Performance Prediction of GPU-based Deep Learning Applications

Eugenio Gianniti
Politecnico di Milano
Milan, Italy
Email: eugenio.gianniti@polimi.it

Li Zhang
IBM T. J. Watson Research Center
Yorktown Heights, NY, Unites States
Email: zhangli@us.ibm.com

Danilo Ardagna
Politecnico di Milano
Milan, Italy
Email: danilo.ardagna@polimi.it

***Abstract*—Recent years saw an increasing success in the application of deep learning methods across various domains and for tackling different problems, ranging from image recognition and classification to text processing and speech recognition. In this paper we propose and validate an approach to model the execution time for training convolutional neural networks (CNNs) deployed on GPGPUs. We demonstrate that our approach is generally applicable to a variety of CNN models and different types of GPGPUs with high accuracy, aiming at the preliminary design phases for system sizing.**

***Index Terms*—Convolutional neural networks; deep learning; performance prediction; general purpose GPUs.**

I. Introduction

Nowadays, convolutional neural networks (CNNs) find application across industries, most notably for image recognition and classification tasks, which represented the first successful adoption of the technique [1]. Ranging from medical diagnosis to public security, deep learning (DL) methods are fruitfully exploited in a wide gamut of products. There is ongoing work on the technique's adaptation for other use cases, like speech recognition [2] and machine translation [3].

Over time, many frameworks have been developed to provide high level APIs for CNN design, learning, and deployment. Among the most well known, we recall Torch, PyTorch, TensorFlow, and Caffe. Usually DL models are trained by relying on GPGPU systems (even in experimental clusters [4]), which allow to achieve from 5 up to 40x time improvement when compared to CPU deployments [5].

In spite of the widespread adoption of DL systems, still there are few studies taking a system perspective which aim at investigating how, e.g., the training time changes when running on different GPGPUs or by varying the number of training iterations or the batch size [5], [6]. DL applications are characterized by a large number of design choices that often do not apply readily to other domains or hardware configurations, up to the point that even advanced users with considerable DL expertise fail at identifying optimal configuration settings [6]. The time required to train a new DL model is generally unknown in advance. Because of this, performance analysis is usually done empirically through experimentation, requiring a costly setup [5]. Performance modeling can help, e.g., to establish service level agreements with end users or to predict the budget to train or run production DL models in the cloud.

In this paper, we present a method to learn performance models for CNNs running on a single GPGPU. Our goal is to lead new users with limited previous experience from an initial test deployment to real scale applications. We propose a gray box *per layer* approach where modeling is performed layer by layer and the only explanatory variable is computational complexity. This technique allows for a great deal of generality, since partial layer predictions are easy to sum, thus obtaining an overall performance estimate for the full CNN, even if the specific network schema has never been considered as part of the training set. Due to this, the approach is preeminently interesting during the initial design stages, for instance to compare different alternative CNNs and deployments in terms of performance. What is more, designers can get a feeling of the resulting performance without ever needing to hit a cluster or the cloud for experiments, which is advantageous regarding both saved work hours and plain monetary savings.

In our experimental campaign, we considered three popular DL models implemented with the Caffe framework. Yet our methodology is not constrained in any way neither to a specific framework, nor to a GPGPU model. The outcomes show that we can obtain per layer models general enough to yield relative errors below 10% on average across different CNN architectures and below 23% in the worst case.

II. Per Layer Model

All CNNs comprise a number of layers belonging to a limited collection of basic categories. Building upon this observation, we propose to learn several linear regression models, in order to characterize common layer types. In this way, it is possible to estimate the performance of a wide range of CNNs even without previous experience with the specific structure, just relying on these low level layer models. Our approach is based on two basic assumptions, in order to make the problem easily tractable and improve model generality.

When working with GPUs, applications attain their best performance when they fully leverage the data parallel architecture, hence we expect CNN designers, as well as users, to tune networks accordingly. Such a consideration means that, mostly, the execution of different layers will not overlap, whence follows that layer running time predictions can just be summed to obtain an estimate of the overall execution time:

$$\hat{t}^{\text{CNN}} = I \sum_{l \in L} \hat{t}_l, \tag{1}$$

Table I
OPERATIONS PER OUTPUT PIXEL

Layer	Forward	Backward
Conv	$H_\text{f}W_\text{f}C_\text{in}C_\text{out}$	$(2H_\text{f}W_\text{f}C_\text{in}+1)\,C_\text{out}$
FC	$H_\text{in}W_\text{in}C_\text{in}C_\text{out}$	$2H_\text{in}W_\text{in}C_\text{in}C_\text{out}$
Loss	$4C_\text{out}-1$	$C_\text{out}+1$
Norm	$5C_\text{out}+C_\text{n}-2$	$8C_\text{out}+C_\text{n}-1$
Pool	$H_\text{f}W_\text{f}C_\text{out}$	$(H_\text{f}W_\text{f}+1)\,C_\text{out}$
ReLU	$3C_\text{out}$	$4C_\text{out}$

where I is the total number of iterations.

