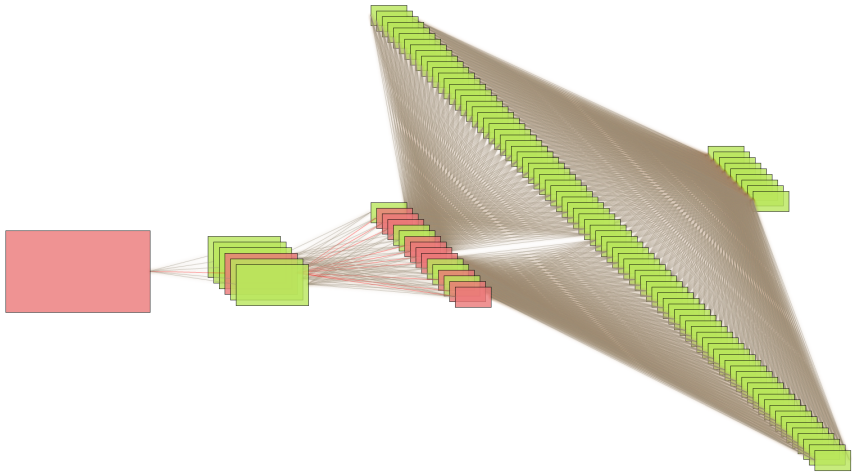
A memory-centric SIMD neural network accelerator balancing efficiency & flexibility

Broere, J.P.

Award date:
2013

A memory-centric SIMD neural network accelerator: Balancing efficiency & flexibility

J.P. Broere (0629355)



Master thesis

Supervisors: Henk Corporaal and Maurice Peemen

Faculty of Electrical Engineering
Eindhoven University of Technology

August 20, 2013

Technische Universiteit Eindhoven University of Technology

Abstract

Nowadays, cameras are everywhere. These cameras can be used to secure properties, as safety for road users, marketing and many other reasons. To perform these tasks, it is required to have object recognition. Object recognition is required at the camera to reduce network dependability and therefore energy. Since the same camera can be used to detect various object, it is required to have a general approach for object recognition.

Convolutional Neural Networks (CNNs) offer this general approach. The advantage of CNNs is that they can be easily *learned* to detect objects and are flexible to support different kind of vision tasks. Running different CNN configurations on an embedded platform is a challenging task. General Purpose processors can not meet the throughput requirement within realistic power constraints, and the known dedicated CNN accelerators do not have the required flexibility. To overcome the limitations of the current state of the art accelerators, a flexible, clustered, Single Instruction, Multiple Data (SIMD) accelerator is designed. The SIMD paradigm is used to efficiently exploit available parallelism in the algorithm. A specialized memory hierarchy is used to reduce the enormous data transfer rates to external memory. To maximize utilization for the varying work load in a CNN, a dedicated optimizing toolflow is developed. The evaluation shows that the proposed solution can achieve 7.4 Giga Operations Per Second (GOPS) with a utilisation of 57% of the available Processing Elements (PEs). Due to the memory hierarchy the required external bandwidth is reduced from 97.5GB/sec to 400MB/sec, for these benchmarks. The power consumption of mapping to the ZedBoard[1] indicates a reduction of $3-30\times$ compared with a GPU implementation, and $6-52\times$ compared with a CPU implementation. The result of this work is a very flexible platform for CNN based vision applications that achieve state-of-the-art performance in throughput and energy.

List of Acronyms

PE Processing Element

LUT Look-up Table

CNN Convolutional Neural Network

MACC Multiply Accumulate

GPGPU General-Purpose computing on Graphics Processing Units

GPU Graphics Processing Unit

SIMD Single Instruction, Multiple Data

GOPS Giga Operations Per Second

Contents

List of Figures

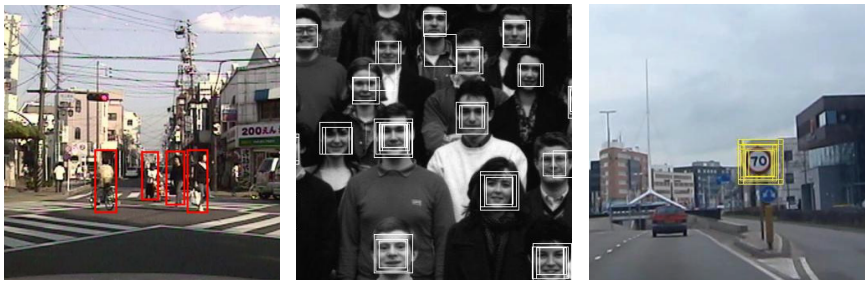
List of Tables

1 Introduction

Nowadays, cameras are everywhere. For privacy and performance reasons camera's require object recognition at camera level. Mobile devices, such as the *Google Glass*, also require executing these vision tasks on the mobile platform. Object recognition applications should be robust and fault tolerant. Many application specific algorithms [2],[3],[4] for object recognition are created in the literature. A more general approach to vision tasks is neural networks. Neural networks emulate the human brain and can be used for multiple vision tasks. The advantage of neural networks is that they can *learn*. When the network is presented with enough examples, it can adapt itself to learn from the examples. Because of the limited free parameters of a neural network the network must generalise to learn the examples, and therefore it can be used for detecting elements that are not in the example set. The advantage of the learning capabilities is that domain specific characteristics do not need to be known, only an example set needs to be created.

Hubel and Wiesel[5] worked on the visual cortex of the brain of a cat and they discovered that the neurons were only stimulated by a small region of the input. Using this knowledge, Le Cun [6] generated a Convolutional Neural Network (CNN) to recognise hand written digits. A CNN is a multilayer neural network designed specifically for detecting 2D objects, with robustness against distortion, scaling and skewing.

There are many other reported applications that use CNN, such as pedestrian detection [7], face detection [8] and speed sign recognition [9]. CNNs can outperform and replace algorithms for vision tasks [10].



(a) Pedestrian detection (b) Face detection (c) speed sign recognition

Figure 1: Example applications of CNNs

Using CNNs for real-time object recognition on HD-video can require billions of computations per second.

By using General-Purpose computing on Graphics Processing Units (GPGPU), real-time object recognition is achieved [9]. This is not a feasible solution for a mobile platform: Graphics Processing Units (GPUs) that have enough compute power to do real-time object recognition use around 100 watts of power. A more application specific solution is needed to reduce the energy requirement.

With a CNN it is possible to do hundreds of computations in parallel, making it possible to operate at a lower frequency, to reduce the power requirement.

In the literature, different accelerators for CNNs are presented. NEC Laboratories America [11] present a massively parallel coprocessor for accelerating CNNs and made it dynamically configurable [12]. The New York University's

Computational & Biological Learning Laboratory has developed NeuFlow[13]: A Runtime Reconfigurable Dataflow Processor for Vision. Both solutions require a large memory bandwidth, are for specific network types and require a general purpose CPU, making it less energy efficient. A general architecture is required that can calculate many different CNNs efficiently with the same hardware. This architecture should reduce the required external memory bandwidth to reduce the energy requirement.
In that way, the same hardware can be used for many vision tasks, which make it cheaper to produce.
In this report I will make the following contributions:

- A scalable, flexible, data transfer efficient architecture for multiple CNNs.
- A complete toolflow to map a CNN to efficiently work on the architecture.
- Peformance and energy analysis of the new architecture with different configurations and CNNs.

This report will start with an explanation of neural networks and the computations involved in section 2. Section 3 shows which features of CNN can be exploited for the architecture. Section 4 describes the designed architecture. The created toolflow is described in section 5. The experiments are described in section 6. Ending with a conclusion in section 7 and future work in section 8

2 Convolutional Neural Networks

Recall that a CNN is a multilayer neural network designed specifically for detecting 2D objects, with robustness against distortion, scaling and skewing. A CNN consists of multiple artificial neurons that are connected to each other, forming a neural network. These artificial neurons emulate biological neurons in, for example, the human brain.

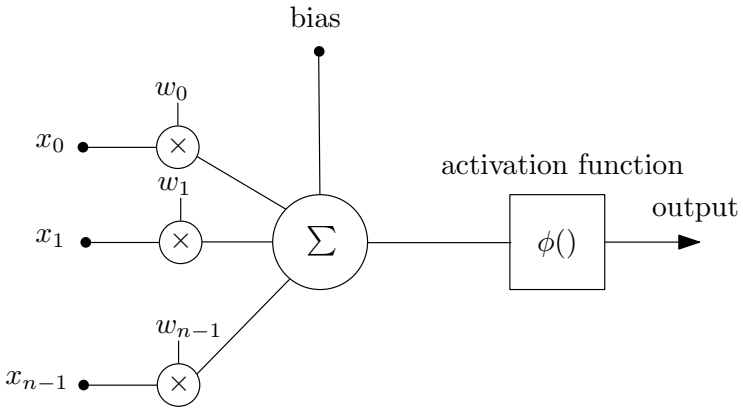


Figure 2: Model of a non-linear neuron

"A *neuron* is an information-processing unit that is fundamental to the operation of a neuron network" [14]. A neuron consist of the following four basic elements:

- A set of connections to other neurons. These connections have a certain weight associated with them
- An adder that sums up all the input signals
- A fixed bias for increasing or decreasing the net input
- An activation function ϕ, limiting the output value.

A graphical representation of a neuron is given in figure 2 while equation 2.1 gives a mathematical representation.

$$\text{output} = \phi(\text{bias} + \sum_{j=1}^{n} w_j x_j) \tag{2.1}$$

A CNN is a specific type of neural network. CNNs consists of several *featuremaps*. A featuremap consists of a set of neurons, neurons of a featuremap are connected to multiple neurons of other featuremaps. The featuremaps are hierarchically divided into *layers*. The featuremaps in a layer only receive input from featuremaps in the previous layer. In figure 3 a small example CNN is depicted. The input consists of a single featuremap, depicted as a rectangle. These rectangle consists of multiple neurons, which are represented as pixels. The first layer consists of three featuremaps, all using the input-featuremap as input. The second layer has two featuremaps with a different connection pattern. The output layer of this CNN consists of only a single featuremap.

Since this network has only a single featuremap as output, it can only be used to *detect* object and not *recognize* them. This is because it is only possible to output a true/false for each location, since the output needs to be quantified. Each connection in this network is displayed with an arrow; such a connection has a associated *kernel*.

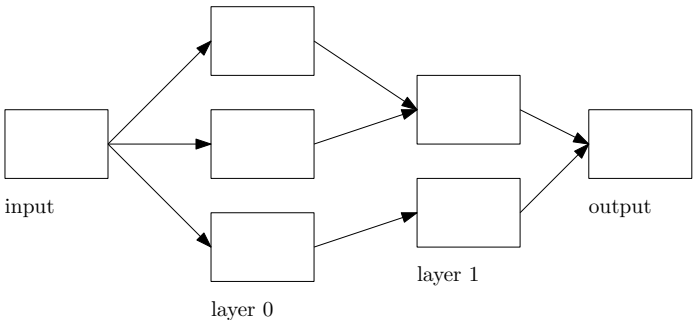


Figure 3: An example of a small CNN

The rest of this section talks about all the computations required to calculate the result of a CNN. It will end with two example CNNs for recognising speed-signs and detecting faces.

2.1 Convolutions

Each connection between featuremaps an associated kernel. The width (l_{kw}) and height (l_{kh}) is constant for all connections between the same layers. The kernel-size describes the number of neurons to which each neuron in a featuremap is connected to. In figure 4 there are two layers with a single featuremap, showing the individual neurons of the featuremaps. The neurons are represented as individual pixels. To which neurons a neuron receives input from is determined by the position in the featuremap. Each connection between two featuremaps has an associated kernel with it, consisting of $l_{kw}l_{kh}$ weights. Weight values of a kernel are obtained by *training* the network with examples.

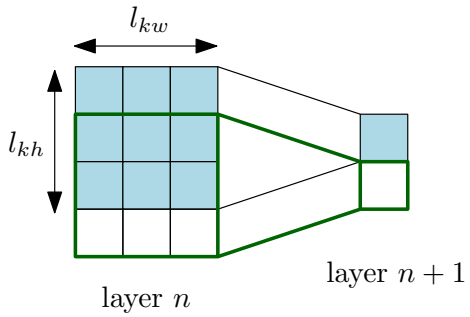


Figure 4: Example with two convolution operations, using a 3×3 kernel

To calculate the output value of featuremap of layer $n+1$, the values of the $l_{kw}l_{kh}$ input values of layer n are multiplied with the associated weights and aggregated. This form of multiplying and adding is called a *discrete convolution* operation. A mathematical representation of a convolution operation is given

in equation 2.2, with W the weights associated with the connection and X the input values. $X_{i,j}$ means the $j'th$ colomn of the $i'th$ row of the input featuremap X. $W_{i,j}$ means the $j'th$ colomn of the $i'th$ row of the weight values in the kernel.

$$Y_{m,n} = \sum_{i=1}^{l_{kw}} \sum_{j=1}^{l_{kh}} W_{ij} X_{i+m,j+n} \tag{2.2}$$

A featuremap can be connected to multiple input featuremaps, in that case each connection has different kernel weights associated with it. To calculate the results, the convolution of equation 2.2 is used on all the featuremaps and aggregated. The bias value of the featuremap is added to the results.

2.2 Subsampling

Subsampling is used to reduce the number of computations and to reduce the sensitify of a featuremap to certain distortions. A layer in a CNN has an associated subsample factor l_{ss}. An optimized model of subsampling [15] is used to reduce the number of operations. This model will merge the convolution and the subsampling into a single step. In this model, the subsampling factor is translated to the amount of input neurons that two consecutive neurons have between them. In figure 5 there is a l_{ss} of 2, therefore the convolution of the second output neuron starts 2 input neurons lower than the convolution of the first neuron.

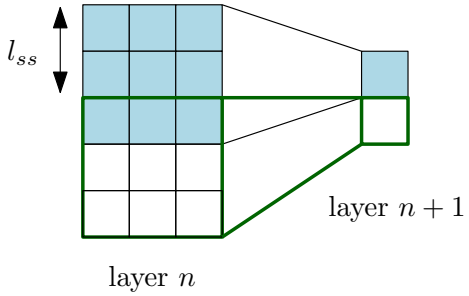


Figure 5: Example of subsampling

2.3 Activation function

When all the convolutions for a single neuron are done and aggregated, the associated *bias* of the featuremap is added. The non-linear activation function is used on this aggregated value. This is the result that neurons of the next layer use as their input. Example activation functions using in CNNs are $\phi(x) = tanh(x)$ or $\phi(x) = (1 + exp(-x))^{-1}$

2.4 Images

An image, such as a videoframe, consists of pixels. With a grayscale image, the pixel can be described by a number $0 \leq \alpha \leq 1$. α determines the brightness of the pixel. An image can be directly used in a CNN: each pixel of the image can be modelled as a neuron with a certain output value. Therefore, there is no preprocessing required of the image to use as input for a CNN.

2.5 Example networks

Different vision applications require different CNNs. CNNs can have different connection schemes, different kernel sizes and different amount of featuremaps. To be able to demonstrate this flexibility of the architecture, two very different CNNs will be used for analyses and testing.
The Speed-Sign Recognition Network is more computation dependant. This is used to demonstrate performance. The Face Detection Network is more data dependent and this is used to demonstrate how well the architecture coops with low memory bandwidth. Both networks have a non-trivial connection scheme, which the architecture should be able to work with.
The requires precision of the values of thesse network is as follows:

- output of the neurons of featuremaps: 1 byte
- weight values: 2 bytes
- intermediate values before using the activation function: 4 bytes

This precision is required to get reliable results out of the CNN.

2.5.1 Speed-Sign Recognition Network

This network is able to detect 8 different speed-signs in videoframes of 1280×720 pixels. The network is inspired by the digit recognition network LeNet-5 [16]. The first layer contains 6 featuremaps with a subsampling factor of 2 and a 6×6 kernel, the second layer contains 16 featuremaps with a specific connection scheme with a 6×6 kernel and again a subsampling factor of 2. This scheme forces the feature maps to learn specific features. The next layer has 80 featuremaps with a 5×5 kernel and is connected to half of the previous featuremaps. The last layer, with specific speed sign information, contains 9 featuremaps with a kernel of 1×1. One featuremap detects if there is a speed-sign at a certain location and the other featuremaps are the specific speed-sign.

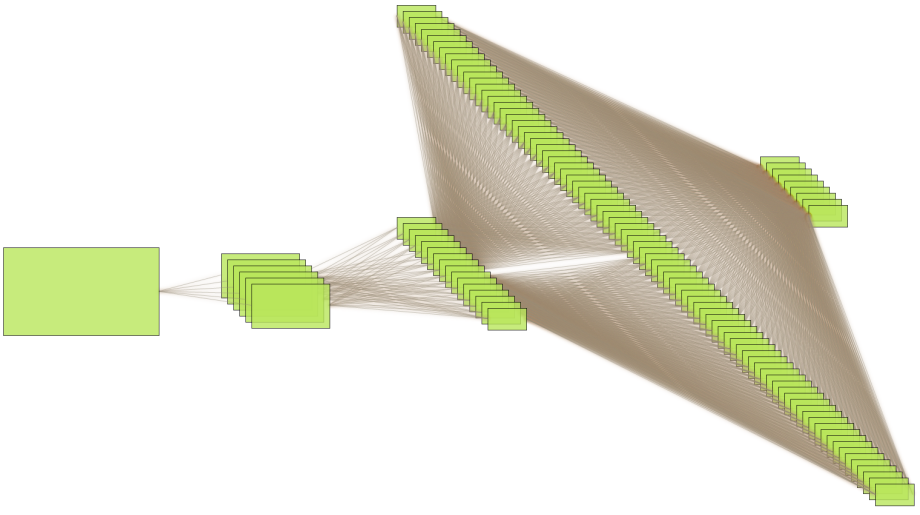


Figure 6: Speed-Sign recognition network

Multiply Accumulate operations / frame (10^6)	1075.9
Output values / frame (10^6)	7.09

2.5.2 Face Detection Network

This network is able to detect faces in videoframes of 1280×720 pixels. The first layer consists of 4 featuremaps with a subsampling factor of 4 and a kernel of 6×6, the second layer consists of 14 featuremaps with a specific connection scheme with a subsampling factor of 2 and a kernel of 3×3. The third layer has an one-on-one connection scheme with a kernel of 6×6 and the last layer consists of one featuremap with a 1×1 kernel. This network can only detect faces, not classify them, and is therefore much smaller than the previous network.

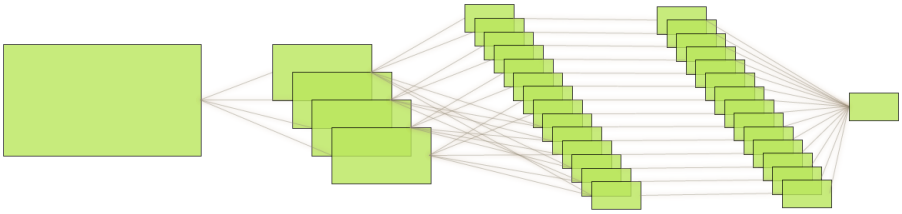


Figure 7: Face recognition network

Multiply Accumulate operations / frame (10^6)	78.95
Output values / frame (10^6)	2.51

2.6 Conclusion

This section described the basic elements of a CNN. The hierarchy featuremaps and layers, the required computations with kernel values. Furthermore, subsampling is explained. Two CNNs are presented to show the required number of computations and the different kind of kernel-sizes and connection schemes.

3 Data and computations

The goal is to design a programmable architecture that achieves high performance for different kinds of CNNs with small energy usage.
With the same architecture, both CNNs with high computation density and with high memory requirements should be calculated with high utilisation.
This section describes and motivates a pattern to go trough the computations. The used pattern supports tiling, this is used to reduce the required external memory bandwidth, and therefore the required energy usage.

3.1 Energy usage

Performing the computations in CNN requires many memory accesses. As an illustrative example to demonstrate energy usage of the memory, assume having a 512MB Lower-Power DRAM consisting of 4 banks and a 36864 byte SRAM that consists of a single bank. Using CACTI of HP [17] with a technology size of $45nm$ results in the following numbers:

	512MB LP-DRAM	**36KB SRAM**
Energy per read (nJ)	0.67	0.013
Access time (ns)	13.9	1.567

Using only the large DRAM for calculating the output values of the Speed-Sign Recognition Network, this would need around $4.9 * 10^9$ memory accesses, resulting in an energy usage of $3.3J$ for the memory accesses.
If it is possible to reduce the number of accesses to the DRAM with a factor of 100, only $(4.9 * 10^7 * 0.67nJ) + (4.85 * 10^9 * 0.013nJ) = 96mJ$ is necessary; a reduction of 34×.
Another advantage of using a smaller memory is the reduced access time. This means that the available bandwidth of the local memory is larger in comparison with the large DRAM.

3.2 Reuse

To reduce the number of accesses to the DRAM, the architecture should reuse the data in the local memory. Peemen[18] calculates the best computations order for a given a local memory size, reusing data as much as possible. Only a part of the input data is stored in local memory, called a tile. When these results are generalised, it becomes clear that moving the tile in the direction of the largest dimension of the featuremap is the most beneficial. The tile should contain all the input feature maps needed to calculate the output featuremaps while also maximizing the number of output featuremaps.
Consider the example in figure 8, the first layer consists of two featuremaps and the second layer consists of three featuremaps. The second layer uses the first layer as input for the calculations. A kernel-size of 4×4 is used for the second layer.
The featuremap of pixel $p00$ is connected to both input featuremaps. To compute the value of this pixel, it would require $4 * 4 * 2 = 32$ input values from external memory. To compute the values of $p00, p01, p10, p11, p20, p21$ it requires $6 * 32 = 192$ input values from external memory.
Keeping the values within the blue square (the tile) in local memory will reduce the number of external memory accesses. With the values currently in local memory, $p00, p10$ and $p20$ can be calculated. If the tile moves one position to

the right, requiring reading 8 pixels from external memory, pixels $p01, p11$ and $p21$ can be calculated. In this case, we only require 40 input values from external memory.

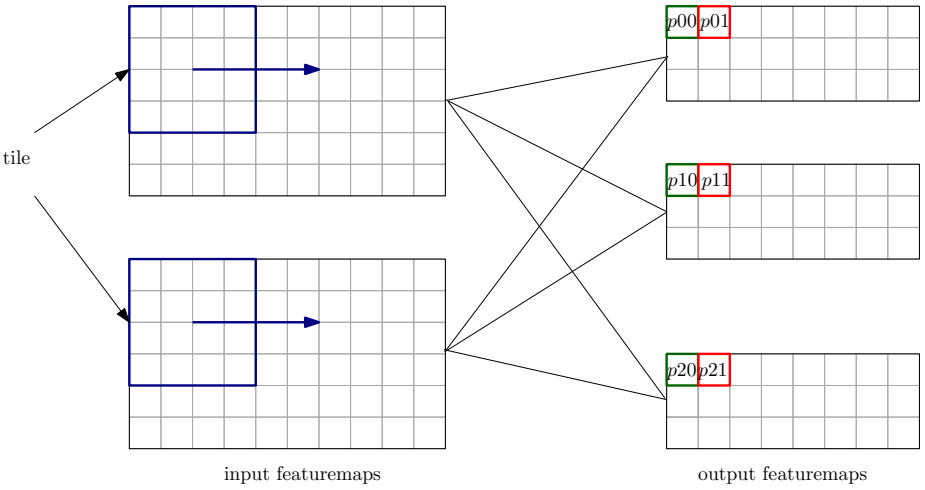


Figure 8: Example network, with a kernel of 4x4.

It is also beneficial to make the tile higher. If the 3D tile in figure 8 would be two pixels higher, it would be possible to calculate 3 pixels with the tile. To calculate the next 3 pixels, $2 * 6$ new values are required: a total reduction of factor 24.
Based on the relation between the tile height and the number of vertical pixels (n) it is possible to calculate the required tile height with equation 3.1

$$\text{tileheight} = (n-1)l_{ss} + l_{kh} \tag{3.1}$$

For the first output element, there are l_{kh} pixels in the vertical direction needed. For every new output element, l_{ss} extra pixels are needed.
Storing the weight values in the local memory is also beneficial. Every pixel in the featuremap uses the same weights values and storing these values in local memory reduces the number memory accesses for weight values with a factor $l_{width} * l_{height}$.

However, increasing the tile over multiple layers imposes a problem. It would create a dependency between calculating PEs on different layers. In figure 9 it is clear that there is a balancing problem: to calculate a single pixel in the last layer requires many more calculations in the previous layers. To make this pipeline balanced, the architecture should support control of individual PEs, which requires much more hardware. Furthermore, the required size of the tiles of the first layer(s) would be very large to support enough data for the final layer.

3.3 Tiling

Is it possible to tile the complete layer, thus making it possible to calculate the complete layer without partial results?

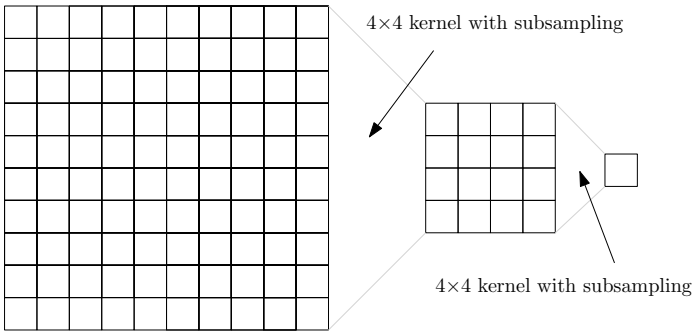


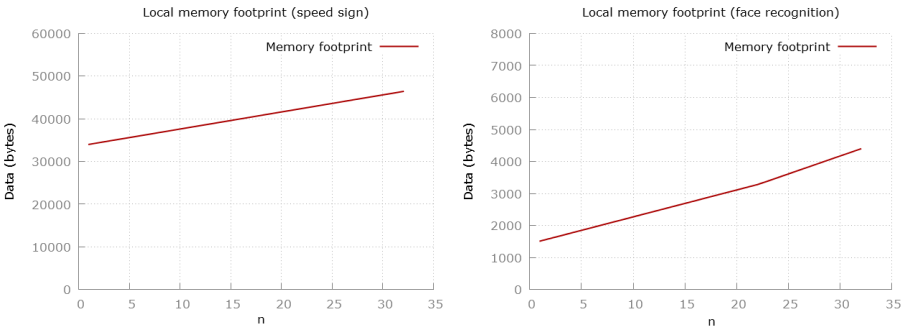
Figure 9: Example of propagation problem with two layers

To calculate the required memory size, the size of the tile and the weight data need to be aggregated. Every layer is solved independently from the other layer, so the required memory is based on the layer that requires the most memory, as described in section 3.2.
Multiplying the tileheight of equation 3.1 with the kernelwidth gives the requiredv size of the input-tile: $(((n-1) \times l_{ss} + k_h) \times k_w)$. The weight values of the kernels should also be kept in local memory. The size of the weight values of a single featuremap is 2 bytes for the bias value, together with all the kernel values. The values of a single kernel consists of $k_h \times k_w \times 2$ bytes, multiplying this with the number of connections of this featuremap fm_{conn} results in the following size for the weight values: $\sum_{fm \in l} 2 + (fm_{conn} \times k_h \times k_w \times 2)$. Combining this results in equation 5.6.

$$\max_{l \in L} \left((((n-1) \times l_{ss} + k_h) \times k_w) + \sum_{fm \in l} 2 + (fm_{conn} \times k_h \times k_w \times 2) \right) \quad (3.2)$$

Using equation 5.6 for the required local memory size on both the example networks with different n results in the plots as shown in figure 10. Most of the data of the Speed-Sign Recognition Network comes from the weight data while most of the data from the Face Detection Network comes from the inputtile.
Even with large n, every layer of both networks can be completely tiled with a local memory of 50 kilobyte. This means that all output featuremaps can be calculated with the tile. This is a feasible memory size for an embedded environment.
To calculate the number of external data requests to fetch input data can be calculated. The number of times the tile shifts down is equal to $\lceil l_{height}/n \rceil$. For each shift down, the number of bytes required is equal to $(l_{width} - 1) * l_{ss} + (l_{kw})$. For the total network, the number of external data requests is given by equation 3.3

$$\sum_{l \in L} (\lceil l_{height}/n \rceil \times ((n-1) \times l_{ss} + k_h) \times ((l_{width} - 1) \times l_{ss} + l_{kw})) \quad (3.3)$$



(a) Speed-Sign Recognition Network (b) Face Detection Network

Figure 10: Memory footprint for both networks.

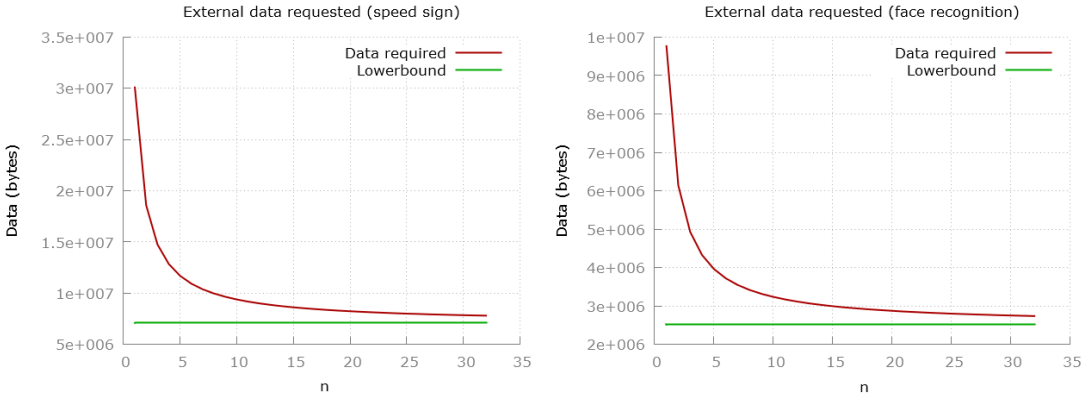


Figure 11: Tile height and external data requested, the lowerbound is calculated with infinite n

Plotting equation 3.3 for both networks is shown in figure 11. As expected, increasing n will decrease the external memory requests. But the cost of the memory footprint will make the reduction of accesses neglectable.

3.4 Parallelism

Due to the data dependencies between two consecutive layers of the network, it is hard to exploit parallelism in that case. There is no data dependency between featuremaps within a layer.
Calculating a single pixel can require multiple convolutions on multiple featuremaps. All these convolutions can be calculated in parallel, and then aggregated. Both the massive parallel coprocessor of NAC Laboratories America [11] and the NeuFlow [13] exploit this parallelism.
A kernel-size independent solution is to let a single PE sequential work on pixels. During every cycle, each PE will execute the same Multiply Accumulate (MACC) operation $y = y + w * x$ of the convolution(s).
The way of computing has two advantages:

- support for a range of kernel sizes with good utilisation

- multiple PEs can share the weight (w) values

3.5 Limitations

When the complete input tile combined with the weight values (the memory footprint as shown in figure 10) of a certain layer does not fit into local memory, the network can not currently be solved with this architecture. In section 8.2 I purpose a solution for this problem by adding an extra hardware piece in the memory controller. Then clusters can work on partial answers, storing partial answers in the off-chip memory.

4 Stream oriented SIMD architecture

In the previous section, the tiling order is determined. This section outlines the architecture that supports this order, while keeping the constraints in mind. Each section will describe a different part of the architecture. A graphical overview of the architecture is shown in figure 12 together with the corresponding section numbers. The presented architecture has MACC-units, called PEs divided into clusters, where each cluster consisting of a local memory, a number of PEs.

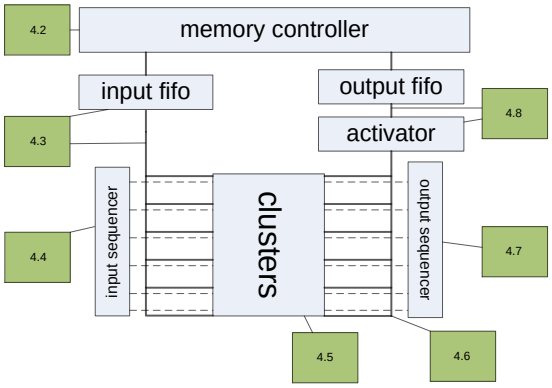


Figure 12: Overview of architecture

The memory controller puts data with a deterministic order in the input fifo. This data is put on the bus and the input sequencer routes the data to the correct clusters. Clusters will calculate pixels and put these on the bus. To ensure the deterministic order, the output sequencer determines which cluster can write on the bus. A single activator is used to do the non-linear activation function on the values.

4.1 Deterministic order

Calculating the result of a CNN is completely deterministic; the input does not change anything about the order and dependencies of the computations. This fact is used to determine an offline calculated and optimized schedule for the complete network. When fixing the order in which data comes from the memory controller into the accelerator, there is no need to check what kind of data it is. The input sequencer it knows the position in the offline calculate schedule and therefore knows where the data needs to go. The order of the output of the accelerator is also fixed. In that way all the memory locations can be calculated and optimized offline, reducing hardware and energy requirements.

4.2 Memory Controller

The memory controller should maximize the external bandwidth by aggregating data requests. The complete data schedule is known, so the memory controller

should exploit this as much as possible to maximize the read and write bandwidth.

4.3 Shared bus

To efficiently communicate data to the clusters, a bus is used that is connected to all clusters. With most connection schemes, input featuremaps are used in multiple clusters. Broadcasting this data over a bus is therefore usefull, because it reduces the number of reads and writes.
The bus is also used to communicate individual cluster information such as control data and weight values. The individual cluster information of all the layers of the network is on no account more than 0.5% of all the data that need to be transferred to the clusters in the two example networks. Therefore it is unnecessary to create an individual data connection to every cluster.
The communication between the shared bus and the memory controller is done via a fifo. In this fifo it is possible to buffer input data from the memory controller.

4.4 Input sequencer

The order that data appears on the bus is determined offline by the compiler. Since clusters do not always need the same input data, a data router is required to offer the data only to the clusters that need this data.
The input sequencer has a memory with an offline calculated sequence file. The file contains a sequence of when which cluster needs to read from the input bus. Further details about the sequence file can be found in section 5.2.2.

4.5 Clusters

It is not feasible to make a single local storage and connect dozens of PEs with it. A more hierarchical model is needed to make this possible. PEs will be grouped in clusters with a smaller local memory. Each cluster works on certain parts of the network, so not all the weight data and possibly not all the input tiles are needed in the local memory. Each cluster will calculate a number of output featuremaps, requiring a number of input featuremaps. Which convolutions needs to be calculated for every output featuremap is described in a connection matrix within the cluster. This connection matrix describes the connection scheme of the part of the network that the cluster is working on.
The architecture consists of multiple clusters where each cluster consist of a local memory, a number of PEs and control hardware.

4.5.1 Cluster Memory

The memory of the cluster is used for three things:

- Connection Matrix
- Weight values
- Input featuremaps

At the beginning of calculating a layer, the connection matrix and weight values are loaded into the cluster. This data does not change throughout the complete layer. The input featuremaps part of the memory is not constant throughout the calculation of a layer. Figure 13 shows the pattern of the first three rounds. The part for the input tiles of the memory is used as a circular buffer.

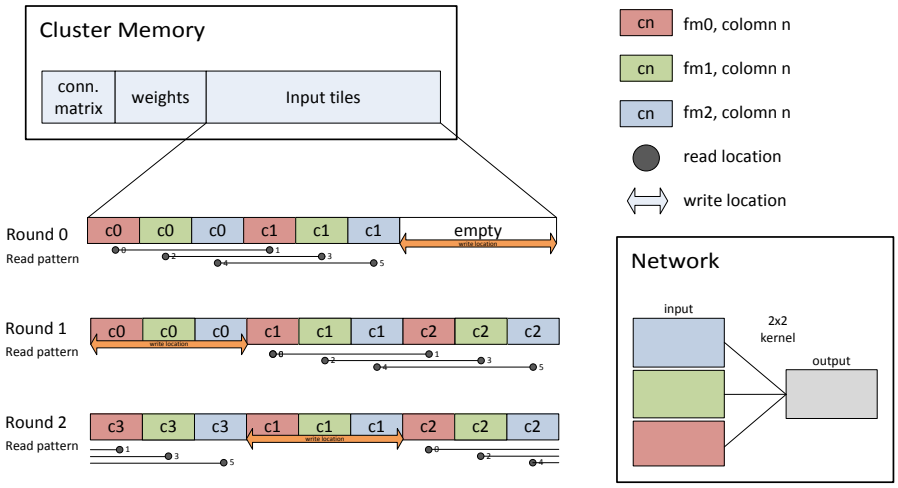


Figure 13: Read pattern of the cluster

The example network in figure 13 shows a small network where 3 featuremaps are connected to a single featuremap with a 2×2 kernel. To compute output pixels, it is required to have two consecutive colomns of the three input featuremaps. While these pixels are being calculated, the next colomns of the featuremaps can be written.
This parallel reading and writing reduces the calculation time of pixels of an output featuremap. In stead of waiting *calculation time + write time*, this is reduced to *min(calculation time, write time)*, with only a small increase in memory requirement.

4.5.2 Registers

Local memories are expensive in area and energy, therefore the number of local memories should be small. Peemen proposed an architecture[18] that needs 6 local memories to supply data to 8 PEs, 4 memories for the input values and 2 for weight values. That system has two disadvantages. First of all, there is a memory limit on both weight values and input values that reduces the flexibility. Secondly, more local memories imply more energy and area usage.
I will propose a system that will make it is possible to supply data to multiple PEs with a single local memory. This solution does not result in a 100% utilisation of the PEs, but it will reduce the energy usage and increase flexibility. The local memory is used for both input values and for weight values. The flexibility of the architecture will be increased by putting the data in the same memory and by not placing a limit on only weight values or input values.
Since one port needs to be used for writing, there is only one read port, therefore a shift register for input values is used to increase memory bandwidth. This is because values read from the memory can be used multiple times.
Figure 14 depicts a module with 4 PEs that supports subsampling up to factor 2. The pixel shift registers are numbers from 0 to 11. There are two weight

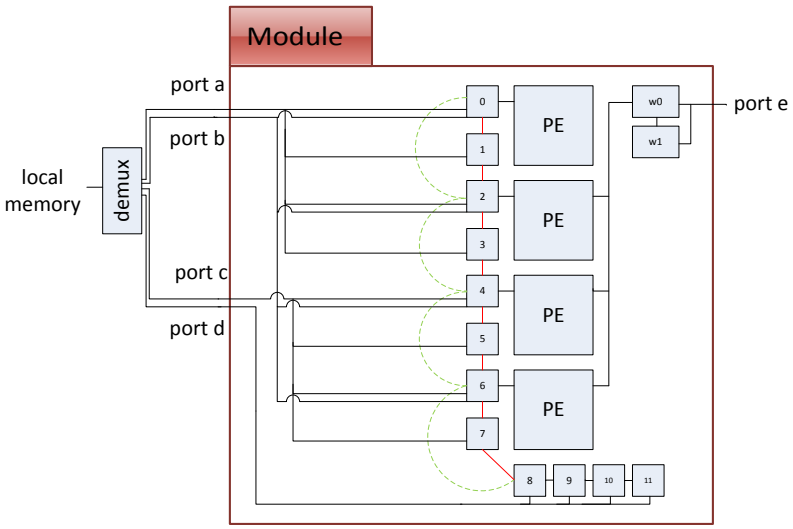


Figure 14: Module hierarchy

cycle	port a	port b	port c	port d	port e	shift/calc enable
0	x					
1			x			
2					x	
3				x		x
4						x
5					x	x
6						x
7					x	x

Table 1: Port usage of the module for the first 5 calculations

shift registers $w0$ and $w1$ for storing the weight values. To start calculations, data is loaded into the shift registers 0 to 7, and extra data is added in registers 8 to 11. When no subsampling is used, after every MACC-operation the pixel data is shifted using the green lines, while with a subsampling factor of 2 the red lines are used.
Assume calculating a 5×5 convolution on a single input featuremap with a subsampling factor of 2. Table 1 shows that in 8 cycles, 5 computations are performed, achieving a utilisation of 63% for the first column with only a single memory. During each cycle, the following pattern is used: *load data, (compute data, shift data)*. When using double buffering, in cycle 4 and 6 you can already load data on port a and port b, achieving 83% utilisation rate for the next colomn. For the complete 5×5 convolution, this will cost $8+(4*6)=32$ cycles, resulting in an overall utilisation of 78%.
Table 2 shows that for larger kernel-sizes, higher utilisation is obtained. Smaller

kernel	no subsampling	subsampling
1×1	50%	not applicable
2×2	50%	44%
4×4	88%	72%
5×5	92%	78%

Table 2: PE utilisation for different kernel sizes

kernel-sizes are mostly memory bound and not computation bound, thus the bottleneck will not be within the clusters.

This model is implemented using Verilog HDL with 8 PEs. At 100MHz, simulated with Xilinx XPower, the power distribution is shown in table 3. In these results there is a factor 3.5 reduction in power using shift registers. With only a small reduction in performance, a substantial energy reduction is achieved, in addition it increases the flexibility of the memory.

Solution	Block RAM	Shift registers	Total
Shift registers	0.00167 W	0.00121 W	0.00288 W
Block RAMs	0.01002 W	-	0.01002 W

Table 3: Power division in different solutions

4.5.3 Convolution operations

Each PE within a cluster works on convolution operations until the pixel of the featuremap is ready to go trough the activation function. The problem with this approach is that all the corresponding featuremaps needed for the calculation of that pixel have to be in local memory. This approach has two advantages: finished pixels go trough an activation function, reducing the size of the data with a factor 4, and in addition intermediate values do not have to be stored in the memory.
To compute a pixel, a PE requires input values and weight values. All the PEs in a cluster calculate the same part of the convolution in parallel, thus sharing the weight values. Sharing the weight values will reduce the bandwidth requirement on the local memory.

4.6 Output bus

The clusters are also connected to the same output bus. At any time, only one cluster will write results on this bus. In this way, it can guarantee a deterministic output order. Subsequently, only a single module can be used for doing the activation function.

4.7 Output sequencer

The output sequencer is the bus-controller for the output bus. Similarly to the input-sequencer, the output sequencer has an offline calculated sequence file that contains a sequence of when which cluster may write on the bus.

4.8 Activator

This activator uses a Look-up Tables (LUTs) to perform the non-linear activation function on the given input values. The LUT does not change throughout the calculations of the network. This single activator is sufficient since the number of activations required in a network is two or three orders of magnitude lower than the number of calculations. Furthermore, the activator sends data directly, via a fifo, to the memory controller. The activator can not work faster than the available bandwidth to the memory controller.

5 Optimizing Toolflow

Running a CNN on this new architecture has a lot of free parameters. Such as the question which cluster is to calculate which output featuremap or what order they are going to calculate them in. Furthermore, sequence files are required for the different components. Finding good schedules and creating the sequence files is manually impossible, so this requires a toolflow. The toolflow consists of two main parts: the scheduler determines which cluster is going to calculate which output featuremap, and determines the output order. The compiler uses the result for the scheduler to generate all the sequence files and datastreams needed for the different hardware elements.

5.1 Scheduling

To demonstrate the problems with scheduling, consider scheduling the second layer of the Speed Sign Detected network as depicted in figure 15 and table 4. A mapping between featuremaps and clusters must be determined.

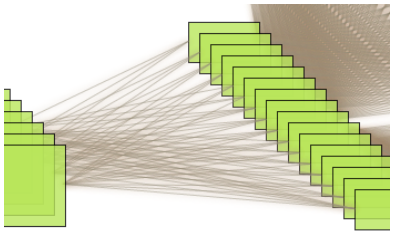


Figure 15: Connection scheme of the second layer of the Speed Sign Detection Network of section 2.5.1

output FMs	**input FMs**
fm_0	0,1,2
fm_1	1,2,3
fm_2	2,3,4
fm_3	3,4,5
fm_4	0,4,5
fm_5	0,1,5
fm_6	0,1,2,3
fm_7	1,2,3,4
fm_8	2,3,4,5
fm_9	0,3,4,5
fm_{10}	0,1,4,5
fm_{11}	0,1,2,5
fm_{12}	0,1,3,4
fm_{13}	1,2,4,5
fm_{14}	0,2,3,5
fm_{15}	0,1,2,3,4,5
total	0,1,2,3,4,5

Table 4: Input FM required for each output FM

A mapping is valid when the total data requirement of the featuremaps mapped to a certain cluster fits into local memory.
Assuming that a cluster i will work on calculating fm_0, fm_5 and fm_{11}. The required input featuremaps are described as a set of sets:
$S_i = \{\{0,1,2\},\{0,1,5\},\{0,1,2,5\}\}$. For each ouput featuremap a bias value may be required. The following equation shows the total amount of weight data:

$$\text{biasdata} = h * |S_i| \tag{5.1}$$

Where h is the amount of bytes of a bias value.

Each output featuremap requires the number of input featuremaps on kernel weights. The total weight data is given in the next equation:

$$\text{weightdata} = i * \sum_{e \in S_i} |e| \tag{5.2}$$

Where i is the amount of bytes of a weights values.
Since tiles of the input featuremaps can be used for all the featuremaps, it only requires the number of unique input featuremaps. The total input data is given in the next equation:

$$\text{inputdata} = j * |\bigcup_{e \in S_i} e| \tag{5.3}$$

Where j is the amount of bytes of a featuremap tile.
The size of the connection matrix is the number of output featuremaps $\times$ the number of input featuremaps. For a certain assignment, this is equal to:

$$\text{connection matrix} = k * |S_i| * |\bigcup_{e \in S_i} e| \tag{5.4}$$

Where k is the size of an entry in the connection matrix in bytes.
Combining these requirements result into the following equation for a valid mapping:

$$\text{valid mapping} = (\text{biasdata+weightdata+inputdata+connection matrix}) \leq mem \tag{5.5}$$

Where mem is the size of the cluster memory
A mapping of S is a when S is partioned into m sets $S_1, S_2, ..., S_m$, where m is the number of clusters. For this partition, the following equation must hold:

$$\forall_{1 \leq x \leq n} \left(\left(h * |S_x| + i * \sum_{e \in S_x} |e| + j * |\bigcup_{e \in S_x} e| + k * |S_x| * |\bigcup_{e \in S_x} e| \right) \leq mem \right) \tag{5.6}$$

If such a division exists, the current layer can be solved with the memory size m.

5.1.1 Complexity of scheduling

Consider an algorithm *CLUSTER-DIVISION(S,h,i,j,k,mem,n)* that returns *true* iff there exists a partition into $S_1, S_2, .., S_n$ such that equation 5.6 holds for this partition.

5.1.2 NP-completeness

To show NP-completeness of the CLUSTER-DIVISION problem, it is required to show that CLUSTER-DIVISION is at least as hard (notation: $\leq_p$) as a known NP-complete problem. 3-PARTITION is a known NP-complete problem. [19] The following theorem shows that CLUSTER-DIVISION is at least as hard as 3-PARTITION:

Theorem 5.1 *3-PARTITION* $\leq_p$ *CLUSTER-DIVISION*

Proof Consider the instance I for *3-PARTITION* consisting of a set A of $3m$ elements $(a_1, a_2, .., a_{3m})$, a bound $B \in \mathbb{Z}^+$, and a size $s(a) \in \mathbb{Z}^+$ for each $a \in A$ such that $l(a)$ is bound by a polynomial in m and such that $\sum_{a \in A} l(a) = mB$ [19]. This bound on $l(a)$ makes *3-PARTITION* still *NP-hard* since *3-PARTITION* is *strongly NP-complete.*
Define the function f that builds a set $S = s_1, \ldots, s_{3n}$ such that $\forall_{1 \leq i \leq 3m} (|s_i| = l(a_i))$. f can be executed in polynominal time because it holds that $\forall_{s \in S}$ s is bounded by a polynomial in m.
Now call *CLUSTER-DIVISION(S=S,h=0,i=1,j=0,k=0,mem=B,n=m).* Using lemma 5.2, this results that *3-PARTITION* is at least as hard as *CLUSTER-DIVISION.* □

Lemma 5.2 *F can be 3-partitioned $\iff$ $f(F)$ can be cluster divided*

Proof $\Rightarrow$ Assume instance $\langle A, B, s \rangle$ can be 3-partitioned. So there exists an assignment such that A can be partitioned into m disjoint sets $A_1, A_2, ...A_m$ such that $\forall_{1 \leq i \leq m} \sum_{a \in A_i} l(a) = B$ S can be partitioned into disjoined sets $S_1, S_2, ..., S_m$ with $\forall_{1 \leq i \leq m} (\forall_{s' \in S_i} (|s'| \in A_i))$. This is possible since $\forall_{1 \leq i \leq 3m} (|s_i| = l(a_i))$.

$\Leftarrow$ Assume instance $\langle S, h = 0, i = 1, j = 0, mem = B, n = m \rangle$ can be cluster-divided into disjoined sets $S_1, S_2, ..., S_m$ such that $\forall_{1 \leq i \leq m} \sum_{e \in |S_i|} \leq B$. Since $\forall_{1 \leq i \leq 3m} (|s_i| = l(a_i))$ and $\sum_{a \in A} l(a) = mB$, $\sum_{s \in S} mB$, dividing S into m disjoined sets can only mean that $\forall_{1 \leq i \leq m} \sum_{e \in S_i} |e| = B$. If there would exists an $1 \leq i \leq m$ such that $\sum_{e \in S_i} |e| < B$, by the Pigeonhole principle, there must also be an $1 \leq i \leq m$ such that $\sum_{e \in S_i} |e| > B$ and that would mean that S is not cluster-divided. Since $\forall_{1 \leq i \leq 3m} (|s_i| = l(a_i))$, we could divide A into disjoined sets $A_1, A_2, ...A_m$ with $\forall_{1 \leq i \leq m} \forall_{e \in A_i} \exists_{e' \in S_i} |e'| = e$. And thus it can be 3-PARTITIONED.

Theorem 5.3 *CLUSTER-DIVISION has a polynomial-time verification algorithm*

Proof CLUSTER-DIVISION verifier V($\langle S, h, i, j, mem \rangle, \langle S' \rangle$)
The verifier only accepts the instance if all the following criteria are true:

- $\forall_{s \in S} (\exists_{s' \in S'} (s \in s'))$
- $\forall_{s' \in S'} (\forall_{e' \in s'} (e' \in S))$
- $|S| = \sum_{s' \in S'} |s'|$
- equation 5.6 holds for S'

Since V can be executed in polynomial time, *CLUSTER-DIVISION* has a polynomial time verification algorithm. □

Theorem 5.4 *CLUSTER-DIVISION is NP-complete*

Proof Combining lemma 5.1 and lemma 5.3, result that *CLUSTER-DIVISION* is NP-complete □

5.1.3 Polynomial time greedy scheduling

In addition to find a solution that meets the memory constraint, a solution that also minimize the execution time is required. Since finding a solution is already *NP-complete*, finding the best solution is thus also NP-complete. So unless *P=NP* it is not possible to find an algorithm that runs in polynomial time.
It is possible that certain layers consists of a 100 featuremaps. If the toolflow will use an algorithm that is exponential in this parameters, it can take months before it is possible to program this architecture.
As a consequence of this, finding an approximation algorithm is the best solution. If this algorithm does not find a solution where the memory constraint is met, you can execute a search on the full statespace, change the network, or need to use more local memory.

5.1.4 Cluster division

Computing a featuremap takes a certain amount of cycles. This workload should be evenly defined over the clusters, in such a way the makespan of all the clusters is minimized.
In the best case, all the jobs of featuremaps can be spread evenly over the clusters. The execution time can not be lower than the largest job, since this jobs needs to be scheduled at a cluster. These results are combined in the following equation for the lowerbound.

$$\max(\frac{1}{|FMs|}\sum_{fm \in FMs} fm_{size},\ max_{fm \in FMs} fm_{size}) \tag{5.7}$$

Where $|FMs|$ is the number of featuremaps and fm_{size} is the number of cycles it takes to compute the result of featuremap fm.
To combine these two dimensions of memory constraint and reducing the makespan, the following score for cluster i is used:

$$\beta_i = \frac{M(i)}{memlimit} + \frac{T(i)}{lowerbound} \tag{5.8}$$

Where $M(i)$ is the current memory requirement and $T(i)$ is the current makespan. Beta needs to be reduced for all clusters. When there is a very large memory limit, β is more dependant on the makespan and therefore will be optimised for makespan. It would be possible to, in stead of add, multiply the two different components of β. This is not considered since the current algorithm gave very good performance, as will be shown in section 5.1.6

Algorithm *Greedy-ClusterDivison*(FMs, n)
$\triangleright$ sort FMs on decreasing execution time
$\triangleright$ determine *lowerbound*
Initialize $M(i) \leftarrow 0$, $T(i) \leftarrow 0$ for $1 \leq i \leq n$
for all $fm \in FMs$ **do**
  for $i = 1 \rightarrow n$ **do**
    determine $M'(i)$ and $T'(i)$ if fm would be added to cluster i
    if $M'(i) \leq$ memlimit **then**
      $\beta_i = \frac{M(i)}{memlimit} + \frac{T(i)}{lowerbound}$
    else
      $\beta_i = \infty$
    end if
  end for
  determine k such that $\beta_k = \min_{1 \leq i \leq n} \beta_i$
  if $\beta_k = \infty$ **then**
    return no_solution
  else
    add fm to cluster k, calculate $T(k) \leftarrow T'(k)$ and $M(k) \leftarrow M'(k)$.
  end if
end for

The algorithm sorts all the featuremap on decreasing execution time. This is because on the end of the algorithm, only small changes are allowed. The featuremap are considered one by one and added to the cluster where *beta* is the lowest, thus decreasing the makespan and the memory usage.

Theorem 5.5 *Greedy-ClusterDivison runs in $O(m \log m + mnl)$ with l the number of input featuremaps, m the number of output featuremaps and n the number of clusters.*

Proof Sorting m featuremaps can be done in $O(m \log(m))$ using quick-sort. To determine the lowerbound, it is necessary to find the largest featuremap and to aggregate all the feautremaps. This is done in $O(m)$. For each output featuremap fm, and for each cluster i, the following is done: determine $M(i)$ and $T(i)$ if fm would be added to cluster i. Determining $M'(i)$ is checking which input featuremaps are not already used in cluster i. Using a boolean array of size l to keep track of these featuremaps, this operation can be done in $O(l)$. Determining $T'(i)$ is done by adding fm_{size} to $T(i)$, which takes $O(1)$. Determening β_i is done in $O(1)$. To keep track of the local minimum, determening k is done inside the loop with an $O(1)$ operation. In total the runningtime of Greedy-ClusterDivison is $O(m \log m + mnl)$ $\square$

Theorem 5.6 *On a fully connected layer, Greedy-ClusterDivison produces the optimal solution for memory usage and makespan*

Proof When scheduling m featuremaps on n clusters by using the pidgeon hole principle, there is a cluster that has at least $\lceil n/m \rceil$ jobs. If a cluster would compute $> \lceil n/m \rceil$ featuremaps, the makespan of this solution would be larger than if all the clusters calculate $\leq \lceil n/m \rceil$ featuremaps. So the optimal solution is when all clusters calculate $\leq \lceil n/m \rceil$ featuremaps.
On a fully connected layer, all the featuremaps have equal execution time and impose the same memory increase when they are added to a cluster. The greedy

algorithm will add the featuremaps where β_i remains the lowest, which is equal to the cluster with the lowest number of featuremaps. This results that no cluster will calculate more than $\leq \lceil n/m \rceil$ and thus results in the lowest memory and lowest makespan. □

5.1.5 Output bus

From the given cluster divison, the sequence of when clusters can write on the bus has to be determined. Recall that this sequence is required to ensure deterministic output order.
Given a certain set of jobs for each cluster, the order in which clusters need the bus needs to be determined. In figure 16 there is an execution timeline of 3 rounds (green,white,green) of 9 jobs, with red meaning writing on the bus.

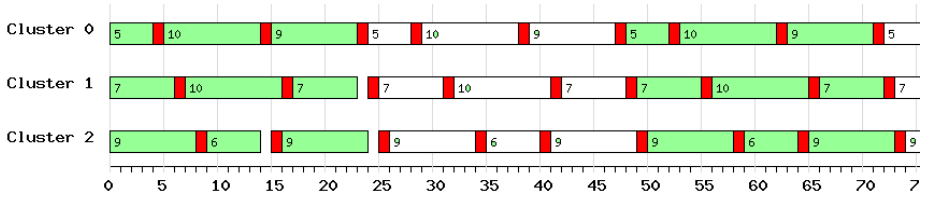


Figure 16: Execution timeline of 3 rounds of 9 jobs

A greedy approach to reduce idle time is determines the first come first serve order. This order can be computed with the following algorithm:

Algorithm *Create-Bus-Order*(C)
Initialize empty list L
for all $c \in C$ **do**
  $t \leftarrow 0$
  for all $fm \in c$ **do**
    $t \leftarrow t + fm_{time}$
    $L \leftarrow L \cup (t, c)$
  end for
end for
▷ sort L increasing
return second(L)

Theorem 5.7 *The running time of Create-Bus-Order is* $O(m \log(m))$*, with* m *the number of output featuremaps*

Proof Every fm is added to a single cluster. For every fm there is a calculation that can be done on $O(1)$, and adding an item to a list that can be done in $O(1)$. The size of list L is equal to $\Omega(m)$. Sorting L can be done in $O(m \log(m))$ using quick-sort.

In the example networks, the output bus does not impose a sizeable problem: the computation time is much larger than the required bus time and this results in a high probability that the bus can be directly used.

speed-sign	round-robin	greedy solution	brute force
layer 1	50	50	50
layer 2	500	450	450
layer 3	-	2400	2400
layer 4	800	800	800
face	round-robin	greedy solution	brute force
layer 1	50	50	50
layer 2	84	84	84
layer 3	-	80	80
layer 4	75	75	75

Table 5: Execution time in cycles of different scheduling algorithms, with the bus, with 8 PEs, 8 Clusters, and miminum memory size

5.1.6 Algorithm results

Results from comparing the polynominal time algorithm to two different algorithm is shown in table 5. The round-robin algorithm schedules the first featuremap on the first cluster, the second on the second, etc. The brute force solution tries all the possibilities.

The greedy solution works better than the round-robin schedule: the round-robin schedule does not find a schedule for layer 3 of both networks. This is because the round robin schedule does not take the memory constraint into account. In layer 3 of Speed-Sign Recognition Network, the mapping generated by round-robin will require that all clusters require all the input featuremaps, while the greedy solution maps it in such a way that the clusters require only half the input featuremaps.

Comparing with the brute-force solution, it shows that the greedy algorithm finds the best solution in all cases for the example network. That the algorithm finds *a* solution for all layers with the minimum memory size is because the minimum memory size is due to the fully connected layers. Using theorem 5.6, these layers can be solved with the minimum memory size. Other layers require less memory than these layers, making it easier to find a solution that fits.

The conclusion that can be drawn from these results is that the greedy solution is better than a round-robin schedule and it produces results comparable with a brute-force approach.

5.2 Compiler

This architecture should work without a general purpose processing unit. This allows for the energy requirement of the whole system to be reduced. To support this, control data is needed for different components of the architecture.
Depicted in figure 17 is a graphical overview of all the data that the compiler produces.

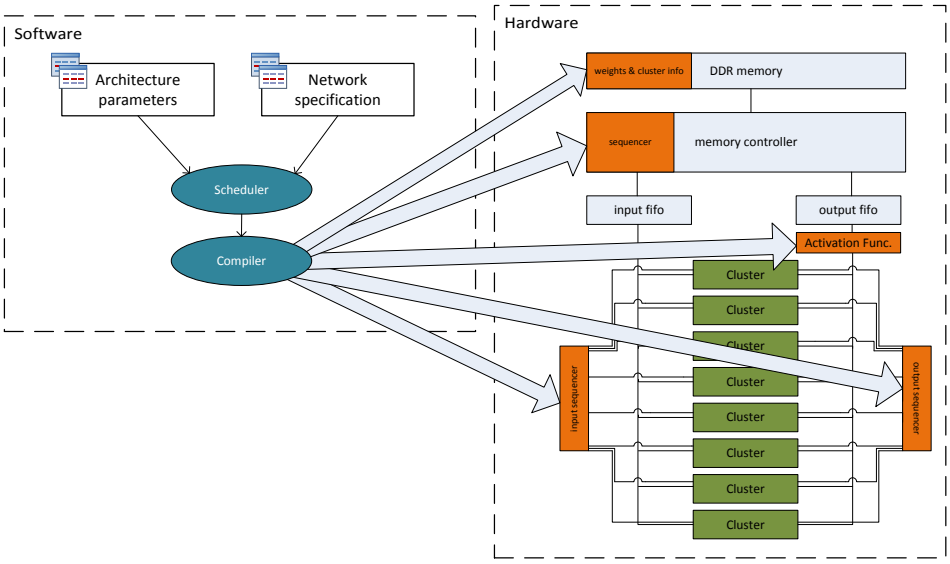


Figure 17: Overview of the toolflow of the compiler

5.2.1 Memory controller sequencer

The sequence files for the memory controller consist of a list of precomputed memory addresses where data needs to be read from and stored. Research on optimising the memory controller is done independently from this report.

5.2.2 Input sequencer

The input sequencer has the task to route the input to the correct cluster(s). Each layer starts with data for all the clusters (kernel size, input size, etc.), followed by individual cluster data (weight data, etc.). After this start-up phase, colomns of featuremaps are presented on the bus. The order in which the data data appears on the bus is deterministic and therefore the input sequencer is also deterministic.
A graphical overview of the content of the sequence file is depicted in figure 18. Every layer is converted to two sequences: one sequence for the startup phase and a sequence for the input data. A sequence consists of several sequence elements: a sequence element consists of a repeat count (how many data elements use this sequence element) and a cluster activation (which clusters need this data). A complete sequence can be repeated multiple times.

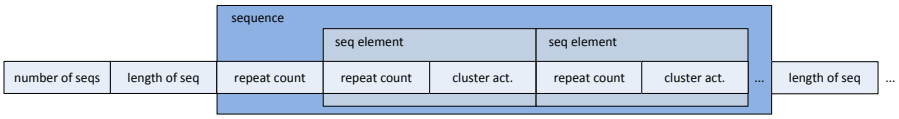


Figure 18: Content of the sequence file

Lemma 5.8 *The size of the input data for the input-sequencer is O(layers*(clusters* (clusters + input featuremaps))*

Proof Each layer has two sequences. The first sequence consists of 1 sequence element for every cluster. The second sequence consist of 1 sequence element for every input feature map. A sequence element consist of a constant size number and `clusters` bits. In total, the input data is equal to O(layers*(clusters*clusters + clusters*input featuremaps)) □

5.2.3 Output sequencer

The output sequencer is the bus controller for the output bus. It tells which cluster can write on the bus. This order is determined by the scheduler. The output sequencer uses the same sequence file structure as the Input sequencer.

Lemma 5.9 *The size of the input data for the output-sequencer is O(layers * (clusters * output featuremaps))*

Proof For every layer the output sequencer has an order in which clusters can write on the bus. Since the output order is fixed, the sequence can be repeated multiple times. A single sequence spans all the output featuremaps. And thus results in a filesize of O(layers * (clusters * output feauturemaps)). □

5.2.4 Weights and cluster information

All the clusters need the following shared information:

- input height
- input width
- subsampling factor
- kernel height
- kernel width
- colomn height (this is for optimization, can be calculated from previous values)

An individual cluster requires the following unique parameters:

- a connection matrix
- number of input featuremaps
- number of output featuremaps
- size of the weight data
- all the weight values

The compiler generates this data in a single stream of data for every layer. By using the startup sequence of the input sequencer, this single stream of data is routed so that every cluster gets the correct individual cluster information.

5.2.5 Activation Function

The compiler will make a static Look-Up Table for calculating the activation function. Quantisation of the activation function is used to generate the LUT for the activator.

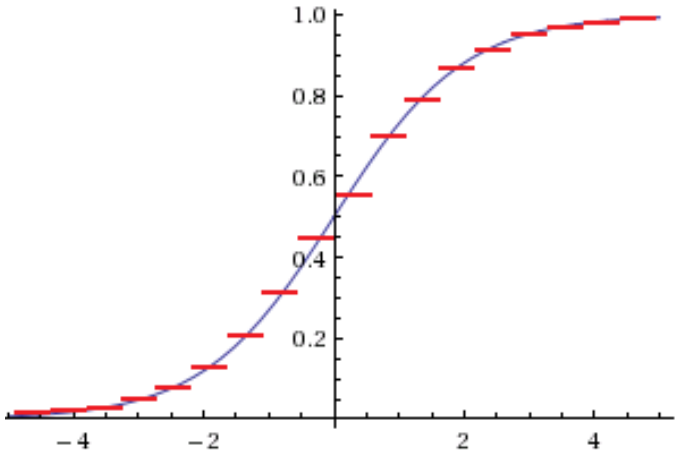


Figure 19: Example of an activation function $\phi(x) = 1/(e^{-x} + 1)$ with LUT-values in shown red

6 Experimental results

To understand performance characteristics, it is required to do performance analysis. To do this analysis of the new architecture, a cycle-based model is used. This model is used to determine bandwidth requirement and to measure throughput performance. The architecture is then compared with recent CNN accelerators known in literature and with a consumer CPU and GPU.

6.1 Experimental setup

The number of cycles a certain convolution with a kernel size $l_{kw} * l_{kh}$, subsampling factor l_{ss} and n PEs, and a memory where it is possible to read ram_{width} bytes per cycle is modelled with the following equations:
Before the module can start on a new column, enough data needs to be loaded into the shift registers, together with 1 weight value.
The required pixels to load is equal to $n * l_{ss}$, since we have n PEs working on the pixels with a subsample factor l_{ss}. The number of clockcycles it takes to load this amount of data is equal to:

$$startUp_0 = 1 + \lceil (n * l_{ss}) / bram_{width} \rceil \tag{6.1}$$

Now data is loaded into the shift registers, the computations can start. To compute the rest of the colomn, an additional $l_{kh} - 1$ bytes are needed to load into the pixel shift registers. In a single clockcycle, it already loaded $\frac{bram_{width}}{2}$ weight values (since a weight value is two bytes). So still $(l_{kh} - \frac{bram_{width}}{2}) * 2)$ bytes of weight values is required for the complete colomn. This results in the following equation for the number of cycles of computing a colomn.

$$colTime = \lceil (l_{kh} - 1) / ram_{width} \rceil + \lceil (l_{kh} - \frac{bram_{width}}{2}) * 2) / bram_{width} \rceil \tag{6.2}$$

With use of double buffering, idle cycles of the previous round can be used to load data for the next column. So the number of startUp cycles it takes for the next colomn is given by:

$$startUp_n = max(0, (startUp - (l_{kh})) \tag{6.3}$$

Combining all the previous results, results in the following formula that computes the number of cycles is take to do a convolution on a module:

$$startUp_0 + ((l_{kw} - 1) * startUp_n) + (l_{kw} * max(colTime, l_{kh})) \tag{6.4}$$

First of all it is determined how many cycles it takes before every cluster can start with the convolution, based on the required individual cluster information. The model assumes that clusters can buffer the number of values from the connection matrix that can be read in a single clock cycle.
Then the number of cycles it takes to calculate all the convolutions and put the results on the bus is determined by simulation of a timeline, assuming a MACC operation can be done in a single clockcycle. This process is repeated for all the layers. The model assumes that the memory controller can read and write data from the fifo's every clockcycle.

6.2 Scalability

In this section the behaviour of the architecture is analysed with different memory bandwidths and with different configurations. This will show the characteristics of the architecture.

6.2.1 Memory bandwidth

To understand the memory bandwidth requirements of the new architecture, scalability with respect to the width of the input bus is tested. Latencies of the memory controller are neglected, every clock cycle new input data can be read from the input fifo. Using an architecture with 16 clusters with 8 PEs, the execution time of both example networks is given in figure 20.

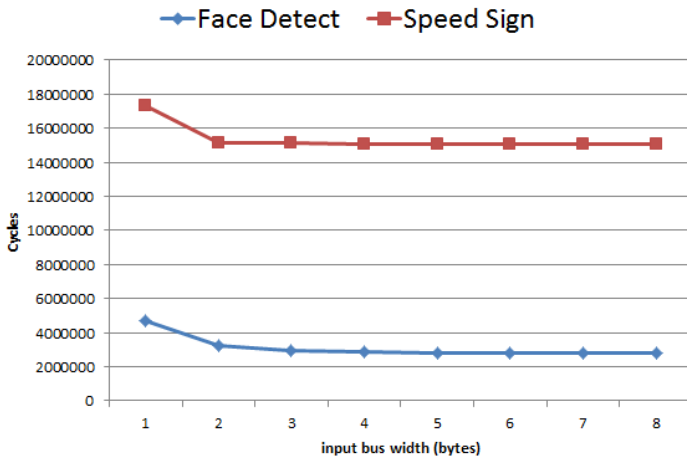
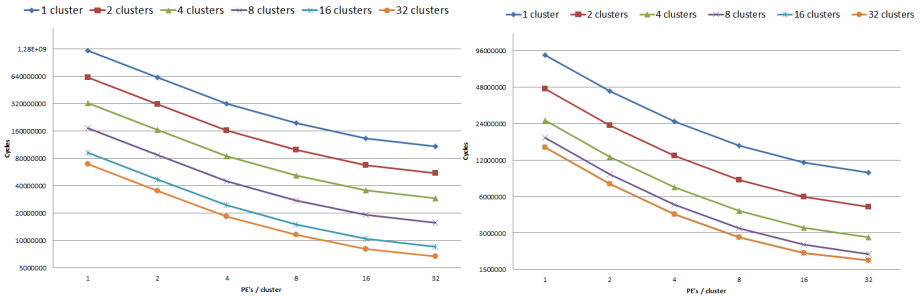


Figure 20: with 16 clusters with 8 PEs

The simulation shows that the Face Detection Network requires a higher memory bandwidth in comparison with the Speed-Sign Recognition Network. While the speed-sign network achieves maximum performance with an input bus width of 2 bytes, the face detect network requires an input bus width of at least 4 bytes to achieve maximum performance. The required memory bandwidth is dependent on the number of clusters and PEs. With only 8 clusters of 2 PEs, a bus width of 1 byte is already sufficient to obtain maximum performance for the Face Detection Network.

At 100MHz an input bus width of 1 bytes results in a bandwidth of 100 MB/s. This is not exactly the same as the required bandwidth from the external memory: To route the date correctly to the clusters, data from at most a single feature map can be send. If the height of the column can not be exactly divided by the bus width, padding is used.
Without reuse of data, the Speed-Sign Recognition Network would require 52GB/frame and the Face Detection Network would require 2.7GB/frame. To be able to solve the Face Detection Network at the same speed, this would impose a memory bandwidth of 97.5GB/sec.



(a) Speed-Sign Recognition Network (b) Face Detection Network

Figure 21: Scalability results of both networks

Layer	16 clusters	32 clusters	Reduction
layer 1	6.628.072	6.628.072	0%
layer 2	9.776.265	9.776.265	0%
layer 3	45.516.888	27.322.827	40%
layer 4	13.00.789	13.009.789	0%

Table 6: Comparison of 16 and 32 clusters with a single PE per cluster

6.2.2 Computational performance

To test if the architecture scales well with more PEs and clusters simulations are performed. Numerous configurations of PEs, clusters and PEs per cluster are tested. Results from these test are shown in figure 21. In this test, the input bus was sufficiently large and the local memories have a width of 4 bytes.
From the results Speed-Sign Recognition Network in figure 21a show that the performance increment between 8 and 16 clusters is much larger than from 16 to 32 clusters. To determine the cause of this, the cycle-count is split op per layer in table 6. Only layer 3 has an advantage of increasing the number of clusters to 32 because layer 3 is the only layer with more than 16 featuremaps. This same problem occurs in the results of Face Detection Network in figure 21b. There is increase in performance going from 16 clusters to 32 clusters. Since a featuremaps can only be computed by a single cluster, with the 32 cluster configuration more than half of the clusters are idle. A solution to increase parallelism when there are more clusters than featuremaps is described in section 8.1.
Overall, increasing the number of clusters and the number of PEs per cluster, decreases the number of cycles. For the Speed-Sign Recognition Network going from a configuration with a single cluster with a single PE to a configuration with 4 clusters with 4 PEs increases the performance with a factor 14.5 with 16× the number of PEs. For the Face Detection Network this increase of PEs results in an increase in performance of factor 12. For more clusters and PEs the increment decreases, because of the limited featuremap parallelism and because it is harder to evenly divide the work over multiple clusters.
Increasing the number of PEs per cluster does not scale completely linear; the PEs use the same BRAM, which has a limited bandwidth.

6.3 Design Space Exploration

In figures 22 and 23 results are shown with sorted execution time with different configurations, all with a bus width of 4 bytes. The configurations where no other solution exists with less or equal number of PEs but with a lower execution time (so called Pareto points) are circled gray. These pareto points are depicted in table 7. An interesting trend in this data is that the pareto points of the Speed-Sign Recognition Network has more clusters, while the Face Detection Network pareto points have more PEs per cluster. This is because the Face Detection Network has less featuremap parallelism. The first layer consists of only 4 featuremaps. Increasing performance on this layer can only be achieved by adding more PEs per cluster when 4 clusters are used.

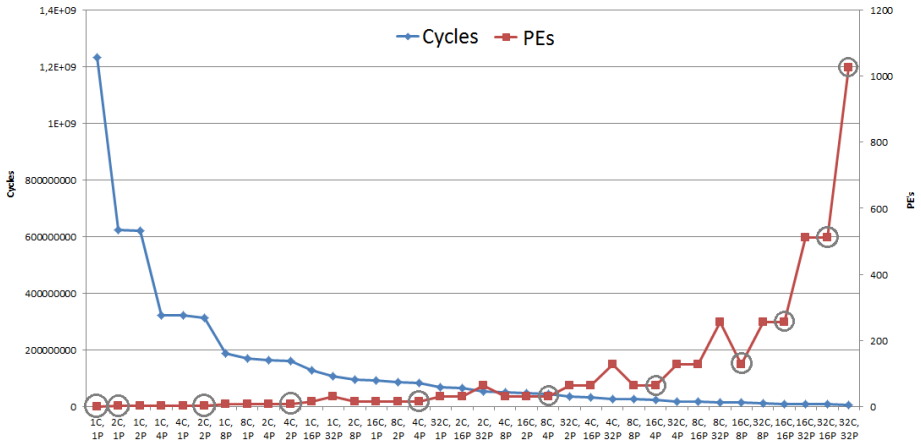


Figure 22: Different configurations for the speed sign network

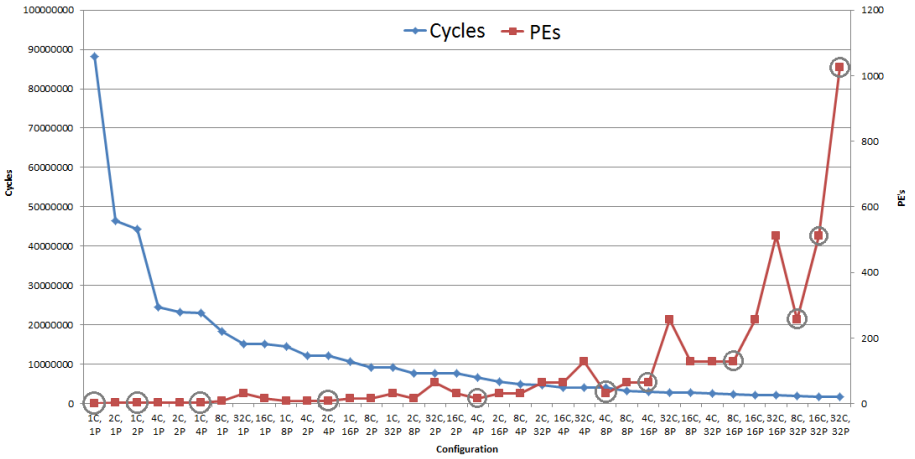


Figure 23: Different configurations for the face detection network

PEs	SS Config	Face Config
1	1C, 1P	1C, 1P
2	2C, 1P	1C, 2P
4	2C, 2P	1C, 4P
8	4C, 2P	2C, 4P
16	4C, 4P	4C, 4P
32	8C, 4P	4C, 8P
64	16C, 4P	4C, 16P
128	16C, 8P	8C, 16P
256	16C, 16P	8C, 32P
512	32C, 16P	16C, 32P
1024	32C, 32P	32C, 32P

Table 7: Parato points of different number of PEs

To show flexibility of the configurations, PE utilisation with multiple configurations is calculated for both example networks, this is depicted in figure 24. Results show that the utilisation for both network stay in the same order of magnitude, mostly differing less than a factor of 2. With less then 4 clusters, they defer less than 25%.
Overall, it is clear that the same configuration can be used for both networks, showing the flexibility of the architecture.

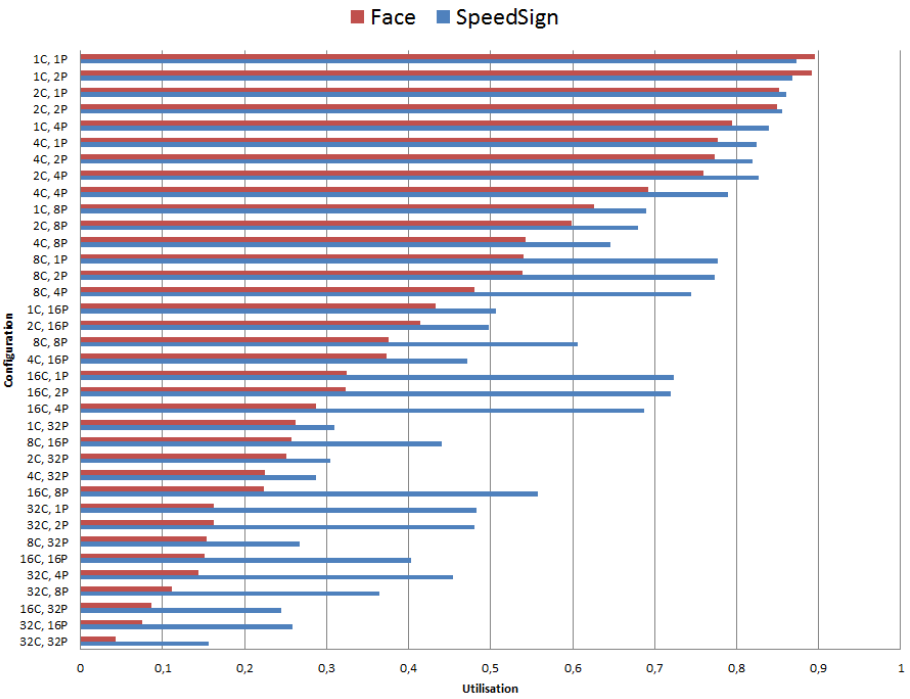


Figure 24: Utilisation with different configurations

6.4 Comparison with other CNN accelerators

There are two CNN accelerators that support multiple networks known in literature. It was not possible to obtain an implementation for both accelerators. To compare the performance of their implementation, the required bandwidth and number of computations is modelled.

6.4.1 NeuFlow

The New York University's Computational & Biological Learning Laboratory has developed NeuFlow: A Runtime Reconfigurable Dataflow Processor for Vision. [13].

The NeuFlow consists of a grid of processing tiles (PTs), a "Smart DMA" and a controller. Between the PTs there is a runtime reconfigurable communication network.

LuaFlow is the compiler for neuFlow, which parses the input, extracts different levels of parallelism and generates the configuration for the system.

In their experimental result they achieve a rate of 147 GMACS, around 92% of the maximum performance with a 9×9 kernel. This result is somewhat biased, normally networks have various kernel sizes.

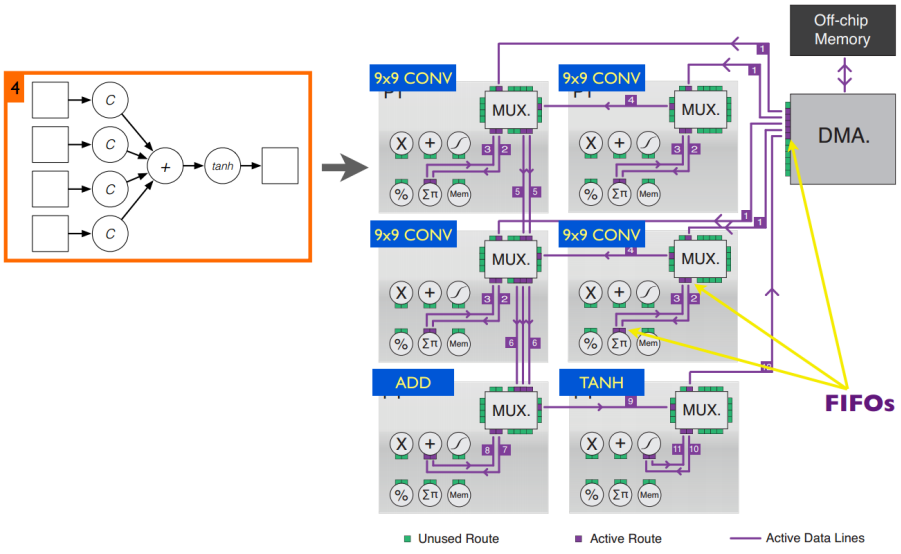


Figure 25: Running a part of the network on the neuflow

To make the Speed-Sign Recognition Network work on the NeuFlow processor[13], the graph needs to be divided into subgraphs that fit on the grid. Each Processing Tile of the grid can do a single convolution. To be able to solve the convolutions of the network, each Processing Tile must consists of at least 25 PEs to be make the convolutions of kernel sizes of 5×5 possible.

Synthesizing NeuFlow on the ZedBoard results in a maximum of $\lceil \frac{220}{25} \rceil = 8$ Processing Tiles. With one tile needed for adding results and one for doing the activation function, it is possible to do a maximum of 6 convolutions in parallel. If the NeuFlow can not store intermediate values, at least 80 processing tiles are needed to do the convolutions, since there is a layer with 80 featuremaps,

	computations (10^6)	data (10^6 bytes)	cycles (10^6)	bandwidth (MB/s)
layer 1	137	0.9	0.91	98
subsampling	0.91	21.9	0.04	54700
layer 2	331	13.47	2.21	610
subsampling	14.3	57.4	0.59	9729
layer 3	866	53.8	5.77	932
layer 4	40	79	6.67	1184
total	1389	265	16.19	54700

Table 8: Results of running Speed-Sign Recognition Network on the NeuFlow

so I assume it can store intermediate result in the off-chip memory. Furthermore, their subsampling technique requires more computations by splitting the subsampling and the convolution in to two separate layers.
Table 8 8 shows the results of running the Speed-Sign Recognition Network on the NeuFlow. In the third layer and fourth layer, all featuremap have too many input featuremaps, this can not b edirectly solved, and it therefore requires storing intermediate values on the external memory. Only reading this intermediate values from the memory is considered in the table. Overall, the reuse is much less because NeuFlow requires that the convolutions need to be calculated completely parallel, making in only possible to reuse data for a maximum of 5 convolutions.
Layer 4 has a kernel of size 1×1, resulting in a maximum utilisation of $\frac{1}{25}$ of the PEs. Therefore the computations of layer 4 takes even longer than the computations of layer 3.
The NeuFlow can calculated the Speed-Sign Recognition Network in 161ms at 100MHz, almost equally fast as the new architecture (146ms). But to be able to achieve this number, a bandwidth of 54.7GB/s is required. In comparison, my architecture only requires 9.9MB of data from the external memory from this network, a reduction of 26×. The highly required memory bandwidth is not feasible on a mobile platform. When this is used on a mobile platform, a memory with a lower bandwidth is used. This will increase the computation time.

6.4.2 Massively Parallel Coprocessor

NEC Laboratories America presents a massively parallel coprocessor for accelerating Convolutional Neural Networks [11] and make it dynamically configurable [12].
The coprocessor consists of computational elements that consists of programmable units that can perform sub-sampling, non-linear functions and convolution primitives (convolvers). The number of convolvers per computational element and the number of computational elements is dynamical configured per layer. If all output images do not require the same number of convolutions, there will be disabled convolvers. If not enough convolvers are available computational elements will produce partial results that need to be stored on off-chip memory. The $k \times k$ convolvers primitive has a fixed k across the coprocessor, and fixed at the hardware level. This result in not fully utilized PEs, when computing for kernel sizes smaller than k. Computational elements can simultaneously use the same input image, resulting in less off-chip memory requests.

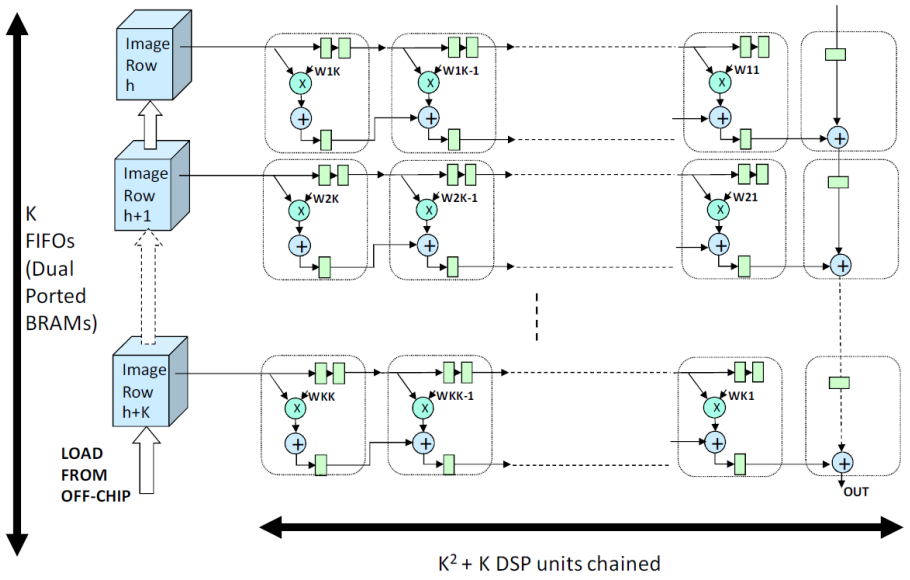


Figure 26: The design of the convolution-hardware of the Massively Parallel Coprocessor

layer	config	computations (10^6)	data (10^6 bytes)	cycles (10^6)	bandwidth (MB/s)
layer 1	2×4	137	0.9	0.68	132
layer 2	2×4	331	8.2	2.0	410
layer 3	1×8	866	34.6	5.77	599
layer 4	1×8	40	56.5	6.67	847
total		1374	132.2	15.12	847

Table 9: Results of computing Speed-Sign Recognition Network on the Massive Parralel Coprocessor

.

With the dynamically configurable coprocessor[12], the number of convolutions per featuremap and the number of featuremaps in parallel can be configured. To be able to solve the Speed-Sign Recognition Network on this implementation, convolvers should support kernel sizes up to 5×5, making 8 convolvers possible at the ZedBoard.
Since there is a sub-sampling primitive, sub-sampling can be done on the fly. Results of computing the Speed-Sign Recognition Network on the Massive Parallel Coprocessor are depicted in table 9. The coprocessor requires a factor of 13 on data, resulting in a required memory bandwidth of 847 MB/s to achieve maximum performance. The coprocessor requires 151ms to compute the result. The massive parallel coprocessor shows some flexibility, but still require a large bandwidth. If we would run two applications on the Massive Parallel Coprocessor, where the second application uses a 8×8 kernel, only 3 convolvers would be possible on the ZedBoard, reducing the total performance and reuse possibilities of the system.

6.5 Comparison with consumer hardware

In this section a comparison will be made between the new architecture on a FPGA, with a CPU and CPU implementation. The Speed-Sign Recognition Network will be calculated, and the execution time and power usage will be measured.

6.5.1 FPGA implementation

To be able to determine energy usage of the new architecture, I have implemented parts of this architecture. The bus and the connection matrix parser are left out of this analyses.
I have synthesized 16 clusters with 8 PEs on the ZedBoard, consisting of a Zynq-7000 SoC XC7Z020-CLG484-1. The estimated execution time of the speedsign recognition network or this configuration at 100MHz is equal to 146ms per frame (achieving 57% utilisation of the PEs with 7.4 GOPSs). The hardware requirements of the synthesation are give in table 10.

	BRAM	slice Registers	Slice LUTs	DSPs
Input sequencer	1	97	390	0
Output sequencer	1	97	390	0
Clusters	32	2592	5808	128
total	34 (24%)	2786 (3%)	6588 (12%)	128 (58%)

Table 10: Different parts and their hardware requirement

Using Xilinx XPower Analyzer to model the power usage of different components, table 11 is obtained.

Part	**Power**
Block RAM (16x)	0.02672 W
Shift registers (16x)	0.01936 W
PEs (256x)	0.13056 W
Input sequencer	0.08046 W
Output sequencer	0.08046 W
Total	0.3376 W

Table 11: Power distribution of different components

Parts of the architecture are not modelled, so the resulting power requirement of 0.34 Watt of table 11 is optimistic. The total specified operating power of the XC7Z020 is equal to 3 Watt. [20]. That estimated operation power also include the *ARM* CPU, which is not used with this architecture.
Using the pessimistic number, the required energy for calculating for a single videoframe is equal to:

$$3 \text{ Watt} * 0.146 \text{ seconds} = 0.438 \text{Joule} \tag{6.5}$$

Using the optimistic power usage, it will require 0.049 Joule. The energy requirement will lie between these numbers.

6.5.2 CPU implementations

A fully optimized, with SSE and multithreading, implementation has been made. With 8 threads this implementation requires 0.081 seconds per frame on the Intel Core i7 3610QM, 2.3GHz. Using Intel Power Gadget[21] it is measured that the core dissipates a total 31.6 Watt when running calculations. The power usage of the DRAM is not calculated, so it is a bit optimistic.

$$31.6 \text{ Watt} * 0.081 \text{ seconds} = 2.5596 \text{ Joule} \tag{6.6}$$

6.5.3 GPU implementation

A fully optimized GPU implementation is made in nVidia CUDA. Running this implementation on the GTX460 results in a execution time of 0.0106 seconds per frame. The GTX460 uses 138 watt[22] under full load. The total energy usage per frame is equal to:

$$138 \text{ Watt} * 0.0106 \text{ seconds} = 1.47J$$

6.5.4 Comparison overview

	Power Usage	Energy Usage	Speed
GPU	138 W	1.47J	94.3 fps
CPU	31.6 W	2.56J	12.3 fps
Accelerator (pessimistic)	3 W	0.44J	6.8 fps
Accelerator (optimistic)	0.34 W	0.049 J	6.8 fps

The power consumption of mapping to the zedboard indicates a reduction of $3 - 30\times$ compared with a GPU implementation, and $6 - 52\times$ compared with a CPU implementation. Because of the lower power usage, the accelerator can be used in an embedded environment.
The accelerator achieves 6.8 fps, because the number of calculations is very large due to the usage of HD video. The ZedBoard is a small FPGA; using a larger FPGA with more DSPs increases the speed, With the same order of energy usage per frame, the quality of the videa can also be downscaled to VGA by decreasing the number of computations by a factor 4.

7 Conclusion

This study shows that it is possible to achieve good compute throughput with a flexible SIMD architecture on CNNs. To overcome external bandwidth limitations, iteration reordering is used in combination with a dedicated memory hierarchy that reduces the transfer bandwidth substantially. The complicated scheduling, which requires to map a network on the architecture, is handled by an optimizing toolflow. As a result, high utilisation rates are achieved for different network configurations, in contrast to other designs that are specific for particular network configurations. This claim is verified with two very different network configurations. Although this architecture is more flexible, the energy efficiency the same. This is achieved by a substantial reduction of external memory transfers and doing the scheduling offline. The results of this study make it possible to map a CNN based vision application to low cost devices. As a result many vision applications that require object recognition can be performed on these low cost embedded platforms. This can improve the use of smart vision systems in daily life, which increases safety, efficiency and comfort during various tasks.

8 Future work

Due to time pressure, there is some research left to be done. This section consists of two key ideas improving the performance and flexibility of the architecture.

8.1 Network splitting

When the number of clusters is more than the maximum number of featuremaps in a layer, there will be idle clusters. This reduces the utilisation and therefor the performance. A solution to this is to split the network into two networks. One network working on the top half of the CNN, the second network working on the bottom half CNN. This solution increases the amount of cluster parallelism, but it does come at a price:

- double the amount of weight values are needed
- the amount of work and data increase because of the boundary problems
- the input must be splitted
- the output must be merged

Using this technique on the Face Detection Network yiels the following increases in calculations:

normal	**splitted**	**increase**
1x720x1280	2x378x1280	5.0%
4x358x638	4x2x186x638	3.9%
14x177x317	14x2x91x317	2.8%
14x173x313	14x2x87x313	0.5%
1x173x313	2x87x313	0.5%

With more layers, the amount of input data required will increase even more. In future work this approach can be added to the toolflow.

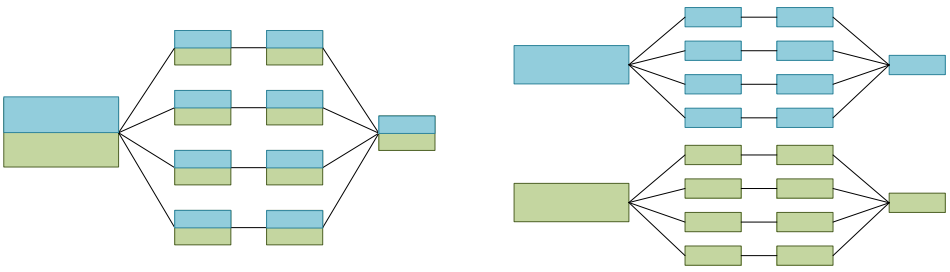


Figure 27: Splitting the network to increase parallelism

8.2 Memory adder

To support larger networks that do not fit into local memory, layers of the network can be split up into two (or even more) sublayers. The resulting pixels from the featuremaps can not be ready to go trough the activation fuction. In that case, intermediate values with large precision are transferred from the clusters to the memory controller. When all the partial results of a featuremap

are available in the global memory, an extra piece of hardware in the memory controller will combine the partial answers and will do the activation function. An example of this flow is depicted in figure 28.

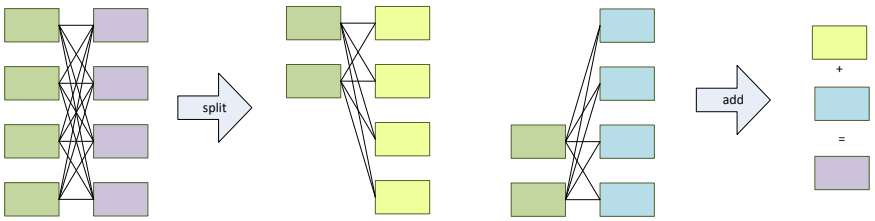


Figure 28: Splitting the network to decrease local memory requirement

This solution should be compared with the more trivial solution: copying intermediate values back to the clusters.

8.3 Hardware implementation

A complete hardware implementation is left to be made. By using this hardware implementation real power usage can be measured. This implementation requires engineering and can be done using the research of this report.

References

[1] "Zedboard.org."

[2] S. Belongie, J. Malik, and J. Puzicha, "Shape matching and object recognition using shape contexts," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 24, no. 4, pp. 509–522, 2002.

[3] T.-C. Chen and K.-L. Chung, "An efficient randomized algorithm for detecting circles," *Computer Vision and Image Understanding*, vol. 83, no. 2, pp. 172 – 191, 2001.

[4] D. Ballard, "Generalizing the hough transform to detect arbitrary shapes," *Pattern Recognition*, vol. 13, no. 2, pp. 111 – 122, 1981.

[5] D. H. Hubel and T. N. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," *J Physiol*, 1962.

[6] Y. Le Cun, L. Jackel, B. Boser, J. Denker, H.-P. Graf, I. Guyon, D. Henderson, R. Howard, and W. Hubbard, "Handwritten digit recognition: applications of neural network chips and automatic learning," *Communications Magazine, IEEE*, vol. 27, no. 11, pp. 41–46, 1989.

[7] M. Szarvas, A. Yoshizawa, M. Yamamoto, and J. Ogata, "Pedestrian detection with convolutional neural networks," in *Intelligent Vehicles Symposium, 2005. Proceedings. IEEE*, pp. 224–229, 2005.

[8] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back, "Face recognition: A convolutional neural-network approach," *Neural Networks, IEEE Transactions on*, vol. 8, no. 1, pp. 98–113, 1997.

[9] M. Peemen, B. Mesman, and C. Corporaal, "Speed sign detection and recognition by convolutional neural networks," in *Proceedings of the 8th International Automotive Congress*, pp. 162–170, 2011.

[10] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun, "What is the best multi-stage architecture for object recognition?," in *Computer Vision, 2009 IEEE 12th International Conference on*, pp. 2146–2153, 2009.

[11] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. Graf, "A massively parallel coprocessor for convolutional neural networks," in *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pp. 53–60, 2009.

[12] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, (New York, NY, USA), pp. 247–257, ACM, 2010.

[13] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pp. 109–116, 2011.

[14] S. Haykin, *Neural Networks and Learning Machines*. Pearson, 2009.

[15] M. Peemen, B. Mesman, and H. Corporaal, "Efficiency optimization of trainable feature extractors for a consumer platform," in *Advances Concepts for Intelligent Vision Systems*, pp. 293–304, Springer, 2011.

[16] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[17] "Hp labs: Cacti - an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model (http://www.hpl.hp.com/research/cacti/)."

[18] M. Peemen, "Memory-centric accelerator design for convolutional neural networks (to appear)," 2013.

[19] M. GAREY and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. A Series of Books in the Mathematical Sciences, W. H. Freeman, 1979.

[20] "Zynq-7000 all programmable soc overview (http://www.xilinx.com/support/documentation/data-sheets/ds190-zynq-7000-overview.pdf)."

[21] "Intel power gadget - http://software.intel.com/en-us/articles/intel-power-gadget-20."

[22] "Geforce gtx 460 review (roundup) - setup, noise, power consumption, heat levels (http://www.guru3d.com/articles-pages/geforce-gtx-460-review-(roundup),13.html)."

List of nomenclature

l_{ss}	the subsampling factor of layer l
l_{width}	the width of the output featuremaps of layer l
l_{height}	the height of the output featuremaps of layer l
l_{kw}	width of the kernel of the convolution of layer l
l_{kh}	height of the kernel of the convolution of layer l
fm_{conn}	the number of incoming connections of featuremap fm