

Sistemi Digitali - Riassunto

**

Created: January 12, 2023 6:15 PM

Tags: Embedded Systems, University, University Notes

Useful Link: <https://virtuale.unibo.it/>

Scopo di questo documento

Questi appunti sono una rivisitazione del documento redatto da *Andrea Salvucci e Lorenzo Felletti*.

Riporto di seguito il loro disclaimer che condivido:

Abbiamo creduto di fare cosa gradita nel mettere a disposizione queste dispense, che sono una rielaborazione ottenuta dal confronto tra slide del corso e registrazioni a nostra disposizione. Ci teniamo a precisare che questo elaborato non intende in nessun modo sostituire le lezioni in presenza, tanto più che potrebbero esserci imprecisioni sparse qua e là.

A tal proposito, invitiamo chiunque voglia farlo a modificare opportunamente il sorgente in Markdown messo a corredo di questo PDF.

N.B.: abbiamo redatto questo documento basandoci sulle lezioni fino all'A.A. 2021/2022, quindi la parte di parallel computing potrebbe essere meno dettagliata di quanto esposto in aula. Viceversa, è presente la sezione sulle reti neurali convoluzionali che, a quanto abbiamo appreso, non viene più esposta.

- *Andrea Salvucci e Lorenzo Felletti*

Parallel Computing

Per velocizzare l'esecuzione si possono usare tecniche di concorrenza e/o parallelismo. Queste tecniche però introducono fonti di ulteriore complessità

- Data Dependency → `op#2` bloccata da `op#1`
`c = a + b; // (op#1)` `y = x + a; // (op#2)`
- Data Race → risultato dipende dal thread che arriva prima ("vince la gara") 

- Più thread accedono alla stessa area di memoria (concorrentemente) e almeno uno modifica il dato → unpredictable behavior 🤦‍♂️ → necessità di meccanismi di sincronizzazione.

Strategie Hardware per Velocizzare l'Esecuzione

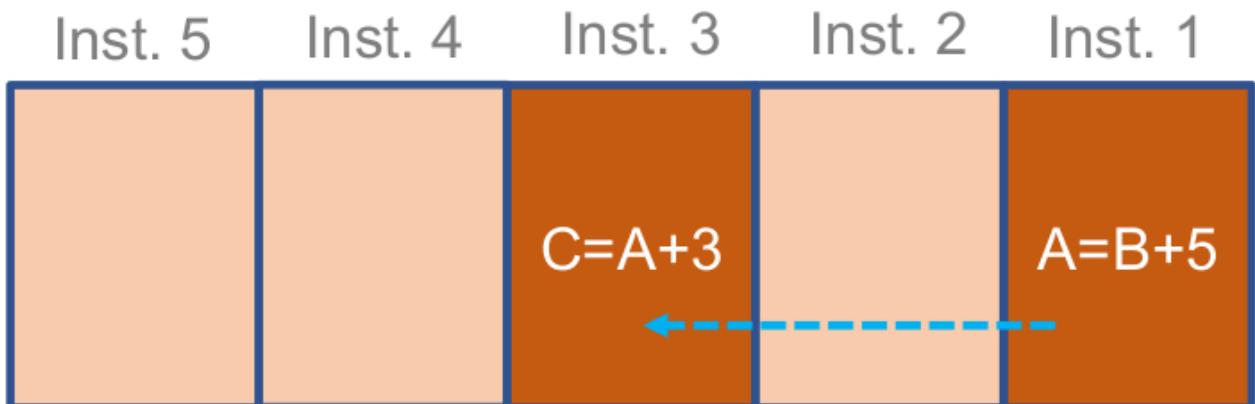
1) Aumentare la frequenza della CPU

- 😊 non c'è problema di data dependency
- 😞 più consumo batteria 🌊 | $pwr \propto (clock)^2$

2) Sfruttare pipelining

Si tratta di eseguire multiple istruzioni in maniera concorrente, ognuna in un diverso stadio di esecuzione (dunque, in pratica *non* è in parallelo).

- 😞 Questa implementazione richiede dei componenti addizionali
- 😞 Problemi di data dependency



😊 ma può essere mitigata grazie al operand forwarding

Operand forwarding (or **data forwarding**) is an optimization in [pipelined CPUs](#) to limit performance deficits which occur due to [pipeline stalls](#).^{[1][2]} A [data hazard](#) can lead to a [pipeline stall](#) when the current operation has to wait for the results of an earlier operation which has not yet finished.

Example [\[edit\]](#)

```
ADD A B C  #A=B+C
SUB D C A  #D=C-A
```

If these two [assembly](#) pseudocode instructions run in a pipeline, after fetching and decoding the second instruction, the pipeline stalls, waiting until the result of the addition is written and read.

Without operand forwarding

1	2	3	4	5	6	7	8
Fetch ADD	Decode ADD	Read Operands ADD	Execute ADD	Write result			
	Fetch SUB	Decode SUB	stall	stall	Read Operands SUB	Execute SUB	Write result

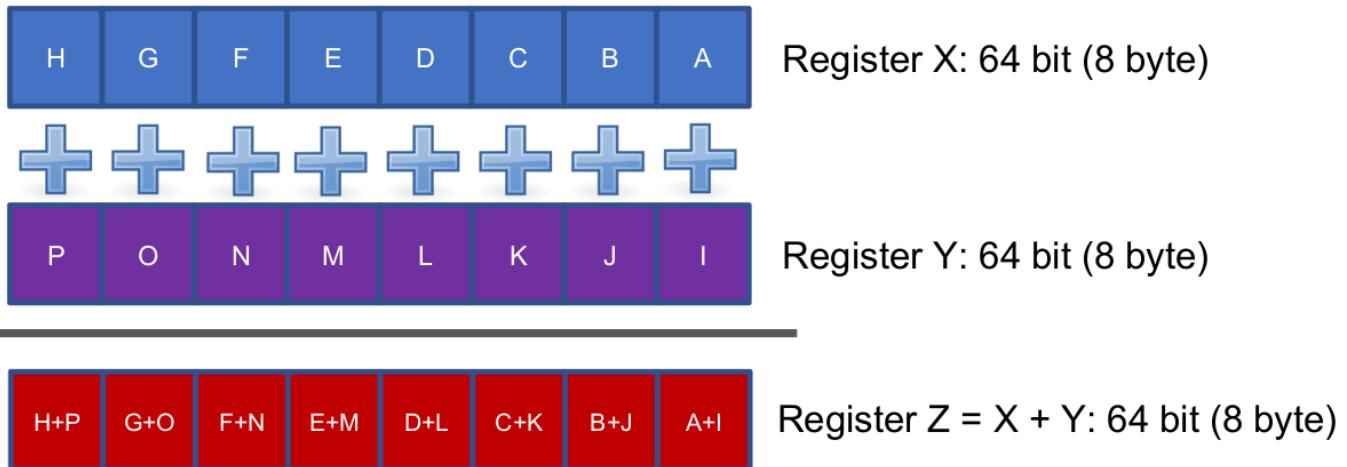
With operand forwarding

1	2	3	4	5	6	7
Fetch ADD	Decode ADD	Read Operands ADD	Execute ADD	Write result		
	Fetch SUB	Decode SUB	stall	Read Operands SUB: use result from previous operation	Execute SUB	Write result

In some cases all stalls from such read-after-write data hazards can be completely eliminated by operand forwarding:^{[3][4][5]}

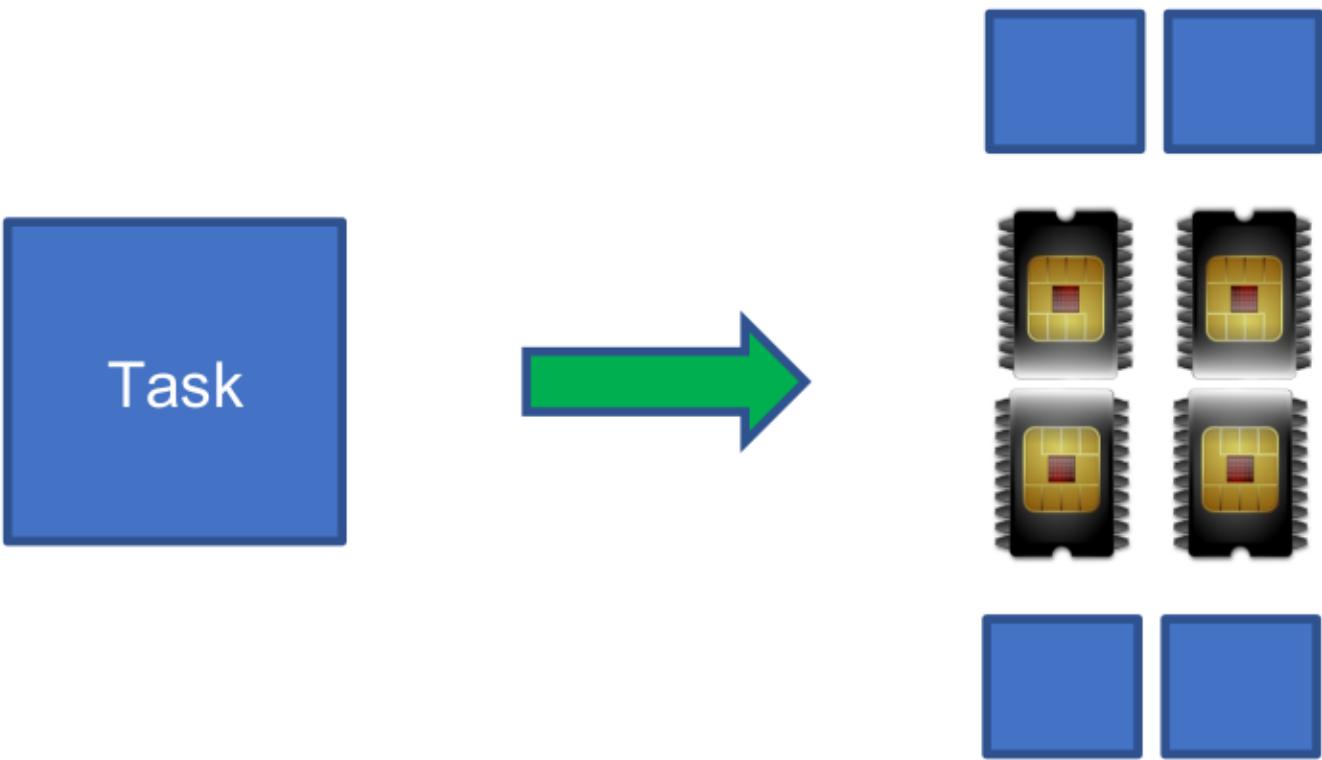
3) Operazioni multiple in un'unica istruzione (SIMD)

Questa strategia consiste nell'eseguire multiple operazioni identiche in una singola istruzione



4) Aumentare il numero di core

Questa strategia consiste nel diffondere la computazione totale in multiple unità di esecuzione (cores/CPUs)



- 😞 Questa strategia potrebbe essere più costosa rispetto a aumentare la frequenza di clock. Inoltre, non è sempre possibile.
- 😞 Problemi di data dependency/race issues

Strategie Software per Velocizzare l'Esecuzione

1) Pipelining

il Pipelining è una strategia molto usata nelle CPU moderne e i compilatori sono in generale molto ottimizzati per sfruttare questa feature.

Le CPU moderne inoltre utilizzano altre strategie quali Out-Of-Order (OOO) execution (o dynamic execution): un processore esegue le istruzioni in un ordine regolato dalla disponibilità di dati di input e unità di esecuzione, piuttosto che dal loro ordine originale in un programma.

2) SIMD instructions

Le migliori prestazioni si ottengono tipicamente agendo sul codice: organizzando i dati in strutture per calcoli SIMD e codificando istruzioni Assembly utilizzando linguaggi ad alto livello (ad es. C o C++)

Usabile quando:

- I dati sono ben organizzati in memoria (cache-friendly)
- La stessa operazione è applicata a più dati
- (ma può essere efficace anche in altri contesti).

Si sfrutta il fatto che la dimensione dei Multimedia Register (MM) è N volte la dimensione del

dato.

Accesso ai dati → un'istruzione può caricare più operandi (packed data).

Processamento → una singola operazione è applicata a più operandi.

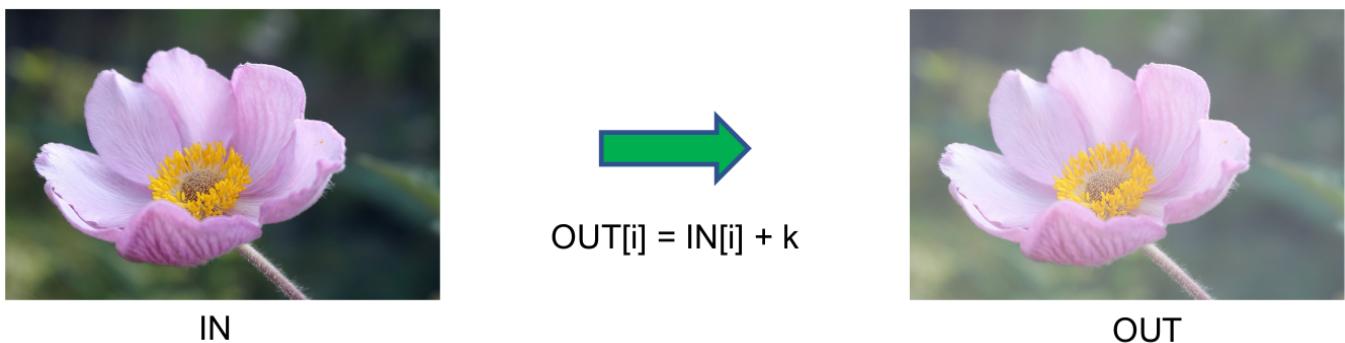
The larger the MM size, the higher the speedup

$groups_of_N_data \implies N_factor_speedup - (un)packing_time$

In theory, fitting and processing groups of N data would allow for an N factor speed-up compared to conventional scalar code (single instruction single data) execution

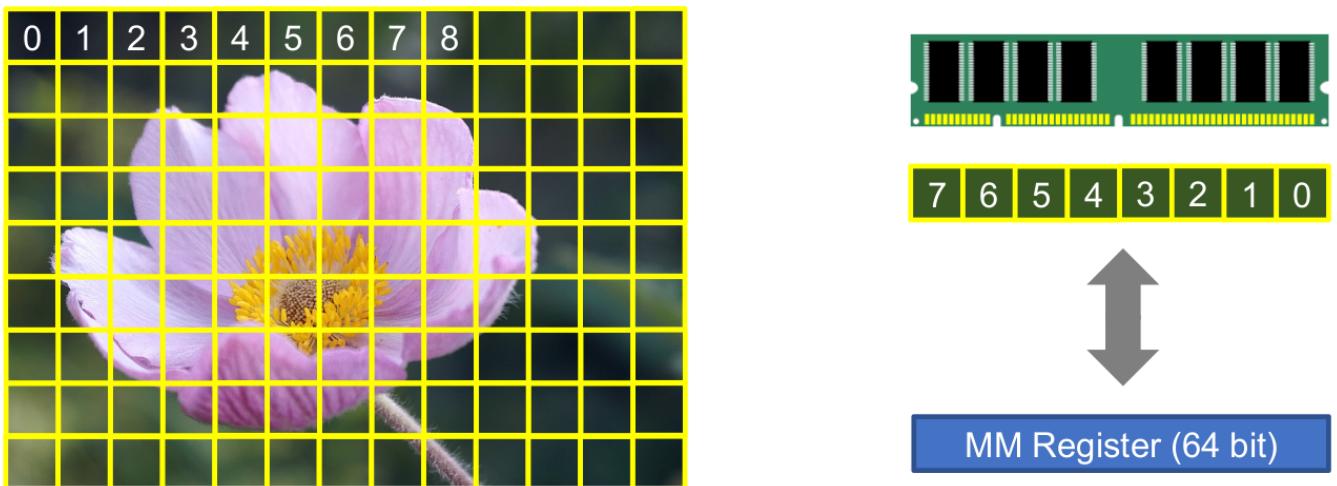
In practice, the speed-up is lower due to the overhead (primarily for packing/unpacking data), yet considerable and not demanding concerning resources and power consumption

Per questo questa strategia è usata spesso con immagini, multimedia e signal processing:



If the data is appropriately stored in memory, a single instruction can load multiple operands (e.g., 8 bytes representing 8 pixels) into a single multimedia register.

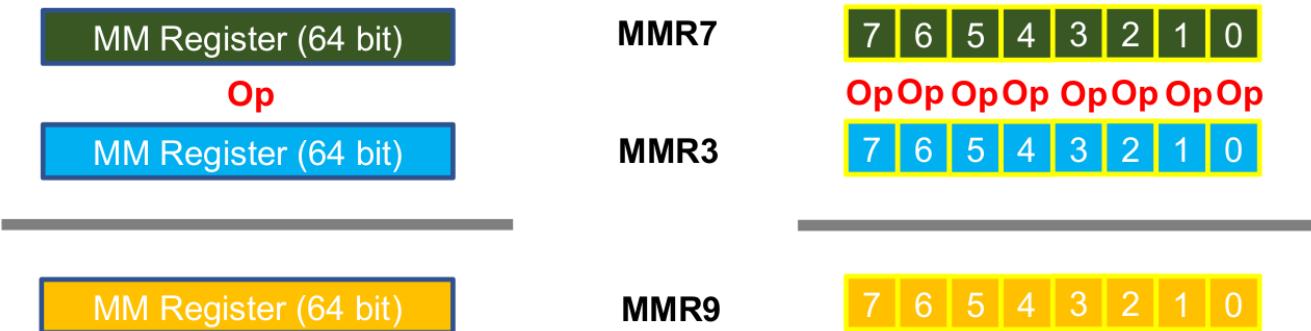
The same paradigm applies when storing data to memory from a multimedia register



Once chunks of packed data are loaded into multimedia registers (e.g., 8 bytes), a single

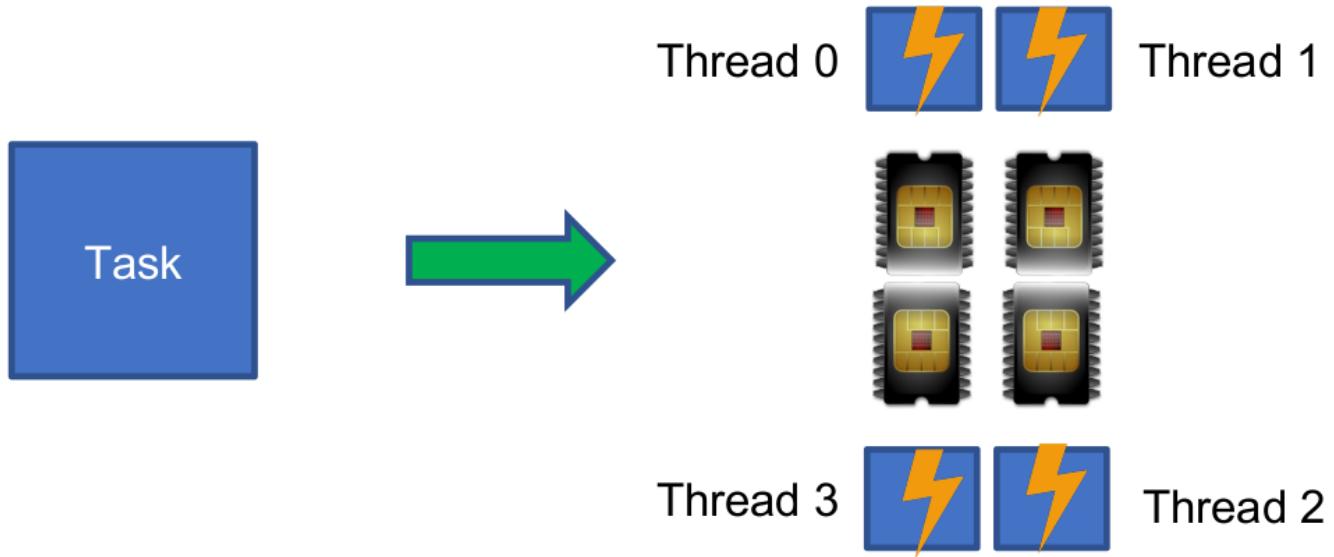
operation can be applied to all operands performing a single multimedia instruction.

MMR9 = MMR7 Op MMR3

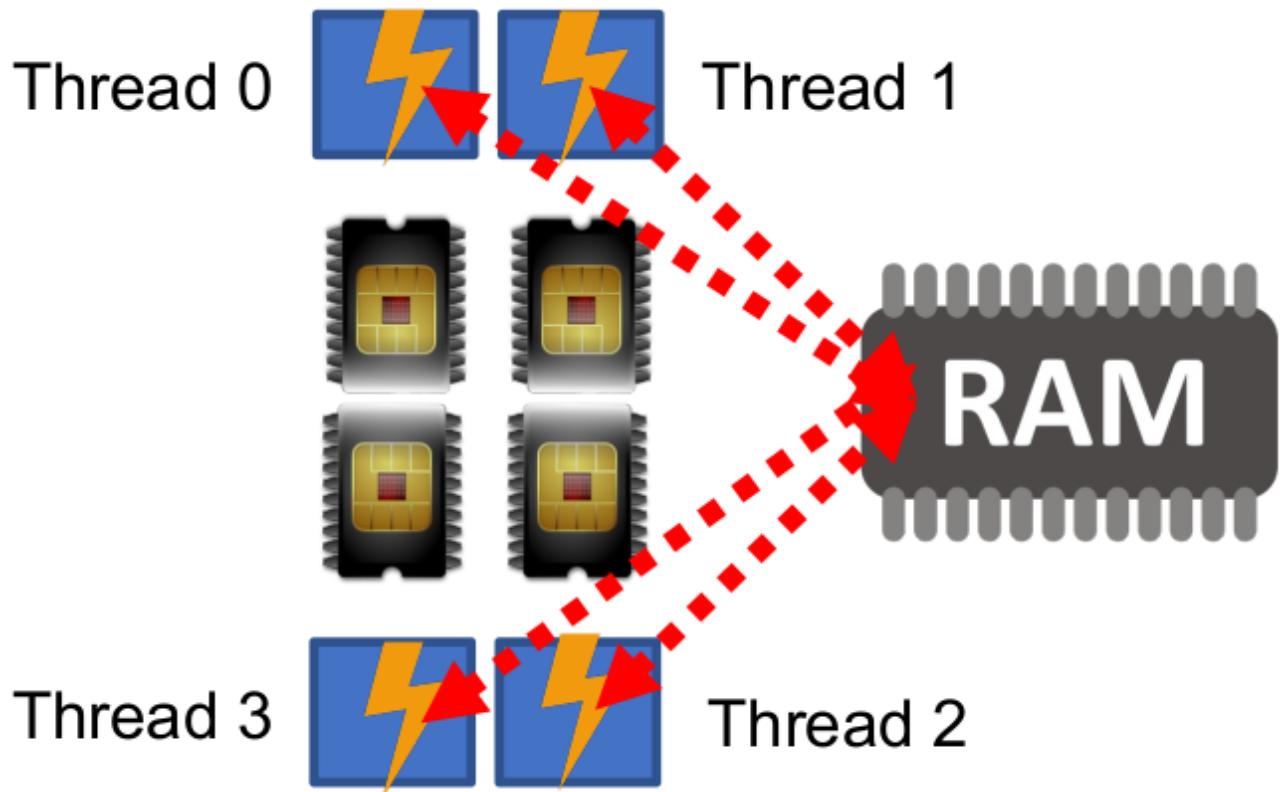


3) Distribuire il carico su più core

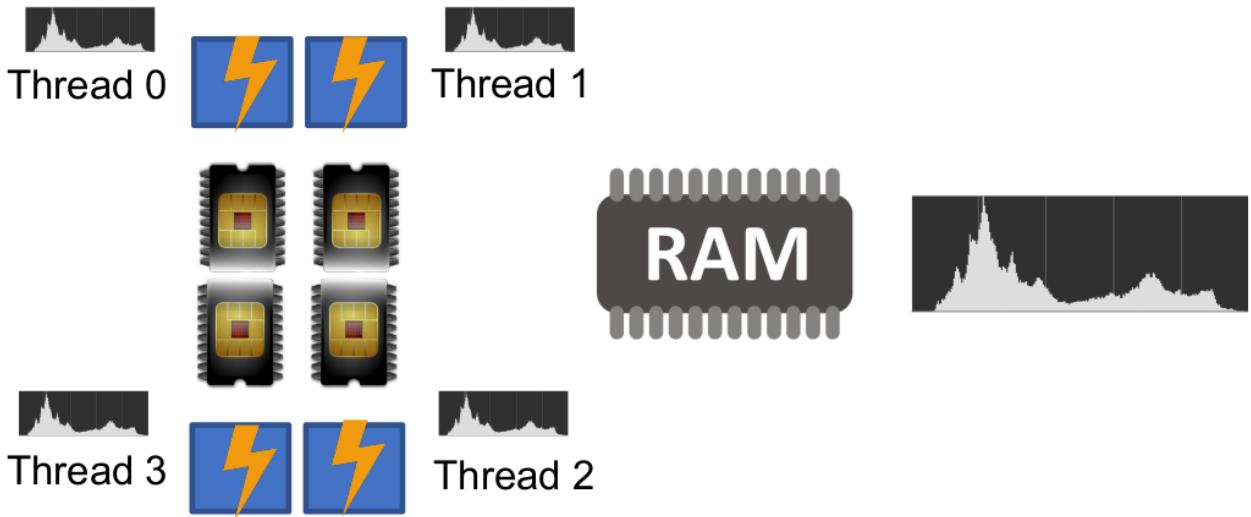
Parallelismo a livello di thread, che eseguono sottotask concorrentemente



- 😔 Bisogna gestire i casi di data dependency/race tramite
 - Mutex, canali, semafori



- Alternativamente, usando più copie private delle strutture dati per ogni thread/core. At the end of all computations, the private data structures are merged. The only synchronization concerns the end of all threads.



this might be faster since, in this specific example, data race is highly likely to happen during concurrent execution, but less elegant.

