**王柏偉 310551166**

**張 軒 310551115**

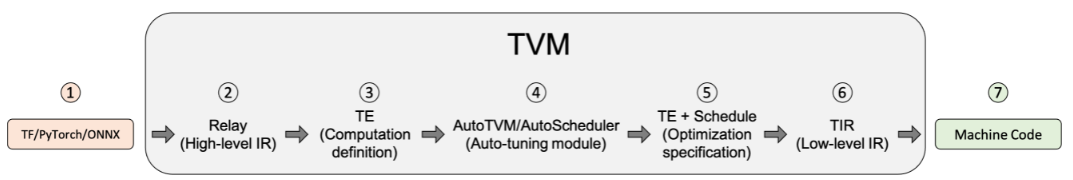
**Topics**

Optimizing tensor operations using TVM

**Background**

TVM is an open source machine learning compiler framework for CPUs, GPUs, and machine learning accelerators. It aims to enable machine learning engineers to optimize and run computations efficiently on any hardware backend.

The diagram below illustrates the steps a machine model takes as it is transformed with the TVM optimizing compiler framework.



As shown in the diagram, after Relay applies graph-level optimization passes to optimize the model, TVM lowers the graph representation to Tensor Expression (TE). TE is a domain-specific language for describing tensor computations. TE also provides several schedule primitives to specify low-level loop optimizations, such as tiling, vectorization, parallelization, unrolling, and fusion.

There often exist several methods to compute the same result, however, different methods will result in different locality and performance. So TVM asks user to provide how to execute the computation called Schedule. A Schedule is a set of transformation of computation that transforms the loop of computations in the program.

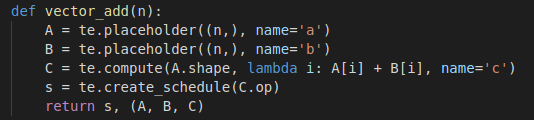
To aid in the process of converting Relay representation into TE representation, TVM includes a Tensor Operator Inventory (TOPI) that has pre-defined templates (TE + schedules) of common tensor operators (e.g., conv2d, transpose).

It is tedious to hand craft kernels like TOPI and the performance are often suboptimal. TVM provides an auto tuning module: AutoScheduler, which generates the search space automatically by analysis the computation definition. It then searches for the best schedule in the generated search space. Note that the auto tuning process is very time consuming.

**Problem tackled**

With the TE expression and a schedule, we can produce runnable code for our target language and architecture, in this case LLVM and a CPU. TVM relies on engineers to write high performance kernel (TE + schedules) to generate efficient code.

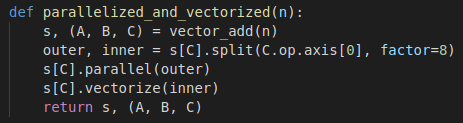
Take vector addition as an example (shown below).



With default scheduling, TVM generates code that serially iterate over the vectors in row major order, which is not efficient. Below is the readable C-style statement returned by TVM after further lowering. In this case, n = 6.



We can add schedule primitives to optimize the computation. As shown below, we add parallel and vectorize primitives to the default schedule.



Below is the readable C-style statement of optimized vector add. We can see that TVM parallelized the loop and grouped 8 iterations together to execute in SIMD instructions on a core.



The vector add example showed that we have to hand-craft high performance TE and schedules for TVM to generate efficient code.

In this report, we will demonstrate some techniques to write high performance kernels for matrix multiplication and compare the throughput with the default implementation.

**Previous related work**

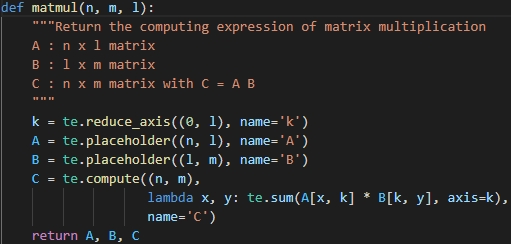
TVM Operator Inventory (TOPI) provides numpy-style generic operations and schedules with higher abstractions than TVM. Different hardware (CPU, GPU, FPGU) and vendors have their own implementations of kernels.

Although TOPI already provides high performance matrix multiplication kernels for different sizes, it’s not easy to dig into the complex code and analysis the impact of the schedules. Thus, we start from the very beginning, hand-craft our own matrix multiplication kernels and analysis the performance. We hope to uncover the influences of different schedule primitives and demonstrate some basic tricks to optimize kernels.

AutoTVM is a legacy auto tuning module of TVM. The developer has to write a schedule template, which typically consists of 20-100 lines of tricky DSL code. This part requires domain expertise of both the target hardware architecture and operator semantics, so it is difficult. On the other hand, AutoScheduler doesn’t need any schedule template and can generate more efficient code in shorter search time.

