Memory-Centric Accelerator Design for Convolutional Neural Networks

Maurice Peemen, Arnaud A. A. Setio, Bart Mesman and Henk Corporaal
Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, the Netherlands
Email: m.c.j.peemen@tue.nl, arnaud.arindra.adiyoso@student.tue.nl, b.mesman@tue.nl, h.corporaal@tue.nl

***Abstract*—In the near future, cameras will be used everywhere as flexible sensors for numerous applications. For mobility and privacy reasons, the required image processing should be local on embedded computer platforms with performance requirements and energy constraints. Dedicated acceleration of Convolutional Neural Networks (CNN) can achieve these targets with enough flexibility to perform multiple vision tasks. A challenging problem for the design of efficient accelerators is the limited amount of external memory bandwidth. We show that the effects of the memory bottleneck can be reduced by a flexible memory hierarchy that supports the complex data access patterns in CNN workload. The efficiency of the on-chip memories is maximized by our scheduler that uses tiling to optimize for data locality. Our design flow ensures that on-chip memory size is minimized, which reduces area and energy usage. The design flow is evaluated by a High Level Synthesis implementation on a Virtex 6 FPGA board. Compared to accelerators with standard scratchpad memories the FPGA resources can be reduced up to 13x while maintaining the same performance. Alternatively, when the same amount of FPGA resources is used our accelerators are up to 11x faster.**

I. Introduction

Advances in sensor technology and compute platforms have reduced camera cost and size, and enabled their usage in numerous applications. Well known examples are systems that monitor traffic at road intersections [1], and the cameras that are currently being integrated in glasses as a state-of-the-art wearable support system [2]. There is a huge potential for further increasing the use of cameras, provided that privacy issues can be handled appropriately. This requires that image processing is performed locally, such that privacy sensitive data can be discarded at the camera level. This onboard processing by embedded computer platforms has to satisfy real-time performance requirements while constraining energy usage, and should be flexible enough to support many applications, which is currently not yet the case. General-purpose processors, with the usual image processing algorithms for object detection and recognition, are notoriously inefficient from an energy perspective, and although a custom accelerator can improve compute performance and energy efficiency this mostly reduces the flexibility.

Recent work shows that Convolutional Neural Networks (CNNs) [3] can outperform, and therefore replace, many combined algorithms for vision tasks [4]. Feature extraction and classification are then combined in a single flexible model that can adapt functionality by simple weight updates. Reported applications are face detection [5], traffic sign detection [6], and many others. An embedded camera platform equipped with a CNN accelerator has the performance advantages of dedicated acceleration without sacrificing flexibility for multiple vision tasks.

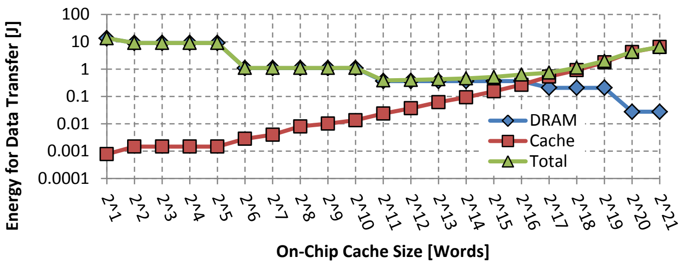

Fig. 1. Data transfer energy for DRAM and On-Chip accesses

A CNN allows sufficient parallelism to execute hundreds of operations in parallel. The current bottleneck in available platforms for efficient utilization of parallelism is data transfer. Evaluating a trained CNN on 720p video involves a large number of convolutions, which in a single layer can require 3.4 billion memory accesses. Without on-chip buffers all accesses are to external memory, requiring huge memory bandwidth and consuming a lot of energy. The number of external accesses can be reduced by on-chip memory that exploits data reuse. Varying on-chip memory size is in essence trading chip area versus memory bandwidth. E.g. 4 MB on-chip memory can reduce the external accesses to 5.4 million.

We estimated the energy of data transfer for varying on-chip memory size with a memory tracing tool [7], and did energy estimation for external [8] and on-chip [9] accesses. From the result, depicted in Figure 1, we conclude that increasing accelerator utilization with more external memory bandwidth is bad for energy. Although on-chip memories can help to increase accelerator utilization, along with the size, the energy consumption per access also increases. In other words, large on-chip memories do not solve the energy problem.

In this paper we present a memory-centric accelerator to improve performance without increasing memory bandwidth. This accelerator uses specialized memories that support the data movement patterns and optimized scheduling for data locality. This combination allows the required buffer size to be minimized and data reuse to be maximized. Specifically, we make the following contributions:

- A configurable accelerator template for CNN, with flexible data reuse buffers. The template supports the different compute patterns in the CNN workload and can be configured to match the external memory bandwidth.
- A memory-centric design flow to synthesize and program the accelerator template. Our design flow uses quick design space exploration to optimize on-chip memory size and data reuse.

- A high-level verification and evaluation of the methodology by FPGA mapping of a speed traffic-sign recognition application.

