Deep Learning Compiler

From the golden days to the era of LLM

Li Shang
lishang@fudan.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.

What have we learned so far?

Parallelism: The first goal is to split a computation intensive workload into many computation intensive tasks running concurrently on many compute units.

Locality: Computation and memory are physically separate. Data movement often becomes the bottleneck.

```
for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {

        // initialize output
        C[i][j] = 0.0f;

        for (int k = 0; k < K; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Optimizations

Tiling

What it is

Partitioning the iteration space into smaller blocks (tiles) to improve locality and parallelism.

What can be optimized

Tile sizes along each dimension (e.g., M/N/K for GEMM).

```
for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {

        // initialize output
        C[i][j] = 0.0f;

        for (int k = 0; k < K; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Example

Split a 4096×4096 matmul into 128×128 tiles to fit into shared memory.

Memory Placement

What it is

Deciding where data resides in memory hierarchy (global, shared, registers).

What can be optimized

Data movement and placement to maximize reuse and minimize latency.

Example

Load tiles of A/B into shared memory to reuse across threads.

Loop Permutation

What it is

Reordering loop nesting to change memory access patterns.

What can be optimized

Loop order (e.g., m–n–k vs. k–m–n).

Example

Switching to k–m–n improves reuse of A[m,k] in matrix multiplication.

```
for (int m = 0; m < M; m++) {
    for (int n = 0; n < N; n++) {
        float tmp = 0.0f;  // accumulator
        for (int k = 0; k < K; k++) {
            tmp += A[m][k] * B[k][n];
        }
        C[m][n] = tmp;
    }
}
```

```
for (int k = 0; k < K; k++) {
    for (int m = 0; m < M; m++) {
        float a = A[m][k];  // reuse
        for (int n = 0; n < N; n++) {
            C[m][n] += a * B[k][n];
        }
    }
}
```

Fusion/Fission

What it is

Fusion merges loops; fission splits loops.

$$Y = \mathrm{GELU}(XW + b)$$

What can be optimized

Balance between data reuse (fusion) and parallelism/resource pressure (fission).

Example

Fuse GEMM + bias + GELU to avoid writing intermediate results to global memory.

Unrolling

What it is

Expanding loop iterations into straight-line code.

What can be optimized

Unroll factor to trade instruction-level parallelism vs. register pressure.

Example

Unroll inner loop of size 8 to reduce branch overhead and increase throughput.

Normal loop

```
for ( int i = 0 ; i < 1024 ;  i++)
    a[i] = i;
```

After Loop Unrolling

```
for ( int  i= 0;  i < 128 ;  i += 8)
 {
  a[i] = i;    a[i+1] = i+1;
  a[i+2] = i+2; a[i+3] = i+3;
  a[i+4] = i+4; a[i+5] = i+5;
  a[i+6] = i+6; a[i+7] = i+7;
}
```

Thread/Block Mapping

What it is

Mapping computation tiles to CUDA threads, warps, and blocks.

What can be optimized

Thread/block dimensions, warp specialization, vector width. For instance, align computation with the warp execution unit to maximize parallel efficiency and reduce synchronization overhead. Enables data reuse in registers, reducing shared memory bandwidth pressure and increasing arithmetic intensity.

Example

Assign each warp to compute a 64×64 tile of output matrix.

Synchronization

What it is

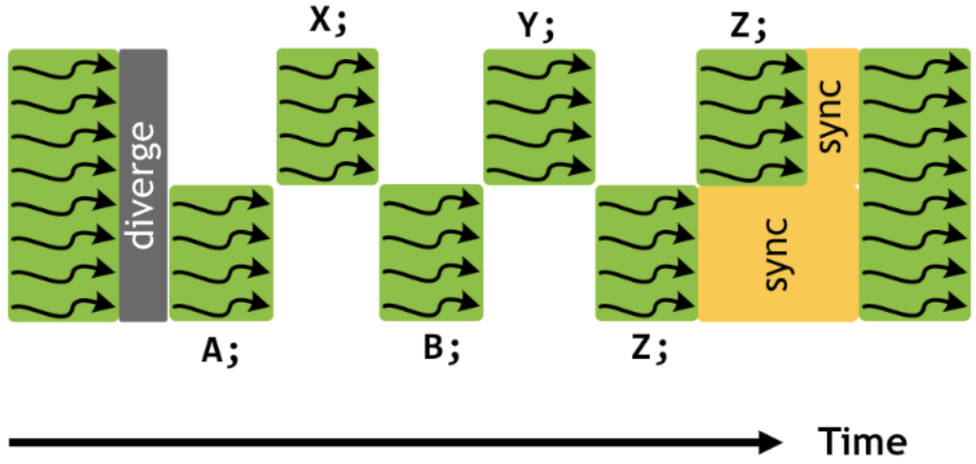
Coordinating execution order between threads.

What can be optimized

Placement and frequency of barriers or async synchronization.

Example

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```



Unified View: GPU Kernel Optimization as a Structured Discrete Search Space

A GPU kernel = a mapping from iteration space → hardware execution + memory hierarchy.

This mapping is parameterized by a set of discrete design choices:

