# A Memory Layout for Dynamically Routed Capsule Layers

Daniel A. Lopez[†], Rui Wu[§], Lee Barford[†‡], Frederick C Harris, Jr.[†]

[†]*Dept. of Computer Science and Engineering*
*University of Nevada, Reno*
Reno, NV

[§]*Dept. of Computer Science*
*East Carolina University*
Greenville, NC

[‡]*Keysight Laboratories*
*Keysight Technologies*
Reno, NV

daniellopez@nevada.unr.edu, wur18@ecu.edu, lee.barford@ieee.org, fred.harris@cse.unr.edu

*Abstract*—**Capsule Networks exploit a new, vector-based perceptron model, providing feature instantiation parameters on top of feature existence probabilities. With these vectors, simple scalar operations are elaborated to vector-matrix multiplication and multi-vector weighted reduction. Capsule Networks include convolutional layers which take the initial input and help it become a tensor. A novel data abstraction maps the individual values of this tensor to one-dimensional arrays but is conceptualized as a 2D grids of multi-dimensional elements. Moreover, loss function thresholds and architectural dimensions were arbitrarily set during the introduction of Capsule Networks. While current machine learning libraries provide abstractions for convolutional layers, a TensorFlow optimization requires structural overhead for a full Capsule Network implementation. They lack simple optimizations specifically for Capsule Network data allocation. This paper presents a scalable GPU optimization for the training and evaluation of Capsule Networks.**

## I. Introduction

Capsule Networks are a novel interpretation of neural networks where the perceptron model has been expanded to groups of artificial neurons which return vectors rather than scalars [8]. In this model, the values of the output vector correspond to specific instantiation parameters of the object being recognized in an image, whereas the length of the vector corresponds to the probability of the feature being detected existing in the image. Analogously to a conventional neural network, the capsule outputs from one layer (after undergoing some dimensionality transformation via transformation matrices) to the next are compiled via some weighted sum-reduction algorithm. The weights for this algorithm are computing by a routing-by-agreement algorithm which, similarly to K-means [5], grants a higher weight to vectors whose outputs tend to cluster together, thus exploiting the rareness of "agreeing" vectors in higher dimensions found in discriminatory learning [4]. Although the authors of [8] present this straightforward method, they also stress that there are many different ways to produce a similar output.

The presented CapsNet architecture in [8] features a convolutional layer, followed by a capsule layer, the latter of which contains a capsule for each feature class defined. The dynamic routing algorithm occurs between the two layers in a network, where the initial "output" vectors of the first capsule layer are composed of the same pixel value from many filter outputs from the previous convolutional layer. In machine learning libraries, individual tasks are distributively organized to accommodate many different devices and easy scaling from a single machine to a distributed system [1]. However, when implementing Capsule Networks, memory resources are wasted when the tensor representing the cube output from the convolutional layer is transformed via these tasks to a set of 8D vectors. Moreover, during reconstruction error generation, although tasks are modeled alongside their dependencies in a graph-based representation in an effort hide latency, redundant or unnecessary tasks may accidentally execute. This GPU acceleration method does not compute the reconstruction error.

The number of vector channels in the reshaping of the convolutional is varied, and thus, the number of filters in the PrimaryCaps layer is $d_l$ times multiplied by the number of vector channels. This paper proposes a set of CUDA kernels which aim to optimize the operations of a capsule network.

The rest of this paper is structured as follows: Background and related work is covered in Section II. Section III presents the methodology from data allocation through algorthmic definitions all the way to loss and activation functions. Results are presented in Section IV including timings, speedup and throughput calculations. Conclusions and future work follow in Section V.

## II. Background and Related Work

Given the embarrassingly parallel nature of some of the filter operations, operations on the convolutional layer may be facilitated with previous abstractions of multidimensional data. Such abstractions allow input and output to be distributed in nature for multiple GPUs [2], [3], [6]. Moreover, although other linear algebra optimizations exist, they rarely include optimizations for the bigger picture for which they will be used.

