CSE-234: Data Systems for Machine Learning, Winter 2025

7: Triton, graph optimization and compilation

Lecturer: Hao Zhang
Scribe: Jiajun Xi, Nikita Johny Kachappilly, Wenting Yang, Chen Si, Qijun Li, Angting Cai, Pooja Pun, Siddharth Nahar, Yuting Yang, Yosen Lin, David Tran, Nishant Rajadhyaksha

1 Multiple Choice Question

MCQ 1) What is a "kernel" in the context of GPUs?
A. A specific section of the CPU used for memory operations.
B. A specific section of the GPU used for memory operations.
C. A type of thread that operates on the GPU.
D. A function that is executed simultaneously by tens of thousands of threads on GPU cores.

Answer: D. By definition, the kernel is a function that is executed simultaneously by many threads on GPU cores.

MCQ 2) What is the function of shared memory in the context of GPU execution?
A. It's HBM.
B. It's used to store all the threads in a block.
C. It can be used to "cache" data that is used by more than one thread, avoiding multiple reads from the global memory.
D. It's used to store all the CUDA cores.

Answer: C. HBM is the global memory. Shared memory (SRAM) is for storing data that can be shared by threads in the same thread block and saving time by avoiding reading from the global memory.

MCQ 3) What is the significance of over-subscribing the GPU?
A. It reduces the overall performance of the GPU.
B. It ensures that there are more blocks than SMPs present on the device, helping to hide latencies and ensure high occupancy of the GPU.
C. It leads to a memory overflow in the GPU.
D. It ensures that there are more SMPs than blocks present on the device.

Answer: B. Over-subscribing means we give enough blocks more than the number of streaming multiprocessors in order to make sure GPU is busy and fully utilized.

MCQ 4) Which of the following is True about GPU Memory?
A. On H100, a CPU process can access an array stored on H100 GPU memory.
B. A thread in a threadblock can access its threadblock-level shared memory.
C. Pinned memory is a part of memory allocated on GPU.
D. print(a) function in C++ can print an array allocated via a = cudaMalloc(..).

Answer: B. CPU cannot access GPU memory before data transfer, and that's also why print(a) C++

function cannot print array allocated by cudaMalloc. Pinned memory is allocated on CPU.

MCQ 5) Which of the following operations is most likely to be limited by arithmetic operations?
A. ReLU Activation
B. Linear layer (8192 outputs, 2048 inputs, batch size 1)
C. Batch normalization
D. Max pooling (3x3 window and unit stride)
E. Layer normalization
F. Linear layer (2048 out puts, 1024 inputs, batch size 512)

Answer: F. Operations like ReLU, Batch Normalization, Max Pooling and Layer Normalization are not arithmetic intensive. Comparing B and F, given that the Flops of matmul operation between matrix with shape (m, n) and (n, p) is $2mnp$, F has a higher Flops than B and thus is more likely to be limited by arithmetic operations.

MCQ 6) When picking a tile size for GEMM, why not always pick the biggest tile size?
A. The tile might not fit on the GPU HBM for some GEMM sizes.
B. The bigger size could result in low parallelism for some GEMM sizes.
C. Larger tiles have lower data reuse.
D. Larger tiles means more data is read, lowering arithmetic intensity.

Answer: B. HBM is pretty large and it's not contributing to matrix tiling. Larger tile means lower number of threads and that results in low parallelism. Larger tiles have more data reuse. By reusing more data, the arithmetic intensity is increased.

2 Triton Programming Model

2.1 Overview

The Triton Programming Model (TPM) strikes a balance between automation and performance, enabling users to manipulate arrays using pointers, even in Python. It allows the definition of tensors in SRAM and the embedding of program interfaces within Python, while also providing Torch primitives for intuitive computations.

However, TPM has certain limitations. Automation is only available to a limited extent, and array dimensions must be powers of two when allocating, partly due to backend optimizations.

2.2 Application on Elementwise Addition

2.2.1 Elementwise Addition Performance: Single Block

Programming Details
The Triton kernel is mapped to a single block of threads, simplifying CUDA programming by abstracting block-level concerns. Rather than focusing on shared memory management, Triton emphasizes how operations are executed at the thread level. Unlike Python, which treats arrays as high-level objects, Triton represents all arrays as pointers, enabling low-level memory access and efficient parallel execution. As shown in Figure 1, Triton provides a Pythonic API that allows different threads to fetch corresponding elements from memory and perform operations in parallel.

```
import triton.language as tl
Import triton

@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N):
  # same as torch.arrange
  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)
```

Figure 1: Elementwise Addition with one block.

Code Explanation
Within the `_add` function, all operations occur within a single block. The function first defines offsets, which act as shared memory, and these offsets are automatically mapped to 1,024 threads. Each pointer is then updated by adding its corresponding offset, ensuring that each thread processes a unique element. Triton guarantees that each thread within the block performs exactly one addition operation on its assigned offset.

Next, the function uses `tl.load` to load elements from memory through cooperative fetching, where each thread loads a specific element based on its offset. This ensures that each pointer correctly references its corresponding data. The computation is then performed in a vectorized fashion, allowing Triton to efficiently map the operation across multiple threads. Each thread is responsible for computing only its assigned element.

Finally, the computed result remains in the thread's register before being stored back into the Z pointer, completing the elementwise addition process efficiently. Outside of the function, the user decides the number of blocks.

2.2.2 Elementwise Addition Performance: Multiple Blocks

As shown in Figure 2, instead of using just one single block, we want to use multiple blocks to maximize GPU utilization, aiming to use as many SMs as possible. Here, we add a new line:

```
offsets += tl.program_id(0) * 1024
```

to separate each block. We also perform boundary checks using masks to avoid out-of-bound references. Specifically, we use:

```
x = tl.load(x_ptrs + offsets, mask=offsets < N)
```

```
import triton.language as tl
Import triton

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


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)
```

Figure 2: Elementwise Addition with multiple blocks.

This programming method is designed so that each thread processes one element, ensuring efficient parallelism. Grid size is calculated using:

```
grid = triton.cdiv(N, 1024)
```

dynamically allocating blocks based on data size. This method ensures complete data coverage.

2.3 Use cases

Single-block threads are suitable for small data sizes that fit within one block. On the other hand, the multi-block approach is more appropriate for large datasets, as it leverages multiple SMs for parallel execution.

3 Triton V/s PyTorch

3.1 Elementwise Addition Performance

PyTorch utilizes handcrafted CUDA kernels for its operators, requiring significant expertise in low-level CUDA programming to develop such optimized implementations. However, as shown in Figure 3, a Triton-based element-wise addition implementation can achieve performance comparable to PyTorch's specialized CUDA kernel for the element-wise addition operation.