- Tiling → define spatial decomposition & parallelism
- Memory placement → define data residency
- Loop permutation → define dataflow ordering
- Fusion / fission → define computation granularity
- Unrolling → define instruction-level expansion
- Thread/block mapping → define execution binding
- Synchronization → define temporal coordination

```
for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {

        // initialize output
        C[i][j] = 0.0f;

        for (int k = 0; k < K; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

What have we learned so far?

Vast optimization space: A discrete search problem over choices such as tiling factors, loop permutation, fusion/fission, unrolling, thread/block mapping, memory placement, and synchronization. .

Real-world performance: Depends on occupancy, register pressure, shared-memory usage, memory coalescing, bank conflicts, warp divergence, instruction mix, tensor-core eligibility, and launch geometry.

Optimization is a hard problem for human brain

What have we learned so far?

Selecting the best transformation is typically a high-dimensional discrete combinatorial optimization problem. Under realistic objectives and constraints, this search is generally intractable in the worst case and is commonly treated with heuristics, ILP/MIP formulations, pruning, or autotuning rather than exact global optimization.

Optimization is a hard problem for human brain

Why do we need a stack?

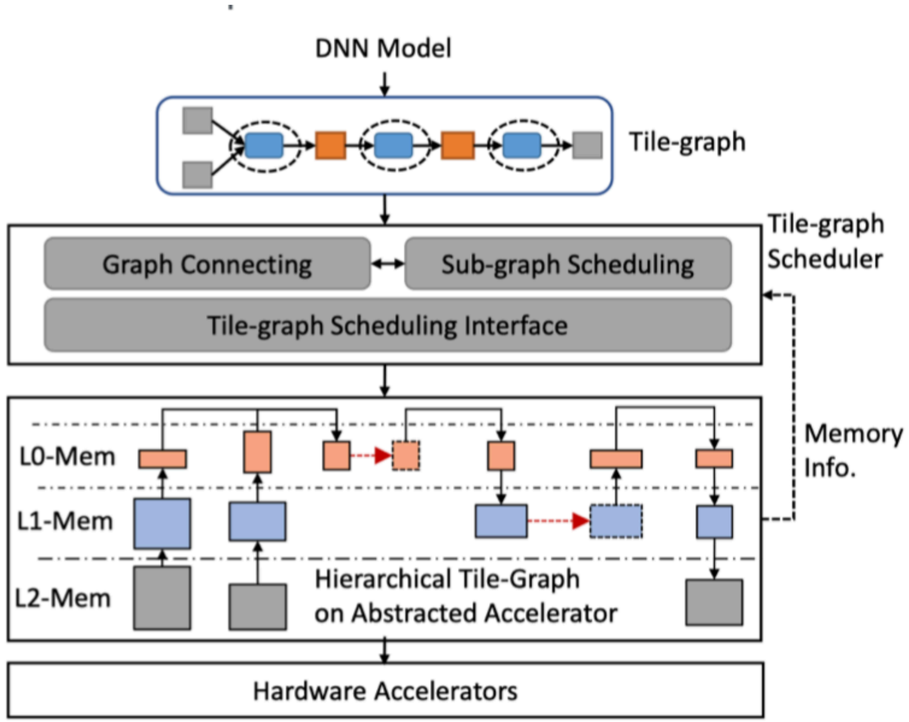
Operator is a mathematically defined function that takes one or more tensors (multi-dimensional arrays of data) as input and produces one or more tensors as output. They represent the building blocks of a neural network or a larger computation.

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

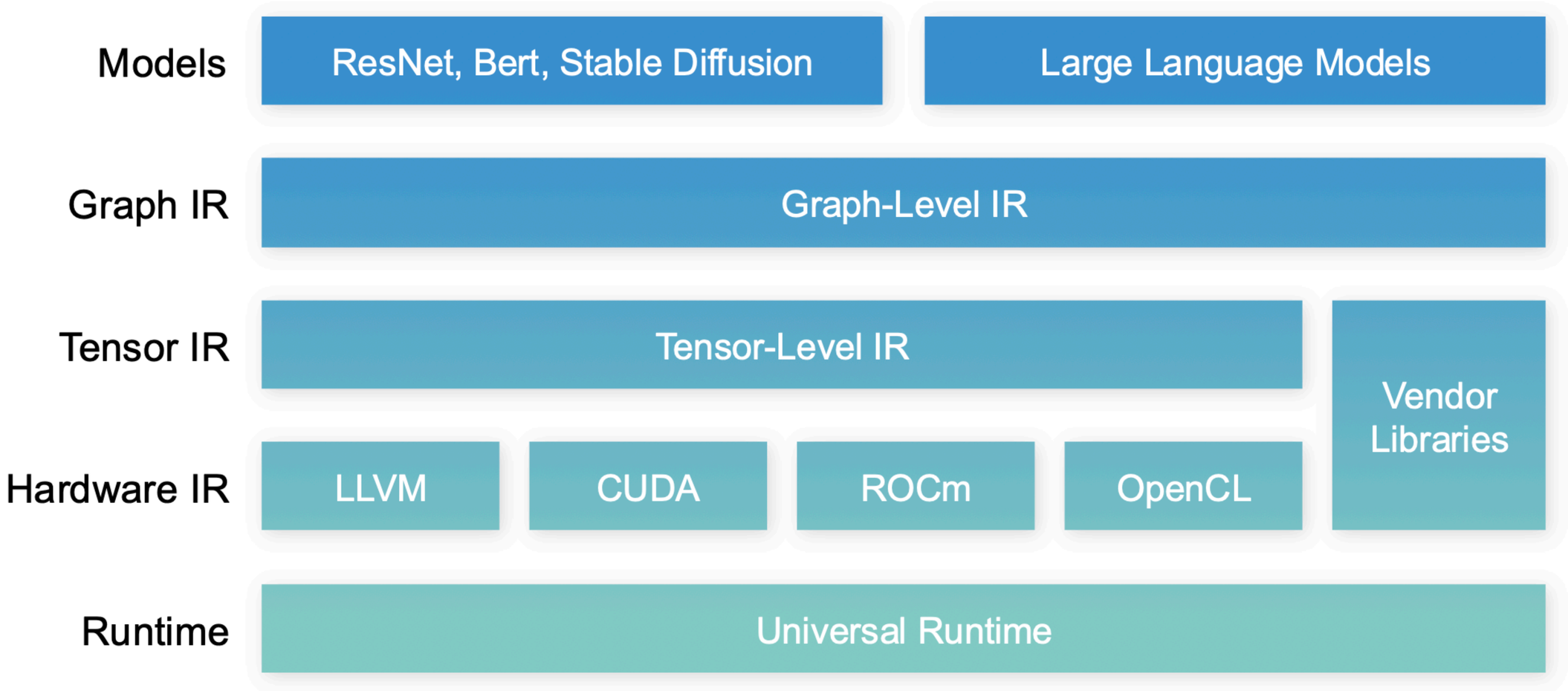
Kernel is a highly optimized computation implementation of an operator for a specific hardware architecture (like a GPU, CPU, or a specialized DNN accelerator).

Why do we need a stack?

- A model is written in a human-friendly form (e.g., linear, attention, activation, conv.)
- Hardware runs low-level hardware instructions (threads memory loads, reg., inst.)
- We need intermediate layers to bridge the gap (model description vs. computation)
- **Stack of representations:** Each layer exposes different structural optimization opportunities



Machine Learning Compilation Abstractions



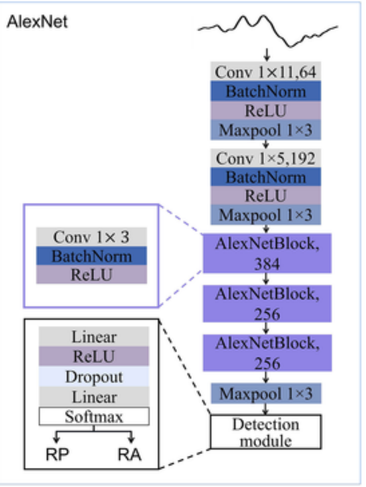
From Models to Hardware: The ML Compiler Stack

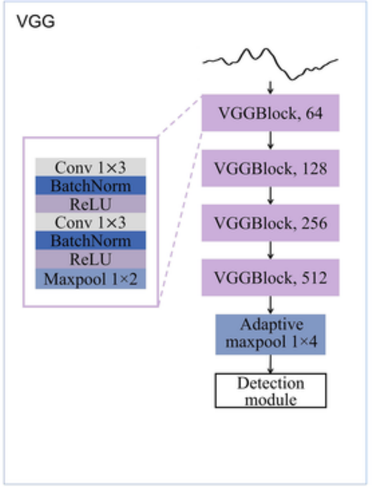
Models

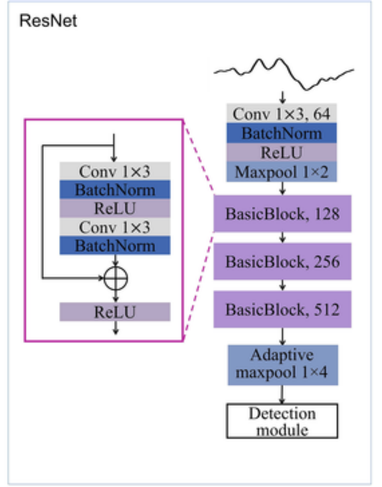
- High-level neural network definitions
- Describes function, not execution
- Organized as layers or modules
- Independent of hardware details

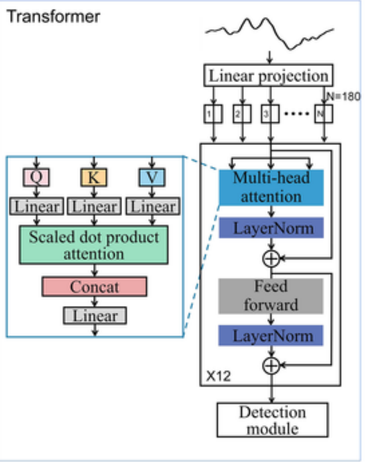
$$Y = \mathrm{GELU}(XW + b)$$

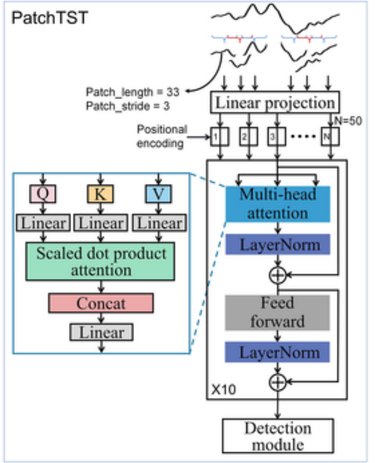
Excellent for expressing ideas, but too abstract for actual execution.

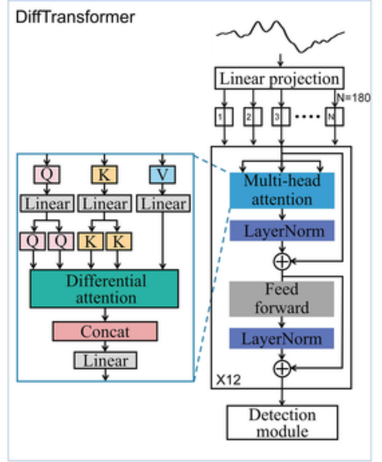


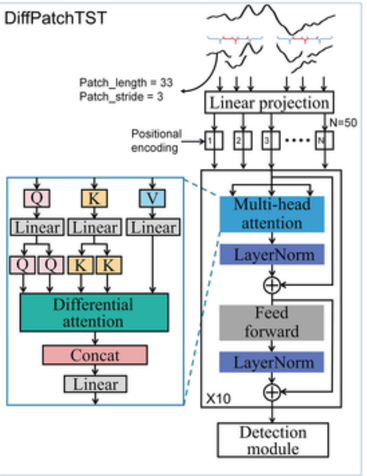


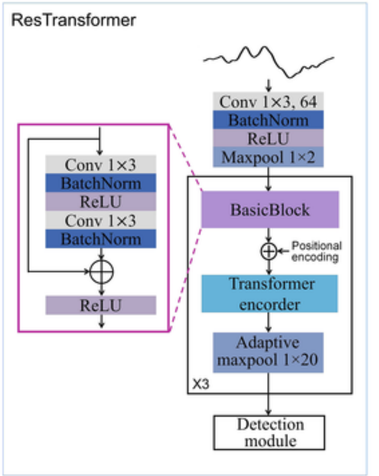


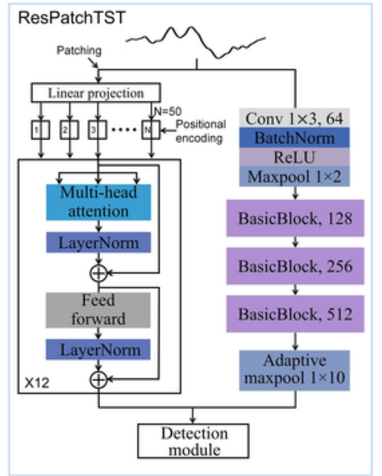










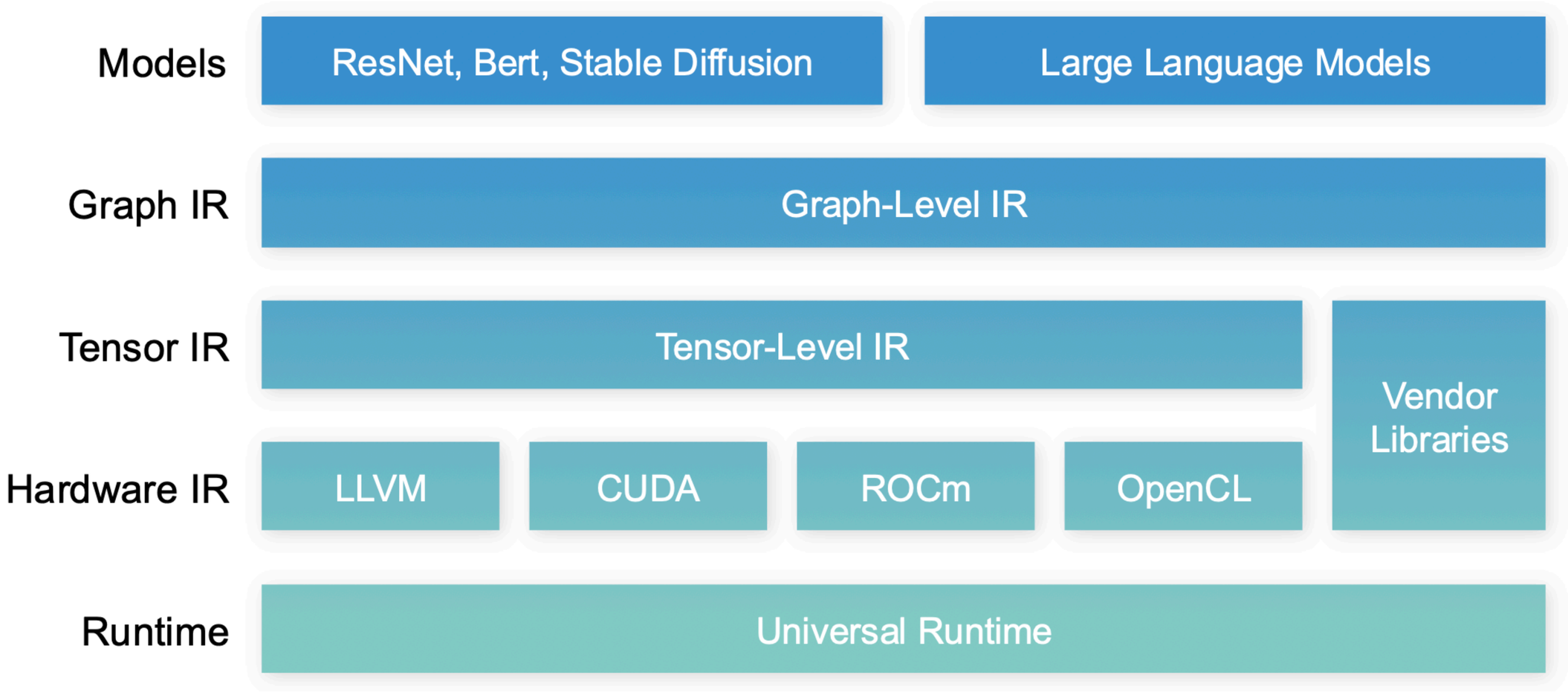


Models: What Can We Optimize Here?

- Computation: Change architecture (e.g., attention variants such as GQA， Mamba)
- Memory: Reduce size (hidden dimension, pruning, quantization, distillation)
- Maximal optimization potential, yet need to keep compiler & hardware in mind

Largest optimization potential, will be covered by the end of the semester...

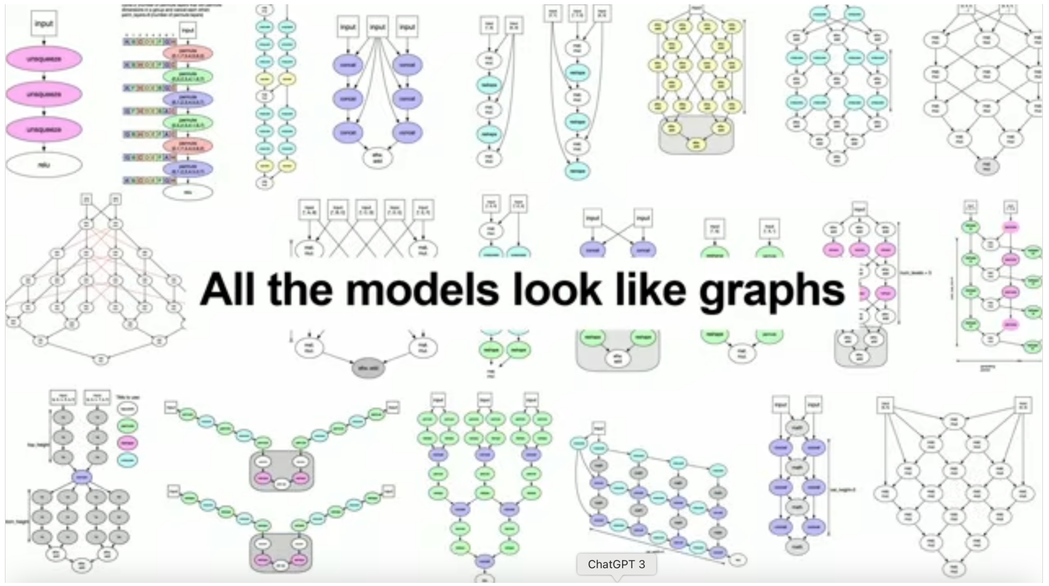
Machine Learning Compilation Abstractions



Models to Hardware: The ML Compiler Stack

Graph IR: What Is This Layer?

- Graph IR means graph intermediate representation
- Computation is written as operators connected by tensors
- Nodes are ops; edges are data dependencies
- The whole computation graph is still visible
- Framework-independent



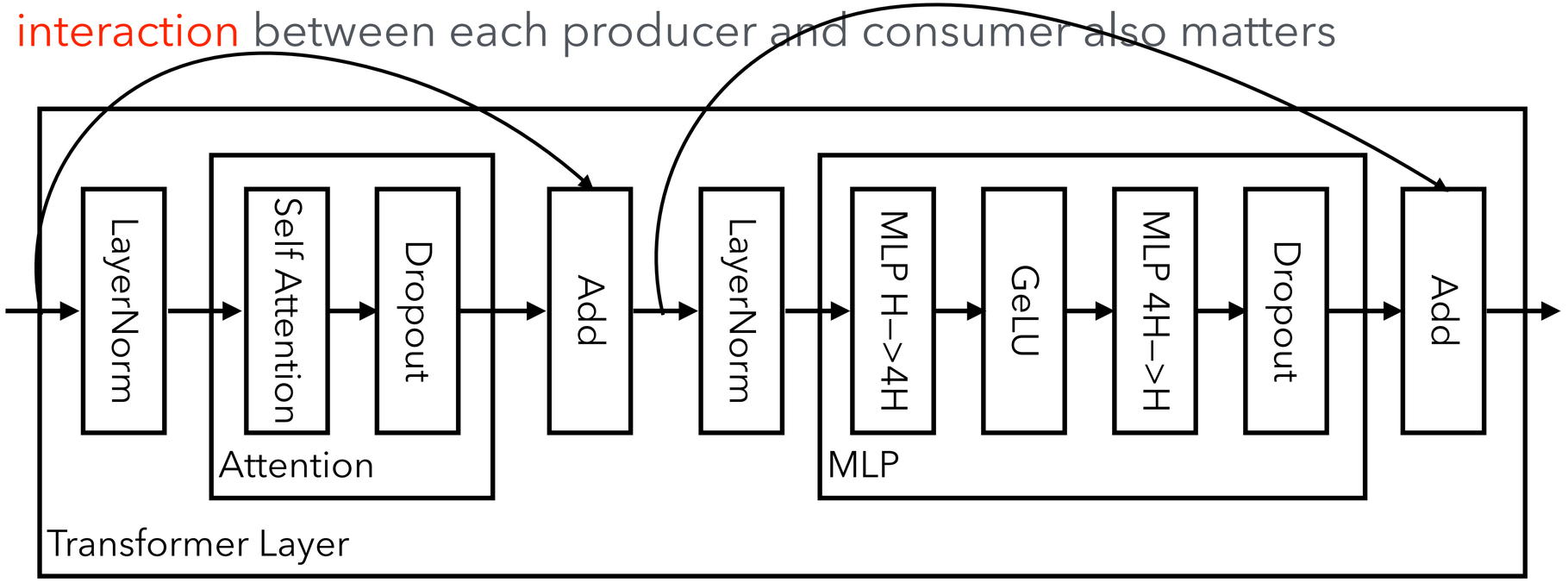
The first layer where the compiler gets a global, structured view of the computation.

```
y = GELU(Linear(x))
```

```
t1 = MatMul(x, W)
t2 = Add(t1, b)
y  = GELU(t2)
```

Graph IR: What Can We Optimize Here?

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



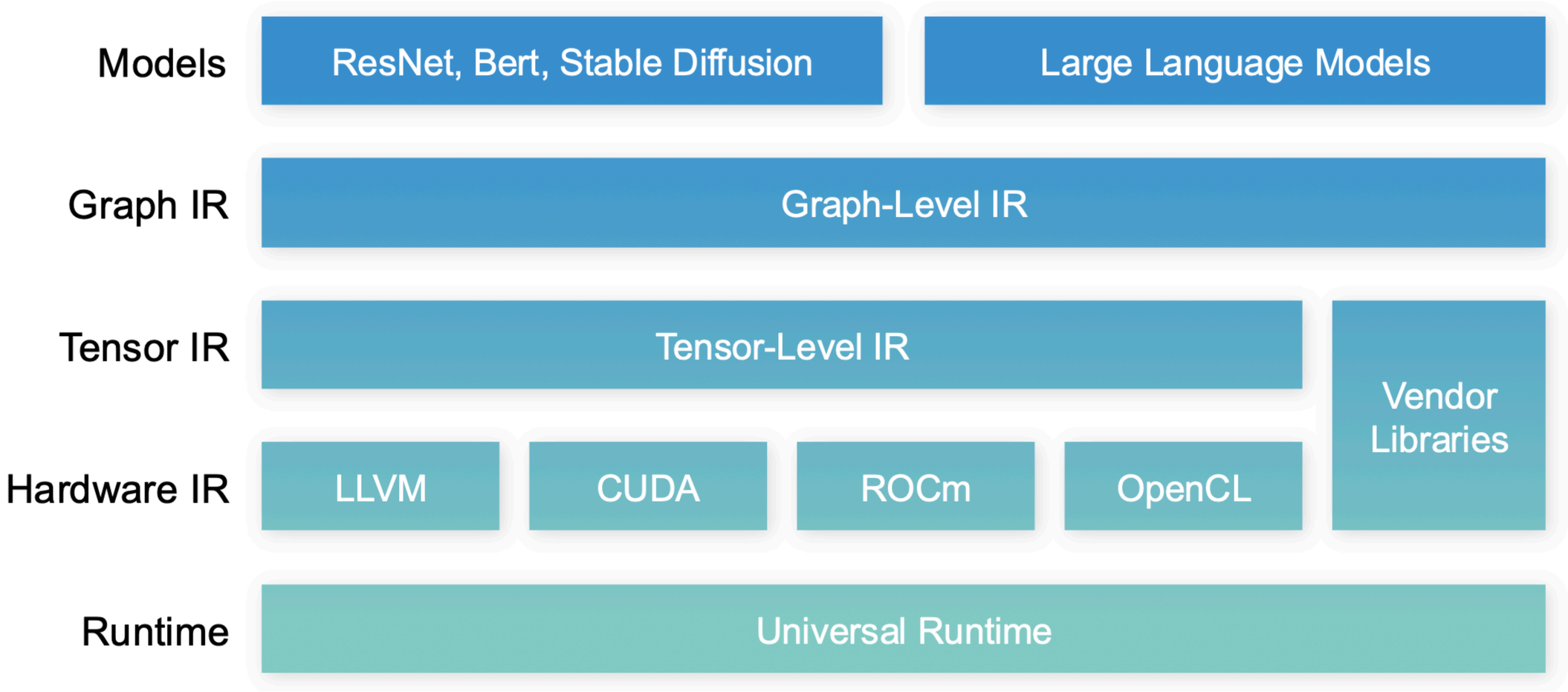
Graph IR: What Can We Optimize Here?

- Fuse adjacent operations
- Remove redundant computation
- Precompute constants
- Partition graph across devices

- Without fusion: MatMul, Add, and GELU run separately
- Intermediate tensor T1 and T2 may be written to memory
- With fusion: Add and GELU may be combined with output stage
- Result: fewer memory reads and writes

```
t1 = MatMul(x, W)
t2 = Add(t1, b)
y  = GELU(t2)
```

Machine Learning Compilation Abstractions



Models to Hardware: The ML Compiler Stack

Tensor IR: What Is This Layer?

- Tensor IR means tensor intermediate representation
- Tensor operations are opened into loops and indexing (loop-and-memory aware form)
- Shapes, reductions, and memory access become explicit
- How should the tensor operation actually be organized
- Bridge between math and hardware

```
t1 = MatMul(x, W)
t2 = Add(t1, b)
y  = GELU(t2)
```

```
for i, j:
  tmp = 0
  for k:
    tmp += X[i,k] * W[k,j]
  tmp += b[j]
  Y[i,j] = GELU(tmp)
```

Tensor IR: What Can We Optimize Here?

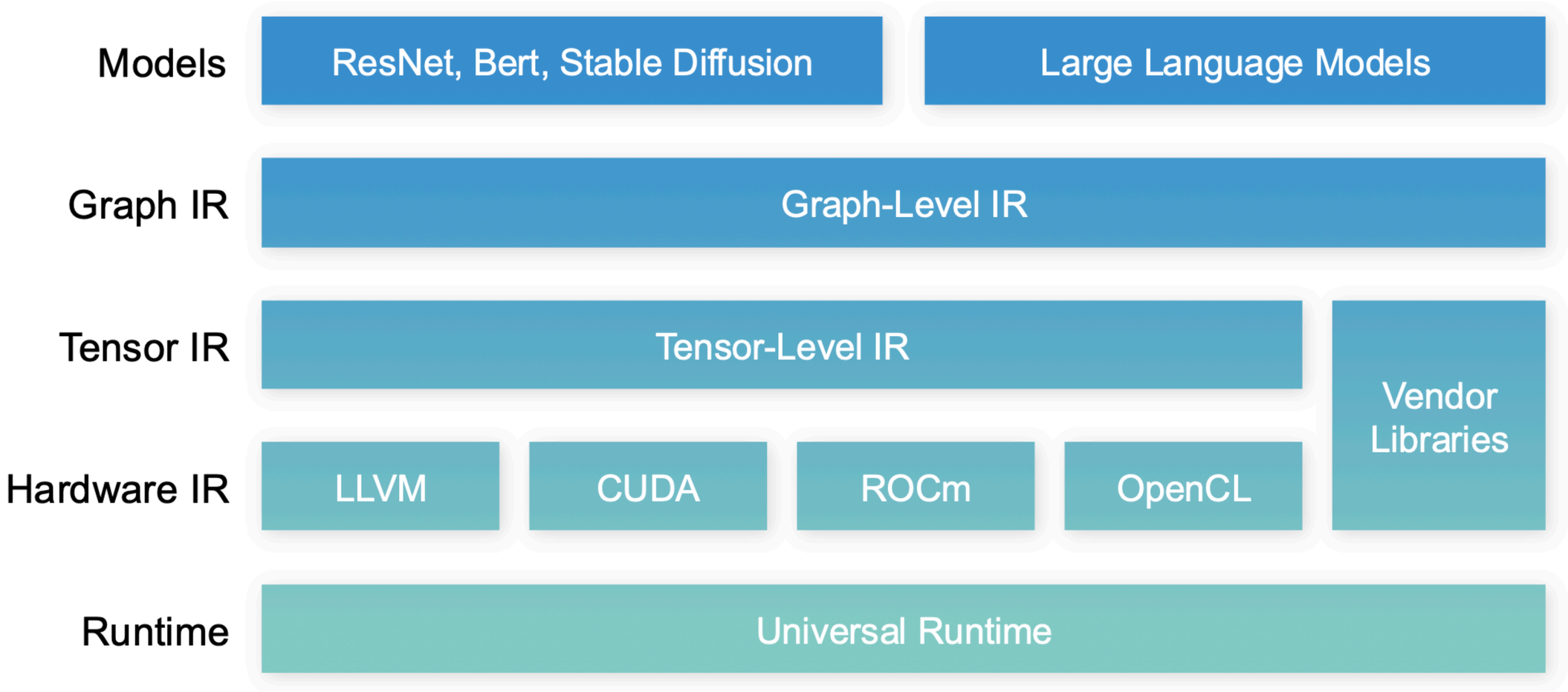
- Break computation into tiles
- Reorder loops for efficiency
- Reuse data in fast memory
- Enable parallel execution

Tensor IR and CUDA operate over the **same underlying degrees of freedom**:

Optimization Dimension	CUDA	Tensor IR
Tiling	manual loops	`split`
Loop order	manual reorder	`reorder`
Thread mapping	`threadIdx/blockIdx`	`bind`
Shared memory	`__shared__`	`cache_read/write`
Sync	`__syncthreads()`	implicit/annotated
Vectorization	intrinsics	`vectorize`

Tensor IR = structured, symbolic encoding of CUDA optimization space

Machine Learning Compilation Abstractions



Models to Hardware: The ML Compiler Stack

Hardware IR: What Is This Layer?

- Hardware-specific representation
- Examples: LLVM, CUDA, ROCm
- Expresses threads, memory, instructions
- Close to actual machine execution

```
for i, j:
  tmp = 0
  for k:
    tmp += X[i,k] * W[k,j]
  tmp += b[j]
  Y[i,j] = GELU(tmp)
```

```
block (bx, by) computes tile Y[bx:bx+64, by:by+64]
threads cooperate to load X and W tiles
accumulate in registers
write tile back to global memory
```

Hardware IR: What Can We Optimize Here?

- Choose thread-block shape and thread mapping
- Improve memory coalescing and shared-memory reuse
- Manage register usage carefully
- Decide whether to use custom kernels or vendor libraries

Optimization becomes less about algebra and more about physical resource management.

- bandwidth
- latency
- occupancy
- register pressure
- synchronization cost

Runtime: What Is This Layer?

- Runtime is the system that actually executes the compiled plan
- It allocates memory, launches kernels, and manages devices
- It handles synchronization and resource coordination
- It decides when and where work runs

```
for i, j:
  tmp = 0
  for k:
    tmp += X[i,k] * W[k,j]
  tmp += b[j]
  Y[i,j] = GELU(tmp)
```

```
allocate buffers
copy / bind inputs
launch kernel
synchronize if needed
pass Y to next operator
```

Runtime: What Can We Optimize Here?

- Overlap computation and data movement
- Reuse memory buffers across operations
- Schedule work to reduce idle hardware time
- Minimize unnecessary synchronization

- LLM inference with KV cache
- Overlap cache updates and compute
- Reuse buffers across steps
- Reduce latency per token

Putting It All Together

At the Models layer, the computation is a meaningful neural network block.

At the Graph IR layer, it becomes a graph of operators.

At the Tensor IR layer, the operators become loop-based tensor programs.

At the Hardware IR layer, those programs become device-specific execution plans.

At the Runtime layer, those plans are actually launched and coordinated.

Performance comes from preserving and exploiting structure at every level.

Take Home Message

- One computation can (also need to) have many representations
- Different layers preserve different kinds of structure
- Optimization depends on what structure is visible
- Good systems design means lowering structure without losing what matters

Performance comes from preserving and exploiting structure at every level.

ML Compiler: A Decade-Long Journey

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.

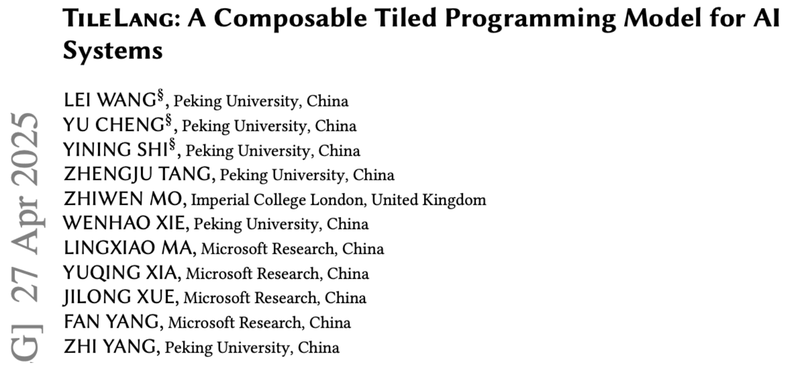
- The focus of deep-learning compilers moved from hardware generality to maximizing performance on a single dominant platform. Triton, an open-source project from OpenAI, emerged as a new focal point. Often described as "CUDA for Python," Triton enables developers to write efficient custom GPU kernels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

The LLM Era (After 2020)

Attention indeed is all you need — so far.

- **TileLang** is a high-performance operator DSL that gives programmers explicit control over scheduling, memory layout, pipelining, and thread binding, allowing them to manually optimize complex kernels beyond the performance ceiling of more automated systems like Triton, at the cost of much higher programming complexity.

- **CUDA-L2** is a reinforcement learning–driven system for GPU kernel optimization that automatically learns to generate high-performance matrix multiplications implementations by directly exploring the discrete design space of tiling, memory hierarchy usage, and thread mapping. It treats kernel design as a sequential decision process and discovers non-obvious scheduling strategies that better exploit data locality and reuse, enabling it to surpass hand-engineered libraries on certain workloads.



GJ 27 Apr 2025

TileLang: A Composable Tiled Programming Model for AI Systems

LEI WANG[§], Peking University, China
YU CHENG[§], Peking University, China
YINING SHI[§], Peking University, China
ZHENGJU TANG, Peking University, China
ZHIWEN MO, Imperial College London, United Kingdom
WENHAO XIE, Peking University, China
LINGXIAO MA, Microsoft Research, China
YUQING XIA, Microsoft Research, China
JILONG XUE, Microsoft Research, China
FAN YANG, Microsoft Research, China
ZHI YANG, Peking University, China

CUDA-L2: Surpassing cuBLAS Performance for Matrix Multiplication through Reinforcement Learning

Songqiao Su, Xiaofei Sun, Xiaoya Li, Albert Wang, Jiwei Li and Chris Shum

DeepReinforce Team

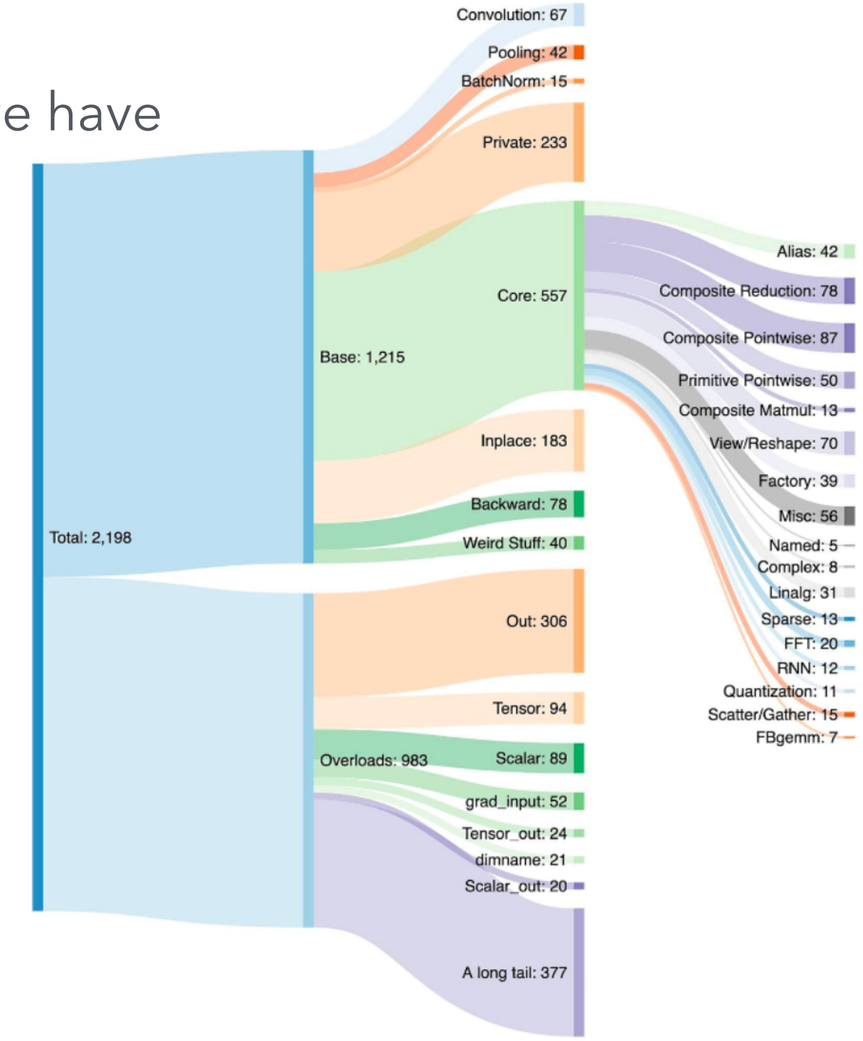
github.com/deepreinforce-ai/CUDA-L2

Pre-LLM Era

Motivation

Actually, we find traditional ways do not work well

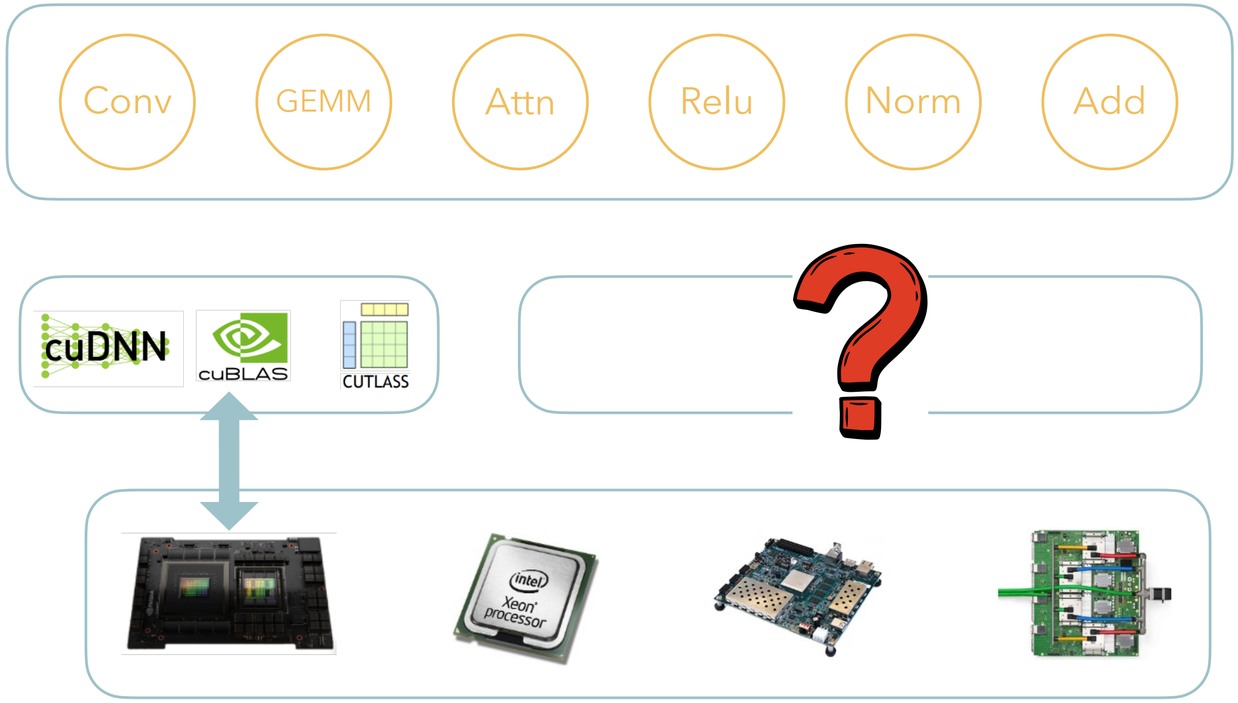
- Deploying any model anywhere is labor-intensive, since we have
  - hundreds of operators



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.

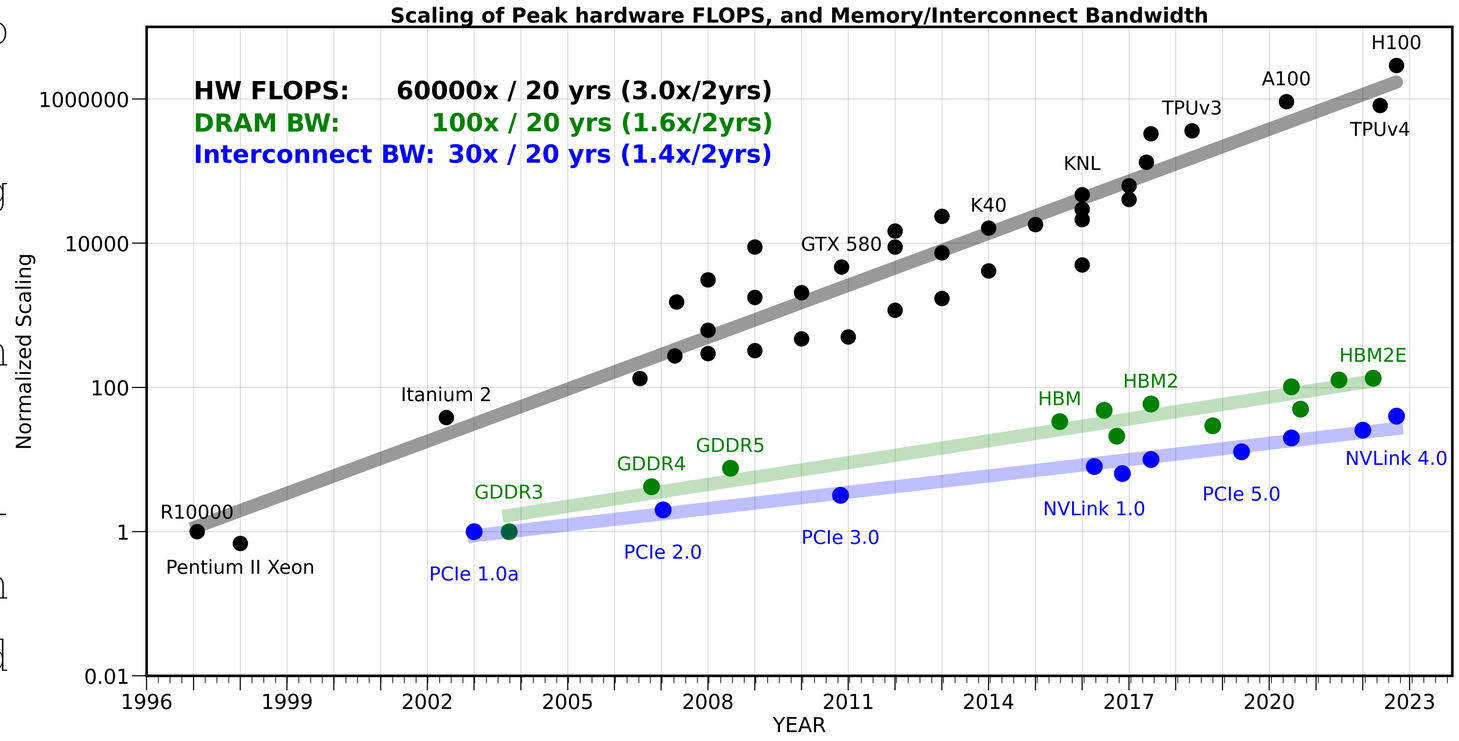
Motivation

Actually, we find traditional ways do not work well

- Evolving neural architectures call for an efficient way to generate kernels with

- Reduces memory access, because intermediate results stay in fast on-chip memory;
- Improves compute utilization, keeping the GPU or accelerator busy;
- And ultimately boosts overall execution efficiency of the model.

The compiler isn't just about portability — it's a tool to close the gap between hardware's theoretical performance and real-world efficiency.



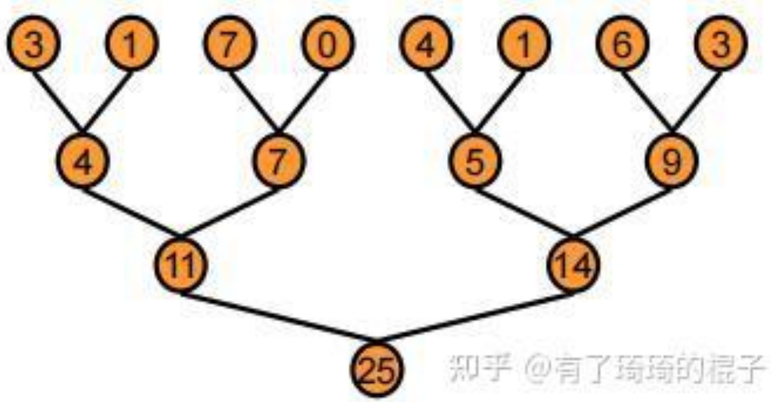
Deep Learning Compiler

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

Example




