Demystifying Deep Learning Compiler Optimizations for Training and Inference

Vishakha Agrawal

vishakha.research.id@gmail.com

Abstract

Deep learning has achieved tremendous success in recent years, powering many artificial intelligence applications. However, deep learning models are computationally intensive to train, requiring massive amounts of data and compute resources. Once trained, deep learning models need to be deployed for inference to make predictions on new data. Hardware used for training differs from hardware used for inference. Deep learning compilers have revolutionized the field of artificial intelligence by optimizing the performance of deep learning models on various hardware platforms. In the current landscape of research on deep learning compilers, there is a notable absence of comprehensive studies that specifically differentiate between compiler optimizations and methodologies for training versus inference. This paper provides detailed description of deep learning compiler optimization, focusing on training, inference separately. We investigate the challenges, opportunities, and design considerations for compilers targeting each phase.

Keywords: Training, Inference, Optimization, ASIC, Fusion, Quantization, Mixed Precision, Dynamic Batching, Pruning

I. INTRODUCTION

The rapid advancement of deep learning models has led to a diverse range of hardware platforms, including general purpose CPUs and GPUs, dedicated ASICs like Google's TPU, and neuromorphic hardware. While optimized linear algebra libraries and vendor-specific libraries have improved computational efficiency, they have limitations.

1) Challenges with Existing Libraries: Traditional libraries, such as BLAS, MKL, and cuBLAS, struggle to keep pace with the evolving deep learning landscape [8]. This results in:
    - Under-utilization of specialized hardware
    - Inefficient computation mapping
    - Limited performance and energy efficiency
2) The Role of Deep Learning Compilers: Domain specific compilers emerge as a solution to bridge the gap between deep learning models and di- verse hardware platforms. These compilers optimize deep learning models for various hardware envi- ronments, ensure performance and adaptability, fos- ter innovation and advancements in deep learning. Key benefits Deep learning compilers offer:
    - Seamless integration of hardware platform
    - Efficient computation mapping

- Optimal performance and energy efficiency
- Adaptability to evolving deep learning models

3) DEEP LEARNING COMPILER FOR TRAINING

Here's the detailed explanation of the deep learning training process, including the role of deep learning compilers(see Fig 1):

- The deep learning training process begins with Data Preparation [2], where relevant data is gathered from various sources, preprocessed to clean and normalize, and transformed into suitable formats. Data augmentation techniques are applied to increase dataset size, and feature engineering selects relevant features to improve model performance. Data is then split into training (80%), validation (10%), and testing (10%) sets.
- Model Architecture is defined, choosing a neural network type, number of layers and neurons, activation functions, and input/output shapes. Regularization techniques are considered to prevent overfitting [7]. A deep learning compiler, such as TVM or Tensor Comprehension, can optimize the model architecture by automatically generating efficient computational graphs and scheduling computations [3].
- During Model Initialization, initial values are set for model weights and biases using techniques like random or Xavier initialization. Deep learning compilers can further optimize initialization by providing efficient memory allocation and data layout transformations.
- The Forward Pass involves passing input data through the network, calculating output values, applying activation functions, and generating predicted outputs. Deep learning compilers can optimize the forward pass by fusing operations, eliminating unnecessary computations, and leveraging hardware-specific optimizations [3].
- Loss Calculation chooses a loss function, calculates the difference between predicted and actual outputs, and computes the loss value. Regularization terms may be added [5]. Deep learning compilers can optimize loss calculation by simplifying computational graphs and reducing memory access.
- The Backward Pass (backpropagation) computes gradients of the loss with respect to model parameters, applies the chain rule, and accumulates gradients for optimization. Deep learning compilers can optimize backpropagation by automatically generating efficient gradient computations and scheduling.

i. Optimization selects an optimization algorithm, updates model parameters based on gradients and learning rate, and adjusts the learning rate during training. Deep learning compilers can optimize optimization algorithm by providing efficient gradient accumulation and parameter update strategies. [1]

ii. Iteration process repeats the forward pass, loss calculation, backward pass, and optimization steps until convergence or a stopping criterion is reached. Deep learning compilers can optimize iteration by parallelizing computations, reducing memory overhead, and leveraging hardware-specific accelerations.

iii. Finally, Evaluation assesses the trained model's performance on unseen data, computes metrics, and selects the best-performing model.

