# A Guide to SW, Compiler & HW Optimizations for Deep Learning

Pascal Bastien
*Department of Electrical and Computer Engineering, Department of Computer Science*
*North Carolina State Universitiy*
Raleigh, USA
pbastie@ncsu.edu

*Abstract*— **In the last decade, machine learning (ML) has experienced great success in its application to various domains such as computer vision (CV), natural language processing (NLP), Generative AI, etc. From simple multi-layer perceptrons (MLPs) containing few layers, to deep neural networks (DNNs) containing as many as 100s of layers and growing, ML neural networks [1] (NN) have increased exponentially in complexity and in size, in the quest to reach accuracy in various tasks comparable to and, at time, better than humans. Such large NNs suffer from compute and memory intensive operations at their core which makes training and inferencing these NNs computationally challenging. To address this issue, there has been considerable efforts form both academia and industry to optimize the storage as well as the computational operations of these NNs to reduce memory footprint as well as speedup processing input data through the NN. These DNN optimizations have spanned multiple stacks; from the software algorithm, compiler, all the way down to the hardware, including dedicated FPGA-based accelerators. Such optimization techniques have been scattered and a comprehensive list seems to be missing. The closest survey of that sort is the work from Sze et al. [1]. Although it provides an overview on all the moving parts in efficient processing of DNNs, the work focuses mostly on hardware optimizations in the context of building NN hardware accelerators. In contrast, the goal of this work is to coalesce all of the best known software, compiler, and hardware optimization techniques into one document, providing a comprehensive coverage to explore past and state-of-the-art (SotA) ML optimization techniques for processing both training and inference, while maintaining accuracy. As such, a user can explore and integrate various optimizations to maximize performance out of their ML training workflow or inference out their end ML deployment product. Lastly, this work also discusses potential research opportunities in microarchitecture for direct implementation of some of these software and compiler optimizations.**

***Keywords— Deep Learning, Software, Compiler, Hardware, Optimization.***

## I. Introduction

DNNs contain multiple layers, each of which contains parameters such as weights and biases, and features such as input feature maps, and output feature maps. They are stored in structures called tensors, which can be thought of as multidimensional matrices/arrays. DNNs implement various computational operations but one of the main computations used are general matrix multiplications (GEMMs) which boil down to Multiply Accumulate (MACs) operations. For multi-dimensional structures such as tensors, these MACs operations can become extremely expensive, storage and compute wise, as the number of dimensions and the size of each dimension increase.

Multiple techniques have been studied by both academia and industry to reduce memory footprints of these tensors as well as speedup the total computations found in these DNNs. This paper focuses on three main levels in the stack for optimizations: software/algorithm, compiler/framework, and hardware, each of which will be discussed in detail in the upcoming sections.

Moreover, this paper primarily focuses on GEMM operations optimizations most densely found in CV models' CNNs structures. However, as underlying structures of DL models share similarities, optimizations techniques for CV could also be extended to other DL models for other tasks such as simple pattern recognizers, NLPs, multi-head dot-product attention mechanisms of Transformers for Generative AI, etc. Additionally, the techniques discussed in this work also covers optimizations for other non-GEMM operations of NNs such as activation and pooling functions, embeddings, etc.

[1]Neural network and model are used interchangeably throughout this paper.

Unlocking performance improvement for training vs inference can feel like a tug of war exercise. In most cases, to provide speedup at inference time, the optimizations applied may add extra steps during training. However, in some cases, since the model is only trained "once" and inferenced "indefinitely", incurring up-front cost during training for a more faster inferencing model, can be worth it, especially in mission critical edge devices where inference have hard time and size constraints requirements. Trying to increase the accuracy of the model also comes at the expense of a more complex training process.

In terms of optimizing for performance, the three aforementioned levels of optimizations usually get adjusted cohesively through the hardware-algorithm co-design practice. However, this work will address them separately as to provide clear boundaries for optimization in case individuals wanted to focus on implementing only specific sections of the stack for their use-case.

Lastly, this work discusses opportunities for hardware to directly implement some of the software and compiler optimizations. Moreover, this discussion will also cover ways that the hardware could theoretically do a much better job implementing these optimizations techniques/ideas.

## II. OPITIMZATIONS

The most readily available optimization a user may attain is through the formulation of their ML algorithm. Two software programs accomplishing the same task but implemented through two different algorithms can incur substantially different performance, both in terms of speed and energy efficiency. Moreover, an algorithm that worked best on one platform can perform worse than a "less efficient" algorithm if ran on a platform that the software architects did not design it for. This discrepancy in performance based on underlying hardware is primarily addressed by compilers. However, the different combinations are usually so vast that best optimization are usually expected at the user level, while the compiler picks up the rest. The following sections will cover the multiple optimization available for users at the software level for DL workloads. Next, the common optimizations performed by specialized DL compilers that users can leverage will be covered. And lastly, common hardware optimizations will be discussed a great deal.

## III. SOFTWARE/ALGORITHM OPITMIZATIONS

Users can impact the performance of their DL models most directly by crafting software and algorithms that keeps the use of memory and compute resources to a minimal. Some of the software/algorithm DL optimizations techniques are as follow:

**Compression** – As the number of layers of a DNN increases, so does the total size of its parameters such as the weights and biases. Introducing such large number of layers sometimes result in certain weights to not contribute much to the accuracy of the DNN. Such weights can be considered as redundant and unnecessary and therefore eligible to be completely removed from the network. Weight removal can be done by assigning said weight to zero, referred to as pruning. Pruning the network is also referred to as introducing sparsity to the network.

**Compressed Sparse Row (CSR)** - A sparse network can then be compressed, reformatted, and represented in new formats such as CSR, effectively reducing the memory footprint to represent the entire newly sparse network.

**Pruning** – Pruning is split into two categories: unstructured and structured, also referred to as fine- and coarse-grained pruning respectively. Unstructured pruning is done by setting specific weights to zero based on a certain threshold value, while structured pruning is setting an entire channel of a filter or an entire filter altogether, effectively preventing activation of the corresponding input feature map (ifmap).

**Magnitude-based Pruning** – The previously mentioned pruning can be considered as magnitude-based pruning since it is the absolute value (magnitude) of the weight that is compared to the zero-assignment threshold. The user could prune a percentage of the model removing the smallest magnitude weights up to that percentage threshold. Pruning can also be done standard deviation removing weights with a set standard deviation from the mean.

**Combinatorial-based Pruning** - Magnitude-based pruning removes weights based on their magnitude, i.e. their relative impact on the forward pass prediction of the network. However, since various combination of weights removed in one layer can differently impact the weights that need to be removed in subsequent layers, finding the ideal combination of weights to be removed is a combinatorial optimization problem. Therefore, although magnitude-based pruning which has been solved using Stochastic Gradient Descent (SGD) has resulted in large compression with minimal loss in accuracy, finding the actual ideal, maximally pruned network, is a combinatorial optimization problem and the de facto SGD can't be used to solve such optimization problems with such constraints. This is where ADMM-based pruning technique is helpful. Although ADMM convergence can require over 200 iterations, it can solve optimization problems with constraints. Numerous efforts to apply ADMM-based pruning to DL can be summarized in [4][5][6][7][8].

**Quantization** - Another software optimization technique for DL is quantization. Datatypes used to represent parameters and activations are reduced in their precision by choosing less bits, i.e. 16, 8, 4, or 3 bits instead of full-precision 32 bits representations. Work such as Deep Compression [9] demonstrate that accuracy of the model can still be maintained while quantizing the model down to 4-3 bits which can lead to a significant reduction in size of the model, up to 35x compression. Mixed precision can also be implemented where different layers in the model are assigned different precision. Moreover, different features of the model such as weight, biases, gradient, activations, etc., can be assigned different precisions [10]. New datatypes such as BF16 and FP16 have also been used to cover more range of decimals number per bits, at the cost of reduced precision.

Implementing a combination of magnitude-based pruning and quantization can provide up to 35x memory footprint reduction, and 3-3.5x speedup on CPU and GPU without compromising accuracy of the model [9].

**Distributed learning** - In addition to pruning models and compressing them which effectively reduces their memory footprint while also providing inference speedups, distributed learning techniques can be applied to parallelize model learning onto various compute engines which can tremendously speedup training of these models. PyTorch provides multi-processing libraries in the context of DL to facilitate weights and gradient sharing amongst processes. Going from one 1 process to 16 processes on a 16 core system can yield speedups of up to 5x. Extrapolating to systems with hundreds/thousands of cores could potentially yield to much greater speedup.

**Computation transforms** – Transforming a computation to a different domain can be better suited for the underlying hardware to process, while still preserving the arithmetic meaning and thus the correctness. Examples of such transform are the Fast-Fourier Transform (FFT), Toeplitz transforms, etc.

## IV. Compiler/Framework Opitmizations

Over the last decades, there has been numerous specialized DL compilers (or frameworks) development. For example, TVM, Tensorflow XLA, oneDNN, TensorRT, have added tremendous support for DL workloads compiler optimizations. More recently, the MLIR framework has been released. Although, not specific to DL, its ability to enable extra levels of IRs can be leveraged for DL optimization applications. This section covers the main optimization opportunities at the compiler level for DL.

**Computational Graph Transformation** – Once the model architecture is defined with its filters and activations shapes, and computation kernels, it can be transformed to and represented as a computational graph. The computational graph can then be saved in TorchScript which is a representation of a model initially described in PyTorch to a language agnostic intermediate representation. This intermediate representation enables graphical optimization for DL, of which the main ones implemented in TVM [3] are described as follow.

**Operator Fusion** – Operator fusion is the concept of fusing a kernel operation that usually follows another, such as activation functions, directly with the corresponding operation, forming a single operation. This reduces the need to fetch the same data multiple times from possibly distant memory, as well as execute an operation in one shot/cycle as opposed to two separate sequential operations. Operator fusion can provide between 1.2-2x speedup [3].

**Constant-Folding** – Certain variables to be computed are always constant no matter the state of the program at runtime. These variables can be precomputed at compile-time and replaced by the resulting constant. That way, computing that value only happens once as opposed to multiple times if it were computed at runtime.

**Static Memory Planning Pass** – This step statically allocates memory for tensors of the model. By "locking" these structures into specific areas of memory, the compiler can make certain assumption which could unlock greater performance during the compiler optimizations passes. Static memory planning can also avoid the need to dynamically allocate memory for the tensors as the model processing executes.

**Data Layout Transformation** – This step reorganizes the layout of data to maximize spatial and temporal locality of data depending on the underlying back-end hardware.

**Memory Latency Hiding** – This technique aims to overlap memory access and kernel execution to hide the relatively longer latency of data fetching from memory. This can be achieved manually by the user through multithreading as a software optimization. However, compilers such as TVM can also implement similar functionalities through its virtual thread lowering, leading to improvements pushing memory and compute bounds closer to roofline curve of the roofline model [3].

In addition to these out-of-the-box optimizations, these DL compilers also provide libraries that can enable further custom optimization for unique scenarios.

## V. Hardware Opitmizations

Out of all the optimization techniques mentioned thus far, the most intuitive way to get performance gains, whether for training or inference, is to leverage Graphical Processing Units (GPUs). GPUs, equipped with multiple processing elements much smaller than CPU cores, were traditionally designed to relieve CPUs of the burden of performing graphics rendering made of computations with parallelizable nature that the CPU cannot exploit. Because of their dominance in processing highly parallelizable workloads, GPUs have been adopted by the NN community to accelerate operations on tensors which share similarities with graphics rendering.

There are numerous avenues to get performance gains in DNN processing from hardware optimizations. Most of these techniques are already highlighted concisely in EPoDNN [1]. Below is a mere summary of the techniques most appropriate for this primer on optimizations for DL.

**Data Reuse** – This technique refers to reusing data as much as possible after it is fetched from memory. Memory operations are in order of magnitudes more expensive relative to computations. The further away data must be fetched from, the higher the cost. As a result, the main goal of any hardware design is to increase data reuse once it's fetched and reduce the need to fetch data multiple times. Therefore, the common theme throughout this section is centered around data reuse, and how to get the most reuse out of a piece of data once it's fetched.

**Processing Elements (PEs)** – PEs are a unit of hardware structure that can process a MAC operation. Dedicated NN accelerators are usually equipped with many PEs, configured in a specific geometry advantageous to the NN workloads intended to be run on them. The main idea of PE geometry is to exploit data reuse, similar to data reuse in cache hierarchy architecting. Such an exercise usually calls for co-design of algorithm and hardware.

**Dataflow and loops** [1] – This section can be thought of as the order in which the algorithm will traverse. During its forward and backpropagation passes, the NN will traverse through and compute operations on multiple tensors such as weights, filters, activations, gradients, etc. Depending on the underlying hardware, the traversal order through these tensors can have substantial impact on data reuse, which can consequently tremendously impact execution time.

**Dataflow Taxonomy** [1] – The common dataflow pattern mentioned in [1] are weight, output, input, stationary flow. These orders depict data access patterns with weights, outputs, inputs, as the outermost loop in the nested-loops traversal, respectively, in order to favor reuse correspondingly. Moreover, data access patterns as Row/Column major orders can be agnostics to the datatypes but favor their row or column.

**PEs' Global Buffer** – Also referred to as a scratchpad, this buffer provides the PEs with extra storage space beyond the PEs' individual register file to hold more data for reuse before they have to fetch data from cache (if available) or beyond. Data movement between the global buffer and the PEs as needed can be orchestrated in various ways based on the workload and the network on chip (NoC) topology (and request-response scheme) being used. Techniques such as overlapping fetching future data while serving currently needed data to the PEs, automatically removing stale data, just to name a few, can also be investigated.

**Network on Chip (NoC)** – NoC is used to describe the connectivity infrastructure between the different hardware structures. Various NoC shapes or topology such as crossbar, star, ring, 2D mesh, 2D and 3D torus, etc., can be chosen based on workload expected.

**Mapping** – With all the possible various hardware configurations, there needs to be a mechanism of directing a set of DL instruction streams on to the chosen hardware configurations. That exercise is called mapping, usually performed by a mapper. It is similar to a compiler's job of compiling a program down to machine code specific to a target processor. The difference now is that this mapper must consider more hardware structures and various configurations some of which were mentioned earlier in this section. Because of such so many more hardware configurability than standard CPU/GPU, a mapping space tend to be a lot bigger which makes finding the optimal hardware configurations based on a specific DL algorithm quite challenging.

**Neural Architecture Search (NAS)** – Because of the large map space, heuristically searching for an optimal mapping may not be realistic. NAS helps this short-coming leveraging ML to catch on pattern that leads to configurations which may provide the best performance.

Notable hardware accelerators implementing the hardware optimization mentioned above are as follow:

**Eyeriss[v2]** - An energy-efficient reconfigurable accelerator for deep convolutional neural networks [12].

**Google's TPU[v3,v4]** – Tensor Processing Unit. Dedicated accelerator designed specifically to accelerate DL kernel computations on tensors.

**NVDLA** – NVIDIA Deep Learning Accelerator, a free and open-source DL accelerator architecture from NVIDIA [13].

**Cambricon-X** – DL accelerator, which started out as a Neural Network Instruction Set Architecture (ISA) proposal back in 2016 [2].

**Intel AMX** – On-CPU accelerator for matrix operations.

## VI. Other Optimzation Technologies

Technologies that were not covered in this paper but worth noting are as follow: processing in memory (memristors), neuromorphic computing, and libraries such as cuSparse, TensorRT, Core ML, etc.

## VII. Potential Optimization Exploration for HW

**Automatic Operation Fusion** – Could be implemented through VLIW instructions. Other parallel sequences of instructions could also be scheduled in VLIW manner and a VLIW co-processor could be used in conjunction with a regular CPU to process DL workloads.

**Automatic DL Workload Detection** - Could hw automatically detect that instruction stream is a DL workload and automatically internally apply quantization? While maintaining certain accuracy? Perhaps a helper thread could be used to maintain "integrity/correctness" up to a certain threshold, of the lower precision ops.

**Cache Coherence** – Would certain cache coherence protocol benefit DL workload? Since DL workloads can tolerate less precision, could we come up with a new coherence protocol that that is much lazier in terms of bus signals updates but still maintain a reasonable level of correctness? Could hw automatically switch to said coherence protocol once it's processing DL workload?

**Memory Consistency** – Since DL workloads tend to be fuzzier and allow certain degree of less precision, would more relaxed consistency be more performant for DL workloads? If a CPU is required to adopt strong consistency model due to other workloads that it needs to also run, could we add a mechanism for the CPU to *temporarily* switch to a more relaxed consistency model that DL workloads could be benefit more from? Could we come up with an even more relaxed consistency model than the most relaxed one which could accelerate DL workloads?

**DL Microarchitecture Structures** – Could we embed microarchitectural structures/states to the hardware that could enable the implementation of certain sw/compiler optimization?

**Dedicated HW for Forward/Backpropagation** – Could we modify hw that could specifically accelerate gradient descent algorithms (SGD, ADAM, etc); the action of reducing the loss function of a network through the updating its weights in accordance with some gradients, or perhaps some other indicators that hw could easily track? Similar to the initial Cambricon ISA proposal [2] but instead of primitively covering common operations found in DL workloads, the hw itself could independently process forward and backward passes of DL training. What would we potentially need to modify?

## VIII. Conclusion

DL workloads' increase in complexity has resulted in larger memory footprint and computation density. Consequently, there has been considerable efforts from industry and academia to address growing memory and computation demand challenges through various forms of optimizations. This paper discussed three major levels of optimizations: software/algorithm, compiler/framework, and hardware optimizations, providing numerous optimization examples in each to gain performance. Lastly, this paper also discussed ways that some software and compiler optimizations could be directly done on hardware and perhaps experience greater improvement in performance.

## Acknowledgment

The author of this paper would like to thank the department of Electrical & Computer Engineer and Computer Science of NC State. Special thanks to the MicroSecure NeuronLabs NC State.

## References

[1] Sze, V., Chen, Y. H., Yang, T. J., & Emer, J. S. (2020). efficient Processing of deep neural networks (Synthesis Lectures on Computer Architecture). *Morgan & Claypool Publishers, Williston*.

[2] S. Liu *et al.*, "Cambricon: An Instruction Set Architecture for Neural Networks," *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, Seoul, Korea (South), 2016, pp. 393-405, doi: 10.1109/ISCA.2016.42.

[3] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L. and Guestrin, C., 2018. {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).

[4] Boyd,S.,Parikh,N.,Chu,E.,Peleato,B.,Eckstein,J.: Distributed optimization and statistical learning via the alternating direction method of multipliers. Foundations and Trends® in Machine Learning 3(1), 1–122 (2011)

[5] Zhang, T., Ye, S., Zhang, K., Tang, J., Wen, W., Fardad, M., Wang, Y.: A systematic dnn weight pruning framework using alternating direction method of multipliers. European Conference on Computer Vision (ECCV) (2018)

[6] Shaokai Ye, Xiaoyu Feng, Tianyun Zhang, Xiaolong Ma, Sheng Lin, Zhengang Li, Kaidi Xu, Wujie Wen, Sijia Liu, Jian Tang, Makan Fardad, Xue Lin, Yongpan Liu, Yanzhi Wang. 2019. "Progressive DNN Compression: A Key to Achieve Ultra-High Weight Pruning and Quantization Rates using ADMM", arXiv:1903.09769

[7] Ao Ren, Tianyun Zhang, Shaokai Ye, Jiayu Li, Wenyao Xu, Xuehai Qian, and Xue Lin, Yanzhi Wang. 2019. ADMM-NN: An Algorithm- Hardware Co-Design Framework of DNNs Using Alternating Di- rection Method of Multipliers. In 2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA. ACM, New York, NY, USA, 14 pages. hps://doi.org/10.1145/3297858.3304076

[8] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. 2020. PatDNN: Achieving Real-Time DNN Execution on Mobile Devices with Pattern-based Weight Pruning. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland. ACM, New York, NY, USA, 16 pages. hps://doi.org/ 10.1145/3373376.3378534

[9] Han, S., Mao, H. and Dally, W.J., 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*.

[10] Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G. and Wu, H., 2017. Mixed precision training. *arXiv preprint arXiv:1710.03740*.

[11] Li, J., Qin, Z., Mei, Y., Cui, J., Song, Y., Chen, C., Zhang, Y., Du, L., Cheng, X., Jin, B. and Ye, J., 2023. oneDNN Graph Compiler: A Hybrid Approach for High-Performance Deep Learning Compilation. *arXiv preprint arXiv:2301.01333*.

[12] Chen, Y.H., Krishna, T., Emer, J.S. and Sze, V., 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, *52*(1), pp.127-138.

[13] NVDLA. http://nvdla.org/.

[14] TensorRT. https://developer.nvidia.com/tensorrt.