A New Approach to Compute CNNs for Extremely Large Images

Sai Wu
Zhejiang Univeristy
Hangzhou, Zhejiang, P.R.China
wusai@zju.edu.cn

Mengdan Zhang
Zhejiang Univeristy
Hangzhou, Zhejiang, P.R.China
zh_mengd@163.com

Gang Chen
Zhejiang Univeristy
Hangzhou, Zhejiang, P.R.China
cg@zju.edu.cn

Ke Chen
Zhejiang Univeristy
Hangzhou, Zhejiang, P.R.China
chenk@zju.edu.cn

ABSTRACT

CNN (Convolution Neural Network) is widely used in visual analysis and achieves exceptionally high performances in image classification, face detection, object recognition, image recoloring, and other learning jobs. Using deep learning frameworks, such as Torch and Tensorflow, CNN can be efficiently computed by leveraging the power of GPU. However, one drawback of GPU is its limited memory which prohibits us from handling large images. Passing a 4K resolution image to the VGG network will result in an exception of out-of-memory for Titan-X GPU. In this paper, we propose a new approach that adopts the BSP (bulk synchronization parallel) model to compute CNNs for images of any size. Before fed to a specific CNN layer, the image is split into smaller pieces which go through the neural network separately. Then, a specific padding and normalization technique is adopted to merge sub-images back into one image. Our approach can be easily extended to support distributed multi-GPUs. In this paper, we use neural style network as our example to illustrate the effectiveness of our approach. We show that using one Titan-X GPU, we can transfer the style of an image with 10,000×10,000 pixels within 1 minute.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; **Image processing**; • **Information systems** → *Query optimization*; • **Theory of computation** → *Distributed computing models*;

KEYWORDS

Convolution Neural Network; Style Transfer; Multi-GPUs

1 INTRODUCTION

From AlexNet [12], Inception network[18] to ResNet [8], CNN (Convolution Neural Network) shows its superior performance in computer vision analysis. It can even outperform humans for particular jobs [8]. One recent interesting application of CNN is to transfer the art style between images [6]. Given a content image and a style image, we try to generate a new image by forwarding all three images through the VGG network [16] and minimizing the style loss and content loss. Since the original approach is too expensive and time-consuming, Justin Johnson et. al. [10] and Dmitry Ulyanov et. al. [19] design different feedforward networks to facilitate the prediction process. In particular, in the learning process, a feedforward network works together with the VGG network to learn the style representations for a specific style image. In the prediction process, we simply forward a content image through the feedforward network to obtain the result image.

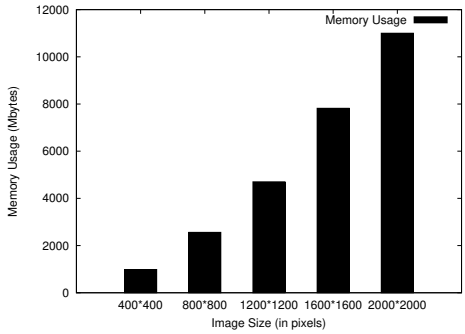


Figure 1: Memory Usage for Style Transfer Network

The new style transfer network can generate a result image with 1,200×1,000 pixels in one second using a Titan-X GPU. Total GPU memory usage is about 3GB. However, to print a 1m×1m poster with 150dpi, we need an image with at least 6,000×6,000 pixels, and the memory usage is about 80GB. We cannot generate such image using a single Titan-X GPU as it only has 12GB memory. We are only able to run our model on CPUs with an extremely long waiting time. Note that the main storage overhead is not caused by parameters in our trained neural network. In fact, one saved model of style transfer network is only about 6MB when trained using Tensorflow. But when a large input image is fed to our model, the convolution result of each CNN layer increases dramatically and uses up all GPU memory.

This is a common problem for neural networks that employ CNNs to process images. The memory requirement increases linearly with the size of an image. Figure 1 shows the memory usage of Johnson's style transfer network [10] which sets up a residual

CIKM'17 , November 6–10, 2017, Singapore, Singapore

ACM ISBN 978-1-4503-4918-5/17/11…$15.00
https://doi.org/10.1145/3132847.3132872

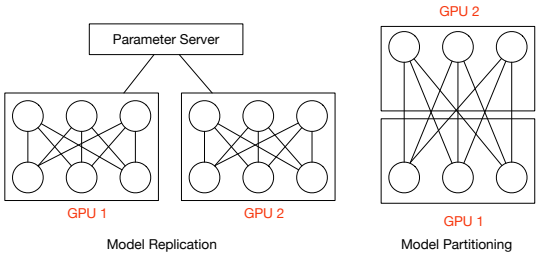


Figure 2: Distributed Learning Model

network with 16 CNNs. Due to limited memory, current GPUs can only process CNN for small to middle size images.

To enable CNN computation for extremely large size images, one possible approach is to distribute the training model to multiple GPUs [4]. For instance, deep learning frameworks such as Tensorflow and Torch support two types of distributed learning, *model replication* and *model partitioning*. As shown in Figure 2, model replication replicates the model to each GPU along with one partition of training data . All GPUs train their own copy of the model and communicate in an asynchronous way, so that they can merge their learned parameters. Model partitioning, on the other hand, maintains a single training model by splitting it into multiple compute-intensive parts and assigning one to each GPU. Those GPU workers can learn the model in a synchronous way.

Model replication can significantly speed up the learning process. But it does not reduce the model size. Therefore, it cannot be used to process CNNs for large images. Model partitioning strategy distributes the model to multiple GPUs, so each GPU only processes a portion of the model. It is a potential solution. However, it does not improve the learning performance. On the contrary, it slows down the whole training and prediction process, because the system incurs high communication overhead when parameters are exchanged between GPUs.

In this paper, we propose a new approach to process CNNs for extremely large images. Our approach can use a single GPU to handle images of any size. The basic idea is straight-forward, namely just splitting an image into equal-size small pieces and computing CNNs for each piece. However, the challenges lie in how to combine the results of those image pieces using correct normalization and padding strategy. Our approach adopts the BSP (bulk synchronization parallel) model by synchronizing the computation after each CNN layer. The main synchronization cost is the cost of shuffling parameters from GPU memory to main memory. Fortunately, after we split images into pieces, the number of parameters has been significantly reduced and so is the synchronization cost. We propose two optimization techniques, delayed normalization and image sampling, to further reduce the processing cost. One advantage of our approach is that it can be extended to multi-GPUs seamlessly. Each GPU is assigned one or multiple image pieces for processing independently. All GPUs can process the convolution computation in parallel with very few synchronization cost. As our experiments show, this type of parallelism effectively improves the performance and is an order of magnitude faster than the in-graph replication approach.

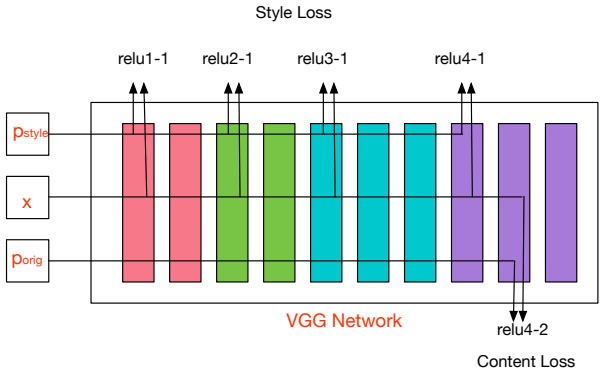


Figure 3: Neural Style Network

In this paper, we use style transfer network as our example to illustrate the idea. However, our approach can be applied to any CNN which is used to process large images. The remaining of the paper is organized as follows. Section 2 introduces some background knowledge about CNN and style transfer network. In Section 3, we present the main idea of our approach. Section 4 evaluates our approach and compares with in-graph replication. Section 5 briefly reviews related work, and we conclude the paper in Section 6.

2 STYLE TRANSFER NETWORK

In this section, we review the basic idea of style transfer network, since we use it as an example to illustrate our approach. Specifically, Leon A. Gatys et. al. [6] designed a CNN network to transfer art styles of famous artists (e.g., Vincent van Gogh) to any new input image. In particular, let p_{orig}, p_{style} and x denote the original image, style image and generated image, respectively. The style transfer network maintains the content similarity between p_{orig} and x, and the style similarity between p_{style} and x. The goal is achieved by forwarding all three images to a VGG-19 network and minimizing the content loss and style loss.

Leon A. Gatys et. al. found that different layers of VGG generate different types of features for the image. The low level layers normally represent textures, colors, brush strokes and other style features, while the high level layers describe the semantics of an image. This is because VGG network is an image classification which can potentially identify objects from images. Based on the observation, the convolution results of VGG layers { ReLU1_1, ReLU2_1, ReLU3_1, ReLU4_1} are used as the style features. And the results of layer ReLU4_2 are used to represent the image content. Figure 3 illustrates the general idea.

Let $F_i^l(x)$ denote the ith feature map in the lth layer of the VGG network for image x. We use $F_{i,j}^l(x)$ to represent the jth value of the ith feature map. The content loss function is defined as the squared-error loss between the two feature representations at layer l:

$$L_{content}(x, p_{orig}, l) = \frac{1}{2}\sum_{i,j}(F_{i,j}^l(x) - F_{i,j}^l(p_{orig}))^2$$

$L_{content}$ measures the pixel-pixel similarity in the lth layer. As Figure 3 shows, we only compute the content loss for layer ReLU4_2.

Figure 4: Original Image (Image is from http://www.nipic.com/show/13005551.html)

Figure 5: Style Image with Naive Splitting

Figure 6: Style Image with BSP Model

Different from the content loss, the style loss should be insensitive to spatial features. So gram matrix G_l at layer l is defined as:

$$G^l_{i,j}(x) = \sum_k F^l_{i,k}(x) F^l_{j,k}(x)$$

It computes the inner product between the ith and jth feature map in layer l. The style loss between p_{style} and x at layer l is computed as:

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G^l_{i,j}(x) - G^l_{i,j}(p_{style}))^2$$

N is the number of feature maps and M is the size of feature map (height times width). The general loss function is a weighted sum of all layers, namely { ReLU1_1, ReLU2_1, ReLU3_1, ReLU4_1}:

$$L_{style}(x, p_{style}) = \sum_{i=0}^{L} w_i E_l$$

Finally, the neural style network is trained to minimize the total loss:

$$L(x, p_{style}, p_{orig}) = \alpha L_{content}(x, p_{orig}) + \beta L_{style}(x, p_{style})$$

α and β are tunable parameters to balance the importance between content similarity and style similarity.

Gatys' approach can produce high quality results. However, its major drawback is that we need to run the training process for every input image. A typical training process with 2000 iterations takes about 20-30 minutes for a small image. This prohibits us from providing an online service. To address the problem, Justin Johnson et. al. [10] and Dmitry Ulyanov et. al. [19] proposed different feedforward networks to speed up the processing.

