Performance Analysis on Automatic Synthesis for Co-Optimized Kernel Generation and Scheduling

Kiran Kumar Rajan Babu, Pavan Hegde
{krajanba, pavanh}@andrew.cmu.edu

Introduction

Frameworks for machine learning perform control flow analysis and subsequently schedule primitive kernel operations, such as a matrix multiplication and convolution, to run on hardware either sequentially or concurrently. Each operator is typically optimized in isolation. As a result, running two or more operators concurrently may yield suboptimal performance on account of physical resources on chip. For example, a kernel *mm()* for matrix multiplication may be optimized assuming it has the full hardware resource. When scheduled to run alongside another operator *X()*, *mm()* and *X()* both perform worse.

The goal of this project was to implement and evaluate a flexible kernel generation pass as an extension to the existing TVM, machine learning compiler framework. This is the first step in creating a co-optimized scheme. Later works may then run exhaustive or non-exhaustive searches across the design space to optimize performance when taking both scheduling and the kernel itself into consideration.

Metrics include performance of kernels generated with different granularities of fused loops and parallelism via threading specifically in the context of CPU. In addition, this report measures and documents the effects of various load balancing choices on the resulting kernel performance.

Background - TVM

The Apache TVM framework implements a compiler infrastructure and Domain-Specific Language (DSL) for tensor processing which pull heavily from previous systems for image processing, specifically Halide. Compilation is split into two phases: correctness (via description of the algorithm) and performance (via scheduling primitives and directives). The programmer starts by creating a naive, default schedule on their algorithm and is able to improve performance without worrying about correctness by changing the scheduling of a given operation. TVM's DSL supports standard operations like blocking, tiling, parallelized loops, loop reordering and more. After fully describing both an operation and schedule, the user can build their function to target their desired hardware - GPUs via CUDA, CPUs via LLVM. This project targets LLVM via CPUs.

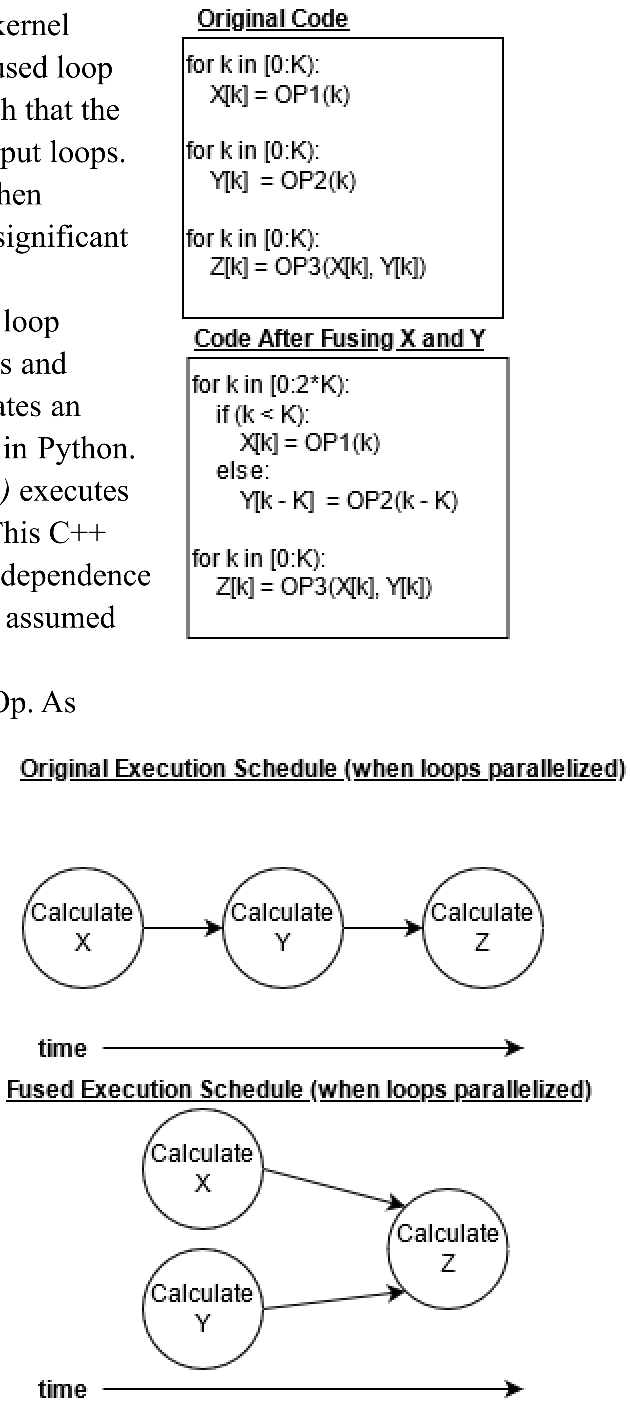
Fused Kernel Generation

Figure 1 outlines a simple example of fused kernel generation. Given the original sequential code, the fused loop scheduling primitive merges the two target loops such that the loop bounds now span the original bounds of both input loops. At first glance this does not seem useful, however when parallelized, the intuition is that this should provide significant speedup when compared to the default schedule.