Software multi-core example with OpenMP

The previous examples pointed out the problem of data dependency, a major issue when dealing with multi-core processing and, in general, when performing parallel/concurrent computations. Given an array A, the following code computes the average value:

```
main()
{
    double sum=0.0;
    for (int i=0; i<1000;i++) sum = sum + A[i];
    printf("The average value is %5.5f\n", sum/1000.0);
}
```

In a parallel execution of the loop, the accumulated sum depends on previous results. Solutions similar to those outlined before can easily address the data dependency issue, but this is not always the case.

OpenMP is a popular library to take advantage of multi-core capabilities

It allows us to enable parallel computations using compilation directives (#pragma)

For instance, a loop can be parallelized as follows:

```
#include <omp.h>
main()
{
    double sum=0.0;
    #pragma omp for reduction(+ : sum)
    for (int i=0; i<1000;i++) sum = sum + A[i]; // execution of N threads, sum
    is a private variable
    printf("The average value is %5.5f\n", sum/1000.0); // Add N sum private
    variables at the end of the the parallel portion of the code (i.e., the
    loop)
}
```

Parallel Architectures

Parallel programming is complicated but can effectively achieve high performance and (sometimes) low power consumption.

Notable examples of devices with highly parallel capabilities are Graphics Processing Units (GPUs).

Initially deployed for computer graphics and gaming, these devices are now the backbone of AI and, in general, for HPC (High-Performance Computing).

For pure computing purposes, programming languages like CUDA (Nvidia) allow the execution of multiple (thousands) tasks concurrently on GPUs.

For high-end devices, energy is not the primary issue, but GPUs are also used in low-power devices such as smartphones/tablets/watches and embedded devices (e.g., Nvidia Jetson).

Many cores CPU

Regarding conventional CPUs, the number of cores is below 100 in most cases. However, some projects/devices push this paradigm further (with hundreds of small cores) to balance performance and power consumption.

In this case, the idea consists of performing many parallel computations with many small cores clocked at a reasonably low frequency to reduce the overall energy drained.

Of course, in all these cases, the programmers and the compilers play a crucial role

Parallel arch for low-powered devices

The need for high-performance on low-power devices emerged in these years and led to custom architectures mainly targeted to AI applications

Un esempio di architetture parallele per dispositivi low-powered sono:

- Neural Engine di Apple
- Tensor Processing Unit di Google.

In most cases, the aim is to balance power efficiency and performance, but the strategy relies on the massive deployment of parallel computations.

Parallel Custom Architectures

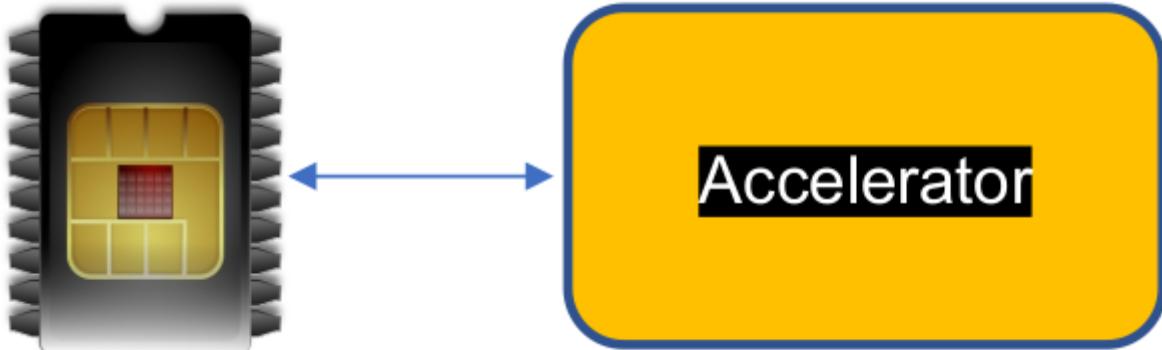
In all previously mentioned cases, the computing architecture is designed and proposed to the end user by a specific manufacturer to satisfy his/her needs.

Nonetheless, a custom architecture to achieve specific requirements (e.g., very low latency) is sometimes mandatory.

In this case, there are devices, such as modern FPGAs, that allows for tailoring the architecture to specific constraints/requirements, losing, on the other hand, the flexibility of a general-purpose approach.

💡 Le FPGA possono essere usate anche per questo scopo: creare architetture parallele che rispettino certi vincoli e requisiti, perdendo, dall'altro lato, la flessibilità di un approccio general purpose.

Per non perdere l'approccio general-purpose, parallel devices serve as accelerators invoked by one or more threads running on a conventional CPU to achieve specific tasks efficiently. This strategy allows taking advantage of high-performance computing capabilities (accelerator) with the flexibility of conventional general-purpose CPU-based systems



FPGA

Introduzione

FPGA aka Field Programmable Gate Array

- Negli anni '70 tutto era fatto con reti logiche → **bad!**
- Negli anni '80 nascono i primi dispositivi programmabili (EPROM, ...)
- Negli anni '90 nascono le FPGA → ❤️

Cosa Sono

Sono dispositivi logici programmabili (o riprogrammabili) on field (sul campo). Basso prezzo, poco tempo per essere fabbricate, supportano linguaggi di alto livello (C/C++, Python) o HDL (VHDL o Verilog). Sono infatti meno costose da progettare e da produrre su larga scala delle schede ASIC (Application Specific Integrated Circuit). Dunque, ideali per la prototipazione (progettazione CPU) e per dispositivi a basso consumo.

- **Componenti**
 - **PL** → parte di logica riprogrammabile (Programmable Logic), il cuore della scheda
 - **PS** → un processore (Processing System)
 - *Parti aggiuntive* (memorie, led, periferiche, ...)

PL può essere configurata per implementare una rete logica a piacere → **easy creation of ad-hoc HW**

- **Utilizzo**

- Per l'implementazione di **reti combinatorie** o **sincrone** (NON asincrone)
- Programmo logica FPGA
- Su PS faccio girare il mio codice (Python/C/...)
- Codice su PS *chiama* a piacere l'FPGA (PL), per *accelerare* specifiche operazioni

- Linguaggi per la **programmazione** delle FPGA

- **HDL** (*Hardware Description Language*) → low-level

Esempi: Verilog, VHDL

- **HLS** (*High Level Synthesis*) → high(er)-level

Permettono di programmare l'FPGA in linguaggi di alto livello (come il C)

- Si specifica in una funzione che implementa il comportamento che vorremmo dalla PL

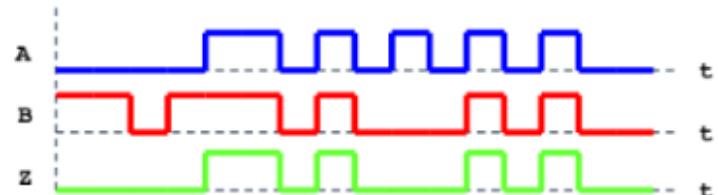
- Il codice C verrà tradotto in HDL

- L'HDL verrà tradotto in **bitstream** (che contiene la descrizione della rete)

NB: bitstream ha dimensione fissa; non varia siccome organizza un numero fisso di risorse



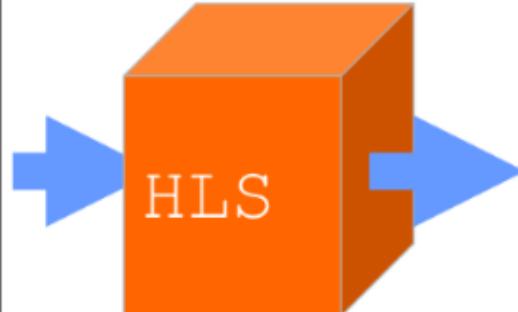
```
entity my_AND is
  Port (A : in  BIT;
        B : in  BIT;
        Z : out BIT);
  . . .
```



Each step is performed by the CPU!

Immagine 1: Sviluppo con HDL

```
#include <math.h>
int function()
{
  int i;
  int r = 0;
  for (i=0;i<100;i++)
  {
    . . . .
    . . . .
  }
  return r;
}
```



```
entity function is
  Port(x : in type;
        Y : in type
        r : out type );
  end function;

  architecture hls of
  function is
    . . . .
    . . . .
  end hls;
```

C, C++, etc



RTL (VHDL, etc)

bitstream

2: Sviluppo con HLS

Come Sono fatte

Le FPGA sono composte da **CLB** (*Configurable Logic Block*), programmabili più volte (a differenza delle ASIC, utilizzate per produzione su larga scala, programmabili 1 sola volta)

- Presenti a *migliaia* nell'FPGA
- Si possono implementare non solo reti logiche ma veri algoritmi
- I *CLB* sono *indipendenti e interconnessi* tra di loro

- *Sparsi* per l'FPGA sono presenti piccoli blocchi di **BRAM** (Block RAM) indipendenti, e unita di calcolo, di solito **moltiplicatori riconfigurabili** (operazione che si usa spesso e agisce da bottleneck)
- Nella board sono presenti periferiche di I/O.

NB: in realtà la logica di configurazione dei CLB è presente in una memoria esterna che all'avvio viene caricata sulla RAM volatile della FPGA.

Esistono molti modelli, produttori e tecnologie di FPGA. In generale, i due aspetti che le discriminano sono:

1. **Struttura** dei CLB

2. **Tecnologia** usata per le connessioni

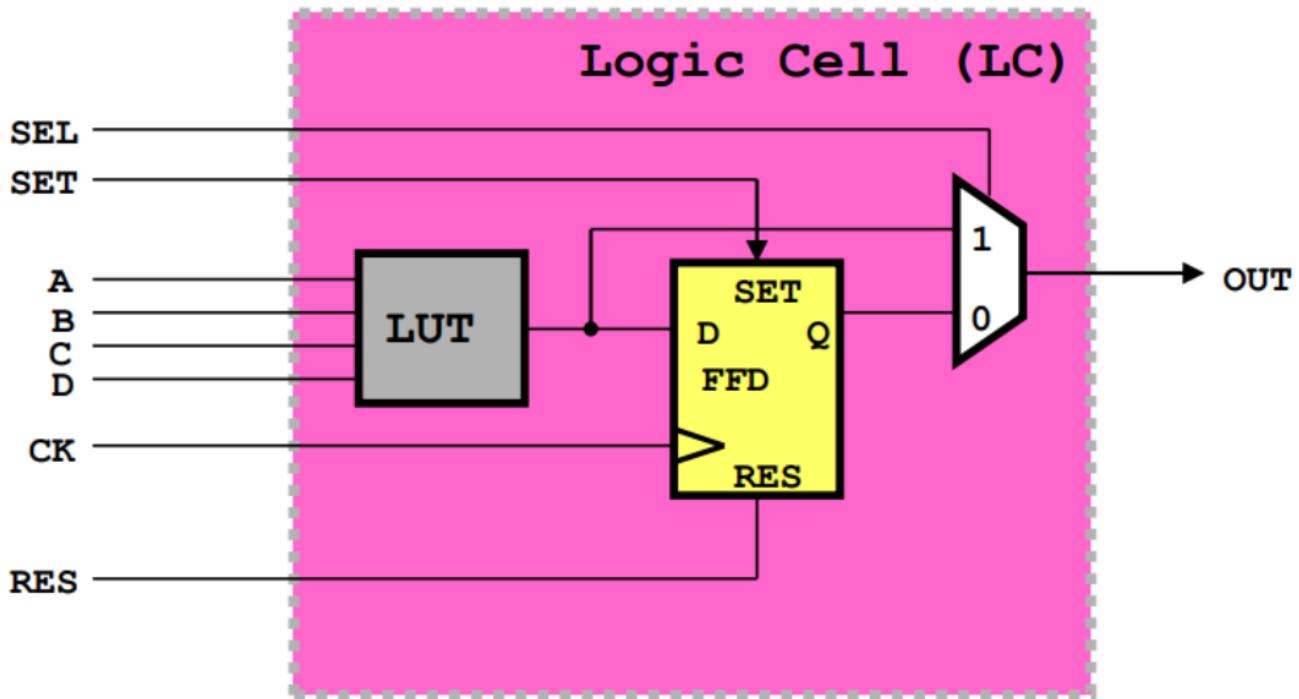
1. *Fusibili* (programmabili 1 sola volta)
2. *Flash memories*
3. *SRAM* (Static RAM) - perde memoria quando si spegne, ma in pochi ms viene ricaricato all'accensione.

Perché i fusibili al giorno d'oggi?

In alcuni contesti avere una memoria non è consigliato. Ad esempio, nel settore aerospaziale → bit flip possibili per le radiazioni nello spazio. Un altro esempio è quando non si vuole correre il rischio che il codice venga copiato illecitamente, cosa possibile se è salvato in una memoria. Infatti, non è possibile “estrarre la disposizione dei fusibili” andando a vedere connessione per connessione se il “filo” è stato interrotto o no.

Struttura CLB

L'unità elementare dei CLB è la cella logica (Logic Cell, LC).



Struttura (semplificata) di una Logic Cell di una FPGA. Si può notare come il MUX in uscita permetta di bypassare l'FFD.

Struttura Logic Cell:

- **LUT** → Look-up Table
 - Implementa funzione combinatoria programmabile
 - Utile perché ha tempo di accesso costante
 - Usabile come RAM o Shift Register
 - “EPROM il cui IN rappresenta l’address del dato da mandare in OUT”
- **FFD** → parte sequenziale sincrona
- **Mux** → per poter bypassare l'FFD (solo per reti combinatorie). Serve per decidere se utilizzare solo l'uscita della LUT o anche il Flip-Flop-D.

💡 LUT e FFD possono essere sfruttati come memoria (RAM/Shift Register), molto piccola e quindi molto veloce.

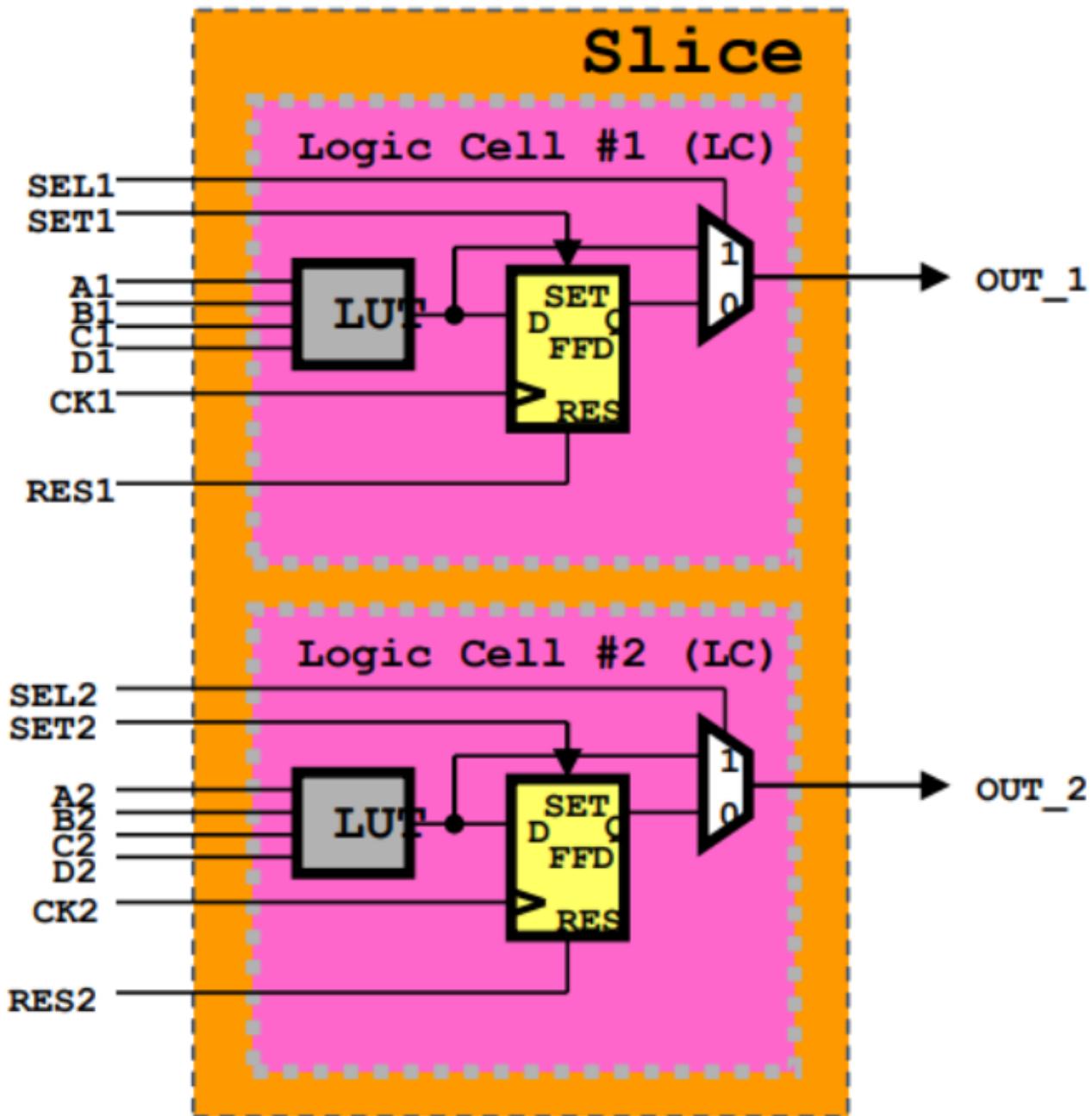
Gerarchia di Cache

Secondo quanto visto, si ha quindi la seguente gerarchia di cache:

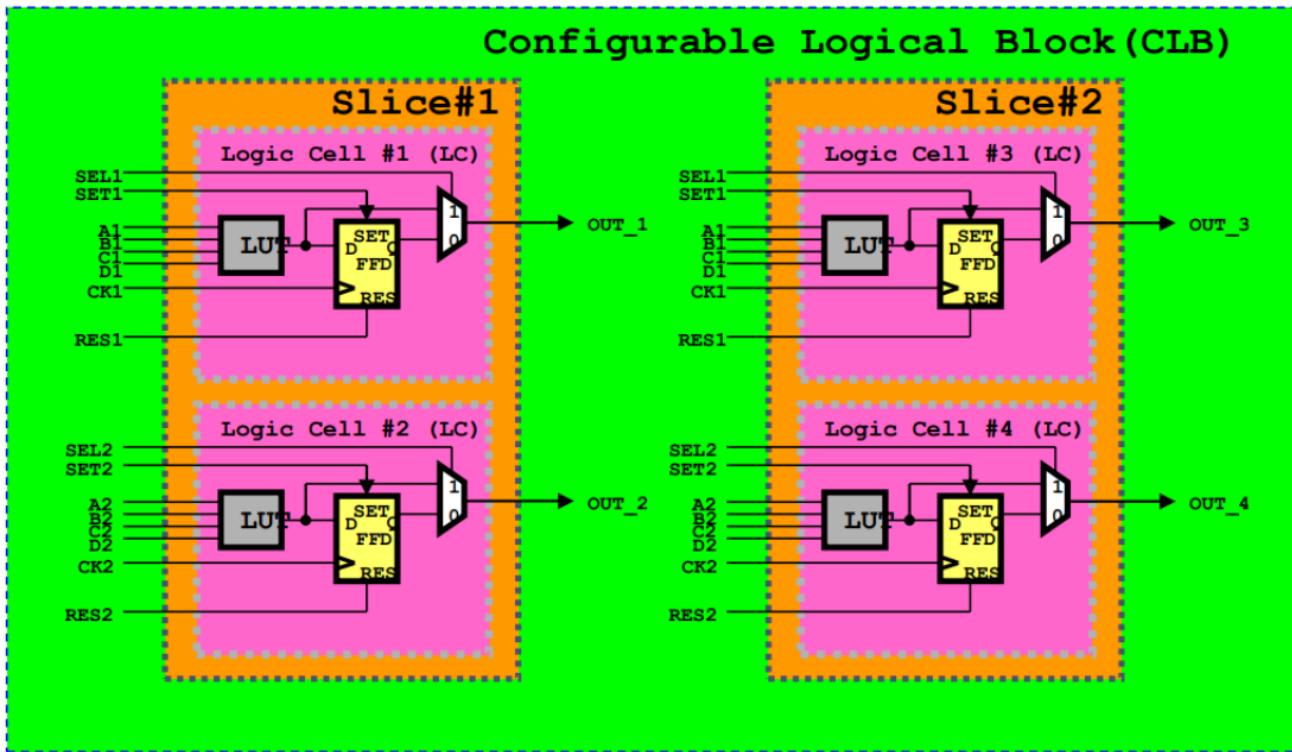
- Memoria esterna (se presente, usata solo per caricare dati nella BRAM) → più lenta, ma più grande
- Block RAM

- FFD/LUT → più veloce, ma più piccola.

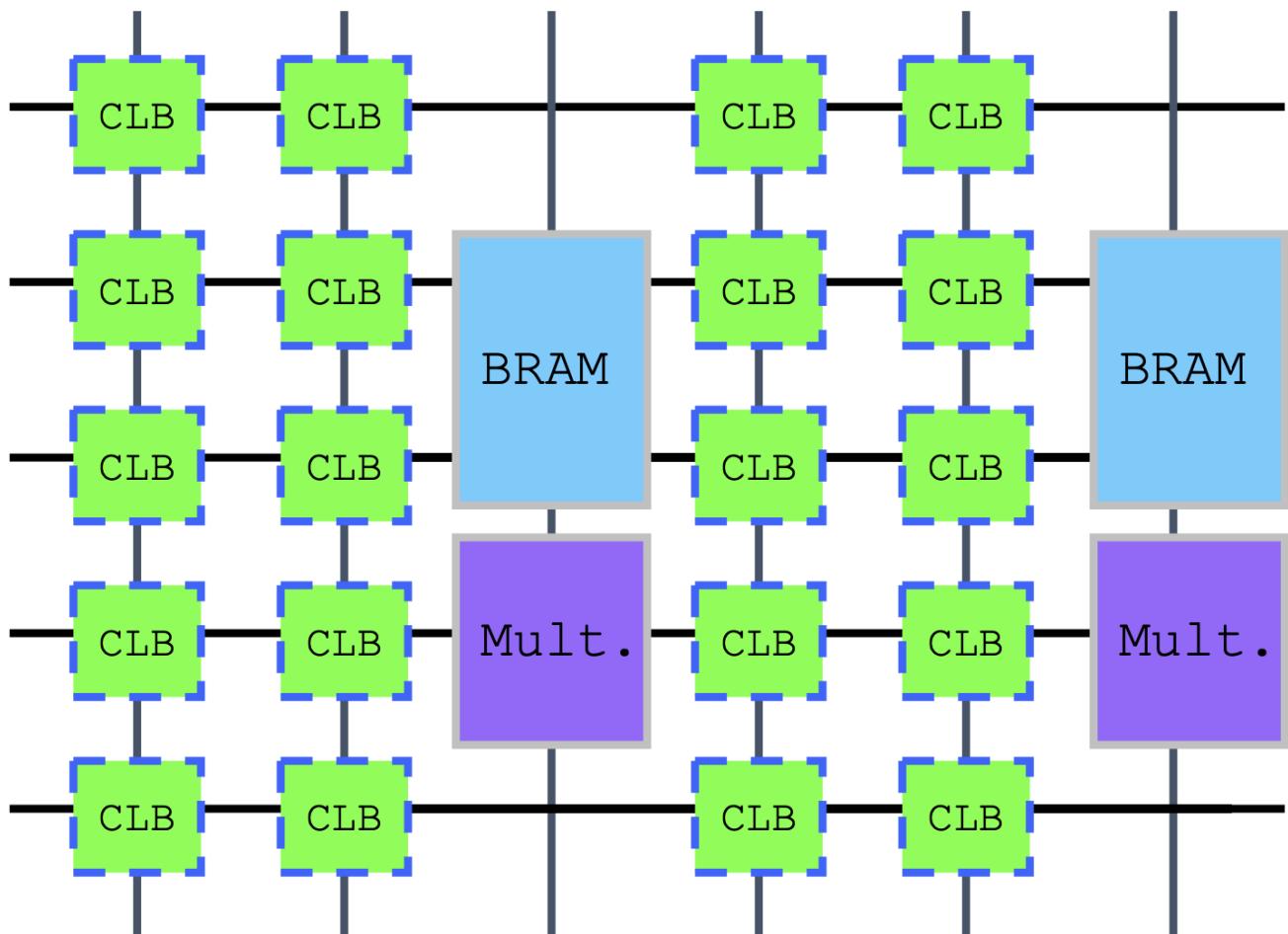
Da Logic Cell a CLB



Le Logic Cell sono raggruppate in Slice.



Più Slice sono raggruppati in un Configurable Logic Block.



Schema di una FPGA. Tanti CLB interconnessi, con BRAM, moltiplicatori e (anche se non visibili nell'immagine) blocchi di I/O sparsi.

Le celle logiche sono raggruppate in Slices, e più slices interconnesse vanno a formare un CLB. Si possono creare retroazioni tra CLB o slice ma non si possono realizzare reti asincrone, solo sincrone e sequenziali.

La Cella Logica può essere riprogrammata in vari modi (shift-register o una memoria RAM distribuita ad esempio). Il compilatore sceglie le connessioni tra CLB, il programmatore produce il codice di configurazione(bitstream) che fa in modo che il compilatore renda la rete veloce, che occupi meno CLB possibili e di massimizzare le performance.

