Accelerating Capsule Networks with Tensor Comprehensions

Prem Nair
Princeton University
pnair@princeton.edu

Nicholas Jiang
Princeton University
nj3@princeton.edu

Abstract

We investigate the utility of Tensor Comprehensions, a recently released library from Facebook AI Research that automatically creates high performance CUDA kernels for arbitrary mathematical operations on tensors. We apply it to optimize specific components of the Capsule Network architecture, a novel neural network by Sabour et al. *[10], which uses many uncommon computation layers that are suboptimal using standard kernel libraries. Combining all of our optimizations, we produce the fastest PyTorch implementation of the Capsule Network architecture as far as we know, accelerating training by 2.2*x *and inference by 3.8*x *compared to our reference implementation.*

1. Introduction

With NVIDIA's release of CUDA in 2007 [7], and the discovery of neural network acceleration on GPUs [9], significant work has gone into the optimization of common neural network layers and tensor operations performed on GPUs. A major milestone in this process was NVIDIA's 2014 release of cuDNN [3], a library of kernels (functions sent to run on the GPU) that implement the most common mathematical operations used by modern convolutional neural networks and their layers, including convolution, pooling, normalization, and activation functions. CUDA also supports BLAS-like tensor operations through cuDNN and cuBLAS. BLAS is a specification of linear algebra routines, including vector addition, matrix-vector multiplication, and matrix-matrix multiplication. This in turn allowed deep learning researchers and engineers to focus on innovation in model architectures instead of directing effort into the low-level specifics of GPU performance.

Nowadays, nearly all commonly used deep learning frameworks are built upon cuDNN/cuBLAS to easily take advantage of its GPU acceleration for common operations. While they provide efficient implementations for this set of operations, they do not supply similarly optimized implementations for non-standard operations. As a result, innovative architectures that make use of uncommon tensor operations or propose new ones are often left unoptimized for GPU acceleration. This makes it harder for research to continue efficiently in these directions. To address this issue, Facebook AI Research (FAIR) recently released Tensor Comprehensions (TC) [11], a library made to automatically generate high-performance CUDA kernels for arbitrary tensor operations.

The architectures most likely to benefit from TC for training and inference are those where the computational bottleneck is a non-standard operation unlikely to be properly optimized by cuDNN or other GPU libraries. We confirm our guess that one such type of model is the recently introduced Capsule Network, by Sabour *et al*. [10]. It proposes a dramatic deviation in architecture and training from typical neural networks, replacing the basic building block of the standard neuron, which takes in an array of scalar values and outputs a single scalar value, with a capsule whose inputs and output are vectors. In this paper, we apply TC's automatic CUDA kernel generation and optimization to multiple tensor operations used by CapsNet, resulting in the following contributions:

- The fastest known (and computationally correct) PyTorch implementation of the CapsNet architecture
- An analysis of some success and failure cases of the TC library
- Suggestions for further optimizations

2. Background

2.1. Capsule networks

Sabour *et al*. [10] introduces capsule networks with dynamic routing, a new neural network paradigm intended to improve upon traditional convolutional neural networks in computer vision. In a normal linear layer, an individual neuron takes in many scalars, weighs them, then sums them, then spits out a scalar which passes through a nonlinearity like ReLU to become the final output. The larger the magnitude of the output, the stronger the indication of some sort

of feature. For efficiency, many neurons' features are calculated simultaneously, making the whole operation equivalent to a matrix-vector multiplication, which is a BLAS routine, and therefore optimized by GPU libraries. The deeper layers in a network convey more abstract features.

Similarly, the fundamental unit of computation in a capsule network is a capsule. It takes as inputs many vectors, which are transformed (a step not found in neurons), weighed, and summed, resulting in a single vector which is then nonlinearly transformed to have magnitude less than 1. Part of this process is equivalent to a batch matrix-vector multiplication, which is also a BLAS routine. A capsule's output vector is intended to capture an explicit parametrization of some feature through its direction, as well as the feature's strength from its magnitude. Output vectors are then treated as a finite resource, and their magnitudes are shared among the capsules in the next layer with weights determined by the so-called dynamic routing algorithm. At a high level, this routing algorithm effectively clusters lower level features into groups that cause strong responses in higher level capsules.

While normal neural networks often exclusively consist of the operations implemented by optimized libraries, CapsNets require many other tricks, including tensor dimension transpositions, a unique nonlinearity applied to vectors instead of scalars, forms of element-wise tensor multiplications, and softmax applied across interior dimensions of a tensor, thus motivating the need for custom kernels. We go into great detail about the specifics of the capsule computation and the dynamic routing algorithm in Section 3.

