A Distributed Scheme for Accelerating and Scaling PSPNet

Ajay Krishna[1], Xiang Gu[1], Andrew Larkin[2], Perry Hystad[2], and Lizhong Chen[1]

[1] School of Electrical Engineering and Computer Science
[2] College of Public Health and Human Sciences
Oregon State University, Corvallis, OR 97331, USA
{krishnaj, guxi, larkinan, perry.hystad, chenliz}@oregonstate.edu

Abstract. Advances in deep learning based image processing techniques have led to their adoption for a wide range of applications, and in tow with these developments is a dramatic increase in the availability of high quality datasets. With this comes the need to accelerate and scale deep learning applications in order to keep up with the size of data and best use available hardware resources. In this paper, we propose a useful scheme to enable accelerated and distributed deployment of PSPNet inferences, which allows scaling across disparate compute clusters with recourse for system instability and heterogeneity of resources. The systemic incorporation of multiprocessing paradigms and pipelining also allows the scheme to mitigate the time penalties of multiscale inference and significantly improves total inference times when used on extremely large datasets. The use of this distributed scheme to process close to 150,000 Google Street View images as part of a public health study has demonstrated excellent promise in addressing issues of deployment, maintenance, crash recovery, and dataset management at scale.

Keywords: semantic segmentation · PSPNet · acceleration · distributed computing · machine learning

Type of Submission: Regular Research Paper

Corresponding Author: Lizhong Chen <chenliz@oregonstate.edu>

1 Introduction

Over the past few years deep learning has become the de facto technique for advanced computer vision tasks, and the ImageNet [3] database and challenges have helped quantify the progress of deep learning models on various computer vision tasks like object detection and image segmentation. An initial breakthrough in performance gains was obtained by the use of GPUs for training a model called AlexNet [6] which won the ILSVRC 2012 challenges and outperformed the runner up by an impressive 10.8%. This set the precedent for the use of GPUs for the training of most, if not all, deep learning models.

The deep learning revolution has been aided greatly by the open sourcing of software libraries like TensorFlow [1] and PyTorch [11] by their developers, Google and Facebook respectively. With CUDA [4] C++ backends for excellent performance on GPUs and user facing Python bindings for easy orchestration of and experimentation with complex deep learning networks, these libraries have lowered the technical barrier to entry for researchers and allowed them to focus more on foundational AI research rather than the minutia of software development.

These days it is quite common to see GitHub repositories with TensorFlow or PyTorch implementations of most popular deep learning models. Often the original authors of the deep learning papers make their own code bases publicly available, aiding greatly in the study of these models. However, these implementations are rarely deferential to considerations of performance and scalability, rendering them inefficient or even infeasible for real world large scale deployments [10]. Such limitations confine the scope of their use merely to academic study. Extensive data curation efforts and increasing deep learning model sizes compound this problem greatly. All these issues and more became prohibitively inconvenient for us when trying to use a TensorFlow implementation of PSPnet inference for a public health study involving enormous datasets and a rather constrained timeframe, motivating the development of the proposed scheme in this paper.

Scaling and large scale deployments present a host of challenges to system administrators. Working with extremely large datasets with hundreds of thousands of images is a logistic nightmare. Keeping track of images to be processed, ones that have already been processed, and even ones that require re-processing due to runtime errors requires precise and co-ordinated logging and synchronization, especially when multiple instances of the model are being run. Such involved administration is time consuming and technically out of reach for many cross disciplinary researchers. Therefore, it is important to address the qualitative shortcomings of deep learning model implementations in addition to improving their quantitatively tangible performance.

To comprehensively address all of the aforementioned issues we have developed a distributed scheme with component enhancements to the original model which aid in acceleration and structural enhancements in deployment. The unified architecture of the original model is decomposed into three separate tasks – pre-processing, deep learning, and image combine. Doing so enables us to pipeline the tasks and increase effective throughput. Performance issues due to poor implementation techniques are also addressed, with each of the three tasks requiring different strategies for acceleration. The deep learning task is accelerated using batch processing, and multiprocessing is used to speed up the image combine task. The combination of all these techniques has yielded a 1.88x increase in inference speed on a single image and a 3.4x increase in effective throughput across the entire dataset.

The rest of the paper is organized as follows. Section 2 provides background on the health study and the base TensorFlow implementation of PSPnet, with a discussion on related works. Section 3 describes the motivating challenges of the original implementations. Section 4 details the proposed component enhancements and scheme architecture. Section 5 illustrates the performance improvements and logistic benefits of our scheme. Finally, Section 6 provides concluding remarks.

2 Background and Related Work

The motivating incident for the development of the proposed scheme was a proposal to use an image segmentation model for the purposes of a wide scale public health study. The study aims to analyze the effects of urban landscapes on the mental health of its inhabitants. In order to do so Google Street View [2] images of select US neighborhoods were obtained from Google's public APIs and the composition of these images needed to be ascertained using the image segmentation model. The downloaded images were curated into a dataset of image pairs. A crowdsourcing effort then followed using this dataset where participants were presented pairs of images and asked to vote for a winner with respect to perceptual attributes like safety, beauty, and greenspace quality. Using these votes a TrueSkill [5] score was computed for each image in the dataset with respect to each perceptual attribute. Finally, a linear regression model was to be developed using the composition metrics obtained from the image segmentation model to predict the TrueSkill scores.

It was decided that for this study PSPNet would be the most appropriate image segmentation model; at the time of the health study's inception PSPNet was the state-of-the-art. A TensorFlow implementation of PSPNet, pretrained on the ADE20k dataset [14, 15] and available from a public GitHub repository [7], was used as the reference implementation. This version of the model has 150 object classes in total including building, shrub, pavement, sky, etc. which are very pertinent to the objective of the study. PSPNet provides support for multiscale inference which has the benefit of improving model accuracy by a substantial amount. In multiscale inference, the input image is rescaled by various scaling factors and each of the rescaled images is run through the model. Doing so allows the model to gather more context about the image composition. When the image is scaled down, the model is able to ascertain the global context of the image. Scaling up the image has the opposite effect where the model is able to focus on parts of the image on a more granular level, yielding local context. Running multiple passes on the same image at different scales yields a number of outputs with slightly varying predictions. All these predictions are collected, scaled down to the original scale, and averaged out to arrive at a final prediction. This prediction is far more accurate than a single scale prediction but has the unfortunate consequence of being much slower if improperly implemented, as is the case with the original implementation we were working with. A solution to this problem is discussed in further sections.

The problem of multiscale inference is one of many we encountered with the original implementation of the model and all of them needed to be addressed without exception for the health study. Before deciding on the development of the proposed scheme, we considered a number of existing solutions but ultimately found all of them inadequate owing to their respective shortcomings. Frameworks like TensorFlow and PyTorch provide facilities to scale deep learning models across multiple GPUs [8, 9]. Data parallel inference is used to process larger batches of images on multiple local GPUs in parallel. This certainly speeds up inference times over the entire dataset since larger batches are being processed, but it does not alleviate the systemic issues of the model implementation discussed in Section 3 such as linearly iterative processing loops for multiscale inference and monolithic single node architecture. Utilizing the benefits of the newer versions of these framerworks would require significant re-implementation of PSPNet, if not from scratch.

In more recent versions of these frameworks multithreaded dataloading is also supported. This proves effective in utilizing multicore processors. The dataloaders, however, did not provide all the facilities we needed. Errors such as missing or corrupted files are not handled by the dataloader and instead needs to be part of inference code. Additionally, the framework's dataloaders cannot handle files located on remote storage servers. Updating the model to use more recent versions of the frameworks might also result in subtle differences in the output and this is something the health study cannot tolerate.