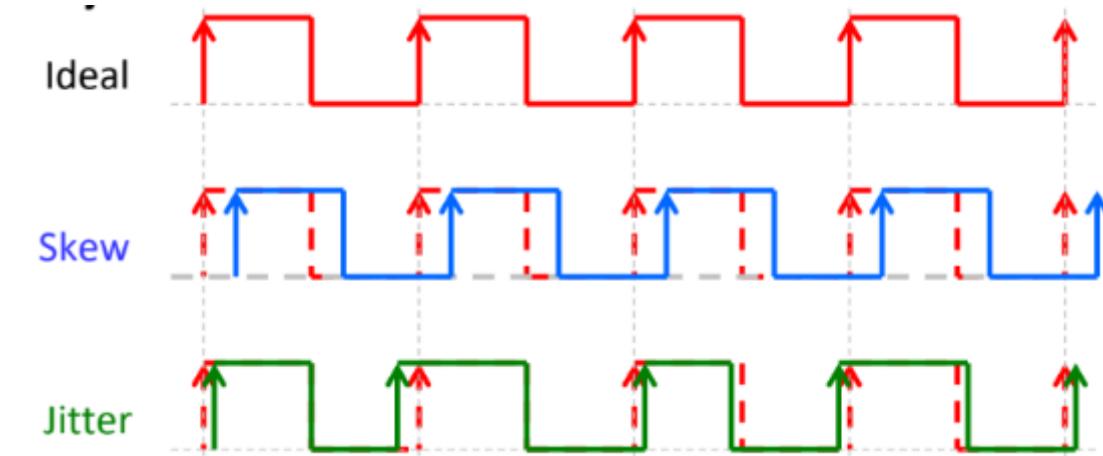
Se la FPGA è occupata all'80-90% bisogna fermarsi, altrimenti ci si impiegherà troppo tempo per realizzare il codice di configurazione oppure otterranno performance ridotte.

⚠ Attenzione, le connessioni tra i CLB sono molte, ma non infinite. Inoltre, problematiche sorgono dal fatto che diversi percorsi introducono diversi ritardi di propagazione.

Clock nelle FPGA

Ogni FPGA ha almeno 1 segnale di **clock**. In genere, vi è 1 segnale per ogni unità funzionale

- $clk_{fpga} \sim 100MHz$
- Bisogna fare attenzione al passaggio dati tra *clock domains* differenti
- **SKEW** → problema di fase (segnale anticipa/ritarda il clock)
- **JITTER** → segnale cambia frequenza dinamicamente.



Per generare clock a una data frequenza si usano **PLL** (*Phase-Locked Loop*) o **DCM** (*Digital Clock Manager*), che sono nient'altro che moltiplicatori/divisori

- Generano segnali stabili a partire da un segnale esterno periodico
- Riducono skew e jitter
- Sono esempi di **IPCORE** ("equivalenti hw di una libreria sw")

Skew e jitter non sono problemi mutualmente esclusivi.

 Nonostante la frequenza di clock non elevata, l'elevato grado di customizzazione e parallelismo ricavabile con l'utilizzo di una FPGA permette di poter competere, e spesso superare, le CPU General-Purpose nell'esecuzione di task specifici. Il tutto consumando relativamente poca energia grazie alla ridotta frequenza di funzionamento. Dunque, per poter essere competitiva, nonostante le sue basse frequenze, è necessario configurarla in modo da ottenere il parallelismo più alto

IP-Core: librerie hardware

IP-Core sta per *Intellectual Property Core*, ed è:

- l'equivalente hw di una libreria sw → implementano una certa funzionalità (memory controller, SERDES, ...) nella FPGA
- Disponibili gratis o a pagamento
 - Alcuni forniti direttamente dal produttore
 - Altri sviluppati da terzi
- Come il sw, può essere copiato (l'approccio SRAM è il più vulnerabile)

 Alcuni IP-Core di uso comune sono:

- DCM o PLL per gestire il clock
- Memory Controllers per gestire i trasferimenti coi dispositivi esterni di memoria
- SERDES per la conversione serie/parallelo di segnali ad alta intensità
- Communication controller: dispositivi per gestire i trasferimenti ad alta larghezza di banda (10/100/1000 GB Ethernet).

Bus Protocols

- Per comunicazione FPGA/esterno o FPGA/ARM CPU
- Insieme di protocolli per periferiche e memorie.

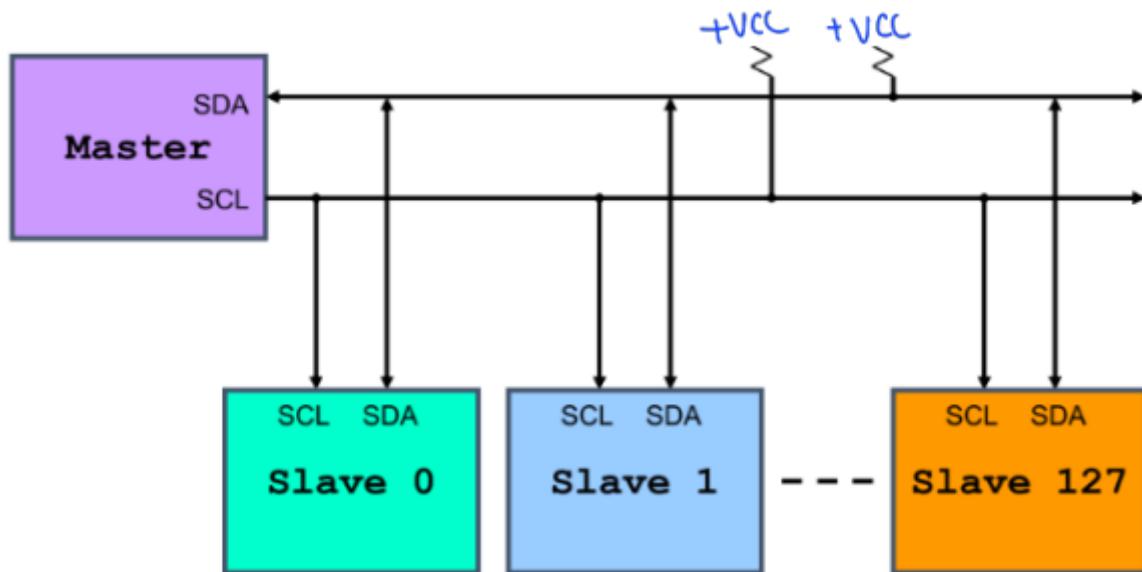
I2C

- Protocollo degli anni '80 **semplice e leggero**, motivo per cui si usa ancora
- seriale
- Master/slave
- Address a 7 bit (+ 1 bit per R/W), quindi riusciamo ad indirizzare 128 dispositivi (compreso il master)
- Usa solo 2 fili → clock (**SCL**) e dati (**SDA**)

- SPI è un protocollo simile, ma con 4 fili.
- La frequenza iniziale è di 100kHz, frequenza current standard=400kHz
 $freq \in \{100KHz, 1MHz\}$

Comunicazione

- Ogni ciclo di bus si divide in due parti
 - Address frame (master specifica l'indirizzo dello slave)
 - Data frame (scambio dati).
- Master: emette clock, emette indirizzo (address frame).
- Il master inizia sempre la comunicazione(asincrona). Ci sono al più 127 slaves (solitamente meno). In molti casi l'indirizzo è assegnato dal produttore e non può essere modificato.
- In ogni comunicazione è solo uno lo slave che si connette con il master per comunicare.



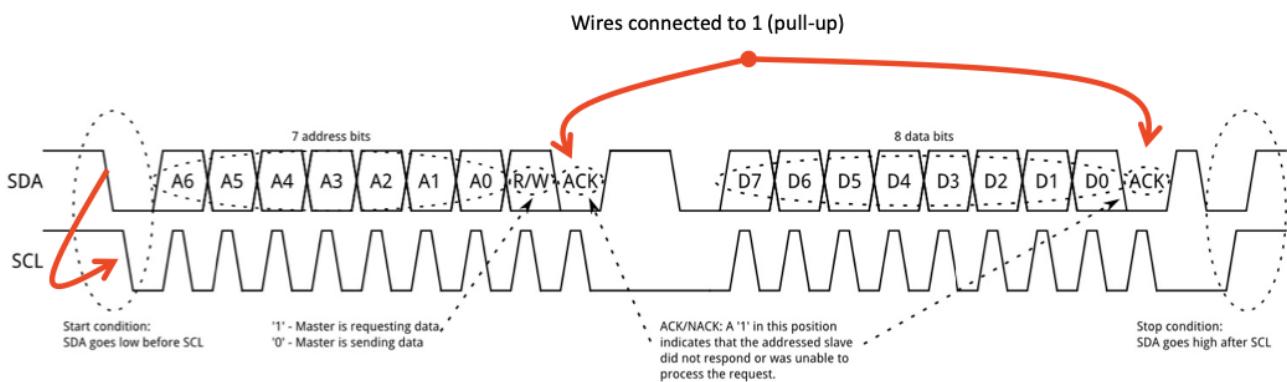
Durante ogni ciclo di bus si hanno due fasi:

1. ADDRESS FRAME:
 - Il master emette l'indirizzo dello slave con il quale intende comunicare (7 bit)
 - Il master specifica se vuole eseguire un'operazione di lettura o scrittura
2. DATA FRAME
 - Write: Il master invia i dati allo slave
 - Read: il master legge i dati dallo slave

Quando nessun ciclo è attivo i due segnali sono collegati a Vcc.

V= High -> non avviene nulla -> stato 1;

V = Low -> inizia la comunicazione -> stato 0



Ciclo di bus I2C.

1. SDA e SCL portati a VCC (1)
2. **ADDRESS FRAME:** Prima SDA, poi SCL vanno a 0 (start condition) -> inizia un nuovo ciclo di bus e gli slave si mettono in ascolto per capire quale sarà l'indirizzo con cui il master vuole comunicare.
3. Master emette address frame (indirizzo slave + R/W bit=7 bit per l'indirizzo + uno per definire l'operazione)
4. Slave risponde con ACK (stop bit), ossia porta a 0 SDA
5. SDA torna a 1, e SCL va a 0 (fine prima fase)
6. **DATA FRAME:** Master torna ad emettere il clock. I dati vengono inviati usando SCL come segnale di sincronismo
7. Scambio dati, seguito da ACK (stop bit)
8. SDA portata a 1 poi a 0, SCL a 0
9. Poi stop condition: SCL e SDA portate a 1 in quest'ordine.

OSS:

Si tratta di una specie di comunicazione asincrona con un segnale di sincronismo usato solo quando serve -> è una comunicazione seriale.

- Il clock è generato da chi invia i dati.
- I produttori devono cablare gli indirizzi degli slaves e per farlo cablano solo una parte degli indirizzi, la parte restante degli indirizzi è scelta con dei jumpers in modo da avere indirizzi modificabili in base alle specifiche esigenze. Di solito si ha un solo bit per cambiare indirizzo in modo da evitare collisioni con altre parti.

Nei componenti vivavo c'è anche il bus I2C -> modulo configurabile come axi.

Utilizzo

- Ottimo per comunicazioni di quantità modeste di dati. Ad esempio, telecamera che trasferisce dati a 10 MB/s ma che può funzionare a diverse risoluzioni, a colori o in scale di grigi: configurabile via software. Bastano pochi dati per configurarla quindi I2C è

sufficiente. Usato anche negli accelerometri giroscopici (funzioni nelle quali devono essere inviati pochi byte al secondo).

AXI

Si divide in 3 protocolli (sviluppati da ARM ❤):

1. **AXI** → complesso, ma veloce. Usato per trasmettere dati ad altissima velocità, vanno specificati gli indirizzi
2. **AXI Lite** → economico, più “lento” (no burst transfer). Utile se non mi interessa una trasmissione ad alta velocità ma il dato è stato mappato con degli indirizzi in memoria
3. **AXI Stream** → *addressless*. Usato per segnali di varia natura, in particolari immagini. Non è memory mapped, è sufficiente sapere quando inviare il flusso di dati (sincronizzazione): meccanismo di ricezione e trasmissione;

I primi due sono memory mapped, il terzo no.

AXI e AXI-Lite

Caratteristiche comuni a AXI e AXI-Lite:

- Memory mapped: dati vengono trasferiti fornendo gli indirizzi con protocolli master-slave
- Comunicazione su **canali bidirezionali** (specifici per read e write)
- **Read**
 - Read Address Channel
 - Read Data Channel
- **Write**
 - Write Address Channel
 - Write Data Channel
 - Write Response Channel
- *Configurable **data parallelism***.

AXI

Caratteristiche chiave: *master/slave*, *indirizzi a 32 bit*, **parallelismo** dati configurabile (tra 32 e 1024 bit), **burst transfer**, segnali di controllo (ACK).



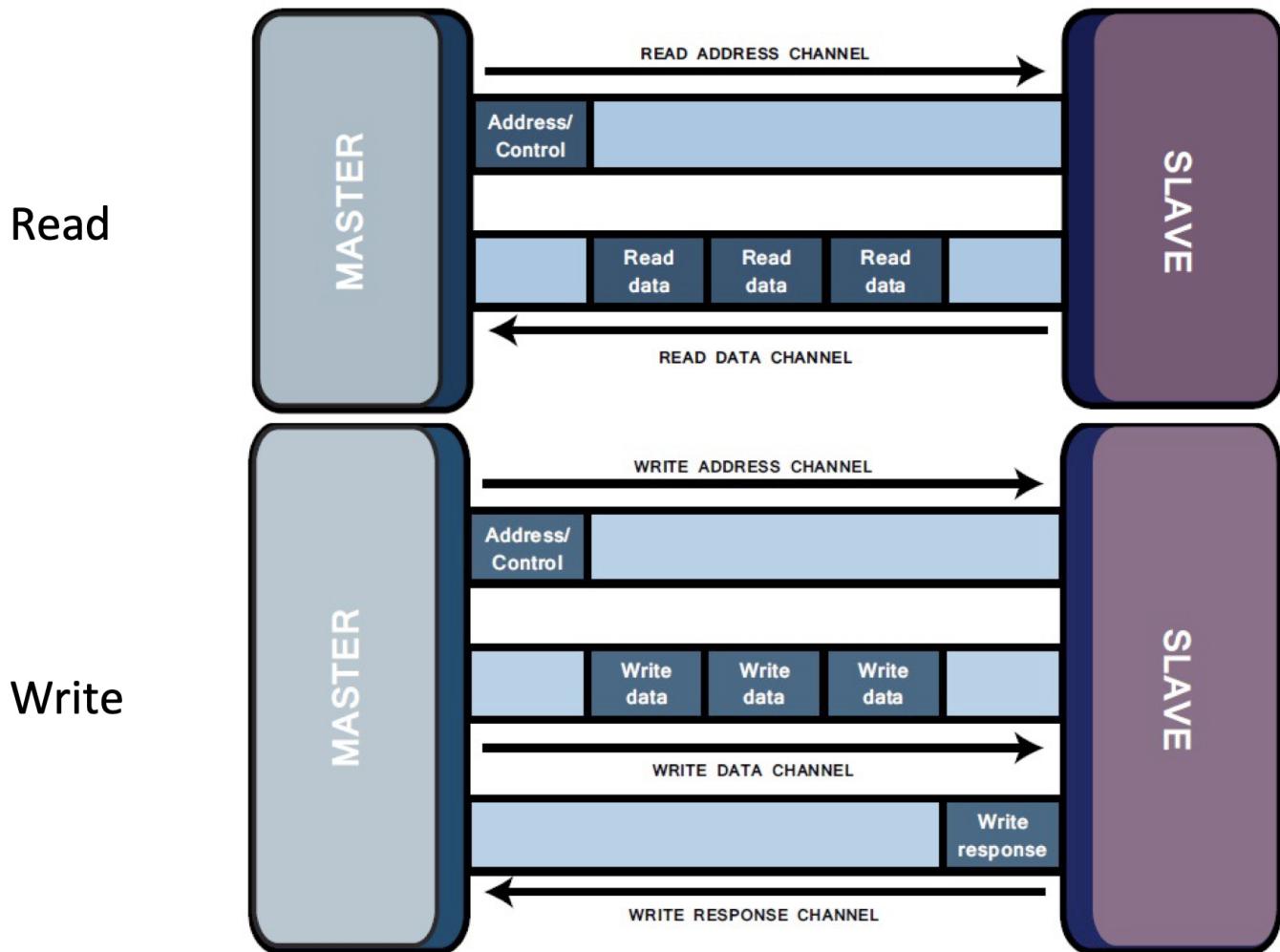
Burst Transfer

Trasferimento (R o W) in sequenza di blocchi contigui.

Fino a 256 trasferimenti in un unico ciclo di clock (o clock allungato) in indirizzi diversi (ma contigui) immettendo un solo indirizzo. Avviene sia in lettura che in scrittura e trasferisce i dati IN SEQUENZA.

Esempio: se si vuole leggere `arr[i]` di un array `arr`, anche i dati fino a `arr[i+N]` saranno letti → indirizzo di lettura emesso ogni `N` cicli .

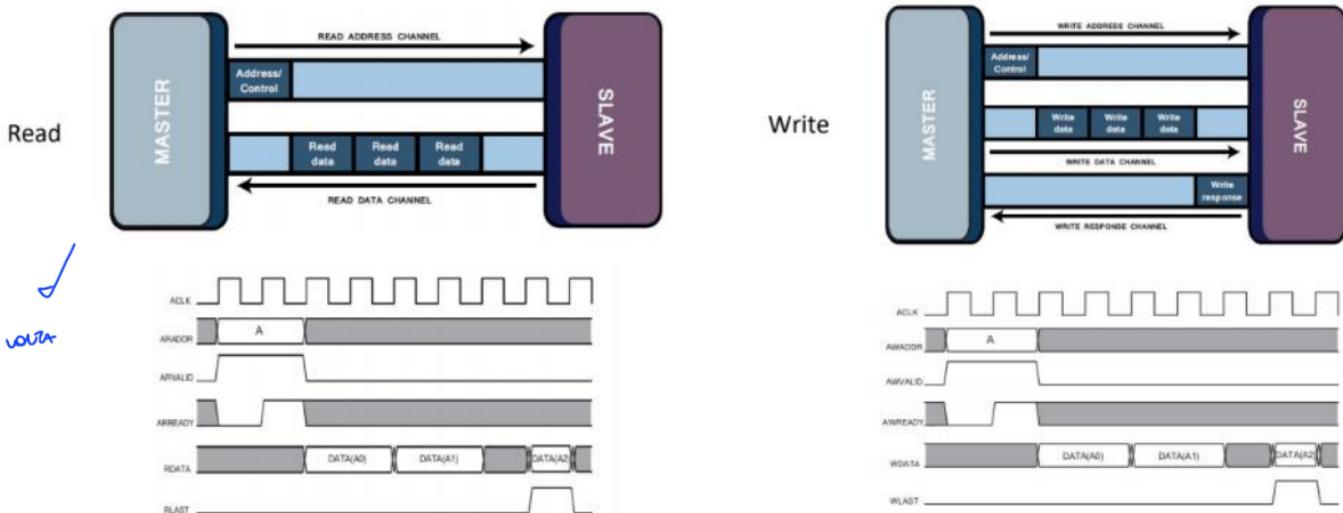
Data la possibilità di burst transfer, AXI è particolarmente indicato per le memorie, a differenza di AXI Lite, più indicato per le periferiche poiché necessita di meno logica.



Esempio di burst R/W AXI.

AXI ha indirizzi a 32 bit e parallelismo configurabile (da 32 a 1024 bit), con un impatto significativo sulle performance.

Il *clock* AXI è *settabile* dal designer del modulo.



AXI Lite

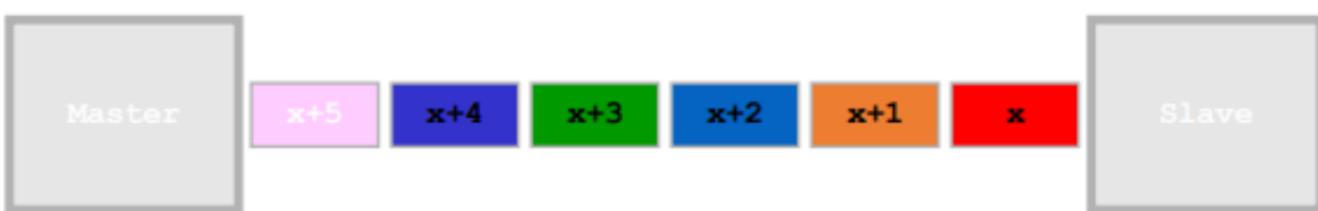
Caratteristiche: *memory-mapped, single transfer* (no burst).

NB: La comunicazione tra master e slave avviene tramite l'axi gp0, ci permette di astrarre la comunicazione a più alto livello senza conoscere tutti i dettagli elettronici dei bus.

I moduli HPIO sono in grado di trasferire grandi moli di dati ma non sono rapidissimi a partire. Se arriva un flusso di immagini, non è un problema scrivere un flusso video in memoria ma è un problema stare al passo con i dati che arrivano. Questo è dovuto alla latenza della memoria, questa

non risponde in un tempo rapido rispetto ai bus. Prima si scrivere i dati in memoria è necessario usare un buffer (FIFO) per compensare la latenza della memoria che necessita di un certo tempo per poter accedere all'indirizzo che sto richiedendo.

AXI Stream



Caratteristiche:

- **addressless** (non memory mapped): La comunicazione tra master e slave avviene senza indirizzamento, perché i dati si presentano sempre in modo sequenziali: sempre con la stessa sequenza di arrivo, perciò, non è necessaria la parte della logica degli indirizzi. Accesso randomico ai dati non è permesso.
 - **master/slave**, con sincronizzazione tra i due tramite protocollo **handshake**
 - numero di trasferimenti non limitato: possibile configurare il **parallelismo** dei dati e si può decidere di trasferire una quantità di dati ben definita (8 bit, ...).
- Addressless** → no random access (servirebbe l'address!!!1!!)

NB: Comodo per segnali audio o immagini (pacchetti dati che hanno sempre la stessa struttura). ARM non supporta l'accesso a questi dati, per farlo bisogna salvare questi dati in memoria convertendoli in un protocollo memory mapped.

Progettazione di FPGA

Progettazione con Linguaggi HDL

HDL sta per "Hardware Description Language"

- Consentono di modellare il comportamento dell'hw → rispetto ai classici linguaggi sequenziali permettono di trattare meglio parallelismo e ritardi
- Utilizzati anche per realizzare ASIC, processori, ... → portabilità ✓
- Standard IEEE.

Progettazione con Linguaggi HLS

HLS sta per "High Level Synthesis"

- Progettazione di moduli custom, ma in linguaggi di più alto livello (es. C/C++) rispetto ai linguaggi HDL
- Più "accessibile" per software engineer con limitate conoscenze di "basso livello". Si scrive un codice C, formato da un header file: parametri e altre cose da includere, e un file di codice vero e proprio dove si descrive ciò che la rete/algoritmo deve fare. Estensione .cpp, che consentono di definire tipi di dati custom con dimensione qualsiasi. `Ap_tipodato.h` -> libreria che consente di definire qualsiasi tipo di dati con una dimensione scelta dall'utente. In questo modo, mediante "`ap_int.h`", si possono definire tipi di dati con dimensione variabile. È possibile definire tipi signed e unsigned:
 - `ap_uint<5>` unsigned a 5 bit
 - `ap_int<5>` signed a 5 bit

OSS: in questo caso al posto di allocare 8 bit per poi buttare via 3 bit, posso utilizzare direttamente 5 bit

Funzione top function -> funzione che si basa sul clock, e che può chiamare/produrre altre funzioni. Nel caso in slide è una funzione void (che ha un led output). Vivado capisce se è un segnale di input o di output, in base alle operazioni che andiamo a fare con questa funzione (solo lettura, solo scrittura o entrambe) e definisce l'interfaccia ai morsetti del modulo che si sta progettando.

NB: noi nel codice non definiamo il clock -> lo capisce da solo il compilatore.

Volatile -> è una direttiva di compilazione. Indica se una variabile sia o meno volatile, cioè serve per evitare che il compilatore faccia ottimizzazioni sulla variabile I/O, a più basso

livello, scrivendo 2 volte sulla variabile si vedranno le due scritture. Lo scopo del codice è la descrizione, non l'ottimizzazione del codice.

Esempio: se assegno ad una variabile un valore, allora il compilatore elimina la 1° assegnazione. Poi in un sistema HLS io devo controllare clock a clock che succede.

Il C non nasce per descrivere l'hardware. Con HLS descrivo il funzionamento dell'hardware con un linguaggio noto ma che non nasce per questa funzionalità, vanno quindi introdotte delle modifiche al linguaggio.

La funzione descrive al compilatore come generare la rete logica (sequenziale) e per fargli capire che cosa deve fare. L'obiettivo è di scrivere un codice che, trasferito all'interno dell'FPGA, sia il più veloce possibile: l'operazione deve terminare in un solo clock.

Testbench

Un main che serve per verificare, con delle printf o con le uscite della rete, se una certa funzione fa quello che si vuole. Ad ogni iterazione che fa la funzione viene richiamata la funzione e vediamo il valore che deve restituire a video. Si può verificare che la funzione faccia ciò che si vuole con un

valore di ritorno noto, cioè usando la return 0 (significa che la funzione ha fatto ciò che si voleva), anziché con valori stampati a video. Verifica che la logica sia corretta e tutti i segnali hanno un'evoluzione temporale ad ogni clock. Possiamo anche stimare ogni quanto tempo il modulo, realizzato in hardware, produce il risultato. È un file che si associa alla funzione che si va a realizzare. Si usa anche per generare forme d'onda.