2.2. Tensor operation optimization

2.2.1 CUDA optimization

To convey the difficulty of writing custom kernels, here we walk through some optimizations drawn from a computer architecture course [1] one might use to improve the performance of something like matrix multiplication to get a sense of what can cause CUDA kernel improvements. If we are trying to multiply matrix **A** by **B** and adding matrix **C** to get **D**, we attempt to do a triple-nested for loop, where in the inner most loop we accumulate

$$d_{ik} = \sum_j a_{ij} b_{jk} + c_{ik}. \tag{1}$$

CUDA's computation model consists of many threads running the same kernel concurrently. CUDA has 3 levels of memory: global, shared, and local. Each successive group is accessible by fewer CUDA threads, is smaller, but is also faster. By tiling the matrices into smaller regions and using shared memory among groups of threads to reduce the amount redundantly read from global memory, we can make the matrix multiplication faster. Too small or too large a tile size degrades performance. Transposing **B** can further help with memory access performance due to the coalesced memory access from **B** now being column-major. The right order of the 3 loops helps, but the number of loop order options increases rapidly for higher dimensional operations.

CUDA also possesses many float primitives, such as fused multiply-add, allowing the accumulation operation $d_{ik} := d_{ik} + a_{i1} b_{1k}$ to be done with a single arithmetic instruction as opposed to two.

Finally, we can use loop unrolling (explicitly listing out the same instructions many times to reduce time spent incrementing loop variables) to further improve throughput. Doing too much rolling causes the kernel code byte size to be too large, slowing down performance.

Most of these optimizations would be similar to steps one might take for CPU-based matrix multiplication optimization, with the exception of the use of tiling and shared memory that is made possible (or necessary) by the CUDA computing/threading model. More sophisticated, math-based optimizations exist for convolutions, which are implemented by cuDNN, such as in Lavin and Gray [6].

One can see that it is difficult for a researcher to spend so much time researching every optimized GPU primitive available, meticulously managing memory and computation bandwidths, and testing different hyperparameters to get good computation efficiency for kernels they come up with. (They already have enough hyperparameters to deal with in their neural nets!) The TC library automates this entire process.

2.2.2 Tensor comprehensions

It is impractical to expect high performance computing knowledge and engineering expertise from deep learning researchers, but the pace of research is also slowed by inefficient research code. FAIR's TC library [11] strives to eliminate the problem by allowing researchers to translate math expressions they write in a custom language directly into performance-tuned kernels.

As an example, we walk through how to specify a matrix-matrix multiplication operation in TC's mathematical notation. Let $\mathbf{A} \in \mathbb{R}^{M \times N}$ and $\mathbf{B} \in \mathbb{R}^{N \times P}$ be the two matrices to be multiplied to yield matrix $\mathbf{C} \in \mathbb{R}^{M \times P}$. The corresponding representation in TC notation is:

```
def mm(float(M,N) A, float(N,P) B) -> (C) {
    C(i, j) +=! A(i, k) * B(k, j)
}
```

TC borrows from Einstein notation to allow conciseness. The dimensions of **C** are inferred from the index variables used, and any variable on the right but not on the left is summed over. The operator `+=!`, means that `C(i,j)` should be incremented by a sum ("+=") over the right hand side, and initialized to 0 ("!").

When compiled by TC, this computes the same output as the following C++ loop:

```
for(int i = 0; i < M; i++) {
  for(int j = 0; j < P; j++) {
    C(i,j) = 0.0f;
    for(int k = 0; k < N; k++) {
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

Note that the loop order of `[i,j,k]` in the C++ code must be arbitrarily chosen whereas the tensor comprehension language is independent of the loop order.

TC's notation is flexible enough that it can express other common tensor operations for neural networks like convolution (including strided and grouped), and it has direct access to CUDA math functions for performance.

2.3. Broadcasting

Broadcasting is a notational convenience. It is a way of performing element-wise tensor operations between tensors with different shapes by implicitly expanding them to have the same shape. As an example [2], take the sum of two tensors, one with dimension 2×3 and one with dimension 1×3 (but fundamentally just a vector of length 3):

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 7 & 8 & 9 \end{bmatrix} \tag{2}$$

$$= \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 7 & 8 & 9 \\ 7 & 8 & 9 \end{bmatrix} \tag{3}$$

$$= \begin{bmatrix} 8 & 10 & 12 \\ 11 & 13 & 15 \end{bmatrix} \tag{4}$$

In this example, broadcasting is performed in (3) along the vertical dimension. We hypothesize that broadcasting is significant not only because it is used throughout CapsNet code, but also because it offers an opportunity for TC kernels to outclass CUDA libraries. TC notation does not require broadcasting due to its more intelligent summation. Those libraries are often made with certain memory access assumptions, like the fact that each element in a 2×3 tensor needs to be loaded, since they are all unique. When broadcasting happens, this is not the case, and the optimal kernel may look different due to the reduced memory cost since many of the elements are the same.

As a basic example, consider the implicit broadcasting of a vector multiplied by a scalar. The scalar could have its memory read by one thread and used by CUDA shared memory neighbors. We further hypothesize that optimizations exist for low dimensional cases but are unlikely to generalize to optimum kernels for higher dimensional tensors with multiple broadcast dimensions. It is impossible to say, since cuDNN is closed source. We test this hypothesis in many of our kernels, particularly in Section 3.5.

3. CapsNet Architecture and Optimizations

For this paper, we focus on optimizing the standard CapsNet that tackles the MNIST classification task (predicting digits $0-9$). We walk through the general architecture as well as the specific TC kernels and optimizations we explored. Figure 1 ([10]) serves as a visual reference for the CapsNet architecture. To skip the implementation details, a summary of all of our optimizations is in Section 3.6.

3.1. Setup

We based our implementation off of the open source Gram.AI implementation [5]. This is because it is the most starred PyTorch implementation on GitHub. The two other most popular implementations are already noticeable less efficient, because they fail to make use of higher dimensional operations [4, 8]. This may be due to simplicity of the code or ease of understanding since these networks are largely educational and not practical for anything yet.

We also note that many of these implementations are incorrect. One by higgsfield [4] fails to correctly compute the outputs of PrimaryCaps describe in 3.4. Gram.AI incorrectly computes the loss function over a $\mathrm{softmax}$ output instead of the pre-$\mathrm{softmax}$ values. This underscores the difficulty of understanding the architecture, making a correct and fast implementation all the more valuable.

We reuse the loss implementation from Gram.AI. We did not compare across models for other neural network frameworks, as it would not be as fair of a comparison, given different levels of library support. Our use of PyTorch is partially from its popularity as a library but also from its status as the only Python framework supported by TC.

In reimplementing Gram.AI's version [5], we noticed three areas where the basic PyTorch code seemed inefficient, and modified these areas with functionally equivalent, more efficient PyTorch code. These changes will be mentioned below as we go through the CapsNet architecture. By optimizing the baseline code, other than the obvious benefits, we can construct a more realistic measurement of the practical value of TC's performance improvements.

We looked for higher-dimensional, atypical tensor operations that we believed would yield the most potential benefit if implemented in TC, identifying 6 such operations in the CapsNet architecture. We also implement a standard convolution and ReLU operation in TC, which lacks theoretical motivation for optimization, but is used as an additional benchmark to test TC's effectiveness. The relevant dimensions used to explain these operations are as follows:

- $BA = 100$: batch size
- $C_0 = 256$: # output channels of Conv1 layer
- $C_1 = 8$: size of PrimaryCaps capsules' vector outputs
- $C_2 = 16$: size of DigitCaps capsules' vector outputs
- $RO = 32 \times 6 \times 6 = 1152$: # capsules in PrimaryCaps layer
- $CA = 10$: # capsules in DigitCaps layer (= # classes)

3.2. Defining the TC gradient kernel

One unfortunate consequence of using the TC library is that the corresponding tensor operation(s) cannot take advantage of automatic differentiation present in PyTorch or other deep learning frameworks. Rather, the gradient of the tensor operation must also

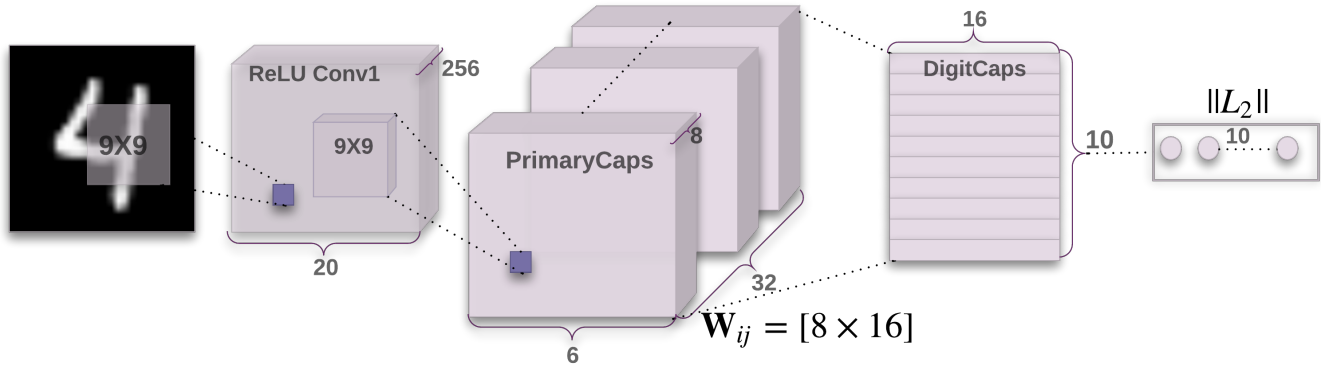

Figure 1. 3-layer CapsNet architecture proposed in Sabour *et al*. [10] to tackle the MNIST classification task. ReLU Conv1 is a standard convolutional layer. PrimaryCaps is a convolutional capsule layer. DigitCaps is a standard capsule layer. Dynamic routing of capsule outputs is performed between the PrimaryCaps and DigitCaps layers. This figure was taken from Sabour *et al*. [10].

