Toward Large-Scale Image Segmentation on Summit

Sudip K. Seal
Computer Science and Mathematics Division
Oak Ridge National Laboratory, USA

Seung-Hwan Lim
Computer Science and Mathematics Division
Oak Ridge National Laboratory, USA

Dali Wang
Environmental Sciences Division
Oak Ridge National Laboratory, USA

Jacob Hinkle
Computational Sciences and Engineering Division
Oak Ridge National Laboratory, USA

Dalton Lunga
National Security Emerging Technology
Oak Ridge National Laboratory, USA

Aristeidis Tsaris
National Center for Computational Sciences
Oak Ridge National Laboratory, USA

ABSTRACT

Semantic segmentation of images is an important computer vision task that emerges in a variety of application domains such as medical imaging, robotic vision and autonomous vehicles to name a few. While these domain-specific image analysis tasks involve relatively small image sizes ($\sim 10^2 \times 10^2$), there are many applications that need to train machine learning models on image data with extents that are orders of magnitude larger ($\sim 10^4 \times 10^4$). Training deep neural network (DNN) models on large extent images is extremely memory-intensive and often exceeds the memory limitations of a single graphical processing unit, a hardware accelerator of choice for computer vision workloads. Here, an efficient, sample parallel approach to train U-Net models on large extent image data sets is presented. Its advantages and limitations are analyzed and near-linear strong-scaling speedup demonstrated on 256 nodes (1536 GPUs) of the Summit supercomputer. Using a single node of the Summit supercomputer, an early evaluation of a recently released model parallel framework called GPipe is demonstrated to deliver $\sim$ 2X speedup in executing a U-Net model with an order of magnitude larger number of trainable parameters than reported before. Performance bottlenecks for pipelined training of U-Net models are identified and mitigation strategies to improve the speedups are discussed. Together, these results open up the possibility of combining both approaches into a unified scalable pipelined and data parallel algorithm to efficiently train U-Net models with very large receptive fields on data sets of ultra-large extent images.

KEYWORDS

applied machine learning, scalable data analytics, deep neural networks, model parallel, image segmentation, U-Net, pipeline.

ICPP '20, August 17–20, 2020, Edmonton, AB, Canada
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8816-0/20/08…$15.00
https://doi.org/10.1145/3404397.3404468

ACM Reference Format:
Sudip K. Seal, Seung-Hwan Lim, Dali Wang, Jacob Hinkle, Dalton Lunga, and Aristeidis Tsaris. 2020. Toward Large-Scale Image Segmentation on Summit. In *49th International Conference on Parallel Processing - ICPP (ICPP '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3404397.3404468

1 INTRODUCTION

Given an $n \times n$ digital image, a segmentation task classifies each of of its n^2 pixels into one of k pre-defined classes in a finite set S of k classes. For example, in a 64×64 biomedical image consisting of bone and muscle tissue, a simple image segmentation task would be to label each of its $n^2 = 4096$ pixels by a class from a pre-defined set $S = \{bone, muscle\}$. Segmentation tasks arise in a wide variety of domain-specific applications. In biomedical imaging, high quality image segmentation is critical for accurate medical diagnosis. Autonomous vehicles rely on fast and accurate segmentation of the driving environment to ensure driver and pedestrian safety. High-throughput data acquisition systems in multiple scientific and engineering domains such as electron microscopy, collision experiments and satellite imagery collect vast troves of image data which require high-speed high-quality segmentation for high-precision knowledge discovery.

Motivation With a resurgence of machine learning (ML) techniques in recent years, deep neural networks (DNN) have enjoyed immense successes in image classification tasks, including segmentation. Major strides have been made in deep learning technologies, especially by the industry stalwarts. Despite their many spectacular successes, efficient training of ML models on large extent images poses a unique set of challenges and continues to remain in its infancy. One of the main reasons for this gap is that most ML tasks that are of interest to the industry involve images of considerably lower resolution than those that are encountered in many scientific domains. For example, the ImageNet data set, against which most ML models for image analytics are benchmarked, consists of images with resolutions that are at most $\sim 10^3 \times 10^3$ pixels. On the other hand, data sets of satellite images currently collected at resolutions of 30 cm (and only expected to increase over time) are typically as large as $10^5 \times 10^5$. Accordingly, DNN models for segmentation of such images can potentially have 10,000-fold more nodes per layer than their smaller counterparts! Similar, and even larger, higher resolution images are anticipated in the next generation of neutron scattering experiments. Segmentation of such large images require DNN models that no longer fit in the memory of a single device necessitating algorithms that can leverage the memory storage of multiple devices to overcome the memory limitations.

Even when image sizes are modest, deeper models are required to identify features that widely vary in sizes. For example, DNN models that can identify and isolate a cat and a building in the same picture requires a larger receptive field than one that needs to identify only a cat or only a building. In DNN architectures, the value of each neuron in a layer depends on a subset of neurons from the previous layer to which they are connected (in fully connected neurons, this subset comprises of all the neurons from the previous layer). For a neuron in a layer, its receptive field is defined by the region of the input image that corresponds to this subset of neurons in the previous layer to which it is connected [6]. Clearly, the activation of a neuron is unaffected by the neurons outside its receptive field. In order to ensure that the DNN is able to "see" both large and small features in the input image, the receptive fields have to be judiciously controlled.

There are multiple ways to increase the effective receptive field of a DNN model. One way is to make the network deeper by stacking more layers, which has been shown to increase the receptive field linearly with the stack size. There are other options like subsampling, which increases the receptive field multiplicatively. Either way, increasing the receptive field increases the model size and, hence, the memory requirements grow very quickly. Clearly, limiting the size of a DNN model to that available on a single device severely restricts its receptive field. Using a parallel approach that gainfully utilizes the memory units of multiple devices enables training DNNs with larger receptive fields.

Accordingly, the main motivation of this study is to assess the effectiveness of two parallel approaches for training large DNN models when they do not fit on a single device, either due to the sheer size of each image sample and/or due to the size of the DNN model to be trained. In keeping with current practice, a (computing) device will refer to a GPU hardware accelerator in this paper.

Challenges The overall memory requirements of a DNN model, henceforth referred to as the *size of the model*, for a segmentation task depends on the fixed size of the input images and the specific architecture of the network model. Here, we focus on the U-Net model as an example network model but the challenges and solution strategies reported are potentially applicable to a broader class of DNNs with similar architectures.

One approach to training large DNN models that do not fit on a single GPU, is to store the data and the model in the larger CPU memory, bringing only portions of it that can reside on a GPU as and when needed during the computations. This approach is sub-optimal as the narrow device-host transfer bandwidth quickly becomes the dominant performance bottleneck and a overwhelming majority of the total training time is spent solely in data transfer. Scientists are, therefore, forced to train on down-scaled images and/or smaller DNN models that can together fit on a single GPU resulting in low quality ML models with compromised predictive capabilities.

Accelerator hardware designed and built specifically for ML workloads is another approach for large model training. These specialized systems are beginning to emerge now. However, hardware solutions are less general and will always present an upper limit in the size of DNN models that can be trained on them. On the other hand, algorithmic software-level parallelization of ML computations offers a more general approach which can be adapted to broader classes of underlying hardware specifications.

A vast majority of existing HPC-enabled DNN adopts what is called the *data parallel* approach in which the training data set is partitioned across multiple devices in such a way that the combined memory requirements of the data partition and the DNN model is fulfilled by each participating device. In this approach, the size of the DNN model is still limited by the amount of available device memory.

Model parallelism relaxes this limitation by parallelizing the computations of a single DNN model across multiple devices. Computations in a typical DNN training involve multiple forward passes through the DNN in which model predictions are computed interspersed with backward passes through the same DNN in which model weight updates are computed. In both passes, computations in a layer depend on the results of computations in the previous layer. This inherent sequentiality makes partitioning single model computations across multiple devices challenging. One approach is to partition the computations in each layer across multiple devices followed by an aggregation of these results before the computations in the next layer is begun. Such an approach requires frequent and computationally expensive all-to-all collectives for data aggregation. Another approach in which the layers of the DNN model are partitioned across multiple devices is called *pipeline parallelism*. This form of parallelism can be combined with data parallelism to enable very large model training if sufficient number of devices can be deployed to each data parallel pipeline.

Related Work Since winning a major international challenge in 2015 by a wide margin, U-Net has become one of the more popular DNN models for image segmentation tasks. There are other variants based on the encoder-decoder principle of DNN architectures. For example, the V-Net [7] and 3D-UNet [2] architectures are extensions of the U-Net architecture to volumetric image data commonly encountered in biomedical applications. U-Net++ [12] is yet another variant that uses a series of nested, dense skip pathways to reduce the semantic gap between the encoder and decoder feature maps to enable more accurate learning. U-Net based DNN architectures have found applications beyond the biomedical domain. For example, the use of U-Net in segmentation and localization of features in geospatial images has been reported in [11].

High-resolution at-scale image analysis has spurred an immense interest in training massive, more sophisticated models with enhanced predictive capabilities. These models require a huge amount of data to train on and are usually too large to fit on a single device. Parallel approaches, therefore, become a natural alternative to train them. Parallel training of DNNs can be viewed along a variety of design approaches. Data parallel training methods have dominated the large-scale HPC-enabled ML landscape over the years. On the other hand, model parallel approaches are only beginning to emerge. [3] reports an early implementation of a model parallel execution based on channel and filter parallelization. Two recently reported model parallelism frameworks based on pipeline parallelism are GPipe [4] and PipeDream [8]. Both frameworks are in their early stages of development. The Mesh-TensorFlow framework is an early attempt to combine both data- and model-parallel execution into a more general distributed training framework [10]. The work presented here is based on the GPipe framework made available in PyTorch [5] and reports an early evaluation of its performance for large-scale image segmentation tasks on the Summit architecture.

Contributions Training large NN models on datasets of large images requires a scalable training framework that can be used to fine-tune the models using appropriate hyperparameter tuning algorithms for best training accuracies. This paper reports on the scalability and performance bottlenecks of such a parallel framework developed by combining sample parallelism (to enable training on large extent images) and pipeline parallelism (to enable training large U-Net models that do not fit on a single GPU). Together, this novel combination is shown to execute U-Net models with 10X more trainable parameters and 4X larger receptive fields than has been reported in the literature. Specifically:

- A theoretical analysis of the U-Net architecture is presented.
- An efficient, sample parallel U-Net training algorithm for the segmentation of arbitrarily large images is presented.
- The sample-parallel algorithm is demonstrated to exhibit near-linear strong scaling speedup on up to 1,536 GPUs on Summit.
- An early evaluation of the GPipe framework for pipeline-parallel execution of U-Net models with over $4X$ increase in the effective receptive field on a single Summit node is presented.
- Pipeline-parallel execution of large U-Net models is shown to deliver $2X$ speedup on 6 GPUs over its sequential execution.

The rest of the paper is organized as follows. The next section provides a brief description of the U-Net model used for all subsequent analysis and experiments. Section 3 analyzes the structure of a general U-Net architecture. In Section 4, an efficient, parallel algorithm to train datasets of large images is presented and its advantages and limitations discussed. Section 5 provides a brief description of a model parallel framework. Its performance evaluation based on subsequent experiments carried out on a single Summit node are reported in Section 6. Conclusions are drawn and future directions are discussed in Section 7.

2 PRELIMINARIES

U-Net was first designed for segmentation of biomedical images [9] and demonstrated to learn very effectively from a limited number of annotated images as is often the case in the biomedical domain. The classic U-Net architecture, shown in Fig. 1, is symmetric. The network has a down-sampling encoder phase followed by an up-sampling decoder phase. The encoder path extracts the spatial features in the input image and captures their contexts. The decoder path localizes the features. The encoder path starts with 64 convolutional layers with 3×3 filters followed by another series of 64 convolutions with 3×3 filters. The resulting feature map is then down-sampled using a max-pool operation with a 2×2 pooling size and stride of 2. This sequence of two convolutional layers followed by a max-pool layer is repeated four times. In each repetition, the number of convolutions is doubled while keeping the filter size fixed at 3×3. The resulting feature map is up-convolved using a sequence of two 3×3 convolution layers and enters the decoder path which is symmetric to the encoder path. In each decoder step,

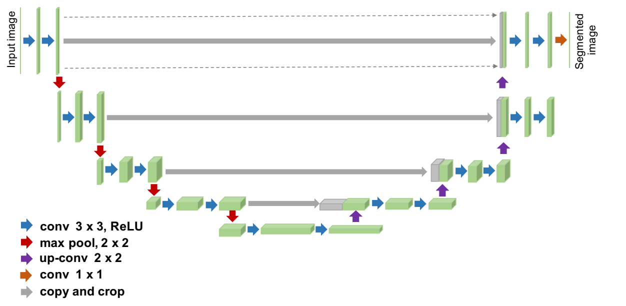


Figure 1: The U-Net architecture. In the original model, each of the first two convolutional layers have a depth of 64 with 3×3 filters. The dashed line indicates that a rectangular annular region of width, say ϵ, remains unsegmented in the output segmented map (see Fig. 2).

the feature map from the previous step is up-sampled using a transposed convolution operation with 2×2 filters. The output of the encoder layer at the same level as that of the up-sampled feature map are then concatenated and passed through two convolution layers with 3×3 filters. This sequence of up-sampling $\longrightarrow$ concatenation $\longrightarrow$ convolution operations is repeated four times halving the number of 3×3 filters at each stage. The final segmented image is obtained using a 1×1 convolution operation. All convolution layers except the last one use the ReLU (Rectified Linear Unit) activation function. The last convolution layer uses the sigmoid activation function. The concatenation operations are a distinguishing aspect of the U-Net architecture. They enable the network to recover spatial information that was blurred out during the down-sampling max-pool operations along the encoder path.

As shown in Fig. 2, a U-Net model segments a $(N - \epsilon) \times (N - \epsilon)$ region within the original $N \times N$ input image. The width, ϵ, of the annular region that remains unsegmented depends on the specifications of the U-Net model and is a function of the number of convolutional layers, number and sizes of filters, pooling size, stride lengths and other network parameters, as shown next.

