Compiling Deep Learning Models for Custom Hardware Accelerators

Andre Xian Ming Chang
Purdue University Department of Electrical Engineering
amingcha@purdue.edu

Aliasger Zaidy
Purdue University Department of Electrical Engineering
azaidy@purdue.edu

Vinayak Gokhale
Purdue University Department of Electrical Engineering
vgokhale@purdue.edu

Eugenio Culurciello
Purdue University Department of Electrical Engineering
euge@purdue.edu

arXiv:1708.00117v1 [cs.DC] 1 Aug 2017

ABSTRACT

Convolutional neural networks (CNNs) are the core of most state-of-the-art deep learning algorithms specialized for object detection and classification. CNNs are both computationally complex and embarrassingly parallel. Two properties that leave room for potential software and hardware optimizations for embedded systems. Given a programmable hardware accelerator with a CNN oriented custom instructions set, the compiler's task is to exploit the hardware's full potential, while abiding with the hardware constraints and maintaining generality to run different CNN models with varying workload properties. Snowflake is an efficient and scalable hardware accelerator implemented on programmable logic devices. It implements a control pipeline for a custom instruction set. The goal of this paper is to present Snowflake's compiler that generates machine level instructions from Torch7 model description files. The main software design points explored in this work are: model structure parsing, CNN workload breakdown, loop rearrangement for memory bandwidth optimizations and memory access balancing. The performance achieved by compiler generated instructions matches against hand optimized code for convolution layers. Generated instructions also efficiently execute AlexNet and ResNet18 inference on Snowflake. Snowflake with 256 processing units was synthesized on Xilinx's Zynq XC7Z045 FPGA. At 250 MHz, AlexNet achieved in 93.6 frames/s and 1.2 GB/s of off-chip memory bandwidth, and 21.4 frames/s and 2.2 GB/s for ResNet18. Total on-chip power is 5 W.

Categories and Subject Descriptors

1.2 [**Hardware for Embedded Systems**]: HW/SW co-design for embedded systems

Keywords

Compiler; FPGA; Deep Neural Networks; Hardware accelerator

1. INTRODUCTION

The deep learning field has grown in popularity in recent years with the success of back-propagation based models. For the past few years, CNNs have consecutively achieved the state-of-the-art accuracy for classification tasks [13, 22, 9] on large image datasets [20]. Those models are embarrassingly parallel, but exploiting parallelism for all existent models is a design problem with room for software and hardware exploration to achieve better performance per power.

Snowflake [7] is a scalable and programmable, low-power accelerator for deep learning with a RISC based custom instruction set. Snowflake architecture was designed to provide high performance, given optimal sequence of instructions. But, manually crafting assembly like instructions can be cumbersome and error prone specially when a model is composed of several layers like in ResNet [9]. Even if one was patient enough to manually write code for some of state-of-the-art deep learning models, further customization on both sides: on the hardware and software would require modifying thousands of lines of assembly code, preventing experimentation on custom system for deep learning.

In this work, we present a compiler, which is responsible for generating instructions and managing data in the main memory. We designed a generic software structure to go from high level model representation from Torch7[4] down to an instruction stream that runs Snowflake. The main contributions of this work are:

- A software framework to generate custom instructions for CNN targeted hardware accelerator.
- Deciding whether to send maps data multiple times per set of kernels or send kernels multiple times per set of maps data for optimal bandwidth usage.
- Communication load balancing to better utilize available memory bandwidth.

Snowflake was implemented on Xilinx's Zynq XC7Z045 FPGA [24]. The system was benchmarked with AlexNet and ResNet18 [13, 9] pre-trained models. At 250 MHz, AlexNet

achieved in 93.6 frames/s and 1.2 GB/s of off-chip memory bandwidth, and 21.4 frames/s and 2.2 GB/s for ResNet18. The following sections present background and related work, overview of Snowflake hardware architecture, overview of instruction set, details of compiler implementation and the obtained results.

2. BACKGROUND AND RELATED WORK

Before designing a compiler for custom deep learning accelerators, lets briefly go through commonly used layers in CNN models that Snowball will be targeting in this paper:

Spatial Convolution (CONV) is a 3D convolution of an input volume with a group of 3D kernels that result in extracted features from the input. Each 3D kernel is associated with a bias value. Number channels is the z-axis of input volume. Input volume is called maps and a slice of it is a map. A window is the size of one kernel convolved with part of input volume. Operations in a compute window are independent, hence it is well suited to multi-core processors: GPUs [21] and other designs using ASIC [3, 2] and FPGAs [6, 26].

Activation unit is a non-linear function that some layer's outputs go through. Some examples are: rectified linear unit (ReLU), tanh and sigmoid. In this work, we only use ReLU.

