2

AIDGE: A Framework for Deep Neural Network Development, Training and Deployment on the Edge

Fabian Chersi, Olivier Bichler, Cyril Moineau, Maxence Naud, Laurent Soutier, Vincent Templier, Thibault Allenet, Inna Kucher, and Vincent Lorrain

CEA, France

Abstract

The last decade has seen Deep Neural Networks (DNNs) become exponentially larger, more capable, more power hungry, and more ubiquitous. This has led to the need to bring machine learning to a wide variety of hardware devices that are either more performing but larger (GPUs) or require less power and are portable (embedded devices). Currently, efficiently deploying these networks on devices requires significant manual effort and a deep knowledge of different tools as well as the hardware's characteristics.

Here we present Aidge, an open-source comprehensive framework designed for simple and fast DNN prototyping, manipulation, optimization, training, testing and deployment. The platform integrates tools for database construction, data pre-processing, network building, and hardware export to various targets.

One of the defining characteristics of Aidge is its modular architecture. More precisely, there is a "Core Module" providing the most common functionalities (e.g. network manipulation, optimization, etc.) that can be extended by so called "plugins" that allow users to add functionalities, such as new quantization algorithms or specific hardware exporters that were not foreseen or implemented during the initial design of the framework.

Aidge is interoperable with the most common frameworks such as PyTorch and Tensorflow, and formats such as ONNX, and it targets hardware such as CPU, DSP, MCU, GPU and FPGA. It can be used with Python, C++ and through a Graphical User Interface.

The open-source framework can be found at:
http://projects.eclipse.org/projects/technology.aidge

Keywords: DNN optimisation, quantization, deployment and compilation, low power, edge devices, graph manipulation, hardware export.

2.1 Introduction and Background

Deep Neural Networks have now reached impressive capabilities in recognizing images, processing natural language, and generating different types of content. There is thus a growing demand to deploy DNN-based applications to a great variety of devices, ranging from GPUs and TPUs [1] in cloud servers, to self-driving cars, to mobile phones and drones, and finally to DSPs, MCUs and FPGAs. Porting AI architectures to these devices is complicated due to the diversity of hardware characteristics, mainly the different functioning of their processing units and the available memory.

Modern Deep Learning (DL) frameworks, such as TensorFlow [2], PyTorch [3] and ONNX [4] utilize a computational graph intermediate representation (IR) to perform manipulations and optimizations, e.g. operator fusion, auto differentiation and dynamic memory management. Besides graph-level manipulations, which are often too high-level, in order to obtain better results, it is usually necessary to perform target hardware-specific operator-level transformations. Currently, the method generally utilized is to deploy generic code developed either for CPUs or GPUs, or to call functions contained in highly engineered and target-specific operator libraries.

These libraries are usually too target-specific, require significant manual tuning and knowledge of the hardware characteristics, and are thus too specialized and opaque to be easily ported to other devices. Presently, major DL frameworks tend to support only a restricted number of hardware back-ends due to the significant engineering support and time required to keep their code up to date. Moreover, even for supported back-ends there is the difficult task of avoiding graph manipulations that produce operators that are not natively supported in the target devices because they would need to fall back to unoptimized implementations.

In order to be able to produce highly optimized and target-specific implementations of desired DNNs, we developed Aidge, a framework containing tools and methods that allow users to act both at the graph-level and at the operator level. Aidge is thus at the same time a neural network graph editor and a compiler that accepts high-level descriptions of DNNs (e.g. in ONNX) and produces low-level code (e.g. in C or assembly) optimized and targeted at chosen hardware back-ends.

One of the great challenges in generating optimized code from high-level descriptions is the fact that different architectures manage operations, data and memory in different ways. For example, Deep Learning Accelerators (DLAs) [5] [1] [6] usually implement optimized tensor compute primitives, while GPUs [7] exploit their massive parallelism, and modern CPUs [8] [9] [10] contain vectorized instructions. Moreover, CPUs and GPUs automatically control pipeline dependencies to hide memory access latency, while for DLAs this has to be explicitly implemented by the developer. All these factors render the creation of a multi-target tool extremely complicated.

2.1.1 Related Work

Although DL models have seen an incredible rise in multiple domains and applications, the same cannot be said about frameworks that allow to easily optimize and deploy them to a wide range of hardware targets.

One way to represent and perform high-level optimizations is through *computation graph domain-specific languages* (DSLs). Examples of these are Tensorflow's XLA [2], DLVM [11] and Glow [12]. Although these representations are well suited for high-level optimizations, they are not apt for low-level operator optimization. To do this, many frameworks resort to lowering procedures to directly generate low-level LLVM or utilize proprietary vendor libraries. Clearly, these methods require considerable engineering effort, considering that they have to be done for every combination of hardware backend and operator variant.

An interesting solution has been proposed in the Halide language and compiler [13] where computing and scheduling are separated. This allows the authors to obtain considerable simplifications in programming and major speed-ups compared to hand-tuned C, intrinsics, and CUDA implementations.

A different optimization method is proposed in Weld [14] where diverse functions can submit their computations in a simple but general intermediate

representation that captures their data-parallel structure. It then optimizes data movement across these functions and emits efficient code for diverse hardware.

DnnWeaver [15] is a framework that automatically generates a synthesizable accelerator for a given (DNN, FPGA) pair from a high-level specification in Caffe. It uses hand-optimized design template to first translate a given high-level DNN specification to its novel ISA that represents a macro dataflow graph of the DNN, then it tiles, schedules, and batches DNN operations to maximize data reuse and best utilize target FPGA's memory and other resources.

Finally, TVM [16] is a DNN compiler that has the capability of optimizing code by searching and combining the best tensor operators. This compiler provides end-to-end compilation and optimization stacks that allow the deployment of DNNs on CPUs, but also mobile GPUs, and FPGA-based devices.

2.2 Our Framework Overview

In this work we present Aidge, a new end-to-end framework for training, optimizing and compiling DNNs especially for low power edge devices. This tool was designed to balance efficient compilation, flexibility, low level control and portability by combining insights from graph analysis and manipulation with methods from structured and functional programming languages.

The platform integrates database construction, data pre-processing, network building or importing, manipulation, optimization, quantization, testing and hardware export functionalities (see Figure 2.1). It is particularly useful for DNN design and exploration, allowing simple and fast prototyping of different DNNs.

With this tool it is possible to define and train multiple topology variations of a network and to automatically compare their performances (in terms of accuracy and computational cost).

One distinctive aspect of Aidge is that it is based on the principle of "modularity", i.e. there is a "Core Module" (see Figure 2.2) that can be extended by so called "plugins" that allow to add new functionalities and to meet needs not foreseen or implemented during the initial design of the framework.

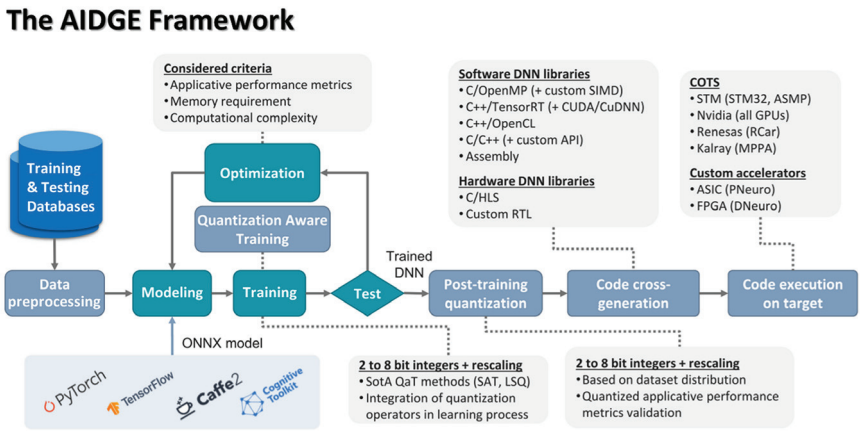

Figure 2.1 Schematic representation of the Aidge Framework with its main components and functionalities

The Core Module is developed entirely in C++ (14) with bindings to Python (>3.7), and includes a set of functions that enable it to:

- Create a computational graph representing a DNN.
- Modify the computational graph (e.g. by deleting, replacing or adding a node).
- Do graph querying/matching to find a specific sequence of operators in the computational graph.
- Instantiate operators.
- Instantiate data structures, such as Tensors.
- Create schedulers (for now only sequential) to execute the computational graph
- Apply graph optimization, such as fusion of operators

Aidge separates the concepts of description and implementation. Operators and data descriptions are abstract, while implementations are target-specific.

For example, the software implementation of a convolution may differ on a GPU or CPU, but the definition of the convolution itself (i.e. its inputs and parameters) does not change. Moreover, the implementation might also change according to the utilized library, for example on an NVIDIA GPU, programming can be done either via CUDA or via TensorRT. For this reason Aidge introduces the notion of "Backend" to define both the hardware target

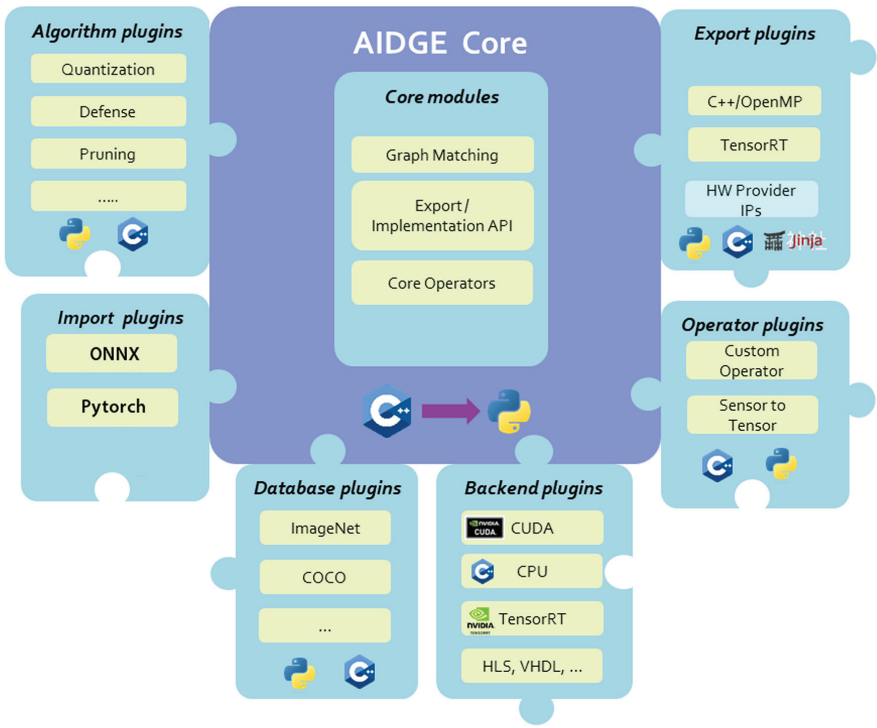

Figure 2.2 Aidge is built upon the concept of modularity with a "Core" component and several "plugins" that complete and extend the framework.

and the library used for the implementation (with its data type and a number precision)

Plugins allow developers and users to add or adapt functionalities of the platform. Different kinds of plugins can be developed (in C++ or Python) using the Aidge API, such as:

- "Recipe plugins", which may allow to load and save the network description in a specific format, or it may consist in a set of optimizer algorithms, for example to reduce the model's cost in terms of memory and computing complexity, or to increase its robustness against external/adversarial attacks.
- "Dataset plugins", which add the capability to load data and labels from a specific dataset.
- "Backend plugins", which register to the Core compiled kernel libraries (e.g. C++, CUDA, HLS) allowing it to execute the computational graph.

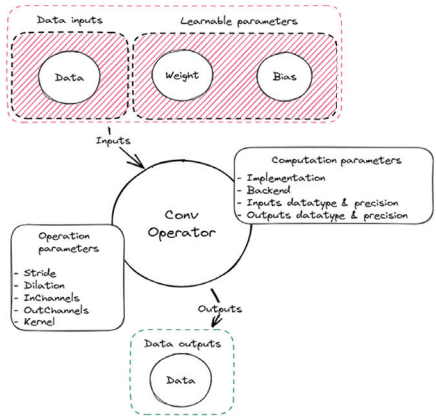

Figure 2.3 The image shows the constituent parts of an example Convolution operator.

- "Operator plugins", which adds the ability to define a new operator in C++ which is not available in the Core.
- "Export plugins", which define a set of rules and methods aimed at adapting the graph to the targeted hardware, and methods to produce source code corresponding to the optimized graph.

2.2.1 Internal Graph Representation

Aidge's low-level architecture is designed to allow the highest flexibility in DNN representation and computation, thus DNN models are represented using a directional "computational graph". This graph is composed of a set of nodes, representing operations (Operators), connected with directed edges, representing the flow of data.

Nodes in this computational graph are defined by three properties: the *connectivity*, the *operation description*, and the *implementation.*

1. Operation description: it describes the operation a node will do (e.g. Convolution, ReLu, Data Provider, etc.) and its attributes (see Figure 2.3). This description is agnostic of the implementation. The attributes are the following:
    - The sizes of the Kernel, Dilation (in case of convolutions), Stride, etc.
    - The number of inputs and their dimensions, datatype and precision;
    - The number of outputs and their dimensions, datatype and precision;

   - A reference to a *forward* (i.e. inference) function implementation;
   - A reference to a *backward* (i.e. train) function implementation.
2. Connectivity: it describes which nodes (proper Operators or Data Providers) are connected to a given node.
3. Implementation: it points to the computational function/kernel used by the Operator for its forward and backward operations. The selection of the right implementation is made via a *registrar system* depending on the following attributes:
   - The Backend, defined by both the hardware target (e.g. CPU, GPU, ...) and available libraries (e.g OpenCV)
   - The DataType (float, int, ...) and Precision (8bits, 16bits, 32bits,...) of the inputs and outputs
   - The DataFormat (NCHW, NHWC, ...)
   - The Kernel, the algorithm chosen to perform the computation

This flexible computational graph description is paired with the ability to use a great variety of data representation (e.g. Tensors, Sparse Tensors, Event Based stimuli, etc.).

2.2.2 Platform interoperability

Thanks to PyBind11, there is a seamless interoperability with Numpy arrays, achieved by defining a *buffer_protocol* in the binding of Aidge Tensors. This allows to use data imported from other frameworks that are compatible with Numpy.

Aidge is interoperable with PyTorch and allows:

- Creating an Aidge Tensor from a PyTorch Tensor
- Running an Aidge (sub)graph within the PyTorch environment.
- Running an Aidge computational graph within the PyTorch environment.

Aidge allows interoperability with Keras by creating a wrapper from a Keras Model through a conversion step via an ONNX file.

Similarly to PyTorch, Aidge can convert Keras tensors by using the Numpy interoperability.

2.2.3 Graph Regular Expression (GraphRegex)

The proposed Aidge's internal graph representation is a powerful tool that combines carefully chosen abstraction levels. The strategy is to adapt the internal representation to narrow the gap between a neural network and

hardware devices. Aidge proposes an innovative way to facilitate the manipulation of the internal graph representation: the Graph Regular Expression or GraphRegex

The Graph Regular Expression combines two main innovations:

1. A description of graph patterns. Taking inspiration from regular expressions from the formal language theory, we introduce a new language to describe a set of graphs starting from a sequence of characters.
2. Graph matching. Aidge provides a function that allows to extract a subset of the graph matching the provided GraphRegex description.

Graph Regular Expression is complementary to other graph transformation methods such as adding and removing nodes or entire parts of the graph.

With GraphRegex it is possible to work on two distinct levels in a graph:

1. At a conditional level, which corresponds to checking the presence of a node in a defined dictionary.
2. At a topological level, which allows to describe the interconnections between symbols.

The topological description, can be compared to classical regular expressions, as it is a form of symbol sequence expression, but extended to the definition of graphs.

At a practical level, this matching method can be subdivided into two distinct stages. First, we describe the desired pattern with the syntax of regular expressions, then we search for that pattern inside the graphs. These two steps together form the GraphRegex Query.

After extracting all the subgraphs corresponding to a GraphRegex Query, it is possible to use an intersection resolution algorithm to obtain intersection-free solutions. However, it is important to note that these algorithms can have a high complexity, which can make their execution time-consuming.

2.2.4 Network optimization

We can define two categories of optimizations: topological ones, which change the structure of the computation graph, and parametrical ones, which change the parameters of the nodes.

An example of topological modification is Tiling. This method splits convolutions in several ones (for example in 4 convolutions, as show in Figure 2.4). All of them are computed independently and concatenated at the end. This manipulation is mathematically exact (lossless).

Here is the practical implementation:

```
import aidge_core

import aidge_backend_cpu

import aidge_onnx

import numpy as np

#  Let's create a small neural network with four layers.

model = aidge_core.sequential([ aidge_core.LeakyReLU(1,
   name= "leakyrelu0" ), aidge_core.Conv2D(3, 32, [3, 3],
   name="conv0"), aidge_core.BatchNorm2D(32, name="bn0"),
   aidge_core.ReLU( name="relu0" ) ])

tiled_conv = aidge_core.get_conv_horizontal_tiling(
   model.get_node("conv0"), 2, 4)

node_to_replace = {model.get_node("conv0"),
   model.get_node("conv0").get_parent(1),
   model.get_node("conv0").get_parent(2)}

aidge_core.GraphView.replace(node_to_replace, tiled_conv)
```

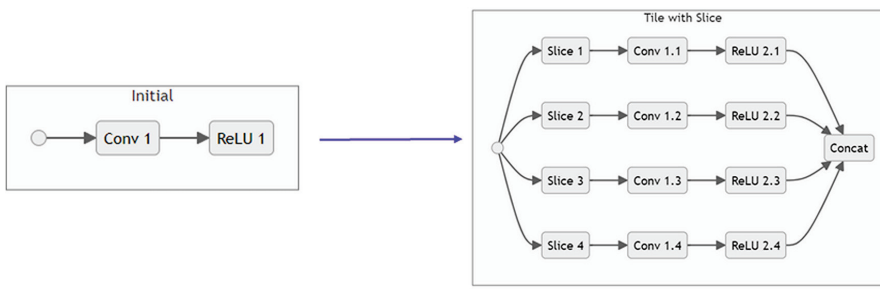

Figure 2.4 Example of operator tiling/splitting: a Conv + Relu subgraph is split into a Slice + 4 Conv + 4 Relu + Concat.

One of the key differentiators compared to other frameworks such as LLVM, is that Aidge applies directly graph modifications, which allows to make global topological changes as opposed to only focus on local ones.

On the other hand, an example of parametrical optimization is quantization after training (PTQ) or during training (QAT). This is a well-established method for reducing memory usage and in most cases, accelerating the

inference. PTQ is very useful when one does not have the time or the possibility to re-run the training and does not need to quantize to more than to 8-bits. If fewer bits are necessary, state-of-the-art QAT methods give very good results. These and other techniques (e.g. LSQ and FracBit QAT) are currently being finalized in Aidge.

2.2.5 Export phase

One of the aims of Aidge is to produce an interpretable, explainable and auditable output. To do this Aidge produces/exports source code files and a number of related resource files that form a complete package.

In Figure 2.5, which summarizes the export strategy, it is possible to see two phases: Export Mapping and Export Implementation.

The first objective of the Export Mapping phase is to modify the computational graph to fit the target hardware by using several optimization techniques (e.g. hardware mapping optimization or graph transformation).

The second objective is the generation of the graph scheduling constrained by the architecture rules of the target and additional project rules imposed by the developer or the user (e.g. the available memory, the available computer resources or the time allocated for the execution).

Taking into account the architecture rules and the project constraints, the scheduler will generate a sequential list of nodes from the optimized graph that will determine how the forward process (i.e. the inference) of the exported DNN will run on the target.

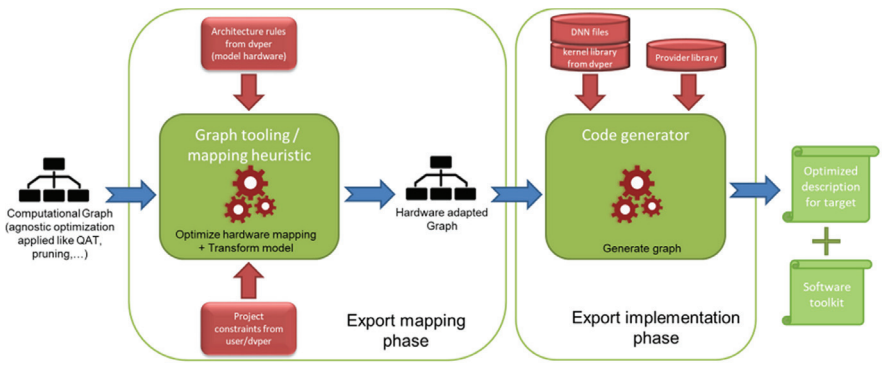

Figure 2.5 Schematic representation of Aidge's export procedure.

The Export Implementation phase aims at producing a source code of the hardware-tuned graph returned by the scheduler. The typical steps for generating source code are the following:

1. Design and export the computation kernels.
2. Export the attributes of the nodes.
3. Export the parameters of the nodes.

Each node of the graph must have an implementation of its forward method in order to use it in the export. Since only the hardware developers really know the characteristics and capabilities of their devices, it is their duty to provide the implementations of the computation kernels. These may be implemented as a kernel library, which is a collection of optimized functions developed by expert programmers targeting the architecture (computing functions, DMA programming, etc. . .).

Together with the kernels, Aidge generates the configuration and parameter files, and also the files that contain the source code of the forward function of the hardware-adapted graph.

The developer has also the possibility to add files to generate a whole Software Toolkit that will provide functionalities such as:

- Compilation or project files to compile the export
- Files to run a whole application of the export
- Set of unitary tests (to test the kernels on board, . . .)
- Input data for tests
- Third party libraries to use board functions
- Resources to check other constraints like security rules or robustness directives
- Memory map files indicating information about the static allocation of the resources used by the

2.3 Conclusion and future work

In this article we proposed Aidge, a framework that allows end-to-end manipulation, optimization and compilation of DNN architectures and their deployment to a vast spectrum of hardware devices ranging CPUs to GPUs, MCUs, DSPs, FPGAs and neuromorphic architectures. Another aim of this framework is to develop and provide reusable hardware building blocks and methodologies that are transversal to all types of architectures.

We would like to point out that one of the driving motivations for the development of this framework was the need to answer industrial challenges for the usability of AI.

For what concerns future work we foresee the extension of the number of target architectures including low power ASICS such as PNeuro [17], STM32, NeuroCorgi [18] and RISC-V.

Moreover, we will focus on graph optimization developing methods for Quantization Aware Training (QAT), Mixed Precision Quantization, Compression, Pruning, and Spike coding.

We also plan to add a greater number of supported functionalities and models such as: Object Detectors, Semantic Segmentation, Multimodal fusion, Attention models (Transformers)

Finally, we will start to tackle on-chip learning capabilities.

The framework is under active development using an open source and collaborative process and can be found at:

projects.eclipse.org/projects/technology.aidge

and

https://eclipse-aidge.readthedocs.io/en/latest/

Acknowledgements

We would like to thank the members of the Aidge development group for their work on the framework and for their valuable contributions to this paper.

This work was supported in part by the Neurokit2E European Project (GA101112268), and the DeepGreen Projet (ANR-23-DEGR-0001).

References

[1] N. P. Jouppi, et al., "In-datacenter performance analysis of a tensor processing," in *Proceedings of the 44th Annual International Symposium*, New York, 2017.

[2] M. Abadi, et al., "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation*, p. 265–283, 2016.

[3] A. Paszk, et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," ArXiv, 2019.

[4] J. Bai, F. Lu, K. Zhang et al., "ONNX: Open Neural Network Exchange," 2019, https://github.com/onnx/onnx

[5] Y.-H. Chen, J. Emer and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*, New York, 2016.

[6] Y. Chen, T. Luo, S. Liu et al., "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 609–622, 2014.

[7] NVIDIA, "H100 Tensor Core GPU," 2023, https://www.nvidia.com/en-us/data-center/h100/

[8] NVIDIA, "Grace CPU," 2023, https://www.nvidia.com/en-us/data-center/grace-cpu/

[9] NVIDIA, "Hopper CPU," 2023, https://www.nvidia.com/en-us/data-center/technologies/hopper-architecture/

[10] Intel, "Intel Core Processor Family," 2023, https://www.intel.com/content/www/us/en/products/details/processors/core.html

[11] R. Wei, V. Adve and R. Schwartz, "DLVM: A modern compiler infrastructure for deep learning systems," ArXiv, 2017.

[12] N. Rotem, et al., "Glow: Graph Lowering Compiler Techniques for Neural Networks," arXiv, 2019.

[13] J. Ragan-Kelley, C. Barnes, S. Adams et al., "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 519-530, 2013.

[14] S. Palkar, J. J. Thomas, D. Narayana et al., "Weld: Rethinking the interface between data-intensive applications," ArXiv, 2017.

[15] H. Sharma, J. Park, D. Mahajan et al., "From High-Level Deep Neural Models to FPGAs," "49th Annual IEEE/ACM International Symposium on Microarchitecture", pp. 1-12, 2016.

[16] T. Chen, T. Moreau, Z. Jiang et al., "TVM: An automated end-to-end optimizing compiler for deep learning," In 13th USENIX Symposium on Operating Systems Design and Implementation, p. 578–594, 2018.

[17] A. Carbon, J. M. Philippe, O. Bichler and et al., "PNeuro: a scalable energy-efficient programmable hardware accelerator for neural networks," Design, Automation & Test in Europe Conference & Exhibition, 2018.

[18] CEA-List, "NeuroCorgi," 2023, https://github.com/CEA-LIST/neurocorgi_sdk.