Efficient Winograd-based Convolution Kernel Implementation on Edge Devices

Athanasios Xygkis[*]
Intel Corporation, Ireland
thanasis.xigis@intel.com

Lazaros Papadopoulos
School of ECE, NTUA, Greece
lpapadop@microlab.ntua.gr

David Moloney
Intel Corporation, Ireland
david.moloney@intel.com

Dimitrios Soudris
School of ECE, NTUA, Greece
dsoudris@microlab.ntua.gr

Sofiane Yous
Intel Corporation, Ireland
sofiane.yous@intel.com

ABSTRACT

The implementation of Convolutional Neural Networks on edge Internet of Things (IoT) devices is a significant programming challenge, due to the limited computational resources and the real-time requirements of modern applications. This work focuses on the efficient implementation of the Winograd convolution, based on a set of application-independent and Winograd-specific software techniques for improving the utilization of the edge devices computational resources. The proposed techniques were evaluated in Intel/Movidius Myriad2 platform, using 4 CNNs of various computational requirements. The results show significant performance improvements, up to 54%, over other convolution algorithms.

1 INTRODUCTION

In the world of Internet of Things, sensors and connected devices generate huge amount of data, on the order of petabytes per second [3]. There is increasing need to perform significant amount of computation closer to the edge rather than transferring large portions of raw data to the cloud, due to communication cost that impacts performance and energy consumption [15]. For applications deployed in drones, autonomous vehicles, robotics and wearables, local data processing by embedded devices is desired, since latency and security risk of relying at the cloud are intolerable. These applications are often enabled by machine learning algorithms, and more specifically by Convolutional Neural Networks (CNNs), which are used to extract meaningful information from raw data. Therefore, the efficient deployment of neural networks in embedded devices will improve near-sensor processing, avoid the expensive data transmission, enable freedom from the cloud and provide low latency along with low energy consumption. This is a significant challenge, due to the embedded systems resource constraints and the increased computational requirements of neural networks.

[*] Also with the School of ECE, NTUA, Greece.

Table 1: 3x3 conv. exec. time in Intel/Movidius Myriad2

Input size	Output maps	Direct convolution (ms)	Straightforward Winograd (ms)
$[56 \times 56 \times 64]$	192	11.3	11.8
$[28 \times 28 \times 96]$	128	4.6	4.7
$[28 \times 28 \times 128]$	192	7.9	7.5

The growth of the embedded applications enabled by CNNs contributed to the availability of various specialized architectures, such as FPGA-based (e.g. NeuFlow [13]), ASICs (e.g. [2]) and heterogeneous SoCs with deep learning processing capabilities. CEVA XM [14], Cadence Tensilica vision DSP [6] and Intel/Movidius Myriad [11] belong to the family of programmable embedded processor-based platforms that rely on a set of vector processing units and on high memory bandwidth to provide computational power within a few Watts of power envelope. Nevertheless, significant effort is required to bring the computational load of state-of-the-art CNNs within the power envelope of such low power edge devices.

A recent trend towards the deployment of neural networks in edge devices is the design of CNNs with limited requirements in computational resources, such as the SqueezeNet [8][5]. Another approach, which is complementary to this and in which this work focuses, is to develop techniques and methodologies for the efficient implementation of CNNs in such systems. Although there exists many algorithmic approaches for improving the performance of CNNs implemented on computing architectures, such as the Winograd [9] and the Strassen [4], little attention has been paid to the efficient implementation of these algorithms on edge devices. Indeed, several recent publications conclude that the metrics of execution time and energy efficiency are largely ignored by mainstream computer vision researchers [8][7]. As an example, Table 1 presents the execution time of 3x3 convolution for various input sizes of GoogleNet deployed in the Intel/Movidius Myriad2 platform [11]. The table shows results for direct (i.e. conventional) convolution and a straightforward implementation of the convolution based on the Winograd algorithm [9]. Although Winograd requires 2.25 times less element-wise multiplications than the direct one, the straightforward implementation, in which each processing unit operates in isolation from the others, provides similar performance. Architectural constraints, such as the limited local memory size of Myriad reduce the performance of the Winograd algorithm. Since Winograd convolution requires significantly more memory space, (the kernel size increases by 78% due to data re-layout and the input