Elementwise Add Performance

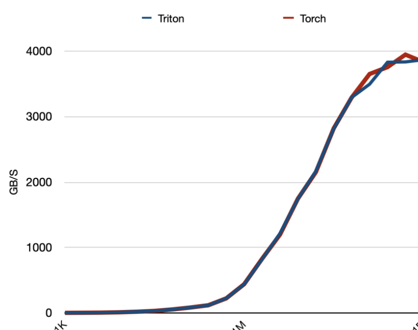


Figure 3: Performance comparison of Triton vs PyTorch on element-wise addition.

3.2 Softmax Performance

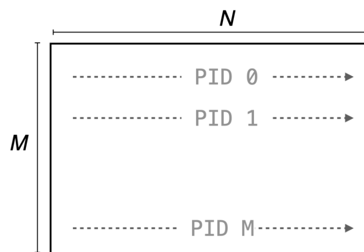
The Softmax function is defined as:

$$y_i = \text{softmax}(x)_i = \frac{e^{x_i}}{\sum e^{x_d}} \tag{1}$$

In common implementations of the Softmax function, such as those used in the programming assignment, we typically construct it using primitive operations like exponentiation and summation provided by PyTorch. However, this approach introduces significant overhead, as each primitive function call incurs separate I/O reads and memory accesses, leading to inefficiencies.

PyTorch optimizes Softmax by implementing it as an end-to-end CUDA kernel, where all the required operations (exponentiation, summation, and division) are fused into a single execution step, as shown in Equation 1. This fusion significantly reduces the number of I/O reads per primitive, improving computational efficiency and lowering memory access latency however is more complicated to write.

Triton Example: softmax




```
import triton.language as tl
Import triton

@triton.jit
def _softmax(z_ptr, x_ptr, stride, N, BLOCK: tl.constexpr):
  # Each program instance normalizes a row
  row = tl.program_id(0)
  cols = tl.arange(0, BLOCK)

  # Load a row of row-major X to SRAM
  x_ptrs = x_ptr + row*stride + cols
  x = tl.load(x_ptrs, mask = cols < N, other = float('-inf'))

  # Normalization in SRAM, in FP32
  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;
  # Write-back to HBM
  tl.store(z_ptr + row*stride + cols, z, mask = cols < N)
```

Figure 4: Sample Triton code for Softmax implementation.

As shown in Figure 4, Triton provides a more intuitive and Pythonic approach to GPU programming,

making it relatively easier to implement operations like Softmax compared to traditional CUDA. In this implementation, each block is responsible for handling an entire row of the matrix, while each thread processes a single column within that row.

We use pointers to determine the memory locations from which each thread should read. Instead of relying on separate primitive operations, which would introduce multiple memory read/write overheads, we utilize SRAM (shared memory) to store intermediate computations. All necessary operations (such as exponentiation, summation, and normalization) are performed in SRAM before writing the final results back to HBM (high-bandwidth memory). This approach significantly reduces I/O latency and enhances computational efficiency.

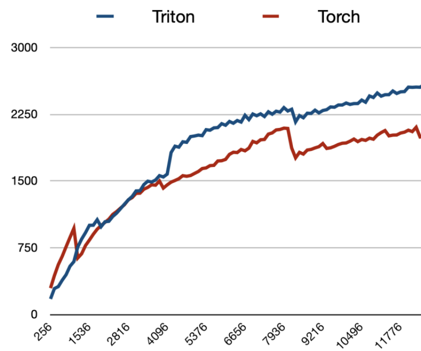


Figure 5: Performance comparison of Softmax implementation in Triton vs PyTorch. Triton's fused kernel optimizes memory access and reduces I/O overhead, leading to improved efficiency.

3.3 Softmax Performance: Triton vs PyTorch

As shown in Figure 5, the y-axis represents latency, while the x-axis denotes the dimensionality of the input.

The performance of the Triton-based Softmax implementation closely matches that of PyTorch's specialized CUDA kernel across certain dimensionality ranges. However, for higher-dimensional inputs, PyTorch outperforms Triton. Despite this, the performance gap is not significantly large, indicating that Triton's flexibility and Pythonic approach do not come at a major cost in performance.

This comparison highlights that while PyTorch's handcrafted CUDA kernel is slightly more optimized for larger dimensions, Triton remains a competitive alternative with a much simpler and more maintainable implementation.

3.4 Why Triton Succeeds

As illustrated in Figure 6, Triton's success can be attributed to its lower barrier to entry and efficient performance with minimal learning overhead.

Unlike CUDA, which requires a significant time investment to achieve high performance, Triton enables users to quickly write efficient GPU programs with a more accessible Pythonic interface. While CUDA can ultimately surpass Triton in performance with sufficient expertise, the learning curve is much steeper.

At the extreme end, SASS (Streaming Assembly)—the hardware-level language for GPUs—provides the best possible performance but demands extensive low-level optimization knowledge, making it impractical

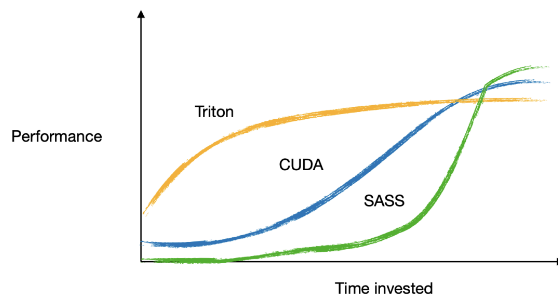


Figure 6: Comparison of performance vs. time investment for Triton, CUDA, and SASS (Streaming Assembly). Triton provides high performance with minimal development effort, whereas CUDA and SASS require more time investment to reach optimal performance.

for most developers. Triton strikes a balance, allowing users to get started quickly while still delivering competitive performance, making it a popular choice for many deep learning and HPC applications.

4 Template-based Graph Optimization

After discussing about **Operator Optimization/Compilation**, we will be discussing about **Graph Optimization**. While rewriting **G** to **G'**, our goal is to make:

- **G' run faster than G**
- **G' output equivalent results**

Template-based Graph Optimization is one of the most straightforward solution to do graph optimization. Engineers manually create (sub-)graph transformation(optimization) templates to guarantee both the **correctness** and the **performance gain**. Based on the templates, exhaustive pattern searching will be done over the dataflow graph and corresponding transformations will be applied.

