The Framework Tax: Disparities Between Inference Efficiency in Research and Deployment

Jared Fernandez [1] **Jacob Kahn** [2] **Clara Na** [1] **Yonatan Bisk** [1] **Emma Strubell** [1]

Abstract

Increased focus on the efficiency of machine learning systems has led to rapid improvements in hardware accelerator performance and model efficiency. However, the resulting increases in computational throughput and reductions in floating point operations have not directly translated to improvements in wall-clock inference latency. We demonstrate that these discrepancies can be largely attributed to bottlenecks introduced by deep learning frameworks. We denote this phenomena as the *framework tax*, and observe that the disparity is growing as hardware speed increases over time. In this work, we examine this phenomena through a series of case studies analyzing the effects of model design decisions, framework paradigms, and hardware platforms on total model latency. Based on our findings, we provide actionable recommendations to researchers and practitioners aimed at narrowing the gap between efficient ML model research and practice.

1. Introduction

Machine learning efficiency is especially important in inference settings when models are used repeatedly and at scale. For example, Meta reports that inference workloads make up 70% of their AI power consumption, with the remaining 30% due to training and development (Wu et al., 2022), while Google attributes 60% of their ML energy consumption to inference (Patterson et al., 2022). Inference is also estimated to make up 80 to 90% of ML cloud computing demand (Barr, 2019; Leopold, 2019).

These concerns have spurred innovations in efficient model architecture design with lesser computational requirements, that reduce number of model parameters and multiply-accumulate operations (MACs) (Tan & Le, 2019; Dai et al., 2020; Sun et al., 2020). Likewise, improvements to GPU hardware accelerators applications have seen similar performance gains with the number of floating point operations per second (FLOPS) growing by over 175% in the past 5 years. Despite this progress, the supposed gains offered by faster hardware and more efficient models are often not realized in deployment settings, where model speed has not improved as seen in Figure 1.

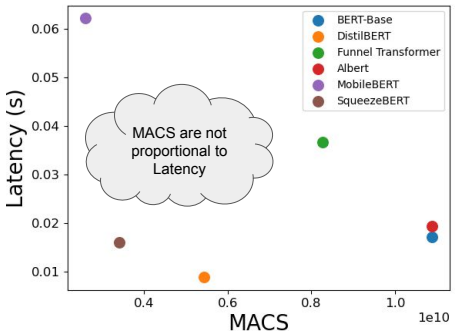

Figure 1. Latency as a function of a model's multiply accumulate operations (MACs) with PyTorch on an NVIDIA 2080ti GPU. Expected relationships do not hold; model complexity and hardware capabilities fail to predict latency due to framework-boundedness.

This misalignment is primarily attributable to overhead incurred by the machine learning frameworks used to implement and execute neural network models. As a manifestation of Amdahl's Law in machine learning systems, we see that despite increases in hardware speeds, the overhead introduced by commonly used deep learning frameworks during kernel dispatch operations is non-negligible and imposes bottlenecks on overall execution time. We refer to this phenomena as the *framework tax*, and show that it exists across deep learning framework paradigms (e.g. eager execution, just-in-time, and ahead-of-time compilation). Due to fixed framework overhead that dominates the execution at small batch sizes, proxy measurements of model efficiency, such as MACs and parameter count, are not predictive of inference latency. Moreover, as hardware performance increases and reduces the time for computational graph execution, machine learning systems are *increasingly* framework-bound at larger batch sizes.

[1]Carnegie Mellon University, Pittsburgh, PA, USA [2]FAIR, Menlo Park, CA, USA. Correspondence to: Jared Fernandez <jaredfern@cmu.edu>.

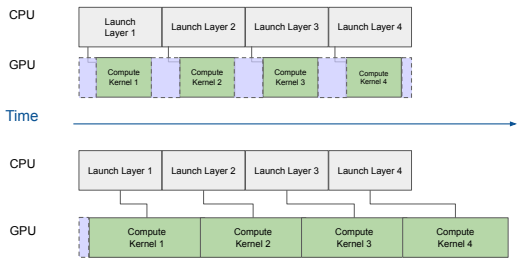

Figure 2. Profiles where execution is bound by CPU operations (above) and by GPU kernel operations (below). Small compute kernels are common at lower batch sizes and in inference. Boxes with dashed lines represent *framework overhead*.

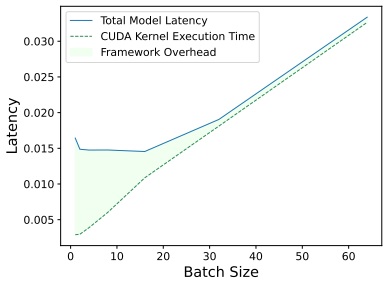

Figure 3. Framework overhead of ResNet-50 in PyTorch for various batch sizes on an RTX-8000. Framework overhead is substantial at low batch sizes but only a small constant at large batch sizes.