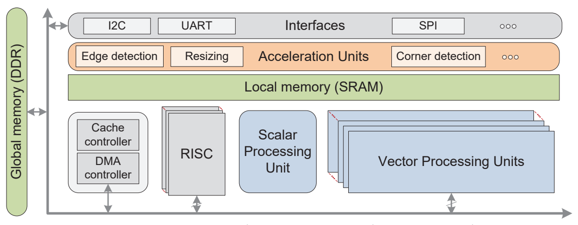


Figure 1: Target architecture schematic diagram.

size increases by 100% due to striding), frequent DMA transfers are required that negatively affect the execution time. Therefore, the results of Table 1 highlight the need for techniques that enable the efficient implementation of the Winograd convolution and similar algorithms in edge devices, by exploiting fine-grain parallelism and improving memory management.

This work is a contribution in "filling the gap" between the proposed algorithmic approaches for reducing the computational requirements of CNNs and the actual implementation of these algorithms in modern heterogeneous edge devices. The complexity and the constraints of these architectures (mainly imposed by the requirements for energy efficiency at the edges of the Internet of Things networks), along with the application requirements for increased real-time performance (e.g. real-time object detection and recognition) put very high requirements on the software side. Therefore, software techniques are required to address challenges, such as the synchronization between the vector processing units, the efficiency of data transfers, data reusability issues and managing of the limited hardware resources.

From the proposed algorithms that focus on increasing the performance of CNNs, we selected the Winograd algorithm that provides efficient convolution of relatively small kernels [9]. The proposed software techniques for the efficient implementation of the Winograd algorithm on edge devices are a combination of typical application-independent data management optimizations widely used in embedded systems and of Winograd algorithm-specific implementation techniques. The application-independent techniques are inspired by existing methodologies for memory assignment and data transfer optimization that have been extensively examined in the literature [1][10] and they are used in this context to increase the impact of the Winograd-specific techniques in the overall CNN inference execution time reduction. The evaluation of the proposed techniques is performed in the Myriad2 embedded platform, in which 4 widely used and computationally intensive CNNs were implemented using performance and energy consumption as key metrics.

The rest of the paper is organized as follows: Section 2 briefly describes the target architectures and their constraints. Section 3 presents the proposed software techniques and the evaluation and discussion follows in Section 4. Finally, in Section 5 we draw conclusions.

2 EDGE DEVICES AND CNN IMPLEMENTATION CHALLENGES

This work focuses on the family of processor-based heterogeneous embedded platforms, which are extremely low power (often <1W) and are often used at the edges of IoT networks to perform tasks enabled by deep learning algorithms, such as object detection and recognition. It includes CEVA XM [14], Cadence Tensilica vision DSP [6] and Intel/Movidius Myriad [11]. A typical high-level schematic diagram is depicted in Fig. 1. The most important common architectural features are the following:

- **Multiple memory hierarchies**: Normally, a local (scratchpad) memory provides low latency and high throughput data access, while a larger global memory is often accessed through DMA transactions. Data are normally operated in the local memory, after being fetched from lower levels of the memory hierarchy.
- **Multiple Vector Processing Units** (VPUs), which usually support VLIW, SIMD and multiply-accumulate operations (MACs) with high efficiency. For instance, CEVA mx-6 and Tensilica Vision P6 integrate 128 and 256 MACs, respectively.
- **RISC processor(s)** that may run an operating system (Myriad2 LEON-OS processor, ARM Cortex i.MX 6 in YouSiP vision DSP based on CEVA platform) or handle tasks such as interrupts, IO, etc..

CNN data storage and management is challenging in edge devices. The Winograd algorithm requires significantly more memory size than the direct convolution. Therefore, from the hardware perspective, the limited size of local memories imposes a significant data management challenge. As a result, the overhead of frequent DMA transaction reduces the performance and increases the energy consumption. Furthermore, in ported memories, such as in Myriad2, stalls may appear under heavy data sharing. Finally, extensive experimentation in Myriad2 has shown that the DMA engine performance is reduced under heavy utilization [12]. Software techniques that optimize data management, improve the utilization of local memories and exploit parallelism can improve both performance and energy efficiency.

