A Black-Box Graph Partitioner for Generalized Deep Neural Network Parallelization

Jaume Mateu Cuadrat[0000−0003−0560−4180], Daon Park[0000−0003−2312−3049], and Bernhard Egger ✉[0000−0002−6645−6161]

Seoul National University, Republic of Korea
{jaume,daon,bernhard}@csap.snu.ac.kr

Abstract. In the quest for higher accuracy, large deep neural networks (DNNs) have grown significantly over the past few years. Training and executing large networks with trillions of parameters requires high-end hardware that is expensive to own or rent. A more economical alternative is to distribute the workload to several less powerful but cheaper machines. To devise an efficient workload division, existing parallelization strategies require users to posses intricate knowledge of the model, available hardware, and algorithms. In this paper, we present BBGraP, a device- and model-agnostic black-box partitioner that computes efficient parallelization plans for deep learning inference. For a given network and hardware configuration, BBGraP generates a data-parallel execution plan for each machine. The initial workload partition is optimized by eliminating redundant operations, data transfers, and synchronization points. As a proof-of-concept, BBGraP is applied to a set comprising three distributed nodes and achieves a 30% reduced latency compared to a single node.

Keywords: Deep Learning · Inference Parallelization · Resource Scheduling

1 Introduction

Deep neural networks (DNNs) are now ubiquitously adopted in many domains [1, 6, 13]. Recent large DNNs, such as the autoregressive language model GPT-3, achieve impressive performance, however, providing the necessary hardware resources to train and run such large models with trillions of parameters has become a challenge [4].

An established alternative to owning hardware is cloud computing. Cloud computing allows businesses to add and remove resources on demand [3]; however, the high-end servers required to train and execute large DNNs are expensive to rent [2]. A more economical alternative is to distribute the workload to several, but less powerful machines that cost only a fraction of high-end servers.

Several techniques distribute a large DNN to a multitude nodes, such as data parallelism [5], model parallelism [9], and intra-layer parallelism [11]. Finding the optimal division is a complex task as the best division depends on the hardware configuration, the number of nodes, and the network [10]. Data parallel workload divisions can split the work along different dimensions, further enlarging the number of possible divisions.

Existing parallelization methods require intricate knowledge of the network and the available hardware to create an efficient work division. In this work, we present BBGraP, a model- and device-agnostic technique to partition and optimize a given DNN

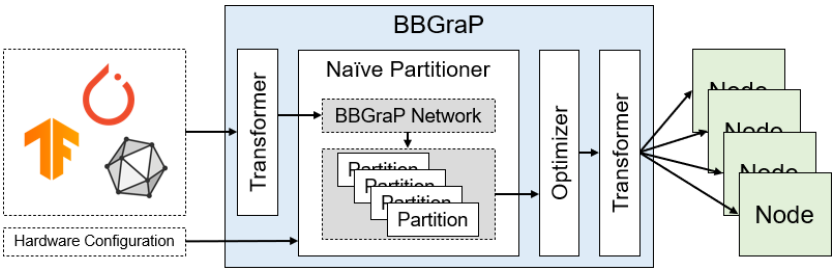


Fig. 1: The BBGraP framework.

workload for a number of heterogeneous nodes. BBGraP distributes a DNN to a variable number of heterogeneous nodes. It first produces an initial distribution and adds the necessary data transfer and synchronization operations. In a second step, a graph optimizer removes unnecessary operators, removes or tailors data transfers, and eliminates synchronization operations that are not required anymore. Initial results show that the parallel workload achieves a 30% lower latency on a three-node configuration compared to a single, more powerful node.

The remainder of this paper is organized as follows. Section 2 discusses the background and related work. Section 3 describes the operation and components of BBGraP. In Section 4, we present preliminary results of workload divisions on up to three homogeneous nodes. Section 5, finally, concludes this paper and discusses future work.

2 Background and Related Work

Distributed deep learning techniques employ data parallelism, model parallelism, and intra-layer parallelism. Data parallelism distributes the data to all workers at the expense of sending the full model to all workers and thus increasing memory requirements. Model parallelism distributes the model layer-by-layer which may cause bottlenecks between the worker nodes. Intra-layer parallelism divides operators between the workers which in turn requires synchronization to maintain operator semantics.

Different types of data partitions and workload distributions have been explored in the past. MoDNN [7] divides the network and sends the data via Wi-Fi to different devices. The network is partitioned layer-by-layer, hence, data synchronization is required at the end of each layer. DeepThings [14] focuses on early layers of the network where size of activations exceeds that of the weights. Layers are divided in the height- and width-dimension. DeepThings is aimed at IoT devices and lacks flexibility when it comes to the shapes of the partitions. DeeperThings [12] focuses on fully-connected layers and layers where the size of the weights is significantly larger than that of input activations. This approach is limited to weight and output partitioning, and only supports specific partitionings. DeepThings and DeeperThings outperform MoDNN because they don't require synchronization after each layer; however, both methods lack flexibility in the type of generated partitionings.

The CSDF partitioner [8] performs both model and data partition. Similar to MoDNN, only weight and output partitionings are supported that require synchronization after each layer. CSDF focuses on throughput, so not all resources are utilized at all times.

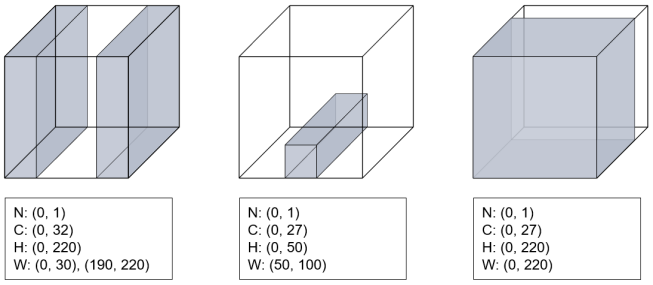


Fig. 2: ShapeMap examples (N: batches, C: channels, H: rows, W: columns).

3 BBGraP: A Black-Box DNN Model Partitioner

BBGraP takes a DNN model and a hardware configuration as inputs. After transforming the network into an internal format, an initial workload partition is created. This partition synchronizes all nodes after each layer. In a second step, BBGraP analyzes and optimizes the execution plan for each node by cropping or eliminating unnecessary data transfers and synchronization operations. Figure 1 depicts the organization of BBGraP.

3.1 Transformer

The transformation step converts an existing network into BBGraP's internal *ShapeMap* format. Figure 2 shows a few examples of ShapeMaps partitioned in various dimensions. A ShapeMap can even define intervals with multiple ranges as shown in the first example of Figure 2. The generated network hierarchy closely resembles the original network and encompasses all required parameters to partition the workload and later regenerate partitioned, stand-alone networks in the original format.

3.2 Partitioner

For the initial partitioning, BBGraP divides each layer into a an evenly distributed workload across all available nodes. The initial partitioning is computed from the last layer to the first using the final output *ShapeMap*. This direction makes it easier to compute the overlaps that occur due to different stride, padding, and dilation parameters. As shown in Equation 1, the output is equally divided by the number of nodes. Table 1 defines the variables used in the equations. The remainder of the integer division O/N_d is distributed to the nodes in a round-robin manner. For example, an 8×8 output feature map partitioned to three nodes in width direction results the partitions $pO_{(w,0)} = 3 \times 8$, $pO_{(w,1)} = 3 \times 8$, and $pO_{(w,2)} = 2 \times 8$.

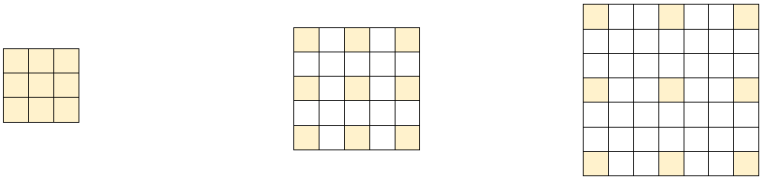
$$pO_{(d,i)} = \lfloor O/N_d \rfloor \tag{1}$$

Once the output dimensions are known, BBGraP computes the input division by taking the kernel size, dilation, stride, and padding into account. First, the true kernel size is realized with Equation 2. A kernel with a dilation $D = 1$ is the kernel itself, but for dilation values larger than 1, the kernel is spread out as illustrated in Figure 3.

$$tK_d = (K_d - 1) \cdot D_d + 1 \tag{2}$$

Symbol	Description
d	Partitioning direction (c, h, or w)
N_d	Number of nodes assigned to direction d
I / O	Original input / output size
$pI_{(d,i)}$ / $pO_{(d,i)}$	Input/output partition size, partitioned in direction d and assigned to node i
K_d	Kernel size in direction d
tK_d	True kernel size in direction d
D_d, S_d, IV_d	Dilation, stride, and interval in direction d
$P_{(d_0,d_1)}$	Padding size in direction of d_0 (h or w) and d_1 (left or right)

Table 1: Table of variables used in the equations



(a) 3×3 kernel with dilation 1. (b) 3×3 kernel with dilation 2. (c) 3×3 kernel with dilation 3.

Fig. 3: 3×3 kernels with different dilation values.

Using the true kernel size, the size of the input is given by Equation 3; Figure 4 illustrates the calculation of the input partition for an output of $3(1 \times 1)$ and a 3×3 kernel with dilation and stride of 2.

$$IV_{(d,i)} = (O - 1) \cdot S_d + tK_d \tag{3}$$

The start and end index of the input data partition for node i are given by Equations 4 and 5. $P_{(d,0)}$ denotes the padding in either width-left or height-left, and $P_{(d,1)}$ denotes the padding in either width-right or height-right direction. For example, $P_{(w,0)}$ refers to padding at the left side of the feature map, and $P_{(h,1)}$ denotes padding at the bottom of the feature map.

$$\text{Left}_{(d,i)} = \begin{cases} 0 & \text{if } i = 0, \\ S_d \cdot \sum_{j=0}^{i-1} pO_{(d,j)} - P_{(d,0)} & \text{otherwise.} \end{cases} \tag{4}$$

$$\text{Right}_{(d,i)} = \begin{cases} IV_{(d,i)} + \text{Left}_{(d,i)} - P_{(d,0)} & \text{if } i = 0, \\ IV_{(d,i)} + \text{Left}_{(d,i)} - P_{(d,1)} & \text{if } i = N_d - 1, \\ IV_{(d,i)} + \text{Left}_{(d,i)} & \text{otherwise.} \end{cases} \tag{5}$$

The input partitioned in direction d for node i is given by Equation 6.

$$pI_{(d,i)} = [\text{Left}_{(d,i)}, \text{Right}_{(d,i)}] \tag{6}$$

Once all partitions have been created, BBGraP inserts concatenation and crop operators after each layer to join the outputs and prepare the inputs for the next layer.

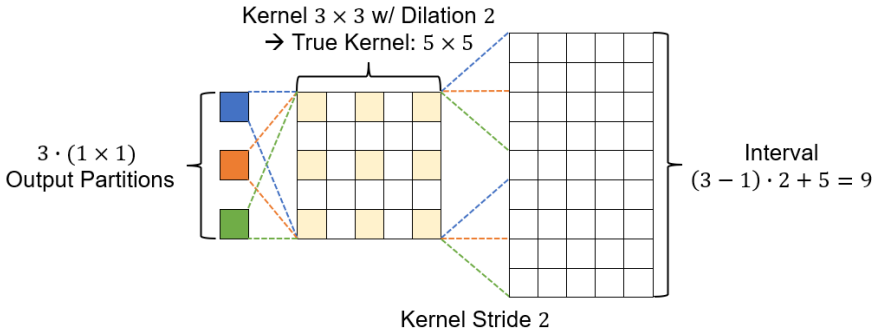


Fig. 4: Example of calculating input interval size.

3.3 Graph Optimizer

The initial partitioning creates a full copy of the output data at each node. This introduces many unnecessary data transfer, synchronization, and crop operators, depending on which parts of the previous layer's output are required as inputs to the following layer. The graph optimizer analyzes the available output and required input data for each node and removes unnecessary operators. As illustrated in Figure 5, there are four distinct cases (b)-(e). Figure 5 (a) shows the initial partitioning with output α of layer $l-1$, the full activations β and the required input γ for layer l.

1. **Input bigger than output**, Figure 5(b): the missing data is fetched from device node 1 and concatenated on device node 2.
2. **Input identical to output**, Figure 5(c): no data dependencies exist, and crop and concatenation operators are deleted.
3. **Input smaller than output**, Figure 5(d-e): if the generated output is larger than the required input on the same node, in most cases, no data dependencies exist and a crop operator is used to reduce the input data to the expected size. It is, however, possible that a data dependency still exists; in that case, the optimizer adjusts the data transfer and crop operator to make sure the necessary data is fetched before the computation continues on device node 2.

4 Evaluation

4.1 Graph Partitioning and Optimization

Fig. 6 illustrates the operation of BBGraP on a small toy network. Figure 6(a) shows the initial network comprising two convolutional layers. The initial partition to three nodes with data transfer operations (crop and concatenate) is shown in Figure 6(b); data dependencies are shown in red. The graph optimizer is able to remove unnecessary data transfer and synchronizations for nodes 1 and 3 since the size of the first convolution's output is identical to the required input for the second convolution as illustrated by Figure 6(c).

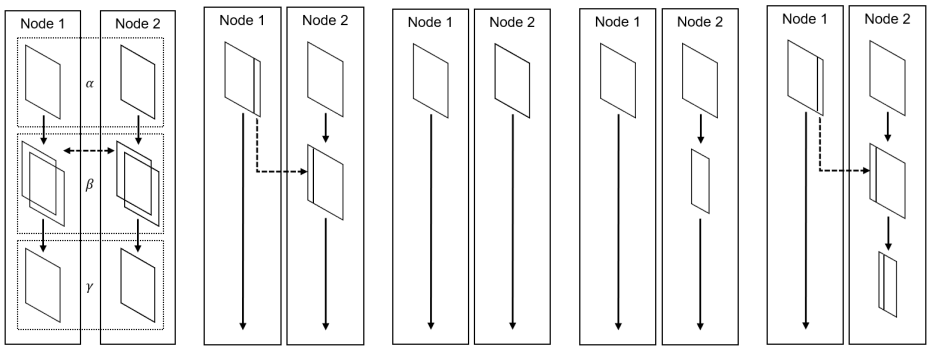


Fig. 5: Graph optimizations: 5a shows the initial state prior to optimization. 5b−5e illustrate the optimization process in dependence of the dimensions of the output and next layer's input.

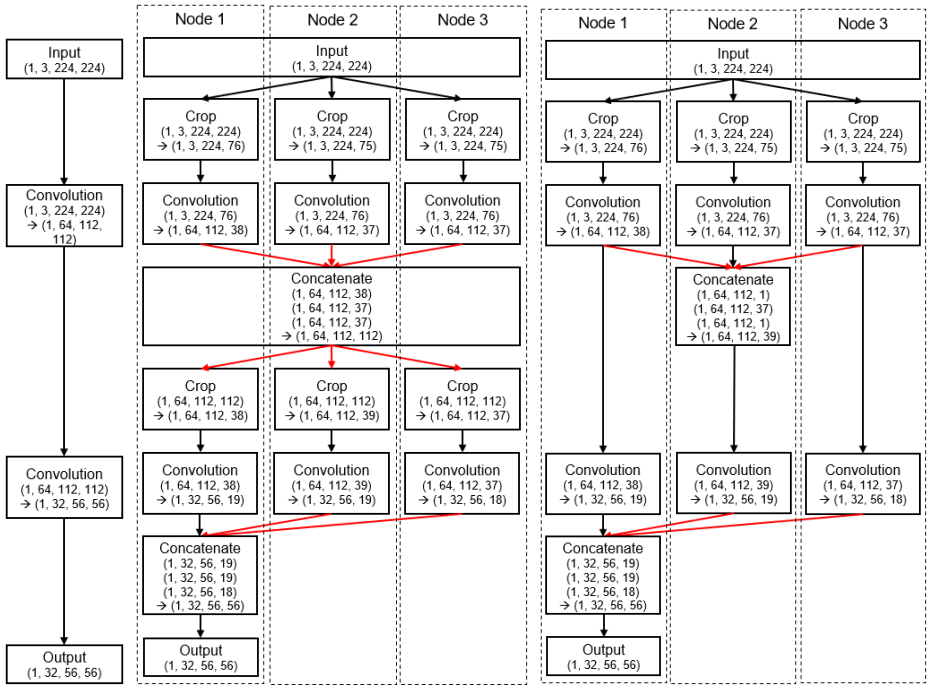


Fig. 6: Original, initial, and optimized graph. Red arrows denote data dependencies/synchronization points between the nodes.

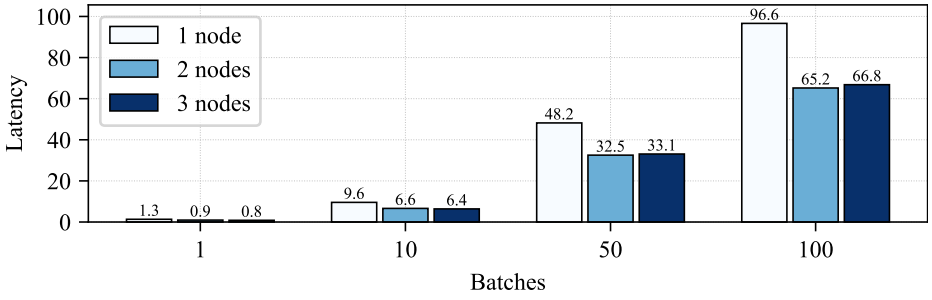


Fig. 7: Latency of ResNet50 inference on 1-3 nodes.

4.2 Latency of Parallel Inference

For a preliminary evaluation of BBGraP, we compare the performance of single-node inference with parallelized workloads to two and three homogeneous nodes. The parallelized network is ResNet50; each node is equipped with an Intel i5-10400 6-core/12-thread processor and sufficient memory. The network operators are executed using Intel's oneDNN library; data transfers and synchronizations orchestrated by a small custom runtime. Due to limitations of the runtime, BBGraP was artificially limited to output channel division.

Figure 7 plots the interference latency of ResNet50 for a batch size of 1, 10, 50, and 100 on one to three nodes. We observe that using two nodes reduces the latency significantly. With three nodes, however, latency does not improve anymore. The culprit is the non-scalable output channel division that requires full synchronization of all output data to all nodes after each convolution. We even observe a small increase in latency for larger batch sizes; this is because our communication library does not use broadcast but transfers the output of one node to all other nodes in a serial manner.

5 Conclusion and Future Work

This paper has presented our ongoing work on BBGraP, a device- and model-agnostic framework that computes an efficient workload division for DNN inference. While the framework still has many shortcomings, the results show the potential of the presented approach.

At the moment, BBGraP only supports homogeneous nodes and both the initial partition, the graph optimization, and the runtime to orchestrate execution on multiple nodes still offer lots of room for improvements. Our ongoing and future work on BBGraP includes support for heterogeneous nodes by providing the computational and memory resources as an input. The graph optimizer is being improved to also consider layer fusion; the large number of potential workload divisions in multiple directions and fused layers will require clever search space pruning to keep the search time reasonable. Finally, we are also working on improving BBGraP's runtime to transfer data more efficiently between nodes, particularly, for scenarios with many nodes.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback and suggestions. This work was funded, in parts, by the Korean National Research Foundation by grants 2022R1F1A1074967, 21A20151113068 (BK21 Plus for Pioneers in Innovative Computing - Dept. of Computer Science & Engineering, SNU), 10077609 (MOTIE/KEIT), and the Samsung Advanced Institute of Technology. ICT at Seoul National University provided research facilities for this study. Bernhard Egger is the corresponding author.

References

1. Ahmedt-Aristizabal, D., Armin, M.A., Denman, S., Fookes, C., Petersson, L.: Graph-based deep learning for medical diagnosis and analysis: past, present and future. Sensors **21**(14), 4758 (2021)
2. Amazon: Amazon ec2 p4 instances: Highest performance for ml training and hpc applications in the cloud (2020), https://aws.amazon.com/ec2/instance-types/p4/
3. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation computer systems **25**(6), 599–616 (2009)
4. Fedus, W., Zoph, B., Shazeer, N.: Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity (2021)
5. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems **25** (2012)
6. Lee, W., Seong, J.J., Ozlu, B., Shim, B.S., Marakhimov, A., Lee, S.: Biosignal sensors and deep learning-based speech recognition: A review. Sensors **21**(4), 1399 (2021)
7. Mao, J., Chen, X., Nixon, K.W., Krieger, C., Chen, Y.: Modnn: Local distributed mobile computing system for deep neural network. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017. pp. 1396–1401. IEEE (2017)
8. Minakova, S., Tang, E., Stefanov, T.: Combining task-and data-level parallelism for high-throughput cnn inference on embedded cpus-gpus mpsocs. In: International Conference on Embedded Computer Systems. pp. 18–35. Springer (2020)
9. Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N.R., Ganger, G.R., Gibbons, P.B., Zaharia, M.: Pipedream: generalized pipeline parallelism for dnn training. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. pp. 1–15 (2019)
10. Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., et al.: Efficient large-scale language model training on gpu clusters using megatron-lm. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–15 (2021)
11. Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., Catanzaro, B.: Megatron-lm: Training multi-billion parameter language models using model parallelism. arXiv preprint arXiv:1909.08053 (2019)
12. Stahl, R., Hoffman, A., Mueller-Gritschneder, D., Gerstlauer, A., Schlichtmann, U.: Deeperthings: fully distributed cnn inference on resource-constrained edge devices. International Journal of Parallel Programming **49**(4), 600–624 (2021)
13. Sun, F., Liu, J., Wu, J., Pei, C., Lin, X., Ou, W., Jiang, P.: Bert4rec: Sequential recommendation with bidirectional encoder representations from transformer. In: Proceedings of the 28th ACM international conference on information and knowledge management. pp. 1441–1450 (2019)
14. Zhao, Z., Barijough, K.M., Gerstlauer, A.: Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **37**(11), 2348–2359 (2018)