```
__global__ void reduce0(float *d_in,float *d_out){
    __shared__ float sdata[THREAD_PER_BLOCK];

    //each thread loads one element from global memory to shared mem
    unsigned int i=blockIdx.x*blockDim.x+threadIdx.x;
    unsigned int tid=threadIdx.x;
    sdata[tid]=d_in[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s<blockDim.x; s*=2){
        if(tid%(2*s) == 0){
            sdata[tid]+=sdata[tid+s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if(tid==0)d_out[blockIdx.x]=sdata[tid];
}
```

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

Kernel 7 on 32M elements: 73 GB/s!

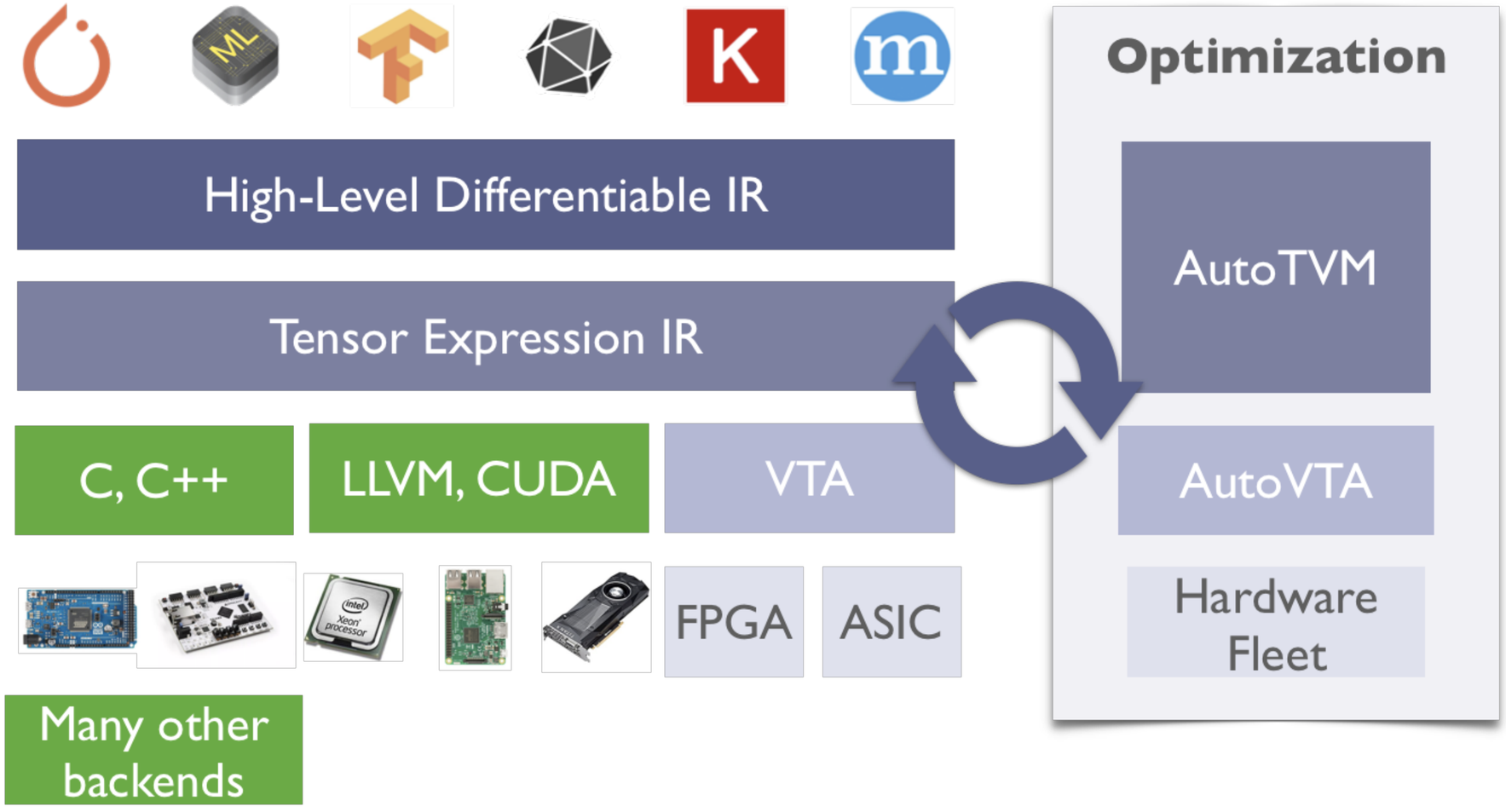
TVM

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 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.

TVM


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

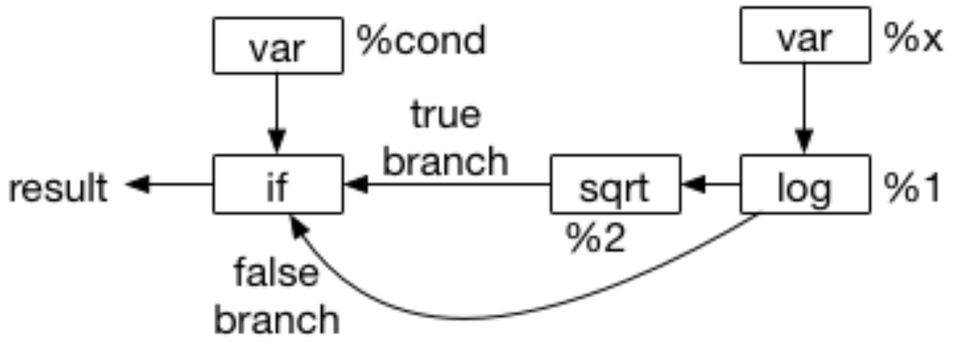
High-level Graph Optimizations

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: 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.

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, defineing 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
```