3 SOFTWARE TECHNIQUES FOR WINOGRAD CONVOLUTION IMPLEMENTATION

In this Section, we present a set of software techniques for the efficient implementation of the Winograd convolution on edge devices that belong to the family of platforms described in Section 2. The proposed techniques focus on data organization and management and improve the memory utilization and the exploitation of parallelism provided by the multiple vector processing units of the target architectures. More specifically, from the embedded systems domain, we selected a set of widely used **data transfer and management optimizations**, which increase the impact of the **Winograd algorithm-specific implementation techniques** that significantly improve the performance of convolution based on the Winograd algorithm.

3.1 Data transfer and management optimizations

The data transfer and management optimizations are application-independent. They improve the utilization of the local memory of edge devices and reduce the frequency of data transfers. They are depicted in Fig. 2 and are described below.

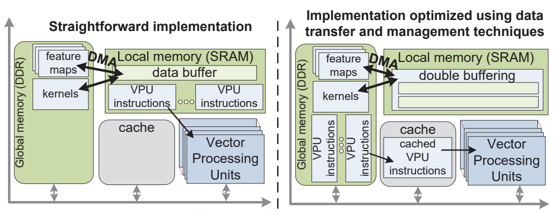
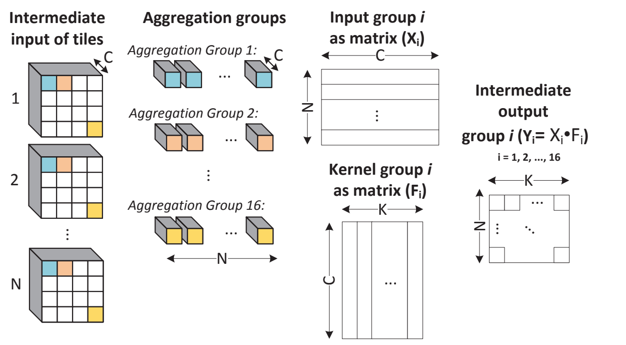


Figure 2: Data transfer and management optimizations.


Intermediate input of tiles
Aggregation groups
Aggregation Group 1:
Aggregation Group 2:
Aggregation Group 16:
Input group i as matrix (X_i)
Kernel group i as matrix (F_i)
Intermediate output group i (Y_i= X_i•F_i)
i = 1, 2, ..., 16
C
N
K
Figure 3: Aggregation in Winograd. N: number of input tiles, C: number of input feature maps, K: number of output feature maps.

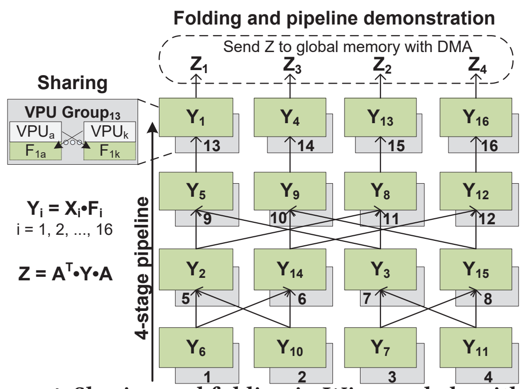


Figure 4: Sharing and folding in Winograd algorithm.

3.1.1 *Increasing local memory available space.* Allocation of vector processors' instruction code in the local memory is often the default memory allocation scheme in embedded systems, such as in Myriad2. However, this scheme imposes frequent DMA data transfers between the local and the global memory during the execution of a CNN inference, due to the limited local memory space available for data. Reducing the frequency of DMA transfers can benefit performance significantly: The effects of DMA data transfers to the performance and the energy consumption in Myriad have been extensively examined in [12]. Allocating the vector processing units' instruction code in the global memory and fetching instructions through the cache subsystem increases the local memory space available for CNN data. Performance loss due to the allocation of instructions in lower levels of memory hierarchy is compensated by performance and energy consumption improvements due to more efficient local memory utilization.

3.1.2 *Double buffering and Overlapping.* Increasing local memory available space for data enables double buffering and overlapping techniques, which are widely used in embedded systems. More specifically, the increased local memory space can be used to allocate a second data buffer, which is used to overlap communication and computation.

3.2 Winograd algorithm-specific implementation techniques

The second group of software techniques focuses on the optimization of the implementation of convolutional layers of CNNs on edge devices. Winograd algorithm for convolution is an efficient way to compute the convolution of small kernels on small input sizes [9]. A typical configuration in which the Winograd algorithm provides numerical stability and efficiency is 3x3 kernels and 4x4 input sizes. The algorithm can efficiently provide the 2x2 output result requiring 2.25 times less element-wise multiplications than the direct convolution. Although we focus on the aforementioned configuration, the following techniques are applicable to other configurations, as well.

The steps of the Winograd algorithm are summarized below:

(1) The input is split into 4x4 tiles with stride 2. The following steps are applied in each tile:
(2) Each 4x4 tile D is transformed into an 4x4 *intermediate input* X, as follows: $X = B^T DB$, where B is a 4x4 matrix with elements -1, 0 and 1 [9].
(3) The 3x3 kernel is similarly transformed offline into a 4x4 *intermediate kernel* F, as well.
(4) The *intermediate output* Y is calculated as follows: $Y = X \odot F$.
(5) Finally, the intermediate output is transformed into a 2x2 matrix Z, which is the result of convolution: $Z = A^T YA$, where A is a 4x2 matrix with elements -1, 0 and 1 [9].

The brief description of the Winograd above considers 2D inputs and outputs. However, in the context of CNNs, the 3rd dimension that represents the multiple feature maps should also be considered. In this case, the dimension of each intermediate input of each tile is Cx4x4 (Depth × Height × Width), where C denotes the number of input feature maps (depth). The straightforward implementation for producing the intermediate output ($Y = X \odot F$) would be to generate the intermediate output of each map separately, followed by stacking of the output results. However, a more efficient approach that eliminates interdependencies and exploits the SIMD feature of the vector processing units is to perform the transformation in a vectorized manner, as shown in Fig. 3. The elements of the intermediate input of each tile are aggregated into 16 different groups, with each group containing elements from all tiles. More specifically, the 1st group contains vectors denoted by x_{00} in the 2D Winograd. Similarly, the 2nd group contains the vectors x_{01} of all the tiles, ..., the 16th group contains the vectors x_{33} of all the tiles. The same is done offline for the kernels. Thus, the intermediate output is generated by performing 16 matrix-matrix multiplications, namely $Y_i = X_i F_i$, for $i = 1, 2, \ldots, 16$. The 16 multiplications will be performed in parallel and combined together to get the final output of the convolution. This data representation enables the *sharing* and the *folding* software techniques, described in the following two subsections.

3.2.1 Sharing. *Sharing* refers to the way that data, which do not fit in the local memory of processing units, are efficiently shared among processing units to retain high data locality and it is depicted in Fig. 4.

In Myriad, although the whole local memory is accessible to all processing units, it is divided in a number of 'slices' and each slice is attached to a specific unit. Each unit can access the attached local memory slice with lower latency compared to the others. Therefore, in the context of Winograd, the solution that provides the highest data locality would be to store each one of the 16 matrices X_i, F_i, Y_i in the part of the local memory where the processing unit that performs the multiplication has the lowest access time. However, since the matrices usually do not fit in the slices, splitting the multiplication among processing units can still retain data locality. For example, the splitting among two units is performed as follows:

$$\left[\begin{array}{c} Y_{i,a} \\ \hdashline Y_{i,b} \end{array}\right] = \left[\begin{array}{c} X_{i,a} \\ \hdashline X_{i,b} \end{array}\right] \left[\begin{array}{c:c} F_{i,a} & F_{i,b} \end{array}\right] \tag{1}$$

Matrices with "a" subscript are stored in the slice of one processing unit, while matrices with "b" subscript are stored in the slice of another unit. As a result, the parts $F_{i,a}$ and $F_{i,b}$ of F_i are shared among the two units, that compute $Y_{i,a}$ and $Y_{i,b}$ respectively. Thus, the processing units of an edge device can be divided into groups, with each group containing the 1/16th of the total units and sharing F_i data to calculate equal parts of a Y_i. Splitting the kernels among the slices of memory leads to better load balancing and more fair memory usage, improving the parallelization scalability.

3.2.2 Folding. The purpose of *folding* is to improve the reusability of data already stored in the highest levels of the memory hierarchy for producing the final output of the convolution. This technique reduces dramatically the overhead imposed by DMA transactions to/from the global memory.

The equation $Z = A^T Y A$ that produces the 2x2 convolution output (step (5) in the description of the Winograd algorithm) can be written as follows (details can be found in [9]):

$$\begin{bmatrix} Z_1 & Z_2 \\ Z_3 & Z_4 \end{bmatrix} = A^T \begin{bmatrix} Y_1 & Y_2 & Y_3 & Y_4 \\ Y_5 & Y_6 & Y_7 & Y_8 \\ Y_9 & Y_{10} & Y_{11} & Y_{12} \\ Y_{13} & Y_{14} & Y_{15} & Y_{16} \end{bmatrix} A \tag{2}$$

where

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

The above transformation can be written as:

$$\begin{aligned} Z_1 &= Y_1 + (Y_5 + Y_9) + (Y_2 + Y_6 + Y_{10}) + (Y_3 + Y_7 + Y_{11}) \\ Z_2 &= -Y_4 + (Y_2 + Y_6 + Y_{10}) - (Y_3 + Y_7 + Y_{11}) - (Y_8 + Y_{12}) \\ Z_3 &= -Y_{13} + (Y_5 - Y_9) + (Y_6 - Y_{10} - Y_{14}) + (Y_7 - Y_{11} - Y_{15}) \\ Z_4 &= Y_{16} + (Y_6 - Y_{10} - Y_{14}) - (Y_7 - Y_{11} - Y_{15}) - (Y_8 - Y_{12}) \end{aligned}$$

The terms grouped in parenthesis appear in more than one of the equations. Therefore, these terms can be computed by a single group of processing units and the result can be forwarded to others. In other words, the final result is computed by accumulating (or *folding*) a series of recurrent sub-results. The latest have also been generated by *folding* as well. Generally, a common subexpression elimination approach is an essential part of the *folding* technique.

Thus, while *sharing* is applied inside each group of processing units and improves the locality of data fetched from the lower levels of memory hierarchy, *folding* improves the reusability of results that are computed by a single group of processing units and be reused by others. As depicted in Fig. 4, results that are locally calculated by each group are forwarded to the next level of processing units groups. These groups accumulate the received results and repeat the same process. After the *folding* operation of the last level is completed, the final output is returned to the global memory with DMA transfer. Thus, *sharing* and *folding* combined together lead to a pipeline, in which the information is continuously propagated through the levels of groups of processing units.

In the case where the number of processing units is not a multiple of 16, both the *sharing* and the *folding* techniques should be adjusted accordingly: Kernel data F_i shared within a specific group of processing units may be used by units that belong to other groups, as well. Additionally, the pipeline steps may be less than four. The suggested approach is that the groups at the top levels should accumulate more produced data than the groups at lower levels, because the reusability of the accumulated results at higher levels is reduced. This approach is expected to reduce the performance decline caused by the limited available processing units.

The Winograd algorithm imposes large and frequent data transactions, due to the increased size of the intermediate input and kernels. As a result, following an approach where each processing unit operates in isolation from the other units leads to exhaustion of the bandwidth between the global and local memory, as it can be derived from the results of Table 1. *Sharing* and *folding* techniques mitigate this issue, by improving the utilization of the local memory. On the other hand, the proposed approach includes synchronization overhead, which appears during the folding step, (i) when processing units of a specific level wait for receiving data from the lower level and (ii) in common shared resources. However, evaluation shows that the synchronization overhead is compensated by the overall performance gain of the CNN inference execution.

4 EVALUATION IN MYRIAD EMBEDDED PLATFORM

The software techniques described in the previous section were evaluated in a set of CNNs acting as critical components of real-life use cases, implemented on the Myriad2 embedded platform.

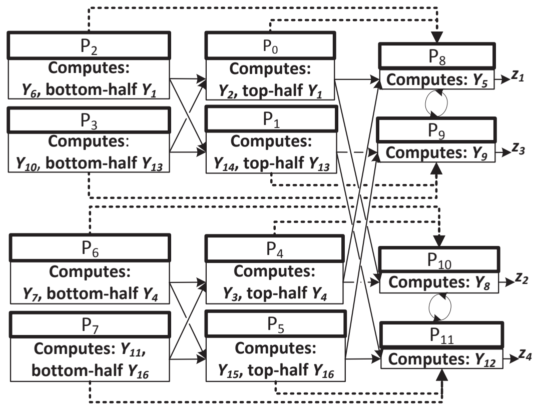
4.1 Evaluation setup and Experimental results

Myriad2 was designed by Intel/Movidius and it is a low power SoC for computer vision and deep learning applications. It integrates 12 vector processing units that operate at 600MHz, 2 RISC processors, hardware accelerators (e.g. edge detection, various filters) a 2MB multi-ported SRAM (local memory) and a 512MB DDR2 DRAM (global memory).

The techniques for the implementation of the Winograd convolution in edge devices were evaluated using 4 widely used CNNs, which are dominated by layers of 3x3 convolution: *GoogleNet*, *SqueezeNet*, *VGG* and *YOLO-tiny*. Details of each CNN are summarized in Table 2, where execution time breakdown results show that a significant percentage of the execution time is spent on the 3x3 convolutional layers (from 46% up to 83%). The proposed

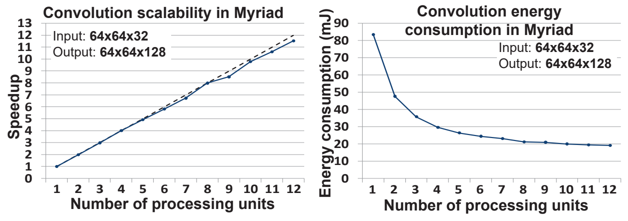
Table 2: Details of CNNs used for evaluation

CNN	input image	output vector	#layers	exec. time (ms)	#3x3 conv. layers	%ms spent in 3x3 conv. layers
GoogleNet	$[224 \times 224 \times 3]$	$[1 \times 1000]$	74	96.20	10	45.51
SqueezeNet	$[227 \times 227 \times 3]$	$[1 \times 1000]$	30	46.73	8	47.48
VGG	$[224 \times 224 \times 3]$	$[1 \times 1000]$	15	733.49	8	83.38
YOLO-tiny	$[448 \times 448 \times 3]$	$[1 \times 1470]$	15	117.49	8	62.23



P2 Computes: Y6, bottom-half Y1
P3 Computes: Y10, bottom-half Y13
P0 Computes: Y2, top-half Y1
P1 Computes: Y14, top-half Y13
P8 Computes: Y5 → z1
P9 Computes: Y9 → z3
P6 Computes: Y7, bottom-half Y4
P7 Computes: Y11, bottom-half Y16
P4 Computes: Y3, top-half Y4
P5 Computes: Y15, top-half Y16
P10 Computes: Y8 → z2
P11 Computes: Y12 → z4

Figure 5: Sharing and folding in Myriad2.



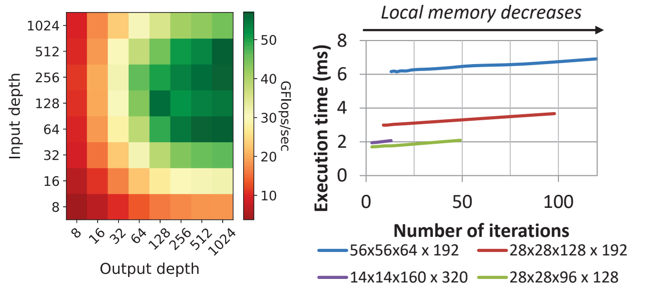
(a) Convolution exec. time vs. number of cores. (b) Convolution energy consumption vs. number of cores.

Figure 6: Software techniques applied in convolution.

implementation of Winograd is compared with (i) direct convolution (i.e. *nxn* convolution as an accumulation of *nxn* times of 1x1 convolutions) (ii) im2col convolution (i.e. *nxn* convolution as 1x1 convolution after altering data layout) and (iii) the combination of the direct and im2col algorithms that results in the lowest execution time in each CNN.

The instantiation of *sharing* and *folding* techniques in Myriad2, which integrates 12 processing units, is depicted in Fig. 5. There exist 3 pipeline stages, and each group contains a single unit. Polling for synchronization is avoided, by leveraging Myriad-specific hardware implemented buffers, which are named *fifos*. There are 12 *fifos* and each one is assigned to a single processing unit. Each *fifo* consists of 16 slots of 8 Bytes. *Fifos* are accessible to application level and can be used for direct message passing between the processing units. Finally, execution time and energy consumption were measured based on features provided by the Myriad2 SDK. All evaluation results refer to the CNN inference execution time and operations are performed under the IEEE 754 fp16 standard.

Before proceeding with the evaluation based on the use cases, we examined the scalability of the Winograd convolution after applying the proposed data transfer and management optimizations.



(a) Conv. depth vs. sharing/folding performance. (b) Conv. execution time vs. pipeline iterations.

Figure 7: Impact of CNN and architectural specifications.

As shown in Fig. 6a, it scales almost linearly with the number of processing units. The energy consumption evaluation results are shown in Fig. 6b. Although power increases with the number of active processing units, the energy consumption is dominated by the execution time, rather than by the power: As the execution time significantly drops with the number of active processing units, due to the increasing parallelism, energy decreases, as well.

The evaluation results are presented in Table 3. For each one of the 4 CNNs the execution time of the whole CNN inference is shown (column *Full Net*) and for the 3x3 layers specifically, is shown in column *3x3 layers*. Each CNN is evaluated with the convolutional layers implemented using direct convolution, im2col, the most efficient implementation of them (*Best combination*) and the Winograd convolution based on the proposed software techniques. The last 2 rows show the execution time improvement of the Winograd implementation over the *Best combination* implementation. The execution time of the full inference is reduced for all CNNs implemented using the Winograd algorithm based on the proposed techniques: 28.8% for GoogleNet, 22.5% for SqueezeNet, 42% for VGG and 24.4% for YOLO-tiny. The execution time results for the 3x3 layers only are presented to highlight the impact of the proposed techniques on the convolutional layers. Significant gains are observed for all CNNs, ranging from 31.7% (YOLO-tiny), up to 54% (GoogleNet).

4.2 Observations and discussion

The data transfer and management optimizations reduce the frequency of DMA transfers by increasing the available local memory space and mitigate their overhead by enabling the overlapping of computation and communication. They also increase the impact of the *sharing* and *folding* techniques which are used to efficiently deploy Winograd 3x3 convolution on edge devices. In this subsection we examine the parameters that affect the performance improvements for the proposed techniques.

Table 3: Evaluation results in ms

	GoogleNet		SqueezeNet		VGG		YOLO-tiny	
	Full Net	3x3 layers	Full Net	3x3 layers	Full Net	3x3 layers	Full Net	3x3 layers
Direct conv.	127.4	50	57	30	742.3	620.5	135.3	88.7
Im2Col conv.	98.9	46	47	20.7	789.7	667.7	126.3	80
Best combination	96.2	43.8	46.7	22.2	733.5	611.6	117.5	73.1
Winograd conv.	68.5	20.2	36.2	11.9	425.6	303.3	88.8	50
Gain (ms)	27.7	23.6	10.5	10.3	307.9	308.3	28.7	23.1
Gain %	**28.8%**	54%	**22.5%**	46.5%	**42%**	50.4%	**24.4%**	31.7%

Using a pipeline to implement the Winograd based on *sharing* and *folding* techniques, enabled by application-independent embedded system optimizations provides significant improvements to the deployment of computationally intensive CNNs in edge devices. It can be derived from Table 3 that GoogleNet implementation improves from 10fps (*Best combination*) to 14fps, SqueezeNet improves from 21fps to 27fps, VGG from 1fps to 2fps and YOLO-tiny from 8fps to 11fps.

The impact of *sharing* and *folding* depends on **the number of layers in which Winograd is applicable**, as well as on the **computational requirements** of these layers. The performance of Winograd varies, depending on the **combination of sizes between the input and output depth**. Fig. 7a presents the performance of the Winograd convolution for a 56x56 input image with various depth sizes. Clearly, very small depths provide poor performance, mainly due to the inability of utilizing the SIMD features of the vector processing units. However, for larger depths, the performance exhibits small variations irrespective of the ratio between input and output depth.

Another important aspect is the efficiency of the pipeline schema shown in Fig. 5. A reduction of the **available local memory space** per processing unit, increases the number of pipeline iterations required to compute a specific convolution. Fig. 7b presents the execution time and the number of iterations for 3x3 Winograd convolution (after applying all proposed optimizations) for various input sizes used in GoogleNet, when the available local memory size ranges from 100 KB (minimum number of iterations) and drops up to 40 KB (maximum number of iterations). Naturally, the number of iterations increases when the available local memory size drops.

