ModelGauge: Inference Profiling of Deep-Learning Models

Calvin B. Gealy[*†], David Langerman[†], Alan D. George[*†]
[*]*Electrical and Computer Engineering, University of Pittsburgh*, Pittsburgh, United States
[†]*NSF Center for Space, High-Performance, and Resilient Computing (SHREC)*, Pittsburgh, United States
{c.gealy, alan.george}@pitt.edu, david.langerman@nsf-shrec.org

***Abstract*—Identifying trends in on-device performance between different deep-learning models is often challenging given the variety of models published and the different devices used in deployment. ModelGauge is a proposed solution that reports the latency, memory, and bandwidth behavior of many different inference configurations. Utilizing ONNX Runtime for CPUs and NVIDIA TensorRT for GPUs, as well as the standardized ONNX model format for model definitions, ModelGauge can easily profile many architectures, allowing deployment engineers easy access to statistics about inference performance. To demonstrate the utility of the tool, we compare 32 different ONNX model definitions and their on-device scaling behavior on an ARM Cortex-A76 embedded CPU, AMD EPYC 9374F server CPU, NVIDIA Jetson Orin Nano embedded GPU, and NVIDIA A100 server GPU. For this study, Pearson correlation is used to show the linear relationship between a metric and a device measurement to characterize behavior and show the utility of bulk data collection. When comparing the number of floating-point operations in a model to the latency for single-image batch inference, the Pearson correlation is highest on the ARM Cortex-A76 at 0.990 and lowest on the highly parallel NVIDIA A100 at 0.388. Across all devices and models tested, the linear trend and Pearson correlation between the number of parameters in a model and the memory is consistently greater than 0.9. Additionally, we propose a new metric found by analyzing the data translation lookaside buffer load miss count and the device latency to help indicate models not using a significant amount of the device. Overall, ModelGauge is useful for gathering statistics about a variety of models across compute at many scales.**

***Index Terms*—deep learning, benchmarking, machine learning, high-performance computing**

I. Introduction

As the number of deep-learning (DL) architectures continues to increase across different problem domains, it is becoming increasingly challenging to determine which model is likely to perform best for a given task. Beyond accuracy, it is also important to verify that potential models reach latency or memory targets for on-device inference given a specific deployment platform. Therefore, a tool that can easily compare many different architectures and report their performance statistics is valuable in modern DL research and deployment.

We propose ModelGauge as a solution to this problem. ModelGauge is built to profile the latency, memory, and bandwidth behavior of DL models across CPU and GPU devices. By providing a single tool to measure these three performance constraints, model practitioners can easily query many architectures to gather additional data and insight into which models will meet their performance targets.

This research was supported by SHREC industry and agency members and by the IUCRC Program of the National Science Foundation under Grant No. CNS-1738783.

ModelGauge utilizes two high-performance inferencing frameworks, ONNX Runtime for CPU and NVIDIA TensorRT for GPU, to provide a realistic analog for runtimes likely to be used in deployment. Additionally, the tool utilizes other standard programs such as Valgrind Massif and NVIDIA Nsight Systems to improve the accuracy of the values measured. ModelGauge will measure any ONNX file in a target directory and report the requested statistics.

To verify the utility of the tool, we compare 32 different ONNX models on four devices: an ARM Cortex-A76 embedded CPU, an AMD EPYC 9374F server CPU, a NVIDIA Jetson Orin Nano embedded GPU, and a NVIDIA A100 server GPU. The 32 models are comprised of seven different base architectures with different variations in model scale and input size. We compare the linear correlation between the floating-point operations (FLOPs) and latency as well as the parameters and memory to showcase the utility of gathering data across a variety of models on a given system. Additionally, we propose a new computational metric for CPUs based on the bandwidth results. This new metric, titled the "Small Model Inefficiency Rate," could help engineers easily identify models that are likely too small on the given device. Given the potential benefits to the DL deployment community, we plan to open-source our tool. Overall, in this research we:

- Propose ModelGauge as an easy tool for gathering the on-device performance of many models on CPU and GPU systems
- Validate ModelGauge using four different devices at opposing compute scales and compute types
- Compare the scalability of 32 ONNX models on the four target devices
- Discover server CPU performance trends more similar to GPU systems than embedded CPU systems
- Propose a new metric for checking if a CPU device is underutilized

II. Background and Related Works

In this section, we will outline some of the related research in the field of DL benchmarks. Most benchmarks are designed to better quantify a system for comparison with others. ModelGauge instead focuses on easy analysis of many different models on a chosen platform. Additional background on the ONNX format and deployment runtimes is included.

A. DL Inference Benchmarks

The field of benchmarks for DL models is wide and continues to grow. Perhaps the most important effort in inference benchmarking is the MLPerf Inference suite [1]. This benchmark is designed to be reproducible across varying systems by quantifying representative workloads and provides rules to fairly compare hardware setups and software packages. This flexible tool has become a popular way to compare the overall performance of a system and its accompanying software. The benchmark fixes the model architectures which are profiled, allowing for the comparison of systems. Similar to MLPerf Inference, MLPerf Mobile Inference is designed to benchmark the performance of ML models on mobile systems, though it will run on non-smartphone devices as well [2]. Devices typically benchmarked with this framework include mobile phones and laptops.

Other smaller projects tend to focus on particular test cases. Kustikova et al. created an inference benchmark that is agnostic to the training framework and focuses on inference for Intel platforms utilizing OpenVINO [3]. They analyze devices by measuring latency, and therefore processing framerate, at multiple batches. EDLAB is another benchmark tool for measuring model inference performance on a variety of edge systems [4]. Specifically, it measures performance on an Intel Neural Compute Stick, NVIDIA Jetson Xavier, and Google Edge TPU. This framework transforms a TensorFlow model into the appropriate type for the given device-specific inference framework. It records device latency, accuracy, throughput, power, and floating-point operations per second.

Finally, Zhang et al. create a benchmark to study how deep-learning hardware and libraries affect inference performance on mobile (i.e., phone) processors [5]. They analyze the results across CPU, GPU, and DSP subprocessors in device system-on-chips. They also compare performance across different inference libraries. Overall, they find large fragmentation among model types, libraries, and devices. They find that library selection can be one of the most impactful decisions for model inference speed.

B. ONNX

ONNX is an open standard for defining models. It is designed to enable transportability between systems and runtimes [6]. Given a DL model in a training framework such as PyTorch or TensorFlow, an ONNX definition of the model can easily be exported to a serialized file that includes details on model operators and contains the trained weights. Downstream inference pipelines can use this standardized definition as an input for their runtime framework.

C. Deployment Runtimes

There are two deployment runtimes utilized in this research. ONNX Runtime is a high-performance library that can utilize many accelerators but is used for deployment on the CPU systems in this research. NVIDIA TensorRT is a high-performance framework for deployment on NVIDIA GPUs. Both these runtimes are detailed below.

1) ONNX Runtime: ONNX Runtime is a model-inference framework backed by Microsoft [7]. It is designed to perform inference on many accelerator types, including GPUs and FPGAs, while also supporting operations with a performant CPU implementation. In this research, only these CPU implementations will be utilized for high-performance inference on the CPU test platforms.

2) NVIDIA TensorRT: NVIDIA TensorRT is designed specifically for high-performance inference on NVIDIA GPUs [8]. This framework can convert models from other formats, including ONNX, and generate performant TensorRT Engines. The Engine is constructed using profiling from the specific device to determine which kernel implementation performs the best on that given platform. TensorRT Engines can be serialized and saved for future use on that device as generating the engine is non-deterministic and can be slow due to the generation-time profiling.

III. Approach

ModelGauge measures the latency, memory, and bandwidth behavior of models. Latency is often important because many applications require models to reach real-time constraints, and therefore, measuring the latency is required. Memory is a finite resource on the device and must be shared by the applications running, including the model inference frameworks. Finally, bandwidth results help to provide lower-level insight into model behavior and constraints on inference performance.

In this section, we will detail the setup for the ModelGauge tool, including the testing parameters used. Additionally, we will highlight the test platforms examined in this study. Next, we will showcase the models tested to demonstrate the data that can be gathered from ModelGauge.

A. ModelGauge

ModelGauge is developed in Python. Command line options allow the user to choose which folder they would like to search. This folder and any subdirectories are searched for '.onnx' files representing the models to be analyzed. Additionally, the user can choose between the two runtimes, as well as name the device they are testing on for saving the results. Results are saved individually for each model in a results folder.

As the parameters for testing slightly differ between the CPU and GPU, they will be annotated separately in the following subsections. Please note that many of these parameters are also configurable through a YAML file. The same base ModelGauge script can be used on the different systems, and runtime dependencies are not loaded until the user selects their runtime type.

Along with the '.onnx' file for a model, the tool will look for a '.info' file with the same name. This file can contain any data stored in JSON format that will then be concatenated to the result file for that model. The info file is useful for storing model information such as version, parameter count, accuracy, or FLOP count.

1) CPU Model Profiling: Latency is relatively simple to measure on the CPU. Simply profiling the wall-clock time of inference on a model allows us to understand the overall latency of model processing. In Python, this can be performed with the built-in `time` package, specifically, the `time.perf_counter()` function. First, the inference is run and not timed for 20 cycles to prime any caches in the device. Then, latency is averaged over 100 inference cycles.

Memory is more complicated to measure. Memory itself can be a loose term due to complex and often obfuscated components of the memory subsystem such as virtual memory and paging. Additionally, Python libraries themselves are often written in C/C++, meaning Python's own memory management may not be aware of all allocated memory. Therefore, to measure the memory for inference, Valgrind Massif is utilized [9]. Massif measures memory accesses and allocation over time. While some memory may still not be fully measured, we have determined Massif to be the best tool given its long history and support. Additional discussion on memory can be found in Section V. Python is set to use a standard malloc function via an environment variable to help with analysis via Massif. A simple Python script that performs inference 10 times is analyzed via Massif. The maximum memory, including allocations on the heap and the stack, is recorded. Memory measurements are averaged across 10 samples. Larger sample counts are time-consuming with Massif.

Bandwidth, or caching behavior, is measured by the Linux `perf` tool [10]. This tool can capture details about the instruction calls, cache hit and miss rates, and other detailed reports of program performance. The main downside to `perf` is that it requires elevated privileges of the kernel to measure these values. Therefore, bandwidth measurements cannot be gathered on all devices when the kernel is more restricted (e.g., shared computing resources hosted by universities). Measured values in this research include: cpu_cycles, instructions, cache-references, cache-misses, L1-dcache-loads, L1-dcache-misses, branch misses, and dTLB-load-misses. Again, a simple Python script runs the model for 100 inferences. All collected values are averaged across 10 samples. Due to the depth of `perf`, recording additional samples can be time-consuming.

2) GPU Model Profiling: Latency is measured using NVIDIA's `trtexec` program [11]. This program is designed to profile NVIDIA TensorRT engines. It will automatically measure the overall latency of the engine. The minimum warmup for priming the caches is set to one second. Data is then gathered for a minimum of five seconds and at least 100 iterations. `trtexec` then reports statistics about the latency average and standard deviation, which is saved with the other results for the model.

Memory is measured utilizing NVIDIA Nsight Systems [11]. Nsight Systems reports a detailed account of memory transfers and allocations on a GPU subsystem for a given program. This detailed information can then be leveraged to understand memory transfers between the host and the device, as well as memory transfers within the device itself. A TensorRT engine is first generated with `trtexec` and is serialized. This engine is then profiled with Nsight Systems with inference performed for one iteration using `trtexec` to analyze the memory required for loading the model, an input, and an output. Since model generation is nondeterministic, this process from engine generation to analysis is repeated 100 times.

Bandwidth is also measured utilizing NVIDIA Nsight Systems. Using one of the TensorRT engines, the execution of `trtexec` is again profiled for 100 iterations. Note that due to the time to generate multiple engines, only one engine is profiled. Measured behavior includes host-to-device transfers and device-to-device transfers. The count, average bytes, average transfer time, and average bandwidth are collected for each transfer type.

TABLE I: Table highlighting the devices under test. Note that the NVIDIA Jetson Orin Nano has a shared-memory architecture.

Type	Device	Class	System Memory	Device Memory
CPU	4-core ARM Cortex-A76 (Raspberry Pi 5)	Embedded	8 GB	-
	32-core AMD EPYC 9374F (Genoa)	Server	768 GB	-
GPU	NVIDIA Jetson Orin Nano	Embedded	8 GB	
	NVIDIA A100	Server	40 GB	40 GB

B. Test Platforms

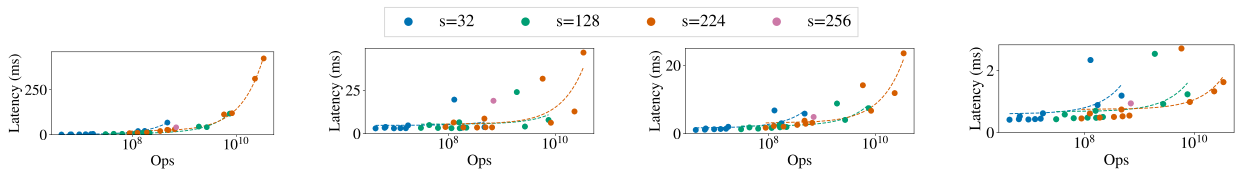
The four test platforms highlighted in the research are shown in Table I. The embedded CPU under test is a quad-core ARM Cortex-A76 processor found in the Raspberry Pi 5. This system features 8 GB of RAM. The server CPU under test is a 32-core AMD EPYC 9374F (Genoa) CPU with 768 GB of system RAM.

The embedded GPU under test is the NVIDIA Jetson Orin Nano. This system has 8 GB of shared memory between the CPU and GPU on the system-on-chip. The server GPU under test is the NVIDIA A100, which has 40 GB of host memory and 40 GB of device memory.

The two server-class systems are housed at the University of Pittsburgh Center for Research Computing. Due to the shared nature of these systems, Linux `perf` could not be run on the AMD EPYC CPU. Therefore, bandwidth results are not included for this device. No additional privileges were required for the GPU measurements.

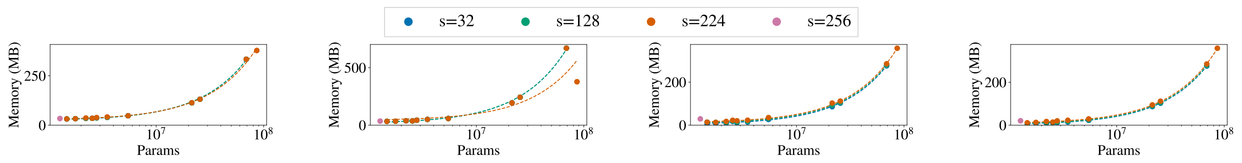
C. Model Analysis

To analyze the usability of ModelGauge, we compared the performance of seven different image-classification models, shown in Table II. The models chosen were meant to compare with commonly used mobile-designed models, such as MobileNetV2 and MobileViT, as well as both convolutional neural



(a) ARM Cortex-A76 (b) AMD EPYC 9374F (c) Orin Nano (d) A100

Fig. 1: Comparing model FLOPs count versus the latency for inference. Error bars represent the standard deviation for each measurement. Dashed lines represent a linear fit for each series of data. Note the semi-log axis.



(a) ARM Cortex-A76 (b) AMD EPYC 9374F (c) Orin Nano (d) A100

Fig. 2: Comparing model parameter count versus the memory consumption for inference. Error bars represent the standard deviation for each measurement. Dashed lines represent a linear fit for each series of data. Note the semi-log axis.

TABLE II: Table highlighting the models tested, any variations tested, and the input image sizes tested. CNNs are retrieved from the TorchVision model zoo [12]. ViTs are retrieved from the HuggingFace Transformers library [13]. Original publication sources are listed in the "Pub" column.

Type	Model Name	Variants	Input Sizes	Pub.
CNN	EfficientNetV2	small	32,128,224	[14]
	MobileNetV2	width_mult ={0.2,0.4,0.6,0.8,1.0}	32,128,224	[15]
	MobileNetV3	small,large	32,128,224	[16]
	ResNet-50	-	32,128,224	[17]
	WideResNet-50	-	32,128,224	[18]
ViT	ViT	-	224	[19]
	MobileViT	-	256	[20]

networks (CNNs) and vision transformers (ViTs). Additionally, to investigate the effect of input size, we varied the input size between 32, 128, and 224 pixels square for the CNNs. The CNNs were downloaded from the Torchvision [12] library and exported to the ONNX format. The ViTs were constrained to set input sizes, which could not be easily changed. These models were downloaded from the HuggingFace transformers library [13].

Additionally, the number of parameters and FLOPs of each model were exported. The number of parameters is found by counting the number of trainable values in the PyTorch models. The number of FLOPs is estimated using the PyTorch-OpCounter library [21]. This additional data will allow us to compare the scaling trends across the different input models. While additional metrics, such as Critical Datapath Length (CDL) [22] would also help explain model performance, our focus in this research is to highlight the ease of gathering data and the quick ability to identify trends amongst many models on varying systems.

IV. Results

The main result of this research is a tool for easily measuring the on-device behavior of many models. In order to exemplify the utility, we highlight the results of gathering latency, memory, and bandwidth data for the models listed in Table II. We explore the linear scaling trends of latency, memory, and bandwidth behavior. On all charts, a dashed line represents the linear fit for each series of data, with a series being a different input image size. Additionally, error bars on each point represent the standard deviation, which is often negligible and not visible. An additional subsection details how bandwidth results can be utilized for developing derived metrics.

By utilizing linear regression and finding the Pearson correlation, we can capture how well a model metric characterizes a linear scaling trend for device performance. While this type of result could theoretically be used for predicting device performance given the metrics of a new model, inference performance is often too complex for one metric alone. Instead, we use these results to see how the linearity of the trends differs across the four devices tested.

A. Latency and FLOPs

The latency trends for the models and devices tested are shown in Figure 1. Here, latency is compared to the number of FLOPs estimated to be in the model. The FLOPs metric is an approximation of the computational complexity of a model assuming each operation would occur serially.

The Pearson correlations between FLOPs and latency on each device are shown in Table III. On the more serial embedded CPU, the correlation is quite strong at 0.990. However, on a highly parallel device such as the NVIDIA A100, the correlation drops significantly to 0.388.

These results match previous research [22], which showed that FLOPs alone are insufficient for explaining performance

TABLE III: Pearson correlation between FLOPs and latency.

Device	Pearson	p-value
ARM Cortex-A76	0.990	p=$8.8 \cdot 10^{-27}$
AMD EPYC 9374F	0.698	p=$9.0 \cdot 10^{-6}$
NVIDIA Jetson Orin Nano	0.878	p=$4.3 \cdot 10^{-11}$
NVIDIA A100	0.388	p=$2.8 \cdot 10^{-2}$

on parallel devices such as GPUs. ModelGauge additionally shows that FLOPs are not a strong linear predictor on server-grade CPUs. On the AMD EPYC 9374F, the Pearson correlation is 0.698. This result highlights that current server-class CPUs have enough parallelism for FLOPs to not scale linearly with performance as they do on the embedded CPU.

B. Memory and Parameters

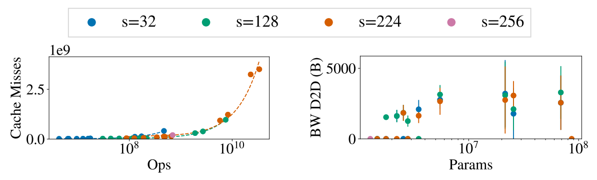
Memory behavior is shown in Figure 2 and is compared to the number of parameters in a model. Across all devices, memory and the number of parameters relate very well, showing near-linear scaling behavior across the tested models. On the AMD EPYC 9374, the outlier with many parameters but below the general trend for s=224 is ViT. Since ViT has a different compute structure than the other CNNs tested, it likely has a different memory layout and requirements.

TABLE IV: Pearson correlation between the number of parameters and memory.

Device	Pearson	p-value
ARM Cortex-A76	0.999	p=$6.9 \cdot 10^{-40}$
AMD EPYC 9374F	0.938	p=$2.5 \cdot 10^{-15}$
NVIDIA Jetson Orin Nano	0.998	p=$9.9 \cdot 10^{-39}$
NVIDIA A100	0.999	p=$1.5 \cdot 10^{-43}$

The trends shown in the graphs are confirmed by the Pearson correlations across the models on the four devices, shown in Table IV. Across all four devices, memory and the number of parameters have a Pearson correlation greater than 0.9. The device with the lowest correlation is the AMD EPYC 9374 at 0.938. This correlation is lower because of the ViT model; when the ViT model is not included in the regression, the Pearson correlation increases to 0.999 (p=$4.3 \cdot 10^{-40}$).

C. Bandwidth



(a) ARM Cortex-A76 (b) Orin Nano

Fig. 3: Comparing bandwidth performance on two devices. Error bars represent the standard deviation for each measurement. Dashed lines represent a linear fit for each series of data. Note the semi-log axis.

The bandwidth behavior varies by platform. Here, we highlight the cache misses on the ARM Cortex-A76 and the average device-to-device (D2D) transfer size on the NVIDIA Jetson Orin Nano, shown in Figure 3. On the CPU device, cache misses correlate higher with FLOPs (Pearson=0.984, p=$4.9 \cdot 10^{-24}$) than with the number of parameters (Pearson=0.816, p=$1.2 \cdot 10^{-8}$).

On the NVIDIA Jetson Orin Nano, no clear trend is evident between the average D2D transfer size and the number of parameters or FLOPs. Additionally, there is often a larger variance in the average D2D transfer size shown by the larger error bars in Figure 3b. Likely, a different perspective than the two common base metrics of the number of parameters and FLOPs is needed to better detail the caching behavior on the GPU.

D. CPU Bandwidth Metric: Small Model Inefficiency Rate

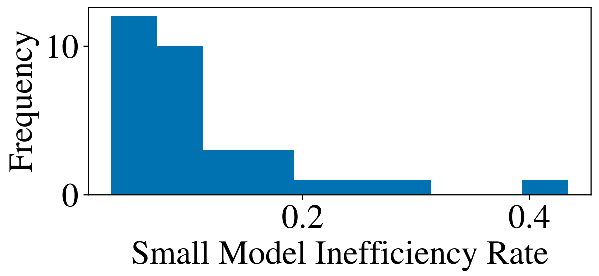


Fig. 4: Histogram showing the frequency of different values for the "Small Model Inefficiency Rate" metric. This metric is found by dividing the dTLB load misses by the latency in milliseconds. Additionally, the value has been scaled down by 10^6 for legibility and discussion.

With the ability to measure bandwidth statistics, we can investigate novel metrics that help to explain performance and model structure. For example, on the CPU device, we can measure the data translation lookaside buffer (dTLB) load misses divided by the model latency, which we will call the "Small Model Inefficiency Rate" (SMIR). Essentially, this value appears to highlight the models that are particularly small and likely not fully utilizing the embedded CPU.

Figure 4 shows a histogram of the tested models. Note that SMIR values have been scaled down by 10^6 for legibility and to ease the discussion of data. We find that in this exponential trend, the models in the tail tend to be small, have small inputs, and have small latencies. Essentially, the models and their inputs appear to be so small that they don't effectively use all the compute capabilities of the device for the number of memory transactions. For example, the largest SMIR belongs to MobileNetV2 at $0.2\times$ scaling with a 32×32 input. All other input sizes for MobileNetV2 at $0.2\times$ scaling have a SMIR greater than 0.15. Additionally, MobileNetV3_small with the smallest input falls in this range.

We believe this metric could be useful for providing insight into the memory access patterns for models. Lower SMIR values show a mix of model types, latencies, and input sizes.

However, larger SMIRs in our testing specifically belong to models with very low latencies and disproportionately larger dTLB load misses. Comparing SMIR across larger devices would help to highlight more of these memory access trends, but the restrictions on the shared server CPU system limit this study. More verification of SMIR is needed to confirm its utility for identifying models that are too small.

V. Discussion

Through the use of ModelGauge, we were able to quickly and efficiently identify performance trends across a variety of models on four different systems. The ability to gather this data represents the most impactful component of our research. In future efforts, we will use this tool to expand our understanding of how model metrics and performance scale across devices.

In this scalability study, we found that the modern server CPU system has enough parallelism to have a lower correlation between FLOPs and latency than the embedded GPU system. Given the 32 cores found on the AMD EPYC 9374F CPU and its high power ceiling, ONNX Runtime is able to perform very quick inference. This result highlights the need for other metrics such as CDL on the server-class CPU system as well as the GPU systems. Given that the linear correlation between latency and FLOPs is lower for the AMD EPYC CPU than it is for the NVIDIA embedded GPU, it also highlights how the server CPU performance is more similar to the embedded and server GPU systems than the embedded CPU system.

It is also worth noting that the AMD EPYC CPU has similar latencies to the Orin Nano GPU. The Orin Nano has an approximate power budget of 15 W for the whole system, while an EPYC CPU has a default TDP of 320 W [23]. Given the similar performance of these two systems with a batch size of one, but vastly different power requirements, a study into throughput differences between the devices would likely help to differentiate the systems.

The Pearson correlation between memory and parameter count is high, even on the GPU devices. This result differs from previous research in [24] where an estimation of GPU memory often fluctuated due to the size of memory required to store the activations. In this study, there was in general a low standard deviation for the memory measurements. Given the previous study in [24] had estimated memory, while this study measures the on-device performance, there likely does exist lower variability in TensorRT Engine memory usage as seen in Table IV. Therefore, studying many different implementations of a TensorRT engine for the same models is likely not needed in future research.

Additionally, we believe our measurements of memory on the CPU to be the most useful. Other measurements of the resident set size or virtual memory size would measure the values currently in RAM or would include shared libraries and other data objects. Massif, instead, focuses on memory allocated to the heap and memory stored on the stack [9]. Essentially, Massif allows ModelGauge to focus on the memory allocations needed specifically for the model under test. We believe, by wrapping Massif into this easy-to-use tool, we provide a fair representation of memory usage for comparing across model types while reducing variability from shared libraries or other memory components.

VI. Conclusions

Measuring the on-device performance of DL models is increasingly important given the growth of many modern architectures and the increasing ubiquity of machine learning. By understanding the latency, memory, and bandwidth behavior of these different models, we can make better-informed decisions about which models we choose in a deployment scenario or develop better architectures for more efficient inference. However, benchmarking performance can be challenging given the number of models in existence and the need for varying performance metrics such as latency and memory.

ModelGauge is our proposed solution to this problem. Through this tool and the use of the standardized ONNX format, many models can easily be measured and quantified across a variety of devices. By using the high-performance inferencing frameworks of ONNX Runtime on CPUs and TensorRT on GPUs, ModelGauge is able to profile expected model behavior on devices ranging from embedded to server scales. Built on top of other industry standard tools such as Valgrind Massif and NVIDIA Nsight Systems, our tool was able to successfully profile many models and their variants across four test devices. With the models tested, we found a high linear correlation between the number of parameters in a model and the memory used across all devices. The latency linearly correlates well to the number of FLOPs on the embedded CPU, but the correlation degrades on the other devices where more parallelism exists. The server CPU system tends to have latency and parameter count trends more similar to GPU systems than to the embedded CPU system.

Additionally, we propose SMIR as a new metric to capture memory access patterns. SMIR is found by dividing the dTLB load misses by the model latency. This metric followed an exponential trend, with models that were particularly small having a disproportionately large SMIR. After further future verification of SMIR on more architectures and more devices, we aim for model deployers to be able to check to see if their model is underutilizing the device. Overall, ModelGauge is designed to be useful to the community, and we plan to open-source the tools so others may use it and add additional devices and runtimes.

VII. Future Research

Development of the ModelGauge and study of standard image-classification models limited the scope of this article. In future research, we plan to utilize this tool to benchmark the scalability of models across CPUs and GPUs targeted at embedded and server workloads. We plan to explore how model metrics scale with device performance. Additionally, further study on the impact of batch size would be useful. Finally, adding support for models with weights stored separately from the definition would allow testing with larger models.

VIII. Acknowledgements

This research was supported by SHREC industry and agency members and by the IUCRC Program of the National Science Foundation under Grant No. CNS-1738783. This research was supported in part by the University of Pittsburgh Center for Research Computing, RRID:SCR_022735, through the resources provided. Specifically, this work used the H2P cluster, which is supported by NSF award number OAC-2117681.

References

[1] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "MLPerf Inference Benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. Valencia, Spain: IEEE, May 2020, pp. 446–459. [Online]. Available: https://ieeexplore.ieee.org/document/9138989/

[2] V. Janapa Reddi, D. Kanter, P. Mattson, J. Duke, T. Nguyen, R. Chukka, K. Shiring, K.-S. Tan, M. Charlebois, W. Chou, M. El-Khamy, J. Hong, T. St John, C. Trinh, M. Buch, M. Mazumder, R. Markovic, T. Atta, F. Cakir, M. Charkhabi, X. Chen, C.-M. Chiang, D. Dexter, T. Heo, G. Schmuelling, M. Shabani, and D. Zika, "MLPerf mobile inference benchmark: An industry-standard open-source machine learning benchmark for on-device AI," in *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu, Eds., vol. 4, 2022, pp. 352–369. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2022/file/a2b2702ea7e682c5ea2c20e8f71efb0c-Paper.pdf

[3] V. Kustikova, E. Vasiliev, A. Khvatov, P. Kumbrasiev, R. Rybkin, and N. Kogteva, "DLI: Deep Learning Inference Benchmark," in *Supercomputing*, V. Voevodin and S. Sobolev, Eds. Cham: Springer International Publishing, 2019, vol. 1129, pp. 542–553. [Online]. Available: http://link.springer.com/10.1007/978-3-030-36592-9_44

[4] H. Kong, S. Huai, D. Liu, L. Zhang, H. Chen, S. Zhu, S. Li, W. Liu, M. Rastogi, R. Subramaniam, M. Athreya, and M. A. Lewis, "EDLAB: A Benchmark for Edge Deep Learning Accelerators," *IEEE Design & Test*, vol. 39, no. 3, pp. 8–17, Jun. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9475509/

[5] Q. Zhang, X. Li, X. Che, X. Ma, A. Zhou, M. Xu, S. Wang, Y. Ma, and X. Liu, "A Comprehensive Benchmark of Deep Learning Libraries on Mobile Devices," in *Proceedings of the ACM Web Conference 2022*. Virtual Event, Lyon France: ACM, Apr. 2022, pp. 3298–3307. [Online]. Available: https://dl.acm.org/doi/10.1145/3485447.3512148

[6] ONNX developers, "ONNX: Open Neural Network Exchange," 2019. [Online]. Available: https://github.com/onnx/onnx

[7] ONNX Runtime developers, "ONNX Runtime," 2021. [Online]. Available: https://onnxruntime.ai/

[8] NVIDIA, "NVIDIA TensorRT," 2023. [Online]. Available: https://developer.nvidia.com/tensorrt

[9] Valgrind Developers, "Valgrind," 2023. [Online]. Available: https://valgrind.org/

[10] L. Torvalds, "Linux perf," 2023. [Online]. Available: https://github.com/torvalds/linux/tree/master/tools/perf

[11] NVIDIA Developers, "TensorRT/samples/trtexec at main · NVIDIA/TensorRT." [Online]. Available: https://github.com/NVIDIA/TensorRT/tree/main/samples/trtexec

[12] PyTorch Developers, "Models and pre-trained weights — Torchvision documentation," 2023. [Online]. Available: https://pytorch.org/vision/stable/models.html

[13] Hugging Face Developers, "Hugging Face Transformers." [Online]. Available: https://huggingface.co/docs/transformers/en/index

[14] M. Tan and Q. Le, "EfficientNetV2: Smaller Models and Faster Training," in *Proceedings of the 38th International Conference on Machine Learning*. Virtual: PMLR, Jul. 2021, pp. 10 096–10 106. [Online]. Available: https://proceedings.mlr.press/v139/tan21a.html

[15] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. Salt Lake City, UT: IEEE, Jun. 2018, pp. 4510–4520. [Online]. Available: https://ieeexplore.ieee.org/document/8578572/

[16] A. Howard, M. Sandler, B. Chen, W. Wang, L.-C. Chen, M. Tan, G. Chu, V. Vasudevan, Y. Zhu, R. Pang, H. Adam, and Q. Le, "Searching for MobileNetV3," in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. Seoul, Korea (South): IEEE, Oct. 2019, pp. 1314–1324. [Online]. Available: https://ieeexplore.ieee.org/document/9008835/

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2016-Decem. Las Vegas, Nevada, USA: IEEE Computer Society, 2016, pp. 770–778.

[18] S. Zagoruyko and N. Komodakis, "Wide Residual Networks," in *Procedings of the British Machine Vision Conference 2016*. York, UK: British Machine Vision Association, 2016, pp. 87.1–87.12. [Online]. Available: http://www.bmva.org/bmvc/2016/papers/paper087/index.html

[19] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," Jun. 2021. [Online]. Available: http://arxiv.org/abs/2010.11929

[20] S. Mehta and M. Rastegari, "MobileViT: Light-weight, general-purpose, and mobile-friendly vision transformer," in *International Conference on Learning Representations*, 2022. [Online]. Available: https://openreview.net/forum?id=vh-0sUt8HlG

[21] L. Zhu, "PyTorch-OpCounter," Sep. 2022. [Online]. Available: https://github.com/Lyken17/pytorch-OpCounter

[22] D. Langerman, A. Johnson, K. Buettner, and A. D. George, "Beyond Floating-Point Ops: CNN Performance Prediction with Critical Datapath Length," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, Sep. 2020, pp. 1–9. [Online]. Available: https://ieeexplore.ieee.org/document/9286182/

[23] "AMD EPYC™ 9374F." [Online]. Available: https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series/amd-epyc-9374f.html

[24] C. B. Gealy and A. D. George, "Characterizing Parameter Scaling with Quantization for Deployment of CNNs on Real-Time Systems," *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 3, pp. 1–35, May 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3654799