Lilou: Resource-Aware Model-Driven Latency Prediction For GPU-Accelerated Model Serving

Qianlin Liang
qliang@cs.umass.edu
University of Massachusetts
Amherst, MA, USA

Haoliang Wang
hawang@adobe.com
Adobe
San Jose, CA, USA

Prashant Shenoy
shenoy@cs.umass.edu
University of Massachusetts
Amherst, MA, USA

ABSTRACT

As deep learning has been widely used in various application domains, a diversity of GPUs are adopted to accelerate DNN inference workloads and ensure Quality of Service (QoS). Robust prediction of inference latency using GPUs within cloud environments facilitates enhanced efficiency and maintains QoS in resource management solutions, such as consolidation and autoscaling. However, latency prediction is challenging due to the vast heterogeneity in both DNN architectures and GPU capacities.

In this work, we present Lilou, an efficient and accurate latency predicting system for a wide range of DNN inference tasks across diverse GPU resource allocations. Lilou employs two techniques. (i) Lilou represents DNNs as directed acyclic graphs (DAGs), and utilizes a novel graph neural network (GNN) model for edge classification to detect the fusion of operators, also known as kernels. (ii) Lilou identifies the GPU features that significantly impact inference latency and learns a predictor to estimate the latency and type of kernels, which are detected in the preceding step. To evaluate Lilou, we conduct comprehensive experiments across a variety of commercial GPUs commonly utilized in public cloud environments, employing a wide range of popular DNN architectures, including both convolutional neural networks and transformers. Our experiment results show that Lilou is robust to a wide range of DNN architectures and GPU resource allocations. Our novel learning-based method surpasses the state-of-the-art rule-based approach in fusion prediction with an accuracy of 97.35%, laying a solid foundation for end-to-end latency prediction that achieves a MAPE of 8.68%, also outperforming existing benchmarks.

KEYWORDS

GPU, Performance, Neural Network, Latency

1 INTRODUCTION

In recent years, deep neural networks (DNNs) have undergone rapid development and have become a fundamental building block for a broad spectrum of AI applications such as autonomous vehicles, video analytics, and recommendation systems. It is increasingly common to use cloud resources to support these applications. As a cloud workload, model serving involves hosting pre-trained models on GPU or CPU resources and providing inference services over the web. Examples of such web-based inference services include image recognition services (e.g., Google Cloud Vision API), large language model (LLM) services (e.g., ChatGPT), and recommendation systems embedded in e-commerce platforms. As AI applications continue to grow, inference has become an increasingly popular cloud workload, and these services are often deployed as web services to enable broad accessibility and scalability. DNN models vary significantly in their size, complexity, and computational requirements. As these services scale, it becomes increasingly important to select the appropriate GPU hardware and resources to serve these models with high performance and cost efficiency.

DNN inference applications often have strict Service Level Objectives (SLO) needs in terms of their latency requirements. At the same time, GPU resources in edge or cloud platforms can be expensive, making cost of model serving an important consideration. For example, Amazon Web Services (AWS)[1] offers GPUs with varied costs ranging from \$0.526 per hour to \$32.77 per hour. Given the diversity of DNN models and their computational complexity, it has become increasingly challenging for a model serving platform to choose the right GPU resources to run each DNN model. An incorrect choice can have cost or performance implications that directly affect the quality of inference services. For example, choosing a low-end GPU for a complex DNN model may not provide sufficient computational resources to meet the desired latency SLO, resulting in poor performance and customer dissatisfaction. Conversely, selecting a high-end GPU for a less complex DNN model may lead to resource underutilization and high cloud costs. Further, many model serving platforms multiplex a single GPU across multiple DNN services to improve utilization and amortize costs. For example, Nvidia GPUs support GPU virtualization through its Multi-Instance GPU (MIG)[30] feature and support fine-grained GPU resource provisioning through its Multi-Process Service (MPS)[31] feature. Such advanced features improve resource utilization and reduce costs, but make the GPU resource provisioning problem more challenging.

In order for a model serving platform to choose the appropriate GPU hardware for a DNN inference workload, it is crucial to estimate the expected latency of executing that DNN model on different GPU configurations. One well-known approach for estimating latency is through empirical profiling, which involves running the DNN model on the target hardware to measure the execution time. However, profiling is a time-consuming process since it can involve executing the model on dozens of available GPU configurations in order to choose the best one. Further, as the number of model and device variants increase (e.g. Neural Architecture Search (NAS)[10, 25] can produce hundreds of model variants for each application), the overhead of exhaustive profiling can quickly accumulate and become impractical in larger settings.

An alternative to empirical profiling is estimating DNN inference latency through modeling. Numerous recent efforts have developed model-driven approaches that use analytic methods or a separate machine learning model to predict the DNN inference latency. One class of approaches focuses on *end-to-end* prediction by modeling the entire DNN and use approaches ranging from linear regression to graph neural network (GNN)[6, 11, 14, 22, 26] to predict the inference latency on a specific GPU hardware. Such approaches require training for each type of GPU and do not generalize easily for unseen hardware or model variants. For example, graph regression methods rely heavily on the graph patterns learned from the training data and often fail to generalize to unseen DNNs. Consequently, other approaches have focused on modeling the internal structure of the DNNs to estimate latency. For example, *layer-regression*[5, 19, 28, 34] approaches predict the latency to execute each layer and then estimate the total latency on the sum of the layer-specific latencies. Since components such as layers are often reused across models, the approach has the potential to generalize across model variants. However, a significant limitation of layer-based approaches is their inability to account for runtime optimization such as layer fusion, which involves combing adjacent layers into a single layer or operation for performance optimization and is common in runtime frameworks for improving performance. Layer-based approaches end up *overestimate* total latency since they focus on individual layers and do not account for latency reduction from fusing layers. To overcome this drawback, kernel-based approaches have been developed[23, 43] , where latency is estimated at kernel, rather than layer granularity. Since a kernel can include one or more layers, including fused layer it can improve the accuracy of the latency estimations. The recent state-of-the-art nn-Meter[43] approach partitions the model into multiple kernels by using handcrafted fusion rules to determine which layer might be fused at runtime, followed by building latency regressors for each kernel to estimate total latency. However, handcrafting fusion rules can be time-consuming and error-prone, and they may need to be changed frequently due to rapid advances in deep learning frameworks. Thus, existing methods suffer from many limitations ranging from inability to handle runtime optimization or inability to generalize to newer models or hardware.

Research contributions. In this paper, we present Lilou, a graph model-based approach for predicting the latency of DNN inference on both dedicated GPUs and arbitrarily provisioned GPU resources. Lilou models the DNN structure as a directed acyclic graph and uses the graph structure itself to automatically infer fusion rules and fusion operations for the DNN graph. It then partitions the overall DNN graph structure into fusion-aware sub-graphs such that layers with the same sub-graph can be fused and executed by a single kernel or operation. Lilou then predicts the latency and kernel type for each sub-graph in a resource-aware manner. Importantly, Lilou adopts GNNs for its sub-graph extraction and sub-graph based latency prediction. We make these design decisions based on the following key observations. 1) Given a runtime platform, the fusion pattern of DNN layers tends to remain relatively stable and consistent across different models. Although GNNs may struggle to generalize to globally unseen graph structures (i.e. graph-level prediction), they remain effective in capturing those unchanged local patterns (e.g. `conv-relu` pattern appears in almost every CNN-based model); 2) similarly, the structure space of the sub-graphs is relatively small. In this case, the strengths of GNNs in learning and representing graph-level information can be leveraged to provide accurate latency prediction and kernel classification; 3) by shifting from the graph-level prediction to sub-graph level, the size of the training dataset increases significantly, which increases the accuracy and generalizability of our models. Unlike prior methods, Lilou can predict DNN latency for both dedicated GPUs and GPUs with arbitrary resource provision. Lilou also generalizes well across model and device variants and overwhelms the need to manually provide rules for layer fusion.

