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.

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 data-flow 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) Architecture in Section 2. The conclusions and possible future work is discussed in Section 5.

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
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 feature maps.

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 $y \leftarrow y + x \times w$. It modifies an array of accumulator values $y$ representing neighboring neurons. By sequentially repeating this operation for all inputs of a 2D convolution window this resource is optimally utilized for different window sizes. The Operation stage in Fig. 2 performs the vector shift operations that are necessary to provide the correct input elements $x$ 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.

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) Module 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 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 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 id $x$ with a number of input feature maps $cnt$, in this case one input feature-map.

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 id $x$ with a number of input feature maps $cnt$, in this case one input feature-map.
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
<layer id="0">
  <outHeight>438</outHeight>
  <outWidth>368</outWidth>
  <kernelHeight>6</kernelHeight>
  <kernelWidth>6</kernelWidth>
  <subHeight>2</subHeight>
  <subWidth>2</subWidth>
  <outMap id="0" cmt="1">0</outMap>
  <outMap id="1" cmt="1">0</outMap>
  <outMap id="2" cmt="1">0</outMap>
  <outMap id="3" cmt="1">0</outMap>
</layer>
</NVEDescriptor>
```

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 ensure 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>
<thead>
<tr>
<th>cycle</th>
<th>MEMS</th>
<th>MEMD</th>
<th>WRID</th>
<th>IREG</th>
<th>VMAC</th>
<th>ACT</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8</td>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-7</td>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| resource table | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
```
Algorithm 2 Schedule Bundling

1: procedure scheduleBundling(B, R)
2: best ← B.first
3: R.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 performance w.r.t. manual. Due to our automatic tools this is done to prevent scheduling conflicts, and is not yet available in our automatic tools.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Time (hours)</th>
<th>Speedup</th>
<th>Data Transfer (MAC/instr.)</th>
<th>Code Size (word)</th>
<th>Code Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture Expert</td>
<td>14.09</td>
<td>16.2</td>
<td>11554</td>
<td>248</td>
<td>10</td>
</tr>
<tr>
<td>Face detection</td>
<td>13.7</td>
<td>14.3</td>
<td>24274</td>
<td>4950</td>
<td>14</td>
</tr>
<tr>
<td>Speed sign detection</td>
<td>12.79</td>
<td>14.9</td>
<td>14094</td>
<td>28040</td>
<td>80</td>
</tr>
</tbody>
</table>

6. REFERENCES