3 Issues in Original Implementation

Figure 1 demonstrates that architecture of the original implementation of PSPnet. The monolithic nature of this architecture tightly couples all critical components of the model in a single nested iterative loop. This loop is what enables multiscale inference in the original implementation. The input image is first scaled to the desired scale level. The rescaled image is then cut up into fixed width block and run through the network to obtain a prediction. This process is repeated within the loop for each scale level to obtain multiple predictions which are combined to obtain a single more accurate prediction. While the approach is simple and straightforward to implement, it forgoes available avenues for acceleration to obtain that simplicity. The process is single threaded and does not fully utilize available CPU resources. While the scaling is being done by the CPU, the GPU is being idled. We address this primary issue in our model because it is paramount that resources, which are scarce to begin with, are not underutilized.

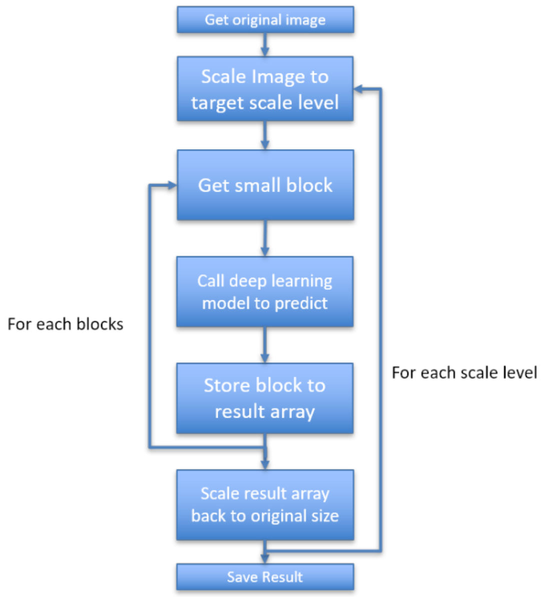
Get original image
Scale Image to target scale level
Get small block
Call deep learning model to predict
Store block to result array
Scale result array back to original size
Save Result
For each blocks
For each scale level

Fig. 1. Structure of the original model

It is typical to house large datasets in dedicated storage servers that are separate from the compute servers. This is certainly true of our use case. The original model is unable to cope with this setup because it expects input images to be available locally. In working with large datasets we found that we constantly needed to copy over small subsets of the images to the compute server to process through the model and delete them once the prediction is obtained. This tedium of manually managing the dataset is error prone and often many images are unaccounted for due to the sheer size of the dataset. A comprehensive logging and data management mechanism, capable of transparently serving remote files to the model and keeping track of progress, was necessary to address this crucial problem.

Another key drawback of the original implementation is the lack of error detection and recovery. Environment changes like system reboot, leak of memory resources, or corrupted input images are catastrophic to the model. Since manual intervention and restarts are necessary, the model needs to be constantly monitored and corrupted images need to be painstakingly removed from the dataset before running the model. Simply running the model is a significant investment of time. A new scheme that can recover from errors and meaningfully relay error logs to researchers would allow passive deployment of the model, freeing researchers from the timesink of system administration.

4 Proposed Scheme

In our proposed approach, the key techniques we used to address the performance issues are model decomposition, pipelining, batch processing, and multiprocessing. Once the model is decomposed into three tasks, each task can be accelerated using techniques appropriate to the specifics of that task. A three stage pipeline is then implemented to improve effective throughput of the model. Finally, the decomposed model is integrated into a deployment scheme that allows easy administration and scaling of the model.

4.1 Decoupling Critical Components

The first and most important concern is decomposing the monolithic model into separate yet logically consolidated components. There are three primary operations being done to obtain the prediction. First, the input image is scaled to the appropriate scale level. Second, the scaled image is run through the deep model to obtain a prediction for the rescaled image. Third and finally, predictions for all of the different rescaled images are combined to obtain a single prediction. Therefore, the decomposition was made along these operational lines. Figure 2 depicts this the process.

