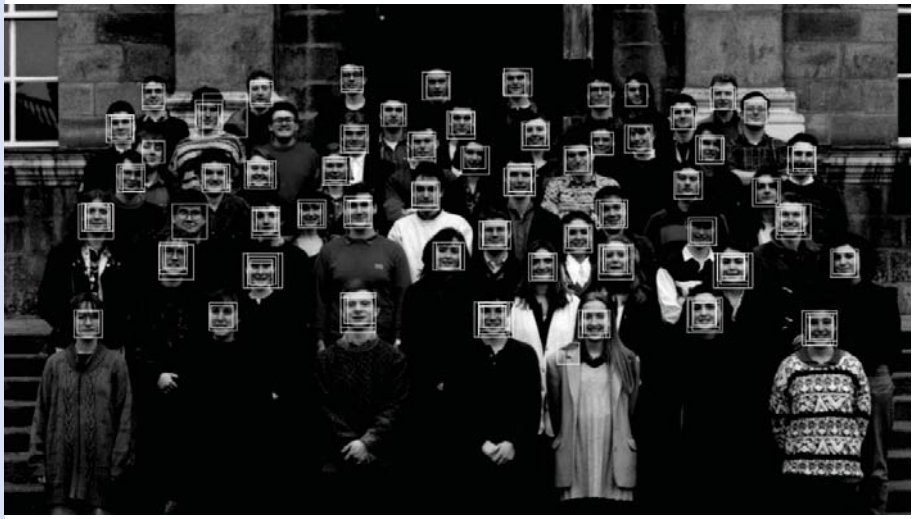
Automatic mapping of convolutional networks on the Neuro Vector Engine

Pramadi, W.

Award date:
2015

Automatic Mapping of Convolutional Networks on the Neuro Vector Engine



Author:
Wisnu Pramadi

Research Advisors:
Henk Corporaal
Maurice Peemen

Date:
August 31st 2016

TU/e

Abstract

Since the last decade Convolutional Networks (ConvNets) achieve state-of-the-art results on recognition tasks. This algorithm classifies complex patterns by performing hierarchical feature extractions, however it requires huge amounts computation and data movement. These requirements do not match with the capabilities of current low power embedded devices. Recently some dedicated hardware accelerators for ConvNets have emerged to answer these problems. These hardware specializations offer flexibility/programmability to support different parameters of ConvNets. To use such flexible architectures users must write programs and their quality determines execution time and energy consumption during run-time. It is error-prone and time-consuming to write such programs manually.

The recent works focused on energy efficient accelerators, but none of them provides a complete and practical programming model. This work is the first that presents an automatic code generation flow to map Convolutional Networks (ConvNets) to a highly specialized VLIW accelerator core targeting the ultra-low power wearable market. By automatically modifying data layout and applying modulo scheduling, the generated code can achieves up to 10% better HW utilization w.r.t. the manually written code by experts. Due the developed feature map fusion technique, which combines programs of multiple feature maps, the generated code has much better locality than code written by experts. It reduces the expensive data transfers to 65% of the transfers in manual code. Finally, this automatic flow reduces the burden of network mapping from several days to mere seconds. With this missing step solved the next generation of smart wearable products can be equipped with intelligent applications.

Contents

Abbreviations

ConvNets	*Convolutional Networks*
ILSVRC	*ImageNet Large Scale Vision Recognition Challenge*
VLIW PE	*Very Long Instruction Word Processing Elements*
NVE	*Neuro Vector Engine*
MAC	*Multiply-Accumulate*
AGU	*Address Generation Unit*
SIMD	*Single Instruction Multiple Data*
DSP	*Digital Signal Processing*
EPIC	*Explicit Parallel Instruction Computing*
FSM	*Finite State Machine*
FIFO	*First In First Out*
DSL	*Domain Specific Language*
XML	*eXtensible Markup Language*
BFS	*Breadth First Search*
ASAP	*As Soon As Possible*
ALAP	*As Late As Possible*
MII	*Minimum Initiation Interval*
DMA	*Direct Memory Access*
BD	*Buffer Descriptor*
CPU	*Central Processing Unit*
GPU	*Graphics Processing Unit*
API	*Application Program Interface*

1. Introduction

In recent years, vision applications become very popular, especially in the consumer market. It ranges from face detection (Figure 1, left) for digital camera, face recognition in social media, image to text for translation, speed sign (Figure 1, right) and pedestrian detection for transportation, and video tracking in surveillance cameras. All of these applications have a similarity, they classify patterns or objects from image or video data. Many algorithms are proposed to address these problems in vision applications. However, every classification problem requires a unique algorithm, which is complex to design and the quality depends on the designer knowledge. Thus, the community has shifted the paradigm of vision application towards machine learning instead of designing specific algorithms to solve the problem. In fact, machine learning outperforms other methods with less effort to design the algorithm. One very popular machine learning algorithm that is used in many vision applications is **Convolutional Networks** (ConvNets) which is originally proposed in [1]. With increasing amounts of data, this algorithm achieves state-of-the-art results on many recognition tasks. It is confirmed by the results on **ImageNet Large Scale Vision Recognition Challenge** (ILSVRC) in the past years where participants such as [2] achieves top performance using ConvNets.

Figure 1 Image annotation results of face and speed sign detection.

A ConvNets consists of several convolution layers where each of these layers performs several simple feature extractions called feature maps. These feature maps across different layers are connected in such a way to construct a hierarchical feature extraction. This hierarchical feature extraction enables the extraction of complicated patterns out of simple feature extractions. The number of feature maps and layers may differ from one application to another depend on the classification complexity and therefore the computation requirements of ConvNets varies. For example, a speed sign detection application requires 1 GMAC per frame which is huge for a video application on embedded platforms. Additionally, a large amount of intermediate results is produced between layers. This huge amount of intermediate results may require large memory space which is usually unavailable on-chip and hence external memory is required. The data transfer is really huge and potentially become a bottleneck. Moreover, the already problematic computation and data transfer also increase with increasing input image size.

ConvNets algorithm has many opportunities for acceleration by exploiting the parallelism since most of the operations can be done concurrently. Data transfer can also be lowered by improving the locality to exploit data reuse. Furthermore, low-precision math can be used to reduce energy per operation in expense of accuracy as stated in the work of [3]. Many accelerations on the general purpose CPU and GPU platform are done to improve the performance of ConvNets, although this remains a challenge for resource-constraints embedded platform where execution time, area, and energy are important. These motivate the community to develop a dedicated architecture for ConvNets. Specialized architecture is required to exploit the parallel nature of the algorithm and

improve locality to reduce the data transfer effort. On the other hand, ConvNets parameters such as layers, feature maps, kernel size may change in different recognition tasks which requires support for flexibility. It is very challenging to reconcile architecture specialization and right amount of flexibility for ConvNets. All the earlier works have focused on efficiently implementing the compute primitives, but none of them provides a complete and practical programming model. They voluntarily ignore programming for simplicity, or they refer to an ad-hoc and therefore impractical assembly program. Prior works on a specialized **Very Long Instruction Word** (VLIW) accelerator for ConvNets are presented in [4] [5] [6] [7]. The core is specifically designed to efficiently run ConvNets algorithm and yet give enough flexibility for different type of networks through VLIW style programming. However, manual programming is still required, which is error prone and time consuming for the users. Additionally, knowledge of the architecture is also required to produce quality code. This work presents a ConvNets code generator for the core mentioned above. The aim is to replace the complicated manual assembly writing by an automatic flow that converts a flexible high level network description into an optimized VLIW assembly program. The contributions of this work are listed as follows:

- An automatic code generation flow to map ConvNets algorithm on the **Neuro Vector Engine** (NVE). The flow consists of task graph construction, scheduling, and local buffer allocation (section 4).
- Code optimization steps to improve the quality of automatic code generation (section 5).
- Evaluations on NVE demonstrator in a real setup (section 6.3.1 and 7.1).
- Evaluations on the code quality of the automatic code generation flow (section 6.3.2 and 7.2).

The results show that the code generator is able to match the code quality of manual programs that are written by architecture experts. In the best case, the automatic code generator achieves 10% better results, whereas in the worst case it has 20% less hardware utilization compared to the manually written programs. The automatic code generator is always better in reducing data transfer due to feature maps combining that increase the amount of data reuse which reduce data transfer up to 65%. Nonetheless, some layers have no reduction at all because there are no shared input feature maps.

This thesis report will discuss further about the ConvNets algorithm in chapter 2. In chapter 3, a flexible hardware accelerator for ConvNets, NVE, is introduced. Later in chapter 4, an automatic code generation flow for NVE is presented. Further optimization steps for the code generation flows are discussed in chapter 5. Next, evaluations are conducted as described in chapter 6 and the results are presented in chapter 7. Several related works are also discussed in chapter 8. Finally, the conclusion and future works are presented in chapter 0.

2. Convolutional Networks

ConvNets is a type of artificial neural network, which has several layers of spatially overlap neurons. Like many other machine learning techniques, ConvNets are trained before one can use them and it involves a large number of datasets. The quality and amount of datasets have direct impact on whether or not the networks can generalize the classification problems and achieve high accuracy. The trained networks are used without additional training. However, using ConvNets requires high compute power and data transfer capabilities. The following subsections discuss the computational overview and improvement opportunities of ConvNets in details.

2.1. Computational Overview

ConvNets consist of several layers where a particular type of processing is done on the inputs, namely convolution, pooling, and classification. These layers are connected in a hierarchical way to find the desired pattern from the input as depicted in Figure 2. In the first layer (L1), several feature maps are extracted from the input image. Next, these feature maps are used in the next layer for the same feature extraction procedure. Last, this hierarchy of feature extractions is ended by classifying the inputs into one of the classes. For different type of classification, the network configuration such as number of layers, the number of feature maps, number of classification output, and feature map connection may differ.

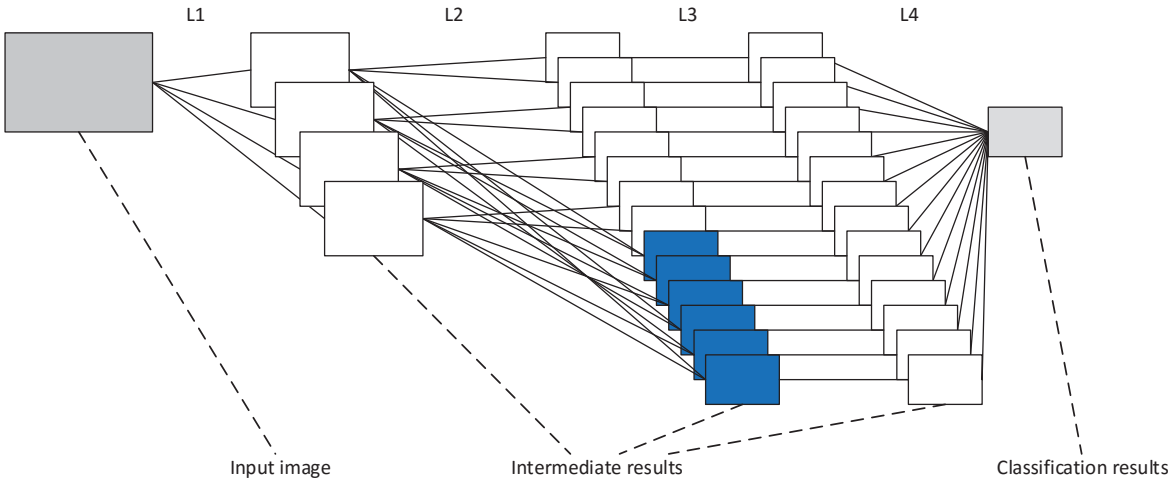


Figure 2 The networks of face detection application.

2.1.1. Convolution Layer

The convolutions are performed to extract interesting features such as edges and corners from the input images. The output of convolutions is calculated from accumulation of multiplication between 2D weights and parts of the input image as presented in Figure 3. These calculations are performed repeatedly all over the image where the convolutions are spatially overlapped with each other.

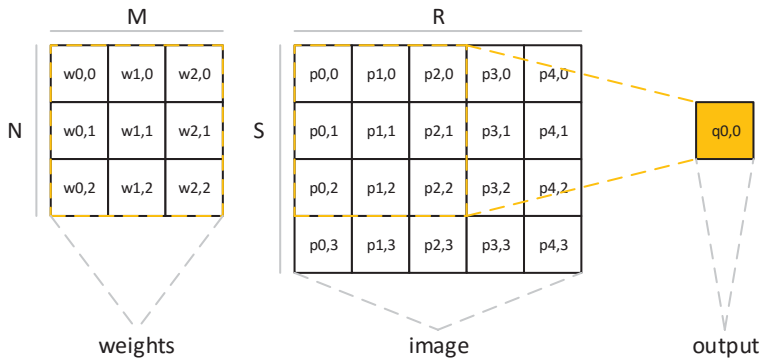


Figure 3 A 3x3 image convolution; 2D weights (left), image (middle), convolution output (right).

The convolution operations are mathematically expressed as in Equation 2.1 where q is the output pixel, b is the bias value, w is the weight, and p is the input pixel. Output pixel q at x, y position (2D Cartesian plane) is a result of an $M \times N$ convolution kernel. All the weights of the kernel are multiplied with input pixels with $M \times N$ dimension at x, y offset position; these multiplications are summed along with the bias value. To compute the entire output feature maps, all output pixels are calculated iteratively for all x, y positions (q in all positions). After the entire convolution is performed, an activation takes place which giving a non-linear saturation effect. Sigmoid function (Equation 2.2) is commonly used for activation, although some recent work proposed other alternatives.

$$q_{x,y} = b + \sum_{j=0}^{N-1} \sum_{i=0}^{M-1} w_{i,j} \times p_{x+i,y+j} \tag{2.1}$$

$$= b + W * P_{x,y}$$

$$S(q) = \frac{1}{1 + e^{-q}} \tag{2.2}$$

2.1.2. Pooling Layer

Small spatial differences may occur over different patterns in image samples. For example a certain object in an image is slightly bigger, smaller, or shifted in a certain direction compared to the rest of the samples; it introduces a small variance over the spatial domain. To ensure the classification results are consistent with small differences in spatial domain, subsampling operations are added to the network in form of pooling layer. The subsampling takes a non-overlapping rectangular region from input images and calculate the average or the maximum value.

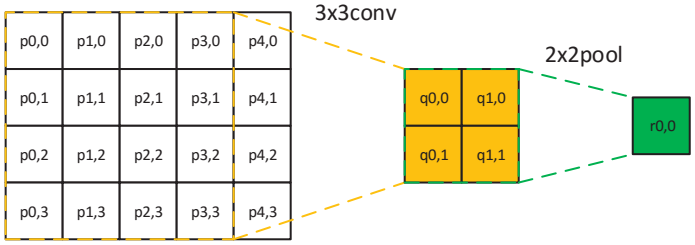


Figure 4 A 3x3 convolution layer followed by a 2x2 pooling layer.

As depicted in Figure 4, four 3x3 convolution produces four output pixels and followed by 2x2 pooling layer where it produces one pixel. The pooling output has smaller dimension compared to its input due to subsampling. Instead of computing the convolution and pooling in two separate layers, the work of [8] combines these two computations as depicted in Figure 5. It shows a 3x3

convolution layer combined with 2x2 pooling layer which produce a convolution of 4x4 window with a stride of 2 for both x and y direction. This reduces the amount of computation required for both layers.

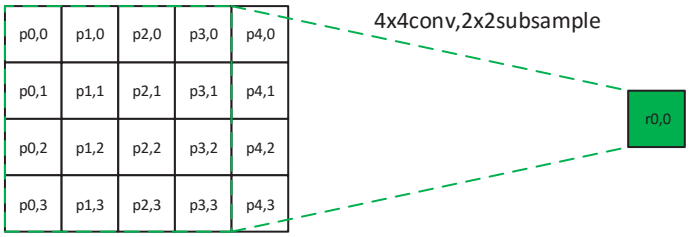


Figure 5 Combination of 3x3 convolution layer and 2x2 pooling layer.

2.1.3. Classification Layer

The last stage of convolutional network is classifying the result based on the extracted features. As shown in Figure 2, the classification layer takes all the feature maps from previous layer and combine them to define in which class the input belong. There are no input overlap across neighboring convolutions in this layer. The example is shown for a single class problem where the classification layer only classifies as *yes* (part of the class) or *no* (not part of the class). For more complex classification where it is required to classify multiple class, the classification layer may have multiple classification at the end of the network. The classification layer produces scale images that contain the information of classification results along with the associated position in the image.

2.1.4. Computational and Data Transfer Requirements

The computational requirements of ConvNets greatly vary depend on the complexity of the classification problem, although the main contributor of the computation is convolutions. As presented in Table 1, face detection which has one classification problem required significantly less number of **Multiply-Accumulate** (MAC) operations compared to speed sign detection which has eight classification problems. We also need to consider the amount of data transfers that are required by ConvNets, especially in high definition input image. The intermediate frames may become too large for on-chip memory to handle which mean the data have to be transferred from and to larger external memory. The table shows the scenario where each pixel is transferred once, whereas in a real case pixels might be transferred multiple time.

Table 1 The required MAC and data transfer of face and speed sign detection.

Recognition Tasks on 1280x720 frame size		Parameters				MAC	Data transfer (MB)
		Number of feature maps	Feature map size	Kernel size	subsample		
Face detection	L1	4	638x358	6x6	2x2	32.9M	1.81
	L2	14	317x177	4x4	2x2	17.95M	1.62
	L3	14	312x172	6x6	1x1	27M	1.47
	L4	1	312x172	1x1	1x1	0.75M	0.72
	Total					78.6M	5.62
Speed sign detection	L1	6	637x357	6x6	2x2	49.1M	2.25
	L2	16	317x177	6x6	2x2	121.2M	2.17
	L3	80	312x172	5x5	1x1	858.6M	4.98
	L4	1	312x172	1x1	1x1	4.3M	4.1
	Total					1.03G	13.5

2.2. Parallelism and Data Reuse Opportunity

Section 2.1.4 presents the huge compute and data transfer requirements of ConvNets. Despite of that, ConvNets have many opportunities for improvement. The independent convolutions can be parallelized to improve the throughput of the algorithm as shown in Figure 6. Several neighboring convolutions are computed at once and an array of output pixels is produced. This parallelism also introduces spatial locality among neighboring convolution which reduce the data transfer compared to sequential computation (left). There are also temporal locality when the vector convolutions move to the next window (right). Some pixels that are used in previous vector convolutions are used again by the next vector convolutions. The pixels in the middle of the vector are completely reused (transfer only once), whereas pixels on the boundaries are transferred again if the reuse distance is larger than the available local buffer. Nevertheless, the average data transfer of the pixels is close to one.

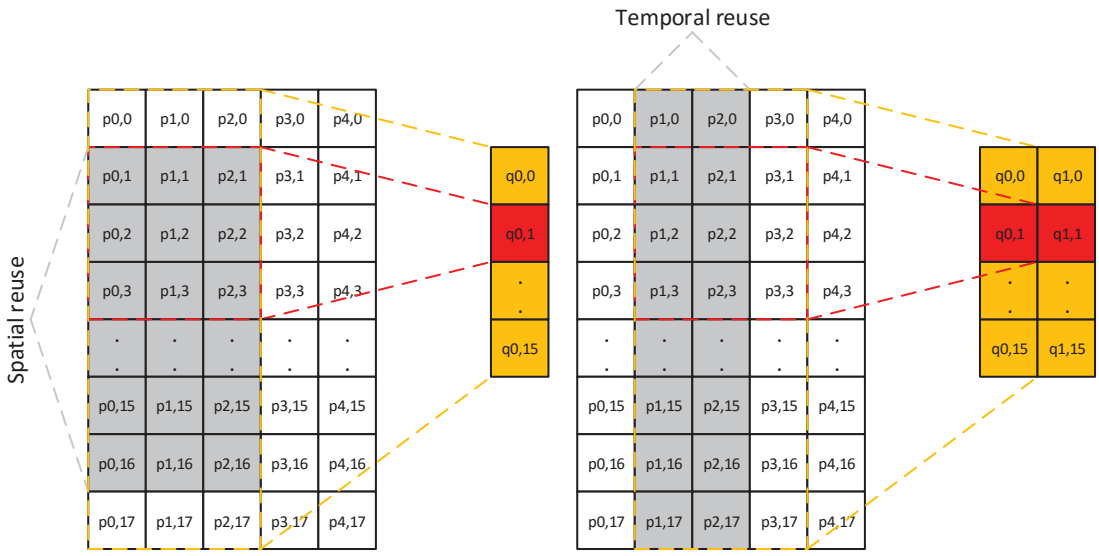


Figure 6 Neighboring convolutions are executed in parallel (vertically).

3. Neuro Vector Engine: A Dedicated ConvNets Accelerator

As discussed in section 2, ConvNets achieve state-of-the-art accuracy, but at the cost of massive compute and data transfer workload. If we aim for mobile application of this algorithm, huge improvements are required in processing efficiency. A well-known solution for this efficiency gap is specialization to dedicated hardware accelerators. Besides compute efficiency, ConvNets also requires flexibility to support different parameters in different type of recognition tasks. The problem with the hardware accelerator design is how to reconcile the specialization of the architecture and provide the right amount of flexibility.

The **Neuro Vector Engine** (NVE) is specifically designed to exploit the parallel nature of ConvNets and exploit data reuse opportunity. Furthermore, the architecture is programmable to give the users flexibility in changing the parameters of the ConvNets applications. Figure 7 presents the NVE which consist of three main components: the datapath, the **Address Generation Unit** (AGU), and the control logic. The data path executes the compute primitive of ConvNets according to the instructions that are fetched from instruction buffer. The AGU updates the address field of the instructions during the execution and the control logic controls the instruction fetch to the datapath according to the configurations on the setting registers. The following sections discuss these components in details and how the architecture is programmed.

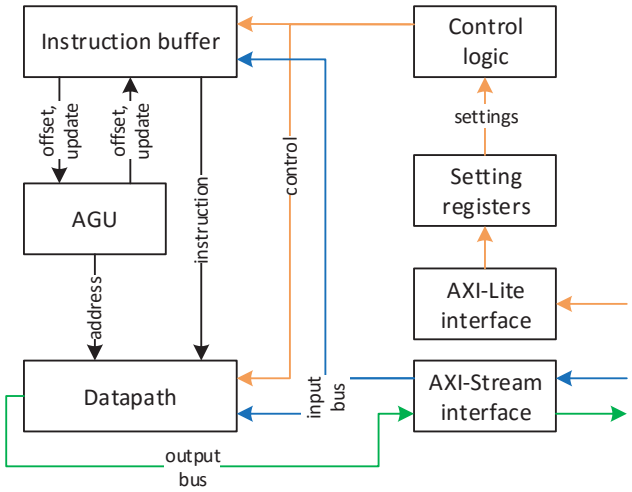


Figure 7 Internal diagram of NVE.

3.1. Datapath

Architecture specialization is key to achieve better performance. Hence, the NVE is specifically designed to efficiently run the computation flow of the ConvNets algorithm. The datapath of the NVE spans across multiple pipeline stages as depicted in Figure 8. The architecture exploits the parallel nature of the ConvNets algorithm through its **Multiply-Accumulate** (MAC) units which are executed in **Single Instruction Multiple Data** (SIMD) style. The NVE is programmable with **Very Long Instruction Word** (VLIW) style where each of the instruction field describe the operation in successive pipeline stages.

①The main operation of convolution is done in the vector MAC where several convolution windows are calculated at the same time. In the current design, the vector MAC supports 16 parallel MAC operations. The computation use 16-bits fixed-point for input weight, 8-bits fixed-point for input pixel, 32-bits accumulation registers to prevent overflow. To compute the convolution, the vector MAC set the bias value to the accumulation registers ($\boldsymbol{accu} = b$). Next, the vector MAC starts calculating the convolution window by performing MAC operations between weights and pixels ($\boldsymbol{accu} = \boldsymbol{accu} + w * \boldsymbol{p}$). These MAC operations are repeated until the entire 2D kernel is calculated. When the vector convolution is done, the results are moved to the activation output. It takes three cycles for the vector MAC to provide results for the activation stage.

②The activation function receives 32-bits results from the vector MAC. However, it only uses 10-bits of the results because the precision is sufficient for ConvNets without significantly degrade the classification accuracy as mentioned in [7]. The activation function is implemented as a lookup table with 1024 entries (10-bits) because it is cheaper compared to actual math operations. It takes two operations for the activation function to produce the output vector because the activation function has a smaller vector size compare to the vector MAC (half the size). Each of these operations takes one cycle to produce results. The results of activation stages are written to the output bus or to local buffer for further processing.

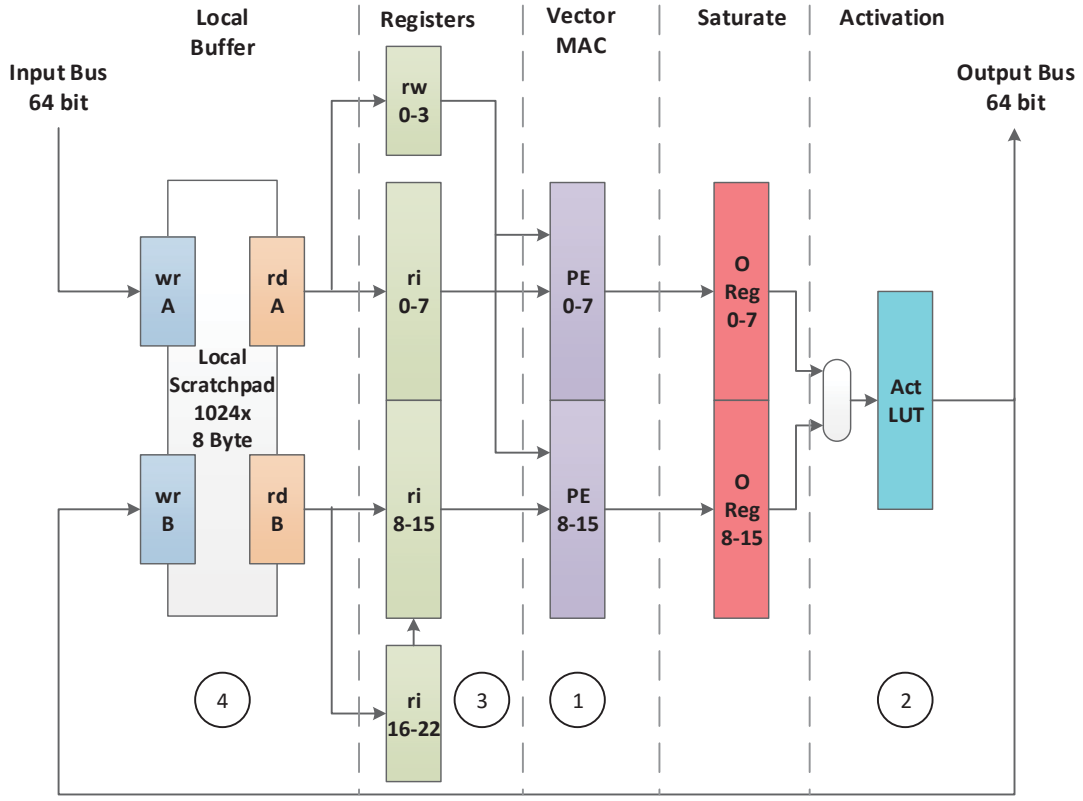


Figure 8 NVE datapath.

③The convolution computation requires a coefficient kernel and pixels which are provided by the registers. Both bias and weight coefficients are broadcasted from the weight register to all MAC (same value for every MAC). The weight register `rw[0-3]` can hold up to 4 coefficients where the broadcasted value can be changed if the register is shifted. For the pixels,

they are provided by the image registers as a vector. Image register `ri[0-15]` provide the pixel vector for MAC operations and `ri[16-22]` provide additional pixels for shift operations. Each of these register operations has a latency of one cycle.

④The local buffer is designed as a *true dual-ported memory*. All data (coefficients and pixels) are transferred from external memory to local buffer through the write port A. For the write port B, it receives data from the output port of the accelerator. It takes two cycles to write data into the local buffer before they can be read. The data that are written into the local buffer are later used by the accelerator by loading them to the registers through the read ports. Read port A is connected to weight register and image register `ri[0-7]` and read port B is connected to image register `ri[8-15]` and `ri[16-22]`. Read operations take two cycles and therefore the corresponding register operations are executed two cycles after the read operation.

3.2. Address Generation Unit

Programmability is very important nowadays, especially with constantly changing applications that require high flexibility to change the algorithms. However, programmability introduces overhead because fetching instructions from main memory requires energy. With increasing complexity of the application, the program size increases and becomes the bottleneck of the application. One solution to this problem is by finding repetitive execution patterns. Additionally, a local instruction buffer is introduced into the design which efficiently reuse repetitive instructions and reduce the energy consumption. This is especially effective for **Digital Signal Processing** (DSP) applications, they often require many iterations with similar operations.

In typical processor, the repetitive part is expressed as a loop iteration, which requires a control instruction (branching) to perform the iteration. The control instruction itself may introduce additional cycles of computing the branching or the loop boundaries and then perform the jump instruction. Instead of introducing control instructions, another possible approach is by providing hardware support for loop execution.

The NVE has a hardware support to iteratively execute a code segment by defining the start address, end address, and number iterations in the setting registers. However, only the MAC, register, and activation instruction fields that are repeated, whereas the local buffer operations require new addresses over iterations. This problem is solved by the address generation unit (AGU) by updating the address field of the VLIW instructions in every iteration. To allow this feature, the setting registers are assigned with information on how the address field should be updated (toggle & increment number).

$$address = offset + update \tag{3.1}$$

$$update = \begin{cases} update + inc, & if\ update + inc \neq toggle \\ 0, & otherwise \end{cases} \tag{3.2}$$

The address field consists of three parts: `offset`, `update` and `mode`. As depicted in Figure 9, the `offset` field gives an offset address and the `update` field increments and toggles its value to give addressing pattern (Equation 3.2); the address of current data is obtained by adding these two fields (Equation 3.1). The `mode` field marks whether or not the instruction requires a pattern addressing. The `update` field increment in every execution of the instruction (`update` = `update` + `inc`) if the `mode` field is enabled. If the `update` field equal to `toggle` value, it is set to 0 instead of incrementing. This produces an address range start from `offset`, increment every `inc`, and reset again if it reaches `toggle`. This circular addressing method is similar to rotating register in the **Explicit Parallel Instruction Computing** (EPIC) architecture.

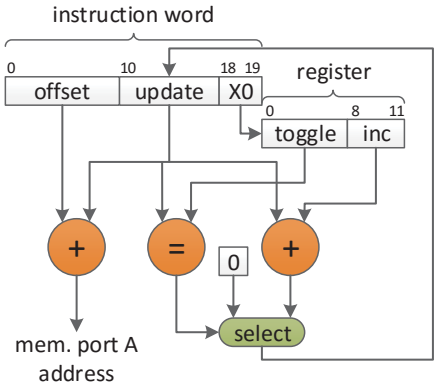


Figure 9 Address generation unit.

This support is useful for executing ConvNets algorithm where new pixel vectors are written into the local buffer, but the rest of the pixels are reused from previous iteration as shown in Figure 10. The first convolution (left) uses pixel column 0-1-2 from the local buffer. In the next convolution window (next iteration), it uses pixel column 1-2-3. Pixel column 3 is not available yet in the local buffer and therefore it is written into the local buffer. Instead of allocating new address space, pixel column 3 uses the space of pixel column 0 because this pixel column is not required anymore in the current iteration. The read address also follows a similar pattern. Supposed that pixel column 0 resides at the address 3, the first convolution reads from address space 3-4-5 to obtain pixel column 0-1-2. In the next iteration, the read address change to 4-5-3 to obtain pixel column 1-2-3.

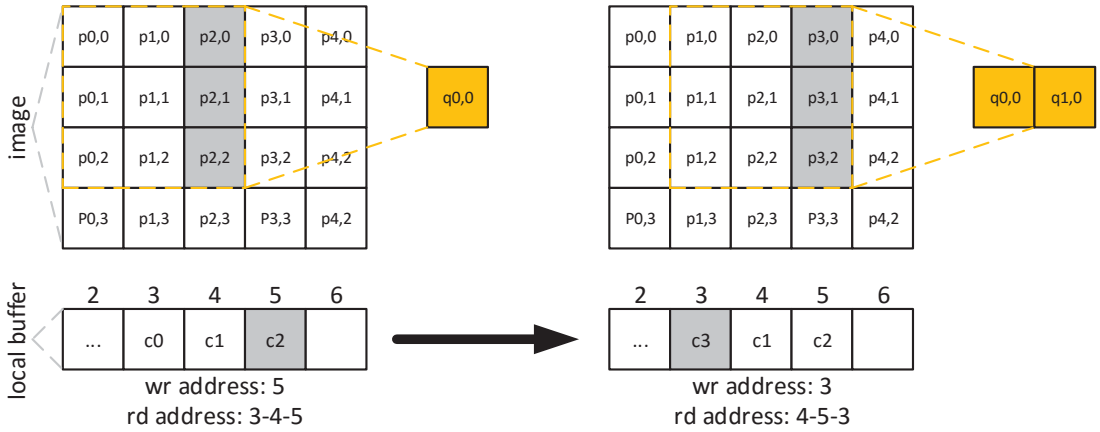


Figure 10 Read and write data using circular addressing.

3.3. Control logic

To run the datapath, a control logic is required to fetch proper instructions. The control logic also controls how the data move in and out the accelerator. To perform this, the control logic requires information on how the program should run. Setting registers are provided inside the NVE to store the settings of the accelerator such as: program size, starting address of repetitive part, and the number of iterations which determine the program flow. An AXI-Lite interface is provided to access these registers from the host processor.

The control logic is implemented as a **Finite State Machine** (FSM) as pictured in Figure 11; the presented figure is a simplified version of the actual implementation. The control logic starts

by loading the instructions into the instruction buffer. Next, it fetches instructions of the initialization part to the datapath. When it reaches the starting address of the repetitive part, the state machine moves to the repetitive execution and keeps track the number of iterations. After a certain number of iterations as specified in the setting registers, the control logic finalizes the execution by running the last part of the code.

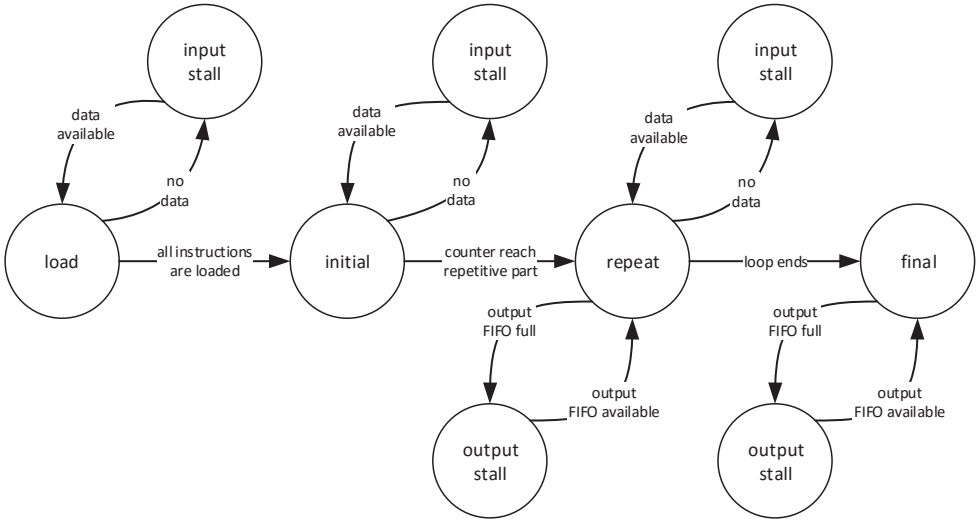


Figure 11 FSM of control logic.

Data are transferred from and to external memory during the execution time. The current implementation uses AXI-Stream interface to move the instructions, coefficients, and pixels. The control logic decides when to write the data from the interface into the instruction buffer or local data buffer and when to write data from the accelerator output to the interface. Conditions such as the unavailability of data upon request and the inability of the output interface to deliver data (caused by full output FIFO) may occur. On such cases, the control logic stalls the accelerator, enter the input or output stall state, to prevent faulty results.

3.4. Execution Flow and Programming

The execution flow of NVE is optimized for computing convolutions. First, the datapath loads the coefficients and pixels into the local buffer. These data are later loaded into the registers. Next, the vector MAC calculates the convolution of several convolution window in parallel. Finally, the results of iteratively computing the MAC are moved into the activation unit and the outputs are produced.

To better understand how the NVE improves the compute efficiency of ConvNets, consider a 3x3 filter example as illustrated in Figure 12. The SIMD execution style exploits spatial locality among neighboring convolution by just shifting the image register. The first vector MAC execution computes the input pixels $p_{0,0} - p_{0,15}$ which are loaded into `ri[0-15]`. Pixels $p_{0,16} - p_{0,17}$ are also loaded into image register `ri[16-22]` to provide additional pixels for shifting. The next MAC operation computes the input of input pixels $p_{0,1} - p_{0,16}$ by only shifting the register and then continue to input pixels $p_{0,2} - p_{0,17}$. Up to this point, the convolution is done for the first column and two more columns remain. The same procedure is repeated for the rest of the columns by loading the pixel column into the register. When the entire window of 3x3 filter is finished, one output vector is produced.

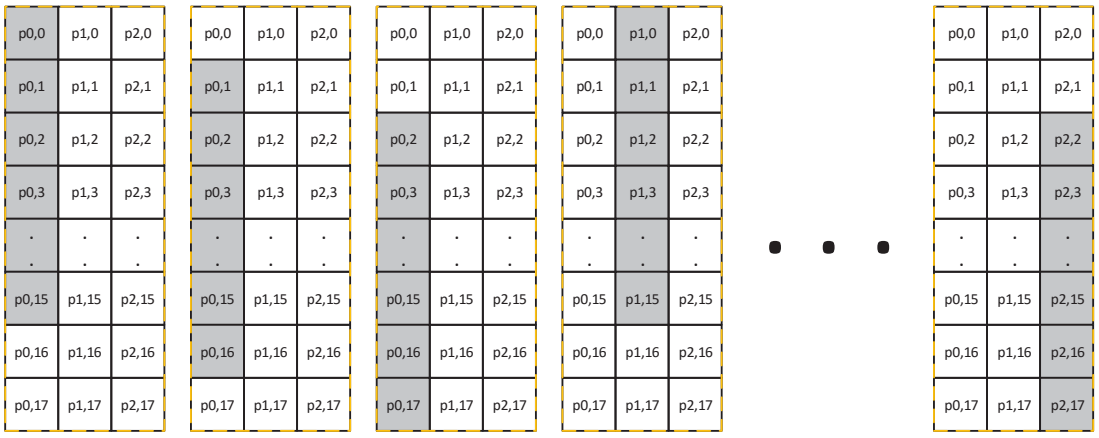


Figure 12 Execution flow of 3x3 filter on NVE.

To perform such computation flow, the programmer must provide instruction sequence for all pipeline stages and arrange those instructions into a VLIW program. There are 6 fields in the VLIW instruction of NVE as depicted in Table 2. The local buffer supports two parallel instruction field of port A (MEMA) and B (MEMB) which support read and write operations. For the register, two distinct instruction fields of weight (WREG) and image (IREG) register are available. Both instruction fields support set operation to load value from read port to the register and shift operation to shift the data within the register. Additional set-shift operation is supported by the IREG to utilize the special image register `ri[16-22]` which can load data to register and shift at the same time. The vector MAC support set bias operation to load bias coefficient from weight register to the accumulation registers and MAC operation to compute the convolution window. The activation output only needs to perform activation operation, although the activation can only be done on half the vector of vector MAC. The activation function can select whether to perform activation on the first 8 or the last 8 of the vector MAC output.

Table 2 List of NVE instructions.

Stage	Supported instructions
MEMA	`rd` `wr`
MEMB	`rd` `wr`
WREG	`set rw` `shift rw`

IREG	`Set ri`
	`set-shift ri`
	`shift ri`

VMAC	`set rb` `mac`
ACT	`act`

The programmer must be careful on scheduling these instructions because there are some architectural constraints applied. Due to register connection with the read ports of local buffer as explained in section 3.1 (Figure 8), set weight register occupied port A which cause conflict with set image `ri[0-7]` if they are scheduled at the same time. The distance between read local buffer operation and set register also requires attention because if the latency between read operation and its set register operation is more than 2 (latency of read operations), the programmer has to ensure that there are no other read operation in between, otherwise the data in the read port will

be overwritten. Another important point is the relation between register operation and MAC operation. The programmer must be sure that the vector MAC receives correct data from the registers and they are not overwritten by successive register operations.

Several constraints and rules of the NVE architecture have been discussed. It is very important to consider these constraints during programming. The next chapter will discuss about the automatic code generation flow, which generates programs that satisfy all the constraints within the architecture.

4. Automatic Code Generation Flow

Section 3 presents a flexible accelerator for ConvNets. However, programming the accelerator as shown in section 3.4 is performed manually in complex assembly. This manual programming effort is a bottleneck for adopting the core and therefore an automatic tool is mandatory.

In this section, an automatic code generation flow for NVE is introduced. As depicted in Figure 13, the code generation flow requires a high level description of the algorithm. Next, the network description is converted into a series of task graphs (task graph generation), more specifically each output feature map has a separate task graph. For each graph a VLIW schedule is created (instruction scheduling). Finally, all individual programs are translated into binaries to produce the executable VLIW program. To implement this flow, *Boost* library [9] is used. *Boost* is an open-source C++ library with many functionalities. In this code generator, XML parser and graph (data structure, creation, and manipulation) functionalities are used. The following subsections discuss all those steps in details.

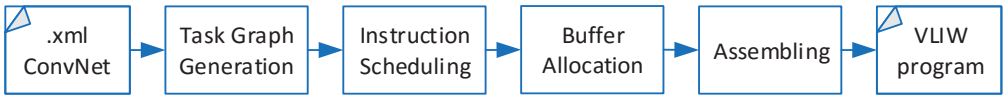


Figure 13 Automatic code generation flow from high level abstraction to executable programs.

4.1. ConvNets Domain Specific Language

Programming a programmable accelerator can be done using common programming language such as C, although this kind of programming language offers much more flexibility than needed. The flexibility of such programming language may raise problems because the users have the freedom to write anything, including operations that are not supported by the accelerator. For highly specialized architecture like NVE, a limitation should be introduced in the programming to prevent mistakes in programming and reduce the effort of producing quality code. The solution is by specifying the networks in high level abstraction; in this case, a **Domain Specific Language** (DSL) using XML format is used.

Listing 1 Network description of first layer face detection application.

```
<?xml version="1.0" encoding="UTF-8"?>
<convNetDescriptor name="Face Detector" layerCount="4">
      <layerDescriptor idx="0">
            <outHeight>358</outHeight>
            <outWidth>638</outWidth>
            <kernelHeight>6</kernelHeight>
            <kernelWidth>6</kernelWidth>
            <subHeight>2</subHeight>
            <subWidth>2</subWidth>
            <outCount>4</outCount>
            <outNet idx="0" cnt="1">0</outNet>
            <outNet idx="1" cnt="1">0</outNet>
            <outNet idx="2" cnt="1">0</outNet>
            <outNet idx="3" cnt="1">0</outNet>
      </layerDescriptor>
</convNetDescriptor>
```

Listing 1 contains a description of a network layer similar to layer 1 in Figure 2 (complete examples are available in Appendix A). The XML lists the ConvNets parameters such as the width and height of the feature-maps, kernel size, and subsample size. Also the connections between feature-maps are specified, e.g. `outNet` specifies the connection of an output feature-map `idx` with a number of input feature maps `cnt`, in this case one input feature-map.

4.2. Task Graph Construction

To construct a task graph, proper instructions must be generated from the high level description. First, computation instructions are generated; a number of set bias and MAC operations are generated based on the kernel size. Because these MAC operations require coefficient and pixel data, weight and image register operations are also generated. The number of image register operations is directly derived from the number of MAC and subsample size. For the weight register operations, they are directly derived from the number of set bias and MAC operations. Next, these register operations require data from the local buffer and hence read local buffer operations are generated according to the register load operations. To perform these read operations, the data must be written beforehand and therefore write operations are generated. Redundant read operations may share the same write operations as long as the data is not overwritten. At last, activation operations are generated.

To explain this procedure in more detail, consider a 3x3 filter as an example. The computation consist of 1 set bias operation and 9 (3x3) MAC operations. The image register operations followed the number of MAC and subsample size, which consist of 1 set, 1 set-shift and 1 shift operation which are repeated 3 times (3 set, 3 set-shift and 3 shift; 9 operations in total). On the other hand, the convolution requires 1 bias and 9 weights which requires 10 weight register operations. The weight register always load 4 coefficients (full capacity) and shift the register to access them in order. This makes 10 weight register operations consist of 3 sets and 7 shifts. For every set register operation, read operations from local buffer are required. The image register has 3 set operations which require 6 read operations because it must read from both ports (set to register (0-15) requires read from port A and B). Additionally, 3 set-shift require 3 read operations through port B. On the other hand, the weight register has 3 set operations which require 3 read operations from port A. This simple example only demonstrates convolutions without subsampling. If the subsample of the convolutions is two, the number of set image register operations become twice due two larger vector size.

After the instruction compositions are decided, these instructions are connected to each other based on their dependencies. First, the set bias and the first vector MAC operations are connected, as detailed in Figure 14 (left). The rest of vector MAC operations are not connected to each other. Because the vector MAC operations require data from the registers, dependencies from register operations to vector MAC operations are connected (middle). Successive register operations are also connected (right); shift register operation depends on previous operation, whereas set register operation independent from previous operation because it overwrites the content of the registers. Anti-dependencies from the vector MAC to the registers are also added as depicted in Figure 15 (left). These anti-dependencies ensure that the weights and pixels are not overwritten until the associated vector MAC operation take place. Finally, dependencies between set register operations and read operations are connected (Figure 15 right). These read operations also have dependencies with write operations. All dependencies have a *distance* parameter representing the minimum latency between adjacent vertices as specified in section 3.4.

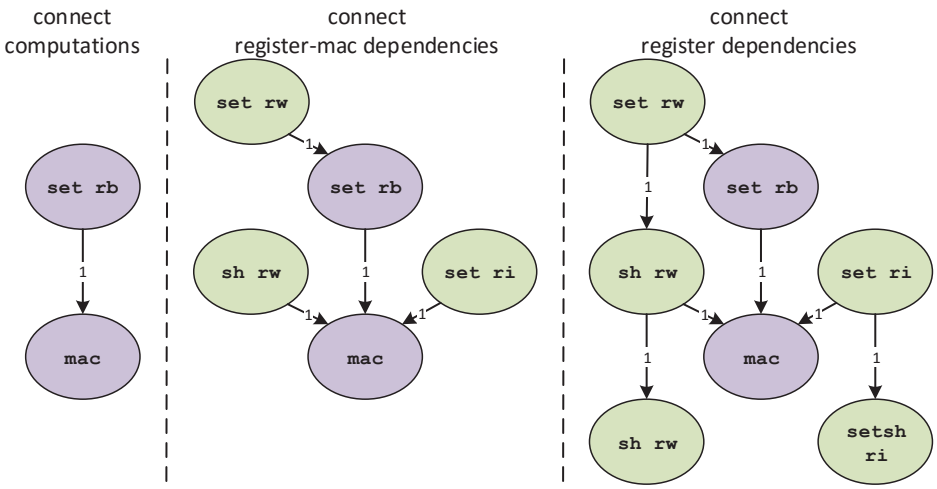


Figure 14 Connecting dependencies between set bias and first MAC operations (left), between register and MAC operations (middle), and between register operations (right).

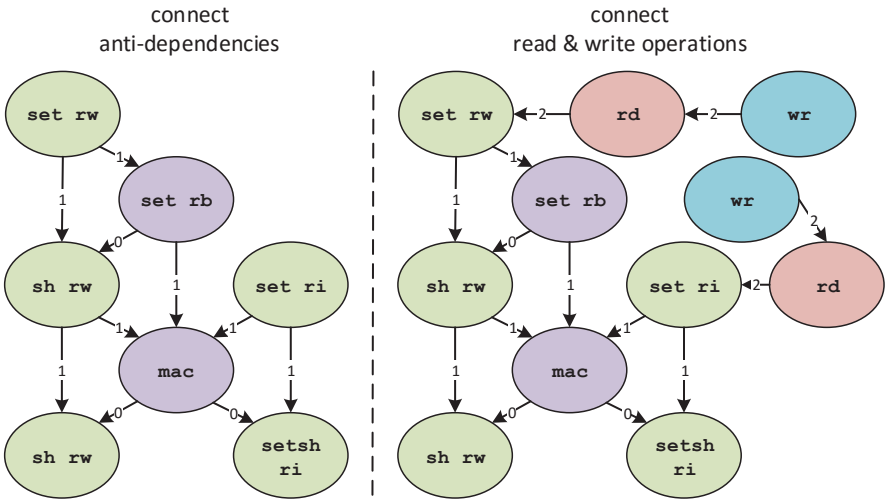


Figure 15 Connecting anti-dependencies between MAC and register operations (left) and dependencies between memory and register operations (right).

4.3. Instruction Scheduling

To schedule the operations of the task graph, as described in section 4.2, a custom scheduler is developed. Algorithm 1 lists the procedure to produce a VLIW schedule from a task graph. Write and activation operations are excluded in this scheme because additional steps are taken to schedule these two operations. The scheduling is performed by visiting the vertices of the graph from bottom to top. The bottom-up direction is chosen because it has a single starting point for graph traversal whereas top-down has many starting vertices. Path information is required to pick a starting point if the top-down approach is used. Scheduling is done by visiting all the vertices and assign a time slot for them. In this algorithm, **Breadth First Search** (BFS) is used to visit and schedule the vertices. The BFS starts by adding the bottom vertex (x) of the task graph (G) to the queue (Q) and then visits all its parents one-by-one. During the visits, a time slot (t) is assigned to

the visited parent vertices (w) based on its child's (v) time slot and the distance (d). All the visited vertices are stored in the queue for next level search of their parents.

Algorithm 1 Instruction scheduling

input: graph G and pointer to bottom vertex x
output: graph G with scheduled vertices

```
create queue Q
create table T
t_x ← 0                                  ◅ assign initial time slot
s_x ← visited                            ◅ assign vertex status
insert x to Q                            ◅ insert the bottom vertex to the queue
while Q is not empty do                  ◅ iterates until the queue is empty
    v ← first element in Q
    for all edges e from w to v in G do  ◅ find all parent vertices
      if i_w ≠ wr and i_w ≠ act then     ◅ exclude write and activation operations
        if s_w ≠ visited then            ◅ first visit
          t_w ← t_v - d_e
          s_w ← visited
          insert w to Q
          if i_w = set r then            ◅ for set register operation only
            while t_w exist in table T do ◅ back-off until no conflict
              t_w ← t_w – 1
            insert t_w to T
        else                             ◅ already visited before
          t ← t_v – d_e
          if t < t_w then                ◅ revise if time slot is earlier
            t_w ← t
```

Instructions in the task graph have many connections. One instruction may depend on more than one and vice versa as depicted in Figure 16. As a result, the BFS may visit the same vertex multiple times and calculate a new schedule. If the calculated slot requires earlier time than the previously assigned, the scheduler updates the time slot to comply with all dependencies; otherwise, the previous time slot is kept. The update of new slot mostly appears due to conflict in set register operations (usage of port A). To handle the conflict, the scheduler keeps track of set register operations that have been assigned using a table (T). Every time the scheduler allocates a time slot for a set register operation, it checks whether the calculated time slot is occupied by another set register operation by comparing to the table. If the schedule is occupied, the schedule is delayed until it finds a proper schedule that is not conflicted with another set register operation (back-off).

Table 3 Partial schedule of 3x3 filter example.

cycle	MEMA	MEMB	WREG	IREG	VMAC	ACT
-9	rd	rd	-	-	-	-
-8	rd		sh	sh	mac	
-7		rd		set	mac	
-6			set			
-5	-	-	sh	setsh	mac	

Table 4 Time slots that are occupied by set register operations.

Set operations	-2	-3	-6	-7

To better understand the scheduling procedure, consider the task graph of 3x3 filter as depicted in Figure 16. Schedule bottom-up in the task graph, the first vertex is assigned at 0 and the parents are assigned based on the child vertex and dependencies. Table 3 presents a partial schedule start from MAC operation at -5. The scheduler visit the parent of MAC operation (-5), which is a set weight register operation (`set rw`) and schedule it at -6. This time slot of -6 is included in the table (Table 4) to prevent another set register occurs at the same time. A conflict occurs when the scheduler visit the other parent vertex, which is a set image register operation. It gives a time slot of -6 as depicted in Figure 16 (see dashed vertices). The scheduler finds that the time slot -6 has been occupied by another set register and therefore the set register operation is rescheduled to -7.

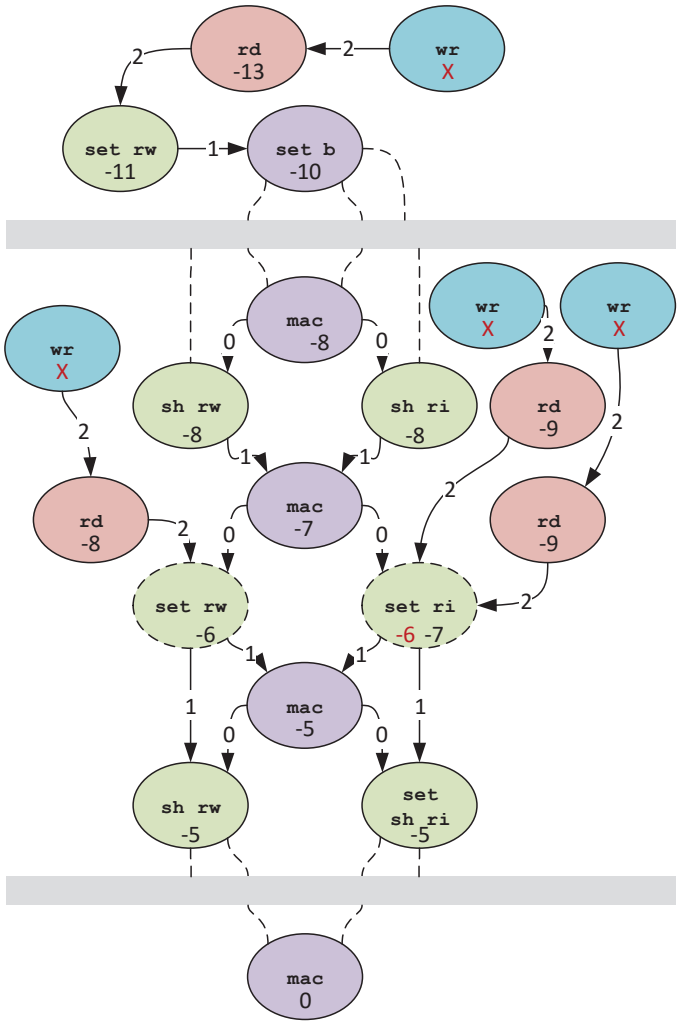


Figure 16 Scheduling of 3x3 filter example.

It is important to note that this instruction scheduling procedure ignores all write operations and activation operations. After this scheduling is done, the activation operations are inserted in **As Soon As Possible** (ASAP) scheme. For the write operations, they are separated into two parts. The first part is the initialization part where it contains the coefficients and pixels write

operations. The second part is the repetitive part where it contains only pixels write operations. The separation of pixel write operations is based on the repetitive pattern of the convolution. When the computation move across iteration (move across neighboring convolution), only some pixels are added to the local buffer and the rest of the pixels are reused. The new pixel write operations are placed in the repetitive because they are called in every iteration and the rest are placed in initialization. The pixel write operations in repetitive part are inserted into the instruction schedule in **As Late As Possible** (ALAP) scheme. This ensures that the write operations fill the schedule gap in the local memory stage. Finally, the time slot is shifted to remove negative time and provide proper cycle time. Figure 17 shows the repetitive part of the 3x3 convolution.

Cycle	MEMA	MEMB	WREG	IREG	VMAC	VACT
1	rd [b w0 w1 w2]					
2	rd [s(i+0) word0]	rd [s(i+0) word1]				
3		rd [s(i+0) word2]	set rw[b w0 w1 w2]			
4	wr [s(i+2) word0]		sh rw	set ri[0-1]	set rb	
5	rd [s(i+1) word0]	rd [s(i+1) word1]	sh rw	setsh ri[2]	mac [rw ri]	
6	rd [w3 w4 w5 w6]		sh rw	sh ri	mac [rw ri]	
7	wr [s(i+2) word1]	rd [s(i+1) word2]		set ri[0-1]	mac [rw ri]	act [s(i) word0]
8	wr [s(i+2) word2]		set rw[w3 w4 w5 w6]			act [s(i) word1]
9	rd [s(i+2) word0]	rd [s(i+2) word1]	sh rw	setsh ri[2]	mac [rw ri]	
10	rd [w7 w8 - -]	rd [s(i+2) word2]	sh rw	sh ri	mac [rw ri]	
11			sh rw	set ri[0-1]	mac [rw ri]	
12			set rw[w7 w8 - -]	setsh ri[2]	mac [rw ri]	
13			sh rw	sh ri	mac [rw ri]	
14					mac [rw ri]	

Figure 17 Repetitive part of 3x3. Write operations fill the empty time slots in MEMA using ALAP.

4.4. Local Buffer Allocation

The VLIW schedule produce by the scheduler does not have any physical address. All the local buffer operations are still labelled as it is, e.g. $rd\ [w0\ w1\ -\ -]$, which is read coefficients $w0$ and $w1$ from memory that are packed into one word. Another example is $wr\ [s(i+1)\ word3]$, which writes vector pixels in column 1 of iteration i; where one column may consist of several words, in this case write word number 3. These relative expressions must be translated according to addressing rules explained in section 3.2.

The allocation procedure starts from the initialization part where it mainly consists of write operations. Since these write operations are called only once, no circular addressing pattern is required and thus the address only need the `offset` field and left the `update` field empty. The circular addressing mode is also disabled to prevent the `update` field to increment. One write operation in initialization part is allocated with one word of memory space. The address is incremented according to the order if the instructions. Next, the allocation process advance to the repetitive part. In this step, the allocator selects which local buffer operations require a circular addressing pattern. Coefficients read in the repetitive part clearly does not require a pattern addressing because the coefficient writes are allocated once at the initialization part and those addresses do not change during the run-time. On the other hand, the pixel write operations in repetitive part are different across iterations. In the first iteration, it writes in an empty address space and in the next iteration it overwrites the pixel vectors that are not used for current convolution. The read pixel operations also perform in similar patterns. The read operations point at a certain address and in the next iteration these addresses change in a circular way. To assign the address of these read and write pixel operations, all `offset` fields are set with the first address that contain pixel data which is usually written in the initialization part. Next, the circular addressing pattern is produced by assigning proper value in the `update` field to point at the correct starting address in the first iteration. The `inc`, `toggle`, and `mode` field also set accordingly to give the pattern.

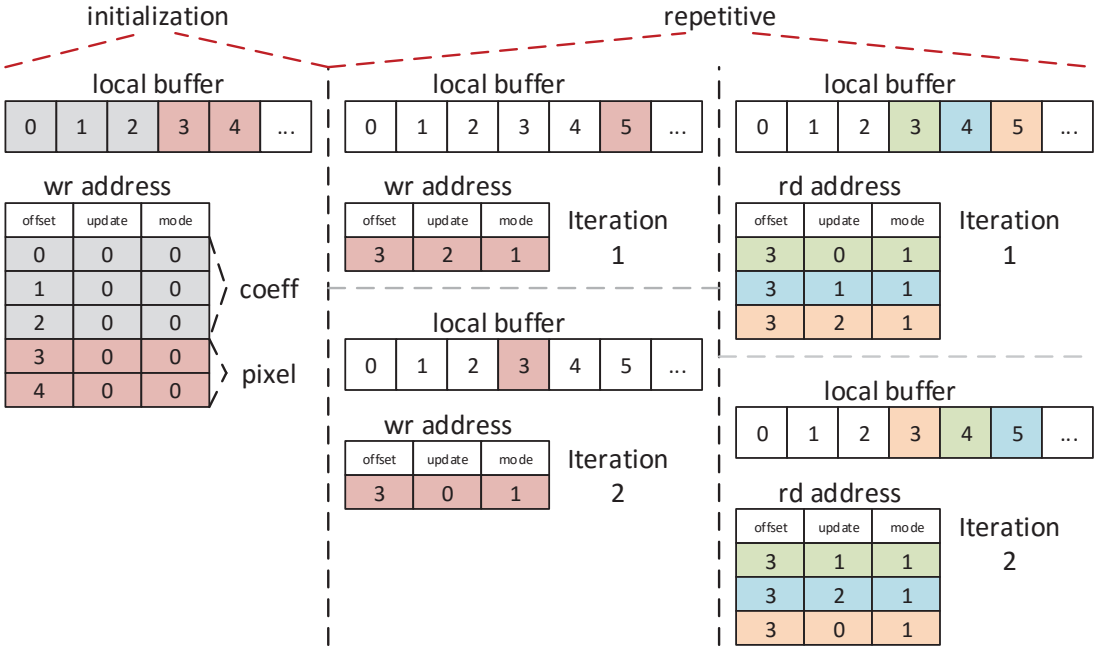


Figure 18 Circular address allocation.

To understand the procedure, consider a circular addressing example of 3x3 convolution as depicted in Figure 10 (section 3.2). Supposed that the coefficients take 3 entries and each pixel column takes 1 entry in the local buffer. During the allocation of initialization part, the coefficients are placed in memory address $0-2$ as shown in Figure 18 (left) with fix addressing. The address does not have any pattern and hence the pattern mode is disable (0). The first two pixel columns also belong in the initialization part because writing the first two columns do not appear as a repetitive pattern in the next iteration. These first two columns are also assigned with fixed addressing. In the repetitive part, there is one pixel write operation because in every iteration only the last column is added to the local buffer. This write operation require pattern because it overwrites the memory in a circular pattern. To generate the address, the `offset` field is assigned to the first address that contains pixel data which is 3 and the `update` field is assigned with 2 to ensure that the address point to empty space in the first iteration. To update the address, circular pattern mode is enabled (1) as depicted in the middle figure. By enabling the `mode` field, the `toggle` and `inc` numbers that are selected are 3 and 1 as depicted in Figure 19. The top part shows the first iteration and the bottom part shows the second iteration where the `update` field has been updated and the write operation point to different address (in a circular way). Similarly, this circular pattern also applied to the read operations. The right figure shows that the read operations have the same offset of 3. In the first iteration, the read requires column 0-1-2 for the convolution and hence the `update` fields are assigned with 0-1-2 to point at the proper address in the local buffer. The bottom figure shows the second iteration where the convolution window move and the read operations read column 1-2-3 in the local buffer. As explained previously, the new vector (column 3) is stored by write operations in the second iteration at the address of column 0. This is because the second convolution does not require column 0 anymore and it can be overwritten by new pixels. Therefore, the read address also change by updating the `update` fields to 1-2-0.

mode	toggle	inc
1	3	1
0	0	0

Figure 19 Select toggle and increment numbers based on mode.

After allocating physical addresses, the automatic code generator converts the schedule into executable binary form. The code generator provides an easy way to generate functionally correct code for the users. A Further comparison of code quality to manually written programs is conducted in section 6.3.2 and 7.2. The manually written programs have better quality compared to the automatically generated programs. In the next chapter, additional optimization steps are proposed to close the quality gap.

5. Code Optimization

Automatic code generation is essential to reduce the time that is required to map ConvNets to the accelerator. Section 4 presents an automatic code generator to map ConvNets on NVE. Nonetheless, there are many problems that prevent the automatic code generator to match the code quality of the manually written program. Code quality can easily become a performance bottleneck in a flexible architecture and well-designed code generator is required to maximize the performance of such design. This section presents strategies that are employed to optimize the automatic flow in the previous section, such as: changing the data layout of coefficients, software pipelining, and feature map fusion.

5.1. Coefficients Layout

Section 4.3 outlines the conflict resolution procedure for the register operation during instruction scheduling. The penalty for solving the conflict is an additional idle slot in the vector MAC pipeline as depicted in Figure 20 (inside the red dashed-line). This happens because there is not enough scheduling freedom to schedule the loads of weight and image register.

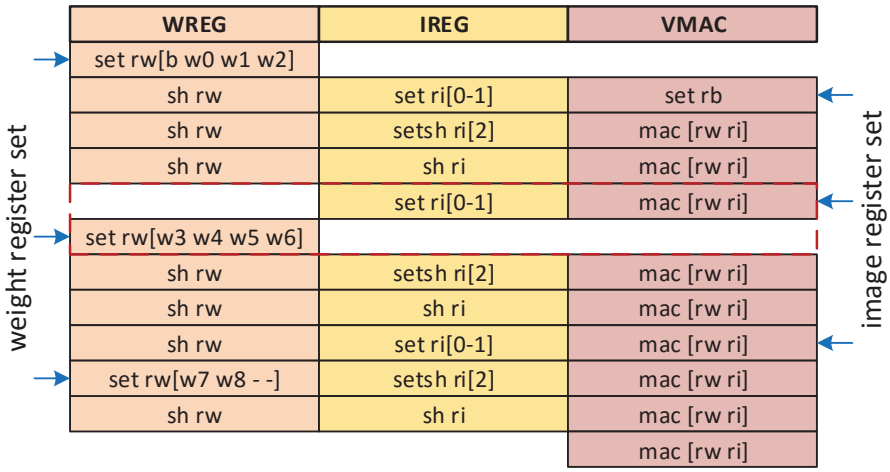


Figure 20 Instruction scheduling results. Blue arrows show the timing of register loads and red dashed-line shows the conflict resolution

The programmer removes this idle time slot by changing the data layout of the weight coefficients to give more scheduling freedom. Instead of always load 4 coefficients, the programmer decides to load less coefficients to prevent conflict with the image register. To adopt this solution, the automatic code generator must be able to find proper data layouts which prevent conflicts. The example of 3x3 filter in Figure 20 always assume weight register loads 4 coefficients ([b w0 w1 w2], [w3 w4 w5 w6], [w7 w8 - -]). The conflict is caused by the second weight register load ([w3 w4 w5 w6]) and the second image register load. To prevent this conflict, the first weight register load is packed with less coefficients ([b w0 w1 -]) as depicted Figure 21. This prevents the conflict which is caused by second weight and image register loads.

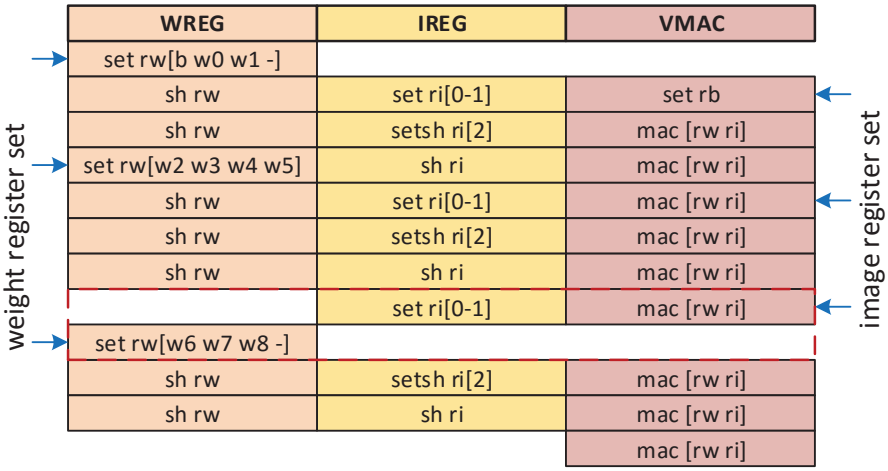


Figure 21 Repacked first weight register load. The conflict remains.

Changing one coefficients layout affects the layout of the following coefficients ([w2 w3 w4 w5], [w6 w7 w8 -]), which cause another conflict during the third register loads. The same procedure is repeated until no more conflict remain as shown in Figure 22. This repacking procedure successfully prevents conflicts between weight and image register. However, in case of 1x1 convolution, this procedure will not find any solution. It is because the image register loads occur in every cycle and there is no free slot to insert the weight register loads. The best option for such case is to load the weight register with maximum capacity.

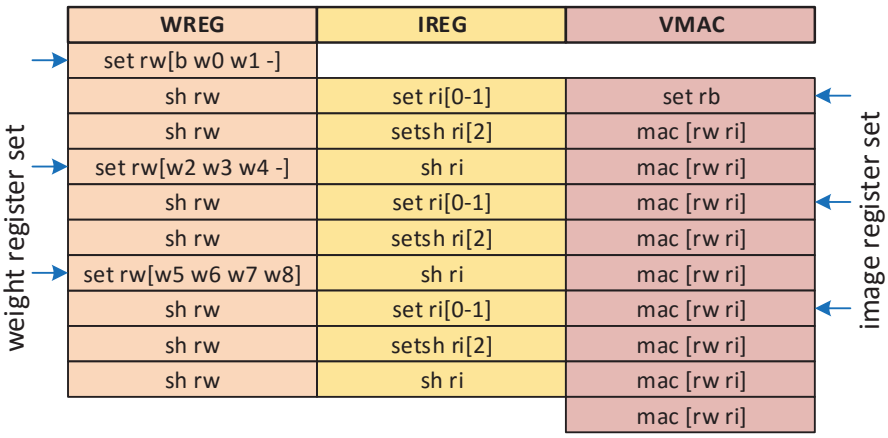


Figure 22 Repeat the repacking until it removes the conflicts.

Changing the data layout of weight register during instruction scheduling is not possible because it also changes the dependencies. Therefore, this procedure is implemented in the task graph construction as shown in Figure 23. The task graph construction search for the best data layout during the selection of weight register operations.

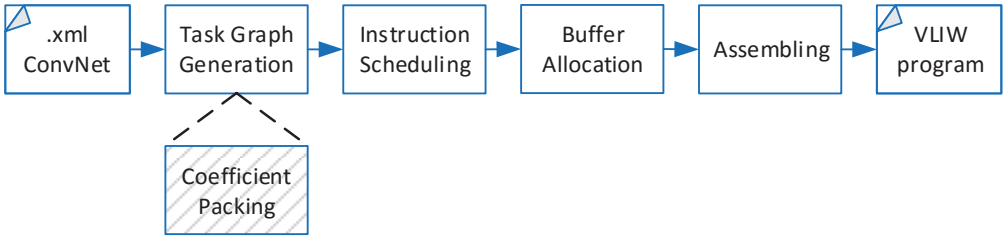


Figure 23 Coefficient packing is included in the task graph generation step.

5.2. Modulo Scheduling

Changing the data layout improve the utilization of the vector MAC. However, there are remaining unutilized slots in the repetitive part of the schedule as depicted in Figure 24. Since this schedule is executed repeatedly, this will leave many unutilized slots across iterations as pictured in Figure 25 (left). To reduce idle slots, the schedule can be partially merged across iterations to save execution time. The next iteration is started before the previous iteration finish as depicted in the right figure. This merging results in three distinct parts: prologue, steady-state, and epilogue. The prologue contains initial part and the epilogue contains the last part of the repetitive schedule. This two program parts are executed only once. The steady-state part contains overlap schedule of two different iterations and this part is executed repeatedly.

Cycle	MEMA	MEMB	WREG	IREG	VMAC	VACT
1	wr [s(i+2) word0]					
2	rd [b w0 w1 -]					
3	rd [s(i+0) word0]	rd [s(i+0) word1]				unutilized
4	wr [s(i+2) word1]	rd [s(i+0) word2]	set rw[b w0 w1 -]			
5	rd [w3 w4 w5 w6]		sh rw	set ri[0-1]	set rb	
6	rd [s(i+1) word0]	rd [s(i+1) word1]	sh rw	setsh ri[2]	mac [rw ri]	
7	wr [s(i+2) word2]	rd [s(i+1) word2]	set rw[w2 w3 w4 w5]	sh ri	mac [rw ri]	
8	rd [w7 w8 - -]		sh rw	set ri[0-1]	mac [rw ri]	act [s(i) word0]
9	rd [s(i+2) word0]	rd [s(i+2) word1]	sh rw	setsh ri[2]	mac [rw ri]	act [s(i) word1]
10		rd [s(i+2) word2]	set rw[w5 w6 w7 w8]	sh ri	mac [rw ri]	
11			sh rw	set ri[0-1]	mac [rw ri]	
12			sh rw	setsh ri[2]	mac [rw ri]	
13	unutilized		sh rw	sh ri	mac [rw ri]	
14					mac [rw ri]	

Figure 24 Unutilized part in 3x3 filter schedule.

To create such schedule, modulo scheduling is used. Modulo scheduling is implemented by finding the issue slot that requires the most time to determine the **Minimum Initiation Interval** (MII). The MII number is used to wrap around the schedule by rescheduling the instruction with modulo MII of its original time slot ($slot_{new} = slot_{old}$ $\boldsymbol{mod}$ MII). In the 3x3 filter example, weight register and vector MAC spend the most time (10 cycle) and therefor the original schedule is rescheduled with modulo 10. The wrap around code reduces the number of instructions and improves the utilization as depicted in Figure 26. This procedure is performed in the automatic flow after the instruction scheduling procedure as pictured in Figure 27.

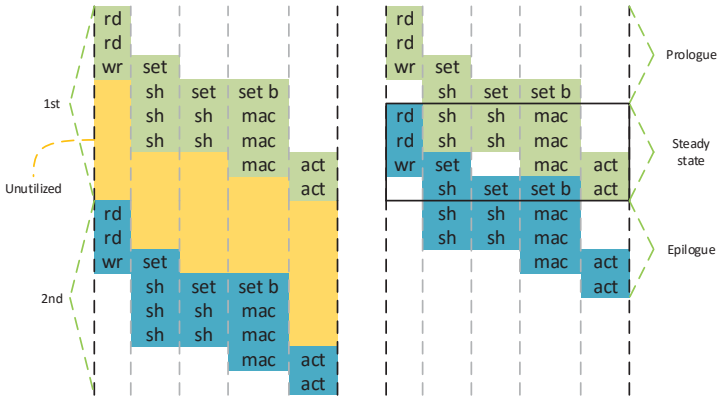


Figure 25 Steady state schedule execution pattern. The left figure shows iteration overhead across loop boundaries and the right figure shows the iterations have overlapping schedule.

Cycle	MEMA	MEMB	WREG	IREG	VMAC	VACT
1	wr [s(i+2) word0]		sh rw	set ri[0-1]	mac [rw ri]	
2	rd [b w0 w1 -]		sh rw	setsh ri[2]	mac [rw ri]	
3	rd [s(i+0) word0]	rd [s(i+0) word1]	sh rw	sh ri	mac [rw ri]	
4	wr [s(i+2) word1]	rd [s(i+0) word2]	set rw[b w0 w1 -]		mac [rw ri]	
5	rd [w3 w4 w5 w6]		sh rw	set ri[0-1]	set rb	
6	rd [s(i+1) word0]	rd [s(i+1) word1]	sh rw	setsh ri[2]	mac [rw ri]	
7	wr [s(i+2) word2]	rd [s(i+1) word2]	set rw[w2 w3 w4 w5]	sh ri	mac [rw ri]	
8	rd [w7 w8 - -]		sh rw	set ri[0-1]	mac [rw ri]	act [s(i) word0]
9	rd [s(i+2) word0]	rd [s(i+2) word1]	sh rw	setsh ri[2]	mac [rw ri]	act [s(i) word1]
10		rd [s(i+2) word2]	set rw[w5 w6 w7 w8]	sh ri	mac [rw ri]	

Figure 26 Modulo scheduling result of 3x3 filter example.

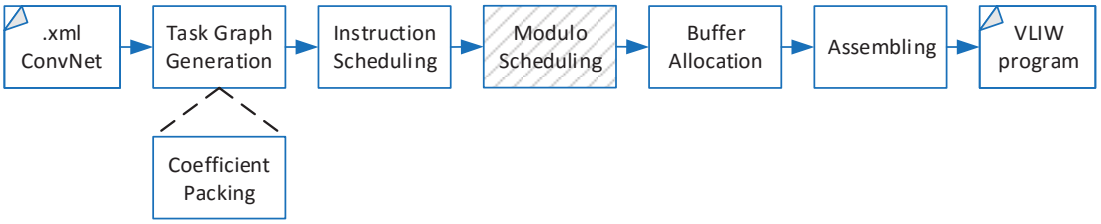


Figure 27 Modulo scheduling is done after instruction scheduling.

5.3. Feature Map Fusion

Feature maps in particular layer may share the same input images among each other. This inputs sharing improves locality if feature maps are combined into single execution. However, combining these individual programs is limited by the capacity of the instruction buffer. Several program combinations may be required to improve locality and conform the capacity constraint.

Consider i feature maps in a particular layer and each of these feature maps has a set of inputs I_x; for example, $I_2 = [0\ 2\ 3\ 4]$ means that feature maps 2 has input feature maps of 0, 2, 3, and 4. The number of input reuse of two combined programs is the intersection of the input sets. Because the combined programs reuse the inputs, redundant write operations are removed and

therefore may reduce the program size. The combined program size is less or equal to the sum of the two programs.

Algorithm 2 Greedy feature map fusion

input: list of individual feature map schedules L
output: list of combined schedules R

$f \leftarrow$ first schedule in L ◅ get first schedule
for all schedule v in list L
  if $s_v > s_f$ **then** ◅ compare size
    $f \leftarrow v$
remove f from L
$g \leftarrow f$ ◅ make a copy of f
while L is not empty **do** ◅ iterates until the list empty
  $\alpha \leftarrow \infty$ ◅ initial minimum difference
  $\beta \leftarrow 0$ ◅ initial maximum intersection
  for all schedule v in list L
    if $|I_f \Delta I_v| < \alpha$ **then** ◅ if the difference is smaller
      $\alpha \leftarrow |I_f \Delta I_v|$ ◅ get the number of different inputs
      $\beta \leftarrow |I_f \cap I_v|$ ◅ get the number of shared inputs
      $c \leftarrow v$ ◅ assign candidate
    else if $|I_f \Delta I_v| = \alpha$ **then** ◅ if the difference is equal
      if $|I_f \cap I_v| > \beta$ **then** ◅ find maximum intersection
        $\alpha \leftarrow |I_f \Delta I_v|$
        $\beta \leftarrow |I_f \cap I_v|$
        $c \leftarrow v$
  $g \leftarrow$ combination of g and c ◅ combine candidate to g
  $h \leftarrow g$ ◅ make a copy of g
  modulo scheduling h ◅ modulo schedule h
  if $s_h >$ `INSTRUCTION_BUFFER_SIZE` **then** ◅ if violates buffer limitation
    modulo scheduling f
    allocate data buffer space for f
    insert f to R ◅ accept f as solution
    $f \leftarrow$ first schedule in L ◅ find a new largest set
    for all schedule v in list L
      if $s_v > s_f$ **then**
        $f \leftarrow v$
    remove f from L
    $g \leftarrow f$ ◅ throw away solution g
  else ◅ if the buffer fits
    $f \leftarrow g$ ◅ copy the solution to f
    remove c from L ◅ remove the candidate from the list

To maximize the data reuse, one must combine two programs that have a maximum intersection of input sets. Consider input sets of two programs $I_1 = [0\ 1\ 2\ 3]$ and $I_2 = [0\ 2\ 3\ 4]$, these input sets have 3 inputs overlap and can be reused if these two programs are combined. However, finding maximum intersection alone does not guarantee an optimal solution. Supposed we have several input sets $I_1 = [0\ 1\ 2\ 3]$, $I_2 = [1\ 2\ 3\ 4]$, $I_3 = [0\ 2\ 3\ 4\ 5]$, and some others (unspecified). The combinations of $I_1 - I_2$, $I_1 - I_3$, and $I_2 - I_3$ have the same number of intersection. However, combining with I_3 give a larger program due to larger set and might limit the opportunity to combine with another program in the next combining step. Hence, combination

of $I_1 - I_2$ maximize locality and also give more freedom to combine with other programs. Despite of that, combining two different sets with more differences increase the size of input set, e.g. combining I_1 and I_3 produce a program with input set $[0\ 1\ 2\ 3\ 4\ 5]$ (larger coverage), and hence increase the opportunity to find more intersection in the next combining step. It is very hard to decide the combination of these programs.

The proposed fusion procedure use a greedy approach to combine these programs. The greedy approach does not guarantee an optimal solution, nevertheless it produce better solution than without combining any programs. Algorithm 2 lists the fusion procedure, L holds the individual schedules and R is the list of combined programs. First, it search for the largest program which is usually has the largest number of input feature maps (largest set). Largest program is selected because it is more likely to completely cover another program (smaller set might be subset of larger set). This largest program is merged with other program that has maximum intersection (β) and minimum difference (α) of input feature maps. This merging is repeated until the total program size reach the maximum limit of the instruction buffer. If fusion procedure reach the limit, the fused program (f) is stored as a valid solution and a new fusion procedure is started from new largest program.

Combining individual programs should be done on the program that has not been scheduled with modulo scheduling, otherwise it will produce a faulty program. This is because modulo scheduling has modified the schedule by merging across iterations. Instead of merging across iteration, the fusion procedure merge two different programs; it started the next program before the previous one finish. Later, modulo scheduling is performed to check the capacity constraint, whereas a copy of original schedule is saved for further combining if there is enough buffer space. The fused program that reach the capacity constraint (or no candidates left) is stored after buffer allocation. The fusion procedure is performed after instruction scheduling as shown in Figure 28, both modulo scheduling and buffer allocation are incorporated in the procedure.

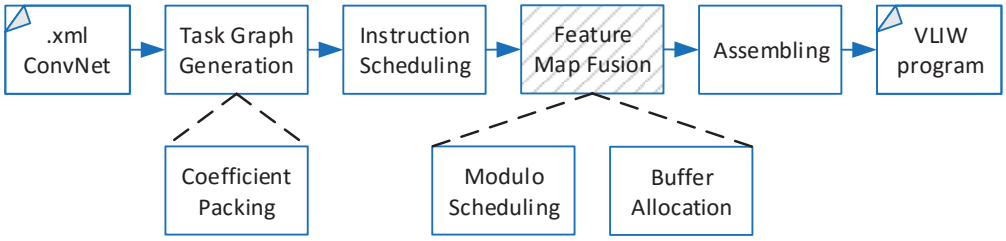


Figure 28 Feature map fusion is done after scheduling. Modulo scheduling and buffer allocation are done during the feature map fusion.

6. Evaluation Methods

To better understand the performance of NVE and how well the code generator works, several evaluations are conducted. To perform these evaluations, benchmark applications are introduced. These benchmarks are used to measure the performance of NVE and a baseline platform for comparison. The same benchmarks are also used to measure the code quality of the manually written code and automatic code generator. This section discusses in details about the benchmark applications, the platforms, and the evaluation methods.

6.1. Benchmark

To evaluate the platforms and code generators, three applications are used, namely: 3x3 filter, face detection, and speed sign detection. The 3x3 filter is a simple application, whereas the face and speed sign detection are ConvNet applications. Appendix A shows the network description (number of layers, number of feature maps, sizes, connections, etc.) of face and speed sign detection in XML format and Table 1 in section 2.1.4 presents the computational and data transfer requirements of both applications. The speed sign detection requires much more computational resources and generates more data traffic compared to face detection. This is because the face detection is a simple classification problem of classifying face and non-face, whereas the speed sign detection has eight different classes.

6.2. Platforms

To understand the performance of NVE, an ARM-based processor is used as a baseline platform. This platform is chosen because it is used in many mobile devices and represents the performance of currently available embedded computing solution in the market. This platform is compared with a demonstrator of NVE to see the difference in performance. The following sections discuss about both platforms in details.

6.2.1. ARM Baseline

The ARM architecture has many variants for a wide range of embedded applications. In this experiment, ARM Cortex-A9 is used as a baseline. The architecture has a 64-bit wide NEON SIMD coprocessor. Applications running on this architecture can be accelerated by exploiting sub-word parallelism of NEON as pictured in Figure 29; the NEON coprocessor can perform up to four 16-bit operations or eight 8-bit operations per cycle. The NEON optimized code can either be generated by the compiler or manually written using NEON intrinsic or assembly code.

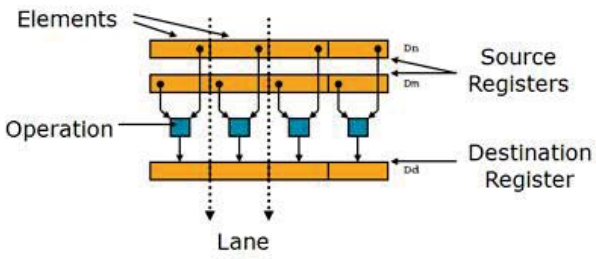


Figure 29 NEON coprocessor

6.2.2. NVE Demonstrator

To demonstrate the performance of the accelerator in a real setup, it is implemented on the Zynq SoC [10] which has a dual-core ARM Cortex-A9 as a host processor and AXI standard interfaces for interconnection to the FPGA part as depicted in Figure 30. Two different AXI standards are used in this setup, namely AXI-Lite and AXI-Stream interface. The AXI-Lite interface is used for accelerator control and AXI-Stream interface is used for data transfer, which consists of instructions, coefficients, and pixels. The AXI-Stream interface is connected to a dedicated *Direct Memory Access* (DMA) which can perform scatter-gather transfer. The scatter-gather mechanism provides a sequence of individual transfers without the need of new commands in between using a sequence of *Buffer Descriptor* (BD). The BDs are arranged as a linked-list and each of them contains information of single transfer such as starting address, size, and pointer to the next BD as depicted in Figure 31.

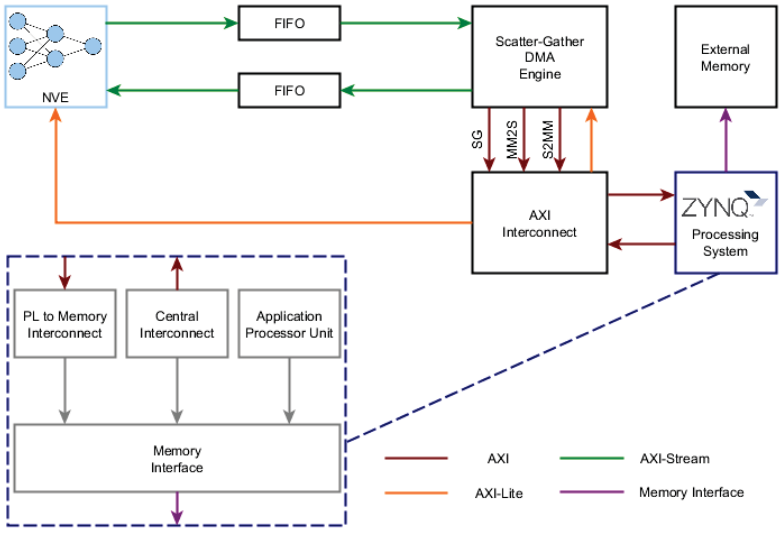


Figure 30 NVE setup on Zynq SoC.

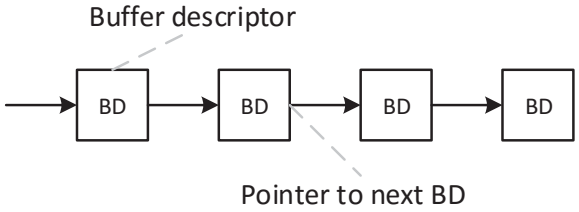


Figure 31 Chain of buffer descriptor (BD).

6.3. Evaluation

The following section discusses how the platforms and the programming approaches are evaluated. For platform evaluation, the benchmark applications are mapped to measure the time performance. As for the programming approaches, the benchmark applications are written into NVE assembly programs both manually and automatically.

6.3.1. Platform Performance

Different scenarios are used to evaluate the performance of ARM baseline and NVE. The performance of ARM baseline is evaluated by porting the face and speed sign detection applications. For the NVE, a simple 3x3 filter is run on the platform. These benchmarks are accelerated using the specific HW features of the platforms to measure the peak time performance that can be achieved. The performances of these porting are plotted into the roofline model of the platforms to see how far the mappings from the theoretical limit of the platform.

6.3.1.1. ARM Baseline

To accelerate the benchmark applications on ARM baseline, loop transformation such as loop tiling is used to improve data locality. The impact of loop tiling differs across layers due to different kernel size and subsample. Thus, several tiling sizes are explored to find the best configuration for each layer.

Other than loop transformation, parallelization is also done to improve the baseline performance. There two approaches that are implemented to parallelize the applications. First, by using the compiler option to automatically vectorized and optimized the program and second by manually re-write the critical parts of the code with NEON intrinsic. The first approach is very straight forward and there are no special steps other than specifying the optimization flags. As for manual optimization, the code is carefully re-written according to instruction latencies as listed in [11]. The order of instructions is very important because it may cause pipeline stall due to data dependencies.

To compute using NEON coprocessor, each of the inputs must be loaded into special vector registers using vector load instruction (`VLD`). A sophisticated features such as interleaved vector load (`VLD`*n*) that can load interleaved of *n* to *n* different registers is also available. This instruction is useful for some layers that has stride-2 accesses where the convolution layers and pooling layers are combined. The ConvNets operations mainly consist of multiply-accumulate operations which take 8-bit pixels and 16-bit weights as inputs. Thus, 4 sub-word parallel operations can be done

using the VMLAL instruction (vector multiply accumulate) which takes 16-bit inputs and produce 32-bit results on NEON coprocessor.

The order of instructions is important to prevent stall due to data dependencies. Consider pseudo codes in Listing 2, the left pseudo code shows an example of 1D convolution. The load of variable A and B is required before MAC operation on C is performed. If the load of B requires a couple of cycles to write back the result to the register, the MAC of C will stall. To prevent this, the loop of n is tiled (with tile size of 3 in the example) and the iteration within the tile is done inside the convolution loop of i. The tile iteration is later unrolled and the operations are rearranged in such a way to hide the latencies as depicted in the right pseudo code. A sequence of independent loads is performed which are not dependent on each other. Later, a sequence of MACs is performed; if the distance between the first MAC and the data load is long enough, it will reduce the stall. These tiling and unrolling strategies are limited by the number of available registers because the unrolled load operations require registers to store the results. If the tiling size is too big, it will cause register spilling which in turn raises the cycle count again.

Listing 2 1D convolution example (left). Loop tiling, interchange, unrolling, and operations rearrangement on the 1D convolution.

```
for(n=0;n<NN;n++)
{
  load C[n]
  for(i=0;i<II;i++)//convolution
  {
    load A[i]
    load B[n+i]
    C[n] += A[i] * B[n+i]
  }
  store C[n]
}
```

```
for(n=0;n<NN/3;n++)
{
  load C[3*n]
  load C[3*n+1]
  load C[3*n+2]
  for(i=0;i<II;i++)//convolution
  {
    load A[i]
    load B[3*n+i]
    load B[3*n+1+i]
    load B[3*n+2+i]
    C[3*n] += A[i] * B[3*n+i]
    C[3*n] += A[i] * B[3*n+1+i]
    C[3*n] += A[i] * B[3*n+2+i]
  }
  store C[3*n]
  store C[3*n+1]
  store C[3*n+2]
}
```

6.3.1.2. NVE Demonstrator

To understand the performance of NVE, a 3x3 filter application is run on the platform. To run the application, the NVE is initialized with settings such as the number of instructions, the boundary of repetitive program, and the number of iterations. Next, DMA transfers the instructions and pixels to the accelerator. The NVE calculates the 3x3 filter and the DMA transfer the output back to main memory. To perform this execution flow, a dedicated software is developed and run on the host processor. This software responsible for initializing the control registers of NVE, configuring the DMA transfers, and measuring the execution time.

The 3x3 filter is optimized for NVE and can achieve top performance. However, different data transfer configuration may yield different overall time performance. The experiments on NVE are repeated for several data transfer configuration to see the impact of transfer size to the time performance.

6.3.2. Code Quality

The code quality between manual and automatic programming is compared using the benchmark applications. The NVE versions of the benchmark applications are manually written by expert programmers. The expert programmers efficiently map the benchmark applications by performing software pipelining technique. They also take the data layout into account to prevent register load conflicts. An automatic programming versions also generated by the code generator using the XML descriptor as presented in Appendix A. There are several versions of automatic programming that are generated. The first version is the automatic programming without any optimization steps. In the second one, coefficient packing is added and in the third and fourth one modulo scheduling and feature map fusion are added respectively. The different versions are generated to reflect the impact of each optimization step on the code quality.

7. Evaluation Results

7.1. Platform Performance

As stated in section 6.3, a series of experiments are conducted to evaluate the platform performance. Face and speed sign detection application are efficiently mapped to ARM baseline and NVE platform is evaluated using a 3x3 filter application with different data transfer scenario. The following sections discuss about the results of the experiments in details.

7.1.1. ARM Baseline

The face detection and speed sign detection are ported on ARM platform to show the performance of running ConvNets applications. NEON coprocessor is used to accelerate the applications. Two approaches are taken in this optimization namely automatic optimization through the compiler and manual optimization using NEON intrinsic. Figure 32 shows the comparison between automatic vectorization using the compiler and manual optimization using NEON intrinsic on both face and speed sign detection. It shows that the manual optimizations have better time performance compared to automatic vectorization with –O3 compiler flag. The face detection achieves 3.3 speed up and the speed sign detection achieves 4.4 speed up.

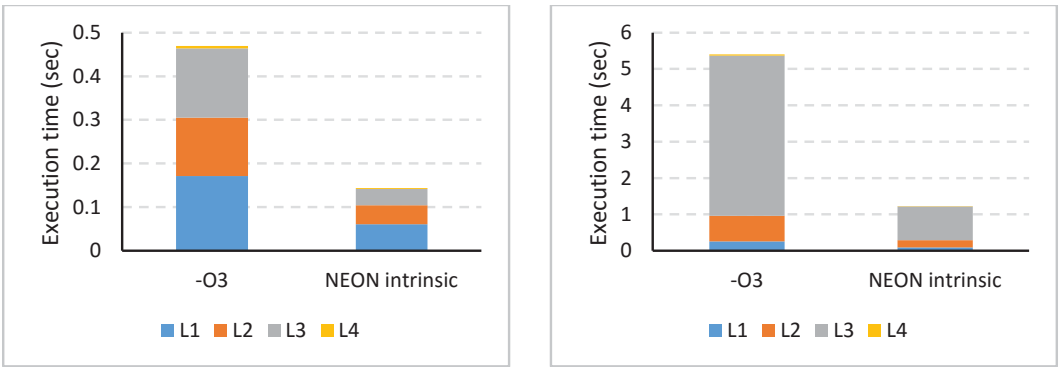


Figure 32 Face detection (left) and speed sign detection (right) applications performance on ARM Cortex-A9 with NEON coprocessor.

To better understand these results, a roofline model of the platform is created. Figure 33 shows the roofline of the ARM Cortex-A9 with a SIMD compute bound and a scalar compute bound. A memory bound is also plotted in this graph. The memory bound is obtained from the stream benchmark that measure the external memory bandwidth. The figure shows the mapping of individual layers in face and speed sign detection. It also shows how far these mappings from the theoretical peak performance. The computational intensity is counted based on the total number of operations and total number of bytes required by the algorithms. The total number of operations only take multiply and add operations into account. The data transfer only count the number of required input pixels and required output pixels which means the pixel transfers are counted once. The attainable operations per second of each application mappings are obtained from the total number of required multiply and add operations divided by the total run-time of the mappings.

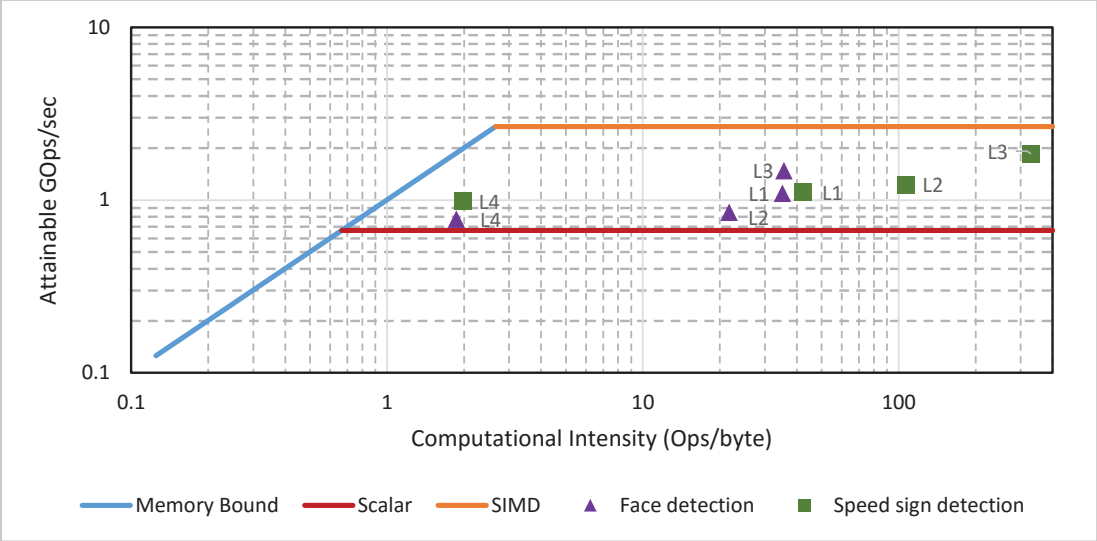


Figure 33 Roofline model of ARM Cortex-A9 + NEON coprocessor.

The best mapping on NEON coprocessor is L3 speed sign detection, which achieves 1.8 GOps/sec. However, it has a significant gap to the theoretical limit of this platform which is around 2.67 GOps/sec. This is even worse for the rest of the mappings as their performance 2 or 3 times less than the peak performance. One explanation for the performance gap is the ratio between effective computation instruction and the total instructions as depicted in Figure 34. The amount of instructions for the computation is half of the total instruction. This becomes even worse since the data dependencies may stall the pipeline, e.g the `vmovl.u8` requires data that are produced by `vld2.8`. Although for different layers the instruction sequence is different, they suffer from the same problem.

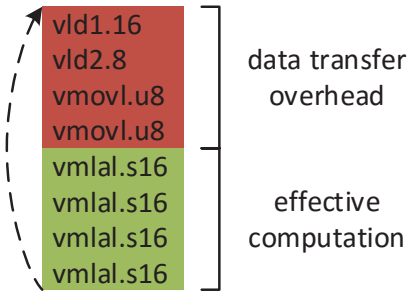


Figure 34 Instruction sequence in convolution loop.

7.1.2. NVE Demonstrator

With current design which has 16 MACs, the NVE has a peak performance of 3.2 GOps/sec at 100 MHz. However, this is hard to achieve when we take the data transfer into account. As explained in section 2.1.4, data movement can easily become a bottleneck for ConvNets implementation. Experiments on how the data transfer affects the performance of NVE are conducted on the demonstrator. An example 3x3 filter application is executed on the platform with different data transfer configurations. The 3x3 filter application consumes one pixel column in every iteration which is transferred from external memory to the demonstrator by the DMA. The DMA accommodate this data consumption pattern using the Scatter-Gather features where it

creates linked-list of descriptors. Each descriptor describe a single transfer of one pixel column. As depicted in Figure 35, this configuration does not perform well (the bottom marker) because most of the time is spent on waiting for the data. Rapidly initiating small data transfer in DMA is costly. The ideal data transfer throughput is shown by the top marker where the pixels are arranged continuously in the memory and the DMA transfer these pixel in single initiation. The demonstrator nearly reach the top performance in this configuration although such case is not a realistic data consumption pattern. It is clear that small data transfers have overhead, so large continuous transfers are preferable although the actual pixels are not contiguous.

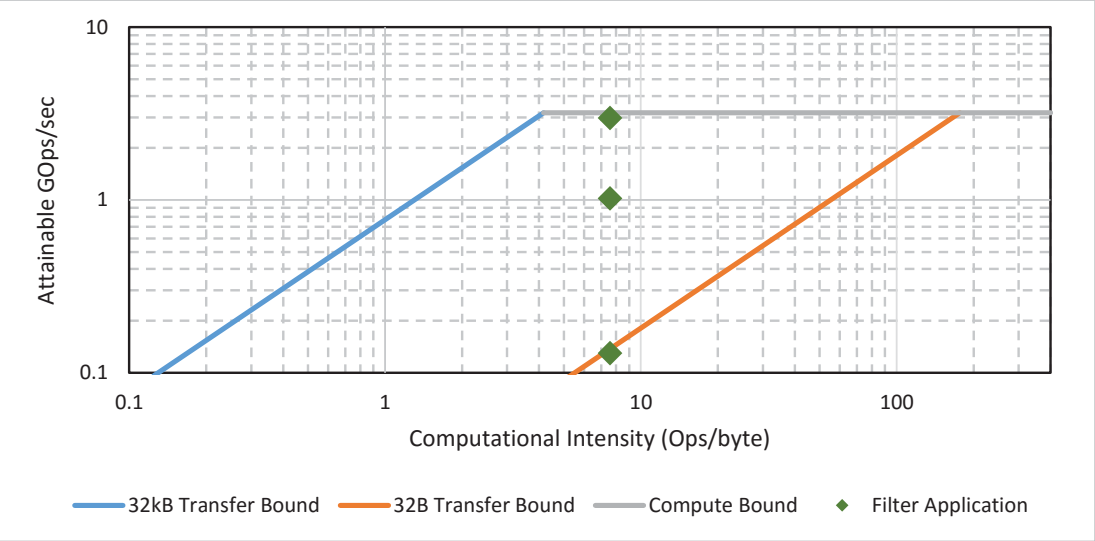


Figure 35 Roofline model of NVE.

One workaround that has been tried in this experiment is reordering the pixel positions in memory. Although the actual data consumption pattern of ConvNets is small pixel vectors, one can reorder those vectors into contiguous pixels using the host processor and then transfer them with DMA in a single continuous transfer. This workaround is shown by the middle marker, it has a lot of improvement compared to the original small data transfer. Nevertheless, this solution is far from the peak performance.

7.2. Code Quality

Generating code with an automatic code generator is fast and easy to users. More importantly, the code generator should generate functionally correct and good quality code. The quality of the code can be expressed as number of computations (in this particular case MAC) per instruction. This represent the number of useful operations executed per cycle. Low quality code may have less number of MACs per instructions due to stalls in the MAC pipeline or loop overhead (additional code to start and finish loop). To evaluate the quality of the automatic code generator, comparisons to manually written code by experts are presented in the following sections.

7.2.1. Automatic without Optimization

In this section, the automatic code generation result without any optimization is evaluated. The first and the second bar in Figure 36 presents the comparison of HW utilization of manually written code and automatically generated code respectively. The graph shows that the automatic

generated code achieves 14% less HW utilization w.r.t. the manual code. This number is acceptable considering the effort of writing manual code, which takes hours per layer even for expert programmers, and automatic code generation is done in seconds. To get better results, further optimization is performed. One obvious step is by removing the additional idle cycles that are inserted during scheduling due to set register conflicts. This optimization step is discuss in section 5.1 by changing the layout of coefficient data in the memory to give more scheduling freedom.

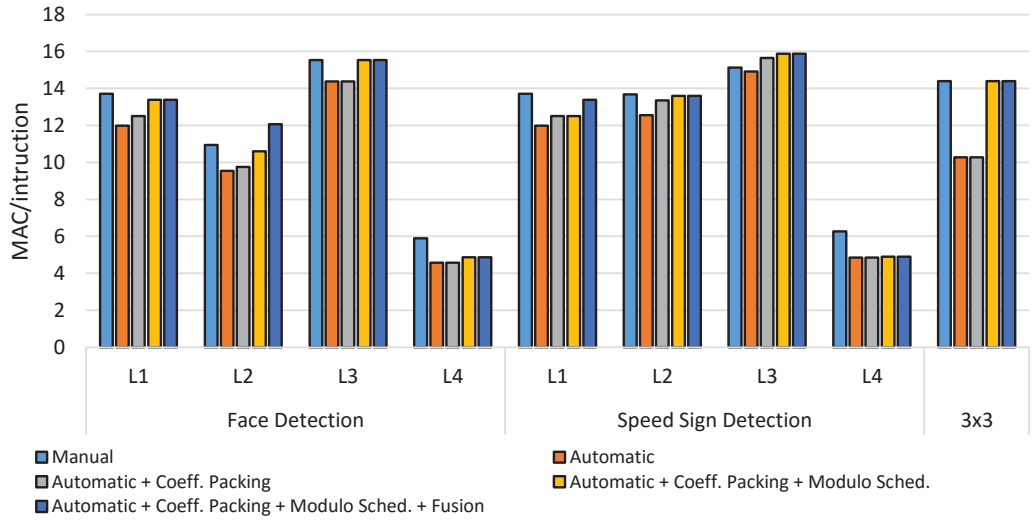


Figure 36 HW utilization comparison between manual programming and several versions of automatic programming.

7.2.2. Coefficients Layout

By changing the data layout of coefficients in the memory, the HW utilization of automatically generated code improved on average of 2% w.r.t the automatic programming without optimization. As shown by the third bar in Figure 36, the performance of L1-L2 face detection and L1-L2-L3 speed sign detection are improving. The L3-L4 face detection and L4 speed sign detection do not improve because there are no conflicts and therefore the coefficient packing strategy does not improve the results. Sometimes, as in case of 3x3 filter, the gains of removing stalls in the MAC stage are canceled by earlier write operations to the local buffer. If one compares Figure 17 and Figure 24 (before and after packing), this canceling effect is visualized.

Changing the data layout alone is not sufficient to improve the HW utilization. Another optimization opportunity is by removing the unutilized HW across loop iterations. This optimization is explained in 5.2, where modulo scheduling is used.

7.2.3. Modulo Scheduling

Modulo scheduling gives an additional improvement of 7% on average to the HW utilization of automatic code generation with coefficient packing. The improvement is quite significant compared to the previous optimization, although the coefficient packing also contributes to the quality of modulo scheduling. In many layers, the automatic code generator achieves similar HW utilization as the manual programs. Sometimes better performance is achieved as demonstrated in L3 of speed sign detection, although this is mainly caused by suboptimal manual code. However if one compares another metric such as data transfers, there is room for improvements. For

example in L1, the automatically generated code requires 3x the amount of data transfer w.r.t. to the manual programs as presented in Figure 37. This is solved by the feature map fusion approach which is presented in section 5.3.

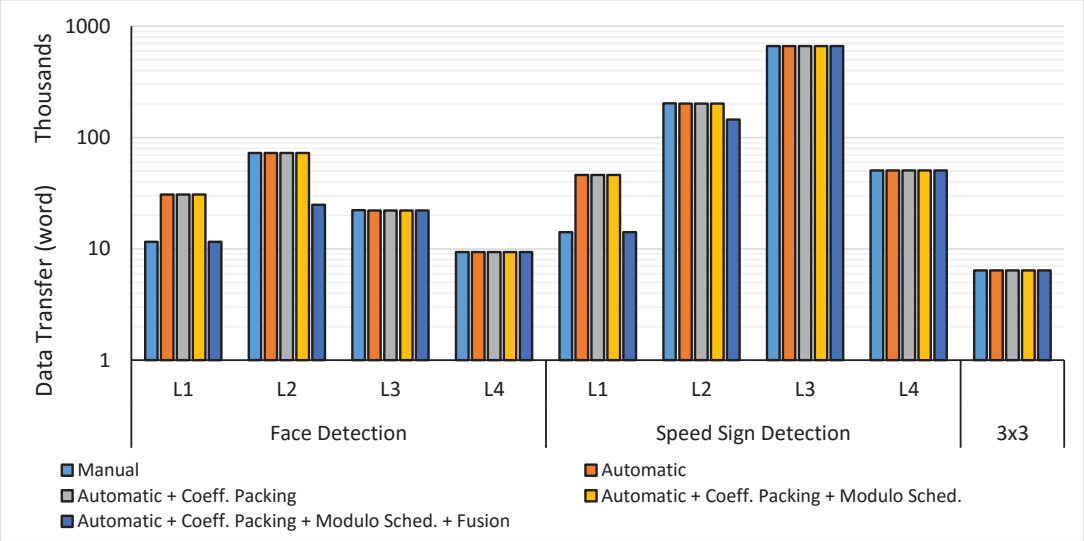


Figure 37 Data transfer comparison between manual programming and several versions of automatic programming.

7.2.4. Feature Map Fusion

The impact of feature map fusion is a possible reduction of data transfer, e.g. in L1 it reduced up to 70% (Figure 37, fourth bar) which is similar to the manual implementation (Figure 37, first bar). The data transfers also reduce in L2 of both applications by up to 65%, because multiple feature maps are combined by the automatic code generator, whereas the expert programmer does not combine the feature maps. Combining feature maps in L2 is difficult and error prone for the programmer. Another good aspect of feature map fusion is that it improves HW utilization by removing redundant write operations in the schedule. As a result, the number of instructions are reduced as shown in L2 face detection and L1 speed sign detection (Figure 36, fourth bar).

As one can observe from Figure 37, feature map fusion does not work for all layers. For example in L3 speed sign detection, the feature maps require large number of instructions which prevent the fusion due to instruction buffer limitation. In other layer such as L3 face detection, the feature maps do not share input and thus there is no opportunity for data reuse.

7.2.5. Possible Improvements

As illustrated in Figure 36, there are some workloads that do not achieve same HW utilization as the manually written programs. For example in L4 of both face and speed sign detection, the HW utilization of automatic code generator is 20% less than the manual one. In this layer, the convolution size is 1x1 and there are always conflict among set register operations. The number of register load operations are also larger compared to computation and hence the NVE spend more time fetching data from local buffer. Interestingly, the programmer decided to use half of the vector size to compute the convolution window in L4. The pixel vector is calculated using the bottom vector (`ri[8-15]`) which is connected to port B and this removes the pixel read pressure from port A, which creates time slots for set weight register operations. This gives more scheduling freedom and hence prevent stall insertions due to conflicts.

Another improvement opportunity is in L1, where three consecutive register loads occur at the beginning of the program which requires stall insertion across iterations. The expert programmers load the first set of coefficients at the bottom of the previous iteration, which removes the stall. Another similar problem is in feature map fusion when the stall also occurs across combined programs. The solution is by reconstructing the task graph of combined programs (after feature map fusion), and redo the entire code generation flow. This requires extra time to generate the code and the improvements are small, in the range of 2%.

The automatic code generator achieves similar performance to expert programmers. In best case, it achieves 10% better HW utilization and in worst case it achieves 20% less HW utilization. The data locality of automatically generated code is always better or at least equal w.r.t. the manually written code. It reduces up to 65% data transfer compared to manually written code. Another important aspect of the presented gain in HW utilization is the size of the workload. To understand the performance according to the workload of individual layers, Table 5 presents the HW utilization of face and speed sign detection where the numbers are weighted with the workload. It shows that the automatic code generator achieves better utilization because it produces better code quality for the layers that dominate the workload, e.g. L2 and L3.

Table 5 HW utilization of face and speed sign detection normalized with the workload of individual layers.

Recognition Task		Workload (%)	HW utilization (MAC/instruction)		Total HW utilization (MAC/instruction)	
			Manual	Automatic	Manual	Automatic
Face detection	L1	42	13.71	13.39	13.6	13.7
	L2	23	10.95	12.06		
	L3	34	15.53	15.53		
	L4	1	5.89	4.87		
Speed sign detection	L1	4.6	13.71	13.39	14.86	15.45
	L2	11.7	13.67	13.6		
	L3	83.3	15.13	15.88		
	L4	0.4	6.26	4.9		

8. Related Work

Prior works on Neuro Vector Engine are presented in [4] [5] [6] [7]. The works present the development of a dedicated accelerator for ConvNets which address the parallelism opportunity and data transfer problem. They present a strategy to lower the data transfer with layer fusion technique. They also present the trade-off in designing the architecture such as arithmetic precision and memory configuration. The proposals aim for efficient and low power application and yet provide enough flexibility for different network configurations. These works are the basis of the thesis project.

Other proposals such as [12] [13] [14] [15] present dedicated accelerators for ConvNets applications which aim for a low power mobile applications. These accelerators achieve high compute performance per watt compared to general purpose CPU and GPU. However, these proposals solely focus on the computation efficiency and ignore the fact that the accelerator requires a certain degree of flexibility to efficiently cover different network parameters.

Collaboration proposals from Inria (France) and ICT (CAS, China) in [16] and [17] proposed programmable architectures that have specialized control instructions. They mention support for automatically generating program for their architecture. Additionally, the work of [17] also support multi-core application mapping. Their recent work of [18] has a different approach; they introduce a hierarchical programming model with a hierarchical finite state machine (HFSM) to program the core, although there are no indication about automatic programming support in this recent work.

Proposal of [14] present a ConvNets accelerator with grid arrangement of PE. The proposal does not introduce any programming flexibility. In their later works of Neuflow architecture that is proposed in [19] and [20], they introduce an automatic tool for programming. Their architecture is programmed in a data flow model. Their compiler convert the algorithm description in flow-graph and convert it into an executable program.

Another work of [21] present a programmable architecture using **Application Program Interface** (API) without any automatic programming support, although in their later work of [22] a compiler is introduced. Their compiler use a high level abstraction which later converted into an executable program.

Despite the existence of automatic programming, all these works lack on details of how their automatic code generator works. In addition, there is no indication on how well their generated code performs compared to manually written code by experts.

An automatic code generation flow for NVE is presented in this thesis report. It uses scheduling heuristic such as bottom-up, BFS, ASAP, and ALAP to compute the efficient schedule. Similar results could be achieved with list scheduling as explained in [23]. A modular compiler infrastructure such as LLVM [24] is available and can be used to create a custom code generation flow. However, the complete compilation flow of LLVM offers so many steps and only four steps could be used in this particular case.

To optimize the code, software pipelining technique such as modulo scheduling which is reported in [25] is used. Modulo scheduling creates code overlaps across loop iterations and reduce the number of instructions. It ensures that there are no violation on intra- or inter-iteration dependence and no resource conflicts.

The work of [26] presents inter-tile optimization technique which reduce the memory access. This approach is quite different from the feature map fusion technique which also aim at reducing memory access. Inter-tile optimization technique is only possible on fully connected

layers if it apply to reduce memory access across feature maps. As for sparse layer connectivity, a search procedure is required such as the feature map fusion technique.

9. Conclusion and Future Work

9.1. Conclusion

Huge compute power requirements of ConvNets become a problem especially for general purpose embedded platform. With increasing popularity of ConvNets, it motivates the community to enable the algorithm on resource-constraint mobile devices. The community proposed customized hardware acceleration for ConvNets which has significantly improved their computational efficiency. The last bottleneck for market adoption of such accelerators is the complexity of programming these accelerators. Supporting flexibility and ease of programming for a custom accelerator are a big challenge.

This work present a code generation flow, from a high level XML ConvNets description to an optimized accelerator VLIW program. From a high level description that is provided by the user, the automatic code generation flow create graph representations of ConvNets workloads. Scheduling procedure is performed on the graphs to create VLIW program for the accelerator. This gives functionally correct code from an easy DSL. Nevertheless, additional optimization steps are required to maximize performance.

A coefficients layout strategy is employed to reduce the amount of conflicts during scheduling which also reduce the amount of additional stalls in the code. By using advanced techniques such as Modulo Scheduling, the hardware utilization is improved. Due to our feature-map combining technique our generated code reduce up to 65% of the data transfer, which answer the challenging data transfer requirements. The automatic code generator generates code that achieves a hardware utilization up to 10% better w.r.t. the manually written code by an expert.

The introduction of the code generation flow enables users to abstract from the accelerator architecture to only a network specification. This abstraction reduced the workload involved when programming a ConvNets vision application from days to minutes. The removal of this last productivity bottleneck enables the adoption of ConvNets in the next-generation mobile devices and bring *smart* features like real-time machine vision and speech recognition to our portable companions.

9.2. Future Work

Overall, the code generator achieves better code quality compared to the expert programmers. Nevertheless, there are interesting options to improve the current design of the code generator. The following sections describe several future directions to improve the code generator.

9.2.1. Aggressive Optimization

In some cases, the code quality of manual programming is better than the automatic code generation as demonstrated in section 7.2. The programmer have several ways to optimize the program such as packing coefficients across iterations and use only half the vector size. These procedures are not available in the automatic code generator and therefore can be added as a more aggressive optimization flow.

The code generator can decide to use half vector size when there are no scheduling freedom (even if the coefficients layout is changed) for register loads such as 1x1 convolution case.

For the coefficients layout, the code generator can decide to pack coefficients across iterations or across program (in case of feature map fusion) if there are any idle time slot in the vector MAC across the loop boundaries or across the feature map boundaries. To perform this, the code generator has to reconstruct the task graph from current program and redo the entire code generation flow.

There is a remaining opportunity to improve data locality because the code generator does not evaluate tiling in a single feature map. In the scenario where there are no further fusion candidates, tiling could be used to compute wider vector with the same HW resources. The code is replicated to compute wider neighboring convolution as long as it fits into the instruction buffer. This will improve the locality within feature maps and also maximize the instruction buffer usage.

9.2.2. Support for Different Local Data Buffer Configuration

The current design of the NVE may change in the future according to the needs of the users. For example, the current design supports true dual-ported memory for the local buffer which is usually available on FPGA, whereas the users might want a different memory configuration for ASIC.

For example, designer can chose a simple dual-ported memory for the local buffer. This introduces a different constraint in the scheduling where the architecture have limited option for reading and writing the data. All register loads will require access to the same ports which increase the number of conflicts.

Another possible changes in the memory is using multiple banks for the local buffer. Supposed that the local buffer use two banks with simple dual-port, the architecture will support two write and two read operations; it is similar to the current design. However, the ports are connected to two different banks which cannot be accessed from one to another. This raise constraints on how to allocate the memory and also how to properly schedule the access. If a certain data is written into one bank, it cannot be read from another bank.

A support for wider port also possible which reduces the loading time to the registers (especially image register with subsampling) due to larger bandwidth. However in a certain case, it may load too many data which are not necessary. A change in the registers maybe required to fit the bandwidth of the wider port. For example the weight register can load more coefficients at once which gives freedom to schedule the operations.

A designer can chose a certain memory configuration that fit their performance requirements and cost. An ideal compiler would have an architecture specification such that it directly generates code for particular architecture instance. One can explore the design space by finding which architecture that generates the best program performance for their applications.

9.2.3. Support for Data Transfer Model and Multicore Configuration

Data movement remains a problem as demonstrated in section 7.1.2. With better code quality which improve hardware utilization, the concerns move to whether or not the system can deliver the require bandwidth for the accelerator. A dedicated data mover is necessary to support the data consumption pattern of such accelerator. The work of [27] and [28] propose a hierarchical memory architecture for video applications. Such concepts can be adopted to address the current problem. With a dedicated data mover unit, a data transfer model can be created to predict the

cost of moving data to and from the accelerator. Using this model, the scheduler can predict the cost of moving the data out of the accelerator or storing them in the local buffer for further processing.

Huge compute requirements of ConvNets may require more compute power than what is available in the NVE. The vector size of NVE should be increased to improve the compute power although in a certain situation larger vector size might not beneficial due to unused MAC unit at the boundary of the image. Another possible approach is by providing multiple accelerator cores in the system. The computation task of ConvNets can be divided over several accelerator cores. The scheduler should support multicore mapping of ConvNets and a data flow model is important to properly explore the design space.

References

[1] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," in *Proceedings of the IEEE*, 1998.

[2] A. Krizhevsky, I. Sutskever and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks.," in *NIPS*, 2012.

[3] S. Gupta, A. Agrawal, K. Gopalakrishnan and P. Narayanan, "Deep Learning with Limited Numerical Precision.," *CoRR,* vol. abs/1502.02551, 2015.

[4] J. P. Broer, "A Memory-Centric SIMD Neural Network Accelerator: Balancing Efficiency & Flexibility," Eindhoven University of Technology, 2013.

[5] M. Peemen, A. A. A. Setio, B. Mesman and H. Corporaal, "Memory-centric accelerator design for Convolutional Neural Networks.," in *ICCD*, 2013.

[6] M. Peemen, "A Data-Reuse Aware Accelerator for Large-Scale Convolutional Networks," 2014.

[7] M. Peemen, "Lowering the Memory Wall for Convolutional Nets: A Dedicated Accelerator with Support for Layer Fusion and Recomputation".

[8] M. Peemen, B. Mesman and H. Corporaal, "Efficiency Optimization of Trainable Feature Extractors for a Consumer Platform.," in *ACIVS*, 2011.

[9] "www.boost.org," [Online].

[10] Xilinx, "Zynq-7000 All-Programmable SoC: Technical Reference Manual," Xilinx, 2015.

[11] ARM, "Cortex™-A9 NEON™ Media Processing Engine: Technical Reference Manual," ARM, 2011.

[12] V. Gokhale, J. Jin, A. Dundar, B. Martini and E. Culurciello, "A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks.," in *CVPR Workshops*, 2014.

[13] J. Jin, V. Gokhale, A. Dundar, B. Krishnamurthy, B. Martini and E. Culurciello, "An Efficient Implementation of Deep Convolutional Neural Networks on a Mobile Coprocessor," in *MWSCAS*, Texas, 2014.

[14] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems.," in *ISCAS*, 2010.

[15] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *FPGA*, 2015.

[16] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen and O. Temam, "DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning," in *ASPLOS*, 2014.

[17] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun and O. Temam, "DaDianNao: A Machine-Learning Supercomputer.," in *MICRO*, 2014.

[18] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen and O. Temam, "ShiDianNao: Shifting Vision Processing Closer to the Sensor," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, Portland, Oregon, 2015.

[19] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello and Y. LeCun, "NeuFlow: A Runtime Reconfigurable Dataflow Processor for Vision," in *CVPRW*, Colorado Springs, 2011.

[20] Phi-Hung Pham, D. Jelaca, C. Farabet, B. Martini, Y. LeCun and E. Culurciello, "NeuFlow: Dataflow Vision Processing," in *MWSCAS*, Boise, 2012.

[21] M. Sankaradass, V. Jakkula, S. Cadambi, S. T. Chakradhar, I. Durdanovic, E. Cosatto and H. P. Graf, "A Massively Parallel Coprocessor for Convolutional Neural Networks.," in *ASAP*, 2009.

[22] S. T. Chakradhar, M. Sankaradass, V. Jakkula and S. Cadambi, "A Dynamically Configurable Coprocessor for Convolutional Neural Networks.," in *ISCA*, 2010.

[23] J. A. Fisher, P. Faraboschi and C. Young, Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools, Morgan Kaufmann, 2012.

[24] "www.llvm.org," [Online].

[25] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing.," in *MICRO*, 1981.

[26] M. Peemen, B. Mesman and H. Corporaal, "Inter-tile reuse optimization applied to bandwidth constrained embedded accelerators.," in *DATE*, 2015.

[27] A. Beric, R. Sethuraman, J. L. van Meerbergen and G. de Haan, "Memory-Centric Motion Estimator.," in *VLSI Design*, 2005.

[28] A. Beric, J. van Meerbergen, G. de Haan and R. Sethuraman, "Memory-Centric Video Processing.," *IEEE Trans. Circuits Syst. Video Techn.,* vol. 18, no. 4, pp. 439-452, 2008.

Appendix A

Face Detection Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<convNetDescriptor name="Face Detector" layerCount="4">
	<layerDescriptor idx="0">
		<outHeight>358</outHeight>
		<outWidth>638</outWidth>
		<kernelHeight>6</kernelHeight>
		<kernelWidth>6</kernelWidth>
		<subHeight>2</subHeight>
		<subWidth>2</subWidth>
		<outCount>4</outCount>
		<outNet idx="0" cnt="1">0</outNet>
		<outNet idx="1" cnt="1">0</outNet>
		<outNet idx="2" cnt="1">0</outNet>
		<outNet idx="3" cnt="1">0</outNet>
	</layerDescriptor>
	<layerDescriptor idx="1">
		<outHeight>177</outHeight>
		<outWidth>317</outWidth>
		<kernelHeight>4</kernelHeight>
		<kernelWidth>4</kernelWidth>
		<subHeight>2</subHeight>
		<subWidth>2</subWidth>
		<outCount>14</outCount>
		<outNet idx="0" cnt="1">0</outNet>
		<outNet idx="1" cnt="1">0</outNet>
		<outNet idx="2" cnt="1">1</outNet>
		<outNet idx="3" cnt="1">1</outNet>
		<outNet idx="4" cnt="1">2</outNet>
		<outNet idx="5" cnt="1">2</outNet>
		<outNet idx="6" cnt="1">3</outNet>
		<outNet idx="7" cnt="1">3</outNet>
		<outNet idx="8" cnt="2">0,1</outNet>
		<outNet idx="9" cnt="2">1,2</outNet>
		<outNet idx="10" cnt="2">2,3</outNet>
		<outNet idx="11" cnt="2">0,2</outNet>
		<outNet idx="12" cnt="2">1,3</outNet>
		<outNet idx="13" cnt="2">0,3</outNet>
	</layerDescriptor>
	<layerDescriptor idx="2">
		<outHeight>172</outHeight>
		<outWidth>312</outWidth>
		<kernelHeight>6</kernelHeight>
		<kernelWidth>6</kernelWidth>
		<subHeight>1</subHeight>
		<subWidth>1</subWidth>
		<outCount>14</outCount>
		<outNet idx="0" cnt="1">0</outNet>
		<outNet idx="1" cnt="1">1</outNet>
		<outNet idx="2" cnt="1">2</outNet>
		<outNet idx="3" cnt="1">3</outNet>
		<outNet idx="4" cnt="1">4</outNet>
		<outNet idx="5" cnt="1">5</outNet>
		<outNet idx="6" cnt="1">6</outNet>
		<outNet idx="7" cnt="1">7</outNet>
```

```
            <outNet idx="8" cnt="1">8</outNet>
            <outNet idx="9" cnt="1">9</outNet>
            <outNet idx="10" cnt="1">10</outNet>
            <outNet idx="11" cnt="1">11</outNet>
            <outNet idx="12" cnt="1">12</outNet>
            <outNet idx="13" cnt="1">13</outNet>
      </layerDescriptor>
      <layerDescriptor idx="3">
            <outHeight>172</outHeight>
            <outWidth>312</outWidth>
            <kernelHeight>1</kernelHeight>
            <kernelWidth>1</kernelWidth>
            <subHeight>1</subHeight>
            <subWidth>1</subWidth>
            <outCount>1</outCount>
            <outNet idx="0"
cnt="14">0,1,2,3,4,5,6,7,8,9,10,11,12,13</outNet>
      </layerDescriptor>
</convNetDescriptor>
```

Speed Sign Detection Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<convNetDescriptor name="Speed Sign Detector" layerCount="4">
      <layerDescriptor idx="0">
            <outHeight>357</outHeight>
            <outWidth>637</outWidth>
            <kernelHeight>6</kernelHeight>
            <kernelWidth>6</kernelWidth>
            <subHeight>2</subHeight>
            <subWidth>2</subWidth>
            <outCount>6</outCount>
            <outNet idx="0" cnt="1">0</outNet>
            <outNet idx="1" cnt="1">0</outNet>
            <outNet idx="2" cnt="1">0</outNet>
            <outNet idx="3" cnt="1">0</outNet>
            <outNet idx="4" cnt="1">0</outNet>
            <outNet idx="5" cnt="1">0</outNet>
      </layerDescriptor>
      <layerDescriptor idx="1">
            <outHeight>177</outHeight>
            <outWidth>317</outWidth>
            <kernelHeight>6</kernelHeight>
            <kernelWidth>6</kernelWidth>
            <subHeight>2</subHeight>
            <subWidth>2</subWidth>
            <outCount>16</outCount>
            <outNet idx="0" cnt="3">0,1,2</outNet>
            <outNet idx="1" cnt="3">1,2,3</outNet>
            <outNet idx="2" cnt="3">2,3,4</outNet>
            <outNet idx="3" cnt="3">3,4,5</outNet>
            <outNet idx="4" cnt="3">0,4,5</outNet>
            <outNet idx="5" cnt="3">0,1,5</outNet>
            <outNet idx="6" cnt="4">0,1,2,3</outNet>
            <outNet idx="7" cnt="4">1,2,3,4</outNet>
            <outNet idx="8" cnt="4">2,3,4,5</outNet>
            <outNet idx="9" cnt="4">0,3,4,5</outNet>
            <outNet idx="10" cnt="4">0,1,4,5</outNet>
            <outNet idx="11" cnt="4">0,1,2,5</outNet>
            <outNet idx="12" cnt="4">0,1,3,4</outNet>
```

```
            <outNet idx="13" cnt="4">1,2,4,5</outNet>
            <outNet idx="14" cnt="4">0,2,3,5</outNet>
            <outNet idx="15" cnt="6">0,1,2,3,4,5</outNet>
      </layerDescriptor>
      <layerDescriptor idx="2">
            <outHeight>172</outHeight>
            <outWidth>312</outWidth>
            <kernelHeight>5</kernelHeight>
            <kernelWidth>5</kernelWidth>
            <subHeight>1</subHeight>
            <subWidth>1</subWidth>
            <outCount>80</outCount>
            <outNet idx="0" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="1" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="2" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="3" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="4" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="5" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="6" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="7" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="8" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="9" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="10" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="11" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="12" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="13" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="14" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="15" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="16" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="17" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="18" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="19" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="20" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="21" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="22" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="23" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="24" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="25" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="26" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="27" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="28" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="29" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="30" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="31" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="32" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="33" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="34" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="35" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="36" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="37" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="38" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="39" cnt="8">0,1,2,3,4,5,6,7</outNet>
            <outNet idx="40" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="41" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="42" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="43" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="44" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="45" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="46" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="47" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="48" cnt="8">8,9,10,11,12,13,14,15</outNet>
```

```
            <outNet idx="49" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="50" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="51" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="52" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="53" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="54" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="55" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="56" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="57" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="58" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="59" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="60" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="61" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="62" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="63" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="64" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="65" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="66" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="67" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="68" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="69" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="70" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="71" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="72" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="73" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="74" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="75" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="76" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="77" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="78" cnt="8">8,9,10,11,12,13,14,15</outNet>
            <outNet idx="79" cnt="8">8,9,10,11,12,13,14,15</outNet>
      </layerDescriptor>
      <layerDescriptor idx="3">
            <outHeight>172</outHeight>
            <outWidth>312</outWidth>
            <kernelHeight>1</kernelHeight>
            <kernelWidth>1</kernelWidth>
            <subHeight>1</subHeight>
            <subWidth>1</subWidth>
            <outCount>1</outCount>
            <outNet idx="0"
cnt="80">0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23
,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,
47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,7
0,71,72,73,74,75,76,77,78,79</outNet>
      </layerDescriptor>
</convNetDescriptor>
```