Week 4 Reading Summaries: TVM + Triton Readings

Amogh Patankar
University of California, San Diego
3869 Miramar Street Box #3037, La Jolla, CA- 92092
apatankar@ucsd.edu

Abstract

The articles for this week's reading summary describe two compilers custom-made for deep learning and neural network computations- TVM, and Triton. TVM is a compiler that allows for graph and operator level optimizations for deep learning algorithms to improve performance. The result is a high performant system that is competitive with hand-tuned libraries for low-power CPU, GPU, and server class GPUs. Similarly, Triton is a language, and a compiler, built around the concept of tiling. Triton is a C-based language for tensor expression on tiles, and tile-level optimizations for exxecution of efficient GPU code.

1 TVM

1.1 Introduction

With a wider and wider range of deep learning models in the present, and with the advent of generative AI, deep learning frameworks are important in order to optimize workload execution. However, these frameworks cannot handle graph level optimizations due to the high-level nature of those programs. Moreover, operator optimizations require manual tuning, which is not scalable. TVM addresses key challenges, and solves them; namely, by 1) leveraging specific hardware features and abstractions, and 2) Creating a large search space for optimization. Deep learning accelerators require specialized hardware to efficiently run workloads, and systems in turn need to generate code that controls pipeline dependencies. With respect to manual tuning, there *must* be a large enough search space to come up with the correct combinatorial choices of memory access, threading pattern, and other HW primitives.

This paper addresses these challenges with tensor expression language, an automatic program optimization framework to find optimized tensor operators, and a graph rewriter to take advantage of high/operator level optimizations.

1.2 Overview

TVM converts a model from an existing framework into a computational graph, then performs high-level dataflow rewrites to produce an optimized graph. Fused operators in the graph are then efficiently coded using a declarative tensor expression language, and still leaves execution details unspecified. TVM uses a ML based cost model to identify the best operator in order to navigate the vast space of possible optimizations for a given hardware target. Only then is the code packaged into a deployable module for execution.

Submitted to 38th Conference on Neural Information Processing Systems (NeurIPS 2024). Do not distribute.

1.3 Optimizing Computational Graphs

Computational graphs in DL are representative of programs using multi-dim tensors, allowing for high-level optimizations without operator implementations. TVM uses graphs for optimizations like oeprator fusion, and more, to improve efficiency.

1.3.1 Operator Fusion

Operator fusion combines multiple operators into one kernel, hence not requiring intermediate results to be stored in memory, significantly reduceing execution time on GPUs and accelerators. Operators can be injective (1-to-1 maps), reduction (sum), complex-out-fusable (conv2d with element-wise output fusion), and opaque (non-fusable, sort). Injective operators fuse into another injective operator, reduction operators fuse with input injective operators, and complex-out-fusable operators like conv2d fuse with element-wise operations. So, computational graphs are transformed into an optimized, fused version.

1.3.2 Data Layout Transformation

Data layout optimization is a method TVM uses to use efficient internal data layouts tailored to hardware like column/row major, or tiling for accelerators by transforming a computational graph. TVM specifies preferred layouts for operators and transforms data layouts between producers and consumers when mismatches occur. High-level optimizations like operator fusion improve efficiency; but, the effectiveness of said optimizations depend on operator library support. Customizing fused kernels for various operators, data layouts + types, and hardware backends isn't sustainable, and so, a code generation approach is taken. By generating diverse operator implementations and enabling scalable and efficient support, the problem can be solved for workloads across multiple hardware targets.

1.4 Generating Tensor Operations

TVM produces efficient code for operators by generating implementations, and choosing one. Here are the TVM-specific features:

1.4.1 Tensor Expression and Schedule Space

TVM uses tensor expression, namely through a tensor expression language, to enable automatic code generation. This process occurs by describing tensor operations through index formulas, with output shapes and element-wise computations without defining loops or execution details. TVM uses schedules to map expressions to code, and applies transformations for hardware optimization. It extends existing primitives and introduces new ones for GPUs and accelerators that support CPUs, GPUs, and TPU-like hardware. TVM tracks loop structures to generate efficient code, with automatic schedule derivation discussed later.

1.4.2 Nested Parallelism with Cooperation

In order to optimize compute-intensive DL workloads, parallelism is very important especially on GPUs with massive parallelism. Traditionally, the model is a fork-join model for nested parallelism where tasks recursively get divided to exploit thread hierarchies, but threads can't share data within the same stage. TVM improves this by supporting cooperative data fetching; threads can collaboratively load data into shared memory, allowing reuse and optimizing GPU performance. Memory scopes are introduced to mark compute stages as shared, ensuring that synchronization is done properly. The approach benefits GPUs and aids in targeting accelerators by tagging memory buffers and creating custom rules.

1.4.3 Tensorization

TVM introduces tensorization, which is the optimization of DL workloads by incorporating specialized tensor compute primitives, such as matmuls and 1D convolution. TVM tries to improve performance and requires a flexible compilation framework to accommodate the various tensor instructions across emerging accelerators. TVM creates an extensible approach by separating hardware-specific

instructions from the compilation schedule through a tensor-intrinsic declaration mechanism. This allows TVM to easily support new hardware architectures. TVM also creates a tensorized schedule primitive to map computations to hardware intrinsics, which can result in significant performance gains, such as a 1.5× speedup when using a low-precision micro-kernel for mobile CPUs.