4.1 Fusion

Fusion is a critical optimization technique used in the realm of compiler design, machine learning frameworks, and computational graph optimizations. In template-based graph optimization, it refers to the process of combining multiple adjacent operators(operations) within a dataflow graph into a single, more efficient operation.

Why Fusion improves performance?

- Reduce I/O

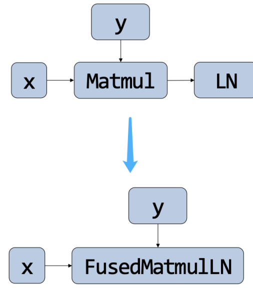


Figure 7: Fusion Example

- Reduce Kernel Launching

While fusion can provide substantial performance improvements, there are several potential **downsides** or challenges associated with this technique, particularly in complex systems:

- Requiring Many Fused Operations: FusedABCOp
- At some point, codebase become unmanageable

4.2 CUDA Graph

Considering the pros and cons of Fusion, one trade-off practice is "CUDA Graph". **CUDA Graphs** are

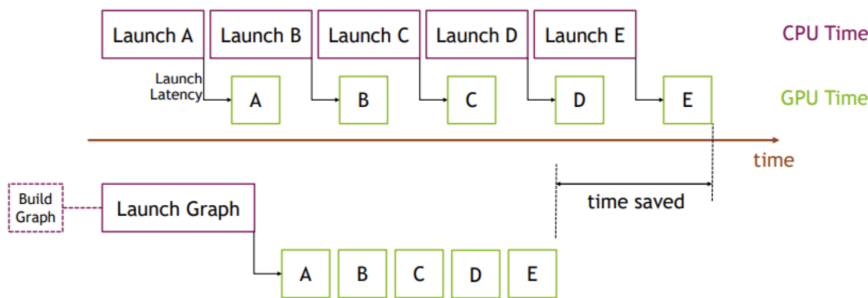


Figure 8: CUDA Graph Example

an advanced feature in NVIDIA's CUDA platform designed to optimize the execution of GPU-accelerated applications. By allowing users to program using **high-level API primitives** and capturing the computational graph at the **CUDA** level, CUDA Graphs provide a structured and efficient way to manage complex sequences of GPU operations, representing a **trade-off** in the realm of Fusion-based optimizations by balancing flexibility, complexity, and performance.
While Fusion focuses on merging operations to enhance efficiency at a granular level, CUDA Graphs optimize the execution flow of entire operation sequences. The decision to use CUDA Graphs, Fusion, or a combination of both depends on the specific requirements of your application, including the nature of the workload, the need for flexibility, and the desired performance characteristics.

4.3 Constant folding

Constant folding evaluates constant expressions at compile time rather than at runtime in order to reduce execution time. Constant folding can also take advantage of algebraic properties of operators such as addition and multiplication. Applying constant folding to a dataflow graph can reduce the number of operators or nodes.

Example. `(x+3)+4` is equivalent to `x+(3+4)` by associativity of addition. `3+4` is a constant expression, so applying constant folding gives `x+7` and eliminates one addition operation.

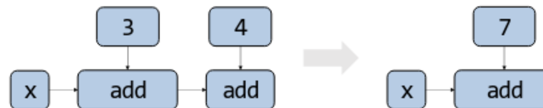


Figure 9: Constant folding on `(x+3)+4`

Example. `x*1` and `x+0`. After constant folding, these are simply `x`.

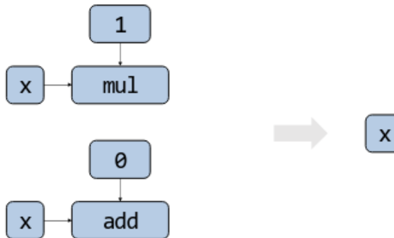


Figure 10: Constant folding on `x*1` and `x+0`

4.4 Common subexpression elimination

In common subexpression elimination (CSE), the compiler finds all occurrences of an expression and replaces it with a single instance holding the computed value. To detect and remove redundant subexpressions, we assign unique value numbers to variables and expressions known to have same value at compile time.

...	...	...
$c = a + b$	$c^3 = a^1 + b^2$	$c^3 = a^1 + b^2$
$d = a$	$d^1 = a^1$	~~$d^1 = a^1$~~
$e = b$	$e^2 = b^2$	$e^2 = b^2$
$f = d + e$	~~$f^3 = d^1 + e^2$~~	~~$f^3 = d^1 + e^2$~~
$d = x$	$f^3 = c^3$	$f^3 = c^3$
....	$d^4 = x^4$	$d^4 = x^4$
	....	....
	CSE hit	DCE hit

Figure 11: CSE and DCE example

Example. Starting from the beginning of the code above, the value numbers are $\{a\colon 1, b\colon 2, a+b\colon 3, c\colon 3, d\colon 1, e\colon 2, f\colon 3\}$. At this point, we have a CSE hit since f has the same value number as $a+b$ and c. To eliminate the redundant computation, we instead assign c to f.

4.5 Dead code elimination

Dead code elimination (DCE) removes code that is executed at runtime but whose result is never used again; for example, assignment to an unused variable. Removing dead code reduces application size and execution time.

Example. Continuing from the previous example, assume the common subexpression elimination pass has already been completed. When we reach the last line, d has value number 4. It is clear that there are no usages of d with value number 1 because d is overwritten before it is read. We have a DCE hit and remove the useless assignment to d on the second line.

Example. Another common template for CSE and DCE is an unreachable control flow branch. At compile time, if one direction of a conditional is never taken, the dataflow graph can be pruned. For example, if the false branch below is never used, we can remove the entire branch by traversing backward from the unused node.

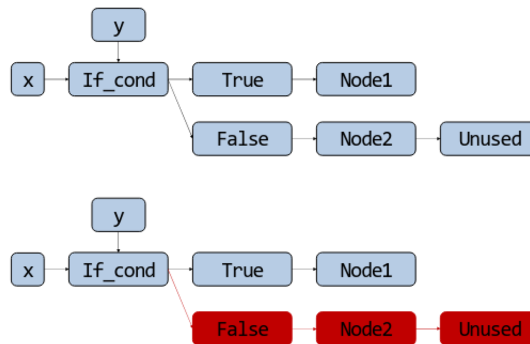


Figure 12: CSE and DCE with unused branch

4.6 Problems of Template-based Graph Optimizations

Problem 1: Greedy Template Matching Limits Performance Gains
In template-based graph optimization, the process is typically greedy, meaning that the graph is constantly replaced with more efficient templates as they are matched. However, this approach can miss opportunities for global optimization. For example, in some cases, applying a template might initially decrease performance (e.g., the example demonstrated in Figure 13 begins by replacing a conv1x1 with a conv3x3), but subsequent transformations could lead to a net performance gain. A greedy approach would never explore such paths because it only applies templates that immediately improve performance. This limitation arises because the search space is restricted to local optimizations, and the system cannot explore temporary performance regressions that might lead to better overall results. To achieve higher performance gains, a larger search space is needed, allowing the optimizer to explore more complex transformations that may not be immediately beneficial but could lead to significant improvements in the long run.

