

Mastering Computer Vision Inference Frameworks

Pierrick Pochelu 

University of Luxembourg FSTM-DCS
Luxembourg
pierrick.pochelu@uni.lu

Oscar Castro-Lopez 

University of Luxembourg FSTM-DCS
Luxembourg
oscar.castro@uni.lu

ABSTRACT

In this paper, we present a comprehensive empirical study to evaluate four prominent Computer Vision inference frameworks. Our goal is to shed light on their strengths and weaknesses and provide valuable insights into the challenges of selecting the right inference framework for diverse situations. Additionally, we discuss the potential room for improvement to accelerate inference computing efficiency.



CCS CONCEPTS

• **Software and its engineering**; • **Computing methodologies**
→ **Computer vision**;

KEYWORDS

software performance, neural networks, inference

ACM Reference Format:

Pierrick Pochelu  and Oscar Castro-Lopez . 2024. Mastering Computer Vision Inference Frameworks. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24 Companion)*, May 7–11, 2024, London, United Kingdom. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3629527.3651430>

1 INTRODUCTION

In the field of deep learning, the deployment of trained neural networks to make predictions, a process also known as inference, is the pivotal moment where these models provide value in applications. Inference deep learning frameworks play a central role in this process by translating the mathematical representation of the neural network into low-level code optimized for specific hardware platforms. These inference frameworks employ a diverse range of optimizations aimed at significantly improving the computational speed of neural network predictions.

While the training phase of deep neural networks is generally a time-bounded computing process with a fixed number of epochs or batches, the inference phase often involves long-term deployment. This is why pursuing lower prediction time has been recognized as a strategically significant endeavor for reducing the financial cost of computing infrastructure [6] and greener computing [5].

Various benchmarks have emerged to assess the speed of deep neural networks, they often focus on the performance of specific

operators like matrix multiplication and convolution. These microbenchmarks, while informative, fall short of comprehensively evaluating the intricate complexities of modern neural networks [19]. Additionally, benchmarks like Dawn Bench [3] have highlighted that neural networks frequently under-utilize computing cores due to memory transfer bottlenecks. Although studies such as MLPerf Inference [15] and ML Bench [13] provide comprehensive assessments of various inference applications, frameworks, and hardware, they generally lack in-depth analysis of the comparison between different inference frameworks and software settings. This paper aims to address this gap by providing fresh perspectives to steer the future development of inference technology and set of tools for reproducibility.

This paper explores the inference performance exhibited by the inference frameworks such as TensorRT [4], ONNX-runtime [16], OpenVINO [7], LLVM MLIR [11], TVM [2]. These inference frameworks employ a diverse range of optimizations aimed at significantly improving the computational speed of neural network inference. Additionally, we also test/compare the performance of Tensorflow XLA [12] which, unlike the aforementioned inference frameworks, is a software environment that applies optimizations for both the training and inference phases.

With the inference frameworks we benchmark different convolutional neural network (CNN) architectures used in computer vision tasks selected for their diversified neural topology: Resnet50[9] VGG19[20] and DenseNet201[10]. For each framework, we collected over 80 data points, encompassing metrics such as prediction throughput (predictions per second), loading time, memory consumption, and power consumption on both GPU and CPU hardware configurations. The code and additional plots are linked in the GitHub repository at the end of the conclusion.

The structure of this paper is as follows. In Section 2, we discuss inference frameworks state-of-art and their optimization techniques. In Section 3, we elaborate on the settings used. In Section 4, we present the experimental results with different metrics. In Section 5, we provide key insights by summarizing the lessons learned from the experiments. Finally, in Section 6 we conclude by showing the importance of this work direction, future work, and GitHub links.

2 POST-TRAINING REPRESENTATION AND OPTIMIZATION

Post-training representation and optimization serve as a critical bridge between model training and efficient inference deployment.

