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

Jiazheng Yuan
Problem

1. Newly invented operator might suffer from performance cost if no available function call is present in the framework library.

2. Even if there is, might not be optimal for the user’s network and dataset, Existing functions might not be tuned for all combination of data sizes.

3. Computation graph are too abstract to capture all detail.
Solution: Tensor Comprehension

(1) a language close to the mathematics of deep learning,

(2) a polyhedral Just-In-Time compiler to convert a mathematical description of a deep learning DAG into a CUDA kernel with delegated memory management and synchronization, also providing optimizations such as operator fusion and specialization for specific sizes

(3) a compilation cache populated by an autotuner.
Current approaches

1. Halide, a image processing library, requiring users to do scheduling by themselves, often too hard for most users, 2d only

2. Active library which generate code on demand, which is still hard to cover all cases.

3. Current deep learning Compilers is on its way, such as XLA and Latte, performance level is still not met on GPU.
Advantage

1. Concise and expressive
2. Specializing a polyhedral intermediate representation and its compilation algorithms to the domain of deep learning
3. General enough to be integrated into other ML frameworks
4. An end-to-end compilation flow capable of lowering tensor comprehensions to efficient GPU code
5. TC closely matches an algorithmic notation
Syntax

Actual TC code:

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

Equivalent c-style pseudo-code:

```c
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);
```

Rule: 1. Index implicitly defined. Bound inferred

2. Index only on right will be reduced

3. Order of evaluation point in iteration space does not matter
Syntax

Initialization: append ! to operator to initialize

```python
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
}
```

Range of variable must be specified using kw, in case they cannot be inferred, in this case, by specifying kw,

It can be inferred that $0 \leq 2i \& 2i + 2 < H$. $0 \leq 2j \& 2j + 2 < W$. Thus each out value is taking the max of four values
Range Inferring

```python
def sgemm(float a, float b,
          float(N,M) A, float(M,K) B) → (C) {
    C(i,j) = b * C(i,j)  # initialization
    C(i,j) += a * A(i,k) * B(k,j)  # accumulation
}
```

1. Making variable as large as possible without going out of bound, if element appeared twice, take the less range (this is taken as an example for compilation process). Loops are implicit, and optimizations are performed automatically.

```python
def conv1d(float(M) I, float(N) K) → (O)
    O(i) = K(x) * I(i + x)
```

2. Find the range of expression containing only one variable, then deduce ones containing more variables using the already obtained range. 0 =< X < N,  0 < i <= M-N.
Integration

1. Translating tensor of other framework to TC’s own (DLPack support, a header only library providing meta info only for converting tensor to TC’S)

2. Overriding operators to generate TC.

3. Currently support Pytorch and caffe2
Workflow description

Can also generate a readable naive cuda for reference
Polyhedral JIT Compilation

schedule trees structure:

1. A band node: defines a partial execution order through one or multiple piecewise affine functions

2. Filter node: binding its subtree to a subset of the iteration domain

3. Context nodes provide additional information on the variables and parameters

4. Extension nodes introduce auxiliary computations that are not part of the original iteration domain

5. Sequence node: specify execution order if necessary
def sgemm(float a, float b, float(N,M) A, float(M,K) B) → (C) {
    C(i,j) = b * C(i,j) # initialization
    C(i,j) += a * A(i,k) * B(k,j) # accumulation
}
Compilation steps

A canonical schedule tree for a TC is defined by an outer Sequence node, followed by Filter nodes for each TC statement. Inside each filtered branch, Band nodes define an identity schedule with as many one-dimensional schedule functions as loop iterators for the statement.

\[
\begin{aligned}
\text{Domain} & : \\
\{S(i, j) \mid 0 \leq i < N \land 0 \leq j < K\} \\
\{T(i, j, k) \mid 0 \leq i < N \land 0 \leq j < K \land 0 \leq k < M\}
\end{aligned}
\]

Sequence
- Filter\{S(i, j)\}
- Band\{S(i, j) \rightarrow (i, j)\}
- Filter\{T(i, j, k)\}
- Band\{T(i, j, k) \rightarrow (i, j, k)\}

(a) canonical \textbf{sgemm}
Command for tuning

tensor_comprehensions.autotune(tc, entry_point, *inputs, starting_options=None, tuner_config=<tensor_comprehensions.tclib.TunerConfig object>, cache_filename=None, load_from_cache=False, store_to_cache=False)

1. either through starting_options or (load_from_cache and cache_filename)

2. Different method of tensor_comprehensions.tclib.MappingOptions can achieve different strategy

3. How to choose starting mapping options? Don’t., better use default: i.e: makeConvolutionMappingOptions()
This tree features an outer band node with i and j loops that became common to both statements, which corresponds to loop fusion. The sequence node ensures that instances of S are executed before respective instances of T enabling proper initialization. The second band is only applicable to T and corresponds to the innermost (reduction) loop k.

After this step, loop of initialization and execution are fused.

Transform command:

```
.scheduleFusionStrategy(<choice of Max, Preserve3Coincident, Min>)
```
Loop Tiling

it converts a permutable schedule band into a chain of two bands with the outer band containing tile loops and the inner band containing point loops with fixed trip count.

Comparing to step (b), tiling is applied which is an additional optimization.

Transform command

.tile(<list of positive integers>)
C style equivalent

for(int i = 0; i < N; i++){
    for(int j = 0; j < K; j++){
        for(int ii = i; ii < min(i + 32, N); ii++){
            for(int jj = j; jj < min(j + 32, K); jj++){
                initialization();
                for(int k = 0; k < M; k++){
                    accumulation();
                }
            }
        }
    }
}
Imperfectly nested tiling is implemented by first tiling all bands in isolation and then sinking parallel point loops in the tree.

Comparing to step(c), access on dimension K is also tiled, thus achieved imperfect loop tiling (not all computations are in inner loop).

Transform command:
.scheduleFusionStrategy(<choice of Max, Preserve3Coincident, Min>)
C style equivalent

for(int i = 0; i < 32 * N; i += 32) {
    for(int j = 0; j < 32 * K; j += 32) {
        for(int ii = i; ii < min(i + 32, N); ii += 1) {
            for(int jj = j; jj < min(j + 32, K); jj += 1) {
                initialization();
                for(int kk = k; kk < min(k + 32, M); kk += 1) {
                    for(int ii = i; ii < min(i + 32, N); ii += 1) {
                        for(int jj = j; jj < min(j + 32, K); jj += 1) {
                            accumulation();
                        }
                    }
                }
            }
        }
    }
}

(d) fused, tiled and sunk
Gpu Mapping

Mapping to GPU, adding parameters. Comparing to step(d), this becomes code on GPU, GPU parameter

Added, such as tx,ty,bx,by, corresponding to block and thread id

This series of steps is autotuning, but user can also specifies dependency graph to tell what is allowed and what is not to be optimized.

Transform command:

.mapToBlocks(<list of 1..3 positive integers>) .mapToThreads(<list of 1..3 positive integers>)
Autotuning and Caching

balance out the cost of JIT compilation, caching and autotuning is used:

- Caching: each entry key is a tuple (TC, input, shapes, target, architecture), Entry is the fastest known version. There could be pre-populated reference implementation to avoid unpredictably long compilation time.
Autotuning and Caching

3 parts to set up

1. a set of starting configurations that worked well for similar TCs, and a few predefined strategies

2. the tuning space dimensions and admissible values for ranges;

3. the type of search—currently a genetic algorithm or random.
Autotuning and Caching

Tuning process: Multiple candidates compiled profiled and run, result used to generate the candidate for next phase,

A database to save all performance data for all versions, useful when additional technique implemented such as Bayesian hyperparameter search

Parameters for tuning: Block size, grid size, fusion strategy, shared memory etc.
### Performance

Wall-clock execution of kernels (in µs). Each kernel ran 1000 times. The top half of each table is Tesla M40 (Maxwell) and the bottom half is Tesla P100 (Pascal); N/A indicates the framework lacked an implementation.

<table>
<thead>
<tr>
<th>(B, M, K, N)</th>
<th>(nil, 128, 32, 256)</th>
<th>(nil, 128, 1024, 1024)</th>
<th>(nil, 128, 4096, 16384)</th>
<th>(500, 72, 26, 26)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p0</td>
<td>p50</td>
<td>p90</td>
<td>p0</td>
</tr>
<tr>
<td>Caffè2 + CUBLAS</td>
<td>33</td>
<td>35</td>
<td>36</td>
<td>127</td>
</tr>
<tr>
<td>ATen + CUBLAS</td>
<td>35</td>
<td>35</td>
<td>36</td>
<td>120</td>
</tr>
<tr>
<td>TC (manual)</td>
<td>32</td>
<td>33</td>
<td>35</td>
<td>441</td>
</tr>
<tr>
<td>TC (autotuned)</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>309</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Caffè2 + CUBLAS</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>107</td>
</tr>
<tr>
<td>ATen + CUBLAS</td>
<td>26</td>
<td>27</td>
<td>27</td>
<td>104</td>
</tr>
<tr>
<td>TC (manual)</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>188</td>
</tr>
<tr>
<td>TC (autotuned)</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>107</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>p0</td>
<td>p50</td>
<td>p90</td>
<td>p0</td>
</tr>
<tr>
<td>Caffè2 + CUDNN</td>
<td>1,672</td>
<td>1,734</td>
<td>1,764</td>
<td>1,687</td>
</tr>
<tr>
<td>ATen</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>TC (manual)</td>
<td>6,690</td>
<td>6,752</td>
<td>6,805</td>
<td>3,759</td>
</tr>
<tr>
<td>TC (autotuned)</td>
<td>666</td>
<td>670</td>
<td>673</td>
<td>1,212</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Caffè2 + CUDNN</td>
<td>1,308</td>
<td>1,343</td>
<td>1,388</td>
<td>1,316</td>
</tr>
<tr>
<td>ATen</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>TC (manual)</td>
<td>3,316</td>
<td>3,339</td>
<td>3,345</td>
<td>2,327</td>
</tr>
<tr>
<td>TC (autotuned)</td>
<td>319</td>
<td>321</td>
<td>322</td>
<td>691</td>
</tr>
</tbody>
</table>
Performance

<table>
<thead>
<tr>
<th>Production Models</th>
<th>1LUT</th>
<th></th>
<th></th>
<th>2LUT</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p0</td>
<td>p50</td>
<td>p90</td>
<td>p0</td>
<td>p50</td>
<td>p90</td>
</tr>
<tr>
<td>Caffe2 + CUBLAS</td>
<td>78</td>
<td>80</td>
<td>82</td>
<td>188</td>
<td>193</td>
<td>207</td>
</tr>
<tr>
<td>ATeN + CUBLAS</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>TC (manual)</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>47</td>
<td>49</td>
<td>52</td>
</tr>
<tr>
<td>TC (autotuned)</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>47</td>
<td>49</td>
<td>52</td>
</tr>
<tr>
<td>Caffe2 + CUBLAS</td>
<td>63</td>
<td>64</td>
<td>66</td>
<td>122</td>
<td>125</td>
<td>128</td>
</tr>
<tr>
<td>ATeN + CUBLAS</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>TC (manual)</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>30</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>TC (autotuned)</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>30</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>MLP1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>p0</td>
<td>p50</td>
<td>p90</td>
<td>p0</td>
<td>p50</td>
<td>p90</td>
</tr>
<tr>
<td>Caffe2 + CUBLAS</td>
<td>123</td>
<td>125</td>
<td>135</td>
<td>146</td>
<td>159</td>
<td>164</td>
</tr>
<tr>
<td>ATeN + CUBLAS</td>
<td>109</td>
<td>110</td>
<td>112</td>
<td>128</td>
<td>142</td>
<td>148</td>
</tr>
<tr>
<td>TC (manual)</td>
<td>150</td>
<td>157</td>
<td>159</td>
<td>344</td>
<td>349</td>
<td>351</td>
</tr>
<tr>
<td>TC (autotuned)</td>
<td>123</td>
<td>125</td>
<td>131</td>
<td>219</td>
<td>224</td>
<td>227</td>
</tr>
<tr>
<td>Caffe2 + CUBLAS</td>
<td>123</td>
<td>133</td>
<td>134</td>
<td>107</td>
<td>113</td>
<td>115</td>
</tr>
<tr>
<td>ATeN + CUBLAS</td>
<td>98</td>
<td>98</td>
<td>99</td>
<td>105</td>
<td>110</td>
<td>112</td>
</tr>
<tr>
<td>TC (manual)</td>
<td>91</td>
<td>92</td>
<td>93</td>
<td>275</td>
<td>279</td>
<td>281</td>
</tr>
<tr>
<td>TC (autotuned)</td>
<td>79</td>
<td>80</td>
<td>80</td>
<td>117</td>
<td>128</td>
<td>129</td>
</tr>
</tbody>
</table>

Wall-clock execution of kernels (in µs). Each kernel ran 1000 times. The top half of each table is Tesla M40 (Maxwell) and the bottom half is Tesla P100 (Pascal); N/A indicates the framework lacked an implementation.
Performance

Evaluation:

Except on transposed matrix multiplication kernel with large enough size, TC performs well comparing to the reference.
Conclusion

1. High performance achieved in a variation of kernels
2. Still has space for improvement, i.e.: register tiling.
3. Claims to be concise, expressive, and easy to understand. It’s hard to compare.
4. Quite new framework, needs time to see how well it develops.