (i.e., each operator's output is consumed only by the next one).
2. Their computation is elementwise-compatible or can be mathematically composed into a single loop nest.
3. Their fusion does not change semantics (i.e., same numerical result).
4. Their memory access patterns align, so the same tensor can stay in registers or caches without materializing to memory.

Operation Type	Typical Hardware Concern	Benefit of Fusion
Conv2D	High FLOPs, heavy memory reads	Combine post-processing to reuse cache
BiasAdd	Low FLOPs, memory-bound	Perform inline with Conv
ReLU	Memory-bound, elementwise	Free to fuse, almost zero overhead
BatchNorm (inference)	Elementwise	Convert to scale + shift, fold in

(1) Conv2D

Mathematically:

$$Y[i, j] = \sum_k X[i, k] * W[j, k]$$

- Takes an input tensor X and weights W.
- Produces an output tensor Y (feature map).
- It's a heavy operation (most of the FLOPs).
- Each output element is independent once convolution is done.

(2) BiasAdd

$$Z[i, j] = Y[i, j] + b[j]$$

- Adds a *bias term* to each output channel.
- This is **elementwise** across the same tensor shape.
- Depends directly on the convolution output.

➡️ **Why fusable:**
Bias addition can be performed *inside the same loop* that computes the convolution.

```
Y = conv2d(X, W)
Z = Y + b
```

```
Z = sum(X * W) + b
```

(3) ReLU

$$A[i, j] = \max(0, Z[i, j])$$

- Applies elementwise activation.
- Independent per element (no cross-channel or spatial dependency).

➡️ **Why fusable:**
ReLU simply transforms the output value after bias addition.
We can fuse it trivially:

```
Z = sum(X * W) + b
```

```
Z = max(sum(X * W) + b, 0)
```

(4) BatchNorm (in inference mode)

$$O[i,j] = \frac{A[i,j] - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} \cdot \gamma_j + \beta_j$$

- Normalizes each channel using learned statistics (μ, σ², γ, β).
- For inference (not training), these are fixed constants.
- Computation is elementwise — each output element depends only on one input element (same position, same channel).

$$O[i,j] = \text{ReLU}\left(\left(\sum_k X[i,k] \cdot W[j,k] + b[j]\right) \cdot s_j + t_j\right)$$

➡️ **Why fusable:**

For inference, batchnorm is just a **per-channel affine transform**:

$$O[i,j] = A[i,j] \cdot s_j + t_j$$

where $s_j = \frac{\gamma_j}{\sqrt{\sigma_j^2+\epsilon}}$ and $t_j = \beta_j - \mu_j s_j$.

```
Z = sum(X * W) + b
```

```
Z = max(sum(X * W) + b, 0)
```

```
Y1 = conv2d(X, W)
Y2 = add(Y1, bias)
Y3 = relu(Y2)
Y4 = batch_norm(Y3)
```

```
for (i, j, k) {
    float val = 0;
    for (r) val += X[i, r] * W[r, j];
    val += bias[j];
    val = max(val, 0);                  // ReLU
    val = (val - mean[j]) / var[j];     // BatchNorm
    Y[i, j] = val;
}
```

Low-Level Tensor Operator Optimizations

This is the “how to compute” layer, implemented in the Tensor Expression (TE) language and later TIR (Tensor IR) in modern TVM. It defines loop structures, tiling, thread mapping, and memory hierarchy usage, the same way a human CUDA programmer would optimize by hand.

Technique	Description	Goal / Hardware Mapping
Tiling / Blocking	Divide large loops (e.g., matrix multiply) into small tiles that fit in cache or shared memory.	Improves data locality; reduces cache misses; enables shared memory reuse on GPUs.
Loop Reordering	Rearrange loop nesting (e.g., `for i, for j` → `for j, for i`).	Ensures contiguous memory access and reduces stride.
Vectorization	Use SIMD instructions (AVX, NEON) to process multiple elements at once.	Exploits hardware vector units for element-wise ops.
Loop Unrolling	Replicate inner loop bodies to reduce branch overhead.	Increases instruction-level parallelism.
Parallelization	Distribute loop iterations across CPU threads or CUDA thread blocks.	Scales computation over multi-core or GPU SMs.
Cache Read/Write (Memory Hierarchy Mapping)	Explicitly place buffers in shared/local memory (`cache_read`, `cache_write`).	Hides memory latency and leverages fast on-chip memory.

Computation & Schedule Separation

- One of the core ideas in deep learning compilers (Halide, TVM, Triton), decouples correctness from performance.
- **Compute definition:** describes what you want to calculate — it's the mathematical logic of an operator or layer, independent of any hardware or optimization decisions.

$$C[i,j] = \sum_k A[i,k] \times B[k,j]$$

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

Computation & Schedule Separation

- **Schedule:** defines how to execute that computation — the order, parallelization, memory layout, and mapping to hardware resources.
  - Split loops into smaller tiles that fit in cache.
  - Reorder loops to improve data locality.
  - Parallelize across GPU threads.
  - Use vectorization (SIMD) instructions.
  - Store temporary results in shared memory to avoid reloading from global memory.

TVM Tensor Expression

```
# Define the compute (what to compute)
C = te.compute((M, N), lambda i, j: te.sum(A[i, k] * B[k, j], axis=k))

# Define the schedule (how to compute)
s = te.create_schedule(C.op)
i, j = s[C].op.axis
io, ii = s[C].split(i, factor=32)
jo, ji = s[C].split(j, factor=32)
s[C].reorder(io, jo, ii, ji)
s[C].parallel(io)
s[C].vectorize(ji)
```

- `te.compute((M, N), ...)`
  Declares a tensor C of shape `(M, N)` *symbolically*. No loops here yet—this is the **mathematical spec**.
- `lambda i, j: ...`
  The element formula for `C[i, j]`.
- `te.sum(A[i, k] * B[k, j], axis=k)`
  Says: "for each `(i, j)`, sum over k the product `A[i, k] * B[k, j]`". This is exactly matrix multiplication:

$$C[i,j] = \sum_{k=0}^{K-1} A[i,k] \cdot B[k,j].$$

  The `axis=k` tells TVM this is a **reduction** over the k dimension.

TVM Tensor Expression

```
# Define the schedule (how to compute)
s = te.create_schedule(C.op)
```

- Creates a **schedule object** `s` tied to the computation that produces `C`. `C.op` refers to the TVM "operation" node that builds `C`.

```
i, j = s[C].op.axis
```

- Extracts the **spatial axes** of `C` (the ones that index its shape).
  Here, `i` iterates rows `0..M−1`, `j` iterates cols `0..N−1`.
  The reduction axis `k` is separate: `k = s[C].op.reduce_axis[0]` if/when you need it.

TVM Tensor Expression

```
io, ii = s[C].split(i, factor=32)
jo, ji = s[C].split(j, factor=32)
```

- **Tiling** (a.k.a. loop splitting) on both `i` and `j` with tile size 32:
  - `i` becomes two loops: an **outer tile index** `io` and an **inner within-tile** index `ii`.
  - `j` becomes `jo` and `ji`.
- Intuition: work on 32×32 **blocks (tiles)** of `C` at a time to improve cache/locality and to match vector/SIMD/GPU warp granularities.

TVM Tensor Expression

```
s[C].reorder(io, jo, ii, ji)
```

```
for io in ...
  for jo in ...
    for ii in ...
      for ji in ...
        ...
```

- Why reorder? Loop order controls:
  - **Memory locality** (how you walk A/B/C),
  - **Parallelization strategy** (which loops to run concurrently),
  - **Vectorization feasibility** (inner loop must be contiguous in memory).

TVM Tensor Expression

```
s[C].parallel(io)
```

- Marks the `io` loop (the outer i-tile) as **parallel**.
  On CPU backends, TVM may spawn threads across `io` tiles (e.g., OpenMP).
  On GPU backends, you'd often bind loops to block/grid—(here we're keeping it generic).

```
s[C].vectorize(ji)
```

- Vectorizes the **innermost j-within-tile** loop.
  This asks the compiler to emit SIMD instructions (e.g., AVX for CPU) or use wide loads/stores when legal.
  Precondition: the memory accessed along `ji` should be contiguous and aligned. Matrix `C` is typically row-major in TVM (last axis contiguous), so vectorizing across `j` is natural.