This project leverages Pratik's original fused loop functionality implementation to provide Python binds and integration with TVM's C++ source. Figure 2 illustrates an example and ease of use of the TVM DSL extension in Python. Internally, the call to schedule.*create_fused_loops(...)* executes the natively built C++ source for the kernel fusion. This C++ source implements basic analysis on the algorithm's dependence graph. First, all output compute operators, which are assumed **independent** both by the pass and TVM's DSL, are encapsulated in an "envelope" operator, FusedLoopOp. As illustrated in Figure 3, when the operation *A[x,y] + 1* with loop bounds 0 to 2 and operation *B[x,y] + 3* with loop bounds 0 to 2 are fused, they are merged into one fused operation with loop bounds 0 to 4. Note that the ranges 0 to 2 were chosen in this example because of the limited threads available in modern CPUs. Should this code be optimized for a modern GPU, the original ranges would be on the order of 1024 (and the fused ranges on the order of 2048).

As seen in Figure 2 and Figure 3, the TVM built-in scheduling primitive, *parallel()*, is used to split the fused loops across threads.

In order to properly encapsulate these operations within a Fused operation, the C++ implementation must perform a variety of copy, object restructure and re-reference operation to properly link with TVM's schedule checkers and expected Python conventions.


Original Code

```
for k in [0:K):
   X[k] = OP1(k)

for k in [0:K):
   Y[k]  = OP2(k)

for k in [0:K):
   Z[k] = OP3(X[k], Y[k])
```

Code After Fusing X and Y

```
for k in [0:2*K):
   if (k < K):
      X[k] = OP1(k)
   else:
      Y[k - K]  = OP2(k - K)

for k in [0:K):
   Z[k] = OP3(X[k], Y[k])
```

Original Execution Schedule (when loops parallelized)
Calculate X
Calculate Y
Calculate Z
time
Fused Execution Schedule (when loops parallelized)
Calculate X
Calculate Z
Calculate Y
time

Figure 1: High Level Fused Kernel Example

```
# Algorithm Description
A = te.placeholder((M,N), name="A")
B = te.placeholder((M,N), name="B")
X = te.compute(A.shape, lambda x, y: A[x,y]+1, name="X")
Y = te.compute(B.shape, lambda x,y: B[x,y]+3, name="Y")
Z = te.compute(X.shape, lambda x,y: X[x,y]*Y[x,y], name="Z")

# Scheduling and Fusing
S = te.create_schedule([Z.op])
inputs = [[X], [Y]]; fuse_outputs = [X, Y]; extents = [M, M]

# Fuse and parallelize the fuse loop
new_outputs = s.create_fused_loop(inputs, fuse_outputs, extents)
s[new_outputs[0]].parallel(new_outputs[0].op.loop_axis)

# Build the function
func_fused = tvm.build(s, [A, B, Z], target=target, name="fuse_test")

# Use it!
a = Tensor(M, N); b = Tensor(M, N); z = zeros(M, N)
func_fused(a, b, z)
```

<u>Figure 2:</u> Fused Loop Creation Example in with TVM Python Interface

Default schedule generated by TVM:

// First set of loops which generates the values for tensor X
```
  for (x.r: int32, 0, 2) "parallel" {
    for (y.r: int32, 0, 10000000) {
      X[((x.r*10000000) + y.r)] = ((float32*)A_2[((x.r*10000000) + y.r)] + 1f32)
    }
  }
```

// Second set of loops which generates the values for tensor Y
```
  for (x.r_1: int32, 0, 2) "parallel" {
    for (y.r_1: int32, 0, 10000000) {
      Y[((x.r_1*10000000) + y.r_1)] = ((float32*)B_2[((x.r_1*10000000) + y.r_1)] + 3f32)
    }
  }
```

// Third set of loops which generates the values for tensor Z (depends on tensor X and tensor Y)
```
  for (x.r_2: int32, 0, 2) {
    for (y.r_2: int32, 0, 10000000) {
      Z_2[((x.r_2*10000000) + y.r_2)] = ((float32*)X[((x.r_2*10000000) + y.r_2)]*(float32*)Y[((x.r_2*10000000) + y.r_2)])
    }
  }
```

