Deep Neural Networks Compiler for a Trace-Based Accelerator

Andre Xian Ming Chang, Aliasger Zaidy, Marko Vitez, Lukasz Burzawa, Eugenio Culurciello
FWDNXT Inc. 1281 Win Hentschel Blvd, West lafayette, USA
achang,azaidy,marko.vitez,lburzawa,euge@fwdnxt.com

Abstract

Convolutional Neural Networks (CNNs) are the algorithm of choice for image processing applications. CNNs are a highly parallel workload that leads to the emergence of custom hardware accelerators. Deep Learning (DL) models specialized in different tasks require programmable custom hardware and a compiler/mapper to efficiently translate different CNNs into an efficient dataflow in the accelerator. The goal of this paper is to present a compiler for running CNNs on programmable custom hardware accelerators with a domain-specific ISA that targets CNNs. In this work, the compiler was evaluated and tested on a hardware accelerator that was presented in [18]. The compiler uses model definition files created from popular frameworks to generate custom instructions. The model goes through static compilation and different levels of hardware aware optimizations that improve performance and data reuse of the generated program. The software also exposes an interface to run on various FPGA platforms, providing an end-to-end solution. Various CNN models were benchmarked on different systems while scaling the number of processing units.

CCS Concepts • **Software and its engineering → Compilers**; • **Hardware** → *Hardware accelerators*;

Keywords DNN, Compiler, accelerator

ACM Reference Format:
Andre Xian Ming Chang, Aliasger Zaidy, Marko Vitez, Lukasz Burzawa, Eugenio Culurciello. 2018. Deep Neural Networks Compiler for a Trace-Based Accelerator. In *Proceedings of 19th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3211332.3211333

1 Introduction

Deep Neural Networks (DNNs) are widely adopted in various areas, such as robotics, security and data analytics. DNNs are composed of highly parallelizable tensor operations, making them well suited to hardware accelerators. Accelerators are a very attractive solution for deploying DNNs especially in the embedded world, where it demands real-time processing under a limited power budget. Several hardware accelerators that use ASIC or FPGA were developed [18, 24]. At the same time, software improvements for optimizing DNN execution on CPU and GPU are under active research [11, 34].

A method to implement a programmable Deep Learning (DL) inference engine is to create a domain-specific instruction set (ISA). Manually crafting assembly-like custom instructions can be cumbersome and error-prone, especially when a model is composed of several layers. Even if one was willing to write custom assembly code, modifying thousands of lines would be required to further customize the hardware and software. The compiler should generate correct and efficient code. It also needs to parse various DNN models created from different high-level frameworks, so that users can use their favorite DL tool.

The goal of this work is to present a compiler (called Jaó) to support DL accelerators with custom ISA and the supporting software elements to execute a model defined in a high-level framework. In this work, the compiler was evaluated and tested for a specific hardware accelerator [18]. The compiler can be extended to address other hardware platforms. This work presents hardware aware optimization methods that aim to improve data reuse and performance. The main contribution of this work is two-fold: it demonstrates that specialized compiler flow leads to better hardware utilization, and ahead of time (AOT) compilation and loop unrolling expose a better schedule of the instructions for hardware accelerators that contain software managed buffers.

The compiler supports ONNX interchange format, allowing it to parse models from different frameworks. The compiler generates custom instructions for various DNN models trained for face identification, image segmentation, style transfer, speech identification, and speech commands. Jaó also provides an interface for users to create their demo applications, resulting in an end-to-end solution. The generated code was benchmarked on different FPGA systems with 256-2048 processing units.

The following sections are going to mention: the literature review, a description of CNNs workloads, a brief overview of the hardware accelerator, the compiler flow and results. The

LCTES'18, June 19–20, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5803-3/18/06.
https://doi.org/10.1145/3211332.3211333

compiler flow is divided into 4 sections: parsing, intermediate code, instruction generation, and deployment.

2 Literature Review

Accelerators provide various ways to control them: defined hardware control signals, extended ISA or domain-specific ISA. [6, 17] are examples of accelerators that expose hardware control signals. [15] presents a compiler for a custom CNN accelerator using Torch5 models. Their approach is to map Torch5 models into a set of pre-defined sequence of control signals for DMA transfers and processing units. Another approach is to extend an existing ISA as seen in [4]. This approach improves the programmability of the accelerator. [27] goes one step further by presenting a custom ISA with 64-bit instructions and 64 registers. This allows the designer to pick which instructions are needed, simplifying the control logic. Custom ISA also allows researchers to experiment with different ISA designs. [25] presents optimizations on controller logic to skip weights and activations.

Higher levels of abstraction ease the representation of the programmer's intent. Lower levels of representation provide a closer match with the hardware capabilities. There are various DL frameworks used to implement DNN: Tensorflow [2], Torch7 [12], Pytorch, MXNet [9] and Caffe [23]. An evaluation of those are presented in [5]. Another software layer is used to optimize tensor operations: XLA [37], Thnets [39] and ngraph [26]. A software back-end provides a link between DL frameworks and the accelerator through high-performance libraries such as cuDNN [11] and MKL [40]. An extensive review of different hardware and software strategies in literature is presented in [35]. In CPU and GPU, CONV is commonly implemented using Toeplitz matrices [11]. CONV is converted into a matrix matrix multiplication and GEMM libraries are used in the back-end [40]. This transformation results in more replicated data. Thus, mapping CONV directly into the accelerator is used in this work to reduce required memory bandwidth. In [4], CONVs are partitioned in tiles with extra overlap regions, called "augmented-tiles", in DRAM to avoid multiple DMA transactions. TVM [8] and Tensor comprehensions [38] present methods for automatic accelerator code generation applying lower level tensor optimizations such as blocking, operator fusion, tiling, and others. Glow [31] proposes an IR that better matches with DNN hardware accelerators.

Finding the best partition and assignment strategy for a specific architecture and generating code that achieves near peak performance is often tackled with heuristic methods. The execution of DNN models is iterated through different compilation parameters, which are automatically tuned. Tuning libraries [3, 30] perform search optimizations to find the best compilation parameter. Those techniques are also applied to the exploration of hardware designs using FPGAs [33]. Use of machine learning to guide the search for optimal hardware/software co-design was shown in [10, 13]. Autotuning methods for optimizing DNN inference using genetic algorithms are shown in [38].