A significant observation based on Fig. 7b is that the synchronization overhead between the processing units is constant, as shown by the constant slope in all lines. Also, this figure highlights the impact of the data transfer and management optimizations: Increasing the local memory available space for useful data reduces the number of required iterations, as well as the execution time of the CNN inference. For, example, for input size 56x56x64, increasing memory space from 40KB to 100KB, reduces the execution time of a single 3x3 convolution by 15%. Finally, the **number of vector processing units** affects the impact of all techniques, since convolution is a compute bound operation.

With respect to the energy consumption comparison between direct and Winograd, both implementations leverage the same number of VPUs, use the DMA engine in a similar fashion and the same matrix-matrix multiplication assembly kernel. Indeed, GoogleNet best combination requires 1.87W, while Winograd 1.84W.

5 CONCLUSIONS

The software techniques proposed in this work can be effectively used to efficiently implement the Winograd algorithm for convolution in CNN-based applications deployed in edge devices that provide a set of vector processing units that access a high bandwidth local memory. This is achieved by combining application-agnostic data management software techniques, along with Winograd-specific ones. The evaluation results based on 4 widely used CNNs show significant performance improvements, leading to efficient deployment of CNN-based applications in low power edge devices.

ACKNOWLEDGEMENTS

We would like to thank the the Eyes of Things project team (http://eyesofthings.eu) for the support and the fruitful discussions. This work has received funding from the European Union's Horizon 2020 research and innovation programmes under grant agreement No 780572 SDK4ED (www.sdk4ed.eu) and under grant agreement No 687628 VINEYARD (vineyard-h2020.eu).

REFERENCES

[1] F Catthoor and et al.. 2013. *Data access and storage management for embedded programmable processors*. Springer Science & Business Media.

[2] L. Cavigelli, D. Gschwend, C. Mayer, S. willi, B. Muheim, and L. Benini. 2015. Origami: A convolutional network accelerator. In *Proc. GLSVLSI'15*. pp. 199–204.

[3] Cisco. 2016. *Cisco Global Cloud Index: forecast and methodology, 2015–2020*. Technical Report.

[4] J. Cong and Xiao B. 2014. Minimizing computation in convolutional neural networks. In *Proc. ICANN'14*.

[5] O. Deniz and et al.. 2017. Eyes of Things. *Sensors* vol.17(5) (2017).

[6] G. Efland, S. Parkh, H. Sanghavi, and A. Farooqui. 2016. High performance DSP for vision, imaging and neural networks. In *Hot Chips 28 Symposium (HCS'16)*.

[7] J Huang and et al.. 2017. Speed/accuracy trade-offs for modern convolutional objects detectors. In *Proc. CVPR'17*.

[8] F. Iandola and K. Keutzer. 2017. Keynote: Small neural nets Aare beautiful: enabling embedded systems with small deep-neural-network architectures. In *Proc. ESWEEK'17*.

[9] A. Lavin and S. Gray. 2016. Fast algorithms for convolutional neural networks. In *Proc. CVPR'16*. pp. 4013–4021.

[10] A. Mallik and et al.. 2011. MNEMEE-An automated toolflow for parallelization and memory management in MPSoC platforms. In *Proc. DAC'11*.

[11] D. Moloney. 2016. Embedded deep neural networks: the cost of everything and the value of nothing. In *Hot Chips 28 Symposium (HCS'16)*.

[12] L. Papadopoulos, D. Soudris, I. Walulya, and P. Tsigas. 2016. Customization methodology for implementation of streaming aggregation in embedded systems. *Journal of Systems Architecture* 66 (2016), pp. 48–60.

[13] PH. Pham, D. Jelaca, C. Farabet, B. Martini, Y. LeCun, and E. Culurciello. 2012. NeuFlow: Dataflow vision processing system-on-a-chip. In *Proc. MWSCAS'12*. pp. 1044–1047.

[14] Y. Siegel. 2016. The path to embedded vision & AI using a low power vision DSP. In *Hot Chips 28 Symposium (HCS'16)*.

[15] S. Yi, C. Li, and Q. Li. 2015. A survey of fog computing: concepts, applications and issues. In *Proc. Workshop on Mobile Big Data*. pp. 37–42.