University of Calabria

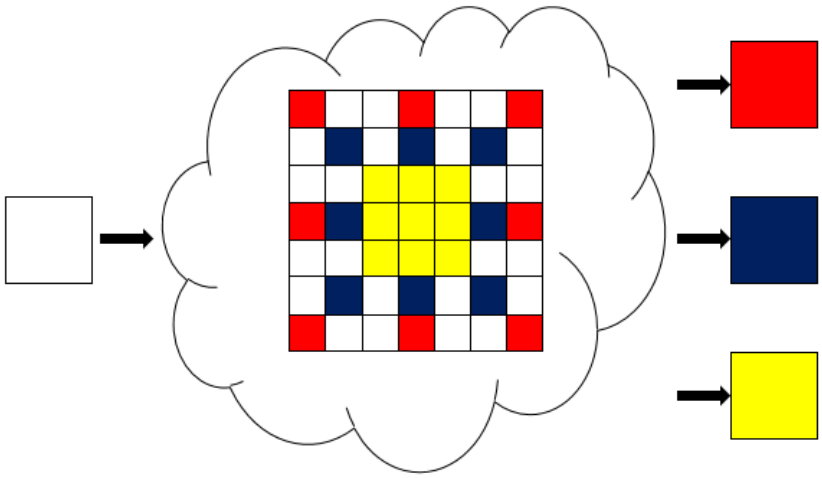
Department of Informatics, Modeling,
Electronics and System Engineering

Rende, Cosenza, Italy

XILINX
| UNIVERSITY PROGRAM

Project report

A SoC implementation of Atrous Spatial Pyramid Pooling for Fully Convolutional Networks



Student: Cristian Sestito

Supervisor: Prof. Stefania Perri

Team number: **xohw19-188**

YouTube link: https://youtu.be/csjqzAHQjns

Contents

1. Introduction

This work aims to introduce a hardware implementation of a Semantic Image Segmentation technique, the *Atrous Spatial Pyramid Pooling* (ASPP) [1] by using the Zedboard. The correct functionality of the whole system is tested through a comparison between the results stored in the DDR3 and the results provided by a MATLAB script which emulates the ASPP behaviour. Semantic Image Segmentation has a primary role in the Image Processing, and it consists of a pixel-level prediction, since it handles the classification and detection of several classes of objects in a frame [2]. Nowadays, this technique is usually performed by using Fully Convolutional Networks (i.e. Neural Networks in which Fully Connected layers are not present).

Biomedical Image Processing is a charming field in which this task is applicable: Magnetic Resonance Imaging (MRI) acquisitions request an accurate segmentation to ensure valid diagnosis and Deep Learning has been proved a valuable choice. Gray Matter (GM) Segmentation is a peculiar example: studies have shown that changes of this tissue are related to neurological diseases, as the Amyotrophic Lateral Sclerosis. [3] have shown a Deep Learning solution to segment GM. This Neural Network consists of several layers, in which each input (*feature map* tensor) is subjected to a transformation: the 3D Convolution is the most recurrent. In particular, the Atrous Spatial Pyramid Pooling layer allows a valid segmentation of the GM by performing several Dilated Convolutions jointly. The utility of this layer has encouraged a study about Dilated Convolutions and Pyramid Pooling, whose utility is not just limited to the biomedical context but is also applicable in several other tasks (e.g. scene parsing).

The Dilated Convolution (also known as *Atrous Convolution*) consists of an upsampled 2D convolution and allows the filtering of each feature map by splitting it into large windows (i.e. fields-of-view according to the Deep Learning terminology). In a canonical convolution, each filter coefficient is applied consecutively to the window, while in the Dilated Convolution it is applied in a sparsely way, laid down by the dilation rate r (i.e. a hop, a hole between adjacent window elements). Fig.1 depicts the situation. This strategy allows the production of dense prediction maps (i.e. output feature maps), by keeping the spatial resolution of each map along the network; this solution performs better segmentation tasks than these performed by the encoder-decoder networks [4], in which spatial resolution is usually destroyed by pooling layers or striding sampling and then recovered by using transposed convolutions. The contemporary execution of dilated convolutions with different upsampling rates overcomes the problem of segmenting objects at multiple scales: the goal is the use

of a small filter kernel (usually a 3×3 one) for all the rates r, so the number of parameters is considerably lower than a traditional segmentation network.

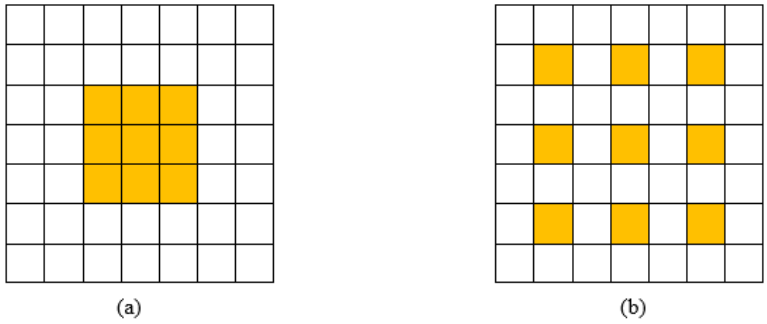


Fig.1: Traditional 2D convolution (a) vs Dilated Convolution (b)

It is useful to formalize the problem in a mathematical manner. Let consider a square matrix of order m (i.e. the window size) where each entry is numbered with a progressive value from 0 to $m^2 - 1$: if the field-of-view size is $k = 3$, the (1) generalizes each 3×3 matrix obtainable by applying any possible dilation rate $r \in \mathbb{N}$, where each value represents the position of the respective element in the $m \times m$ matrix.

$$\boldsymbol{M_r} = \begin{pmatrix} \frac{m^2-1}{2} - r(1+m) & \frac{m^2-1}{2} - rm & \frac{m^2-1}{2} + r(1-m) \\ \frac{m^2-1}{2} - r & \frac{m^2-1}{2} & \frac{m^2-1}{2} + r \\ \frac{m^2-1}{2} - r(1-m) & \frac{m^2-1}{2} + rm & \frac{m^2-1}{2} + r(1+m) \end{pmatrix} \quad (1)$$

In this case, $\boldsymbol{r}$ = [6,12,18,24] so 4 parallel dilated convolutions are performed by covering fields-of-view of order $k_{e,r} = k + (k-1)(r-1)$; therefore, $\boldsymbol{k_e}$ = [13,25,37,49].

Since a 2D convolution of 3D feature maps is just the sum of the homologous filtered values in each intermediate output feature map, the process can be speeded up by parallelizing these convolutions. Therefore, FPGAs are a valid choice; moreover, parallelism is an intrinsic property of the Atrous Spatial Pyramid Pooling, so it falls well in a FPGA design.

The ASPP network also includes a Global Average Pooling (GAP) layer, which aims to accurately locate [5] the segmented object in the final prediction map, by collecting the arithmetic mean of each feature map; a 1×1 convolution layer is also present: it consists of a feature map scaling and it basically acts as a feature pooling layer. In the design, Dilated Convolutions and Global Average Pooling have been implemented by using hardware resources, while the Zynq Processing System performs 1×1 convolution because it is a simple product between each feature map value and a coefficient.

2. Design

This section provides a concise description of the design and the software control handled by the Processing System.

2.1. Hardware

The whole system includes 4 key components, as depicted in Fig.2:

- The **ASPP Core**, which implements the neural network;
- The **Data DMA**, which is responsible for transferring the feature maps between the ASPP Core and the external DDR3;
- The **Weights DMA**, which is responsible for transferring the coefficients between the external DDR3 and the ASPP Core;
- The **ZYNQ Processing System**, which aims to control the ASPP Core and to configure DMAs properly; it also performs 1×1 convolution.

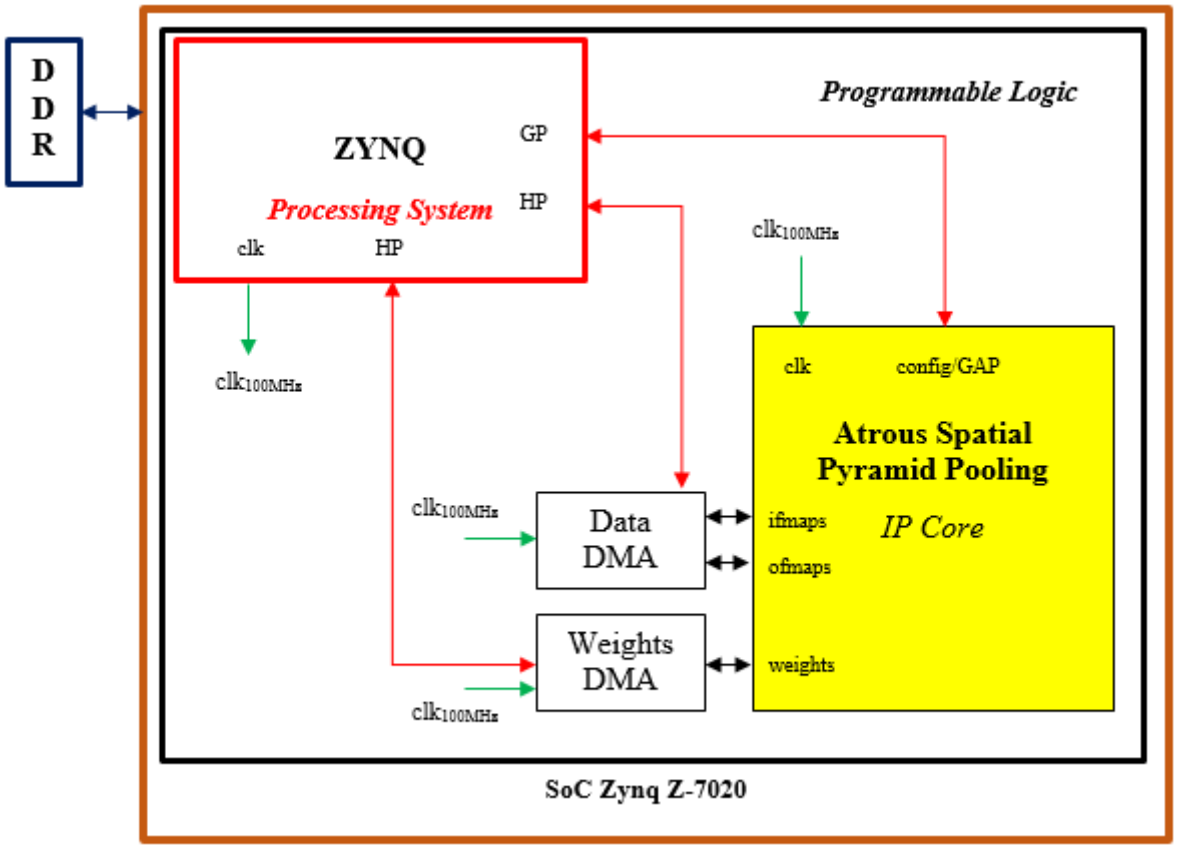


Fig.2: The Embedded System

2.1.1. The ASPP Core

The Atrous Spatial Pyramid Pooling IP Core (Fig.3) is highly parameterizable, since it has been described by using VHDL parametric-like constructs (e.g. *for… generate*).