The intuition of their idea is similar to the adversarial networks [7]. Basically, the VGG loss network is consider as the "Discriminator", who judges whether the generated image x is equal to p_{orig} in content and p_{style} in style respecitively. For a given style image p_{style}, the feedforward network works as the "Generator" to create synthetic images to cheat the Discriminator. Once the training process completes, we do not need the Discriminator anymore. Any new input image can be stylized as p_{style}.

Both Justin Johnson and Dmitry Ulyanov's network consist of a set of CNNs. In this paper, we use Justin Johnson's network to illustrate our approach and its performance. In particular, our generator network contains five residual blocks [8], four convolution layers and two deconvolution layers. Table 1 shows the structure (the input/output image is 256×256. If not, we will scale them to a particular size.). The discriminator network is still the VGG network shown in Figure 3.

Table 1: Structures of Generator

Layers	Descriptions
Input Image	size 256×256
convolution layer	32 features, kernel size 9×9
convolution layer	64 features, kernel size 3×3
convolution layer	128 features, kernel size 3×3
5 residual blocks	all 128 features
deconvolution layer	64 features, kernel size 128×128
deconvolution layer	32 features, kernel size 256×256
convolution layer	3 features, kernel size 256×256

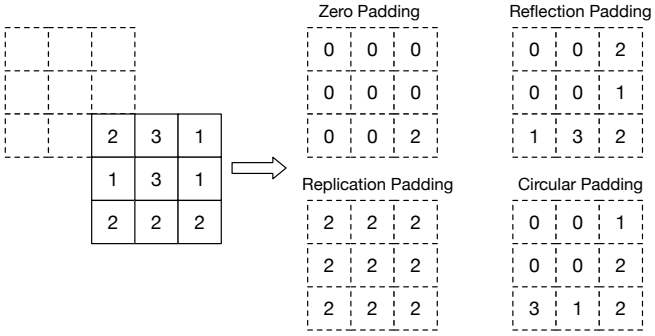


Figure 7: Padding Techniques

3 BSP MODEL FOR CNN

In this section, we present the general idea of our distributed CNN computations. As a matter of fact, our approach follows the BSP (Bulk Synchronous Parallel) model. A CNN layer is split to workers and computed in parallel. A synchronization process is used to collect results from workers and merges them to generate statistics for the whole image. The synchronization process then distributes the statistics to all workers and start a new round of computation. Before delving into details of our approach, we first discuss a simple solution.

3.1 Naive Image Splitting Approach

A straightforward way to address the memory shortage problem is to split an image into equal-size portions. Given an input image with size $W \times H$, we can split it into equal-size sub-images horizontally or vertically. E.g., we can set two scale factor, f_w and f_h, to split the image, and the size of each sub-image is $\frac{W}{f_w} \times \frac{H}{f_h}$. f_w and f_h are determined based on the size of available GPU memory. In particular, CNN computation of each $\frac{W}{f_w} \times \frac{H}{f_h}$ sub-image cannot overrun our GPU memory. In the same time, we should reduce also the number of sub-images as much as possible to improve the performance.

For example, given the image shown in Figure 4 (resolution is 1024×640), we can split it into four 256×640 images (namely, $f_w = 4$ and $f_h = 1$). In this way, the image size is reduced by four and so is the requirement for GPU memory. In style transfer network, those sub-images are passed to the pre-trained feed-forward CNN networks to produce stylish images, which are further combined back as a 1024×640 image. However, we find that the result image (Figure 5) shows clear segmentations at edges of sub-images. For more results, please refer to Figure 10 to 13 in our experiments.

The reason is two-fold. First, when computing CNN for images, we normally adopt some padding techniques to fill in missing pixel values. There are four widely used padding techniques, namely zero padding, replication padding, reflection padding and circular padding. In Figure 7, given a 3×3 image, suppose we use a 3×3 kernel to do the convolution. When computing for the first element, we need to fill in 8 empty pixel values. Different padding approaches will give us different results.

- Zero padding just simply fills the outer padded region with all zeros.
- Replication padding replicates the nearest pixels to the outer padded region.
- Reflection padding reflects the input image along the intersect border.
- Circular padding applies the reverse reflection strategy. E.g., reflecting the right side of the image to the left padded region and reflecting the bottom side of the image to the top padded region.

By default, to perform convolution for the whole image, we normally use the replication padding or zero padding. However, such padding approaches will sharpen the image border, which causes the segmentation effect between sub-images.

Second, in convolution network, batch normalization technique is applied to speed up the convergence and reduce the internal covariate shift [9]. Specifically, let $B = \{x_1, x_2, ..., x_n\}$ be the values of a mini-batch. The variance is computed as:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \frac{1}{n}\sum_{j=1}^{n}x_j)^2 \tag{1}$$

The output value of normalization is then computed as:

$$y_i = \gamma \frac{x_i - \frac{1}{n}\sum_{j=1}^{n} x_j}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{2}$$

γ and β are learned in the neural network. ϵ is a small constant for numerical stability.

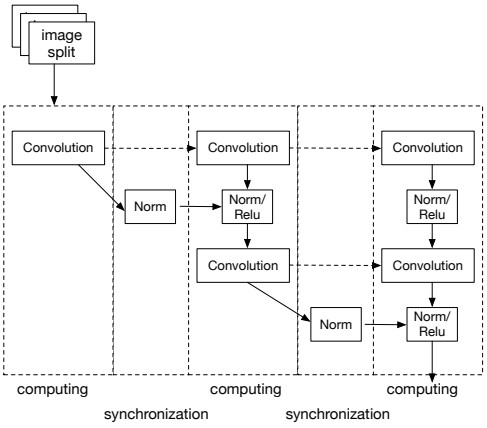


Figure 8: BSP Model

Now, suppose we split the image into two sub-images. We get two small batches $B_1 = \{x_1, ..., x_m\}$ and $B_2 = \{x_{m+1}, ..., x_n\}$. If B_1 and B_2 go through the CNN independently, we will obtain two normalization results which only apply sub-images' local means and variances in the computations. Therefore, even if we merge those pieces back to one image, we will end up with a result that is visually different from the one generated for the whole image.

3.2 BSP Model

The above discussion shows that to enable the split-and-merge CNN computation, we must redesign our padding and normalization process. In this paper, we learn from the distributed graph processing system [15] and adopt the BSP model.

3.2.1 Adaptive Padding. To smooth textures near image edges, we adaptively select a proper padding approach. Given a sub-image, we use different padding techniques for its borders when performing the convolution. We call a border internal border, if it is between two small images. On the other hand, a border is called external border, if it is a portion of the border of the whole input image.

- For internal border, we use a novel padding technique, dictionary padding. In particular, for padding pixels along an internal border, we will use values in the corresponding position of original image. E.g., let $img(i, j)$ returns the pixel value at column i and row j of the original image. If the left-top pixel of a sub-image is at position (x, y) in the original image, the jth top padding value along the ith column of the sub-image is $img(i, y - j)$.
- For external border, we use circular paddings to address the recursive pattern problem. Namely, bottom objects of an image can be connected to top objects in the same image.

We also set a padding width to further smooth the transition between sub-images. Suppose we use a kernel with size $M \times M$ to compute the convolution, a typical padding width is $M - 1$. Therefore, we add $M - 1$ more pixels to each border of the sub-image. So the size of the image becomes $(\frac{W}{f_w} + 2M - 2) \times (\frac{H}{f_h} + 2M - 2)$. We find that a larger padding width normally results in a better transition result between sub-images. In this paper, we set padding

width to 64. We also experiment with other padding widths and find that setting padding width to 50 is enough for most images.

3.2.2 Two-Phase Processing. Figure 8 shows the general idea of our BSP approach. The whole computation consists of two phases, computation phase and synchronization phase. In the computation phase, each sub-image goes through one convolution layer independently. If we have multiple GPUs, we can assign one sub-image to one GPU and do the computation in parallel. As mentioned before, normalization values of sub-images are different from those computed for the whole image. To address the problem, we add a synchronization phase after each computation phase. All sub-images send their current convolution results out to CPU, which has a much larger memory and will compute the global normalization values for them. In next computation phase, sub-image will use new normalization values to continue their next layer convolution processing.

This strategy follows the idea of BSP model. We establish a worker thread for each sub-image (Algorithm 1). If multiple workers are assigned to one GPU, we need to load and save current neural network results of each specific worker to disk, because we do not have enough GPU memory to maintain them all. The thread transfers the input image into a three-channel (RGB values) matrix and loads the pre-trained CNN model into GPU. It iteratively enters the computation phase (Algorithm 4) which does the real convolution work. This may be costly for a large image. At the end of each computation, we transfer the convolution results to the master thread on CPU.

The master thread on CPU helps all GPU workers synchronize their process (Algorithm 3). After it receives responses of all GPU workers, the master thread starts the normalization computation. Normalization results are forwarded back to GPU workers, who can start the next layer computations.

Algorithm 1 Worker(Image *img*, Model M)

```
1: Matrix V = ImageToMatrix(img)
2: while TRUE do
3:     loadModelToGPU(M)
4:     if M.nextLayer=NULL then
5:         return
6:     Compute(V, M)
7:     V = WaitForMessageFromCPU()
8:     M.save()
```

Algorithm 2 Compute(Matrix V, Model M)

```
1: if not M.firstLayer then
2:     Layer L = M.getNextReLULayer()
3:     V = L.doReLU(V)
4: Layer L = M.getNextCNNLayer()
5: Matrix R = L.doConvolution(V)
6: SendToCPUMaster(R)
```

3.3 Optimizations

We observe that the main bottleneck of our basic approach is the I/O cost of saving and loading data model and the transmission cost between CPU and GPUs. Each GPU may need to process multiple sub-images and even one sub-image's intermediate results will use up all available GPU memory. Repeatedly forwarding those intermediate results to CPU for synchronization incurs high overhead. Therefore, we propose two optimization techniques to reduce the data exchanged between CPU and GPUs.

Algorithm 3 Master(int K)

```
1: MessageSet S = null
2: int layer_no=0
3: while TRUE do
4:     Message m = waitForNextMessage()
5:     S.add(m)
6:     if S.size==K then
7:         var, mean = getStatistic(S)
8:         for i=0 to K do
9:             Matrix V = doNormalization(var, mean, S[i])
10:            SendToGPUWorker(V, S[i])
11:        layer_no++
12:        S.removeAll()
```

3.3.1 Delayed Normalization. To compute normalization results, we scale the image convolution results based on the global mean and variance of the whole image (Equation 1 and 2). In fact, only the global mean and variance need to be synchronized between sub-images. Convolution results of each sub-image are not necessarily sent to CPU. We can compute the global mean and variance from the local mean, variance and number of pixels per feature in each sub-image. As we split the image into equal-size sub-images, the number of pixels of a feature at the same layer are all same for sub-images. In this way, we only need to send local mean and variance to CPU for synchronization, which can compute the global mean and variance.

This approach significantly reduces the overhead of synchronization. Less information is transferred between CPU and GPU memory. However, since we no longer send out the convolution results to CPU, we cannot do the normalization at our CPU. Instead, the normalization is delayed until each GPU worker receives the global mean and variance. So the *Compute* function is revised as below:

Algorithm 4 Compute(Matrix V, Model M)

```
1: V=M.load()
2: var, mean = getLastStatisticFrom()
3: V = doNormalization(var, mean, V)
4: if not M.firstLayer then
5:     Layer L = M.getNextReLULayer()
6:     V = L.doReLU(V)
7: Layer L = M.getNextCNNLayer()
8: Matrix R = L.doConvolution(V)
9: M.save(R)
10: SendToCPUMaster(R)
```

One advantage of this approach is that if we do the normalization on GPU, we can exploit its parallelism to speed up the processing, which is much more efficient than doing normalization on CPU.

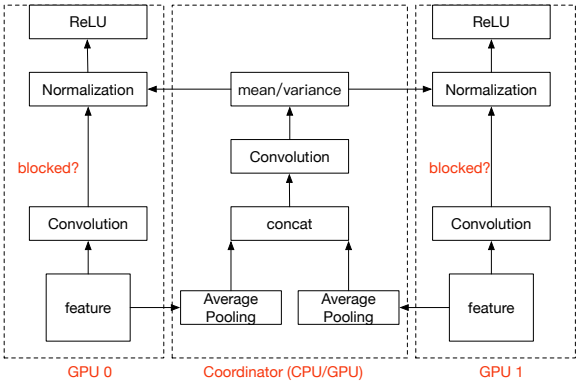


Figure 9: Normalization via Sampling

3.3.2 Normalization via Sampling. The delayed normalization technique reduces the synchronization costs between GPUs and CPU. However, after computation of each CNN layer, GPU workers are still blocked until receiving the global variance and mean value. In most cases, the number of sub-images is an order of magnitude larger than the number of available GPUs. Each GPU, on the other hand, can only maintain the model of one sub-image in its memory. Therefore, if the computation of one sub-image is blocked for synchronization, we need to evict its model to disk. So another sub-image can be loaded into GPU for processing. After receiving the global variance and mean value, the GPU needs to load the intermediate results of each sub-image one by one to perform the normalization. This type of periodically saving and loading incurs high I/O overhead, especially for a large size of image, and compromises the benefit of using parallelism.

To address the problem, we adopt an approximate approach. We observe that the statistics of CNN intermediate results are very much similar for an image with different resolutions. For instance, suppose the original image has a resolution of 512×512. We can obtain a new image by adopting a simple average/max pooling technique to downsize the image to 256×256. If both images pass through the same CNN layer (namely, using the same set of kernels), the mean and variance of generated feature pairs (features generated by the same kernel for two images) are approximately the same.

This observation motivates our "normalization via sampling" approach. Figure 9 shows the general idea. Suppose we split the input image into N sub-images (x_0,...,x_{N-1}). Let $F_i^l(x_j)$ denote the ith feature map in the lth layer of the VGG network for image x_j. We define feature group $\mathcal{G}_i^l$ as

$$\mathcal{G}_i^l = \{F_i^l(x_j) | 0 \leq j < N\}$$

Feature group $\mathcal{G}_i^l$ combines all the ith feature maps of sub-images at layer l. By concatenating them together, we actually reconstruct the ith feature map for the whole image. Therefore, if we send all feature groups to a coordinator, the coordinator can do the normalization for the whole image. However, that is still expensive and not applicable, since this approach does not reduce the size of features. To address the problem, we apply the pooling technique. Before fed to the next convolution layer, each feature map generated by previous layer is sampled via an average pooling approach. We use $\pi(F_i^l(x_j))$ to represent the pooling result of feature $F_i^l(x_j)$. After pooling, the coordinator obtains a smaller feature group $\bar{\mathcal{G}}_i^l = \{\pi(F_i^l(x_j)) | 0 \leq j < N\}$, which is a down-sampled feature map. After obtaining all feature groups, the coordinator performs the convolution as the normal GPU worker. Because we are manipulating much smaller feature maps, we have enough memory to hold all necessary intermediate results, and the computation is also less expensive.

After the convolution, the coordinator computes mean and variance of the results, and forwards them to GPU workers to do the normalization. Note that this is an approximation of the normalization, because we simulate the mean and variance of the whole image convolution result by using the ones computed from down-sampled image features. However, the results show that the approximation can produce good enough result.

In this approach, GPU workers perform their convolution concurrently. After they finish their convolution processing, they are blocked, since they need to wait for statistic results from the coordinator to do the normalization. In practice, blocking is seldom triggered because coordinator is handling much smaller features and can always complete its processing before GPU workers. So in most cases, GPU worker can start their normalization immediately after the convolution computation completes.x

The sample image size may affect the final convolutional results. In our experiments, for an image with resolution 10,000×10,000, we down-scale it to a sample image with 1,000×1,000 pixels. The results are almost the same compared to the non-sampling approach. However, if we further reduce the size of sample image to 512×512, we observe that a lot of details of the image are missed. Hence, how to select a proper sample image is still requiring more research. Currently, we just generate a sample image in the highest possible quality that can be still processed by one GPU.

4 EXPERIMENTS

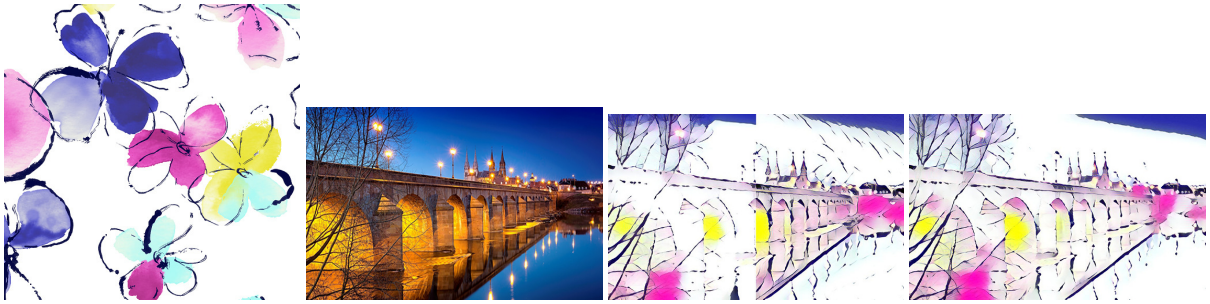
We deploy our model on a server equipped with 8 Titan-X GPUs, 2 4-Core Xeon CPUs, 32GB DRAM, 2TB disk and 512GB SSD. We implement our schemes on Tensorflow 1.0. The style transfer network is trained using the MSCOCO dataset[1]. We reuse part of Logan Engstrom's codes[2]. The default settings are shown in below Table 2.

For comparison, we implement the original fast style transfer network proposed by Justin Johnson et. al.[10], denoted as *basic* approach in our diagram. We also implement a model partitioning approach, which splits different CNN layers to different GPUs. In particular, we can assign the first two CNN layers to GPU 1, the third and fourth to GPU 2 and so on. After we complete the computation of the first two layers, we forward intermediate results to GPU 2, which can continue the computation. However, we find that this is even slower than a single GPU, since we need to shuffle a lot of parameters among GPUs. Instead, we just use a single GPU to perform a single layer computation each time. After the

[1] http://mscoco.org/explore/
[2] https://github.com/lengstrom/fast-style-transfer

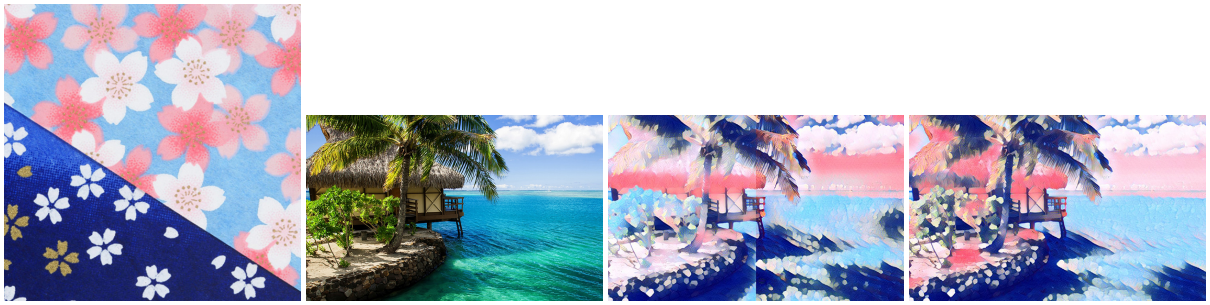
(a) Style Image (b) Content Image (c) Result of Naive Approach (d) Result of Our Approach

Figure 10: Result of Style Transfer 1



(a) Style Image (b) Content Image (c) Result of Naive Approach (d) Result of Our Approach

Figure 11: Result of Style Transfer 2



(a) Style Image (b) Content Image (c) Result of Naive Approach (d) Result of Our Approach

Figure 12: Result of Style Transfer 3

computation finishes, the GPU saves current layer to DRAM and loads the next layer to resume the computation. In this way, we avoid shuffle parameters between GPUs. We represent this approach as *layer partitioning* approach in our diagram. In our experiments, we test both the basic BSP approach and the BPS with sampling approach.

4.1 Image Style Transfer Results

We first check whether our approach can effectively generate large graphs. We implement a naive split-merge approach. It simply partitions the image into multiple sub-images and passes each sub-image to the style-transfer neural network. Then, the styled results are merged back to one image. In Figure 10 to 12, we show the style image, input image, style-transferred results using naive approach and style-transferred results of our BSP approach. In all results of naive approach, we can observe a clear segmentation between

(a) Style Image (b) Content Image (c) Result of BSP with Sampling (d) Result of BSP

Figure 13: Result of BSP with Sampling

(a) Style Image (b) Content Image (c) Result of BSP with Sampling (d) Result of BSP

Figure 14: Result of BSP with Sampling

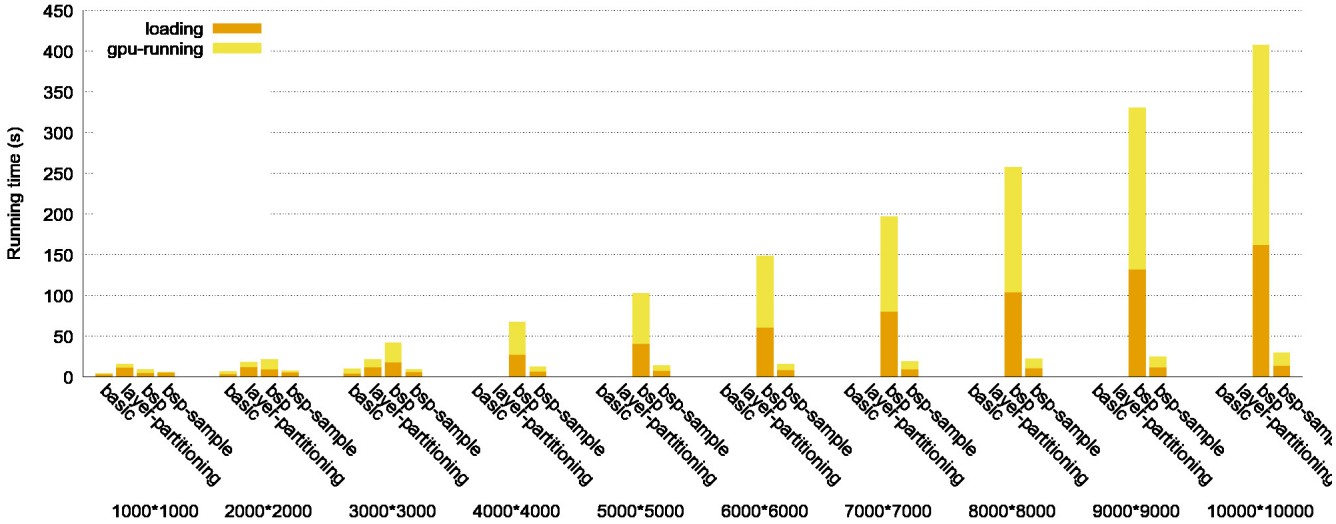


Figure 15: Effect of Image Size

image splits, especially in the background, while our BSP approach can produce good styled images. This shows that by adopting the new padding and normalization technique, BSP approach can effectively smooth the sub-images and simulate the CNN computations on the whole image.

Once we complete the computation of one CNN layer for all sub-images, we need output their results to CPUs to perform the normalization. To avoid high computation overheads during the synchronization process, we apply the sample-based normalization approach. Instead of waiting for the CNN computation of the whole

Table 2: Experiment Settings

Parameters	Default Values
Sub-Image Size	256×H (H is the height of original image)
GDDR Size	8GB
VGG Network	VGG19
Feed-forward Network	Five Residual Blocks[10]
Training Image Size	256×256
Style Image Size	512×512

image, we use a small sample image to get the approximate normalization values. This can dramatically reduce the processing cost. However, it may affect the final styled images. In Figure 14 to 15, we show the result images with/without sampling. They are almost identical, except that the images with sampling lack some texture details compared to the non-sampling approach, but the quality defect is within tolerance.

4.2 Performances of Different Approaches

In this experiment, we compare the performance of four different approaches for image style transfer. We range the image size from 1,000×1,000 to 10,000×10,000. In other words, we increase the test image size by two orders of magnitude. Figure 15 summarizes the results. We record two types of costs. The model loading cost refers to the initial cost of reading the neural model from disk to GPU and rebuilding the model graph. This mainly incurs the I/O cost and GPU GDDR access cost. The GPU computation cost refers to the cost of computing the CNNs, which also includes the synchronization cost for BSP approaches. In Figure 15, we can see that BSP and BSP-sample approach can generate images of all sizes. The other two approaches, the basic and layer-partitioning, cannot produce images above 3,000×3,000 pixels. For basic approach, we need to load the full image into GDDR and go through all CNN layers. So it consumes huge GPU memory. To our surprise, we thought that layer-partitioning should support larger images. But it does not perform any better than the basic approach. It can produce images at most in 3,500×3,500 pixels. This is because for a large image, some CNN layer generates too many features which cannot be maintained in GPU memory. In other words, even computing a single CNN layer for a large image can overwhelm the GDDR.

For BSP and BSP-sample, the loading cost and GPU computation cost increase for larger images. However, compared to BSP, BSP-sample is still very efficient. It can even output a 10,000×10,000 image within 50 seconds, while BSP requires more than 400 seconds. The performance gap further increases, if we test for larger images. This result shows that BSP-sample approach is more scalable with regards to the image size.

Finally, we test the performance of our approaches on multi-GPUs. We increase the number of GPUs from 1 to 8 and the results are shown in Figure 16. Specifically, we partition the images into equal-size sub-images and randomly assign sub-images to GPUs. After completing the computation of one sub-image, the GPU will ask for the next one. If all sub-images have been processed, we perform the normalization and resume the computation for the next CNN layer. We find that simply increasing the number of GPUs does not necessarily improves the performance. First, the loading cost cannot be reduced by employing more GPUs, since each GPU needs to fully loaded the neural models (e.g., VGG network). Second, the GPU cost can be partially reduced, because each GPU now handles fewer sub-images. However, such improvement is neutralized by the synchronization cost. More GPUs lead to higher communication overhead. We should synchronize the processing of those GPUs and send back the general normalization results to them. So in our case, the best result is achieved when we use 6 GPUs. For BSP-sample approach, when we increase the number of GPUs to 2, we almost half the computation cost. However, the data loading cost soon becomes the bottleneck. So further increasing the number of GPUs does not introduce any further improvement.

5 RELATED WORK

5.1 Distributed Framework

Training deep neural network is very expensive, sometimes lasting for several days or a few weeks. E.g., training the VGG network on a computer with 4 titan-black GPU takes about 2-3 weeks [16]. One possible solution is to split the training work into tasks and disseminate them to different machines. Motivated by this idea, deep belief network [4] is designed by Google, which proposes the “parameter-sever" architecture. It partitions the training data among several machines and replicates the neural model to each of them. Adam [3] further extends the idea by optimizing and balancing workload computation and communication. It uses 30x fewer machines to train a large 2 billion connection model for ImageNet classification job. Singa [20] supports more complex distributed training model, where machines are grouped for a specific task and different groups can synchronize their process. Besides the distributed computation, another solution to improve the efficiency of neural network training is via the design of new hardware [2]. This is beyond the topic of this paper.

5.2 Deep Convolutional Neural Network

Convolutional Neural Network (CNN) can preserve spatially local correlations when generating features, which makes it a perfect fit of image processing. It was shown that the model consisting of multiple CNNs tends to outperform any existing approaches in image classification. AlexNet [13], VGG [16], GoogLeNet [17], Inception network[18] and ResNet [8] are recent image classification CNN models, which represent the state-of-the-art techniques in image classification. One trend of those models is that they become deeper and deeper by integrating more CNN models. For instance, ResNet can support more than 1,000 CNN layers and still can be trained in a manageable way. Due to their high accuracies, those models are also used as a pre-trained model to improve the accuracy of image segmentation, object recognition and image processing jobs. Besides image processing, CNN is also adopted in NLP processing [5][11], speech processing [21] and auto-driving [1]. It is becoming one of the most popular neural networks in real applications.

5.3 Style Transfer

Style transfer enables a user to apply the art style of a famous artist (e.g., Vincent van Gogh) to his/her own photos. The core idea is to iteratively transform the result image, so that the loss among

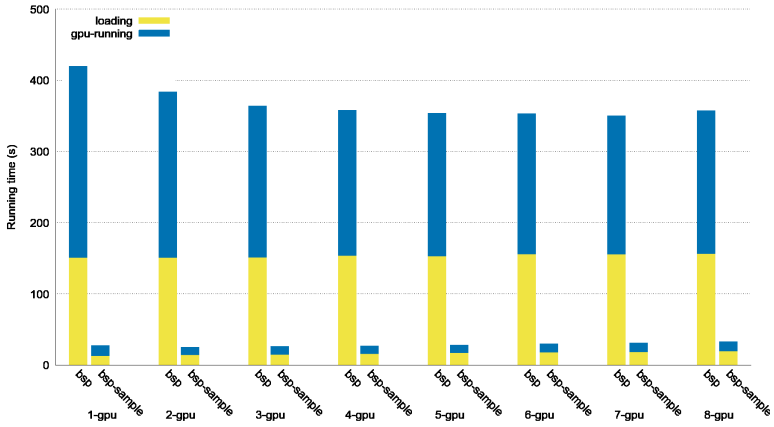


Figure 16: Effect of Number of GPUs

the result image, style image and input image is minimized. Gatys et. al. [6] uses the pre-trained VGG network to compute the losses, which is costly since we need to start the training process for every different input image. As a result, Johnson et. al. [10] and Ulyanov et. al. [19] develop different CNN-based feed-forward networks. The feed-forward networks, once trained for a specific art style, can generate new result images efficiently. This strategy is similar to the GAN (Generative adversarial networks) [7]. The feed-forward network is the generator, while the VGG network is the discriminator. Recently, Adobe proposes a style transfer solution for realistic photos [14]. Before transforming an image, it first identifies semantic objects, so we can apply the style textures to the corresponding objects.

6 CONCLUSION

Convolutional Neural Network (CNN) is becoming one of the most popular neural network structures. However, it is normally expensive to train. In this paper, we use the style transfer application as an example to show how our new CNN computation approach works. Our approach can use a single GPU to handle images of any size, or it can leverage multi-GPUs to significantly improve the performance. It splits an image into equal-size small pieces and computes CNNs for each piece. A new normalization and padding strategy is proposed to combining the partial results. Our approach adopts the BSP (bulk synchronization parallel) model by synchronizing the computation after each CNN layer. Our experiments show that we can improve the CNN computations by 3x to 5x, and can handle extremely large images (resolution larger than 100,000×100,000).

ACKNOWLEDGMENTS

This research is supported by the National Basic Research Program of China (GrantNo. 2015CB352402) and National Science Foundation of China (GrantNo. 61661146001).

REFERENCES

[1] Chenyi Chen, Ari Seff, Alain L. Kornhauser, and Jianxiong Xiao. 2015. DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving. In *ICCV*. 2722–2730.

[2] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO)*. 609–622.

[3] Trishul M. Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *OSDI*. 571–582.

[4] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *NIPS*. 1232–1240.

[5] Cícero Nogueira dos Santos and Maira Gatti. 2014. Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts. In *COLING*. 69–78.

[6] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. 2015. A Neural Algorithm of Artistic Style. *CoRR* abs/1508.06576 (2015).

[7] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *NIPS*. 2672–2680.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*.

[9] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *ICML*. 448–456.

[10] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. 2016. Perceptual Losses for Real-Time Style Transfer and Super-Resolution. In *ECCV*. 694–711.

[11] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *PEMNLP*. 1746–1751.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*. 1106–1114.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*. 1097–1105.

[14] Fujun Luan, Sylvain Paris, Eli Shechtman, and Kavita Bala. 2017. Deep Photo Style Transfer. *CoRR* abs/1703.07511 (2017). https://arxiv.org/abs/1703.07511

[15] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2009. Pregel: a system for large-scale graph processing. In *PODC*. 6.

[16] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014).

[17] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *CVPR*. 1–9.

[18] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *CVPR*.

[19] Dmitry Ulyanov, Vadim Lebedev, Andrea Vedaldi, and Victor S. Lempitsky. 2016. Texture Networks: Feed-forward Synthesis of Textures and Stylized Images. In *ICML*. 1349–1357.

[20] Wei Wang, Gang Chen, Tien Tuan Anh Dinh, Jinyang Gao, Beng Chin Ooi, Kian-Lee Tan, and Sheng Wang. 2015. SINGA: Putting Deep Learning in the Hands of Multimedia Users. In *Proceedings of the 23rd Annual ACM Conference on Multimedia Conference, MM '15, Brisbane, Australia, October 26 - 30, 2015*. 25–34.

[21] Chao Weng, Dong Yu, Michael L. Seltzer, and Jasha Droppo. 2015. Deep Neural Networks for Single-Channel Multi-Talker Speech Recognition. *IEEE/ACM Trans. Audio, Speech & Language Processing* 23, 10 (2015), 1670–1679.