3 ANALYSIS OF THE U-NET MODEL

In this section, the structure of a U-Net model is analysed and key relations related to the memory requirements for the segmentation

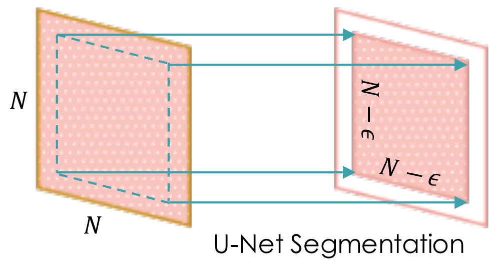


Figure 2: U-Net model takes an input image of size $N \times N$ and outputs a segmented image of size $(N - \epsilon) \times (N - \epsilon)$.

of a fixed-size input image are derived. The size of the output image of a convolutional layer is given by:

$$O = \frac{N - K + 2P}{S} + 1$$

where O is the output image size, N is the input image size, K is the filter size, P is the padding and S is the stride length. All input images are assumed to be squares of pixel dimension $N \times N$. Thus, the difference d between the input and output image sizes after a single convolution is:

$$d = \frac{N(S-1) + K - 2P}{S} - 1 \tag{1}$$

Let the L levels of a U-Net model be denoted by $\ell = 0, 1, 2, \cdots, L-1$ and the number of convolutional layers for a single filter in a level be denoted by n_c. Also, let the number of filters in level ℓ be denoted by n_f^ℓ and n_f denote the number of $K \times K$ filters in the first level. In this analysis, we make the following two assumptions that hold true in practice:

Assumption 1. *The filter size K, the padding P, the stride length S and the number of convolutional layers n_c for a single filter are assumed to remain unchanged at each level.*

Assumption 2. *A 2×2 max-pool layer connects one encoder level to the next and a 2×2 up-convolutional layer connects one decoder level to the next.*

In a U-Net model, the number of filters double with every level. Therefore, $n_f^\ell = 2^\ell n_f$. Additionally, let I_ℓ and X_ℓ denote the input and output image sizes at an encoder level ℓ while J_ℓ and Y_ℓ denote the input and output image sizes at a decoder level ℓ. Based on the above assumptions, the input and output image sizes at any given level ℓ of a U-Net satisfy:

$$X_\ell = I_\ell - dn_c, \qquad Y_\ell = J_\ell - dn_c \tag{2}$$

Given a $N \times N$ input image to a U-Net, the input image size at an encoder level ℓ is:

$$I_0 = N, \; I_1 = \frac{I_0 - dn_c}{2} = \frac{N - dn_c}{2}, \; I_3 = \frac{I_2 - dn_c}{2} = \frac{N - 7dn_c}{2^3}, \cdots$$

yielding

$$I_\ell = \frac{N - (2^\ell - 1)dn_c}{2^\ell} \tag{3}$$

where $\ell = 0, 1, 2, \cdots, L-1$.

At the deepest level $L-1$, the size of the output image is $X_{L-1} = I_{L-1} - dn_c$. The size of the input image to the decoder at level $L-2$ is twice that of the output image size at the deepest level. Therefore, $J_{L-2} = 2X_{L-1} = 2I_{L-1} - 2dn_c$ and the size of the output image at the decoder level $L-2$ is:

$$Y_{L-2} = J_{L-2} - dn_c = 2I_{L-1} - 3dn_c$$

Similarly, the input and output image sizes at level $L-3$ of the decoder are:

$$\begin{aligned} J_{L-3} &= 2Y_{L-2} = 4I_{L-1} - 6dn_c \\ Y_{L-3} &= J_{L-3} - dn_c = 4I_{L-1} - 7dn_c \end{aligned}$$

Continuing in this fashion, it is straightforward to show that:

$$J_{L-\ell'} = 2^{\ell'-1} I_{L-1} - 2(2^{\ell'-1} - 1)dn_c \tag{4}$$

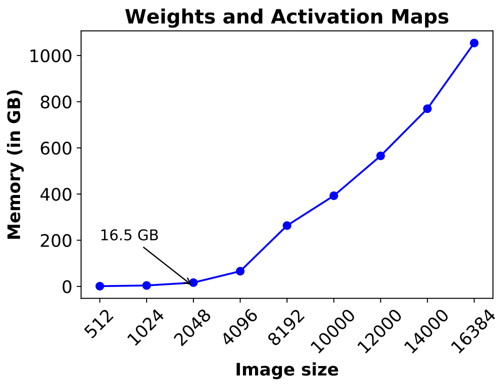


Figure 3: Memory requirements of a U-Net model with increasing image size.

where $\ell' = L - \ell = 2, 3, \cdots, L$. Therefore, the size of the output image of a U-Net, which is at level $\ell = 0$ ($\ell' = L$), is:

$$Y_0 = J_0 - dn_c = N - (2^{L-1} - 1)dn_c - (2^L - 1)dn_c$$

where Eqn (3) has been used with $\ell = L - 1$ and Eqn (4) has been used with $\ell' = L$. Using the preceding equation, it is straightforward to show that the difference $2\epsilon = I_0 - Y_0$ between the sizes of the input and output images of the U-Net model is:

$$\epsilon = (3 \cdot 2^{L-2} - 1)dn_c \tag{5}$$

For the U-Net in [9], $L = 5$ and $d = n_c = 2$, which yields $\epsilon = 92$.

Note that along the decoder path, the size of the input image at any level has to be at least as large as $K + dn_c$, where K is the filter size. Therefore, we have:

$$Y_{\ell-2} > K + dn_c, \quad Y_{\ell-3} > K + dn_c, \quad \cdots \quad Y_0 > K + dn_c$$

which yields:

$$\sum_{\ell=0}^{L-2} Y_\ell = \sum_{\ell=0}^{L-2} (J_\ell - dn_c) > (L-1)(K + dn_c)$$

Using Eqn (3) and Eqn (4) in the above relation yields the following condition:

$$\begin{aligned} N &\geq (2^L + 2^{L-1} - 1)dn_c + \left(\frac{L-1}{2^{L-1} - 1}\right) 2^{L-2} K \\ \Rightarrow N &> (2^L + 2^{L-1} - 1)dn_c + 2^{L-2} K \end{aligned} \tag{6}$$

where we have used the fact that the factor within the parenthesis in the second term is maximum when $L = 2$, the minimum possible number of levels in a U-Net. Thus, a U-Net with a specified architecture can only segment images whose sizes are greater than a threshold size that depends on the parameters of the U-Net architecture via Eqn (6).

4 SAMPLE PARALLEL U-NET TRAINING

This section presents a parallel strategy to train a U-Net model for large image segmentation. As mentioned earlier, the memory demands of training a U-Net on very large images is significant. Figure 3 shows the memory required (in GB) to train a standard U-Net model on data sets of varying input image sizes. As expected,

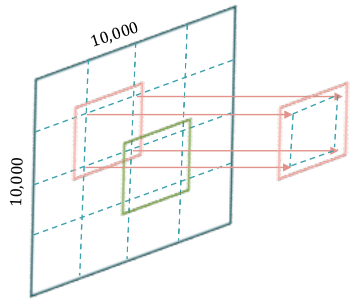


Figure 4: Example of a 10000×10000 image sub-divided into a 4×4 array of non-overlapping tiles (shown by the dashed lines). Segmenting each padded tile independently is equivalent to segmenting the original 10000×10000 image.

the needed memory scales as $\sim N^2$. Specifically, the memory needed to train a U-Net model on data sets of 10000×10000 images is about 393 GB which is significantly more than the storage offered by any state-of-the-art GPU. Down-sampling the images with large strides and/or reducing the model depth both reduces the total memory footprint, but at the cost of adversely affecting the predictive quality of the resulting models. For localized features, smaller sub-regions of interest from the original (large) images can be selected to train the U-Net model but such an approach assumes a priori knowledge of the training samples which are likely to be both subjective as well as error prone.

Here, we present a U-Net training algorithm for large images that leverages the properties of the U-Net architecture to train multiple smaller models such that the collective segmentations of the smaller models is identical to that of the original large image by a single U-Net model if it could be held in memory.

The key observation is that a U-Net model segments a $(N - \epsilon) \times (N - \epsilon)$ area of an $N \times N$ input image (see Fig. 2). This ϵ-width defines the receptive field of the U-Net model and, as such, the segmentation of any part of the image is unaffected by pixels that are more than ϵ pixels away. This property is analogous to finite-difference methods in which computations at a grid point is unaffected by values that are away by more than a certain number of grid points depending on the order of the finite-difference method.

The algorithm proceeds by sub-dividing each original $N \times N$ image in the data set into an array of non-overlapping $T' \times T'$ tiles. See Fig. 4. Each tile is then padded by its surrounding annular region of width ϵ in the original image. Let the resulting width of each tile be denoted by T. Segmenting each $T \times T$ tile using independent smaller U-Net models, each with ϵ receptive field, is then equivalent to segmenting the original $N \times N$ image using a larger U-Net model with ϵ receptive field. Depending on the architecture of the underlying computing hardware, the tile size is chosen such that the storage needed to hold the tile, weights and activation maps of the U-Net model is less than the device memory.

The parallel strategy described above is highly scalable. No communication is required between the individual model computations on each padded tile. As such, this algorithm is almost embarrassingly parallel except for the collective calls during weight updates at the end of each iteration. In principle, this algorithm can be appropriately modified to train segmentation models on data sets of arbitrarily large images as long as the memory demands for segmenting each padded tile is satisfied by a single device memory. This flexibility makes it attractive for easy data parallel U-Net training on ultra-high resolution image data sets using massively-parallel hardware-accelerated supercomputers like Summit.

However, good parallel runtime performance of this strategy comes at a cost of extra computations within the ϵ-annular regions. To better understand this trade-off, let us denote the total number of GPUs by $P = q^2$ and, for simplicity, assume $N = T' \cdot q$ where T' is an integer. Let an $N \times N$ image be sub-divided into a $q \times q$ array of non-overlapping sub-images, each with dimension $T' \times T'$. Each tile is padded with ϵ pixels on each side resulting in padded tiles of dimension $T \times T$ where $T = T' + 2\epsilon$. Assuming there is enough memory in a GPU, a U-Net model with ϵ-sized receptive field segments a $T' \times T'$ inner region of any $T \times T$ input tile as shown in Fig. 2 and Fig. 4.

Here, our focus is on segmenting large images on the Summit supercomputer. A Summit node houses six Volta V100 GPUs. Each Volta V100 GPU on Summit has 16 GB of memory. From Fig 3, it is clear that when $T = (T' + 2\epsilon) \gtrsim 2000$, the memory of each GPU on Summit will saturate. Using the example in Fig. 4, one strategy to train a U-Net model with receptive field $\epsilon = 92$ (like the original model published in [9]) on a data set of 10000×10000 images would be to sub-divide each image into a $q \times q$ array of $T' \times T'$ non-overlapping tiles where:

$$\begin{aligned} & q = \lceil \tfrac{N}{T'} \rceil = \lceil \tfrac{10000}{2000-184} \rceil = 6 && \text{(1 Summit node)} \\ \Rightarrow \quad & P = q^2 = 36 && \text{(6 Summit nodes)} \end{aligned}$$

Then, a U-Net segmentation model can be trained on a data set with, say, 10000 images of size 10000×10000 using the sample parallel strategy as follows:

- Partition each 10000×10000 image into roughly 36 non-overlapping tiles, each of dimension roughly $\frac{10000}{6} \times \frac{10000}{6}$.
- Pad each tile by the neighboring $w = \frac{184}{2} = 92$ pixels to form tiles of dimension $(\frac{10000}{6} + 184) \times (\frac{10000}{6} + 184)$.
- Use a U-Net model with receptive field $\epsilon = 92$ to train each $(\frac{10000}{6} + 184) \times (\frac{10000}{6} + 184)$ tile on a single Summit GPU and independently obtain a non-overlapping segmented tiles of dimension $\frac{10000}{6} \times \frac{10000}{6}$.
- Assuming a training data set with 10000 samples and a batch size of 100, a U-Net model with receptive field $\epsilon = 92$ can be efficiently trained using 3600 Summit GPUs (or 600 Summit nodes).

Recall that *not* all computations performed on each GPU are useful. The parallel strategy proposed here uses a tile of dimension $T \times T$ as an input to the U-Net model only to output a segmented tile of dimension $T' \times T'$ where $T' < T$. Let E denote the ratio of the total volume of computations to the total volume of non-repeated computations. Then, E will scale as follows:

$$E \sim O\left(\frac{T^2}{T'^2}\right) = O\left(\frac{(N/q + 2\epsilon)^2}{(N/q)^2}\right) \sim O\left(1 + q\frac{4\epsilon}{N}\right) \tag{7}$$

ignoring second order terms. When the total amount of computations is equal to the amount of non-repeated computations, $E = 1$.

Ideally, E should be as close to 1 as possible. In other words, since N and ϵ are fixed by the input image size and the model architecture, respectively, the tiling size should be chosen such that:

$$q\frac{4\epsilon}{N} \ll 1 \Rightarrow q \ll \frac{N}{4\epsilon} \tag{8}$$

The above relation highlights a fundamental limitation of the parallel approach described so far. Scientists are interested in training U-Net models with large receptive fields so that large features can be identified along with smaller features in the image. Receptive fields of U-Nets (and other DNNs) are increased by growing the depth L of the network. From Eqn (5), it is clear that $\epsilon \sim O(2^L)$. But, increasing the receptive field ϵ would require decreasing the value of q to keep E closer to 1. In other words, for a fixed image size, larger receptive fields implies division of the original images into a smaller array of larger tiles which along with the model weights and activation maps may not fit in the device memory. On the other hand, if the original images are subdivided into larger arrays of very small tiles, then $E \gg 1$ rendering the algorithm inefficient in terms of resource utilization.