Max pooling (Maxpool) is a down-sampling technique to achieve data invariance and to compress feature representation. Max pooling is element-wise comparison and its result is the highest value in a processing window.

Average pooling (Avgpool) is similar to Max pooling, but instead of getting the highest value in a window, it averages out its values. Average pooling can be implemented as a CONV with a single weight value of inverse of window size. Multiplying and accumulating all values in a window with this weight gives the average value of a window.

Residual addition or bypass is used in ResNet models [9]. The output values of a CONV are element-wise added with a previous layer's input. In hardware, we want to add those bypass values as output results are being produced by a CONV layer to save communication cost. Thus we need to keep track of previous input layers and to conditionally issue an extra instruction.

Fully connected (FC) layers are used to map the features into a classification vector that has the "final answer" of the network and it is usually the last layer of a CNN model. FC layer is a data movement intensive operation because it provides limited data reuse. Thus, memory bandwidth is a bottleneck for running FC layers. Weight compression and weight pruning based on network sparsity are techniques that lower memory bandwidth requirement for this type of workload [8, 12].

[5] 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. Custom hardware and instruction generation software was developed for Caffe [11] in [25, 1]. Snowball is the first to generate to custom instructions for hardware accelerator from Torch7 [4] or Pytorch models.

The system in [25] maps FC layers and CONV layers into a uniformed control representation: input or weight major processing. This work also uses uniform representation, but with a finer granularity defined as trace, which is any contiguous sequence of multiply and accumulate. This allows finer hardware and algorithmic optimizations.

Memory transfer friendly computation tiling for CNN accelerators was explored in [18, 19]. In [1], block tiling with x-y axis ordering was used. They store tiles with extra overlap regions, called augmented-tiles, in DRAM to avoid multiple DMA transactions. Snowball also stores overlapped regions but it tiles at the granularity of row strips with channel major ordering to lower overlapped data replication. This lowers the required memory bandwidth.

Other domain-specific instructions set for CNN were presented in [15], can be added into Snowball, because they also use vector compute instructions and scratchpad on-chip memory loads instructions. The intermediate representation and the techniques presented can be also be integrated into conventional frameworks [14], which is left for future work.

3. SNOWFLAKE HARDWARE OVERVIEW

Snowflake was presented in [7]. This section summarizes the main hardware concepts that will be needed to develop a compiler. For readers' convenience Snowflake's diagram is shown in figure 1.

The main building block of Snowflake's convolution engine are 16 bit multiply and accumulate units (MACs). A vector MAC (vMAC) is comprised of 16 MACs, that process 256 bits in one cycle. A compute unit (CU) is composed of 4 vMACs. Each vMAC has a private kernel scratchpad buffer (WBuf) and every vMAC in a CU shares the input data through the maps scratchpad buffer (MBuf). Data transfer time is overlapped by MAC compute time by using double buffer strategy.

Numerous CUs can be grouped into compute clusters. A compute cluster has a control unit, which is a RISC based pipeline. There are 4 load/store units that access the host main memory through DMA using AXI protocol. Vector comparator units (Pool Unit) are used for max-pool operations. RISC based instructions are loaded into the instruction cache (I$). The synthesized Snowflake was aimed for embedded system workloads, thus only one cluster with 4 CUs was instantiated. Each maps bank has 64 KB and each vMACs weight buffer is 8 KB. Instruction cache is 4 KB. Two ARM Cortex-A9 CPUs function as the host processor for the Snowflake implementation. Snowflake is clocked at 250 MHz and the ARM CPUs are at 666 MHz.

3.1 Control pipeline

The control pipeline provides the CUs control signals given a stream of instructions from the instruction cache. It is a pipeline with 5 stages: fetch, decode, dispatch, execute and register writeback.

In the **fetch stage**, instructions are read from the instruction cache. The instructions are 32 bit. The **decode stage** turns fetched instructions into signals for operation modes, sign extends immediate values and gives addresses to access the register file. Decode stage also performs true dependency or read-after-write (RAW) hazards detection. RAW causes decode stalls, which is a pipeline stall but not necessarily a CU stall. The **dispatch stage** is responsible for identifying the resources needed for the current instruction and issuing register file read. Snowflake has 32 32 bit registers. If the decoded instruction is a vector operation, then CUs receive signals to fetch data and start processing a vector. Scalar computations are implemented in execute stage. The **execute stage** gets the dispatch signals and starts the compute

