Optimal Tiling Strategy for Memory Bandwidth Reduction for CNNs

Leonardo Cecconi[1], Sander Smets[2](✉), Luca Benini[1], and Marian Verhelst[2]

[1] DEI, University of Bologna, Bologna, Italy
leonardo.cecconi@studio.unibo.it, luca.benini@unibo.it
[2] ESAT-MICAS KU Leuven, Leuven, Belgium
{sander.smets,marian.verhelst}@esat.kuleuven.be

Abstract. Convolutional Neural Networks (CNNs), are nowadays present in many different embedded solutions. One of the biggest problems related to their execution is the memory bottleneck. In this work we propose an optimal double buffering tiling strategy, to reduce the memory bandwidth in the execution of deep CNN architecture, testing our model on one of the two cores of a Zynq®-7020 embedded platform. An optimal tiling strategy is found for each layer of the network, optimizing for lowest external memory $\rightleftharpoons$ On-Chip memory bandwidth. Performance test results show an improvement in the total execution time of 50% (cache disabled/34% cache enabled), compared to a non double buffered implementation. Moreover, a 5x lower external memory $\rightleftharpoons$ On-Chip memory double buffering memory bandwidth is achieved, with respect to naive tiling settings. Furthermore it is shown that tiling settings for highest OCM usage do not generally lead to the lowest bandwidth scenario.

1 Introduction and Related Works

Nowadays, mobile devices aim to implement Neural Networks for tasks like face recognition, fault detection, speech processing and so on. The great number of computations involved in this kind of algorithms requires an efficient usage of the limited available resources in order to keep performance at an acceptable level. [3] or [6] look towards dedicated hardware on FPGA, while other works like [5] target multi-core mobile solutions (e.g. the Odroid-XU platform) to exploit parallelism. Despite significantly enhancing the performance of Neural Networks, design tools for these solutions often lack of an optimal strategy to tackle the memory bottlenecks. Some level of transfer-computation pipelining can be introduced, but fails to get performances close to hand crafted solutions. More structured reuse techniques are a common way to reduce the external bandwidth. Works like [10,11] address this issue by means of loop transformations and data tiling. In [12] for example, a tiling strategy is proposed for two-dimensional data processing in multi-core architectures. The effectiveness of these techniques has been assessed for both non-cached and cached scenarios, where data tiles become an easier fit for cache blocks [8].

J. Blanc-Talon et al. (Eds.): ACIVS 2017, LNCS 10617, pp. 89–100, 2017.
https://doi.org/10.1007/978-3-319-70353-4_8

In this work, a double buffering system will be implemented by means of a Direct Memory Access (DMA) on-chip unit, to transfer data between the external Double Data Rate (DDR) memory an the On-chip memory (OCM). The contribution of this work is in developing a data tiling strategy for the double buffering system, tailored to specifically target a CNN architecture, taking into account data dependencies, changing shape of the layers throughout the network and underlying DMA transfer specifications and characteristics. Modeling the DDR $\rightleftharpoons$ OCM data transfer costs, optimal tiling parameters are found, by means of an exhaustive search over a constrained search space. This proves to improve performance both in cached and non-cached scenarios. Furthermore a comparison between the performances of naive tiling parameters and optimal ones, for similar OCM size usage, will lead to the evidence that using the most of the available OCM space does not necessarily lead to the best performances in terms of external memory bandwidth.

In Sect. 2 of the paper, an overview of CNN architecture and operations will be presented, together with a brief description of the network chosen for testing. Section 3, will focus on the embedded platform used for the tests and the double buffering system and tiling mechanisms. Following, Sect. 4 will be dedicated to introduce and develop the Optimal Tiling model and its solution for minimum data bandwidth. Section 5 is dedicated to give out details about the implementation on the embedded target, focusing on the programming of the DMA controller. Finally in Sect. 6 the results of the tests are reported and discussed.

2 Network Structure

2.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a specific type of feed-forward artificial Neural Network, proven to be very effective in tasks like image recognition and classification. The neurons of these networks are arranged in three dimensions and every layer of the network transforms one volume of activations to another one by means of a differentiable function [4].

