MIOpen: An Open Source Library For Deep Learning Primitives

Jehandad Khan[0000−0003−4479−1871], Paul Fultz[0000−0002−3423−2315], Artem Tamazov[0000−0002−7427−8676], Daniel Lowell[0000−0002−8929−7837], Chao Liu[0000−0002−6943−07919], Michael Melesse[0000−0001−5663−9733], Murali Nandhimandalam[0000−0001−5427−3184], Kamil Nasyrov[0000−0003−2026−7100], Ilya Perminov[0000−0003−0486−5821], Tejash Shah[0000−0003−0354−5674], Vasilii Filippov[0000−0003−0559−0380], Jing Zhang[0000−0001−8114−1080], Jing Zhou[0000−0002−4294−3985], Bragadeesh Natarajan[0000−0001−6848−0694], and Mayank Daga[0000−0002−2637−302X]

AMD Inc.
Mayank.Daga@amd.com

Abstract. Deep Learning has established itself to be a common occurrence in the business lexicon. The unprecedented success of deep learning in recent years can be attributed to: an abundance of data, availability of gargantuan compute capabilities offered by GPUs, and adoption of open-source philosophy by the researchers and industry. Deep neural networks can be decomposed into a series of different operators. MIOpen, AMD's open-source deep learning primitives library for GPUs, provides highly optimized implementations of such operators, shielding researchers from internal implementation details and hence, accelerating the time to discovery. This paper introduces MIOpen and provides details about the internal workings of the library and supported features.
MIOpen innovates on several fronts, such as implementing fusion to optimize for memory bandwidth and GPU launch overheads, providing an auto-tuning infrastructure to overcome the large design space of problem configurations, and implementing different algorithms to optimize convolutions for different filter and input sizes. MIOpen is one of the first libraries to publicly support the bfloat16 data-type for convolutions, allowing efficient training at lower precision without the loss of accuracy.

Keywords: Convolution, Deep Learning, GPU, HIP, Machine Learning, MIOpen, OpenCL® , Performance.

1 Introduction

Deep Learning has burgeoned into one of the most important technological breakthroughs of the 21st century. The use of deep learning has garnered immense success in applications like image and speech recognition, recommendation systems, and language translation. This in turn advances fields like autonomous driving and disease diagnosis. GPUs have played a critical role in the advancement of deep learning. The massively parallel computational power of GPUs has been influential in reducing the training time of complex deep learning models hence, accelerating the time to discovery [29]. The availability of open-source frameworks like TensorFlow and PyTorch is another cornerstone for the fast-paced innovation in deep learning [1, 24].

The deep learning frameworks decompose the models as either a computational graph or a sequence of operations [12, 22]. These high-level operations are then compiled down to a series of hardware specific high-performance primitives. These primitives in deep learning are akin to BLAS (Basic Linear Algebra Subprograms) [18] in linear algebra and high performance computing. Availability of a library which provides highly optimized implementations of such primitives enables the deep learning researchers to focus on their science and leaves the burden of developing such primitives on the hardware vendors. The library then provides a simple and callable application programming interface (API) to enable seamless integration with client libraries and be flexible so that new features may be added easily.

MIOpen is AMD's deep learning primitives library which provides highly optimized, and hand-tuned implementations of different operators such as *convolution*, *batch normalization*, *pooling*, *softmax*, *activation* and layers for *Recurrent Neural Networks (RNNs)*, used in both training and inference [9]. Moreover, MIOpen is fully open-source including all its GPU kernels; complementing AMD's open-source ROCm stack [4]. MIOpen is the *first* to extend the open-source advantage into GPU vendor libraries thereby, continuing to embark on the same ethos as the deep learning community.

As deep learning has gained critical acclaim over the years, substantial research has been conducted to accelerate it. One optimization technique called *fusion* has been recognized to be more potent than others [19]. Fusion allows to fuse or collapse several neural network layers thereby, optimizing on 1) memory bandwidth requirements by requiring less data to be moved between host and GPU memories, and 2) GPU kernel launch overheads by launching fewer GPU kernels compared to the vanilla, non-fused neural network. Aside from discrete primitives, MIOpen also offers a fusion API which allows the frameworks to fuse some of the operations mentioned above. MIOpen fusion can be used to accelerate both convolution and recurrent neural networks.

Another area that has flourished with the popularity of deep learning is open-source graph compilers [19], [26], [16], [6]. Graph compilers further the relevance of deep learning to wide-spread applications by generating the implementations of aforementioned operators instead of relying on hardware specific libraries. However, generating high-performance implementations of two operators, convolution and GEMM, is extremely cumbersome without inherent knowledge of the underlying hardware. Therefore, the graph compilers rely on libraries like MIOpen for these operators. MIOpen's open-source nature enables a plethora of optimization opportunities which were not possible before. For example, fusing an operator generated by the compiler with MIOpen's convolutions. MIOpen facilitates these optimization by breaking down complex operators like convolutions into several simple and small operators and providing high-performance implementations of these simple operators to the graph compiler. This MIOpen feature is called *composable kernels*.

The primary aim of MIOpen is to provide access to high-performance kernels, support several data-types, and also support as many hardware targets as required. To that end, MIOpen supports four different data-types: `float32, float16, bfloat16, and int8`, and two programming models: OpenCL® [23] and HIP. The kernels in MIOpen are backed by both high-level language as well as hand-tuned assembly implementations. MIOpen also provides an auto-tuning infrastructure to achieve maximum performance on the user's hardware and software environment.

This document provides an under-the-hood look at the MIOpen library providing detailed information about the functionality of the library as well as introduce MIOpen's capabilities to users and developers. The rest of the paper is organized as follows: Section 2 describes some prior work, Section 3 describes the overall design philosophy of the library and provides details about kernel compilation, abstractions used to localize those details in the library, tuning infrastructure for improving kernel performance and MIOpen's support for OpenCL® and HIP. This is followed by Section 4 which provides details about the supported operations; primarily the convolution operation. Section 5 describes MIOpen's Fusion API for merging different operations for increased performance, this is followed by some usage statistics and performance comparisons in Section 6. Section 7 presents conclusion and future work.

2 Related Work

Developing hardware-optimized libraries for most critical and time-sensitive operations is a well-known practice. For linear algebra such libraries are known as BLAS (Basic Linear Algebra Subsystem) and have different implementations for different systems [5, 18, 28]. In similar spirit different deep learning libraries have been written, to make it easier for client applications to implement different deep learning primitives. Alex Krischevsky's `cuda-convnet` is one of the initial libraries to implement convolutions and inspired many others [27], [11]. Chetlur et al. developed cuDNN, a deep neural network library for nvidia GPUs [7]. MIOpen falls in this category since it provides a C programming language based API for deep learning primitives. While these libraries aim to accelerate deep learning primitives on GPUs, research also been conducted to improve the performance of inference only loads on different CPUs such as MKL-DNN [11].

Most of the above mentioned libraries focus on lower level optimization opportunities. An orthogonal approach is to abstract this detail behind a domain specific language (DSL). This technique has already been successfully applied to other domains such as computer vision and linear algebra [13, 14, 25]. Vasilache et al. developed *Tensor Comprehensions*, which takes a similar approach and designs a language which can infer tensor dimensions and summation indices automatically [27]. However, such an approach makes it complicated to support a wide array of platforms and hardware targets as is required of MIOpen.

2.1 MIOpen and higher level frameworks

The above libraries are augmented by a community of frameworks which enable researchers and practitioners to express their computation pipeline using a host language (typically Python™ or some other higher level language) [12] [1]. These frameworks in turn call out libraries such as MIOpen for efficient implementation of the primitives required to implement the computation in those graphs. Frameworks strive to support a wide array of hardware and applications, for instance both TensorFlow and PyTorch already support MIOpen as a backend aimed at AMD GPUs. Thus a user can seamlessly change the hardware target without changing their application code.

3 Overall Design

This section describes the MIOpen's design philosophy using the convolution operation as an example.

3.1 Kernels and Solvers

Mapping a problem description to a particular kernel requires MIOpen to determine the file which contains the required GPU kernel, the name of the kernel in the file and the compiler arguments required to compile it. Typically, there is more than one kernel which can perform similar operations. However, each kernel has a unique set of constraints and may result in different performance due to differing code optimizations and input dimensions of the problem.

All this information is grouped in MIOpen classes collectively called *solvers*. These classes together *solve* for the best convolution kernel given a problem description. This construct creates a layer of abstraction between the rest of the MIOpen library and kernel specific details, thus all the details of a kernel are completely localized.

If a developer wishes to add a new kernel to the MIOpen, all that is required is to add the source code for the kernel and implement the associated solver, thereafter the kernel may be selected automatically.

3.2 Auto tuning infrastructure

In general, any high-performance code leverages auto-tuning for choosing the parameters that may change with the underlying architecture as well as the problem description thereby, impacting performance. MIOpen is not an exception to this rule. This requires that all tunable kernels be tuned for known configuration to achieve maximum performance. Once known, these tuning parameters can be shipped with MIOpen or, the user may employ the same infrastructure to tune MIOpen kernels for custom configurations.

A solver encapsulates the constraints for the tuning parameters as well as the interface machinery to launch tuning instances. The tuning parameters create a grid of possible values of the kernel tuning parameters and the tuning infrastructure compiles and launches a unique kernel for each of these combinations using a pruned search space approach. Once a kernel is tuned and the optimum tuning parameters are known, they are serialized to a designated directory on the user's system for future retrieval.

3.3 Kernel compilation and caching

Launching a kernel requires setting up the compilation parameters and invoking a device-code compiler to generate the binary object. MIOpen device-code consists of kernels written in OpenCL® , HIP [3] and GCN assembly [2], which may be compiled using *clang* [8].

Since compiling a kernel is a costly and time-consuming procedure, MIOpen employs two levels of caching to improve the runtime performance of the library. This design choice is tightly coupled with how device-code compilers compile and load compute-kernels from the binaries.

Once an kernel file is compiled, it is cached to disk to avoid future compilations of the same source-file with the same parameters. Due to the caching effects described above, it is recommended that the user's application performs a *warmup* iteration so that MIOpen's different caches can be populated. Such runs will ensure that subsequent network invocations are accurately timed without the effects of disk I/O or compilation delays. This limitation is not unique to MIOpen and is also applicable to other high-performance libraries.

3.4 HIP and OpenCL® backends

MIOpen supports applications that use the OpenCL® and HIP programming models [3]. All the APIs remain consistent from the client application's perspective, the only difference is in the creation of `miopenHandle` structure, which is created either with a HIP stream or an OpenCL® device context. Internally the HIP backend compiles the kernel using an appropriate complier depending on the kernel source type. Subsequently, the compiled binary object is loaded and passed off to the runtime for execution.

4 Machine Learning Primitives

4.1 Convolution

Most modern neural networks employ convolution as a central operation. Its usefulness and popularity make it a critical piece of the machine learning puzzle, particularly in image processing.

The numerical complexity of the convolution operation and it's diverse set of inputs make it difficult to generalize accross multiple hardware architectures. Different algorithms have been proposed to compute the convolution of a filter and an image, among them MIOpen provides implementation for the Winograd algorithm [17], a *direct* algorithm and using the matrix-matrix multiplication (GEMM) operation [12] as well as the Fast Fourier Transform.

The best performing algorithm is rarely readily apparent on a given architecture for a set of input and filter dimensions. To assess the relative performance of these kernels and return the best performing kernel, MIOpen employs the *find step* before the actual convolution operation. For this step, the user constructs the necessary data structures for the input/output image tensors as well as the convolution descriptor specifying the properties of convolution such as striding, dilation, and padding. The user then calls the

MIOpen convolution `Find` API which allows MIOpen to benchmark all the applicable kernels for the given problem configuration, this information is returned in an array of type `miopenConvAlgoPerf_t`. This enables the library to adjust for any variations in the user hardware and also allows the user to balance the trade-off between execution time and additional memory that may be required for some algorithms.

Types of Supported Convolutions

Transpose Convolution Transposed Convolution (also known as deconvolution or fractionally-strided convolution) is an operation typically used to increase the size of the tensor resulting from convolution. The standard convolution operation reduces the size of the image, which is desirable in classification tasks. However, tasks such as image segmentation require the output tensor to have the same size as the input. MIOpen supports transpose convolution required by such networks and may be enabled by setting the `miopenConvolutionMode_t` in `miopenConvolutionDescriptor_t` to `miopenTranspose`.

Depthwise convolution In depthwise convolution, the input is separated along the depth (channels) and then is convolved with a filter that is also separated along the same axis. The results are stacked into a tensor. Separating out the process of finding spatial correlation and cross channel correlations, results in fewer parameters as compared to regular convolution. Smaller and more efficient neural networks with depthwise separable convolutions have applications in training on embedded systems such as mobile phones.

Grouped convolutions Group convolutions were introduced in Alexnet [15], to reduce the memory required for convolution operation. Grouped convolutions are able to achieve accuracy similar to non-grouped convolutions while having fewer parameters. Further details may be found in [15].

The function `miopenSetConvolutionGroupCount` may be used to set the group count for a groupwise convolution. To perform a depthwise convolution use the same function to set group count to the number of channels [10].

Composable Kernels Different variations of the convolution operation discussed above as well as the variety of algorithms that may be used to implement them make it difficult to develop efficient kernels. One solution to tackle this complexity is to break down these operations into reusable modules that can be universally used by different implementations of different algorithms, and express a kernel as a composition of these modules.

Development work would fall into one of the following categories: 1) describe an algorithm with a hardware-agnostic expression, 2) decide how to map the hardware-agnostic expressions into hardware-dependent modules, 3) implement and optimize the hardware-dependent modules for specific hardware. Breaking down these primitives into smaller modules opens new doors to optimization that may fuse these modules together.

This new kernel programming model is referred to as *composable kernels* in MIOpen. MIOpen v2.0 includes an implementation of the `implicit GEMM` convolution algorithm, using the composable kernel programming approach. Further details about this novel programming paradigm will be published in the future.

4.2 Batch Normalization

Batch normalization is a very successful technique for accelerating deep neural network training. There are two versions of batch normalization supported in MIOpen: `Per-activation` and `Spatial` batch normalization. Per-activation batch normalization is typically positioned after a fully connected layer in a network. Batch normalization for convolution layers is termed `spatial` in that it learns separate scaling(γ_i) and bias(β_i) parameters for each channel, the resulting transform is applied to all the activations in a single feature map.

MIOpen supports the batch normalization operation for both training and inference. They all accept the mode parameter from the `miopenBatchNormMode_t` enum, Which has two modes `miopenBNPerActivation`, which does element-wise normalization for a fully connected layer and `miopenBNSpatial` which does normalization for convolutions layers. For more information see [20] and [21].

4.3 Recurrent Neural Networks

MIOpen supports three RNN types prevalent in the industry and research: vanilla RNN, LSTM and, GRU and two kinds of activation function for the hidden state of vanilla RNN neuron: Rectified Linear Unit (ReLU) and hyperbolic tangent (Tanh). Furthermore, information through the RNN may flow in the forward direction (unidirectional RNNs) or both in the forward and backward directions (bi-directional RNNs). MIOpen supports all three RNN types in the unidirectional `miopenRNNunidirection` as well as the bidirectional model `miopenRNNbidirection`. Some RNN layers take input sequences directly from the output of a previous layer while others require a transform to align the intermediate vector dimension or simply to achieve better results. MIOpen satisfies this requirement by supporting two input types: 1) `miopenRNNlin-ear`, which performs a linear transform before feeding the input to the neuron, and 2) `miopenRNNskip`, which allows a direct input into the neuron. Similarly, bias to the neural network may be added or removed by choosing the mode `miopenRNNWith-Bias` or `miopenRNNNoBias`.

The dependence of current state on the previous state as well as different RNN configurations make it difficult to achieve high computational efficiency on a GPU platform. Prevalent frameworks such as TensorFlow encapsulate the state updating functions of the RNN neuron in a cell format to achieve better compatibility in different modes, though the impact of the data layouts and computation procedures on performance is neglected. MIOpen handles the RNN computation by taking advantage of two powerful ROCm platform GEMM libraries (1) rocBLAS for the HIP backend, and (2) MIOpenGEMM for the OpenCL® backend, which are augmented by specialized MIOpen kernels for other primitive functions.

MIOpen achieves high computational efficiency for RNNs by batching together different time steps and performing them as single GEMM operation. The is made possible due to the independent input vectors at different time points.

In addition to the operations mentioned above, other operations required to support popular neural network architectures are also supported by MIOpen.

5 Fusion API

Most neural networks are data-flow graphs where data flows from one direction and is operated upon as it moves from one layer to another. While conceptually data is flowing only in one direction, the underlying kernels implementing these operations have to read data from the global memory, operate on the data and then write the result back for layers down the pipeline. This is necessary due to the limited on-chip memory of the GPUs given the large image and filter sizes in neural network architectures.

However, not all operations require that data be read from and written back to the global memory each time. That is some operations may be fused to increase the compute efficiency of these kernels. This merger of the operations to be performed by a single kernel may be termed as *kernel-fusion*.

As a simple example let us consider an addition operation followed by a rectified linear unit (ReLU) operation. In this case, the intermediate result need not be written back to the main memory, and both the operations may be performed while the individual data elements are in the on-chip memory. Another common sequence of operations is convolution followed by a bias (addition) and ReLU operation. It must be kept in mind that fusions for other operators are much more involved such as the fusion of the convolution and batch normalization operation.

The MIOpen library offers the fusion API to facilitate the efficient fusion of such operations; it allows the user to specify a sequence of operations that are desired to be fused. Once the user specifies this sequence, MIOpen decides the applicable kernel and compiles it; all this information is encapsulated in the `miopenFusionPlanDescriptor` data structure [21].

If merging of the required fusion sequence is feasible, the compilation step of the fusion plan will return success; thereafter the user would supply the runtime arguments for the kernels such as parameters for different operations. Following which, the user would execute the fusion plan with data pointers for the input and output data. The advantage of separating the compilation step from the argument structure is that the fusion plan which has been compiled once, need not be compiled again for different input values. Further details and example code can be found at [20].

6 Results

This section highlights the performance improvements that MIOpen is able to offers particularly in convolution as well as some supported fusions. To date, the primary beneficiary of Machine Learning progress has been machine vision as well as Natural Language processing. In machine vision, the convolution operation is the primary workhorse due to the low number of parameters required to learn as compared to regular neural networks as well as the favorable mathematical properties. However, the parameters associated with the convolution operations in different deep convolution neural networks have changed considerably. The early CNNs employed larger filter sizes to reduce the height and width of the feature maps and simultaneously increase the number of feature maps. For instance, LeNet employed filters of size 5×5 while, Alexnet [15] contained filters of size 5×5 as well as 11×11. However, recently networks have almost exclusively relied on smaller filter sizes namely only 1×1 and 3×3 coupled with striding to reduce the size of the feature map.

Figure 1 shows the relative speedup of different convolution configurations as compared to MIOpen's im2col+GEMM implementation. The configurations shown therein have been selected randomly from different popular networks such as GoogLeNet, Inception v3, and Inception v4 for image classification. The y-axis in Figure 1 shows $\log$ of the speedup obtained by MIOpen, while the x-axis shows the labels for different configurations. Each label shows, respectively, the filter height, filter width, input channels, image height, image width, output channels, padding (height) and padding (width) separated by a hyphen (-).

Figures 1a, 1c and 1e depict the performance gains for kernels with filter height and width equal to 1 (1×1 convolutions) in the forward, backwards-data and backwards-weights directions respectively. While mathematically 1×1 convolutions may be described as a pure GEMM operation, still MIOpen may provide substantial performance benefit in certain cases. Similarly, Figures 1b, 1d and 1f show the performance benefit attained for non-1×1 kernels in the forward, backward-data and backward-weights directions respectively.

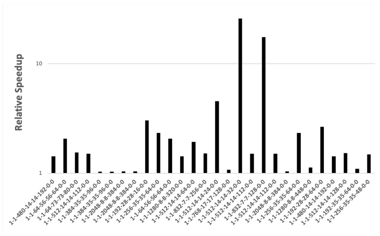
As mentioned in Section 4 MIOpen employs the Winograd algorithm for applicable convolutions while the 1×1 convolutions are primarily serviced by kernels written in GCN ISA. Due to the efficiency of the Winograd algorithm, MIOpen can speed up many 3×3 convolutions, however, on larger filter sizes it is not as effective due to granularity loss. Wherein MIOpen's other convolutional kernels step in to provide speedup, however, in some cases, this speedup is not substantial. The MIOpen team is continuously working on new algorithms to improve performance in these areas.

7 Conclusions and Future Work

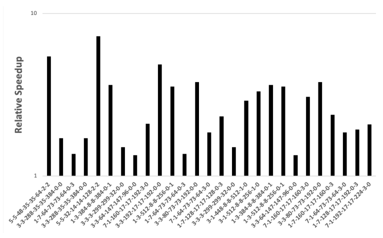
This paper identified some of the challenges faced by a high performance computing library and some of the mechanisms implemented in MIOpen to address these challenges were presented. The open source nature of MIOpen makes it easy for researchers and academics to experiment and implement novel solutions to these problems, the authors look forward to constructive feedback from the community.

Acknowledgements

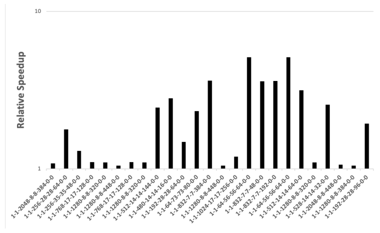
The MIOpen team would like to gratefully acknowledge the valuable contributions of Alex Lyashevsky, James Newling, and the GitHub user `ghostplant` as well as the support of the open source community.



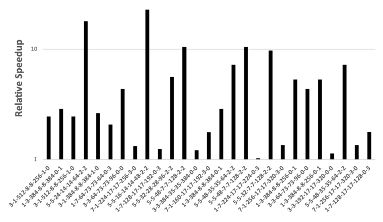
(a) 1x1 filter size (Forward)



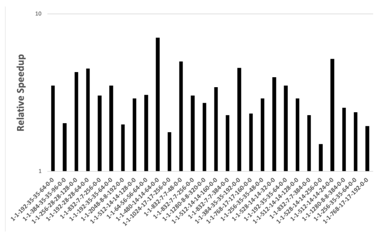
(b) non 1x1 filter sizes (Forward)



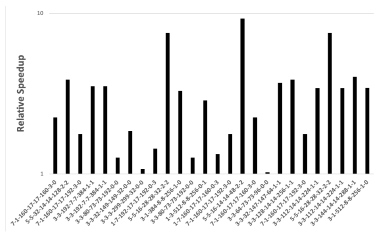
(c) 1x1 filter size (Backward Data)



(d) non 1x1 filter sizes (Backward Data)



(e) 1x1 filter size (Backward Weights)



(f) non 1x1 filter sizes (Backward Weights)

Fig. 1. Relative performance improvement for different convolution configurations as compared to im2col+GEMM

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: TensorFlow: A system for large-scale machine learning. In: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16). pp. 265–283 (2016)
2. AMD GCN ISA. https://developer.amd.com/resources/developer-guides-manuals ,last accessed 2020/07/15
3. AMD HIP. https://github.com/ROCm-Developer-Tools/HIP, last accessed 2020/08/13
4. AMD Inc: ROCm - Open Source Platform for HPC and Ultrascale GPU Computing, https://github.com/ROCmSoftwarePlatform, last accessed 2020/08/14
5. Belter, G., Jessup, E.R., Karlin, I., Siek, J.G.: Automating the generation of composed linear algebra kernels. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. p. 59. ACM (2009)

6. Chen, T., Moreau, T., Jiang, Z., Shen, H., Yan, E.Q., Wang, L., Hu, Y., Ceze, L., Guestrin, C., Krishnamurthy, A.: TVM: end-to-end optimization stack for deep learning. arXiv preprint arXiv:1802.04799 pp. 1–15 (2018)
7. Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E.: cuDNN: Efficient Primitives for Deep Learning. arXiv pp. 1–9 (2014). https://doi.org/10.1002/polb.23894, http://arxiv.org/abs/1410.0759
8. Clang: a C language family frontend for LLVM, http://clang.llvm.org/ last accessed 2020/07/18
9. Hochreiter, S., Schmidhuber, J.: Long Short-Term Memory. Neural Computation **9**(8), 1735–1780 (1997)
10. Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv preprint arXiv:1704.04861 (2017)
11. Intel MKL-DNN, https://software.intel.com/en-us/articles/introducing-dnn-primitives-in-intelr-mkl
12. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. arXiv preprint arXiv:1408.5093 (2014)
13. Kjolstad, F., Kamil, S., Chou, S., Lugato, D., Amarasinghe, S.: The tensor algebra compiler. Proceedings of the ACM on Programming Languages **1**(OOPSLA), 77 (2017)
14. Kjolstad, F., Kamil, S., Ragan-Kelley, J., Levin, D.I., Sueda, S., Chen, D., Vouga, E., Kaufman, D.M., Kanwar, G., Matusik, W., et al.: Simit: A language for physical simulation. ACM Transactions on Graphics (TOG) **35**(2), 20 (2016)
15. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. In: Advances in neural information processing systems. pp. 1097–1105 (2012)
16. Lattner, C., Pienaar, J.: MLIR Primer: A Compiler Infrastructure for the End of Moore's Law. Google Research (2019)
17. Lavin, A., Gray, S.: Fast algorithms for convolutional neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 4013–4021 (2016)
18. Lawson, C.L., Hanson, R.J., Kincaid, D.R., Krogh, F.T.: Basic linear algebra subprograms for fortran usage. vol. 5, pp. 308–323. ACM New York, NY, USA (1979)
19. Leary, C., Wang, T.: XLA: TensorFlow, compiled. TensorFlow Dev Summit (2017)
20. MIOpen: Documentation, https://rocmsoftwareplatform.github.io/MIOpen/doc/html, last accessed 2020/08/14
21. MIOpen: AMD's library for high performance machine learning primitives, https://github.com/ROCmSoftwarePlatform/MIOpen, last accessed 2020/05/15
22. MLIR: Multi-level Intermediate Representation for Compiler Infrastructure. https://github.com/tensorflow/mlir, last accessed 2020/09/03
23. Munshi, A.: The OpenCL specification. In: Hot Chips 21 Symposium (HCS), 2009 IEEE. pp. 1–314. IEEE (2009)
24. 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. In: Advances in neural information processing systems. pp. 8026–8037 (2019)
25. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. Acm Sigplan Notices **48**(6), 519–530 (2013)
26. Rotem, N., Fix, J., Abdulrasool, S., Catron, G., Deng, S., Dzhabarov, R., Gibson, N., Hegeman, J., Lele, M., Levenstein, R., et al.: Glow: Graph lowering compiler techniques for neural networks. arXiv preprint arXiv:1805.00907 (2018)

27. Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W.S., Verdoolaege, S., Adams, A., Cohen, A.: Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. arXiv **2** (2018), http://arxiv.org/abs/1802.04730
28. Whaley, R.C., Dongarra, J.J.: Automatically tuned linear algebra software. In: SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing. pp. 38–38. IEEE (1998)
29. You, Y., Zhang, Z., Hsieh, C.J., Demmel, J., Keutzer, K.: ImageNet training in minutes. In: Proceedings of the 47th International Conference on Parallel Processing. p. 1. ACM (2018)