A flexible memory shuffling unit for image processing accelerators

Xie, R.Z.

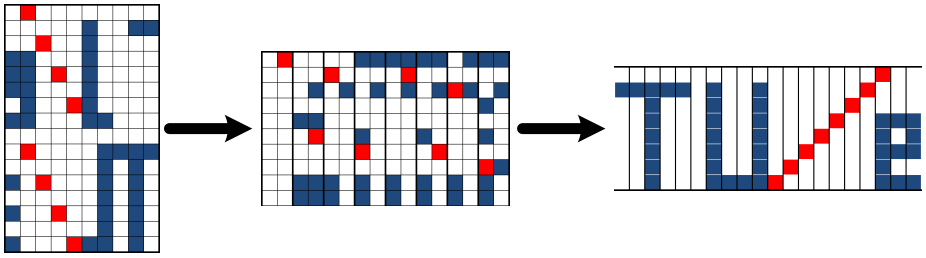
Award date:
2013

Faculty of Electrical Engineering
Eindhoven University of Technology

Master Thesis

A Flexible Memory Shuffling Unit for Image Processing Accelerators

November 11, 2013



Author:
Rui Zhou Xie

Supervisors:
Henk Corporaal
Maurice Peemen

Abstract

Nowadays devices with embedded cameras are found everywhere. These devices are able to perform complex tasks, such as image processing, object recognition and more. Such tasks easily require 200 operations per pixel and in addition each pixel is read and transferred many times, which consume a substantial amount of energy and execution time. However the number of transfers can substantially be reduced by exploiting data reuse by optimizing for locality. Often these optimizations result in more complex memory access patterns. In practice data transfers are performed by a DMA (Direct Memory Access) controller, which is able to transfer large consecutive block efficiently. However, large consecutive blocks do not match with the complex access patterns that are required for good locality. As a result, the attained bandwidth is only a portion of the maximum bandwidth. To improve bandwidth for locality optimized access patterns, this work proposes a memory shuffling unit that provides an interface between DMA and the accelerator IP (Intellectual Property). To provide enough flexibility in the access patterns, the DMA controller and the shuffling unit are programmable. Programming these units requires many parameters, which makes it manually complex and error prone. To abstract away from the complexity, a tool flow is developed that analyzes the schedule, claims memory, and generates the DMA transfers and shuffle instructions. Our shuffling unit increases memory bandwidth by 300x and decreasing the energy consumption by 300x compared to supplying the data in patterns directly from the off-chip memory.

Contents

1 Introduction

Nowadays portable devices with embedded cameras are found everywhere, they can be found in devices such as notebooks, tablets or even smartphones. Moreover, many companies are even trying to take a step further, a well known example is google glass [21] where a processing unit, camera and other sensors are integrated into a pair of glasses. These devices are potentially used for many graphical functionalities such as editing images, detecting objects in photographs or recording video. These devices should be able to perform these complex image algorithms with a decent performance, and in many cases even have real-time requirements. In addition they require extremely high energy efficiency to last more than a few minutes on battery power.

General purpose processors do not meet the high energy requirements set by those portable devices. By customization, which results in specialized accelerators it is possible to meet these requirements. These kind of heterogeneous architectures consumes 500x less energy and 500x better performance for a specific task [11]. Hence heterogeneous architectures, combining general purpose processors and specialized accelerators already dominate mobile systems, such as the Tegra [22].

Section 1.1 will give an introduction into data transfers for accelerators. The problem description of this project is described in section 1.2. Afterwards the outline of this thesis is given in section 1.3.

1.1 Data transfers

One of the open issues is the data transfers to and from accelerators. For image processing where each pixel easily requires 200 operations, the pixels needs to be transferred many times. This results in a huge bandwidth requirement. Fortunately the bandwidth can be reduced by reusing data.

A cache mechanism is commonly used in combination with a general purpose processor to exploit the locality of reference. Since for a general purpose processor the data access patterns are not known in advance, moreover it can vary drastically depending on the application being processed. To accomodate for that, a cache is designed to support a wide range of applications, this is done by means of selecting a replacement policy that performs well on average. A few examples of these policies are *eviction* or *least recently used*, to handle these policies area and energy is required [4]. Though for known access patterns these policies are not necessary, it is therefore more efficient to use scratchpad memories.

Writing to scratchpad memories requires manual instructions, this can be done by a general purpose processor. Instead it is more efficient in terms of energy and throughput, to use a DMA (Direct Memory Access) unit, which is specialized in transferring data.

1.2 Problem description

When focussing on applications which use image processing, a wide range of them are based on the same basic operation, namely 2D convolution. A few well known examples are photo filtering and augmented reality applications.

There are accelerators specialized in convolution operations having a small storage (for area, energy and flexibility), that can process data efficiently when data is provided in certain patterns. On the other hand, there is a DMA that can efficiently transfer large blocks of data. The problem is that these large blocks of data do not match with the patterns that are favorable for the accelerator, there is no efficient interface available between these two units.

To provide a flexible and efficient interface this work proposes a memory shuffling unit. The contributions of this thesis are summarized below:

1. A flexible interface is proposed to connect a DMA to an image processing accelerator. This interface reduces the off-chip memory bandwidth by allowing the accelerator to exploit data reuse. Furthermore the interface is sufficiently flexible within the application domain.

2. To outline the benefits and the costs of this interface a thorough analysis is given, that includes throughput, area and energy studies.

3. To relieve the programmer from the complexity of programming individual DMA transfers an automatic toolflow is developed. The flow defines the read and write instructions and the shuffle instructions as well.

1.3 Outline

This thesis is organized as follows. Section 2 gives a short introduction to the background information required for this project. Section 3 discusses reuse in image processing algorithms and how it can be exploited. Section 4 explains a concept of an interface able to reorder data. Section 5 presents implementation details of the proposed interface. In section 6 a toolflow is proposed which makes it possibly to automatically generate instructions for the proposed interface. Section 7 explains the toolflow using a complex image processing algorithm. Experimental evaluation is done in section 8 and the thesis concluded in section 10.

2 Background

This work includes performing experiments in hardware, in order to be able to rapidly prototype and analyse this hardware a *Field-Programmable Gate Array (FPGA)* is used. First the FPGA and its relevant components are described in section 2.1, afterwards an introduction is given in DMA data transfers in section 2.2. The results of the performed experiments regarding data transfers are show in 2.3. This section will be concluded in section 2.4.

2.1 Field programmable gate array

For this thesis a *Zedboard (rev c)* is used to perform the experiments. This board includes a *Zynq-7000 XC7Z020 SoC* [24] and is composed of the following major function blocks:

- Processing System (PS)
  - Application processor unit (APU)
  - Memory interfaces
  - I/O peripherals (IOP)
  - Interconnect
- Programmable Logic (PL)

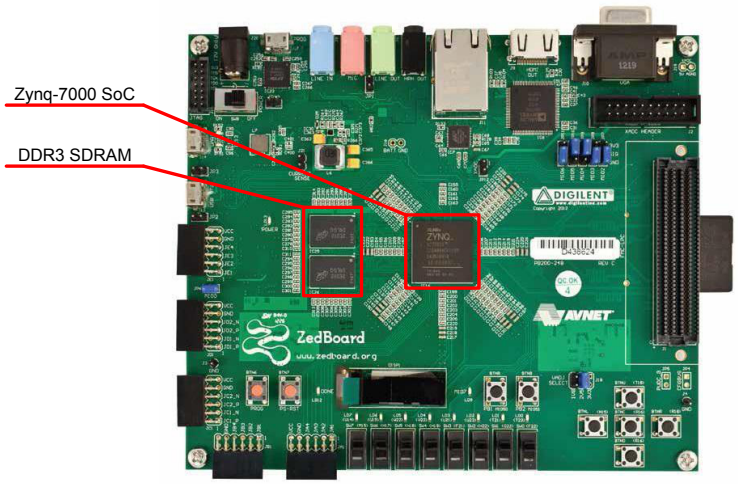


Figure 1: Zedboard (rev c)

The components of importance in this project are indicated in figure 1. First is the off-chip memory *DDR3 SDRAM* [14], there are two units, each having a size of 2GB on board. The designed hardware for this project is done on the programmable logic of the *Zynq-7000 XC7Z020 SoC*, where the *APU* is used to easily control and benchmark the hardware.

To prototype on the *Zedboard*, Xilinx tools are used to rapidly test and benchmark the hardware. In particular new IP's were written and synthesized

using *Vivado HLS* [12], which converts C/C++ code into HDL (Hardware Description Language). This IP can then be tested by adding it into a simple test setup, using the EDK.

For these tools Xilinx provides a catalog with commonly used IP cores, also for this project a few of these IP's are used to simplify the design part. Namely the *AXI DMA* [2] is used for transferring data and *AXI TIMER* [3] is used to perform benchmarks.

2.2 Data transfer

For a general purpose processor, data access patterns can vary drastically depending on the loaded application. To solve this problem a cache, which performs well on average, is commonly used in combination with a general purpose processor. In a heterogeneous architecture, where different processing units are specialized with respect to certain applications, access patterns are known before hand. Due to this reason it is not worthwile to add a cache, instead a scratchpad memory is used in combination with a DMA. In this setup, the DMA is responsible for the off-chip communication, in order to gain more insights in the efficiency of these data transfers, this section will illustrate the bandwidth a DMA can provide.

2.2.1 DMA instruction

To illustrate the interfaces and components required to initiate a data transfer, a basic setup of the hardware is shown in figure 2. Using this setup a DMA instruction will be explained.

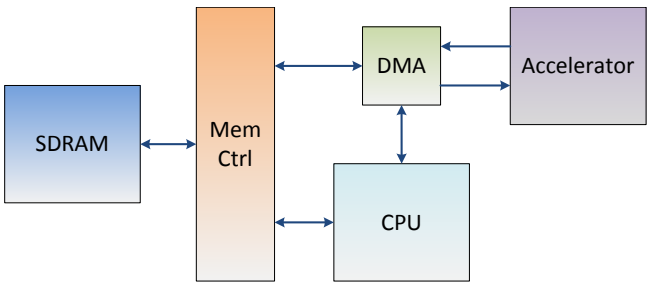


Figure 2: Benchmark setup

As depicted the CPU is directly connected to the DMA, it has access to a few registers to control it. The DMA contains two identical sets of registers, one for writing to the SDRAM and one for reading. To initiate a transfer, three registers need to be set:

- Start bit
- Address
- Size (in bytes)

For example, if one sets the corresponding parameters to issue the DMA to read from the SDRAM. The DMA will react to it by reading from the SDRAM

(through the memory controller) in bursts of data (figure 3). In this figure, there is a fifo that represents blocks of data that can be received by the DMA over time. The grey areas represent timestamps in which requested data is available. Where the width represents the width of the interface and the burst length is a static parameters.

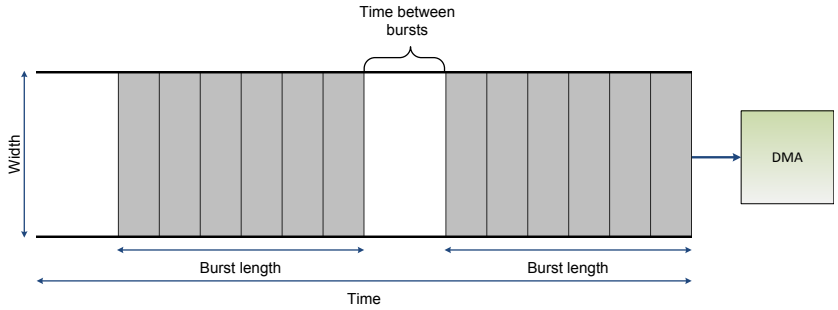


Figure 3: SDRAM bursts

After the DMA receives the data from the SDRAM, this data will be send to the accelerator. This does not happen in bursts, the DMA will send data whenever the interface is available. Therefore the rate at which the DMA can send data to the accelerator depends on the width of the interface and the rate at which the accelerator can fetch data from it.

2.2.2 Parameters

From a single data instruction, one can identify many parameters that can influence the bandwidth. Whereas in the previous section the flow of a data transfer is described. This section will list and describe the parameters in more detail before showing the experimental results.

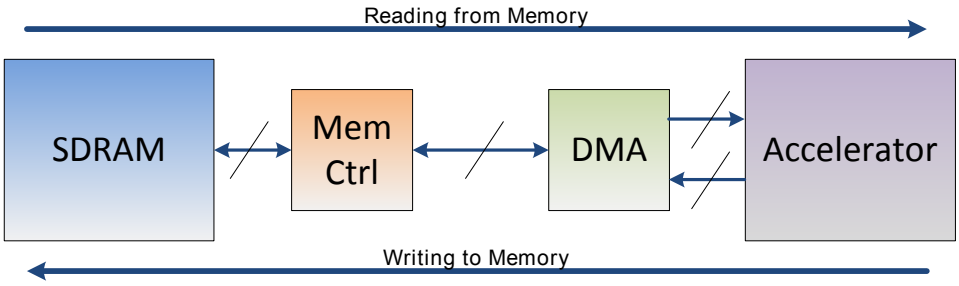


Figure 4: Bandwidth benchmark setup

Figure 4 shows a simple block diagram of the different components in the setup that can have influence on the bandwidth. Note that the CPU is not illustrated in the diagram, even though it is responsible for issueing memory transfers, it does not have any influence on the bandwidth.

From the components and the interfaces in the figure, the following parameters can be identified:

- Clock frequency
- Data transfer size
- SDRAM burst length
- Bus width of memory interface
- Bus width of accelerator interface

Note that the clock frequency and the data transfer size are trivial parameters that hold in general. Increasing the clock frequency will obviously increase the bandwidth, however in terms of energy consumption it is favorable to keep this parameter as low as possible. Even though it is a valid parameter, the clock frequency will be kept constant for the experiments, since the effects with respect to the bandwidth is predictable. Furthermore there is the data transfer size, in practice this parameter varies with respect to the context. Nevertheless if the goal is to achieve a high bandwidth, one has to keep this in mind.

For data transfers without any overhead the efficiency (eff_t, theoretical) can be computed as follows:

$$\text{eff}_\text{t} = \frac{\#bytes}{t_{transfer}}$$

Where $\#bytes$ is the number of bytes transmitted and $t_{transfer}$ is the number of cycles it takes, these two values have a linear relation.

For the data transfer itself, this analysis is correct. However, to initiate data transfers the DMA requires instructions, such as the start address, length and start bit. Therefore initiating a data transfer requires a relatively large amount of time, including this overhead in the formula gives:

$$\text{eff}_\text{p} = \frac{\#bytes}{t_{transfer} + T_{dma}}$$

From this can be deduced that increasing the size of the transfers, will decrease the effect of the overhead.

As covered in the previous section, data transfers to and from the SDRAM happen in bursts. Larger bursts can increase the efficiency of the SDRAM and therefore has influence on the rate at which data can be retrieved from the SDRAM (and vice versa). In order to explain the effect of this parameter, the functioning of an SDRAM has to be explained in more detail.

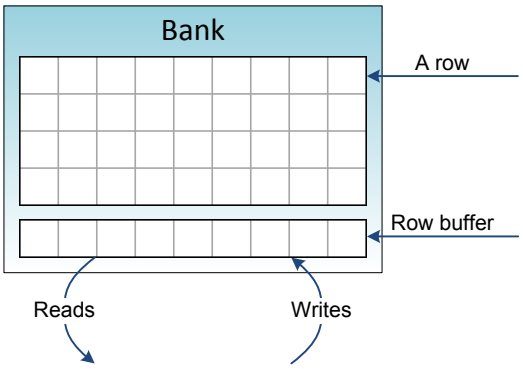


Figure 5: SDRAM bank

An SDRAM consists of multiple memory banks, where each bank consists multiple rows and a row buffer (figure 5). For every SDRAM access, the address of the request is first decoded into a bank, row and column addresses using a memory map [1]. Using the bank and row addresses a bank can be selected and the corresponding row can be requested. This row will then be loaded into a row buffer, which stores the most recently activated row. Now a number of reads or writes can be issued to access the columns in the row buffer, where the number of reads or writes is the burst length. The described addresses are illustrated in figure 6.

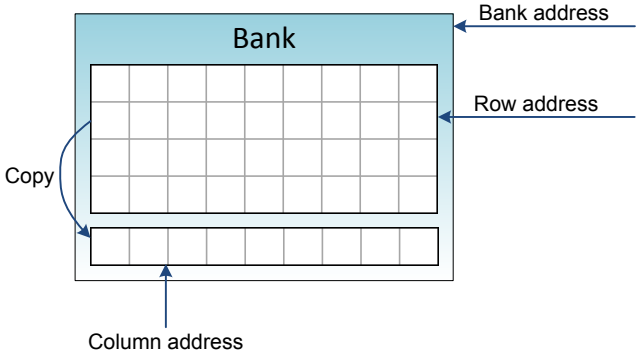


Figure 6: SDRAM addresses

The size of each read or write depends on the width of the interface connected to the memory controller. Trivially it is more efficient to issue as many read or write requests as possible, however this depends on the amount of data that is requested, the size of the row buffer and the width of the interface. If the amount of data is small, there is no use to have a large burst length, since a majority of the requested data will be discarded. Furthermore, since one wants to reduce the number of row activations, alignments becomes an important issue. By allocating large sequences of data to column addresses equal to 0, the start of a row, one can ensure the minimum number of activations.

Introducing the width of the interface between the memory and the DMA

as W_{mem}, the size of a burst in bytes S_{burst} can be computed as:

$$S_{burst} = W_{mem} * L_{burst}$$

Where L_{burst} represents the length of the burst. So when a data transfer size is specified in the register of the DMA, these are split up in bursts, where each burst has the same size. The number of burst can be denoted as N_{burst}:

$$N_{burst} = \left\lceil \frac{S_{transfer}}{S_{burst}} \right\rceil$$

Here $S_{transfer}$ is the size of the entire transfer, which is set in the register of the DMA. If $S_{transfer}$ is not a multiple of S_{burst}, data will automatically be discarded by the DMA to match the total size of $S_{transfer}$.

2.3 Experiments

This section will present the results of the performed experiments, the delivered bandwidth are measured while varying the parameters described in the previous section.

First the experimental setup is outlined, this is illustrated in figure 2. A CPU is used to prepare the SDRAM, setup DMA transfers and measure the time in cycles. Before the initiation of the DMA the CPU starts the timer until an interrupt is generated by the DMA.

Figures 7 and 8 show the experimental results of setting the interfaces to 32 bits for reading and writing while varying the size of the transfers and burst length.

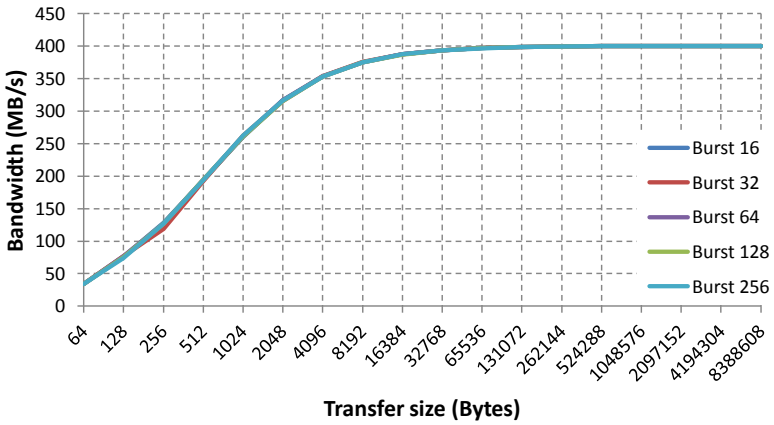


Figure 7: Reading from SDRAM using 32b interfaces

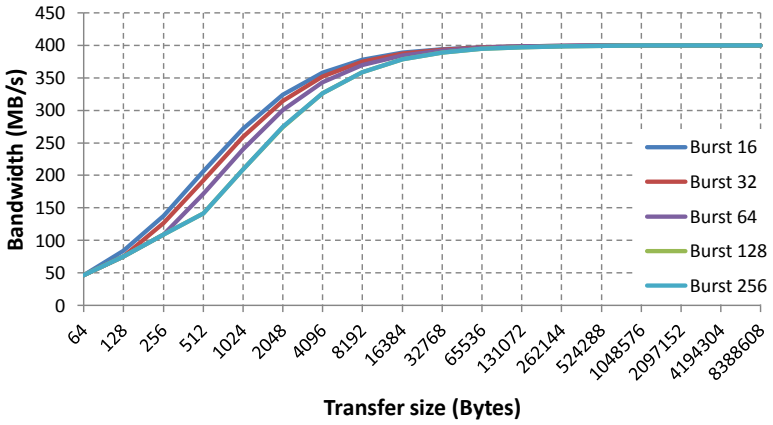


Figure 8: Writing to SDRAM using 32b interfaces

For the measurements having an interface width of 32 bits, one can find that the bandwidth gradually increases as the size in bytes increases, nearly reaching the theoretical maximum above sizes larger than 8KB. This behaviour was to be expected, as the size of the transfer increases, the effect of the overhead T_{dma} decreases.

Figures 9 and 10 show the experimental results of setting the interfaces to 64 bits for reading and writing while varying the size of the transfers and burst length.

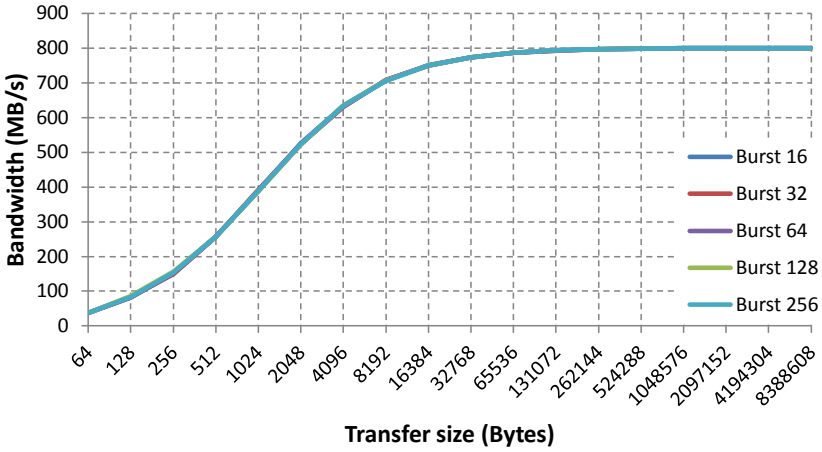


Figure 9: Reading from SDRAM using 64b interfaces

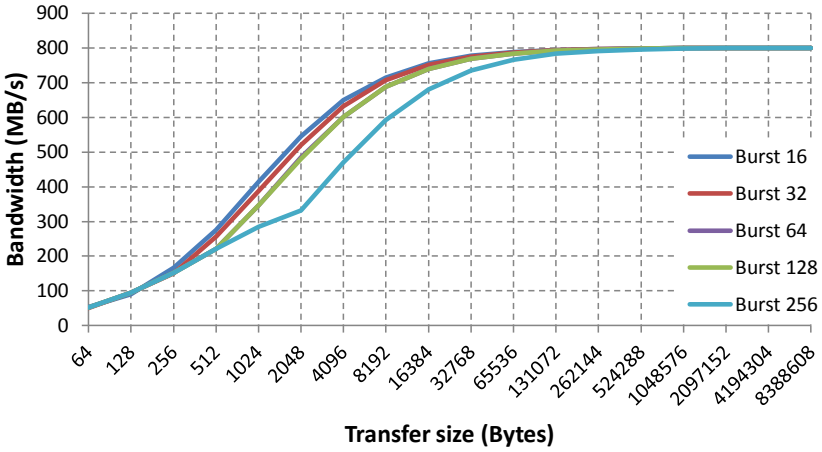


Figure 10: Writing to SDRAM using 64b interfaces

The same results can be seen while setting the width of the interface to 64 bits. The main difference is that the maximum bandwidth doubled, this can be explained by the fact that the interface allows to send twice as much data per cycle.

For both sets of measurements, one can find that increasing the burst does not improve the bandwidth, instead it decreases. This is very counter-intuitive since the burst rate should decrease the latency by decreasing the number of activations in the SDRAM, especially for larger data transfer sizes.

2.4 Conclusions

From the results of the experiments can be concluded that transfers of consecutive data can be performed efficiently. One can almost reach the theoretical maximum bandwidth, provided that the transfer size is large enough. However this means that a large on-chip memory is required to allow efficient transfers.

3 Motivation of locality and reuse

In the previous section, some experiments are performed to show the bandwidth a DMA unit can provide when transferring consecutive blocks of data. However, those measurements do not show the performance when transferring data in patterns that are favorable for 2D convolutional image processing. This class of algorithms requires a high bandwidth, because a lot of data needs to be transferred multiple times. The required bandwidth can be reduced by changing the data access pattern such that the accelerator is able to reuse data.

The reduction of data transfers required to perform the operations will also reduce the energy consumption. The focus of this section will be to explore how to allow the accelerator to reuse data and show experimental results. In section 3.1 the reuse of data is explained, section 3.2 describes how the reuse of data can be exploited and what the consequences it has on the data access patterns. Section 3.3 shows the results when transferring those patterns using a DMA and this chapter is concluded in section 3.4.

3.1 Pixel reuse

In order to explain how to reuse data, an example of the convolution operation is illustrated. In convolution a filter is used, also known as a *kernel* which is a small matrix with a certain width W_K and height H_K (where the K denotes kernel). The operation of such a kernel to process an image is illustrated in figure 11.

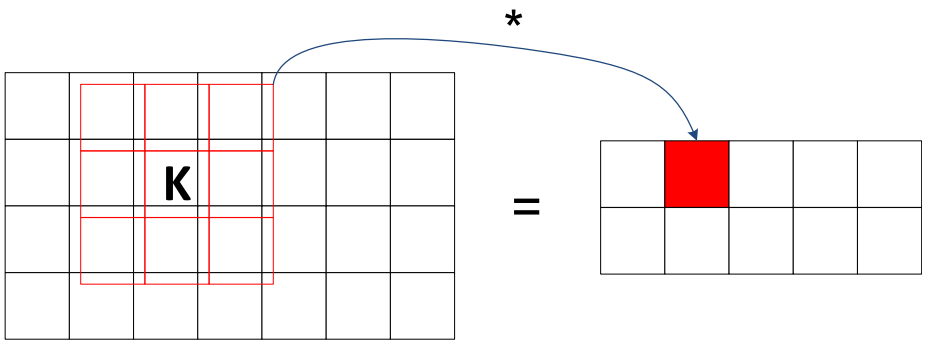


Figure 11: A simple convolution

In the figure, a very small image is being convolved with a 3x3 kernel, where the kernel is slided over the image, the kernel and its overlapping values are multiplied and summed, resulting in a value in the output image. The fact that only a small area, in this case a block of 3x3, is required to compute is a characteristic of window-based algorithms, where the output does not depend on the entire image, but only a small portion of it.

An important property is the overlap of the different blocks. Assume the red block is currently being computed by the accelerator, which implies that those pixels are loaded in the local memory. Afterwards, one could choose to slide the kernel either horizontally or vertically. Due to the overlap of the blocks, not the whole block needs to loaded, instead loading a small portion is sufficient.

3.2 Exploiting reuse

From the previous section can be concluded that in convolution alike image processing algorithms it is possible to reuse data by providing neighbouring data. The most obvious way to do that is by including a large memory in the accelerator, where all the required data can be stored. This option allows the accelerator to fully use the available reuse within an image, however this requires a large amount of local memory, which is expensive in terms of energy and area, especially when multiple accelerators are used. Moreover, optimizing for locality using this method decreases the flexibility:

- If the user wants to process an image larger than the available local memory, the accelerator might be unable to do so.
- There are applications in the image processing domain, which require even more memory. For instance, one might need to process several images at a time.

Fortunately there are a few options that require less local memory and still allowing to reuse the data.

3.2.1 Row buffers

An easy option is to fully optimize reuse is by loading a few entire rows equal to the height of the processing window, as depicted in figure 12. The figure represents an image, which is entirely stored on the off-chip memory. Here the gray areas are loaded in the local memory and the block representend by **P** is the kernel. In this situation all available reuse can be exploited in the first set of rows.

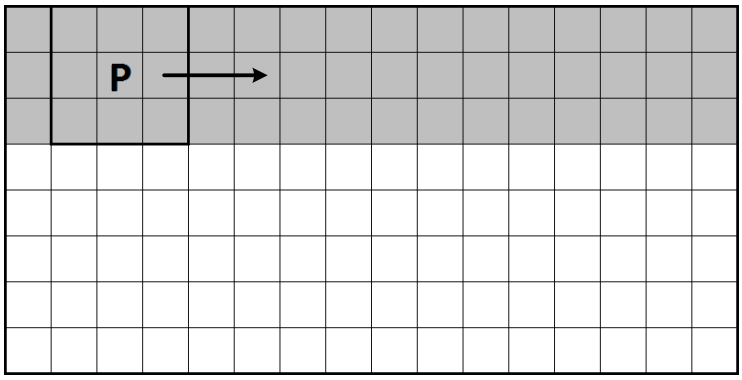


Figure 12: Row buffers

Once the first rows are all processed by **P** the next row can be loaded, while discarding the first row. This allows **P** to exploit all the reuse available in the following set of rows as well. This method effectively allows one to exploit all reuse available in an image, however it requires a fairly large local memory as well.

3.2.2 Column buffers

To minimize the memory requirements of the accelerator, the data transfers have to be minimized. Only a small amount of pixels need to be send to compute the next block according to figure 11. This method can efficiently be implemented for general cases, as illustrated in figure 13. By shifting the kernel to the right, the start of each row $W_K * H_K$ needs to be send, whereas for the rest of the computations only require H_K new pixels.

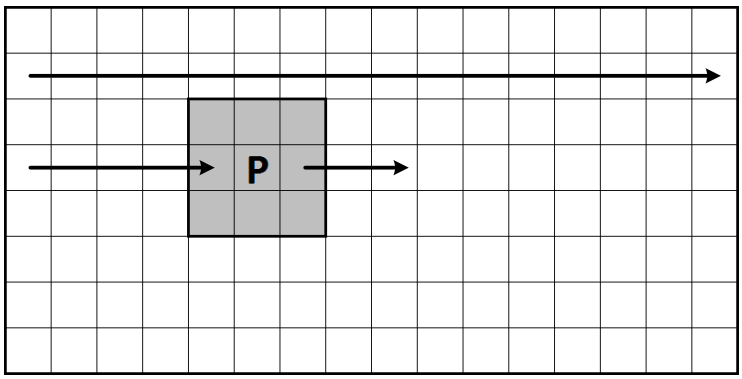


Figure 13: Column buffers

In terms of reuse, it only reuses data while processing in the horizontal direction. The amount of data that needs to be transferred can be calculated by:

$$N_D = H_O * (W_K * H_K) + (W_O - 1) * H_O * H_K$$

Where N_D denotes the number of data that needs to be transferred, W_O and H_O represent the width and height of the output image respectively. Where the first term computes the total number of pixels to be reloaded (H_O times) and the rest of the time only H_K new pixels are required.

Note that this method only benefits from the reuse when moving the processing window to the right. When a new row needs to processed, there is a overlap with the previous row, however that property is not used. One could benefit from it by increasing the size of the window loaded in the accelerator.

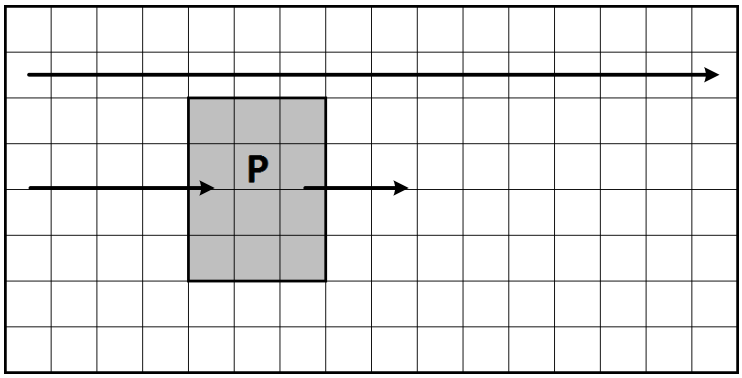


Figure 14: Column buffers extended

For example, while in figure 13 three pixels are send at a time, figure 14 shows an example of sending four pixels. The idea is that by sending columns of three, only one output value can be computed. By increasing the size of the columns to four, two output values can be computed at the same time.

In other words one can increase the height of the input columns to increase the height of the output columns, where each pixel in the output column can be computed in parallel, which depends on the accelerator. The height of the output columns is a parameter denoted by H_{OC}, from which the height the input columns (H_{IC}) can be computed as:

$$H_{IC} = (H_{OC} - 1) + H_K$$

$$N_D = \left\lceil \frac{H_O}{H_{OC}} \right\rceil * (W_K * H_{IC}) + (W_O - 1) * \left\lceil \frac{H_O}{H_{OC}} \right\rceil * H_{IC}$$

Where figure 13 required 288 to be transferred, by extending the size of the columns to four, the number of transferred bytes can be reduced to 192.

Note that this method can also be used from top to bottom (*row buffers*), since in many cases the images are wider than they are high, thus more reuse can be exploited using *column buffers*.

3.3 Experiments

In the previous sections data patterns to optimize for locality were explored. In this section experimental results are presented when transferring data in patterns according to *column buffers* or *column buffers extended* using a DMA.

Before presenting the results of the experiments, the pattern in which data is send is first illustrated in figure 15. Here the data is send in columns, where the number in the columns indicate the order in which the data is send.

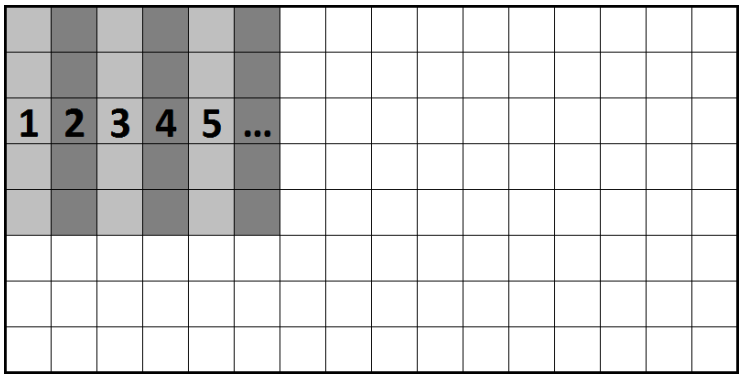


Figure 15: Column pattern

Figures 16 and 17 show the actual results of the experiments. For each entry in the column a separate DMA transfer is initiated where only one pixel is used. As described in section 2 the size of a single transfer depends on the width of the interface and the burst size. As only one pixel is needed, both parameters are minimized.

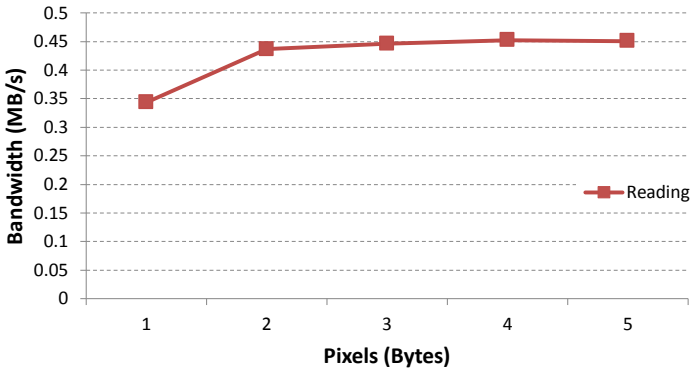


Figure 16: Reading column patterns

Figure 16 shows reading these patterns from the SDRAM, one can see that the bandwidth decreases by nearly three orders of magnitude compared to transferring a large block of data (figure 7). These results were expected, since multiple requests issued send for transferring a small amount of data, causing a huge overhead. Furthermore for each request to the SDRAM a burst of data is received, from which only one pixels is read and the rest is discarded.

The data is send in columns to the accelerator, which means that the outputs are retrieved in columns as well. Therefore the DMA controller has to accomodate for that as well, the experiments are shown in figure 17.

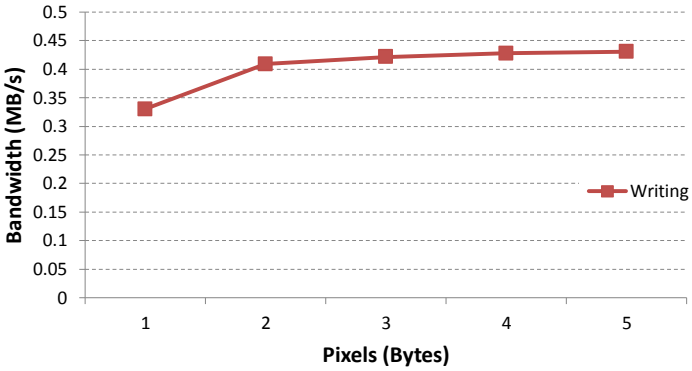


Figure 17: Writing column patterns

The results are similar as the measurements shown in 16, this is also due to the high overhead of the data transfers.

3.4 Conclusions

This chapter shows data access patterns beneficial for image processing accelerators. In other words, sending data in these patterns requires less on-chip memory. Using these patterns it is important to reuse data to decrease the amount of data to be transferred. Experiments are performed in which the DMA is used to send data in these patterns, however the bandwidth decreases by nearly 3 orders of magnitude when compared to transfers of large consecutive blocks. Even though these patterns are beneficial for image processing accelerators, the decrease in bandwidth is to large to be of any use.

4 Data reordering

In previous chapters it is shown that the DMA can provide a high bandwidth if the transfer consists of large blocks of data. On the other hand there is the accelerator, which can reduce the amount of data to be transferred if data is delivered in certain patterns. However, these patterns do not match the transfers the DMA can handle efficiently.

In order to provide the data in patterns to the accelerator while at the same time achieving a high bandwidth, the accelerator and the DMA accesses should be separated. This can be achieved in the form of an interface, that can receive large amounts of data while at the same time output this same data in patterns favorable for the accelerator.
Outputting data in a different order implies that this interface requires a buffer memory. Adding the interface, the setup is illustrated by 18.

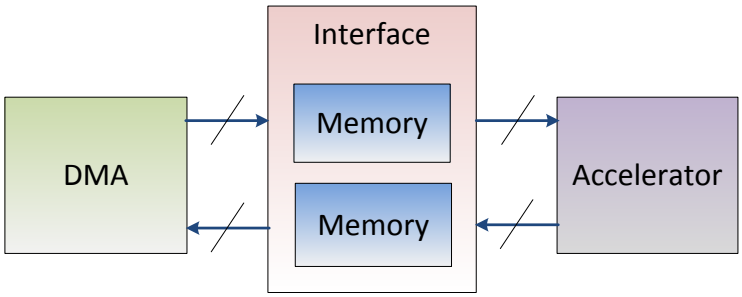


Figure 18: Interface for DMA/accelerator

In this setup, two memories are included because the data moves in two directions, and the data transfers of these two directions are unrelated. An issue that can be identified here is, is that data is written and read from the same memory which can cause collisions and makes this setup very error sensitive. A solution to resolve these collisions is by using a *ping pong* method. What happens is that the memory is divided into two smaller memories, the idea is that one unit reads one of the memories while the other unit writes to the other. This is a property that can be forced by the interface. The result of the modification is shown in figure 19, here the exclusivity of memory usage is illustrated by the switches.

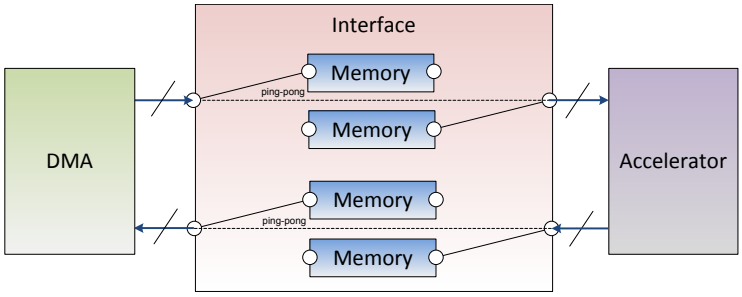


Figure 19: Interface including ping pong buffers

4.1 Reordering

By studying the details of the reordering mechanism, one can derive that the order of the data received from the DMA differs a lot compared to the order of the access patterns. For instance, if the data received by the DMA is writting to the memory within the interface in a consecutive order. The output of the interface (input for the accelerator) is scattered over the memory. Gathering data from the scattered positions in a memory is a time consuming task. The proposal is to split the memories into banks, where different banks are accessible in parallel. This structure allows access to multiple addresses in parallel, hence allowing to gather the correct data with a high throughput.

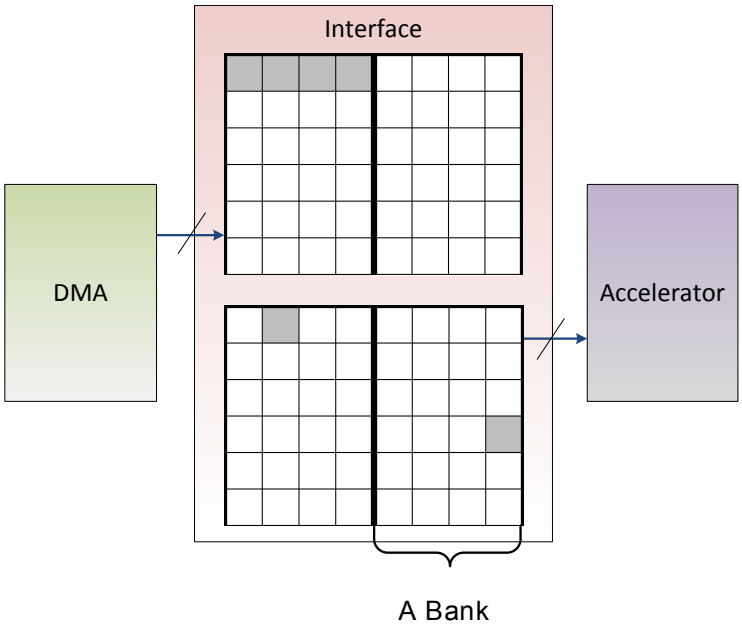


Figure 20: Banks concept

This concept is shown in figure 20, the transfers from the DMA towards the accelerator are illustrated. Each row represents one address, thus one access is required to access a row. The gray areas represent the areas being accessed simultaneously. In this example the DMA unit accesses the memories in a sequential order. In theory the accelerator can select any data as long as it only accesses one address in each memory. Only two memories are illustrated in the figure for illustrational purposes, however it is possible to add more memories to increase the flexibility.

4.2 Flexibility

For the reordering of data, this work focusses on convolution alike access patterns. These algorithms have multiple parameters, e.g. kernel size, subsampling factor, image or feature map size, the proposed interface can accomodate for that. How this can be realized will be explained in detail in section 5.

In commercially sold SoCs, different accelerators are used for different tasks, e.g. video decoding or face detection [22][15], from which many of them are based on convolution alike data access patterns. Therefore by focussing on data access patterns for the convolution operation, enables the interface to be used for a large range of accelerators.

5 Shuffling unit

A shuffling unit is proposed to provide an interface between the DMA and the accelerator. The unit allows a high transfer rate while at the same time provides access patterns required to exploit data reuse by optimizing locality for image processing algorithms. In other words increasing spatial locality for the DMA and maximize temporal locality for the accelerator.

5.1 Architecture

The basic architecture of the proposed memory shuffling unit can be found in figure 21, here the accelerator is divided into a computation unit and memory shuffling units. Furthermore there are FIFO buffers placed between the shuffling units and the accelerator, these are placed to facilitate for the speed differences between these components. For instance, if the shuffling unit wants to send data to the accelerator while it is busy, instead of being stalled, the shuffling unit can continue processing after writing the data to the FIFO.

In this setup the shuffling in is responsible for receiving the data from the DMA and supply the data to the accelerator in the correct pattern. The shuffling out receives the results from the accelerator, reorders the data and sends it back to the SDRAM through the DMA.

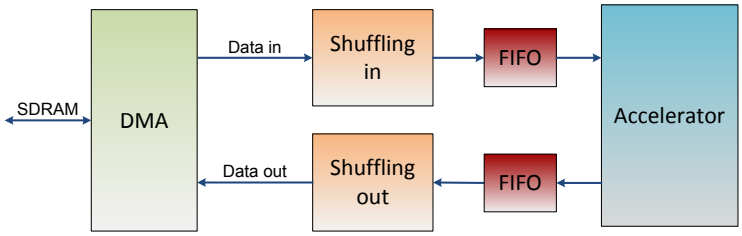


Figure 21: Including the shuffling units

The shuffling units can receive and provide large amounts of data from and to the DMA. Whereas on the other side, the shuffling unit can supply and retrieve data in complex patterns, optimized for locality, from the computation unit. Hence it is assumed that the computation unit has the required memory and intelligence to make use of this [8].

Note that there is only one DMA present in this setup, this means that data transfers can only happen in one direction at a time, as there is only one connection available to the SDRAM. This means that the bandwidth can be improved by adding another DMA, where one DMA is responsible for reading and the other for writing.

5.2 Memory

In order to send the same data in a different order requires buffer memory within the shuffling units, which is shown in more detail in figure 22.

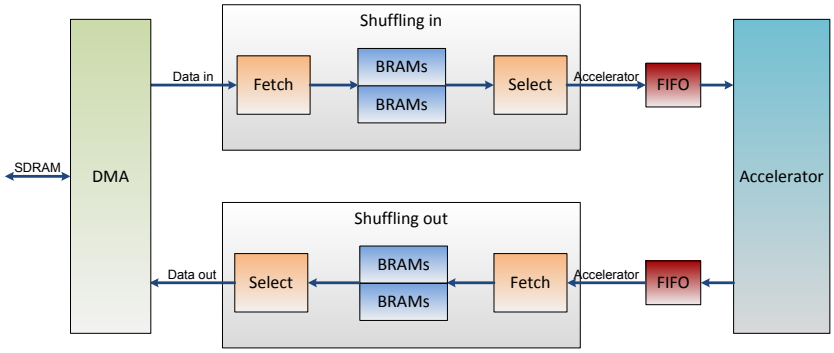


Figure 22: Fetch & select units

Each shuffling unit has a two memories consisting of multiple block RAMs and fetcher and selector units. Multiple memories allow the fetcher and selector unit to process data at the same time without having the risk to interfere with each other.

Multiple block RAMs allow a high flexibility in reading and writing, since each block RAM can be accessed at the same time [7]. The fetcher takes care of incoming data, it writes the data from the DMA into the block RAMs. The selector is responsible for selecting the correct data from the block RAMs and send it to the next unit.

5.3 Memory Patterns

To perform a shuffling action, the fetcher and selector should work together. However, depending on the accelerator connected to the shuffling unit, it should be able to supply different patterns. This changes the order in which the selector selects data, to be able to do that efficiently, the fetcher should also change the pattern in which the data is written.

To outline the inner working of the shuffling unit, an education example of 2D convolution is given. To describe the example the following notation based uppon [6] is used.

- N_B Number of banks
- N_P Bank width

It is preferable to set N_P equal to the width of the incoming interface, this way each input has the same size as an entry in the memory. Moreover N_B is set equal to the width (in bytes) of the outgoing interface, guaranteeing to be able to send data every cycle even if one byte is required from every bank simultaneously, provided that the data is sorted well. For now, it is assumed that all interfaces are 4 bytes wide. As shown in a previous section for the convolution operation it is favorable to send data in columns to the accelerator, from this information a few properties are derived.

1. For every column, a single item is accessed from a row.
2. Neighbouring rows are accessed in parallel.

These properties can be translated into data accesses in multiple memory banks. Since only one item from a row is accessed at a time, it can be concluded that a row can safely be written in the same bank. The fact that neighbouring rows are accessed simultaneously, implies that these rows should be folded over the number of banks. A pattern that could be used for convolution is illustrated in figure 23.

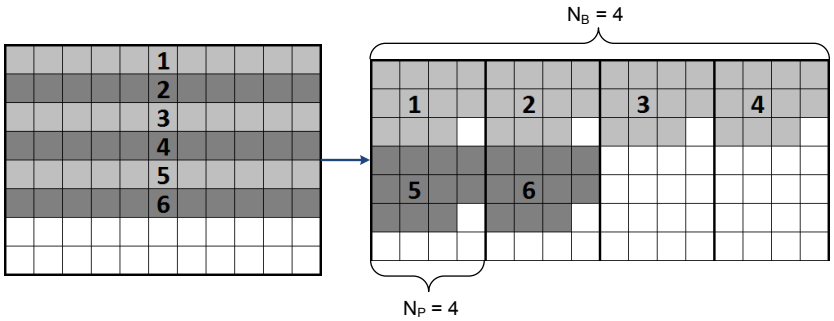


Figure 23: Fetch pattern for convolution

The block on the left is an image, where the lines are numbered. On the right the lines of the image are mapped on the memory banks.

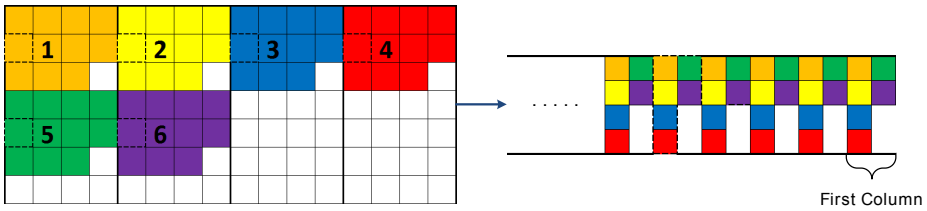


Figure 24: Select pattern for convolution

Afterwards the select unit can select columns from the memory banks, which in this example is a set of 6 pixels, this is depicted in figure 24. Where the memory banks are shown on the left side and the output fifo to the right. Here each row is illustrated with a different color, so each column consists of 6 pixels with different colors. Note that the operations of the fetch and select are similar for reordering output data.

5.4 Programmability

In order for the shuffling unit to work efficienty with a large range of image processing accelerators, it needs to be flexible. That means the DMA and the fetch and select units should be programmable, this is managed by a scheduler as depicted in figure 25.

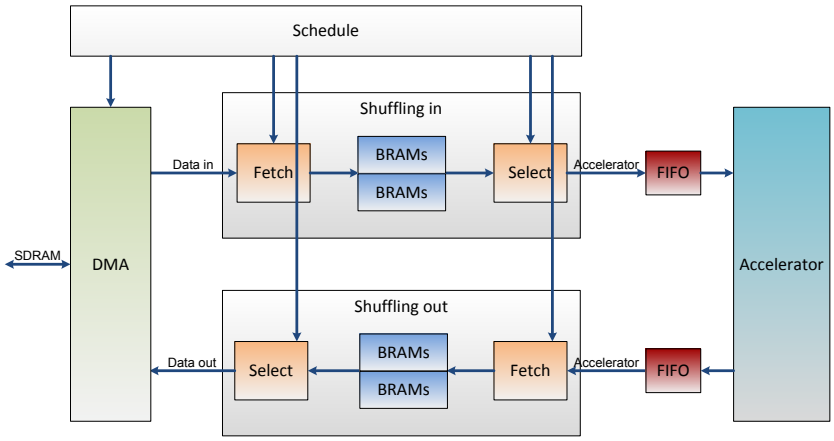


Figure 25: Programmable shuffle unit

Just like a DMA instruction, a shuffle instruction processes a large amount of data. Each shuffle instruction corresponds to both the select and fetch units, since they both have to process the same block of data. Between the fetch and select units there is a ping pong buffer, this means that for every instruction, the select should wait for the fetch unit is finished such that the ping pong buffer will be flipped. This results in a pipelined flow of the fetch and select units 26.

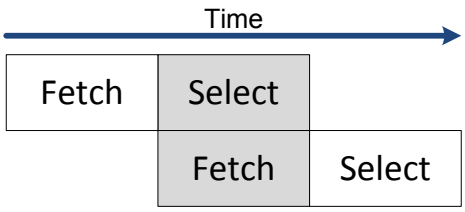


Figure 26: Fetch & select pipelined

5.5 Implementation

The architecture is implemented in hardware using *Vivado HLS*, as input language C/C++ is used, which is converted to a HDL by the tool. *Vivado HLS* offers easy creation of interfaces, due to these benefits, this tool allowed us to rapidly design prototypes of the shuffling unit.

This section will show samples of code written in *Vivado HLS* that are essential for the properties of the shuffling unit.

```
#define MEM_SIZE        8192
#define BLOCK_FACTOR    4
typedef int             BANK_TYPE;
BANK_TYPE mem_in0[BLOCK_FACTOR][MEM_SIZE/BLOCK_FACTOR];
BANK_TYPE mem_in1[BLOCK_FACTOR][MEM_SIZE/BLOCK_FACTOR];
```

Since a ping-pong mechanism is used, two identical sized memories are allocated. Each consisting of *BLOCK_FACTOR* banks (which is 4 in this case).

```
#pragma HLS ARRAY_PARTITION variable=mem_in0 block factor=4 dim=1
#pragma HLS RESOURCE variable=mem_in0 core=RAM_2P_BRAM

#pragma HLS ARRAY_PARTITION variable=mem_in1 block factor=4 dim=1
#pragma HLS RESOURCE variable=mem_in1 core=RAM_2P_BRAM
```

To define an appropriate structure regarding the block RAMs special commands are required in the form of pragmas. Using the *HLS ARRAY_PARTITION* pragma, the mapping of the array on top of block RAMs is specified. Afterwards it is specified to use simple dual port block RAMs to create this memory.

Besides having two memories, also two instruction variables are required, since the instructions fed to the select unit needs to be delayed. Which instruction and memory to be used for the fetch and select units is decided by a one-bit value i. A *last* variable is required to indicate the last instruction.

```
icInstr instr[2];
ap_int<1> i = 0;
unsigned int last;
```

This function takes care of reading instructions and the last bit.

```
last = readInstr(ctrl, instr[i]);
```

In the following segment the code representing figure 26 is shown. Where the fetch and select units are represented by functions, where each function receives a memory to read or write from and an instruction as parameters. First the fetch unit initiates the first instruction, afterwards the while loop is entered, where the fetch and select units run in parallel. In this loop the select unit processes instruction i, while the fetch unit is processing $i + 1$. Since the instructions to the select unit is delayed by one, this needs to be compensated after the last instruction is received (when the loop breaks). Note that the if statement acts as a switch to connect the fetch and select units to different memories and the code within the if statement guarantees exclusive access.

```
fetch(in, mem_in0, instr[i]);

i ^= 0x01;

while(!last) {
#pragma HLS DEPENDENCE variable=instr intra true
  last = readInstr(ctrl, instr[i]);

  if(i) {
    fetch(mem_in1, instr[1]);
    select(mem_in0, instr[0]);
  } else {
    fetch(mem_in0, instr[0]);
    select(mem_in1, instr[1]);
  }

  i ^= 0x01;
}

if(i)
  select(out, mem_in0, instr[0]);
else
  select(out, mem_in1, instr[1]);
```

6 Toolflow

In the previous section the shuffling unit is introduced to efficiently transfer data to and from the image processing accelerator. This unit is flexible and requires to be programmed according to the needs of the accelerator. However the programming happens with custom instructions, where these instructions depend on many parameters, e.g. interface widths and alignments in the hardware. All these parameters makes the programming process hard to construct and error-sensitive. That is why a toolflow is proposed to relieve the user from this complex task.

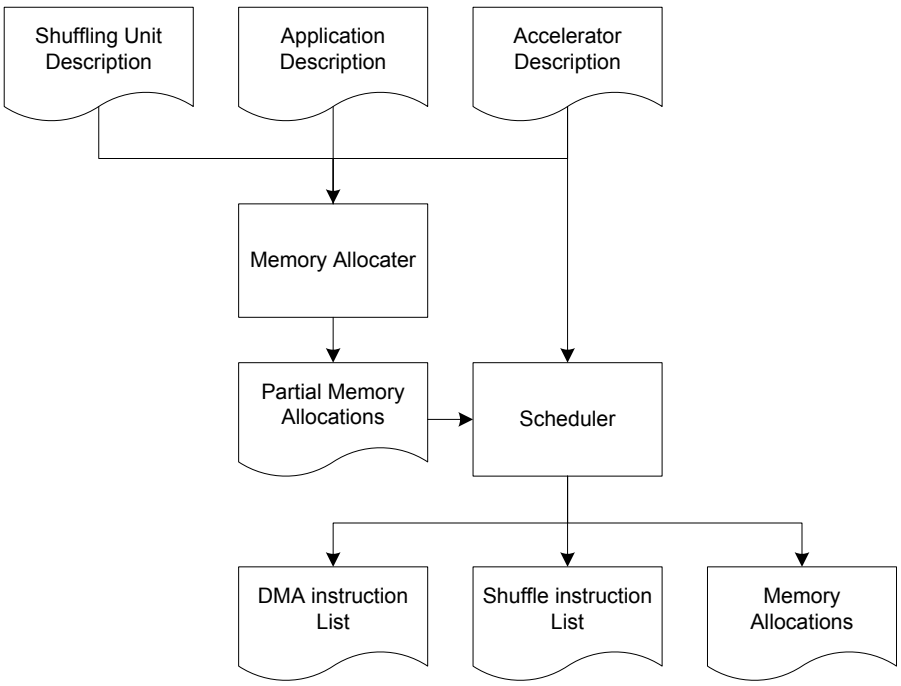


Figure 27: The toolflow

The goal of the toolflow is to generate shuffling instructions, though these shuffling instructions are also related to the DMA instructions, on top of that, the relative memory allocations depend on the DMA instructions. For the DMA to operate correctly, the memory allocations have to be aligned. Note that for the data transfers it is assumed that *column buffers extended* (figure 14) is used.

As depicted in figure 27, the following information are required as input:

- **Shuffling unit description**: this description contains information about the shuffling unit, allowing prediction of the behaviour.
  - **Interface widths**: required for alignment of the data. Incorrect alignments can result in undefined behaviour of the DMA.
  - **Memory size**: this parameter bounds the number of bytes the shuffling unit can receive.
  - **Number of banks**: the data send to the shuffling unit is written into multiple banks, in order to analyze the behaviour this parameter is required.
- **Application description**: information about the application can be found in this description, e.g. input dimensions and coefficients.
- **Accelerator description**: this description is required to know the behaviour and limits of the accelerator, e.g. the buffer size and number of kernels it can compute simultaneously.

6.1 Partial memory allocations

The partial memory allocations is an intermediate result created by the *memory allocator* process (figure 27). In this step the application description is used to extract all the allocations required on the SDRAM, e.g. input, intermediate, output and metadata. It is assumed all data is allocated in a large aligned block of data. In addition since all transfers are performed by a DMA controller all the values within this list must be aligned.

For example, consider computing of the partial memory allocations that are required for a convolutional neural network. This kind of network has one input image, several output data and intermediate data within each layer. The metadata is extra data that is required to perform the computations, e.g. the network coefficients.

6.2 Shuffle and DMA instructions

The DMA instructions is a list containing all the data transfers required for the accelerator to run an application. Each instructions consists of the following values:

1. Relative address
2. Size in bytes
3. Direction

The relative address is based on the partial memory allocations, because it describes the data required for this application. Furthermore, these transfers are directed in either direction as the accelerator also produces output data.
The shuffle instructions are closely related to the DMA instructions and can be generated at the same time. The toolflow will simulate the behaviour of the accelerator, since the simulator knows what data the accelerator expects, it can

generate the required shuffling and corresponding DMA instructions to provide data in the correct patterns. Note that each shuffling unit is related to multiple DMA instructions, this idea is further explained using figure 28.

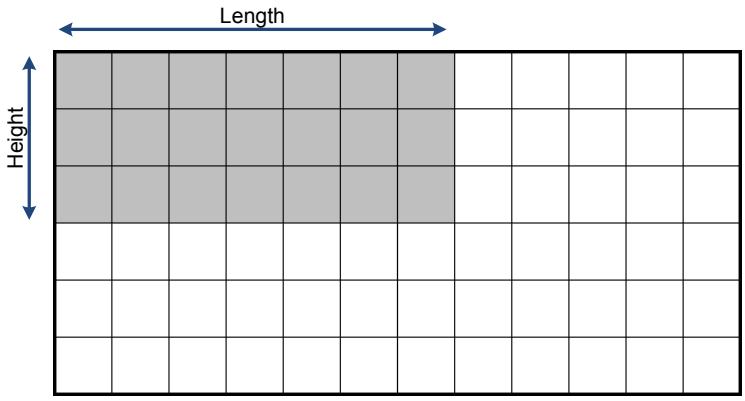


Figure 28: Shuffle instruction generation

Assume the gray area needs to be processed, though this area is not a consecutive block, hence multiple data transfers are required. This whole block needs to be shuffled as one entity, therefore three data transfers are required for one shuffle instruction in this example.

The maximum height of this area depends on the kernel size and the accelerator description since the accelerator can manage a number of computations at a time (*column buffers extended*, section 3). Using this maximum height, a length can be computed depending on the size of the memory within the shuffling unit. Here it is beneficial to use a length as long as possible for efficient DMA transfers. Note that it is possible to reduce the height of the transfers to increase the length. In other words increasing the DMA efficiency for the level of parallelism in the accelerator.

The effect gained here is that practically a larger image is cut in smaller blocks of data, where each block is processed separately. The manner in which an image is partitioned is shown illustrated in figure 29.

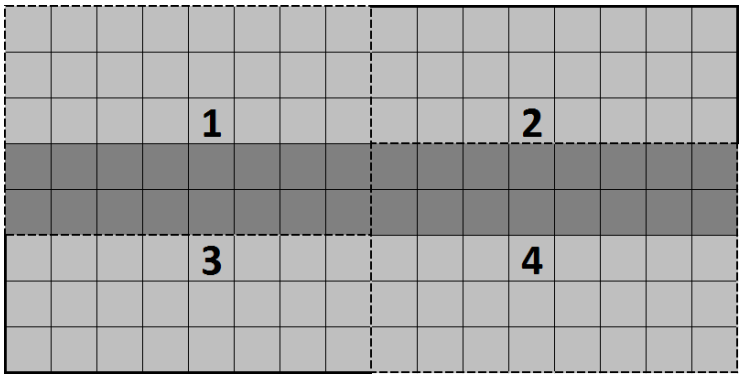


Figure 29: Partitioned image

In this figure an image is partitioned in four blocks, where between these blocks there is overlap (indicated by a darker color), the size of the overlap depends on the processing parameters. As mentioned, the height and length depends on the accelerator and the memory in the shuffling unit. So for wider images, it could be necessary to horizontally split the image in more than 2 partitions. The overlapping blocks of data are assumed to be transferred twice, hence simplifying the memory management within the shuffling unit for the price of exploiting less reuse.

6.3 Memory Allocations

While the memory allocations are computed in the partial memory allocations, some algorithms require additional memory allocations. For instance due to the lack of resources in the accelerator, such that one needs to store intermediate results of computations. These memory allocations, can be computed while simulating the behaviour of the accelerator.

6.4 Toolflow results

In the previous subsections the different results from the toolflow are listed, this section will describe how these different results are related and how they are used to control the shuffling unit.

The results of the toolflow consists of the memory allocations, DMA instructions and shuffle instructions, where the DMA instructions matches with the memory allocations and in turn the shuffle instructions are closely related to the DMA instructions. The memory allocations describes a list of relative addresses for all the data which is required for the computations, for which a large and aligned block of data is allocated. The designated memory locations of the input and metadata (e.g. coefficients) are required to be filled before the computations can start. The result of the toolflow is depicted in figure 30.

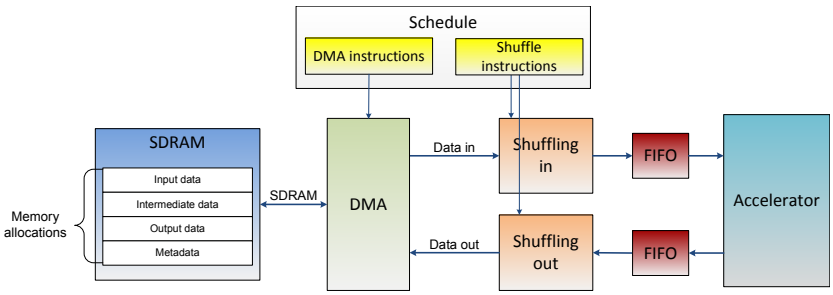


Figure 30: Theoretical concept of toolflow results

In the figure, the data is allocated and the *scheduler* now has access to the DMA and shuffle instructions. This scheduler is responsible for supplying instructions to the DMA and the shuffle units. In our setup the DMA instructions are set by the CPU whereas the shuffling instructions are send using an extra DMA unit. Once the data allocations are done, the DMA and shuffling units can be initiated by supplying the instructions.

7 Convolutional Neural Networks

In the previous sections, a shuffling unit was proposed based on the idea to reduce data transfers, the focus was for applications based on the convolution operation. This section will perform a case study for CNN's (convolutional neural networks). These networks have shown to provide a large level of parallelism and are flexible in terms of feature extraction and classification of images. Due to these properties CNN's are successfully applied in the domain of object recognition [16][10][13]. However a CNN requires a lot of data, therefore this section will describe the manner in which computations are performed in a CNN, from this study, data patterns can be found which are able to exploit locality of access [18].

As the name suggests, a CNN is a network of neurons, in this case artificial neurons. Each operation is performed by a such an artificial neuron and is illustrated by figure 31.

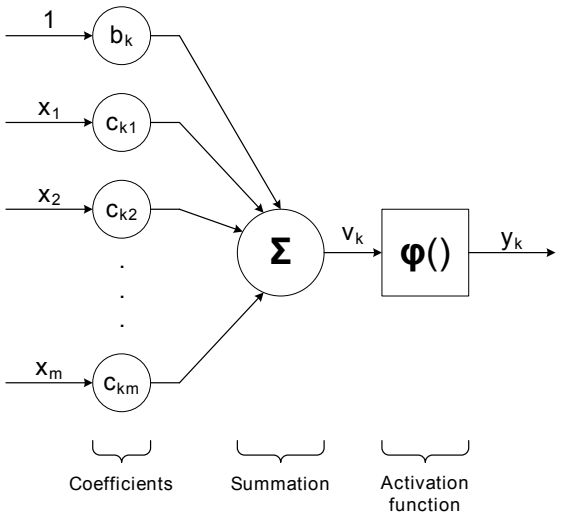


Figure 31: An artificial neuron

Here the x values are the inputs, c the coefficients, v_k an intermediate value and y_k the output. The inputs are multiplied with the coefficients and summed afterwards. Note the b_k, also known as the *bias*, which is similar to a coefficient, the main difference is that this value remains constant, as illustrated by a value of 1 at the input.

The activation function $\varphi()$ is a non-linear function that bounds the summed value. The activation function will prevent a dominating value from an input to propagate through the entire network.
The operation of an artificial neuron can mathematically be expressed as:

$$v_k = \sum_{i=1}^{m} c_{ki} x_i + b_k$$
$$y_k = \varphi(v_k)$$

The operation described by an artificial neuron is applied on a patch of data in an image, to process an entire image many artificial neurons are used to shift

on top of the image. Note that this operation is very similar to a convolution, the main differences are the bias value and the activation function, nevertheless the data patterns are identical to that of a convolution.

7.1 Multi-stage architecture

A CNN is a technique that uses multiple layers to process a single image, for each layer the same operation is performed with different coefficients and dimensions. Figure 32 illustrates the concept of a layer.

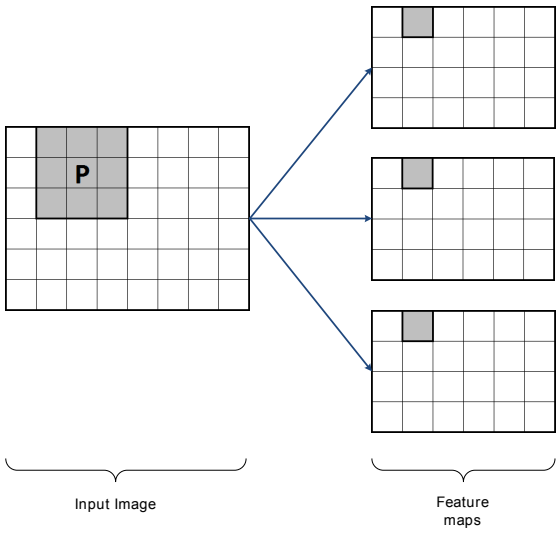


Figure 32: First layer

On the left, an input image can be found, this input is used to compute the multiple feature maps, those are the results from this layer. In this case three feature maps are computed, where each of them is the result of the computation with a different kernel.

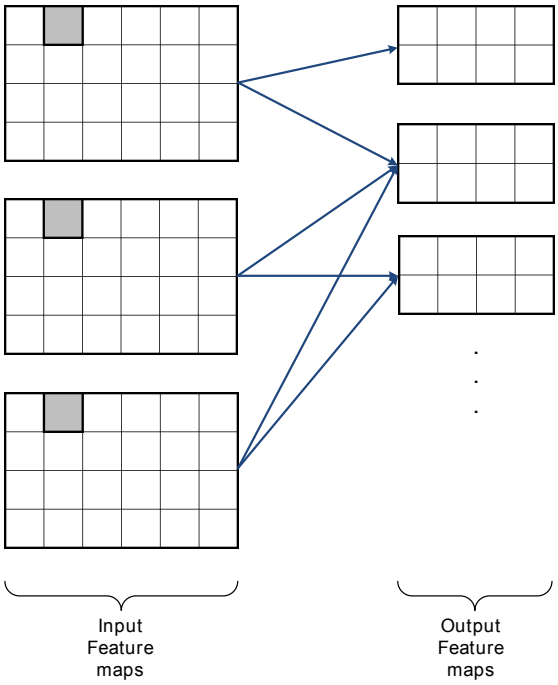


Figure 33: Layers

The power of a CNN lies in the fact that there are multiple layers, whereas the first layer can only detect simple features. The next layers can combine the detected from the previous layers to identify more complex features, this process is also known as *feature extraction*. Figure 33 shows computations between two successive layers, where the left side represents the input feature maps and on the right are the output feature maps. In this figure every arrow represents a computation of an entire feature map with the same kernel. Note that not each output feature map require the same amount of input feature maps, this does happen in practice, though in this example the arrows are added randomly for illustrational purposes. Furthermore the number of layers and number of feature maps within a layer depends on the functionality one wants to achieve.

7.2 Processing Feature maps in parallel

Due to the multi-stage architecture there are many levels of parallelism provided, which allows for efficient processing. To be able to exploit this level of parallelism, the data patterns are required to change drastically compared to conventional convolution based algorithms. The following subsections will explain the concepts in more detail and solutions will be provided allowing large amounts of reuse in addition to *pixel reuse*.

7.2.1 Feature map reuse

From figures 32 and 33 one can find that certain inputs are required to compute multiple outputs. Though the concept of exploiting reuse as explained in section 3 only assumes one output. Fortunately this concept can be generalized, where the input can be transferred using the same method as for conventional convolution operations. Instead of using one kernel to convolve with the input, multiple kernels are used, the only consequence of this modification is that the output becomes more complex. As parts of multiple output feature maps are interleaved.

7.2.2 Intermediate data reuse

The concept of intermediate data is explained using figure 33, from this can be conluded that there are cases where an output feature map requires multiple inputs. The consequence is that if not all the required input feature maps are available, it would result in a large amounts of intermediate data that needs to be transferred to the SDRAM.

For instance, in figure 33 the bottom output feature map is being computed, it would require two input feature maps. Of course these two can be send at the same time in an interleaved fashion, however these two inputs are also required by the middle output feature map, where the middle one requires one more input. To efficiently use this property, the result is that a whole layer is transferred simultaneously to prevent intermediate data.

7.3 CNN interleaved pattern

The conclusion of the previous section for the most efficient data transfers is to transfer parts of all the input feature maps in an interleaved fashion. Where also all the output feature maps are being computed at the same time and interleaved. This concept is illustrated in figure 34.

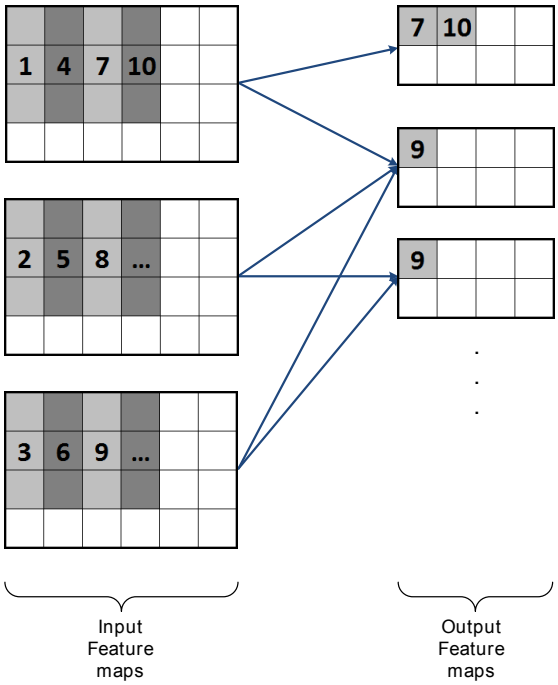


Figure 34: CNN interleaved pattern

The number in the input denotes the order in which the data is presented to the accelerator and the number in the output denotes after which transfer specific data can be computed. In this figure it is assumed that the input and output are directly read from or written to the shuffling unit.

For the shuffling unit this interleaving adds two complications, as multiple inputs are processed at the same time, multiple inputs needs to be read at a time. This implies a larger memory size with respect to a conventional convolution operation. Furthermore the addressing of the data is more complex due to the interleaving, the same complications hold for the output as well.

8 Experimental evaluation

For this section a test setup is created including the shuffling unit, this is done for the *Zedboard (rev c)*. An abstract layout of the setup is presented in figure 35.

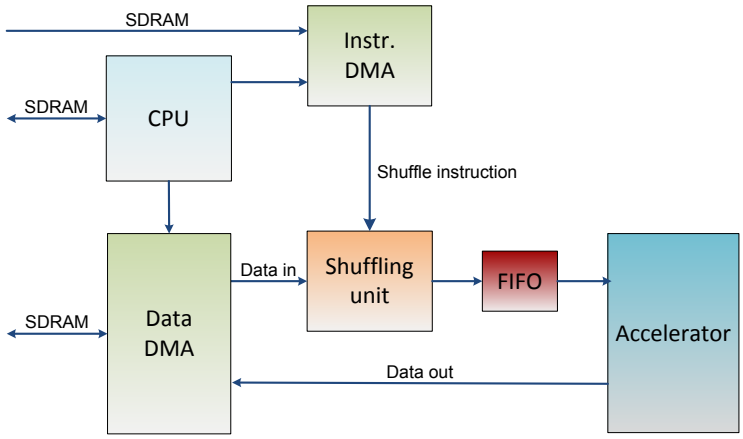


Figure 35: Shuffle unit setup

In section 5 a *scheduler* is presented which supplies DMA and shuffle instructions, to simplify the test setup a CPU is used as a replacement. Also note that two DMA units are used, the *Data DMA* sends data to the shuffling unit and the *Instr. DMA* sends the shuffle instructions, such that the shuffling unit can process multiple instructions independently of the CPU.

This setup operates as follows; first the shuffle instructions are prepared on the SDRAM, once this is done, the *Instr. DMA* is initiated to send the instructions as a stream. Afterwards the CPU controls the *Data DMA* to send the data to the shuffling unit.

Note that only measurements of sending data towards the accelerator are performed. This explains the lack of an output shuffling unit in figure 35.

8.1 Throughput

This section quantifies bandwidth results when supplying different data patterns. Here a single layer of a CNN is assumed, where the computation and subsampling layers are merged [17]. This CNN is depicted in figure 36, there are 3 input feature maps and 3 output feature maps. Even though this CNN is small, it contains all the kinds of reuse as explained in previous sections. In order to gain more insight, different settings, such as image size, kernel size and subsampling are used, however the connectivity will remain the same.

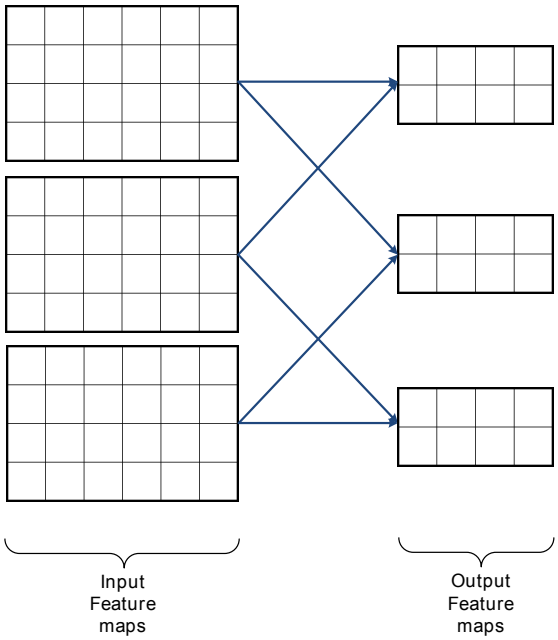


Figure 36: Throughput CNN setup

For this setup multiple sets of parameters are used to illustrate the effects, the sets are listed in table 1. The output column height (H_{OC}) denotes the potential parallelism that could be exploited by the accelerator and increases the level of pixel reuse of the proposed pattern.

ID	**Image size**	**Kernel size**	**Subsampling**	**Output column height (H_{OC})**
1	800x600	5x5	2	8
2	800x600	5x5	1	4
3	800x600	5x5	2	4
4	800x600	3x3	1	4
5	1920x1080	5x5	2	8
6	1920x1080	5x5	1	4
7	1920x1080	5x5	2	4
8	1920x1080	3x3	1	4

Table 1: Overview of parameter sets

A few different methods will be used to perform experiments, these are listed below:

- *Shuffling unit*: the proposed interface is used to transfer the data in the proposed patterns.
- *DMA inefficient*: identical patterns at outputted by the *shuffling unit* are provided directly by the DMA controller.
- *DMA lines*: individual pixels lines of a feature map are transferred as consecutive blocks to the accelerator. The accelerator computes the result of a single input feature map, which causes additional data transfers of the intermediate output result. For this method the accelerator needs to store a few lines of data, resulting in a small local memory.
- *DMA blocks*: whole input feature maps are transferred as consecutive blocks to the accelerator. To prevent intermediate output results, a single output feature map is computed at a time. As a result input feature maps are required to be transferred multiple times. Moreover a large memory size is required since whole feature maps are stored on the local memory.

The selected methods have varying characteristics in terms of reuse, a clear overview of the reuse these methods can exploit is given by table 2, the numbers are based on parameter set 2.

Method	**Pixel reuse**	**Feature map reuse**	**Intermediate data reuse**
Shuffling unit	50.59%	100%	100%
DMA inefficient	50.59%	100%	100%
DMA lines	100%	100%	0%
DMA blocks	100%	0%	100%

Table 2: Reuse overview of a few methods

The table shows that for the proposed schedule 50.59% reuse of the available is used. This value depends on the processing parameters, from which most are application dependent, except for the output column height (H_{OC}). This parameter is also described in section 3 and increases the reuse. Note that H_{OC} only affects data transfers for *shuffling unit* and *DMA inefficient*. The bandwidth results of the methods can be found in figure 37.

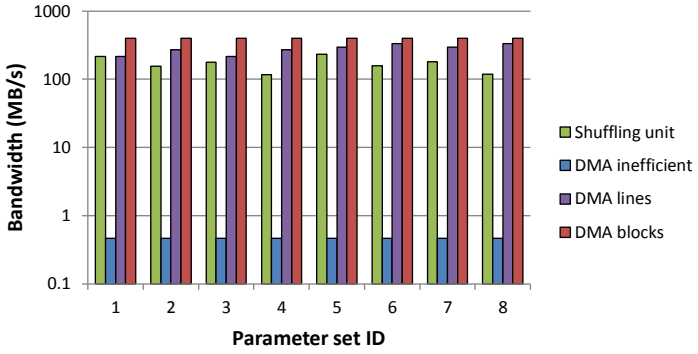


Figure 37: Bandwidth evaluation

For *DMA blocks* entire images are send to the accelerator regardless of the processing parameters, this results in the fact that the bandwidth nearly reaches the theoretical maximum. *DMA lines* sends lines, since more transfers are initiated more overhead is created compared to *DMA blocks*. Furthermore, it produces intermediate computations where the amount depends on the processing parameters, which also influences the bandwidth.

The shuffling unit performs reasonably well in terms of bandwidth, it shows a nearly 2.5x bandwidth reduction compared to *DMA blocks*. This is due to the smaller transfers compared to *DMA blocks* and because *select* forms a bottleneck in the current design of the shuffling unit. As expected, *DMA inefficient* performs extremely bad compared to the other implemenations, mainly due to the huge overhead of the transfers. It shows a difference of 2 orders of magnitude compared to the *shuffling unit*.

From the bandwidth results, the effective bandwidth $\mathrm{BW}_{\mathrm{eff}}$ can also be computed, this can be expressed as:

$$\mathrm{BW}_{\mathrm{eff}} = \mathrm{BW} * \frac{\#bytes_{act}}{\#bytes_{transfer}}$$

Where BW is the measured bandwidth, $\#bytes_{transfer}$ the transferred number of bytes and $\#bytes_{act}$ the number of bytes that actually needed to be transferred. The results are depicted in figure 38.

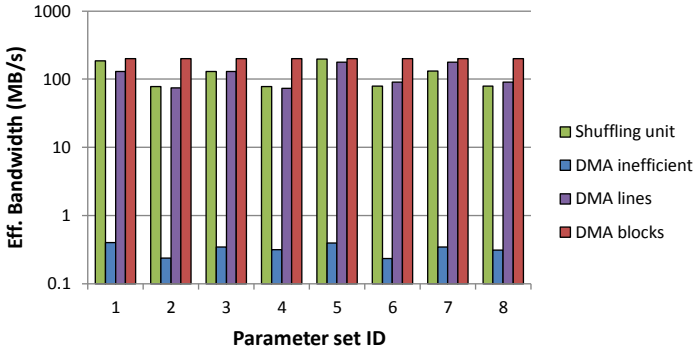


Figure 38: Effective bandwidth evaluation

DMA blocks stays stable since the amount of data transferred does not depend on the parameters. However the effective *DMA lines* decreases compared to *DMA blocks*. Note that the effective bandwidth of *DMA lines* depends in particular on the subsampling factor. From this figure can be concluded that for measurements including subsampling, the efficiency of *DMA lines* nearly approaches *DMA blocks*. Because a higher subsampling factor decreases the number of intermediate data.

The graph also shows that *shuffling unit* moves in the same trend as *DMA lines*, however this is not only due to the subsampling factor. As mentioned before the *select* unit forms the bottleneck, this means that larger transfers to the accelerator is more efficient. In other words, the effect of the bottleneck decreases as the number of lines to be shuffled increases. Furthermore one can find that larger transfers does not increase the effective bandwidth by much, because larger transfers imposes a start-up latency.

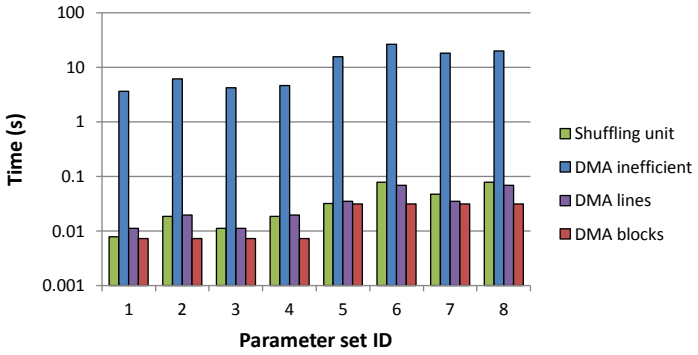


Figure 39: Latency evaluation

In figure 39 the latency is plotted according to the number of bytes send and the bandwidth results. From this figure the same conclusions can be made as for figure 38.

8.2 Area

The resource usage of a setup without shuffling units (figure 2) is compared with a setup including a shuffling unit in table 3. Note that all these designs include at least one DMA and a timer for benchmarking purposes.

	Design without shuffling unit		**Design including shuffling unit**	
Resource	**Usage**	**Percentage**	**Usage**	**Percentage**
Registers	4613	4%	7182	6%
LUTs	3903	7%	5770	10%
RAMB36E1	4	1%	23	7%
RAMB18E1	1	1%	3	1%
DSP48E1	0	0%	3	3%

Table 3: Resource usage of design with and without shuffle unit

Table 3 presents the resource requirements for designs with and without shuffling unit. Here one can see that including a shuffling unit increases the resource usage, this design also includes an additional DMA for providing instructions to the shuffling unit.

The largest increase is the use of RAMB36E1 of about 7%, this was to be expected as a large part of the shuffling unit is memory. The increase of LUTs and registers can be explained by the logic required for the shuffling unit. For the current implementation the *shuffling unit* claims the largest amount of area, however it only uses a small amount of the available resources. The current setup is not realistic as there are no real accelerators included, which will be responsible for the highest resource requirements.

This interface can replace a substantial part of the memory requirements within multiple accelerators. This property can reduce the resource penalty for this interface. Also a few DSP48E1 are required for the shuffling unit, these are used to perform address computations. In table 4 the resources for only the shuffling unit is presented. Note that this is a pessimistic estimation made by *Vivado HLS*.

Resource	**Usage**	**Percentage**
Registers	1286	1%
LUTs	2567	5%
RAMB36E1	0	0%
RAMB18E1	35	12.5%
DSP48E1	3	3%

Table 4: Resource usage of shuffling unit

From this table one can find that the shuffling unit uses a considerable amount of the resources with respect to the entire design. However, the whole design

consists only of the required components to test this shuffling unit, without real accelerators. Note that in the estimation 0 RAMB36E1 are used and 35 RAMB18E1, this is because in the implementation 32 RAMB18E1 are replaced by 16 RAMB36E1.

As there is no hardware generated to implement *DMA blocks* and *DMA lines* the following tables show the block RAM requirement with respect to the memory usage of the different parameter sets.

Image size (Bytes)	**Kernel size**	**Subsampling**	**RAMB18E1**	**Percentage**
800x600	5x5	2	698	249.29%
800x600	5x5	1	1161	414.64%
800x600	3x3	1	1167	416.79%
1920x1080	5x5	2	3026	1080.71%
1920x1080	5x5	1	5040	1800%
1920x1080	3x3	1	5051	1803.93%

Table 5: Minimum memory usage for *DMA blocks*

Table 5 shows the block RAM usage of *DMA blocks*. In the table one can find that the memory usage is extremely high, because whole feature maps are stored in the local memory. None of the tested setups is feasible on the used platform.

Image size (Bytes)	**Kernel size**	**Subsampling**	**RAMB18E1**	**Percentage**
800x600	5x5	2	3	1.07%
800x600	5x5	1	4	1.43%
800x600	3x3	1	2	0.71%
1920x1080	5x5	2	7	2.5%
1920x1080	5x5	1	9	3.21%
1920x1080	3x3	1	7	2.5%

Table 6: Minimum memory usage for *DMA lines*

In table 6 the memory usage with respect to the parameters for *DMA lines* is listed. The block RAM usage requirements is substantially lower compared to *DMA blocks*, since only individual pixel lines need to be stored locally. The number of block RAMs requirement for the *shuffling unit* is only 8x larger than *DMA lines*. Though the *shuffling unit* is much more flexible and is usable for more accelerators.

8.3 Energy

For the energy analysis, *Xilinx power analyzer* [23] is used to accurately estimate the power consumption of the designs. A minimal design of a setup in which consecutive blocks of data is send and received from dummy IP's is compared to a design including the shuffling unit.

No hardware implementations are made for *DMA blocks* and *DMA lines*. Therefore a simple design including a varying memory size is created for estimation purposes. Figure 40 shows the estimated power consumption while varying the amount of local memory.

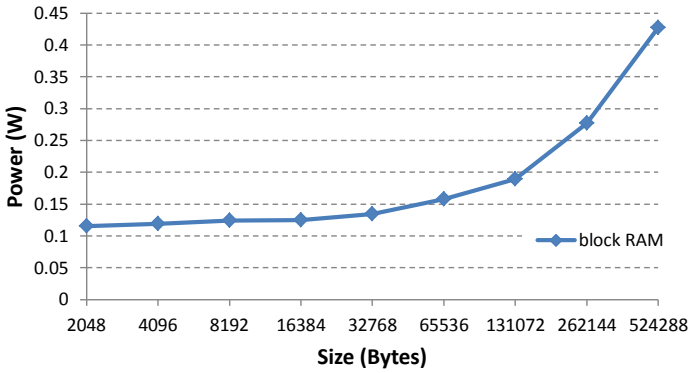


Figure 40: Power consumption for memory units

This figure shows a linear relation between the power consumption in terms of memory size. Where the base is approximately 0.114W and an estimated increase of 0.001W per block RAM for larger values, this will be used for estimations. The following tables will list power consumption estimations for the actual designs.

	Design without shuffling unit		**Design including shuffling unit**	
On-Chip	**Power (W)**	**Percentage**	**Power (W)**	**Percentage**
Clocks	0.021	17.5%	0.027	15.4%
Logic	0.005	4.2%	0.007	4%
Signals	0.007	5.8%	0.012	6.9%
BRAMs	0.010	8.3%	0.052	29.7%
Leakage	0.077	64.2%	0.078	44.6%
Total	0.120	100%	0.175	100%

Table 7: Power consumption evaluation

In table 7 the power consumption of a design with and without shuffling unit is presented. Due to the increase of the amount of logic and a drastic increase in the number of block RAMs, the power consumption including the shuffling unit is approximately 1.5x higher.

From these results the energy consumption can be computed using the results obtained in the throughput section, these are plotted in figure 41. For *DMA lines*, the results are based on the measurements performed by 40 and for *DMA blocks* the results are estimated. Note that these estimations are only on based memory sizes, for these two methods additional logic is required to select the correct data from the memory. In other words, the estimations gives a lower bound for the power consumption of *DMA lines* and *DMA blocks*.

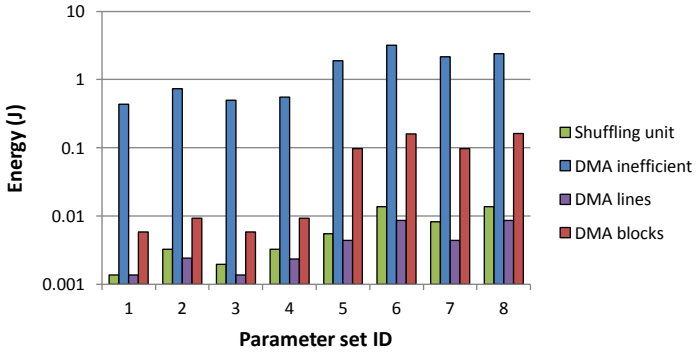


Figure 41: Energy consumption evaluation

In terms of energy consumption, *DMA blocks* and *DMA inefficient* performs extremely bad compared to the other two methods. This is due to the large memory size and inefficient data transfers respectively. Here the energy consumption of *DMA inefficient* is a factor 170x higher compared to the *shuffling unit*. From the figure one can see that *DMA lines* performs approximately 1.4x compared to the *shuffling unit*.

8.4 Conclusions

In this chapter an implementation including a shuffle unit is compared to a few different methods without. This unit shows a reasonable performance in terms of bandwidth, while at the same time the energy consumption and area is also reasonably low. Note that *DMA lines* outperforms the *shuffling unit* on every aspect. The transfers of *DMA lines* are equal transfers to the *shuffling unit*, however no shuffling is performed. So basically *DMA lines* gives an upperbound, however it can only handle certain data patterns and is not programmable at all. Hence the advantage of the *shuffling unit* is its flexibility. Furthermore the memory within *DMA lines* can only be used for a specific task, thus cannot be shared among multiple accelerators, increasing the area requirements.

9 Related work

Many others focus on improving the efficiency of heterogeneous architectures by reducing the off-chip memory requirements. There are many approaches that can help tackle this issue, such as focussing on the accelerator. In [20] a flexible convolution engine is proposed which can be used for convolution-like operations, here the focus is an efficient accelerator. This solution does reduce the required amount of data, though it does not take into account the efficiency of transfers from the SDRAM.

Others solve this issue by separating memories from the accelerator. Where these memories are connected through the accelerator by means of a *crossbar switch*. Where DMA controller are responsible for communication with the off-chip memory, the reason for multiple DMA's is to hide the start up latencies for initiating data transfers [9]. In this architecture there are critical dependencies between the accelerator and the buffers, this is solved by a large *crossbar switch* connecting all the accelerators with all the buffers. These buffers need to be sufficiently large to increase the energy efficiency of the data transfers, all this concludes in a large area requirement.

In [5] data access patterns are analyzed by means of a polyhedral model, from this they identify access patterns in the off-chip memory. Based on this knowledge, these accesses are reorderd, such that these SDRAM requests happen more efficiently. However, the data accesses by the accelerator itself is not changed, this is solved by a separate unit including on-chip memory. This approach increases the efficiency of the SDRAM accesses at the price of a small unit. However they do not consider restructuring the loop structure of the accelerator to better exploit data locality.

Where [19] also analyses access patterns using a polyhedral model, from this as data access structures of the accelerator itself is modified. Where on-chip memory is used to allow the accelerator to reuse data. However, they only consider the amount of data required from the off-chip memory and not its efficiency.

In [18] an accelerator template is given to efficiently compute CNNs, which contains a memory subsystem also supplying data in efficient patterns. However the memory subsystem is less flexible compared to the *shuffling unit*. Furthermore no analysis is given for the efficiency of data transfers to and from the SDRAM.

My proposal is inspired by the work of [7], where the idea is to have a flexible memory structure which can offer a high bandwidth. The addition of my proposal are the *fetch* and *select* units to provide flexible interfaces for multiple accelerator. In contrast to [9] this work assumes flexible reading and writing units around the memories, allowing a higher level of flexibility. Therefore this unit can be shared by multiple accelerators.

10 Conclusions and Future work

The issue identified during the introduction was that there was no efficient interface available to connect a DMA with an image processing accelerator. On one side, there is the DMA which allows very fast data transfers for large consecutive blocks of data, while at the other side the image processing accelerator is able to reuse data when the data is supplied in complex patterns.

In this project a shuffling unit is proposed forming an interface between a DMA and an image processing accelerator. This unit is inspired by the work of [7], which allows a high throughput and a high level of flexibility. In addition a toolflow is proposed that takes care of instruction generation for the shuffling unit, relieving the user from this complex task. This toolflow can generate memory allocations, shuffling instructions and DMA instructions by analyzing the application, based on the hardware parameters of the shuffling units and properties of the accelerator. During this project, the shuffling unit is thoroughly analyzed for performance, area and energy consumption. The analysis is performed using varying image processing parameters and is presented in section 8.

With this work we can provide image processing accelerators with the required bandwidth. Due to the reordering of data accelerators can operate more efficiently, furthermore the flexibility allows easy integration of multiple accelerators. In other words, the shuffling unit allows new low cost and high throughput applications to be implemented on mobile devices for everyday tasks.

10.1 Future work

In this work a case study is shown of a shuffling unit, more work is required to be able to analyse this unit. In the current experiments the benefits of including a shuffling unit is not analyzed in a realistic setup. This is due to the fact that the shuffling unit is tested using dummy accelerators, which simply receives the data without further processing. One can gain more insights about the benefits of a shuffling unit once it is used for multiple real accelerators. In addition to that, the shuffling unit allows a lot of flexibilty, this work only focussed on the convolution operation. So it would be worthwile to explore the flexibility of this interface, also this can then be tested in combination with multiple accelerators.

For the current architecture a simple DMA is used, this requires the CPU to manually set DMA for every instruction. The result is that the overhead created by initiating data transfers and the data transfers themselves happen sequentially. To reduce the overhead the simple DMA could be replaced by a DMA supporting scatter gather mode. This mode allows the user to prepare multiple instructions and send them to the DMA at once. The result is that the DMA can pipeline the initiation and the actual data transfers, resulting in a higher bandwidth. Furthermore it relieves the CPU from having to check on the DMA the whole time.

The current shuffling unit aligns the transfers, this means that different sets of image processing parameters can have a large effect on the bandwidth. This issue can be solved by aligning the transfers, however this also involves more complex addressing of the scratchpad memory.

References

[1] B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable sdram memory controller. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '07, pages 251–256, New York, NY, USA, 2007. ACM.

[2] AXIDMA. *LogiCORE IP AXI DMA*. Xilinx, v6.03a edition, December 2012. PG021.

[3] AXITimer. *LogiCORE IP AXI Timer*. Xilinx, v1.03a edition, July 2012. DS764.

[4] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, CODES '02, pages 73–78, New York, NY, USA, 2002. ACM.

[5] Samuel Bayliss and George A. Constantinides. Optimizing sdram bandwidth for custom fpga loop accelerators. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, FPGA '12, pages 195–204, New York, NY, USA, 2012. ACM.

[6] A. Beric, G. de Haan, J.L. van Meerbergen, and J.A.J. Leijten. *Video Post Processing Architectures*. PhD thesis, University of Technology, Eindhoven, 2008.

[7] A. Beric, J. van Meerbergen, G. de Haan, and R. Sethuraman. Memory-centric video processing. *IEEE Trans. Cir. and Sys. for Video Technol.*, 18(4):439–452, April 2008.

[8] Peter Broere. A memory-centric simd neural network accelerator: Balancing efficiency & flexibility. Master's thesis, University of Technology Eindhoven, August 2013.

[9] Yu-Ting Chen, Jason Cong, Mohammad Ali Ghodrat, Muhuan Huang, Chunyue Liu, Bingjun Xiao, and Yi Zou. Accelerator-rich cmps: From concept to real hardware. In *Computer Design (ICCD), 2013 IEEE 31st Internation Conference on*, 2013.

[10] C. Garcia and M. Delakis. Convolutional face finder: a neural architecture for fast and robust face detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(11):1408–1423, 2004.

[11] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 37–47, New York, NY, USA, 2010. ACM.

[12] Vivado HLS. *Vivado Design Suite User Guide*. Xilinx, v2012.4 edition, December 2012. UG902.

[13] Quoc Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg Corrado, Jeff Dean, and Andrew Ng. Building high-level features using large scale unsupervised learning. In *International Conference in Machine Learning*, 2012.

[14] Micron. *DDR3 SDRAM*, April 2013. MT41J128M16.

[15] OMAP5. *OMAP543x Multimedia Device*. Texas Instruments, 2.0 edition, July 2013. SWPU249X.

[16] M. Peemen, B. Mesman, and C. Corporaal. Speed sign detection and recognition by convolutional neural networks. In *Proceedings of the 8th International Automotive Congress*, page 162170, 2011.

[17] Maurice Peemen, Bart Mesman, and Henk Corporaal. Efficiency optimization of trainable feature extractors for a consumer platform. In Jacques Blanc-Talon, Richard P. Kleihorst, Wilfried Philips, Dan C. Popescu, and Paul Scheunders, editors, *ACIVS*, volume 6915 of *Lecture Notes in Computer Science*, pages 293–304. Springer, 2011.

[18] Maurice Peemen, Arnaud Setio, Bart Mesman, and Henk Corporaal. Memory-centric accelerator design for convolutional neural networks. In *Computer Design (ICCD), 2013 IEEE 31st Internation Conference on*, 2013.

[19] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '13, pages 29–38, New York, NY, USA, 2013. ACM.

[20] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A. Horowitz. Convolution engine: balancing efficiency & flexibility in specialized computing. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 24–35, New York, NY, USA, 2013. ACM.

[21] Thad Starner. Project glass: An extension of the self. *Pervasive Computing, IEEE*, 12(2):14–16, 2013.

[22] Tegra. Nvidia tegra multi-processor architecture. Technical report, February 2010.

[23] XilinxPowerTools. Xilinx, v14.5 edition, March 2013. UG733.

[24] Zynq-7000. Xilinx, v1.6.1 edition, September 2013. UG585.