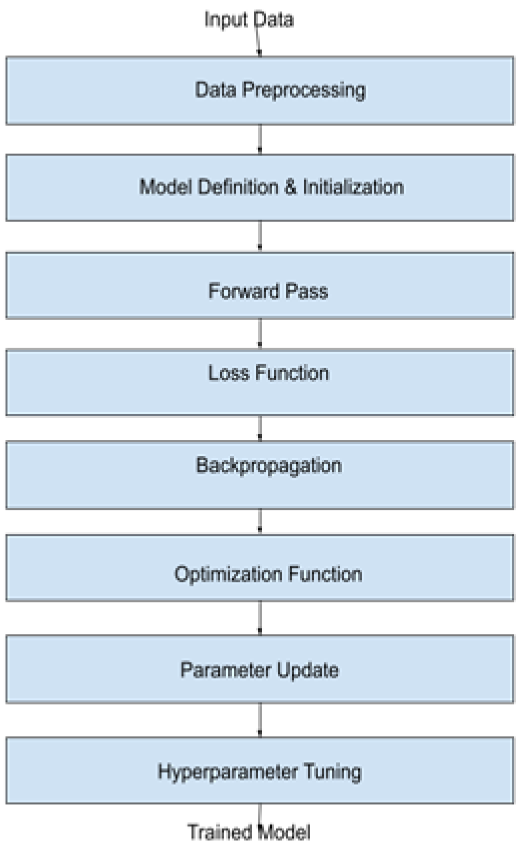

Fig. 1. Deep Learning Training Process

A. *Training Specific Compiler Optimizations*

When designing a compiler for training deep learning models, several key considerations must be taken into ac- count. Firstly, the compiler should efficiently handle com- plex computational graphs, optimizing operations such as convolution, matrix multiplication, and activation functions [8]. Additionally, memory management is crucial, requir- ing strategies for memory allocation, data layout transfor- mation, and reuse to minimize memory access overhead. The compiler should also leverage hardware specific optimizations for various architectures, including GPUs, TPUs, CPUs, and FPGAs. Furthermore, considerations for paral- lelization, distributed training, and asynchronous computation are essential for scalability. Support for diverse deep learn- ing frameworks, models, and data formats is also vital. Following are training specific compiler optimizations, that can greatly improve the performance:

- Automatic Mixed Precision: Compilers can automatically adjust data types during training to use lower precision (like FP16) for most computations while keeping critical calculations (like weight updates) in higher precision (FP32).It reduces memory bandwidth requirements and speeds up training on compatible hardware without sac- rificing model accurac [9].
- Memory Optimization: Memory optimization techniques are essential for efficient back-propagation in deep learn- ing: Gradient checkpointing: Stores intermediate gradi- ents at specific intervals, reducing overall memory us- age [4]. Memory reuse: Recycles memory buffers for gradient computations to minimize allocation overhead. These methods help deep learning compilers reduce memory consumption, enabling faster training and better performance on hardware with limited memory.

- Parallelization: Deep learning compilers employ vari- ous parallelization optimizations to enhance performance. These include data parallelism [12], which distributes training across multiple devices, and model parallelism [10], which splits the neural network itself. Pipeline parallelism [6] combines aspects of both by dividing the model into stages on different devices. Compiler techniques like operator fusion and vectorization improve efficiency, while GPU-specific optimizations leverage hardware capabilities. Distributed training coordinates work across multiple machines, and dynamic batching improves hardware utilization. By carefully implementing these parallelization strategies, deep learning compilers can dramatically improve the performance and scalability of neural network computations.

B. *Main Challenges of Deep Learning Compilers for Training*

Building effective deep learning compilers is a complex task, and several challenges must be addressed:

- Training often involves dynamic graph structures, espe- cially in models like RNNs or those with conditional computations. Compilers must handle graph mutations and runtime-dependent control flow, which is more com- plex than static inference graphs.
- Training requires storing intermediate activations for backpropagation, leading to high memory usage. Com- pilers must implement sophisticated memory reuse strate- gies and trade-offs between computation and memory.
- Optimizing for multi-device and multi-node training sce- narios. Compilers need to handle data parallelism, model parallelism, and efficient communication patterns.
- Handling variable batch sizes and input shapes during training. Compilers must generate flexible code that can adapt to runtime shape changes efficiently.
- Efficiently saving and restoring model states for long- running training jobs. Compilers need to optimize for quick serialization and de-serialization of large model states.
- Providing meaningful debug information and perfor- mance profiles for complex training graphs. Compilers must preserve source-level information and instrument code for detailed runtime analysis.
- Balancing compilation time with runtime performance, especially for rapidly iterating research models. Compil- ers must employ strategies to reduce compilation over- head while still producing highly optimized code.

II. Seamlessly integrating with popular deep learning frame- works and their ecosystems. Compilers need to support various front-ends and operator libraries used in training workflows.

DEEP LEARNING COMPILER FOR INFERENCE

Here’s the detailed explanation of the deep learning infer- ence process, including the role of deep learning compilers(see Fig 2)

- Input Preparation involves preprocessing input data, which may include resizing, normalization, and data augmentation. Deep learning compilers can optimize input preparation by automatically generating efficient preprocessing pipelines and leveraging hardware-specific optimizations.
- The deep learning inference process commences with Model Loading, where the trained

model is loaded into memory from storage, encompassing weights, biases, and architecture. A deep learning compiler plays a crucial role in this step by optimizing model loading through techniques like model pruning, quantization, and compression, reducing memory footprint and accelerating loading times.

- The Forward Pass step passes input data through the loaded model, performing computations layer by layer, including convolution, pooling, activation functions, and fully connected layers. Deep learning compilers optimize the forward pass by fusing operations, eliminating unnecessary computations, and scheduling computations to minimize memory access and maximize parallelization.
- The Output Generation step produces the final output, which may include class probabilities, regression values, or feature embeddings. Compilers can optimize output generation by simplifying computational graphs and re- ducing memory overhead. Optional post-processing steps are then applied, such as non-maximum suppression, bounding box regression, or output formatting. Deep learning compilers can accelerate post-processing by pro- viding optimized implementations of these techniques and leveraging hardware-specific accelerations.

The following analysis outlines the mechanisms involved in deep learning inference, with a specific focus on compiler infrastructure.

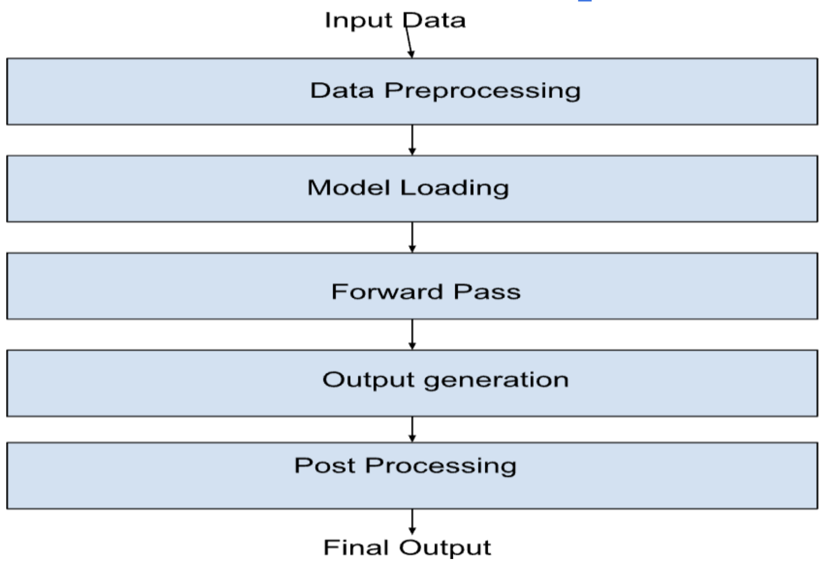

Fig. 2. Deep Learning Inference Process

A. *Inference Specific Compiler Optimizations*

When designing a compiler for inference deep learn- ing, several key considerations must be taken into ac- count. The compiler should efficiently handle complex neu- ral network architectures, optimizing computations, mem- ory access, and hardware utilization to minimize latency and maximize throughput. It should support various deep learning frameworks, models, and data formats, while also providing flexibility for customization and extension. Additionally, considerations for model pruning, quanti- zation, and knowledge distillation are essential for re- ducing computational overhead and memory footprint. The compiler should also leverage hardware-specific opti- mizations for CPUs, GPUs, TPUs, and other accelerators, and optimize for power consumption and thermal efficiency. Furthermore, support for dynamic batching [3] and operator fusion [11] can significantly improve inference performance. Effective compiler design requires balancing competing ob- jectives, including performance, power consumption, memory usage, and flexibility.

Compiler optimizations for deep learning Inference can vary depending on hardware i.e. whether is edge inference or cloud inference. Edge inference occurs on resource-constrained de- vices like smartphones, IoT devices, or embedded systems. While Cloud inference leverages powerful servers, often with specialized hardware.

When considering resource focus, edge computing emphasizes optimization for limited memory, computational power, and energy consumption, while cloud computing prioritizes high throughput and scalability. In terms of precision, edge systems often utilize lower precision formats like INT8 and FP16 to enhance efficiency, whereas cloud environments can afford to use higher precision formats such as FP32 and FP64 when accuracy is paramount.

Regarding model size, edge computing emphasizes model compression and efficiency to fit within constrained resources, while cloud infrastructure can accommodate larger and more complex models.

Additionally, edge computing prioritizes low latency to support real-time applications, whereas cloud systems typically focus on achieving high throughput by managing multiple inferences simultaneously. Finally, edge computing is tailored for specific devices, such as ARM processors and mobile GPUs, while cloud computing leverages powerful and often specialized hardware,including high-end GPUs, TPUs, and FPGAs. Here are edge inference-specific compiler optimizations:

- Weight Pruning Removing (or zeroing out) weights that have minimal impact on the model’s output, often using a threshold to determine which weights to prune, reduces model size and computation requirements, speeding up inference without significantly sacrificing accuracy.
- Quantization Converting model weights and activations from floating-point representation to lower-bit formats (e.g., INT8, FP16) to decrease memory usage and com- putational complexity, enables faster execution and lower power consumption on hardware that supports lower precision, significantly improving inference efficiency.
- Dynamic Input Optimization Compilers can optimize how models handle varying input sizes and shapes, allowing for more efficient memory usage and computation, en- hances the flexibility of models during inference, reducing overhead and improving throughput.

Here are Cloud Inference specific compiler optimizations: Cloud inference leverages powerful servers, often with specialized hardware. Optimizations focus on:

- Parallelism This includes data parallelism—where multi- ple inputs are processed simultaneously across different devices; model parallelism—where large models are split across multiple devices; and pipeline parallelism—where inference tasks are organized into pipelines to enhance efficiency.
- Accelerator-Specific Optimizations Tailors code genera- tion and execution strategies for specific hardware accelerators, such as GPUs and TPUs, leveraging their unique capabilities.
- Batch Processing: Batch processing is another impor- tant optimization strategy, featuring dynamic batching to group incoming requests for more efficient processing and kernel

optimization to enhance throughput when handling large batches.

- Scalability Scalability is also a significant focus; auto scaling allows resources to be adjusted dynamically based on the current load, while load balancing distributes infer- ence tasks across multiple servers to prevent bottlenecks.

B. *Main Challenges of Deep Learning Compilers for Inference*

Deep learning compilers for edge inference face unique challenges due to the constraints of edge devices and the requirements of real-time processing. Here are the main chal- lenges:

- With limited storage on the edge reducing model size while maintaining accuracy is a key challenge. Compilers need to implement and optimize various compression techniques like pruning, quantization, and knowledge distillation.
- Another one is targeting a wide range of edge de- vices with different architectures (CPUs, GPUs, NPUs, DSPs).Compilers must generate optimized code for di- verse hardware, often with limited documentation or support.
- In order to meet the strict real-time processing demands of many edge applications, compilers need to minimize overhead and optimize for low-latency execution paths.
- Fragmented Software Ecosystems: Dealing with diverse operating systems, drivers, and software stacks on edge devices. Compilers need to ensure compatibility and op- timal performance across various software environments.
- Providing effective debugging and profiling tools for edge deployments. Compilers must generate code that can be easily debugged and profiled with minimal overhead.

Cloud inference also poses several challenges for deep learning compilers, particularly in terms of optimization, per- formance, and security.

- Dynamic Model Updates and Versioning: Cloud infer- ence requires frequent model updates and versioning. Compilers must efficiently manage model updates, ensure backward compatibility, and optimize model execution for new versions.
- Real-time Monitoring and Analytics: Cloud inference demands real-time monitoring and analytics to ensure optimal model performance. Compilers must provide insights into model execution, latency, and throughput to facilitate optimization and debugging.
- Integration with Cloud Services and APIs: Cloud in- ference involves integrating with various cloud services and APIs. Compilers must provide seamless integration with these services to enable scalable and secure model deployment.
- Cost Optimization and Billing: Cloud environments incur costs based on resource utilization. Compilers must op- timize model execution to minimize costs, provide cost estimation and prediction, and integrate with cloud billing systems.

III. CONCLUSION

Despite the distinct computational demands, resource con- straints, and optimization strategies inherent to training and inference processes, there is a notable absence of literature that explicitly differentiates between compilers specialized for these two crucial phases of deep learning. This observation un- derscores the need for a more nuanced and targeted approach within the research community. While existing compilers often address both training and inference,the unique challenges posed by each phase warrant more focused investigation. Training, with its emphasis on back propagation, gradient computation, and iterative optimization, presents a different set of challenges compared to inference, which prioritizes low latency, energy efficiency, and deployment across diverse hardware platforms. Furthermore, the evolving landscape of deep learning applications, from edge computing to large- scale distributed systems, amplifies the importance of this distinction.

REFERENCES

[1] Mayukh Bhattacharyya. Regularization for deep learning: A taxonomy. *https://towardsdatascience.com/gradient-accumulation-overcoming- memory-constraints-in-deep-learning-36d411252d01*, 2020.

[2] Jason Brownlee. *Data preparation for machine learning: data cleaning, feature selection, and data transforms in Python*. Machine Learning Mastery, 2020.

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *USENIX Sympo- sium on Operating Systems Design and Implementation*, 2018.

[4] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[5] Tobias Grosser. Regularization for deep learning: A taxonomy. *Polyhe- dral Compilation. https://polyhedral.info.*, February 4, 2020.

[6] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. *GPipe: efficient training of giant neural networks using pipeline parallelism*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[7] Jan Kukacˇka, Vladimir Golkov, and Daniel Cremers. Regularization for deep learning: A taxonomy. *arXiv preprint arXiv:1710.10686*, 2017.

[8] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, 2020.

[9] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations*, 2018.

[10] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Prin- ciples*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Im- plementation*, PLDI 2021, page 883–898, New York, NY, USA, 2021. Association for Computing Machinery.

[12] Chao Peng, Tete Xiao, Zeming Li, Yuning Jiang, Xiangyu Zhang, Kai Jia, Gang Yu, and Jian

Sun. Megdet: A large mini-batch object detector. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6181–6189, 2018.