### A. Existing Tensor Flow Implementations

TensorFlow provides an API to a model-loss centric framework; a model is defined as a directed acyclic graph of tasks that eventually lead to a loss function which is minimized through a solver [1]. This generalization allows it to attempt to complete tasks that are independent of one another in parallel, as well as distributing a task across many nodes and potential multi-core devices. Generalized as it may be, this has serious pitfalls, as resources may be wasted on needless tasks, simply because a later dependent task decides not to used based on other input. The framework seizes the control flow and may waste computational power on values that are thrown away. However, the solvers in this framework still enable a Tensorflow implementation to reach around 90% accuracy within hours sequentially, and within minutes with GPU computation.

Moreover, TensorFlow back-end computation is accelerated by cuDNN, a deep neural network library.

GPU-accelerated primitives specific for convolutional neural networks (as well as other deep neural network formats) are provided in cuDNN, a library released by NVIDIA [2]. These primitive include layer definitions forward and backward propagation for convolutional layers which are accelerated through very specific indexing filter values allocated to these layer. As with normal batching techniques, the first most dimension of their tensor primitive is allocated for the batch size in all tensors used in computation [4].

The host-only API provided enables the initialization and utilization of the filter kernel with no human interaction, enabling speedup for their back-end processing. Although it is generally not required to manually update or check the individual values in these tensor, the authors found it easier to implement a simple, straightforward convolutional pass. In this implementation, the input (image), filter, and output tensors are allocated sequentially.

An admittedly performance-hindering method; back propagation is done with the use of built in "atomicAdd" functions, since the same filter value has an effect on multiple output values during forward propagation and thus, gets the same influence in back propagation. In Section V, using these primitives is considered.

### B. Computational Walk through

The CapsNet Architecture is defined as a convolutional layer (PrimaryCaps), followed by a capsule layer, which reinterprets the output of the convolutional layer as its own output, followed by a smaller capsule layer (DigitCaps). The principle novel computation in capsule networks lies in the operations between capsule layers during the forward propagation stage. Here, lower-level, lower-dimensional capsules undergo dimensional transformation and a dynamically weighted reduction to become the output of a higher-level, higher-dimensional capsule network. This transformation is analogous to multiplying individual scalars by weights in feed-forward, fully-connected layers.

The lower-dimensional capsules has "outputs" by the feature map output of a convolutional layer with ReLU activation. In PrimaryCaps, the number of filters must be divisible by the lower dimension to reshape the feature map outputs as vector maps. For referential integrity, the lower dimension, $d_l$, will be 8, and the higher dimension, $d_{l+1}$ will be 16. Therefore, depth wise, $d_l$ sized increments may be considered a vector map, and the number of lower level capsules is $d_l$ times the number of vector channels. The number of vector maps (and consequently, the number of lower-level capsules) are varied in this paper to study the speed up effectiveness. Furthermore, these methods will use the MNIST data set of hand written digits of $28 \times 28$ pixels.

*1) Forward Propagation:* First, the lower level capsules, $j$, produce an output vector, $\hat{\mathbf{u}}_{j|i}$, for each higher level capsule, $i$, once for each possible output class, estimating the parameters of *their* output vector, $\mathbf{v}_i$, which is transformed to the dimensional space of the higher layer by an evolved transformation matrix, $\mathbf{W}$, such that $\hat{\mathbf{u}}_{j|i} = \mathbf{W}_{ij}\mathbf{u}_i$. Each higher level capsule then computes a dynamic weighted sum of these vectors as their output, $\mathbf{v}_j$.

For a vector to calculate its output, the weighted sum result, $\mathbf{s}_j$, undergoes a vector squishing activation function, $\mathbf{v}_j = \frac{\|\mathbf{s}_j\|^2}{1+\|\mathbf{s}_j\|^2}\frac{\mathbf{s}_j}{\|\mathbf{s}_j\|}$. This is analogous to the sigmoid activation functions usually applied onto the weighted sums in traditional capsule networks.

The dynamic weights, $\mathbf{c_i}$, are updated by computing the log prior probabilities, $\mathbf{b_i}$, which are iteratively updated. During each iteration, the probabilities, $\mathbf{b_i}$ are incremented by the scalar product of the activated weighted sum, $\mathbf{v}_j = squash(\mathbf{s}_j), \mathbf{s}_j = \sum_i c_{ij}\hat{\mathbf{u}}_{j\|i}$, and the vector in question, $\hat{\mathbf{u}}_{j\|i}$.

The final capsule vectors outputted encode the probability of the existence of that feature in the length of the vector, while encoding the instantiation parameters of the pose of that feature in the orientation of the vector. The orientation parameters are heavily determined by the transformation matrix, which are optimized using $\mu$ momentum, instead of the Adam optimizer. The network will classify an image as being part of class $i$ from $k$ classes with the maximum length, $\|\mathbf{v_i}\|$.

*2) Back Propagation:* In back propagation, a corresponding error value is computed as a factor of the length of the vectors. This error value is the combination of the gradient of the loss function, multiplied by the derivative of the vector squashing activation function, effectively representing the error gradient towards which the free $\mathbf{W}$ values scattered throughout the network collectively inch towards. These are weighted by their respective $c$ values to produce the error gradients, $\delta\mathbf{u}_{j|i}$, for all $i$ output capsule vectors.

The error gradients are then transformed into the dimension of the lower level capsules, by being multiplied by the transpose of the original transformation matrix, $\mathbf{W}_{ij}^T$. Before then, however, a matrix product of these error gradients, and the original inputs to the network become part of the $\Delta\mathbf{W}_{ij}$. The output of the lower level capsules, however, are in truth, the rearranging of the output of the earlier convolutional layer. Back propagation occurs in this layer as normal.

*3) Sequential Batch Updating:* Given the time it takes to train a network, and the potential bias the ordering of the training examples gives to the network, different techniques were created in order to speed up processing and reduce potential bias and equalize the change all training examples provide. The latter is important to greatly increase the chances the network will converge to a more global optima. Mini-batching is one such technique.

In mini-batching, a batch of input images are provided to the network, where forward, and subsequently, back propagation are computed in parallel to one another. Afterwards, the resulting $\Delta\mathbf{W}$'s are reduced from all these "layers" to provide one main error, by which the network is updated. This paradigm is used in machine learning frameworks such as TensorFlow.

All matrices and vectors used in these computations are allocated as 1D arrays and indexed very precariously in the proposed method. Traditional mini-batching compilations of images would include higher complexities in these indexing. Moreover, the reduction of these $\Delta\mathbf{W}$'s from multiple devices would increase the communication needed between the host and all potential devices, thereby reducing potential scaling benefits. Therefore, this method does not use this technique, but rather computes forward and back propagation for each image sequentially, accumulating the error in $\mathbf{W}$ and then applying it to $\mathbf{W}$ at the end.

## III. Methodology

The network is trained with several forward and back propagation passes for images from a training data set with periodic weight updates. To go through all data points is an epoch, and several epochs are performed in an effort to minimize the overall network loss, and conversely, maximize accuracy. Network accuracy evaluation is done after training, where a different testing data set is used to eliminate biased estimations.

### *A. Data Allocation*

All data structures are allocated using Unified memory, where data movement is optimized by the device scheduler. Since everything is allocated with 1D arrays, memory management and organizing is highly significant, and this method presents one way to arrange the data.

Data may be thought of as a $k \times t$ grid of potentially multi-dimensional elements in row-major ordering, where an element $i, j$ corresponds to class $i$ and lower level capsule $j$. An example of the data layout may be found in Fig. 1, where $\hat{\mathbf{u}}$ is being created for each higher-level capsule column-wise, from each lower-level capsule row-wise. The lower level capsule outputs, $\mathbf{u}_{i,j}$, are represented in the middle tensor, and are duplicated along each column. Although this is potential memory storage waste during forward propagation, the extra storage will be used to save appropriate $\delta\mathbf{u}_{i,j}$ during back propagation.

For the capsule layer interface operations, there are a total of two 1D element grids, $\mathbf{b}$ and $\mathbf{c}$, three vector-element grids, $\mathbf{u}$, $\hat{\mathbf{u}}$, and $\mathbf{v}$, ($\mathbf{v}$ has a height of 1, and shares the dimensionality of $\hat{\mathbf{u}}$) and three matrix-element grids, $\mathbf{W}$, $\Delta\mathbf{W}$, and $\mathbf{W}_{\text{velocity}}$, the latter two of which are used for updating.

The preceding convolutional layer, however, requires simpler, sequentially indexed (channel, then depth if applicable, then height, then width) of 3 and 4 dimensional arrays. These are required for the input, $\mathbf{x}$, the output, $\hat{\mathbf{x}}$, and the filters of the arrays. Much like the $\mathbf{W}$ in the capsule layer, the filters have two other equally sized companion arrays, to hold the errors, and the velocities required in momentum updating.

### *B. Algorithmic Definitions*

In Procedure 1, forward propagation is given the image as a vector, $\mathbf{x}$, and requires the use of the dynamic routing procedure defined in [8].

The *Rearrange* method interprets the output tensor from the convolutional layer as a list of vectors as captured down the output depths. These vectors are then undergo the vector squash activation function, the non-linear function which facilitates discriminatory learning by scaling vector lengths close to zero and long vectors closer to one. Afterwards, as is illustrated the middle tensor shown in Fig. 1, these vectors are duplicated along the "columns", representing each of the DigitCaps classes. The Hadamard product, represented by $\otimes$, then produces distinct $\hat{\mathbf{u}}_i, j$ used in dynamic routing.

**Algorithm 1** Forward Propagation

```
1: procedure FP($\mathbf{x}$)
2:     $\hat{\mathbf{x}} \leftarrow PrimaryCaps.FP(\mathbf{x})$
3:     $\mathbf{u} \leftarrow Duplicate(Activate(Rearrange(\hat{\mathbf{x}})))$
4:     $\hat{\mathbf{u}} \leftarrow \mathbf{W} \otimes \mathbf{u}$
5: return $Routing(\hat{\mathbf{u}}, 3, 2)$          ▷ This is defined in [8]
```

On the other hand, during back propagation, the corresponding label to the vector, $y_{\mathbf{x}}$, is provided to calculate the error functions. This is performed by the *DerivativeActivationAndLoss* kernel. With the remaining $\mathbf{c}$ values which were set during dynamic routing, a $\delta\mathbf{u}$, corresponding to lower level capsules are generated by multiplying $\mathbf{v}$ with the corresponding transpose transformation matrix, $\mathbf{W}^T$, and scaled with $\mathbf{c}$. Before continuing, however, $\Delta\mathbf{W}$ needs to be incremented by the matrix product of the previous input $\mathbf{u}^T$ and the error gradient for the output, $\delta\mathbf{v}$. After $\delta\mathbf{u}$ has been calculated for all $j$ along the columns, they are reduced to the left, to compile all the error gradients proposed by each higher level capsule, before having each undergo another "unsquashing". This inverse activation is performed to match the initial activated squashing done during forward propagation. Finally, these vectors are rearranged and handed back to the convolutional layer for tradition convolutional back propagation.

