TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

VLIW code generation for a Convolutional Network Accelerator

Citation for published version (APA):
Peemen, M. C. J., Wisnu Pramadi, W., Mesman, B., & Corporaal, H. (2015). VLIW code generation for a Convolutional Network Accelerator. In S. Stuijk (Ed.), *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems, SCOPES 2015, 1-3 June 2015, St. Goar, Germany* (pp. 117-120). Association for Computing Machinery, Inc. https://doi.org/10.1145/2764967.2771928

DOI:
10.1145/2764967.2771928

Document status and date:
Published: 01/01/2015

Document Version:
Accepted manuscript including changes made at the peer-review stage

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 04. Oct. 2023

VLIW Code Generation for a Convolutional Network Accelerator

Maurice Peemen, Wisnu Pramadi, Bart Mesman, and Henk Corporaal
Eindhoven University of Technology, the Netherlands
m.c.j.peemen@tue.nl, wisnu.pramadi@student.tue.nl, b.mesman@tue.nl, h.corporaal@tue.nl

ABSTRACT

This paper presents a compiler flow to map Deep Convolutional Networks (ConvNets) to a highly specialized VLIW accelerator core targeting the low-power embedded market. Earlier works have focused on energy efficient accelerators for this class of algorithms, but none of them provides a complete and practical programming model. Due to the large parameter set of a ConvNet it is essential that the user can abstract from the accelerator architecture and does not have to rely on an error prone and ad-hoc assembly programming model. By using modulo scheduling for software pipelining we demonstrate that our automatic generated code achieves equal or within 5-20% less hardware utilization w.r.t. code written manually by experts. Our compiler removes the huge manual workload to efficiently map ConvNets to an energy-efficient core for the next-generation mobile and wearable devices.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers*; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (Hybrid) Systems*

General Terms

Algorithms, Performance, Theory

Keywords

Convolutional Networks, VLIW, Compilation, Code Generation, Software Pipelining

1. INTRODUCTION

Since the last decade *machine learning* has become the dominant paradigm in machine vision applications. Rather than imposing our real-world knowledge of objects into static algorithms, machine learning extracts this knowledge from a rich collection of examples. Especially one category of algorithms, called Deep Learning and Convolutional Networks (ConvNets) has caused this shift by setting the state-of-the-art across a broad range of applications [9, 7, 8].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SCOPES '15, June 01 - 03, 2015, Sankt Goar, Germany
© 2015 ACM. ISBN 978-1-4503-3593-5/15/06 ...$15.00
DOI: http://dx.doi.org/10.1145/2764967.2771928

Although ConvNets achieves superior results for machine vision, it lacks an attribute that is crucial for mobile and wearable applications, and that is energy-efficiency. The rather large computational workload and data intensity has motivated optimized implementations on CPUs [2] and GPUs [4], but these do not fit the constrained (less than 1 Watt) mobile power budget. Our community is well aware of the trend towards heterogeneous computing where architecture specialization is used to achieve high performance at low energy [6]. A few research groups exploited this customization paradigm to design highly specialized, and thus highly efficient, hardware which could enable excellent machine vision for mobile devices [1, 5, 3]. They achieve most of their efficiency gains by tuning data storage structures to the dataflow and data-locality requirements of the algorithm.

The main challenge in accelerator design is to reconcile architecture specialization and flexibility. Especially, the right level of flexibility is key for the adoption of a new core. ConvNets have many parameters such as the layers, feature maps, and kernels, which are different for every task. Hence the architecture should support different parameters efficiently and it should have a programming model. 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.

In this paper we present a ConvNet compiler for a highly specialized Very Long Instruction Word (VLIW) accelerator core [11]. Our 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. Our main contributions that achieve this goal are:

1. A VLIW code generation flow, from a high level `XML` network description to assembly output. (Section 3)
2. We use list scheduling with backtracking, additionally important optimization steps, such as modulo software pipelining. In addition, the usage of HW address generators is automated. (Section 3.3)
3. We evaluate our method against expert manual assembly coding. Performance is max 20% worse compared with code written by human architecture experts. (Section 4)

First this paper gives an overview about ConvNets and the Neuro Vector Engine (NVE) Archtecture in Section 2. The conclusions and possible future work is discussed in Section 5

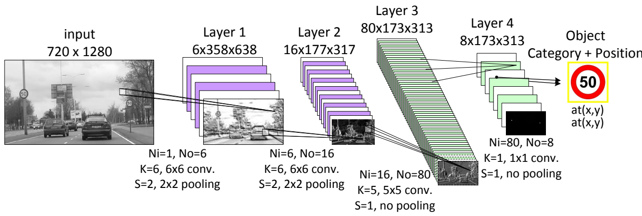

Figure 1: ConvNet for speed sign recognition

2. BACKGROUND

2.1 Convolutional Networks

ConvNets as depicted in Fig . 1, are implemented as a series of connected 2D convolution filters. The coefficients of these filters are obtained by a learning process on a labeled dataset, this ensures that important features are extracted from the input image. E.g. Layer 1 extracts simple features like edges, in further layers these are combined into more complex features such as corners and crossings. In the last layer high level features are combined into decisions such as the detection of a speed sign at a location in the frame and which speed it represents. The filter operations differ over successive layers as illustrated by Fig. 1 e.g., different window sizes, subsampling, and connected featuremaps.

2.2 The Neuro Vector Engine (NVE)

The varying workload in a ConvNet causes that a single efficient compute operator (like in a systolic dataflow based design) is often too specialized and therefore underutilized. In our previous work we proposed the Neuro Vector Engine (NVE) [11], a programmable accelerator core with a high degree of flexibility. To obtain a good balance between efficiency and flexibility the core employs the Single Instruction Multiple Data (SIMD) principle i.e., we can change the operation per instruction, but share control overhead over many parallel execution units.

2.2.1 *Vector Data-Path*

Fig. 2 illustrates the pipelined data path of the core NVE, as the name implies all possible data operations are performed on vectors. The main compute primitive is a *Vector Multiply Accumulate* with scalar $\mathbf{y} \leftarrow \mathbf{y} + \mathbf{x} \times w$. It modifies an array of accumulator values $\mathbf{y}$ representing neighboring neurons. By sequentially repeating this operation for all inputs of a 2D convolution window this resource is optimally utilized for different windows sizes. The *Register Operation* stage in Fig. 2 performs the vector shift operations that are necessary to provide the correct input elements $\mathbf{x}$ and broadcast the weight coefficient w. If the input window is processed the results are saturated and normalized by an activation function implemented as a vector look up table. Data reuse of overlapping windows is exploited by the *Local Scratchpad Buffer* that holds vectors of input neurons and coefficients.

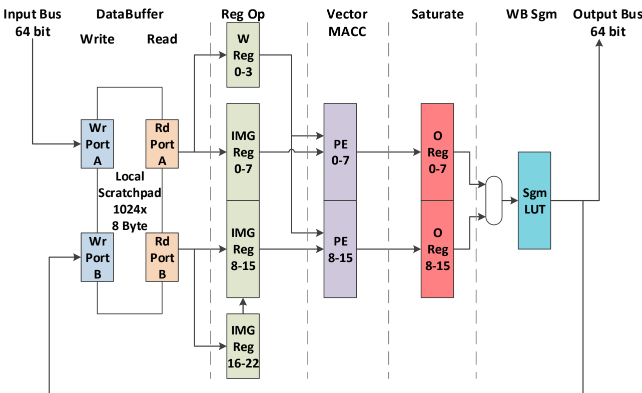

Figure 2: Pipelined accelerator datapath

Write Local	Read Local	Reg Operation	EX VMAC	WB Sgm
	Rd a 0, [b0 w0-1]	Set i3, Shift W	MAC,w6, c2 i0	
	Rd ab 4 5, [c0,i0-15]	Shift i0 i1, Shift W	MAC,w7, c2 i1	
Wr[c3,i0-7], 4	Rd b 6, [c0,i16-17]	Set W0	MAC,w8, c2 i2	
	Rd a 1, [w2-4]	Set i0 i1, Shift W	Set, b0	
	Rd ab 7 8, [c1,i0-15]	Set i3, Shift i0 i1, Shift W	MAC,w0, c0 i0	
Wr[c3,i8-15], 5	Rd b 9, [c1,i16-17]	Shift i0 i1, Set W1	MAC,w1, c0 i1	
	Rd a 2, [w5-8]	Set i0 i1, Shift W	MAC,w2, c0 i2	Sigm0, Wr
	Rd ab 10 11, [c2,i0-15]	Set i3, Shift i0 i1, Shift W	MAC,w3, c1 i0	Sigm1, Wr
Wr[c3,i16-17], 6	Rd b 12, [c2,i16-17]	Shift i0 i1, Set W2	MAC,w4, c1 i1	
	NOP	Set i0 i1, Shift W	MAC,w5, c1 i2	

Figure 3: VLIW steady-state for 3x3 convolution

2.2.2 *VLIW Programming Model*

The control of the successive stages in the architecture is distributed over *issue slots*, implemented as a Very Long Instruction Word (VLIW) core. Each slot is very specific, so the instruction with is small. Using instructions creates flexibility but it causes overhead. To reduce the instruction overhead we use two techniques. **1) SIMD:** Operate on vectors as discussed above this increases the amount of useful work per instruction. **2) Modulo Software Pipelining:** We create a schedule of instructions (Steady State) that is repeated very often. As a result, a lot of instruction reuse can be exploited by an instruction buffer which substantially reduces the instruction transfer overhead.

A simple example of the steady-state of a 3x3 convolution filter is given in Fig. 3. One execution of the steady-state (10 cycles) results in 16 neighboring 3x3 convolution outputs, requiring is 144 MACC ops. For simple programs manual construction of schedules is feasible. However complex ConvNet layers require over 200 instructions in the steady-state; additionally prologue and epilogue must be created. Manual programming is very-time consuming and error prone, so an automatic optimizing compiler is key for the NVE.

3. AUTOMATIC CODE GENERATION FLOW

Our compilation flow, as depicted in Fig. 4, requires an `XML` network description as input. First, the `XML` is converted into a series of task graphs, more specifically each output feature-map has a separate task graph. For each graph a schedule is created, these are later combined by an optimizing bundling procedure. The result is a very efficient executable VLIW program.

3.1 XML Network Specification

As stated before we use an `XML` ConvNet description as input instead of a program. Listing 1 contains a description of a network layer similar to layer 1 in Fig. 1. As shown the `XML` lists size parameters as the width and height of the feature-maps and convolution kernels. Also the connections between feature-maps are specified, e.g. `outMap` specifies the connection of an output feature-map `idx` with a number of input feature maps `cnt`, in this case one input feature-map.

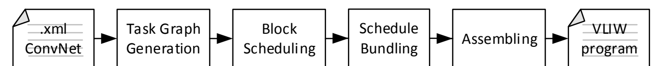

Figure 4: Automatic code generation flow

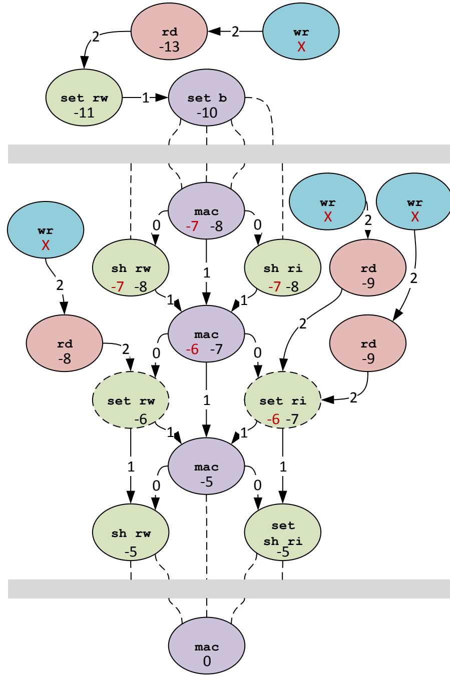

Figure 5: Partial DAG for a 3x3 convolution kernel

3.2 Task Graph Construction

Graph construction is started by connecting the `set bias` and the successive `MAC` operations according to the kernel size, as detailed in Fig. 5. Next, the required register ops to provide data are generated and dependencies are connected. The same is performed for the reads from and writes to the local scratchpad. All dependencies have a *distance* parameter representing the minimum latency between adjacent vertices. In a stage the distance is always one cycle, e.g. adjacent MAC ops have a distance of one. Across stages the distance varies, e.g. scratchpad ops are pipelined so moving data to a register has a distance of two.

3.3 Instruction Scheduling

For scheduling of the vertices, we use bottom-up *Breadth First Search (BFS)* through the DAG. The bottom-up approach is a good heuristic that minimizes the number of scheduling conflicts. In Fig. 5 the assigned schedule positions are numbered in the vertices. However architecture constraints sometimes cause conflicts on `set rw` (set weight) and `set ri` (set img reg) operations, see the dashed vertices. If a conflict is detected it is solved by a backtracking procedure that revises the schedule from the conflict point.

Listing 1: Convolutional network description of first layer face detection in XML

```xml
<NVEDescriptor>
 <layer idx="0">
  <outHeight>638</outHeight>
  <outWidth>368</outWidth>
  <kernelHeight>6</kernelHeight>
  <kernelWidth>6</kernelWidth>
  <subHeight>2</subHeight>
  <subWidth>2</subWidth>
  <outCount>4</outCount>
  <outMap idx="0" cnt="1">0</outMap>
  <outMap idx="1" cnt="1">0</outMap>
  <outMap idx="2" cnt="1">0</outMap>
  <outMap idx="3" cnt="1">0</outMap>
 </layer>
</NVEDescriptor>
```

Algorithm 1 Task Graph Scheduling

1: **procedure** CREATESCHEDULE(G,v)
2: create queue Q
3: create table T
4: Q.push(v)
5: v.schedule $\leftarrow$ 0
6: v.status $\leftarrow$ `visited`
7: **while** not Q.empty() **do**
8: $v \leftarrow Q$.pop()
9: **for all** edges e from w to v in G.edges(v) **do**
10: **if** w.instruction $\neq$ `wr` and w.instruction $\neq$ `act` **then**
11: **if** w.status $\neq$ `visited` **then**
12: Q.push(w)
13: w.schedule $\leftarrow$ v.schedule - e.distance
14: w.status $\leftarrow$ `visited`
15: **if** w.instruction $=$ `set r` **then**
16: **while** T.exist(w.schedule) **do**
17: w.schedule $\leftarrow$ w.schedule - 1
18: T.insert(w.schedule)
19: **else**
20: newSchedule $\leftarrow$ v.schedule - e.distance
21: **if** newSchedule $<$ w.schedule **then** ▷ revised
22: w.schedule $\leftarrow$ newSchedule
23: Q.push(w)
24: **for all** edges $\bar{e}$ from $\bar{w}$ to w in G.edges(w) **do**
25: **if** Q.exist($\bar{w}$) **then**
26: Q.remove($\bar{w}$)
27: $\bar{w}$.status $\leftarrow$ `not visited`

Algorithm 1 lists the scheduling procedure that constructs the schedule as given in Table 1. Starting bottom-up in the DAG of Fig. 5 a conflict occurs when scheduling two `register set` ops at cycle -6 (detected in the resources table), so one of them is moved to -7. However, the already scheduled `MAC` on -6 has a problem which is detected after scheduling the connected parents. The backtrack procedure revises the scheduling of the `MAC` to -7 (insert a stall) and updates visited parents to ensures a correct schedule.

After scheduling the `read`, `register`, and `MAC` operations, the `activation` operations are scheduled by an *As Soon As Possible* heuristic. The scratchpad `write` operations are split into a prologue and steady-state part. The prologue contains coefficient-writes and the first set of image-writes. The steady-state part writes the non-overlapping image-writes with an *As Late As Possible* heuristic.

The bundling procedure combines schedules (output feature-maps) to increase data reuse over shared input feature-maps. This combining stops when the instruction buffer is full, in this case a new bundle is started. Algorithm 2 lists the bundling procedure, `B` holds the individual schedules and `R` is the resulted program list in partitions.

During the bundling process, *Modulo Scheduling* and *Data Buffer Allocation* are performed. Similar to earlier work [10] we count the resource-minimum initiation interval of the steady state and wrap around to reduce the number of instructions. *Data Buffer Allocation* generates addresses for all buffer accesses, for the steady state repetitive patterns are extracted and stored in the address generation unit (AGU), which implements a rotating buffer.

Table 1: Graph schedule and resource table

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

resource table	-2	-3	-6	-7

Algorithm 2 Schedule Bundling

```
1:  procedure SCHEDULEBUNDLING(B, R)
2:    best ← current ← B.first
3:    B.remove(B.first)
4:    for all schedule b in B do
5:      combine(current,b)                  ▷ store in current
6:      temp ← current                      ▷ make a copy
7:      moduloSchedule(current)
8:      if current.count > capacity then    ▷ exceeded, use best
9:        moduloSchedule(best)
10:       allocateBuffer(best)
11:       r ← Assemble(best)                ▷ create binary
12:       R.insert(r)                       ▷ store program partition
13:       best ← current ← b                ▷ start new
14:     else                                ▷ bundling fit
15:       best ← current ← temp ▷ prepare for next bundling
16:   moduloSchedule(best)
17:   allocateBuffer(best)
18:   r ← Assemble(best)
19:   R.insert(r)
```

4. EXPERIMENTAL EVALUATION

Our generated code quality is evaluated by comparison with assembly coding by architecture experts. Table 2 presents code metrics such as, effective number of MAC ops per instruction, data transfers, code size, and number of code partitions. For some layers, performance in MAC/instruction is equal but often the automatic code has 5-20% less performance w.r.t. manual. A manual programmers can reorder the content of weight vectors this is done to prevent scheduling conflicts, and is not yet available in our automatic tools.

Due to feature map combining (bundling) the automatic generated schedules often have less data transfers. Especially, for dense ConvNet layers like layer 2 and 3 combining removes many redundant image loads. For a manual programmer this option is more difficult to explore because it generates larger and more complicated programs. Automatic code generation has a huge productivity advantage for the programmer, e.g. an architecture expert spends about 4 hours to code a complex layer, our automatic flow performs this in a few seconds.

5. CONCLUSIONS

Customized hardware acceleration for ConvNets has significantly improved their computational efficiency. The last bottleneck for market adoption of such accelerators is the complexity of programming these accelerators. We presented a new compilation flow, from a high level XML ConvNet description to an optimized accelerator VLIW program. By using advanced techniques such as Modulo Scheduling and HW address generation units we can generate code that achieves a hardware utilization at max 20% lower than code written by an expert. Due to our feature-map combining technique our generated code always has better or similar locality, which reduces the challenging data transfer requirements.

The introduction of our compiler flow enables users to abstract from the accelerator architecture to only a network specification. This abstraction reduced the workload involved when programming a ConvNet 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.

Table 2: Manual coding vs auto code generation

		Utilization [MAC/instr.]	Data transfer [word]	Code size [word]	Code partition
Architecture Expert Manual Coding					
3x3 filter		14.39	6401	21	1
Face detection	L1	13.7	11554	238	1
	L2	10.95	72784	974	14
	L3	15.53	22232	910	14
	L4	5.89	9378	37	1
Speed sign detection	L1	13.7	14104	342	1
	L2	13.67	202744	4950	16
	L3	15.13	663040	31040	80
	L4	6.26	50645	203	1
Our Automatic Code Generation					
3x3 filter		13.08	6399	20	1
Face detection	L1	12.79	11544	240	1
	L2	10.35	31230	658	2
	L3	15.53	22190	868	2
	L4	4.87	9364	50	1
Speed sign detection	L1	12.79	14094	350	1
	L2	12.78	182960	4336	13
	L3	15.13	660720	28640	80
	L4	4.88	25605	283	1

6. REFERENCES

[1] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In *ISCA'37*, 2010.

[2] K. Chellapilla, S. Puri, and P. Simard. High performance convolutional neural networks for document processing. In *10th International Workshop on Frontiers in Handwriting Recognition*, 2006.

[3] 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'19*, 2014.

[4] D. C. Ciresan, U. Meier, J. Masci, L. Maria Gambardella, and J. Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *IJCAI*, 2011.

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

[6] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA'37*, volume 38, pages 37–47. ACM, 2010.

[7] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In *IEEE 12th International Conference on Computer Vision*, 2009.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS'25*, 2012.

[9] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In *ICML'24*, 2007.

[10] J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero. Swing module scheduling: a lifetime-sensitive approach. In *PACT*, 1996.

[11] M. Peemen, B. Mesman, and H. Corporaal. A data-reuse aware accelerator for large-scale convolutional networks. In *NeuroArch Workshop at ISCA '41*, 2014.