DIPPM: A Deep Learning Inference Performance Predictive Model Using Graph Neural Networks

Karthick Panner Selvam(✉) and Mats Brorsson

SnT, University of Luxembourg, Kirchberg, Luxembourg
{karthick.pannerselvam,mats.brorsson}@uni.lu

Abstract. Deep Learning (DL) has developed to become a corner-stone in many everyday applications that we are now relying on. However, making sure that the DL model uses the underlying hardware efficiently takes a lot of effort. Knowledge about inference characteristics can help to find the right match so that enough resources are given to the model, but not too much. We have developed a DL Inference Performance Predictive Model (DIPPM) that predicts the inference *latency*, *energy*, and *memory usage* of a given input DL model on the NVIDIA A100 GPU. We also devised an algorithm to suggest the appropriate A100 Multi-Instance GPU profile from the output of DIPPM. We developed a methodology to convert DL models expressed in multiple frameworks to a generalized graph structure that is used in DIPPM. It means DIPPM can parse input DL models from various frameworks. Our DIPPM can be used not only helps to find suitable hardware configurations but also helps to perform rapid design-space exploration for the inference performance of a model. We constructed a graph multi-regression dataset consisting of 10,508 different DL models to train and evaluate the performance of DIPPM, and reached a resulting Mean Absolute Percentage Error (MAPE) as low as 1.9%.

Keywords: Performance Prediction · Multi Instance GPU · Deep Learning Inference

1 Introduction

Many important tasks a now relying on Deep learning models, for instance in computer vision and natural language processing domains [3,14]. In recent years, researchers have focused on improving the efficiency of deep learning models to reduce the computation cost, energy consumption and increase the throughput of them without losing their accuracy. At the same time, hardware manufacturers like NVIDIA increase their computing power. For example, the NVIDIA A100[1] GPU half-precision Tensor Core can perform matrix operations at 312 TFLOPS. But all deep learning models will not fully utilize the GPU because the workload

[1] https://www.nvidia.com/en-us/data-center/a100/.

J. Cano et al. (Eds.): Euro-Par 2023, LNCS 14100, pp. 3–16, 2023.
https://doi.org/10.1007/978-3-031-39698-4_1

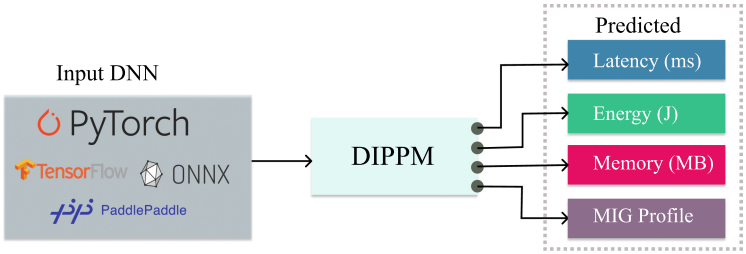


Fig. 1. DIPPM can predict the Latency, Energy, Memory requirement, and MIG Profile for inference on an NVIDIA A100 GPU without actually running on it.

and number of matrix operations will vary according to the problem domain. For this reason, NVIDIA created the Multi-Instance GPU (MIG[2]) technology starting from the Ampere architecture; they split the single physical GPU into multi-isolated GPU instances, so multiple applications can simultaneously run on different partitions of the same GPU, which then can be used more efficiently.

However, determining the DL model's efficiency on a GPU is not straightforward. If we could predict parameters such as inference latency, energy consumption, and memory usage, we would not need to measure them on deployed models which is a tedious and costly process. The predicted parameters could then also support efficient Neural Architecture Search (NAS) [5], efficient DL model design during development, and avoid job scheduling failures in data centers. According to Gao et al. [7], most failed deep learning jobs in data centers are due to out-of-memory errors.

In order to meet this need, we have developed a novel *Deep Learning Inference Performance Predictive Model* (DIPPM) to support DL model developers in matching their models to the underlying hardware for inference. As shown in Fig. 1, DIPPM takes a deep learning model expressed in any of the frameworks: PyTorch, PaddlePaddle, Tensorflow, or ONNX, and will predict the latency (ms), energy (J), memory requirement (MB), and MIG profile for inference on an Nvidia A100 GPU without running on it. At the moment, the model is restricted to inference and the Nvidia A100 architecture, but we aim to relax these restrictions in future work. As far as we are aware, this is the first predictive model that can take input from any of the mentioned frameworks and to predict all of the metrics above.

Our contributions include the following:

- We have developed, trained and evaluated a performance predictive model which predicts inference latency, energy, memory, and MIG profile for A100 GPU with high accuracy.
- We have developed a methodology to convert deep learning models from various deep learning frameworks into generalized graph structures for graph learning tasks in our performance predictive model.

[2] https://docs.nvidia.com/datacenter/tesla/mig-user-guide/.

– We have devised an algorithm to suggest the MIG profile from predicted Memory for the given input DL model.
– We have created an open-sourced performance predictive model dataset containing 10,508 graphs for graph-level multi-regression problems.

Next, we discuss our work in relation to previous work in this area before presenting our methodology, experiments, and results.

2 Related Work

Performance prediction of deep learning models on modern architecture is a rather new research field being attended to only since a couple of years back. Bouhali et al. [2] and Lu et al. [15] have carried out similar studies where a classical Multi-Layer Perceptron (MLP) is used to predict the inference latency of a given input DL model. Their approach was to collect high-level DL model features such as batch size, number of layers, and the total number of floating point operations (FLOPS) needed. They then fed these features into an MLP regressor as input to predict the latency of the given model. Bai et al. [1] used the same MLP method but predicted both the latency and memory. However, the classical MLP approach did not work very well due to the inability to capture a detailed view of the given input DL model.

To solve the above problems, some researchers came up with a kernel additive method; they predict each kernel operation, such as convolution, dense, and LSTM, individually and sum up all kernel values to predict the overall performance of the DL model [9,16,19,21,23,25]. Yu et al. [24] used the wave-scaling technique to predict the inference latency of the DL model on GPU, but this technique requires access to a GPU in order to make the prediction.

Kaufman et al. and Dudziak et al. [4,10] used graph learning instead of MLP to predict each kernel value. Still, they used the kernel additive method for inference latency prediction. However, this kernel additive method did not capture the overall network topology of the model, and instead it will affect the accuracy of the prediction. To solve the above problem, Liu et al. [13] used a Graph level task to generalize the entire DL model into node embeddings and predicted the inference latency of the given DL model. However, they did not predict other parameters, such as memory usage and energy consumption. Gao et al. [6] used the same graph-level task to predict the single iteration time and memory consumption for deep learning training but not for inference.

Li et al. [12] tried to predict the MIG profiles on A100 GPU for the DL models. However, their methodology is not straightforward; they used CUDA Multi-Process Service (MPS) values to predict the MIG, So the model must run at least on the target hardware once to predict the MIG Profile.

Most of the previous research work concentrated on parsing the input DL model from only one of the following frameworks (PyTorch, TensorFlow, PaddlePaddle, ONNX). As far as we are aware, none of the previous performance prediction models predicted Memory usage, Latency, Energy, and MIG profile simultaneously.

Table 1. Related Work comparison

Related Works	A100	MIG	GNN[a]	Multi-SF[b]	Latency	Power	Memory
Ours (**DIPPM**)	✓	✓	✓	✓	✓	✓	✓
Bai et al. [1]	–	–	–	–	✓	–	✓
Bouhali et al. [2]	–	–	–	–	✓	–	–
Dudziak et al. [4]	–	–	✓	–	✓	–	–
Gao et al. [6]	–	–	✓	–	✓	–	✓
Justus et al. [9]	–	–	–	–	✓	–	–
Kaufman et al. [10]	–	–	✓	–	✓	–	–
Li et al. [12]	✓	✓	–	–	–	–	–
Liu et al. [13]	–	–	✓	–	✓	–	–
Lu et al. [15]	–	–	–	–	✓	✓	✓
Qi et al. [16]	–	–	–	–	✓	–	–
Sponner et al. [19]	✓	–	–	–	✓	✓	✓
Wang et al. [21]	–	–	–	–	✓	–	–
Yang et al. [23]	–	–	–	–	✓	–	–
Yu et. al. [24]	✓	–	–	–	✓	–	–
Zhang et al. [25]	–	–	–	–	✓	–	–

[a] Using Graph Neural Network for performance prediction
[b] Able to parse DL model expressed in Multiple DL Software Framework

Our novel Deep Learning Inference Performance Predictive Model (DIPPM) fills a gap in previous work; a detailed comparison is shown in Table 1. DIPPM takes a deep learning model as input from various deep learning frameworks such as PyTorch, PaddlePaddle, TensorFlow, or ONNX and converts it to generalize graph with node features. We used a graph neural network and MIG predictor to predict the inference latency (ms), energy (J), memory (MB), and MIG profile for A100 GPU without actually running on it.

3 Methodology

The architecture of DIPPM consists of five main components: Deep Learning Model into Relay IR, Node Feature Generator, Static Feature Generator, Performance Model Graph Network Structure (PMGNS), and MIG Predictor, as shown in Fig. 2. We will explain each component individually in this section.

3.1 Deep Learning Model into Relay IR

The Relay Parser takes as input a DL model expressed in one of several supported DL frameworks, converts it to an Intermediate Representation (IR), and passes this IR into the Node Feature Generator and the Static Feature Generator components.

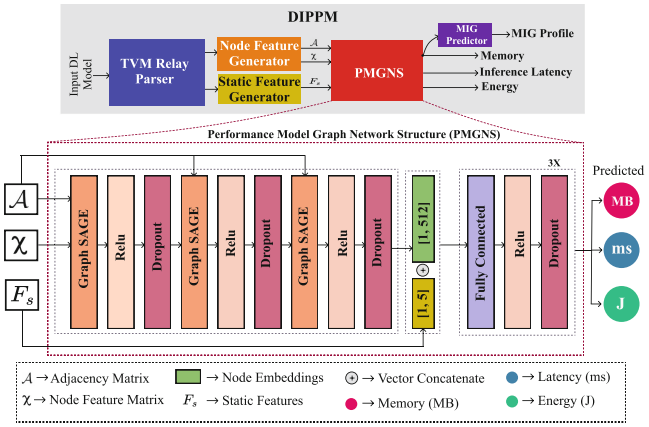


Fig. 2. Overview of DIPPM Architecture

Most of the previously proposed performance models are able to parse the given input DL model from a single DL framework, not from several, as we already discussed in Sect. 2. To enable the use of multiple frameworks, we used a relay, which is a high-level IR for DL models [17]. It has been used to compile DL models for inference in the TVM[3] framework.

We are inspired by the approach of converting DL models from different frameworks into a high-level intermediate representation (IR), so we incorporated their techniques into our architecture. However, we couldn't directly employ relay IR in DIPPM. To overcome this, we developed a method explained in Sect. 3.2. It involves parsing the Relay IR and transforming it into a graph representation with node features.

It allows parsing given input DL models from various frameworks, including PyTorch, TensorFlow, ONNX, and PaddlePaddle. However, for the purposes of this study, we have focused on the implementation and evaluation of the framework specifically within the PyTorch environment. We pass this DL IR to the subsequent components in our DIPPM architecture.

3.2 Node Feature Generator

The Node Feature Generator (NFG) converts the DL IR into an Adjacency Matrix ($\mathcal{A}$) and a Node feature matrix ($\mathcal{X}$) and passes this data to the PMGNS component.

The NFG takes the IR from the relay parser component. The IR is itself a computational data flow graph containing more information than is needed for our performance prediction. Therefore we filter and pre-process the graph by post-order graph traversal to collect necessary node information. The nodes in the IR contain useful features such as operator name, attributes, and output

[3] https://tvm.apache.org/.

Algorithm 1. Algorithm to convert DL model IR into a graph with node features

CreateGraph takes input IR and filters it by post-order traversal. Collect node features for each node and generate a new graph $\mathcal{G}$ with node features, finally extract node feature matrix $\mathcal{X}$ and adjacency matrix $\mathcal{A}$ from $\mathcal{G}$.

1: **function** CreatGraph(IR) ▷ IR from Relay Parser Component
2: $\mathcal{N} \leftarrow filter\_and\_preprocess(IR)$
3: $\mathcal{G} \leftarrow \emptyset$ ▷ Create empty directed graph
4: **for each** $node \in \mathcal{N}$ **do** ▷ where *node* is node in node_list $\mathcal{N}$
5: **if** $node.op \in$ [operators] **then** ▷ Check node is an operator
6: $\mathcal{F}_{oh} \leftarrow one\_hot\_encoder(node.op)$
7: $\mathcal{F}_{attr} \leftarrow ExtractAttributes(node)$
8: $\mathcal{F}_{shape} \leftarrow ExtractOutshape(node)$
9: $\mathcal{F}_{node} \leftarrow \mathcal{F}_{oh} \oplus \mathcal{F}_{attr} \oplus \mathcal{F}_{shape}$
10: $\mathcal{G}.add\_node(node.id, \mathcal{F}_{node})$ ▷ Nodes are added in sequence
11: **end if**
12: **end for**
13: $\mathcal{A} \leftarrow GetAdjacencyMatrix(\mathcal{G})$
14: $\mathcal{X} \leftarrow GetNodeFeatureMatrix(\mathcal{G})$
15: **return** $\mathcal{A}, \mathcal{X}$
16: **end function**

shape of the operator, which after this first filtering step are converted into a suitable data format for our performance prediction. In the subsequent step, we loop through the nodes and, for each operator node, generate node features Fnode with a fixed length of 32 as discussed on line 9 in Algorithm 1.

The central part of the NFG is to generate an **Adjacency Matrix** ($\mathcal{A}$) and a **Node feature matrix** ($\mathcal{X}$) as expressed in Algorithm 1. $\mathcal{X}$ has the shape of $[N_{op}, N_{features}]$, where N_{op} is the number of operator nodes in the IR and $N_{features}$ is the number of features. In order to create node features $\mathcal{F}_n$ for each *node*, first, we need to encode the node operator name into a one hot encoding as can be seen on line 6 in Algorithm 1. Then extract the node attributes $\mathcal{F}_{attr}$ and output shape $\mathcal{F}_{shape}$ into vectors. Finally, perform vector concatenation to generate $\mathcal{F}_n$ for a node. We repeat this operation for each node and create the $\mathcal{G}$. From the $\mathcal{G}$, we extract $\mathcal{A}$, $\mathcal{X}$ that are passed to the main part of our model, the Performance Model Graph Network Structure.

3.3 Static Feature Generator

The Static Feature Generator (SFG) takes the IR from the relay parser component and generates static features $\mathcal{F}_s$ for a given DL model and passes them into the graph network structure.

For this experiment, we limited ourselves to five static features. First, we calculate the $\mathcal{F}_{mac}$ total multiply-accumulate (MACs) of the given DL model. We used the TVM relay analysis API to calculate total MACs, but it is limited to calculating MACs for the following operators (in TVM notation): Conv2D, Conv2D transpose, dense, and batch matmul. Then we calculate the total number of

convolutions F_{Tconv}, Dense F_{Tdense}, and Relu F_{Trelu} operators from the IR. We included batch size F_{batch} as one of the static features because it gives the ability to predict values for various batch sizes of a given model. Finally, we concatenate all the features into a vector $\mathcal{F}_s$ as expressed in Eq. 1. The feature set $\mathcal{F}_s$ is subsequently passed to the following graph network structure.

$$\mathcal{F}_s \leftarrow \mathcal{F}_{mac} \oplus \mathcal{F}_{batch} \oplus \mathcal{F}_{Tconv} \oplus \mathcal{F}_{Tdense} \oplus \mathcal{F}_{Trelu} \tag{1}$$

3.4 Performance Model Graph Network Structure (PMGNS)

The PMGNS takes the node feature matrix ($\mathcal{X}$), the adjacency matrix ($\mathcal{A}$) from the Node Feature Generator component, and the feature set ($\mathcal{F}_s$) from the Static feature generator and predicts the given input DL model's memory, latency, and energy, as shown in Fig. 2.

The PMGNS must be trained before prediction, as explained in Sect. 4. The core idea of the PMGNS is to generate the node embedding z from $\mathcal{X}$ and $\mathcal{A}$ and then to perform vector concatenation of z with $\mathcal{F}_s$. Finally, we pass the concatenated vector into a Fully Connected layer for prediction, as shown in Fig. 2. In order to generate z, we used the graphSAGE algorithm suggested by Hamilton et al. [8], because of its inductive node embedding, which means it can generate embedding for unseen nodes without pretraining. GraphSAGE is a graph neural network framework that learns node embeddings in large-scale graphs. It performs inductive learning, generalizing to unseen nodes by aggregating information from nodes and neighbors. It generates fixed-size embeddings, capturing features and local graph structure. With a neighborhood aggregation scheme, it creates node embeddings sensitive to their local neighborhood, even for new, unobserved nodes.

We already discussed that we generate node features of each node in the Sect. 3.2. The graphSAGE algorithm will convert node features into a node embedding z which is more amenable for model training. The PMGNS contains three sequential graphSAGE blocks and three sequential Fully connected (FC) blocks as shown in Fig. 2. At the end of the final graphSAGE block, we get the generalized node embedding of given $\mathcal{X}$ and $\mathcal{A}$, which we concatenate with $\mathcal{F}_s$. Then we pass the concatenated vector into FC to predict the memory (MB), latency (ms), and energy (J).

3.5 MIG Predictor

The MIG predictor takes the memory prediction from PMGNS and predicts the appropriate MIG profile for a given DL model, as shown in Fig. 2.

As mentioned in the introduction, the Multi-instance GPU (MIG) technology allows to split an A100 GPU into multiple instances so that multiple applications can use the GPU simultaneously. The different instances differ in their compute capability and, most importantly, in the maximum memory limit that is allowed to be used. The four MIG profiles of the A100 GPU that we consider here are: 1g.5gb, 2g.10gb, 3g.20gb, and 7g.40gb, where the number in front of "gb"

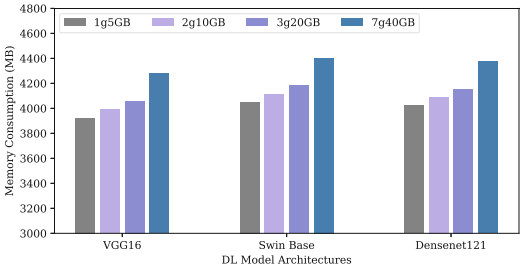


Fig. 3. MIG Profile comparison of three different DL models memory consumption on A100 GPU. We used batch size 16 for VGG16 and Densenet121 model and batch size 8 for Swin base model

denotes the maximum amount of memory in GB that the application can use on that instance. For example, the maximum memory limit of 1g.5gb is 5 GB, and 7g.40gb is 40GB. For a given input DL model, PMGNS predicts memory for 7g.40gb MIG profile, which is the full GPU. We found that this prediction can be used as a pessimistic value to guide the choice of MIG profile. Figure 3 shows manual memory consumption measurements of the same DL model inference on different profiles. The results show no significant difference in the memory allocation of DL in the different MIG profiles even though the consumption slightly increases with the capacity of the MIG profile. The memory consumption is always the highest when running on the 7g.40gb MIG profile.

As mentioned, PMGNS predicts memory for 7g.40gb, so we claim that predicted memory will be an upper bound. Then we perform a rule-based prediction to predict the MIG profile for the given input DL model, as shown in Eq. 2. Where α is predicted memory from PMGNS.

$$\text{MIG}(\alpha) = \begin{cases} \text{1g.5gb}, & \text{if } 0gb < \alpha < \text{5gb} \\ \text{2g.10gb}, & \text{if } \text{5gb} < \alpha < \text{10gb} \\ \text{3g.20gb}, & \text{if } \text{10gb} < \alpha < \text{20gb} \\ \text{7g.40gb}, & \text{if } \text{20gb} < \alpha < \text{40gb} \\ \text{None}, & \text{otherwise} \end{cases} \tag{2}$$

4 Experiments and Results

4.1 The DIPPM Dataset

We constructed a graph-level multi-regression dataset containing 10,508 DL models from different model families to train and evaluate our DIPPM. The dataset distribution is shown in Table 2. To the best of our knowledge, the previous predictive performance model dataset doesn't capture memory consumption, inference latency, and energy consumption parameters for wide-range DL models on A100 GPU so we created our own dataset for performance prediction of DL models.

Table 2. DIPPM Graph dataset distribution

Model Family	# of Graphs	Percentage (%)
Efficientnet	1729	16.45
Mnasnet	1001	9.53
Mobilenet	1591	15.14
Resnet	1152	10.96
Vgg	1536	14.62
Swin	547	5.21
Vit	520	4.95
Densenet	768	7.31
Visformer	768	7.31
Poolformer	896	8.53
Total	**10508**	100%

Our dataset consists of DL models represented in graph structure, as generated by the Relay parser described in Sect. 3.1. Each data point consists of four variables: $\mathcal{X}$, $\mathcal{A}$, $\mathcal{Y}$, and $\mathcal{F}_s$, where $\mathcal{X}$ and $\mathcal{A}$ are the Node feature matrix and Adjacency Matrix, respectively, as discussed in Sect. 3.2, and $\mathcal{F}_s$ is the static features of the DL model as discussed in Sect. 3.3. We used the Nvidia Management Library[4] and the CUDA toolkit[5] to measure the energy, memory, and inference latency of each given model in the dataset. For each model, we ran the inference five times to warm up the architecture and then the inference 30 times, and then took the arithmetic mean of those 30 values to derive the $\mathcal{Y}$, where $\mathcal{Y}$ consists of inference latency (ms), memory usage (MB), and energy (J) for a given DL on A100 GPU. We used a full A100 40 GB GPU, or it is equivalent to using 7g.40gb MIG profile to collect all the metrics.

4.2 Enviroment Setup

We used an HPC cluster at the Jülich research centre in Germany called JUWELS Booster for our experiments[6]. It is equipped with 936 nodes, each with AMD EPYC 7402 processors, 2 sockets per node, 24 cores per socket, 512 GB DDR4-3200 RAM and 4 NVIDIA A100 Tensor Core GPUs with 40 GB HBM.

The main software packages used in the experiments are: Python 3.10, CUDA 11.7 torch 1.13.1, torch-geometric 2.2.0, torch-scatter 2.1.0, and torch-sparse 0.6.16.

4.3 Evaluation

The Performance Model Graph Network Structure is the main component in DIPPM, and we used the PyTorch geometric library to create our model, as

[4] https://developer.nvidia.com/nvidia-management-library-nvml.
[5] https://developer.nvidia.com/cuda-toolkit.
[6] https://apps.fz-juelich.de/jsc/hps/juwels/booster-overview.html.

Table 3. Settings in GNN comparison.

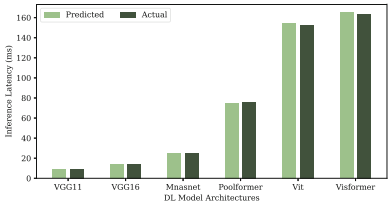
Setting	Value
Dataset partition	Train (70%) / Validation (15%) / Test (15%)
Nr hidden layers	512
Dropout probability	0.05
Optimizer	Adam
Learning rate	$2.754 \cdot 10^{-5}$
Loss function	Huber

Table 4. Comparison with different GNN algorithms and MLP with graphSAGE, we trained all the models for 10 epochs and used Mean Average Percentage Error for validation. The results indicate that DIPPM with graphSAGE performs significantly better than other variants.

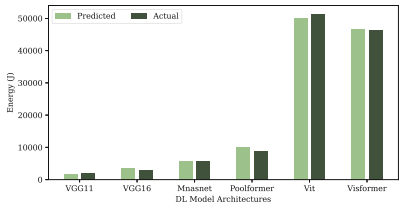
Model	Training	Validation	Test
GAT	0.497	0.379	0.367
GCN	0.212	0.178	0.175
GIN	0.488	0.394	0.382
MLP	0.371	0.387	0.366
(Ours) GraphSAGE	**0.182**	**0.159**	**0.160**

shown in Fig. 2. We split our constructed dataset into three parts randomly: training set 70%, validation set 15%, and a test set 15%.

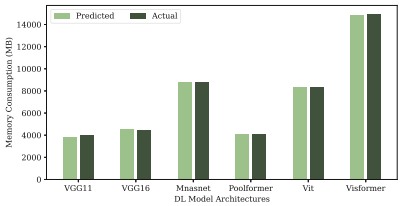
In order to validate that graphSAGE performs better than other GNN algorithms and plain MLP, we compared graphSAGE with the following other algorithms:, GAT [20], GCN [11], GIN [22], and finally, plain MLP without GNN. Table 3 summarizes the settings used. The learning rate was determined using a learning rate finder as suggested by Smith [18]. The Huber loss function achieved a higher accuracy than mean square error, which is why we chose that one. For the initial experiment, we trained for 10 epochs and used Mean Average Percentage Error (MAPE) as an accuracy metric to validate DIPPM. A MAPE value close to zero indicates good performance on regression prediction. Table 4 shows that graphSAGE gives a lower MAPE value in all of the training, validation, and test datasets. Without using a GNN, MLP gives 0.366 of MAPE. With graphSAGE, MAPE is 0.160 on the test dataset which is a significant improvement on a multi-regression problem. We conclude that graphSAGE outperforms other GNN algorithms, and MLP because of its inductive learning, as discussed in Sect. 3.4. After this encouraging result we increased the number of epochs for training our DIPPM with graphSAGE to increase the prediction accuracy. After 500 epochs, we attained MAPE of 0.041 on training and 0.023 on the validation dataset. In the end, we attained 1.9% MAPE on the test dataset. Some of the DIPPM predictions on the test dataset are shown in Fig. 4.



(a) Inference latency (ms).



(b) Energy (J).



(c) Memory consumption (MB).

Fig. 4. Comparison of actual value with DIPPM predicted values on the test dataset. Results show that DIPPM predictions are close to the actual predictions.

4.4 Prediction of MIG Profiles

In order to verify the MIG profile prediction for a given DL model, we compared the actual MIG profile value with the predicted MIG profile from the DIPPM, as shown in Table 5. To calculate the actual suitable MIG profile, we divide actual memory consumption by the maximum memory limit of the MIG profiles. The higher the value is, the more appropriate profile for the given DL model. For example, the predicted memory consumption for densenet121 at batch size 8 is 2865 MB. The actual memory consumption for the 7g.40gb MIG profile is 3272 MB. The actual memory consumption of 1g.5GB is 2918 MB, the percentage is 58%. Which is higher than other MIG profiles. Results show that DIPPM correctly predicted the MIG profile 1g.5gb for densenet121. It is interesting to note that the densent121 models are from our test dataset and the swin base patch4 model is not in our DIPPM dataset but a similar swin base model family was used to train DIPPM. The convnext models are completely unseen to our DIPPM, but it's still predicting the MIG profile correctly.

4.5 DIPPM Usability Aspects

DIPPM takes basic parameters like frameworks, model path, batch, and input size, and finally, device type. As of now, we only considered A100 GPU; we are working to extend DIPPM to various hardware platforms. With a simple python API call, DIPPM predicts memory, latency, energy, and MIG profile for the given model, as can be seen in Fig. 5.

Table 5. DIPPM MIG profile prediction for seen and unseen DL model architectures. (densenet*: seen, swin*: partially seen, convnext*: unseen).

Model	Batch size	Predicted		Actual				
		MIG	Mem	Mem	1g.5gb	2g.10gb	3g.20gb	7g.40gb
densenet121	8	1g.5gb	2865	3272	**58%**	30%	15%	8%
densenet121	32	2g.10gb	5952	6294		**60%**	30%	16%
swin_base_patch4	2	1g.5gb	2873	2944	**52%**	27%	14%	7%
swin_base_patch4	16	2g.10gb	6736	6156		**59%**	30%	15%
convnext_base	4	1g.5gb	4771	1652	**61%**	31%	16%	8%
convnext_base	128	7g.40gb	26439	30996				**77%**

```
import dippm
import torchvision

model = (torchvision.models.vgg16(pretrained=True)).eval()

predicted = dippm.predict(model, batch=8, input="3,244,244", device="A100")
print("Memory {0} MB, Energy {1} J, Latency {2} ms, MIG{3}".format(*predicted))
```

Fig. 5. An example code demonstrating the utilization of DIPPM for performance prediction of a VGG16 deep learning model with a batch size of 8.

5 Conclusion

We have developed a novel Deep Learning (DL) Inference Performance Predictive Model (DIPPM) to predict the inference latency, energy, and memory consumption of a given input DL model on an A100 GPU without running on it. Furthermore, We devised an algorithm to select the appropriate MIG profile from the memory consumption predicted by DIPPM. The model includes a methodology to convert the DL model represented in various frameworks to a generalized graph structure for performance prediction. To the best of our knowledge, DIPPM can help to develop an efficient DL model to utilize the underlying GPU effectively. Furthermore, we constructed and open-sourced[7] a multi-regression graph dataset containing 10,508 DL models for performance prediction. It can even be used to evaluate other graph-based multi-regression GNN algorithms. Finally, we achieved 1.9% MAPE on our dataset.

Acknowledgment. This work has been done in the context of the MAELSTROM project, which has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955513. The JU receives support from the European Union's Horizon 2020 research and innovation program and United Kingdom, Germany, Italy, Switzerland, Norway, and in Luxembourg by the Luxembourg National Research Fund (FNR) under contract number 15092355.

[7] https://github.com/karthickai/dippm.

References

1. Bai, L., Ji, W., Li, Q., Yao, X., Xin, W., Zhu, W.: Dnnabacus: toward accurate computational cost prediction for deep neural networks (2022)
2. Bouhali, N., Ouarnoughi, H., Niar, S., El Cadi, A.A.: Execution time modeling for CNN inference on embedded GPUs. In: Proceedings of the 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings, DroneSE and RAPIDO 2021, pp. 59–65. Association for Computing Machinery, New York, NY, USA (2021)
3. Brown, T.B., et al.: Language models are few-shot learners. In: Proceedings of the 34th International Conference on Neural Information Processing Systems, NIPS 2020, Curran Associates Inc., Red Hook, NY, USA (2020)
4. Dudziak, L., Chau, T., Abdelfattah, M.S., Lee, R., Kim, H., Lane, N.D.: BRP-NAS: prediction-based NAS using GCNs. In: Proceedings of the 34th International Conference on Neural Information Processing Systems, NIPS 2020, Curran Associates Inc., Red Hook, NY, USA (2020)
5. Elsken, T., Metzen, J.H., Hutter, F.: Neural architecture search: a survey. J. Mach. Learn. Res. **20**(1), 1997–2017 (2021)
6. Gao, Y., Gu, X., Zhang, H., Lin, H., Yang, M.: Runtime performance prediction for deep learning models with graph neural network. In: Proceedings of the 45th International Conference on Software Engineering, Software Engineering in Practice (SEIP) Track, ICSE 2023. IEEE/ACM (2023)
7. Gao, Y., et al.: Estimating GPU memory consumption of deep learning models. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1342–1352. ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA (2020)
8. Hamilton, W.L., Ying, R., Leskovec, J.: Inductive representation learning on large graphs. In: Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS 2017, pp. 1025–1035. Curran Associates Inc., Red Hook, NY, USA (2017)
9. Justus, D., Brennan, J., Bonner, S., McGough, A.: Predicting the computational cost of deep learning models. In: 2018 IEEE International Conference on Big Data (Big Data), pp. 3873–3882. IEEE Computer Society, Los Alamitos, CA, USA (2018)
10. Kaufman, S., et al.: A learned performance model for tensor processing units. In: Smola, A., Dimakis, A., Stoica, I. (eds.) Proceedings of Machine Learning and Systems, vol. 3, pp. 387–400 (2021)
11. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: International Conference on Learning Representations (ICLR) (2017)
12. Li, B., Patel, T., Samsi, S., Gadepally, V., Tiwari, D.: Miso: exploiting multi-instance GPU capability on multi-tenant GPU clusters. In: Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022, pp. 173–189. Association for Computing Machinery, New York, NY, USA (2022)
13. Liu, L., Shen, M., Gong, R., Yu, F., Yang, H.: Nnlqp: a multi-platform neural network latency query and prediction system with an evolving database. In: Proceedings of the 51st International Conference on Parallel Processing, ICPP 2022. Association for Computing Machinery, New York, NY, USA (2023)
14. Liu, Z., et al.: Swin transformer: hierarchical vision transformer using shifted windows. In: 2021 IEEE/CVF International Conference on Computer Vision (ICCV), pp. 9992–10002. IEEE Computer Society, Los Alamitos, CA, USA (2021)

15. Lu, Z., Rallapalli, S., Chan, K., Pu, S., Porta, T.L.: Augur: modeling the resource requirements of convnets on mobile devices. IEEE Trans. Mob. Comput. **20**(2), 352–365 (2021)
16. Qi, H., Sparks, E.R., Talwalkar, A.: Paleo: a performance model for deep neural networks. In: 5th International Conference on Learning Representations, Conference Track Proceedings, ICLR 2017, Toulon, France, 24–26 April 2017. OpenReview.net (2017)
17. Roesch, J., et al.: Relay: a new IR for machine learning frameworks. In: Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL 2018, pp. 58–68. Association for Computing Machinery, New York, NY, USA (2018)
18. Smith, L.N.: Cyclical learning rates for training neural networks. In: 2017 IEEE Winter Conference on Applications of Computer Vision (WACV), pp. 464–472 (2017)
19. Sponner, M., Waschneck, B., Kumar, A.: Ai-driven performance modeling for AI inference workloads. Electronics **11**(15) (2022)
20. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., Bengio, Y.: Graph attention networks. In: International Conference on Learning Representations (2018). Accepted as poster
21. Wang, C.C., Liao, Y.C., Kao, M.C., Liang, W.Y., Hung, S.H.: Toward accurate platform-aware performance modeling for deep neural networks. SIGAPP Appl. Comput. Rev. **21**(1), 50–61 (2021)
22. Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How powerful are graph neural networks? In: International Conference on Learning Representations (2019)
23. Yang, C., Li, Z., Ruan, C., Xu, G., Li, C., Chen, R., Yan, F.: PerfEstimator: a generic and extensible performance estimator for data parallel DNN training. In: 2021 IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence), pp. 13–18 (2021)
24. Yu, G.X., Gao, Y., Golikov, P., Pekhimenko, G.: Habitat: a runtime-based computational performance predictor for deep neural network training. In: Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC 2021) (2021)
25. Zhang, L.L., et al.: Nn-meter: towards accurate latency prediction of deep-learning model inference on diverse edge devices. In: Proceedings of the 19th Annual International Conference on Mobile Systems, Applications and Services, MobiSys 2021, pp. 81–93. Association for Computing Machinery, New York, NY, USA (2021)