In designing, implementing and evaluating Lilou, we make the following contributions:

- *DNN Latency Modeling on GPU.* We design Lilou, a graph model-based latency predictor for accurately predicting a wide range of DNN inference latencies spanning various GPU and GPU configurations.
- *Fusion Rule Learning.* We propose a novel learning-based method to learn DNN layer fusion rules and predict fusible operators by performing edge-level binary classification using Graph Attention Network (GAT)[39] and LSTM[17]. We show that significantly improves the precision, recall, and accuracy of layer fusion detection. Notably, unlike traditional rule-based fusion detection methods that necessitate the handcrafting of rules, our method is fully automatic and can be integrated into MLOps frameworks to adapt to data drift, including updates in DNN architectures and software optimizations.
- *Performance Datasets.* To demonstrate the effectiveness of Lilou, we create and propose two large benchmark datasets `dnnFuse` and `dnnKernels`, which consist of architectures and performance of 51 published and representative DNN models, including CNNs and transformers, with *different* graph structures on 30 different GPU partitions. The datasets are designed for training GNNs. `dnnFuse` contains 91k nodes, which represent layer features, and 134k edge labels, which indicate if two nodes can be fused. `dnnKernels` comprises 2.2 million sub-graphs, which contain layers that can be fused and executed by one GPU kernel. The dataset includes latency and kernel information for these sub-graphs and the corresponding primitive layers that make up of them. In total, `dnnKernel` contains 1 million edges and 3 million nodes. To the best of our knowledge, these datasets are the first to incorporate DNN layer fusion data, providing comprehensive resources for further analysis and optimization of DNN performance.
- *Implementation and Evaluation.* We evaluate Lilou on these datasets and compare it with state-of-the-art approaches. Our results show that Lilou achieves overall 98.35% accuracy in fusible edge prediction and 8.68% MAPE in end-to-end latency prediction, significantly outperforming the start-of-the-art approaches. Additionally, we conduct a comprehensive analysis to illustrate the reasons behind our approach's superior accuracy.

2 BACKGROUND

This section provides background on DNN model serving in the cloud, DNN performance optimization, and graph neural networks.

2.1 DNN Model Serving in the Cloud

DNN model serving[8, 35] is a computational paradigm that hosts trained DNN models, providing inference services via a well-defined

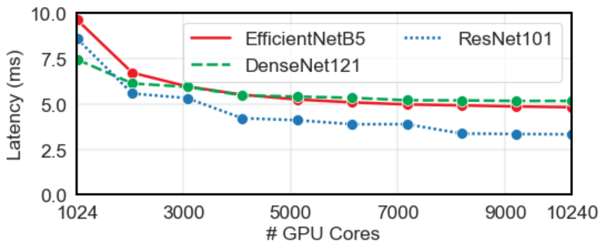


Figure 1: Number of GPU cores vs. latency. Limiting cores using MPS on Nvidia A10G GPU.

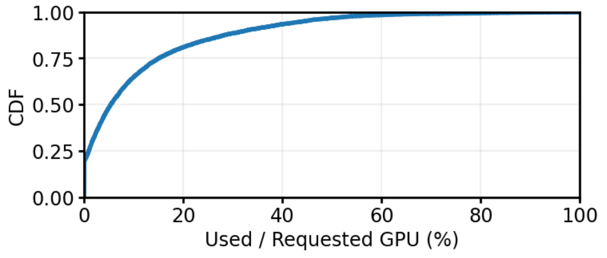


Figure 2: GPU utilization for inference tasks in Alibaba Platform for Artificial Intelligence.

interface, which allows seamless integration of deep learning functionality into a wide range of applications. DNN model serving has gained popularity in cloud environments due to the increasing availability and diversity of pre-trained models in domains, such as computer visions[15, 18, 38] and natural language processing[9].

Cloud-based model serving often involves the use of GPUs to accelerate the processing of DNN workloads, and these services usually have strict SLOs to ensure high performance and low latency for their users. In recent years, cloud providers have substantially expanded their offerings of GPU-based instances, catering to the growing demand for high-performance computing and machine learning workloads. These GPU devices, such as NVIDIA's Tesla and Ampere series, are available across a wide range of cloud platforms such as AWS, Google Cloud, and Microsoft Azure. This extensive selection of GPU devices in the cloud enables users to choose the most suitable option based on their specific performance and budget requirements. Moreover, emerging techniques such as MPS and MIG provide users with even greater flexibility in resource allocation for their workloads. MPS allows multiple applications to share a single GPU with limited resource provisioning, while MIG partitions a single GPU into multiple instances with dedicated resources. These advancements contribute to more efficient resource management. Figure 1 shows an example of serving DNNs using MPS on Nvidia A10G GPU. It allows limiting resource allocation for each model, at the cost of higher latency.

However, despite the growing demand for GPU in datacenters, GPU utilization often remains low. Figure 2 shows the GPU utilization distribution of inference tasks in Alibaba PAI (Platform for Artificial Intelligence)[40]. As shown by the figure, over 80% of tasks use less than 20% of the GPU resources requested. One of the reasons for low GPU utilization is the lack of accurate predictions of DNN performance under different GPU configurations. As a result, developers often overprovision resources to ensure SLOs are met. This conservative approach can lead to resource wastage and increase operational costs, highlighting the need for accurate performance prediction and efficient resource provisioning methods.

2.2 DNN Performance

Before deployment, DNN models usually undergo a series of optimization steps, shown in figure 3. These optimizations can significantly improve their performance in production environment and have become essential to support efficient execution of DNN models across various hardware. Common DNN runtimes and compiler frameworks include TensorRT[32], TVM[7], XLA[36], and AITemplate[4]. In the following, we briefly introduce the neural network structure and common optimization techniques.

Trained Neural Network. The optimization pipeline takes a trained DNN as input, which is represented as a directed acyclic computational graph in a universal format such as ONNX[3]. In the graph, nodes correspond to the individual layers, such as convolutional layers, fully connected layers, and batch normalization layers. Each layer has a set of hyperparameters (e.g. the number of hidden channels) that significantly affect the computational requirements and thus the inference latency. The edges specify the execution dependencies between layers, outlining the order in which the DNN processes information through its architecture.

Layer fusion. Layer fusion is one of the major optimization techniques applied to DNN computation graphs, where multiple adjacent layers in the network are combined into a single layer or operation. For example, a convolutional layer followed by a ReLU activation layer can be fused into a single layer by incorporating the activation function directly into the convolution operation. By doing so, the intermediate memory storage and additional overhead, such as launching separate kernels, are eliminated. As a result, the inference latencies are significantly improved.

Kernel tuning. An operation can be implemented using various algorithms. For instance, a convolutional layer may be implemented using GEMM, Winograd, or FFT algorithms, each of which exhibits different performance characteristics on specific hardware. Kernel tuning involves selecting the optimal algorithm for a given operation based on the target GPU platform, thereby improving the overall latency.

Generate machine code and execute. Finally, the optimized model is compiled to generate executable code for target hardware. Although the structure of some DNNs may allow parallel layer execution, kernels on GPUs are often executed *sequentially*. This is because concurrent kernel execution might adversely interfere with each other, resulting in higher overhead and less predictable execution time for GPU operations[2]. As a result, summing the latencies of individual kernels provides a reasonable approximation of the total DNN latency.

Additional optimizations, such as precision calibration and graph pruning, can also be applied to DNNs. However, these techniques may involve latency-accuracy trade-offs and typically require careful design by human experts. We assume these steps are completed before the DNNs enter the runtime optimization pipelines.

2.3 Graph Neural Network

Graph Neural Networks are a class of deep learning models designed to handle graph data. GNNs leverage graph-based operations, such

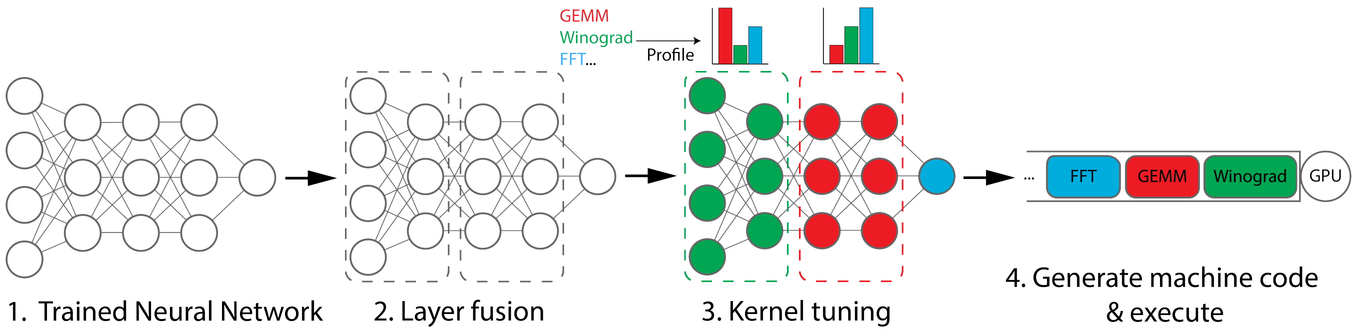


Figure 3: DNN runtime optimization steps.

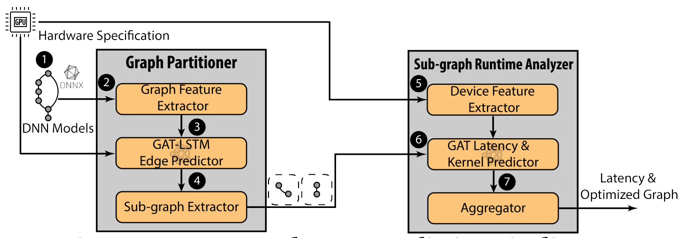


Figure 4: Lilou DNN latency prediction pipeline.

as message passing, graph convolutions (e.g. GCN[21]), and graph attention (e.g. GAT[39]), to propagate information through the network and learn meaningful representations of nodes and edges.

There are three general types of prediction tasks on graphs: 1) graph-level, which predicts labels for the entire graph; 2) node-level, which targets individual nodes; and 3) edge-level, which aims to predict labels for each edge. Prior works[6, 11, 14, 22, 26] have proposed predicting the end-to-end DNN latency by performing graph-level prediction, and shown significant accuracy improvements, compared to traditional layer-wise methods. However, these approaches fail to generalize to unseen graphs. In Lilou, we employ both edge- and graph-level predictions, resulting in enhanced generalizability across various graph structures.

3 DESIGN

Our main focus in designing Lilou is to accurately predict latency across a diverse range of DNN models on various GPU configurations. To do that, our method should have the following three desired attributes: ① the ability to identify fusible operations, ② sensitivity to resource allocations, and ③ operation without necessity for manual rule crafting. We start by providing an overview of Lilou prediction workflow and subsequently describing Lilou's key design components. Finally, we provide details of collecting our benchmark and training datasets.

3.1 Lilou Workflow

Figure 4 shows the prediction pipeline of Lilou.

❶ First, the system receives a DNN in a universal format, such as ONNX, along with the target GPU hardware configuration as input.
❷ The graph feature extractor component of the Graph Partitioner converts the DNN into a general graph format and extracts node features based on the computational semantics of DNN layers.

Type	Name	Description	Example Value
Operator	Operator type	Operator type or layer type.	Conv2d
Memory	Input size	The total number of elements in input tensors.	150528 (i.e. $1\times 3\times 224\times 224$)
	Output size	The total number of elements in output tensors.	1000
	Parameter size	The total number of weights and bias in this layer.	9472
Computation	FLOPs	The total number of floating point operations performed by this layer.	115806208

Table 1: Node features.

❸ This encoded graph and hardware type are then fed into a GAT-LSTM model to predict if an edge connects two fusible nodes for all edges in the graph.
❹ Once all edges are labeled, the sub-graph extractor partitions the origin graph into multiple sub-graphs, such that all nodes within the same sub-graph can be fused and executed by a single kernel or operator.
❺ To incorporate hardware semantics, the device feature extractor is used to extract performance-related features from the hardware configuration.
❻ Then, each sub-graph output from sub-graph extractor is then combined with the device features and fed into another GAT to predict the latency and kernel type of the sub-graph.
❼ Finally, the aggregator reassembles the sub-graphs into an optimized graph and sums up the latencies to calculate the total latency of the DNN.

3.2 Lilou Design Components

There are two core components in Lilou. The *Graph Partitioner* analyzes input DNN structures and *Sub-graph Runtime Analyzer* predicts their runtime performance. Both components employ learning-based methods for prediction. We next describe the design and functionality of each component.

3.2.1 Graph Partitioner. The goal of the graph partitioner is to divide the input DNN into partitions, each can be executed by a single function or GPU kernel. This design enables us to model DNN latency by analyzing each partition independently. The graph partitioner consists of three components — graph feature extractor, GAT-LSTM edge predictor, and sub-graph extractor.

Graph Feature Extractor. The job of this component is to extract layer features and convert the input DNN to a general graph format

$G = (V, E)$, such that $V \in \mathbb{R}^{n \times d}$ represents the layers in the DNN, where n is the number of layers and d is the dimension of the layer features. $E = \{(v_i, v_j)\}$ for all $v_i, v_j \in V$ such that the output of v_i is the input of v_j. To represent the structural and computational semantics of the DNNs, we extract an effective set of layer features shown in table 1. The operator type indicates the computational complexity and the optimization methods (e.g. fusion) that may apply to it. The input, output, and parameter size of the layer can affect the memory access, communication overhead, and fusibility. FLOPs represents the computational requirement of the layer.

GAT-LSTM Edge Predictor. To partition a DNN by kernel, it's crucial to identify the layers that can be fused together. *One of our key observations is that this fusibility relationship can be represented by labeled edges.* Specifically, two layers can be fused only if they are connected by an edge. More generally, k layers, $V' = \{v_1, \cdots v_k\}$ where $v_i \in V$, can be fused only if there exists a set of edges $E' \subseteq E$ such that the sub-graph $\mathcal{F}(V', E')$ is connected. Based on this observation, we define the set $U = \bigcup E'$, for all fusible sub-graphs $\mathcal{F}(V', E') \subseteq G$ as *fusible* edges. The primary task of the GAT-LSTM edge predictor is to classify whether each edge in the graph is a fusible edge or not.

We formally define the task as follows. Given a *directed* DNN computational graph $G(V, E)$ and target hardware type H. Let $m = |E|$ denotes the number of edges in the graph and $\mathbb{B} = \{0, 1\}$ denotes the boolean domain, where 1 corresponds to a fusible edge and 0 to a non-fusible edge. Our goal is to learn a mapping function f:

$$f : V \times E \times H \rightarrow \mathbb{B}^m \tag{1}$$

We employ a GAT-LSTM model to learn f. Our model consists of one GAT layer followed by one LSTM layer. We use GATs to extract node features because GATs are specifically designed to process graph-structure data. GATs are capable of exploiting the local structure and neighborhood information of nodes using message-passing and attention mechanisms. They naturally model relational information and dependencies between nodes, which is crucial for our fusibility prediction task. Each GAT layer has 128 hidden channels and is designed as GAT-GraphSizeNorm-ReLU pattern, where GraphSizeNorm[12] is used to normalize node features and defined as:

$$\mathbf{x}' = \frac{\mathbf{x}'}{\sqrt{|V|}} \tag{2}$$

We use an LSTM layer with 128 hidden channels to encode the edge features. The inputs of the LSTM layer are sequences with length 2, $[\mathbf{x_i}, \mathbf{x_j}]$, where $\mathbf{x_i}, \mathbf{x_j}$ are node embeddings output from the GAT layer for node v_i, v_j and $(v_i, v_j) \in E$. We make this design decision because we want to preserve the layer order information, which can significantly affect the fusion decision. For example, `conv-relu` can be fused but `relu-conv` cannot. Finally, a linear layer is used to make the final classification. Similar to many other classifiers, cross-entropy loss is used to train this model.

Sub-graph Extractor. The job of the sub-graph extractor is to divide the original graph into sub-graphs, based on fusible labels of the edges. Recall that fusible edges are defined by fusible layers. Conversely, given the predicted fusible edges, we can determine the fusible layers. The key observation is that one layer can only be executed by one kernel, in other words, kernels are disjoint sets of layers. Based on the observation, we partition the graph's nodes into disjoint sets, where each set represents a group of nodes connected by predicted fusible edges. To do that, we apply `Union-Find` data structure. We set up the data structure such that each node initially belongs to its own unique set containing only itself. Then, for all predicted fusible edge $(v_i, v_j) \in U$, we merge nodes v_i and v_j into a single set. Consequently, the resulting sets identify sub-graphs in which the layers can be fused together. The pseudocode for extracting sub-graphs is shown in algorithm 1.

Algorithm 1: Extract sub-graph

```
Input: $V$ a set of nodes in the origin DNN graph; $E$ a set of
       edges in the origin DNN graph, with predicted
       is_fusible label.
Output: Assign group label for all $v_i \in V$ such that
        $v_i$.group == $v_j$.group if and only if $v_i, v_j$ can be
        fused and executed by one kernel.
1  Function Find(parents, v):
2      if parents[v] != v then
3          parents[$v$] = Find(parents, parents[$v$])
4          return parents[$v$]
5      else
6          return v
7      end
8  End Function
9  Function Union(parents, $v_i$, $v_j$):
10     $p_i$ = Find(parents, $v_i$)
11     $p_j$ = Find(parents, $v_j$)
12     if $p_i$ != $p_j$ then
13         parents[$v_j$] = $p_i$
14     end
15 End Function
16 parents[$v_i$] = $i$
17 for $e = (v_i, v_j) \in E$ do
18     if e.is_fusible then
19         Union($v_i, v_j$)
20     end
21 end
```

Through the Graph Partitioner, Lilou effectively detects the fusion rules (goal ①) and successfully eliminates the necessity for manual rule crafting (goal ③).

3.2.2 *Sub-graph Runtime Analyzer.*

The objective of the sub-graph runtime analyzer is to estimate the overall latency of the DNN on target hardware by individually analyzing the sub-graphs produced by the graph partitioner. The sub-graph runtime analyzer consists of three components — Device Feature Extractor, GAT Latency & Kernel Predictor, and Aggregator.

Device Feature Extractor. Similar to the graph feature extractor, we extract an effective set of device features $\mathcal{D}$ shown in table 2 to represent the memory and computational semantics of the target hardware. These features are concatenated with the node features in the sub-graphs in order to integrate hardware knowledge into the prediction process.

Type	Name	Description	Example Value
Version	Compute capability	This feature identifies the set of features supported by GPU.	8.6
Memory	Memory bus width	The amount of data in bits that can be transferred at one time.	384
	Memory clock rate	The speed of GPU's memory in GHz.	6.251
Computation	Number of cores	The number of GPU cores.	10240
	Number of SM	The number of stream multi-processor.	80
	Compute clock rate	The speed of GPU cores in GHz	6.251

Table 2: Device features

GAT Latency & Kernel Predictor. We employ another GAT model to estimate the latency of each sub-graph generated by graph partitioner. Furthermore, we also predict the specific kernel that will execute the sub-graph. Since different kernels have different execution characteristics, knowing which kernels are used can help identifying potential performance bottlenecks and better understanding the predicted latency.

We formally define the tasks as follows. Given an *undirected* graph $G = (V, E)$, let $\mathbb{K}$ denotes the kernel type domains, where $k \in \mathbb{K}$ is a specific kernel implementation. Our goal is to learn a mapping function g:

$$g : V \times E \times \mathcal{D} \rightarrow \mathbb{R}, \mathbb{K} \tag{3}$$

By incorporating device-specific attributes, Lilou achieves the ability to be cognizant of the resource allocation, thereby addressing goal ②. We empirically choose to use undirected graphs in this task because the directions appear to have minimal impact on the latency and kernel implementation. Our model consists of three GAT layers followed by one linear classifier layer. Each GAT layer has 256 hidden channels and is designed as the GAT-GraphSizeNorm-ReLU pattern. We use `GlobalMeanAggregation` to aggregate node features to graph features. To train this multi-task model, we employ both *root mean squared error*(RMSE) and *cross entropy loss*(CE) to formulate the loss function, such that:

$$\mathcal{L} = RMSE(\hat{y}_{reg}, y_{reg}) + CE(\hat{y}_{cls}, y_{cls}) \tag{4}$$

where $\hat{y}_{reg}, y_{reg}$ are predicted values and ground truth values of latency; $\hat{y}_{cls}, y_{cls}$ are predicted values and ground truth values of kernel type.

Aggregator. The aggregator's role is to combine the latencies of all sub-graphs to predict the final end-to-end latency for the DNN. As the execution of each sub-graph is independent and the DNN must execute all of them, the aggregator models the end-to-end latency as the sum of the individual sub-graph latencies. In practice, the aggregator also reassembles the sub-graphs into an optimized graph where each node represents a fused kernel, labeled with the predicted kernel type. This facilitates further analysis for users interested in understanding the optimization process.

Models	# variants	# nodes		# edges		# kernels		GFLOPs		Latency(ms)	
		min	max	min	max	min	max	min	max	min	max
ConvNeXt	4	576	1080	657	1233	276	543	0.09	34.16	1.91	141.40
DenseNet	4	378	618	912	2420	723	2111	0.06	7.78	4.3	34.99
EfficientNet	8	239	815	312	1080	164	548	0.01	5.05	1.04	40.08
GoogLeNet	1	139	139	165	165	63	63	0.03	1.50	0.74	5.12
MNASNet	4	99	99	108	108	54	54	0.003	0.50	0.42	4.55
MobileNet	3	141	170	174	199	54	102	0.003	0.29	0.49	3.01
RegNet	15	121	360	136	413	55	196	0.007	91.66	1.02	402.41
ResNet	4	49	360	56	409	23	158	0.04	11.54	0.81	36.01
ResNeXt	1	241	241	273	273	107	107	0.30	14.60	3.05	61.32
SqueezeNet	2	65	65	72	72	30	30	0.004	0.82	0.23	3.15
Wide Resnet	1	241	241	273	273	107	107	0.47	22.78	4.42	61.01
GPT-2	2	686	1358	807	1599	3660	7260	240.04	707.62	13.58	1002.22
BERT	2	621	1221	730	1438	3720	7320	112.64	348.16	5.63	448.71

Table 3: DNN model characteristics in our dataset.

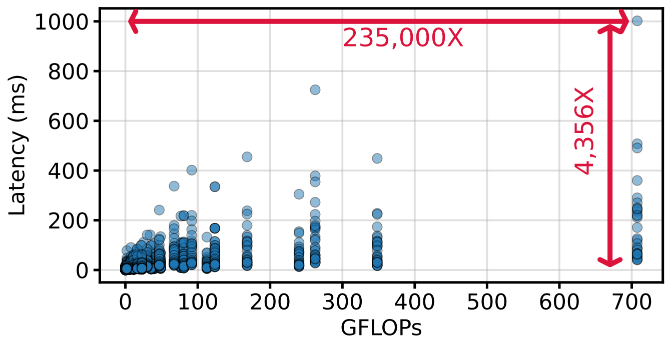


Figure 5: FLOPs vs. latency for various DNNs under diverse GPU configurations. Each data point represents a DNN tested under a specific GPU setup within our dataset.

3.3 Dataset Collection

To train our GAT models and evaluate the effectiveness of our system, we propose two representative datasets, `dnnFuse` and `dnnKernels`, that include runtime optimization and performance data for a diverse spectrum of DNN models and GPU configurations. When generating DNN models, our design principle is two-fold. Firstly, we aim to demonstrate the generalizability of Lilou in handling various DNN structures (i.e. DNNs with different number of layers and layer connections). Secondly, we aim to illustrate Lilou's precision in predicting latency across varying input sizes, thereby reinforcing its robustness and accuracy. Table 3 summarizes the DNN models included in our datasets. As shown, there are 13 distinct representative DNN families. Each family is further divided into one or more variants, each has a unique model structure. For example, in the `DenseNet` family, we have four different variants — `DenseNet121`, `DenseNet161`, `DenseNet169`, and `DenseNet201`; in the `GPT-2` family, we have two different variants — `GPT2` and `GPT2-large`. As a result, we have 51 model variants in total, covering a wide spectrum of node size, edge size, kernel size, and FLOPs. Additionally, we profile each model across varying input sizes (e.g. image resolution) to gain insights into their performance characteristic under different operational conditions. For example, for CNN models, we use image dimensions with heights and widths set to 32, 64, 128, 169, 192, and 224.

When creating GPU configurations, we consider three typical cloud GPUs — Nvidia T4, Nvidia V100, and Nvidia A10G. We employ the MPS to configure the GPU resources provided to the DNN execution. Specifically, we profile the DNNs' performance across GPU allocations ranging from 10% to 100% of the GPU capacity in 10% increments. Consequently, we have 30 GPU configurations that span a broad spectrum of GPU capacities.

We create our datasets based on these DNNs and GPU configurations. The `dnnFuse` dataset contains layer features in table 1 for 91,752 layers and edge labels for 134,346 edges. The `dnnKernels` dataset contains both layer and device features shown in table 1 and 2 respectively, for 2,752,560 layers. It also contains latencies and kernel types for 2,217,240 kernels. Figure 5 showcases the broad spectrum of FLOPs and latency values spanning across different models under various GPU configurations in our datasets. Our datasets span a substantial range, with latency varying up to $4,356\times$ and FLOPs varying up to $235,000\times$. Even for the same DNN architecture with the same FLOPs, the latency can vary up to $80\times$ based on hardware configurations, making accurate latency prediction challenging.

4 IMPLEMENTATION

In this section, we describe our implementation of Lilou. We implement Lilou in Python with 4,315 lines of code (LOC). As shown in figure 6, our implementation consists of three phases — data collection, GNN training, and latency inference.

DNN Data Collection. As mentioned in §3, we collect our DNN data on three Nvidia GPUs. The specifications of these GPUs are shown in table 4. We use MPS to configure the GPU capabilities. However, MPS only restricts the number of SMs and cores provided to an application. For example, if the MPS constraint is set to 50% for a process on Nvidia T4 GPU, the process has access to 20 SMs and 1280 GPU cores. Other configurations, specifically the memory bus width, clock rate, and GPU clock rate, will remain unaffected.

We collect our DNN models from `torchvision`[27] and HuggingFace [41]. We employ TensorRT(8.4.2) with CUDA 11.7 as the runtime for DNN inference tasks because of its detailed logging of runtime optimization and precise profiling information, which are crucial for our analysis. We profile our DNN models using the `trtexec` tool and Nvidia Nsight Systems, which support kernel-level profiling and logging of runtime optimization. To ensure accurate profiling, we monitor the variance of measurements. If the coefficient of variance exceeds 5%, we redo the profiling. Lastly, since the tools do not systematically produce layer fusion data, we implement a parser to extract them from the raw log outputs. All our datasets are collected in a standard dataset format described in [29].

GNN Training. We implement our GNN models using PyTorch Geometric (2.1.0)[13] and train them using `PyTorch`(1.12.0)[33]. We normalize the node features by subtracting the mean of each feature and a division by the standard deviation. We apply one-hot encoding for categorical features, such as layer operator type. We train our GAT-LSTM edge predictor model with a learning rate of 1e-2, a batch size of 32, and over 20 epochs. We employ the Adam optimizer[20] and a one-cycle learning rate scheduler[37]. We store the model weights that yield the highest accuracy on the validation set. We train our GAT latency and kernel predictor using the same strategy except with a learning rate of 5e-4 and batch size of 512 over 100 epochs.

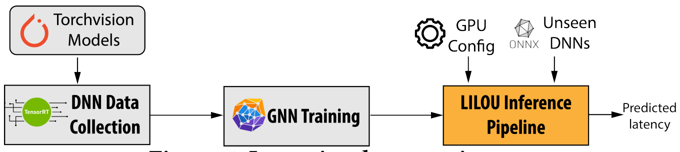


Figure 6: Lilou implementation.

Hardware	Architecture	Memory		Compute		
		width	clock	# cores	# SMs	clock
Nvidia T4	Turing	256	5.001	2560	40	1.56
Nvidia V100	Volta	4096	0.877	5120	80	1.53
Nvidia A10G	Ampere	384	6.251	10240	80	1.71

Table 4: GPU specifications. Memory bus widths are in bits. Clock rates are in GHz.

Lilou Inference Pipeline. In the inference stage, Lilou accepts DNNs in ONNX[3] format and a target GPU configuration vector. The feature extractor is implemented using the `ONNX`(1.10.2)[3] Python library for determining the shapes of inputs, outputs, and parameters for each layer based on input size. The FLOPs for each layer are computed using the `thop` Python library (0.1.0)][24]. Subsequently, these extracted features are provided as input to the trained models from previous stage to generate the predicted results.

5 EVALUATION

This section demonstrates the evaluations of Lilou through our benchmark datasets (table 3) on three cloud GPUs (table 4).

5.1 Experimental Setup

Baselines. We start by describing our experimental setup. We compare Lilou with the following baselines: (1) Linear Regression (LR), (2) BRP-NAS[11], and (3) nn-Meter[43]. For LR, we use FLOPs and parameter size to estimate the latency. BRP-NAS performs graph regression for latency prediction using GNNs. nn-Meter is the state-of-the-art method that first detects fusion kernels using handcrafted rules and then estimates the kernel latency individually. Since we use different hardware than they used in the original works, we retrain the models on our datasets using their source code.

Settings. We use a 5-fold cross-validation approach to evaluate Lilou. The 51 DNNs, each with distinct architectures, are randomly partitioned into 5 subsets. For each iteration, one subset is used as test set while the remaining 4 subsets are further randomly divided into a training set (70%) and a validation set (10%). This setting is used to train our GNNs and all baselines.

Hardware. We evaluate Lilou using GPUs shown in table 4 and MPS configurations described in §3.3. However, since none of the baselines support resource-aware prediction, we only compare Lilou with the baselines (§5.2 and §5.3) on Nvidia A10G GPU. We demonstrate Lilou's ability for resource-aware latency prediction using all aforementioned GPUs in §5.4.

5.2 Fusion Prediction Evaluation

We first evaluate the effectiveness of Lilou in predicting operator fusion, which directly affects the accuracy of end-to-end latency prediction. Since LR and BRP-NAS do not predict operator fusion,

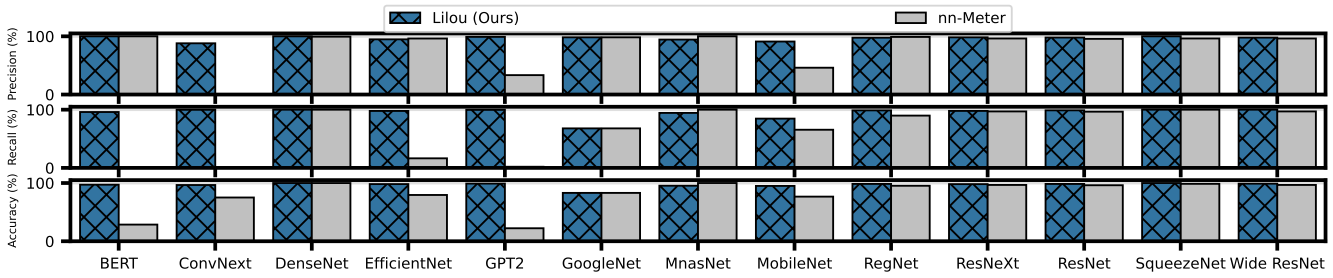


Figure 7: Evaluation results for fusion prediction. Lilou achieves overall higher precision, recall, and accuracy than the state-of-the-art handcrafted method.

Fused operations	Lilou	nn-Meter
`conv-relu`	99.83%	99.01%
`sigmoid-mul`	99.92%	0.00%
`batchnorm-relu`	100.00%	100.00%
`add-sqrt-div-mul-add`	100.00%	0.00%
`div-erf-add-mul-mul`	100.00%	0.00%
`hardsigmoid-mul`	100.00%	0.00%

Table 5: Prediction accuracy for various fusion patterns.

we only compare our Graph Partitioner with the state-of-the-art nn-Meter handcrafted fusion detector by comparing their edge prediction accuracy. Since nn-Meter does not inherently perform edge prediction, we employ an equivalent method to evaluate its performance in this domain. We directly run its source code for GPU kernel detection to identify the fusible operations. Then, edges that connect these predicted fusible operations are considered positive predictions in our edge prediction evaluation.

Figure 7 shows a comparative analysis of precision, recall, and accuracy between different predictors across various DNN model families. As shown, Lilou generally outperforms nn-Meter, demonstrating superior results across most evaluation instances. Specifically, Lilou achieves overall 97.21% precision, 98.22% recall, and 98.35% accuracy, significantly better than nn-Meter's 92.02% precision, 31.66% recall, and 73.27% accuracy. The performance of nn-Meter is fairly competitive on models like `ResNet` due to their well-known and frequently-used layer patterns such as `conv+relu`. These commonly used patterns are easily identifiable by the predefined reules of nn-Meter. However, its performance tends to diminish when encountering models with less common or more complex layer patterns. For instance, the current version of nn-Meter fails to recognize the `conv+sigmoid+mul` pattern in `EfficientNet`, leading to a substantial reduction in recall. In a more extreme example, the `ConvNext` model, characterized by numerous point-wise operations that are not included in the predefined rules, completely stumps nn-Meter, resulting in zero precision and recall. Additionally, while the nn-Meter's pre-defined rules are applicable to various CNN models, they do not generalize well to transformer-based models such as `BERT` and `GPT-2`. It fails to recognize many fusion patterns. For instance, our findings indicate that the fusible sequence of `Matmul-Transpose-Reshape` operations, which is prevalent in transformer models, is not recognized by nn-Meter. This limitation stems from the initial development phase, during which the rules were crafted without considering transformer architectures.

We further confirm this observation by dissecting the accuracy according to frequently used fusion patterns. As shown by table 5, nn-Meter yields high accuracy for fusion patterns that it has predefined, such as `conv-relu`. However, when encountering patterns absent from its collection of rules, including a variety of point-wise fusion patterns, its accuracy drops to 0. In contrast, Lilou is able to learn various rules from large dataset and identify the patterns accurately. Although it's possible to incrementally add newly discovered fusion rules to nn-Meter, attempting to exhaustively cover all potential fusion scenarios could be prohibitively time-consuming. As demonstrated, Lilou offers a more efficient solution by leveraging large-scale data to learn and accurately identify these rules. Lilou's capability in accurately predicting the fusible operators establishes a robust foundation for precise end-to-end latency prediction of DNNs, as will be demonstrated in the next section.

Key Takeaways. *Lilou achieves a remarkable accuracy of 98.35%, significantly outperforming the state-of-the-art handcrafted rule-based nn-Meter, which attains an accuracy of 73.27% in fusion prediction across a broad spectrum of CNN and transformer models. This superior performance is attributed to Lilou's effective learning and identification of diverse fusion patterns.*

5.3 End-to-end prediction evaluation

Next, we examine the effectiveness of Lilou in predicting the end-to-end DNN latency by comparing its performance against all baselines. Given the significant variation in latency among models, even within the same family, we compare using normalized metrics such as Mean Absolute Percentage Error (MAPE) and Root Mean Square Percentage Error (RMSPE).

Figure 8 illustrates the MAPE and RMSPE results obtained by different predictors across various model families. Lilou achieves overall better performance than all baselines. Specifically, on average, Lilou achieves a MAPE of 8.68%, which is significantly lower than the 30.24% of nn-Meter, the 130.90% of LR, and the 93.42% of BRP-NAS. The performance of BRP-NAS is notably inferior, as it struggles to generalize the graph representation it learned to unseen graphs. LR also performs poorly because it is agnostic of DNN architectures and fusion optimization. The latency error of nn-Meter is closely tied to the accuracy of its fusion prediction. For models where it accurately predicts fusion (e.g. `ResNet`), it can achieve

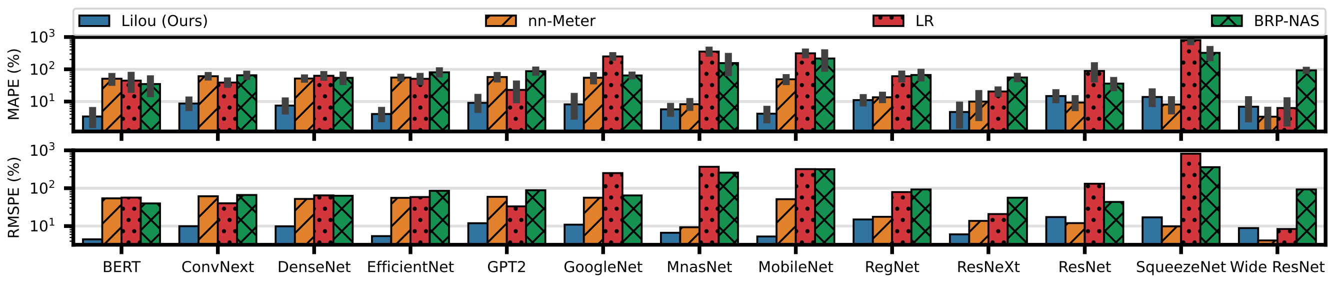
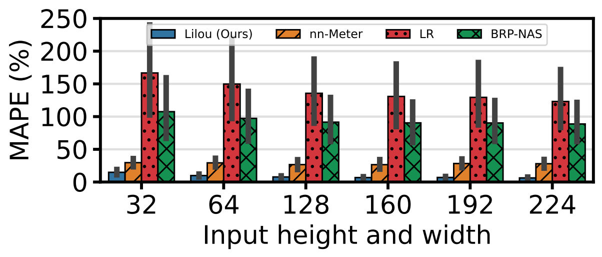
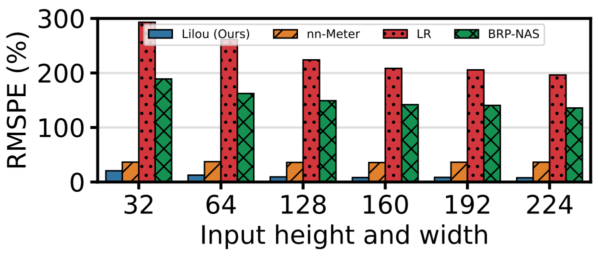


Figure 8: End-to-end DNN latency prediction comparison for all baselines.



(a) MAPE



(b) RMSPE

Figure 9: Prediction errors for different input sizes.

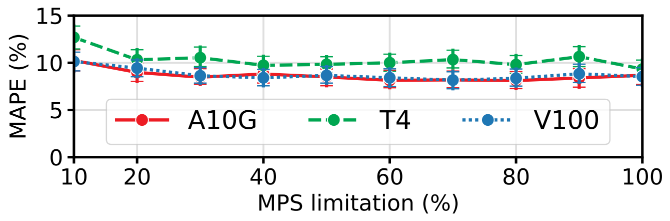


Figure 10: Mean DNN latency prediction error on all GPU configurations.

comparable latency error of Lilou. However, for models with low fusion prediction accuracy (e.g. `EfficientNet`), the latency error increases significantly. Notably, for models like `ConvNext`, where nn-Meter achieves zero precision and recall in fusion prediction, it performs even worse than LR and BRP-NAS.

We now take a step further to evaluate the predicted latency error with respect to varying input sizes. We group the DNNs by the height and width of the input ranging from 32 to 224. As shown in figure 9, all predictors tend to have higher latency errors on small input sizes. Prediction errors of LR and BRP-NAS fluctuate significantly across input sizes. In contrast, Lilou shows minimum variation in prediction error, shedding light on the consistency of Lilou across diverse input sizes.

Key Takeaways. *Lilou demonstrates remarkable generalizability and accuracy in DNN latency prediction, achieving a MAPE of 8.68%, which is a substantial improvement over the 30.24% achieved by existing state-of-the-art baseline.*

5.4 Resource-aware Prediction

In this section, we evaluate the effectiveness of Lilou in predicting DNN latency under different GPU and GPU partitions, which is crucial for efficient and QoS-aware resource management. To the best of our knowledge, Lilou is the first system to provide DNN latency prediction under custom hardware configuration. Although nn-Meter allows predicting DNN latency on different devices, it requires building separate predictors, while Lilou employs one predictor for all models and hardware configurations.

Models	A10G		T4		V100	
	MAPE (%)	RMSPE (%)	MAPE (%)	RMSPE (%)	MAPE (%)	RMSPE (%)
BERT	3.44	4.48	4.84	6.30	4.38	5.66
ConvNeXt	8.68	9.92	8.81	11.23	7.94	9.52
DenseNet	7.51	9.84	8.71	10.21	6.66	8.92
EfficientNet	4.11	5.44	5.68	7.12	4.82	6.04
GPT2	8.17	10.91	11.25	12.92	7.84	10.51
GoogLeNet	8.17	10.91	11.25	12.92	7.84	10.50
MNASNet	7.53	6.66	13.04	16.30	9.40	12.02
MobileNet	4.21	5.31	8.31	10.28	6.00	8.01
RegNet	11.11	14.97	11.19	14.71	9.20	12.65
ResNet	14.81	17.32	9.08	12.00	14.43	19.06
ResNeXt	4.75	6.06	7.47	8.62	6.92	8.45
SqueezeNet	13.87	17.01	12.79	14.95	15.32	18.64
Wide Resnet	6.91	8.85	9.18	10.76	8.15	10.17

Table 6: End-to-end latency prediction error on different GPUs.

Table 6 presents the results of latency prediction error for each model family on the three GPUs shown in table 4. As shown, the system achieves accurate and robust prediction across all evaluated GPUs, which have vastly different architectures and capability. Specifically, on average, Lilou achieves 8.66%, 10.33%, and 8.76% of MAPE on Nvidia A10G, T4, and V100 GPUs respectively.

To further show the flexibility and generalizability of Lilou, we customize GPU capability using MPS as described in §4. Figure 10

Operations	Time
Data collection	1 week
Train Graph Partitioner	10 minutes
Train Sub-graph runtime analyzer	1 hour

Table 7: Time cost for building Lilou.

shows the MAPE for all MPS configurations on all GPUs. Impressively, although the resource allocation varies significantly, Lilou maintains stable latency errors across all GPU configurations. We observe a minor increase in error rate at 10% MPS configuration. This can be attributed to the substantial fluctuations in latency when the provisioned resources are relatively limited (figure 1).

Key Takeaways. *Lilou consistently delivers accurate and robust DNN latency predictions across diverse GPU architectures and resource allocations.*

5.5 Kernel Prediction Evaluation

In this section, we examine the effectiveness of Lilou in predicting specific kernels that will be used to execute the sub-graphs at runtime. In Lilou, each sub-graph is designed to be executed by a single kernel, while different kernels can implement the same sub-graph. The kernel selection depends on the hardware, resource allocation, and graph structures. There are 140 unique GPU kernels that are involved for executing the 51 DNNs in our datasets. We group these kernels by the layers they implement and their underlying libraries and show the prediction accuracy in figure 11. On average, Lilou achieves overall 88.63% top-1 accuracy. We achieve high accuracy for ElementWise, PointWiseV2, Reformat, Scale, and Shuffle operations. However, the accuracies for Convolution and Pooling operations are relatively low. The reason is that TensorRT selects kernels based on runtime latency profiling. Specifically, it measures the latencies of all kernel candidates and selects the one with the lowest latency. However, because many kernels share similar implementations, their latency differences are minor, leading to potentially inconsistent ranking across different runs. We relax our evaluation metric by considering top-5 accuracy. Lilou can reach overall 98.84% accuracy and a minimum of 93.82% accuracy across all operations.

Key Takeaways. *Lilou is able to accurately predict runtime kernels for various DNNs across diverse GPU architectures and resource allocations.*

5.6 System overhead

Finally, we evaluate the system overhead. The main overhead comes from the time cost for data collection and training the GNN models. As shown in table 7, collecting data from different GPUs takes approximately one week. Training the Graph Partitioner takes less than 10 minutes and training the Sub-graph runtime analyzer takes about 1 hour on Nvidia A10G GPU. Notably, these processes are fully automated, which simplifies the extension to new devices.

6 RELATED WORK

DNN latency prediction is an open research research problem. Because of the complexity and diversity of DNN architectures, early works[16, 42] simply employ traditional metrics used in performance evaluation of computational systems, such as FLOPs, to build regressors for DNN latency prediction. FLOPs-based DNN latency regression is widely used because of its simplicity and strong correlation between FLOPs and execution time. Specifically, FLOPs in a DNN is a simple metric that can be easily calculated from the architecture without executing the network, and networks with more FLOPs will often require more time to execute. However, FLOPs-based DNN latency regressors do not account for the characteristics of different layers. Therefore, while they can provide a useful first approximation, more sophisticated methods are needed for accurate DNN latency prediction.

To address the layer- or operator-agnostic problem of FLOPs-based regressors, layer-level modeling methods[5, 19, 28, 34] are proposed. The key observation of these methods is despite the huge amount of different DNN variations, all these DNN architectures consist of basic underlying building blocks/primitives, such as convolutional and fully connected layers, which show similar execution characteristics per type of layer. Given this observation, these works model the latency of the basic layers and sum them up as the model latency. However, layer-level modeling does not take into account runtime optimization factors such as operator fusion, which can significantly impact actual latency. Consequently, compared to Lilou which explicitly considers the effect of operator fusion, layer-level modeling yields less accurate predictions.

The recent advancements in Graph Neural Networks have opened up new possibilities for DNN latency prediction. The inherent graph structure of DNN provides an ideal scenario to leverage GNNs for modeling and predicting their latency. BRP-NAS[11] and DNNPerf[14] take computation graph of DNNs as input of the GNNs and predict the latency of the DNNs at graph-level. Sectum[22] employs GNNs to detect if a given model would trigger memory over-commitment, which can significantly affect the inference latency, under a certain hardware configuration. These works have shown that GNNs can learn both layer characteristics and fusion optimizations. However, the generalizability of graph-level GNN predictors is poor. When applied to unseen models, their accuracy drops significantly. While Lilou also leverages GNNs, it employs them in a distinct manner. Instead of learning fusion operations implicitly and conducting graph-level predictions, we explicitly harness GNNs to learn fusion operations through edge-level predictions.

The recent state-of-the-art nn-Meter[43] proposes dividing a whole model inference into kernels, the execution units on a device, and conducting kernel-level prediction. It performs kernel detection by applying handcrafted fusion rules. However, as we have shown, rule-based kernel detection can be time-consuming and error-prone. The error in fusion prediction could directly translate to error in latency, resulting in inaccurate predictions. In contrast, Lilou applies learning-based method to detect fusion operations, offering a more efficient and accurate solution.

7 CONCLUSION

In this paper, we propose Lilou, a learning-based system designed for predicting latency across diverse GPU resource allocations for a wide range of DNN inference tasks. Lilou innovatively utilizes edge classification on DNN computation graphs to learn operation fusion rules. We build resource-aware latency predictor by incorporating

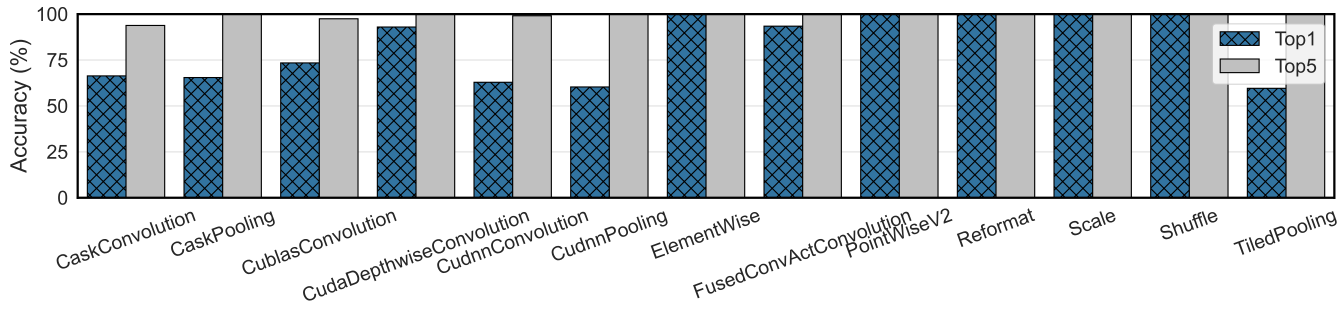


Figure 11: Kernel type classification accuracy.

GPU hardware features, enabling latency prediction even under custom GPU resource allocations. We evaluate Lilou by comparing it with state-of-the-art methods on a large dataset containing a wide range of DNN models. We also show Lilou's robustness and generalizability by evaluating it under different GPU resource allocations.

REFERENCES

[1] Amazon Web Services. 2023. Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/.

[2] Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. 2017. GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*. 104–115. https://doi.org/10.1109/RTSS.2017.00017

[3] Junjie Bai, Fang Lu, Ke Zhang, et al. 2019. ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx.

[4] Xu Bing, Zhang Ying, Lu Hao, Chen Yang, Chen Terry, and Ajit Mathews. 2022. Faster, more flexible inference on GPUs using AITemplate, a revolutionary new inference engine. https://ai.facebook.com/blog/gpu-inference-engine-nvidia-amd-open-source/.

[5] Ermao Cai, Da-Cheng Juan, Dimitrios Stamoulis, and Diana Marculescu. 2017. Neuralpower: Predict and deploy energy-efficient convolutional neural networks. *arXiv preprint arXiv:1710.05420* (2017).

[6] Yuji Chai, Devashree Tripathy, Chu Zhou, Dibakar Gope, Igor Fedorov, Ramon Matas, David M. Brooks, Gu-Yeon Wei, and Paul N. Whatmough. 2023. PerfSAGE: Generalized Inference Performance Predictor for Arbitrary Deep Learning Models on Edge Devices. *ArXiv* abs/2301.10999 (2023). https://api.semanticscholar.org/CorpusID:256274662

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[8] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 613–627.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *ArXiv* abs/1810.04805 (2019).

[10] Xuanyi Dong and Yi Yang. 2020. NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search. In *International Conference on Learning Representations (ICLR)*.

[11] Łukasz Dudziak, Thomas Chau, Mohamed S. Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas D. Lane. 2020. BRP-NAS: Prediction-Based NAS Using GCNs. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 879, 11 pages.

[12] Vijay Prakash Dwivedi, Chaitanya K. Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. 2020. Benchmarking Graph Neural Networks. *ArXiv* abs/2003.00982 (2020).

[13] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[14] Yanjie Gao, Xianyu Gu, Hongyu Zhang, Haoxiang Lin, and Mao Yang. 2023. Runtime Performance Prediction for Deep Learning Models with Graph Neural Network. In *ICSE '23*. IEEE/ACM. The 45th International Conference on Software Engineering, Software Engineering in Practice (SEIP) Track.

[15] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015), 770–778.

[16] Benedict Herzog, Stefan Reif, Judith Hemp, Timo Hönig, and Wolfgang Schröder-Preikschat. 2022. Resource-Demand Estimation for Edge Tensor Processing Units. *ACM Trans. Embed. Comput. Syst.* 21, 5, Article 58 (oct 2022), 24 pages. https://doi.org/10.1145/3520132

[17] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (nov 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[18] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. 2016. Densely Connected Convolutional Networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), 2261–2269.

[19] Daniel Justus, John Brennan, Stephen Bonner, and Andrew Stephen McGough. 2018. Predicting the Computational Cost of Deep Learning Models. In *2018 IEEE International Conference on Big Data (Big Data)*. 3873–3882. https://doi.org/10.1109/BigData.2018.8622396

[20] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014).

[21] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*.

[22] Yan Li, Junming Ma, Donggang Cao, and Hong Mei. 2022. Sectum: Accurate Latency Prediction for TEE-hosted Deep Learning Inference. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. 906–916. https://doi.org/10.1109/ICDCS54860.2022.00092

[23] Zhuojin Li, Marco Paolieri, and Leana Golubchik. 2023. Predicting Inference Latency of Neural Architectures on Mobile Devices. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering* (<conf-loc>, <city>Coimbra</city>, <country>Portugal</country>, </conf-loc>) *(ICPE '23)*. Association for Computing Machinery, New York, NY, USA, 99–112. https://doi.org/10.1145/3578244.3583735

[24] Zhu Ligeng et al. 2018. THOP: PyTorch-OpCounter. https://github.com/Lyken17/pytorch-OpCounter.

[25] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. 2018. Progressive Neural Architecture Search. In *Computer Vision – ECCV 2018: 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part I* (Munich, Germany). Springer-Verlag, Berlin, Heidelberg, 19–35. https://doi.org/10.1007/978-3-030-01246-5_2

[26] Liang Liu, Mingzhu Shen, Ruihao Gong, Fengwei Yu, and Hailong Yang. 2023. 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* (Bordeaux, France) *(ICPP '22)*. Association for Computing Machinery, New York, NY, USA, Article 78, 14 pages. https://doi.org/10.1145/3545008.3545051

[27] Sébastien Marcel and Yann Rodriguez. 2010. Torchvision the Machine-Vision Package of Torch. In *Proceedings of the 18th ACM International Conference on Multimedia* (Firenze, Italy) *(MM '10)*. Association for Computing Machinery, New York, NY, USA, 1485–1488. https://doi.org/10.1145/1873951.1874254

[28] Diana Marculescu, Dimitrios Stamoulis, and Ermao Cai. 2018. Hardware-Aware Machine Learning: Modeling and Optimization. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (San Diego, CA, USA). IEEE Press, 1–8. https://doi.org/10.1145/3240765.3243479

[29] Christopher Morris, Nils M. Kriege, Franka Bause, Kristian Kersting, Petra Mutzel, and Marion Neumann. 2020. TUDataset: A collection of benchmark datasets for learning with graphs. *CoRR* abs/2007.08663 (2020). arXiv:2007.08663

[30] Nvidia Corporation. 2023. Multi-Instance GPU (MIG). https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html.

[31] Nvidia Corporation. 2023. Multi-Process Service (MPS). https://docs.nvidia.com/deploy/pdf/\CUDA_Multi_Process_Service_Overview.pdf.

[32] Nvidia Corporation. 2023. TensorRT SDK. https://developer.nvidia.com/tensorrt.

[33] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[34] Hang Qi, Evan R. Sparks, and Ameet Talwalkar. 2017. Paleo: A Performance Model for Deep Neural Networks. In *Proceedings of the International Conference on Learning Representations*.

[35] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 397–411. https://www.usenix.org/conference/atc21/presentation/romero

[36] Amit Sabne. 2020. XLA : Compiling Machine Learning for Peak Performance.

[37] Leslie N. Smith and Nicholay Topin. 2017. Super-convergence: very fast training of neural networks using large learning rates. In *Defense + Commercial Sensing*.

[38] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. *ArXiv* abs/1905.11946 (2019).

[39] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJXMpikCZ

[40] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 945–960. https://www.usenix.org/conference/nsdi22/presentation/weng

[41] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2019. HuggingFace's Transformers: State-of-the-art Natural Language Processing. *ArXiv* abs/1910.03771 (2019). https://api.semanticscholar.org/CorpusID:268093756

[42] Gingfung Yeung, Damian Borowiec, Adrian Friday, Richard Harper, and Peter Garraghan. 2020. Towards GPU Utilization Prediction for Cloud Deep Learning. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association. https://www.usenix.org/conference/hotcloud20/presentation/yeung

[43] L. Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. 2021. nn-Meter: towards accurate latency prediction of deep-learning model inference on diverse edge devices. *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services* (2021).

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009