Problem 2: Scalability Issues Due to Variations in Graph Definitions
The second major problem with template-based graph optimization is its lack of scalability due to the

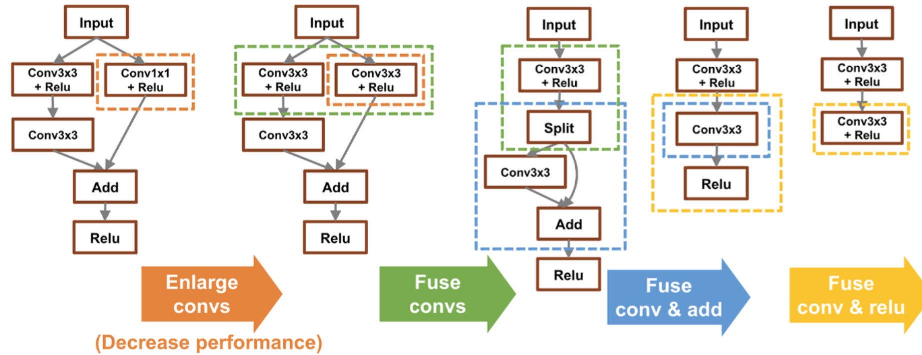


Figure 13: Example of a greedy template-based optimization failing to find a globally optimal transformation.

variations in graph definitions across different ML operators, and graph architectures, and hardwares. Each hardware backend (e.g., GPUs, TPUs) and each ML model architecture (e.g., ResNet, BERT) may require different optimization templates to achieve optimal performance. This means that for every combination of hardware, ML operators, and model architectures, human experts must manually design specific templates, and new operators and graph structures require more rules.

Given the vast number of combinations—approximately 300 ML operators, thousands of model architectures, and dozens of hardware backends — this approach becomes prohibitively labor-intensive. Companies would need to form dedicated teams to write and maintain these templates for every new model and hardware configuration, leading to high overhead and making the process unsustainable as the number of models and hardware platforms continues to grow. This manual effort is not scalable and cannot keep up with the rapid evolution of ML models and hardwares.

Problem 3: Lack of Robustness and Guaranteed Performance
In addition to the scalability problem, the third problem with template-based graph optimization is its lack of

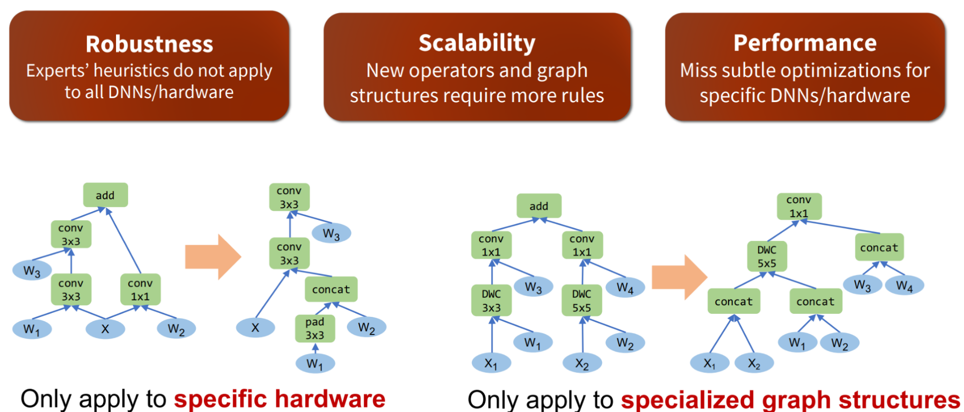


Figure 14: Problems of template-based graph optimizations

robustness and the inability to guarantee performance improvements in all scenarios. The greedy heuristics of template matching and rules used by human experts to design templates are often specific to certain hardware or model architectures and may not generalize well to other configurations. For example, as shown in Figure

14. As a result, performance gains from template-based optimizations are not guaranteed, as optimizations may miss subtle opportunities for performance improvements that are specific to certain DNNs or hardware backends. This lack of robustness and performance guarantees makes template-based optimization less reliable compared to automated approaches that can explore a wider range of transformations and verify their correctness and performance benefits.

5 Automate Graph Transformation

5.1 Introduction

The basic approach for graph optimization with template-based method has the following disadvantages:

- There are approximately 200-300 different ML operators that can exist in a dataflow graph. The organization of these operators, which is reflected by graph architectures, and different hardware backgrounds, pose challenges to manually create and optimize templates for all cases.
- Experts' heuristics do not apply to all DNNs and hardware settings.
- Adding new operators in the algorithm would need exponential amount of new templates to get created and designed.

Therefore, Automate Graph Transformation is introduced to automatically generate replacements for subgraphs.

5.2 TASO Workflow

Figure 15 demonstrates the workflow of Tensor Algebra SuperOptimizer (TASO) - a representative example of Automate Graph Transformation algorithm.

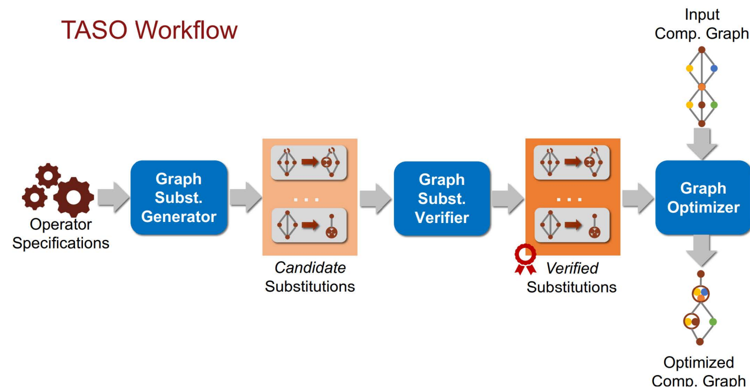


Figure 15: Workflow for TASO - a typical example of Automate Graph Transformation

5.2.1 Graph Substitution Generator

The first key component of TASO's workflow is the **graph substitution generator**, which provides a dictionary of equivalent ML operator subgraphs. The key idea for graph substitution is to find many set of

"equivalent subgraphs" that are easy to substitute like what we used to do in template-based approaches.

To automatically generate such a dictionary, we

1. Enumerate **all possible** graphs up to a fixed size, which are composed by the ML operators.
   For instance, we can generate approximately 66M such graphs with up to 4 ML operators, by various of graph compositions.
2. Evaluate which "pair" of graphs are equivalent – suitable for substitute.
   - Considering the mass amount of math proofs needed, it's impossible to mathematically prove whether two graphs are equivalent.
   - Instead, we give each of them the same random input and check if their outputs are the same.
     In the up-to-4-op example, we can filter out 28744 substitutions from the 66M candidate graphs.

5.2.2 Pruning Repeated Graphs

The Graph Substitution Generator produces 28,744 substitutions, which is a large number. Running pattern matching on this scale is not efficient and does not guarantee performance improvements. To address this, we prune the substitutions using simple mathematical rules. Examples include:

- **Variable Renaming**: As shown in Figure 16, two graphs are equivalent if they differ only in variable names.
- **Common Subgraph**: In Figure 16, the graphs differ only in the order of operations, A+(B×C) being equivalent to (B×C)+A.

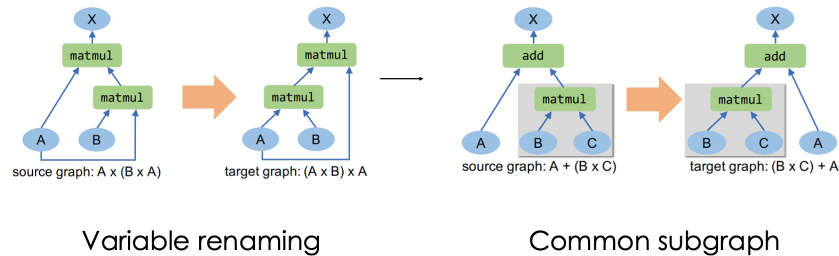


Figure 16: Examples of Pruning Graphs

By applying these rules, we reduce the search space to 743 substitutions in our running example.

5.2.3 Substitution Verifier

Among the pruned subset of substitutions, how many can we truly trust? As previously mentioned, we verify the outputs using a few random inputs. However, can we guarantee that these substitutions hold true for the entire input space?
Mathematically, if $f(a) = g(a) \& f(b) = g(b)$, does this imply $f(x) = g(x) \forall x \in X$?

To address this, we need a formal verification method. The steps to achieve this are as follows:

- **Define Specifications for Operators**: Write down the mathematical properties and characteristics for each operator. Intuitively, writing operator specifications is easier than manually performing optimizations. For example, for the convolution operator:
  - **P1**: Convolution is distributive over concatenation.
  - **P2**: Convolution is bilinear.
- **Use an Automated Theorem Prover**: Employ a compiler tool or automated theorem prover to determine whether two graphs are equivalent based on the mathematical specifications. Figure 17 illustrates how substitutions are verified.

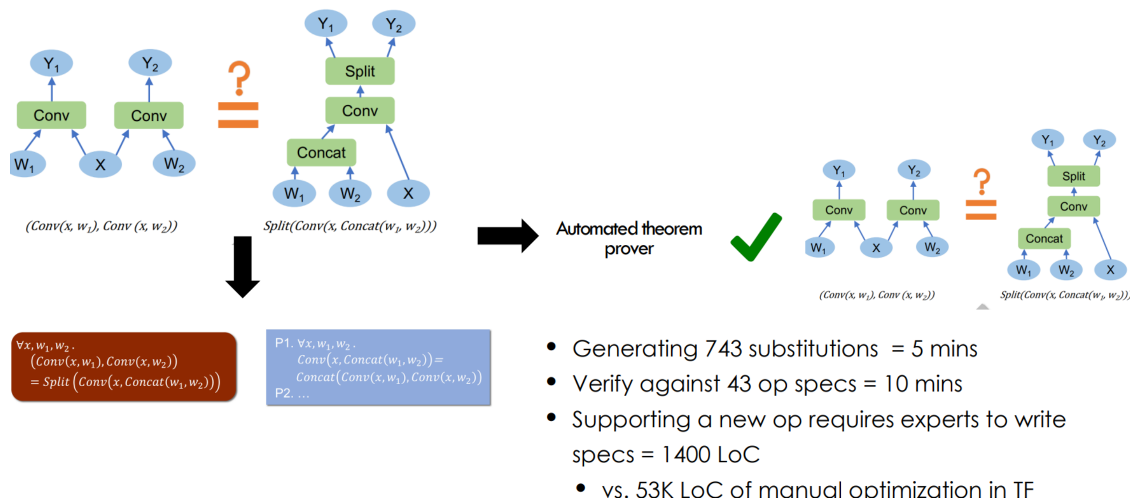


Figure 17: Verification of Graph Substitutions

Using these techniques, we can automate the graph optimization process. Generating 743 substitutions after pruning takes approximately 5 minutes, and verifying these substitutions against operator specifications takes about 10 minutes. This represents a significant improvement over manual optimization and the manual creation of substitutions.

5.2.4 Incorporating Substitutions

Once all possible verified substitutions have been identified, the next step is to apply them to create a more efficient computational graph. However, it is not immediately clear whether these substitutions will actually improve execution speed compared to the original graph. A practical way to solve this problem is through a trial-and-error approach. This involves selecting a verified substitution, matching patterns in the graph, and running the modified graph on a TPU to measure its performance. By recording these performance results, a 'Cost Model' can be built. This model estimates the total computational cost by adding up the costs of individual operations. Each operation's cost is measured on the target hardware, helping to guide the search for better substitutions. To efficiently optimize the graph, the process involves traversing the graph, applying substitutions, calculating the cost, and using backtracking to refine the choices. This entire process is automated, removing the need for manually creating templates and performing pattern matching.

5.3 Performance (as of 2019)

The graph below illustrates the performance differences when using automated graph transformation in models such as BERT, ResNet, and NasNet. The results show that this automated approach can outperform graph optimizations created by human experts. When comparing the performance of TASO with other methods, it is observed that TASO's workflow is even faster than expert-designed optimizations from TensorFlow XLA and TensorRT.

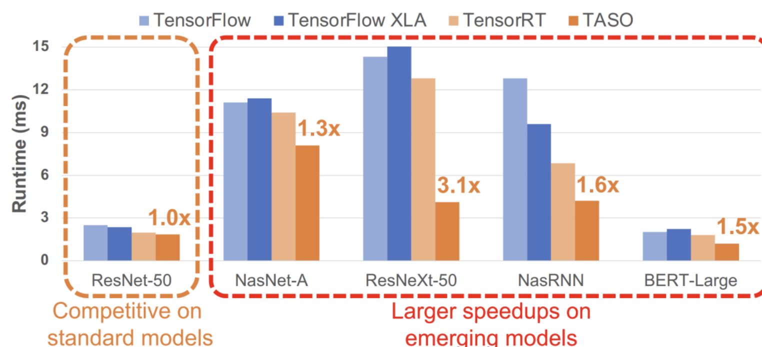


Figure 18: Performance plots (for year 2019)

5.4 Summary

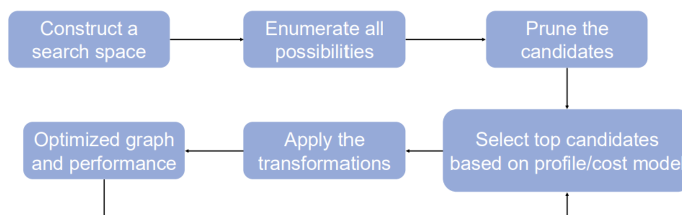


Figure 19: Workflow of Graph Optimizations

The basic workflow for graph optimization is outlined in the figure. First, a search space is created and all the possible combinations of operators are enumerated. Then, heuristics are used to filter out candidates that are either redundant or incorrect. Next, the best candidates are selected based on the Cost Model, and transformations are applied to generate an optimized graph. Once the graph is optimized and real performance data is collected, the Cost Model can be updated and refined. This creates a feedback loop that helps improve the accuracy of future optimizations.

5.4.1 Limitations

- The search space may not be large enough to include the best possible optimized graph. However, expanding the search space makes the process complex and time-consuming, creating a trade-off.
- The search process can be slow since it relies on trial and error method.
- Evaluating the resulting graph is expensive because real performance data is obtained after multiple iterations, which consumes GPU resources before improving the Cost Model.

5.5 Partially Equivalent Transformation Introduction

We have automated the task of graph optimization by searching in the search space of fully equivalent transformations. But are there any other possibilities for optimization?

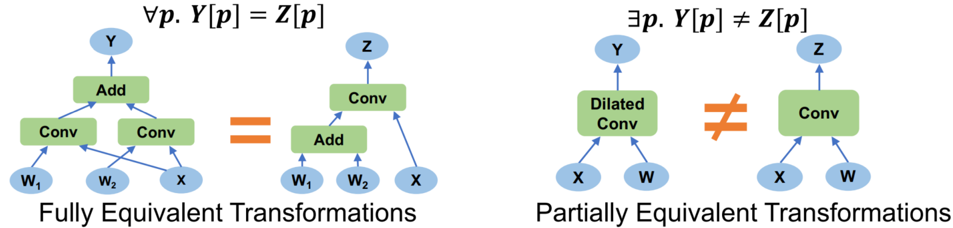


Figure 20: Illustrations of FET and PET.

Let's look into an example in Figure 20. We have two transformations: fully equivalent and partially equivalent. The fully equivalent transformation (FET) gives the same result as the original graph but it is missing some optimization opportunities. The partially equivalent transformation (PET) has better performance, but it gives a different result than the original graph. Is there a way for us to utilize the better performance of PETs while also preserving the correctness of the original graph? How about we exploit the larger space of PETs and then correct their results?

5.5.1 A Motivating example of *conv2d*

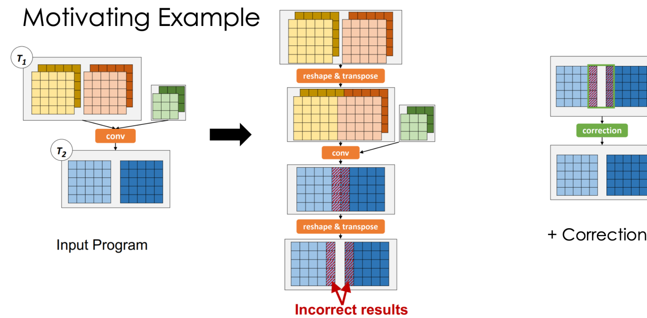


Figure 21: An example of PET on *conv2d* operation.

In Figure 21, we are applying *conv2d* operation on two different matrices using the same filter. Let's think of a partially equivalent transformation (PET) where we concatenate the input matrices and apply the *conv2d* operation on the concatenated matrix. Applying *conv* operation on a single matrix is 1.2 times faster than applying it on two matrices, since we are reducing kernel launches and memory I/O. But, as we can see from the figure, the resulting matrix from the PET has incorrect results in the boundary columns. So, to preserve the correctness, we need to perform correction on the PET results.

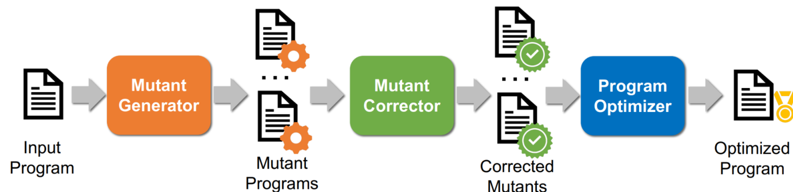


Figure 22: Partially Equivalent Transformations

5.6 Partially Equivalent Transformations Workflow

Figure 22 shows how we generate partially equivalent transformations and correct them. First, given an input program, we generate mutants, which are like substitutions but are not fully equivalent. We then apply these mutants to the original program to get a faster version and then check if the mutated program's results match the original program's result. If they do not match, we apply mutant corrector to correct the results. Since we have added additional operators in the graph to correct the results, it will possibly increase the entire graph size. We will then manually fuse operators together to optimize the performance.

The two important steps of this process are mutant generator and mutant corrector. So, how do we mutate a program? And how do we correct the results once we apply the mutation?

5.6.1 Mutant Generator

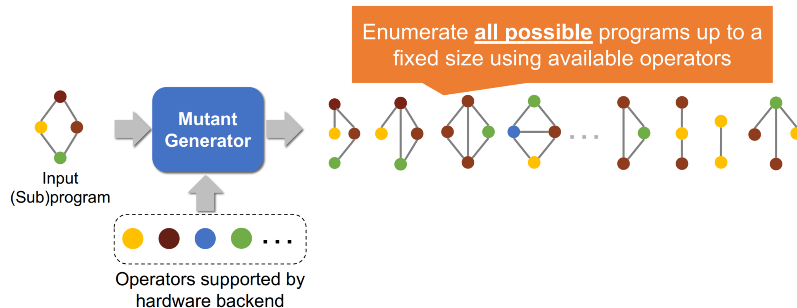


Figure 23: Mutant Generator

We generate mutant programs by enumerating all possible programs up to a fixed size using available operators. Unlike substitution generators, we are not going to stop at just mathematically equivalent programs, instead we are going to enumerate partially equivalent programs as well. To make the search space manageable, we are going to find transformations with equal shapes as our partially equivalent transformations.