Optimized schedule generated by Create fused loop:

// Fused the first and second set of loops using create_fused_loops
```
  for (fused.r: int32, 0, 4) "parallel" {
    if (fused.r < 2) {
      attr [meta[FusedLoopOp][0]] "fused_branch_scope" = 0;
      for (y.f.r: int32, 0, 10000000) {
        fuse_test.v0[((fused.r*10000000) + y.f.r)] = ((float32*)A_2[((fused.r*10000000) + y.f.r)] + 1f32)
      }
    } else {
      attr [meta[FusedLoopOp][0]] "fused_branch_scope" = 1;
      for (y.f.r_1: int32, 0, 10000000) {
        fuse_test.v1[(((fused.r*10000000) + y.f.r_1) - 20000000)] = ((float32*)B_2[(((fused.r*10000000) + y.f.r_1) - 20000000)] + 3f32)
      }
    }
  }
```

// Third set of loops remains the same as the default schedule
```
  for (x.r: int32, 0, 2) {
    for (y.r: int32, 0, 10000000) {
      Z_2[((x.r*10000000) + y.r)] = ((float32*)fuse_test.v0[((x.r*10000000) + y.r)]*(float32*)fuse_test.v1[((x.r*10000000) + y.r)])
    }
  }
```

<u>Figure 3:</u> TVM Generated Kernel (Denoted in TIR which can later be reduced to LLVM or CUDA)

Experimental Setup

Experimental results were gathered through empirical testing of generated kernels. For each kernel tested, Python code similar to that shown in Figure 2 was built into native functions and timed using TVM's evaluator object. For each test performed, execution time was measured for both the default parallelized schedule and fused loop parallelized schedules and then averaged over 30 runs.

On account of bugs with the *create_fused_loop()* implementation, this analysis included only outer loop fusion.

All tests were performed on readily available hardware within a Lubuntu VM, namely on an Intel® Core™ i7-1065G7 CPU @ 1.30GHz × 8 (capable of handling 8 hardware threads).

To evaluate the effects of sweeping thread count, we ran the code by varying the outer loop's dimension from M = 1 to 8. We did this by considering the fact that our hardware is capable of handling upto 8 threads as we parallelize across the outer loop's axis.

One more thing which we considered for the performance testing was load balancing and load unbalancing cases. In the load balancing case we made the X and Y as add instructions (same instructions) and both X and Y instructions have equal amounts of work to do which is intuitive as they perform N^2 computations. On the other hand, the load unbalanced case, X and Y were different instructions with different execution times(X has add which is lower execution time) and also X had lesser number of computations (N computations) to perform than Y (N^2 computations) in this case. We thought concurrent-scheduling of these type of instructions would be interesting to see. As we expected the load unbalanced test acted as a stress-test for the create-fused loop schedule as we will show in the results later in this report.

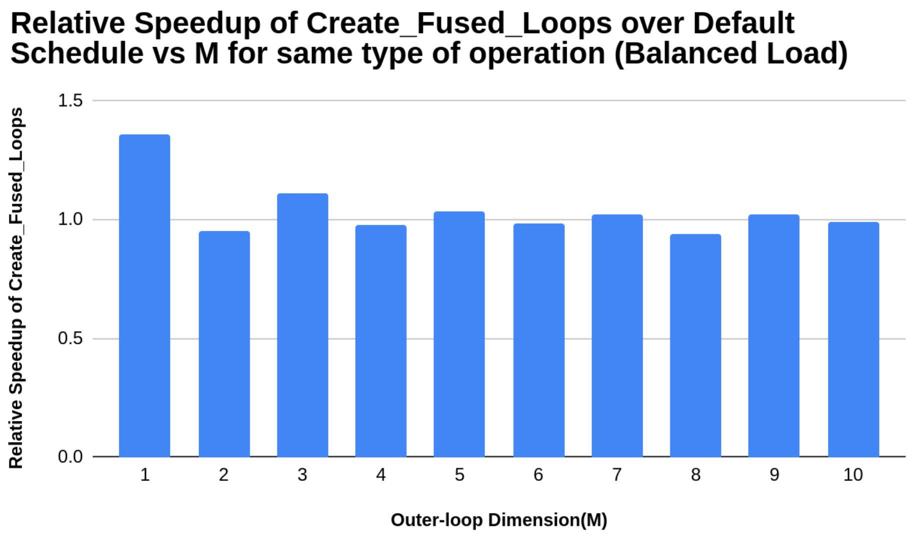
Code snippet for balanced load:

```
A = te.placeholder((M,N), name="A")
X = te.compute(A.shape, lambda x,y: A[x,y]+1, name="X")
B = te.placeholder((M,N), name="B")
Y = te.compute(B.shape, lambda x,y: B[x,y]+3, name="Y")
Z = te.compute(X.shape, lambda x,y: X[x,y]*Y[x,y], name="Z")
```

