CSE 291A/DSC 291: Data Systems for Machine Learning, Spring 2026

7: Triton, graph optimization, memory

Lecturer: Hao Zhang

Scribes: Gaurav Joshi, Prince Modi, Hritik Bharucha, Tongfei Yang, Nishchay Mahor, Ayush Govind, Ishit Gulati, Karthik Nandagudi Raviprakash

1 Introduction and Recap

This lecture connected low-level GPU optimization ideas to the broader themes of ML compilation and graph transformation. It began with a short recap of GPU execution concepts through multiple-choice review questions. The recap revisited the meaning of a GPU kernel, the role of shared memory, why over-subscribing the GPU helps hide latency, and how to reason about whether an operator is memory-bound or compute-bound. The lecture also briefly returned to the GEMM tiling problem and emphasized an important systems lesson: larger tile sizes are not always better, because practical performance depends on resource limits such as available registers, SRAM, and parallelism.

The recap motivated a broader question. In previous lectures, making matrix multiplication fast required carefully selecting implementation details such as thread count, register usage, and shared-memory allocation. One approach is to hand-tune kernels, enumerate many possible configurations, and then profile them. Another is to automate this process through compilation.

2 From Matmul Tuning to ML Compilation

In the previous lectures, we established a two-level tiling strategy to optimize Matrix Multiplication (Matmul). The first level is **SRAM-level tiling**, where we allocate a shared memory block of size $S \times L$ to be accessed globally by all threads within a thread block. The second level is **register-level tiling**, where each individual thread computes a smaller $V \times V$ sub-matrix, keeping values locally in registers to accumulate the results.

While the underlying logic of this nested loop is straightforward, implementing it in practice (e.g., writing the actual CUDA kernel) reveals a massive performance cliff: how do we choose the optimal values for the tile sizes S, L, and V? These variables dictate the exact configuration of thread blocks, register usage, and shared memory allocation. The optimal choice is highly sensitive to both the specific processor architecture (available SRAM, maximum registers per thread) and the exact dimensions of the matrices being multiplied.

To resolve this configuration problem, there are roughly two paths:

- **Expert-Crafted (Manual Tuning):** Developers write scripts to perform grid searches or binary searches, empirically testing and profiling different values until the best configuration is found.
- **Operator Compilation:** Building a compiler that automatically evaluates the hardware specifications and matrix shapes to deduce the optimal configuration without exhaustive manual testing.

At its core, a traditional software compiler translates human-readable imperative code (like C++) into machine code instructions. Machine learning compilation applies this exact paradigm to deep learning. However, in ML systems, the "human code" is the high-level model definition (such as a Transformer or ResNet). This model is first converted into an intermediate **Dataflow Graph**. The ML compiler's job is to ingest this graph alongside the target hardware specifications and automatically generate highly optimal machine code tailored to that specific environment.

3 Big Picture of ML Compilation

The main challenge for the ML compiler community can be summarized in a single goal: ***Automatically generate optimal configurations and machine code given a user's high-level ML code and target hardware.***

To successfully translate a high-level model down to bare metal, modern ML compilers must solve problems across three distinct levels of abstraction:

1. **Programming-level compilation (Program Translation).** Users typically write models using dynamic, imperative code (e.g., standard Python/PyTorch). The compiler's first job is to automatically parse and transform this arbitrary code into a static, well-defined, and compilable representation, namely, the Dataflow Graph.
2. **Graph-level compilation (Graph Transformation).** Once a static dataflow graph is generated, the compiler applies mathematical and structural transformations. The goal is to convert the original graph G into a mathematically equivalent but much faster graph G' (e.g., by pruning extra nodes or fusing operations together to reduce memory I/O).
3. **Operator-level compilation (Kernel Generation).** Finally, the compiler must automatically generate the actual hardware-specific kernel code for individual nodes in the graph. Instead of a human writing CUDA, the compiler generates the highly efficient code to execute a specific convolution or Matmul shape on the target hardware.

3.1 Market Drivers and Notable Compilers

Why is there such a massive demand for ML compilers? The answer lies in hardware fragmentation.

In the **cloud and data center market**, hardware is largely monopolized by NVIDIA GPUs. Because the hardware target is relatively uniform, it is economically viable for NVIDIA to hire expert engineers to hand-write perfectly optimized kernels (like cuDNN or cuBLAS). In this specific domain, it is very difficult for an automated compiler to beat an expert human.

However, the **edge computing market** (mobile phones, IoT devices, local laptops) is as big, if not bigger, than the cloud. This market is deeply fragmented across vendors like Apple, Qualcomm, and Huawei. It is economically infeasible to hire human experts to manually rewrite and tune kernels for every single edge chip. Automated ML compilers are the only scalable way to deploy efficient models across this diverse hardware landscape.

To address this, several notable compilers have emerged across the industry and academia:

- **XLA (Accelerated Linear Algebra):** Google's flagship compiler. Originally shipped alongside TensorFlow, XLA survived as a vital piece of infrastructure because it serves as the essential backend required to translate and execute operations on Google's TPUs.

- **TVM (Tensor Virtual Machine):** A pioneering academic project that helped formalize modern ML compiler research, specifically demonstrating the viability of automated operator-level compilation and search space exploration.
- **PyTorch 2.0 (TorchDynamo):** While PyTorch originated as a purely imperative, define-by-run framework, the 2.0 update introduced powerful just-in-time (JIT) compilation capabilities. This allowed developers to maintain the flexibility of Python while compiling the underlying dataflow graphs to strike for maximum hardware performance.
- **Modular:** A rising entity in the ML compilation space founded by Chris Lattner, the original creator of the universally adopted LLVM compiler infrastructure. Modular represents a push to bring rigorous, classical systems-compilation expertise directly into the machine learning ecosystem to build next-generation AI execution engines.

3.2 Design Trade-offs Across ML Compilers

Although these systems all fall under the umbrella of ML compilers, they differ significantly in design philosophy and practical trade-offs.

A useful way to compare them is along three axes:

(1) Static vs Dynamic Execution. Systems like XLA assume relatively static graphs, which enables aggressive whole-graph optimization. In contrast, PyTorch 2.0 (TorchDynamo) operates dynamically by extracting subgraphs from Python execution, allowing it to support control flow and dynamic shapes at the cost of some global optimization opportunities.

(2) Search-based vs Template-based Optimization. TVM emphasizes automated search over a large space of operator implementations, using learned cost models to guide optimization. In contrast, systems like TensorRT rely more heavily on predefined optimization templates and heuristics.

(3) Usability vs Peak Performance. Lower-level systems (e.g., CUDA) achieve the highest performance but require expert knowledge. Fully automated compilers improve usability but may miss specialized optimizations. Modern systems such as Triton and TorchDynamo aim to balance these competing goals.

These trade-offs highlight a key systems insight: *no single compiler design simultaneously optimizes flexibility, performance, and portability.* Instead, practical ML systems combine multiple approaches.

4 Operator Compilation

A concrete example discussed in the lecture was *loop splitting*. A simple loop can be transformed into nested loops that expose more structure for parallelism and locality. This kind of transformation is representative of how compilers create a search space of low-level program variants.

These challenges are compounded by the need to handle many operators across diverse hardware targets while keeping the search space tractable.

4.1 Operator Compilation as a Structured Search Problem

A useful way to formalize operator compilation is as a search problem over a space of equivalent programs. Starting from a high-level specification, the compiler generates many semantically equivalent implementations that differ in execution strategy.

These differences arise from transformations such as:

- Loop tiling and blocking (affecting memory locality)
- Loop reordering (affecting access patterns)
- Parallelization strategy (thread/block mapping)
- Memory placement (registers vs shared memory vs global memory)

Even for a simple operator, the number of possible implementations grows combinatorially with the number of transformation choices. As a result, exhaustive enumeration is infeasible.

This leads to a key abstraction:

$$\text{Operator Compilation} = \text{Search over Program Transformations}$$

The main challenge is therefore not just generating implementations, but exploring this space efficiently.

5 Search with Learned Cost Models

To avoid profiling every candidate implementation exhaustively, the lecture introduced the idea of using a learned cost model. Instead of evaluating each candidate directly on hardware, the compiler predicts the runtime or quality of a candidate using a learned model.

This changes the problem from pure brute-force search into a combination of generation, prediction, and selective evaluation. A search planner proposes promising implementations, and the learned cost model estimates which ones are likely to perform well. In practice, this reduces search cost and can make operator compilation much more scalable.

5.1 Why Learned Cost Models Are Necessary

A naive approach to operator compilation would benchmark every candidate implementation directly on hardware. However, this is prohibitively expensive due to the size of the search space.

Learned cost models address this by approximating the performance of a candidate without executing it. These models typically take as input features such as:

- Loop structure and tiling parameters
- Memory access patterns
- Hardware-specific characteristics

and output a predicted runtime or efficiency score.

Designing these features is itself non-trivial, as they must capture performance-relevant properties while remaining general across operators and hardware.

Key insight: the cost model does not need to be perfectly accurate. It only needs to rank candidates well enough to guide the search toward promising regions.

This results in a hybrid optimization loop:

1. Generate candidate implementations
2. Predict their performance using the cost model
3. Evaluate only the most promising candidates on hardware
4. Update the model based on measured results

This approach significantly reduces the cost of search while maintaining high-quality solutions.

The lecture also alluded to emerging directions where more expressive models, including large language models, may assist in generating or guiding program transformations, though this remains an active area of research.

Overall, this reframes compilation as a data-driven optimization problem rather than a purely rule-based process.

6 Triton as a Middle Ground

The lecture then presented Triton as a case study in operator compilation. Before describing Triton itself, the lecture briefly disambiguated the name: this is the open-source kernel DSL originated at OpenAI, not the UCSD mascot, and not NVIDIA's Triton Inference Server. Triton was positioned along a spectrum of programming models for GPUs:

Left end – device-specific DSLs (e.g., CUDA, Qualcomm DSL, Apple's Metal Shading Language, AMD ROCm). These languages are designed to expose every relevant feature of one specific hardware target. Their pros are clear: developers have access to low-level functionality and can squeeze out the last bit of performance, fully unleashing the underlying device. The costs, however, are equally pronounced:

- Steep learning curve. Two lectures of CUDA are typically only enough to write a single working kernel; reaching expert proficiency is realistically a one- or two-year effort.
- The strongest CUDA programmers are inside NVIDIA itself, which makes it hard for outside developers to match expert-level performance.
- Optimization is extremely time-consuming because every detail must be specified, and CUDA codebases are long, dense, and difficult to read compared to high-level languages like Python.

Right end – fully automatic ML compilers. These match the original ML-compiler premise: the user writes a program in a high-level language, hands the program plus the target hardware to a compiler, and receives back an efficient kernel. The pros are equally clear: very fast iteration and prototyping, since users can express ideas in familiar high-level constructs. But the cons are significant:

- Many algorithmic ideas cannot be cleanly represented because users must respect the compiler's operator and control-flow conventions.
- Custom data structures and unconventional memory layouts are difficult or impossible to express.
- Code generation itself is a deeply unsolved problem, and there is no guarantee that compiler-generated code will match the performance of expert-written kernels.

Triton sits between these two extremes. It is simpler to write than CUDA but more expressive than a graph compiler; it is necessarily harder than writing pure Python, but it gives developers fine-grained control over tiling and memory placement without the massive boilerplate of low-level thread-block code. The lecture's central point was not that Triton fully replaces either side, but that it occupies a practical sweet spot between programmability and performance.

7 Triton Programming Model

The Triton programming model is built on three concrete design choices that together define how kernels look and behave:

1. **Embedded directly in Python via the `@triton.jit` decorator.** Code beneath the decorator is parsed and compiled by Triton rather than executed as ordinary Python. From the developer's perspective, this means writing GPU kernels in a language that is already familiar to most ML practitioners.
2. **Heavy reliance on pointer arithmetic.** This is unusual for Python, where pointers are normally hidden, and is closer in spirit to C or C++. Triton exposes tensor pointers explicitly so that users can construct addresses by adding offsets to base pointers and then dereference them. This makes memory access patterns far more visible than in standard high-level tensor code.
3. **Shape constraints, in particular power-of-two dimensions.** Requiring power-of-two block sizes simplifies memory management and downstream optimizations such as vectorization and alignment.

A consequence of these choices is that a Triton kernel corresponds to a block-level unit of work. The programmer is responsible for specifying how data is partitioned across program instances, similar to how a CUDA programmer reasons about thread blocks. As a result, Triton exposes the structure of parallel execution more directly than a graph compiler does, while still remaining far more compact than CUDA: the low-level details of individual threads, register allocation, and per-SM scheduling are hidden behind the Python-flavored API.

8 Example: Elementwise Addition

Triton was introduced through a simple elementwise vector addition kernel,

$$z = x + y.$$

This example was used to demonstrate how GPU kernels can be written in a Python-like style while still exposing the core ideas of parallel execution and memory movement.

Version 1: single thread block. The first version maps the entire 1024-element operation to a single thread block (one SM). Walking through the body of the kernel:

- The kernel is marked with `@triton.jit` so that Triton, not the Python interpreter, parses the body.
- An array of offsets from 0 to 1023 is generated using `tl.arange(0, 1024)`, in a syntactic style that closely resembles ordinary Python.

- Pointer arithmetic constructs the actual GPU memory addresses by adding the offsets to the base pointers `x_ptr` and `y_ptr`. These pointers reference GPU memory directly, which is a departure from typical Python semantics.
- `tl.load` fetches all 1024 elements of x and y simultaneously into fast on-chip SRAM. By Triton convention, allocations default to SRAM, even when the syntax looks like ordinary Python.
- The addition $z = x + y$ is performed on values held in SRAM/registers.
- `tl.store` explicitly writes the 1024 results back to global memory (HBM). Writes to global memory must be explicit; there is no implicit flushing from SRAM.

Listing 1: Triton elementwise addition, single thread block

```
@triton.jit
def _add(z_ptr, x_ptr, y_ptr):
    offsets = tl.arange(0, 1024)
    x = tl.load(x_ptr + offsets)
    y = tl.load(y_ptr + offsets)
    z = x + y
    tl.store(z_ptr + offsets, z)
```

Version 2: scaling across many thread blocks. The single-block version is correct but cannot scale beyond one SM. To use the full GPU, the kernel must be parallelized across many thread blocks, similar to a CUDA grid launch. The launch site computes how many blocks are needed using ceiling division:

$$\text{grid} = \texttt{triton.cdiv(N, 1024)},$$

so that an arbitrary input size N, not necessarily a multiple of 1024, is fully covered. Inside the kernel, each block determines which slice of the array it owns by calling `tl.program_id(0)`: block 0 handles elements 0–1023, block 1 handles 1024–2047, and so on. Because the last block may overshoot the end of the array when N is not a multiple of the block size, a boundary mask `mask = offsets < N` is constructed and passed to both `tl.load` and `tl.store` so that out-of-bounds accesses are safely suppressed.

Listing 2: Triton elementwise addition with block indexing and masking

```
@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N):
    offsets = tl.arange(0, 1024)
    offsets += tl.program_id(0) * 1024
    x = tl.load(x_ptr + offsets, mask=offsets < N)
    y = tl.load(y_ptr + offsets, mask=offsets < N)
    z = x + y
    tl.store(z_ptr + offsets, z, mask=offsets < N)
```

This example is important because it shows that Triton keeps the decomposition of work explicit, similar to CUDA, while presenting a cleaner and more compact interface: the user writes Python-flavored code, while the low-level details of thread allocation and SM mapping are handled by the Triton API.

For this simple operation, Triton was shown to closely match native PyTorch performance, even though those baseline kernels are already highly optimized.

9 Example: Softmax in Triton

The next example considered softmax as a more realistic operator. Unlike vector addition, softmax requires exponentials, reductions, normalization, and row-wise execution over a matrix. While straightforward to express using high-level tensor operations, implementing it efficiently on GPUs is significantly harder.

If softmax is composed from many primitive operators, intermediate values may be repeatedly written to and read from global memory, and multiple kernels may need to be launched. A fused kernel can avoid much of this overhead.

In Triton, each program instance was assigned one row of the input matrix. The row was first loaded into SRAM, converted to FP32 for numerical stability, and shifted by subtracting the maximum value before exponentiation. The kernel then computed exponentials, summed them, divided each entry by the row sum, and stored the normalized result back to memory.

This illustrates the practical appeal of Triton: kernels that would be cumbersome in CUDA can be expressed more naturally while preserving control over memory placement and execution structure.

Listing 3: Triton softmax kernel sketch

```
@triton.jit
def _softmax(z_ptr, x_ptr, stride, N, BLOCK: tl.constexpr):
    row = tl.program_id(0)
    cols = tl.arange(0, BLOCK)
    x = tl.load(x_ptr + row * stride + cols,
                mask=cols < N,
                other=-float('inf'))
    x = x.to(tl.float32)
    x = x - tl.max(x, axis=0)
    num = tl.exp(x)
    den = tl.sum(num, axis=0)
    z = num / den
    tl.store(z_ptr + row * stride + cols, z, mask=cols < N)
```

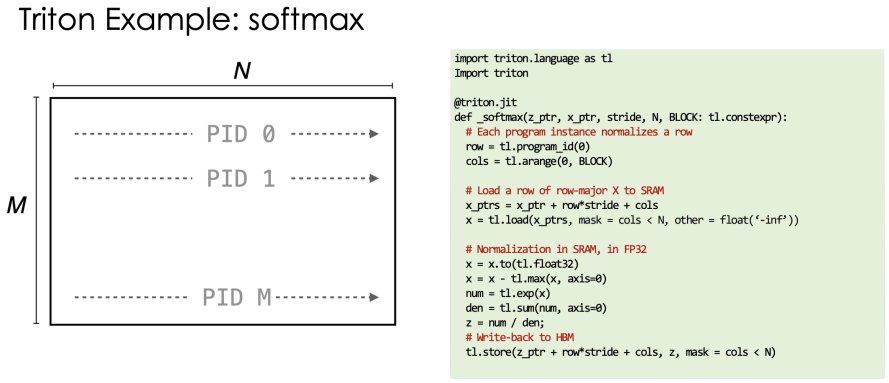

Figure 1: Row-wise softmax execution in Triton. Each program instance processes one row, performs normalization in SRAM, and writes the result back efficiently.

The fused Triton softmax kernel was shown to consistently outperform standard composed PyTorch implementations, with gains of roughly 25% in several settings as row sizes increased.

10 Why Triton Appears Effective

Triton appears effective because it occupies a strong middle ground between raw CUDA programming and fully automatic compiler systems.