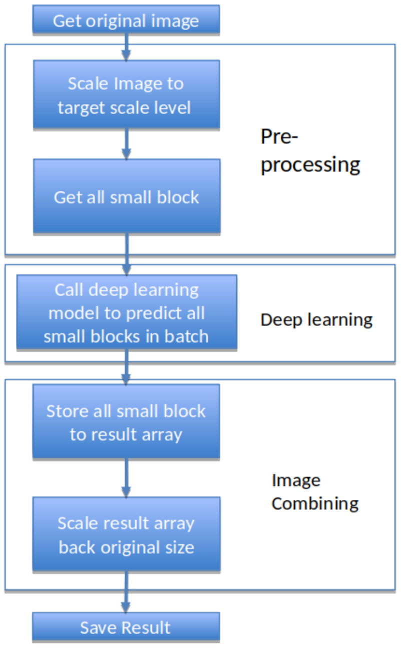
Get original image
Scale Image to target scale level
Get all small block
Pre-processing
Call deep learning model to predict all small blocks in batch
Deep learning
Store all small block to result array
Scale result array back original size
Image Combining
Save Result

Fig. 2. Model decomposition

The purpose of the three tasks is as follows:

- The **pre-processing** task takes in the input image and rescales it to all desired scale levels. From these rescaled images it generates the smaller blocks that are input to the deep learning task.
- The **deep learning** task takes in all image blocks generated by the pre-processing tasks and runs them through the deep model to obtain predictions.
- The **image combine** task combines all the results from the deep learning task, scales the results back to the original image size, and computes an average of all the results to obtain a final prediction for the input image.

Having obtained three distinct and independent tasks, we can implement a three stage pipeline. Although the time cost of a single image does not decrease, pipelining increases throughput across the entire dataset which, for our purposes, was a paramount concern. It should be noted that each of the three tasks are performed on either the CPU or the GPU but never both. The pre-processing and image combine phases are CPU tasks while the deep learning phase is a GPU task. This separation is very useful because when the three tasks are pipelined as shown in Figure 3 below, the issue of inadequate usage of resources goes away since neither the CPU nor the GPU is ever idle for too long waiting on a specific task. As soon as the pre-processing by the CPU is done and the result is dispatched to the GPU, the CPU is tasked with the next pre-processing task. Similarly, when the GPU is done with a deep learning task, another deep learning process is waiting in the pipeline queue.

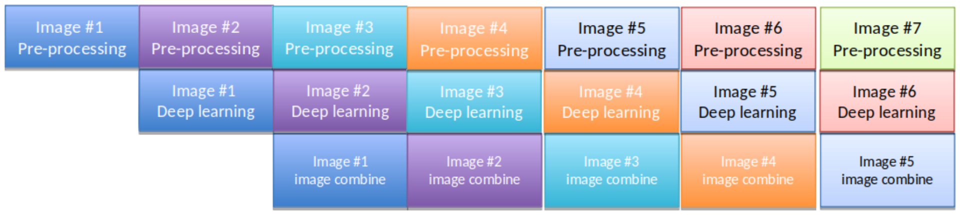

Fig. 3. Three-stage pipeline

4.2 Batch Processing and Multiprocessing for Multiscale Inference

When the pre-processing task completes, the input image is available to the GPU task at all required scales level, with each of the rescaled images having been split into smaller blocks. This means that the deep learning task can process all image blocks in a single batch. The benefit of batch processing is that the processing is done in parallel and is largely invariant with regard to batch size, if the GPU has enough memory to accommodate the batch. This allows us to overcome the time penalty of multiscale processing found in the original model because each rescaled image is not processed linearly one after another but rather simultaneously in a single batch. Batch processing is a technique that is agnostic to the model being used and can be generalized to any deep learning model that is being run on a GPU.

Having obtained the individual results from the deep learning task, the image combine task needs to combine and average the results to obtain the final prediction. This task can be accelerated by the use of multiprocessing as shown in Figure 4.

Multiple processes are spawned in parallel and the relevant scale level arguments are passed to all of them. Each process then takes the output blocks pertinent to its assigned scale level from the deep learning task, scales them according to the scale level argument, and adds them to the final result.