**Solution**

In the following program, we first declare the placeholders A and B for both inputs by specifying their shapes through tvm.te.placeholder. Both A and B are Tensor objects, which we can feed data later.



In order to increase the performance of matrix multiplication, we try some schedule primitives to specify low-level loop optimizations, such as reordering, parallelization, tiling, and write cache.

Default:

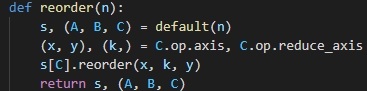
The default schedule implementation will like below



Reordering:

The original implementation will access the B matrix column by column, this approach will hurt the performance due to the cache miss, so we change the axes order from (x,y,k) to (x,k,y) by the reorder primitive, so we will process all matrix row by row.

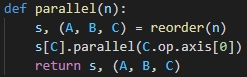
The schedule primitives we implement will like below



Parallelization:

After reordering, we compute the results of a row in C, each row can be computed in parallel, so we can make the schedule parallelize the axis x.

The schedule primitives we implement will like below

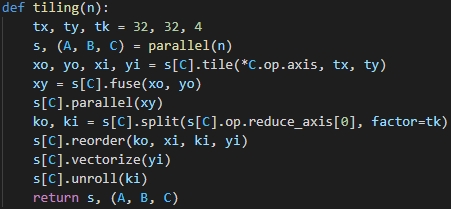


Tiling:

One commonly used strategy is tiling matrices into small blocks that can be fitted into the cache, we can decompose this matrix multiplication into multiple small ones

C[x:x+tx, y:y+ty] = sum(np.dot(A[x:x+tx,k:k+tk], B[k:k+tk,y:y+ty]) for k in range(0, n, tk))

The schedule primitives we implement will like below

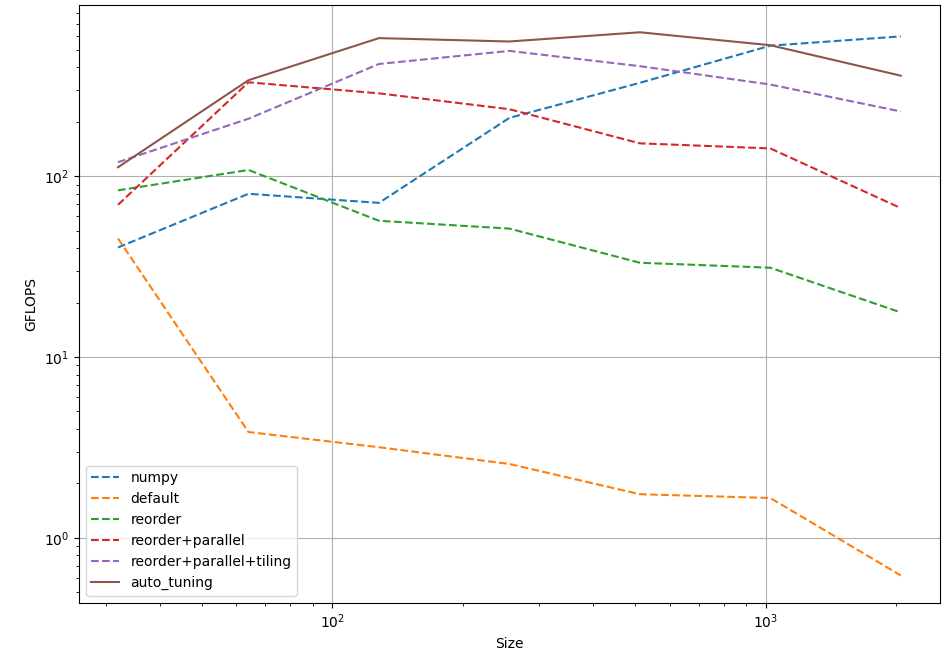


**Evaluation**

For the optimization for several schedule primitives, we compare the throughput with the default implementation.

We plot the line chart in the below Figure, the x axis represents the matrix size with 32/64/128/256/512/1024/2048 and the y axis represents the throughput with GFLOPS.

We can see the figure that it increases obviously while we only add the reorder optimization, the throughput increases with the more optimization, but the auto tuning scheme is still better than what we do.



**Work of each person**

王柏偉: Solution, Evaluation

張 軒: Background, Problem tackled, Related work

**Reference**

[1] <https://dada.cs.washington.edu/research/tr/2017/12/UW-CSE-17-12-01.pdf>

[2] <https://tvm.apache.org/docs/tutorial/introduction.html>

[3] <https://tvm.apache.org/docs/tutorial/tensor_expr_get_started.html>

[4] <https://tvm.apache.org/2021/03/03/intro-auto-scheduler>

[5] <https://tvm.apache.org/docs/tutorial/auto_scheduler_matmul_x86.html>