Co-simulazione e forme d'onda

Dal testbench in C/C++ e dall'output della sintesi è possibile esaminare le forme d'onda.

Questo è utile per capire cosa fa la rete realizzata osservando l'evoluzione temporale dei segnali sia interni che esterni. Questo si fa lanciando la co-simulazione (selezionando VHDL e tutte le tracce),

questa produce file che possono essere visualizzati come forme d'onda. È utile per comprendere in modo più fine l'evoluzione della rete nel tempo. Per i nostri scopi non è essenziale. Se rispetta latency e interval siamo abbastanza certi che una volta importata in vivado funzionerà abbastanza

tranquillamente.

Codice RTL: come un codice HDL che descrive il funzionamento della rete nel tempo con una serie di assegnamenti. Premo export RTL e trasferisco in vivado il codice HLS.

Mettiamo un range inferiore (ma non sotto i 10/20MHz come frequenza di clock).

Interfacce per porte

Vivado mette a disposizione diversi protocolli per le interfacce:

- `ap_none` – è il tipo di protocollo più semplice, non ha nessuna interfaccia di protocollo esplicito, nessun segnale di controllo addizionale, e nessun overhead hardware associato.

Tuttavia, implica che il timing delle operazioni di input e output è indipendente e correttamente gestito

- **ap_stable** — Simile ad ap_none, non richiede segnali di controllo aggiuntivi o hardware apposito. La differenza è che ap_stable è inteso solo per input che non cambiano frequentemente che sono generalmente stabili (tranne a causa dei reset, come dati di configurazione). Gli input non sono costanti ma nemmeno richiedono di essere registrati.
- **ap_hs** — protocollo di ‘handshaking’, è un superset di `ap_ack`, `ap_vld`, `ap_ovld`. Usato sia per le porte di input che di output, e mette in atto un processo di handshaking tra il produttore e l'utilizzatore di un dato; include anche le transizioni di validazione del dato e acknowledge. Per questo, richiede 2 porte di controllo e un overhead associato. È, comunque, un modo robusto di passare dati, senza la necessità di usare un timing esterno. Semplice, richiede pochi fili per la comunicazione, quindi è facile, veloce e affidabile. Non richiede overhead nella comunicazione. Utile per sincronizzare due moduli.
 - `ap_ack` — Con questo protocollo si ha un comportamento differente per input e output: Per gli input, viene aggiunto un segnale di output acknowledge che viene settato alto nello stesso ciclo di clock in cui l'input è letto. Per gli output, è aggiunto un segnale di input acknowledge. Dopo ogni scrittura dalla porta di output, il progetto deve attendere che l'ACK di input deve essere TRUE (alto, 1) prima di poter proseguire l'esecuzione.
 - `ap_vld` — Una porta addizionale è usata per validare un dato. Per le porte di input, è aggiunto un segnale di input che qualifica i dati di input come validi; per porte di output, è aggiunto un segnale di output che diventa TRUE nei clock cycles dove i dati di output sono validi. Si tratta di un segnale qualificante che definisce la validità di un segnale istante per istante
 - `ap_ovld` — Come `ap_vld` ma implementato solo nelle porte di output, o nei segnali di output di una porta bidirezionale.
- **ap_memory** — Permette di comunicare con le memorie, memory-based, supporta transazioni ad accesso random con la memoria, e può essere usato sia per porte di input, output e bidirezionali. L'unico tipo compatibile con questo protocollo sono gli array, che corrispondono alla struttura di una memoria. Richiede segnali di controllo per abilitare il clock e la scrittura, nonché una porta di indirizzo. Più oneroso da implementare e costoso da un punto di vista delle risorse, ma necessario in alcuni casi.
 - `bram` - La stessa cosa di `ap_memory` con l'eccezione che quando si utilizza IP Integrator, le porte non vengono mostrate come singole porte, ma raggruppate in un'unica porta. Utili per comunicare con le memorie interne dell'FPGA (Bram).
- **ap_fifo** - Il protocollo FIFO è compatibile con gli array, garantisce che siano accessibili in sequenza piuttosto che in ordine casuale. Non richiede alcuna informazione di indirizzo, e quindi è più semplice da implementare rispetto all'`ap_memory`. È usato sia per porte di

input che di output, ma non per quelle bidirezionali. Le porte di controllo associate indicano se la coda della FIFO è piena o vuota, a seconda della direzione della porta, e assicurano che l'elaborazione sia "in stallo" per evitare il superamento o il sotto funzionamento.

- **ap_bus** - Il protocollo ap_bus è un'interfaccia bus generica che non è legata a uno standard bus specifico, e può essere utilizzata per comunicare con bus bridge, che può quindi arbitrare con un bus di sistema. Supporta singole operazioni di lettura, singole operazioni di scrittura, e burst transfer, coordinati tramite una serie di segnali di controllo. Oltre a questa interfaccia bus generica, il supporto specifico per le interfacce bus AXI può essere integrato in una fase successiva, utilizzando una direttiva di sintesi delle interfacce:
 - **m_axi** – Interfaccia per protocollo AXI master
 - **s_axilite** – interfaccia per AXI Slave Lite
 - **axis** – interfaccia per AXI stream

Argument Type	Variable			Pointer Variable			Array			Reference Variable		
	Pass-by- value			Pass-by-reference			Pass-by-reference			Pass-by-reference		
Interface Type	I	IO	O	I	IO	O	I	IO	O	I	IO	O
ap_none	D			D						D		
ap_stable												
ap_ack												
ap_vld						D						D
ap_ovld					D							D
ap_hs												
ap_memory							D	D	D			
ap_fifo												
ap_bus												
ap_ctrl_none												
ap_ctrl_hs			D									

Block level protocols can be applied to the return port - but the port can be omitted and just the function name specified

Key:
 I : input
 IO : inout
 O : output
 D : Default Interface

Supported Interface
Unsupported Interface

Non sempre per qualsiasi tipo di variabile si possono usare tutti i protocolli senza distinzione.

Direttive e ottimizzazione rete

Il modo in cui vivado inserisce le direttive di comunicazione è con il comando `#pragma`.

`pragma HLS INTERFACE ap_ctrl_hs port=return` -> Con questa direttiva (specificata alla TOP function) eliminiamo i segnali di accesso al modulo esterno, altrimenti, la top function, di default, incapsula i segnali per controllare il modulo esterno (per resettarlo `ap_RST`, bloccarlo `ap_start`, monitorarlo `ap_done`, `ap_ready`, `ap_idle`).

- **LATENCY**: È il numero di cicli di clock necessari al modulo per generare il risultato a fronte di un nuovo input;
- **INTERVAL/ THROUGHPUT** (II or Initiation Interval): numero di cicli di clock necessari prima di poter elaborare un nuovo dato: quante volte possiamo richiamare il modulo nel

codice.

Latency e throughput/interval non sono la stessa cosa. Si pensi al DLX: nel caso del DLX sequenziale Latency e Interval sono coincidenti mentre nel caso del DLX pipelined la Latency è 5 clock (senza stalli) mentre l'Interval è pari a 1. Simili metodologie si applicano per rendere i moduli HLS più efficienti mediante delle opportune direttive.

La simulazione non ci dice quanto tempo impiega il codice né quante risorse FPGA utilizza per eseguire tale conversione.

Ottimizzazione della rete in Vivado HLS

Come migliorare le prestazioni della rete logica risultante dalla sintesi tramite semplici modifiche al codice. L'operazione di modulo è un'operazione complessa e fa sì che il sintetizzatore debba allocare una rete per far fronte alle velocità del clock. Per velocizzare tutto, si può togliere l'op di modulo e

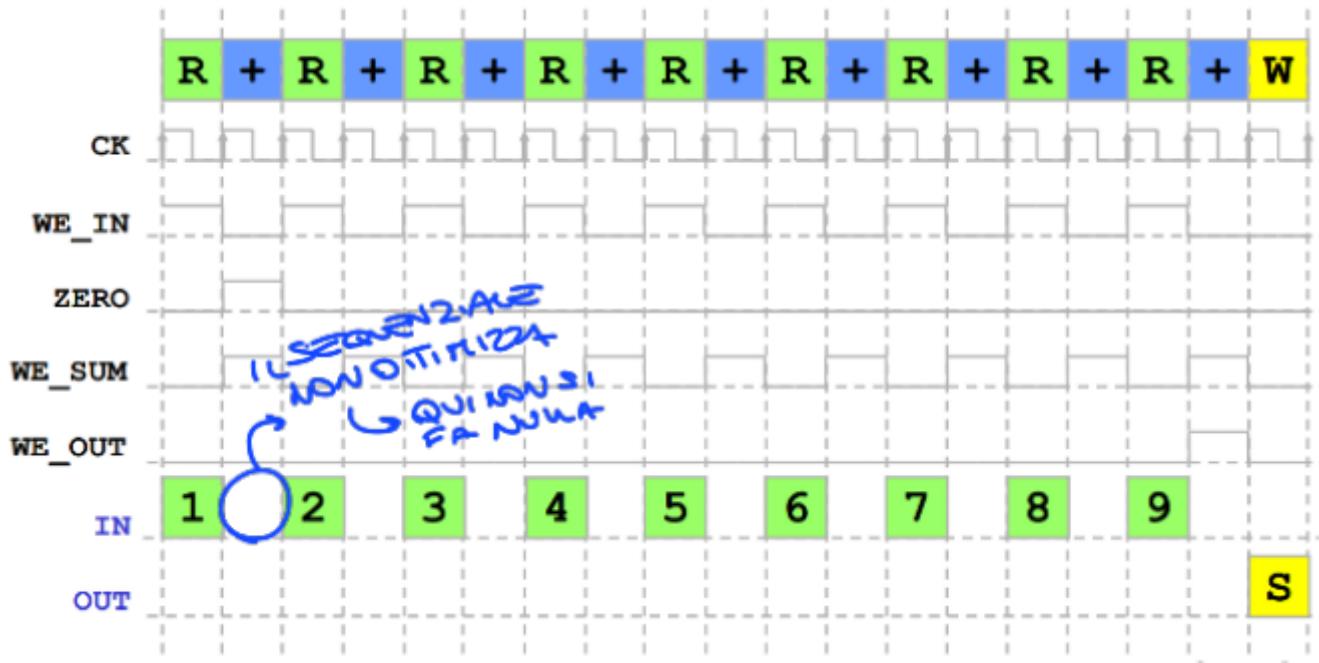
fare un'op di confronto che è più semplice da eseguire e più veloce (ha una latenza inferiore).

Richiamo il contatore ad ogni clock, inoltre la frequenza di clock incrementa. Dal target del clock di almeno 10ns possiamo raggiungerne una più elevata (6,53ns). Si tratta dello stesso codice, cambiato di una riga -> il modo in cui si scrive il codice ha un impatto cruciale sulla rete finale, in termini di

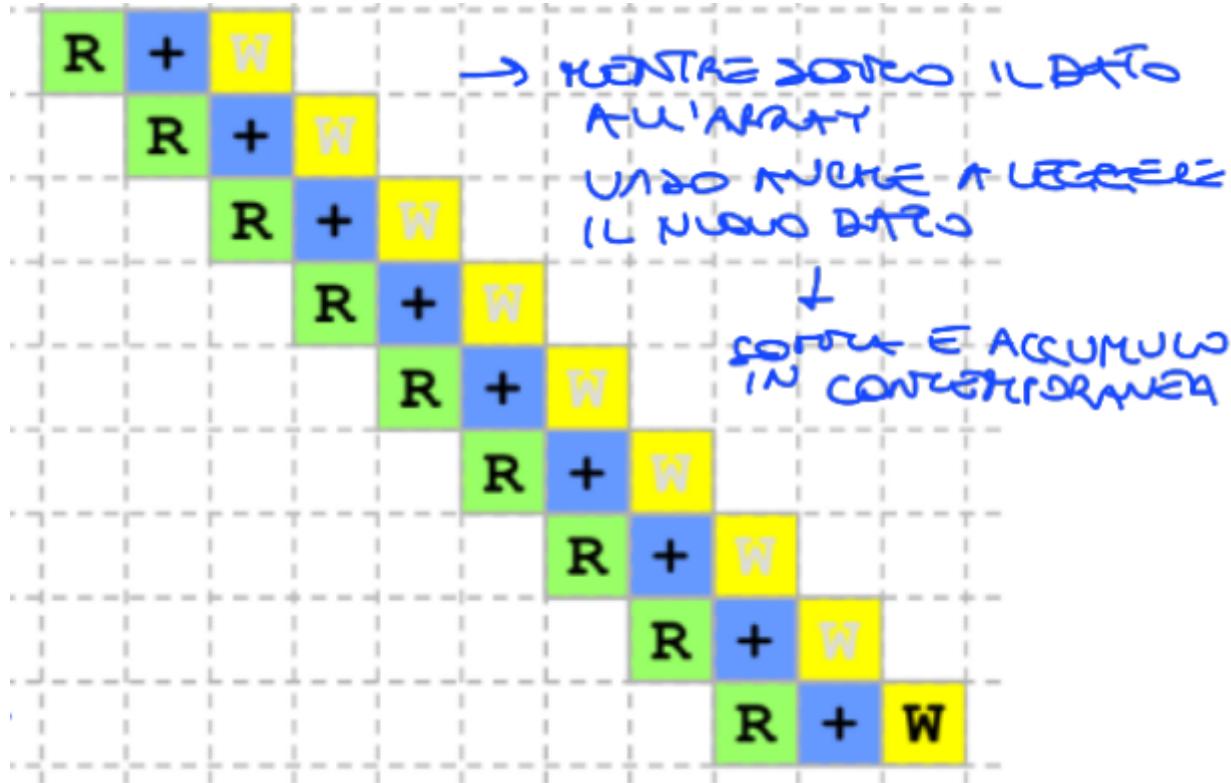
capacità di elaborazione della rete e di risorse utilizzate.

In Vivado l'opzione analisi ci aiuta da questo punto di vista. Sull'asse x c'è il clock. L'analisi ci fornisce la visuale temporale del tempo impiegato dell'operazione all'interno del codice.

Ci sono anche altre direttive (oltre la `#pragma`) in c per poter scrivere questo tipo di codice (direttive di compilazione) per renderla il più possibile ottimizzata.



1. **PIPELINING** -> solitamente, permette di ridurre la Latenza con un overhead non particolarmente elevato in termini di risorse utilizzate. Consiste nel dividere in loop in vari stadi che possono essere eseguito in parallelo. Non bisogna modificare il codice, è sufficiente inserire il comando `#pragma HLS PIPELINE II=1` (voglio porre l'interval ad 1, e quindi essere in grado di ricevere 1 nuovo dato di input ad ogni clock). Non modifica i LUT o FF o la struttura dell'FPGA.



Nell'esempio viene mostrata una somma di array di 9 elementi--> 9 cicli di lettura + 1 ciclo finale di scrittura. Il risultato della somma è disponibile sull'uscita dopo 19 cicli (latency=19). Le iterazioni del loop sono 9 (tripcount=9) e ciascuna impiega 2 cicli di clock (1 per leggere, 1 per eseguire la somma)

OSS: è buona prassi mettere un tag "loop:" che indica il loop che si vuole mettere in pipeline.

OSS: se ci sono più loop innestati allora il pipelining va applicato a loop più interno, altrimenti il programma esplode.

2. **LOOP UNROLLING** -> non è necessaria alcuna modifica al codice, è sufficiente una

```
#pragma : #pragma HLS UNROLL factor=8 <= Descrizione di una somma di un array di 9 elementi con Loop Unrolling (ci impiega 4 clock e serve replicare 4 sommatori). In questo caso factor=8 perchè si parla di un array di 8 elementi, ma bisognerebbe partizionare l'array su varie zone di memoria: #pragma HLS ARRAY_PARTITION variable=input_array complete dim=1
```

Il programma deve supportare l'accesso multiplo ai dati: deve accedere contemporaneamente a più elementi dell'array. Costringe ad utilizzare più risorse o componenti con più uscite-ingressi. Infatti, in questo esempio, i 4 sommatori ci saranno

sempre: posso sì fare 4 operazioni contemporaneamente rispetto al pipelining vs mi servono 4 volte le risorse (in vivado si vede dal numero di FF e LUT).

Fattore unroll: Operazioni da eseguire in contemporanea per clock. Il loop unrolling è la forma più aggressiva del pipelining.



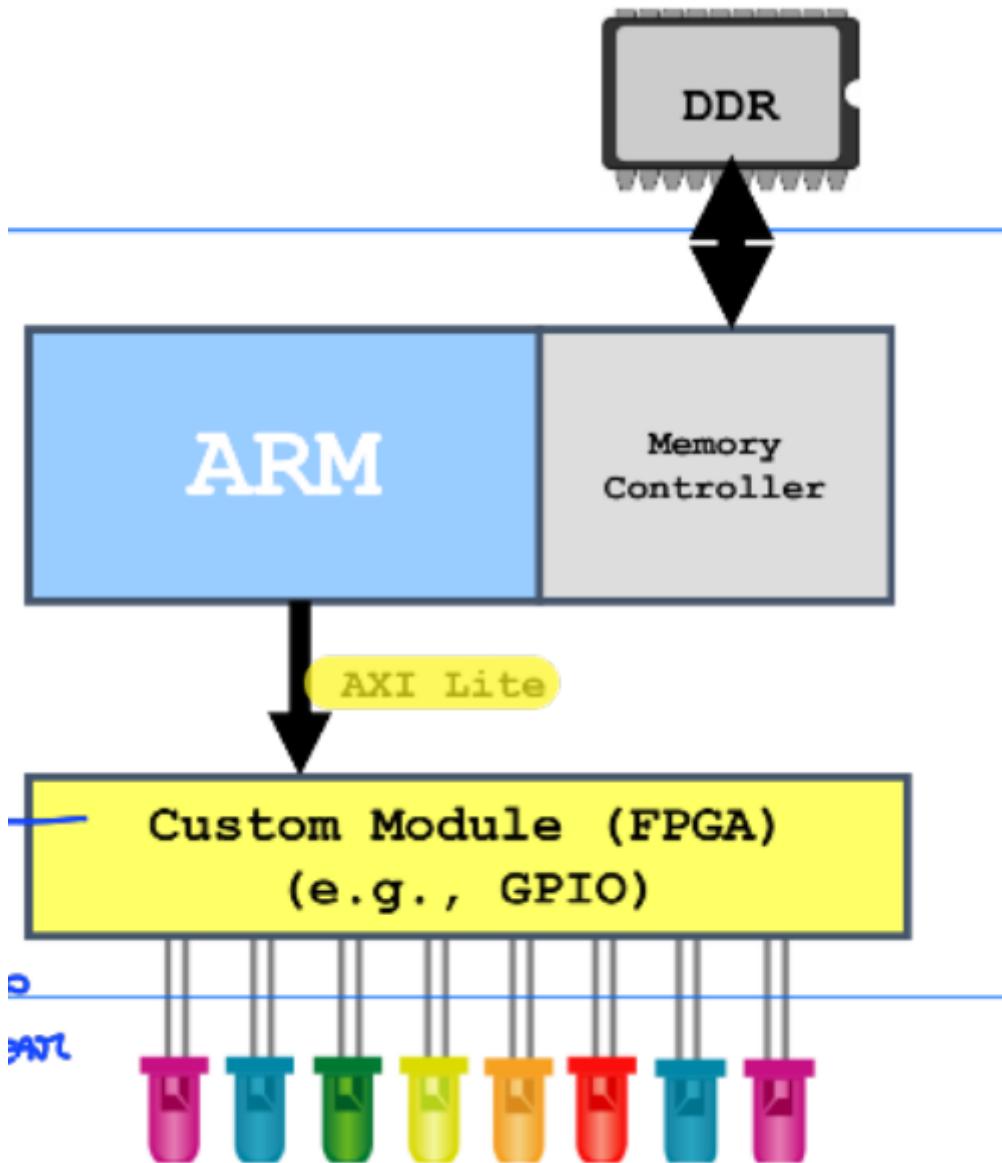
Progetti vari

Fasi sviluppo di un progetto su Zynq:

1. Creazione del dispositivo (C/C++) e validazione Vivado HLS
2. Progettazione grafica in Vivado
3. Sviluppo Software C/C++ per specifici OS (Vivado SDK)

Accensione di un led

GPIO: Dispositivo di input nella forma di un IP Core (sempre un registro di output)



Spazio degli indirizzi ZYNQ:

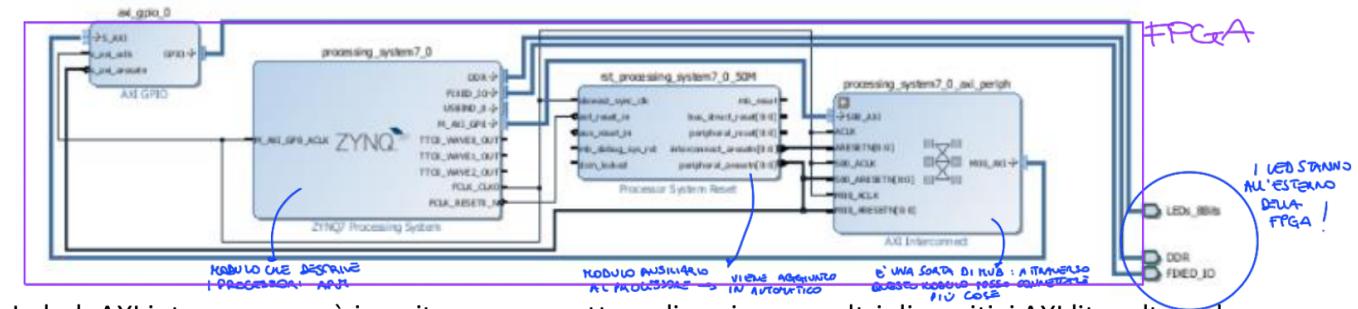
- Memoria (Esterna) e spazio degli indirizzi (32 bit) sono condivisi tra:
 - ARM Cortex A9_0
 - ARM Cortex A9_1
 - FPGA
 Dato che la memoria è condivisa, ci possono essere corse critiche --> a livello HW sono risolte perchè c'è un unico memory controller
- Controllore singolo per la memoria (dentro il PS) e singolo spazio di indirizzamento per memoria e dispositivi (entrambi ARM e FPGA), che sono memory-mapped nel range: 0x00000000 -> 0xFFFFFFFF. Il fatto che ci sia un unico spazio di indirizzamento perchè cos' ARM e FPGA possono comunicare semplicemente scrivendo in memoria. OSS: i vari pin sono con indirizzi cablati dal produttore.
- La logica programmabile (PL) (FPGA) è connessa al DDR memory controller attraverso porte High Performance HP 0, 1, 2, 3. Sono porte con elevato rate di trasferimento (GB/s),

e permettono sia la lettura che la scrittura.

NB: il GPIO è dentro l'FPGA, ma i led sono fuori da essa -> bisogna creare collegamento tra i pin FPGA e pin esterno -> si fa in vivado.

Progetto Vivado

Il progetto Vivado, controlla gli 8 LEDs attraverso un'interfaccia IP core (GPIO) AXI lite:



Un hub AXI interconnesso è inserito per permettere di aggiungere altri dispositivi AXI lite, oltre ad un modulo di reset controller per il reset del sistema (Processor System Reset).

Progetto HLS di un modulo configurabile mediante protocollo AXI lite

Mediante direttive (Vivado HLS) è possibile definire:

- Protocollo del modulo/blocco (ap_start, etc)
- Interfaccia per ciascuna porta di I/O
 - a. ap_none per il range_counter
 - b. ap_none per l'uscita led_output
 - c. ap_none per l'uscita output_value
- Risorse
 - d. AXI lite per il range

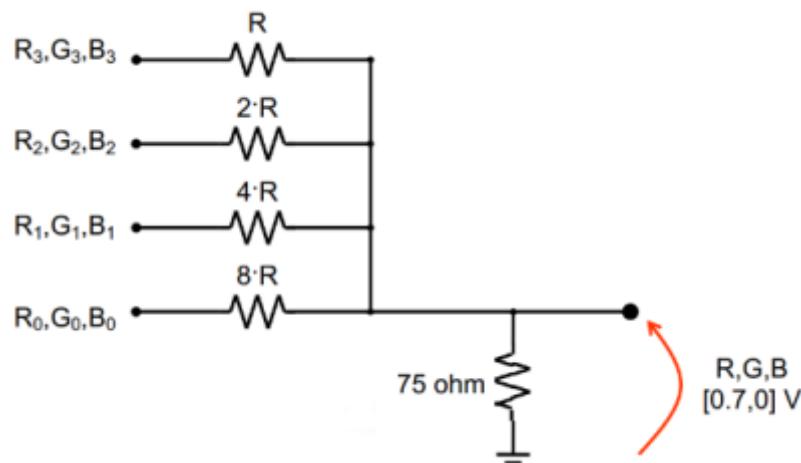
Progetto HLS di un'interfaccia VGA

La VGA è una semplice interfaccia che consente di pilotare un monitor analogico utilizzando dei segnali digitali. Sono supportate varie risoluzioni e frame rate.

5 segnali:

- 3 componenti di colore R, G, B
- 2 segnali di sincronismo: V_SYNC (verticale) e H_SYNC (orizzontale). Indicano quando comincia e finisce una riga/colonna (dimensioni immagine a schermo). Vanno da 0 a 1, in (0,0) comincia una nuova immagine (punto in alto a sinistra nello schermo)
I segnali digitali di RED, GREEN e BLUE a n bit sono convertiti in tre segnali analogici tra 0 e 0.7 V mediante un DAC (l'interfaccia è analogica ma il segnale di partenza lo creiamo digitale). Nel caso della zedboard, n=4 e il DAC è costituito da una serie di resistenze (4 bit)

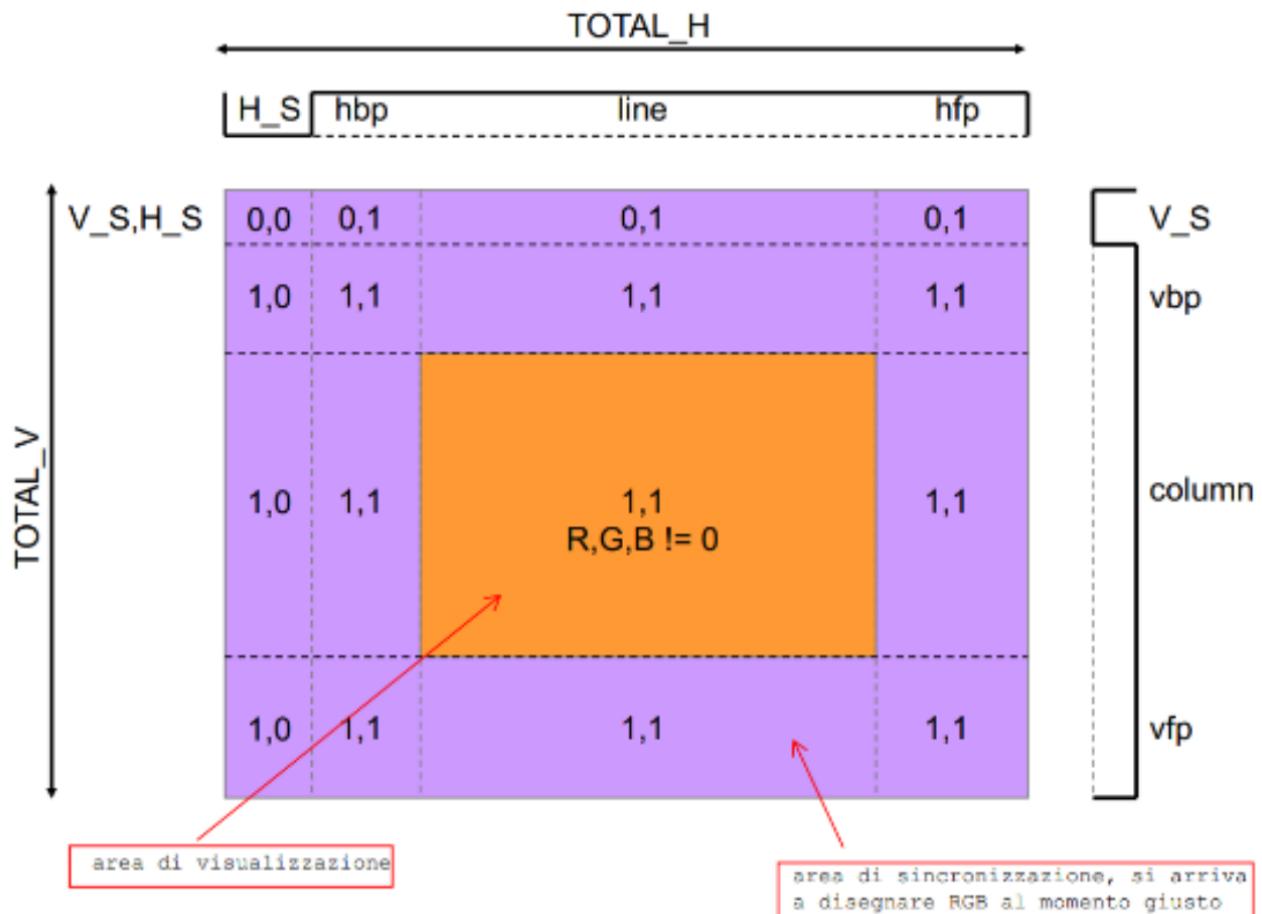
per colore).



Trasmissione di immagini da FPGA ad altro

Con VGA si passa per la memoria per evitare le difficoltà di sincronizzazione, dunque i dati(immagini) vengono salvati in memoria in modo da tenerli pronti per la visualizzazione appena la VGA li richiede.

Utilizza un circular frame buffer per tenere traccia dell'ultimo frame depositato in memoria, utile per rilassare i vincoli tra controller VGA, ARM e FPGA.



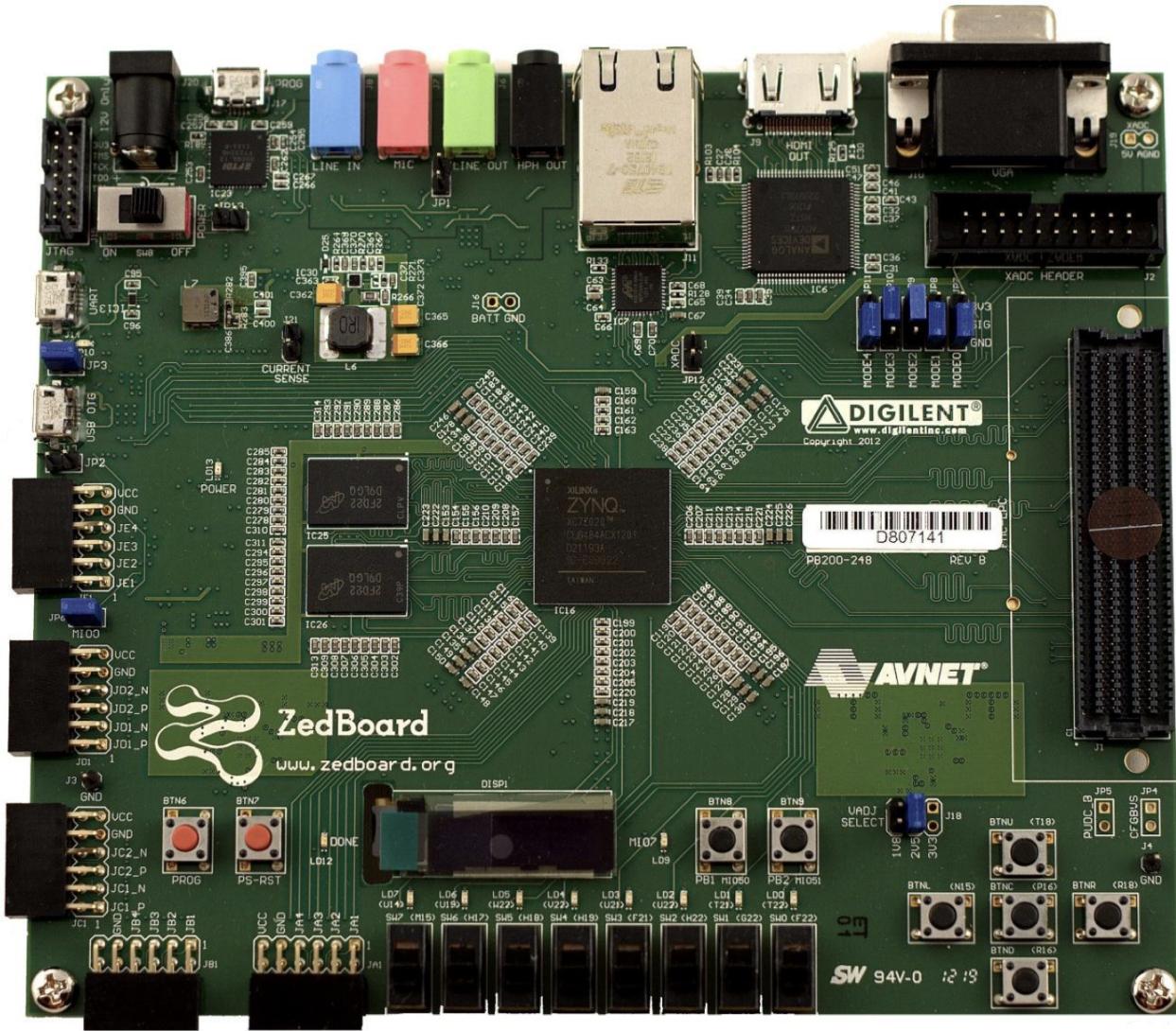
Zynq & Pynq

- FPGA prodotte da Xilinx ❤
- Composte da FPGA + 2 CPU ARM + memoria condivisa tra PS e PL per comunicazione (in alternativa al bus protocol)
- Programmazione
 - Vivado HLS → creazione di blocchi custom o pre-esistenti
 - Vivado SDK → programmazione CPU
- Basso consumo  → ideali per embedded, ma freq. CPU minore dei PC
- Progettate per sfruttare al meglio il parallelismo.

La ZedBoard

La ZedBoard di base ha:

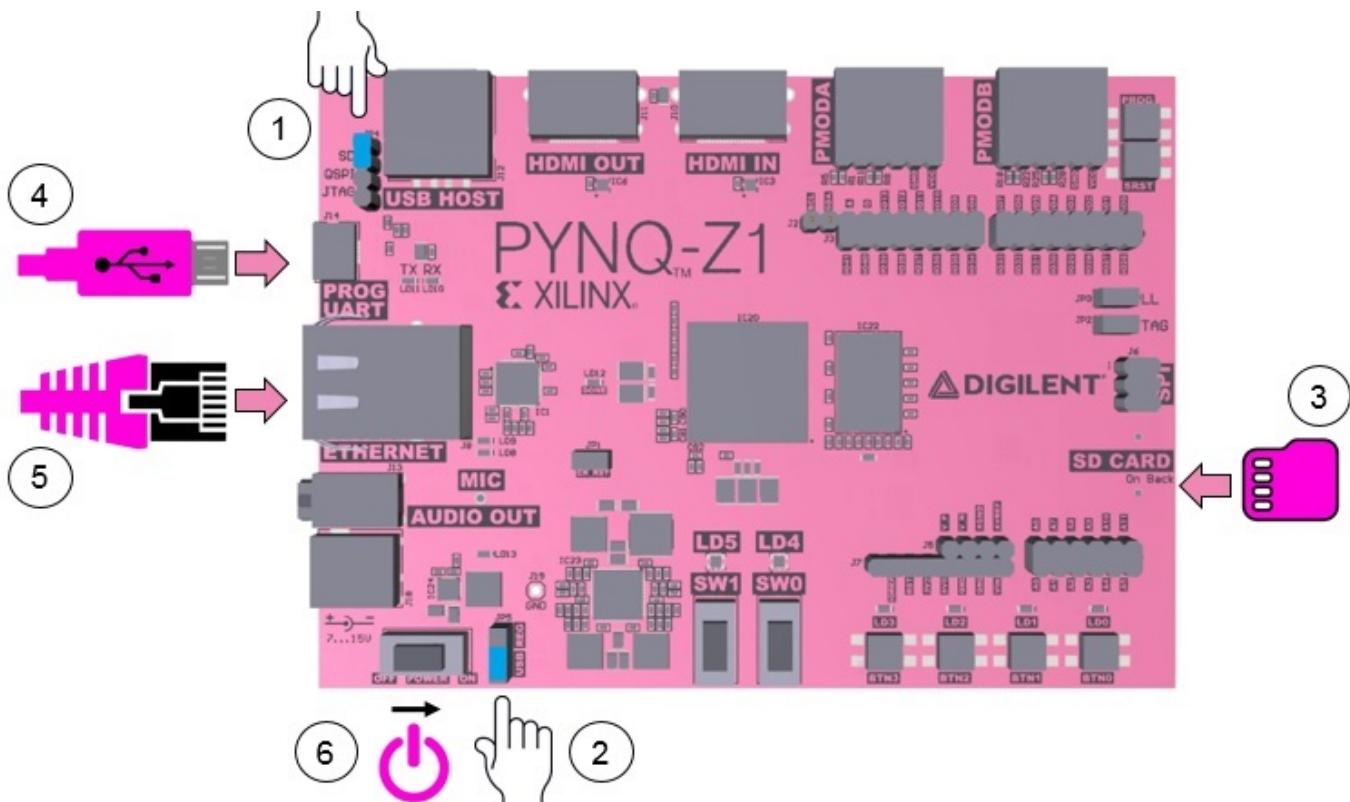
- 8 led e 8 switch
- 5 pulsanti disposti a croce
- 2 pulsanti addizionali
- 5 Pin mode (porte di espansione): Si hanno a disposizione 5 pin-mode, A e B sono i più interessanti in quanto collegati con la FPGA. Mentre il pin connesso alla ARM serve per inviare comandi I2C e comandare la telecamera, inizializzandola e programmandola in maniera appropriata. Sono connettori che hanno 12 pin. 8 di questi sono per i segnali, poi su ogni pin mode ne abbiamo 2 riservati alla massa (GND), e due per l'alimentazione che è 3,3V. Ogni pixel della telecamera è codificato con 8 bit quindi servirono 3 pin mode.
- Display led che si può comandare attraverso lo Zynq.
- Connettori per l'acquisizione di segnali audio.
- Connettore HDMI e il relativo controller
- Oscillatore (che è un clock a 100MHz) che genera segnali a frequenza programmabile.
- Scheda Ethernet utilizzabile attraverso una semplice libreria, che si usa per trasmettere le immagini elaborate in FPGA in streaming
- Convertitore digitale e anche un connettore sulla dx che consente di comunicare direttamente con il pin dello Zynq in totale libertà. Nel progetto serve indicare come connettere i segnali ai pin collegati ad oggetti già presenti nella scheda stessa.
- Ci possiamo scrivere anche con vivado HLS e progettare un controllore per inviare segnali ad un monitor o ad un connettore VGA.



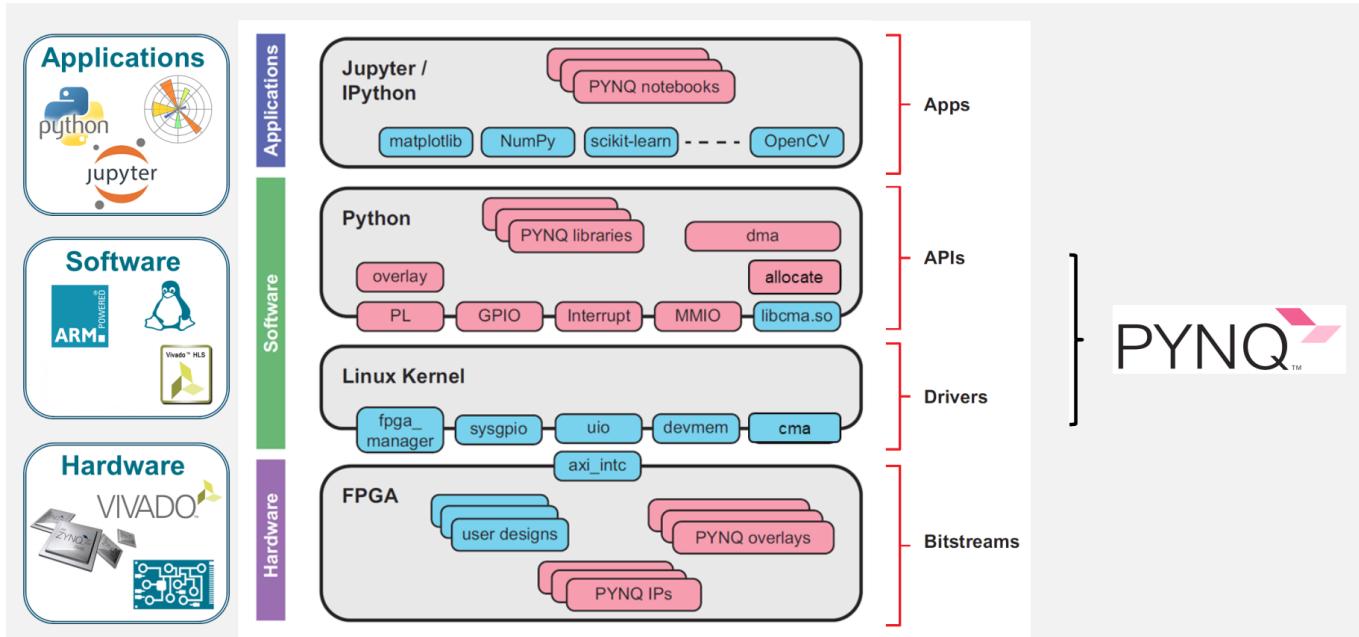
La celeberrima ZedBoard/Zynq.

La Board PYNQ

Una board molto simile alla Zedboard, con la particolarità di avere uno stack software tale da permetterci di **interagire in Python** con la parte di logica riconfigurabile.



Schema della PYNQ.



Lo stack della PYNQ.

Allo strato più basso dello stack si trova la logica riconfigurabile dell'FPGA, ma anche un processore ARM dual-core. Vi sono poi dei driver che permettono il dialogo tra il mondo Python e l'FPGA.

Più in alto nello stack si trovano dei moduli Python ad-hoc e infine, a livello applicativo, si possono addirittura scrivere dei Notebook con Jupyter.

Gli Overlay

Il Processing System può demandare l'esecuzione di alcune funzioni alla Programmable Logic attraverso la chiamata di specifiche librerie hardware chiamate Overlay. Queste possono sia accelerare funzionalità software, sia implementare funzionalità di più basso livello letteralmente mappate sull'hardware e richiamate poi dal processore.

Questo può essere utile per far eseguire all'FPGA determinate parti di codice che possono essere ben parallelizzate. Non si delega l'esecuzione ad un linguaggio più efficiente, come ad esempio avviene con `ctypes` in Python, ma proprio ad un **hardware dedicato!**

Si possono vedere gli overlay come **una vera e propria libreria software**. Infatti, gli overlay mettono a disposizione una signature.

 Tramite il Processing System (ARM) è possibile eseguire operazioni direttamente sulla Programmable Logic (FPGA).

Gli overlay sono tipicamente scritti da esperti, ma i programmati possono chiamarli via Python in modo trasparente, senza la necessità di conoscerne dei dettagli.

Overlay d'Esempio

Dato un bitstream, la creazione di un overlay segue questa procedura

```
from pynq import Overlay

overlay = Overlay('/home/xilinx/tutorial/tutorial_1.bit')
```

Si recupera poi una sorta di “puntatore” all'IP-core definito nel bitstream

```
add_ip = overlay.scalar_add
help(add_ip)
```

Il metodo `help` stampa l'oggetto IP che viene estratto dall'overlay.

Se si vuole, si può creare un driver specifico per l'overlay, che espone una chiamata `add` in modo tale da rendere più **trasparente** l'interazione con lo sviluppatore.

Per farlo, semplicemente si crea un oggetto che eredita da `UnknownIP` e che definisce il metodo `add` con già cablati gli indirizzi a cui leggere e scrivere

```

from pynq import DefaultIP

class AddDriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

    bindto = ['xilinx.com:hls:add:1.0']

    def add(self, a, b):
        self.write(0x10, a)
        self.write(0x18, b)
        return self.read(0x20)

```

N.B.: gli indirizzi a cui scrivere e leggere sono dati dalla sintesi HLS.

Ora si può ricaricare l'overlay, che non sarà più un oggetto di classe `UnknownIP` o `DefaultIP` e si può utilizzarne la chiamata `add`

```

overlay = Overlay('/home/xilinx/tutorial_1.bit')

overlay.scalar_add.add(15,20) # 35

```

Vivado

- *Vivado* → design del progetto, creazione HDL wrapper
 - *Vivado SDK* → codice eseguito dall'ARM
 - *Vivado HLS* → creazione IP-Core custom
- In una evaluation board con una FPGA, tutto ciò che è esterno all'FPGA vera e propria, quando si programma la FPGA, è da considerarsi come esterno (e va dichiarato con `make_external`, anche se fisicamente sulla stessa board. **N.B.:** i LED, gli switch e i bottoni sono esterni!

Quando si crea un progetto Vivado è necessario specificare la board sulla quale si vuole eseguire il progetto (es. Zedboard Zynq).

Esempio Hello LED

Progetto su Zedboard Zynq.

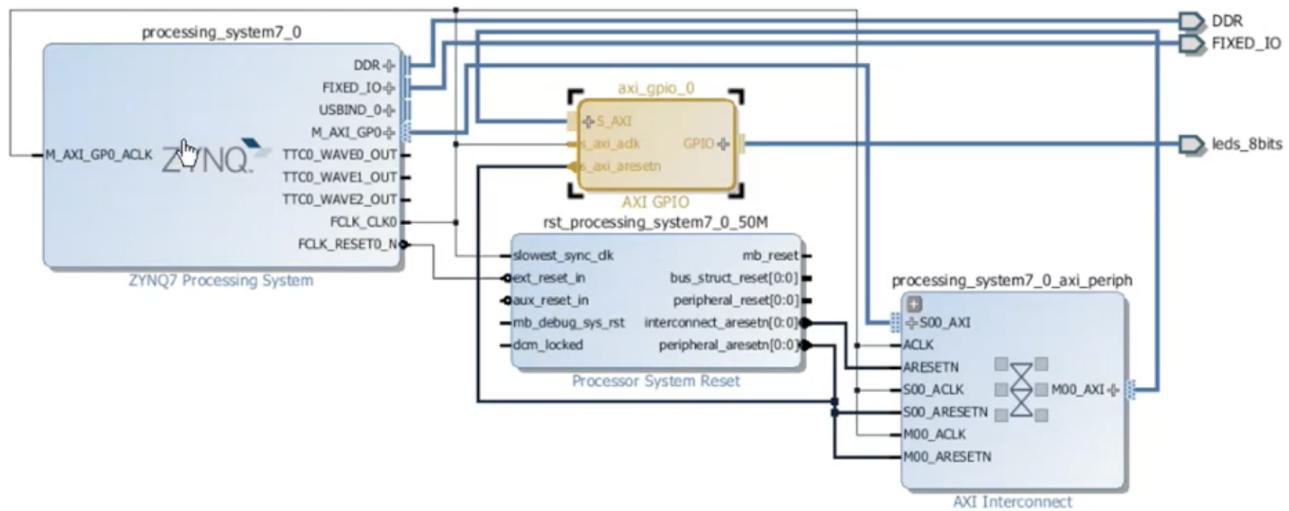
- ARM e FPGA comunicano con AXI Lite

- Nella parte FPGA si mette un registro (si può pensarlo come 8 FFD) mappato nello spazio di indirizzamento dell'ARM
- La FPGA manderà a ogni pin dei LED della Zedboard un bit del registro.

È necessario specificare nel progetto che si vogliono utilizzare i processori ARM (che altrimenti resterebbero spenti di default). Per inserirli basta ricercare *Zynq* negli IP-Core disponibili. Il blocco *Zynq* viene con, tra l'altro, vari segnali già predisposti dall'ambiente (tra cui un master AXI).

Progetto Vivado

- Si aggiunge un blocco *Zynq* per la CPU ARM, che farà da master AXI Lite
- Si aggiunge un blocco per il registro (di sola scrittura) AXI Lite, semplice e adatto ad applicazioni che non richiedono velocità elevate; si specifica che la larghezza del registro sarà di 8 bit (di default sono 32)
- Si specifica che il registro in uscita è da collegare ai LED (questo grazie al fatto che il sistema già conosce la board)
- Si esegue la “Block Automation” e la “Connection Automation” per connettere i blocchi in automatico.



Risultato finale. Le due fasi di automazione hanno aggiunto i collegamenti tra i blocchi, compresi quelli esterni, e ha aggiunto due blocchi: uno per resettare l'ARM (Processor System Reset) e uno che funge da hub (AXI Interconnect) tra ARM e registro AXI Lite.

Prima di poter eseguire sulla board, bisogna eseguire gli step di creazione di un wrapper HDL (“come funziona il sistema visto ai morsetti?”), validazione del design (opzionale, ma consigliato) che va a cercare se ci sono segnali non connessi o altre cose “strane” e, per ultimo, la generazione del bitstream.

Progetto Vivado SDK

In SDK si può creare del codice di test che sfrutti il progetto custom che si è creato.

In SDK si può scegliere il Sistema Operativo per il quale si vuole sviluppare (Linux, standalone, ...).

Vivado HLS

- Progettazione di IP-Core in C/C++, con eventuale main di test (*testbench*)
- Variabili di I/O dichiarate `volatile` per evitare compiler optimizations
Esempio: `void func(volatile bit *led_out)`
- `ap_int.h` per definire tipi di dato con dimensione variabile (un)signed → `ap_int<size>`, `ap_uint<size>`
 - `ap_uint<5>` → unsigned int di 5 bit

Vivado HLS fornisce diverse opzioni di ottimizzazione. Due esempi notevoli (descritti meglio più avanti) per i loop sono:

- Pipelining
- Loop unrolling.

Latency e Interval

- *Latency* - # cicli di clock necessari al modulo per generare il risultato
- *Interval* (o Throughput) - # cicli di clock necessari prima di poter processare un nuovo input

*Latency e interval **non** sono la stessa cosa!*

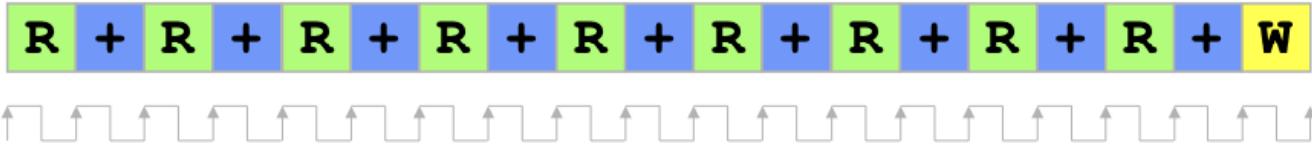
 **Ricordate?** Nel DLX pipelined riuscivamo ad avere (a meno di stalli) una latency di 5 clock ed un interval di 1 clock.

Cosimulazione

Dal testbench e l'output della sintesi è possibile esaminare le forme d'onda, a patto di avere un software esterno di simulazione capace di leggere il file apposito con le forme d'onda.

Ottimizzazioni

Si consideri l'esempio seguente: calcolo della somma degli elementi di un vettore di `N` elementi, `N=9`.

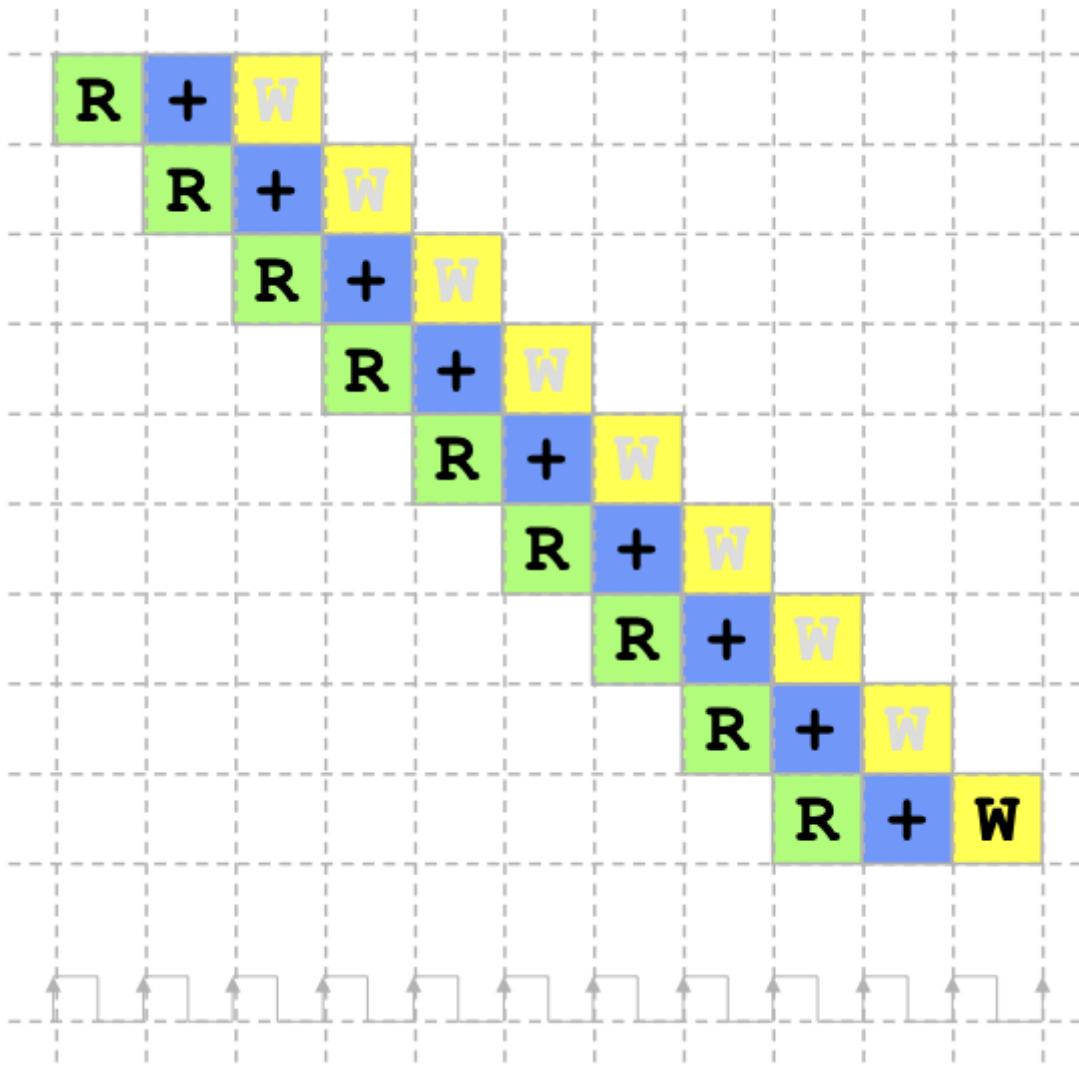


Per 9 elementi, servono 19 cicli di clock per completare l'esecuzione: ogni iterazione è composta di due fasi da 1 ciclo di clock l'una, una read e un'addizione, quindi per 9 elementi servono 18 cicli di clock. Infine, serve un'operazione di write del risultato, che porta il totale a 19.

Il codice C di HLS che risolve l'esempio è

```
void media(volatile int input_array[LENGTH], volatile int *average_value)
{
    int temp_sum = 0;
    for (int i = 0; i < LENGTH; i++) {
        temp_sum = temp_sum + input_array[i];
        *average_value = temp_sum;
    }
    return;
}
```

- **Pipelining** → Dividere l'esecuzione di un'iterazione in N step indipendenti che possono essere eseguite parallelamente ad ogni iterazione



La versione pipelined dell'esempio riduce la latency a 11 cicli di clock (dai 19 di partenza).

- Pragma `#pragma HLS PIPELINE II=1` → `II=1` indica che l'iteration interval desiderato è di un ciclo di clock (si desidera elaborare un dato ad ogni clock, non è detto che il compilatore ci riesca).
- **Loop Unrolling** → “srotolare il loop” eseguendo ad ogni iterazione (parallelamente) quello che “normalmente” viene eseguito in `N` iterazioni, e iterare a step di `N`
 - Il loop si “muove” a incrementi (step) di `N`
Si immagini qualcosa del genere: `for (i = 0; i < NUM_ITERAZIONI; i += N)`
 - In ogni iterazione si eseguono `N` calcoli.
 - Per poterlo implementare i dati devono supportare l'accesso multiplo
 - La replicazione di risorse necessaria porta a maggior costo in termini di risorse allocate (più CLB banalmente). Replicazione di risorse? Sì, se si vogliono fare, ad esempio, `N` somme contemporaneamente si avrà bisogno di `N` full adder che lavorano in parallelo.

Protocolli

È possibile utilizzare diversi protocolli in Vivado HLS per le variabili di I/O e per I/O a livello di blocco.

Block-Level I/O

Per I/O a livello di blocco, utilizzando la pragma `ap_ctrl_none` per la top-function, si specifica che non si vuole che vengano predisposti segnali per controllare quando la funzione inizia/finisce le operazioni, è idle, o pronta per un nuovo input. Di default è applicato `ap_ctrl_chain`.

Interfacce per Porte

Per le variabili di I/O (quelle nella signature della top-function) è possibile specificare l'utilizzo di diversi protocolli:

- `ap_none` → nessun segnale di controllo, nessun overhead hardware, ma il timing di I/O deve essere correttamente gestito in maniera autonoma
- `ap_fifo` → protocollo FIFO, compatibile con array di argomenti acceduti in maniera sequenziale; non richiede che siano generate informazioni sugli address; non può essere usato per porte bidirezionali
- `axis` → interfaccia per AXI Stream
- `s_axilite` → interfaccia per uno slave AXI Lite
- `m_axi` → interfaccia per un master AXI.

L'utilizzo è il seguente: `#pragma HLS INTERFACE <iface name>`.

Introduzione Alle CNN

Le **Neural Network** (NN) sono metodologie di *Machine Learning* che prendono *ispirazione dalla biologia* del sistema nervoso.

Artificial Neuron: $out = F_{att}(\sum x_i w_i + b)$

- F_{att} - funzione di attivazione
- x_i - ingresso i-esimo
- w_i - peso i-esimo
- b - bias, costante da sommare alla somma pesata degli ingressi.

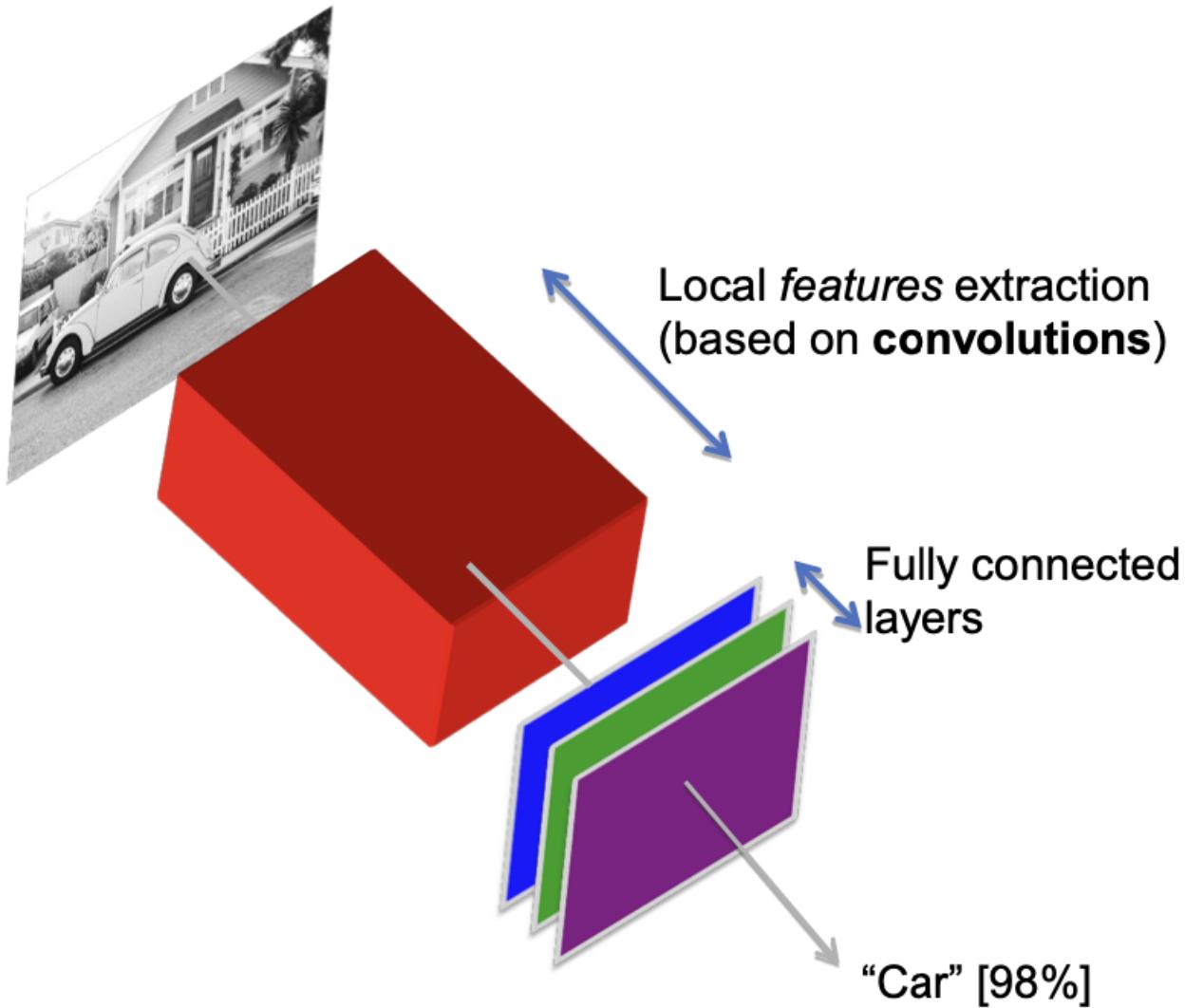
Una NN è fatta di più **livelli**, ossia "strati" di neuroni.

Ogni livello aggiunge uno strato di elaborazione alla rete e:

- Riceve in input dallo strato precedente
- Manda in output allo strato successivo.

Le NN vanno addestrate per trovare i migliori valori per i pesi (e il bias) per ogni neurone, per risolvere uno specifico task.

Convolutional Neural Networks



Schematizzazione di una CNN.

Sono reti neurali basate su layer *convoluzionali*, degli strati cioè specializzati per eseguire delle operazioni dette di **convoluzione**.

Per eseguire un'operazione di convoluzione si ha bisogno de:

- Il **dato d'ingresso**, per semplicità si immagini un'immagine in grayscale, visualizzabile come una matrice $H \times W$ di interi a 8 bit in cui il generico elemento $M[i, j]$ codifica

l'intensità luminosa del pixel.

- Il **kernel** (o *filtro*) convoluzionale, ossia una matrice quadrata $N \times N$ di numeri interi, chiamati anche “**pesi**”.

La singola operazione di convoluzione si fa tra l'immagine e il kernel.

Si supponga di avere un kernel 3×3 . Si sovrappone il kernel ad un pixel qualsiasi dell'immagine; l'operazione di convoluzione non fa altro che sostituire il valore di quel pixel con un nuovo valore, calcolato come media pesata (coi pesi del kernel) dei pixel circostanti.

La convoluzione si fa per tutti i pixel dell'immagine “scorrendo” il kernel da sinistra a destra, dall'alto verso il basso.

Un problema che sorge è: “*cosa fare dei pixel della cornice*”? Dato che bisogna **centrare** il kernel su ogni pixel dell'immagine originale, i pixel della riga 0, ad esempio, non hanno pixel immediatamente sopra con i quali fare la media pesata. Questo piccolo inconveniente si può risolvere in alcuni modi:

- **Creando artificialmente una cornice** delle dimensioni richieste (ad esempio se il kernel è 3×3 è sufficiente una cornice di un pixel per ogni lato), in cui i pixel della cornice hanno valori
 - Tutti zero (cornice nera)
 - Replica specchio i pixel esistenti immediatamente sotto, a destra o a sinistra
- Scartando un certo numero di righe e colonne, ponendo il kernel al centro del primo pixel immediatamente convolvibile. Ad es: se il kernel è 3×3 , lo si centra come prima convoluzione in posizione $M[1,1]$
 - Così facendo non c'è molta perdita di dettaglio, anche perché culturalmente si tende ad avere poca informazione ai bordi dell'immagine e molta al centro.

I Filtri Convoluzionali Nelle Reti Neurali

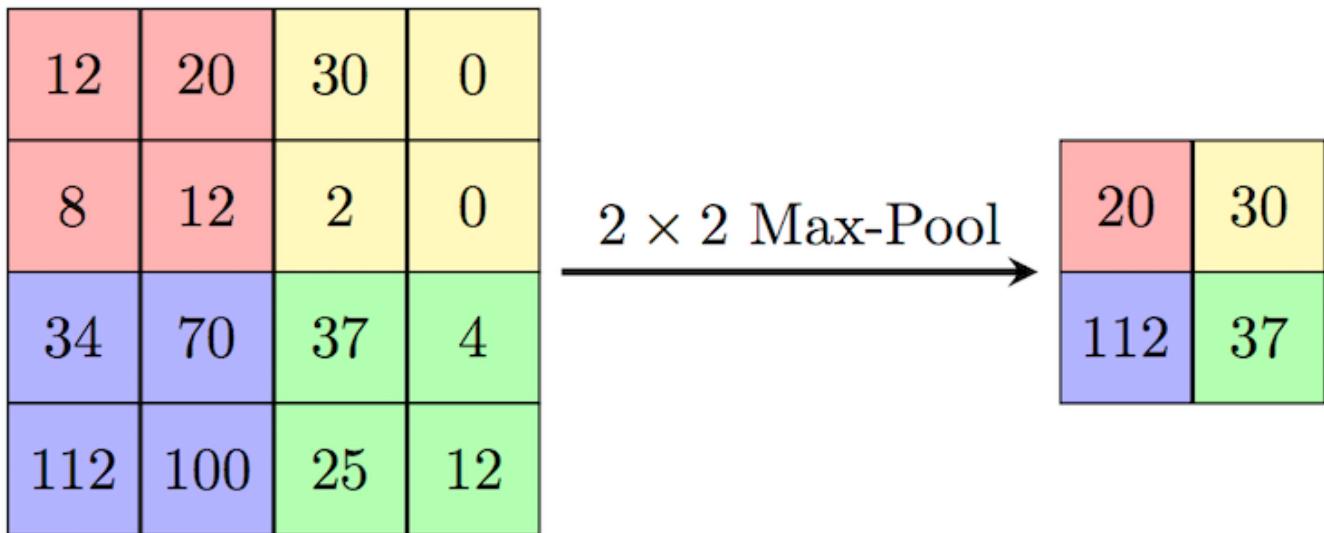
I **filtri convoluzionali** sono da molti anni impiegati nelle applicazioni di fotoritocco, ad esempio per inserire dei filtri, sfocare immagini, eccetera. Questo perché ad ogni singolo kernel corrisponde un effetto finale diverso.

Nelle reti neurali convoluzionali, soprattutto per l'elaborazione di immagini, si usano degli stadi in cui sono presenti filtri a cascata nell'ordine di grandezza delle centinaia. Questo perché esistono alcuni filtri in grado di porre in evidenza delle feature interessanti, come ad esempio i bordi dei soggetti particolarmente contrastati.

Allora, si può invertire il compito e “chiedere” alla rete di **apprendere** un certo numero di kernel sulla base di un dataset che è fatto da immagine + ground truth value per costruire un modello in grado di predire che una certa immagine corrisponde ad una certa classe.

Pooling

La dimensionalità delle immagini può raggiungere valori molto alti e il task di convoluzione, che già di per sé è abbastanza oneroso, può diventare inaffrontabile. Per questo esistono degli stadi intermedi tra convoluzioni che sono realizzati dai cosiddetti livelli di **pooling**.



Esempio di max-pooling.

Consistono in processamenti intermedi atti a ridurre sensibilmente la dimensionalità di un certo dato. Riprendendo l'esempio di un'immagine, si suppona (semplificando) di avere una matrice 4x4 in uscita da una serie di convoluzioni. Un esempio di pooling (in questo caso il **max pooling**) potrebbe essere quello di suddividere la matrice in ingresso in quattro quadranti e di conservare su una matrice di output 2x2 il valore massimo di ognuna delle sottomatrici.

Altri metodi, al posto del valore massimo, usano la media o altri parametri, ma il concetto alla base non cambia: si aggregano i dati di partenza in una struttura notevolmente più piccola, che conserva comunque un certo quantitativo dell'informazione di partenza.

I Layer Finali (Fully Connected)

Prendono in input un dato dei livelli precedenti e lo sottopongono al processo di **flattening** (lo riportano in una sola dimensione, un array monodimensionale).

Applicano poi una funzione *non-lineare* e infine una *soft-max*, la quale crea una *distribuzione di probabilità* su un array che rappresenta l'output. I valori di questo array finale sono compresi tra (0, 1) e la somma di tutti i valori dell'array deve essere esattamente 1.

Ad esempio: per addestrare un modello a classificare se una data immagine appartenga ad una delle classi *{Cane, Gatto, Cavallo}*, operativamente si vuole in output un vettore *Out* che ha tre componenti di probabilità:

$Out[0] = P(\text{immagine} = \text{Cane})$
 $Out[1] = P(\text{immagine} = \text{Gatto})$
 $Out[2] = P(\text{immagine} = \text{Cavallo})$

Ad esempio, se la funzione soft-max restituisce $Out = (0.99, 0, 0.01)$, questo vettore è da interpretarsi come “*l’immagine rappresenta al 99% un cane*”.

In Python, ad esempio: si crea una list che mappa l’indice del vettore con la classe

```
labels = ["Cane", "Gatto", "Cavallo"]
```

Supponendo che in uscita dalla NN si abbia l’output come distribuzione normalizzata di probabilità, si può comunicare all’utente il risultato nel seguente modo

```
print("L'immagine è stata riconosciuta come un " + labels[np.argmax(out)])
```

 Il metodo `np.argmax` di NumPy, dato un vettore, restituisce l’*indice* del valore massimo.

Embedded Computer Vision

Acquisizione Dell’Immagine

Il Sensore

Per acquisire un’immagine dal mondo reale si ha innanzitutto bisogno di un **sensore**, cioè di un dispositivo fotosensibile che trasforma la radiazione luminosa in informazione adatta all’elaborazione o alla visualizzazione.

Esistono principalmente due tecnologie che permettono di realizzare un sensore:

- **CMOS** (più utilizzata)
- **CCD** (usata solo contesti specifici, ad esempio l’astrofotografia).

Entrambe sono accomunate dal fatto che il sensore è sostanzialmente una matrice di **fotodiodi**, che trasformano la luce che catturano in un segnale elettrico.

Il vantaggio del CMOS è che è più economico, meno complesso, e consuma meno corrente. D’altra parte, la tecnologia CCD è meno suscettibile al rumore e permette di avere un’immagine di qualità più elevata.

Modalità di Acquisizione

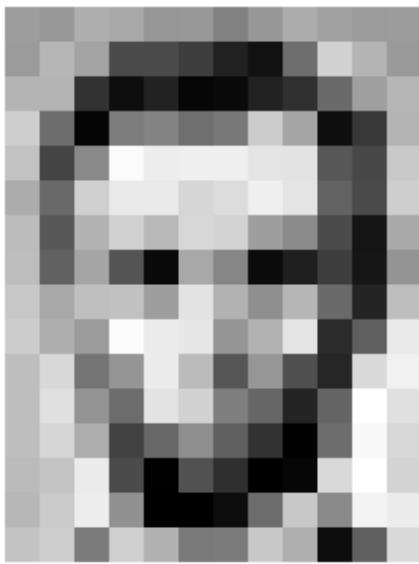
Si hanno due alternative per acquisire un'immagine:

- **Rolling shutter:** l'acquisizione della luce **non** avviene nello stesso istante per ogni pixel, non si crea cioè un'istantanea perfetta, ma è come se il sensore scansionasse la scena per righe da sinistra a destra e dall'alto verso il basso. Parti della scena diverse sono quindi acquisite in momenti diversi, seppur molto vicini → In caso di acquisizione di una scena molto dinamica potremo avere delle distorsioni
- **Global shutter:** tutti i pixel del sensore vengono acquisiti nello stesso istante e quindi viene prodotta effettivamente una istantanea della scena → serve una grande capacità di parallelizzazione.

Spazio Colore

Grayscale

Con questo modello di colore è possibile acquisire la scena e codificare la **luminosità** entro un range, ad esempio [0, 255] se si utilizzano 8 bit, ottenendo un'immagine in scala di grigi



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	257	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	105	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	85	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	19	96	218

157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	105	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	85	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	19	96	218

Esempio di un'immagine in scala di grigi.

Colore

Per rappresentare i colori un solo canale per ogni pixel non può bastare, perché può al massimo rappresentare l'intensità della luce in scala di grigi. Si devono prevedere più canali per ogni immagine → uno per ogni colore primario (altri colori ottenibili con sintesi additiva).

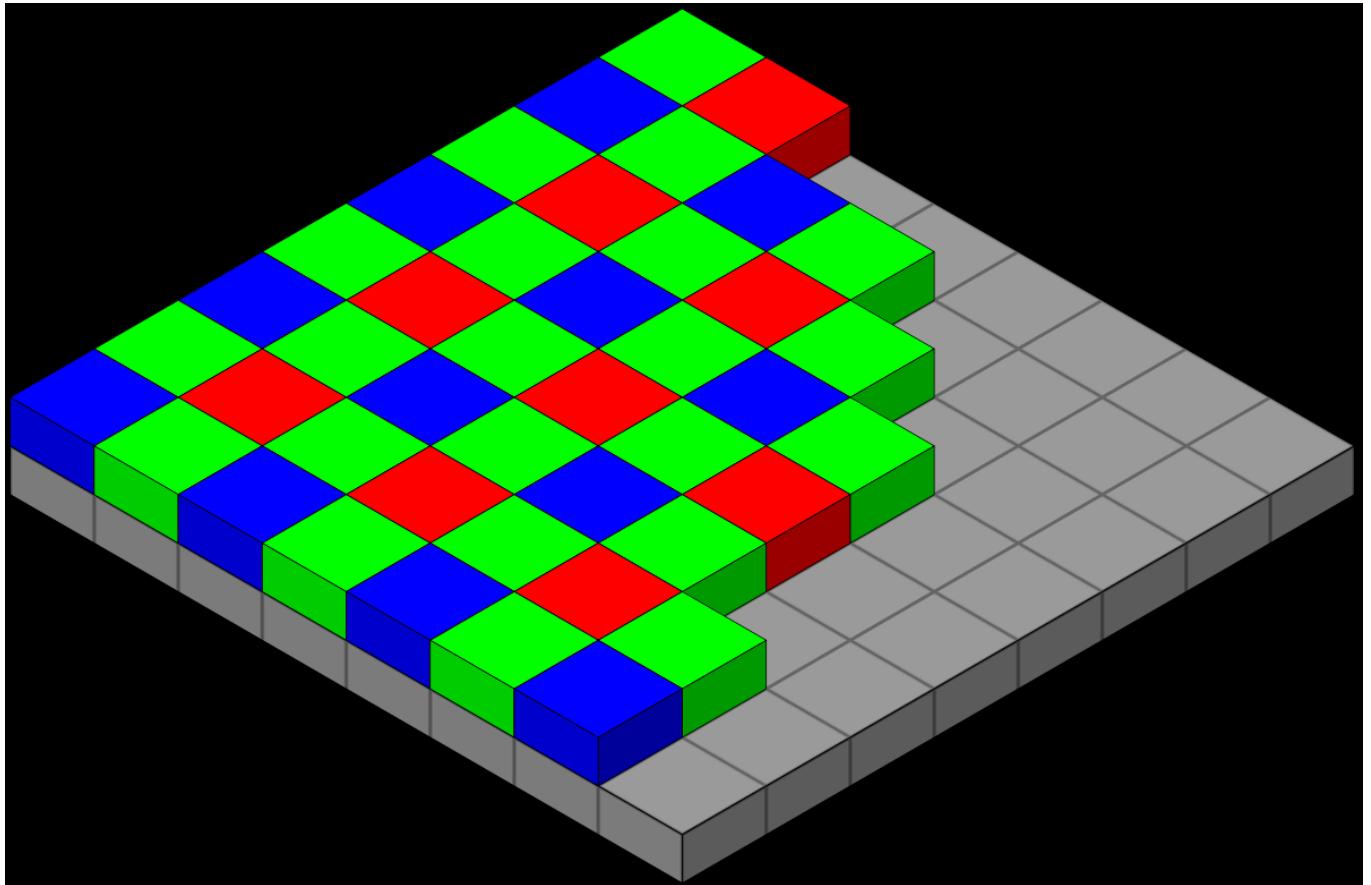
Pattern Bayer

Avere, per ogni singolo punto che si vuole rappresentare, ben tre fotosensori, ognuno sensibile ad una delle tre diverse radiazioni (Rosso, Verde e Blu) significa triplicare il numero di

componenti elettronici in ogni sensore (\rightarrow un po' too much).

Lo schema Bayer permette di **risparmiare** sui sensori senza troppa perdita di informazione.

Consiste in una mascheratura del sensore con una *matrice* di filtri, ognuno dei quali fa sì che il dispositivo fotosensibile che sta sotto riesca a registrare una sola lunghezza d'onda.



Bayer Pattern.

Ogni pixel quindi acquisisce il livello di luminosità di un solo canale e inferisce il valore degli altri due facendo un'interpolazione tra i pixel contigui.

Codifica YUV (o YCbCr)

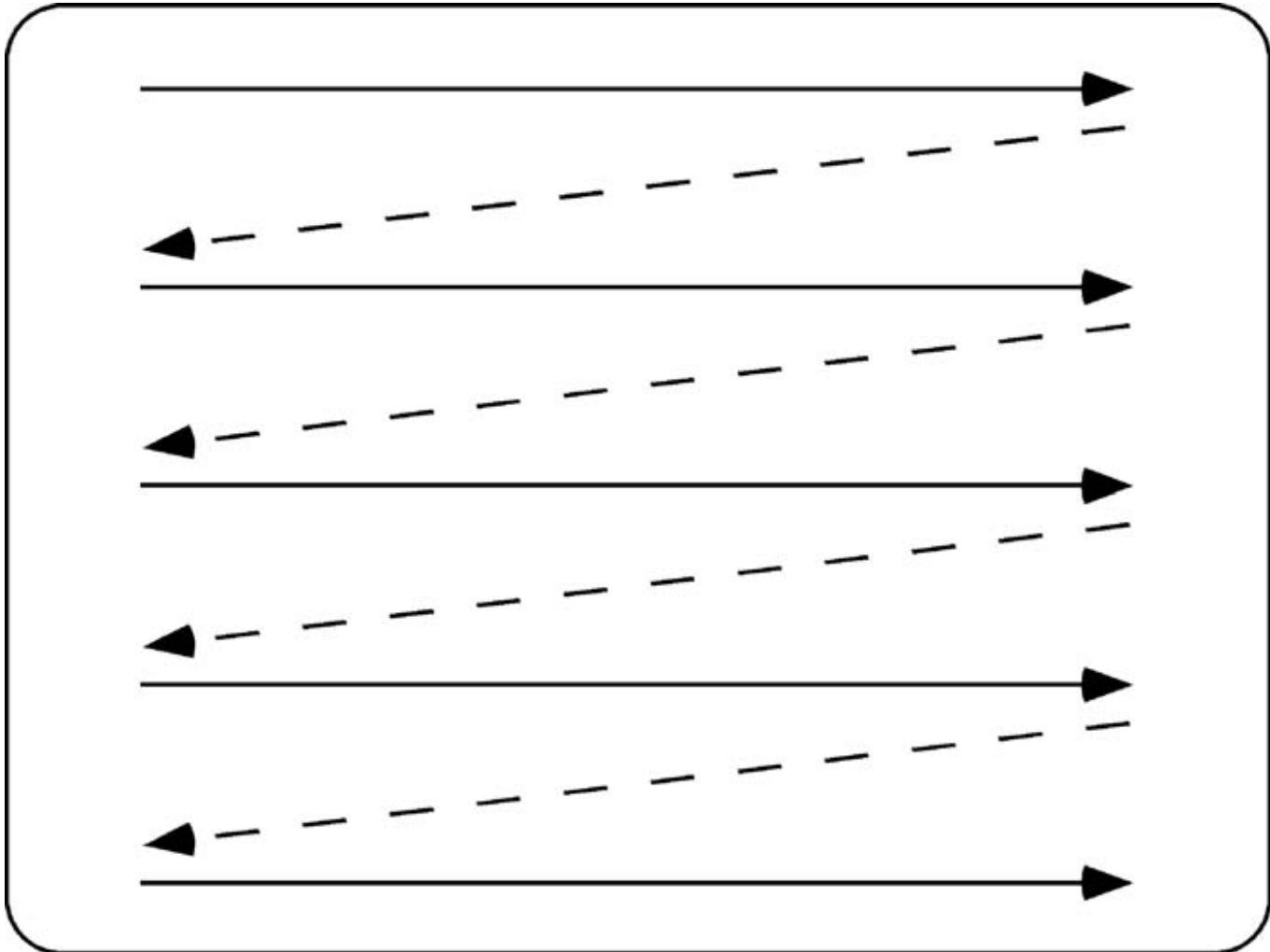
Ogni immagine codificata in YUV ha tre componenti:

- **Luminanza Y**: rappresentazione della luminosità (come in scala di grigi)
- **Crominanza UV**: U e V sono le informazioni del colore rispettivamente per la radiazione blu e quella rossa.

Si precisa che le componenti U e V vengono campionate un pixel sì e uno no, orizzontalmente o verticalmente. Se ogni pixel è rappresentato con 16 bit, una possibile scelta delle proporzioni è 8:4:4.

La Trasmissione Dell'Immagine

I pixel di un'immagine vengono inviati secondo il pattern del *raster scan*: da sinistra a destra e dall'alto verso il basso.



Raster scan schematizzato.

Lo streaming dei pixel che viene generato da un sensore **non può essere fermato** → occorre quindi predisporre un metodo di acquisizione che stia al passo il più possibile.

Si definisce **PIXEL_CLOCK** la frequenza con cui il sensore genera il singolo pixel, e **framerate** il numero di frame (immagini) che il sensore riesce a generare ogni secondo.

💡 Osservazione: per mantenere lo stesso framerate aumentando, però, la risoluzione, il **PIXELCLOCK** deve *diminuire*. $\text{Px}\{\text{clk}\} \propto \text{framerate}^{-1}$

Lo streaming dei pixel include, insieme ai veri e propri pixel, anche alcuni segnali riservati, ad esempio i valori 0,1,2,3. Questi valori, che si tolgono dai possibili valori del pixel, il dispositivo di acquisizione può interpretarli come segnali di **inizio linea** e **inizio frame**, ma anche come segnale di **pixel valido**.

-	-	-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-	-	-
A	B	C	D	E	F	G	H				
I	J	K	L	M	N	O	P				
Q	R	S	T	U	V	W	X				
Y	Z	S	a	b	c	d	e				
-	-	-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-	-	-
A'	B'	C'	D'	E'	F'	G'	H'				
I'	J'	K'	L'	M'	N'	O'	P'				
Q'	R'	S'	T'	U'	V'	W'	X'				
Y'	Z'	S'	a'	b'	c'	d'	e'				
-	-	-	-	-	-	-	-	-	-	-	-

Un esempio possibile di trasmissione è il seguente:

1. All'inizio il segnale di PIXEL_VALID è a zero
2. Ad un certo punto il segnale di FRAME_VALID va a Vcc \Rightarrow sta per iniziare il frame
3. Nel momento in cui sia LINE_VALID che PIXEL_VALID vanno a Vcc, allora l'immagine che si sta ricevendo ha significato, in particolar modo si sta ricevendo una riga
4. La riga è finisce quando il segnale LINE_VALID torna a zero
5. La nuova riga inizia quando si verificano le stesse condizioni del punto 3
6. Quando il segnale di FRAME_VALID torna a zero il frame termina
7. Si può ricevere un nuovo frame nel momento in cui FRAME_VALID torna a Vcc.

Python

È un linguaggio sviluppato nei primi anni '90.

- **Interpretato:** script da dare in pasto ad un interprete (invece di compilazione in assembly o bytecode intermedio) Vantaggi:
 - prototipazione e debugging molto veloce
 - diffusione del codice molto veloce
 - veloce da imparare per i non addetti ai lavori
- Svantaggi:
 - un'applicazione non-nativa, come quelle Python, non sarà mai perfettamente ottimizzata per la macchina sottostante

- **Object-Oriented**, multiple inheritance
- **Dynamically typed**
- Possiede molte **librerie** per **Data Analysis** e **Machine Learning**, proprio perché gli esperti di altri settori come Data Science e Statistica tendono a lavorare con un linguaggio semplice da imparare come Python.

💡 Python vs C

Feature	Python	C
Codice	Interpretato	Compilato
Tipizzazione	Dinamica	Statica
Velocità	😔	😊
Object-Oriented	Sì	No (C++ sì)
Ereditarietà	Sì, multipla	No (C++ sì, multipla)
Portabilità codice	Facile	Un po' meno facile
Deallocazione della memoria dinamica	c'è il garbage collector	va fatta a mano con primitive come la free(...)
Sintassi	meaningful-whitespaces	Token speciali ; {, ...

Python 2.x vs Python 3.x

La coesistenza tra la versione 2 (supporto cessato in Gennaio 2020) e la versione 3 continua da anni. I due linguaggi differiscono solamente per alcuni aspetti: quello più plateale, e che può portare a problemi se non se ne tiene conto, è la **divisione fra interi**.

In Python 2.x la divisione tra interi restituisce un valore intero senza resto

```
a = 5/2 # a = 2
```

In Python 3.x invece quello che accade è che il risultato è un valore floating point

```
a = 5/2 # a = 2.5
```

Il Gestore dei Pacchetti

Le librerie sono installabili attraverso il package manager **pip**, che in genere è installato insieme con Python. I pacchetti che sono disponibili si possono installare in questo modo

```
pip3 install [package_name]
```

L'Ambiente Virtuale

È possibile, e anche caldamente **consigliato**, per ogni progetto Python creare **virtual environment**

```
virtualenv -python3 venv
source venv/bin/activate
```

Così facendo, il progetto girerà all'interno di una **sandbox**, nella quale saranno installate le versioni dei pacchetti da utilizzare, senza conflitti con eventuali altre versioni usate in altri progetti o nell'ambiente globale.

Virtualenv è una libreria che va prima installata con pip (`pip3 install virtualenv`).

Tipi Primitivi e Casting

I tipi primitivi Python sono:

- Integer
- Float
- String (racchiudibili indifferentemente tra apici singoli o doppi)
- Boolean.

È possibile il casting, ma solo quando questa operazione è “sensata”:

```
age = "25"
next_age = str(int(age) + 1) # risultato: stringa "26"

height = 180.4
integer_height = int(height) # risultato: 180
```

Le Collection

Tuple

Sono astrazioni di livello molto alto per gestire liste di elementi.

- Collezioni immutabili di dati separati da virgola, eventualmente racchiuse tra parentesi tonde

```
users = ('admin', 'andrea', 'lorenzo', 'giorgio')
```

- **Indicizzate** `users[N]` → `N`-esimo elemento dell'array

Liste

- Collezione **mutabile** di oggetti, anche di tipo *disomogeneo*, **indicizzata**

```
my_list = ["apple", "banana", "cherry"]
```

Dictionary

- Collezione **mutabile** di **key-value pairs**, **indicizzata**

```
my_dict = {
    "brand": "Pontiac",
    "model": "Firebird",
    "year": 1987,
    "owner": "Dwight Schrute"
}
```

Funzioni

- Definibili con la parola chiave `def`

```
def pow(x, y=2): # 2 è valore di default per y
    return x ** y
```

- Se un parametro non è passato ed esiste un valore di default, quest'ultimo viene usato

```
cube = pow(2, 3) # 8, perché 3 "sovrascrive il valore di default"
square = pow(2) # 4, viene usato il valore di default
```

Scope

- Tutte le **variabili definite all'interno di una funzione** rimangono **locali** ad essa e quindi non visibili all'esterno.
- Si possono definire ed utilizzare variabili globali, che sono tutte quelle variabili definite fuori da una funzione.

```
x = 655 # variabile globale

def myFunction():
    global y # variabile globale
```

```
y = 657
```

```
myFunction()  
print(x + y) # 1312
```

Oggetti

Python consente di definire classi ed istanziare oggetti.

```
class Person(object):  
    def __init__(self, name):  
        self.name = name  
  
    def introduce_yourself(self):  
        print("Hello, my name is  
{0}".format(self.name))  
  
p1 = Person('Slim Shady')  
p1.introduce_yourself()
```

Il costruttore si indica con il metodo riservato `__init__`, che viene invocato ogni volta che si istanzia un nuovo oggetto.

La parola chiave `self` è analoga al `this` Java, e permette di referenziare l'oggetto su cui si sta invocando il metodo (costruttore incluso).

Non è possibile definire più costruttori, ma usando i valori di default non ve ne è la necessità.

NumPy

Scientific computation library. Offre molte utilities per gestire array n-dimensionali omogenei (dati dello stesso tipo, default `f64`). Ciò che accomuna tutte le possibilità di utilizzo che ci offre numpy è quello di usare **strutture dati n-dimensionali**. Essendo scritta prevalentemente in C, è piuttosto ottimizzata.

Strutture Dati N-Dimensionali

Creazione array:

- Passando una lista
 - Di default il tipo di dato sarà un `np.float64`. Importante: i tipi di dato devono essere assolutamente omogenei

```

list_of_values = [3.141, 2.718, 1.414]
x = np.array(list_of_values)

# si può anche specificare il tipo di dato desiderato per l'elemento
# generico
int_list_of_values = [20, 2, 33]
y = np.array(int_list_of_values, dtype=np.int32)

```

- zeros, ones passando la shape voluta (versione `_like` per ricalcare la shape di un array già esistente), eye per matrice identità, arange passando un range start, stop, step, random per valori casuali

```

my_zeros_array = np.zeros((2,2)) # matrice nulla 2x2

my_ones_array = np.zeros(3) # [1. 1. 1.]

my_identita = np.eye((2,2), dtype=np.int32) # [[1 0], [0 1]]

```

Ci sono almeno due modi per **inizializzare** un numpy array ad un valore *costante*

```

all_fortytwo = np.full((2,2), 42) # [[42. 42.], [42. 42.]]

all_fortytwo_v2 = np.ones((2,2))*42

```

Ogni array ha degli attributi (es. `dtype`, `shape`, ...).

Espansione e Rimozione Delle Dimensioni di un Array

Visualizzare la shape è importantissimo per verificare che un'operazione tra numpy array sia concessa!

```

x = np.array([2., 5., 3.])
y = np.ones((2,4,1,2,3))

print(x.shape) # (3,)
print(y.shape) # (2, 4, 1, 2, 3)

```

La somma tra i due array di cui sopra è difficile da visualizzare. Si può immaginare che sia element-wise, ma ciò avrebbe senso solo se le shape dei due array fossero esattamente identiche. Numpy, cercherà in ogni modo di “far saltare fuori” una possibile somma tra due array, ma non garantisce la buona riuscita o che il risultato sia semanticamente sensato.

Dato un array, si può cambiare la sua forma (aggiungere dimensioni, rimuovere un asse, rimuovere gli assi di dimensione 1, ecc..), l'espansione delle dimensioni si effettua con il metodo `expand_dims`

Dato un array `x`

```
x = np.full((2,2,3),7)
print(x.shape) # (2, 2, 3)
```

Si può incrementare il suo numero delle dimensioni **senza alterarne il contenuto** nel seguente modo

```
x = np.expand_dims(x, 0)
print(x.shape) # (1, 2, 2, 3)
```

Il secondo argomento, `0`, indica che si vuole aggiungere una dimensione alla posizione `0` della shape di `x`. Si può espandere **in coda alla shape** indicando come indice `-1`

Con il metodo `squeeze` si possono rimuovere una o più dimensioni, a patto che quella o quelle dimensioni siano `1`

```
x = np.full((20, 1, 1), 42)
# una matrice 20x(1x1) è all fin fine un vettore di 20 scalari, no?
x_squeezed = np.squeeze(x)
assert x_squeezed.shape == (20,)
```

Si può **decidere la dimensione precisa** da eliminare tramite l'attributo `axis`, di default `None`. Per farlo, bisogna sapere che quel preciso `axis` sia monodimensionale, altrimenti numpy lancerà un errore.

Accesso Agli Elementi di un NumPy Array

Dato un array si può accedere all'i-esimo elemento:

- Tramite **notazione a indice** `arr[i]`
- Con il **metodo** `item`, `digits.item(3)`
N.B.: Fortunatamente, gli elementi di un numpy array **NON sono immutabili!** 😊

Slicing Degli Array

Si può selezionare un sottoinsieme dei numpy array con la **slice notation**

```
digits_greater_5 = digits[6:]
digits_between_2_9 = digits[2:9]
even_digits = digits[::-2] # cifre multiple di quell'indice a partire
dall'indice 0
```

Concatenazione di Array

Tramite il metodo `concatenate` è possibile “appendere” un array ad un altro, sempre che ci sia compatibilità tra le shape

```
x = np.full((4,3),9)
y = np.zeros((4,1))

x_y = np.concatenate([x,y], axis=-1) # si fa inferire l'asse a numpy

assert x_y.shape == (5,3)
```

Funzionalità Matematiche Base di NumPy

Somma e sottrazione element-wise tra array di numpy sono forniti in modo molto trasparente quando le shape tra i due vettori sono identiche. Queste operazioni vengono eseguite in **maniera molto efficiente**.

```
all_sevens = np.full((2,2),7)
all_ones = np.ones((2,2))

all_eights = all_sevens + all_ones
assert np.array_equal(all_eights, np.full((2,2),8))

all_sixes = all_sevens - all_ones
assert np.array_equal(all_sixes, np.full((2,2),6))
```

Potremmo voler sommare o sottrarre elementi che non hanno la stessa shape, ad esempio sommare un array con uno **scalare**

```
y = 1 # scalare

all_eights_broadcasted = all_sevens + y
assert np.array_equal(all_eights_broadcasted, all_eights)
```

Quello che fa NumPy sotto è fare broadcasting e quindi espandere la shape di `y` per farla matchare con quella dell'altro array. Tutto questo vale anche per la sottrazione.

N.B.: il **broadcasting non è sempre possibile**.

💡 Due shape sono compatibili per il broadcasting **se sono identiche** oppure se differiscono tra di loro **per una sola dimensione** che deve essere 1 per una delle due shape.

Sono compatibili, ad esempio, le seguenti coppie di shape:

- `(4, 2, 3)` e `(4, 1, 3)`
- `(2, 1)` e `(2,)`
- `(1, 2, 3, 4)` e `(3, 2, 3, 4)`

Qualora una coppia di shape non sia compatibile con il broadcasting, Python lancerà un `ValueError`.

Ci si può far furbì e rendere una coppia di shape compatibile con una `expand_dims` 😎

Anche la moltiplicazione element-wise è possibile con Numpy tramite il metodo `np.multiply`.

Il prodotto riga per colonna tra due matrici, invece, è eseguibile con `np.matmul`.

Operazioni sugli array:

- Somma/sottrazione con `+-` grazie all'operator overloading
 - *element-wise* se la shape degli array è la stessa
 - “fittando” se la shape è diversa (*broadcasting*), ad esempio sommare uno scalare a un array
- Prodotto per uno scalare
- Moltiplicazione
 - Element-wise con `*`
 - `matmul` → con array di `dim > 2` viene applicato il broadcasting (array trattato come uno stack di matrici)
- Applicazione di una condizione con `where`, di solito si acquista in efficienza se si usa questo metodo invece di algoritmi tradizionali implementati attraverso dei **for loop**.

Importare Moduli C: `ctypes`

Spesso, non si riesce ad avere un elevato livello di efficienza perché non si trova un metodo nativo NumPy adeguato allo scenario.

Esiste la possibilità di importare delle librerie dinamiche ed eseguire delle funzioni scritte in linguaggio C.

Diversi moduli per Python permettono questa possibilità, il più semplice è `ctypes`

```
import ctypes
```

Con `ctypes`, si fa uso di librerie dinamiche, le quali corrispondono a sorgenti con estensione `.dll` su Windows e a shared libraries in ambiente GNU/Linux con estensione `.so`.

Si crea una funzione che ci **restituisce una struttura dati di tipo** `void` `custom` in C, in un file che chiamato, ad esempio, `my_library.c`. Definire gli argomenti come puntatori a `void` permette di evitare di definire delle strutture interne alla funzione, acquistando in efficienza.

```
void sum_of_arrays(const void *a, const void *b, void *result, const int len) {
    for(int i = 0; i < len; i++)
        *result[i] = *a[i] + *b[i]
}
```

Si noti che questo è solo un “esempio giocattolo” che serve solamente a far capire tutta la procedura di scrittura della libreria dinamica. Nelle slide del corso è presente un esempio più completo con allocazione dinamica di memoria e cicli for innestati, si faccia riferimento a quell'esempio per una lettura più completa.

Passando all'ambiente Python, si importa `ctypes` e si compila direttamente la libreria dinamica

```
import ctypes
from ctypes import *
import numpy as np
import os

os.system("gcc -fPIC -shared -o myLibrary.so my_library.c")
lib = cdll.LoadLibrary("./myLibrary.so")
```

In questo modo, direttamente dallo script, si è compilata la shared library e la si è caricata nell'oggetto `lib`. Per puntare alla funzione, adesso, non rimane che fare come di seguito

```
sum_of_arrays = lib.sum_of_arrays
```

N.B.: prima di invocare `my_dot_product` **occorrono degli accorgimenti**. Non si può assolutamente passare dei numpy array così come sono, ma si deve convertirli in delle rappresentazioni di dati che `ctypes` mette a disposizione

```

a = np.full((10), 3) # [3. 3. 3. 3. 3. 3. 3. 3. 3. 3.]
b = np.full((10), 7) # [7. 7. 7. 7. 7. 7. 7. 7. 7. 7.]
result = np.zeros((10)) # [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

a_p = c_void_p(a.ctypes.data)
b_p = c_void_p(b.ctypes.data)
result_p = c_void_p(result.ctypes.data)
len_p = c_int(10)

# finalmente si può chiamare la funzione, rispettando la signature
sum_of_arrays(a_p, b_p, result_p, 10)

```

OpenCV

- Libreria Open Source per Computer Vision.

Offre funzioni per:

- Gestire e manipolare immagini
- Corner Detection
- Calibrazione della camera
- Flusso Ottico
- ...

Per aprire un'immagine in OpenCV (`cv2` nei prossimi esempi)

```
img = cv2.imread('img_path')
```

Una volta aperta un'immagine, viene restituito un numpy array contenente i valori di ogni pixel per ogni canale.

Il formato di default di OpenCV è BGR. Sono forniti metodi, tra gli altri, per riportare i canali in RGB, o convertirli in grayscale (`cv2.COLOR_BGR2RGB` , `cv2.BGR2GRAY` , ...).

Convoluzione di Kernel

Per effettuare la convoluzione tra un kernel e l'immagine, è sufficiente definire il kernel e chiamare l'apposita funzione `filter2D` , passandogli l'immagine e il kernel.

```
cv2.filter2D(src, ddepth, kernel)
```

- `src` è l'immagine sorgente
- `ddepth` è la depth desiderata in output (`-1` per mantenere la stessa di `src`)

- `kernel` è il kernel da usare.

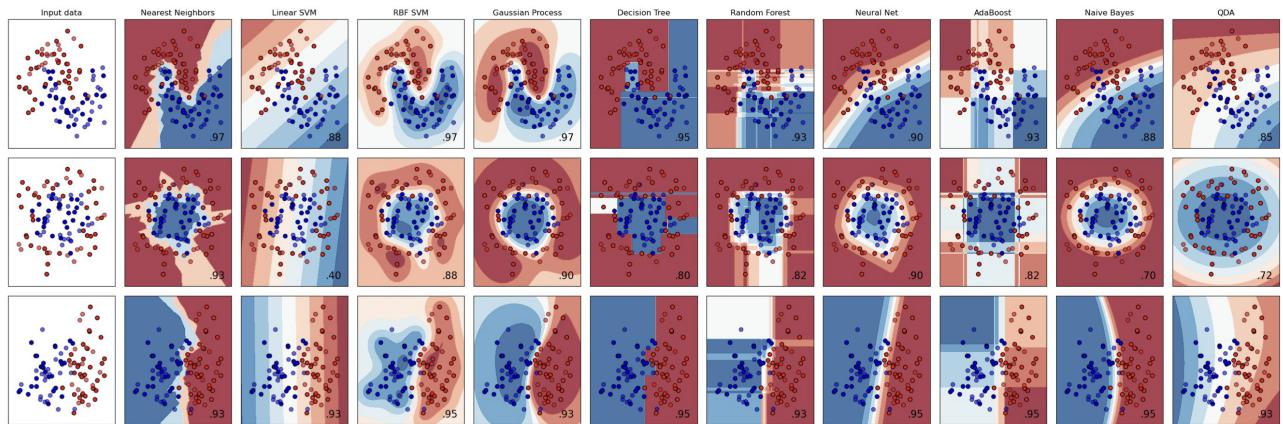
Scikit-Learn

- Open Source Machine Learning library
- Algoritmi out-of-the-box per classificazione, regressione e clustering.