This enables the synthesis of accelerators that are very efficient in terms of utilization, FPGA resources and external bandwidth requirements. Compared to accelerators with standard scratch-pad memories, the buffer resources can be reduced up to a factor 13 while maintaining performance. In addition, when the same amount of FPGA resources is used our accelerators are up to 11 times faster.

The paper is organized as follows. Section II discusses related work. Section III gives a computational overview of the CNN algorithm. Section IV presents the hardware accelerator template. Section V outlines the complete design flow. We present the evaluation of our method in Section VI, and the conclusions in Section VII.

II. Related work

Acceleration of the CNN algorithm for FPGA or VLSI has been done by others. They have focused on the filtering part, because that represents 90% of the computational workload of the CNN. Systolic implementations seem to be a natural fit, because they are very efficient at that filtering. Recently, two design proposals were built around systolic implementations of the 2d convolution operation: NEC labs developed a convolutional coprocessor [10] and an accelerator design is reported in [11]. However, systolic implementations are very inflexible. Therefore, these proposals had to resort to complex arbitration and routing logic to share inputs and connect outputs of the convolvers to other resources. Furthermore, systolic implementations support only convolutions up to the implemented kernel size, e.g. up to 7x7 convolutions. When a network requires a reduced kernel size, the hardware utilization drops. In addition, the amount of utilized reuse is limited because there is no flexible memory hierarchy with on-chip buffers. As a result, both accelerators require a high memory bandwidth to deliver performance. For example, the accelerator developed by NEC labs requires 3 memory banks, each with an independent memory port for 5.8 GB/s external memory bandwidth, whereas our solution requires an external bandwidth of 150 MB/s.

In our work the kernel size is not restricted, nor does it influence utilization. This is due to having the convolution operations performed by Single Instruction Multiple Data (SIMD) type of Processing Elements (PEs). Moreover, inspired by the work on memory hierarchies for block matching algorithms [12], we use similar BRAM-based multi-bank on-chip buffers to minimize the required bandwidth.

We have complemented this technique with a design flow that uses off-line scheduling to optimize the iteration order for data reuse from the on-chip buffers. Iteration ordering to optimize locality for CPU caches has also been studied by others. The work of [13] demonstrates that nested-loop applications can be automatically optimized for data locality. This idea is further extended with multiple transformations as the polyhedral model from [14]. In [15] the polyhedral model is used to synthesize optimized accelerators by High Level Synthesis. However, time consuming empirical tuning is necessary to optimize the size of the on-chip buffers.

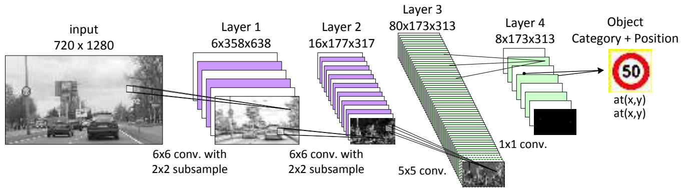

Fig. 2. CNN speed sign recognition in a video frame with merged feature extraction layers

Compared to [15], our accelerator is more flexible because it is programmable with configuration words, such that varying workloads can be used.

III. Recognition with CNN

In this section we outline the feed-forward recognition phase of the CNN algorithm. The recognition mode is performed on embedded platforms with real-time performance and energy constraints, which is the part that our work focuses on. Since training, used to compute the filter kernel coefficients, is performed off-line, it is not considered in this work. As a result we assume that a trained network is available for the feed-forward recognition phase.

A. Algorithm

Traditional vision applications use feature extraction and classification as two distinct steps. The feature extractor largely influences the classification accuracy. Because this extractor is often developed manually, the system performance depends on the ability of the designer to come up with a set of rules to extract an appropriate set of features. In contrast, a CNN combines the two steps in a single trainable model. We outline the model with an example of a speed sign recognition application for a car driver support system [6]. The architecture of the model is depicted in Figure 2. First, the network performs feature extraction by a cascade of trainable convolution and subsample operations. In these layers simple features such as edges are extracted, which are combined in the next layers to detect more complex features such as corners or crossings. Secondly, the features, represented as feature maps, are classified by feed forward neural network layers. The final outputs describe whether there is a sign and to which category it belongs.

A significant portion of the computational workload occurs in the feature extraction layers. By merging the convolution and subsample layers we have demonstrated that the workload is substantially reduced, with equal recognition accuracy [16]. In addition, the required data transfer for the intermediate results is reduced. With the algorithmic modification the network can still be expressed as a series of filters. For example the operation in layer 1 is given in (1) and can be described as a 2d convolution with a step size of 2 positions.

$$y[m,n] = \phi(b + \sum_{k=0}^{K-1} \sum_{l=0}^{L-1} w[k,l] x[mS+k, nS+l]) \quad (1)$$

In the expression, the input image is represented by x, the weight kernel by w, and y is the output feature map. The subsample factor S is used by the indices of x. This general expression is used for all layers in the network. Since classification layers do not use subsampling, the factor S is 1 for layer 3 and 4. The bias value b and the evaluation of the sigmoid function ϕ are used in all layers. The kernel size can vary from layer to layer and is defined by K and L.

In general, a layer in the network converts Q input images $X_1 ... X_Q$ to R output images $Y_1 ... Y_R$. For example, in layer 1 the input image is filtered 6 times by (1), each time with a different weight set. However, in layers 2-4 the network uses multiple input feature maps to compute each output map. For this situation, the definition of (1) changes to:

$$y_r = \phi(b + \sum_{q \in Q} w_{r,q} * x_q) \tag{2}$$

The operator * in (2) represents the parameterized convolution operation of (1) with subsample factor.

B. Practical implications

The huge amount of data reuse in the algorithm has practical implications that should be considered before implementation. For example, the ordering of operations and selection of parallelism are key parameters that influence data transfer and computational resource usage. The parameter design-space we consider contains 4 levels:

- Convolution kernel level, the multiplication of a small window in x with weights in w.
- Neuron level, $y[m, n]$ values in a feature map see (1).
- Input image level, input feature maps that should be combined as shown in (2).
- Output image level, multiple output feature maps in a layer.

Each level in the algorithm has a different data transfer pattern and contains an amount of data reuse. In addition, different layers or network configurations change the properties of the 4 levels. For example, in figure 3 the first layers have much parallelism at the kernel level. Layer 4, on the other hand, has no kernel level parallelism because the kernel size is 1x1. As a result, an accelerator must be flexible to be efficient for each CNN layer.

To refer accurately to the different operations in a CNN, a nested-loop description of each layer is used. An example description of Layer 3 of the speed sign network is given in Listing 1. Outer loop r represents the output feature maps and loop q the connected input feature maps. Additionally, loop m and n describe the feature map rows and columns, respectively. Finally, loop k and l describe the rows and columns, respectively, of the convolution operation.

```
for(r=0; r<R; r++){            //output feature map
  for(q=0; q<Q; q++){          //input feature map
    for(m=0; m<M; m++){        //output values in a map
      for(n=0; n<N; n++){
        if(q==0){Y[r][m][n]=Bias[r];}
        for(k=0; k<K; k++){   //kernel operation
          for(l=0; l<L; l++){
            Y[r][m][n]+=W[r][q][k][l]*X[q][m+k][n+l];
          }
        }
        if(q==Q-1){Y[r][m][n]=sigmoid(Y[r][m][n]);}
      }
    }
  }
}
```

Listing 1. Example loop-nest representing Layer 3 of the speed sign CNN

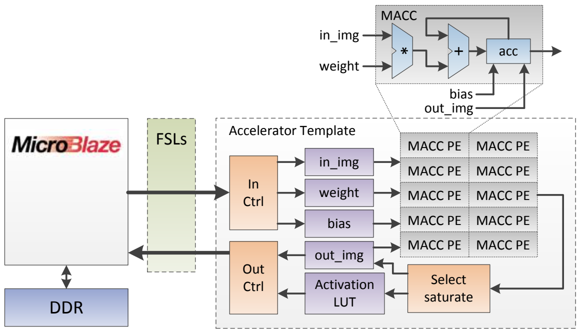

Fig. 3. CNN accelerator template connected to a host processor for control.

IV. CNN Accelerator Template

The accelerator template is presented in a design flow for Xilinx FPGAs. For control purposes the accelerator is connected to a MicroBlaze host processor. This processor sends configuration parameters such as the number of feature maps and their dimensions. When the accelerator is in operation, the host processor streams data in and out though Fast Simplex Link (FSL) connections. A high-level overview of the accelerator template is given in Figure 3.

Our accelerator template possesses two types of flexibility: it is configurable, and programmable. Configurability is used to design accelerators that are matched to the available platform resources. Programmability is used to maintain performance with the varying workload over CNN layers. To achieve these targets a cluster of SIMD type of Multiply Accumulate (MACC) PE is used to accelerate the convolutions. Each PE sequentially computes a neuron value in a feature map. Hence, iterations of loop m and n are divided over different PEs, and iterations of loop k and l are performed by one PE. To maximize utilization of the PE cluster we use flexible reuse buffers that exploit the predictable data access patterns in a CNN. The sigmoid activation functions are evaluated by lookup tables.

A. Flexible memory subsystem

The memory subsystem facilitates the flexibility and increases communication bandwidth by exploiting data reuse in memory access patterns. This is implemented by a series of dual ported FPGA Block RAMs (BRAMs). In Figure 4 the communication is depicted in more detail. The example shows four PEs that compute neighboring output results in a row of

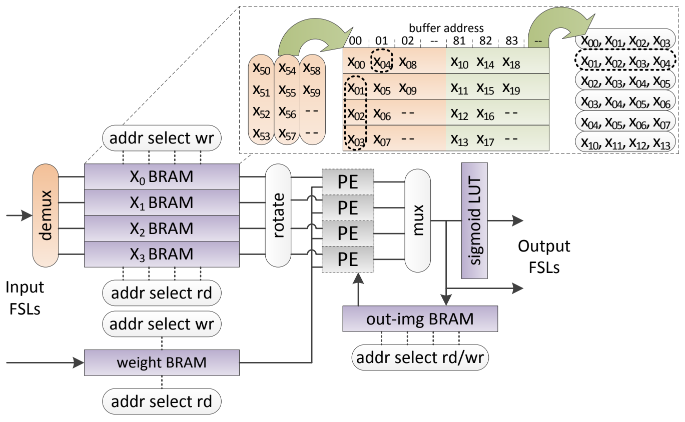
buffer address
00 01 02 -- 81 82 83 --
X_{00} X_{04} X_{08} X_{10} X_{14} X_{18}
X_{01} X_{05} X_{09} X_{11} X_{15} X_{19}
X_{02} X_{06} -- X_{12} X_{16} --
X_{03} X_{07} -- X_{13} X_{17} --
X_{50} X_{54} X_{58}
X_{51} X_{55} X_{59}
X_{52} X_{56} --
X_{53} X_{57} --
$X_{00}, X_{01}, X_{02}, X_{03}$
$X_{01}, X_{02}, X_{03}, X_{04}$
$X_{02}, X_{03}, X_{04}, X_{05}$
$X_{03}, X_{04}, X_{05}, X_{06}$
$X_{04}, X_{05}, X_{06}, X_{07}$
$X_{10}, X_{11}, X_{12}, X_{13}$
addr select wr
demux
X_0 BRAM
X_1 BRAM
X_2 BRAM
X_3 BRAM
rotate
PE
PE
PE
PE
mux
sigmoid LUT
Output FSLs
Input FSLs
addr select rd
addr select wr
out-img BRAM
addr select rd/wr
weight BRAM
addr select rd

Fig. 4. Programmable buffer architecture with interconnect system. The depicted input feature map buffer is configured as a cyclic row buffer.

a feature map (parallel execution of four iterations of loop n). Because weight values are shared in a feature map, these are broadcasted to the PEs. The small set of values in the weight BRAM is reused for the other neurons in the feature map.

1) Row based storage: In addition to the weights, each cycle the PE require a series of input feature map values, x in (1). If there is no subsampling, the four PEs first require $[x_{00}\ x_{01}\ x_{02}\ x_{03}]$, secondly $[x_{01}\ x_{02}\ x_{03}\ x_{04}]$, etc. The read bandwidth to send the values of x to the PEs is created by using four BRAM banks. The writing controller is configured to store successive values cyclic over banks. As depicted in Figure 4, the read controller is configured to read four addresses that are rotated in the correct order. Since the buffer for 5x5 convolution should only hold 5 input lines, the values of the first line that are used can be overwritten by new values located 5 lines further on in the feature map. To support the read and write patterns, modulo addressing is used in two dimensions: for the banks, and for input feature map lines.

2) Compute feature maps in parallel: The number of PEs and BRAM banks can be increased to exploit more parallelism. For large numbers the rotate logic would become complex. As a result, parallelism is exploited over output feature maps that share input feature maps (parallelism over iterations of loop r). This configuration of the template is depicted in Figure 5. The number of weight BRAMs increases because each output feature map requires a different weight kernel. Since input feature map values are shared, these are broadcasted to both groups of PEs.

3) Partial column based storage: Another modification to the template involves parallelism over neighboring output results of a column instead of a row (parallelism over iterations of loop m). In this configuration, the PEs require patterns such as $[x_{00}\ x_{10}\ x_{20}\ x_{30}]$, and $[x_{10}\ x_{20}\ x_{30}\ x_{40}]$, etc. To support this pattern small portions of columns are loaded in the input feature map buffer. Portions of other required input feature maps are stored at deeper positions of the buffer. In Figure 5 this is depicted by column $[y_{00}\ y_{10}\ y_{20}\ y_{30}]$. Each extra input feature map in the memory reduces the number of communications for the temporal result y_r in (2) by reuse of Y over iterations of loop q in Listing 1.

4) Column based storage with subsampling: Due to the flexibility of the reuse buffers subsampling factors are directly supported. If (1) contains a subsample factor $S > 1$ the

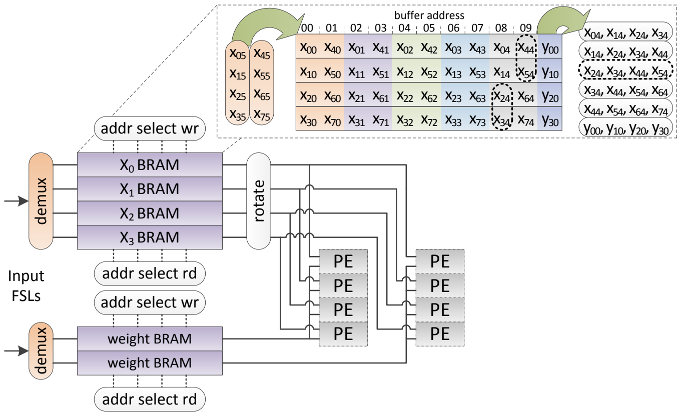

Fig. 5. Accelerator template that exploits parallelism of feature maps that share input maps. The reuse buffer stores portions of feature map columns.

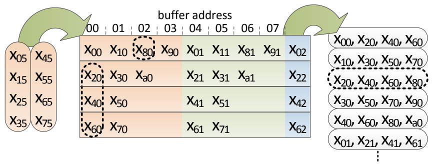

Fig. 6. Reuse buffer addressing scheme that supports reading with a subsample factor of two.

parallel feature map neurons are not direct neighbors. As a result a different pattern must be send to the PEs, e.g. S=2, $[x_{00}\ x_{20}\ x_{40}\ x_{60}]$, $[x_{10}\ x_{30}\ x_{50}\ x_{70}]$, etc. Figure 6 shows one of the possible writing and reading schedules that can be used to implement a subsample factor of 2.

B. Parameterized HLS Templates

SIMD type of PEs, in combination with programmable reuse buffers, result in a very flexible accelerator. Multiple parameters of a CNN are programmed in the execution schedule, which can change during execution. Examples are: convolution kernel size (loops k and l), feature map size (loops m and n), number of input feature maps (loop q), and subsample size. These parameters ensure that the accelerator supports a variety of layer configurations. In addition it is possible to configure parameters of the template that influence the hardware instantiation. These parameters are fixed after synthesis; examples are the number of PEs, connectivity, supported addressing modes, buffer configuration, and buffer depth.

For quick evaluation of accelerator configurations we synthesize the template with the Vivado HLS (AutoESL) tools from Xilinx. This allows us to use a high-level accelerator description in C, and to use HLS directives to specify the hardware configuration. The configuration involves partitioning of the flexible memory banks, or pipelining of operations in the MACC PE. Due to size constraints of the paper we do not further specify these directives, but refer to the documentation [17] for more information. Modulo addressing logic is implemented as variable increments and a test for the used modulo value.

V. Design Flow

If reuse buffers have enough content the MACC PEs can continue processing. Because bandwidth towards the buffers is relatively small, reuse of data is exploited to increase bandwidth for reading from the buffers. Our design flow selects the best computation schedules to maximize data reuse for a buffer size restriction. This goal is achieved by loop transformations such as interchange and tiling on the nested-loop description of a CNN layer. A schematic overview of the design flow is depicted in Figure 7. This section briefly outlines the design flow, detailed information can be found in our extended technical report [18].

1) Use memory access function for modeling: The first steps in the design flow involve the detection of data reuse patterns in a CNN layer. This is performed by modeling memory access patterns as *affine access functions* (linear functions

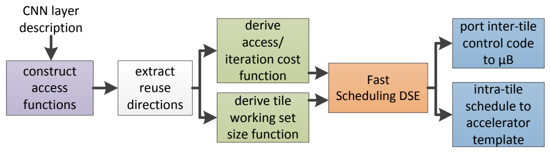
CNN layer description
construct access functions
extract reuse directions
derive access/ iteration cost function
derive tile working set size function
Fast Scheduling DSE
port inter-tile control code to µB
intra-tile schedule to accelerator template

Fig. 7. Design space exploration flow for locality optimized schedules.

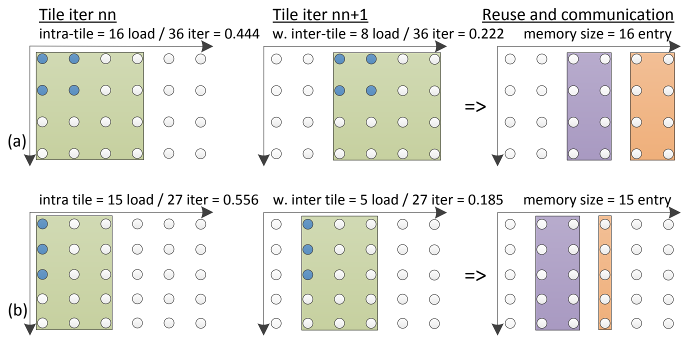
Tile iter nn
intra-tile = 16 load / 36 iter = 0.444
Tile iter nn+1
w. inter-tile = 8 load / 36 iter = 0.222
Reuse and communication
memory size = 16 entry
(a)
=>
intra tile = 15 load / 27 iter = 0.556
w. inter tile = 5 load / 27 iter = 0.185
memory size = 15 entry
(b)
=>

Fig. 8. Memory access/iteration and working set size for different schedules for implementation of 3x3 convolution.

of variables and loop iterators with a constant) [13]. Tools such as PET [19] automatically extract such representations. We convert access functions into two analytical models that model the effects of a possible schedule. The first computes the working set size in the accelerator. The second is a cost function that models the number of external accesses. The combination of the two is used to optimize intra- and inter-tile reuse for a bounded working set size.

2) Memory effects of different schedules: The models are clarified by the effects of different schedules, as depicted in Figure 8. It illustrates the data access patterns when reading an input image for a 3x3 convolution. The real design flow also models transfer and storage of coefficients, intermediate results, and outputs. Figure 8(a) illustrates a schedule that starts four PEs in a 2x2 grid. The bullets represent memory locations, and the blue ones are the start addresses for the four PEs. To compute the four results, the working set size spanned by the green box is required; this defines the accelerator memory size requirement. In addition, we model the communications per iteration, defined as: *the number loads divided by the compute iterations*. As shown, a reduction of communications is achieved when successive tiles reuse their overlapping values. This reuse is illustrated with a purple box, and the remaining communication by an orange box. Figure 8(b) shows a better schedule regarding data reuse by maximizing the overlap between successive tiles. Further more, the memory footprint is reduced to 15 values instead of 16. This example shows that the ordering of iterations has a substantial effect on communications and local memory size.

3) Scheduling Design Space Exploration: The space of possible accelerator schedules is large, especially if the allowed buffer size is increased. This search space increases exponentially with the number of CNN dimensions (or loop dimensions in Listing 1). In practice the optimization problem is manageable because the memory size constraint is a

TABLE I. OPTIMAL SCHEDULE PER LOCAL MEMORY SIZE CONFIGURATION

mem. size	32	64	128	256	512	1k	2k	4k
tile dir.	n	n	n	n	n	n	n	n
r	1	1	3	3	3	7	7	14
q	3	1	1	2	4	4	8	8
m	1	3	4	5	5	9	10	20
n	1	1	1	1	1	1	1	1
k	1	5	5	5	5	5	5	5
l	5	5	5	5	5	5	5	5
access/iter.	0.356	0.176	0.109	0.066	0.046	0.029	0.019	0.014
set size	31	63	127	255	495	1023	2030	4040

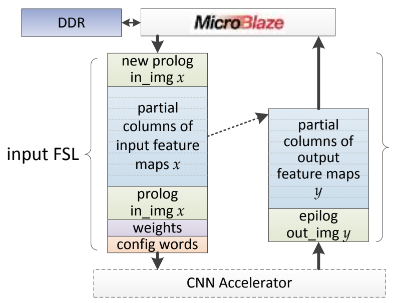

Fig. 9. Schematic overview of the pipelined communication stream

monotonic function. As a result it is not necessary to check all configurations, because many are invalid regarding memory size. On a standard laptop the optimization procedure for a series of memory sizes, as depicted in Table I, is performed in two seconds. The table present the optimal schedules that minimize external accesses for layer 3 of the speed sign recognition CNN. Each column represents a schedule that corresponds to a memory size constraint. Furthermore, it contains the modeled number of external communications per iteration.

4) Pipelined Tile Processing: The proposed schedules are implemented as a pipelined communication stream. A control part of the schedule runs on a MicroBlaze host processor, it communicates the required data for a tile to the accelerator. Because communication goes through FIFO based FSL connections, it is possible to overlap communication and computation. A graphical representation of the communication flow is depicted in Figure 9. The control part starts by loading a parameter set in the accelerator. Next, the prolog part is communicated: it fills the buffers with the data that is required before processing can start. Finally, a number of stream communications is performed, and as a result the accelerator returns the results. This sequence is repeated until a CNN layer is complete and a new parameter set must be communicated.

A. Accelerator Instantiation

To instantiate the accelerator template with the obtained schedules, an HW/SW integration flow is used. Figure 10 gives an overview of the flow that configures the HW template of Section IV. The top left part contains the scheduling design space exploration, from which the optimal schedules are used

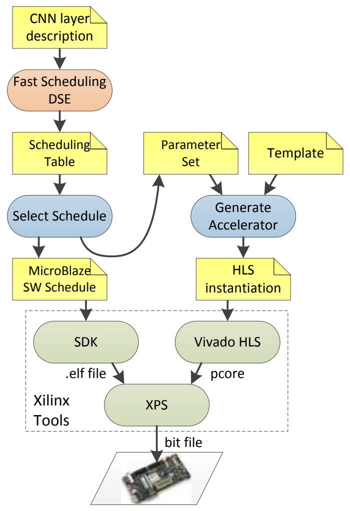

Fig. 10. Accelerator design, mapping and configuration flow.

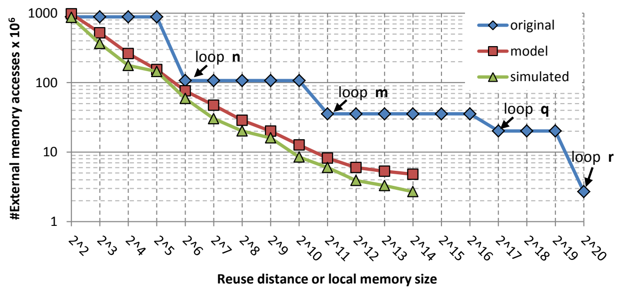

Fig. 11. External access histogram for different schedules of CNN layer 3

to select the template parameters. The parameter set and the hardware template are manually converted into a HLS instantiation of the accelerator. In the left part of the design flow the selected schedule is manually converted to control software for the MicroBlaze host processor.

VI. Experimental Evaluation

For evaluation of our method we analyze the effect of different schedules on the external memory accesses. The SLO [7] reuse profiling tool is used to compare the original schedule of Listing 1, with the optimized schedules of Table I. The results are depicted in Figure 11, for the original iteration order the external communications are reduced when a new loop level fits in the buffer. The memory accesses estimations of the model, and the simulations of the schedules are plotted in the same figure. The graph indicates that our models are somewhat pessimistic compared to the simulated results. However, the substantial reduction of external communications and buffer size is remarkable.

A. Accelerator performance

To quantify the performance of the improved schedules, we compare with the original schedule. The differences are measured by mapping with the accelerator template to a Xilinx ML-605 Virtex 6 FPGA board. The system clock frequency of presented results is 150 MHz.

1) Original schedule: For the mapping, two degrees of freedom are evaluated: the amount of parallelism (MACC PE), and local buffer size. Parallelism influences the compute time, and buffer size the data transfer time. For the buffer size three interesting design points are extracted from Figure 11. These points mark buffer sizes where reuse of a new loop level is utilized; they are annotated as loop m, q, and r.

Figure 12 shows the execution time of the mappings versus the number of instantiated MACC PEs. Obviously, the execution time does not scale well with the amount of parallelism that is used, and in particular the data transfer time is responsible. In addition, the mapping that utilizes all reuse with big local buffers (loop r) does not scale well beyond 8 MACC PEs. This is due to big local memory structures, which have significant read and write delays.

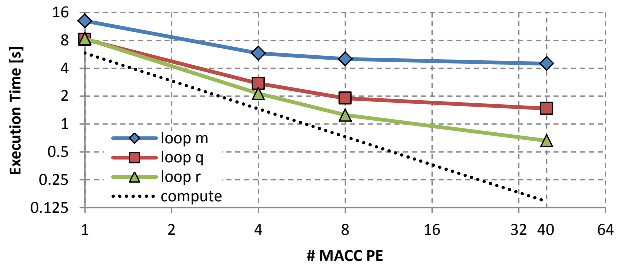

Fig. 12. Accelerator performance for the original iteration order

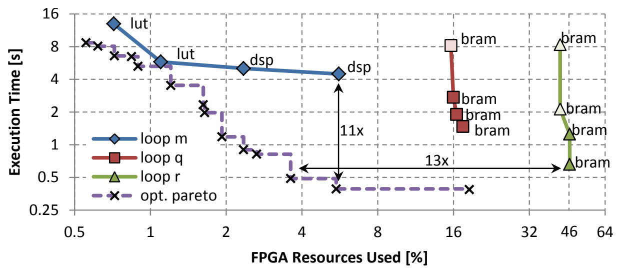

Fig. 13. FPGA resource utilization for the original iteration order

The FPGA resource usage for the three accelerator configurations are depicted in Figure 13. The resource usage is defined as: *the maximum of the percentages for instantiated DSPs, BRAMs, LUTs and flip-flops of the total available for each category*. The text label next to each design point indicates which resource is critical. The accelerator designs of loop q and r, are dominated by BRAMs because the local buffers store 55000 and 500000 elements, respectively.

2) Optimized schedules: Figure 14 shows the execution time of mappings generated with locality optimized schedules. Each line in the graph represents a column of Table I. The scaling in execution times shows that more buffer size is required when the number of PEs is increased. Since this reduces the compute time, it becomes impossible to overlap data transfer, and as a result data transfer becomes dominant. Since, the external bandwidth is known, it is possible to predict data transfer problems in advance. Table II shows the compute time for one single MACC PE and predicted data transfer time for an accelerator with 32 or 64 entry local buffer sizes. If data transfer time is smaller than compute time, total execution time is close to the theoretical optimum. This happens because the accelerator overlaps data transfer and compute time. This also holds for mappings with large local memories e.g., for the 4096 entry mapping compute and transfer times are balanced for 20 PEs, but for 40 PEs there is an imbalance that saturates execution time.

TABLE II. The modeled balance between compute and transfer time

Compute	Data transfer		Data transfer	Compute	
1 PE	acc 32	acc 64	acc 4096	20 PE	40 PE
5828 ms	8225 ms	4066 ms	323 ms	291 ms	146 ms

The FPGA resource usage for optimized accelerators is depicted in Figure 15. For these designs, the amount of required BRAMs, DSPs, LUTs and flip-flops are better balanced. To compare resource efficiency with the original iteration order, the pareto-points of Figure 15 are plotted in Figure 13. The comparison shows that locality optimized accelerators can

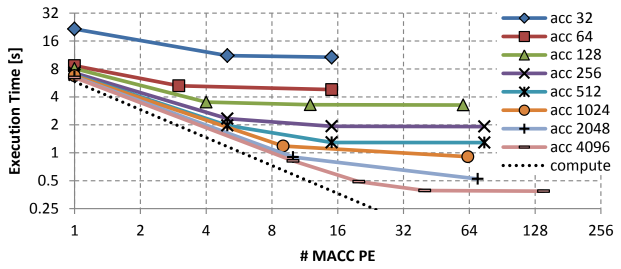

Fig. 14. Accelerator performance for optimal iteration orders

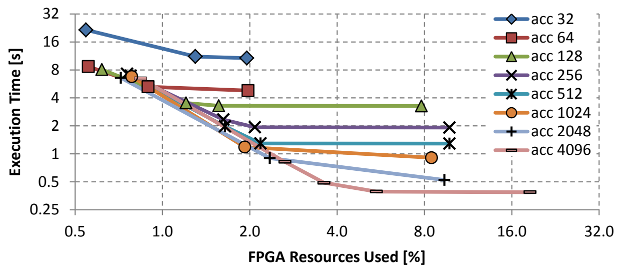

Fig. 15. FPGA resource utilization for optimal iteration orders

gain up to 11x better performance with equal resource cost. Furthermore, locality optimized accelerators require up to 13x fewer resources with equal performance.

B. Memory bandwidth limitations

Due to the small memory bandwidth of the MicroBlaze core, the accelerator of the previous section does not scale beyond 20 MACC PEs. By packing multiple memory accesses in larger words, the theoretical bandwidth is increased from 37.5 MB/s to 150 MB/s. For this case, the 32-bit width of the FSL link is the limiting factor. The dashed lines in Figure 16 show the scaling improvement that is achieved by packing memory accesses. With 150 MB/s, the accelerator can efficiently utilize 140 MACC PE for the convolutions.

VII. Conclusion

In this paper we demonstrated that a memory-centric design method for CNN accelerators can achieve a substantial performance increase. This increase is mainly caused by efficiency improvements in the data access patterns. As a result, resource costs such as external memory bandwidth and FPGA resources are modest. By using an HLS accelerator template with a flexible memory hierarchy, we ensure that a broad range of CNN configurations is supported. In addition, our design flow for on-chip buffer management is novel, since our optimization flow quickly analyzes the possible scheduling alternatives and returns the best schedules that minimize external communications and buffer size. The accelerator template and the design-flow are combined into an end-to-end solution that drastically reduces development time for efficient CNN accelerators for an embedded vision platform. As a result we were able to analyze and verify our method extensively on a Virtex 6 FPGA board with the Xilinx tool chain. For the analysis we used a CNN vision application for speed sign recognition on a 720p HD video. The CNN model does not change over varying vision applications. Only the network configuration and weight coefficients change if a different application is started. Due to the flexibility of our template and mapping flow, CNN accelerators prove to be suitable for many types of vision tasks. This opens new possibilities for flexible smart camera platforms that can perform a range of vision tasks on efficient hardware.

A. Future work

Although the presented design flow is complete, some elements need to be added. This mainly involves engineering work to increase the ease of use. For example the construction of the HSL instantiation of the accelerator requires manual code transformations of the template and the parameter set. In our future work the generation of HLS instantiations will be automated by C++ template functions. Furthermore, the external memory transfers of the platform are facilitated by a MicroBlaze soft core. In our future work these transfers are performed by a dedicated DMA controller in the system. This will further improve scaling to use many more PEs efficiently.

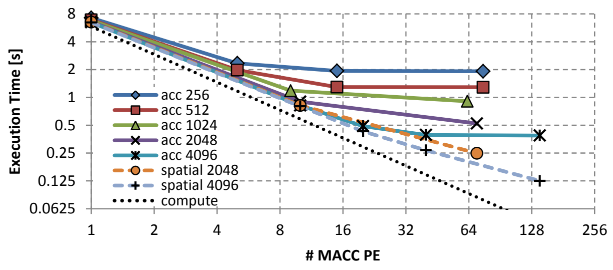

Fig. 16. Accelerator performance with memory access grouping

References

[1] S. Kamijo, Y. Matsushita, K. Ikeuchi, and M. Sakauchi, "Traffic monitoring and accident detection at intersections," *Intelligent Transportation Systems, IEEE Transactions on*, vol. 1, no. 2, pp. 108–118, 2000.

[2] T. Starner, "Project glass: An extension of the self," *Pervasive Computing, IEEE*, vol. 12, no. 2, pp. 14–16, 2013.

[3] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, 1998.

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

[5] 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*, vol. 26, pp. 1408–1423, 2004.

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

[7] K. Beyls and E. D'Hollander, "Refactoring for data locality," *Computer*, vol. 42, no. 2, pp. 62–71, feb. 2009.

[8] K. Chandrasekar *et al.*, "Drampower: Open-source dram power & energy estimation tool," 2012.

[9] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *Proc. MICRO*, 2007, pp. 3–14.

[10] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *ISCA*, 2010, pp. 247–257.

[11] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Proc. Embedded Computer Vision Workshop*, 2011.

[12] A. Beric, J. van Meerbergen, G. de Haan, and R. Sethuraman, "Memory-centric video processing," *IEEE Trans. Cir. and Sys. for Video Technol.*, vol. 18, no. 4, pp. 439–452, 2008.

[13] M. Wolf and M. Lam, "A data locality optimizing algorithm," in *Proc. PLDI '91*, 1991, pp. 30–44.

[14] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," *SIGPLAN Not.*, vol. 43, no. 6, pp. 101–113, jun. 2008.

[15] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *Proceedings of the ACM/SIGDA international symposium on FPGA*, 2013, pp. 29–38.

[16] M. Peemen, B. Mesman, and H. Corporaal, "Efficiency optimization of trainable feature extractors for a consumer platform," in *ACIVS*, ser. LNCS, vol. 6915, 2011, pp. 293–304.

[17] Xilinx, *Vivado Design Suite User Guide, High-Level Synthesis, UG902*.

[18] M. Peemen, B. Mesman, and H. Corporaal, "Optimal iteration scheduling for intra- and inter tile reuse in nested loop accelerators," Eindhoven University of Technology, Tech. Rep. ESR-2013-03, 2013.

[19] S. Verdoolaege and T. Grosser, "Polyhedral extraction tool," in *Proc. of IMPACT'12*, 2012.