3 Deep Neural Networks

DNNs are composed of various layers of operations connected. Before designing a compiler for DL accelerators, let us briefly go through commonly used layers in DNN models:

Spatial Convolution (CONV): Spatial convolution is the core layer of CNNs, accounting for more than 90% of the operations for most of CNN models. CONV is the multiplication and accumulation of values from an 3D input tensor (maps) with a set of 3D kernels that produce a 3D output volume. Each 3D kernel is associated with a bias value and produces a 2D output plane, also referenced as feature maps. Each output pixel of this plane was created by striding the kernel along with a window of the input. Padding is added to the corners of the input plane so that the kernel can cover the corners of the input. Padding is also used to create different output sizes. The main parameters of a CONV are dimensions of the input, kernel sizes, stride, and padding. From those parameters, the output size is calculated using equation 1. Only equation with subscript x is shown, but an equivalent with subscript y should be inferred.

$$o_x = \lfloor \frac{i_x - k_x + 2 * pad_x}{s_x} \rfloor + 1 \quad (1)$$

where i_x, i_y and i_p is input width, height and planes. And o_x, o_y and o_p is output width, height and planes. k_x, k_y and k_p is kernel width, height and planes. k_p and i_p are equal. s_x is the window stride along x. pad_x is number of columns padded on left and right. Usually, zeros values are used for padding. Reflection padding uses the corner values as padding instead of zeros. In any case, the compiler uses the appropriate addressing and computation sizes to avoid sending padding values to the co-processor.

Transposed Convolution (TCONV): CONVs produce o_x and o_y that are less or equal to i_x and i_y. Thus, the sizes of feature planes shrink in the deeper layers. To reverse this process, transpose convolution is a special case of CONV that recovers the sizes of the feature planes. TCONV have o_x and o_y greater than or equal to its i_x and i_y. The padding defines the number of rows or columns padded in the output of the TCONV. Similar thought applies for stride. TCONV also have an output padding, in which TCONV's input has an extra row or column in one of the corners so that the output width and height matches. It is possible to compute TCONV using an equivalent CONV. TCONV with stride greater than 1 is a CONV with padding/gaps in between each row and column. The gap size is stride minus 1. The location of the extra row or column added for output padding varies between frameworks. A good visualization of the TCONV operation is shown in [14].

Expanding the input is an expensive price for the accelerator since it increases the required memory footprint by $(s_x - 1) * (i_x - 1)$. The compiler does not send the padded input, it uses the appropriate access addresses to avoid sending padding/repeated data from memory to the co-processor.

Fully connected (FC): Fully connected layer is a matrix-vector multiplication. The matrix parameters are learned so that they map its input into an output vector with a learned linear function. FC is used in the last few layers of CNNs to provide a classification vector. FC is also largely used in Recurrent Neural Networks (RNNs) [22]. FC layer is a data movement intensive operation because it provides limited data reuse. In [24], they show that CNNs are compute intensive whereas RNNs are bandwidth intensive. Memory bandwidth is a bottleneck for RNN FPGA based accelerators [6]. Weight compression and weight pruning based on network sparsity are techniques that lower memory bandwidth requirement for this type of workload [19, 20]. FC layers can also be viewed as a CONV with unitary kernel sizes and strides. i_p is the input vector size, which is also the width of the matrix. And o_p is the output size, which is also the height of the matrix. This allows the reuse of most algorithms used in CONV for FC.

Average pooling (Avgpool) and Max pooling (Maxpool): Average pooling takes the average of the values in a window of the feature map. Avgpool also can be implemented as a CONV with a single weight value of the inverse of window size ($\frac{1}{k_x * k_y}$). Multiplying and accumulating all values in a window produce the average value of a window. Pooling is a down-sampling technique to achieve data invariance and to compress feature representation. Max pooling is element-wise comparison and its output is the highest value in a window. The input and output size relationship is the same as shown in 1.

Activation unit: Non-linear functions are applied to some layer's outputs so that the model will reproduce non-linear relations. Some examples of activation units used are rectified linear unit (ReLU), softmax, tanh and sigmoid.

Add, Multiply and Concatenation: These are important vector operations in linear algebra. Given a multiply and accumulate engine, add is $y = x * 1 + b$, where y is output, x is one input vector and b is another vector. Multiply is $y = x * b + 0$. Vector add is mainly used as a bypass connection between layers. This idea was introduced in ResNet models [21]. Concatenation is used to group feature maps from different layers. This was introduced in GoogLeNet models [36].

4 Inference Engine

The inference engine of choice used in this work was presented in [18]. Its main compute engine is a 16 bit multiply and accumulate unit (MAC). The data precision of choice of the MACs is fixed point Q8.8. A vector MAC (vMAC) is composed of 16 MACs, and 4 vMACs plus a vector max-pool unit (vMAX) are grouped into a compute unit (CU). Each vMAC has a private double buffered kernel buffer (WBuf) and all vMACs in a CU share the input data through the maps buffer (MBuf). A compute cluster (CC) is composed of 4 CUs, and multiple CCs can be implemented, each containing 256 MACs. The inference engine's 32- bit instructions are stored in the instruction buffer (I$). Each CC has a control unit, which is a pipeline that executes those instructions. The control unit has thirty-two 32- bit registers to store scalar values. The custom ISA has 13 different instructions, which implement four different functionalities: data movement, compute, flow control and memory access. The details of each instruction are presented in [18]. For this paper, the most relevant instructions are: MAC, MAX, VMOV, LD, TMOV, Branch and SYNC. MAC multiplies and accumulates a contiguous sequence of data from MBuf and WBuf. MAX compares two blocks of data from MBuf. MAC and MAX send results back to MBuf. VMOV pre-loads data to the compute unit to set the initial value for MAC. It is used to add the bias or implement the residual addition. LD sends data from external memory to MBuf, WBuf or I$. TMOV sends data from MBuf to external memory. Branch is used to create loops and if-else conditions. SYNC is an instruction that ensures all CUs or CCs are synchronized.

The accelerator used in this work has two modes of operations: cooperative (COOP) and independent (INDP). In COOP mode, all MACs in one vMAC work together to produce one value of the output map. Each MAC processes a different channel of one kernel, and the results of all 16 MACs are added together by an extra adder called gather adder to produce one value. Each vMAC produces one 1 output feature map (o_p). WBuf for each vMAC has a different kernel. The maps are broadcast to all CUs MBuf, thus 4 CUs produce 16 output values along o_p.

In INDP mode, all MACs in one vMAC work independently on different kernels and map values from Mbuf are broadcast to produce 16 different output map values. Each WBuf in a CU has a multiple of 16 kernels, thus a CU produces 64 output values along o_p. Different partitions of the maps are sent to different MBuf and the same set of 64 kernels is broadcast across CUs. The 4 CUs produces 4 vectors of 64 values stored back to their own MBuf.

In general, COOP mode gives high-utilization for high data reuse cases: large channel and/or large window size. Independent mode is useful for small channel cases used in most models initial layers, which need to extract features from an input image of channel 3 (RGB).

5 Parsing

The first step towards generating code for a custom accelerator is to gather information about the model's architecture. There are various high-level DL frameworks [2, 9, 12, 23]

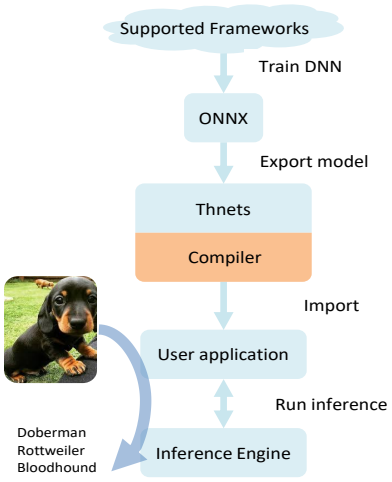

Figure 1. Workflow to run inference on the custom hardware accelerator. DNN models are trained using a DL framework and imported to Jaó using thnets.

being used today. Each represents DNNs differently, and they exploit different optimizations for deploying it on CPU or GPU systems. ONNX [1] is an exchange format that allows exporting and importing models created from different DL frameworks. Adopting ONNX allows users to deploy models that were trained on any DL framework supported by ONNX. Figure 1 shows the workflow to run a model on the hardware accelerator.

Thnets [39] is used to read a model file exported from ONNX. A list of layer objects is created to represent the layer computation sequence in the hardware accelerator. The objects contain information needed to generate code. Layer fusion is performed in this step to create layers that match the hardware capabilities. For example, the vector add operation present in ResNet models is fused with a convolution layer. Non-linear operations, such as MFM in [41] and ReLU, are also merged with convolution layers. An example is shown in figure 2, parts 1 and 2.

Main memory shared between the host and the inference engine is managed by software. Memory regions are allocated and maps are accessed in a double buffering fashion. When maps are shared among non-sequential layers, then extra regions are allocated to retain maps for later layers to use. Using two memory regions for sequential layers saves main memory space compared to allocating memory for each layer. This is important for embedded systems applications in which main memory is limited. In some models, such as GoogLeNet [36] and ResNet [21], some layers share their input and output. Those layers are labeled according to their parallel path. Later the labels are translated into memory addresses.

After creating the model list, each layer goes through a decision step that processes each layer's information and its neighboring layers to decide how to decompose and generate instructions for them. The choice of which *mode* (COOP/INDP) to use is defined by the layer parameters. If kernel size provides enough computational cycles and its channel size is multiple of 16 then COOP is used otherwise INDP is used. A *compute order* choice is based on whether to reuse weights or maps. Weight and maps tile sizes are defined by their buffer sizes. The number of tiles needed is the ratio between the total size and the tile size. An estimate of the data to be sent is calculated. And the option that requires the least data transfer is chosen.

LSTMs [22] and GRUs are also supported in this framework. They are executed as a group of FC layers. Once the FC layer's output vectors are produced, element-wise add, multiply, sigmoid and tanh are executed accordingly to create the final output. The recurrence states are saved in memory for multiple iterations.

6 Intermediate code

After the model parsing task completes, the compiler needs to partition the input and weights data and map the dataflow onto the hardware. DNN accelerators [29], are composed of an on-chip memory buffer (Buf) to store data, a group of processing elements and a control core. This leads to 3 main operations: load, compute and store. A sequence of load, compute and store is grouped into a compute step. Each step consumes part of the layer's input and produces part of the layer's output. The compiler creates a list of compute steps based on the layer parameters.

The limits imposed by Buf size and layer parameters are first calculated before creating the compute list. Based on these limits, load objects (LO) are created such that a balance between input data and output data coexists in the same Buf. LO sends data from external memory into Buf. The algorithm aggregates as many weights as possible that fit in WBuf. After LO creation, compute objects (CO) are generated based on the data available in MBuf and WBuf.

For a CONV, the minimum input necessary to produce an output is $k_x \times k_y \times k_p$. Maps data is arranged with planes first, then columns and rows last (p, x, y). To ensure data is contiguous and to avoid issuing multiple LD instructions, $i_x \times k_y \times k_p$ is needed to create one output row. The division order is rows first, then columns and planes last and a greedy approach tries to create as many output rows it can. If the input rows and output row doesn't fit into the MBuf, then other strategies need to be used. The input rows are divided into parts, which requires multiple LD instructions to put the maps data into the MBuf. In COOP mode, parts of the output row can be sent to different MBufs, which requires multiple TMOV instructions. Another approach is to divide the planes into parts and a compute step would create partial results. In ResNet models, it is common to have a CONV followed

by an ADD. For this fused layer, the MBuf will contain the CONV's input, output, and the ADD's input.

CO contains information about the vector compute operations necessary for producing a number of output values. This encompasses up to 3 loops: stride on the y-axis, the x-axis and accumulate. The accumulate loop issues multiple instructions that accumulate the results before producing an output pixel. This is because not all data needed to produce an output value is contiguous. CO will be translated into nested loops of vector instructions to perform multiply and accumulate or compare. The loop boundaries are a repeat variable and the data access address is incremented by an offset variable. CO also has an extension with variables for vector register load, which implements residual add and bias addition.

There are 3 types of CO: MAC, MAX and COPY. MAC CO generates instructions to multiply and accumulate values from MBuf and WBuf, and it conditionally creates VMOV instructions. MAX CO generates instructions to compare values in MBuf. COPY uses self-comparison (MAX instructions) to copy MBuf values to a different location in MBuf.

Lastly, store objects (SO) are created to return the output in MBuf to external memory. Compute step creation is shown in figure 2, part 3. In the example, assume input data to CONV is in m0, which is also the input to a residual add in a following RESADD layer. m1 is another memory location that has the output of the previous CONV and it is the input of the CONV part of the RESADD. w0 and w1 are other memory locations for weights. Precise address offsets were omitted. The created compute schedule has 2 fully unrolled loops. Depending on the CONV layer, this schedule loops all LO KBufs for each MBufs, or vice-versa. Optimization passes are applied to this initial schedule described in algorithm 1.

Algorithm 1 Initial operation schedule

```
for all k in K_parts do
  LO k
  for all m in M_parts do
    LO m
    CO (m * k)
  end for
  SO
end for
```

Each operation has a set of address ranges. The operation dependencies are verified based on these ranges and the type of operation. Every CO uses data from a LO or previous CO and creates data for a SO or for a following CO. The list of compute steps needs to guarantee that all LOs needed by a CO happened before it. All results of a CO are stored with a following SO or consumed by another CO. Following these observations, optimization passes are applied to a compute schedule. Algorithm 2 describes a pass to group COs based on LOs. This exposes redundant LOs across nested iterations. Figure 3 shows an example of the pass. In the example, assume there are 2 kernel parts k(0) and k(1). And there are 3 maps parts that produce 2 stores. First, compute steps with maps loads are grouped. Next redundant loads are removed, producing a schedule that requires less data transfer.

Algorithm 2 Group LOs and remove redundant LOs

```
for all c in computes do
  co ← CO in c
  co_next ← find next CO that accesses same data of co
  move co_next to after co
end for
for all c in computes do
  so ← SO in c
  co ← last CO that produces so
  move so to after co
end for
for all c in computes do
  lo ← LO in c
  lo_next ← LO in next of c
  if lo_next == lo then
    remove lo_next
  end if
end for
```

Compute steps with LOs accessing the same data are grouped in a sequence. This causes some of the LOs to become redundant, thus improving data reuse. LOs that are already present in M/WBuf or LOs that are not used by any CO are removed. SO must be moved after all COs that produces it. Finally, another pass is applied to double buffer, such that a compute step will pre-fetch data for the following compute step. A LO in the next compute step is moved to a previous one if that doesn't create any true dependency.

7 Instruction generation

Once a list of compute steps is created, a code generation phase converts each load, compute and store object into a list of instructions. Most of the computational structure was sketched as a result of compute step creation. This phase takes care of instruction-level optimizations, such as register assignment, branch delay slot filling, loop creation and unrolling. Each object has a corresponding function that creates instructions. Code generation for a compute step is shown in figure 2 part 4.

Instructions are grouped into basic blocks (BB). A list of BB is created per compute cluster. Each BB runs in sequence. The destination of any control flow instruction cannot be at any other BB. This way makes scanning instructions for potential optimizations and error checking bounded within a BB, rather than all or a fixed number of instructions.

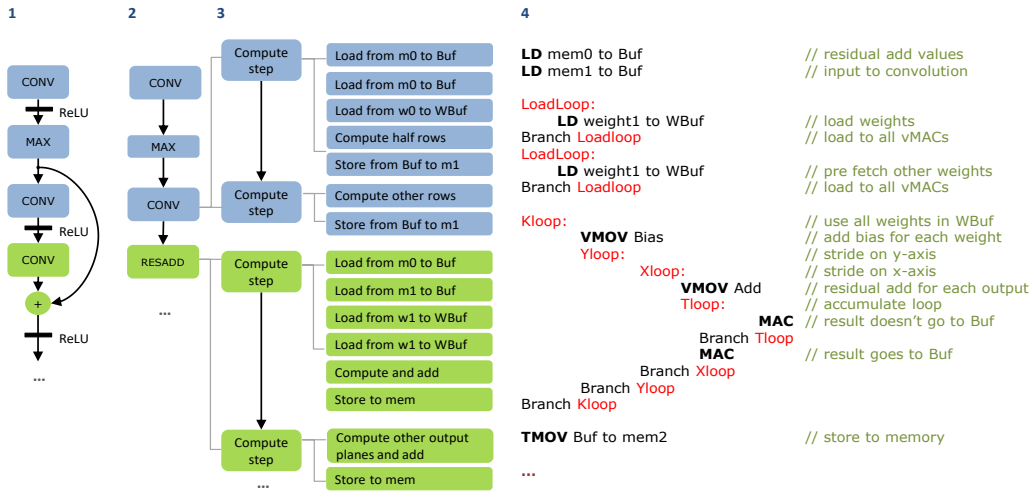

Figure 2. Example of instruction generation for part of ResNet model. 1- ResNet model layers. 2- list of layers. 3- Creation of compute steps that contains a list of LO, CO and SO. 4- Shows the main instructions generated from a compute step.

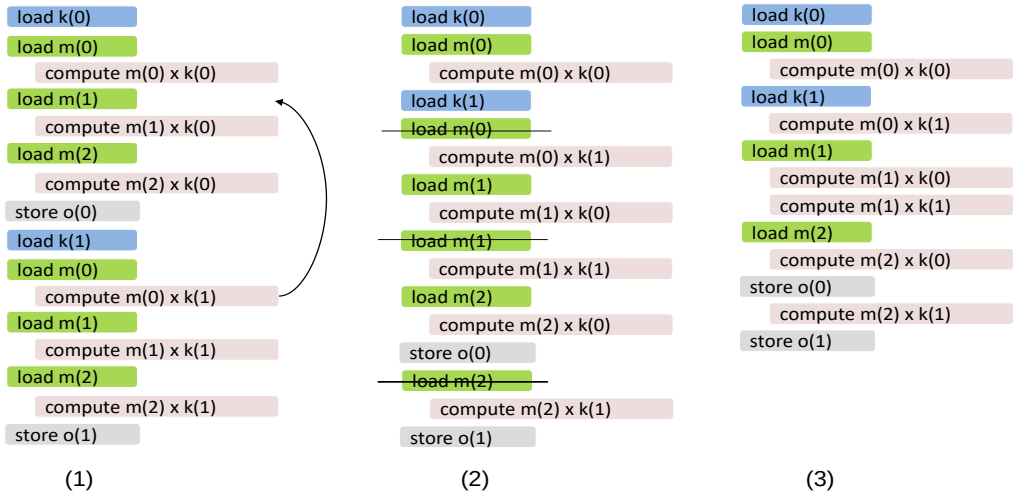

Figure 3. Example of algorithm 2. 1- Group the COs with same LOs together. The arrow in the diagram shows moving a compute step upwards. 2- Remove redundant LOs indicated by a line crossing the objects that are removed. 3- Resulting schedule before double-buffering pass.

The instruction cache is separated into 2 banks of 512 instructions. For each bank, a load is needed to load instructions to another bank. The compiler groups all the instructions into sections of 512 ensuring that there isn't a branch from one instruction bank to another. The compiler merges BBs to form instruction banks (groups of 512 instructions). Some instructions that are not in a loop or if-else condition from the following BB are moved to the instruction bank to so that I\$ is better utilized. Absence of branches across instruction banks is ensured by the BBs. At the beginning of each instruction bank, a load for the following bank is inserted. In the end, a jump to the next bank is inserted to align the section of useful instructions to 512.

In MAC CO, the accumulate loop issues multiple instructions that accumulate the results before producing an output pixel. This is because not all data needed to produce an output value is contiguous. In the case of a CONV with k_x and k_y equal to 3. 3 MAC instructions with 3 different addresses in MBuf and WBuf are needed. CO creates the accumulate loop that issues a MAC and increments each address by a constant. MAC CO code generation function unrolls the accumulate loops if they are small enough. They also add VMOVs in case of CONV+ADD. If 2 consecutive output values need to be compared like in CONV+MFM layers in LightCNN [41], then 2 sets of MAC instructions are created in sequence. Loop over all kernels in WBuf, y-loop and x-loop are conditionally

created. MAX CO and COPY CO also create those loops if needed.

SO conditionally creates sync instructions to avoid data corruption. Load, vector compute, and store instructions have variable latency and can be issued in parallel. In cases when there is a latency mismatch, a sync instruction is needed to avoid vector data hazards. The inference engine has multiple compute clusters, each with an independent control core, MBuf, and WBuf. There is one sync instruction that synchronizes the execution of all clusters. For example, if a cluster finishes producing half of the outputs in a CONV, it will wait at the sync instruction for the other cluster to also reach a barrier and complete the CONV layer before going to the second layer. Another possibility is to send different inputs for all clusters in which barrier isn't needed.

As instructions are created, they are checked for inconsistencies in their arguments: immediate is above a certain number of bits, the register value has overflown, special registers can only be used with some instructions, etc. Instructions are also labeled to determine some properties of the instruction: whether they cause WAW or RAW on the scalar registers, they are in a branch delay slot, they are redundant, they are in nested loops, or they have an immediate that will be modified (e.g: for address reallocation). After a BB is created, instruction optimizations are applied to the BB.

Vector instructions (MAC/MAX) take a variable amount of cycles to produce a result. Within these cycles we want to hide all other necessary operations: loop control, conditional branches, buffer address increment and load instructions. If those operations take on average more cycles than MAC/MAX latency then CUs would stall.

Read after write (RAW) is when an instruction reads from a register that was just written to in less than 4 cycles gap. In the case of RAW, the instruction will stall up to 4 cycles to get access to the required register. A common situation is shown in 4, where some instructions set some registers to have the MBuf and WBuf addresses for a MAC/MAX. To avoid RAW between the registers sets and the MAC/MAX instructions, some instructions for setting addresses are grouped, using different registers. The following set of MAC/MAX won't be stalled due to RAW. Pre-loading addresses into different registers solves the issue as long as there are enough registers as shown in 4. This instruction-level transformation is necessary for CONV/TCONV with small kernel size, which has low MAC latency, or max-pool layers. For example, the execution time for AlexNet's second max-pool layer reduced from 0.53 to 0.31 ms. And TCONV with $3x3$ kernel, $2x2$ stride, $32x32x64$ input, $o_p = 64$ reduced the execution time from 2.202 to 1.498 ms.

An array determines if a register is alive or dead, and this determines if instructions can use a particular register. It also determines how far an instruction can be moved without affecting other instructions. Redundant instructions or dead

```
MOV R0 maddr          // put Buf address in R0
MOV R1 kaddr          // put KBuf address in R1
MAC R0 R1             // uses R1. RAW 4 stall cycles
ADD R0 += offset      // increment Buf address in R0
ADD R1 += offset      // increment KBuf address in R1
MAC R0 R1             // uses R1. RAW 4 stall cycles
ADD R0 += offset
ADD R1 += offset
MAC R0 R1             // uses R1. RAW 4 stall cycles
                      // total: 12 stall cycles
```

```
MOV R0 maddr          // put Buf address in R0
MOV R1 kaddr          // put KBuf address in R1
MOV R2 maddr+offset
MOV R3 kaddr+offset
MOV R4 maddr+2.offset
MOV R5 kaddr+2.offset
MAC R0 R1
MAC R2 R3
MAC R4 R5             // uses R5. RAW 2 stall cycles
                      // total: 2 stall cycles
```

Figure 4. Example of code transformation to overlap bookkeeping operations with MAC/MAX instruction latency.

code are eliminated. Branch delay slot filling follows a similar approach to RAW, in which potential independent scalar instructions inside the loop are moved into an empty delay slot.

8 Instruction deployment

The last task is to run the instructions. After code generation, weights data is arranged to make kernel loads contiguous. For example, in INDP mode each MAC processes a different kernel so each WBuf of each vMAC has a group of 16 kernels. Each CU has a group of 64 kernels assuming kernel loads are in broadcast mode. If 2 groups of kernels fit in WBuf, then kernel 0 to 15 and 64 to 79 must be in sequence so that one load is needed. Bias values are attached at the beginning of each kernel.

An external memory shared between the accelerator and host contains all the data needed for execution. The memory layout reserves memory locations for temporary intermediate results for layers, input, output, weights, and instructions. Inference engine's LD and ST units access those locations to run. The memory layout, arranged weight data and instructions are saved in a file, which can be read by a decoding program to bypass recompilation of the same DNN model. Instructions that access memory are labeled in the code generation phase, and a reallocation table is created and saved.

It is possible to instantiate an accelerator on different FPGA cards with one host processor. In this case, separate software objects are created for each FPGA card. Different FPGAs can run different models or different inputs.

The inference engine provides some configuration registers that enable an initial load instruction to populate the instruction buffer with the first set of instructions. Another register to count the amount of data sent to and received from memory. The software polls the output counter register

to check whether processing has finished or not. The measurements are acquired from these registers are explained below:

- Execution time: is measured between setting the accelerator's start register and until all expected output values are produced.
- Expected time: is the time that it would have taken given the accelerator running at peak performance. It is calculated with equation 2, where *MACunits* is the total number of MAC units. Each MAC unit can do 2 Ops per cycle.
- Performance: is the ratio between the amount of operations, which is calculated from the DNN model, and execution time.
- Efficiency: is the ratio of expected time with measured run-time.
- Required bandwidth: is calculated as the total amount of data transferred divided by the expected execution time.
- Achieved bandwidth: is the ratio between the measured amount of data transferred with the measured execution time.

$$Expected\_time = \frac{Ops}{2.MACunits.freq} \tag{2}$$

For validation purposes, the compiler has a software implementation of the model's layers using Q8.8 to simulate the inference engine's compute operations. The accelerator's output for each layer is compared with CPU implementation of the layer to check for correctness. A functional simulator that executes custom instructions were created to debug and experiment with the generated code.

9 Results

The hardware accelerator parameters used in this work are 512 KB of WBuf and 256 KB of MBuf, 4 KB of I$ and 256 MACs per CC. An accelerator at 187 MHz was implemented using AC510 [28], which contains HMC memory and Xilinx KU060.

Figure 5 shows measured performance (top), bandwidth (middle) and efficiency (bottom) for some DNN models. The required bandwidth is shown in striped bars. This shows that the compiler and the accelerator are capable of running various DNN model architectures at high efficiency. The figure also demonstrates that the system is able to scale its performance across CC and FPGA cards.

Input size of choice is $224 \times 224 \times 3$. For LightCNN9 input size is $128 \times 128 \times 1$, Inception-v3 is $299 \times 299 \times 3$, Linknet [7], styletransfer [16] and yolov3 [32] are $256 \times 256 \times 3$. The execution time doesn't account for the linear layers.

Using EX750 backplane multiple AC510 cards were added. The measurements were obtained on 1 FPGA (1f) or 2 FPGAs (2f), using 1 input image (1i) or 2 images (2i) and using 1 CC (1cc) or 2 CCs (2cc). For example, in yellow 2cc1f1i means that 1 image was distributed into 2 CCs within 1 FPGA. In red, 2cc1f2i 1 image was processed by each CC on 1 FPGA. The maximum bandwidth that we achieved on one FPGA is 7 GB/s. 2cc1f1i shows 2× performance boost as expected from using 2× more MACs on same number of operations compared to 1cc1f1i. A larger system processes multiple images (more operations) in parallel achieving the same execution time of 1cc1f1i. Thus performance for 2cc1f2i and 1cc2f2i is 2× of 1cc1f1i. 1cc4f4i performance is 4×.

LSTMs and GRUs are also supported in this framework. Table 1 shows the execution time of 2 LSTM/GRU layers with a sequence length of 100. Different hidden layer sizes were used. Both models are memory bound workloads for accelerators with small on-chip buffers. This shows that the compiler is able to support applications that require RNN models.

Table 1. LSTM and GRU execution times

Type	Size	Exec. Time [ms]
GRU	128	9.7
GRU	512	76.4
LSTM	128	13.4
LSTM	512	100.1

9.1 Instruction analysis

Table 2 shows that majority of CONV layers are executed in COOP mode, thus most of the layers have i_p as a multiple of 16. LD length is in 64 B granularity, so LD with a length of 1 transfers 64 B of data. The MAC trace length is represented with 12- bit so their max is 4096. CONV layers in the benchmark models don't have $k_x \times i_p$ larger than 2048. The LD length on average is larger than the ST length.

The compiler generates an instruction stream that achieves performance comparable to handcrafted instructions as shown in table 3. The results in this table were measured with an early prototype system with 1 CU system, running at 142 MHz. In the table, CONVs parameters are, respectively, input size, kernel size, input plane, output plane, stride, and padding. *Auto* stands for compiler-generated code and *hand* is handwritten code. Auto-generated code has higher instruction count (437 more), but it achieves similar efficiency to hand-optimized code. We have only compared some AlexNet layers because models in handwritten instruction are human error-prone and tedious.

9.2 Loop rearrangement for bandwidth constraints

Unlike GPUs and ASIC designs, FPGA accelerators are limited mostly by their off-chip memory bandwidth. The required bandwidth for a layer is a ratio between the total amount of data transferred by the expected execution time. Loop rearrangement is a method that reduces the total amount

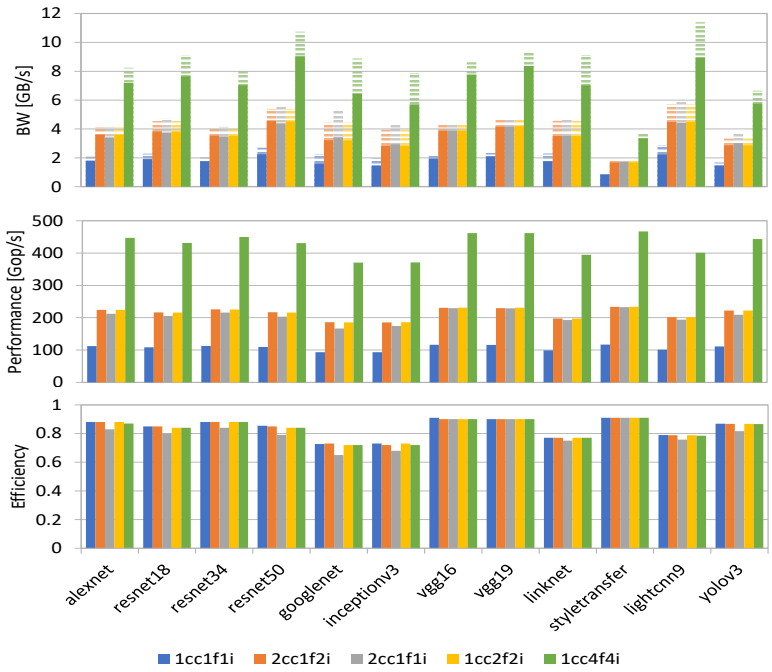

Figure 5. Bandwidth, performance and efficiency measurement of inference engine's execution for different DNN models with compiler-generated code. The required bandwidth is striped bars and measured bandwidth is the solid bar.

Table 2. Instruction parameters in different models. INDP and COOP columns show the percentage of the MAC instructions modes. LD shows the average LD length. MAC is the average trace length. ST shows the average ST length.

Model	INDP	COOP	LD	MAC	ST
alexnet	12	88	616	427	80
resnet18	7	93	367	291	112
resnet34	4	96	416	340	99
resnet50	9	91	472	362	117
inceptionv3	1	99	100	239	110
googlenet	8	92	164	265	111
vgg16	2	98	485	471	128
vgg19	2	98	504	533	124
lightcnn9	16	84	201	182	156
linknet	6	94	266	204	146
styletransfer	6	94	60	263	79
yolov3	2	98	87	331	22

of data movement by exploiting data reuse, which leads to memory bandwidth savings. Some CONV layers have large kernels, whereas others have large maps, but usually neither completely fits into the buffer. Maps and kernels need to be partitioned and processed in buffer sized tiles. A map tile needs to go through each kernel tile, leading to repeated kernel loads when the next map tile is loaded. Alternatively,

Table 3. Hand optimized code (hand) versus auto-generated instructions (auto) for some AlexNet layers.

Layer	Code	Time [ms]	Eff. [%]
13x13,3x3,192,384,1,1	Hand	11.11	99.5
	Auto	11.08	99.7
13x13,3x3,384,256,1,1	Hand	14.84	99.3
	Auto	14.77	99.8
13x13,3x3,256,256,1,1	Hand	9.89	99.3
	Auto	9.86	99.6

a kernel tile needs to be processed with every map tile, resulting in repeated map loads for the following kernel tile. The total amount of data moved is different depending on kernel/map load repetition for a particular CONV layer. The compiler estimates the amount of data to be transferred for both configurations and chooses the one that sends fewer data.

ReuseK is an abbreviation for repeated maps data and reuseM is an abbreviation for repeated kernel data. The difference between these two in terms of performance (top), bandwidth (middle) and efficiency (bottom) are shown in figure 6. The required bandwidth is shown in striped bars. The measurements were made on 1CC 1F for various DNN models. This shows that reuseK leads to lower memory bandwidth requirements for most DNNs. For some models, reuseM and reuseK doesn't show a significant difference.

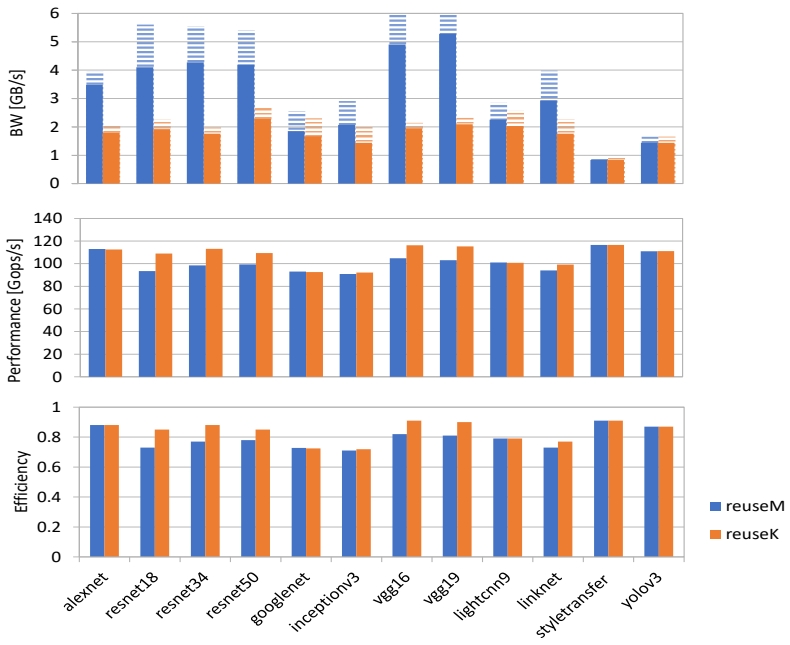

Figure 6. Required memory bandwidth, bandwidth, performance, and efficiency on AC510 using reuseM or reuseK mode for various DNN modes.

9.3 Discussion

In this paper, 4 different levels of the compiler flow were presented. Each level contains hardware aware optimizations. Parsing can fuse layers according to the primitives in the hardware's ISA. Intermediate code optimization reorders code based on internal buffer size. Code generation optimizes the order of instructions based on RAW and WAW dependencies. Deployment distributes the workload for multiple accelerators.

In comparison, TVM [8] is a generic deep learning compiler that can target various architectures: GPU, CPU, VTA [29]. VTA is an FPGA based hardware accelerator. Figure 7 shows the roofline plot for VTA and this work. The dashed lines denote the maximum that the accelerator can achieve. The data points denote what is achieved given the execution of CONVs from resnet18 mentioned in [29]. The closer the points are to their respective colored dashed line the better.

DNNVM [42] presents an FPGA system to accelerate DNN models. Figure 8 shows the FPS comparison against this work. A 4 CC system was used for this comparison. This shows that our system can achieve higher FPS given the techniques presented in this work.

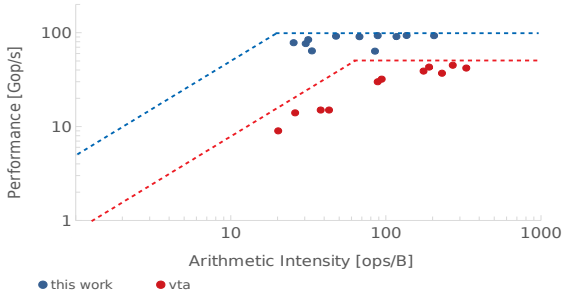

Figure 7. Roofline comparison with VTA [29].

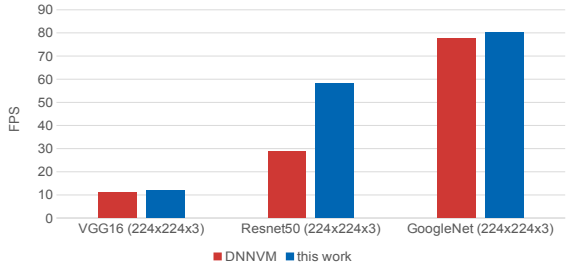

Figure 8. FPS comparison with DNNVM [42].

10 Conclusion

This work presents a compiler that takes a model definition created with popular deep learning frameworks and produces code for a custom DNN accelerator. This work contributes to the adoption of custom hardware accelerators with domain-specific ISA in embedded or server-based applications. Our future work involves improvements in implementing support for more DNN models.

Acknowledgment

This work was supported by FWDNXT Inc. and Micron

References

[1] 2017. Open Neural Network Exchange. (2017). https://github.com/onnx/onnx

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[3] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on*. IEEE, 303–315.

[4] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. 2017. Neurostream: Scalable and Energy Efficient Deep Learning with Smart Memory Cubes. *arXiv preprint arXiv:1701.06420* (2017).

[5] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. 2015. Comparative study of deep learning software frameworks. *arXiv preprint arXiv:1511.06435* (2015).

[6] Andre Xian Ming Chang, Berin Martini, and Eugenio Culurciello. 2015. Recurrent neural networks hardware implementation on FPGA. *arXiv preprint arXiv:1511.05552* (2015).

[7] Abhishek Chaurasia and Eugenio Culurciello. 2017. LinkNet: Exploiting Encoder Representations for Efficient Semantic Segmentation. *arXiv preprint arXiv:1707.03718* (2017).

[8] Tianqi Chen. 2017. TVM: An End to End IR Stack for Deploying Deep Learning Workloads on Hardware Platforms. (2017). http://tvmlang.org/2017/08/17/tvm-release-announcement.html

[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[10] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. *arXiv preprint arXiv:1805.08166* (2018).

[11] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[12] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*.

[13] Christophe Dubach. 2009. Using machine-learning to efficiently explore the architecture/compiler co-design space. (2009).

[14] Vincent Dumoulin and Francesco Visin. 2016. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285* (2016).

[15] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. 2011. NeuFlow: A runtime reconfigurable dataflow processor for vision. In *CVPR 2011 WORKSHOPS*. 109–116. https://doi.org/10.1109/CVPRW.2011.5981829

[16] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. 2015. A neural algorithm of artistic style. *arXiv preprint arXiv:1508.06576* (2015).

[17] Vinayak Gokhale, Jonghoon Jin, Aysegul Dundar, Berin Martini, and Eugenio Culurciello. 2014. A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*.

[18] Vinayak Gokhale, Aliasger Zaidy, Andre Xian Ming Chang, and Eugenio Culurciello. 2017. Snowflake: An efficient hardware accelerator for convolutional neural networks. In *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on*. IEEE, 1–4.

[19] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. 2016. Ese: Efficient speech recognition engine with compressed lstm on fpga. *arXiv preprint arXiv:1612.00694* (2016).

[20] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. *CoRR* abs/1602.01528 (2016). http://arxiv.org/abs/1602.01528

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). http://arxiv.org/abs/1512.03385

[22] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[23] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.

[24] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. *arXiv preprint arXiv:1704.04760* (2017).

[25] Patrick Judd, Alberto Delmas, Sayeh Sharify, and Andreas Moshovos. 2017. Cnvlutin2: Ineffectual-Activation-and-Weight-Free Deep Neural Network Computing. *arXiv preprint arXiv:1705.00125* (2017).

[26] Jason Knight. 2017. Technical Preview of IntelÂő NervanaâĎć Graph. (2017). http://ngraph.nervanasys.com/docs/latest/

[27] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. 2016. Cambricon: An instruction set architecture for neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 393–405.

[28] Micron 2017. *AC-510 UltraScale FPGA with Hybrid Memory Cube*. Micron. http://picocomputing.com/wp-content/uploads/2016/01/AC-510_Product_Brief.pdf

[29] Thierry Moreau, Tianqi Chen, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. VTA: An Open Hardware-Software Stack for Deep Learning. *arXiv preprint arXiv:1807.04188* (2018).

[30] CPLEX Optimization et al. 1993. Using the cplex callable library and cplex mixed integer library. *CPLEX Optimization, Incline Village* (1993).

[31] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Satish Nadathur, Jakob Olesen, et al. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. *arXiv preprint arXiv:1805.00907* (2018).

[32] Mohammad Javad Shafiee, Brendan Chywl, Francis Li, and Alexander Wong. 2017. Fast YOLO: a fast you only look once system for real-time embedded object detection in video. *arXiv preprint arXiv:1709.05943* (2017).

[33] Hardik Sharma, Jongse Park, Emmanuel Amaro, Bradley Thwaites, Praneetha Kotha, Anmol Gupta, Joon Kyung Kim, Asit Mishra, and Hadi Esmaeilzadeh. 2016. Dnnweaver: From high-level deep network models to fpga acceleration. In *the Workshop on Cognitive Architectures*.

[34] Daniel Strigl, Klaus Kofler, and Stefan Podlipnig. 2010. Performance and scalability of GPU-based convolutional neural networks. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*. IEEE, 317–324.

[35] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *arXiv preprint arXiv:1703.09039* (2017).

[36] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper With Convolutions. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[37] The XLA Team. 2017. TensorFlow compiled. (2017). https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html

[38] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[39] Marko Vitez. 2017. Thnets. (2017). https://github.com/mvitez/thnets

[40] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon PhiâĎć*. Springer, 167–188.

[41] Xiang Wu, Ran He, Zhenan Sun, and Tieniu Tan. 2015. A light CNN for deep face representation with noisy labels. *arXiv preprint arXiv:1511.02683* (2015).

[42] Yu Xing, Shuang Liang, Lingzhi Sui, Xijie Jia, Jiantao Qiu, Xin Liu, Yushun Wang, Yu Wang, and Yi Shan. 2019. DNNVM: End-to-End Compiler Leveraging Heterogeneous Optimizations on FPGA-based CNN Accelerators. *arXiv preprint arXiv:1902.07463* (2019).