Code snippet for unbalanced load:

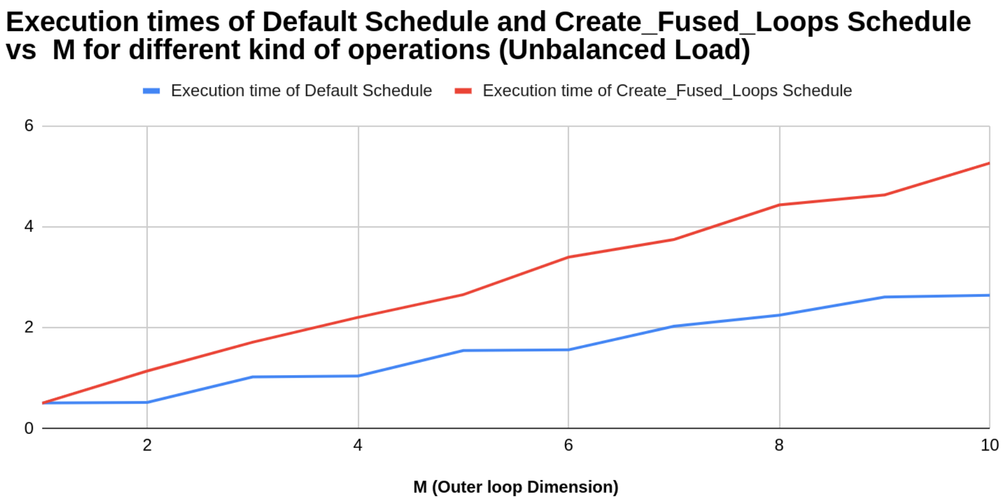
```
A = te.placeholder((M,N), name="A")
X = te.compute(A.shape, lambda x, _: A[x,0]+1, name="X")
B = te.placeholder((M,N), name="B")
Y = te.compute(B.shape, lambda x,y: te.sigmoid(te.exp(B[x,y])), name="Y")

Z = te.compute(X.shape, lambda x,y: X[x,y]+Y[x,y], name="Z")
```

Results and Evaluation



We believe that the relative speedup oscillates for odd *M* and even *M* because the latency of the last chunk of work is hidden by the penultimate chunk of work when *M* is odd due to unequal split of work between threads. The free thread picks up the unequal chunk of work and completes it in parallel with the penultimate chunk of work.

The latency hiding does not happen when *M* is even but there is also an overhead of evaluating an extra branch inside Create_fuse_loops schedule compared to Default schedule which makes the even *M* case slower for Create fused loops schedule compared to Default Schedule.



Due to implicit Barriers, threads which finish early (threads which run the add operation) have to wait for slower threads(threads which run the exp(sigmoid(x)) to finish before starting the next batch of work.

Surprises

As mentioned before, we only ran with outer loop fusion. When trying to fuse inner loops, we ran into odd bugs where fusion resulted in extra, dead loops which had no impact on algorithm correctness, however did cause unnecessary looping during dynamic execution. After some deep diving we were able to track down the bug however “fixes” we tried resulted in TVM’s schedule checker yelling at us.

Conclusions and Future Work

Loop fusion in the context of simultaneous kernel operations presents an intriguing optimization to “speeding” up workloads in highly parallelizable tensor operations. Based on the result from this study fusion is not particularly beneficial in case of CPUs on account of the low number of available threads. Despite the negative results from this study, we do still expect it to be highly beneficial in case of GPU where we can make use of hundreds of threads. GPUs also leverage significantly different synchronization primitives when compared to CPUs.

This study also demonstrated that, on a CPU, parallelized fused loops do not perform well for concurrent scheduling of unbalanced load (dissimilar operations which have different execution times) likely due to implicit barriers.

Future work would include bug fixes to the existing implementation, expanding the pass to encapsulate more TVM operation types (for example ScanOp operations for dependent compute), and most importantly, evaluation on GPU systems.

Acknowledgements and Distribution of Work

We would like to give special thanks to Pratik Fegade (ppf@andrew.cmu.edu) for his major contributions and phenomenal help towards this project including, but not limited to, his expertise with TVM and initial implementation of loop fusion.

Work for the remainder of the project was split evenly between Kiran and Pavan. Pavan aided with initial Python runs, TVM build environment setup, and C++ source analysis while Kiran also worked on the initial Python runs, data collection and polished evaluation harness.