**Algorithm 2** Back Propagation

```
1: procedure BP($y_{\mathbf{x}}$)
2:     $\delta\mathbf{v} \leftarrow DerivativeActivationAndLoss(\mathbf{v}, y)$
3:     $\delta\mathbf{u}_{ij} \leftarrow \delta\mathbf{c}_{ij}\mathbf{W}_{ij}^T\delta\mathbf{v}_j$
4:     $\Delta\mathbf{W}_{ij} \leftarrow \Delta\mathbf{W}_{ij} + \delta\mathbf{v}_j\mathbf{u}_{ij}^T$
5:     $\delta\hat{\mathbf{x}} \leftarrow DerivativeActivation(ColReduction(\delta\mathbf{u}))$
   return $PrimaryCaps.BP(\delta\hat{\mathbf{x}})$
```

### *C. Loss and Activation Functions*

The error function applied to the back vectors include the derivative of two functions, the loss function aforementioned and the partial derivative of the vector activating-squash function. The loss function and its derivative include $T_k$ which is set if an instance of class $k$ is present in the image. The original vector activation squash-function and its corresponding derivative may be seen in Eq. 1 and Eq. 2.

The activation function multiplies a non-linear squashing scalar with a normalized vector. The partial derivative function

$$\begin{vmatrix} \mathbf{W}[0,0] & \mathbf{W}[0,1] & \dots & \mathbf{W}[0,k] \\ \mathbf{W}[1,0] & \mathbf{W}[1,1] & \dots & \mathbf{W}[0,k] \\ \dots & \dots & \dots & \dots \\ \mathbf{W}[t.0] & \mathbf{W}[t,1] & \dots & \mathbf{W}[t,k] \end{vmatrix} \bigotimes \begin{vmatrix} \mathbf{u}[0] & \mathbf{u}[0] & \dots & \mathbf{u}[0] \\ \mathbf{u}[1] & \mathbf{u}[1] & \dots & \mathbf{u}[0] \\ \dots & \dots & \dots & \dots \\ \mathbf{u}[t] & \mathbf{u}[t] & \dots & \mathbf{u}[t] \end{vmatrix} = \begin{vmatrix} \hat{\mathbf{u}}[0,0] & \hat{\mathbf{u}}[0,1] & \dots & \hat{\mathbf{u}}[0,k] \\ \hat{\mathbf{u}}[1,0] & \hat{\mathbf{u}}[1,1] & \dots & \hat{\mathbf{u}}[0,k] \\ \dots & \dots & \dots & \dots \\ \hat{\mathbf{u}}[t.0] & \hat{\mathbf{u}}[t,1] & \dots & \hat{\mathbf{u}}[t,k] \end{vmatrix}$$

Fig. 1. The $d_{l+1} \times d_l$ transformation matrices, shown in the left most tensor, are multiplied element wise with the $d_l$ dimensional outputs from the lower level capsules. These outputs are stored column-wise in the middle tensor, but are duplicated by column-wise for each higher-level capsule. The Hadamard product of these tensor produces $d_{l+1}$ sized vector inputs to the higher level capsules in the right most tensor.

$$squash(\mathbf{s}_j) = \frac{\|\mathbf{s}_j\|^2}{1 + \|\mathbf{s}_j\|^2} \frac{\mathbf{s}_j}{\|\mathbf{s}_j\|} \tag{1}$$

$$\frac{\partial}{\partial\|\mathbf{s}_j\|}[squash] = \frac{2\|\mathbf{s}_j\|}{(\|\mathbf{s}_j\|^2+1)^2} \frac{\mathbf{s}_j}{\|\mathbf{s}_j\|} \tag{2}$$

$$L_k = T_k \max(0, m^+ - \|\mathbf{v}_k\|)^2 + \lambda(1 - T_k)\max(0, \|\mathbf{v}_k\| - m^-)^2 \tag{3}$$

$$\frac{d}{d\|\mathbf{v}_k\|}[L_k] = \begin{cases} -2T_k(m^+ - \|\mathbf{v}_k\|) & \|\mathbf{v}_k\| < m^+, \|\mathbf{v}_k\| \le m^- \\ 2\lambda(T_k - 1)(m^- - \|\mathbf{v}_k\|) & \|\mathbf{v}_k\| \ge m^+, \|\mathbf{v}_k\| > m^- \\ 2(\lambda(T_k - 1)(m^- - \|\mathbf{v}_k\|) + T_k(\|\mathbf{v}_k\| - m^+)) & \|\mathbf{v}_k\| < m^+, \|\mathbf{v}_k\| > m^- \end{cases} \tag{4}$$

Eq. 1 shows the original activation algorithm proposed, whereas its derivative is shown in Eq. 2. Eq. 3 is the loss function proposed by [8] as a function of the length of output vector. Eq. 4 is the derivative of the loss function with respect to the length of the instantiation vector. Note the normalization factor, $\frac{\mathbf{s}_j}{\|\mathbf{s}_j\|}$ remains present in both equations. Constant hyper-parameters: $m^+ = 0.9$, $m^- = 0.1$, $\lambda = 0.5$

is determined by the scalar portion of the activation function, and effectively scales the error from the loss function to the length of the output instantiation vector. The original vector loss function may be seen in Eq. 3 and its appropriate loss function is Eq. 4.

## IV. Results

To measure the given speed up of these methods, capsule networks were generated with varying numbers of vector channels. This effectively changes the number of lower level capsules by multiples of the convolutional layer's height and width, which were set to 6 for referential integrity's sake. The time taken to perform forward propagation, backward propagation and epoch timings are reported, along with the equivalent speed up and throughput calculations.

Varying the number of channels in the tensor also helps better profile the program better through its throughput measurements. Reported below are the time taken to perform forward propagation, backward propagation and epoch timings, along with the equivalent speed up and throughput calculations. All timings seen in Figure 2 and Figure 3 are an average 30 statistical runs, after removing the highest and lowest outliers. To ensure sequential optimization, the sequential version uses the Armadillo library for linear algebra operations (matrix-vector multiplication) [9].

In Figure 2, back propagation is shown to have a higher percentage of data computation rather than data movement, since data movement is not a hindering factor on CPU-based operations. Parallel timings in Figure 3 further illustrate the GPU bottleneck since back propagation is faster than forward propagation. The single image trend lines show the computation time of processing an entire image, combining forward and back propagation.

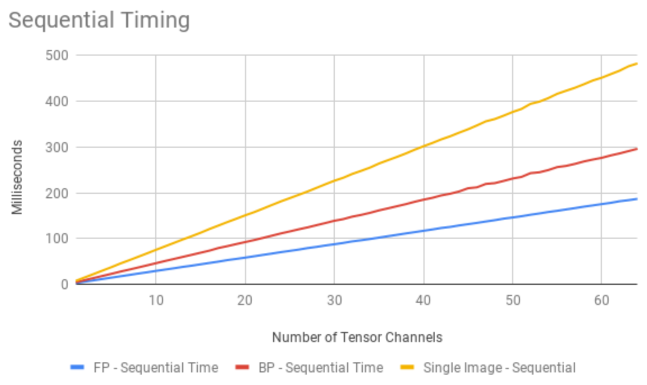

Fig. 2. Forward propagation takes less time than back propagation in the sequential version of these methods since there is no data movement (despite the inner loop found in Dynamic Routing [8]. This indicates back propagation is more computationally intensive. These methods are not multi-threaded.

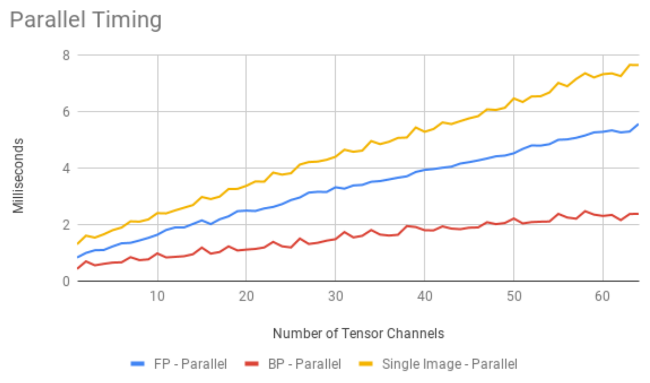

Fig. 3. Back propagation is clearly faster than forward propagation due to lack of communication overhead in data movement. However, the steadily increasing gap between the two also indicates computation overhead in back propagation scales better with these methods than forward propagation.

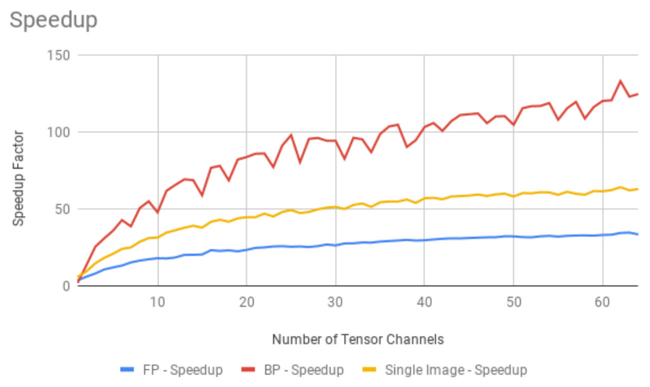

Fig. 4. Speedup of these methods start to slow down between 10-15 tensor channels (360-540 lower level capsules) as these methods increase due to Amdahl's law. Note that back propagation, which only communicates resulting $\mathbf{v}_j$ errors back to the host, a constant $k \times d_{l+1} = 160$ values, has higher speedup than forward propagation, which requires the movement of a $28 \times 28$ sized image from the MNIST dataset. [7].

The sequential version of these methods are completed with the help of the Armadillo library for linear algebra operations (matrix-vector multiplication) [9]. Fig. 2 shows, however, back propagation is shown to be more computationally intensive, since data movement is not a hindering factor on CPU-based operations. Parallel timings in Fig. 3 further illustrate this point, as near linear trends are also shown, similar to the sequential version, but back propagation is clearly faster than forward propagation, although not by much. However, the gap between forward propagation and back propagation steadily increases over larger capsule layer sizes, despite the amount of data being transferred being the same. This indicates the back propagation problem scales better to GPUs as opposed to forward propagation. This comparison is made more apparent since forward propagation adds initial data transfer overhead.

## A. Speedup

To measure the speedup, capsule networks were generated with varying numbers of vector channels. This changes the lower level capsule layer size, dependent on the convolutional layer height and width, both set to 6 for the sake of referential integrity. Thus, the number of rows in the $\mathbf{u}$ grid (and other grids) is based off the height (6), the width (6), and the tensor channel size. For the equal tensor channel values found in [8], forward propagation obtained 32x speedup and back propagation obtained 116x speedup. These methods were able to obtain up to 33x speed up for forward propagation and 130x speed up for back propagation procedures alone.

## B. Throughput

Traditionally, efficiency is calculated in multi-CPU application to measure resource and memory exploitation in distributed algorithms. However, GPU speedup is accompanied with throughput; how many floating point operations (FLOPS) are computed in a given amount of time. For GPUs, throughput focuses on the bandwidth of the data flow rather than hardware architecture; an more appropriate, important distributed algorithm metric. For the graph seen in Figure 5, throughput

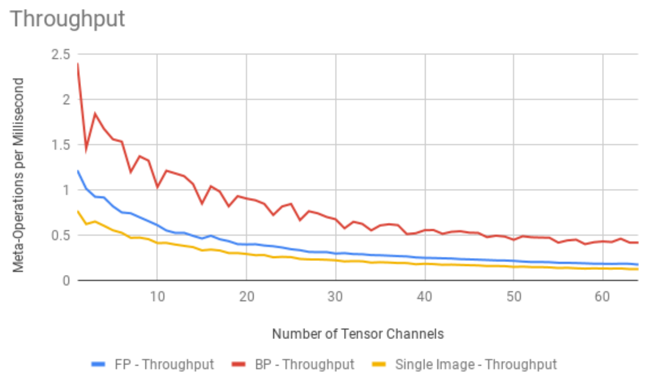

Fig. 5. The throughput is measured by a factor the number of floating points required to compute (not the operations) at the variable layer divided by the amount of time taken to complete the meta-operation. These floating points are the ones for the interim layer only.

was computed solely on the amount of computation used during the variable layer, the convolutional output to tensor, for easy comparison. In forward and back propagation, this equates to a $6 * 6$ grid, multiplied by the appropriate number of depth channels, divided by the processing time in seconds. This is similar for single image processing, where throughput is obviously slower due to the concatenation of these two operations. The warp-based valleys found during the speed up of the program also make an impact here; computation is less efficient when hardware resource allocation is not optimized. There is no surprise that back propagation can produce a higher throughput than forward propagation, even when the hidden layers are increased. Contributing to this advantage are lack of required CPU-to-GPU data communications and no iterative dynamic routing necessary.

## V. Conclusion and Future Work

This paper illustrates data layout methods, and shows their effectiveness in CUDA based architectures for Capsule Networks. Although cumbersome, these methods provide increased manipulation of lower level arrays while maintain high level grid abstraction intact; increasing flexibility than machine learning libraries, such as TensorFlow.

Though other major batch-based techniques were not used, single data point processing was increased almost 20x for 40 vector-tensor channels, equating to 1440 lower level capsule, each having unique $8 \times 16$ ($d_l \times d_{l+1}$) weight matrices. Increasing another dimension of complexity may become a single GPU issue, given the data limits and how the size of these transformation $\mathbf{W}$'s grows. However, in a future, heterogeneous distributed version of these methods, data points may be partitioned per device, and $\Delta\mathbf{W}$'s may be reduced between these devices, as a single point of communication during batch updating.

### A. Using cuDNN Primitives

The methods in this paper involves low-level organization of the equivalent weight elements between capsule networks, and the rearrangement of incoming convoluted vector data from the convolution layer. Although cuDNN primitive use very specific ways of arranging their tensors in memory without a clear way of accessing the values in the convolutional layer filters themselves, their use as a convolutional layer to this method could aid in potential future speedup. However, the tensor indexing for individual values must be either transformed to reshape as the tensor shown here. To lower kernel call overhead even further, some of the original computation kernels may be rewritten to output into the required tensor shape.

### B. Parallel Mini-Batching

To achieve multiple image processing for each forward and back propagation step, an extra dimension in the proposed arrays could be used to hold the same information for varying inputs. After a backward pass, all values for $\Delta\mathbf{W}$ would have to be reduced along this new axis, however, before being applied to $\mathbf{W}$. The complexity of this reduction problem increases depending on both the solver and potential distributed systems, for memory restrictions, and bottleneck data movement overhead, respectively.

### C. Multiple Parameter Varying

In efforts to remain truthful to source material, $d_l$ was set to 8 and $d_{l+1}$ was set to 16. These values seem to be arbitrary, as no real explanation as to the dimensionality is mentioned. However, as the orientations of the vectors in these space represent the instantiation of the perceived feature in each capsule, it could be argued that the limited space helps the vector "point" in the most likely direction. Therefore, increasing these values will most likely give more space and wiggle room for these outputs. Nevertheless, the $\mathbf{W}$ matrices contain $d_l \times d_{l+1}$ degrees of freedom; increasing this may add too much volatility to the space of transformations.

## References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[2] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[3] Adam Coates, Brody Huval, Tao Wang, David J. Wu, Andrew Y. Ng, and Bryan Catanzaro. Deep learning with cots hpc systems. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III–1337–III–1345. JMLR.org, 2013.

[4] Wei Di, Anurag Bhardwaj, and Jianing Wei. *Deep learning essentials: your hands-on guide to the fundamentals of deep learning and neural network modeling*. Packt Publishing, Birmingham, UK, 1 edition, 2018.

[5] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7):881–892, 2002.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[7] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[8] Sara Sabour, Nicholas Frosst, and Geoffrey E. Hinton. Dynamic routing between capsules. *CoRR*, abs/1710.09829, 2017.

[9] Conrad Sanderson. Armadillo c++ linear algebra library, June 2016.