be implemented in TC notation in order to be used in the backward pass during training. Additionally, we found that due to current limitations in the library, TC is currently unable to represent the backward pass of convolutions. For these and other operations where only the forward pass was implemented, the generated kernels were only used during inference and not during training.

3.3. Conv1 layer

The first capsule layer is a standard convolution layer with 9×9 kernels, a stride of 1, and ReLU activation. It takes in a batch of input images in the form of a tensor $\mathbf{I} \in \mathbb{R}^{BA \times 28 \times 28}$ and outputs a tensor $\mathbf{O} \in \mathbb{R}^{BA \times C_0 \times 20 \times 20}$ (there are C_0 total kernels).

As convolution followed by ReLU is a typical operation already optimized by cuDNN for GPU performance, we did not expect significant performance benefits from TC optimization but rather used it as a test to ensure that TC was able to generate a kernel with comparable performance. We wrote a equivalent TC implementation named CONV1 for the forward pass of this operation.

3.4. PrimaryCaps layer

The second capsule layer is a convolutional capsule layer made up of 32 feature channels where each channel is comprised of 8-dimensional capsules (instead of 1-dimensional features/node activations in a standard convolutional layer). The individual convolutions are 9×9 kernels with a stride of 2. The operation takes the tensor $\mathbf{O} \in \mathbb{R}^{BA \times C_0 \times 20 \times 20}$ output by the Conv1 layer and a tensor kernel $\mathbf{K} \in \mathbb{R}^{32 \times C_0 \times 9 \times 9 \times C_1}$ and outputs a tensor $\mathbf{X'} \in \mathbb{R}^{BA \times 32 \times 6 \times 6 \times C_1}$. After convolution, the output tensor is reshaped by collapsing the channels, height, and width dimensions into a single dimension to yield a tensor $\mathbf{X} \in \mathbb{R}^{BA \times RO \times C_1}$, on which the squash nonlinearity is run along the last dimension.

This high-dimensional strided operation does not perfectly match up with normal convolution due to the C_1 dimension at the end of the involved parameters. Therefore, it is cannot be optimized by some cuDNN convolution template and is thus a prime candidate for TC optimization which we call PRIMARYCAPS CONVOLUTION (TC). Unfortunately, after writing the TC kernel and attempting to optimize it, the TC autotuner kept hanging and failed to make progress in kernel optimization.

Within standard PyTorch, there are still many different ways of implementing this convolutional capsule layer operation. The reference implementation [5] (1) performs C_1 standard stride-2 convolutions with C_0 input channels and 32 output channels on a 20×20 image with a 9×9 kernel, resulting in a 6×6 region, (2) collapses the non-batch dimensions ($32 \times 6 \times 6$) to get one dimension of size RO, and (3) concatenates the C_1 output tensors along a new dimension to yield a $BA \times RO \times C_1$ output tensor. By the term collapse, we mean that we simply combine the indices, like how a 3×3 matrix can be equivalently represented as a vector of length 9. This is a free operation since no data is moved in the process, as long as the dimensions are adjacent. The reverse operation of splitting is also free for adjacent dimensions.

The reference implementation's repeated convolutions are a point of potential inefficiency, so we also test an implementation that instead (1) performs a *single* convolution with C_0 input channels and $32 \times C_1 = 256$ output channels with the same stride and kernel size, (2) splits the 256 channels into C_1 and 32 for free, (3) collapses the now adjacent dimensions to get RO, then transposes it to dimensions $BA \times RO \times C_1$. We call this optimization PRIMARYCAPS CONVOLUTION (PYTORCH).

3.4.1 Squash nonlinearity (TSquash)

While most conventional neural networks use ReLU, or $\max(x, 0)$, as their activation function, CapsNets use squash, which is defined on a vector as

$$\text{squash}(\vec{x}) = \frac{|\vec{x}|^2}{1 + |\vec{x}|^2} \frac{\vec{x}}{|\vec{x}|} \tag{5}$$

$$= \frac{|\vec{x}|\,\vec{x}}{1 + |\vec{x}|^2}. \tag{6}$$

The squash performed along the vectors of the C_1 dimension of the output tensor is another potential area for improvement. Furthermore, this operation can be combined with the previous transposition operation into a single kernel that performs both transposition and squashing. We implement this operation as a kernel that takes in a 3-dimensional tensor in $\mathbb{R}^{BA \times C_1 \times RO}$, applies the squash operation along the C_1 dimension, and transposes the tensor to dimensions $\mathbb{R}^{BA \times RO \times C_1}$. For this operation we only defined the forward pass in TC and call the optimization TSQUASH.

TSQUASH is an example of layer fusion. Layer fusion combines two or more operations that normally would occur in separate kernels. The performance improves largely because intermediate values do not require storage in global memory, thus saving time. The previously mentioned CONV1 optimization fused convolution and ReLU into one kernel, for example.

3.5. DigitsCaps layer

The third capsule layer is a capsule layer with a ($C_2 = 16$) 16-dimensional capsule for each digit class, each receiving input from all capsules in the previous layer. The DigitCaps layer first transforms the output it receives in the previous layer by performing a high dimensional tensor operation that takes in input $\mathbf{X} \in \mathbb{R}^{BA \times RO \times C_1}$ and weight tensor $\mathbf{W} \in \mathbb{R}^{CA \times RO \times C_2 \times C_1}$ and outputs $\mathbf{U} \in \mathbb{R}^{CA \times RO \times C_2}$.

This was the second area where we noticed a potential inefficiency in the reference PyTorch code. They perform matrix multiplication of the input tensor **X** by the weight tensor **W** instead of the other way around (multiplying **W** by **X**). The latter approach seems to be more standard as it is most similar to BLAS-like batch matrix-vector multiplication, whereas the Gram.AI version expressed an equivalent row vector-matrix multiplication, so we changed the weight dimensions and operation accordingly. We call this optimization **U** CALCULATION (PYTORCH).

The equivalent TC operations is as follows:

```
U(ba,ca,ro,c2) +=!
    W(ca,ro,c2,c1) * X(ba,ro,c1)
```

In addition to rewriting the PyTorch implementation of this operation, we also attempted to optimize it using the above TC formula. While this operation is similar to standard batch matrix-vector multiplication, the additional dimensions require broadcasting and reusing of the same matrices for multiplication multiple times, which entails multiple accesses of the same memory location, something that may not be optimized by standard kernels which usually only use each matrix and vector once. Because of this, we suspected TC may yield performance benefits compared to standard baseline kernels.

For this operation, we defined both the forward and backward passes of the tensor comprehension named **U** CALCULATION (TC), allowing optimized kernels to be used during both training and inference.

3.5.1 Permuting tensor dimension order

Since we represented **W** as a 4D tensor, as opposed to a more naive implementation which would treat it as a large bank of matrices, we have the option to permute the order of the dimensions of **W**. There are 24 orders for the 4 dimensions, with the inner (more right) dimensions being "more" contiguous in GPU memory than the outer (more left) dimensions. By testing all possible orders, we were able to find a more efficient kernel than just picking one of them for **U** CALCULATION (TC).

3.5.2 Dynamic routing algorithm

Between the PrimaryCaps layer output and DigitCaps layer output, the dynamic routing algorithm must be performed to determine what fraction of each PrimaryCaps capsule's output vector is sent to each DigitCaps capsule as input. The routing algorithm iteratively allocates the magnitude of vectors across DigitCaps capsules, feeding them toward capsules which agree with them (as measured by a dot product with the DigitCaps capsule's output). The CapsNet implementation we use has 3 iterations for routing, each iteration receiving $\mathbf{B_i} \in \mathbb{R}^{BA \times CA \times RO \times C_2}$ from the previous iteration with $\mathbf{B_0}$ equal to the zero tensor. The routing algorithm for each iteration i is as follows:

1. $\mathbf{C_i} \in \mathbb{R}^{BA \times CA \times RO \times C_2}$ is equal to the softmax of $\mathbf{B_i}$ along the RO dimension.
2. $\mathbf{S_i} \in \mathbb{R}^{BA \times CA \times C_2}$ is calculated by taking the element-wise multiplication of **U** and $\mathbf{C_i}$ and summing along the RO dimension. The equivalent TC formula is:

   ```
   S(ba,ca,c2) +=!
       C(ba,ca,ro,c2) * U(ba,ca,ro,c2)
   ```

3. $\mathbf{V_i} \in \mathbb{R}^{BA \times CA \times C_2}$ is calculated by taking squash of $\mathbf{S_i}$ along the C_2 dimension
4. $\mathbf{B_{i+1}} \in \mathbb{R}^{BA \times CA \times RO \times C_2}$ is calculated by taking the element-wise multiplication of **U** and $\mathbf{V_i}$ with $\mathbf{V_i}$ broadcast along the RO dimension and summing along the C_2 dimension to yield $\mathbf{B_{temp}} \in \mathbb{R}^{BA \times CA \times RO}$. This value is then added to $\mathbf{B_i}$ (by broadcasting $\mathbf{B_{temp}}$ along the C_2 dimension) to yield $\mathbf{B_{i+1}}$. The corresponding TC formula is:

   ```
   B_temp(ba,ca,ro) +=!
       U(ba,ca,ro,c2) * V(ba,ca,c2)
   B_new(ba,ca,ro,c2) =
       B_old(ba,ca,ro,c2) +
       B_temp(ba,ca,ro)
   ```

The third inefficient area was the implementation of the softmax used in step 1. They perform unnecessary transposes of $\mathbf{B_i}$ instead of using the built-in `dim` argument of PyTorch's softmax function. We corrected this in our implementation and call the optimization SOFTMAX.

We wrote TC optimization code for steps 2, 3, and 4 as they deal with high-dimensional matrix multiplications and reductions which suggest they may be potential points of optimization for TC. For step 2 we wrote TC kernels for both the forward and backward passes while for steps 3 and 4 we only wrote the forward pass. The TC optimizations are named as follows: 2 is **S** CALCULATION, 3 is SQUASH, and 4 is **B** CALCULATION.

After the last iteration of the routing algorithm, the final **V** is passed through three linear layers and used to infer the class and calculate loss. The details of this loss and the decoding step are not relevant to this paper and are available in Sabour *et al*. [10].

3.6. Summary of optimizations

In total, we attempt 3 PyTorch optimizations and 7 TC optimizations over the reference implementation. The PyTorch optimizations are:

1. **U** CALCULATION (PYTORCH). Reorder the tensor multiplication in the calculation of **U** to be **W@X** instead of **X@W** (@ is the PyTorch operator for matrix multiplication).

2. SOFTMAX. Fix step 1 of the dynamic routing algorithm to take advantage of the `dim` argument in PyTorch's softmax operation instead of performing unnecessary transposes.
3. PRIMARYCAPS CONVOLUTION (PYTORCH). Convert iterative convolutions and concatenation in the PrimaryCaps layer to a single convolution followed by reshaping and transposing.

The TC optimizations are:

1. CONV1. Perform standard convolution + ReLU for the first layer.
2. PRIMARYCAPS CONVOLUTION (TC). Perform capsule convolutions for the PrimaryCaps layer. Note that this TC kernel failed to autotune and so is not useable.
3. **U** CALCULATION (TC). Calculate **U** by performing equivalent of batch matrix-vector multiplication in TC.
4. **S** CALCULATION. Calculate **S** in the dynamic routing algorithm by taking element-wise multiplication of two tensors and summing across the RO dimension.
5. **B** CALCULATION. Calculate **B** in the dynamic routing algorithm by taking element-wise multiplication of input tensors with broadcasting, summing along the C_2 dimension, then performing broadcasted addition.
6. SQUASH. Perform standard squashing used in the dynamic routing algorithm.
7. TSQUASH. Perform both squashing and transposition in the PrimaryCaps layer after PRIMARYCAPS CONVOLUTION (PYTORCH).

4. TC kernel autotuning

TC can define a baseline inefficient kernel given an input string. However, if one knows the sizes of the input tensors in advance, one can use TC to automatically search for better kernels tuned for those sizes. TC uses a genetic algorithm approach, where a population of kernels which are defined by some parameters are bred over many generations to produce the fastest possible kernel.

At each generation, they are all benchmarked on GPU, and depending on their performance, a certain percentage of them will survive. Some number of the best performing kernels are guaranteed to survive to the next generation. The surviving kernels exchange some of their parameters in a manner similar to DNA, and seed the next generation of kernels. While the specifics of the algorithm and the translation of parameters to kernels are obscure TC implementation details, we control a few hyperparameters. For most kernels, we set the population size to 300, we guarantee the survival of 10 "elites" per generation, and we run the kernels for 5 generations. In the case of CONV1, which is standard convolution, we only use a population of 100. PRIMARYCAPS CONVOLUTION (PYTORCH) failed to run in the autotune system. We attribute this to the library's alpha state.

We use a population of 300 because we found repeated initializations of the standard population of 100 resulted in different local optima for kernel performance. By increasing the population, we can approximate many smaller repeated trials.

5. Benchmarking

There are several implementations of capsule networks that replicated the state of the art MNIST performance of [10] in PyTorch, but as described in Section 3.1, we compare to the Gram.AI implemention [5]. To ensure a fair comparison, we control for any differences in our implementation:

Hyperparameters. Many capsule network-specific parameters need to be held equal across tests between implementations. This includes the layers and sizes of the layers of the network, batch size, the number of dynamic routing iterations, and the training scheme (SGD with same nonzero weight decay and no regularization), so the same number of operations are performed.

PyTorch-specific details. We use the same parameters to initialize the `DataLoader`, which gets the MNIST dataset from disk into memory. Since this is a performance benchmark, we also ensure the usage of cuDNN and enable PyTorch's internal benchmarking-based kernel selection to give it the fastest speed possible without Tensor Comprehensions.

5.1. Measurement

For the purposes of accuracy and relevance, we measure the time to completion of training one epoch (iterating through, thus benchmarking the combined forward/backward pass), to assess improvements in training efficiency. We also measure the time to completion for inference through an entire epoch, which focuses just on the forward pass. This will be evaluated at varying batch sizes. These are the most important metrics because they reflect the end to end real cost of the two main operations one does with neural networks. That's why we choose not to separate the backward pass by itself, in addition to the fact that its timings are harder to correctly isolate.

5.2. Equivalence of optimizations

We made sure to verify that all PyTorch and TC optimizations were functionally equivalent to the reference implementation and still represented more efficient implementations/kernels.

To test for equivalence for kernels involving network parameters, we transferred weights between a Gram.AI implementation to our own model, then ran a forward pass of a dataset batch through both networks. We compared the maximum absolute difference in the output tensors and verified that it was negligible and attributable to floating point precision limitations. In this case, we looked at both the direct probability outputs given by vector magnitude as well as the reconstructions of the CapsNet decoder component (not described here).

For kernels that did not involve network parameters, we also conducted smoke tests with random input tensors to verify the outputs were the same across Gram.AI component, our PyTorch component, and sometimes the corresponding TC component.

We note that training is not sufficient to show correctness, as it is likely that networks with many parameters on MNIST will achieve high accuracy even with subtle implementation errors.

6. Experiments

In this section, we conduct a series of experiments to evaluate the performance effects of our optimizations, including our

Opt.	Dir.	Batches	Ref	Ours	Speedup
CONV1	f	100k	75.2	64.8	1.16
U CALC	f	50	0.872	1.67	0.52
U CALC	f/b	50	2.73	5.75	0.47
S CALC	f	1k	0.401	0.080	5.01
S CALC	f/b	1k	3.38	27.2	0.12
B CALC	f	1k	2.46	0.579	4.25
SQUASH	f	100k	6.75	6.11	1.10
TSQUASH	f	100k	43.7	6.48	6.74