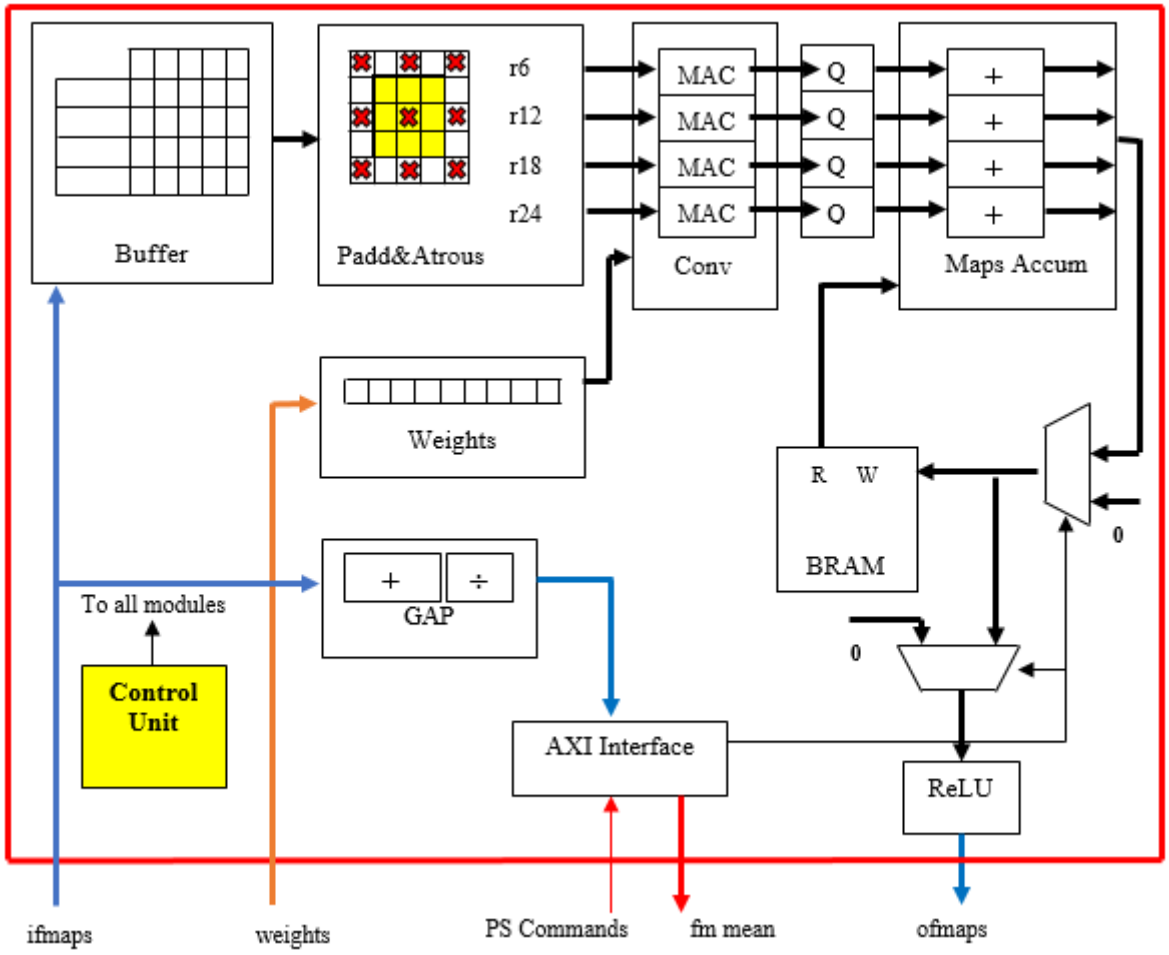


Fig.3: The Atrous Spatial Pyramid Pooling IP Core

In the following lines, fixed values will be considered to better understand the workflow.

The component receives a 200×200 feature map (8-bit grayscale format) and a 3×3 coefficient kernel (8-bit signed format). The former is stored in a sliding-window *Buffer* (Fig.4) which aims to arrange pixels properly for the following 2D convolution; the latter is stored in a shift-register (*Weights*).

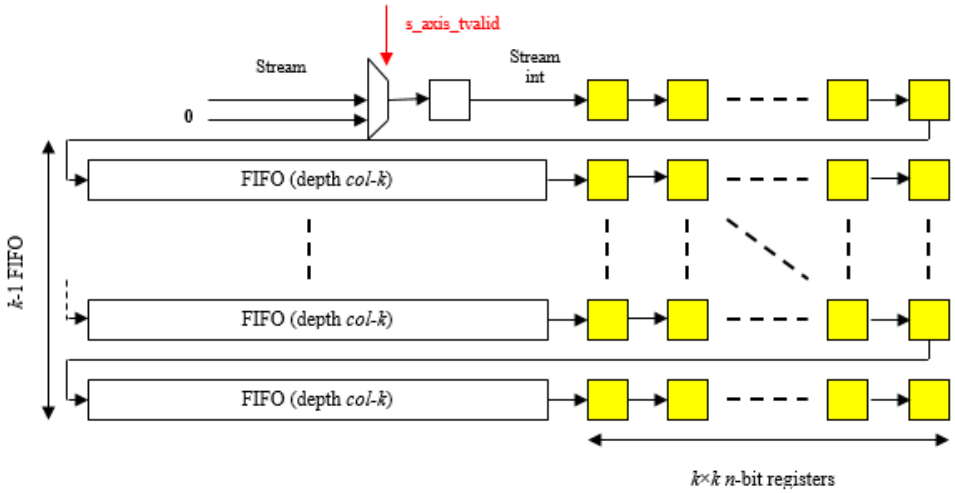


Fig.4: The sliding-window buffer

After a certain latency, the *Padd&Atrous* (Fig.5) module performs zero-padding (which handles edge pixels properly) and arranges each current window in 4 Atrous 3×3 windows (according to ***r*** = [6,12,18,24], as described in section 1). For each frame line, a counter handles the columns; if the column is interested in zero-padding, each decoder provides a valid selection signal for the muxes in order to send a zero-value instead of the current pixel.