We identify common decisions that emerge in during the design of efficient model architectures, inference environments, and hardware accelerators and isolate choices likely to mislead a practitioner. We analyze the performance of convolutional neural networks (CNNs) and transformer models in eager execution PyTorch, just-in-time compiled TorchScript, and an ahead-of-time compiled ONNX runtime. We perform our study across seven different GPUs from the Pascal, Turing, and Ampere Nvidia GPU microarchitectures. We provide a series of recommendations for ML researchers and practitioners presented through a collection of case studies.

2. Framework Overhead

Deep learning frameworks often dispatch computation for asynchronous execution on highly parallelized hardware accelerators, as shown in Figure 3. For sufficiently large compute kernels, such as those seen during training, models achieve near maximum GPU utilization as measured by the difference between total execution time and active GPU time (Zhu et al., 2018). However, during inference, smaller input sizes frequently lead to suboptimal GPU utilization as the rapid executing kernels do not saturate the fixed cost framework overhead incurred from CPU operations (i.e. kernel launch, computational graph construction, control flow, and device synchronization; See Figure 2).

When the execution of GPU kernel computation is blocked by CPU framework operations such as kernel dispatches, the model's execution becomes *framework-bound*. In this setting, latency is constant regardless of batch size or number of MACs computed. For settings where latency is dependent on the execution of computational kernels and data movement, we refer to models as being *compute-bound*.

3. Experimental Setting

We evaluate on batch sizes from 1 to 128 and measure average latency of 100 forward passes after 10 warmup passes. GPU core and memory utilization are measured with the PyTorch profiler, Nvidia NSight, and NVIDIA Management Library (NVML), respectively. We randomly generate image data in the 224 x 224 RGB NCHW format, and language data as random embedding sequences of length 128.

4. Framework Considerations

Frameworks adhering to an eager mode of graph construction, such as PyTorch or TensorFlow 2.0, incur overhead from execution inside an interpreted environment given that the time between kernel launches is related to the speed at which kernels are enqueued, which is in turn affected by host-side latency. To mitigate this, inference frameworks such as TorchScript attempt to reduce interpreter overhead and introduce computation graph-level optimizations. Static runtimes can remove interpreter overhead entirely by compiling graphs ahead of time; the ONNX runtime provides further optimizations through graph rewrites that can fuse operators such as GEMMs and convolution layers with subsequent activations or norms. Such fusion operations can reduce memory footprint and result in more efficient kernels, decreasing overall framework overhead.

Unsurprisingly, model latency decreases when performing inference with frameworks and runtimes that support more aggressive optimizations as seen in Figure 4. These increases are especially pronounced at low batch sizes where inference is framework-bound.

For batch size 1, TorchScript and ONNX provide an average FP16 speed up of 34.16% and 71.38% over PyTorch, respectively. As batch sizes increase and models become compute bound, there is substantially less difference in latency across frameworks as the majority of execution time is spent awaiting the completion of kernel execution.

Although these speedups suggest that inference should always be executed in specialized inference runtimes, static construction of computational graphs may be slower when considering real input data where examples may differ in size, leading to input sparsity. We consider framework opti-

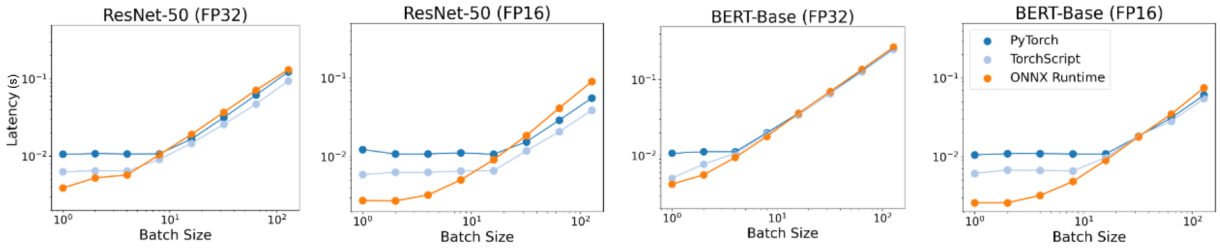

Figure 4. Latency vs. batch size for baseline models in FP16 and FP32 on RTX-8000. Framework boundedness exists for all models at small batch sizes where framework overhead dominates runtime regardless of Framework overhead is most prominent in smaller models executed in half precision (FP16) on slower frameworks.

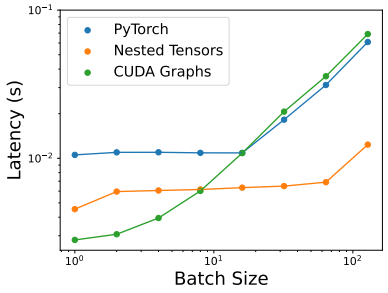

Figure 5. Different framework optimizations lead to latency improvements in different regimes. CUDA Graph kernel serialization reduces launch overhead in the framework bound regime, whereas sparse computation reduces latency at larger batch sizes.

mizations that utilize kernel serialization and sparse nested tensor operations in Figure 5. To construct sparse inputs, we simulate samples for sentence processing by generating variable length sequences and padding to the maximum sequence length of 128. Sentences are randomly generated according to the sequence length distribution of the Penn Treebank (Taylor et al., 2003), with an average length of 20.92 and a standard deviation of 10.18 tokens. We see that CUDA Graphs remove framework overhead at low batch sizes, but the efficiency gains from sparse tensor optimizations lead to latency reductions for larger batch sizes.

5. Model Design Decisions

We select ResNet-50 (He et al., 2016) and BERT-Base (Devlin et al., 2018) as representative architectures of the CNN and transformer architecture paradigms.

Although, the number of FLOPs required for model inference scales with the input batch size we observe that for small inputs latency is constant in Figure 4. Both ResNet-50 and BERT-Base exhibit framework boundedness for small inputs, where latency is constant regardless of batch size.

While execution in both single (FP32) and half precision (FP16) is framework bound, slower computation and increased data movement for FP32 execution leads single precision execution to become compute bound at lower batch size. At larger batch sizes, reduced FP16 precision yields expected reductions in latency as compared to FP32.

5.1. Scaling Model Depth & Width

Scaling model hidden dimensions and number of layers are common techniques for exploring tradeoffs between FLOPs and performance. For ResNet-50, we examine ResNet-18, 101, and 152 (He et al., 2016) and Wide ResNet-50 (Zagoruyko & Komodakis, 2016). Similarly, we compare 12-layer BERT-Base against 6-layer DistilBERT (Sanh et al., 2019) and variants of BERT with varying depth and width.

Increasing number of layers, leads to increases model latency in both the framework- and compute-bound regimes as each added layer operation requires an additional CPU-GPU dispatch, as seen in Figure 3.

Counterintuitively, increases in model width lead to *no increase in latency* at low batch sizes. In this setting, computation is framework bound and total runtime is constant despite the wider operations requiring more floating point operations. However, larger compute kernels causes these wider models to become compute-bound at lower batch sizes. Once compute-bound, latency scales more rapidly with batch size for wide models.

5.2. Downsampling and Hierarchical Pooling

Self-attention layers are notoriously expensive as their computational complexity scales quadratically with sequence length. Efficient transformer architectures often attempt to reduce these costs by downsampling the sequence to reduce length and total MACS (Dai et al., 2020; Kitaev et al., 2020; Wang et al., 2020a). We examine the performance of the Funnel Transformer, which applies average pooling every 4 layers, which attains comparable downstream performance to BERT with 42% fewer total MAC operations through sequence length reduction.

In Figure 8, we see that Funnel Transformer's reduction in total FLOPs does not translate to increased inference speed. Added pooling layers introduce additional steps to

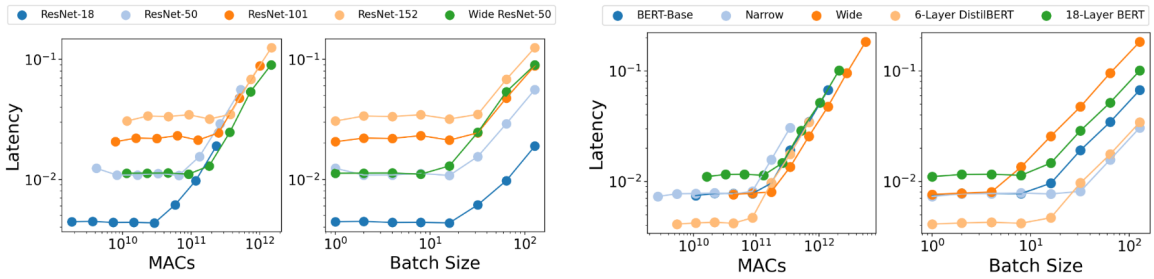

Figure 6. Comparison of latency for BERT and ResNet variants that scale model *width* and *depth*. Increases in model depth add more framework overhead, whereas increases in model width lead to faster transitions to compute boundedness. ResNet-50 and its Wide ResNet-50 variant are equivalent in latency up to batch size 8 despite having over 3x fewer parameters.

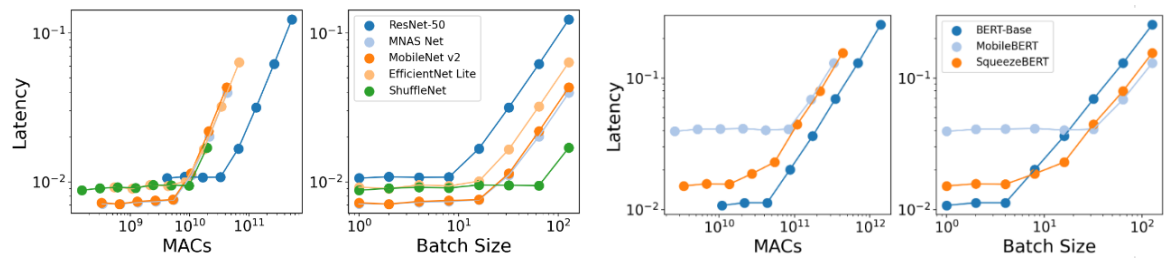

Figure 7. Latency of CNN and Transformer models using *efficient variations* of convolution and self-attention operations. All of the variants observe lower latency at large batch sizes, but have worse FLOP utilization. Efficient Transformer variants are slower than BERT at small batch sizes due to the introduction of more layers.

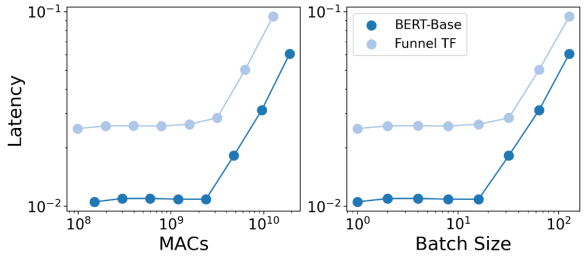

Figure 8. Comparison of latency for BERT and Funnel Transformer. Despite using fewer total MAC operations, Funnel Transformer is slower than BERT for inference due to the introduction of additional intermediate pooling layers.

the computation graph, and increase the model's framework overhead. Funnel Transformer is framework bound at a much higher latency than BERT at low batch sizes, and remains slower even at larger batch sizes.

5.3. Efficient Mobile Operations

Low-FLOP operations are commonly used as substitutes for dense convolution and linear operations in edge settings. In Figure 7, we examine models utilizing grouped convolutions: SqueezeNet (Iandola et al., 2016) and SqueezeBERT (Iandola et al., 2020); inverted residual bottlenecks MobileNet (Sandler et al., 2018), EfficientNet (Tan & Le, 2019), MNasNet (Tan et al., 2019), and MobileBERT (Sun et al., 2020).

While these operations reduce the number of FLOPs required for execution, they exhibit worse per-FLOP latency and poor memory utilization as compared to baseline models. In fact, for the case of the transformer architectures, both grouped convolutions and inverted bottleneck operations in these efficient architectures lead to worse latency than baselines. Similar to pooling operations, these operations introduces substantial framework overhead that slows down execution for low batch sizes.

6. Hardware Considerations

In Figure 9, we observe that framework-bound inference emerges across multiple generations of consumer, workstation, and datacenter Nvidia GPUs. As the speed of the accelerator increases, the relative execution speed of the compute kernels decreases while the total framework overhead due to CPU kernel dispatch operations remains constant. These observations indicate that framework bounds on model execution will continue to worsen as hardware improves unless commensurate improvements are made to deep learning frameworks.

7. Discussion

Computational graph optimizations and compilation improve latency. Removal of host language dependencies and graph optimizations provides substantial speedups over

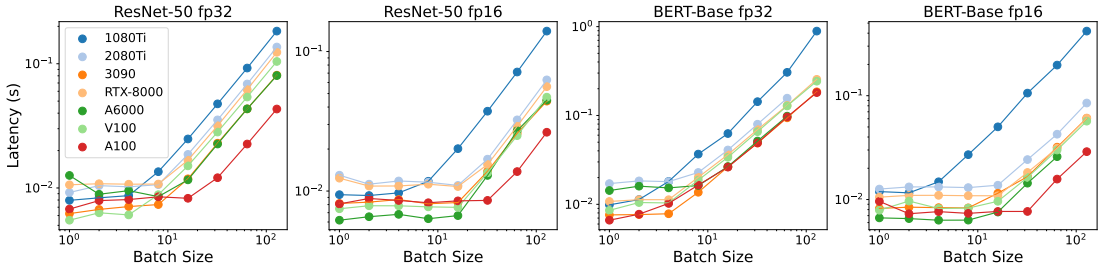

Figure 9. Framework overhead becomes increasingly prominently with newer generations of GPU. See Fig **??** in Appendix E for an alternative view featuring throughput across generations of GPUs

eager frameworks for inference at low batch sizes. However, feature completeness for operators and control flow varies across graph optimizers and compilers. For example, the FNet architecture (Lee-Thorp et al., 2021) relies on FFTs as a swap-in for self-attention. Currently, FFT operations are not currently supported by ONNX or TorchScript. As expected, FNet executed in PyTorch outperforms BERT executed in ONNX RT despite less framework overhead – with a 10.31% speedup at batch size 1.

Dynamic computational graphs can be faster for input sentences with variable lengths. Dynamic computational graphs can leverage input sparsity to reduce latency when processing variable length text. For example, PyTorch with sparse tensor optimizations reduces the latency of BERT-Base using static CUDA graphs by 80.56% at batch size 128 when processing sparse inputs.

At large input sizes, framework overhead from graph operations is negligible. For batch sizes larger than 16, we find there is minimal latency difference across models, inference runtimes, and frameworks. In the compute-bound regime, number of FLOPs is still a poor latency predictor due to variable execution time for different operations. For example, mobile architectures that depend on inverted-residual layers are memory inefficient and are much slower per-FLOP than standard convolution and GEMM layers.

For framework-bound models, model depth is a reasonable proxy for latency. Number of floating point operations is a poor indicator of latency in a framework-bound setting, as total runtime is generally constant and tied to framework overheads and the size of the computational graph. In framework-bound models, the size of the computational graph is related to model depth.

Estimations of latency for models deployed in production settings must account for their target framework and hardware platform. Model development frequently occurs using eager execution research frameworks. However, deployment often occurs in inference runtimes and on mobile devices or specialized hardware. This misalignment can mislead the development of "efficient" models and result in claimed gains that do not translate to real-world deployment settings. As such, researchers should be clear in specifying the setting of their "efficiency" gains, such as the target frameworks and hardware platform, when developing new methods. For example, techniques such as hardware-aware neural architecture search which leverage direct latency measures must also control for framework choices to account for this mismatch.

SA Dim	FC Dim	Batch	Seq Len	Latency	Throughput
768	3072	1	128	0.0136	0.0136
768	3072	4	128	0.0134	0.0034
768	3072	1	512	0.0134	0.0134
1536	6144	1	128	0.0134	0.0134

Table 1. Latency of BERT PyTorch models on RTX-8000. Scaling along batch sizes and model width shows no increase in latency.

Using higher-performing hardware does not necessarily improve end-to-end performance. Framework overhead limits the impact of improved hardware as it limits utilization. This trend will continue as ML-specific hardware advances without efforts to address software bottlenecks. For example, single-example inference with both BERT is slower using an A100 than using a V100 GPU despite a 2.75x increase in peak computational throughput.

8. Conclusions

In this work, we conduct an study of neural networks from the CNN and transformer architecture paradigms across a variety of software and hardware platforms. We show that inference performed with these large neural networks, which was previously assumed to be compute bounded, is in fact limited by overhead incurred by deep learning frameworks – a phenomena we refer to as the *framework tax*. In particular in the inference setting when compute kernels can be executed quickly, we show that existing methods of designing efficient model architectures are in fact limited by limitations in framework design.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Al-Rfou, R., Alain, G., Almahairi, A., Angermueller, C., Bahdanau, D., Ballas, N., Bastien, F., Bayer, J., Belikov, A., Belopolsky, A., et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv e-prints*, pp. arXiv–1605, 2016.

Barr, J. Amazon ec2 update-infl instances with aws inferentia chips for high performance cost-effective inferencing, 2019.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Chen, S., Wang, C., Chen, Z., Wu, Y., Liu, S., Chen, Z., Li, J., Kanda, N., Yoshioka, T., Xiao, X., et al. Wavlm: Large-scale self-supervised pre-training for full stack speech processing. *IEEE Journal of Selected Topics in Signal Processing*, 16(6):1505–1518, 2022.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, 2018.

Dai, Z., Lai, G., Yang, Y., and Le, Q. Funnel-transformer: Filtering out sequential redundancy for efficient language processing. *Advances in neural information processing systems*, 33:4271–4282, 2020.

Dehghani, M., Arnab, A., Beyer, L., Vaswani, A., and Tay, Y. The efficiency misnomer. *arXiv preprint arXiv:2110.12894*, 2021.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Gleeson, J., Gabel, M., Pekhimenko, G., de Lara, E., Krishnan, S., and Janapa Reddi, V. Rl-scope: Cross-stack profiling for deep reinforcement learning workloads. *Proceedings of Machine Learning and Systems*, 3:783–799, 2021.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Hooker, S. The hardware lottery. *Communications of the ACM*, 64(12):58–65, nov 2021. ISSN 0001-0782. doi: 10.1145/3467017. URL https://doi.org/10.1145/3467017.

Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

Iandola, F. N., Shaw, A. E., Krishna, R., and Keutzer, K. W. Squeezebert: What can computer vision teach nlp about efficient neural networks? *arXiv preprint arXiv:2006.11316*, 2020.

Janapa Reddi, V., Kanter, D., Mattson, P., Duke, J., Nguyen, T., Chukka, R., Shiring, K., Tan, K.-S., Charlebois, M., Chou, W., et al. Mlperf mobile inference benchmark: An industry-standard open-source machine learning benchmark for on-device ai. *Proceedings of Machine Learning and Systems*, 4:352–369, 2022.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678, 2014.

Kahn, J. D., Pratap, V., Likhomanenko, T., Xu, Q., Hannun, A., Cai, J., Tomasello, P., Lee, A., Grave, E., Avidov, G., et al. Flashlight: Enabling innovation in tools for machine learning. In *International Conference on Machine Learning*, pp. 10557–10574. PMLR, 2022.

Kitaev, N., Kaiser, Ł., and Levskaya, A. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

Lee-Thorp, J., Ainslie, J., Eckstein, I., and Ontanon, S. Fnet: Mixing tokens with fourier transforms. *arXiv preprint arXiv:2105.03824*, 2021.

Leopold, G. Aws to offer nvidia's t4 gpus for ai inferencing. *URL: https://web. archive. org/web/20220309000921/https://www. hpcwire. com/2019/03/19/aws-upgrades-its-gpu-backed-ai-inference-platform/(visited on 2022-04-19)*, 2019.

Lin, Z., Feng, L., Ardestani, E. K., Lee, J., Lundell, J., Kim, C., Kejariwal, A., and Owens, J. D. Building a performance model for deep learning recommendation model training on gpus. *arXiv preprint arXiv:2201.07821*, 2022.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Patterson, D., Gonzalez, J., Hölzle, U., Le, Q., Liang, C., Munguia, L.-M., Rothchild, D., So, D. R., Texier, M., and Dean, J. The carbon footprint of machine learning training will plateau, then shrink. *Computer*, 55(7):18–28, 2022. doi: 10.1109/MC.2022.3148714.

Reddi, V. J., Cheng, C., Kanter, D., Mattson, P., Schmuelling, G., Wu, C.-J., Anderson, B., Breughe, M., Charlebois, M., Chou, W., et al. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 446–459. IEEE, 2020.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

Sun, Z., Yu, H., Song, X., Liu, R., Yang, Y., and Zhou, D. Mobilebert: a compact task-agnostic bert for resource-limited devices. *arXiv preprint arXiv:2004.02984*, 2020.

Tan, M. and Le, Q. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pp. 6105–6114. PMLR, 2019.

Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., and Le, Q. V. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2820–2828, 2019.

Taylor, A., Marcus, M., and Santorini, B. The penn treebank: an overview. *Treebanks: Building and using parsed corpora*, pp. 5–22, 2003.

Tokui, S., Oono, K., Hido, S., and Clayton, J. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pp. 1–6, 2015.

Wang, S., Li, B. Z., Khabsa, M., Fang, H., and Ma, H. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020a.

Wang, Y., Wei, G.-Y., and Brooks, D. A systematic methodology for analysis of deep learning hardware and software platforms. *Proceedings of Machine Learning and Systems*, 2:30–43, 2020b.

Wu, C.-J., Raghavendra, R., Gupta, U., Acun, B., Ardalani, N., Maeng, K., Chang, G., Aga, F., Huang, J., Bai, C., Gschwind, M., Gupta, A., Ott, M., Melnikov, A., Candido, S., Brooks, D., Chauhan, G., Lee, B., Lee, H.-H., Akyildiz, B., Balandat, M., Spisak, J., Jain, R., Rabbat, M., and Hazelwood, K. Sustainable ai: Environmental implications, challenges and opportunities. In Marculescu, D., Chi, Y., and Wu, C. (eds.), *Proceedings of Machine Learning and Systems*, volume 4, pp. 795–813, 2022. URL https://proceedings.mlsys.org/paper/2022/file/ed3d2c21991e3bef5e069713af9fa6ca-Paper.pdf.

Zagoruyko, S. and Komodakis, N. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

Zhang, X., Zhou, X., Lin, M., and Sun, J. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 6848–6856, 2018.

Zhou, X., Chen, Z., Jin, X., and Wang, W. Y. Hulk: An energy efficiency benchmark platform for responsible natural language processing. *arXiv preprint arXiv:2002.05829*, 2020.

Zhu, H., Akrout, M., Zheng, B., Pelegris, A., Phanishayee, A., Schroeder, B., and Pekhimenko, G. Tbd: Benchmarking and analyzing deep neural network training. *arXiv preprint arXiv:1803.06905*, 2018.

Zhu, H., Phanishayee, A., and Pekhimenko, G. Daydream: Accurately estimating the efficacy of optimizations for {DNN} training. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 337–352, 2020.

A. Implementation Details

CPU	GPU	GPU Arch	Core Count	Tensor Cores	Clock Rate (GHz)	Memory (GB)	Mem BW (GB/s)	FP16 TFLOPS
Intel Xeon Silver 4110	**1080Ti**	Pascal	3584	-	1.38	11	484.4	22.78
Intel Xeon Gold 6242	**V100**	Volta	5120	640	1.23	32	897	28.26
Intel Xeon E5-2630	**2080Ti**	Turing	3584	544	1.35	11	616	26.90
Intel Xeon E5-2630	**RTX-8000**	Turing	4608	576	1.40	48	672	32.62
AMD EPYC 7282	**3090**	Ampere	10496	328	1.35	24	936	35.89
Intel Xeon Silver 4110	**A6000**	Ampere	10752	336	1.41	48	768	38.71
Intel Xeon 8339HC	**A100**	Ampere	6912	432	1.215	40	1935	77.97

Table A1. Details on hardware platforms used in our experiments, ordered by Nvidia microarchitecture generation.

Framework	Graph Compilation	Kernel Serialization	Latency	SMs Active	Warp Occupancy
PyTorch	None	None	10.54 ms	2.6%	0.9%
PyTorch with TorchScript	Just-in-Time	None	6.14 ms	18.5%	3.0%
PyTorch with CUDA Graphs	None	Yes	2.82 ms	57%	9.2%
ONNX RT	Ahead-of-Time	None	2.56 ms	22.3%	9.5%
ONNX RT with CUDA Graphs	Ahead-of-Time	Yes	2.11 ms	59%	20.3%

Table A2. Comparison of SM Activity Across Frameworks for BERT-Base at batch size 1.

Full hardware details are described in Table A1. We use PyTorch 1.12.1 with CUDA 11.6 and Python 3.8.13. We use ONNX Runtime 1.7.0 with CUDA 11.1.1 and cuDNN v8.0.4.3. Baseline experiments are run on a compute node with an Nvidia RTX-8000 GPU and an Intel Xeon E5-2630 CPU with 32 GB of DDRAM memory.

B. Results Using Precompiled CUDA Graphs

CUDA Graphs are an optimization that enable compilation and serialization of multiple CUDA kernels and remove the need for repeated kernel dispatch calls. As such they can significantly reduce framework overhead, however they still observe low hardware utilization.

Batch Size	Latency (ms)	SMs Active	Warp Occupancy	Memory Throughput (R — W)
1	2.81	57.0%	9.2%	5.3% — 2.3%
2	3.07	63.4%	17.6%	13% — 8%
4	3.95	64.4%	25.6%	9.5% — 9.2%
8	6.02	73.4%	32.4%	13.0% — 13.1%
16	10.84	78.1%	39.6%	16.5% — 15.3%
32	20.65	82.8%	42.4%	21.7% — 16.5%
64	35.77	93.5%	47.9%	24.9%— 19.1%

Table A3. GPU Utilization of BERT-Base executed in PyTorch with CUDA Graphs on an RTX-8000.

C. Related Work

C.1. Neural Network Frameworks and Runtimes

Specialized frameworks enable execution of deep learning workloads by implementing and providing APIs for tensor operations, gradient calculation, and construction of computational graphs. Frameworks generally fall into the following design paradigms (Kahn et al., 2022):

- **Eager Execution**: A computational graph is constructed from a series of operations that are executed as soon as they are called from an interpreter. Examples include: PyTorch (Paszke et al., 2019), TensorFlow 2.0 Eager (Abadi et al., 2015), Chainer (Tokui et al., 2015).

- **Deferred Execution**: A series of operations are defined and executed on sample data to generate a dataflow graph that can then be just-in-time (JiT) compiled. Examples include: PyTorch TorchScript, Jax (Bradbury et al., 2018), Theano (Al-Rfou et al., 2016), Caffe (Jia et al., 2014).

Framework	Compilation	Kernel Serialization	Latency	SMs Active	Warp Occupancy
PyTorch	Eager	None	10.54 ms	2.6%	0.9%
PyTorch with TorchScript	Just-in-Time	None	6.14 ms	18.5%	3.0%
PyTorch with CUDA Graphs	Eager	Yes	2.82 ms	57%	9.2%
ONNX RT	Ahead-of-Time	None	2.56 ms	22.3%	9.5%
ONNX RT with CUDA Graphs	Ahead-of-Time	Yes	2.11 ms	59%	20.3%

Table A4. Comparison of SM Activity Across Frameworks for BERT-Base at batch size 1. CUDA Graphs enable higher GPU utilization but still observe low levels of SM activity and warp occupancy.

- **Static**: A computational graph is pre-defined, compiled, and executed inside a specialized runtime; allowing for aggressive, global ahead-of-time compiler (AoT) optimizations. Examples include: ONNX Runtime, TensorFlow 1.0, MXNet (Chen et al., 2015), Nvidia TensorRT, and TVM (Chen et al., 2018).

While naively, the deferred and static settings appear to provide obvious benefits in production settings – the machine learning research community continues to rely heavily on eager mode frameworks for their ease of use. This further exacerbates the community divide, where models are designed under different assumptions than deployed.

Deferred and static execution frameworks leverage JiT or AoT compilation of their computational graphs allowing for optimizations such as operator fusion. However, they are limited in model expressivity due to reduced support for structures such as control flow and dynamic input sizes. In contrast, models under the eager execution paradigm allow for more flexibility but incur larger amounts of framework overhead running inside an interpreted environment.

C.2. Efficiency Metrics & Efficient Model Design

To measure model efficiency, metrics intrinsic to the model architecture are often reported such as the number of trainable parameters, number of floating point (FLOPs) or multiply-accumulate (MACs) operations. Number of trainable parameters is a frequently reported metric as a proxy for memory usage. However, parameter count is often not predictive of model speed (Zhou et al., 2020) and techniques such as weight tying can reduce in parameter counts without reducing the amount of computation (Lan et al., 2019).

Optimizing for FLOP counts has motivated the development of model architectures that achieve comparable task performance under a fixed FLOP budget (Dai et al., 2020) and scaling model width and depth to reduce overall size (Sanh et al., 2019). Low-FLOP operations, such as grouped convolutions (Iandola et al., 2016; Zhang et al., 2018) and inverted residual bottlenecks (Sandler et al., 2018; Sun et al., 2020), have been developed as substitutes for their standard, dense counterparts. However, the tensor operation primitives and underlying hardware support varying levels of parallelism for different arithmetic operations, leading to low correlation between FLOPs and latency.

Previous works examining the relationship between the mentioned efficiency measures have shown that these metrics often do not correlate well with each other, nor with direct measurements of latency in large scale training (Dehghani et al., 2021; Wang et al., 2020b). We extend this analysis to this inference setting and show that some of these metrics breakdown entirely due to additional performance bottlenecks that emerge at different scales of computation.

C.3. Hardware and Platform Performance Analysis

Efforts to establish common benchmarks leverage reference models and hardware platforms with target latency or accuracy (Reddi et al., 2020; Janapa Reddi et al., 2022; Zhou et al., 2020). Although these efforts have led to improvement in end-to-end latency, they often abstract away the underlying frameworks, compilers, backends, and hardware platforms. While general purpose improvements in hardware and software kernels may lead to improvements across all models, it has been argued that solely focusing on performance optimization of a limited set of model architectures and runtimes may lead to overspecialization (Hooker, 2021).

Previous analysis of the computational properties of hardware accelerators has largely focused on the training setting in which larger kernels and batch sizes hide framework overhead that emerges in the inference setting (Wang et al., 2020b; Zhu et al., 2020; 2018). Analyses of overhead in machine learning systems has primarily focused on domain-specific applications, such as reinforcement learning and recommendation systems settings (Gleeson et al., 2021; Lin et al., 2022), where simulation and memory access dominate execution time. These prior efforts are largely restricted to a small set of

reference model architectures and have not directly examined the relationship between model and platform in inference across scale.

D. Additional Experiments: Speech

In Figure 10, we examine the behavior of the WavLM (Chen et al., 2022) model which consists of a CNN encoder followed by transformer encoder layers. Audio inputs are simulated as 2 second sequences sampled at 16 kHz to create 32,000-dimensional floating point inputs. As with transformers and CNNS, we observe that WavLM exhibits framework bound behavior but quickly transitions to being compute-bound due to the large audio sequence lengths in Figure 10,

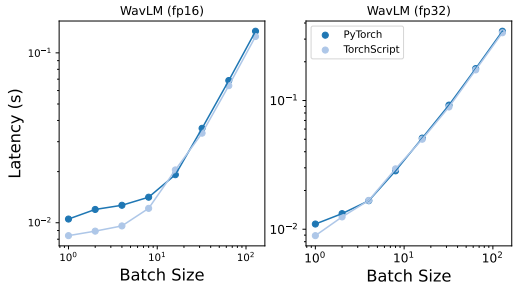

Figure 10. Transformer-based speech models exhibit framework boundedness but transition to compute-bound at small batch sizes due to long sequence lengths.

E. Additional Figures

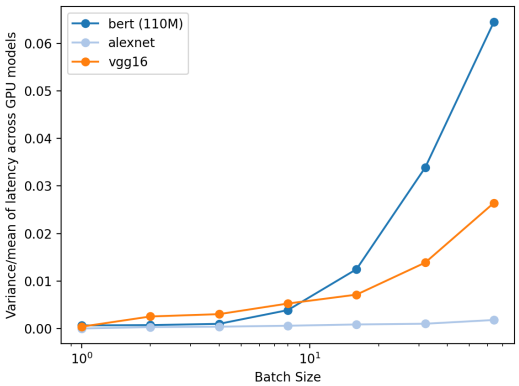

Figure 11. Framework boundedness is apparent at low batch sizes, as latency is consistent across accelerators. For larger models, framework boundedness dissipates for smaller bath sizes as inference becomes compute bound. Hardware used are the same as prior benchmarks.