Accelerating Model Search with Model Batching

Extended Abstract

Deepak Narayanan, Keshav Santhanam, Matei Zaharia
Stanford University

ABSTRACT

GPUs have become the computing platform of choice for deep learning applications. However, leveraging the ever-increasing computational power of these GPUs is challenging for many workloads, resulting in poor resource utilization. In this work, we explore techniques to utilize the GPU more effectively for certain deep learning tasks; in particular, we propose simultaneously running multiple models (called a *model batch*) on the same GPU (kernel-level parallelism), which also allows data preprocessing to be shared among the different models. Our results demonstrate performance gains of up to 9.2× for training and 13.5× for inference compared to the traditional 1-model-per-GPU configuration.

1 INTRODUCTION

Over the last five years, deep learning has become ubiquitous for a variety of tasks, including image recognition, machine translation, and speech recognition. To train these deep learning models more efficiently, GPUs have become the default computing platform [4]. Modern GPUs are capable of computational throughputs in excess of several Teraflops; leveraging all of this computational capacity for certain problem domains is a challenge.

For image-related tasks, Convolutional Neural Networks (CNNs) have proven to be the model of choice. Recent research has shown that shallow CNNs (< 10 layers) can often be effective in certain task regimes (for example, analyzing video streams [7, 16, 17]). These smaller models have also been deployed with great success in mobile and embedded settings [5, 18] where compute and memory resources are limited. However, training and inference of these shallow CNNs on a GPU proves to be inefficient; our experiments show that the processing pipeline is often bottlenecked by input preprocessing, which includes input decoding, data augmentation, etc. In addition, training and inference in these regimes often do not contain enough floating point operations to sufficiently saturate the GPU's many execution units. Sequence-to-sequence models like LSTMs, which have inherently *less parallelism*, are also unable to fully utilize GPUs [2].

Training a high-accuracy deep learning model for a specific task is not easy; in practice, deep learning models often require hyperparameter and architecture tuning [3, 6, 10] for optimal accuracy. This usually entails training a number of similar models [12, 19], and picking the one with highest validation accuracy. In addition, a number of state-of-the-art models for a variety of tasks [13] are in fact *ensembles*. This raises the natural question: can we leverage the fact that users frequently utilize a number of similar models to achieve better GPU utilization? To this end, we designed and built ModelBatch, a system that batches models in addition to inputs, to improve hardware utilization by performing training or inference on a number of similar models at once.

SysML'18, February 2018, Stanford, California USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

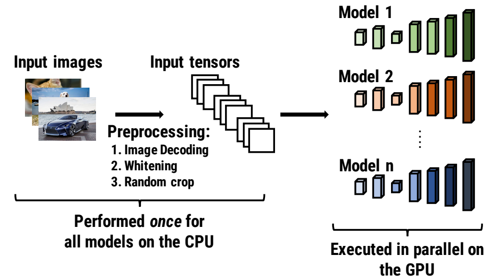


Figure 1: ModelBatch architecture for image classification models. Input images are preprocessed once across all models and then offloaded to the GPU. Model computations are launched in parallel.

There are two main benefits to this approach: (a) Better GPU utilization by giving the GPU more floating point operations (in the form of multiple models), and (b) Amortization of I/O and preprocessing over the many models being trained, shifting the bottleneck of the end-to-end pipeline back to compute.

Our experiments using ModelBatch demonstrate that batching models does in fact improve GPU utilization.

2 SYSTEM DESIGN

ModelBatch consists of two main stages: 1) A preprocessing step, performed on the CPU, that is shared among the different models, and 2) A computation step (convolutions, matrix multiplications, pooling, etc.), performed on the GPU and on a per-model basis. To obtain good performance while batching CNN models, ModelBatch also needs to choose the right convolution algorithm. ModelBatch's architecture is shown in Figure 1.

Preprocessing. For image classification models, ModelBatch's preprocessing step involves image decoding, as well as other data augmentation. As we show in Figure 3, preprocessing is often a bottleneck, especially for relatively shallow architectures, so amortizing this cost over multiple models is often beneficial. After preprocessing, input data must be moved from the CPU to the GPU – ModelBatch hides the latency of this data movement by double buffering [14], ensuring that the transfer of one minibatch to the GPU is performed concurrently with processing of the previous minibatch.

Computation. ModelBatch's computation step involves launching the kernels associated with the different models in parallel. This is made possible using NVIDIA's CUDA streams. Kernels on different streams can be launched (and hence executed) in parallel. For example, two GEMM kernels operating on disjoint inputs can be executed in parallel on different GPU Streaming Multiprocessors when placed on separate streams. ModelBatch creates a new CUDA stream for each independent model and schedules kernels for a particular model exclusively on its respective stream.

Optimizing convolutions. Previous work has demonstrated that the choice of convolution algorithm significantly impacts performance of CNNs [9, 11]. Furthermore, the optimal algorithm for a

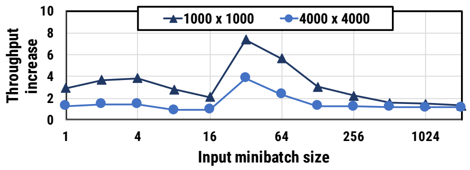


Figure 2: Throughput increase of launching 64 kernels in parallel compared to a single kernel for the forward pass of 1000×1000 and 4000×4000 fully connected layers, for different input minibatch sizes. Commonly used input minibatch sizes usually range from 16 to 256.

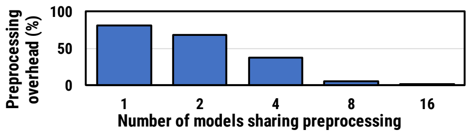


Figure 3: Overhead of preprocessing (computed as the percentage of time spent in the preprocessing step) while training a stripped-down AlexNet model. The number of models sharing preprocessing is varied from 1 to 16, and preprocessing is *not* pipelined with computation.

single, serial convolution often requires a large amount of scratch workspace, leading to poor memory locality and out-of-memory errors when used in parallel. ModelBatch addresses this by scaling the maximum scratch workspace memory limit with the inverse of the model batch size, and sharing scratch workspace memory between layers in a single model.

3 EVALUATION

Our evaluation seeks to answer the following questions:

- How helpful is concurrent kernel execution for GPU utilization?
- How much of a bottleneck is input preprocessing in realistic deep learning workloads?
- How effective are ModelBatch's optimizations when used in conjunction?

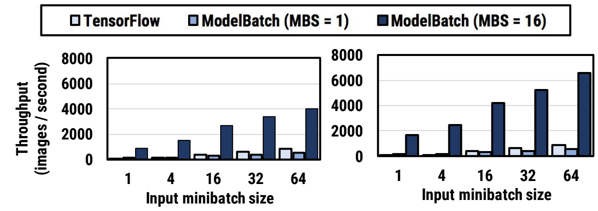
Experiment Setup. All experiments in this section were run on a machine with 512 GB of memory and 28 CPU cores, and an NVIDIA P100 GPU. Experiments were run using CUDA 8.0 and CuDNN 6.0. Where relevant, we used TensorFlow 1.3 [1], compiled from source.

Concurrent Kernels. We illustrate the effectiveness of concurrent kernel launching in Figure 2, which shows the throughput increase gained by launching 64 kernels in parallel compared to a single kernel, for the forward pass of two fully connected layers (1000×1000 and 4000×4000) commonly seen in CNN architectures like AlexNet [8] and OverFeat [15], for input minibatch sizes varied from 1 to 2048.

We observe three key results: (a) Model batching helps with GPU utilization for *both* layer sizes, (b) The benefits of model batching are more pronounced for the smaller layer (5.63× for input minibatch size of 64), and (c) A single model does not reach peak device throughput, even for very large input minibatch sizes (not pictured in Figure 2).

Input Preprocessing. Figure 3 shows the overhead of preprocessing (computed as the percentage of time spent in the preprocessing step) while *training* a 5-layer CNN (inspired by the 8-layer AlexNet network [8]) executed in TensorFlow on ImageNet data (each image is 227×227 pixels) – this is similar to the specialized models used in [7]. Different numbers of models are trained while *sharing* the same input pipeline. Model computations are executed *serially*.

We highlight two key results: (a) The preprocessing overhead for a single model is high (81.51%), and (b) Sharing preprocessing among multiple models reduces this overhead drastically (1.94% when preprocessing is shared across 16 models).



(a) Training. (b) Inference.

Figure 4: Throughputs (in images per second) for training (left) and inference (right) of 16 identical 5-layer AlexNet-like models on ImageNet, using TensorFlow (which processes a single model at a time), ModelBatch with a model batch size of 1, and ModelBatch with a model batch size of 16, for different input minibatch sizes.

End-to-End Training and Inference. Finally, we show the benefit of using ModelBatch in an end-to-end setting. We use the same 5-layer stripped-down AlexNet model as before, on the ImageNet dataset.

Figure 4a shows the results for training. We run ModelBatch with model batch sizes (MBS) of 1 (no model batching) and 16, and also run the same model on TensorFlow as a baseline. We observe that ModelBatch with MBS of 1 is competitive with TensorFlow, and that ModelBatch with MBS of 16 improves throughput by up to 9.2× compared to ModelBatch with MBS of 1, and by up to 7.2× compared to TensorFlow.

Figure 4b shows the results for inference. As before, ModelBatch with MBS of 1 is competitive with TensorFlow. In addition, ModelBatch with MBS of 16 improves throughput by up to 13.5× compared to ModelBatch with MBS of 1, and by up to 10.9× compared to TensorFlow.

4 FUTURE WORK

ModelBatch is intended to be effective for a range of model architecture types, even though this paper only considered image classification CNNs. In particular, we are interested in evaluating how well model batching works for sequence-to-sequence models like LSTMs and GRUs, which have inherently less parallelism [2]. We are also interested in studying how model batching can be used for latency hiding purposes in distributed settings. Finally, we wish to test ModelBatch on a model / hyperparameter search application, like Hyperband [10] or Google's Neural Architecture Search [19], which currently take hundreds of GPU hours to run to completion. These applications feature training and inference on similar but *not* identical models, which makes scheduling the computation of the different models more challenging.

5 CONCLUSION

ModelBatch aims at accelerating model search workloads by *batching* models, allowing for concurrent kernel computation, and shared data preprocessing. ModelBatch produces speedups of up to 9.2× and 13.5× on training and inference workflows respectively.

ACKNOWLEDGMENTS

We thank Cody Coleman, Edward Gan, Daniel Kang, Pratiksha Thaker, and the many members of the Stanford InfoLab for their valuable feedback on this work. This research was supported in part by affiliate members and other supporters of the Stanford DAWN project – Google, Intel, Microsoft, Teradata, and VMware – as well as industrial gifts and support from Toyota Research Institute, Juniper Networks, Keysight Technologies, Hitachi, Facebook, Northrop Grumman, NetApp, the NSF under grants DGE-1656518 and CNS-1651570, and DARPA under grant No. FA8750-17-2-0095 (D3M).

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, Vol. 16. 265–283.

[2] Jeremy Appleyard, Tomas Kocisky, and Phil Blunsom. 2016. Optimizing Performance of Recurrent Neural Networks on GPUs. *arXiv preprint arXiv:1604.01946* (2016).

[3] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-parameter Optimization. *J. Mach. Learn. Res.* 13 (Feb. 2012), 281–305. http://dl.acm.org/citation.cfm?id=2188385.2188395

[4] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. 2017. DAWNBench: An End-to-End Deep Learning Benchmark and Competition. In *NIPS ML Systems Workshop*.

[5] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

[6] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. 2017. Population Based Training of Neural Networks. *arXiv preprint arXiv:1711.09846v2* (2017).

[7] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1586–1597.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[9] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4013–4021.

[10] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2016. Hyperband: Bandit-based configuration evaluation for hyperparameter optimization. (2016).

[11] Karas Pavel and Svoboda David. 2013. Algorithms for efficient computation of convolution. In *Design and Architectures for Digital Signal Processing*. InTech.

[12] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Quoc Le, and Alex Kurakin. 2017. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041* (2017).

[13] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision* 115, 3 (2015), 211–252.

[14] José Carlos Sancho and Darren J Kerbyson. 2008. Analysis of double buffering on two different multicore architectures: Quad-core Opteron and the Cell-BE. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 1–12.

[15] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. 2013. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229* (2013).

[16] Haichen Shen, Seungyeop Han, Matthai Philipose, and Arvind Krishnamurthy. 2016. Fast video classification via adaptive cascading of deep models. *arXiv preprint arXiv:1611.06453* (2016).

[17] Xin Wang, Yujia Luo, Daniel Crankshaw, Alexey Tumanov, and Joseph E Gonzalez. 2017. IDK Cascades: Fast Deep Learning by Learning not to Overthink. *arXiv preprint arXiv:1706.00885* (2017).

[18] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2017. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. *CoRR* abs/1707.01083 (2017). arXiv:1707.01083 http://arxiv.org/abs/1707.01083

[19] Barret Zoph and Quoc V Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).