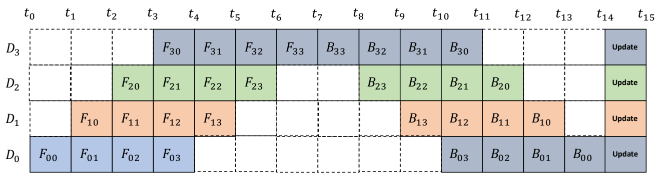
5 PIPELINE PARALLEL U-NET TRAINING

U-Net, like any other DNN, can be viewed as a sequence of layers, each performing a set of array-based operations. Computations in these layers, during forward and backward propagations, depend on computations performed in the previous layer. This establishes a strict sequentiality in the forward and backward computations that needs to be preserved to ensure correctness. In a pipeline-parallel execution, the sequence of network layers are partitioned across multiple devices and the computations are distributed layer-wise in such a way that the sequentiality of overall computations is preserved. This kind of a partitioning strategy enables training of much larger DNN models since their sizes are no longer limited by the memory storage of a single device but can harness the combined memory units of multiple devices.

GPipe implements pipeline parallelism by partitioning all the layers in the DNN into K partitions called *cells*. Computations within each cell are performed on a single device as long as the memory requirement of the cell and its associated weights and activations maps fit within the device memory.

When a DNN model is trained, the training samples are divided into subsets called *mini-batches*. A loss function is computed after all the samples in the mini-batch are fed forward through the DNN. Based on this loss function, model weights are updated after a backward propagation of the entire network during which it computes the weight updates using a very high-dimensional gradient descent procedure. Thus, a pipeline algorithm for model training will have to ensure that each mini-batch of samples is processed in a manner that preserves the computational sequentiality described above to ensure correctness with respect to a non-pipelined execution of the same mini-batch on a single device (if it were possible).

For computational efficiency, the GPipe framework splits a mini-batch with N samples into M smaller *micro-batches* and pipelines the execution of each set of micro-batches over the K cells. Each



t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12} t_{13} t_{14} t_{15}
D_3
F_{30} F_{31} F_{32} F_{33} B_{33} B_{32} B_{31} B_{30} Update
D_2
F_{20} F_{21} F_{22} F_{23} B_{23} B_{22} B_{21} B_{20} Update
D_1
F_{10} F_{11} F_{12} F_{13} B_{13} B_{12} B_{11} B_{10} Update
D_0
F_{00} F_{01} F_{02} F_{03} B_{03} B_{02} B_{01} B_{00} Update

Figure 5: Pipelined execution of 4 mini-batches (shown in 4 colors) on four devices D_0, D_1, D_2 and D_3. Each mini-batch is partitioned into 4 equal-sized micro-batches (each shown as a colored square). The total time, assuming each pipeline stage takes equal time, is shown as $t_0, t_1, \cdots, t_{15}$.

micro-batch is composed of $\frac{N}{M}$ samples. This strategy of partitioning a mini-batch into micro-batches, as shown in Fig. 5, reduces the idle time per device. In addition, GPipe implements rematerialization, a memory reducing algorithm that enables DNN training with sub-linear memory at the cost of modest extra computations, to reduce activation memory requirements [1]. A more detailed discussion of the GPipe framework can be found in [4].

6 PERFORMANCE

A primary goal of this study is to assess the computational performance of large-scale image segmentation workloads on the Summit supercomputer. Performance studies of ML workloads differ fundamentally from their conventional scientific computing counterparts. In the latter, time-to-solution with constant or varying workloads is a major criterion for evaluation. These criteria usually translate to strong or weak scaling speedups. A tacit assumption in the definitions of such performance metrics is that the final solution remains largely unchanged even as the time-to-solution decreases with increasing parallelism. This is because the global ordering of the underlying mathematical operations in scientific computing workloads remains unchanged with increasing parallelism. ML workloads are, however, stochastic by definition and the global ordering of the numerical steps during training changes as the granularity of parallelism changes. Consequently, reductions in training times due to increased parallelism does not necessarily translate to commensurate reduction in time-to-solution which is an iterative process and measured by the accuracy of the final prediction(s) of the ML model. Techniques such as hyperparameter tuning are used to tune the model parameters for improved model accuracy but, to do so effectively, fast and scalable training is a must. Here, we only report the computational performance of the parallel strategies described here in terms of their training speed and other related computational metrics, with the understanding that a deeper study on hyperparameter tuning of DNN models for accuracy improvements using these fast parallel strategies, while extremely essential, remains outside the scope of this effort.

Computational Platform: Summit has 4,600 compute nodes, each equipped with two IBM POWER9 processors and six NVIDIA Volta V100 accelerators. Each node houses 512 GB of DDR4 memory for use by the POWER9 processors and 96 GB of High Bandwidth Memory (HBM2) divided equally amongst all 6 GPU accelerators (16GB/GPU). Each node also has 1.6TB of non-volatile memory for use as burst buffers. Each processor has 22 cores with support

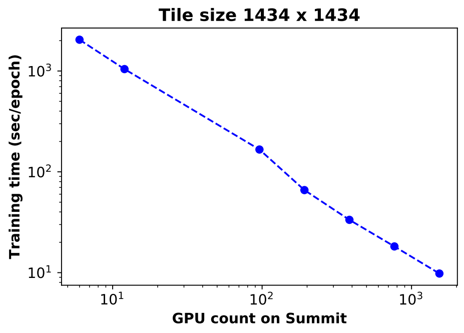


Figure 6: Training time per epoch using the sample parallel algorithm exhibits near linear speedup as the number of GPUs grows from 1 to 256 Summit nodes (6 to 1536 GPUs).

for four hardware threads per core. The POWER9 processors are connected via dual NVLINK capable of 256 GB/s transfer rate in each direction. The V100 GPUs use NVIDIA's NVLink interconnect for CPU-to-GPU and GPU-to-GPU data transfer.

Models: For this study, we use four U-Net models, referred to as *small*, *medium-1*, *medium-2* and *large* models. The models *medium-1* and *medium-2* have roughly the same number of trainable parameters but they differ in the number of levels and the number of convolutional layers. In each model, a 2×2 max pool layer with a stride of 2 connects any two levels along the encoder arm of the U-Net and a 2×2 up-convolution connects any two levels along the decoder arm. The filter size in all the convolutional layers is 3×3. Other details of the models are provided in Table 1.

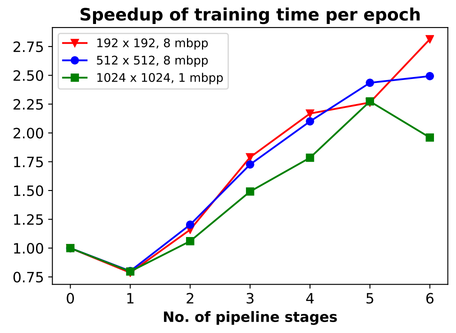
6.1 Sample Parallel U-Net Training

Here, we present the computational performance of the sample parallel algorithm for training large images sub-divided into non-overlapping tiles (see Fig. 4). For this experiment, 5120 tiles, each with dimension 1434×1434, were used for training, of which 640 tiles were used for validation and an additional 640 for testing.

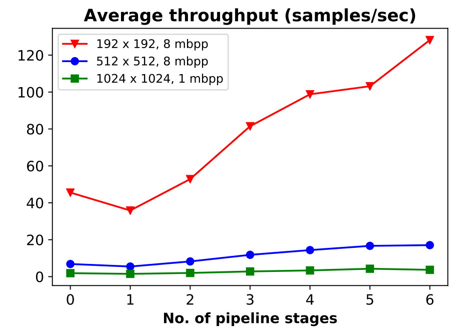
Figure 6 shows the strong scaling performance of training the *medium-1* model on the data set described above. The y-axis plots the wall-time required per epoch where an *epoch* refers to one complete cycle through the entire training set. As described in Section 4, the scaling is nearly linear with the number of hardware accelerators. The number of GPU accelerators was varied from 6 (1 Summit node) to 1536 (256 Summit nodes).

Model	Levels	Conv. Layers/Level	Parameters
Small	5	2	72,301,856
Medium-1	5	5	232,687,904
Medium-2	6	2	289,357,088
Large	7	2	1,157,578,016

Table 1: Architecture of the U-Net models used for performance study.



(a)



(b)

Figure 7: Speedup in U-Net execution time using the *medium-1* model and its throughput as the number of pipeline stages vary. The pipeline stage denoted by 0 refers to the baseline non-pipelined execution and that denoted by 1 refers to a pipelined execution with only one stage.

The advantage of this approach is clear. Sample parallelism scales to data sets of any image size as long as each is sub-divided into smaller tiles and the total memory requirement for the model and the (padded) tiles fits in the memory of a single GPU. For the U-Net model described above, the tile size (including the ϵ-padding of 92 pixels) appropriate for a Summit GPU was $\sim$ 1434. Increasing the receptive field increases the ϵ-padding ultimately exceeding the GPU memory limit. Consequently, on a single Summit node, as long as the padded tile size is maintained at approximately 1434×1434, the original image can be of any size. Larger tiles often hold larger features that can only be detected with larger receptive fields. While it is possible to increase the model size to support larger receptive fields, the tile size will need to be reduced appropriately to accommodate the combined memory on a GPU device. However, as shown in Eqn (7), doing so will quickly increase the amount of extra computations in the ϵ-annular regions greatly diminishing resource utilization.

6.2 Pipeline Parallel U-Net Training

The computational performance of pipeline-parallel execution of the U-Net model is presented in this section using the models listed in Table 1 with varying input tile sizes. In each case, a total of

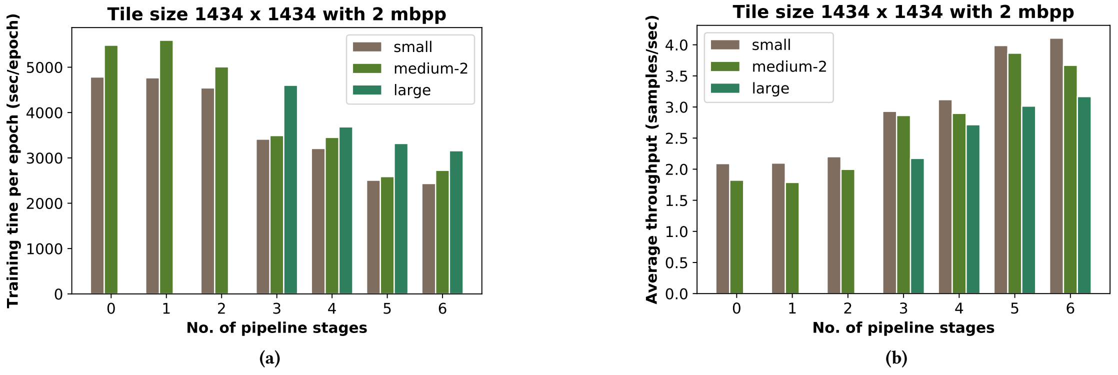


Figure 8: Dependence of training time and throughput on the number of pipeline stages. The pipeline stage denoted by 0 refers to the baseline non-pipelined execution and that denoted by 1 refers to a pipelined execution with only one stage.

10,000 tiles were trained in each epoch. As such, the comparison between the various modes of parallel execution are made in terms of training time per epoch.

Figure 7(a) shows the speedup in training time per epoch for a fixed U-Net model (*medium-1* model with ∼ 233 million training parameters) with increasing number of pipeline stages for four different sizes of input tiles. Figure 7(b) shows the corresponding throughputs measured in terms of input tiles trained per second. In both figures, 0 pipeline stages refers to a baseline non-pipelined sequential execution on one GPU while 1 pipeline stage refers to a pipelined execution but with only one pipeline stage. Figure 7 shows that the latter performs worse in all cases. This under-performance is due to the additional bookkeeping overhead of parallel pipeline execution that are absent in a sequential execution on a single GPU.

Figure 7(a) shows a $2.8X$, $2.5X$ and $2X$ speedup in the training time per epoch using 6 pipeline stages for tiles of sizes 192, 512 and 1024, respectively. Pipeline execution scales better with decreasing tile size as the number of pipeline stages increase. Increasing the number of pipeline stages increases the total communication rounds between the pipeline stages while increasing the tile size increases the volume of communication in each round. Together, this increases the communication-to-computation ratio and degrades the performance of pipeline parallelism for larger tile sizes. The *medium-1* model was also trained on 1434×1434 tiles but executions failed with out of memory errors on 1, 2, 3 and 4 pipeline stages but succeeded with 5 and 6 pipeline stages requiring 5696.318 secs and 5428.294 secs, respectively, to complete.

Using 1434 × 1434 tiles, Fig. 8 shows the performance of training three different U-Net models with varying number of GPUs in terms of their training time per epoch and the corresponding average throughput. The models have ∼ 72 million, ∼ 289 million and ∼ 1.1 billion training parameters, respectively. As the number of pipeline stages is increased from 1 to 6, the speedup for the *small* models varies as 1.00, 1.05, 1.40, 1.49, 1.90 and 1.97, respectively. The corresponding speedup factors for the medium-2 model are 0.98, 1.10, 1.57, 1.59, 2.12 and 2.01, respectively. The large model failed to execute with 1, 2 and 3 pipeline stages. The speedup for a fixed number of pipeline stages increases with the model size. This is because the *medium-2* model is four times the size of the *small* model and requires proportionately larger volume of computations for the same number of pipeline stages. Since the speedup is measured with the sequential execution as the baseline, large models favor larger speedup for a fixed number of pipeline stages.

Pipelined performance of a U-Net model depends on a number of parameters and not just the tile size the models are trained on or the number of trainable parameters in the model. The total memory and GPU utilization of a pipelined execution of a U-Net model depends on two main factors - how the training data is mapped across multiple GPUs and how the neural network model is distributed across them.

As described in Section 5, a mini-batch of training data is subdivided into smaller micro-batches in the GPipe framework to reduce the overall idle time of the GPUs. To distribute a single DNN across multiple devices, the torchGPipe framework defines a group of consecutive DNN layers that run on a single device together to be a *partition* and the number of layers in each partition to be the *balance*. So, the total storage per GPU is determined by the size of each micro-batch and the size of the partition it is assigned.

In the experiments reported here, for a given balance, the number of micro-batches assigned to each partitions in the balance, referred to as *micro-batches per partition (mbpp)*, is kept equal. In other words, even though the micro-batches assigned to two different GPUs are equal in size, they may be operated upon by unequal number of network layers depending on the balance.

NVIDIA defines GPU utilization as the percent of time over the past sample period during which one or more kernels was executing on the GPU and memory utilization as the percent of time over the past sample period during which global (device) memory was being read or written. Figure 9 shows the average memory usages and average GPU utilizations to train three different sized models using a 6 stage pipelined execution. These three models have the same number of convolutional layers per level (n_c) but different depths (L). The total number of layers in the *small*, *medium-2* and *large* models are 109, 129 and 149, respectively, and were executed with the following balances: *small* {14, 24, 30, 22, 12, 7}, *medium-2* {16, 26, 38, 26, 12, 11} and *large* {18, 30, 44, 30, 14, 13}.

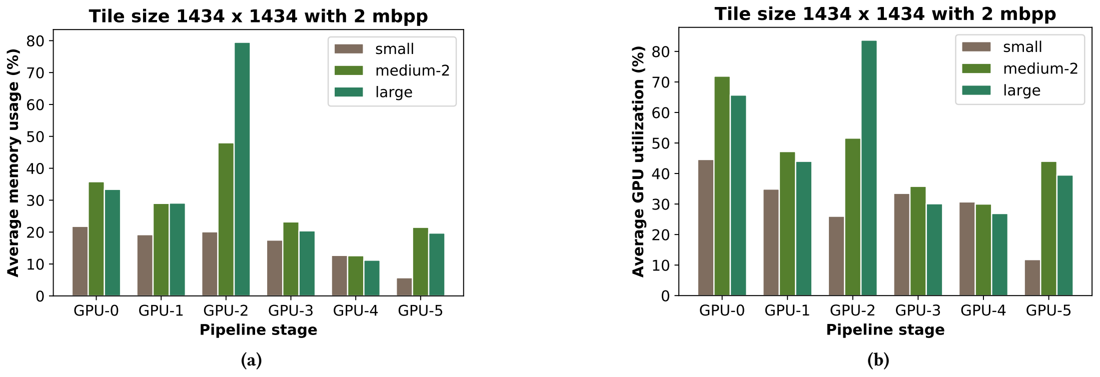


Figure 9: The average memory usage and GPU utilization for training three different U-Net models using 6 pipeline stages. Each model has the same number of convolutional layers per level but different depths.

Figure 10 shows the average memory usages and average GPU utilizations to train two different models with different number of convolutional layers per level but the same depths using a 6 stage pipelined execution with the following balances: *small* {14, 24, 30, 22, 12, 7} and *medium-1* {25, 60, 75, 50, 20, 11}.

It is clear from Fig. 9 and Fig. 10 that resource utilization by a GPU is proportional to the number of layers mapped to it. However, choosing the right balance to increase resource utilization is non-trivial. To understand this better, let E_ℓ denote the memory requirement at an encoder level ℓ and $D_{\ell'}$ denote the same at a decoder level $\ell' = L - \ell$ where $\ell = 0, 1, 2, \cdots, L-1$. In an encoder level ℓ, an $I_\ell \times I_\ell$ input image is operated upon by n_c convolutional operations, each with n_f^ℓ filters. Therefore, using Eqn (3), we have:

$$E_\ell = O\left(I_\ell^2 + 2^\ell n_f \sum_{i=1}^{n_c} (I_\ell - i \cdot d)^2\right) \tag{9}$$

where $n_f^\ell = 2^\ell n_f$ and Eqn (1) has been used. Similarly, at any decoder level ℓ', the memory required can be estimated exactly as above except that the extra memory required due to the copy-and-crop operation of the skip connection at that level has to be accounted for. Together, Eqn (4) yields:

$$D_{\ell'} = O\left(2^{\ell'} n_f \left(2I_{\ell'}^2 + \sum_{i=1}^{n_c} (I_{\ell'} - i \cdot d)^2\right)\right) \tag{10}$$

where $\ell' = L - \ell = 2, 3, \cdots, L$. Eqn (9) and Eqn (10) together describe the memory profile of a U-Net architecture up to an overall constant. The resulting profiles for the four test models are shown in Fig. 11. The profiles not only demonstrate the expected storage trend along the encoder and decoder phases of a U-Net but also exposes the subtle differences that contribute to the non-uniform resource utilization profiles in Fig. 9 and Fig. 10.

The four memory profiles in Fig. 11 can be understood better based on the architecture details. The parameters $K = 3$, $S = 1$ and $P = 0$ remain the same in the four models. Therefore, for all four models, the value of $d = 2$. Using this value of d and Eqn (5), the ϵ-padding for the four models are 92 (*small*), 230 (*medium-1*), 188 (*medium-2*) and 380 (*large*). Thus, the *large model* tested here already supports a receptive field that is over four times that of the *small* model, which is the standard U-Net model.

In the absence of skip connections, the memory requirement at each decoder level is always less than that of the corresponding encoder level. However, additional storage is required for the copy-and-crop skip connection at every decoder level. A smaller ϵ-padding implies that the additional memory required at each decoder level for the skip connections increases. The *small* model has the thinnest unsegmented annular region. Therefore, the memory footprint of the output level after accounting for the extra copy-and-crop from the skip connection exceeds that of the input level. On the other hand, the *medium-1* and *large* models both have very large ϵ-padding as a result of which the extra memory requirement from the skip connections is considerably less leading to lower memory requirements along the decoder path compared to the encoder path. The ϵ-padding of the *medium-2* model is such that the needed memory remains roughly equal at each encoder and decoder level.

Based on Fig. 11, a straightforward balance in which each partition is assigned equal number of network layers will lead to severe load imbalance and non-uniform resource utilization (memory usage and/or GPU utilization). In fact, depending on the memory needed to store the micro-batches assigned to each partition, the largest partition in the balance should satisfy the combined memory requirements of the network layers mapped to it and the size of the micro-batch assigned to the partition. It is due to a violation of this condition that the *medium-1* model failed to execute on 3 and 4 pipeline stages whereas the *medium-2* model, which is only slightly larger than *medium-1*, succeeded (see Fig. 8 and Fig. 11).

While the memory profiles modelled by Eqn (5), Eqn (9) and Eqn (10) provide a good heuristic to generate TorchGPipe partitions for better and more uniform resource utilization, the partition boundaries must be chosen with care such that inter-GPU communication overhead is minimized. The overhead of communication between pipeline stages is significantly lower when the partition boundaries are placed right after a max-pool layer. This reduces the communication volume by half compared to a partition whose boundary is between two convolutional layers.

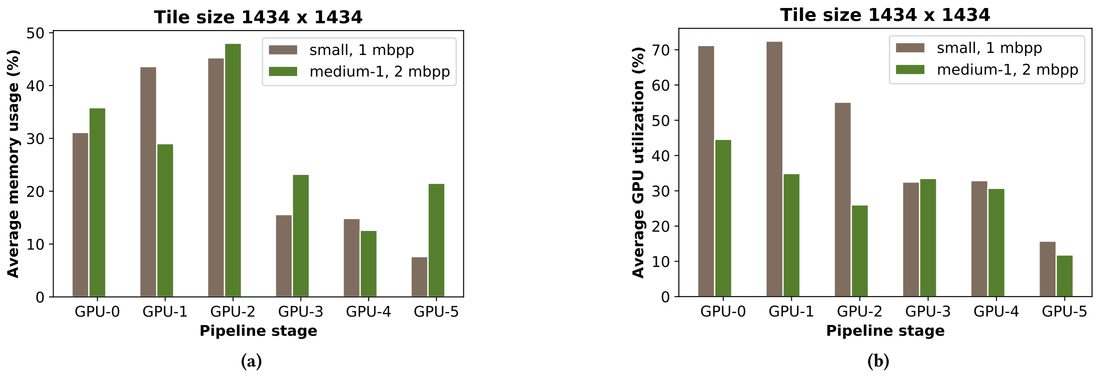


Figure 10: The average memory usage and GPU utilization for training three different U-Net models using 6 pipeline stages. The two models have the same depth but different number of convolutional layers per level.

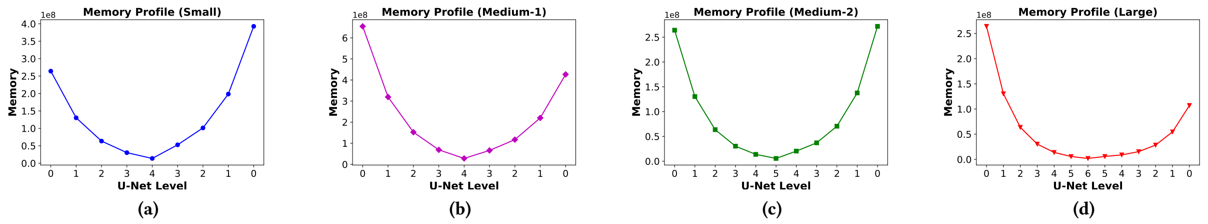


Figure 11: Memory profiles (upto an overall constant) of the four U-Net models (see Table 1) using Eqn (9) and Eqn (10). The x-axes show the level numbers as seen by an image passing through the network.

Thus, a load balancing algorithm for more efficient pipelined training of U-Net models should (a) not violate the global sequentiality of the DNN computations, (b) find a balance that partitions the network layers appropriately taking into account its memory profiles given by Eqn (9) and Eqn (10), and (c) attempt to place the partition boundaries right after max-pool layers of the model.

It is important to recall that the input images to the pipelined models experimented in this section are individual $T \times T$ ϵ-padded tiles and belong to larger original images. The sample parallel approach enables U-Net training of arbitrarily large original images as long as they are sub-divided into tiles with the appropriate ϵ-padding.

7 CONCLUSIONS AND FUTURE WORK

The work presented here is motivated by the need to train large DNN models for semantic segmentations of ultra-high resolution images that arise in a number of scientific domains. Training these large models is extremely memory intensive due to (a) large extents of each individual training sample, and (b) deeper, and hence larger, DNN models capable of supporting larger receptive fields needed for segmentations of both large- and small-scale image features.

To address the issue of training large images, a sample parallel U-Net training algorithm that exhibits near linear speedup on 1534 GPUs was presented. To overcome the memory limitations of a single GPU to train large U-Net mpdels, a pipeline-parallel training framework called TorchGPipe that implements the GPipe framework in PyTorch was evaluated for its baseline performance on a single Summit node (6 GPUs). Pipeline parallelism was shown to further boost the training speed by two to three fold while supporting deeper models with $\sim 4X$ larger receptive field and an order of magnitude more training parameters than the standard U-Net models reported in the literature. The advantages, limitations and performance bottlenecks of the two approaches were discussed and supported by formal analyses where necessary. To the best of our knowledge, the analysis of a U-Net model and a performance study of the sample parallel algorithm along with that of a pipeline-parallel execution of U-Net models on the Summit supercomputer have not been reported before.

The performance results presented in this paper indicate that a good load balancing algorithm will significantly improve resource utilization during pipelined training, opening up the possibility of further speeding up its single node performance. The sample parallel algorithm in combination with single-node pipelined execution is, therefore, expected to enable training of deeper U-Net models on data sets of ultra-high resolution images. In conjunction with data parallel training, the combined advantages of the three parallel approaches - data, sample and pipeline parallelism - will be merged in a future work to dramatically improve the speed and quality of large-scale semantic segmentation tasks on leadership-class heterogeneous architectures such as the Summit supercomputer.

ACKNOWLEDGMENTS

This research is sponsored by the AI Initiative as part of the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the US Department of Energy under contract DE-AC05-00OR22725. This research used resources at the Oak Ridge Leadership Computing Facility, a DOE Office of Science User Facility operated by the Oak Ridge National Laboratory.

REFERENCES

[1] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. (2016). http://arxiv.org/abs/1604.06174

[2] O. Cicek, A. Abdulkadir, S. S. Lienkamp, T. Brox, and O. Ronneberger. 2016. 3D U-Net: learning dense volumetric segmentation from sparse annotation. In *International conference on medical image computing and computer-assisted intervention*. 424–432.

[3] Nikoli Dryden, Naoya Maruyama, Tim Moon, Tom Benson, Marc Snir, and Brian Van Essen. 2019. Channel and Filter Parallelism for Large-Scale CNN Training. In *Proceedings of Supercomputing (SC19)*.

[4] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *33rd Conference on Neural Information Processing Systems (NeurIPS 2019)*.

[5] Chiheon Kim, Heungsub Lee, Myungryong Jeong, Woonhyuk Baek, Boogeon Yoon, Ildoo Kim, Sungbin Lim, and Sungwoong Kim. 2020. torchgpipe: On-the-fly Pipeline Parallelism for Training Giant Models. *ArXiv* abs/2004.09910 (2020).

[6] Wenjie Luo, Yujia Li, Raquel Urtasun, and Richard Zemel. 2016. Understanding the Effective Receptive Field in Deep Convolutional Neural Networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*. 4905–4913.

[7] F Milletari, N Navab, and S Ahmadi. 2016. V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation. In *Proceedings of the Fourth International Conference on 3D Vision*. 565—-571.

[8] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Gregory Ganger, Phillip Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. 1–15.

[9] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *Medical Image Computing and Computer-Assisted Intervention*. 234–241.

[10] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. In *32rd Conference on Neural Information Processing Systems (NeurIPS 2018)*.

[11] Z Zhang, Q Liu, and Wang. Y. 2018. Road Extraction by Deep Residual U-Net. In *IEEE Geoscience and Remote Sensing Letters*, Vol. 15. 749 – 753. Issue 5.

[12] Zongwei Zhou, Md Mahfuzur Rahman Siddiquee, Nima Tajbakhsh, and Jianming Liang. 2018. UNet++: A Nested U-Net Architecture for Medical Image Segmentation. In *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support*. 3–11.