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

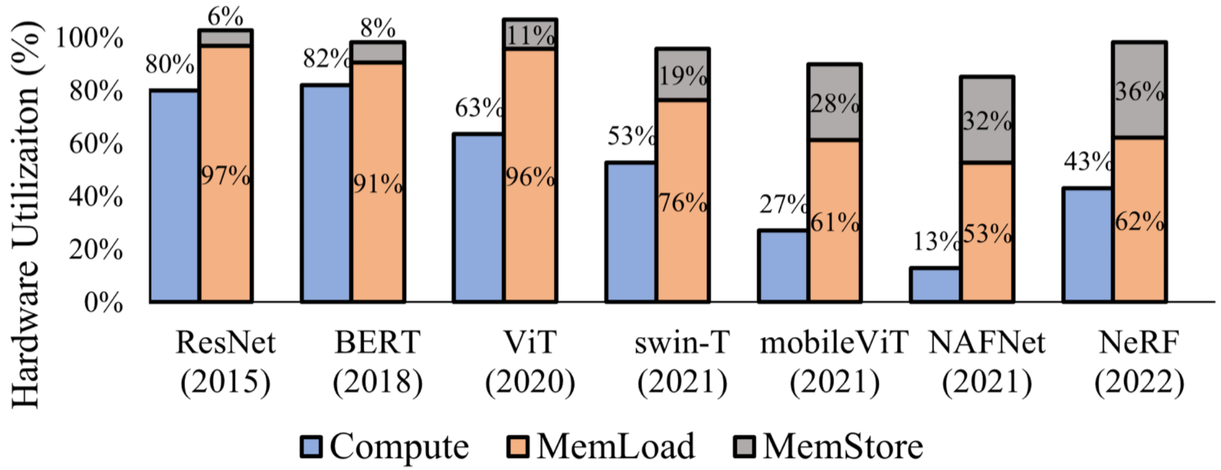
🧱 Summary: TVM Optimization Hierarchy

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

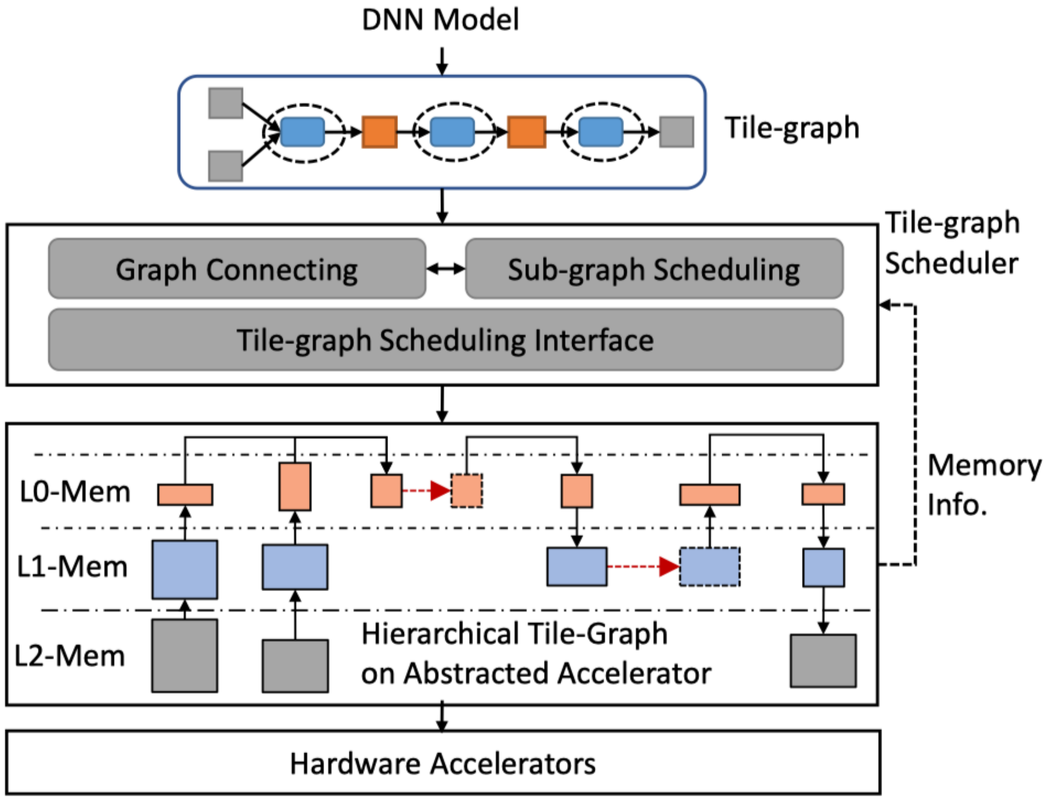


Deep Learning Compiler

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

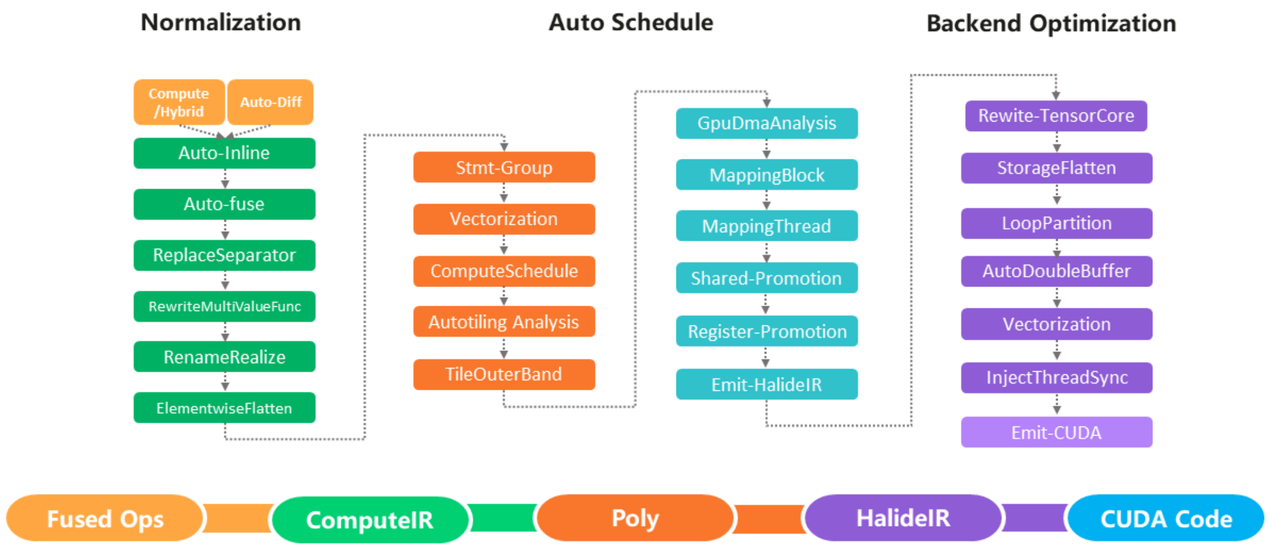
With Tile Abstraction
Compute multiple ops in-place per tile
Most intermediate data stays in on-chip memory
Compute units always busy — **compute-bound**
Minimal read/write — **less I/O, more FLOPs**



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

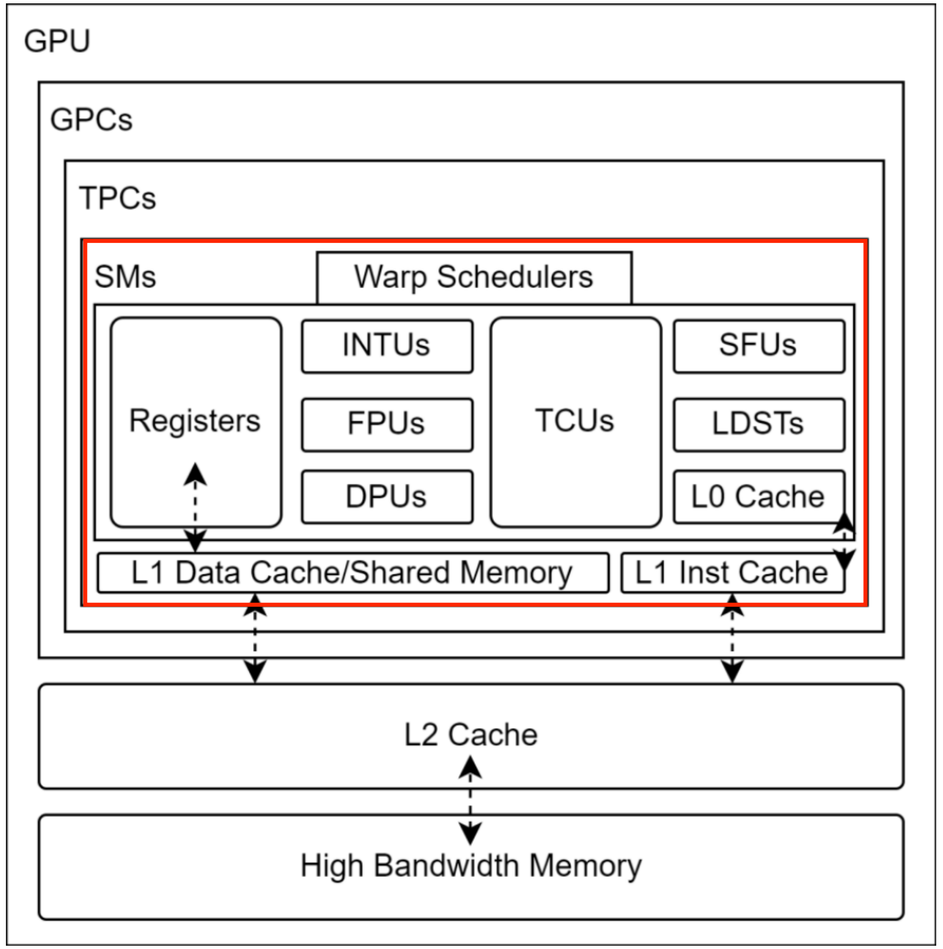
- Triton **Python + CUDA-level performance.**
  - to provide an open-source environment to write fast code at higher productivity than CUDA, but also with higher flexibility than other existing DSLs

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

Tile-level program and hides thread-level parallelism inside vector operations

```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

```cpp
#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;
}
```

- explicit thread index
- manual vectorization (`float4`)
- manual memory access
- explicit register use