5.6.2 Detection and Correction the mutation results

After mutation, if the results are equivalent, it turns out to be a fully equivalent transformations, we don't need correction. If it is partially equivalent transformations, we need to correct the results back.

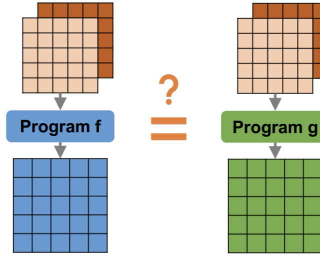


Figure 24: Comparing Whether results of f and g equal

Given the g is the mutant of f, one way to verify the results is to do the enumeration in which we check the results one by one and the time complexity is O(m x n), which is not efficient.

To make things faster, we try to reduce m and n.

5.6.3 Reducing dimension of n

We can reduce n of computational costs in neural networks by verifying the correctness of computations using only a small subset of outputs instead of evaluating the entire function. This approach works for approximately 80% of computations because neural network operations are largely multi-linear. A function is multi-linear if its output is linear with respect to each input. This means the function satisfies additivity ($f(..., X + Y, ...) = f(..., X, ...) + f(..., Y, ...)$) and scalar multiplication ($f(..., aX, ...) = a \cdot f(..., X, ...)$). By leveraging these properties, computations can be simplified, reducing the overall cost while maintaining accuracy.

Operator	Description
add	Element-wise addition
mul	Element-wise multiplication
conv	Convolution
groupconv	Grouped convolution
dilatedconv	Dilated convolution
batchnorm	Batch normalization
avgpool	Average pooling
matmul	Matrix multiplication
batchmatmul	Batch matrix multiplication
concat	Concatenate multiple tensors
split	Split a tensor into multiple tensors
transpose	Transpose a tensor's dimensions
reshape	Decouple/combine a tensor's dimensions

Figure 25: Important ML Operators are multi-linear

This table shows that multi-linear is very common e.g. matmul, conv in Machibe Learning Operations.

Theorem 1: For two multi-linear functions f and g, if $f = g$ for $O(1)$ positions in a region, then $f = g$ for all positions in the region.

This Theorem is important because we can now check a few positions to assert the results' equivalent. For example, when we apply conv operator, we only need to check one position in one region, which largely

reduce the computation costs from n to r (region number) and reduce $O(mn) \to O(mr)$, r (# regions) $\ll n$

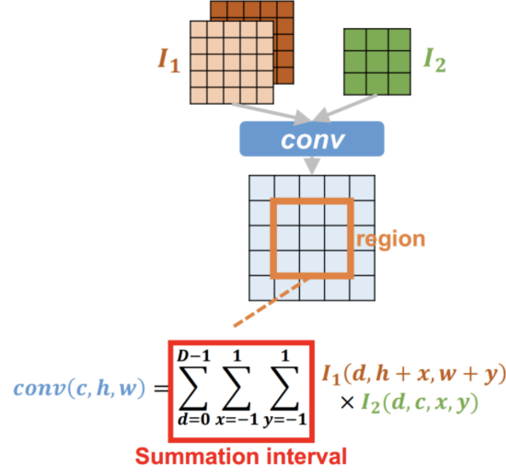


Figure 26: Conv Example in n reduction

5.6.4 Reducing dimension of m

When comparing $f(I)[p]$ and $g(I)[p]$, if there exists any input I such that $f(I)[p] \neq g(I)[p]$, we can conclude that f and g are inequivalent. However, enumerating all possible inputs m is computationally expensive, with a complexity of $O(mn)$. Thus, we aim to reduce m.

- **Key Idea**: Random testing relies on the assumption that if $f(I)[p] \neq g(I)[p]$, then the likelihood of failing to detect this inequivalence over t random inputs is extremely low.
- **Probability Analysis (Theorem 2)**:
  - Assume that $f(I)[p] \neq g(I)[p]$ occurs for some inputs I. For a single random input I, the probability of detecting $f(I)[p] \neq g(I)[p]$ is:

    $$P = 1 - \frac{1}{2^{31}}$$

    This stems from the fact that a 32-bit random input space has 2^{31} possibilities.
  - Over t random inputs, the probability of failing to detect inequivalence in all tests is:

    $$\left(\frac{1}{2^{31}}\right)^t$$

If t random tests pass without detecting inequivalence, the likelihood of f and g being inequivalent becomes negligibly small. Random sampling replaces exhaustive enumeration with a probabilistic guarantee of equivalence, dramatically reducing computational cost.

5.6.5 Complexity Analysis

- Initial complexity: $O(mn)$, where m is the number of possible inputs and n is the output size.

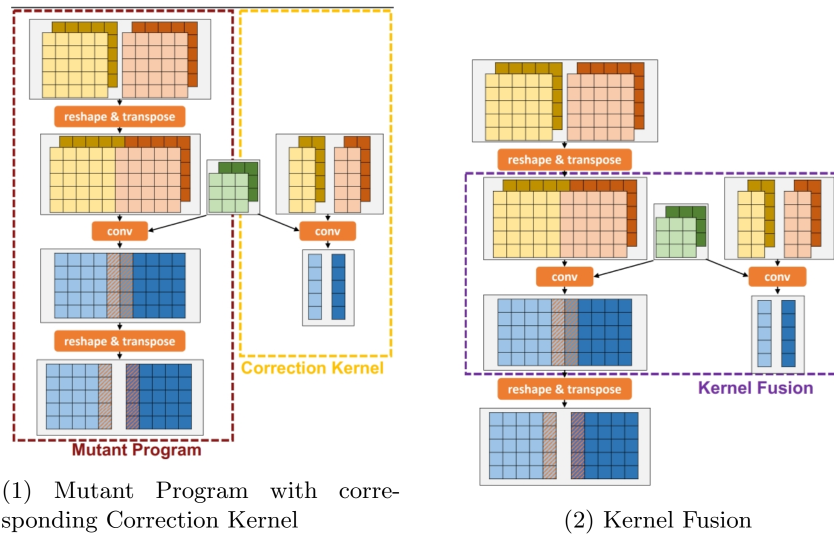
- Using random sampling: Reduce m to t, where t is the number of random tests.
- Final complexity: $O(tr)$, where $t \ll m$ and $r \ll n$ (with additional methods to reduce r).