TVM Tensor Expression

```
# Pseudocode of the scheduled loop structure
for io in parallel range(ceil_div(M, 32)):      # parallel tiles along rows
    for jo in range(ceil_div(N, 32)):          # tiles along cols
        for ii in range(32):                    # inside a row tile
            # ji will be vectorized (SIMD)
            for ji in vectorized range(32):     # inside a col tile
                acc = 0
                for kk in range(K):             # reduction (not yet tiled)
                    acc += A[io*32 + ii, kk] * B[kk, jo*32 + ji]
                C[io*32 + ii, jo*32 + ji] = acc
```

Computation & Schedule Separation

- Tiling (split) improves locality: you reuse A's row slice and B's column slice for a 32×32 block of C before moving on, reducing cache misses / global memory traffic.
- Reordering ensures we iterate over tiles first, then within-tile indices—this groups data accesses nicely.
- Parallel(io) exposes coarse-grain parallel work across independent tiles.
- Vectorize(ji) exploits SIMD for the most contiguous memory direction (the inner-most loop), boosting per-core throughput.

Concept	Description	Example
Compute Definition	What to compute — pure mathematical description	`C[i, j] = sum(A[i, k] * B[k, j])`
Schedule	How to compute — loop ordering, memory usage, threading	Tile size, parallelization, caching
Separation Benefit	Allows auto-tuning, portability, optimization	TVM, Halide, TensorIR

Automated Optimization — AutoTVM & Meta Schedule

Manually tuning all schedule knobs is infeasible — the search space is exponential. TVM automates this process with AutoTVM (rule-based + ML) and the newer Meta Schedule (reinforcement learning + evolutionary search).

Concept	Description	Example
Compute Definition	What to compute — pure mathematical description	`C[i, j] = sum(A[i, k] * B[k, j])`
Schedule	How to compute — loop ordering, memory usage, threading	Tile size, parallelization, caching
Separation Benefit	Allows auto-tuning, portability, optimization	TVM, Halide, TensorIR

Automated Optimization — AutoTVM & Meta Schedule

(a) Cost Modeling

- Build a **performance predictor** (ML model) trained on measured runtimes.
- Features include tile sizes, loop unroll factors, vector widths, memory access patterns.
- The model estimates latency before running the kernel.

Automated Optimization — AutoTVM & Meta Schedule

(b) Search Strategy

- Use techniques like simulated annealing, Bayesian optimization, or evolutionary algorithms.
- The search space is explored adaptively, guided by the cost model.

Automated Optimization — AutoTVM & Meta Schedule

(c) Measurement and Feedback

- Candidate schedules are compiled, run on the *real device*, and measured.
- Results feed back to improve the cost model iteratively.
- After convergence, TVM exports the **best performing schedule** as a “tuning log”.

This process is called **auto-tuning**, producing results comparable to vendor libraries like cuDNN or MKL.

Automated Optimization — AutoTVM & Meta Schedule

Meta Schedule doesn't need templates because it operates directly on the program IR (TensorIR) and systematically applies transformation rules to generate optimization candidates—eliminating the need for human-defined templates that AutoTVM depended on.

- **Less human effort:** No need to handcraft templates for each operator or hardware target.
- **More general:** Works on *any TensorIR-defined operator*, including user-defined ops.
- **Faster innovation:** When new hardware appears, Meta Schedule can adapt automatically via transfer learning or re-tuning.
- **Fully end-to-end:** Optimizes entire models without relying on manually prepared building blocks.

System	Optimization Search Space	Who Defines It	Example
AutoTVM	Parameterized template	Human	"Try tile size (8, 16, 32) and unroll factor (2, 4)"
Meta Schedule	IR transformation space	Compiler	"Split, fuse, reorder, parallelize automatically"