At one extreme, low-level CUDA or assembly-style kernels can achieve the highest possible performance, but they require substantial expertise and engineering time. At the other extreme, fully automatic compiler approaches provide convenience, but often struggle to capture specialized optimizations for rapidly changing workloads.

Triton offers a more practical trade-off. It gives developers direct control over tiling, memory access, fusion, and parallel decomposition, while using a much more productive Python-like interface. In many realistic workloads, this allows efficient custom kernels to be developed quickly without the steep learning curve of CUDA.

More broadly, operator optimization relies on ideas such as vectorization, memory layout design, tiling for matrix multiplication, kernel fusion, and accelerator-aware parallelism. Triton was presented as a practical framework for applying these ideas without forcing optimization to be entirely manual or entirely automatic.

11 Graph Optimization

After discussing operator-level performance, the lecture shifted to graph optimization. The goal is to rewrite an original graph G into another graph G' such that G' computes equivalent outputs but executes more efficiently.

A straightforward strategy is to use hand-designed graph transformation templates. Human experts write rewrite rules for common subgraphs, and the compiler applies them by pattern matching. This approach is intuitive and can produce large performance gains in important cases.

11.1 Operator Fusion

The first graph optimization example was operator fusion. For example, a graph containing a matrix multiplication followed by layer normalization can be replaced with a fused operator. Fusion improves performance mainly by reducing intermediate I/O and decreasing kernel launch overhead.

At the same time, the lecture emphasized a practical downside: if one tries to create fused implementations for too many combinations of operators, the number of cases grows rapidly, and the resulting codebase becomes hard to maintain.

11.2 Mega-kernels

Instead of fusing a small subgraph, one approach attempts to collapse a much larger portion of the dataflow graph into a single kernel launch. This can further reduce intermediate memory traffic and launch overhead, but it also magnifies the main drawback of fusion. In practice, supporting many such fused combinations leads to specialized one-off kernels and quickly makes the system difficult to scale and maintain.

11.3 CUDA Graphs as a Practical Systems Optimization

A major practical issue in GPU execution is **kernel launch overhead**. The host CPU launches kernels asynchronously to the GPU, so it can continue issuing kernels while earlier ones execute. However, when kernels are very small, the fixed launch cost can become comparable to or larger than the kernel execution time itself. In that regime, performance is dominated by repeated launch latency.

To reduce this overhead, the lecture introduced **CUDA Graphs**. Instead of manually fusing many kernels into one custom operator, CUDA Graphs capture a sequence of GPU operations as a single executable graph. The programmer still writes code using ordinary high-level primitives, but the runtime records the overall execution pattern and launches the captured graph as a unit. This preserves modularity while reducing repeated launch overhead.

Thus, CUDA Graphs provide many of the benefits of fusion without requiring explicit fused operators such as `FusedABCDE`. The dependency order among kernels is preserved, but the launch cost is amortized across the full sequence. This makes CUDA Graphs especially useful in modern training and inference pipelines.

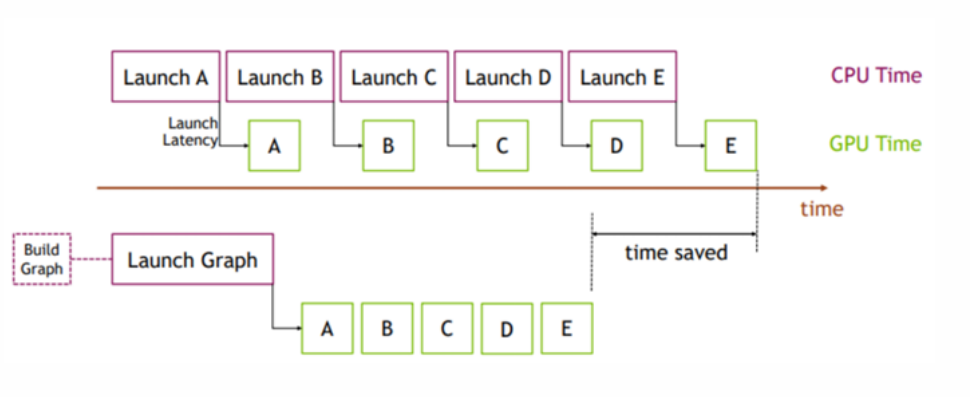

Figure 2: CUDA Graphs reduce repeated kernel launch overhead by capturing a sequence of GPU operations and launching them as a single graph.

11.4 Classical Graph Optimization Templates

Before moving to automatic graph transformation, the lecture reviewed several standard compiler techniques that simplify graphs and remove unnecessary work. One basic template is **constant folding**. If a graph contains operations involving constants known at compile time, those constants can be combined into a simpler equivalent expression. For example,

$$x + 3 + 4 \rightarrow x + 7$$

A related optimization is **identity elimination**, where operations that do not change the result are removed, such as

$$x \times 1 = x \qquad \text{and} \qquad x + 0 = x$$

These rewrites reduce graph size and simplify later optimization passes.

Another standard technique is **Common Subexpression Elimination (CSE)**. CSE detects when the same intermediate result is computed multiple times and replaces those duplicate computations with a single shared result. If two branches compute the same transformation, the compiler can compute it once and reuse it. This reduces redundant arithmetic and memory movement.

The lecture also reviewed **Dead Code Elimination (DCE)**. DCE removes nodes whose outputs are never used in producing the final result. Conceptually, the pass scans backward from the outputs and deletes computations that do not contribute to them. This also applies to unused branches in control-flow-style graph structures.

11.5 Greedy Graph Rewriting and Hardware Dependence

A common strategy is to apply these templates **greedily**: find a matching pattern, rewrite it, simplify the graph, and repeat. In many cases this works well, since each local simplification reduces computation, memory traffic, or kernel overhead.

However, graph optimization is also hardware-dependent. A transformation that improves runtime on one accelerator may reduce performance on another. The lecture emphasized that a rewritten graph could be faster on a V100 GPU but slower on a K80. Thus, semantic correctness alone is not enough; the performance impact of a rewrite depends on the target hardware.

The lecture also noted that useful optimization paths are not always monotonic. Some transformations may temporarily slow the graph down but expose later rewrites that lead to a much better final result.

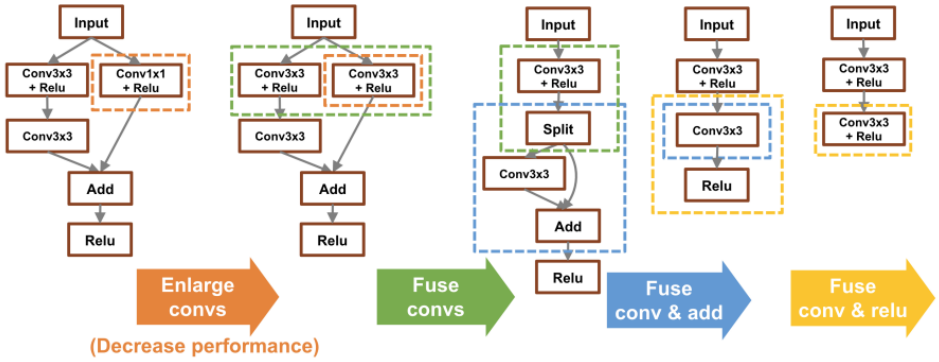

Figure 3: Greedy graph optimization applies a sequence of local rewrites, but the final performance gain may still depend strongly on the target hardware.

12 Limits of Template-Based Graph Optimization

Although template-based rewriting is useful, it does not scale well to modern machine learning workloads. The lecture described this as a combinatorial problem involving hundreds of operator types, thousands of possible graph structures, and multiple hardware backends. With this many combinations, it becomes infeasible to manually design graph optimization rules for every case.

Three limitations were highlighted. First, there is a **robustness** issue: expert-designed heuristics often work well only for specific networks or hardware targets. Second, there is a **scalability** issue: every new operator or architecture may require new rewrite rules. Third, there is a **performance** issue: handcrafted templates may miss optimizations that matter for particular model-hardware combinations.

13 Toward Automated Graph Transformation

To overcome the limits of manual templates, the lecture introduced **automated graph transformation**. The idea is to replace manually designed rewrites with a system that can generate candidate substitutions, verify correctness, and search over the verified rewrites to find faster implementations. Instead of relying only on expert-written rules, graph rewriting becomes a broader search problem over equivalent tensor-algebra expressions.

A representative example is the **TASO-style workflow**. The system begins with operator specifications, generates candidate graph substitutions, verifies semantic equivalence, and then uses the verified substitutions to optimize the original computation graph. The significance of this workflow is that it reduces dependence on manually curated rewrite libraries. Instead, the system can systematically enumerate and validate possible rewrites, which is especially useful when the search space is too large for human designers to manage directly.

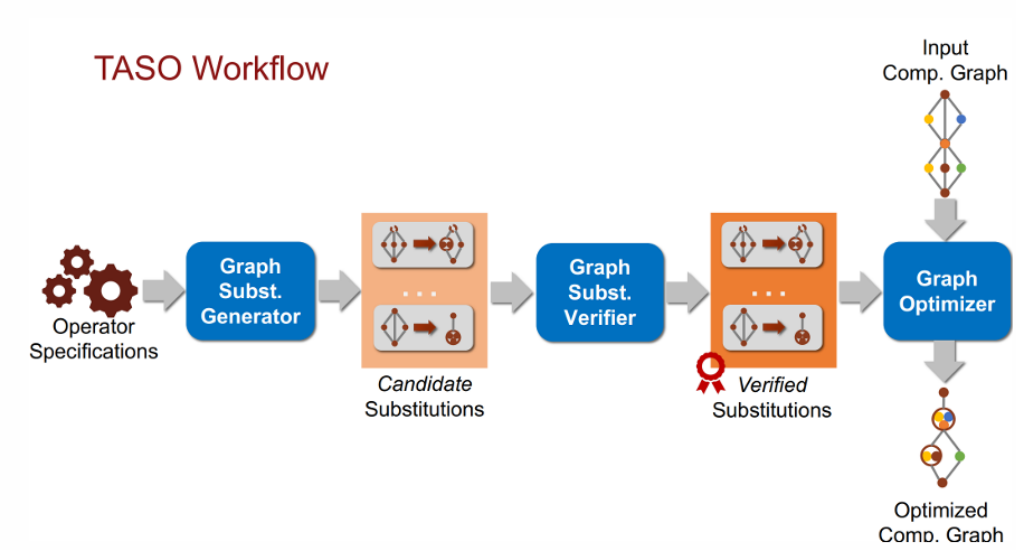

Figure 4: Automated graph transformation workflow: candidate substitutions are generated, verified, and then used by a graph optimizer to construct a more efficient computation graph.

The lecture also explained why this topic receives less emphasis in the current course offering. Earlier, graph-level compiler research was especially compelling because model architectures were highly diverse. Today, many practical workloads are concentrated around transformer-based language models and diffusion models. Since these workloads are more uniform, it is often more useful to focus on direct optimization techniques for these dominant architectures than on fully general graph-rewrite automation.

At the same time, this area remains relevant when architectural diversity increases again. Emerging ideas such as linear attention, loop transformers, and other transformer variants may create new graph structures that benefit from broader graph-optimization frameworks.

14 Retrospective and Big Picture

14.1 Overview and Motivation

The lecture closed with a retrospective on the ML compiler landscape, situating the day's techniques within a broader historical arc. The central question posed was: *why has the community largely shifted away from standalone ML compilers*, even after 500 or more compiler papers were produced in a concentrated period?

To appreciate the answer, it helps to understand how the field began and how each system addressed the limitations of its predecessors.

14.2 Origins: Halide (circa 2013)

The earliest relevant system was **Halide**, which dates to roughly 2013. Importantly, Halide was not originally designed for machine learning at all. Its domain was image processing and graphics rendering pipelines, which involve many of the same mathematical structures as ML operators: reductions, stencils, sliding windows, and element-wise maps. The key insight of Halide was to separate the *algorithm* (what to compute) from the *schedule* (how and when to compute it, including tiling and parallelism decisions). This separation turned out to be highly relevant to ML because the challenge of choosing tile sizes and loop orders for neural network operators is structurally identical to the challenge Halide was built to solve. In this sense, the mission to automate math-heavy operator compilation predates the deep learning era.

14.3 The 2016-2017 Wave: XLA, TensorRT, cuDNN, and ONNX

Around 2016-2017, the landscape changed rapidly as deep learning frameworks reached wide adoption. Google released TensorFlow and quickly faced the practical problem of compiling operators efficiently across CPUs, TPUs, and GPUs. To solve this, Google began investing heavily in **XLA (Accelerated Linear Algebra)**, their first major ML compiler. XLA was not entirely open-source in its early form, which made it difficult for the broader community to study or build upon. Nevertheless, its central purpose is automatically generating efficient kernel code for arbitrary TensorFlow operations on TPUs that established a clear template for what ML compilers should do.

During this same period, NVIDIA produced **TensorRT** and **cuDNN**, two libraries that are often discussed alongside compilers but operate at somewhat different levels of the stack:

- **cuDNN** provides highly hand-tuned implementations of deep learning primitives (convolutions, pooling, normalization) specifically optimized for NVIDIA GPUs. It is not a compiler in the traditional sense but rather a library of expert-crafted kernels.
- **TensorRT** is closer to a compiler in that it takes a trained neural network, applies graph-level optimizations (fusion, precision calibration), and produces a specialized inference engine. It is designed for inference deployment rather than training.

Both cuDNN and TensorRT were developed during the period when convolutional neural networks (CNNs) dominated the field. As a result, their performance profiles and internal optimizations were heavily shaped by CNN workloads, and they include many specialized operators dedicated to convolution variants.

ONNX (Open Neural Network Exchange) also emerged in this period. Rather than a compiler, ONNX is a template language and interchange format that allows models to be defined, exchanged, and

run across different frameworks and hardware backends. It addresses program- and graph-level portability rather than kernel-level performance, and it enabled a growing ecosystem of tools to operate on a shared model representation.

14.4 The 2018 Breakthrough: TVM

The year 2018 saw the publication of **TVM (Tensor Virtual Machine)**, which the lecture described as groundbreaking work. TVM's central premise was ambitious: given an arbitrary operator and an arbitrary hardware target, automatically compile efficient kernel code without requiring hand-written implementations. TVM introduced a formal notion of the search space over loop transformations, a learned cost model (the AutoTVM system) to navigate that space efficiently, and a hardware-agnostic intermediate representation that could target CPUs, GPUs, FPGAs, and specialized accelerators.

TVM's impact was significant for two reasons. First, it demonstrated that automated operator compilation could produce competitive performance across diverse hardware, not just on the NVIDIA GPUs for which cuDNN was optimized. Second, it helped formalize the research agenda around ML compilers by providing a clean framework for studying the search-schedule-codegen pipeline.

14.5 The 2019–2020 Period: MLIR and FlexFlow

By 2019–2020, the field was producing a large volume of compiler research. Two notable systems from this period are **MLIR** and **FlexFlow**.

MLIR (Multi-Level Intermediate Representation) was initiated by Google and later adopted by the LLVM community. Its goal was to establish a shared, extensible intermediate representation that could serve as the backbone for a wide variety of compilers and transformations. Rather than building yet another end-to-end ML compiler, MLIR provides the infrastructure: a dialect system that allows different levels of abstraction to coexist and interoperate, enabling teams to write compiler passes that work at the level of tensor algebra, hardware intrinsics, or anything in between. MLIR is best understood as an attempt to prevent the proliferation of incompatible IRs that had characterized the previous generation of compilers.

FlexFlow addressed graph-level compilation with a specific focus on parallelization strategies. Its key contribution was a framework for automatically discovering how to partition a computation graph across multiple devices, going beyond the standard data-parallelism and model-parallelism strategies that practitioners typically hand-tune.

14.6 The Present: Torch Dynamo and the Shift Away from Compilers

The most widely used compilation infrastructure today is **Torch 2.0 and its Dynamo frontend**, which the lecture described as having succeeded largely because of PyTorch's enormous user base rather than through a purely technical superiority over earlier compilers.

Torch Dynamo works by intercepting PyTorch's Python execution at the bytecode level, capturing subgraphs of operations dynamically, and lowering them through a compilation backend. This design makes it compatible with arbitrary PyTorch code, including code with Python control flow and dynamic shapes that earlier static-graph compilers could not handle.

The broader question the lecture posed - *why the community has shifted away from dedicated ML compilers* has several intertwined answers:

1. **Hardware consolidation in practice.** Despite the theoretical appeal of hardware-agnostic compilers, NVIDIA's dominance in training hardware means that cuDNN, cuBLAS, and more recently FlashAttention-style hand-tuned kernels continue to set the performance ceiling. Automated compilers often cannot match expert-written kernels on the most important workloads.

2. **The rapid pace of model architecture change.** When Transformers replaced CNNs as the dominant architecture, the fused operators and graph templates that had been carefully built for convolution-heavy models became less relevant. This highlighted a fundamental weakness of template-based approaches: they require sustained human investment to stay current as architectures evolve.

3. **Dynamic models and control flow.** Modern research code frequently uses dynamic shapes, conditional computation, and per-sample branching that are difficult or impossible to represent in a static dataflow graph. Systems like Torch Dynamo that work at the Python level sidestep this problem by compiling sub-graphs dynamically, accepting some overhead in exchange for generality.

4. **Framework inertia.** The ML community writes in PyTorch. Any compiler that requires users to rewrite their models or adopt a new programming paradigm faces adoption barriers that pure performance gains cannot always overcome. Torch 2.0's design where compilation is a drop-in addition requiring minimal code changes reflects a lesson learned from the more invasive frameworks that preceded it.

14.7 The Retrospective in Context

The arc traced by this timeline is one of increasing pragmatism: early systems sought to fully automate compilation in a hardware-agnostic way; later systems accepted more constraints in exchange for broader usability and compatibility with the dominant framework ecosystem. The current state of the art is a pragmatic combination of hand-tuned libraries for the most performance-critical operators, Triton-style DSLs for custom kernels, and dynamic compilation infrastructure like Torch Dynamo for everything else.

Finally, the lecture returned to the course's larger systems stack: dataflow graphs, autodiff, graph optimization, parallelization, operator optimization or compilation, and runtime support. This helped place the day's material in context and showed how operator-level and graph-level optimizations fit into the end-to-end deep learning systems pipeline.

15 Conclusion

Overall, this lecture connected low-level optimization ideas to compilation and graph rewriting for machine learning systems. The first half focused on operator compilation and Triton as a practical middle ground between CUDA and graph compilers. The second half examined graph optimization techniques such as fusion, CUDA Graphs, constant folding, common subexpression elimination, and dead code elimination. The lecture ended by emphasizing that while manual templates remain useful, scalable and hardware-aware automation is increasingly necessary in modern ML systems.