Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions

Introduction:

Tensor Comprehensions is a C++ library and mathematical language that helps in bridging the gap between researchers, who communicate in terms of mathematical operations, and engineers focusing on the practical needs of running large-scale models on various hardware backends. The main differentiating feature of Tensor Comprehensions is that it represents a unique take on Just-In-Time compilation to produce the high-performance codes that the machine learning community needs, automatically and on-demand.

Purpose and application:

Tensor Comprehensions (TC) is a fully-functional C++ library to *automatically* synthesize high-performance machine learning kernels using Halide ISL and NVRTC or LLVM. TC additionally provides basic integration with Caffe2 and PyTorch. This library is designed to be highly portable, machine-learning-framework agnostic and only requires a simple tensor library with memory allocation, offloading and synchronization capabilities.

Tensor Comprehension(TC) is a notation based on generalized Einstein notation for computing on multi-dimensional arrays. TC greatly simplifies ML framework implementations by providing a concise and powerful syntax which can be efficiently translated to high-performance computation kernels, automatically.

Know nothing about GPU programming? Still write high-performance deep learning using TC.

With Tensor Comprehensions, their vision is for researchers to write ones idea out in mathematical notation, this notation automatically gets compiled and tuned by the system, and the result is specialized code with good performance.

In the paper , they provide:

- a mathematical notation to express a broad family of ML ideas in a simple syntax
- a C++ frontend for this mathematical notation based on Halide IR
- a polyhedral Just-in-Time (JIT) compiler based on Integer Set Library (ISL)
- a multi-threaded, multi-GPU autotuner based on evolutionary search

Backend:

Tensor Comprehensions uses the Halide compiler as a library. It is build on Halide's intermediate representation (IR) and analysis tools, and it is paired with polyhedral compilation techniques, so that we can write layers using similar high-level syntax but without the need to explicitly say how it is going to run. There are also ways to make language even more concise by eliminating the need to specify loop bounds for reductions.

Tensor Comprehensions use **Halide** and **Polyhedral Compilation** techniques to automatically synthesize CUDA kernels with delegated memory management and synchronization. This translation performs optimizations for general operator fusion, fast local memory, fast reductions and JIT specialization for specific sizes. Since they dont try to own or optimize memory management, their flow is easily and efficiently integrated into any ML framework and any language that allows calling C++ functions.

Contrary to classical compiler technology and library approaches, Polyhedral Compilation allows Tensor Comprehensions to schedule computation of individual tensor elements on-demand for each new network. At the CUDA level, it combines affine loop transformations, fusion/fission and automatic parallelization while ensuring data is correctly moved through the memory hierarchy.

To drive the search procedure, they also provide an integrated multi-threaded, multi-GPU autotuning library which uses **Evolutionary Search** to generate and evaluate thousands of implementation alternatives and select the best performing ones. Just call the **tune** function on the Tensor Comprehension and watch the performance improve, live; stop when one is satisfied. The best strategy is serialized via protobuf and reusable immediately or in offline scenarios.

support the DSL is given:

Tensor Comprehensions (TC) are a notation for computing on multi-dimensional arrays that borrows from the Einstein notation (a.k.a. summation convention):

1. index variables are defined implicitly by using them in an expression and their range is inferred from what they index;

2. indices that appear on the right of an expression but not on the left are assumed to be reduction dimensions;

3. the evaluation order of points in the iteration space does not affect the output

Importantly, the nesting order (`i` then `k`) is arbitrary: the semantics of a tensor comprehension is always invariant by loop permutation. TC allows in-place updates, but preserves a functional semantics that is atomic on full tensors: the semantics is to read RHS expressions in full before assigning any element on the LHS. This specification is

important in case the LHS tensor also occurs in RHS [30]: the compiler is responsible for checking the causality of in-place updates on element-wise dependences. They chose to reuse the `out` tensor across all comprehensions, indicating the absence of temporary storage. Iteration variables are always non-negative. Unless specified otherwise in a `where` clause, each iteration variable is assumed to start at 0. Inference is computed from input arguments to output tensors, setting up a constraint-based analysis problem across all the affine array accesses in a TC function.

Integration:

TC backend is agnostic to frameworks and is designed in a way such that the integration with any ML framework can be easy. TC backend is based on DLPack tensors which is very lightweight header library for describing the tensor. A DLPack tensor is a simple struct which has information like data pointer, tensor size, tensor strides, tensor type etc. DLPack doesn't do any memory allocations and rather provides the meta information about the tensor. Hence converting a tensor for example torch tensor to DLPack tensor doesn't involve any copies and is very cheap. Further, since TC itself doesn't do any data allocations, it infers the output tensor shapes for a given TC and input sizes. The output tensors shapes are sent back after the compilation and the framework needs to allocate storage for output tensors using that information.

Additionally, they provide a simple "identity" polyhedral mapping option to generate a naive, readable,CUDA reference implementation that may be run and checked for correctness on a single GPU thread and shake off simple problems.they are planning a simple LLVM JIT compilation pass to make that reference implementation more generally usable on a CPU, but this is not yet implemented. Lastly, They also provide a path to emit a series of library calls, a useful fallback to default implementations backed by CUDNN.