Syncfree optimizers and compiler improvements for efficient model training

Shreyas Subramanian
*Amazon Web Services**
Seattle, USA
subshrey@amazon.com

Corey Barrett
*Amazon Web Services**
Seattle, USA
corbarre@amazon.com

Yanming Wang
*Amazon Web Services**
Seattle, USA
yanmwang@amazon.com

Ricky Das
*Amazon Web Services**
Seattle, USA
dasritwi@amazon.com

Harish Tummalacherla
*Amazon Web Services**
Seattle, USA
hartum@amazon.com

Lokeshwaran Ravi
*Amazon Web Services**
Seattle, USA
lokravi@amazon.com

Vinay Kumar Burugu
*Amazon Web Services**
Seattle, USA
vinaybur@amazon.com

Yi-Hsiang Lai
*Amazon Web Services**
Seattle, USA
yihsian@amazon.com

***Abstract*—Deep learning training compilers accelerate and achieve more resource-efficient training. We present a deep learning compiler for training consisting of three main features, a syncfree optimizer, compiler caching and multi-threaded execution. We demonstrate speedups for common language and vision problems against native and XLA baselines implemented in PyTorch.**

***Index Terms*—Training, compilers, deep learning**

I. Introduction

Efficient utilization of neural networks during training and inference, has long been a challenging endeavor. The advent of the transformer model architecture revolutionized the potential of sequence-based neural networks. Thus, facilitating parallelization and distribution techniques for gradient descent on a broader scale. However, the substantial growth in model size associated with this progress has given rise to predicaments concerning GPU availability and memory. In particular, numerous models encounter issues related to either their large size surpassing the capacity of a single GPU's memory or may have less optimal graphs resulting in lower utilization of the allocated GPU compute resources available. Pre-trained language and vision models can vary greatly in size, further adding to the complexity of this challenge. For example, while CLIP [1] uses a base of 63 million parameters, CoCa [2] leverages 2.6 billion parameters, yet masked VLM [3] is only 4 million parameters. The performance advantages of smaller models is clear, and yet the potential for higher accuracy provided by larger models remains a research area, as presented in GPT-3 [4].

Compilation has the potential to provide the best of both worlds, reducing the overall GPU memory footprint of the neural network without sacrificing accuracy. In this work, we propose a specific set of compilation optimizations to speed up the GPU training of large models. We present a strong case where it is clearly a more optimal use of resources to leverage compilation for existing models and showcase training time speedups in all configuration settings.

* All work was done when authors were affiliated with Amazon Web Services (AWS), Seattle, USA

II. Related Work

There are a variety of methods available to reduce the GPU memory footprint of deep learning models in order to accelerate training. Here, we only focus on compilation-based approaches, and not other techniques to improve model training efficiency such as distillation, quantization, pruning, data or model parallel training. Recently, compilers have been used in machine learning to more efficiently utilize the available hardware. Compilers can combine sets of elementary operations into a single operation, greatly improving the execution speed. Generally, the compiler framework processing flow for machine learning models can be divided into five layers [5]. These layers consist of front end, intermediate representation, high level optimization, low level optimization and back end. Several compilers have been developed recently that differ in how they implement one or more of the above mentioned layers. Halide [6] is one of the earliest compilers for machine learning models, that was originally developed for image processing. Portability in Halide is achieved by applying efficient scheduling methods and high level abstraction. An optimized pipeline is generated by the Halide autoscheduler, that can be adapted to Xeon and ARM CPUs as well as GPUs.

TVM [7] is another compiler framework for mapping deep neural networks to low level optimized code. TVM included high-level intermediate representations similar to a programming language instead of data flow representations, and other optimizations such as the combination of adjacent operators which reduce data transfer between on-chip buffer and off-chip memory, data layout transformations, and automation of operator level optimizations for improving GPU and specialized accelerator performance. This stack of optimizations are

specific to TVM, and the interplay between these optimizations are important in deciding how final training performance is affected. Other works such as DLVM [8], introduce various optimizations in their compiler stack such as a graph-based intermediate representation that optimizes linear algebra steps, linear algebra fusion, matrix multiplication reordering and some additional compiler optimization steps. Once again, we note that these are very specific choices that make up the entire compilation stack.

Tensor Comprehension [9] is another deep learning compiler framework, that introduces an expressive domain-specific language that enables the compilation flow to generate highly optimized GPU code automatically. This is achieved by mapping the high-level representation into a polyhedral model for exploring optimization scheduling. nGraph [10] utilizes efficient memory management and data layout abstraction for compiler optimization. An nGraph framework bridge utilizes a transformer for compiling or interpreting the intermediate representations resulting in highly optimized code for a specific back-end. Glow [11] is another machine learning compiler for heterogeneous hardware that uses a two-phase intermediate representation. Domain specific optimizations are carried out by the optimizer using the high-level intermediate representations, whereas the compiler utilizes the low-level intermediate representations to perform optimizations such as static memory allocation, copy elimination and instruction scheduling.

Another popular compiler for neural networks is the XLA (Accelerated Linear Algebra) project for TensorFlow and PyTorch [12]. XLA relies on JIT compilation (just-in-time) techniques to analyze the user-provided graph at run-time. For the interested reader, [5], [13] provide comprehensive surveys of machine learning compilers that provide additional reading on various existing machine learning compilers. The aforementioned compiler options are popularly used in practice as "black box" compilers. Generally, practitioners simply conduct several experiments with these options to decide which compiler to use, and it is unclear if a specific combination of steps represented in a chosen compilation stack can work well for all kinds of data and task types.

The rest of this paper is organized as follows - In the next section we describe the three main features of our custom compiler. We then provide feature specific results and end-to-end results.

III. Methods

Our custom compiler is comprised of the following three main features:

1) Syncfree Optimizers
2) Compilation Caching
3) Linking Parallelization

In the sections that follow, we dive deeper into the above three features individually. We then show experimental results that showcase speedups achieved for 1) individual features mentioned above, and 2) end-to-end speedup results where all features are used in our compile stack across different models.

A. Syncfree Optimizers

XLA [12] by default is used along with native optimizers. PyTorch native optimizers do an *inf/NaN* check before the optimizer step call. However this causes an extra CPU synchronization step. In the case of XLA, this CPU sync leads to an execution of the lazy tensor graph sequence, which leads to an average increase of 20% in training time latency across multiple models in two different domains, namely NLP and CV. PyTorch native optimizers don't use CUDA kernels with the in-kernel short-circuit prologue, so they need the CPU-based control flow decision with a syncing $.item()$ call. This $.item()$ call causes overheads and overall slower training times. To resolve this issue our solution proposes the following *sync-free* optimizer. The optimizer's native CUDA kernels must take an additional pointer that points to the GPU memory of a one-element found_inf device tensor, and then the prologue of each optimizer kernel checks this value on the device and internally makes the decision to skip or run the actual check. This way, the CPU Python code can enqueue *inf/NaN* checking ops that populate found_inf on the device. The sync-free optimizer kernel uses found_inf tensor to make the right decision by checking its device pointer's value. In other words, the CPU can blindly enqueue an inf-check operation followed by $optimizer.step()$ every iteration and allow the kernels to either skip or run the check, without needing a CPU-side control flow decision.

B. Compilation Caching

XLA optimization primarily works by compilation for the first few examples, and then ideally provides a speedup on the training data after it. This implies that it takes a larger number of training examples for XLA to come to parity with the native framework and can only exceed in performance afterwards. Therefore, it has a compilation overhead, without which the training throughput would be much higher from the first example. For distributed training, the compilation time is multiplied by the number of GPUs as this step is repeated for every GPU. Therefore we propose a methodology to reduce the compilation overhead to speedup the overall training run. We can reasonable assume that with distributed training, the final compiled graph from each instance should be the same. Thus, compilation caching for distributed training is essentially the same as caching for a single process. Fig 1 shows the time taken for compilation pipeline steps in XLA.

We observe that:

1) There is no significant difference When comparing High Level Operation (HLO) IRs before and after each compilation step.
2) When considering all steps until the buffer assignment step, the only major difference are the cuBLAS GEMM instructions
3) The difference in these instructions do not affect correctness or overall performance

With this in mind, we compute pre-compiled caches and insert them as shown in Fig 2. These pre-compiled caches can

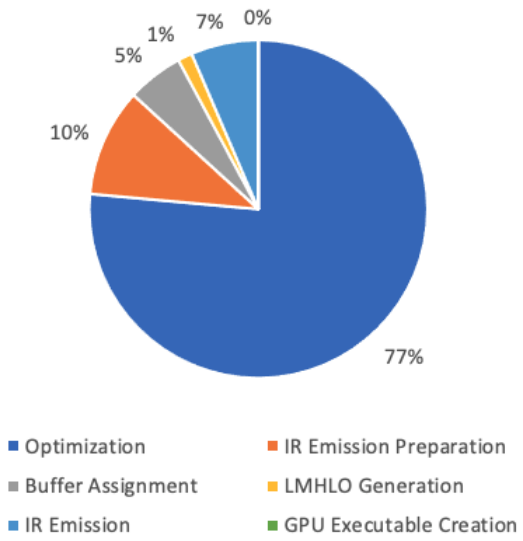

Fig. 1. Time distribution across different XLA GPU pipeline Steps

be further saved and reused into our future training workflows to cut down our compilation overhead.

C. Linking Parallelization

Building upon the previous optimization, we further optimize XLA runtime by multi-threading the compilation step in XLA. We restore the constant folding optimization to achieve a performance uplift across different models.

IV. Results

In this section we present our results based on modifications to the open-source PyTorch XLA compiler with the three main features explained previously - syncfree optimizer, compilation caching and linking parallelization. We first break down the speed-up and test individual components before testing end-to-end speedup results. We conduct experiments across multiple NLP and CV models, multi-GPU and multi-node multi-GPU types. For all of our end-to-end speedup experiments on NLP models, except facebook/bart-base, we use a sequence length of 128 on the *wikitext* dataset for 50 epochs. For end-to-end speedups with facebook/bart-base we use the xsum summarization dataset for 1 epoch. Finally, we use *Automatic Mixed Precision(AMP)* for all of our experiments.

Table I shows the overall performance uplift against the native PyTorch framework, for the models tested, across two optimizers - SGD and Adam - both with and without *syncfree optimizer* change. We see speedups across all models and optimizers tested. We also perform an ablation study in table II with a batch size sweep for bert-base-uncased to further demonstrate the broad usefulness of this optimization across different batch sizes.

Second, we document the performance uplift before and after *compilation caching* in a distributed setting in table III. We see up to a 90% speedup in compliation across models tested.

TABLE I
COMPARISON OF XLA WITH SYNCFREE OPTIMIZERS V/S BASELINE XLA NATIVE PYTORCH FRAMEWORK ON A P3.2X LARGE(TESLA V100) INSTANCE

Model Name	**Speedup v/s XLA Baseline**		**Speedup v/s Native Frame-work**	
	SGD	***Adam***	***SGD***	***Adam***
albert-base-v2	1.33	1.35	1.62	1.66
distilgpt2	1.28	1.35	1.45	1.53
facebook/bart-base	1.06	1.09	1.23	1.14
gpt-2	1.11	1.14	1.87	1.94
roberta-base	1.14	1.15	1.34	1.36
bert-base-uncased	1.21	1.19	1.22	1.23
ViT	1.1	1.12	1.13	1.14
resnet-50	1.08	1.11	1.45	1.47
densenet121	1.18	1.17	1.09	1.06

TABLE II
COMPARISON OF XLA WITH SYNCFREE OPTIMIZERS V/S BASELINE XLA FOR BERT-BASE-UNCASED ACROSS MULTIPLE BATCH SIZES ON A P3.2X LARGE (TESLA V100) INSTANCE

Batch Size for Bert-Base Uncased	**Speedup v/s XLA Baseline**	
	SGD	**Adam**
1	1.29	1.35
2	1.32	1.48
4	1.41	1.46
8	1.58	1.58
12	1.66	1.60
16	1.71	1.69
20	1.7	1.63
24	1.6	1.6
28	1.53	1.51
32	1.51	1.46
64	1.28	1.29
96	1.21	1.20

TABLE III
COMPARISON OF XLA WITH COMPILATION CACHING V/S BASELINE XLA ON 4 NODES OF P3DN.24XLARGE - (8 TESLA V100 32GIB GPUS PER NODE)

Model Name	**Speedup in 1st Epoch**	**Speedup in 2nd Epoch**	**Compilation Improvement**
albert-base-v2	1.29	1.2	81%
gpt2	1.59	1.63	56%
distilgpt2	1.7	1.72	68%
roberta-base	1.88	1.85	92%
ViT	1.54	1.48	62%
resnet-50	1.75	1.73	76%
densenet121	1.76	1.7	83%

Third, as shown in the Table IV that measures the benefits of linking parallelization, we are able to see up to 50% compilation time speedup across models such as Roberta-base, GPT2, DistilGPT2, Albert-base V2, and CV models like ViT, Resnet-50 and Densenet121 with no loss of model performance. We also tested speedup when distributed training was run across

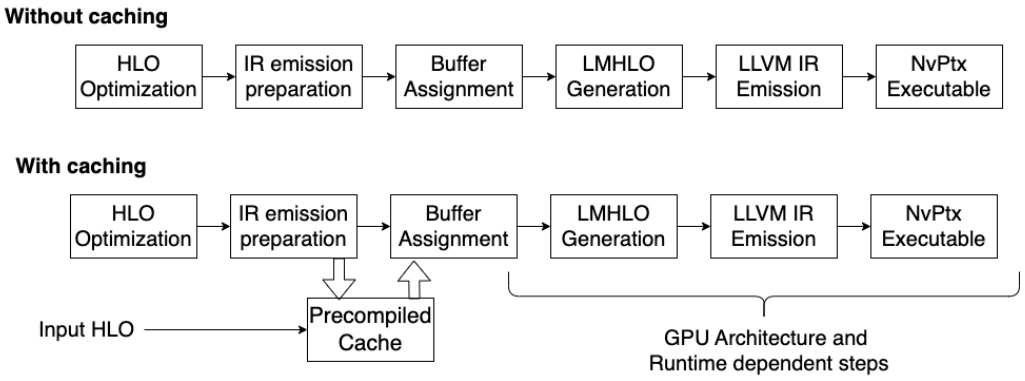

Fig. 2. Pipeline Difference between without caching(baseline) and with caching(ours)

TABLE IV
COMPARISON OF XLA WITH LINKING PARALLELIZATION OPTIMIZATION V/S BASELINE XLA ON A P3.16X LARGE MEASURED PER GPU IN SECONDS

Model Name	Linking parallelization run-time/GPU (in sec)	Baseline run-time/GPU (in sec)	Improvement %
albert-base-v2	139	168	21%
gpt2	218	337	55%
distilgpt2	132	198	50%
facebook/bart-base	346	492	42%
roberta-base	416	567	36%
ViT	237	257	8%
resnet-50	319	338	6%
densenet121	390	394	1%

multiple instances with Elastic Fabric Adapter (EFA).[1] We see roughly a 1.4x speedup (geometric mean) across a variety of NLP workloads. Table V shows the performance gains as well as comparison with the native framework with DDP.

TABLE V
COMPARISON OF XLA WITH EFA IMPROVEMENT V/S BASELINE XLA AND NATIVE PYTORCH FRAMEWORK ON 4 P4D.24XLARGE(A100 GPU) INSTANCES

Model Name	Speedup v/s XLA Baseline	Speedup v/s Native Framework
albert-base-v2	1.47	1.53
gpt-2	1.33	1.11
distilgpt2	1.25	1.07
roberta-base	1.32	1.15
ViT	1.55	1.27
resnet-50	1.4	1.48
densenet121	1.37	1.24

Finally we conducted a comparison against the recently released PyTorch 2.0 compile stack. The PyTorch 2.0 compile stack involves multiple tools - TorchDynamo, AOT Autograd, PrimTorch and TorchInductor. More information about PyTorch 2.0 can be found here - https://pytorch.org/get-started/pytorch-2.0/. We conducted several experiments and measured end-to-end speed up. Table VI shows results for fixed batch size tests. We observe that based on the limited tests conducted, we outperform Pytorch 2.0's Inductor-based compilation on some experiments, while Inductor outperforms our stack in others in an almost 50-50 split, within the limited set of experiments we conducted. We hope that this encourages researchers to try to experiment with complimentary approaches to obtain an even higher speedup overall.

TABLE VI
FIXED BATCH SIZE EXPERIMENTS COMPARING TORCH INDUCTOR TO OUR COMPILATION STACK USING PYTORCH 2.0

Model name	Batch size	Speedup over eager (Inductor)	Speedup over eager (ours)	Speedup (Ours vs Inductor)
bert-base-uncased	128	1.16x	1.13x	0.97x
camembert-base	128	1.16x	1.16x	0.99x
distilgpt2	128	1.48x	1.73x	1.17x
distilgpt2	144	1.48x	1.74x	1.17x
distilroberta-base	128	1.06x	1.25x	1.18x
EleutherAI/gpt-neo-125M	64	1.57x	1.84x	1.17x
EleutherAI/gpt-neo-125M	96	1.58x	1.94x	1.22x
facebook/bart-base	16	1.55x	0.76x	0.49x
google/electra-small-discriminator	256	0.88x	0.78x	0.88x
google/vit-base-patch16-224-in21k	128	1.08x	1.52x	1.4x
google/vit-base-patch16-224-in21k	176	1.06x	1.67x	1.56x
roberta-base	128	1.12x	0.69x	0.61x
t5-base	8	1.11x	0.89x	0.79x
xlnet-base-cased	128	1.48x	0.8x	0.54x

We also conduct experiments for appropriately tuned batch sizes for eager mode, torch inductor and our stack. Results of this experiment with a ml.g5.2xlarge instance can be seen in table VII. Similar experiments were run on an ml.p3.2xlarge instance; results can be seen in VIII below.

[1]EFA is Amazon's high bandwidth networking interface which features a custom OS bypass hardware interface using `libfabric`. https://aws.amazon.com/hpc/efa/

TABLE VII
TUNED BATCH SIZE EXPERIMENTS COMPARING TORCH INDUCTOR TO OUR COMPILATION STACK USING PYTORCH 2.0 ON A ML.G5.2XLARGE INSTANCE TYPE

Model Name	**Best Eager Batch Size**	**Best Inductor Batch Size**	**Best Batch Size (ours)**	**Eager Time To Train (seconds)**	**Speedup Over Eager (inductor)**	**Speedup Over Eager (ours)**
Albert-Base-V2	126	160	128	4894	1.87x	2.0x
Bert-Base-Uncased	160	192	128	3212	1.16x	1.12x
Bert-Large-Uncased	74	80	57	3701	1.15x	1.0x
Camembert-Base	160	186	128	4937	1.18x	1.14x
Distilbert-Base-Uncased	248	260	200	1919	1.1x	1.15x
Distilgpt2	128	159	159	3491	1.5x	1.79x
Distilroberta-Base	176	180	158	2587	1.07x	1.24x
EleutherAI/gpt-Neo-125M	100	112	124	3692	1.62x	1.95x
Facebook/bart-Base	19	24	15	3583	1.59x	1.24x
Google/electra-Small-Discriminator	256	288	192	1212	0.89x	0.98x
Google/mt5-Small	12	16	13	4162	1.58x	1.38x
Google/pegasus-Xsum	4	5	2	3857	1.44x	0.68x
Google/vit-Base-Patch16-224-In21k	128	96	56	3219	1.02x	1.12x
gpt2	64	87	70	4058	1.48x	1.3x
Microsoft/deberta-Base	60	60	77	2415	1x	1.02x
Microsoft/deberta-V3-Base	37	40	48	2923	0.89x	0.81x
Roberta-Base	128	145	116	3836	1.12x	1.12x
Roberta-Large	65	74	50	5117	1.16x	1.04x
Sentence-Transformers/ all-Mpnet-Base-V2	128	128	128	3381	1.14x	1x
T5-Base	10	12	6	4820	1.15x	1.36x
T5-Small	32	32	18	2482	1.12x	1.53x
Xlnet-Base-Cased	144	160	132	6183	1.52x	0.8x

TABLE VIII
TUNED BATCH SIZE EXPERIMENTS COMPARING TORCH INDUCTOR TO OUR COMPILATION STACK USING PYTORCH 2.0 ON A ML.P3.2XLARGE INSTANCE TYPE

Model name	**Best Eager batch size**	**Best Inductor batch size**	**Best batch size (ours)**	**Eager time to train (seconds)**	**Speedup over eager (inductor)**	**Speedup over eager (ours)**
albert-base-v2	94	116	106	3415	1.65x	1.72x
bert-base-uncased	114	132	70	2476	1.1x	0.79x
bert-large-uncased	48	54	35	2849	1.1x	0.65x
camembert-base	105	130	68	3837	1.06x	0.81x
distilbert-base-uncased	172	177	130	1492	1.06x	0.95x
distilgpt2	100	116	117	2582	1.36x	1.54x
distilroberta-base	122	128	104	2004	1.04x	0.93x
EleutherAI/gpt-neo-125M	62	77	82	2695	1.5x	1.66x
facebook/bart-large	6	8	5	3485	1.46x	0.9x
google/electra-small-discriminator	159	231	159	1378	1.03x	1x
google/mt5-small	11	15	12	3394	1.22x	0.95x
google/vit-base-patch16-224-in21k	182	188	177	2967	1.08x	1.59x
gpt2	101	128	108	5594	1.67x	1.73x
microsoft/deberta-base	79		98	3153	x	1.38x
microsoft/deberta-v3-base	52	60	64	3611	0.97x	1.18x
roberta-base	96	100	60	2968	1.08x	0.8x
roberta-large	40	46	32	3989	1.1x	0.7x
sentence-transformers/ all-mpnet-base-v2	81	131	-	2757	1.08x	1x
t5-small	22	25	13	1928	1.09x	0.91x
ViT	512	512	-	3436	1.0x	1x
xlnet-base-cased	96	112	70	4856	1.36x	0.35x

V. Conclusion

In this work, we introduced a new deep learning compilation stack with three key innovations - 1) syncfree optimizers, 2) compilation caching, and 3) linking parallelization - to accelerate training across a variety of models and tasks. Our experiments demonstrate substantial speedups compared to eager execution and other compilation stacks like TorchInductor.

Though no single stack dominates across all settings, our proposed techniques match or considerably outperform alternatives in the majority of experiments conducted. The performance gains highlight the promise of leveraging compilation techniques specialized for deep learning workloads. Further refinements to the optimizations and caching policies could expand the benefits to an even wider range of models and training scenarios.

References

[1] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever, "Learning transferable visual models from natural language supervision," *CoRR*, vol. abs/2103.00020, 2021. [Online]. Available: https://arxiv.org/abs/2103.00020

[2] J. Yu, Z. Wang, V. Vasudevan, L. Yeung, M. Seyedhosseini, and Y. Wu, "Coca: Contrastive captioners are image-text foundation models," 2022. [Online]. Available: https://arxiv.org/abs/2205.01917

[3] G. Kwon, Z. Cai, A. Ravichandran, E. Bas, R. Bhotika, and S. Soatto, "Masked vision and language modeling for multi-modal representation learning," 2022. [Online]. Available: https://arxiv.org/abs/2208.02131

[4] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[5] Y. Xing, J. Weng, Y. Wang, L. Sui, Y. Shan, and Y. Wang, "An in-depth comparison of compilers for deep neural networks on hardware," in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, 2019, pp. 1–8.

[6] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," vol. 48, 06 2013, pp. 519–530.

[7] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated End-to-End optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen

[8] R. Wei, L. Schwartz, and V. Adve, "Dlvm: A modern compiler infrastructure for deep learning systems," 2017. [Online]. Available: https://arxiv.org/abs/1711.03016

[9] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," 2018. [Online]. Available: https://arxiv.org/abs/1802.04730

[10] S. Cyphers, A. K. Bansal, A. Bhiwandiwalla, J. Bobba, M. Brookhart, A. Chakraborty, W. Constable, C. Convey, L. Cook, O. Kanawi, R. Kimball, J. Knight, N. Korovaiko, V. Kumar, Y. Lao, C. R. Lishka, J. Menon, J. Myers, S. A. Narayana, A. Procter, and T. J. Webb, "Intel ngraph: An intermediate representation, compiler, and executor for deep learning," 2018. [Online]. Available: https://arxiv.org/abs/1801.08058

[11] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, J. Montgomery, B. Maher, S. Nadathur, J. Olesen, J. Park, A. Rakhov, M. Smelyanskiy, and M. Wang, "Glow: Graph lowering compiler techniques for neural networks," 2018. [Online]. Available: https://arxiv.org/abs/1805.00907

[12] "XLA - tensorflow, compiled," https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html.

[13] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, "The deep learning compiler: A comprehensive survey," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 708–727, mar 2021. [Online]. Available: