Reading Summary Week 4

Aakash Agrawal
HDSI
University of California San Diego
San Diego, CA, 92092
`aaa015@ucsd.edu`

1 Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations

Vendor libraries like cuBLAS and cuDNN are highly optimized for specific hardware but support only a limited range of operations. They lack portability and require significant manual effort, making them less useful when dealing with unsupported operations or targeting different hardware.

Other domain-specific languages (DSLs) based on loop synthesis or polyhedral machinery perform well only for a specific class of problems like depthwise-separable convolutions but are usually much slower than vendor libraries in practice and lack expressivity.

Triton addresses this by offering a DSL based on tiling, along with a compiler. It provides programmers with explicit control over memory management and parallel execution, facilitating the creation of performance-competitive, portable operator kernels. Triton leverages a C-like syntax and an LLVM-based Intermediate Representation (IR) to define tensor operations in terms of tile operations. This approach enables the application of various tile-level optimizations to compile these programs into efficient GPU code, matching the performance of hand-optimized vendor libraries while enhancing portability. However, these benefits come at the expense of increased programming efforts.

1.1 The Triton C language

Triton C is a language for writing tensor operations. It's easy for those who know CUDA or low-level GPU programming, and it's also friendly for people using deep-learning compilers. It's not optimized for low-level hardware. The language closely resembles CUDA but incorporates modifications to support operations with NumPy-like semantics using a Single Program Multiple Data (SPMD) approach. These modifications include:

- **Syntax Enhancements**:
  - Special syntax for declaring multi-dimensional arrays (tiles), distinct from traditional nested arrays.
  - Built-in functions such as `dot` and `trans` to facilitate tile operations, broadcasting, and other matrix computations.
- **Semantic Enhancements**:
  - A built-in tile type with operations designed to handle implicit broadcasting and conversion rules efficiently.
- **SPMD Programming Model**:
  - Each kernel in a Triton program is single-threaded at the source level and is automatically parallelized behind the scenes. This approach results in simpler kernels that do not require CUDA-like concurrency primitives such as shared-memory synchronization or inter-thread communication.

Benefits of Built-in Tile Semantics:

- Simplifies the structure of tensor programs by abstracting low-level performance details such as intra-tile memory coalescing and cache management.
- Enables compilers to automatically apply performance optimizations, improving efficiency and reducing manual tuning.

1.2 The Triton IR

Triton IR is an LLVM-based intermediate representation designed for tile-level program analysis, transformation, and optimization. It enables complex optimizations that are not directly achievable in C. Triton IR extends LLVM IR with features essential for tile-level data-flow and control-flow analysis.

- Triton-IR supports **tile-level data-flow analysis** with multi-dimensional tiles as the central data structure. It introduces reshape and broadcast instructions for efficient broadcasting and extends scalar operations (e.g., add, load) to work on tiles.
- Support for **tile-level control-flow analysis**: A challenge in Triton-IR is the inability to express divergent control flow within tiles; this is mainly because tile elements cannot be accessed individually. Triton-IR enables control flow within tiles without relying on traditional branching using two new instruction types:
  - `cmpp`: Returns two opposite predicates for comparisons.
  - `psi`: Merges instructions from different predicated streams.

1.3 The Triton-JIT compiler

The Triton-JIT compiler compiles and optimizes Triton-IR programs into efficient machine-level code. This process includes a combination of tile-level machine-independent optimizations, machine-dependent optimizations, and an auto-tuner.

1.3.1 Machine Independent Passes

These include generic code improvements/optimizations that are independent of any hardware. They focus on improvements that enhance efficiency across different architectures. These include:

- **Pre-Fetching**: Tile-level memory accesses inside loops can incur major latency overheads. Triton-JIT compiler detects loops and predicts which memory location will be accessed next and preloads data when necessary to reduce stalls.
- **Peephole Optimizations** at the tile level: A peephole optimizer is a technique that looks at small sequences of instructions and replaces them with more efficient ones in order to reduce redundant computations, simplify expressions, and improve performance. Example: simplifying transposition chains: $(X^T)^T = X$.

1.3.2 Machine Dependent Passes

These include hardware-specific code optimizations, such as:

- **Hierarchial Tiling**: Hierarchical tiling breaks down tiles into smaller micro- and nano-tiles to better utilize a machine's compute capabilities and memory hierarchy.
- **Memory Coalescing**: Triton-JIT order threads within each micro tile behind the scenes, which helps avoid un-coalesced memory accesses. Since memory is usually accessed in large chunks from DRAM, this strategy reduces the number of memory transactions required to load the tile column.
- **Shared Memory Allocation**: This pass enhances the performance of tile operations with high arithmetic intensity (AI) by temporarily storing operands in shared memory, reducing redundant memory accesses.
- **Shared Memory Synchronization**: This pass ensures program correctness by automatically inserting barriers in GPU source code. It detects read-after-write (RAW) and write-after-read (WAR) hazards using forward data-flow analysis.

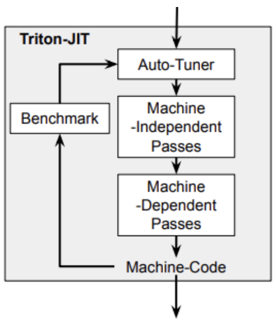

Figure 1: Overview of Triton-JIT

1.3.3 Auto-Tuner

Traditional auto-tuners use pre-written templates to optimize performance for specific tasks. Triton-JIT, on the other hand, automatically adjusts parameters like tile size and memory layout to improve performance by analyzing Triton-IR programs directly. Triton differs from **TVM** by enabling the **automatic inference of execution schedules**, whereas TVM requires them to be manually specified (as detailed in Section 2).

1.4 Evaluation

The numerical experiments section evaluates performance by comparing Triton to vendor libraries like cuBLAS and cuDNN, as well as compiler frameworks such as auto-TVM and PlaidML.

- **MatMul**: Triton matches cuBLAS performance in most tasks and achieves over 90% of the device's peak performance in some cases. It is also $2 - 3\times$ faster than other existing DSLs.
- **Convolutions**: In the case of Dense Convolutions, Triton outperforms cuDNN implementation of IMPLICIT GEMM for ResNet. For Shifted Convolutions, Triton's implementation of a fused shift-conv module outperforms cuBLAS's hand-written shift kernel.

2 TVM: An Automated End-to-End Optimizing Compiler for Deep Learning

TVM (Tensor Virtual Machine) is an ML compiler that provides graph-level and operator-level optimizations to improve the performance and portability of deep learning workloads across diverse hardware backends. A key innovation of TVM is its automated operator optimization, which leverages learning-based **cost models** to efficiently explore optimization strategies and select the best-performing implementations.

As discussed in Triton, highly engineered, vendor-specific operator libraries (like cuBLAS) require significant manual tuning, making them highly specialized and difficult to port across diverse hardware platforms. This forces deep learning workloads to either avoid graph optimization that yields operators not supported by vendors or use suboptimal implementations of these operators.

A large optimization search space and the need for specialized hardware accelerators that offload most scheduling and optimization to the compiler stack present significant challenges in compiler design. Let's explore how TVM addresses these challenges. There are four key modules in the system design of TVM:

1. Optimizing Computational Graphs
2. Generating Tensor Operations
3. Hardware-Aware Optimization Primitives
4. Automated Optimization

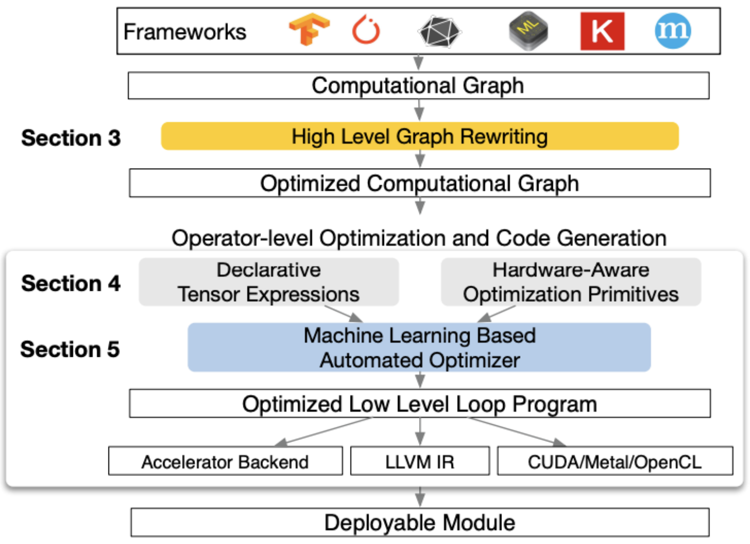
Frameworks
Computational Graph
Section 3
High Level Graph Rewriting
Optimized Computational Graph
Operator-level Optimization and Code Generation
Section 4
Declarative Tensor Expressions
Hardware-Aware Optimization Primitives
Section 5
Machine Learning Based Automated Optimizer
Optimized Low Level Loop Program
Accelerator Backend
LLVM IR
CUDA/Metal/OpenCL
Deployable Module

Figure 2: System Overview of TVM

2.1 Optimizing Computational Graphs

TVM takes a model from frameworks like TensorFlow, MXNet, or PyTorch and converts it into a computational graph. It then applies high-level dataflow rewrites to generate an optimized graph, leveraging the graph representation to perform various optimizations. These graph-level optimizations include:

- **Operator Fusion**, which fuses multiple small operations together, greatly reduces the execution time;
- **Constant Folding**, which pre-computes graph parts that can be determined statically, saving execution costs;
- **Static Memory Planning Pass**, which pre-allocates memory to hold each intermediate tensor; and
- **Data Layout Transformations**, which converts a computational graph into one that can use better internal data layouts for execution on the target hardware.

These graph-level optimizations are limited by the effectiveness of the operators they implement. TVM proposes a code generation approach that can produce various implementations for the operators in a given model.

2.2 Generating Tensor Operations

TVM generates efficient code for each operator by creating multiple valid implementations for different hardware backends and selecting the best one. It follows **Halide**'s compute and schedule separation concept (separating what needs to be computed from where and how it needs to be scheduled in hardware), making it easier to optimize for multiple hardware platforms at once.

The code generation process uses a tensor expression language that supports common arithmetic and math operations but does not define execution details or loop structures. This flexibility allows for hardware-specific optimizations across different backends.

2.3 Hardware-Aware Optimization Primitives

Schedule denotes a mapping from a tensor expression to a low-level code that defines how the computation should be executed in the hardware. It consists of fundamental transformations, known as **schedule primitives**, which restructure the computation while preserving its logical equivalence.

2.3.1 Nested Parallelism with Cooperation

GPUs use nested parallelism to speed up deep learning tasks by breaking them into smaller parallel subtasks. Traditionally, each thread works independently without accessing its peers' data (shared-nothing parallelism). A more efficient approach is **cooperative fetching**, where threads work together to load shared data into memory, reducing redundant memory accesses and improving performance.

2.3.2 Tensorization

Deep learning workloads rely on tensor operations like matrix multiplications and convolutions. To speed these up, modern hardware uses tensor compute primitives, similar to SIMD vectorization but designed for multidimensional data. Since different accelerators have unique tensor instructions, a flexible approach is needed. TVM solves this by **separating hardware intrinsics from scheduling**, allowing developers to easily define and integrate new tensor operations, making it adaptable to future hardware.

2.3.3 Explicit Memory Latency Hiding

Latency hiding overlaps memory operations with computation, ensuring the processor stays productive while waiting for data. On CPUs, this is achieved through simultaneous multi-threading or pre-fetching, whereas GPUs rely on rapid context switching of many warps of threads. Specialized accelerators like TPUs use a **decoupled access-execute** (DAE) architecture that decouples memory access from computing.

In Decoupled Access-Execute (DAE) architectures, synchronizing the instruction stream is crucial for maximizing compute utilization. However, manually handling low-level synchronization is complex. TVM simplifies this with **virtual thread** lowering, which converts a high-level parallel program into a single instruction stream. This stream includes explicit synchronization cues that help the hardware maintain pipeline parallelism and efficiently hide memory access latency.

2.4 Automated Optimization

With a rich set of schedule primitives, the challenge is to find the optimal operator implementation for each model layer. This involves choosing the best loop order, memory optimizations, tiling size, and loop unrolling factor, creating a large search space. The automated schedule optimizer consists of two key components:

- *Schedule Explorer* – Generates promising new schedule configurations.
- *Cost Model* – Predicts the performance of each configuration to guide optimization.

Developers can also optionally bake their domain-specific knowledge when specifying possible schedules using a schedule template specification API.

2.4.1 ML-based Cost Model

There are several approaches for identifying the best schedule.

1. **Auto-tuning**: Auto-tuning explores the entire search space of configurations to find the best one. However, it requires numerous experiments to identify an optimal configuration, making it computationally expensive.
2. **Using a pre-defined cost model**: This helps narrow the search space for a specific hardware backend, avoiding the need to test all possibilities. However, this method lacks generality, as a new cost model must be created for each different hardware.
3. **Statistical Approach using a schedule explorer**: This approach strikes a balance between auto-tuning and predefined cost modeling by proposing configurations that may improve an operator's performance. This is achieved using an ML model that takes a lowered loop program as input and predicts a relative order of runtime costs for different configurations.

The choice of ML model to be used by the explorer depends on two key factors: **speed** and **quality**. For a ML model to be useful, its inference and refitting time should be smaller than the time it takes to measure the performance.

The authors compare **XGBoost** and **TreeRNN** for predictive quality and find them similar. However, XGBoost is twice as fast in prediction and significantly faster to train. Consequently, they choose XGBoost as the default cost model for their experiments.

Once a cost model is chosen, schedule exploration uses it to pick promising configurations for testing. In each iteration, the explorer selects candidates based on the model's predictions, tests them, and updates the model with the results.

To efficiently explore large search spaces, **simulated annealing** is used. The explorer starts with random configurations, adjusts them step by step, and accepts changes that lower the predicted cost. The process continues until it finds the best configurations, with exploration continuing after each update.

2.5 Performance Evaluation

The authors provide an end-to-end evaluation of TVM based on real-world deep learning workloads (ResNet and MobileNet) on four different types of platforms:

1. **Server-Class GPU Evaluation**: TVM outperforms other baselines like MXNet, Tensorflow(v1.1), and TensorFlow XLA with speedups ranging from 1.6x to 3.8x due to both joint graph optimization and operator optimization. TVM is also able to find faster kernels as compared to MXNet's handcrafted kernels.
2. **Embedded CPU Evaluation**: TVM generates operators that outperform the hand-optimized TFLite versions for both ResNet and MobileNet workloads.
3. **Embedded GPU Evaluation**: The baseline used was the ARM Compute Library (v18.03). TVM outperformed the baseline on three models for both float16 and float32, with speedups ranging from 1.2x to 1.6x.
4. **FPGA Accelerator Evaluation**: The authors performed an end-to-end evaluation on a ResNet workload in a FPGA-based accelerator and found 40x speedups for convolutional layers. This experiment highlights TVMs' ability to adapt to new architecture and the intrinsic hardware they expose.