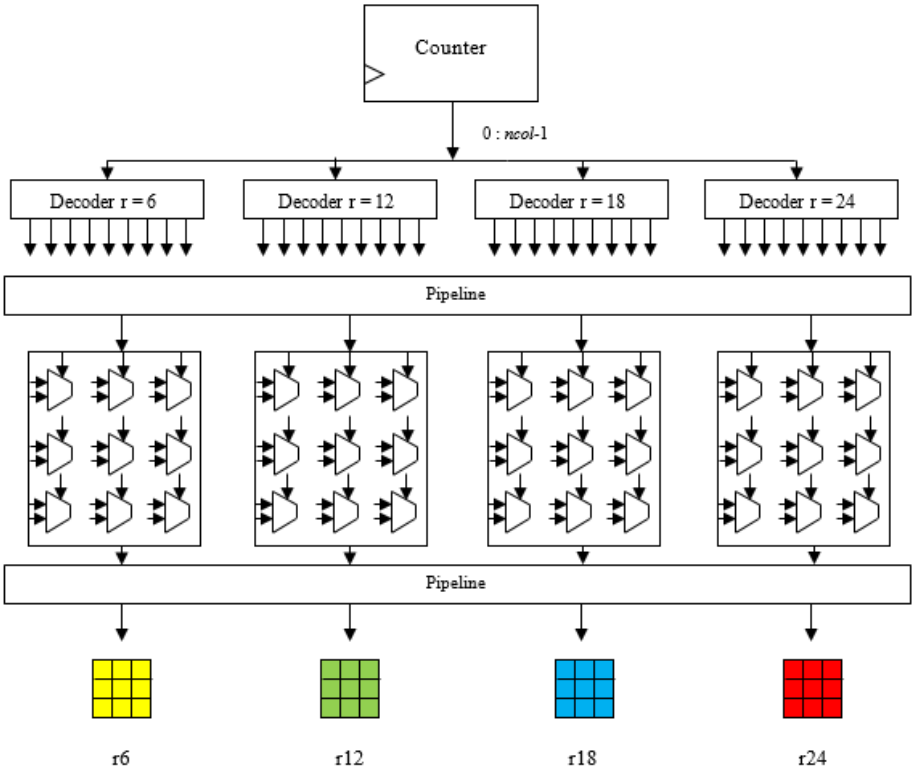


Fig.5: The Padd&Atrous module

After that, 2D convolution is performed and, for each clock cycle, 4 parallel 20-bit signed filtered pixels are provided; hence, these values are quantized[1] to an 8-bit signed format. The *Maps Accum* module handles the sum of homologous filtered pixels in several feature maps to extend convolution to 3D fmaps according to the input. In the first cycle, each filtered pixel is added up to a zero value, since there is no previous corresponding filtered pixel. Each quartet is arranged in a 32-bit word and stored in a Simple Dual Port BRAM with a depth 200×200 = 40k (according to the feature map size). BRAM has been implemented by using the *Block Memory Generator* IP Core [6].

Therefore, the next feature map is provided, and the cycle repeats itself in the same way. In this case, each filtered pixel is added up to the previous homologous word.

When the last feature map is provided to the IP core, after the accumulation, the *Rectified Linear Unit* (ReLU) is performed. This non-linear function emulates the impulse-coding style typical of human brain, and hence of a canonical neural network. Its mathematical expression and an example are provided in Fig.6.

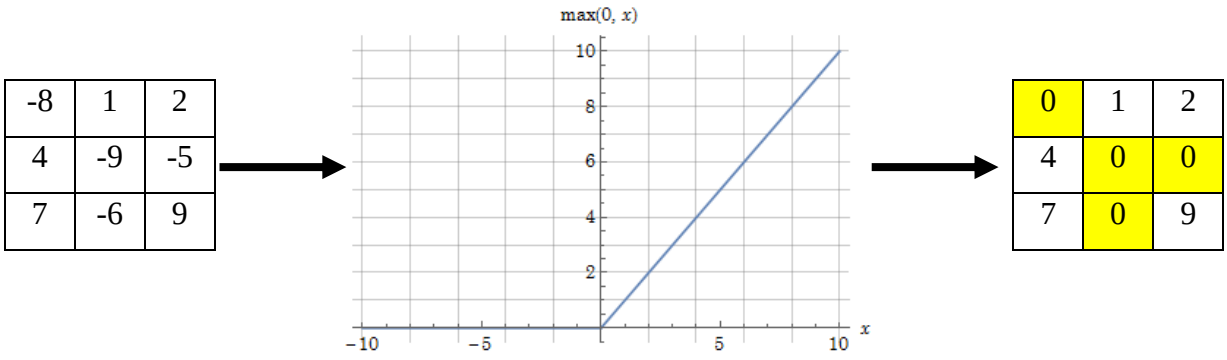


Fig.6: The Rectified Linear Unit

The Global Average Pooling is performed together with the convolutions: when each pixel is provided to the IP Core, a 24-bit unsigned accumulator collects the sum of all the pixels; at the end, a bit-shift is performed: it emulates the division between this sum and the number of elements of the feature map (40k); since 40k is not a power of 2, the division algorithm would be complex. So, since 2D convolution is quantized, it has been decided to quantize the mean pooling too, through a division by the nearest power of 2 (2^{15} = 32 768) and it consists of a right-shift of 15 positions.

