High Performance Computing

02. Ottimizzazioni in HLS

Gianluca Brilli
(Gianluca.Brilli@Unimore.it)
AA 2018-2019

Obiettivi

- L’obiettivo di questa esercitazione è quello di consolidare le conoscenze che abbiamo appreso nel blocco di esercitazioni precedente ed introdurre alcuni nuovi concetti importanti quando si effettua Hardware design con HLS.
- Nello specifico vedremo come scrivere la DDR dall’FPGA, tramite il protocollo **AXI4 Master**.
- Inoltre vedremo nel dettaglio le **ottimizzazioni** che possono essere svolte per parallelizzare i nostri IP.

HLS – Esercitazione 04

- Come prima esercitazione introduttiva vediamo come **scrivere in DDR**, andando a realizzare un modulo hardware che emula il funzionamento della *memcpy*.

```
void copyMem(uint8_t *dst, uint8_t *src, size_t bytes) {

#pragma HLS INTERFACE m_axi port=dst offset=slave bundle=ddr
#pragma HLS INTERFACE m_axi port=src offset=slave bundle=ddr

...
...
```

- Gestiamo il numero di elementi ed i segnali di controllo del modulo tramite **AXI-Lite**.

HLS – Esercitazione 04

- Il parametro offset = slave permette di gestire in maniera dinamica l'indirizzo nel quale andare a scrivere i dati in RAM.
- Il trasferimento dell'indirizzo verso l'FPGA viene gestito tramite un'interfaccia di tipo **AXI-Lite**, dopo la sintesi verrà generato un driver del tipo *X<modulo>_Set_<param>();*
- Per esempio supponendo di voler scrivere l'array src all'indirizzo *0x10000000*:

- *Xsum_Set_src(&XCopyMemInstance, 0x10000000);*

Ottimizzazioni – BRAM e Registri

- Tipicamente conviene posizionare all'interno delle memorie più veloci i dati che vengono acceduti più di frequente.
- All'interno della Zynq 7000 sono presenti le seguenti risorse:

Name	BRAM_18K	DSP48E	FF	LUT
Available	280	220	106400	53200

- I Flip-Flops sono presenti in maniera minore rispetto alle BRAM_18K, ma sono più veloci.
- Le **variabili** sono memorizzate all'interno di registri (blocchi di FF). Gli **array** sono memorizzati in BRAM.

- *Come vedremo, è perciò fondamentale trasferire parte dei dati in DDR in queste memorie.*

Ottimizzazioni – Loop Unrolling

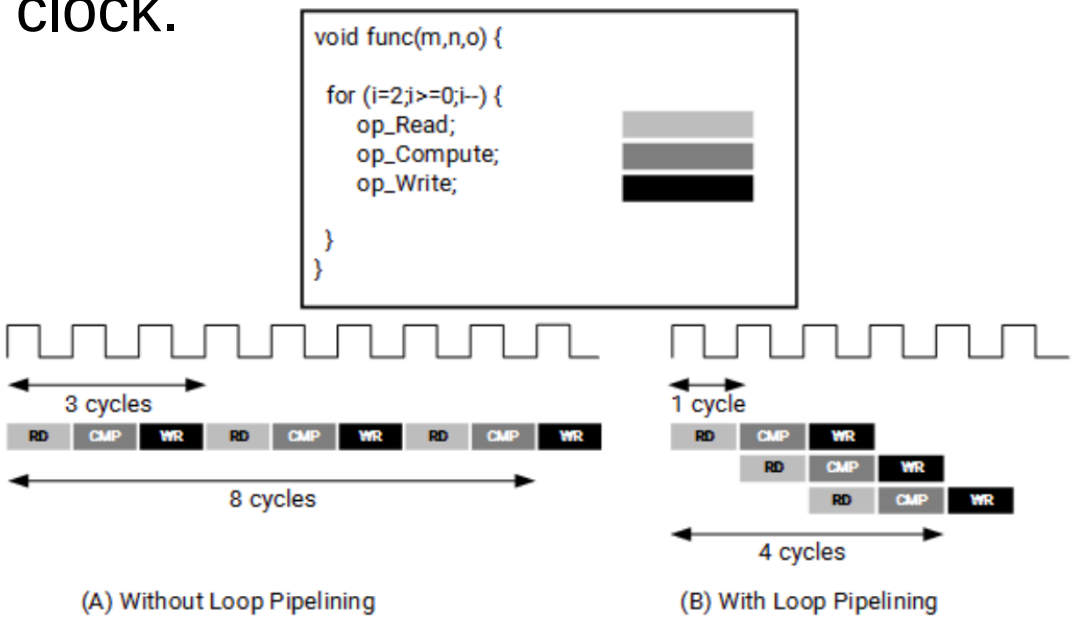
- Permette l'esecuzione in parallelo delle iterazioni all'interno di un loop.

```
for (int i = 0; i < n; ++i)
{
    #pragma HLS UNROLL

    c[i] = a[i] + b[i];
}
```

- Aumenta notevolmente il grado di parallelismo, ma è importante tenere a mente che aumenta anche il numero di risorse utilizzate di un fattore n.
- Inoltre aumenta esponenzialmente anche il tempo necessario per la sintesi *(anche da minuti a ore)*.

Ottimizzazioni – Pipelining

- Permette l'esecuzione di più istruzioni in un singolo ciclo di clock.




```
for (int i = 0; i < n; ++i)
{
    #pragma HLS PIPELINE

    c[i] = a[i] + b[i];
}
```

- A differenza del loop unrolling, il numero di risorse ed il tempo di sintesi restano più o meno invariati.

Convoluzione – Cos'è?

- In generale la convoluzione è un'operazione matematica, che date due funzioni ne produce una terza, definita come segue:

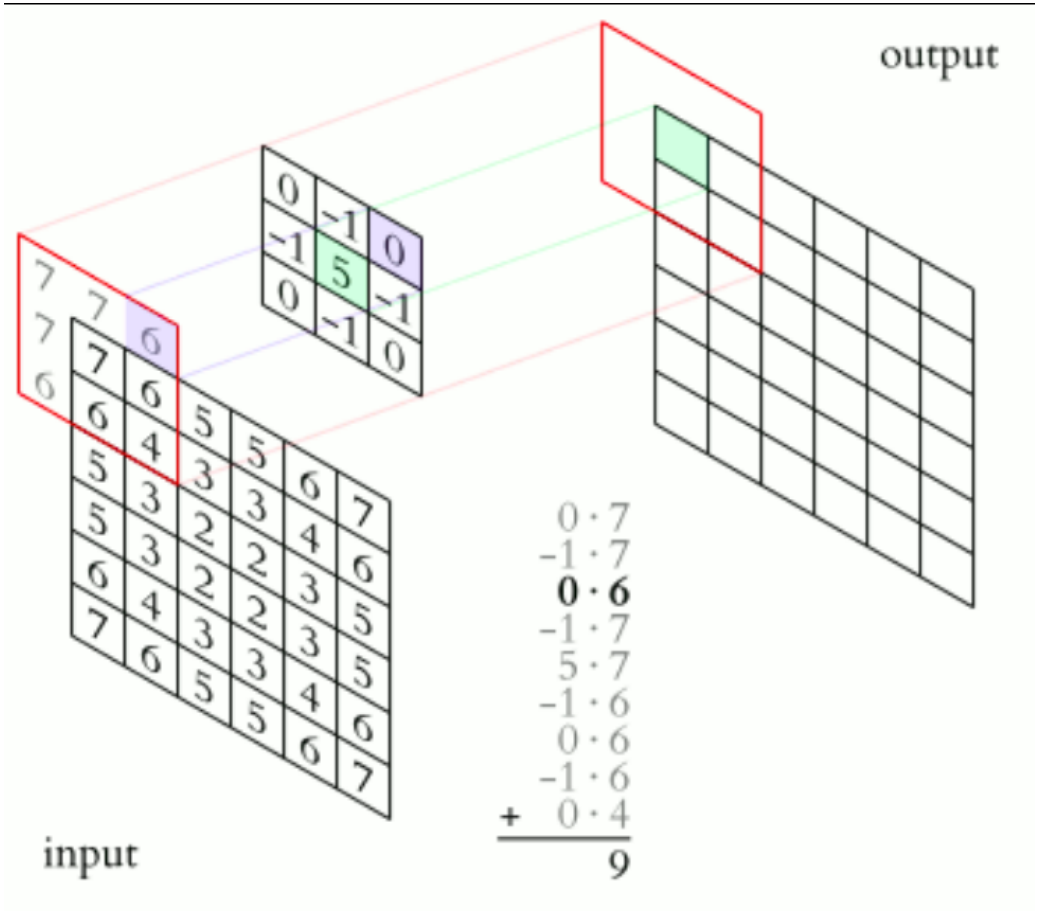
$$(f*g)(t)=\int f(\tau)g(t-\tau)d\tau$$

- Nel caso delle immagini, le funzioni f e g sono rispettivamente due canali, limitati nell'intervallo [0, 255], quindi la precedente formula si può esprimere come:

$$(f*g)(x,y)=\sum_{i=0}^{M}\sum_{j=0}^{N}f(x,y)g(x-i,y-j)$$

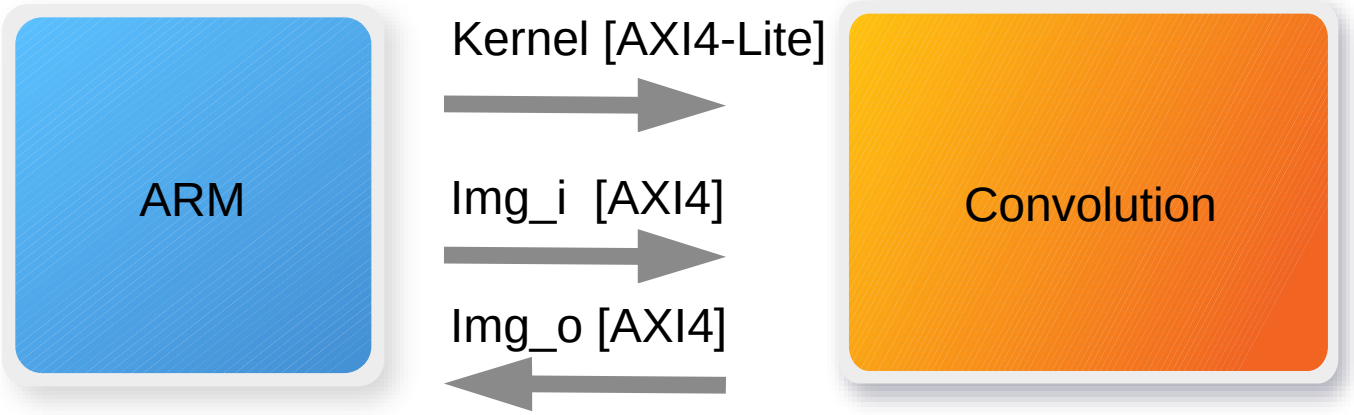
Convoluzione – Cos'è?

- Tale operazione possiamo rappresentarla nel seguente modo:



Esercitazione 04

- Provare a realizzare il design seguente:



Esercitazione 04

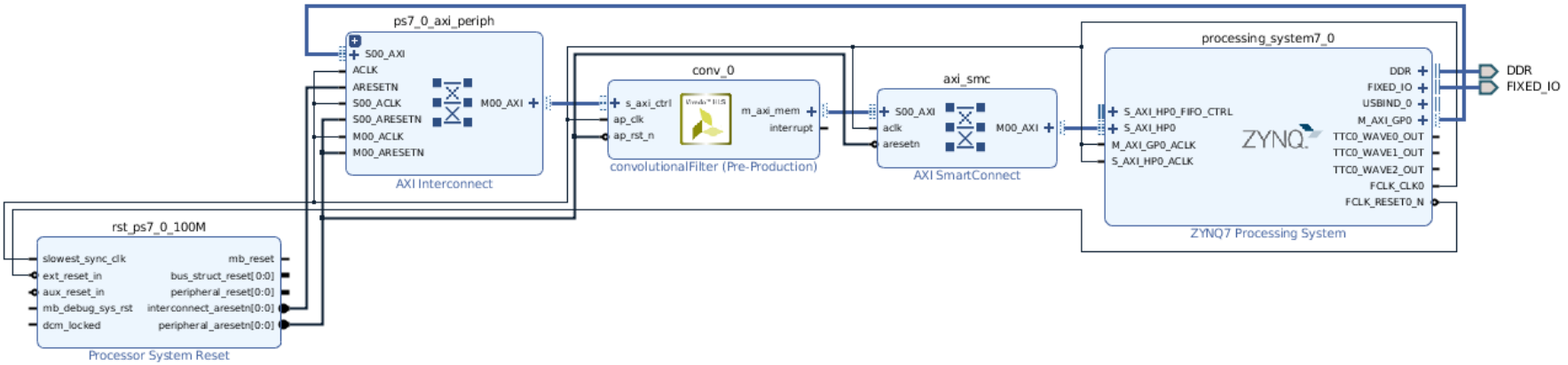
```c
#include "conv.h"

void conv(
    uint8_t    image[ROW_IMG][COL_IMG][CHN_IMG],
    uint8_t newImage[ROW_IMG][COL_IMG][CHN_IMG],

    float * filter, int filterDim){

    float sum;

    int i, j, d, m, n, mm, nn;
    int kCenterX, kCenterY;
    int rowIndex, colIndex;

    // find center position of kernel (half of kernel size)
    kCenterX = filterDim / 2;
    kCenterY = filterDim / 2;

    for(i=0; i < ROW_IMG; ++i)
    {
        for(j=0; j < COL_IMG; ++j)
        {
            for(d=0; d < CHN_IMG; ++d)
            {
                sum = 0;
                for(m=0; m < filterDim; ++m)
                {
                    mm = filterDim - 1 - m;

                    for(n=0; n < filterDim; ++n)
                    {
                        nn = filterDim - 1 - n;

                        // index of input signal, used for checking boundary
                        rowIndex = i + (kCenterY - mm);
                        colIndex = j + (kCenterX - nn);

                        // ignore input samples which are out of bound
                        if(rowIndex >= 0 && rowIndex < ROW_IMG && colIndex >= 0 && colIndex < COL_IMG) {
                            sum += image[rowIndex][colIndex][d] * filter[filterDim * mm + nn];
                        }
                    }
                }
                newImage[i][j][d] = (uint8_t)((float)fabs(sum) + 0.5f);
            }
        }
    }
}
```

- Partiamo dal seguente frammento di codice.
- **Cicli di clock** necesari a completare la convoluzione:
- **61 210 081**

Esercitazione 04 - Test

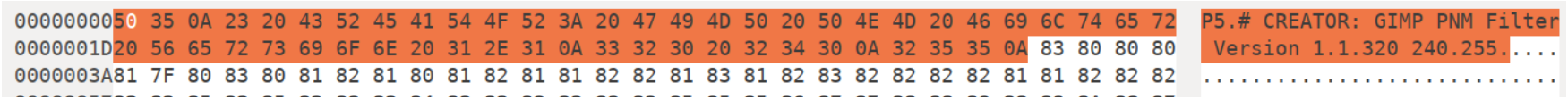
- Testiamo alcune ottimizzazioni e verifichiamo come cambia il numero di cicli di clock necessari e l'utilizzazione delle risorse, nello specifico:
  - *BRAM*: in modo da evitare di accedere alla DDR.
  - *Loop Unrolling*: provare ad effettuare unrolling sui cicli for più interni dell'algoritmo di convoluzione.
  - *Loop Pipelining*.

Esercitazione 04 - Design



Test: Immagine

- Una volta sintetizzato e programmato l'FPGA, per verificare se la nostra convoluzione effettivamente funziona, dobbiamo caricare un'immagine di prova dal nostro PC.
- Per semplicità utilizziamo immagini in **formato PPM**, questo formato contiene un header molto semplice nel quale è presente la dimensione dell'immagine, segue poi l'immagine effettiva codificata con **24 bit per pixel** nel seguente formato: *R0 G0 B0 R1 G1 B1 R2 G2 B2, ...*



Test: Precaricamento DDR

- Per il caricamento dell'immagine sulla memoria della Zynq utilizziamo il tool **XSCT** *(Xilinx Software Command Line Tool*).
- Per prima cosa dobbiamo collegarci alla board tramite i seguenti comandi:

```
xsct% connect

xsct% target 1
```

Test: Precaricamento DDR

- Successivamente, tramite i comandi riportati in basso dobbiamo inizializzare la DDR della board, che dopo l'accensione viene mantenuta in **stato di reset**:

```
xsct% source design_1_wrapper_hw_platform_0/ps7_init.tcl

xsct% ps7_init
```

Test: Precaricamento DDR

- Infine scriviamo l'immagine sulla DDR con il seguente comando:

```
xsct% mwr -bin -file ritratto.ppm 0x10000000 19214
```

- Il penultimo parametro è l'**indirizzo di partenza** su cui scriveremo la nostra immagine. E' importante utilizzare un indirizzo abbastanza alto per evitare di sovrascrivere i segmenti del programma.
- L'ultimo parametro è il **numero di word** di cui si compone il file *(words = bytes / 4)*.

Test: Precaricamento DDR

- La lettura è praticamente analoga alla scrittura:

```
xsct% mrd -bin -file out.ppm 0x10000000 19214
```

- Dopo aver dato questo comando, verranno scritti sul file out.ppm (sul nostro PC), *19214*4* bytes a partire dall'indirizzo *0x10000000* sulla DDR della Zynq.
- L'immagine processata dopo aver eseguito l'algoritmo in FPGA.

Test: Filtri utilizzati

```
float gaussianFilter[25] = {

        1/256.0f,  4/256.0f,  6/256.0f,  4/256.0f, 1/256.0f,
        4/256.0f, 16/256.0f, 24/256.0f, 16/256.0f, 4/256.0f,
        6/256.0f, 24/256.0f, 36/256.0f, 24/256.0f, 6/256.0f,
        4/256.0f, 16/256.0f, 24/256.0f, 16/256.0f, 4/256.0f,
        1/256.0f,  4/256.0f,  6/256.0f,  4/256.0f, 1/256.0f
};

float edgeDetectionFilter[9] = {

        0,  1, 0,
        0,  0, 0,
        0, -1, 0
};
```

Test: Edge Detection

Test: Gaussian Blur