5.6.6 Correct the Mutant

Step 1: Recompute Incorrect Outputs Using the Original Program

The first step identifies regions of the output that are incorrect and recalculates them using the original program, which serves as the ground truth.

- **Identification of Errors:** The mutant program optimizes certain operations but may introduce inaccuracies. By comparing the mutant program's outputs with those from the original program, we can pinpoint the erroneous regions.
- **Localized Correction:** Instead of re-executing the entire program, only the specific incorrect regions are recomputed, significantly reducing the computational cost.



(1) Mutant Program with corresponding Correction Kernel

(2) Kernel Fusion

Figure 27: Two steps of mutant correction.

Step 2: Opportunistically Fuse Correction Kernels with Other Operators

To further reduce the overhead introduced by the correction process, kernel fusion is applied. This technique integrates correction operations into existing program operations.

- **Reduced Data Movement:** By fusing the correction kernel with other operators, the need for intermediate data movement is minimized, leading to faster execution.
- **Parallel Efficiency:** Kernel fusion enables better utilization of hardware resources by executing correction operations alongside regular computations.
- **Performance Optimization:** The overall performance impact of error correction is mitigated, making the mutant program more efficient.

5.7 Overall Workflow Review

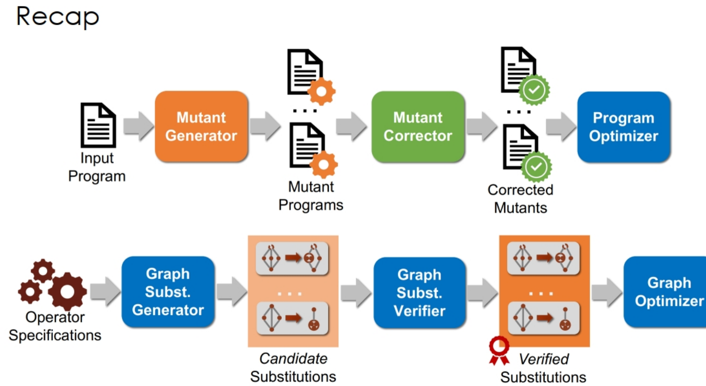


Figure 28: Recap of Automated Graph Transformation workflow.

Aspect	Partially Equivalent Transformations	Graph Substitution
Core Idea	Explores a larger optimization space by allowing partially equivalent transformations that are not strictly mathematically equivalent but may achieve higher performance.	Focuses on generating and applying mathematically equivalent substitution rules based on strict operator specifications.
Components	1. **Mutant Generator**: Enumerates all possible partially equivalent transformations using hardware-supported operators. 2. **Mutant Corrector**: Detects errors in mutant programs and corrects outputs with correction kernels and kernel fusion. 3. **Program Optimizer**: Further optimizes the corrected program for better performance.	1. **Graph Subst.Generator**: Generates substitution rules based on operator specifications and mathematical properties. 2. **Graph Subst.Verifier**: Ensures correctness of substitution rules via symbolic derivation and automated theorem proving. 3. **Graph Optimizer**: Applies validated substitution rules to optimize the computational graph.

Table 1: Comparison of Partially Equivalent Transformations and Graph Substitution

6 ML Compiler Retrospective

The compiler research is roughly started in 2013. Halide is one of the first compilers to conduct matrix tiling and operation compilation, though it is developed for graphics rendering instead of machine learning.

When machine learning started to take off, people paid more focus on optimizing machine learning workload by compilers, and XLA came out as the first ML compiler developed by Google during 2016 to 2017. However, XLA is very low-level and hard to understand by machine learning researchers. At the same time, other communities like TensorRT, cuDNN and ONNX are pushing at a different direction from automated compilation: they handcrafted fuse operations in templates and let users decide which one to use from the library.

In 2018, TVM came out as a famous open-source compiler. In 2019 and 2020, MLIR was developed by Google as a language (or a "layering" compiler) that builds machine learning intermediate representation

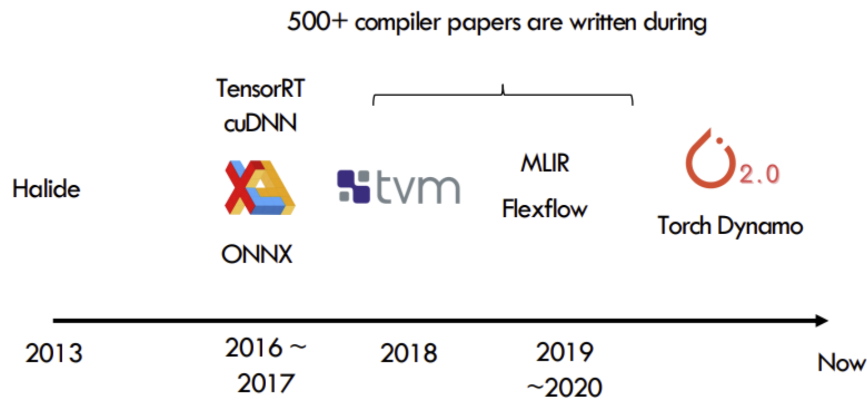


Figure 29: ML Compiler Retrospective.

in the compiler layer across different frameworks. Up to today, one of the most famous compilers is Torch Dynamo, which can be regarded as an intermediate representation for torch.

In the past several years starting from 2018, there are more than 500 compiler papers written, but nowadays the community is shifting away from the compiler. The professor proposes that one of the primary reasons is compilers are developed for automatic optimization on different models, operators and hardwares, under the assumption that new architectures and hardwares will keep emerging with high diversity; however, the transformer architecture dominates the model area and Nvidia GPU dominates the hardware area since 2020, such that people simply focus on how to optimize transformer on Nvidia GPU and there is no longer need for developing compilers to adapt to different models and hardwares.

7 Contributions

- Jiajun Xi: Section 1, 6; Overleaf formatting and editing
- Yosen Lin: Section 2
- Nishant Rajadhyaksha: Section 3
- Angting Cai: Section 4 introduction, 4.1, 4.2
- David Tran: Section 4.3, 4.4, 4.5
- Yuting Yang: Section 4.6
- Chen Si: Section 5.1, 5.2.1
- Siddharth Nahar: Section 5.2.2, 5.2.3
- Nikita Johny Kachappilly: Section 5.2.4, 5.3, 5.4, 5.4.1
- Pooja Pun: Section 5.5, 5.6 introduction, 5.6.1
- Wenting Yang: Section 5.6.2, 5.6.3
- Qijun Li: Section 5.6.4, 5.6.5, 5.6.6, 5.7