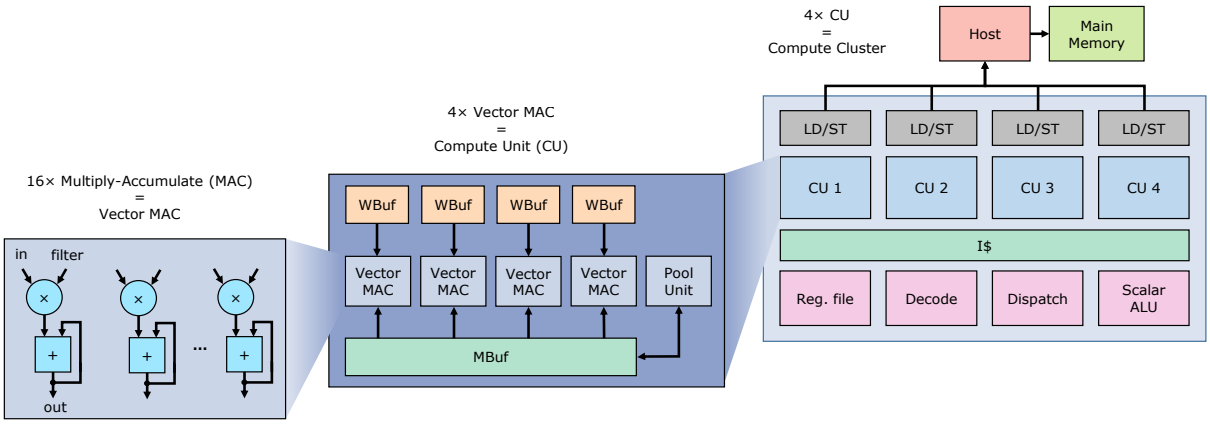


Figure 1: Snowflake architecture block diagram. On the left, a vMAC is a group of MAC units. In the center, a CU is composed of a group of vMACs with data buffers. In the right, a group of CUs forms a compute cluster that shares the control unit and load/store units.

resources for a scalar operation. Scalar unit is composed of multiplier, adder and comparator. Previous stages are single cycle latency, whereas the execute stage is 2 cycle. Finally, the **register write-back stage** writes scalar results to the register file.

4. CUSTOM INSTRUCTION SET

Snowflake's instruction set contains 13 instructions: MOV, MOVI, ADD, ADDI, MUL, MULI, MAC, MAX, VMOV, BLE, BGT, BEQ and LD. The 32 bit instructions are grouped into four categories: data movement, compute, flow control and memory access. Most instructions have in common four properties: 4 bit operand code, 1 bit mode select, 5 bit register selects (destination and two source registers) and a immediate field.

Data movement: Instruction MOVI moves a 23 bit immediate value into a register. MOV moves data between registers with a optional 5 bit left shift. VMOV is a vector move from buffer into compute units. It fetches a buffer block and load it into an operand register of a compute unit defined by select. VMOV is used to load the CONV/FC layer's bias into the MACs or to load the bypass values in residual add for ResNet.

Compute: ADD is a simple register to register add and ADDI is register with immediate add. MUL and MULI are register to register and register to immediate multiply, respectively. The vector compute units are controlled by MAC/MAX instructions. MAC multiplies and accumulates from contiguous sequence of data (trace) in maps and weights buffers. MAX is another vector instruction that has similar behavior to MAC. It performs comparisons with a retained previous vector. After a vector instruction finishes, a store to buffer or store to main memory is issued, thus there is not an explicit store instruction.

Snowflake's MAC instruction 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. In independent mode, all MACs in one vMAC work independently on different kernels and map values are broadcast to produce 16 different output map values. More details on custom instructions are presented in [7].

Flow control: BLE, BEQ, BGT are branch instructions that compare the value at Rs1 to the value at Rs2. The immediate is the PC offset when the condition is true. Branches take 4 cycles to go through the pipeline, thus it leaves 4 branch delay slots to be filled with instructions. Only one pair of true RAW dependent instructions is allowed in the branch delay slots.

Memory access: LD instruction loads data from main memory to one of Snowflake's buffers. Snowflake can have multiple load units that can start independent load streams, within the off-chip memory bandwidth constraints. LD have select modes that allow a processing choice of weights broadcast and different maps, or maps broadcast with different weights.

5. SNOWBALL

Given the custom hardware constraints, the compiler is responsible to orchestrate the hardware for all sorts of layers. How to load balance the communication ports for better bandwidth utilization, how to partition maps and kernels to fit in on-chip buffers, how to issue compute and load instructions without stalling and how to balance between loops and unrolled instructions are some software design decisions.

The compiler performs three major tasks: parse high level representation of a model, instruction generation and instruction deployment into hardware. Each task has steps that are described in following subsections.

5.1 Model parsing

In this task, the start point is a model representation from Torch7, and the end point is a data structure that contains all the information to easily generate Snowflake instructions. There are 5 steps to reach our goal.

Thnets[1] is an open-source library that provides means to read a Torch7 model representation file and to convert it into a C data structure. Using Thnets' functions, **step 1** loads the parameters of each layer in the model into a layer

[1] `https://github.com/mvitez/thnets`

object. This step scans through all layers and ignores non-sequential inter-layer relations that happens in parallel layer paths. The layer objects are serialized into a doubly linked list. Snowflake will process each element in the list in sequence.

In some models, such as GoogLeNet [22] and ResNet [9], not all the layers are sequential. Some layers share their input and output, thus some layers in the list are labeled according to their parallel path. **Step 2** scans the model to get the inter-layer relations, and creates a dependency label for each layer object. This label indicates whether the layer is only connected to its previous and next layers or not. This label translates into how each layer share their maps data in pre-allocated main memory regions.

Step 3 processes each layer's information and its neighboring layers to decide how to decompose and generate instructions for different layers. Snowflake hardware parameter object is globally shared among functions to create hardware dependent structures: maps tile, kernel tile and load objects. The main hardware constraints are:

- Instruction cache size: Snowflake instruction cache is double banked with 512 instructions per bank. But branching across instruction banks is not permitted. This affects how loops are broken down.
- Data buffer size: this defines how to decompose the maps and weights into tiles. It also affects the required memory bandwidth, since smaller tiles means more overlapped data is loaded more frequently.
- Memory bandwidth: computation stalls happen when required data has not arrived into the buffer, which is caused by the memory bandwidth constraint. It affects whether to loop kernels or maps. This will be explained in section 6.2.
- Instruction latency: vector instructions require variable latency 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. Another reason why a CU can stall is because there was not enough MAC/MAX instructions issued in sequence, not leaving enough cycles to overlap with bookkeeping instructions.

Based on these constraints and the layer parameters, the compiler sets decision variables that chooses mode (COOP or INDP) to use, chooses loop breakdown based on instruction cache size, sets tile size limit based on data buffer size, chooses whether is better to fix map data and loop through kernels or vice-versa based on memory bandwidth constraint.

Step 4 breaks down the layers' maps and weights data into tiles that fits into the data buffer. The maps are decomposed in tiles with output row granularity, meaning that each tile produces output row(s). Weights are decomposed in tiles with single kernel granularity. Single kernel size is input channels times window size. Based on the decision made in step 2 and the neighboring layers, the tiles can have different data sizes. For instance, if we broadcast weights, then each CU works on different map tiles and the maps need to be decomposed such that all CUs will have the same amount of work. Inevitably, some remaining tiles won't be big enough to share among all CUs. Then some CUs must be disabled. Another example is in the context of ResNet where a CONV followed by a bypass needs two input maps: one for the CONV and one for the bypass. This special CONV needs to use both maps buffer banks simultaneously. Double buffering is done by using different buffer regions.

After creating a list of tiles to process, each element of the list will create operation lists in **step 5**. For each tile, there are two major operations: load and compute. A load list carries information necessary for a load instruction: stream length, memory address and buffer address. Each load object is associated with a different tile. The load list is created taking into account subsequent tiles loads, so that Snowflake can process a tile while loading data for the next one.

Compute objects contain information about a set of vector compute operations in a window, the number repetition in x and y directions and x and y offsets. This allows grouping of striding windows with same window size into one object. For example, multiple compute objects are needed for CONVs with padding, whereas a single object suffices in a CONV without padding. Compute objects will be translated into nested loops of MAC/MAXP instructions. The loop boundaries are the repeat variable and the vector instruction read buffer address is incremented by the offset variable. The compute object also has an extension with variables for VMOV if the layer is a CONV with bypass.

An example of Snowflake's data structure is shown in figure 2. A layer object can be one of the layer types: CONV, Maxpool, Avgpool, FC and Residual add. A layer has two lists of tiles: one for kernel and one for maps. Each tile object can instantiate a list of windows and a list of loads. A kernel tile does not have window list because compute operations are defined by maps tiles. Load list is in the tile list that is not being repeated in a loop. In the example shown in figure 2, all kernels are looped through each maps tile. Kernel tile object contains repeat variable that defines the kernel loop boundaries. Hence, there is one kernel tile object. Kernel loads are implicit in a repeating tile object parameters, thus there is not a load list in the kernel tile. If maps are being repeated in loop for each different kernel tile, then kernel tile objects have load list and maps wont.

5.2 Instruction generation

After the model parsing task completes, the compiler goes through the lists and create instructions accordingly. Each tile object creates a block of instructions that will be concatenated with other tile objects instructions. The compiler inserts load for the following instruction cache bank at the beginning of each instruction block and inserts a jump to next instruction bank at the end to allow instruction double buffering. Hence, before generating instructions, we need to predict how many instructions are needed for a tile and check if it fits into the instruction cache constraint, so that we can correctly insert instruction loads and jumps. The prediction step passes through all objects and creates a temporary instruction block for each tile. If the predicted number of instruction in a block is larger than instruction cache bank size, then different instruction generation strategy must be used for that tile. After an instruction count profile for each tile is generated, the program knows where it can safely insert an instruction load for next bank. Then the permanent instruction stream is created.

Most of the instruction stream structure was sketched in the data structure as a result from model parsing in section

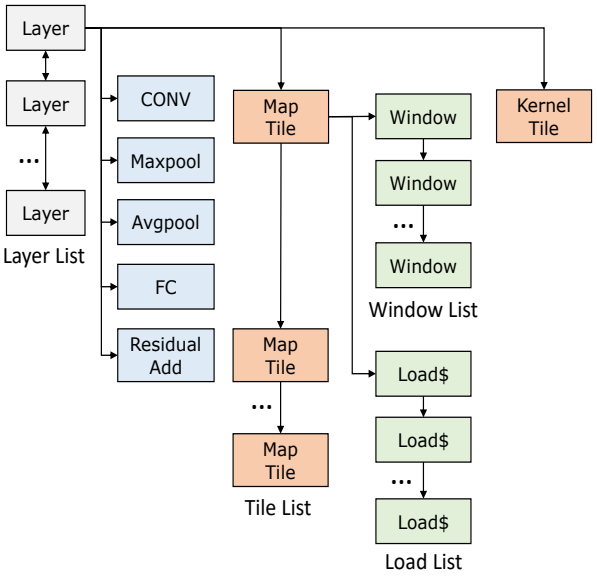


Figure 2: Example Snowball's data structure. The arrows represent pointers. Objects are grouped into lists. Head and tail pointers of the lists are omitted.

5.1. There are three main goals that instructions blocks need to accomplish:

- Initialization or register reset: the first tile of the model needs to populate all buffers in Snowflake creating a initialization latency. The following tiles needs to reset some loop control registers and set the reserved registers to the correct output locations.
- Compute data: each tile has their compute list, which defines a maximum of 3 nested loops. The inner loop is to accumulate traces, the second loop is stride along x-axis and the outer loop is stride along y-axis. Respectively, the loop limits are defined by kernel height, repeat x variable and repeat y variable.
- Load data: buffers needs to be updated for each compute tile, so that the following compute section can proceed without stalling for data. Both weights buffer and maps buffer loads need to be inserted in between compute operations, so that data coherence is maintained.

Figure 3 shows an example of instructions for a CONV tile, where T denotes trace loop, X x-axis loop, Y y-axis loop and K kernel loop (Kloop). The compiler first generates instructions for the initial map and weight buffer loads with the parameters in the first load object. Then it creates a kernel loop based on the kernel tile object. If the map tile window objects have non-unity repeat Y then a Y loop is created. Inside Y loop, T and X loops are created depending on the window objects variables. The load for the next tile, which is the second load MBuf object, is inserted in the X loop. The load insertion is needed because the compiler needs to guarantee that previously issued vector instructions have finished using a buffer bank before issuing new data load to that buffer bank. One way is to issue 16 vector instructions that will fill the trace buffer with new vector instructions, which guarantees that there no old vector instructions pending.

Depending on the CONV case, some of the loops are removed. For example, 1×1 CONVs don't have trace loop. Depending on bandwidth constraint, map and kernel tile objects can be generated, such that the kernel loop can become map loop, which means maps are sent multiple times for one tile of kernel.

Window objects can be broken down into multiple loops, which is based on the instruction latency versus instruction size trade-off. Snowball can only put limited amount of bookkeeping work in between two consecutive vector instructions. If those operations takes on average more cycles than MAC/MAX latency then CUs would stall because there is not enough MAC/MAX instruction being issued in sequence. A CONV with higher MAC latency allows more freedom to add flow control instructions, making the instruction stream more generic and with fewer instructions. On the other hand, 1×1 CONVs have lower MAC latency, which restricts the number of bookkeeping instructions that can be overlapped with the MAC instruction's latency. For example, if a MAC takes 16 cycles to finish, then having 20 instructions (loads, branches and other bookkeeping) in between MAC instructions will stall the CUs. In this case, breaking the loops or full loop unrolling will reduce the amount of bookkeeping instructions between consecutive instructions. But this increases the instruction count. One extreme is that all loops are completely unrolled.

Window objects also have the buffer address of bias value associated with each output map produced by a kernel, so a VMOV is needed before each Kloop iteration. For a residual add, a VMOV for each write-back MAC instruction is necessary because output value is added with a bypass value. Note that only the last MAC instruction of a trace loop is a write-back MAC instruction. VMOV also adds additional bookkeeping instructions for incrementing and resetting addresses and word select. This causes issue in extreme cases when there are not enough MAC instructions to hide the bookkeeping operations, like in the last 1×1 CONVs of ResNet18 and ResNet50.

Another part is that load instruction can become comparably large. For example, if weights are not broadcast, then there will be a load for each weights buffer. In a 4 CU system, there will be 16 weight LDs plus load ID bookkeeping operations. For a low MAC/MAX latency CONV case, the 16 loads should be spread out and interleaved with the MAC/MAX instructions. Load unit balance is also necessary to better utilize the available memory bandwidth. It is better to break a single large load transaction into multiple smaller loads to prevent the CUs from stalling for incoming data.

Instruction granularity optimization: register assignment, branch delay slot filling and instruction reordering are topics for future work. For this paper, register assignment is statically defined to avoid unnecessary register saving instructions. Branch delay slots are filled and instruction order is manually optimized for a small subset of the main tasks: compute and load. Finding the sweet spot between fully handwritten code and generic pieces of optimized instructions that achieves high-performance for most use cases is up to further study.

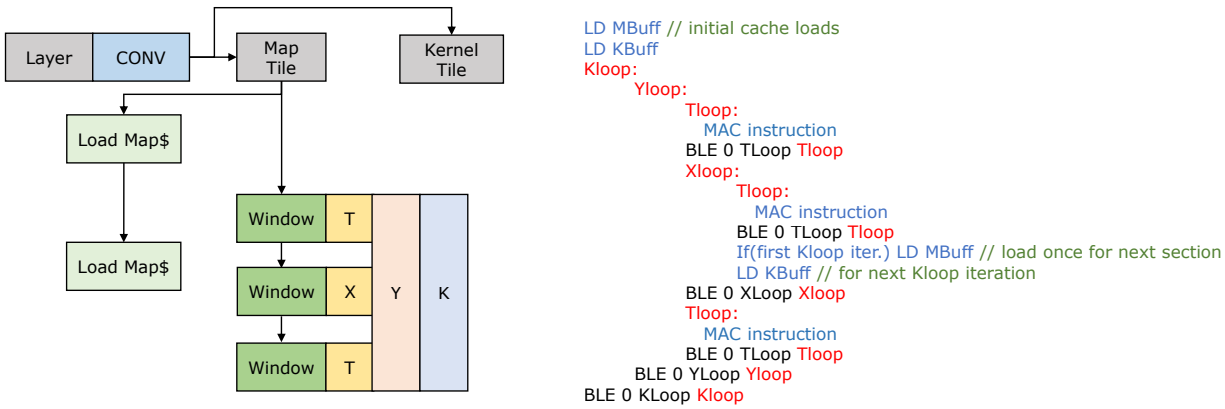


Figure 3: Example of instruction generation for a CONV layer. Different layers create different number of objects, but they all become MAC/MAX for window objects and LD for load objects.

5.3 Instruction deployment

Last task is to run Snowflake. This task loads and arranges weights and biases data from a trained model. The weights and bias need to be arranged differently based on the workload break down and the compute decision made earlier. For instance, each vMAC works on a different kernel in COOP mode, whereas in INDP mode each vMAC works on 16 different kernels, hence in INDP we need to group 16 kernels in one vMAC weight buffer. Instruction deployment task also need to load the input image for the first layer of the model.

Snowflake uses CMA (Contiguous Memory Allocator) for memory to FPGA communication. All data need to be placed into CMA allocated region of memory. Different regions in CMA are allocated according to layer dependencies (step 2 in section 5.1). Different regions are allocated for each layer's weights. After that, some configuration registers enables an initial load instruction to populate Snowflake's instruction cache with the first set of instructions. The software polls an output counter register to check whether processing has finished or not. Finally, for validation purposes, we wrote a software implementation of the model's layers using Q8.8 to simulate Snowflake's compute operations. Result checking allows layer by layer validation.

The data representation of choice for hardware and software was Q8.8, which has been shown to have insignificant accuracy degradation compared to neural networks implemented in 32 bit floating point [10]. Nevertheless, other number representations can be used in the system. Pretrained ResNet18 [2] was profiled on ImageNet dataset [20] using Q8.8 and Q5.11 fixed point precisions. Top-5 accuracy using 32 bit float was 89% , Q8.8 was 84% and Q5.11 was 88%.

6. RESULTS

This section presents Snowball's results. We compared hand optimized instruction stream versus code generated from Snowball. Performance results for AlexNet, ResNet18 and ResNet50 model were measured. Finally, this section presents a discussion of techniques used in Snowball: kernel or maps data loop and communication load balance.

[2]`https://github.com/facebook/fb.resnet.torch`

6.1 Snowball generated instructions

Using all the techniques described in previous section, Snowball generates an instruction stream that achieves performance comparable to handcrafted instructions as shown in Table 1. 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 execution time compared to hand optimized code, which exploits manual optimizations such as filling branch delay slots and instruction reordering. We have only compared some AlexNet layers because models in handwritten instruction are human error prone and tedious. The results for auto-generated instruction for models are shown in table 2. 224×224 images was used as model's input.

Table 1: Hand optimized code (hand) versus auto-generated instructions (auto).

Layer	Code	Time [ms]
27x27,5x5,64,192,1,2	Hand	3.256
	Auto	3.261
13x13,3x3,192,384,1,1	Hand	1.627
	Auto	1.624
13x13,3x3,384,256,1,1	Hand	2.188
	Auto	2.187
13x13,3x3,256,256,1,1	Hand	1.462
	Auto	1.458

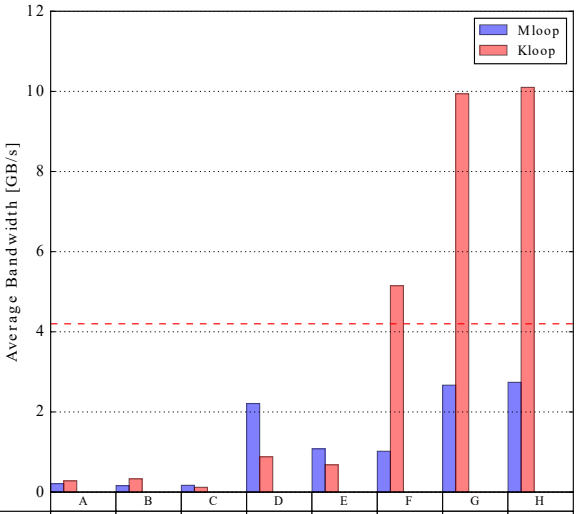
The compiler results show its main contributions: to provide means to test models, ensure output correctness and to allow further exploration. Some inefficiency is caused by cold buffer misses, memory bandwidth limitation and non-overlapped Maxpool layers. Execution time for all models does not account for FC layer times, since FC layers are inherently bandwidth limited operations. Those issues will be addressed in future hardware and software developments.

6.2 Loop rearrangement for bandwidth constraints

Unlike GPUs and ASIC designs, FPGA accelerators are limited mostly by their off-chip memory bandwidth. While a

Table 2: Results for models using Snowball

Model	Exec. Time [ms]	BW [GB/s]
AlexNetOWT	10.68	1.22
ResNet18	46.77	2.25
ResNet50	218.61	1.87



	A	B	C	D	E	F	G	H
Map size	224x224	27x27	128x128	56x56	56x56	28x28	14x14	14x14
Kernel size	11x11	5x5	3x3	1x1	1x1	1x1	1x1	1x1
In channels	3	64	32	256	256	512	1024	1024
Out channels	64	192	128	512	128	1024	2048	1024
Stride	4	1	1	2	1	2	2	2
Pad	2	2	1	0	0	0	0	0

Figure 4: Required memory bandwidth in Mloop or Kloop mode for various CONV examples.

GPU's optimized memory interconnects can achieve a high bandwidth of 112 GB/s [16], Xilinx ZC706 board [24] can achieve 4.2 GB/s bi-directional bandwidth with the AXI ports [23].

Loop rearrangement is a method that reduces total amount of data movement between main memory and hardware accelerator leading 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, 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. Figure 4 shows some examples of CONVs that have lower bandwidth requirement with maps load repetition and vice-versa. Mloop is abbreviation for repeated maps data and Kloop is abbreviation for repeated kernel data. A red dashed line indicates the memory bandwidth limit of the development board. CONVs A and B are from AlexNet model. Their required memory bandwidth is below the limit, so choosing between Mloop or Kloop wont significantly affect performance. CONVs G and H are examples from Resnet50 model and their required memory bandwidth is above the limit for the Mloop mode. Kloop mode is necessary for those layers.

6.3 Communication load balance

Snowflake has 4 load/store units, and properly distributing LD instructions to all units prevents CU stalls due to data transfer. Issuing a single map load to a unit while distributing kernels for all units will lead to unbalanced load units workload. A better approach is to break the maps data into multiple load instructions and distribute evenly with the kernel loads. Communication load balancing optimizes the load unit usage, and thus results in better usage of available memory bandwidth. Load imbalance is a measure of how evenly data is distributed. The percent imbalance metric equation 1 is commonly used [17]. Where L_{max} is maximum load for any load unit and μ_L is the mean load over all load units.

$$C_L = (\frac{L_{max}}{\mu_L} - 1) \times 100\% \quad (1)$$

Table 3: Speed up versus load imbalance.

Load Balance [%]	Speed up
5	1.658
17	1.656
42	1.652
102	1.644
114	1.297
132	1.000

Table 3 shows the speedup achieved by reducing the load imbalance on a CONV 1×1 with 1024 input channels, 2048 output channels and stride 2. The load imbalance percent is measured and averaged out for all tiles. The worst imbalance in the table 3 is the case when kernel and maps uses two load units. The measured speedup in the execution time is compared with the worst load imbalance. This shows that finer load balancing has gains in performance up to a certain limit when data loads are mostly overlapped with vector computation. From this point, reducing memory latency by better load distribution results in diminishing improvements.

7. CONCLUSIONS

This work presented a complete software design flow from high level model definitions created with popular deep learning tools (Torch7) down to custom architecture for accelerating deep learning. Snowball was implemented to provide hardware usability, while efficiently utilizing all hardware resources for various CNN workloads. This work addresses software design points, such as model structure parsing, workload breakdown, loop rearrangement and memory access balancing. Those techniques were tested on the Snowflake custom accelerator, but they can be applied to other custom accelerators.

8. REFERENCES

[1] E. Azarkhish, D. Rossi, I. Loi, and L. Benini. Neurostream: Scalable and energy efficient deep learning with smart memory cubes. *arXiv preprint arXiv:1701.06420*, 2017.

[2] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini. Origami: A convolutional network accelerator. In *Proceedings of the 25th Edition*

on Great Lakes Symposium on VLSI, GLSVLSI '15, pages 199–204, New York, NY, USA, 2015. ACM.

[3] Y.-H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *SIGARCH Comput. Archit. News*, 44(3):367–379, June 2016.

[4] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[5] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *CVPR 2011 WORKSHOPS*, pages 109–116, June 2011.

[6] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello. A 240 g-ops/s mobile coprocessor for deep neural networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2014.

[7] V. Gokhale, A. Zaidy, A. X. Ming Chang, and E. Culurciello. Snowflake: An efficient hardware accelerator for convolutional neural networks. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2017 - in press.

[8] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: efficient inference engine on compressed deep neural network. *CoRR*, abs/1602.01528, 2016.

[9] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[10] J. L. Holi and J.-N. Hwang. Finite precision error analysis of neural network hardware implementations. *IEEE Transactions on Computers*, 42(3):281–290, 1993.

[11] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[12] D. Kadetotad, S. Arunachalam, C. Chakrabarti, and J.-s. Seo. Efficient memory compression in deep neural networks using coarse-grain sparsification for speech applications. In *Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference on*, pages 1–8. IEEE, 2016.

[13] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[14] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[15] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen. Cambricon: An instruction set architecture for neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 393–405. IEEE Press, 2016.

[16] Nvidia. Gtx 960 specs, 2017.

[17] O. Pearce, T. Gamblin, B. R. De Supinski, M. Schulz, and N. M. Amato. Quantifying the effectiveness of load balance algorithms. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 185–194. ACM, 2012.

[18] M. Peemen, A. A. Setio, B. Mesman, and H. Corporaal. Memory-centric accelerator design for convolutional neural networks. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 13–19. IEEE, 2013.

[19] M. Peemen, R. Shi, S. Lal, B. Juurlink, B. Mesman, and H. Corporaal. The neuro vector engine: Flexibility to improve convolutional net efficiency for wearable vision. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pages 1604–1609. IEEE, 2016.

[20] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[21] D. Strigl, K. Kofler, and S. Podlipnig. Performance and scalability of gpu-based convolutional neural networks. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 317–324. IEEE, 2010.

[22] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[23] Xilinx. *AXI Interconnect v2.1*, Nov. 2015. Vivado Design Suite.

[24] Xilinx. *ZC706 Evaluation Board for the Zynq-7000 XC7Z045 All Programmable SoC*, Sept. 2015. v1.5.

[25] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *Proceedings of the 35th International Conference on Computer-Aided Design*, ICCAD '16, pages 12:1–12:8, New York, NY, USA, 2016. ACM.

[26] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 161–170, New York, NY, USA, 2015. ACM.