Table 1. TC optimization results. Direction specifies if timing is performed for just forward or for both forward and backward passes. Times for reference and TC-optimized operations represent the time to perform the operation on the specified number of batches.

Opt.	Train	Test	Speedup
GRAM.AI	0.203	0.092	1.00 / 1.00
U CALC (PYTORCH)	0.174	0.066	1.17 / 1.40
ABOVE + SOFTMAX	0.124	0.045	1.64 / 2.07
ABOVE + PCAPS CONV	0.093	0.032	2.19 / 2.87
ABOVE + TC OPTS	—	0.024	— / 3.79

Table 2. Overall optimization results showing performance improvements in training and test times. Times are seconds/batch of 100 calculated by taking the average across 5 epochs for train and 10 epochs for test, excluding 1 epoch of warmup time in both cases. All times are adjusted by subtracting 0.004 seconds representing the amount of time to load the data. The two values in the Speedup column represent training/testing speedups.

performance-optimized TC kernels for the previously mentioned tensor operations and PyTorch optimizations over our reference implementation [5].

We used a GTX 1080 Ti, and made sure to also tune our kernels on the same type of GPU. Disk speed was not a factor in our experimental calculations since MNIST is loaded entirely to memory. For all end to end experiments for overall inference and training performance, we ran an epoch to warm up, then 5 epochs for training and 10 epochs for inference benchmarks. We used the Anaconda distribution of Python 3.6 and PyTorch 0.3.1 with cuDNN 7. We acknowledge that benchmarks on other GPUs or on other versions of cuDNN may have achieved different results.

6.1. TC optimizations

Table 1 shows the performance effects of the autotuned TC kernels on the tensor operation times when compared to the same operations without TC optimizations in our implementation (called Ref in the table). Note that **U** CALC represents the best performing kernel of the 24 permutations for the **U** CALC (TC) operation. Some of our TC kernels show clear improvements over existing kernels while others are not as efficient. We can clearly see that the two backward kernels generated by TC for **U** CALC (TC) and **S** CALC are not as efficient as existing cuDNN kernels, taking approximately 2X and 8X longer respectively. In general, we found that TC support for backward passes to be less complete than for forward optimization. Many standard operations such as convolutions cannot have their backward pass expressed easily or at all in TC, and the kernel generation for backward passes is largely inferior to default kernels even for non-standard high-dimensional tensor operations.

In addition to having a slower backward pass, **U** CALC (TC) is also approximately twice as slow in the forward pass as the baseline kernel. This suggests that the existing batch matrix-vector multiplication optimizations in cuDNN are applicable to this higher dimensional batch matrix-vector multiplication and so TC is unable to achieve better performance. It in fact achieves significantly worse performance. This may also be attributable to the current lack of implementations for some CUDA optimizations in TC that would be implemented in cuDNN.

CONV1, **S** CALC, **B** CALC, SQUASH, and TSQUASH all display performance increases in the forward pass, with **S** CALC, **B** CALC, and TSQUASH showing particularly dramatic speedups of 5.01X, 4.25X, and 6.74X respectively. These 3 kernels and the CONV1 kernel all take advantage of layer fusion which is likely the cause of their high performance compared to cuDNN kernels. CONV1 fuses convolution and ReLU, **S** CALC and **B** CALC both fuse element-wise multiplication with summation along a dimension, and TSQUASH fuses the squash operation with transposition. Since summation along a dimension and transposition require more complicated memory access logic than element-wise ReLU, it is understandable that the speedups generated by TC kernels are far higher for **B** CALC, **S** CALC, and TSQUASH than CONV1. Additionally, CONV1 as a standard convolution followed by ReLU is likely more optimized by cuDNN than the other higher-dimensional tensor operations.

SQUASH does not benefit from this concept of layer fusion as it writes to an intermediate variable and so has no obvious theoretical benefits over the PyTorch implementation, and yet it is still better. The 1.10X speedup for SQUASH is the lowest out of all our beneficial optimizations and is likely just due to TC finding a better standard kernel for this novel operation than cuDNN provides. One explanation is that cuDNN kernels may only have a few templated versions that do not fit CapsNet dimensions just right.

Initially when testing the TC kernels, we ran each for 50 batches and found them all to perform worse than the reference implementations. It was only when running them on more batches that we found performance benefits. We believe the constant costs associated with TC are higher than the reference implementations and thus running times with small batches are dominated by these constant factors, leading to deceptively low performance gains. An example of a constant factor associated with TC kernels and not with baseline kernels is the time required to load the kernel from the cache on disk.

6.2. Full network optimization

In addition to optimizing and analyzing individual tensor operations, we also measured the performance gains for the whole network training and testing times as a result of both our PyTorch and TC optimizations. Table 2 displays the speedups achieved by the addition of different optimizations. The effects of all optimizations are are calculated through accumulated application, applied in the order: **U** CALC (PYTORCH), SOFTMAX, PCAPS CONV, and TC OPTS.

The PyTorch optimizations all show significant speedups over reference [5], confirming our beliefs that the original implementations were inefficient. In particular, it is interesting to note how inefficient implementations of standard tasks such as matrix multiplication order and softmax can lead to significant performance gains with our corrections to these two components yielding a 1.64X speedup in training time and 2.19X speedup in test time. The addition of the PRIMARYCAPS CONVOLUTION optimization, converting the concatenation of repeated convolutions to the transposition and reshaping of a single convolution, also had large performance benefits further speeding up training by 33% and testing by 39%.

When calculating the effects of TC optimizations on the overall network training and testing time we decided to include them all at once as we have already analyzed their individual operation-scope speedups and wnat to evaluate the efficacy of TC as a whole on network inference time. In the table, TC OPTS includes the TC-generated forward pass kernels for CONV1, **S** CALC, **B** CALC, SQUASH, and TSQUASH since these are the kernels that showed performance gains. Note that there are no training times when including TC OPTS as they lack backward pass kernels, or the backward kernels are too slow. In total, these TC optimizations boost performance by 24.3% compared to our implementation with only PyTorch optimizations, yielding an overall network inference time speedup of 3.79X when compared to the reference implementation [5].

7. Conclusion

In this paper we sought to improve upon existing implementations for the CapsNet architecture by utilizing automatically optimized TC kernels to speed up high-dimensional tensor operations. Our model, including both PyTorch optimizations and TC optimizations, achieves a 2.2X speedup in training and a 3.8X speedup in inference when compared to our reference implementation. We also discovered that TC optimizations are particularly effective when taking advantage of layer fusion, combining multiple PyTorch operations into a single TC operation, and had the ability to accelerate performance by over 6X for some operations. Furthermore, we found that while TC shows significant performance boosts for inference, its ability to optimize kernels for the backward pass still lags behind cuDNN and occasionally fails altogether with the gradients of certain operations unable to be expressed within the current extent of TC semantics.

7.1. Future work

There are many potential directions for future work building off the results of this paper. We discovered a 4^{th} implementation inefficiency in our reference implementation in the use of **B** in the dynamic routing algorithm. While **B** is supposed to be implemented as a $BA \times CA \times RO$ tensor, the reference implementation defines it as a $BA \times CA \times RO \times C_2$ tensor, unnecessary replicating redundant information 16 times along the C_2 dimension. We believe removing this redundancy would yield performance benefits but were unable to do so in this paper due to time constraints.

In order to gain more insight into the efficient kernels generated by TC it may be useful to examine the actual C++ code for the CUDA kernels generated by TC. We found that TC has an option to print out the corresponding C++ code for its CUDA kernels but felt that analyzing them was beyond the scope of this paper.

This paper also acts as a proof of concept that TC can be used to optimize high-dimensional tensor operations which naturally leads into the exploration of using TC in similar architectures. One potential use case might be delving into 3D capsule networks where it is possible that TC optimizations may show even more significant performance boosts compared to cuDNN kernels due to the added complexity of the additional dimension.

References

[1] 5kk73 GPU assignment website 2014/2015.

[2] Broadcasting semantics.

[3] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *arXiv:1410.0759 [cs]*, Oct. 2014. arXiv: 1410.0759.

[4] higgsfield. Capsule-Network-Tutorial: Pytorch easy-to-follow Capsule Network tutorial, Apr. 2018. original-date: 2017-12-27T06:25:41Z.

[5] K. Iwasaki. capsule-networks: A PyTorch implementation of the NIPS 2017 paper "Dynamic Routing Between Capsules", Apr. 2018. original-date: 2017-11-02T14:47:54Z.

[6] A. Lavin and S. Gray. Fast Algorithms for Convolutional Neural Networks. *arXiv:1509.09308 [cs]*, Sept. 2015. arXiv: 1509.09308.

[7] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, Mar. 2008.

[8] T. Omernick. pytorch-capsule: Pytorch implementation of Hinton's Dynamic Routing Between Capsules, Apr. 2018. original-date: 2017-11-02T03:37:23Z.

[9] R. Raina, A. Madhavan, and A. Y. Ng. Large-scale Deep Unsupervised Learning Using Graphics Processors. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 873–880, New York, NY, USA, 2009. ACM.

[10] S. Sabour, N. Frosst, and G. E. Hinton. Dynamic routing between capsules. In *Advances in Neural Information Processing Systems*, pages 3859–3869, 2017.

[11] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv:1802.04730 [cs]*, Feb. 2018. arXiv: 1802.04730.