The second aspect to take care of is the choice of features to feed into the regression models. A simplistic idea could be using all the various hyper-parameters as features, but this would make for a difficult to interpret and hardly generalizable formulation. On the other hand, we propose to summarize all the relevant characteristics in a single feature: layers computational complexity in terms of simple primitives available on GPGPUs, a good metric for layer workload.

To exemplify the derivation of computational complexity from network hyper-parameters, here we discuss the method for convolutional layers. The convolutional part operates on 3D tensors. Let us denote with C the number of channels, H the number of rows or height, W the number of columns or width. The amount of zero padding on each side of the matrices is P, whilst the stride is S. Subscripts distinguish properties of the input, output, and filters, e.g., H_in, H_out, H_f. Cardinalities are used as a shorthand for index sets, as in $i \in H_\text{in}$.

Some layers just apply predetermined operations, possibly depending on hyper-parameters under users' control. In contrast, convolutional and fully connected layers have a set of learnable weights that evolve during the training phase via back propagation. The number of weights depends on their hyper-parameters. Each output channel is obtained by convolving a different filter with the input tensor, hence the count of learnable parameters is given by:

$$(H_\text{f}W_\text{f}C_\text{in}+1)\,C_\text{out}. \tag{2}$$

Convolution entails multiplying filters of size $H_\text{f}W_\text{f}C_\text{in}$ element-wise with input activations and producing as output the sum of all these partial products and an additive bias, hence there are C_out filter-bias pairs that contribute all the entries in the tensor plus one coefficient.

The 3D tensors involved in CNNs contain all the partial values, called activations, obtained via the incremental transformations operated by filters. In practice, layers take an input tensor and apply a filter to its entries, thus yielding an output tensor with a possibly different layout. It is possible to compute output dimensions given layer hyper-parameters, specifically filter sides, padding around the edges, and stride. Tensor sizes are relevant because they appear in the formulas for computational complexity, since every output activation comes from one of the several applications of the filter to its inputs: it is common to consider complexity per pixel, as the overall layer operations count is always directly proportional to $H_\text{out}W_\text{out}$. In particular, continuing our example, convolutional layers can be formalized as the following expression for all $(i,j,k) \in H_\text{out} \times W_\text{out} \times C_\text{out}$:

$$y_{ijk} = b_k + \sum_{t \in H_\text{f}} \sum_{u \in W_\text{f}} \sum_{l \in C_\text{in}} w_{tulk} x_{\tilde{\imath}\tilde{\jmath}l}, \tag{3}$$

where $\tilde{\imath} = \varphi(i,t)$ and $\tilde{\jmath} = \psi(j,u)$, while x and y are, respectively, input and output activations. φ and ψ associate output and filter indices to the input needed to convolve each activation and their specific functional forms depend on the CNN and its hyper-parameters, in particular padding and stride, but they do not affect the derivation of complexity. Overall, you multiply all the weights times the input activations and accumulate the products on the bias once per output channel, hence convolutional layers require $O(H_\text{f}W_\text{f}C_\text{in}C_\text{out})$ operations per output pixel. Similar considerations enable determining the computational complexity for back propagation. Without spelling out the full details for space limitations, note that propagating deltas to biases requires one operation per channel, whilst doing so for parameters and inputs costs twice as much as the forward pass, because it fundamentally amounts to following the convolution backwards once for weights and once for activations. All in all, each output pixel requires $O((2H_\text{f}W_\text{f}C_\text{in}+1)\,C_\text{out})$.

Table I shows formulas for all the kinds of layers, with the computational complexity per output pixel for both the forward and backward passes. Now, using these formulas it is possible to build a dataset where the operations count is associated with the measured execution times of both passes: given this data, we can build a series of models where computational complexity is the only explanatory variable. For every layer category and direction we learn, following the theory for linear regression, a model of the form:

$$t_l = \beta_{0l} + \beta_{1l} c_l + \varepsilon_l, \tag{4}$$

is considered, where t_l is the execution time, c_l the complexity, and $\varepsilon_l \sim \mathcal{N}(0, \sigma_l^2)$ random errors. With the estimated coefficients $\hat{\boldsymbol{\beta}}$ it is possible to predict both forward and back propagation time, then all the relevant contributions are added to obtain the time taken for one iteration. Multiplying by the overall number of iterations and summing the terms due to each layer, as in (1), yields a prediction for the full run. In particular, it is possible to predict both training and deployment execution times, depending on whether back propagation terms are included in the sum.

The choice of using only computational complexity as independent variable confers a lot of generality, allowing to learn a set of models on data coming from a limited selection of CNNs and to apply it nonetheless to different networks. Anyhow, not adding explicitly any contribution related to hardware in the formulation makes every trained model specific of the deployment where data is extracted.

III. EXPERIMENTAL RESULTS

In this section we report numerical results to support and validate our proposed modeling techniques. In order to provide a reproducible experimental setting, we consider AlexNet [1], GoogLeNet [7], and VGG-16 [8] as CNNs, while the training and validation datasets are the ones released

for ILSVRC2012.[1] We collected data from a GPU-equipped computational node, which has an Intel Xeon E5-2680 v2 2.80 GHz 10-core processor, an NVIDIA Quadro M6000 GPU, and runs CentOS 6.8. The considered GPU boasts 3072 CUDA cores, 12 GB dedicated RAM with a 317 GB/s bandwidth, overall amounting to 7.0 TFLOPS.

We performed several runs of the three CNNs with varying batch sizes and iterations numbers. In every configuration we collected mainly two logs, one for the regular learning procedure and one for timing runs. The Caffe framework is instrumented to record the precise execution times for all the layers when the *timing* flag is set. Datasets for the per layer models associate the average layer running times and the complexity determined via the formulas in Table I. On the other hand, we extracted from ordinary execution logs the time taken to perform both the training and testing phases of the CNNs, thus obtaining validation data.

A. Preliminary Observations

Since per layer models can be considered gray box approaches, they provide not only a performance prediction device, but also some interesting insight about the way CNNs work. Some layers, though different, can be merged into a single category without degrading the goodness of fit. Convolutional and fully connected layers are an intuitive example of this behavior. A less obvious similarity was found between rectified linear units and dropout layers. Likely the similar performance behavior is due to the fact that both need to loop through activations and act based on a point-wise condition.

Pooling, in contrast, requires the adoption of multiple categories based on stride. In order to highlight this aspect, in the following we report some preliminary results on GoogLeNet from the GPGPU deployment described in detail in Section III-B. In these plots, different markers represent data coming from timing runs with different batch sizes, the solid line is the model of layer execution times against complexity obtained via linear regression, and the dotted lines are the boundaries of the 95% confidence interval around the predicted mean layer time. Figure 1a clearly shows two different behaviors for layers operating at $S = 1$, which mostly lay on the smaller slope line, and for the ones with $S > 1$, aligned on greater times. On the other hand, Figures 1b and 1c prove that separate linear models can effectively fit the measurements. Our interpretation of this phenomenon is that the change in stride affects GPGPUs' memory access patterns, causing a degradation in performance.

Table II reports the linear regression coefficients by category, both for the forward pass and back propagation. Following the notation established in equation (4), this table shows the estimated intercepts and slopes. Convolutional and fully connected layers achieve the best marginal efficiency, followed by normalization and, on a similar level, all the other categories. Our statement is motivated by the lower slopes needed to fit the data: this means that, when the complexity increases by a fixed amount, the convolution/fully connected category suffers a relatively smaller impact in comparison to other layer kinds. Such effect corroborates the observations previously discussed with respect to the operations and time breakdowns.

[1] http://www.image-net.org/challenges/LSVRC/2012/

Table II
LINEAR REGRESSION MODELS, NVIDIA QUADRO M6000

Category	$\hat{\beta}_0^{\text{fw}}$ [ms]	$\hat{\beta}_1^{\text{fw}}$ $\left[\frac{\text{ms}}{\text{op}}\right]$	$\hat{\beta}_0^{\text{bw}}$ [ms]	$\hat{\beta}_1^{\text{bw}}$ $\left[\frac{\text{ms}}{\text{op}}\right]$
Conv/FC	$1.83\text{E}{-1}$	$3.43\text{E}{-10}$	$3.15\text{E}{-1}$	$3.65\text{E}{-10}$
Norm	$1.64\text{E}{-2}$	$7.11\text{E}{-9}$	$1.01\text{E}{-1}$	$6.87\text{E}{-9}$
Pool $S = 1$	$2.23\text{E}{-2}$	$1.27\text{E}{-8}$	$1.52\text{E}{-1}$	$1.93\text{E}{-8}$
Pool $S > 1$	$1.44\text{E}{-2}$	$1.45\text{E}{-8}$	$6.11\text{E}{-3}$	$5.67\text{E}{-8}$
ReLU/Drop	$8.91\text{E}{-3}$	$1.17\text{E}{-8}$	$1.18\text{E}{-2}$	$1.33\text{E}{-8}$

Table III
PER LAYER MODEL VALIDATION, NVIDIA QUADRO M6000

Network	Batch	Iterations	T [ms]	$\varepsilon_{\text{r}}^{1}$ [%]	$\varepsilon_{\text{r}}^{\text{GN}}$ [%]
AlexNet	16	100	3427	−0.71	−21.30
AlexNet	16	150	4880	4.60	−17.10
AlexNet	16	230	7488	4.52	−17.16
AlexNet	32	100	4914	1.20	0.68
AlexNet	32	150	6987	6.77	6.22
AlexNet	32	230	10719	6.72	6.16
AlexNet	64	100	8209	−1.16	15.10
AlexNet	64	150	11555	5.34	22.65
AlexNet	64	230	17790	4.91	22.16
GoogLeNet	16	100	7666	2.93	2.93
GoogLeNet	16	150	10823	9.36	9.36
GoogLeNet	16	230	16578	9.47	9.47
GoogLeNet	32	100	13201	−7.19	−7.19
GoogLeNet	32	150	18874	−2.63	−2.63
GoogLeNet	32	230	28542	−1.27	−1.27
GoogLeNet	64	100	24299	−13.68	−13.68
GoogLeNet	64	150	34088	−7.70	−7.70
GoogLeNet	64	230	52468	−8.05	−8.05
VGG-16	8	100	18440	−2.98	−15.96
VGG-16	8	150	26157	2.60	−11.13
VGG-16	8	230	39954	2.99	−10.79
VGG-16	16	100	33283	−2.07	−9.41
VGG-16	16	150	46621	4.87	−2.99
VGG-16	16	230	72084	4.00	−3.79
VGG-16	32	100	62875	−1.40	−5.43
VGG-16	32	150	88732	4.81	0.52
VGG-16	32	230	136685	4.32	0.06

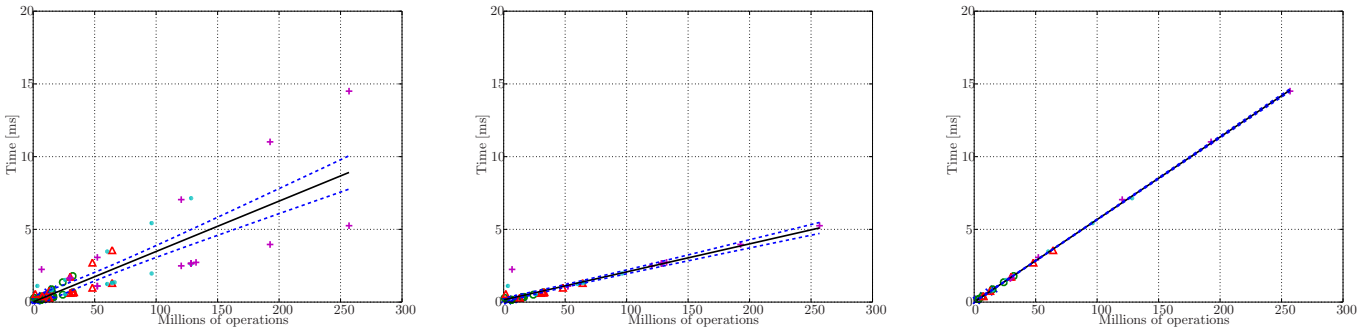
B. Per Layer Model Validation

In this section the accuracy which can be achiaved by our proposed per layer modeling approach is discussed. As accuracy metric we consider signed relative errors:

$$\varepsilon_{\text{r}} = \frac{\hat{t} - T}{T}, \tag{5}$$

where T denotes the measured time and $\hat{t}$ is the predicted time, so that negative values highlight too conservative predictions, which can be helpful if these models are to be used to enforce a deadline. Both the measured times T and the predictions $\hat{t}$ refer to the total time taken for CNN training.

Table III lists the achieved accuracy, alongside CNN, batch size, number of iterations, measured time T. For comparison, we report both the relative errors obtained by predicting with models learned on same CNN timing runs, in column $\varepsilon_{\text{r}}^{1}$, and the ones gotten when using the models for GoogLeNet across



(a) All strides (b) $S = 1$ (c) $S > 1$

Figure 1. GoogLeNet pooling layers, backward pass

all the considered networks, in column $\varepsilon_{\mathrm{r}}^{\mathrm{GN}}$. In this way we underline the generalization capabilities of this approach.

Following a common practice, the batch size varied according to a geometric progression of ratio 2, ranging from 16 to 64, except for VGG-16, which could not run at batch size 64 due to memory constraints. As far as accuracy is concerned, when using GoogLeNet's models errors in most cases remain below 20% and the average is 9.29%. When considering CNN-specific models, instead, errors generally remain below 10% and the average settles at 4.75%.

IV. Related Work

DL popularity is steadily increasing thanks to its impact on many application domains (ranging from image and voice recognition to text processing) and has received a lot of interest from many academic and industry groups. Advances are boosted by enhancements of the deep networks structure and learning process (e.g., dropout [9], network in network [10], scale jittering [11]) and by the availability of GPUs, which allows to gain up to 40x improvement over CPU systems. The work in [5] provides a comparative study of several frameworks, namely, Caffe, Neon, Theano, and Torch, by analyzing their extensibility and performance and considering both CPUs and GPUs.

The authors in [6] present solutions to minimize the total training time of CNNs, given the network architecture of the DL model, the target dataset, and the available computational resources. The authors of [12] profile and build models for a range of applications, run either on CPUs or GPUs. They exploit principal component analysis and regression to investigate performance on heterogeneous processors. Along the same lines, the authors of [13] describe Qilin, a technique for adaptively mapping computation onto CPUs or GPUs, depending on application as well as system characteristics.

The contribution closest to this work can be found in [14], where the authors focus on the deployment of CNNs on mobile devices. According to this consideration, they focus on the forward pass and predict the overall execution time based on an estimate of the time taken for matrix multiplications. This approach entails extracting tensor sizes from network specifications and associating them to their expected response times, according to platform-specific benchmarks.

V. Conclusion

In this paper we proposed a modeling approach to predict the performance of CNNs. Our models enable prediction with less than 10% on average and 23% worst case relative error even when applied to networks never seen during training. On top of their generalization capability, these models also provide insights into the performance characteristics of CNNs.

Acknowledgment

Eugenio Gianniti and Danilo Ardagna's work has been partially funded by the ATMOSPHERE project under the European Horizon 2020 grant agreement 777154.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *NIPS*, 2012.

[2] T. N. Sainath, B. Kingsbury, G. Saon, H. Soltau, A. Mohamed, G. E. Dahl, and B. Ramabhadran, "Deep convolutional neural networks for large-scale speech tasks," *Neural Networks*, vol. 64, 2015.

[3] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, vol. abs/1409.0473, 2014.

[4] Y. Wang, L. Zhang, Y. Ren, and W. Zhang, "Nexus: Bringing efficient and scalable training to deep learning frameworks," in *MASCOTS*, 2017.

[5] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative study of Caffe, Neon, Theano, and Torch for deep learning," *CoRR*, vol. abs/1511.06435, 2015.

[6] S. Hadjis, C. Zhang, I. Mitliagkas, and C. Ré, "Omnivore: An optimizer for multi-device deep learning on CPUs and GPUs," *CoRR*, vol. abs/1606.04487, 2016.

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *CVPR*. IEEE, 2015.

[8] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *ICLR*, 2015.

[9] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *JMLR*, vol. 15, 2014.

[10] M. Lin, Q. Chen, and S. Yan, "Network in network," *CoRR*, vol. abs/1312.4400, 2013.

[11] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *JMLR*, vol. 11, 2010.

[12] A. Kerr, G. Diamos, and S. Yalamanchili, "Modeling GPU-CPU workloads and systems," in *GPGPU-3*. ACM, 2010.

[13] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *MICRO*. ACM, 2009.

[14] Z. Lu, S. Rallapalli, K. Chan, and T. La Porta, "Modeling the resource requirements of convolutional neural networks on mobile devices," in *MM*. ACM, 2017.