TVM Optimization Hierarchy

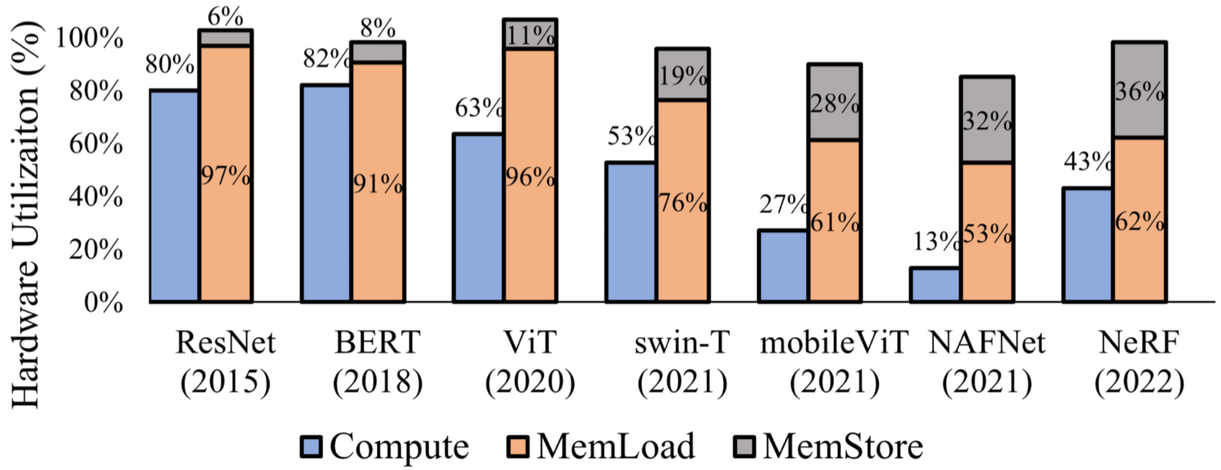
🧱 Summary: TVM Optimization Hierarchy

Layer	Focus	Techniques	Automated by
High-Level (Relay)	Whole-model graph simplification	Operator fusion, layout transform, memory reuse	Graph optimizer
Mid-Level (TE/TIR)	Per-operator scheduling	Tiling, unrolling, parallelization, vectorization	AutoTVM / Meta Schedule
Low-Level (Runtime)	Hardware binding	Target-specific codegen (LLVM, CUDA, Metal, Hexagon)	Codegen & auto-tuner

Deep Learning Compiler

Tile, tile, it's the tile

- The DNN compute is more and more memory-bound
  - Backbone operators like GEMM and Conv are highly optimized
  - For chips, the gap between compute power and memory bandwidth is widening

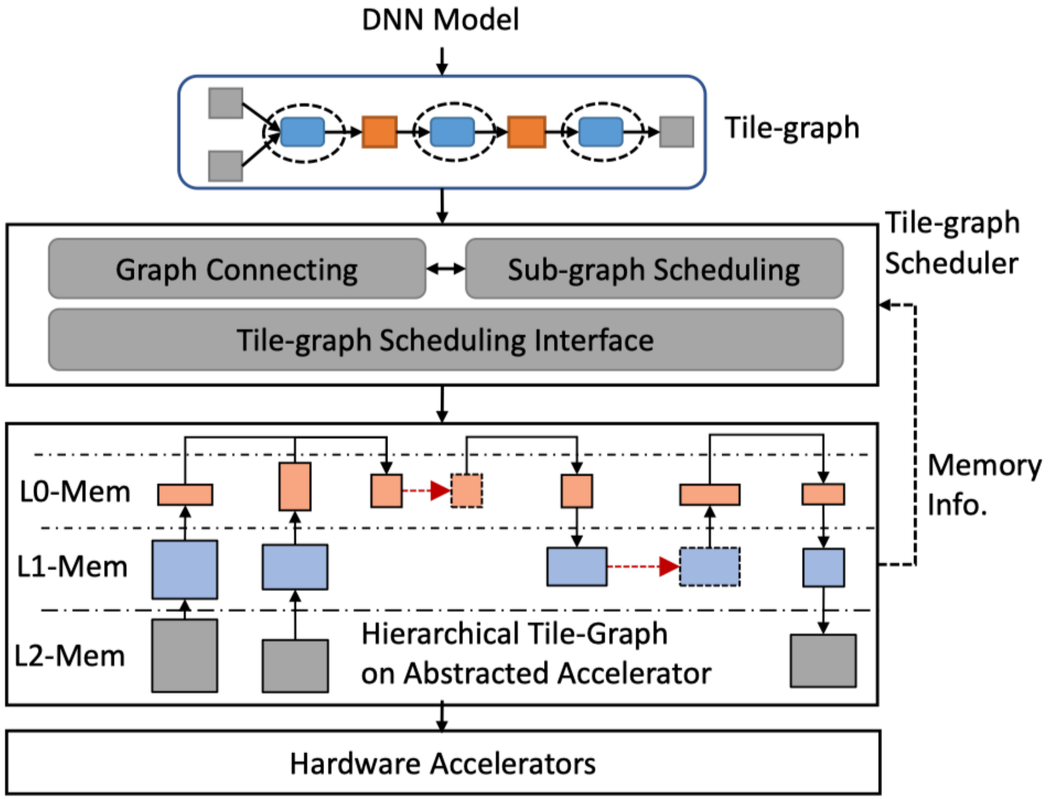


Deep Learning Compiler

Tile, tile, it's the tile

- Each operator enjoys a different pattern of memory access and parallelism due to different compute semantics
- Propagate the tile config through dependence

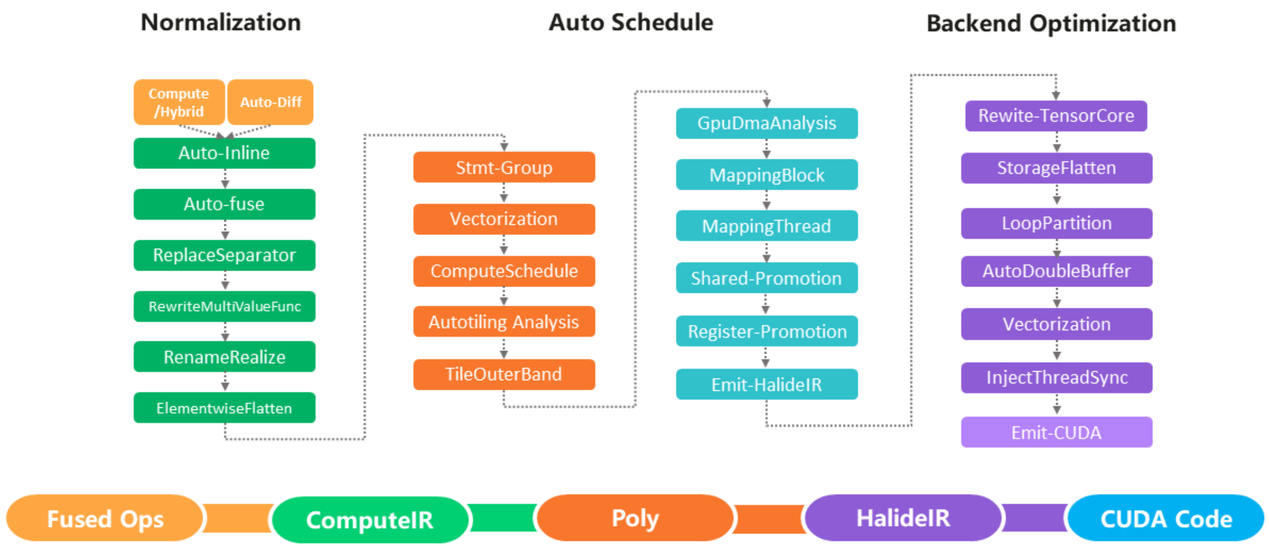
With Tile Abstraction
Compute multiple ops in-place per tile
Most intermediate data stays in on-chip memory
Compute units always busy — **compute-bound**
Minimal read/write — **less I/O, more FLOPs**



Deep Learning Compiler

I hate/fear Deep Learning Compiler

- The DLC is just a magic black box, and it's OK when it runs well, however when bug shows up, you must debug the compilation pipeline instead of the code



Deep Learning Compiler

The golden days have gone

- We only care about Transformer, Attention+FFN
- We believe Transformer is all we need
- Huge Capex almost goes to NVIDIA

LLM Compiler

Triton

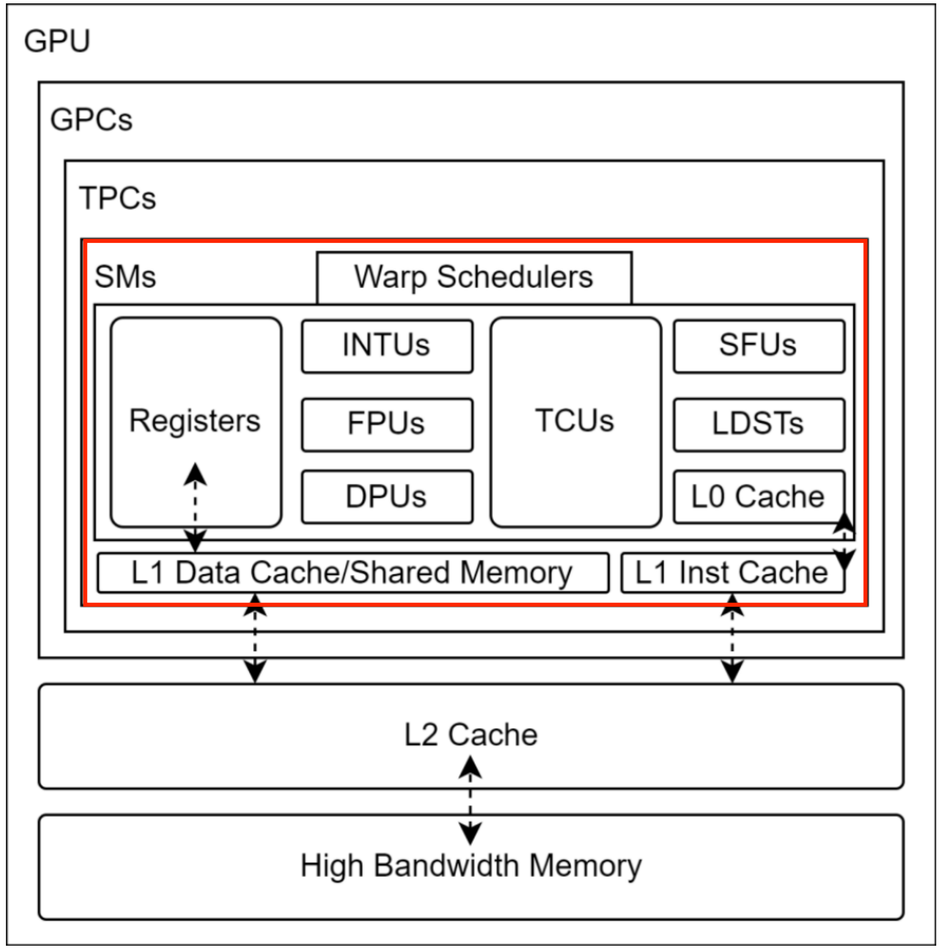
- Triton **Python + CUDA-level performance.**
  - to provide an open-source environment to write fast code at higher productivity than CUDA, but also with higher flexibility than other existing DSLs

LLM Compiler

Triton

- With Triton, you only need to know that a program is divided into multiple blocks

	CUDA	Triton
Memory	Global/Shared/Register	Automatic
Parallelism	Blocks/Warps/Threads	Mostly Blocks
Tensor Core	Manual	Automatic
Vectorization	.8/.16/.32/.64/.128	Automatic



LLM Compiler

Triton

- @triton.jit: python decorator

1. Analyzes the Python function decorated with @triton.jit
2. Generates LLVM / PTX code based the pass arguments and constants
3. Caches the compiled binary
4. Launches it immediately on the GPU

```python
import triton.language as tl
import triton

@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N):
  # same as torch.arange
  offsets = tl.arange(0, 1024)
  # create 1024 pointers to X, Y, Z
  x_ptrs = x_ptr + offsets
  y_ptrs = y_ptr + offsets
  z_ptrs = z_ptr + offsets
  # load 1024 elements of X, Y, Z
  x = tl.load(x_ptrs)
  y = tl.load(y_ptrs)
  # do computations
  z = x + y
  # write-back 1024 elements of X, Y, Z
  tl.store(z_ptrs, z)

N = 1024
x = torch.randn(N, device='cuda')
y = torch.randn(N, device='cuda')
z = torch.randn(N, device='cuda')
grid = (1, )
_add[grid](z, x, y, N)
```

Dynamic specialization: adapt to tensor sizes, hardware, and constants (BLOCK_SIZE, etc.)

Python integration: write GPU kernels directly in Python syntax.

Caching: after the first run, reuses the compiled binary (fast subsequent launches).

Optimization: uses LLVM/MLIR/ptxas backend optimizations based on runtime parameters.

LLM Compiler

Triton

- @triton.jit: python decorator

Program	A single instance of your kernel (like a block)	`blockIdx`
Thread	A scalar or vector of elements inside a program	`threadIdx`
Grid	A collection of programs	Launch configuration
Mask (predicate)	Condition that controls valid elements	`if (idx < N)`
@triton.jit	Marks function for runtime compilation	`__global__`
tl.load / tl.store	Load/store with optional mask	`ld.global` / `st.global`
tl.arange	Vector of thread offsets	`threadIdx.x` sequence

```python
import triton.language as tl
import triton

@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N):
  # same as torch.arange
  offsets = tl.arange(0, 1024)
  offsets += tl.program_id(0)*1024
  # create pointers to X, Y, Z
  x_ptrs = x_ptr + offsets
  y_ptrs = y_ptr + offsets
  z_ptrs = z_ptr + offsets
  # load elements of X, Y, Z
  x = tl.load(x_ptrs, mask=offset<N)
  y = tl.load(y_ptrs, mask=offset<N)
  # do computations
  z = x + y
  # write-back elements of X, Y, Z
  tl.store(z_ptrs, z, mask=offset<N)

N = 192311
x = torch.randn(N, device='cuda')
y = torch.randn(N, device='cuda')
z = torch.randn(N, device='cuda')
grid = (triton.cdiv(N, 1024), )
_add[grid](z, x, y, N)
```

LLM Compiler

Triton

- @triton.jit: python decorator
- program_id(): get the block id
- musk: predicate to load the data
- @triton.autotune: instantiate kernels using configs

```
import triton.language as tl
import triton

@triton.autotune(configs=
  [triton.Config('TILE': 128),
  triton.Config('TILE': 256)]
@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N, TILE: tl.constexpr):
  # same as torch.arange
  offsets = tl.arange(0, TILE)
  offsets += tl.program_id(0)*TILE
  # create pointers to X, Y, Z
  x_ptrs = x_ptr + offsets
  y_ptrs = y_ptr + offsets
  z_ptrs = z_ptr + offsets
  # load elements of X, Y, Z
  x = tl.load(x_ptrs, mask=offset<N)
  y = tl.load(y_ptrs, mask=offset<N)
  # do computations
  z = x + y
  # write-back elements of X, Y, Z
  tl.store(z_ptrs, z, mask=offset<N)

N = 192311
x = torch.randn(N, device='cuda')
y = torch.randn(N, device='cuda')
z = torch.randn(N, device='cuda')
grid = lambda args: (triton.cdiv(N, args["TILE"]), )
_add[grid](z, x, y, N)
```

LLM Compiler

Triton

```
#define FETCH_FLOAT4(pointer) (reinterpret_cast<float4*>(&(pointer))[0])
__global__ void vec4_add(float* a, float* b, float* c)
{
    int idx = (threadIdx.x + blockIdx.x * blockDim.x)*4;
    float4 reg_a = FETCH_FLOAT4(a[idx]);
    float4 reg_b = FETCH_FLOAT4(b[idx]);
    float4 reg_c;
    reg_c.x = reg_a.x + reg_b.x;
    reg_c.y = reg_a.y + reg_b.y;
    reg_c.z = reg_a.z + reg_b.z;
    reg_c.w = reg_a.w + reg_b.w;
    FETCH_FLOAT4(c[idx]) = reg_c;
}
```

```
import triton.language as tl
import triton

@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N):
  # same as torch.arange
  offsets = tl.arange(0, 1024)
  offsets += tl.program_id(0)*1024
  # create pointers to X, Y, Z
  x_ptrs = x_ptr + offsets
  y_ptrs = y_ptr + offsets
  z_ptrs = z_ptr + offsets
  # load elements of X, Y, Z
  x = tl.load(x_ptrs, mask=offset<N)
  y = tl.load(y_ptrs, mask=offset<N)
  # do computations
  z = x + y
  # write-back elements of X, Y, Z
  tl.store(z_ptrs, z, mask=offset<N)

N = 192311
x = torch.randn(N, device='cuda')
y = torch.randn(N, device='cuda')
z = torch.randn(N, device='cuda')
grid = (triton.cdiv(N, 1024), )
_add[grid](z, x, y, N)
```