For every layer of the network, the input features map is convolved with a stack of 2D kernels to obtain another 3D feature map and repeat the process. The 3D convolution is the operation that gives this type of Network its name. According to the purpose of the CNN, the last few layers of the network can be *fully-connected*, for classification or even *deconvolutional layers* if the goal is to reconstruct some sort of output image or map.

Focusing on the 3D convolution operation, the arguments required for this function are essentially four: an input 3D map of features, a set of kernels of weights (3D as well), a set of biases and a 3D output map to accumulate and store the computed features.

Let $w_{i,j}^{n,m}$ be the (i, j) element of the kernel referred to input map n and contributing to output map m. Let $x_{s,t}^{n}$ be the element in position (s, t) and belonging to input map n. Finally, let $y_{s,t}^{m}$ the element in position (s, t) and belonging to output map m and b^m the bias to be added to the elements of output map m. The 3D convolution can then be expressed as follows:

$$y^m_{\frac{s}{str},\frac{t}{str}} = actf\left(\sum_{n=0}^{N-1}\sum_{i=-K_s/2}^{K_s/2}\sum_{j=-K_s/2}^{K_s/2} w^{n,m}_{i,j}\cdot x^n_{s-i,t-j} + b^m\right)$$

where actf is the activation function (usually relu or tanh) and where the kernels are considered squares of size $K_s \times K_s$. The s and t coordinates are update with stride str so that $y^m_{\frac{s}{str},\frac{t}{str}}$ only integer indexes occur.

2.2 Test Network

Without limiting the generality of this work, we will apply the proposed approach in this paper to the *contracting section* (layer *Conv 1* to layer *Conv 6_1*) of FlowNet S [7], as it is a good example of a deep CNN whose weights and maps do not fit the relatively small On-Chip memories of embedded targets.

The MAC operations involved in FlowNet S are in the order of several billions per input image pair, moreover, the deep layers lead to a great number of weights that do not fit the small On-Chip memory of a typical embedded target. In detail, the total size of weights and biases for the contractive section of FlowNet is approximately 96 MBytes, while the memory space needed to allocate the maps is roughly 32 MBytes. Moreover this amount of data is transferred to and from the external memory multiple times during the computations and this strongly affects performance. Fetching data from the external memory is more expensive in terms of power and time, but becomes a necessity when the OCM is not big enough to host all the necessary data. For this reason a Double Buffering system, together with an efficient tiling strategy, become an important step in reducing access times and enhance data reuse to limit the bandwidth.

3 System Structure

3.1 Embedded Target

The system of choice for testing is Zedboard, a development board based on the Xilinx Zynq®-7020 All Programmable SoC [2]. This platform also features two external 512 MB DDR3 memories. The Zynq®-7020 SoC is composed by two main sections: the Processing System (PS) and the Programmable Logic (PL). The PS consists of dual-core ARM® Cortex®A9, 32KB of L1 cache per core, 512 KB of shared L2 cache, 256 KB of On-Chip memory (OCM), a DMA controller, a DDR controller and other peripherals. The two levels of cache sit in between the processing core and the OCM and can be enabled or disabled programmatically. The choice of a SoC from the Zynq family is supported by other works on CNNs, like [14], where this SoC has been proven the right choice especially for the possibility of building hardware accelerators in the PL side to speed up computations.

3.2 Double Buffering

To reduce the impact of data transfers on performance, data can be moved to a physical memory *closer* to the processor, where *closer* implies a shorter access time. Typically, these memories are also much smaller and cannot contain all data to run deep CNN architectures like FlowNet.

Double Buffering is a technique to overlap data transfers between external and On-Chip memories with CPU computations, at the cost of an extra hardware unit, a DMA. The key point of this method is that the DMA constantly fills the OCM with the data needed by the CPU, so that the CPU communicates with the short access time OCM only. As such, Double Buffering masks the long access time from the large external memory. To ensure data consistency, the OCM memory is divided into two buffers, one to receive data moved by the DMA and the other one accessed by the CPU. At the end of each computational unit, the pointers to the two buffers are swapped and the same process repeats.

3.3 Tiling

The convolution operation, which underpins the CNN, requires 3 different data to operate: input, weights and output to accumulate the result. The question that arises at this point is how to partition the available OCM space among these three when transferring data with the double buffering system. Tiling is the strategy by which all the necessary layer data is partitioned in smaller chunks that are transferred by the double buffering system and fit the OCM.

Let us consider a generic convolutional layer of the network. All the three data types involved are essentially three-dimensional maps. The boundaries of these maps are represented in Fig. 2. For the input and output maps the names used throughout this work are ROW, COL and N respectively for the s, t and u directions. The tiling process consists of cutting smaller three-dimensional sections out of these three volumes. The volume of the tiles depends on the available OCM space, on the dependency between the entities and on the tiling strategy.

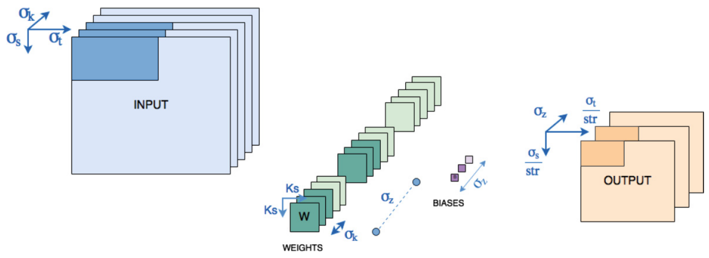


Fig. 1. CNN layer tiling parameters

In Fig. 1, all the tiling parameters are represented with the names and symbols that will be adopted later for the mathematical model. In particular for each new computational unit, the DMA transfers a $\sigma_s \times \sigma_t \times \sigma_k$ input tile, a $\frac{\sigma_s}{str} \times \frac{\sigma_t}{str} \times \sigma_z$ output tile and $\sigma_k \times \sigma_z$ kernels of weights.

The constants $\mathrm{K_s}$ (kernel size) and str (stride) are specific for the layer, while the four σ parameters are the degrees of freedom of the problem, bounded by the map dimensions (Fig. 2) and constrained by the OCM size as it will be explained in Sect. 4.

Because of the nature of the convolution operation, choosing σ_s and σ_t for the input tile, fixes the s and t parameters also for the output. Furthermore the choice of σ_k and σ_z determines the total number of $K_s \times K_s$ kernels to be fetched and the third dimension of respectively input and output tiles. Because of the *granularity* of the convolution algorithm, the minimum amount of data that can be tiled for each layer, for a computational unit to work, is a $K_s \times K_s \times 1 \times 1$ input tile, $K_s \times K_s + 1$ for one kernel and one bias, plus a single element to hold and accumulate the output.

It is important to also stress that each layer of the network works with kernels, input and output maps of very different shapes and this is why there is no such a thing as an overall optimal tiling parameter sets. Each layer of the CNN has its own characteristics and needs to be optimized accordingly.

4 Optimal Tiling

4.1 Cost Model

In this work the memory bottleneck is the addressed problem, hence the aim of the OCM partitioning will be to minimize the external memory bandwidth. To represent the impact of a transaction on the external bandwidth, a linear model is used that takes into account how data are transferred by the DMA with respect to their layout in memory. In other words, the cost of accessing and transferring data sequentially is different from the cost of accessing the same amount of data but fragmented in many memory blocks.

The linear model used is similar to the one presented in [10]. While the model in [10] is linear in two parameters, the one used in this work introduces an extra parameter, p, to better represent the cost of a DMA transaction.

$$T = C + s \cdot p + n \cdot t \tag{1}$$

In the cost function T, the parameter C represents the cost of initiating a new DMA transfer and is a startup cost independent from the amount of data to be moved or their memory layout. The extra parameter p, is introduced to make a distinction between the startup cost C and the additional cost of accessing non sequential elements in memory (scattering) within the same DMA call. This extra parameter was also used by [12], in modeling the transfer of 2D data tiles, with a similar cost function. The distinction between p and C is important because there is no need to issue a new DMA instruction when non sequential elements are accessed, thanks to the *scatter-gather* capabilities of the PL-330 DMA controller. The parameter p is multiplied by the number of address jumps,

s, occurring for the transfer. Lastly, t is the cost for transferring a single element, that has to be multiplied by the total number of transferred data elements n.

The layout of data in memory is *row-major*. This means that to cut out a $\sigma_s \times \sigma_t \times \sigma_k$ tile in a certain position, σ_t elements can be transferred sequentially. Then one needs to update the source address to the next row, before proceeding with another σ_t sequential transfer. The same applies for moving from one input feature to the next one. As a consequence, the cost for accessing a $\sigma_s \times \sigma_t \times \sigma_k$ input tile can be modeled as in Eq. 3. To derive the cost functions for weights (T_{wb}) and outputs (T_{out}), a reasoning similar to the one used for the input tiles is applied. Because of the $K_s \times K_s$ granularity of the convolution algorithm being used, whole kernels are fetched for every tiling iteration.

$$T_{in} = C + \sigma_s \cdot \sigma_k \cdot p + \sigma_s \cdot \sigma_t \cdot \sigma_k \cdot t \tag{2}$$

$$T_{out} = 2 \cdot (C + \frac{\sigma_s \cdot \sigma_z}{str} \cdot p + \frac{\sigma_s \cdot \sigma_t \cdot \sigma_z}{str^2} \cdot t) \tag{3}$$

$$T_{wb} = C + (\sigma_z + 1) \cdot p + (\sigma_k \cdot \sigma_z \cdot Ks^2 + \sigma_z) \cdot t \tag{4}$$

Each of the cost functions will have to be multiplied by the number of times the specific transfer is issued (DMA calls). The number of DMA calls depends on how the tiling nested loop is sorted. The pseudo code for the tiling loop is reported in Listing 1.1.

```
for(st = 0; st < ROWin; st+= Ss)
 for(tt = 0; tt < COLin; tt+= St)
   for(kt = 0; kt < Nin; kt+= Sk){
       DMA_DDR_to_OCM(INPUT);
     for(zt = 0; zt < Nout; zt+= Sz){
       DMA_DDR_to_OCM(WEIGHTS);
       DMA_DDR_to_OCM(OUT);
     CONV_TILE(INPUT,WEIGHTS,OUTPUT);
       DMA_OCM_to_DDR(OUT);
     }
  }
```

Listing 1.1. Tiling Loop pseudo-code

The meaning of the loop boundaries are shown in Fig. 2. The loop order affects the bandwidth and there exist different techniques to determine which loop order is the best in terms of bandwidth. This aspect will not be analyzed in this work where the focus is instead how to optimize the tiling for a given nesting order. The total number of DMA calls per layer (D_{in}, D_{wb}, D_{out} for input, weights and output) according to the given loop order, are expressed as follows:

$$D_{in} = \frac{ROWin \cdot COLin \cdot Nin}{\sigma_s \cdot \sigma_t \cdot \sigma_k} \tag{5}$$

$$D_{wb} = D_{out} = \frac{ROWin \cdot COLin \cdot Nin \cdot Nout}{\sigma_s \cdot \sigma_t \cdot \sigma_k \cdot \sigma_z} \tag{6}$$

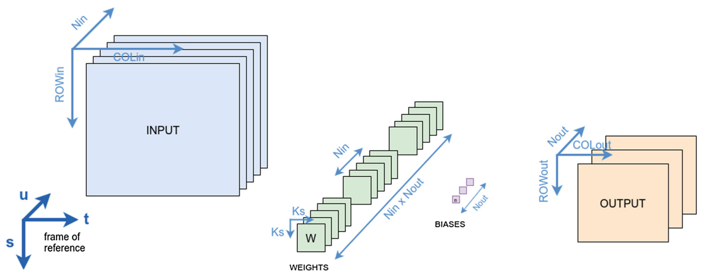


Fig. 2. CNN dimensions (loop boundaries)

The overall cost function is then derived by summation of the T_* by D_* products and can be expressed as follows:

$$T_{tot} = \frac{ROWin \cdot COLin \cdot Nin}{\sigma_s \cdot \sigma_t \cdot \sigma_k} \cdot (T_{in} + \frac{Nout}{\sigma_z} \cdot (T_{out} + T_{wb}))$$

4.2 Constraints

Once the cost functions have been defined, the focus can be moved on the constraints that affect the σ parameters. The first restriction comes from the size of the OCM of the target. In particular input and output tiles, together with weight kernels and biases have to occupy less then half the OCM size, for the double buffering to work correctly. This limit can be expressed as:

$$\sigma_s \cdot \sigma_t \cdot \sigma_k + \frac{\sigma_s \cdot \sigma_t \cdot \sigma_z}{str^2} + \sigma_k \cdot \sigma_z \cdot K_s^2 + \sigma_z \quad \leq \quad \frac{OCM_{size}}{2}$$

On top of this, other constraints are related to the underlying tiling loop and the way data is fetched. Firstly the tiling size along a specific direction has to be a divisor of the map size in that direction. Moreover, all the σ parameters have to be positive integers smaller than the map dimension for that direction.

4.3 Optimal Parameters

Applying this set of constraints greatly reduces the search space of the problem, allowing *exhaustive search* techniques to be used without having to deal with very long simulation times.

To derive the optimal tiling settings for the presented double buffering strategy, the tiling parameters are swept for every layer of the test network individually. The results are presented in Sect. 6.

5 Implementation

5.1 CNN Framework

The FlowNet network is implemented within the Caffe [9] software environment. The Caffe framework is meant to run on PC workstations and has many dependencies on external high-level libraries and tools. For this reason and performance issues, the choice has been not to install Caffe on the embedded target.

From the Caffe implementation however, a binary file can be extracted, containing all the pre-trained weights and biases of the network.

To run CNNs on the Zynq®-7020 target, a custom stand-alone C framework has been implemented to exactly emulate all the data processing that takes place in Caffe.

5.2 PL-330 Programming

At boot time weights and maps are fetched from an SD card and loaded onto one of the two external DDR3 modules where they can be accessed by either the CPU or other memory controllers on the Zynq®-7020.

The Zynq®-7020 SoC features an ARM® PL-330 DMA [1] controller (DMAC). The DMAC is programmable by means of a small and variable length instruction set that provides a flexible method of specifying the DMA operations. The usage and restrictions of these instructions can be found in [1].

The key to enable full transfer flexibility in the PL-330 is to assemble ad-hoc DMAC instruction programs in the system memory space, that the DMAC can access for execution. In particular different DMAC programs can be assigned to different channels simultaneously and executed in a quasi-parallel round-robin fashion.

In the case of CNNs four different programs (for input, output read, output write back and weights) are built from four different code templates whose parameters are updated at runtime. In the most general scenario, full control over the three tiling dimensions has to be provided. This means that *Scatter-Gather* capabilities have to be exploited to cut out $\sigma_s \times \sigma_t \times \sigma_k$ tiles from the input.

The main issue regarding a three-dimensional tiling is that the DMAC instruction set only supports two-level-deep nested loops. The strategy used in this work is to loop unroll in the leftover dimension thus repeating the same code template multiple times.

```
DMA_LOAD src_addr ;init src, dest, transfer config
DMA_LOAD dest_addr
DMA_LOAD dma_config
 LOOP_0 cnt0:
        LOOP_1 cnt1:  ; transfer consecutive data
                DMA_LOAD data from DDR
                DMA_STORE data to OCM
        END_LOOP1
        LOOP_1 cnt2:  ; update src address
                ADD Imm16_bit to src_addr
        END_LOOP1
                ADD Imm16_bit to src_addr
 END_LOOP0
 ...
   ;LOOP0 cnt0 repeated sigma_k times (unrolling)
 ...
DMA_SEND_EVENT ev_id  ;end of transfer interrupt
DMA_END
```

Listing 1.2. Input transfer DMAC pseudo-code

The nested loop (*cnt0* and *cnt1*) in Listing 1.2 takes care of the first two dimensions, rows and columns, cutting out $\sigma_s \times \sigma_t$ tiles, while the unrolling is adopted to loop through the third one for σ_k times. In between the nested loop, a single loop (*cnt2*) is instantiated to update the *source address* to the new memory location by means of a series of 16-bit immediates additions. Finally, a *DMA_SEND_EVENT* instruction is issued to signal the end of transfer to the processor and the program ends. To prevent the convolution operations to run before the transfer of all the necessary data has completed, software barriers in the C code have been used in between the loop levels. (These barriers consist of software watchdog down-counter that are reset at the end of each transfer completion, preventing them from elapsing. The execution is suspended on the barriers when the data to be computed has not been completely transferred).

6 Results

6.1 Performance on Hardware Target

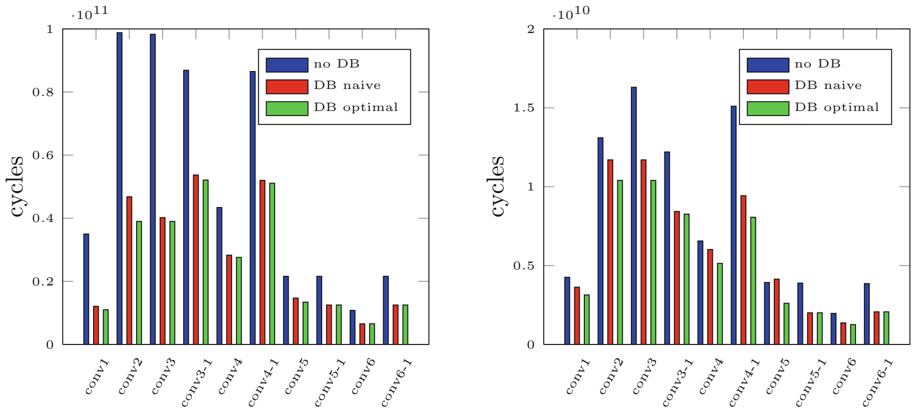
To estimate the effectiveness of the optimal tiling parameter set, another *naive* set of parameters has been used for comparison. These parameters have been selected with the only aim of using a high OCM space percentage (at least above 80%) for the tiling of each layer. Handcrafting the parameters to occupy most of the OCM space is a reasonable choice. This is because as found in different works, like [10], bigger OCM sizes lead to better performances when it comes to double buffering.

As the tiling strategy aims to alleviate the memory bottleneck to increase performance, the first considered evaluation metric is the number of execution cycles for the ARM®Cortex®A9, running at 667 MHz. Figure 3 shows the execution cycles for the contractive layers of FlowNet, both with the L1 and L2 levels of cache disabled (left) and enabled (right). For both cases, three different scenarios are considered: the non-optimized implementation (blue), where no double buffering system is present and all the data are fetched by the CPU from the external DDR; an optimized implementation with double buffering at a naive (non-optimal) set of parameters (red) and the optimized implementation with the double buffering system running at the optimal parameter set (green).

The advantage of using a double buffering system is clear from the results. The total non-cached execution time goes from 1573 s to 838 s for the naive Double Buffering (DB) Tiling and 805s for the optimal DB, hence with an overall improvement of almost 2x, with a peak of 2.4x for the second layer. Regarding the naive versus optimal tiling, the difference is present but not so marked, especially in the deeper layers.

Moving on to the cache-enabled setting, the double buffering system is still moving data between the DDR and the OCM but now two levels of cache sit in between the OCM and the CPU and the caches are flushed after every computational unit.

In this setting, the impact of the external bandwidth on performance is partially masked because of the presence of the cache. The cache size (512 KB), bigger than the OCM, greatly reduces the external memory bandwidth and masks the benefits of the double buffering solution. Nevertheless it is important to underline that still a tangible improvement is present exploiting the double buffering and still the optimal set performs better than the naive one.



(a) Cache disabled

(b) Cache enabled

Fig. 3. Execution cycles for the FlowNet contractive layers, for scenarios with cache disabled and enabled. (Color figure online)

6.2 Data Transfer Profiling

As a second evaluation metric, the external memory bandwidth can be evaluated. For this test, the convolution operations of the layers have been disabled, removing the internal bandwidth and computations cycles and allowing to profile the external data movements (DDR $\rightleftharpoons$ OCM) under the double buffering for naive and optimal settings.

The results in Fig. 4 clearly demonstrate that the optimal tiling strategy is more effective than a naive approach. Note that both approaches use roughly the same (high) percentage of the available OCM space, as demonstrated in Table 1. Therefore, it is generally not true that the highest SPM usage leads to the lowest external memory bandwidth. This is particularly clear for layer 7, where the optimal result is achieved with a smaller SPM space than the one used by the naive tiling.

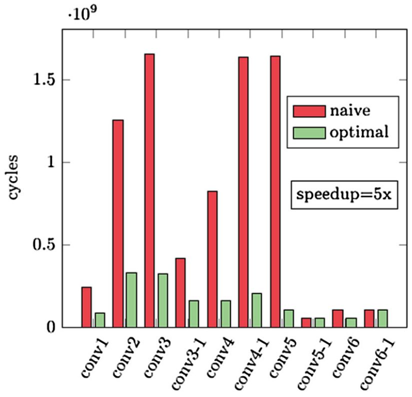


Fig. 4. Naive versus optimal tiling transfer cycles

Table 1. OCM space usage % per layer

	conv_1	conv_2	conv_3	conv_3.1	conv_4	conv_4.1	conv_5	conv_5.1	conv_6	conv_6.1
Naive	86	92	96	83	91	89	88	88	80	91
Opt.	88	98	98	86	85	88	76	86	90	91

7 Conclusion

An optimal tiling strategy for the double buffering system has been proven to be effective in removing the memory bottleneck for a deep CNN architecture. When applied to the FlowNet CNN, this improves the overall execution cycles by 50% (with cache disabled/34% with cache enabled). Moreover a 5x improvement is brought by the use of optimal tiling parameters has been assessed, with respect to the use of naive ones when it comes to data transfers between the external DDR memory and the On-Chip memory. Furthermore, another key point that flows from the results is that high percentages of On-Chip memory space usage do not necessarily lead to a lower transfer time. This statement does not imply that there is no reason to prefer bigger On-Chip memories to small ones. Despite the findings, the general trend is that performances increase with bigger OCMs, as found in different works such as [13]. Because of this, finding an optimal set of parameters to transfer data tiles in a deep CNN architecture has been proven to be an important step of the optimization process.

Acknowledgment. The work of S. Smets was supported by a Doctoral Fellowship of the Research Foundation Flanders (FWO).

References

1. PrimeCell DMA Controller (PL330) (2007)
2. Zedboard, Zynq Evaluation and Development Hardware Users Guide (2014)
3. Al Maashri, A., Cotter, M., Chandramoorthy, N., DeBole, M., Yu, C.-L., Narayanan, V., Chakrabarti, C.: Hardware acceleration for neuromorphic vision algorithms. J. Sig. Process. Syst. **70**(2), 163–175 (2013)
4. S. C. class. Cs231n: convolutional neural networks for visual recognition (2016)
5. Conti, F., Pullini, A., Benini, L.: Brain-inspired classroom occupancy monitoring on a low-power mobile platform. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, pp. 610–615 (2014)
6. Farabet, C., Martini, B., Corda, B., Akselrod, P., Culurciello, E., LeCun, Y.: Neuflow: a runtime reconfigurable dataflow processor for vision. In: 2011 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), pp. 109–116. IEEE (2011)
7. Fischer, P., Dosovitskiy, A., Ilg, E., Häusser, P., Hazırbaş, C., Golkov, V., van der Smagt, P., Cremers, D., Brox, T.: Flownet: learning optical flow with convolutional networks. arXiv preprint arXiv:1504.06852 (2015)
8. Huang, Q., Xue, J., Vera, X.: Code tiling for improving the cache performance of PDE solvers. In: 2003 International Conference on Parallel Processing, Proceedings, pp. 615–624. IEEE (2003)
9. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. arXiv preprint arXiv:1408.5093 (2014)
10. Kandemir, M., Ramanujam, J., Irwin, M.J., Vijaykrishnan, N., Kadayif, I., Parikh, A.: Dynamic management of scratch-pad memory space. In: Design Automation Conference, Proceedings, pp. 690–695. IEEE (2001)
11. Kodukula, I., Ahmed, N., Pingali, K.: Data-centric multi-level blocking. In: ACM SIGPLAN Notices, vol. 32, pp. 346–357. ACM (1997)
12. Saidi, S., Tendulkar, P., Lepley, T., Maler, O.: Optimizing two-dimensional DMA transfers for scratchpad based MPSoCs platforms. Microprocess. Microsyst. **37**(8), 848–857 (2013)
13. Yang, X., Wang, L., Xue, J., Tang, T., Ren, X., Ye, S.: Improving scratchpad allocation with demand-driven data tiling. In: Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, pp. 127–136. ACM (2010)
14. Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., Cong, J.: Optimizing FPGA-based accelerator design for deep convolutional neural networks. In: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 161–170. ACM (2015)