1.4.4 Explicit Memory Latency Hiding

Latency hiding is the idea of overlapping memory ops with computation, in order to maximize memory/compute utilization. CPUs use multithreading and prefetching, while GPUs and specialized DL accelerators use DAE or threadwarps. Programming DAE accelerators requires synchorinization; so, TVM uses a virtual threading scheduling primitive. TVM starts with a high level multi-threaded program schedule, then inserts synchronization ops for correct execution. In hiding latency, the performance of algos improves for ResNet layers.

1.5 Automating Optimization

1.5.1 Schedule Space Specification

TVM utilizes schedule template specification to declare schedule options, incorporating domain-specific knowledge when needed. This unlocks automated exploration of a vast search space of configurations, and the schedule optimizer searches through billions of potential configurations, guided by the template and the tensor expression language.

1.5.2 ML-Based Cost Model

TVM utilizes a machine learning model to predict the performance of different schedule configurations as opposed to exhaustive auto-tuning. The model that is trained with real-time measurement data, helps reduce the search space. The focus on relative runtime predictions rather than absolute execution times helps to improve accuracy over time. TVM balances the need for quality predictions and fast execution, and uses various methods like gradient tree boosting (XGBoost) for efficient predictions.

1.5.3 Schedule Exploration

Using the cost model, the schedule explorer selects promising configurations, employing parallel simulated annealing algorithm to iteratively find better configurations. TVM can then search more efficiently by rejecting configurations with higher predicted costs and converging on lower-cost configurations. The exploration process is continuous and updates as new data from experiments becomes available.

1.5.4 Distributed Device Pool and RPC

TVM uses a distributed device pool and RPC framework to scale hardware trials, then managing optimization jobs across multiple devices. It automates compilation, execution, and profiling processes, especially useful for embedded devices, where tuning and cross-compilation are hard and inefficient.

1.6 Evaluation

TVM evaluates their framework, noting that it optimizes DL system SW-HW codesign as compared to other options.

2 Triton

2.1 Introduction

Deep Neural Networks (DNNs) and generative AI has now changed AI accelerators and HW, through parallel computing devices like GPUs, and vendor libraries like cuBLAS/cuDNN. These libraries are limited in supporting tensor operations, which has led to the development of DSLs, and pairing them with microkernels and high level tiling abstractions. Triton is a system comprising a C-like language for tensor programs, LLVM-based Intermediate Representation for tile-level optimization and a compiler for efficient GPU code generation.

2.2 Related Work

Triton has a few different approaches to optimizing deep learning computations. Existing frameworks rely on hand-selected and optimized libraries like cuBLAS, some use domain-specific languages (DSLs). Pre-existing Tensor-level IRs use predefined templates, the polyhedral model automates compilation, and loop synthesizers enable manual schedule optimization. Triton integrates tile-level operations, offers greater flexibility, and automatic schedule inference.

2.3 Triton-C

2.3.1 Syntax

Triton-C uses ANSI-C and CUDA-C, but has a few cahnges for semantics and programing model. Primarily, special syntax for tile declarations, built in functions (eg. dot, trans, etc.), and broadcasting using the "newaxis" keyword. Some additional changes include the "" prefix for control flow.

2.3.2 Semantics

- Tile: Tile semantics simplifies tensor programs by hiding performance details regarding memory, cache, and HW utilization.
- Broadcasting: Provides rules to perform padding and broadcasting conversions from scalar to array implicitly.

2.3.3 Programming Model

Triton follows the SPMD programming model, just like many other support libraries and frameworks; however, it makes each kernel single-threaded and parallelized. Triton assigns each kernel with a set of global ranges that vary. This simplifies kernel execution by eliminating concurrency primitives.

2.4 Triton-IR

2.4.1 Structure

- Modules: Each unit of compilation is called a module, compiling independently and linked by a linker.
- Functions: Consist of return type, name, and arguments list if required. Function and parameter attributes can be specified.
- Basic Blocks: Straight line code sqeuences that may contain terminator instructions.

2.4.2 Support for Tile-Level Dataflow Analysis

- Types: Multi-dim types are declared using similar syntax to vectors in LLVM-IR.
- Instructions: These instructions are used to retile, and support broadcasting semantics.

2.4.3 Support for Tile-Level Control-Flow Analysis

Triton-IR doesn't have native support for divergent control flow in tiles; using Predicated SSA (PSSA) form and ψ -functions, adding cmpp for dual predicates and psi for merging predicated instruction streams.

2.5 Triton-JIT Compiler

2.5.1 Machine-Independent Passes

- Prefetching: Tile level memeory ops are avoided by detecting loops and adding prefetching code.
- Tile-level Peephole Optimization: Algebraic properties introduced as tiles are used as operations.

2.5.2 Machine-Dependent Passes

- Hierarchical Tiling: TritonIR allows automatic enumeration and optimizatino of nested tiling without polyhedral machinery.
- Memory Coalescing: TritonIR's backend can order threads internally within microtiles.
- Shared Memory Allocation: Tile level operations store operands in fast shared memory, by calculating the live range.
- Shared Memory Synchronization: R/W from and to shared memory are asynchronous; RAW and WAR hazards are detected then barriers are inserted.

2.5.3 Autotuner

As compared to hand-written parameterized code templates, Triton-JIT extracts spaces from Triton-IR by concatenating meta-parameters.

2.6 Experiments

In all experiments (MatMul, Convolutions, etc.), Triton is on-par with vendor librarise like cuBLAS, cuDNN, etc, in some cases even more performant.

2.7 Conclusions

Triton's language and compiler are effective to compiling tiled-NN computations into machine code. Triton-IR also enables optimization passes, and Triton-C is a useful language to implement efficient kernels.