The whole IP Core is controlled by a Finite State Machine (FSM – *Control Unit*). Feature map and weight transfer is handled by an AXI4-Stream interface whose signals are controlled by the FSM. An AXI4-Lite interface, whose VHDL script is provided by Xilinx IP Core Generator, manages 2

[1] Bit quantization is necessary to overcome resource limitation (since intermediate feature maps are stored in BRAM) and to adapt data to the available bandwidth for the DMA transfer. In section 3.1 more details are provided.

signals provided by the Processor: the *start*, which opens all the operations, and the *ReLU enable*, which activates the clock enables of the ReLU registers; this interface allows the transfer of the mean value of each feature map, too.

Each module consists of at least a pipeline stage: this choice aims to speed the performances up. Furthermore, each register is clock-enabled so it is in a quiescence state when a module must not perform operations: this solution minimizes power consumption; moreover, this solution allows blocking of the IP Core when AXI4-Stream signals *slave ready* or *slave valid* go suddenly down.

2.1.2. The DMAs

Each feature map is sent through a Direct Memory Access modality, by using Xilinx DMA IP Core [7]. Both read and write channels are enabled: the former handles input feature maps, while the latter the output ones.

Each coefficient is sent in the same way. This DMA is configured in Micro-Mode operation since it handles only 9 weights; the read channel is the only one enabled. Both DMAs are configured by the Processor, therefore they work in Direct Mode.

2.2. Software

A C software routine runs on the Zynq Processing System (PS). First, input feature maps and coefficients are stored in the external DDR3; then the PS enables the ASPP Control Unit.

The PS configures properly the Weights DMA, according to Direct Mode [7]:

Step 1. DMA Reset;

Step 2. DMA MM2S Port Activation;

Step 3. Choice of DDR Source Address from which data read starts.

Step 4. Choice of Byte to transfer.

After that, first feature map is sent to the ASPP core in a similar way. After a polling control which allows the full transfer, the PS receives the mean pooling current value through the AXI Interface and stores it into the DDR. This process is repeated for all the feature maps.

During the last cycle, the PS enables ReLU registers and the Data DMA is ready to receive output values. At the end, the PS performs the 1×1 convolution.

2.2.1. Test setup

The design has been implemented by using the ZedBoard. In the following section, test setup instructions are provided.

1) Program the SoC and run the C script in the SDK. The main ASPP actions are printed to the Serial Port (e.g. in Fig.7).

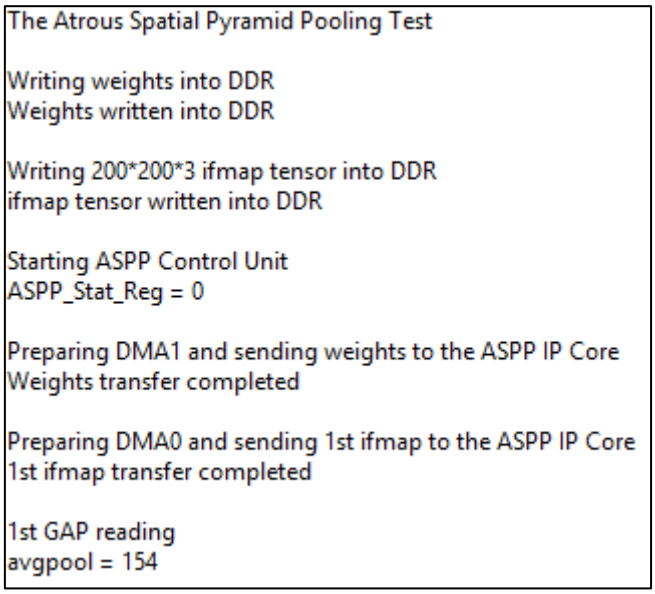


Fig. 7: The Serial Port

2) Write ofmap results, stored in the DDR, in a text file, by using the XSCT prompt, as in Fig.8.

```
XSCT Process
xsct% set logfile [open "C:\\Users\\Cristian\\Dropbox\\MATLAB\\hw_results.txt" "w"]
file7bf43fec0
xsct% puts $logfile [mrd 0x01300000 40000]
xsct% close $logfile
```

Fig.8: The XSCT Prompt

3) Open the text file (e.g. in Fig.9) and delete the address column.

```
13270DC:   03031010
13270E0:   0F0C040C
13270E4:   0B150A06
13270E8:   0A040A0C
13270EC:   0C060809
13270F0:   0A0B0F0F
13270F4:   0E060411
13270F8:   0A110C04
13270FC:   0B0F140A
```

Fig.9: A part of the text file

4) Run the MATLAB script; a positive message will be printed to the Command window (i.e. "*Correct results*").

2.3. Design reuse

The ASPP Design is highly reusable, since most of the VHDL scripts have been written in a parametric style (e.g. in Fig.10). In particular:

- The *Buffer* consists of 2 components: several rows of D N-bit registers and several FIFOs of D N-bit registers.
- The *Padd&Atrous* module consists of a N-bit counter with an upper limit MAX (for the Zero-Padding management), several decoders (with an arbitrary input bit dimension N, which provide valid selectors to the muxes according to the fmap size COL and the padding size P) and several 2to1 muxes with an arbitrary word size N. The top-level structure provides a number of decoders and mux banks equal to the rate array size. In this case size(*rates*) = 4 (this is a global parameter, handled by a package definition).
- The *Weights Unit* consists of several (equal to size(*rates*)) rows of D N-bit registers and a control unit which handles AXI4-Stream interface related to the coefficient transmission between the DMA1 and the ASPP IP core. In this case the transmission refers to only 9 weights since a 3×3×1 filter has been used, but an arbitrary F×F×D tensor could be sent.
- The *Conv* Module consists of size(*rates*) parallel 3×3 MACs. Each MAC component consists of custom Multipliers (A N-bit unsigned, B N-bit signed) And Adders (A,B N-bit signed) implemented by using the DSP48E1 slices.
- The *fmaps Accumulator* provides the sum between homologous filtered values; the number of operands refers to size(*rates*).
- A 2to1 N-bit mux bank handles BRAM location reset during the last cycle.
- The *BRAM*, implemented by using the Block Memory Generator, refers to 40k 32-bit words. Hence, BRAM reuse is guaranteed by using the IP GUI properly.
- The *ReLU* module consists of size(*rates*) parallel ReLU units. Each ReLU unit performs this non-linear function if enabled by the Processing System. A 2to1 mux bank handles sleep-phase and awake-phase.
- The *Control Unit* orchestrates IP operations. It is reusable, since it is handled by latency parameters which are:
  - ✓ Buffer Latency: $\frac{K_{e\_max}-1}{2} * (col + 1) + 1$.

- ✓ MAC Latency: 5 (1 mult, 4 sums).
- ✓ Frame Latency: $row * col$.

- The *Global Average Pooling* consists of a N-bit accumulator (where N refers to the worst case related to a feature map (i.e. if each pixel refers to an 8-bit format, its maximum value is 255; if map sizes are *row* and *col*, then N = $\lceil \log_2(255 * row * col) \rceil$), a register in which mean value is stored and a FSM which handles the accumulation and register enabling when necessary.
- The AXI4-Interface script is provided by Xilinx.

```
SD : for i in 0 to RATE_WIDTH-1 generate
    SD_i: sel_decoder_param generic map(N => CNTB, COL => COL, P => R(i))
          port map(clock => clock, reset => reset, clock_enable => clock_enable_int,input => count_int, output => se
end generate SD;

-- Generic Dilated 3x3 Window. Refer to the documentation for more details.
WI: for i in 0 to RATE_WIDTH-1 generate
    win_int(i)(0) <= win_in((KE*KE-1)/2-R(i)*(1+KE));
    win_int(i)(1) <= win_in((KE*KE-1)/2-R(i)*KE);
    win_int(i)(2) <= win_in((KE*KE-1)/2+R(i)*(1-KE));
    win_int(i)(3) <= win_in((KE*KE-1)/2-R(i));
    win_int(i)(4) <= win_in((KE*KE-1)/2);
    win_int(i)(5) <= win_in((KE*KE-1)/2+R(i));
    win_int(i)(6) <= win_in((KE*KE-1)/2-R(i)*(1-KE));
    win_int(i)(7) <= win_in((KE*KE-1)/2+R(i)*KE);
    win_int(i)(8) <= win_in((KE*KE-1)/2+R(i)*(1+KE));
end generate WI;

MUXi: for i in 0 to RATE_WIDTH-1 generate
    MUXj: for j in 0 to WEIGHT_WIDTH-1 generate
        MUXij : multiplexer_2to1_param generic map(N => DATA_WIDTH) port map(input_1 => win_int(i)(j), input_2 => (c
    end generate MUXj;
end generate MUXi;
```

Fig.10: Parameterized VHDL example: part of the Padd&Atrous component

The whole design is available at https://github.com/csestito94/ASPP_HW.

3. Results

In this section, details about the implementation phase are provided. After a discussion about the main design challenges, results in terms of resource utilization, power consumption and performances are presented.

3.1. Design challenges

During the design phase, two main challenges have been encountered: the parameterization level and the size of each bit word.

A well parameterized design lets the designer adapt network features quickly. Since hardware design deals with the available resources and their organization, it is very difficult to abstract all the network features (better known as hyperparameters) as it happens by using only software. In section 2.3, the most relevant parameterization choices have been just presented.

The size of each bit word deals with two key elements of the design: the BRAMs included in the ASPP IP Core and the DMAs which act as a bridge between the neural network and the external DDR3. After the convolution, the 8-bit word size increases to 20-bit; this size should be extended to a 32-bit format in order to store it in a BRAM. Each cycle produces 4×40k filtered value, so 5120000 bit ≈ 5000 kb. The SoC has 140 36kb-BRAM, so the whole information would fill 5000 kb/36kb ≈ 139 BRAM! Therefore, it is not a good idea because BRAM utilization would be ≈ 100% and no one would be available for other tasks. Moreover, the Zynq Z-7020 Processor has 4 HP Port, so it could handle a maximum of 4 DMAs jointly. If each DMA held a feature map value, 4 ones would be necessary for all the rates; therefore, no DMA would be available for the coefficients. All these reasons have encouraged bit quantization from 20 to 8 bit.

3.2. Resource utilization

The utilization about the single ASPP IP Core is provided in Table 1, in which information about Slices are reported. BRAM utilization is 26.07% (36 36kb-BRAM, 1 18kb-BRAM), while DSP utilization is high: 73 DSP (33.18%) due to the presence of 4 parallel convolution processes and the accumulation logic for 3D fmaps and for the Global Average Pooling.

Site Type	Used	Available	Util%
Slice LUTs	3718	53200	6.99
LUT as Logic	870	53200	1.64
LUT as Memory	2848	17400	16.37
LUT as Shift Register	2848		
Slice Registers	1660	106400	1.56
Register as Flip Flop	1660		

Table 1: ASPP IP Core Utilization

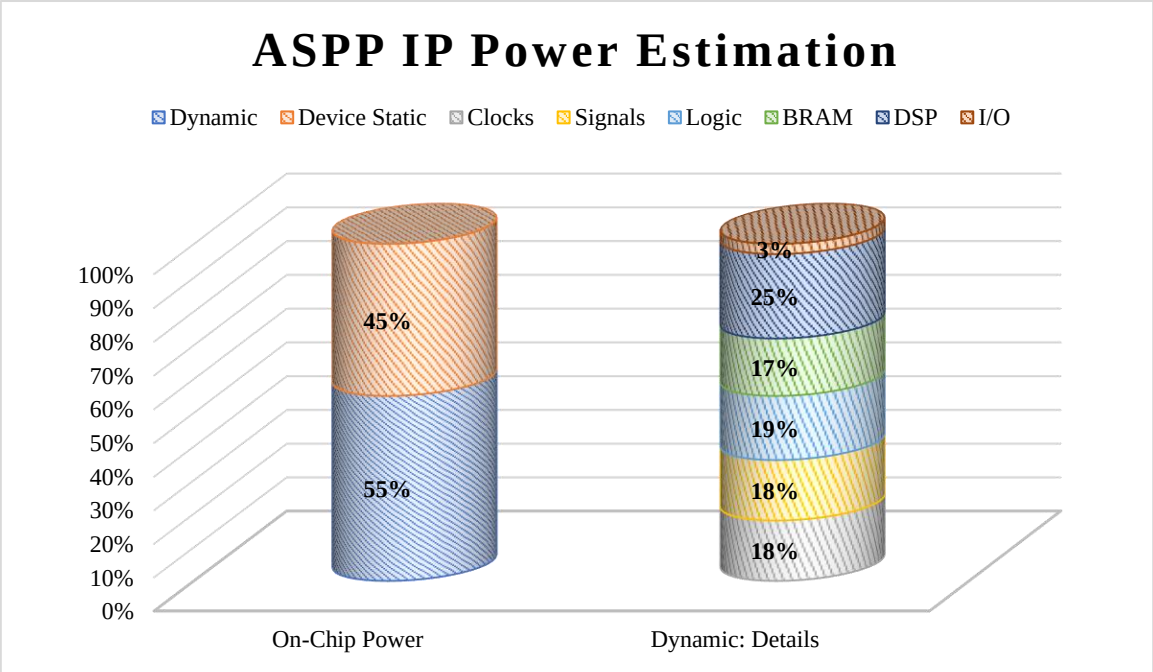
The utilization about the whole system is instead provided in Table 2.

Site Type	Used	Available	Util%
Slice LUTs	8583	53200	16.13
LUT as Logic	5014	53200	9.42
LUT as Memory	3569	17400	20.51
LUT as Distributed RAM	502		
LUT as Shift Register	3067		
Slice Registers	7561	106400	7.11
Register as Flip Flop	7561		
F7 Muxes	1	26600	< 0.01

Table 2: System Utilization

3.3. Power consumption

The ZedBoard doesn't allow a direct power consumption analysis, so it has been handled through an estimation by using a SAIF file. The report in Table 3 refers to the ASPP IP Core, since a simulation of the whole block design isn't granted.



Total On-Chip Power (Confidence Level: *High*)	**0.241 W**
Device Static	0.108 W
Dynamic	0.133 W
Clocks	0.024 W
Signals	0.023 W
Logic	0.026 W
BRAM	0.023 W
DSP	0.033 W
I/O	0.003 W

Table 3: ASPP power consumption

3.4. Performances

The system clock frequency is f = 100 MHz. According to the network timing[2], as in Fig.11, each output feature map is available after (2):

$$t_{ofm*4} = (31 \cdot 44834 + 44835) \cdot 10^{-8} s \approx 14.35\, ms \qquad (2)$$

[2] This timing refers to the control unit. In the picture, each block represents a different state.

According to the Worst Negative Slack, the maximum frequency achievable is (3):

$$MaxFreq = \left\lfloor \frac{1}{(10 - 1.182) \cdot 10^{-9}} \frac{1}{s} \right\rfloor = 113\ MHz \qquad (3)$$

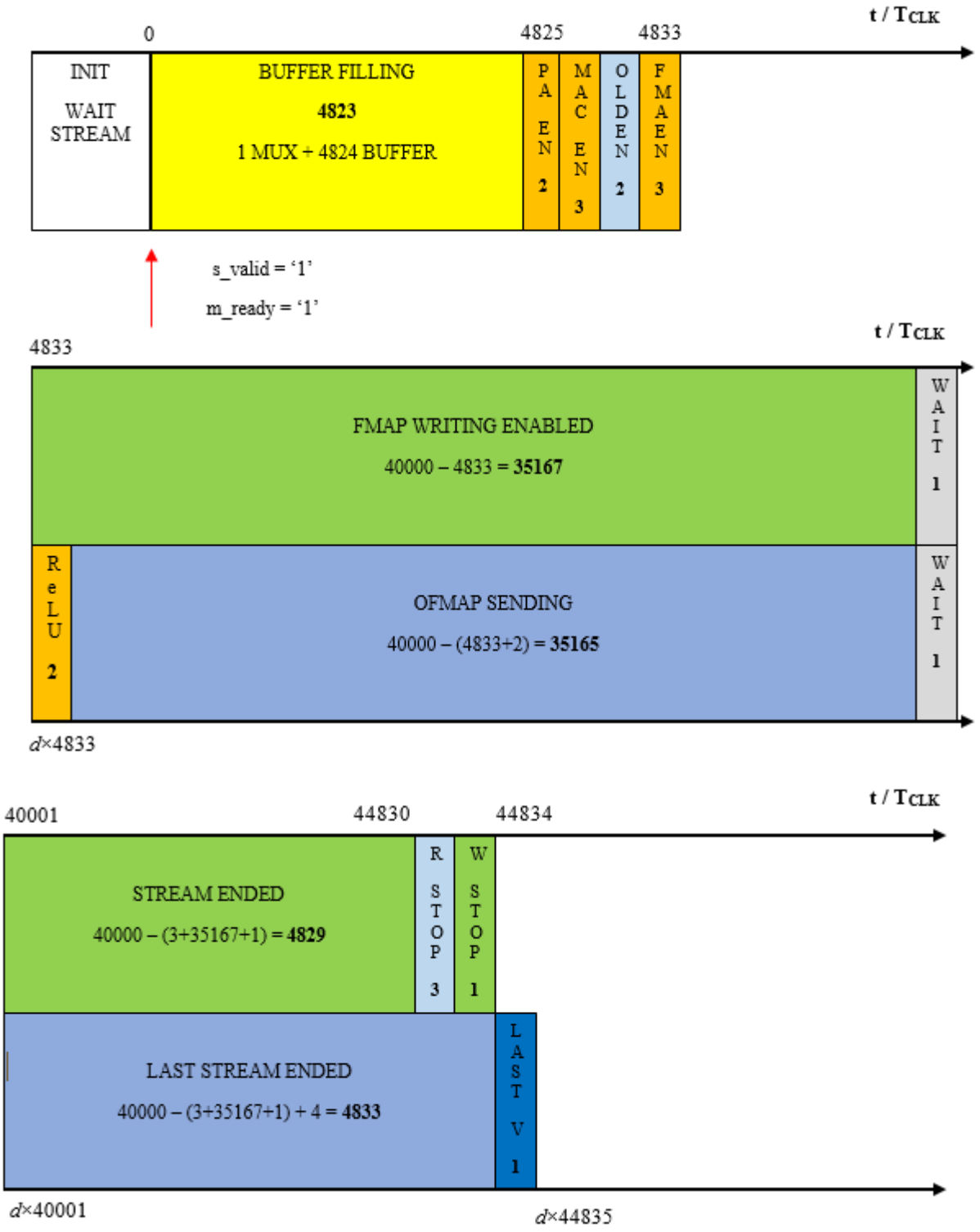


Fig.11: ASPP Timing

4. Conclusions

This work has introduced a novel IP core solution for the Atrous Spatial Pyramid Pooling for the task of Semantic Segmentation by using Fully Convolutional Networks.

Based on the Spinal Cord Gray Matter segmentation [3], the ASPP IP Core represents a valid prototype which aims to be improved thanks to its high parameterization capability: therefore, it can be used in several other applications as expected by the ASPP genesis [1].

The intrinsic parallelization provided by the ASPP algorithm is the main goal for a FPGA implementation choice; the remarkable literary background about the Image Processing has even more encouraged the hardware design. Despite the 4 parallel dilated convolutions and the high size of the convolutional windows ($k = 49$), the FFs utilization has been shown minimal (1.69%) thank to a careful VHDL design which has allowed a good synthesis. This result encourages the replication of this hardware in wider SoCs.

The IP Core, which basically performs typical Fully Convolutional Neural Network operations (convolutions, mean pooling, ReLU), has been put into a complete embedded system and this choice has allowed the designer the development of SoC design skills (e.g. the data transfer from an external memory to an internal IP and vice versa by using the AXI solution).

The tests performed have shown the correct mathematical functionality through a comparison between the results stored in the external DDR and the results provided by a MATLAB script, but no effective segmentation ability has been tested. Therefore, this initial work requires future developments: first, the design could be optimized, by exploiting the capability of more powerful SoCs; for example, the processing time could be reduced, by using a double buffering solution [8] and the parallelization level could be increased so each output feature map could be provided in a really competitive time.

As described in section 3.1, the design has required a bit quantization but no information about segmentation accuracy is available. Therefore, it could be useful investigate it by using real datasets (i.e. real feature maps and weights); if the analysis shows an important degradation, the quantization rate will be reduced.

Hence, the bet about this task is surely right since it could lead to valid results.

5. References

[1] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 40, no. 4, pp. 834–848, Apr. 2018.

[2] X. Liu, Z. Deng, and Y. Yang, "Recent progress in semantic image segmentation," *Artif. Intell. Rev.*, pp. 1–18, Jun. 2018.

[3] C. S. Perone, E. Calabrese, and J. Cohen-Adad, "Spinal cord gray matter segmentation using deep dilated convolutions," *Sci. Rep.*, vol. 8, no. 1, p. 5966, Dec. 2018.

[4] H. Noh, S. Hong, and B. Han, "Learning Deconvolution Network for Semantic Segmentation," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1520–1528.

[5] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning Deep Features for Discriminative Localization," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2921–2929.

[6] Xilinx and Inc, "Block Memory Generator LogiCORE IP, PG058(v8.3)," 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_3/pg058-blk-mem-gen.pdf. [Accessed: 19-Mar-2019].

[7] Xilinx and Inc, "AXI DMA LogiCORE IP, PG021 (v7.1)," 2018. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf. [Accessed: 19-Mar-2019].

[8] S. Mittal, "A survey of FPGA-based accelerators for convolutional neural networks," *Neural Comput. Appl.*, pp. 1–31, Oct. 2018.