```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)
```

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.

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.

What is next...

Triton vs TileLang：“Ease of Use vs. Control”

Design Philosophies of Two High-Performance Operator DSLs

From automation to explicit scheduling: memory layout, pipelining, thread binding, and cross-platform compilation optimization

Key Questions

1. Why are Triton and TileLang built on fundamentally different design philosophies?
2. Which hardware details does each DSL hide or expose?
3. What dimensions of parameters are optimizable?
4. How does TileLang push beyond the performance ceiling of Triton?

Triton：Design Philosophy Overview

The core idea of Triton is: Let the user write block-level logic, while the compiler automatically handles scheduling, memory access optimization, and pipelining.

Key Characteristics

1. Python-style SPMD (Single Program, Multiple Data) model: The user writes code that represents the behavior of a *parallel thread group*, and the compiler maps it onto CUDA thread hierarchies.

2. Hides low-level details such as shared memory and thread scheduling: Users do not need to reason about bank conflicts, warp mapping, or memory coalescing.

3. Automatically generates high-performance loop structures for matrix operations: Includes software pipelining, asynchronous loads, and Tensor Core lowering.

4. Well-suited for GEMM, FlashAttention, and local operator optimization: Achieves 80–90% of hand-written CUDA performance with only tens of lines of code.

Limitation

Because scheduling is treated as a black box, it is difficult to push beyond the performance ceiling for complex operators.

TileLang：Design Philosophy Overview

The core idea of TileLang is: Give users explicit control over scheduling and memory layout, enabling fine-grained performance optimization through manual composition.

Key Characteristics

1. Tile-based abstraction built on TVM TensorIR: Scheduling primitives (split, fuse, reorder, cache, tensorize) are directly controlled by the user.
2. Explicit control over memory hierarchy: Users can fully define layouts in shared memory, registers, and global memory, including tiling, swizzling, and alignment to eliminate bank conflicts.
3. Explicit thread binding: Users can directly specify threadIdx.x/y and design warp-level specialization.
4. Strong cross-backend potential: Built on TVM infrastructure, enabling portability across NVIDIA, AMD, Intel, Ascend, MLU, etc.

Trade-offs

- Advantage: Enables extreme performance optimization for complex kernels (e.g., attention variants, quantized kernels)
- Disadvantage: High learning curve and significant development complexity

Memory Layout Differences

Triton: Implicit layout

- Compiler decides how tiles are placed in shared memory or registers
- Works well for simple cases
- Performance may degrade in complex fused kernels

TileLang: Explicit layout

- Users can control: Tile arrangement in shared memory
- Swizzling to avoid bank conflicts
- Padding and alignment for Tensor Core efficiency

Optimizable parameters

- Tile sizes (M/N/K)
- Memory scope (global/shared/local)
- Layout transformations (swizzle/pad/transpose)
- Alignment (32/64/128 bytes)

Pipelining Differences

Triton: Automatic pipelining

User specifies only num_stages, while the compiler:

- Overlaps load/store/compute
- Prefetches future tiles
- Uses async load + barriers

Pros: simple

Cons: no guarantee of optimal pipeline structure

TileLang: Explicit pipelining

Users control:

- Pipeline depth (2/3/4 stages)
- Double buffering
- Prefetch distance
- Load/compute ordering

Optimizable parameters

- Pipeline stages
- Prefetch distance
- Buffer count
- Execution ordering

Thread Binding Differences

Triton: SPMD abstraction

- Thread behavior is implicit
- One Triton program ≈ one CUDA block
- Warp/thread assignment handled by compiler

Result: convenient but opaque

TileLang: Explicit binding

- Direct mapping of threads/warps to tasks
- Supports warp specialization (e.g., Hopper warp-group pipeline)
- Enables fine-grained vectorization

Optimizable parameters

- Thread/block hierarchy
- Warp-level pipeline
- Vector width
- Execution group size

Observability and Debugging

Triton challenges

- Heavy compiler transformations obscure PTX
- Difficult to trace performance bottlenecks
- Hard to diagnose issues like:
- Bank conflicts
- Warp divergence

TileLang advantages

- Scheduling decisions are explicit in code
- Debugging is systematic:
  - Adjust tile size
  - Reorder loops
  - Modify thread binding
  - Change caching strategy
  - Tune tensorization

Result: tuning is interpretable and reproducible

Summary

Triton excels at

- Standard operators: Linear, Softmax, LayerNorm
- FlashAttention (v1/v2)
- Rapid prototyping of GEMM/Conv
- Filling gaps in training frameworks

TileLang excels at

- Complex attention variants (FlashMLA, MQA, grouped attention)
- Quantized kernels (INT4/INT3 packed layouts)
- Manual GPU memory/Tensor Core optimization
- Complex fused kernels (e.g., GEMM + Bias + GELU)

Final Takeaway

Triton → automation, high productivity, strong baseline performance

TileLang → explicit control, higher performance ceiling, better for complex and cross-hardware optimization

If your goal is to quickly build a high-performance kernel → use Triton

If your goal is to push performance limits or optimize across hardware → use TileLang

CUDA-L2: Surpassing cuBLAS Performance for Matrix Multiplication through Reinforcement Learning

Songqiao Su, Xiaofei Sun, Xiaoya Li, Albert Wang, Jiwei Li and Chris Shum

DeepReinforce Team

github.com/deepreinforce-ai/CUDA-L2

LLM performance is bottlenecked by custom GPU kernel development

- LLMs increasingly rely on specialized code (FlashAttention, Mamba, custom convolutions, fused operators), and writing optimized kernels is slow, error-prone, with deep hardware knowledge
- Hardware diversity is rising: A100 → H100 → GB300, NPU, TPU, custom accelerators

Key motivation: *learned optimization can surpass the best human-written GEMM kernels used by NVIDIA libraries.*

There have been a rapidly growing body of work using LLM+RL based approach for kernel optimization, implementation, verification and testing (30+).

CUDA-L2 summary

CUDA-L2 demonstrates that LLM-guided reinforcement learning can outperform even NVIDIA's heavily engineered libraries (cuBLAS and cuBLASLt) on half-precision GEMM (HGEMM) kernels across 1000 different (M, N, K) configurations.

Its core finding is that an LLM, when trained with execution-time rewards, profiling signals, and contrastive feedback, can autonomously discover hardware-optimized strategies that human experts rarely explore due to the vastness of the search space.

CUDA-L2 beats cuBLAS/cuBLASLt on real hardware

Across 1000 GEMM shapes on A100:

- +19.2% faster than cuBLAS (best of NN/TN layouts)
- +16.8% faster than cuBLASLt-heuristic
- +11.4% faster than cuBLASLt AutoTuning (its strongest configuration)
- +22–28% faster than PyTorch’s matmul

These results confirm that even the highly optimized industry kernels are not globally optimal, and machine learned optimization can surpass human intuition.

CUDA-L2 automatically learns architecture-specific scheduling behaviors

The LLM, trained with RL, consistently rediscovers optimization patterns aligned with fundamental GPU performance principles:

- When to use larger tiles for memory reuse
- When to shrink tiles to avoid register pressure
- When to increase or reduce pipeline stages
- How to tune prefetch distances
- When to enable block-swizzling for better L2 locality

Across 1000 shapes, the system learns general scheduling “principles” without being explicitly taught them.

The performance improvement is highest for small and mid-sized GEMMs

- For small matrices (low MNK), speedups reach 1.3×–1.4×
- For large matrices (high MNK), speedups converge toward cuBLAS due to GPU saturation

This suggests that human-written kernels underperform most severely when workloads are irregular or too small to saturate the GPU—precisely where RL excels at exploring bespoke solutions.

Case 1: Zero-padding to unlock better tile sizes (Novel Trick)

The phenomenon:

- Many matrix dimensions (e.g., M = 8192) are divisible only by limited tile factors.
- CUDA-L2 chooses BM = 160, even though 160 does not divide 8192.

What it does: It pads the matrix to 8320 rows (+1.6%) to enable this tile size.

Result:

- BM = 128 → poor performance
- BM = 160 with padding → +15.2% faster than cuBLASLt AutoTuning
- BM = 256 → register pressure kills performance

Why it's surprising:

- Humans avoid padding because it adds useless FLOPs.
- CUDA-L2 learns that *better tiling outweighs small padding overhead*.

Case 2: Multi-step prefetching (e.g., K+4 lookahead) rather than standard K+1

Standard kernels prefetch one tile ahead.

CUDA-L2 often uses:

- Prefetch of K, K+1, K+2, K+3, K+4 ahead
- Priming the pipeline at the start

```
for (k_tile = 0; k_tile < K; k_tile += TILE_K) {
    load A[:, k_tile]
    load B[k_tile, :]
    compute
}
```

Benefit:

- On A100, async copy latency is high; prefetch depth >1 hides memory stalls dramatically.
- Human hand-tuned kernels rarely explore such deep prefetch because it’s architecture-sensitive.

Case 3: Direct wide (128-bit) register → shared-memory writes

CUTLASS often uses:

```
registers → temporary tensor → shared → global
```

CUDA-L2 identifies cases where the temporary tensor is unnecessary (right shape, alignment, and ordering,) and instead uses wide 128-bit copies directly:

```
registers ⇒ shared memory (uint128_t vectorized store)
```

Benefit:

- Fewer memory instructions
- Higher throughput on Ampere shared memory
- Cleaner epilogue pipeline

Case 4: Double-buffered register fragments with ping-pong scheduling

CUDA-L2 learns to allocate *two* register fragments:

- While buffer A is used for MMA compute,
- buffer B prefetches the next tile, and vice versa.

Result:

- Removes load stalls
- Greatly speeds up kernels with large K
- Automatically avoided when register pressure would overflow

Humans often do double buffering, but CUDA-L2 learns *when* it is worthwhile on each shape (register pressure hence occupancy drops vs. load stalls).

Case 5: Staggered A/B prefetch scheduling

Effect:

- Increases instruction-level parallelism
- Smooths memory traffic, avoiding bandwidth bursts
- Matches asynchronous memory pipelines more effectively

This is a delicate optimization that depends on the precise instruction mix and warp scheduling.

Instead of:

```
code

prefetch A
prefetch B
compute
```

CUDA-L2 discovers:

```
code

prefetch A
compute
prefetch B
```

Case 6: CTA block swizzling tuned per shape

Block swizzling rearranges CTA order to improve L2 reuse.

CUDA-L2 discovers:

- Small GEMMs → often no swizzle
- Medium GEMMs → small stride swizzle
- Large GEMMs → large stride swizzle (e.g., 512–16,384)

This level of shape-specific swizzle tuning is almost never done manually.

Overall Insight

CUDA-L2 demonstrates that LLMs can act as GPU performance engineers, discovering:

- tile shapes humans would not try,
- multi-stage pipelines difficult to reason about manually,
- layout transformations beyond human heuristics,
- and complex interactions between registers, shared memory, and tensor cores.

This is an evidence that learned optimization can surpass the best human-written GEMM kernels used by NVIDIA libraries.

LLM4AI Infra： Beat it at its own game

Specialized post-trained kernel models. These works train or post-train LLMs specifically for kernel generation, usually with execution-aware rewards, RL, or synthetic data. Representative examples include AutoTriton, CUDA-L1, CUDA-L2, Dr. Kernel, CUDA Agent, and DRTriton.

Agentic iterative search systems. Instead of relying only on model weights, these systems run a closed loop of generate → compile → verify → profile → refine. Representative examples include CUDA-LLM, AutoKernel, StitchCUDA, and KernelFalcon.

Hardware-aware / profiling-aware optimization. These methods feed profiling counters, access patterns, or architectural bottlenecks back into the search loop so optimization is guided by hardware evidence rather than blind trial and error. Representative examples include SwizzlePerf, PRAGMA, KernelBand, and recent work using DSL plus speed-of-light guidance.

Evolutionary and population-based search. These systems maintain multiple candidate kernels and evolve them over time, which helps avoid local optima in a rugged search space. Representative examples include GPU Kernel Scientist, cuPilot, K-Search, KernelFoundry, AVO, and Kernel-Smith.

Cross-hardware and non-CUDA expansion. This cluster extends the paradigm beyond NVIDIA CUDA into heterogeneous hardware, alternative backends, and newer kernel abstractions. Representative examples include KernelEvolve, KernelCraft, CuTeGen, and the earlier QiMeng line of work.

Benchmarks and evaluation infrastructure. This line focuses on making the field measurable and comparable, especially around correctness, latency, tuning budget, and baseline fairness. KernelBench is the central benchmark, and newer work expands evaluation to emerging hardware and more realistic agent settings.