TVM vs. Triton: Two paths to efficient deep learning kernels

- TVM TE + Schedule → compiler-centric abstraction
- Triton → programmer-centric DSL
- Both seek portable, high-performance GPU kernels
  - **TVM stack:**
    Parser → IR → Fusion → Schedule → Codegen → Runtime
  - **Triton usage:**
    PyTorch 2.0 Inductor → calls Triton kernels as “leaf ops”

Both frameworks aim to bridge deep learning and hardware efficiency, but their philosophies differ—TVM automates scheduling; Triton simplifies manual GPU kernel writing.

Two Mental Models

	TVM TE + Schedule	Triton
Developer defines	*What + How (separate)*	*What = How (integrated)*
Optimizer type	Compiler search	Programmer intuition
Analogy	“Compiler engineer”	“CUDA programmer in Python”

TVM abstracts optimization, while Triton lets developers control threads, memory, and tiling directly.

Compute vs. Schedule vs. Inline Kernel

- TVM
  - compute: pure math (e.g., C = A × B)
  - schedule: loop tiling, binding, memory layout
  - compile → different backends (CPU/GPU/ASIC)
- Triton
  - kernel code = both algorithm + execution
  - program defines block/grid, memory access inline

Targeting different hardware

- TVM:
  - one compute IR → many targets (LLVM, CUDA, OpenCL)
  - only schedule differs
- Triton:
  - GPU-centric (CUDA / ROCm)
  - no CPU backend

- TVM can retarget a model from NVIDIA to ARM CPU by rescheduling.
- Triton is ideal for GPU kernels embedded in PyTorch, not for CPUs or edge NPUs.

Operator Fusion

- TVM:
  - Graph-level automatic fusion (compute_at, fuse)
  - Compiler ensures dependency safety
- Triton:
  - Manual but simple: inline math in same kernel
  - Common for fused MLP + activation, FlashAttention

Auto-Tuning

	TVM AutoTVM / MetaSchedule	Triton @autotune
Search space	large (tile / split / unroll / vector)	small (BLOCK / warps / stages)
Search time	minutes–hours	seconds–minutes
Guidance	learned cost model	direct benchmarking
Output	optimal schedule IR	best kernel config

TVM explores thousands of possibilities per op, whereas Triton tries dozens; the difference reflects their goals—TVM: portability + automation, Triton: fast iteration.

Take Home Message

The “principles” of DLC haven’t aged: start from the workload, and use compute–schedule separation, IR-level transformations, auto-tuning, and tile/block abstractions to move performance back onto the compute units.

- The “techniques” are changing in the LLM era: for fixed, high-value operator families, DSLs like Triton + lightweight auto-tuning often reach peak performance faster; for diverse or new hardware, Meta Schedule remains a key lever for cross-platform performance.

- Looking ahead: enable co-evolution of algorithm engineers and compilers—retain the compiler’s global view and automation while providing hardware-proximal, controllable, and efficient abstractions (e.g., Block/Tile). This is a viable path for repositioning DLC in the LLM era.

Homework

PyTorch Conference 2025

https://events.linuxfoundation.org/pytorch-conference/program/schedule/

Reference

- PyTorch 2.0 Overview
- Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines
- TVM: An Automated End-to-End Optimizing Compiler for Deep Learning
- Ansor: Generating High-Performance Tensor Programs for Deep Learning
- Roller: Fast and Efficient Tensor Compilation for Deep Learning
- Welder: Scheduling Deep Learning Memory Access via Tile-graph

Reference

- Ladder: Enabling Efficient Low-Precision Deep Learning Computing through Hardware-aware Tensor Transformation
- AI and Memory Wall
- Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations