PRoof: A Comprehensive Hierarchical Profiling Framework for Deep Neural Networks with Roofline Analysis

Siyu Wu
wusiyu@buaa.edu.cn
Beihang University
Beijing, China

Hailong Yang
hailong.yang@buaa.edu.cn
Beihang University
Beijing, China

Xin You
youxin2015@buaa.edu.cn
Beihang University
Beijing, China

Ruihao Gong
gongruihao@sensetime.com
SenseTime Research
Beijing, China

Yi Liu
yi.liu@buaa.edu.cn
Beihang University
Beijing, China

Zhongzhi Luan
07680@buaa.edu.cn
Beihang University
Beijing, China

Depei Qian
depeiq@buaa.edu.cn
Beihang University
Beijing, China

ABSTRACT

The increasing diversity of deep neural network (DNN) models and hardware platforms necessitates effective model profiling for high-performance inference deployment. Current DNN profiling tools suffer from either *limited optimization insights* due to the missing correlation between high-level DNN layer design and low-level hardware performance metrics, or *prohibitive profiling overhead* due to the large amount of performance measurement through hardware performance counters. Meanwhile, the roofline model has been widely used in the high-performance computing (HPC) domain for identifying performance bottlenecks and guiding optimizations. However, it lacks hierarchical (e.g., kernel/operator/layer), fine-grained, multi-platform support for profiling DNN models.

To overcome the above limitations, we propose PRoof, a versatile DNN profiling framework, that can effectively attribute the hardware performance metrics back to the model design. In addition, PRoof does not require massive hardware profiling and thus mitigates the large profiling overhead. Specifically, our approach correlates the profiled result of each layer to their conceptual layer design by effectively handling layer fusion. Our approach also provides an analytical model to predict the floating-point operations (FLOP) and memory accesses of DNN models without massive profiling. We demonstrate the effectiveness of PRoof with representative DNN models across a wide range of hardware platforms. Derived from PRoof's profiling results, we obtain several insights to provide useful guidance for model design and hardware tuning.

CCS CONCEPTS

• **Computing methodologies → Machine learning**; • **Computer systems organization → Parallel architectures**; • **General and reference → Performance**.

KEYWORDS

DNN profiling, hardware performance metrics, roofline model

ACM Reference Format:
Siyu Wu, Hailong Yang, Xin You, Ruihao Gong, Yi Liu, Zhongzhi Luan, and Depei Qian. 2024. PRoof: A Comprehensive Hierarchical Profiling Framework for Deep Neural Networks with Roofline Analysis. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3673038.3673116

1 INTRODUCTION

In recent years, deep neural networks (DNNs) have emerged as the most prevalent method in the field of artificial intelligence. The evolution of DNN models such as Transformer [27], ViT [7], Swin [15], Diffusion [22] retains an ever-increasing trend of model parameter and computation demand. To reduce model inference latency and increase throughput, researchers have proposed various optimization techniques such as point-wise convolution, distillation, and neural architecture search (NAS), aiming at reducing computation complexity while preserving acceptable accuracy. However, the effectiveness of these optimization techniques may not result in a real-world performance improvement with overheads like additional memory access [20] or vary across different hardware platforms (e.g., CPU, GPU, and NPU), leading to inconsistent performance gains, and in some cases, even negative effects.

To better understand the inference performance of DNNs, fine-grained layer-wise profiling of models is necessary. However, the complexity of the software stack for DNN inference poses some challenges: although DL frameworks such as PyTorch [19] can perform model inference on their own, its performance is often not optimal due to the need to balance the convenience of development,

Table 1: Profiling tools for DNNs.

Type *(Example)*	Mapping to model design	Production performance	Hardware metrics
DL frameworks *pytorch-OpCounter*	✓	✗	✗
Inference runtime's built-in profilers *OpenVINO's benchmark_app*	✗ (optimized backend layer only)	✓	✗
Hardware profilers (if provided) *Nsight Compute*	✗ (kernel name only)	✓ (when profiling inference runtimes)	✓
PRoof (ours)	✓	✓	✓✓ (measured or predicted)

platform universality, and model training functions. This has led to the emergence of various DNN inference runtimes dedicated to the inference deployment of DNNs, such as TensorRT [5], OpenVINO [12], ONNX Runtime [18], some of which are provided by hardware vendors to achieve optimal performance on their specific hardware, but usually are closed source and proprietary. During the deployment, the model will first be exported from a DL framework like Pytorch into a general self-contained format (typically ONNX [8]) and then imported into the production inference runtime. The inference runtime will then optimize the model by transforming the compute graph to significantly improve its performance (e.g., operator fusion), and the optimized model is typically saved in a proprietary format for later production run with the inference runtime, where the optimized layer (later called the backend layer) will be different from the original model design. Therefore, for further model design and hardware optimization, developers need to profile the optimized production DNN model with metrics reflecting specific hardware characteristics and utilization and attribute the problematic backend layers to layers in model design.

Unfortunately, to the best of our knowledge, there is no profiling tool that can accomplish the above goals. Specifically, we examined the existing DNN profiling tools and classified them into three types, as shown in Table 1. The first type is the profiling tools associated with the DL framework, such as PyTorch's built-in profiler which only provides the latency of the model layers, and *pytorch-opcounter* which is based on PyTorch, further counts the number of theoretical FLOP of these model layers and thus calculates the FLOP/s[1] performance it achieves. However, these metrics reflect the performance of the inference of the model directly on the specific DL framework, and do not reflect its performance on an optimized production inference runtime for model deployment. Besides, these profilers provide insufficient hardware performance metrics (e.g., only FLOP/s) and lack the memory access characteristics that are equally important for hardware optimization.

For the second type, the built-in profilers for the inference runtime, only provide the latency of the backend layer, which is difficult to map back to the layer in the model design and thus poor insights for model design optimization. They also lack hardware performance metrics for further fine-grained optimizations.

For the third type, the hardware profilers such as Nsight Compute can provide fine-grained hardware performance metrics and latencies of kernels, but they cannot map to the layers by themselves to provide insights for model design optimization. Besides, these vendor-provided hardware profilers often lack portability to support the ever-increasing hardware platforms and models.

Among the existing profiling techniques, the roofline model [28] has gained widespread adoption in high performance computing due to its effective hardware and software optimization guidance. The roofline model consists of hardware peak performance, memory bandwidth, attained performance (typically in FLOP/s), and arithmetic intensity of the target application. By analyzing each layer (i.e., operator) of the model in a roofline model, one can observe the performance bottleneck of the model on a particular hardware intuitively for model design optimization guidance. Conversely, for hardware optimization, the roofline model can be used to obtain the performance requirements needed for a model's inference. For instance, a lower arithmetic intensity of a model requires higher memory bandwidth for achieving better performance. We present a case study for each of these two aspects in Section 4.5 and Section 4.6, respectively.

Constructing the roofline model requires latency, FLOP count, and memory access metrics. Although these values can be obtained from hardware profilers, they cannot correlate the collected performance metrics to the layer in either inference runtime or model design without extra engineering efforts (e.g., NVTX instrumentation). Specifically, to construct a layer-wise roofline model for DNNs with effective optimization guidance, we need to address the following two non-trivial challenges.

1) Lack of effective mapping between the runtime optimized layer (a.k.a., backend layer) and the conceptual model design layer, which is critical to understanding and pinpointing the performance inefficiency for both model developers and hardware architects. With progressive optimization by the DNN inference runtimes, the model often results in a highly optimized model for inference, which is significantly different from the conceptual model designs. For example, the most common layer fusion optimization results in multiple conceptual model layers being fused into a single backend layer. Moreover, additional backend layers may be introduced by the model runtime for tensor format and data type conversion. For some platforms, the backend layer may be further lowered to one or more operations (e.g., CUDA kernel), which should also be mapped correctly. Consequently, when performance bottlenecks are identified, it is challenging to map the profiling results back to the conceptual layers for effective performance optimization guidance.

Especially when it comes to the co-optimization of model design and hardware efficiency, the kernel-level performance metrics produced by standalone vendor profiling tools are insufficient. We need to integrate an understanding of the role of original model layers within the model architecture with their hardware performance metrics. This allows us to modify the model design to achieve better hardware performance while maintaining model effectiveness, as demonstrated in the case study in Section 4.5.

[1]For INT8 and other integer quantized models, the metric should be integer operation per second (OP/s). For simplicity, we use the term FLOP/s to represent both floating-point and integer operations per second in this paper.

2) Lack of a generic metric measurement methodology to collect the FLOP/s and memory bandwidth for each backend layer. One commonly adopted approach leverages profiling tools to measure the model performance via hardware performance counters provided by hardware vendors (e.g., Nsight Compute [4]), which is the most applicable method for certain scenarios. However, not all platforms offer these kinds of tools, especially some new hardware or embedded devices (e.g., Raspberry Pi). Besides, some vendors' tools incur significant profiling overheads that hinder the effective profiling of larger models on some slower devices (e.g., edge GPU or CPU).

To address the first challenge, we propose a novel and effective mapping strategy that bridges the gap between runtime optimized layer and the conceptual model layer. By leveraging the model design and vendor profiling tools, we can establish the correct mappings between the backend layers and the conceptual layers without any modifications to the runtimes. However, for certain model runtimes, obtaining the correct mapping is not an easy task. For example, NVIDIA TensorRT adopts an internal optimizer named *Myelin*, which does not provide any information about the mapping either in the execution logs or its built-in profiling tools. The limited mapping information can only be obtained through the Nsight Systems [3], which is not enough to understand the profiling results. To enhance compatibility with diverse inference runtimes, we have designed a set of universal methods to extract the correct mappings. These methods search the computational graph and leverage context and data dependencies to extract the correct mappings between the backend layers and the conceptual layers, even with limited information from the runtimes. The process does not require human effort and can handle complex operator fusion correctly.

To address the second challenge, we propose a generic and hardware-independent analytical model to estimate the amount of computation (FLOP) and memory access (read/write bytes) of each backend layer as a supplement or alternative to the actual measurement method with the vendor tools. We then employ the latency of each backend layer reported by the runtime to ascertain the achieved FLOP/s and memory bandwidth, requiring only latency measurements, easily obtained via the runtime's built-in profiler. Our analytical model is inspired by the observation that DNNs tend to have static control flow during inference and their operator implementations typically leverage on-chip SRAM adequately for data reuse, thereby greatly reducing the occurrence of duplicate memory accesses. Thus, we can estimate the FLOP of each operator in the model based on the input shape and operator type and estimate the memory accesses of an operator by its input and output sizes with rules associated with operator types. Our analytical model has demonstrated acceptable accuracy as an alternative solution, compared to the measured ones, with no platform limitations and negligible analytical overhead.

Based on the above two novel profiling techniques, we propose PRoof (DNN Profiler with Roofline analysis) [2], a comprehensive hierarchical DNN inference profiling framework for guiding model design and hardware optimization through the entire DNN software stack. PRoof can be adopted with or without an applicable hardware profiling tool from a vendor. PRoof accepts an ONNX model as input and automatically performs layer-wise and end-to-end roofline analysis. For intuitive optimization guidance, PRoof implements a data-viewer to provide a user-friendly visualization of the profiled results, including the layer-wise latency, FLOP/s, and memory bandwidth, as well as the roofline chart.

Specifically, this paper makes the following contributions:

- We propose a series of effective strategies to map the backend layers back to the model design layers, that can be applied to different model runtimes. Based on the proposed mapping strategies, we can apply more instructive and accurate roofline analysis.
- We propose a novel roofline profiling method that predicts the FLOP and memory accesses of a model through analytical modeling. This approach reduces the excessive overhead associated with hardware performance counter profiling and is operable on platforms lacking profiling support.
- We present PRoof, an efficient profiling tool for DNN models across various hardware platforms. PRoof can provide a visual report including end-to-end and layer-wise roofline analysis.
- We evaluate PRoof with 20 representative models on 7 different hardware platforms and show the useful insights derived from PRoof's profiling results.

2 RELATED WORK

Deep learning frameworks like PyTorch [19] and DNN inference runtimes [5] usually provide built-in profiling tools. However, such tools only offer relatively primitive profiling support, primarily focusing on basic measurements such as layer-wise inference latencies.

Moreover, the built-in profilers of the deep learning frameworks only measure the performance of inference within the framework. In real-world model deployment, a dedicated and optimized DNN inference runtime is often utilized for better performance. Due to the significant performance gap, the built-in profilers struggle to accurately reflect the performance of the actual model deployment. It is even more challenging to guide performance from the perspective of inference hardware. In contrast, profilers provided by DNN inference runtimes can accurately represent the performance of the actual model deployment. However, they only profile the latency of the backend layer, which is optimized by the runtime and cannot further reflect its performance characteristics from the hardware perspective. Additionally, due to the disparity between the backend layer and the original model layer, it is difficult to trace back to the original model when diagnosing performance issues.

Some hardware platforms provide lower-level general-purpose roofline collection tools, such as Intel VTune [1] and NVIDIA Nsight Compute [4], which accurately measure and construct roofline models based on hardware performance counters. However, such low-level measurements can only provide performance insights at the granularity of the underlying operators (e.g., GPU kernel), which is difficult to associate with the model layer and has little performance insights for model designers. Furthermore, due to the limited hardware performance counters, profilers need kernel replays to collect all necessary performance metrics, which incurs significant overhead and does not apply to all scenarios.

[2]https://github.com/PRoof-framework/PRoof

There are also some research studies on the roofline model and cross-software-stack profiling techniques. In terms of profiling from the perspective of inference hardware, there have been some works that can perform layer-wise roofline analysis on the model. Some of them are based on measurements with hardware performance counters [14, 29], which still suffer from poor universality and high overheads. For instance, XSP [14] proposes a way to associate the original model layer with the underlying GPU kernel on TensorFlow through distributed trace collection. However, XSP is limited to the deep learning framework running on GPUs. XSP also depends on the log output of each level within the entire software stack. Since many models cannot provide logs that meet the requirements when they are running, the profiling techniques provided in XSP are not universal. Other approaches are based on prediction, like MD-Roofline [17], but they cannot be applied to DNN inference runtime, and operator fusion is not considered.

There are also other emerging profiling techniques for tensor or dataflow programs. Benanza [13] is an optimization space exploration tool based on microbenchmarks, but it mainly focuses on the deep learning framework and cannot be applied to more efficient DNN inference runtimes for model deployment. Unlike PRoof, such micro-benchmark-based approaches cannot provide model and hardware designers with useful insights about hardware efficiency and bottlenecks. For distributed training, dPRO [11] considers some hardware performance characteristics and optimizations such as tensor fusion during profiling, but it provides performance guidance for distributed training strategies, which is orthogonal to PRoof. Daydream [30] is a profiling tool for simulating optimizations on DNN training, which cannot be easily adopted for DNN inference within dedicated runtimes and provides the perspective of inference hardware. NVIDIA DLProf [2] is a deep learning profiling tool for NVIDIA platforms, but it can only provide latency-based statistics associated with GPU kernels, which cannot provide further insights from the hardware perspective.

3 DESIGN AND IMPLEMENTATION

3.1 Design overview

The overall structure of PRoof is shown in Figure 1. The main part of PRoof is designed as a CLI program that accepts an ONNX model and a specified DNN inference runtime (a.k.a., *backend*) as input. The ONNX model format is widely used and can be easily exported from nearly all commonly used deep learning frameworks. Then, PRoof collects all necessary performance metrics through model analysis and measurements. The collected data can be viewed through the PRoof dataviewer, providing an intuitive visual report for performance optimization guidance.

PRoof mainly consists of two components: the *analysis representation* and the *backend workflow*. The analysis representation component parses the ONNX model and converts it into its internal representation based on ONNX nodes, providing a set of classes and interfaces for model analysis, which are universal for all backends. The backend workflow component performs tests on the DNN runtime, utilizing the results and information from the analysis representation to conduct the roofline analysis. The remainder of this section provides a detailed description of each of these parts.

3.2 Analysis representations

The analysis representation contains a number of predefined classes of operators that describe how their FLOP and memory accesses are computed. The ONNX model is transformed into a combination of these operator objects, as well as a set of their tensor information, which forms the *Analyze Representation*. The *Optimized Analyze Representation* can represent the analysis of the optimized model after optimization (e.g. operator fusion) in the backend.

3.2.1 Operator defines. The *Operator defines* are a set of operator classes that correspond to an ONNX Node. They maintain information about operator types, attributes, and input/output tensors, and define rules for predicting FLOP and memory accesses based on this information.

For the prediction of FLOP, the number of each basic computation can be determined by the type, properties, and shape of the input or output tensor of each operator. For basic computation operations (e.g., multiply-accumulate (MAC), add, comparison, etc.), we map them to the theoretical number of FLOP according to the underlying device characteristics[3]. The FLOP of the operators can be further calculated by accumulating the FLOP of all basic operations. For some of the basic computational operations (such as division, logarithm, etc.), the actual FLOP performed on different hardware platforms may vary, but due to their limited share in the whole model, the error is within an acceptable range.

The prediction of the amount of memory access, which is also designed to be platform-independent, will be obtained by a simple formula as shown in Equation 1.

$$\begin{aligned} Memory = &\sum_{p \in params} Size_p + batch\_size \\ &\times (\sum_{i \in input} Size_i + \sum_{o \in output} Size_o) \end{aligned} \tag{1}$$

In addition, some operators need special treatment. For example, convolution operators may have a large stride and a small kernel shape, where not all the data in their input tensor will be loaded. Additionally, operators like *Shape* or *Reshape* do not actually read or copy the tensor's content. These rules are also defined in the *operator defines*.

3.2.2 Analyze Representation. *Analyze Representation* is PRoof's internal representation of the model, which contains a set of objects from the operator classes that correspond to all operators (ONNX nodes) in the model. It also includes information about all tensors, such as their shape (obtained by performing shape inference on ONNX model) and the associated operators.

3.2.3 Optimized Analyze Representation. The *Optimized Analyze Representation* and a special operator define named *_FusedOp* are used to express the model after backend optimization (mainly operator fusion). It will eventually be transformed by layer mapping to a similar structure as the backend layers.

Specifically, *_FusedOp* is a virtual operator to express a fused operator. The *_FusedOp* is generated from a set of existing operators, which represents these operators fused into a single fused operator. The *_FusedOp* maintains a subgraph of these original operators and

[3]For instance, a MAC operation will be treated as 2 FLOP

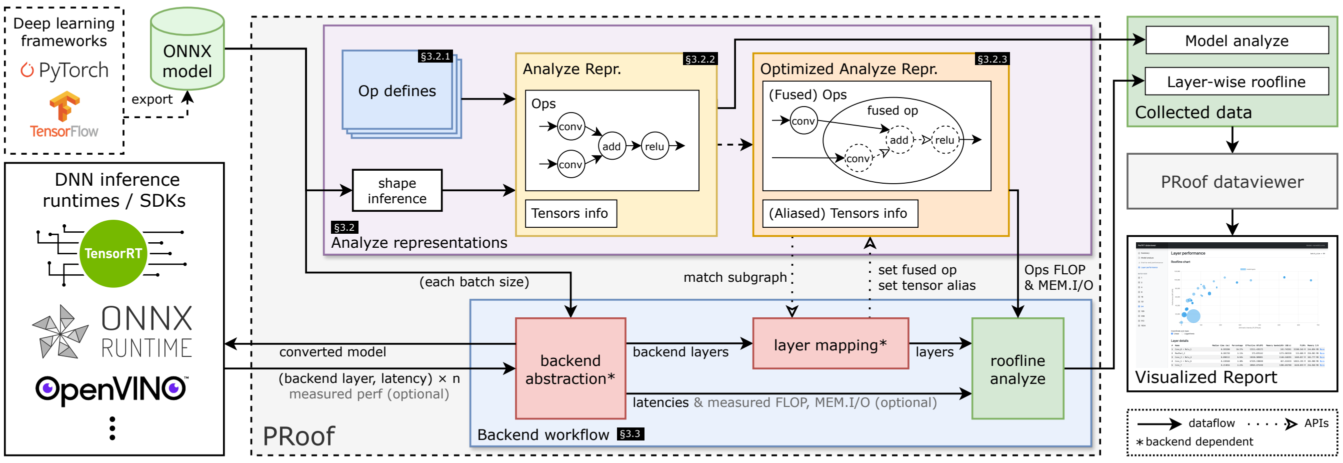


Figure 1: Overview of PRoof.

finds the input and output tensor of the subgraph as the input and output for itself.

The FLOP prediction of the fused operator is obtained by summing the FLOP of all the original operators. For the prediction of memory access amount, we assume that the intermediate tensors in the fused subgraphs will no longer need to be passed through DRAM and will be passed directly on-chip. Thus, only the input and output tensors of the subgraphs need to be considered. This simple but effective strategy can significantly improve accuracy for scenarios containing operator fusion compared to directly summing the memory accesses of unfused operators.

The *Optimized Analyze Representation* is derived from the *Analyze Representation*, which represents the fused model with a set of *_FusedOp* objects. Initially, the *Optimized Analyze Representation* is identical to the Analyze Representation and is converted to the fused model through a series of API calls provided in PRoof.

Since the backend DNN runtime determines the exact way to perform operator fusion, the *Optimized Analyze Representation* needs to obtain information from the runtime, usually using the limited information provided by its built-in profiling functionality, thus implying its relationship within the original model. However, the information given by these runtimes is usually indirect and incomplete. We need to look up and match on the computational graph, or even guess the missing information based on the computational graph and data dependencies. In order to minimize the development cost of supporting each runtime, we have designed several common interfaces for searching and matching on computational graphs within the Optimized Analyze Representation, which are utilized in the layer mapping step in the backend workflow.

3.3 Backend workflow

In DNN runtimes, optimizations such as operator fusion will result in a difference between the original layers and the backend layers, where a backend layer may correspond to a single layer or multiple fused layers of the original model. Since the latency information we can obtain from the DNN runtime is associated with the backend layer, the layer-wise roofline analysis will also be performed at the granularity of the backend layer.

To backtrack from the performance information of the backend layer to the original model layers, we need to obtain the mapping of the backend layer to the model layer. Specifically, the backend runtime needs to provide both latencies at the backend layer granularity and the direct or indirect mapping of the backend layer and the model layers. According to our observation, common DNN runtimes such as TensorRT, OpenVINO and ONNX Runtime can provide such information.

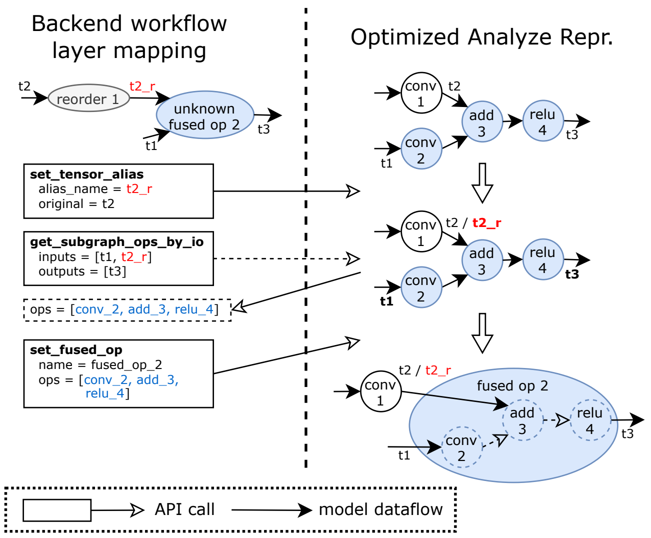
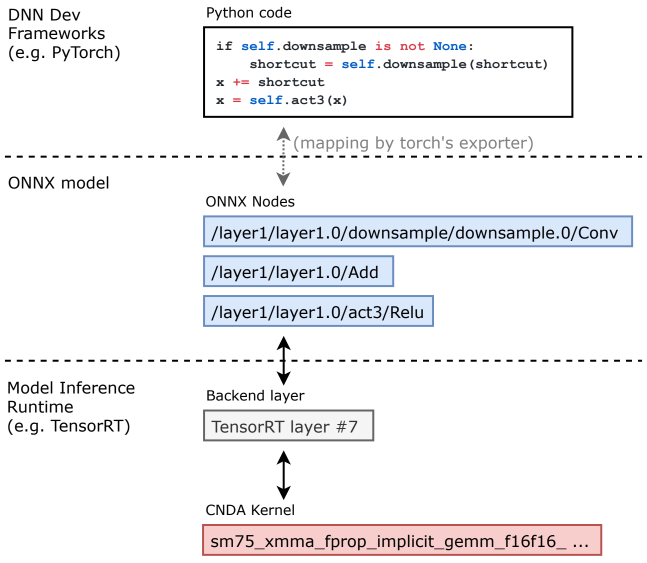


Figure 2: An example of layer mapping. The left side shows a part of the backend layers and the API calls in mapping. The right side shows the transformation process for the part of model representation in Optimized Analyze Representation.

The backend workflow mainly consists of *backend abstraction*, *layer mapping*, and *roofline analysis*. The *backend abstraction* can abstract different DNN runtimes into a unified interface, *layer mapping* leveraging interfaces from *Optimized Analyze Representation* and deriving mapping information. Finally *backend abstraction* combines the above results and predicts the FLOP and memory access amount for the backend layers if needed (when measured performance metrics are not available). The backend abstraction and layer mapping are specified for each DNN runtime, while the other parts are generic.

Layer mapping can determine the mapping of the backend layer to the layers in the original model (a.k.a., ONNX nodes) and transform the *Optimized Analyze Representation* into a structure equivalent to the fused model in the DNN runtime. Specifically, layer mapping uses the information about the backend layer given in the DNN runtime to obtain the mapping, so there are different implementations for each DNN runtime.

Figure 2 illustrates an example of a typical mapping process for a part of the backend layer for ONNX Runtime, which contains an additional backend layer (*reorder_1*) inserted by the DNN runtime. This layer transforms the original tensor *t2* into *t2_r* (usually a datatype or format conversion), and a backend layer (*fused_op_2*) fused from the original operators (*conv_2, add_3, relu_4*). The mapping process leveraging the 3 interfaces on *Optimized Analyze Representation*: *get_subgraph_ops_by_io* to find the subgraph by its input and output tensor; *set_tensor_alias* and *set_fused_op* to apply the transform on itself, the final result is consistent with the structure of the backend layer, its equivalence with operators seen by vendor profiling tools and can be linked to obtain metrics in real measurement mode, but it maintains the composite relationships of the original model layers.


DNN Dev Frameworks (e.g. PyTorch)
Python code
(mapping by torch's exporter)
ONNX model
ONNX Nodes
/layer1/layer1.0/downsample/downsample.0/Conv
/layer1/layer1.0/Add
/layer1/layer1.0/act3/Relu
Model Inference Runtime (e.g. TensorRT)
Backend layer
TensorRT layer #7
CNDA Kernel
sm75_xmma_fprop_implicit_gemm_f16f16_ …

```
if self.downsample is not None:
    shortcut = self.downsample(shortcut)
x += shortcut
x = self.act3(x)
```

Figure 3: An example of some DNN layers of ResNet-50 in the full software stack.

Additionally, if the runtime backend further lowered the backend layer to more low-level operations, like CUDA kernels, which is the target of the vendor profiling tool and its mapping information needs to be maintained as well. This part usually relies on other profiling tools from vendors, such as Nsight System. Finally, we maintained the full stack mapping information of the DNN model, shown as Figure 3. This mapping is bidirectional, allowing us to correlate model design details with their hardware performance metrics.

4 EVALUATION

4.1 Experimental setup

We have selected hardware platforms for common hardware architectures in a variety of common scenarios, covering Data center, Desktop, and Edge scenarios, and several different architectures such as NVIDIA GPUs, x86 CPUs, ARM CPUs, and a mobile NPU in the latest generation Intel Core Ultra laptop CPU (Meteor Lake), as shown in Table 2. For performance metrics, we use the vendor's hardware profiling tool on Data center GPU and Desktop GPU, while using the analytical model for others. For each hardware, we will choose a batch size and data type that is reasonable and fully utilizes the hardware.

For model evaluation, we have selected some models commonly adopted from data centers to edges, which include a classical CNN model (ResNet [9]), lightweight or modified CNN models (EfficientNet [24, 25], MobileNet [10, 23], ShuffleNet [16]), Transformer CV models (ViT [7], Swin [15]), a modern MLP CV model (MLP-Mixer [26]), a Transformer NLP model (BERT [6]), and a Diffusion model (UNET of Stable diffusion [22]). They are all exported as ONNX models via PyTorch [19], as shown in Table 3, where we report the theoretical GFLOP at bs=1 that are obtained from PRoof's analytical model. Our modified version of ShuffleNetV2 x1.0 model (#14) is also included, which will be discussed in Section 4.5.

4.2 Accuracy of prediction method

To verify the accuracy of PRoof's predicted FLOP and memory access bytes, we compared it on the NVIDIA A100 with the measured values obtained via NVIDIA Nsight Compute (NCU), on the 5 most representative models. The evaluation results are shown in Table 4. In summary, the prediction error is within an acceptable range as an alternative method, resulting in substantially lower profiling time (a few seconds total).

During the experiments, we discovered that NCU incorrectly calculated FLOP on Tensor Cores, resulting in an approximate integer multiple difference from the predicted values. This issue with NCU was confirmed by the official staff on the NVIDIA official forums[4]. Specifically, NCU calculates FLOP based on the number of `HMMA.` or `IMMA.` instructions executed on the Tensor Cores. However, different instructions on different GPU architectures perform varying numbers of FLOP, while NCU uses a fixed number of 512. This calculation is only correct for the `HMMA.884.F32.F32` instruction on the Volta architecture. For correction, we refer to some existing work on Tensor Cores analysis and reverse engineering [21] to determine the correct number of FLOP per instruction based on the specific GPU architecture and kernel type, which is further multiplied by the count of HMMA/IMMA instructions from NCU for correct total numbers of FLOP.

[4]https://forums.developer.nvidia.com/t/227563/6

Table 2: Hardware for evaluation.

Hardware	Scenarios	Runtime
NVIDIA A100 PCIE-40GB	Data center GPU	TensorRT 8.6.1
NVIDIA RTX 4090	Desktop GPU	TensorRT 8.6.1
Intel Xeon Gold 6330	Datacenter CPU	ONNX Runtime 1.15.0
NV. Jetson Xavier NX	Edge GPU	TensorRT 8.4.1
NV. Jetson Orin NX 16GB	Edge GPU	TensorRT 8.5.2
Raspberry Pi 4B	Edge CPU	ONNX Runtime 1.14.1
NPU 3720 (in Intel Ultra9 185H)	Mobile NPU	OpenVINO 2024.0.0

Table 3: Models for evaluation.

#	Model name	Type	ONNX Nodes	Params (M)	GFLOP*
1	DistilBERT base	Trans.	435	67.0	48.718
2	Stable Diffusion	Diffu.	5343	859.5	4747.726
3	EfficientNet B0	CNN	239	5.3	0.851
4	EfficientNet B4	CNN	476	19.3	3.209
5	EfficientNetV2-T	CNN	487	13.6	3.939
6	EfficientNetV2-S	CNN	504	23.9	6.030
7	MLP-Mixer (B_{16})	MLP	497	59.9	25.403
8	MobileNetV2 0.5	CNN	100	2.0	0.205
9	MobileNetV2 1.0	CNN	100	3.5	0.621
10	ResNet-34	CNN	89	21.8	7.338
11	ResNet-50	CNN	122	25.5	8.207
12	ShuffleNetV2 x0.5	CNN	584	1.4	0.084
13	ShuffleNetV2 x1.0	CNN	584	2.3	0.294
14	Shuf. v2 x1.0 mod	CNN	156	2.8	0.434
15	Swin tiny (P_4 W_7)	Trans.	1465	28.8	9.133
16	Swin small (P_4 W_7)	Trans.	2839	50.5	17.723
17	Swin base (P_4 W_7)	Trans.	2839	88.9	31.183
18	ViT tiny (P_4 W_7)	Trans.	786	5.7	2.558
19	ViT small (P_4 W_7)	Trans.	786	22.1	9.298
20	ViT base (P_4 W_7)	Trans.	786	86.6	35.329

We tests the models on NVIDIA A100 with FP16 datatype and a batch size of 128. As shown in Table 4, *Memory* indicates the size of the memory access in bytes. *Prof. time* indicates the additional time required for NCU to perform profiling, which is negligible for our analytical model. The *Diff. from NCU* indicates the percentage deviation of our predicted values relative to the NCU. The inaccuracy of the memory access amount is small, which proves our assumptions about memory access behavior in the DNN workload.

While some models have a higher difference in FLOP, this is because the FLOP in our analytical model has a variance of semantics with FLOP in NCU. The former may be called Model FLOP, which only counts the calculations required to accomplish the model inference. The latter may be called Hardware FLOP, which includes implementation overhead such as padding and memory address calculation. The Model FLOP may be more meaningful for model performance because a higher FLOP/s calculated by Hardware FLOP may not reflect higher real performance. Generally, the Model FLOP could potentially offer additional insights when compared with Hardware FLOP, while also serving as a relatively accurate alternative.

4.3 End-to-end roofline analysis

In this section, we perform an end-to-end roofline analysis of multiple models on different devices to visualize the hardware efficiency of each model on the device and identify possible bottlenecks.

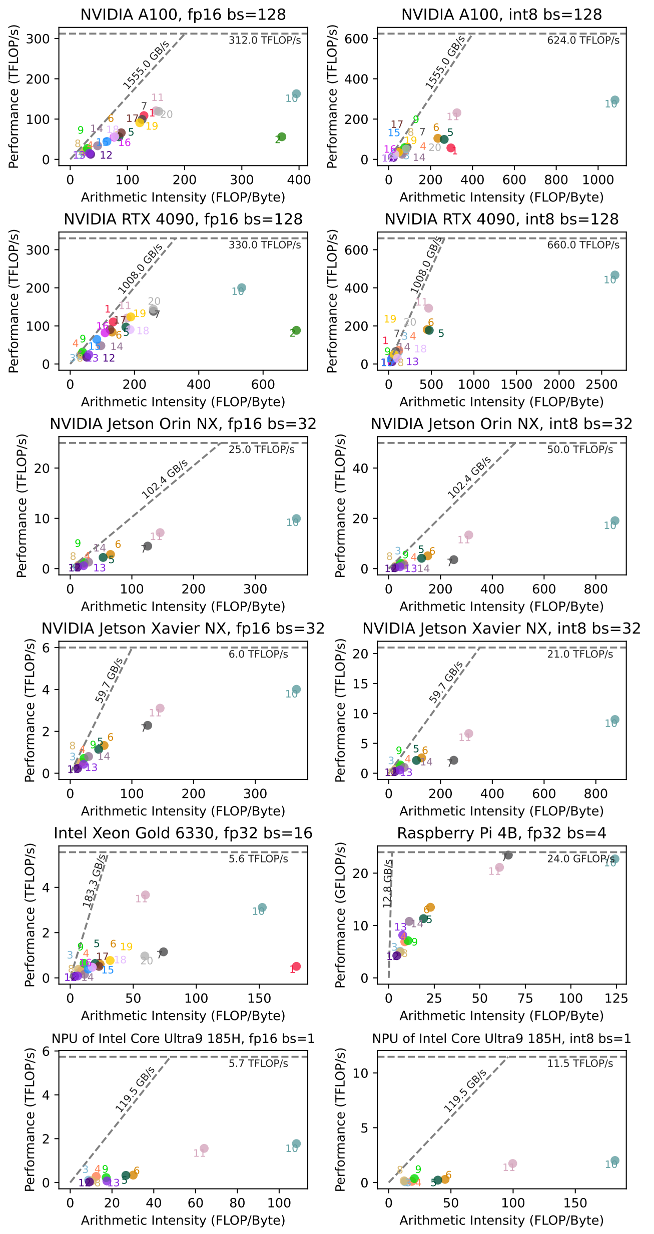


Figure 4: End-to-end roofline analysis for models.

We evaluate all models on data center and desktop platforms, and all models except Transformer and diffusion models on the edge platform because these models are larger and less commonly used on edge devices. On each device, we use its optimal batch size. The results are shown in Figure 4. The numbers beside the data points in the figure indicate the serial numbers of the models in Table 3. [5]

[5]For Stable Diffusion (#2), we run one iteration of its UNET with a latent size of 128x128 and a batch size of 4. It was not tested on CPU or Edge platforms, nor on GPU with int8 datatype due to the failure of TensorRT during the model conversion.

Table 4: Accuracy of FLOP and Memory access prediction as the alternative method.

			Analytical model		NVIDIA Nsight Compute			Diff. from NCU	
Model name	Latency (ms)	Nodes	GFLOP	Memory (MB)	GFLOP	Memory (MB)	Prof. time (s)	FLOP	**Memory**
EfficientNetV2-S	16.644	504	771.794	11669.419	962.575	11820.696	1327	-19.82%	**-1.28%**
MobileNetV2 1.0	3.894	100	79.452	3521.010	104.492	3474.114	343	-23.96%	**+1.35%**
ResNet-50	8.918	122	1050.435	7052.921	1072.227	7150.855	395	-2.03%	**-1.37%**
Swin small	43.935	2839	2268.528	28897.395	2414.215	31431.407	1930	-6.03%	**-8.06%**
ViT tiny	5.308	786	327.382	4059.092	298.195	3826.516	483	+9.79%	**6.08%**

Based on these results, we are able to make some interesting analyses. First of all, powerful GPUs like NVIDIA A100 and RTX 4090 have high performance due to their Tensor Cores that accelerate matrix operations. However, when actually performing model inference, only a small number of models have achieved FLOP/s rates exceeding half of the peak FLOP/s. Many of the models in the lower left corner of the roofline chart are notably limited by memory bandwidth.

NVIDIA Jetson Orin NX, the successor to Xavier NX, approximately doubles the peak FLOP/s and memory bandwidth, resulting in significantly higher performance on DNNs. However, the power consumption is also higher (25W MAX compared to 15W MAX for the Xavier NX). These NVIDIA Jetson devices also have a relatively high peak FLOP/s due to the presence of Tensor Cores.

For CPU platforms such as Intel Xeon Gold 6330 and Raspberry Pi 4B, despite having SIMD instruction sets such as AVX512 and NEON, their theoretical FLOP/s remain low due to the limited number of execution units (FPUs). For the Raspberry Pi 4B, due to the internal AXI bus bandwidth limitation of its BCM2711 SoC, the actual reachable memory bandwidth is only about 5.5 GB/s at maximum, further limiting its inference performance.

Finally, we also conducted tests on the Neural Processing Unit (NPU) within the latest Intel Core Ultra series notebook processor (Meteor Lake), which was launched in Q1 2024. This NPU is commercially known as Intel AI Boost, with an internal name of NPU 3720. Despite utilizing Intel's latest Deep Neural Network (DNN) runtime, OpenVINO 2024, only a small portion of models were able to successfully perform inference on it. Furthermore, its performance significantly deviated from its theoretical value of 5.7 TFLOP/s or 11.5 TOP/s under fp16 or int8 (2048 fp16 MACs or 4096 int8 MACs per cycle @ 1.4 GHz).

4.4 Layer-wise roofline analysis

DNN models usually have complex structures, so for performance analysis, profiling from layer granularity is necessary. The layer-wise roofline analysis can visualize the distribution of efficiency and arithmetic intensity at each layer of the model, thereby guiding the design and optimization of the model from both model and hardware perspectives.

In this section, we show the layer-wise roofline of *ResNet-50*, *Vit tiny*, *EfficientNet B4*, and *EfficientNetV2-T* models on NVIDIA A100 (fp16, batch size 128), as shown in Figure 5 (*we resort to the analytical model method in model (b) because one of the dependencies in our measurement method, the NVIDIA DLProf, crashes during the process).

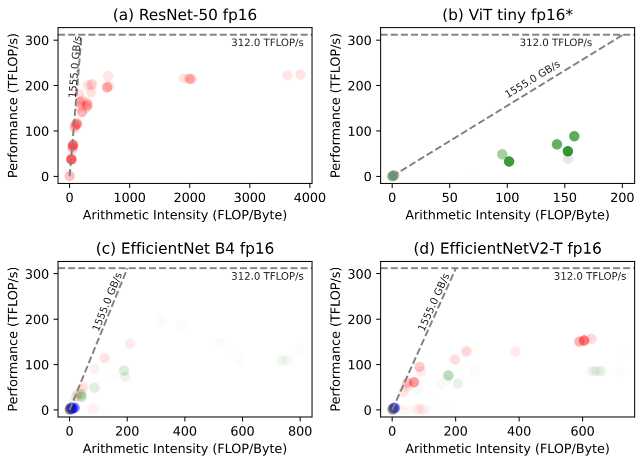


Figure 5: Layer-wise roofline analysis for several models on NVIDIA A100.

In Figure 5, each data point represents a backend layer in the model, and its opacity indicates its share in the model inference, with more opacity indicating a larger share. In Figure 5(a), we can see that *ResNet-50* consists of many backend layers with different arithmetic intensities. Some of these layers are limited by the memory bandwidth. However, in general, its relatively time-consuming backend layers are distributed in an interval of higher arithmetic intensities and have relatively high FLOP/s. This explains the relatively high overall FLOP/s of *ResNet-50* in the previous section.

The *Vit tiny* depicted in Figure 5(b) is a Transformer model with moderate overall hardware efficiency. The backend layers, which contain the *MatMul* (matrix multiplication) operator, are marked in green. These correspond to the matrix multiplication in the attention and linear layers of the model, exhibiting relatively high arithmetic intensity and FLOP/s.

In Figure 5(c), *EfficientNet B4* is a model with lower hardware efficiency, with 17.242 TFLOP/s. We can observe that it has a large number of backend layers with low arithmetic intensity. On the other hand, *EfficientNetV2-T* in Figure 5(d) has a similar total FLOP but higher hardware efficiency at 37.586 TFLOP. We marked depth-wise convolution in blue, point-wise convolution in green, and other convolutions (including traditional convolution) were marked as red. It can be seen that the low hardware efficiency mainly stems from the depth-wise convolution. In its successor, EfficientNetV2 [25], some combinations of the point-wise and depth-wise convolutions are changed back to traditional convolution, which has higher arithmetic intensity and relatively high hardware efficiency, resulting in better overall hardware efficiency. We specifically corroborate this in our roofline analysis, where the replaced traditional convolution achieved higher FLOP/s due to its higher arithmetic intensity (deeper red point).

4.5 Guiding model design optimization

ShuffleNetV2 is a lightweight CNN network that effectively balances speed and accuracy. However, through the use of PRoof, we have discovered that there is still potential to increase speed without sacrificing accuracy on a data center GPU.

On the NVIDIA A100, while utilizing PRoof's end-to-end profiling, we noticed that the ShuffleNetV2 x1.0 model only reached 12.153 TFLOP/s (with fp16 datatype and a batch size of 2048). This is in contrast to the theoretical performance announced by the vendor, which is 312 TFLOP/s.

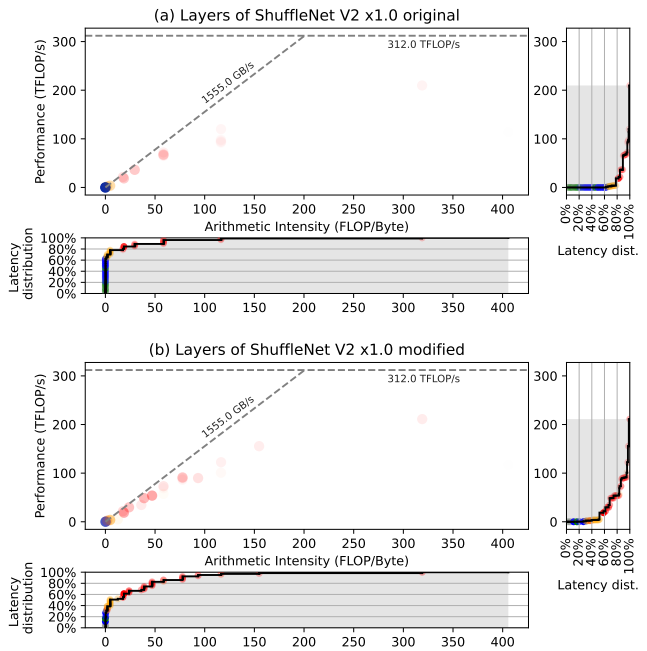


Figure 6: Layer-wise roofline analysis for the original and modified *ShuffleNetV2 x1.0* models (fp16, batch size of 2048). We added the bar charts aside to have a better view of the latency distributions of the model layers along the two axes of the roofline chart since some points overlap in the roofline chart.

We performed a layer-wise roofline analysis on the original model, as shown in Figure 6(a). In this experiment, we utilize the prediction mode of PRoof to obtain metrics that demonstrate its effectiveness. The depth-wise convolution layers are marked in orange, while the other convolution layers (mainly point-wise) are marked in red. They contribute the majority of the model's FLOP but only account for about 40% of the total latency. The most time is taken by the transpose layer (blue) and the data-copy layer (green), which are mainly produced by the Shuffle operation in ShuffleNetV2.

The Shuffle operation is the main idea of the ShuffleNet family. It is used to reduce FLOP by only performing convolution on half of the activation channels in each block. Then, the Shuffle operation is used to re-order the channels to perform convolution on different channels in the next block. However, the Shuffle operation is implemented by the *Transpose* and data copy layers during model runtime, and these operations are inefficient and memory-intensive.

Considering the NVIDIA A100 is a powerful GPU with a high peak FLOP/s but rather limited memory bandwidth, the idea for optimization is to trade off FLOP savings in exchange for a lower amount of memory access. Using PRoof, we can analyze the performance of Conv layers within different types of base blocks at various stages of the original model design. This allows us to determine the most effective positions of Conv layers to modify. We modified the model design of the ShuffleNetV2's non-downsampling basic block while keeping the downsampling block unchanged. We removed the Shuffle operation and doubled the input channel and output channel numbers at the first and last point-wise convolutions in these blocks, to cover all the channels without the Shuffle operation. Additionally, we appended an Add operation to the bottom for residual connections, which is implicitly accomplished in the original Shuffle operation, as shown in Figure 7.

These knowledge of the original model ensures that our modifications do not lead to issues with the model's effectiveness. Compared to using NCU alone, the NCU can only obtain a series of poorly recognized kernel names, it allows for only rough classification and statistics. This limitation makes it impossible to use it to perform optimizations such as the one in this case.

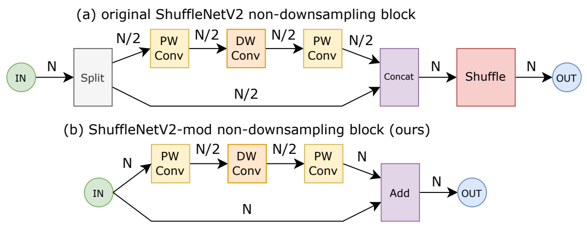


Figure 7: Our optimizations to ShuffleNetV2.

We re-trained the modified model (as well as the original model under the same configuration). The modified model's size and accuracy are slightly higher than the original one (due to the increased Conv channel number). Although the FLOP is higher, the modified model has a 64.45% increased throughput (30.1 ms vs 49.5 ms) at a batch size of 2048 (the batch size reached maximum throughput for both models). It is also much faster in smaller batch sizes for latency-sensitive applications, as shown in Table 5.

The layer-wise roofline analysis for the modified model is shown in Figure 6(b). Transpose and data copy layers take significantly less time. Additionally, since the model is under a memory bottleneck, the modified convolution layer with higher FLOP does not result in performance degradation.

4.6 Guiding hardware optimization

In this section, we performed hardware optimization to maximize the specific DNN workloads under a limited power budget by tuning

Table 5: Effectiveness of the modified *ShuffleNetV2 x1.0* model.

Model	Params (M)	ImageNet Top-1 Accuracy (%)	Batch size	GFLOP	Latency (ms)	Throughput (images/s)	Achieved GFLOP/s	Achieved memory bandwidth (GB/s)	Speedup
Original	2.271	68.9%	1	0.294	0.528	1894	556.759	34.026	-
			128	37.630	3.2479	39410	11585.843	530.486	-
			2048	602.079	49.543	41338	12152.612	555.062	-
Modified	2.804	70.1%	1	0.434	0.380	2632	1141.680	54.855	**1.39x**
			128	55.578	2.184	58608	25451.294	790.130	**1.49x**
			2048	889.255	30.126	67981	29518.047	895.042	**1.64x**

Table 6: Achieved roofline peak (measured by PRoof running an assembled pseudo ONNX model including a series of MatMul and memory copy operators of different sizes in TensorRT) and power at different clock speeds.

#	GPU clock (MHz)	Memory clock (MHz)	FLOP/s (T)	Memory BW (GB/s)	Power (W)
1	918	3199	13.620	87.879	23.6
2	918	2133	13.601	62.031	21.3
3	510	3199	7.433	54.002	15.7
4	510	2133	7.426	53.017	13.6
5	510	665	7.359	15.177	11.5

the clock speed for different hardware components with the help of PRoof.

For the edge scenario, due to limitations such as device cooling and battery life, we need to keep the hardware system running at a limited power, possibly below its maximum support. The question is how to maximize the performance of the workload within a limited power budget. We performed the evaluation on the NVIDIA Jetson Orin NX with a 15W power budget (25W max). The workload is *EfficientNetV2-T* with fp16 datatype and a batch size of 128, on TensorRT.

To limit the power, we can limit the maximum clock speed for components on Jetson hardware using the `nvpmodel` command. We have identified the critical components as GPU cores and EMC (memory clock), while other components (such as the CPU) are not the bottleneck for our workload. Therefore, we lower their clock speed to conserve power.

Firstly, we performed a roofline peak test at several clock frequencies to establish a baseline, as shown in Table 6. We observed that lowering the GPU clock resulted in a decrease in both the Achieved FLOP/s and Memory Bandwidth (#1 vs #3), and lowering the memory clock led to a decrease in the Achieved Memory Bandwidth (#1 vs #2).

Then we performed a layer-wise analysis for the target model under clock speed #1 (maximum performance), shown as Figure 8. The depth-wise convolution (marked in yellow) and point-wise convolution (red) take about 70% of the latency, while other point-wise operators and *ReduceMean* operators are marked in purple. Considering that the GPU clock is more sensitive to the performance of this model since the computationally intensive layer takes up half of the latency, we will first select an optimal memory clock speed, and then tune the GPU clock within the power budget.

Memory clock speeds available to be selected are 3199, 2133, and 655 MHz (we skip 204 MHz which is not useful), which will

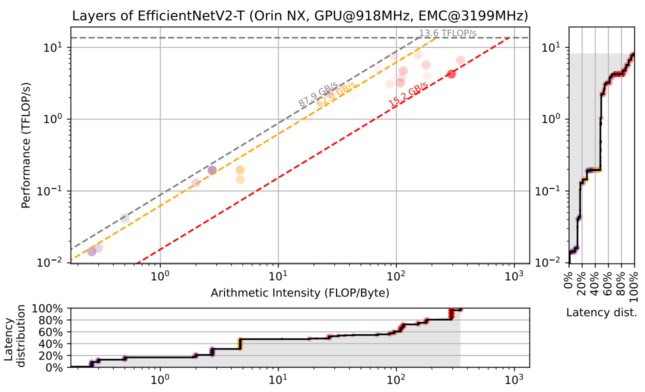


Figure 8: Layer-wise roofline analysis for *EfficientNetV2-T* (fp16, batch size of 128).

limit memory bandwidth to 87.9, 62.0, and 15.2 GB/s. We also added the last two of them as bandwidth lines to Figure 8, in yellow and red. We can observe that if we lower the memory clock speeds from 3199 to 2133 MHz, only a small amount of layers (above the yellow line) will be affected slightly, which will only lead to a small performance drop and it's a worthwhile trade-off to the power. But if we lower from 2133 to 655 MHz, most layers (above the red line) will be affected massively and the trade-off is not worth it.

Yet, we select the memory clock speeds of 2133 MHz. Then, we could perform a simple binary search for the GPU clock just below the power budget of 15W within a few tests, which is 612 MHz. The latency and power consumption retrieved from the *jtop* tool are 320.1 ms and 14.7W, respectively, which are faster and more efficient than the stock power profiles within the power budget.

Detailed results are shown in Table 7. The CPU cores have been grouped into 2 clusters with individual clock settings or can be turned off. The stock profile named "15W" (#2) uses a different value of 252 for an undocumented GPU power setting called `TPC_PG_MASK`, and we found it to be less efficient than other power profiles where it is set to 240. We also tested other profiles for comparison, which verified our analysis of memory clock speeds (#4 - #6), and our selection of clock speeds is optimal (#7 - #9).

5 CONCLUSION

This paper presents PRoof, a framework for profiling DNN models from a hardware perspective. PRoof provides end-to-end and layer-wise roofline analysis of the DNN models within DNN inference runtime. Within this framework, we propose a scheme to backtrack

Table 7: Performance and power of *EfficientNetV2-T* under different power profiles.

Profile	#	Clock speed (MHz)			Latency (ms)	Power (W)
		CPU	GPU	EMC		
stock *"MAXN"*	1	729/729	918	3199	211.4	23.2
stock *"15W"**	2	729/off	612	3199	514.5	13.6
stock *"25W"*	3	729/729	408	3199	462.1	14.2
Comparisons	4	729/off	918	3199	211.3	22.5
	5	729/off	918	2133	232.7	19.2
	6	729/off	918	665	568.0	12.4
	7	729/off	612	3199	317.5	16.6
	8	729/off	612	665	584.6	10.9
	9	729/off	510	3199	378.1	15.1
Optimal (ours)	10	729/off	612	2133	**320.1**	**14.7**

the backend layer to the original model layer after optimization and fusion by DNN runtimes, as well as an analytical model for predicting FLOP and memory access. We have implemented PRoof support on TensorRT, OpenVINO, and ONNX Runtime and conducted experiments on 20 models across 7 different hardware platforms. We validate the accuracy of the analytical model as an alternative method, provide unique insights on hardware and model design through the final end-to-end and layer-wise roofline analysis, and present case studies. These results demonstrate PRoof's ability to assist in model design and hardware tuning.

For future work, we plan to add more model inference runtime support to the PRoof implementation in order to cover more platforms. Furthermore, with the increasing popularity of large-scale inference, we aim to investigate the adaptation of PRoof to distributed environments.

ACKNOWLEDGMENTS

This work is supported by National Key Research and Development Program of China (Grant No. 2023YFB3001801), National Natural Science Foundation of China (No. 62322201, 62072018, U23B2020 and U22A2028), and the Fundamental Research Funds for the Central Universities (YWF-23-L-1121 and JKF-20240198). Hailong Yang is the corresponding author.

REFERENCES

[1] Intel Corporation. 2024. Intel VTune Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html
[2] NVIDIA Corporation. 2024. DLProf User Guide. https://docs.nvidia.com/deeplearning/frameworks/dlprof-user-guide/index.html
[3] NVIDIA Corporation. 2024. *Nsight Systems | NVIDIA Developer.* https://developer.nvidia.com/nsight-systems
[4] NVIDIA Corporation. 2024. NVIDIA Nsight Compute | NVIDIA Developer. https://developer.nvidia.com/nsight-compute
[5] NVIDIA Corporation. 2024. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt
[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[7] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. arXiv:2010.11929 [cs.CV]
[8] The Linux Foundation. 2024. Open Neural Network Exchange. https://onnx.ai/
[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 770–778.
[10] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).
[11] Hanpeng Hu, Chenyu Jiang, Yuchen Zhong, Yanghua Peng, Chuan Wu, Yibo Zhu, Haibin Lin, and Chuanxiong Guo. 2022. dPRO: A Generic Performance Diagnosis and Optimization Toolkit for Expediting Distributed DNN Training. *Proceedings of Machine Learning and Systems* 4 (2022), 623–637.
[12] Intel. 2024. OpenVINO™ toolkit: An open source AI toolkit that makes it easier to write once, deploy anywhere. https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit/overview.html
[13] Cheng Li, Abdul Dakkak, Jinjun Xiong, and Wen-mei Hwu. 2020. Benanza: Automatic μBenchmark Generation to Compute" Lower-bound" Latency and Inform Optimizations of Deep Learning Models on GPUs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 440–450.
[14] Cheng Li, Abdul Dakkak, Jinjun Xiong, Wei Wei, Lingjie Xu, and Wen-mei Hwu. 2020. XSP: Across-stack profiling and analysis of machine learning models on GPUs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 326–327.
[15] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. 2021. Swin Transformer: Hierarchical Vision Transformer Using Shifted Windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV).* 10012–10022.
[16] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. 2018. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European conference on computer vision (ECCV).* 116–131.
[17] Tianhao Miao, Qinghua Wu, Ting Liu, Penglai Cui, Rui Ren, Zhenyu Li, and Gaogang Xie. 2022. MD-Roofline: A Training Performance Analysis Model for Distributed Deep Learning. In *2022 IEEE Symposium on Computers and Communications (ISCC).* IEEE, 1–8.
[18] Microsoft. 2024. Optimize and Accelerate Machine Learning Inferencing and Training. https://onnxruntime.ai/
[19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).
[20] Zheng Qin, Zhaoning Zhang, Xiaotao Chen, Changjian Wang, and Yuxing Peng. 2018. Fd-Mobilenet: Improved Mobilenet with a Fast Downsampling Strategy. In *2018 25th IEEE International Conference on Image Processing (ICIP).* 1363–1367. https://doi.org/10.1109/ICIP.2018.8451355
[21] Md Aamir Raihan, Negar Goli, and Tor M Aamodt. 2019. Modeling deep learning accelerator enabled gpus. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* IEEE, 79–92.
[22] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition.* 10684–10695.
[23] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 4510–4520.
[24] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning.* PMLR, 6105–6114.
[25] Mingxing Tan and Quoc Le. 2021. Efficientnetv2: Smaller models and faster training. In *International conference on machine learning.* PMLR, 10096–10106.
[26] Ilya O Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, et al. 2021. Mlp-mixer: An all-mlp architecture for vision. *Advances in neural information processing systems* 34 (2021), 24261–24272.
[27] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
[28] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (apr 2009), 65–76. https://doi.org/10.1145/1498765.1498785
[29] Charlene Yang, Yunsong Wang, Thorsten Kurth, Steven Farrell, and Samuel Williams. 2021. Hierarchical roofline performance analysis for deep learning applications. In *Intelligent Computing: Proceedings of the 2021 Computing Conference, Volume 2.* Springer, 473–491.
[30] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. 2020. Daydream: Accurately estimating the efficacy of optimizations for {DNN} training. In *2020 USENIX Annual Technical Conference (USENIX ATC 20).* 337–352.