CS3423: Compilers - II

Mini-Assignment #2 - Summary of Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions

Soumi Chakraborty

ES19BTECH11017

Abstract

This paper highlights the problem that occasionally, existing operators are not enough to aptly deal with new computation requirements arising out of research problems. Existing operators are often not capable of computing the required data in the most efficient manner, which acts as a bottleneck for any following research. Each computation problem might require a custom operator to get the best and the most optimised results. The paper then proposes the following as solutions:

- A language called Tensor Comprehensions (TC)
- A polyhedral Just-In-Time (JIT) compiler to convert deep learning directed acyclic graphs (DAGs) to a CUDA kernel with various kinds of optimisations
- A compilation cache populated by an autotuner

1 Introduction

Neural networks expressed as graphs of tensor operations have recently been solved by GPU-accelerated algorithms which use back-propagation. Such frameworks have low latency and power usage since they are implemented using hardware accelerators, and are suitable for many applications. However, there are some custom frameworks where computation graphs are unable to effectively use the hardware accelerators unless they use a pre-defined set of library functions.

1.1 Motivation

Building an operator for each use case is both not feasible and very time-consuming. The idea of an active library that generates the most optimised code as needed sounds like a fantastic solution, however, the process would involve going through a huge number of combinations to generate the best possible operator. This is an incredibly time-consuming and infeasible solution.

Domain-specific languages like Halide solve some of the issues but it’s rather complicated for non-experts to use effortlessly.

Deep learning compilers like XLA and Latte optimise across operators and data size but are still not up to par and lack in areas like complex scheduling and mapping transformations.

Thus, the solution must optimise deep parallelism, memory hierarchies, hardware features, and offer abstraction without regret.

1.2 Contributions

To offer abstraction without regret, TC is designed in a way such that its concise syntax is good at managing memory and mapping to parallel platforms. To optimise deep parallelism, memory hierarchies and hardware features, a polyhedra framework is specialised towards deep learning with a dedicated autotuner. The polyhedral framework was the best choice since it improved the performance in multiple ways.

The paper presents the following contributions:

- A concise and expressive high-level language to write ML tensor computations in Einstein notation
- A compilation flow that lowers tensor expressions to efficient GPU code
- Specialising polyhedral intermediate representation and compilation algorithms to the domain of deep learning through kernel fusion and multi-level parallelism
- An autotuning framework using JIT compilation
- And a language that can be easily integrated into ML frameworks like PyTorch

2 Related Work

Active libraries, which are code generators, have tried to solve the issue of parallelising and optimising compilation by using feedback-directed optimization. They try to choose the best combination for the compiler based on the output they get. A couple of examples are ATLAS and BTO. However, such libraries are often not very feasible as explained above.

This is where domain-specific languages (DSL) are required. DSL compilers are capable of producing code that is highly optimised for each use case because they use domain-specific constructs, which is precisely what the paper is offering a solution to. Halide, TACO, Simit, and OoLaLa are some such DSL compilers that have different applications; for example, Halide is for image processing whereas OoLaLa is for linear algebra.

TC is a DSL compiler that aims to be general and not constrained to a specific use-case, by making use of the research conducted on loop-nest optimisation and parallelisation.

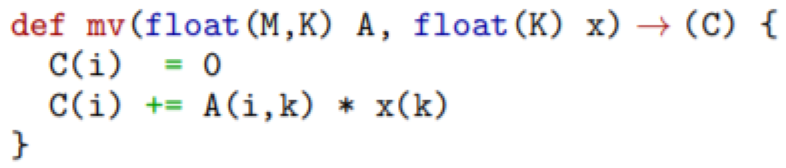
TC is also not an embedded DSL and only uses C++ to implement optimisation.

3 Tensor Comprehensions

We will now examine the syntax of TC as mentioned in snippets in the paper. Note: the code snippets have been taken directly from the paper.

The first example compares how the code for a matrix-vector product in TC differs from that in C-type languages.

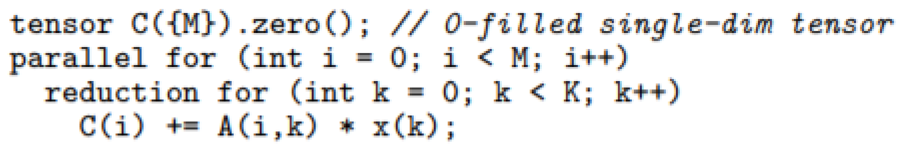
TC code:



```
def mv(float(M,K) A, float(K) x) → (C) {
  C(i)  = 0
  C(i) += A(i,k) * x(k)
}
```

In TC, the range of indices are inferred from their usage, and the index variables like i and k are also declared implicitly. Moreover, induced that only appear on the right side of an expression are assumed to be the dimension over which to reduce over, i.e., in this example, the index variable k only appears in the RHS, which will result in C being a vector of length M.

The syntax for the same function in C would be along the lines of the pseudo code pasted below:



```
tensor C({M}).zero(); // 0-filled single-dim tensor
parallel for (int i = 0; i < M; i++)
  reduction for (int k = 0; k < K; k++)
    C(i) += A(i,k) * x(k);
```

In TC, the entire RHS of an expression is evaluated before updating the LHS. This ensures that update expressions use the previous values of the variable being updated without any unexpected results.

TC also provides provisions to initialise variables during reduction without an explicit statement dedicated to doing so. Appending a ‘!’ symbol to the reduction operator is enough to do so. Thus, the previous matrix-vector product function reduces to this:

```
def mv(float(M,K) A, float(K) x) → (C) {
  C(i) +=! A(i,k) * x(k)
}
```

Properties like these allow us to write ML code in just a few simple lines. Most of the time TC is capable of inferring the range of indices from the usage. However, whenever there might be ambiguity, one can also explicitly define the range, like so:

```
def maxpool2x2(float(B,C,H,W) in) → (out) {
  out(b,c,i,j) max=! in(b,c, 2 * i + kw, 2 * j + kh)
    where kw in 0:2, kh in 0:2
}
```

The where keyword defines the range. In scenarios where ambiguity might arise with respect to inferring the range directly, an error is thrown and the user is asked to define the range explicitly using a where clause. TC infers the range of indices over rounds by maintaining a list of unresolved variables. In each round, each unresolved variable is checked for whether it is out of bounds. Finally, the range is decided by taking the maximum of the intersection of the ranges of the unresolved variables.

TC's operators are not black-box functions, which makes experimenting with them very easy.

TC also makes data layout transformations easy and aids in processes like data tiling and hierarchical decompositions. Implicit data tiling refers to the process of doing calculations without creating copies in memory. Data tiling can be achieved in TC by reshaping vectors and adjusting indices.

*[**NOTE:** some phrases and terms have been lifted directly from the paper so as to not lose out on any important information.]*