The use of batch processing and multiprocessing reduces the time overhead of multiscale inference to the point of almost converging multiscale and single scale inference times. We can also adjust batch sizes to ensure that the deep learning task utilizes the GPU to the fullest. The multiple processes of the image combine task run on separate CPU threads which furthers our objective of effective utilization of CPU resources.

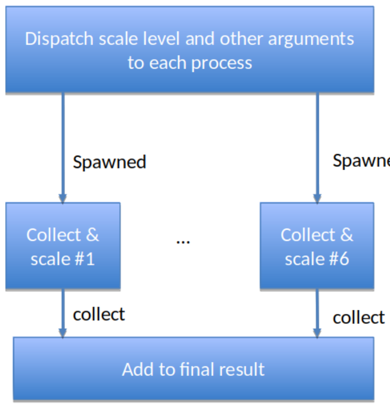

Fig. 4. Multiprocessing in image combine task

4.3 Multi-node Deployment

Batch processing and multiprocessing address issues of performance. In order to solve the logistical challenges of deployment at scale, however, the decomposed model needs to be scaffolded by a distributed scheme capable of utilizing disparate resources, provide logging and data management facilities, and have a graceful fallback in case of errors. Figure 5 displays our scheme possessing all these qualities. The scheme is a client-server architecture [13] enabled by the pervasive use of the Socket.IO [12] python library for metadata transfer and SQL database for logging progress and errors.

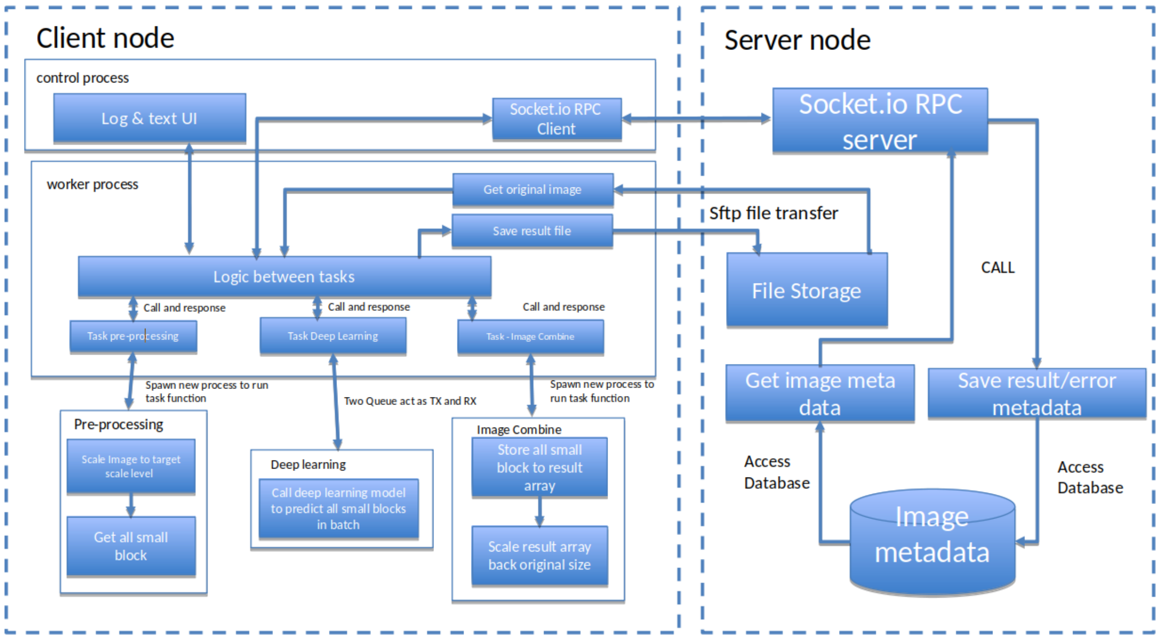

Fig. 5. Architecture of the deployment scheme

Server Node: The server node is deployed on the file storage server. The backbone of the server node is a SQL database that contains metadata about the images in dataset such as filename, file location, and whether a particular file has been processed. The server node is responsible for serving up images from the

storage server by use of the SFTP protocol when clients request specific files. Metadata is served by use of the Socket.IO server to which Socket.IO clients connect to from the client nodes.
Client Node: The client node is deployed on the compute server. The client node is primarily composed of two processes – the control process and the worker process. The control process contains the Socket.IO client which connects to the analogous server on the server node to request file metadata. Having obtained the metadata from the server node, the worker process is first tasked with obtaining the relevant file from the storage server by use of the SFTP protocol. The worker process then spawns separate processes for each of the three model tasks – pre-processing, deep learning, and image combine.

The steps involved in the deployment and call-response engagement between a client and a server node is described below:

– The database is first initialized with the list of images in the dataset. In this state the database indicates that none of the images have been processed.
– The server and client nodes are started on the respective servers.
– The client node's control process, by use of the Socket.IO client, requests a task.
– The server node queries the database to first figure out which of the images are yet to be processed and then furnishes metadata about one such image to the client node.
– The client node now uses SFTP file transfer to obtain the unprocessed image file from the storage server. File location is part of the metadata provided by the server node.
– The worker node spawns the three model tasks to process the image.
– The result of the inference is passed by the client node to the server node which writes it to the database.
– The above steps are repeated until all the images are processed.

It should be noted that since the three model tasks are pipelined, the worker process requests new image metadata as soon as the pre-processing task completes, and the next image is queued up for processing. The deployment scheme has been used to scale PSPNet across multiple compute servers. Separate client nodes can be deployed to each of the available compute servers, all of whom are served by a single server node on the storage server. Communication between client nodes and the server node is asynchronous and messages are queued. Since the server node is computationally lightweight, multiple server nodes can be deployed on a single storage server in case a single server node is unable to adequately serve all client nodes. The ACID properties (Atomicity, Consistency, Isolation, and Durability) of the database implicitly disallows contention issues and stale data during data access by multiple server nodes.

5 Evaluation

5.1 Evaluation Setup

For the purposes of testing, a Nvidia V100 GPU paired with a 24 core, 48 thread Intel Xeon Platinum 8168 was used. Testing of multiscale inference was performed using 4, 3, 2, and 1 scale levels, where "1" means single scale inference. Scale levels are picked between [0.5, 0.75, 1.0, 1.25]. All four scale levels were used for 4-scale inference, [0.75, 1.0, 1.25] scale levels were used for 3-scale inference, [0.5, 1.0] scale levels were used for 2-scale inference, and finally the image at original scale was used for single scale inference. Time measurements were taken programmatically by utilizing related system time modules and packages to minimize interference with other processes.

5.2 Single Inference Speedup

Figure 6 demonstrates the inference times for the original model and our modified model. It can be observed that with the original model inference times for multiscale inference increases almost linearly with the number of scale levels used. This is as expected since the original model used a iterative loop to linearly process the image at the various scale levels. With our modifications, however, the time penalty is largely overcome. The pre-processing task time increases with increase in scale levels and so does the image combine task time. However, multiprocessing measures mitigates the latter while the former is a negligible increase relative to the total inference time.

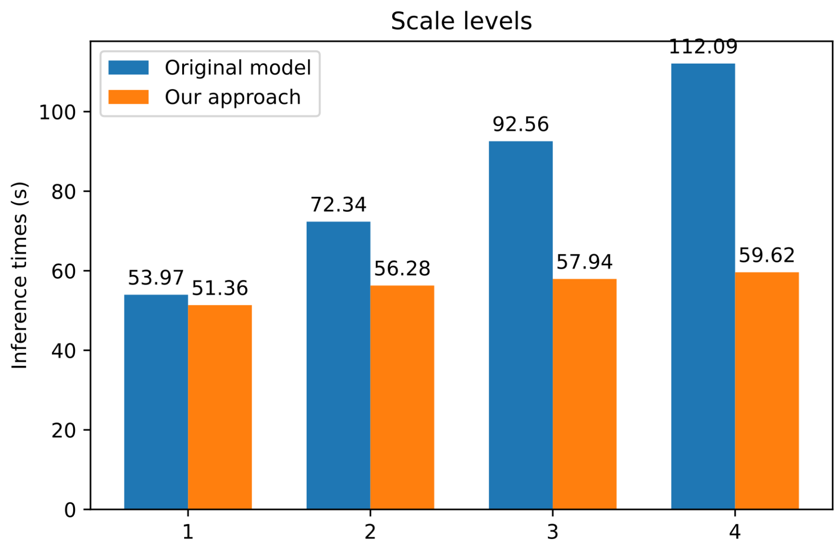

Fig. 6. Comparison of inference times between original model and our approach

Table 1. Time to perform each of the three model tasks.

Scale levels	Pre-process (s)	Deep learning (s)	Image combine (s)	Total (s)
4	1.72	29.29	28.61	**59.62**
3	1.30	29.20	27.44	**57.94**
2	0.91	29.22	26.15	**56.28**
1	0.45	29.22	21.69	**51.36**

Table 1 gives a more detailed breakdown of the time taken by the individual tasks. The deep learning and image combine tasks take comparable time while the pre-processing is an order of magnitude faster.

Table 2 demonstrates that the pre-processing time scales linearly with the scale levels since it needs to rescale and cut up the original image and this is done in a linear code block. The deep learning task time remains unchanged due to batch processing. All the image blocks are loaded into the GPU and processed simultaneously. The image combine task times reveal the benefits of multiprocessing. For single scale inference a single process is needed to collect and scale result blocks to arrive at the final prediction. Multiscale inference requires spawning and synchronizing multiple processes which adds overhead. However, the overhead is manageable especially compared to the cost of multiscale inference with the original model. An exploration of image combine times with a non-parallel implementation demonstrates the enormous benefit of the multiprocessing paradigm.

Table 2. Time to perform image combine task with and without multiprocessing.

Scale levels	With multiprocessing (s)	Without multiprocessing (s)
4	28.61	78.72
3	27.44	60.00
2	26.15	39.89
1	21.69	19.97

5.3 Pipelining Speedup

We were working with datasets with hundreds of thousands of images and typically used 4-scale inference to process all of the images. Pipelining enabled us to increase throughput over the dataset by a factor of 3.4.

For this testing we used 100,00 images, a subset of our dataset. Table 3 compares time metrics to process this subset of data with our model and the original model.

Table 3. Time to process 100,00 images using our model and original model.

Time (s)	**Our model**	**Original model**
Total time	3,303,028	11,235,537
Per image average	33.03	112.36

A point of note is the lag time between the pre-process task and deep learning task in the pipeline. The three component tasks of our model take different amounts of time to complete, with the pre-process task being an order of magnitude faster than the subsequent two. This is responsible for the aforementioned lag time. This does not however mean that the CPU is completely idle because it is being put to use for the image combine task. This improves resource utilization compared to the original model.

5.4 Additional Benefits of Proposed Deployment Scheme

Besides acceleration, the proposed deployment scheme is able to largely automate the administration of the model at scale. Once the database is initialized and the nodes started, the scheme does not need any manual intervention. The nodes can be setup to restart in case of system reboots and since the server can provide state information using the metadata in the database, the restarted nodes can resume without needing to reprocess any images.

The error logging facility provided by the scheme is also useful especially when working with very large datasets. Corrupted images and missing files are very commonplace and our scheme gracefully handles such incidents. The presence of corrupted images is virtually impossible to detect even with the more comprehensive data cleansing efforts. Even visual inspection of images is often inadequate to detect subtle encoding errors since they appear undamaged and such errors are only spotted when the model tries to parse the file. In these instances, errors are succinctly logged and the scheme moves on to the next image in the dataset. This is in contrast to the original model which fails catastrophically when it encounters such errors and requires constant monitoring and restarts by a system administrator.

Scaling to elastic resources is another strength of the scheme. When more GPUs or CPUs become available more instances of the client node can be spun up with ease to consume those resources. If the resources are reclaimed by another user, the running client node can quit at any moment (without waiting for all the assigned images to be processed) while not affecting other client nodes.

The above salient features of the scheme were designed as a reaction to the various problems we encountered while attempting to use a deep learning to process very large datasets at scale and therefore provide a useful and practical solution for the needs of researchers attempting to do the same at scale.

6 Conclusion

In this paper we have presented a distributed scheme to accelerate PSPNet and scale it across disparate resources. Structural changes to the model enabled us to decompose the model and accelerate the individual components using batch processing, multiprocessing, and pipelining. In aggregate, our approach accelerates 4-scale inference on a single image by a factor of 1.88 while effective throughput across the entire dataset increases by a factor of 3.4. The deployment scheme that surrounds the model components enables dataset management and processing at scale across a range of heterogeneous resources, while ensuring appropriate and effective use of resources.

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015), http://tensorflow.org/, software available from tensorflow.org
2. Anguelov, D., Dulong, C., Filip, D., Frueh, C., Lafon, S., Lyon, R., Ogale, A., Vincent, L., Weaver, J.: Google street view: Capturing the world at street level. IEEE Computer **43**, 32–38 (06 2010). https://doi.org/10.1109/MC.2010.170
3. Deng, J., Dong, W., Socher, R., Li, L., Kai Li, Li Fei-Fei: Imagenet: A large-scale hierarchical image database. In: 2009 IEEE Conference on Computer Vision and Pattern Recognition. pp. 248–255 (2009). https://doi.org/10.1109/CVPR.2009.5206848
4. Ghorpade, J., Parande, J., Kulkarni, M., Bawaskar, A.: GPGPU processing in CUDA architecture. CoRR **abs/1202.4347** (2012), http://arxiv.org/abs/1202.4347
5. Herbrich, R., Minka, T., Graepel, T.: Trueskill(tm): A bayesian skill rating system. In: Advances in Neural Information Processing Systems 20. pp. 569–576. MIT Press (January 2007), https://www.microsoft.com/en-us/research/publication/trueskilltm-a-bayesian-skill-rating-system/
6. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1. p. 1097–1105. NIPS'12, Curran Associates Inc., Red Hook, NY, USA (2012)
7. Kryvoruchko, V.: Pspnet-keras-tensorflow. github.com/Vladkryvoruchko/PSPNet-Keras-tensorflow (2017)
8. Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., Chintala, S.: Pytorch distributed: Experiences on accelerating data parallel training. Proc. VLDB Endow. **13**(12), 3005–3018 (Aug 2020). https://doi.org/10.14778/3415478.3415530, https://doi.org/10.14778/3415478.3415530
9. McKerns, M.M., Strand, L., Sullivan, T., Fang, A., Aivazis, M.A.G.: Building a framework for predictive science. CoRR **abs/1202.1056** (2012), http://arxiv.org/abs/1202.1056
10. Munappy, A., Bosch, J., Olsson, H.H., Arpteg, A., Brinne, B.: Data management challenges for deep learning. In: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). pp. 140–147 (2019). https://doi.org/10.1109/SEAA.2019.00030
11. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates, Inc. (2019), http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
12. Rauch, G.: Socket.io: the cross-browser websocket for realtime apps. (2012), http://socket.io/
13. Sulyman, S.: Client-server model. IOSR Journal of Computer Engineering **16**, 57–71 (01 2014). https://doi.org/10.9790/0661-16195771
14. Zhou, B., Zhao, H., Puig, X., Fidler, S., Barriuso, A., Torralba, A.: Scene parsing through ade20k dataset. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 5122–5130 (2017). https://doi.org/10.1109/CVPR.2017.544
15. Zhou, B., Zhao, H., Puig, X., Fidler, S., Barriuso, A., Torralba, A.: Semantic understanding of scenes through the ADE20K dataset. CoRR **abs/1608.05442** (2016), http://arxiv.org/abs/1608.05442