The optimization is generally done in two steps, high-level and low-level optimizations. While high-level optimizations focus on algorithmic and architectural enhancements, low-level optimizations delve into the intricacies of code generation and hardware-specific

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE '24 Companion, May 7–11, 2024, London, United Kingdom

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0445-1/24/05...\$15.00
<https://doi.org/10.1145/3629527.3651430>

performance tuning. The synergy between these optimization layers empowers deep learning frameworks to deliver both accuracy and speed in real-world applications. However, different inference frameworks have different implementations.

One fundamental high-level optimization strategy is known as "fusing," a technique that consolidates multiple operations into a single kernel launch. For example, sequences of convolutions can be merged into one single convolution [1]. Fusing offers several advantages, including the elimination of slow intermediate tensor storage, improved cache utilization, and the removal of synchronization barriers between operations. Notably, these optimization methods, such as XLA [12], not only accelerate inference but can also benefit the training phase. Moreover, certain fusing operations involve mathematical simplifications, like combining adjacent convolution layers together without lowering the accuracy of the model [8].

Additional high-level optimizations encompass techniques such as constant-folding and static memory planning. Constant-folding evaluates constant expressions before executing the program and static memory planning creates a plan to reuse ahead and use intermediate tensor buffers.

On the other end, low-level optimization entails converting the computational graph into highly optimized low-level code tailored for specific hardware platforms. Drawing upon decades of compilation expertise, this optimization category encompasses sub-expression elimination, vectorization, loop ordering, tiling, unrolling, threading patterns, and memory caching/reusing strategies. Notably, the field benefits from two noteworthy low-level compilers and optimizers, namely TVM [2] and LLVM MLIR [11], both enriched with tensor types, which enable efficient code generation and execution.

3 EXPERIMENTAL SETTINGS

3.1 Evaluated Neural Networks

To assess the performance of different inference frameworks, we utilize a diverse set of convolutional neural networks, including VGG19, ResNet50, DenseNet201, and EfficientNetB0. Overall, these networks offer a range of characteristics, such as varying depths, widths (parametric ratio to the number of layers), and densities, providing a comprehensive evaluation of the frameworks' capabilities. The details of these architectures are summarized in Table 1.

Table 1: Summary of the CNNs architectures used in the benchmark.

	$\frac{\#param.}{\#layers}$	$\#layers$	$\#jumps$	$\#param.$	Jump type
VGG19	7.26M	19	0	138M	No jumps
ResNet50	0.52M	50	16	26M	Additions
DensetNet201	0.1M	201	98	20M	Concatenations
EfficientNetB0	0.06M	89	25	5.3M	Mult. and Add.

3.2 Evaluated Machine Specifications

Our benchmarking experiments are conducted on two distinct machines, namely Machine A and B. Machine A is equipped with Tesla

V100 SXM2 GPUs and its CPU is a dual-socket Intel(R) Xeon(R) CPU E5-2698 v4. Whereas, Machine B has NVIDIA Amper A100 PCI-E GPUs and its CPU is a single-socket AMD EPYC 7F52. Table 2 presents the full details for both machines.

For the Software Stack, we maintain consistent software versions across both machines. Our assessment involves popular inference frameworks, including TensorFlow 2.6, TensorRT 8.0, ONNX-runtime 1.10, and OpenVINO 2021. To facilitate the benchmarking process, we utilize neural network converters such as tf2onnx 1.9.3 and LLVM 14.0.

Machine A operates on Ubuntu with Python 3.9, while Machine B runs on CentOS with Python 3.8. It's worth noting that specialized real-time operating systems may enhance latency determinism and speed but could potentially impact throughput negatively. We also explored MLIR (onnx-mlir 0.2 framework [11]) on Machine B; however, it only supports ResNet50. In all our benchmarks, Tensorflow benefits from acceleration via XLA [12] (Accelerated Linear Algebra).

Table 2: Summary of the characteristics of the machines used in the benchmark.

Feature	Machine A	Machine B
GPU model	Tesla V100 SXM2	Amper A100 PCIE
GPU # of cores	5,120	6,912
GPU clock speed	1,312MHz-1,530MHz	765MHz-1,310MHz
GPU memory	16GB	40GB
GPU board cons.	300 watts	250 watts
CPU model	Intel XEON E5-2698 v4	AMD EPYC 7f52
CPU # of cores	80	16
CPU clock speed	1.2GHz-3.6GHz	2.5GHz-3.5GHz
CPU memory	512GB	256GB
OS	Ubuntu	CentOS
Python version	3.9	3.8

3.3 Graph Optimization Settings

To discuss optimization settings and the effect of each performance we discuss the obtained speed up by ablation (starting from the mostly well optimized settings and discussing the impact of changing a specific setting).

- **Tensorflow XLA:** We freeze the computational graph, which means all weights are stored in read-only memory. We enable the "optimize_for_inference_lib" optimizer, although it doesn't significantly impact performance. XLA is enabled on the GPU because disabling it reduces speed by 15%. However, enabling XLA multiplies initialization time by a factor of six on the range of neural networks. We do not observe a performance gain from enabling XLA on the CPU. Therefore, it remains disabled.
- **ONNX-RT (ONNX-runtime)** [16]: We enable caching, as disabling it reduces speed by approximately 3%. We also enable maximum graph-level optimization, and using the default optimization settings reduces speed by 8%.
- **OpenVINO** We enabled two settings: "NCHW" and convolution fusing. NCHW stands for batch (N), channels (C),

height (H), and width (W). NCHW is a data format, a way to represent a tensor in memory and all input images are converted into this format for this framework. In a nutshell, convolution fusing combines or merges multiple convolutional operations within a CNN model into a single, more efficient operation. When convolutional fusing is disabled the execution speed is reduced by 9%. We disabled concatenation optimization because there was no significant gain in any experiment. In fact, with Densenet201 concatenation optimization reduced the performance by 1%.

- **MLIR** [11] (Multi-Level Intermediate Representation from the LLVM project): MLIR is still under development, but we could compile the Resnet50 graph with the "-O3" optimization level for performance. This optimization level represents the highest level of optimization provided by the compiler.

These settings have been carefully configured to ensure optimal performance for each framework, enabling a fair and informative comparison of their capabilities.

4 EXPERIMENTAL RESULTS

After the training phase, the importance of performance metrics can significantly differ depending on the application at hand. The prioritization of specific metrics over others is deeply influenced by the unique requirements of each use case. In this study, we conduct assessments focusing on prediction speed, memory utilization, power consumption, and model loading time.

4.1 Prediction Speed

Our assessment of prediction speed encompasses three key scenarios, each measured differently:

- **Batch Applications:** In scenarios where neural networks predict a substantial workload of data samples, optimizing the batch size becomes critical to maximize predictions per second, referred to as throughput. Throughput is quantified as the number of predictions per second.
- **Data Flow Applications:** For use cases involving sequential data sample prediction, such as real-time embedded systems and Markov Decision Processes in deep reinforcement learning, we evaluate latency, represented as the number of milliseconds required for a single prediction.
- **Irregular Batch Applications:** In situations where data samples arrive irregularly, such as web services serving multiple client requests, a dynamic batch computing approach is essential. Here, we carefully balance latency for responsiveness and throughput when the service faces heavy request loads.

Figure 1 and 2 show the throughput results for machines A and B, respectively. Each figure has results for the inference frameworks by varying the batch size and the model. Notice the vertical axis is log2-scale. Please note that we have opted not to display latencies for predicting a single data point (data flow applications) since these results exhibited a strong correlation with batch size 1. In cases where there are no bars in the figure, it indicates out-of-memory issues (e.g., VGG with TensorRT) or compilation errors (e.g., EfficientNet with OpenVINO).

4.2 Memory Consumption

Optimizing memory usage within the inference framework unlocks various GPU utilization possibilities, effectively reducing the need for investments in multi-GPU configurations and minimizing their associated power demands. The diverse memory consumption scenarios encompass serving larger models, enhancing accuracy through the management of asynchronous ensembles [14], and efficiently handling independent applications [18]. Figure 3 shows the memory consumption of an ensemble model, quantified as the combined storage occupied by the neural network on disk and the current batch of features propagating through the layers. The ensemble results from diverse topologies: VGG19 has wider convolutions, Resnet50 is deeper, and Densenet201 includes numerous jumps between layers.

4.3 Power consumption

Figure 4 presents the power consumption (Watt/sec.) of different inference frameworks ("TRT" for TensorRT, "TF" for Tensorflow, "ORT" for ONNX-RT) with batch sizes 1, 32, and 128. The model running is the ensemble of VGG19, Resnet50, and DenseNet201.

We use the following command to estimate GPU power draw:

```
$ nvidia-smi -i $GPUID --format=csv --query -- \
  ↪ gpu=power.draw --loop-ms=3000
```

Where \$GPUID is the corresponding GPU identifier hosting the neural network.

It's worth noting that the Relative Standard Deviation (RSD) of instantaneous power consumption (watt) can be high, approximately around 20%. The oscillations observed in the estimated GPU consumption are attributed to a combination of factors including instruction flows, dynamic voltage and frequency, temperature regulation, and measurement error. This underscores the significance of averaging power consumption over multiple sampling to obtain a more representative value.

Power consumption is a pivotal metric, with implications for ecological sustainability, energy costs, and thermal management. We express power consumption in terms of watt-seconds required by the inference system to predict a fixed quantity of data samples, denoted as D . The actual value of power consumption is influenced by the specific neural network, chosen inference engine, and batch size. These measurements can be further converted into equivalent units such as CO2 emissions, energy expenses, or thermal dissipation.

Equation 1 is used to describe power consumption denoted as E . The variable D stands for the quantity of data samples. Whereas variable W represents the mean instantaneous power consumption in watts during the prediction period T in seconds.

$$E = D \times W \times T \tag{1}$$

Equation 1 provides insights into the relationship between the throughput of an inference system and its power consumption. It appears that the relationship adheres to a power law $y = \alpha x^\beta$, with α and β coefficients for different GPU generations.

The measurement teaches us two lessons for sustainable computing. First, model speed and power consumption are linked in a predictable way, their correlation is above 0.95. Second, the power

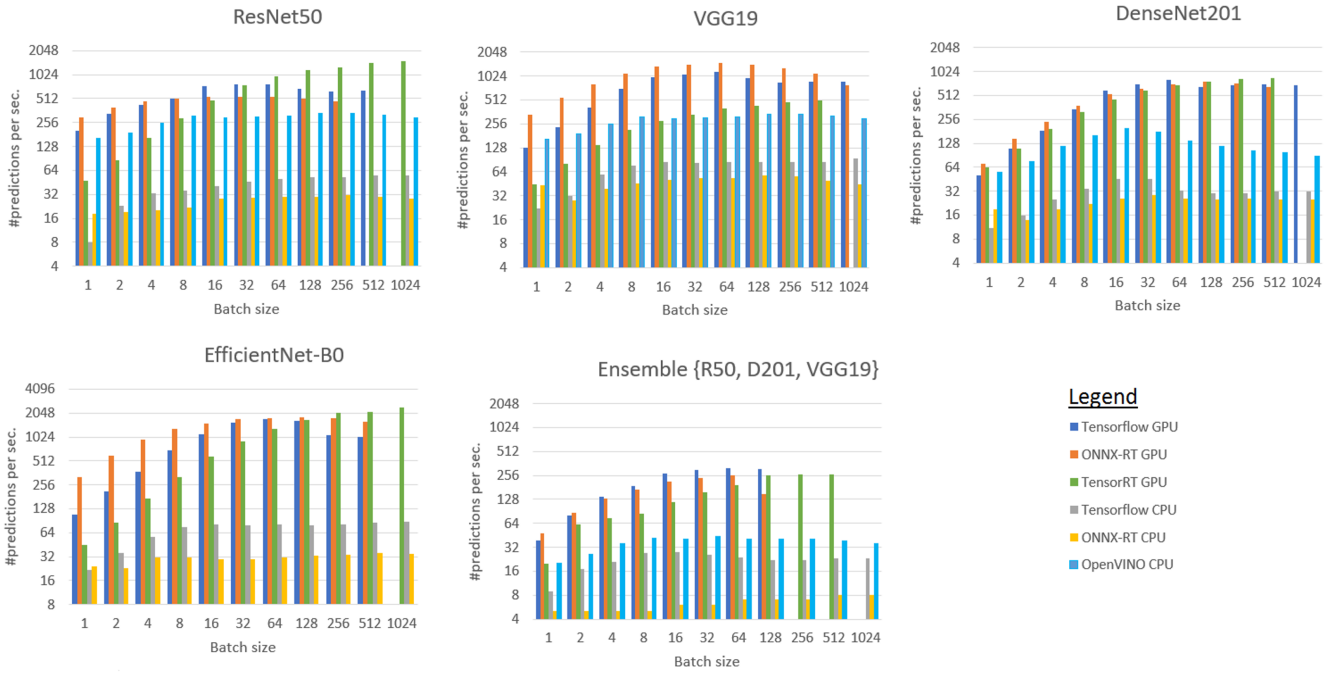


Figure 1: Machine A. Inference framework comparison with different batch size values.

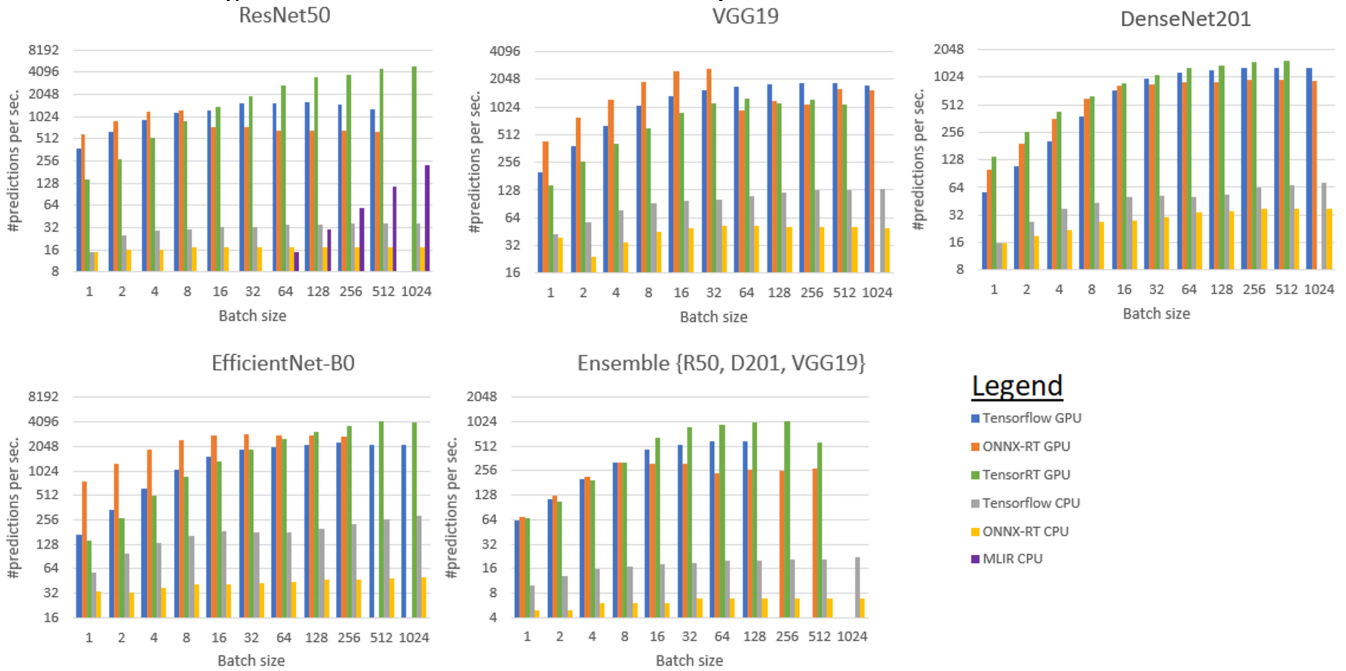


Figure 2: Machine B. Inference framework comparison with different batch size values.

law curvature shows us that to a certain degree, improving the model parallelism may not reduce significantly the power consumption. This reduction in the trend can be interpreted like the following: maximizing cores utilization reduces computing time

(T) but increases instantaneous power consumption (W), less efficient internal parallelism keeps the cores idle which takes more computing time but consumes less power.

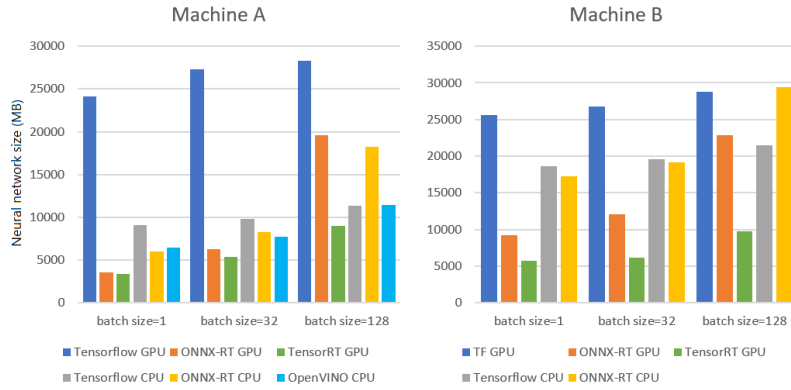


Figure 3: Memory consumption of the model's ensemble with different frameworks varying the batch size.

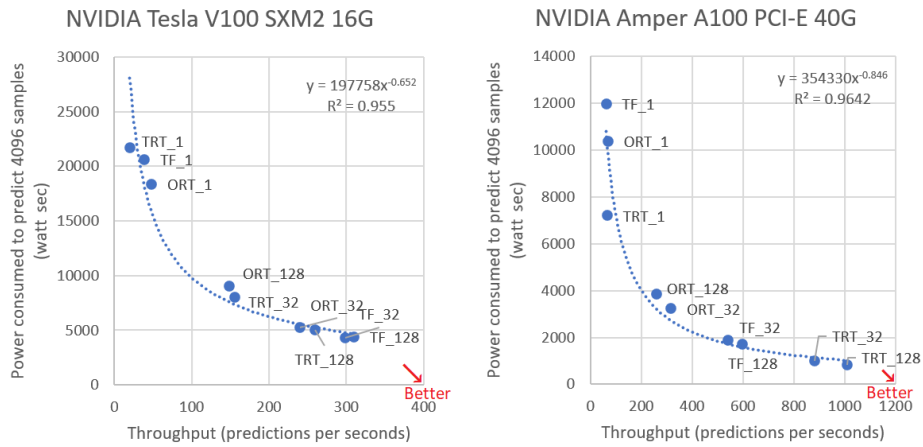


Figure 4: Plot of the throughput and power consumption with the ensemble model varying the batch size.

4.4 Loading Time

Loading time refers to the duration from the neural network's representation on disk to its readiness in memory for making predictions. This metric is of particular interest in applications requiring rapid service setup to accommodate peak demand periods, such as elastic cloud services [17]. In all our benchmarks, we enable or implement caching systems to measure loading times.

The loading times for the ensemble model (containing VGG19, Resnet50, and Densenet121) are displayed in Table 3. The data is organized by machine and processing unit (CPU/GPU), with faster loading times listed before slower ones for clarity.

TensorFlow with XLA optimization exhibits slower loading times due to the current absence of caching for optimized graphs on disk. However, disabling XLA optimization reduces loading time by a factor of 6 but it keeps staying the slowest inference framework to load the model.

5 KEY TAKEAWAYS

In this section, we offer valuable insights derived from our benchmarking outcomes. These insights are designed to aid deep learning practitioners in selecting the most suitable inference framework for their specific requirements.

Table 3: Results of loading the models categorized by machine and processing units.

Machine	Device	Framework	Time (seconds)
A	CPU	OpenVINO	3.4
		TensorFlow	8.7
		ONNX-RT	8.7
	GPU	TensorRT	3.6
		ONNX-RT	5.7
		Tensorflow XLA	42.5
B	CPU	ONNX-RT	0.5
		Tensorflow	2.9
	GPU	ONNX-RT	2.1
		TensorRT	3.9
		Tensorflow XLA	29.2

The performance of GPU-based frameworks, including ONNX-RT, TensorRT, and TensorFlow with XLA, had results in response to different scenarios (batch size values and model architecture). For low-latency and sporadic request scenarios, ONNX-RT had the best results. TensorRT shines in high-throughput scenarios with larger batch sizes. TensorFlow optimized with XLA shows the good speed with high-density networks (i.e. VGG19).

For CPU inference, Intel OpenVINO consistently outperformed other CPU-based frameworks, making it a strong choice for CPU-centric deployments. MLIR shows promise and should be considered for future use as it continues to mature.

It is widely acknowledged by deep learning practitioners that optimizing the batch size for speed's sake is an empirical process. A "sufficiently large" batch size value allows inference frameworks to harness the full power of a particular processing unit. However, if the batch size value is too large, it can result in cache memory issues, potentially slowing down execution. It is worth noting that some inference frameworks have specific constraints on tensor shapes (such as TensorRT with VGG19 and batch size 1024). Furthermore, very large tensor shapes may lead to memory crashes due to indexing element errors.

In terms of GPU memory usage, TensorFlow XLA GPU consumes significantly more memory than other technologies but the gap is slightly reduced when the batch size increases. Conversely, TensorRT stands out for having the lowest overall memory footprint. For CPU memory usage, Tensorflow XLA exhibits the largest memory footprint, particularly noticeable with batch sizes 1 and 32. However, with a batch size of 128, ONNX-RT for CPU experiences a considerable increase in memory usage. The most memory-efficient option for CPU is OpenVINO.

With the power consumption metric, we provide a link between computing speed and power consumption for a fixed amount of data samples. This leads us to the conclusion that optimizing the computing time reduces the power consumption up to a certain extent.

6 CONCLUSION AND FUTURE DIRECTIONS

We benchmark four deep learning inference frameworks: TensorRT, ONNX-runtime, OpenVINO, and LLVM MLIR and a diverse array of neural network architectures and configurations. Some inference frameworks are still missing such as TVM and will be introduced later in our repo.

Our study has yielded valuable insights into the domain of machine learning inference optimization. We learned that selecting the ideal inference framework from the multitude of options available can be a daunting task. In addition to that, our findings underscore the importance of aligning the specific choice with the final application requirements and hardware environment. Therefore fast experimentation of the application under different settings is desirable to optimize it, this is what we published in our GitHub repo.

Looking ahead, more in-depth analysis will lead our future explorations in the design of inference development tools and inference systems. We anticipate that our work will inspire research and innovation leading to more efficient and effective machine learning solutions for making easier the development and deployment of optimized neural networks.

Supplementary materials, including the GitHub repository link and full-resolution figures, will be provided upon acceptance for the double-blind reviewing process.

REFERENCES

[1] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on*

Microarchitecture (MICRO), 1–12. <https://doi.org/10.1109/MICRO.2016.7783725>

[2] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (Carlsbad, CA, USA) (OSDI'18)*. USENIX Association, USA, 579–594.

[3] Cody Coleman, Daniel Kang, Deepak Narayanan, Luigi Nardi, Tian Zhao, Jian Zhang, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. 2019. Analysis of DAWNbench, a Time-to-Accuracy Machine Learning Performance Benchmark. *SIGOPS Oper. Syst. Rev.* 53, 1 (jul 2019), 14–25. <https://doi.org/10.1145/3352020.3352024>

[4] Pooya Davoodi, Chul Gwon, Guangda Lai, and Trevor Morris. 2019. TensorRT inference With TensorFlow. GPU Technology Conference.

[5] Radosvet Desislavov, Fernando Martinez-Plumed, and José Hernández-Orallo. 2023. Trends in AI inference energy consumption: Beyond the performance-vs-parameter laws of deep learning. *Sustainable Computing: Informatics and Systems* 38 (2023), 100857. <https://doi.org/10.1016/j.suscom.2023.100857>

[6] Dominic Divakaruni, Peter Jones, Sudipta Sengupta, Liviu Calin. 2018. Amazon Elastic Inference: Reduce Learning Inference Cost. *AWS re:Invent 2018*.

[7] Yi Ge and Monique Jones. 2018. Inference With Intel. AI DevCon 2018.

[8] Mathilde Guillemot, Catherine Heusele, Rodolphe Korichi, Sylvianne Schnebert, and Liming Chen. 2020. Breaking Batch Normalization for better explainability of Deep Neural Networks through Layer-wise Relevance Propagation. *CoRR abs/2002.11018* (2020). arXiv:2002.11018 <https://arxiv.org/abs/2002.11018>

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778. <https://doi.org/10.1109/CVPR.2016.90>

[10] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2261–2269. <https://doi.org/10.1109/CVPR.2017.243>

[11] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization (Virtual Event, Republic of Korea) (CGO '21)*. IEEE Press, 2–14. <https://doi.org/10.1109/CGO51591.2021.9370308>

[12] Chris Leary and Todd Wang. 2017. XLA: Tensorflow, Compiled!. In *Tensorflow developer summit*.

[13] Yu Liu, Hantian Zhang, Luyuan Zeng, Wentao Wu, and Ce Zhang. 2018. MLbench: Benchmarking Machine Learning Services against Human Experts. *Proc. VLDB Endow.* 11, 10 (jun 2018), 1220–1232. <https://doi.org/10.14778/3231751.3231770>

[14] P. Pochelu, S. G. Petiton, and B. Conche. 2021. An efficient and flexible inference system for serving heterogeneous ensembles of deep neural networks. In *2021 IEEE International Conference on Big Data (Big Data)*. IEEE Computer Society, Los Alamitos, CA, USA, 5225–5232. <https://doi.org/10.1109/BigData52589.2021.9671725>

[15] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 2020. MLPerf Inference Benchmark. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (Virtual Event) (ISCA '20)*. IEEE Press, 446–459. <https://doi.org/10.1109/ISCA45697.2020.00045>

[16] Dmytro Dzhulgakov Sarah Bird. 2017. ONNX. In *Workshop NIPS2017*.

[17] Rahul Sharma. 2019. Amazon Elastic Inference: Reduce Deep Learning Inference Cost. GPU Technology Conference.

[18] Fuxun Yu, Di Wang, Longfei Shangguan, Minjia Zhang, Chenchen Liu, and Xiang Chen. 2022. A Survey of Multi-Tenant Deep Learning Inference on GPU. *arXiv e-prints*, Article arXiv:2203.09040 (March 2022), arXiv:2203.09040 pages. <https://doi.org/10.48550/arXiv.2203.09040> arXiv:2203.09040 [cs.DC]

[19] Wei Zhang, Wei Wei, Lingjie Xu, Lingling Jin, and Cheng Li. 2019. AI Matrix: A Deep Learning Benchmark for Alibaba Data Centers. *CoRR abs/1909.10562* (2019). arXiv:1909.10562 <http://arxiv.org/abs/1909.10562>

[20] Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun. 2016. Accelerating Very Deep Convolutional Networks for Classification and Detection. *IEEE Trans. Pattern Anal. Mach. Intell.* 38, 10 (Oct. 2016), 1943–1955. <https://doi.org/10.1109/TPAMI.2015.2502579>