Classificazione, Regressione e Clustering

Classificazione

- Goal: attribuire una categoria, label, tra un set di categorie predeterminate, a ogni input

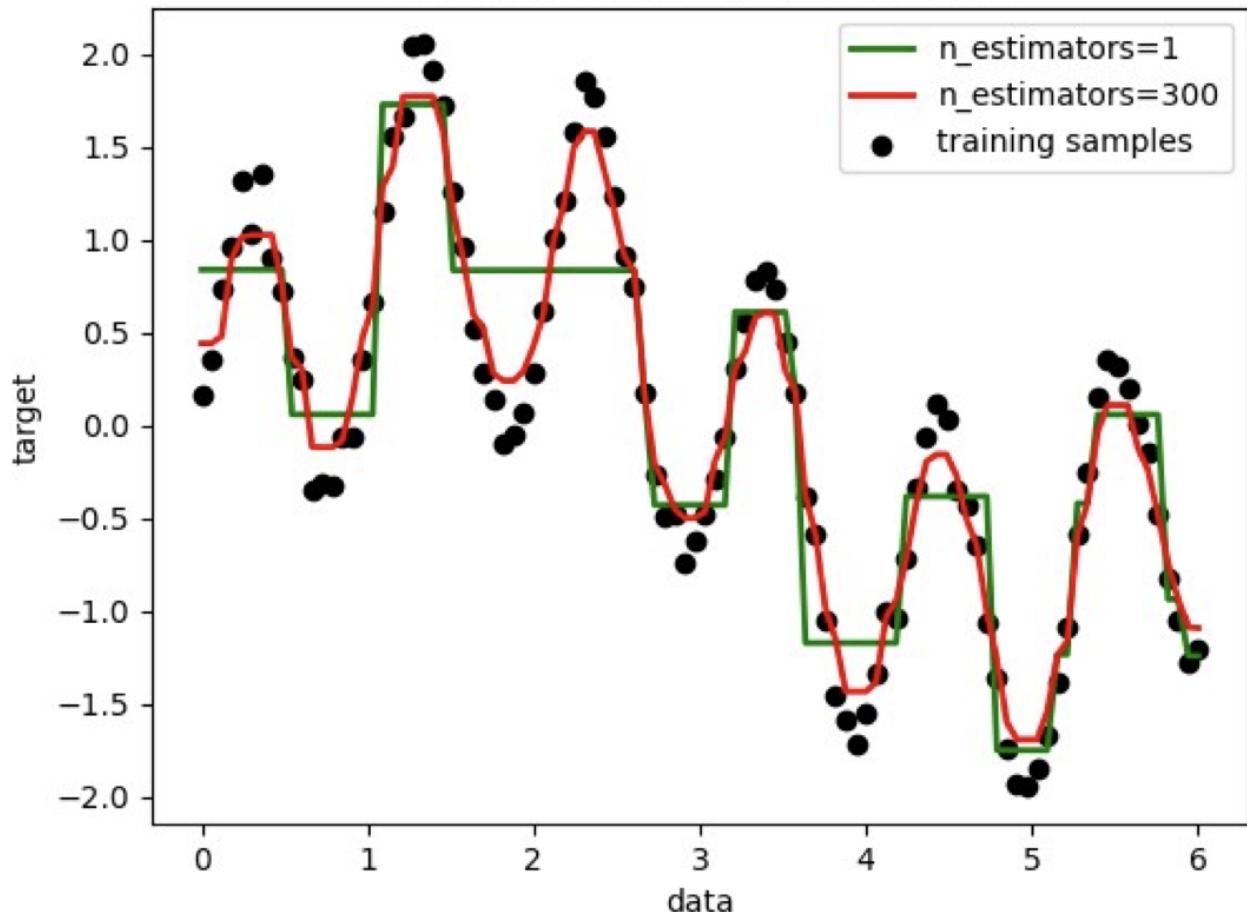


Esempio di risultato della classificazione di tre set in due classi (blu e rossa), effettuato con diversi algoritmi di Machine Learning.

Regressione

- Goal: stimare una funzione da dei dati di input

Boosted Decision Tree Regression

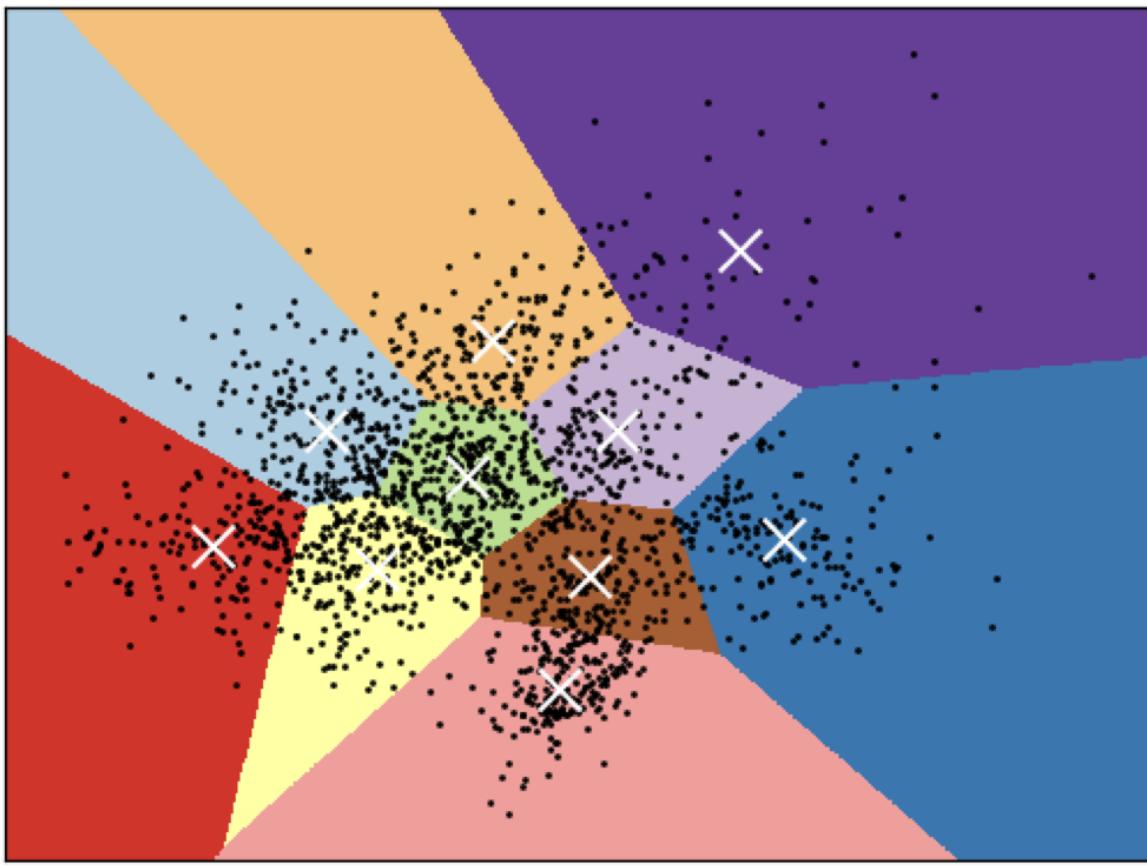


Esempio di regressione.

Clustering

- Goal: raggruppare i dati in cluster (gruppi) con proprietà comuni, senza conoscenza pregressa sui gruppi (a differenza della classificazione)
 - In alcuni definendo alcuni parametri, come il numero di cluster da trovare.

K-means clustering on the digits dataset (PCA-reduced data)
Centroids are marked with white cross



Esempio di clustering.

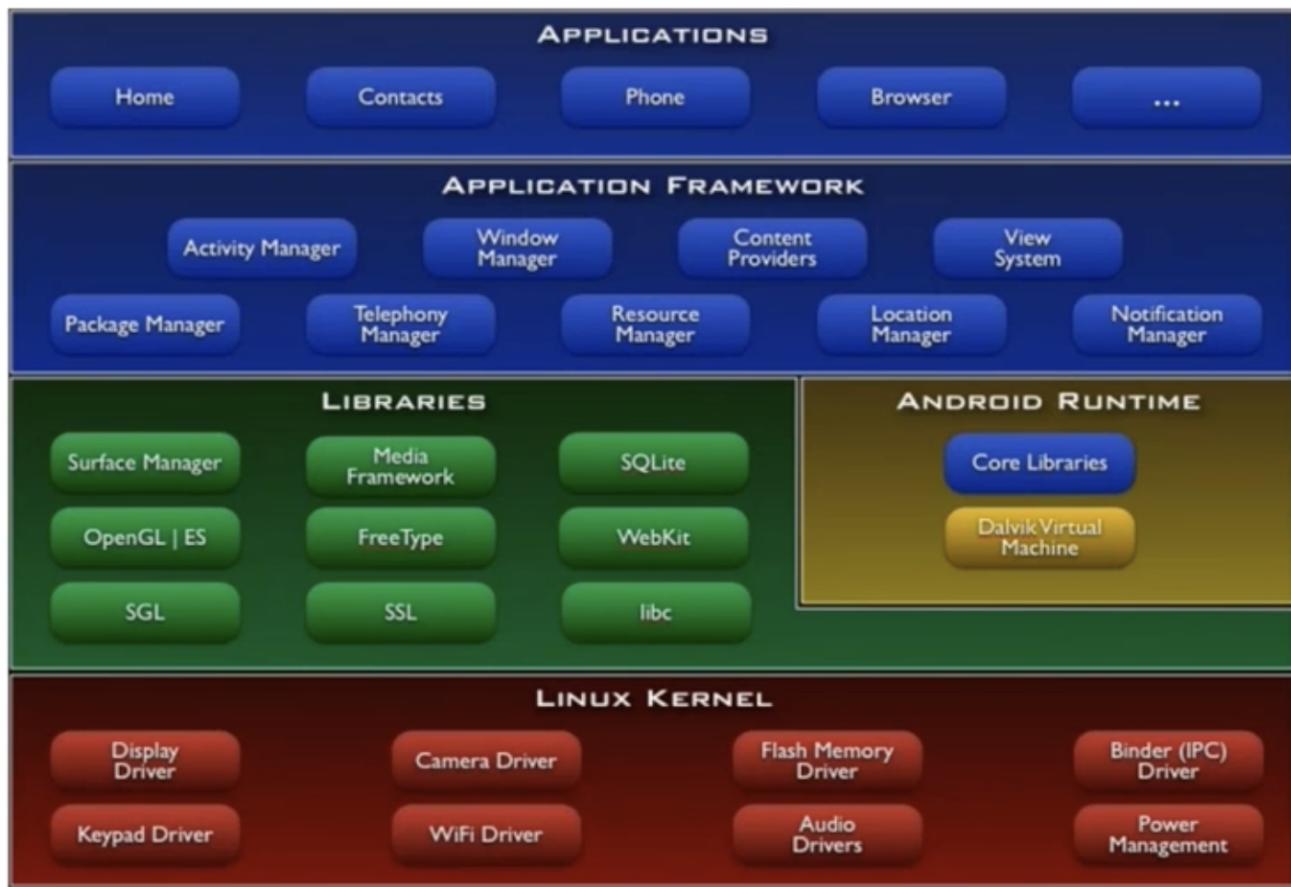
Algoritmi Disponibili

- Tree Classifiers
- Nearest-Neighbor Classifiers
- K-Means
- ...

Questi algoritmi, rispetto a quelli di Deep Learning, sono più efficienti ma, dall'altra parte, richiedono esplicito “features engineering” da parte dello sviluppatore.

Android

- *Open Source, Linux-based OS, pensato per mobile*
- Presenta uno stack (JVM-based) per sviluppare applicazioni mobile.



Architettura dello stack Android.

Stack Android

- **Applications** → livello più alto dello stack. Le applicazioni sono generalmente sviluppate in Java/Kotlin
- **Application Framework** → fornisce le classi usate per sviluppare applicazioni Android (API di vario tipo) come l'Activity Manager (per il controllo del ciclo di vita dell'applicazione) o il View System (per controllare l'interfaccia grafica utente).
- **Libraries** → Librerie che implementano la maggior parte delle Java libraries (accesso a database `android.database`, al sistema operativo `android.os`, ecc...)
- **Android Runtime** → JVM modificata pensata per il mobile, fino alla versione 5.0 Lollipop, era la Dalvik, ora la JVM si chiama Art.
 - Ogni app gira nella propria VM.

CameraX API

Interagire con la fotocamera di un dispositivo, **non è in generale un compito semplice** perché la varietà di modelli in commercio è molto vasta, si pensi che telefoni diversi hanno sensori diversi, con risoluzioni diverse, spesso anche più di una fotocamera.

CameraX è uno stack di API che ci dà una mano perché:

- è una semplice interfaccia per accedere alla camera in maniera consistente su diversi device (con diverso hardware)
- Precedentemente, Camera/Camera2 fornivano API molto più complicate, e con funzionalità device-specific in alcuni casi. CameraX è offre un'astrazione di più alto livello.

Interfacce offerte (**use cases**):

- *Preview* → mostrare a schermo una preview dell'immagine
- *Image Capture* → salvare immagini ad alta risoluzione localmente
- *Image Analysis* → eseguire algoritmi su immagini acquisite
- *Video Capture* → salvare video con audio localmente.

CameraX offre diversi meccanismi di callback per eseguire un certo comportamento al verificarsi di determinate condizioni (es. quando un'immagine viene catturata).

Chaquopy

- *Python SDK* per sistemi Android
- Completamente *integrato* con Android Studio e Gradle
- *API* per chiamare Java/Kotlin da Python e viceversa
- *Supporto* alle più popolari *librerie* Python (matplotlib, opencv, ...)

Linguaggi di Riferimento

I linguaggi di riferimento per lo sviluppo Android sono *Java* e *Kotlin*.

Entrambi i linguaggi sono basati sulla *JVM*. Entrambi sono Object-Oriented con feature di FP, ossia Functional Programming (più Kotlin che Java).

Il linguaggio di riferimento prima del 2019 è stato Java, mentre in quell'anno Google ha deciso di rendere Kotlin (che comunque mantiene la piena interoperabilità con Java) il linguaggio di riferimento.

Principali Differenze tra Java e Kotlin

Kotlin, rispetto a Java, offre una *sintassi* molto più *concisa*, con un approccio spesso dichiarativo. Ma non solo, oltre alla sintassi diversa, che è la parte più “visibile”, offre anche diverse funzionalità moderne, che sono il vero motivo che ha portato Google al cambio di rotta. Tra le più notevoli differenze ci sono:

| Feature | Java | Kotlin |

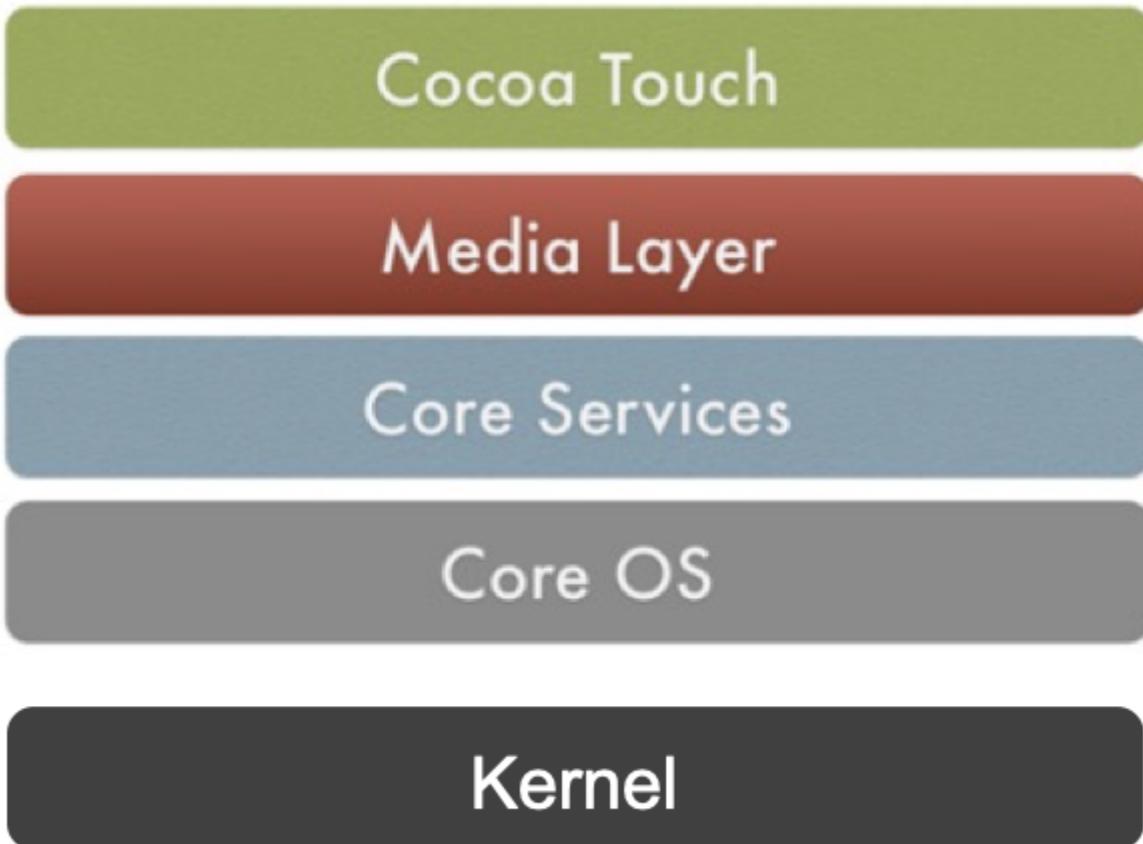
| --- | --- | --- |

Null-safety No Sì (nullable types)
Mutabilità opzionale No Sì (val vs. var
Funzioni come First-Class Citizens No Sì
Supporto alla creazione di DLS https://en.wikipedia.org/wiki/Domain-specific_language No
Sì

iOS

- *UNIX-based*, mobile-oriented Operating System
- Developed by Apple.

Architettura



Architettura di iOS.

- **Cocoa** → Application layer. Agisce da interfaccia con cui l'utente interagisce con l'OS. Fornisce alcuni framework di alto livello (EventKit, UIKit, Gamekit, ...)
- **Media Layer** → Permette di gestire le interazioni grafiche e audio del sistema. Implementa tre principali categorie di framework per grafica, audio e video

- **Core Services** → Fornisce framework base per la gestione del lifecycle
- **Core OS** → Astrazione di basso livello per gestire sicurezza, networking, ...
- **Kernel** → Implementa le funzionalità di più basso livello.

A parte il kernel (scritto in Assembly/C/C++), gli altri livelli sono perlopiù scritti in Objective-C o (più recentemente) Swift.

Sviluppo di Applicazioni

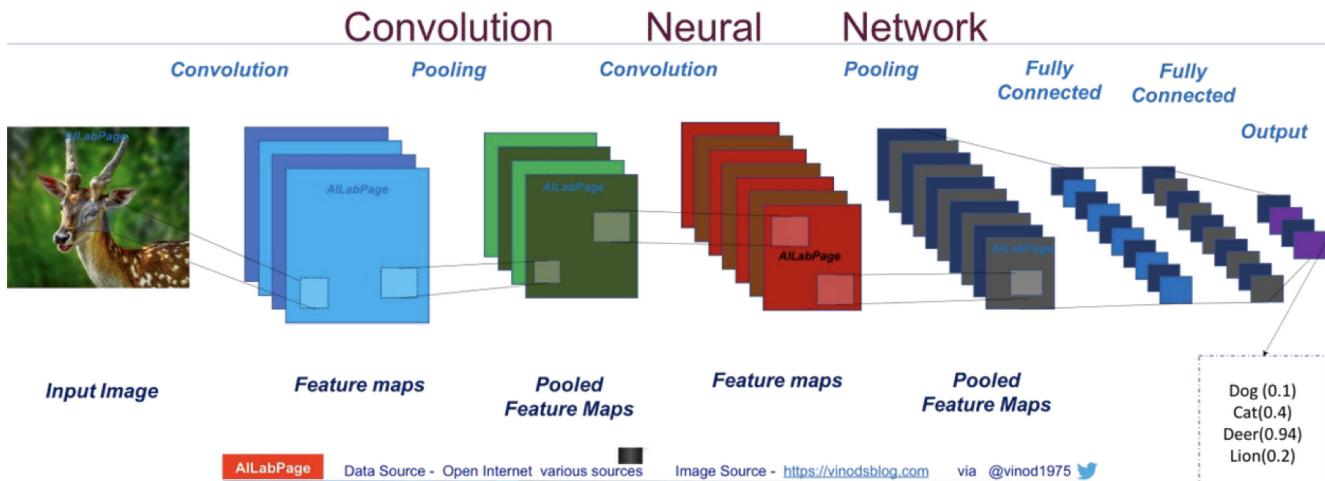
L'IDE ufficiale per lo sviluppo è Xcode, che permette di creare applicazioni per tutto l'ecosistema Apple.

Il linguaggio di riferimento per lo sviluppo è Swift. Swift usa l'LLVM Compiler e il runtime Objective-C. Non è un linguaggio garbage-collected (a differenza di Java, Kotlin, Python, ...) e utilizza invece l'*Automatic Reference Counting* (ARC) come metodo per la gestione della memoria.

Sviluppo di Applicazioni che Usano la Camera

AVFoundation è il framework che supporta lo sviluppo di applicazioni con contenuti audiovisivi in iOS. Tra le altre API, fornisce il supporto alla fotocamera con *AVCapture*.

Librerie Per il Deep Learning



Esempio di struttura di una CNN.

Le CNN risolvono problemi che si possono categorizzare in:

- Problemi di **classificazione**

- Problemi di **regressione**.

Questo è fatto in due step:

- Training - si addestra la rete a risolvere un problema
- Testing - si testa la “bravura” della rete.

Una rete è costruita come una sequenza di moduli, ognuno caratterizzato da un insieme di parametri (pesi).

Training e Testing

Training

- Fase *iterativa*
- La rete processa un’immagine e predice l’output (**forward-pass**). L’output predetto è comparato con l’output reale (label) assegnato all’immagine
- Misurando la differenza tra output predetto e label, si può quantificare l’errore
- Si ottimizza la rete per minimizzare l’errore.

L’“errore” è chiamato “differentiable loss”. Si minimizza iterativamente stimandone il gradiente e “muovendosi” in direzione opposta.

Funzione di aggiornamento dei pesi:

$$\theta_j = \theta_j - \alpha \frac{\delta}{\delta \theta_j} \mathcal{L}(\theta) \quad \forall \theta_j \in \theta$$

α è il learning rate (tasso di apprendimento), un’*iperparametro* che regola l’“intensità” del cambiamento. Un learning rate troppo elevato, porta la rete a “focalizzarsi troppo” sull’ultimo esempio che gli è stato fornito nel forward-pass, senza “memoria” di ciò che ha già appreso.

Dalla seguente formula, si può ricavare la derivata parziale di ogni peso rispetto alla loss function, e aggiornare i pesi di conseguenza, con quello che è chiamato **backward-pass**:

$$\frac{\delta f}{\delta x} = \frac{\delta f}{\delta g} \frac{\delta g}{\delta x}$$

TensorFlow

- Open Source deep learning framework sviluppato da Google
- API per Python, Java, C/C++, ...
- Due principali versioni

- 1.x → scrittura codice divisa in due fasi (creazione **computational graph**, creazione della sessione per eseguirlo)
- 2.x → **eager execution** per evitare la divisione in due fasi
- I modelli addestrati con TF possono essere portati su mobile grazie a **TF-Lite**
 - Addestrati con TensorFlow “normale” su dispositivi “potenti”
 - Portable su mobile con la versione Lite.

Perché “Tensor” Flow?

Un tensore è un array multidimensionale:

- 0-D → scalare
- 1-D → vettore
- 2-D → matrice
- 3-D → “array di matrici”.

Immagini Come Tensori

Un’immagine RGB può essere vista come un tensore 3-D → una matrice `altezza x larghezza` per ognuno dei tre canali colore.

Spesso, nei framework di deep learning le immagini vengono trattate in fase di addestramento come array 4-D:

1. Indice dell’immagine (1^a, 2^a, ...)
2. Canale R
3. Canale G
4. Canale B.

Il framework, durante l’addestramento, ad ogni iterazione non agisce su un dato (ad esempio un’immagine), ma su un batch (insieme) di `K` dati, con `K` minore del numero totale di dati in input:

- Ottimizzando su più dati in ingresso alla volta, si migliora l’ottimizzazione, perché l’ottimizzazione del backward pass sarà “più vicina” alla distribuzione delle classi.

Grafo Computazionale

La rete può essere vista come un grafo che in ogni pass viene percorso in una sola direzione (nel forward pass in avanti, nel backward all’indietro), in cui ogni nodo è uno stadio di elaborazione.

 In TensorFlow, un grafo viene definito ogni volta si scrive nel codice un'operazione tra tensori.

Il grafo:

- Specifica che operazioni eseguire ("come fluiscono i tensori")
- "Innalza" il livello dell'astrazione
- Permette di analizzare i vari stadi indipendentemente (per diagnostica, performance evaluation, ...)
- Il compilatore può usare la conoscenza del grafo per "tagliare" branch del grafo non utilizzati, risparmiare memoria, ... → può ottimizzare il grafo
- Il grafo può avere più output "specializzati" e utilizzabili selettivamente.

Il grafo, essendo un'astrazione, introduce un overhead → di questo overhead ci si può "liberare" con TF-Lite, ad esempio:

- Togliendo funzionalità del grafo usate solo in addestramento, ma non utili durante l'esecuzione lato mobile.

N.B.: In TF 1.x il grafo è immutabile.

Tipi di Tensore

- Variable - tensore il cui valore può cambiare compiendo operazioni su di esso
 - I pesi della rete sono organizzati in tensori variabili
- Constant - tensore costante
- Placeholders (1.x only) - "segnaposti" utilizzati in TF 1.x per gestire le due fasi.

I tensori hanno un rango (rank), ossia il numero di dimensioni.

| Rank | Entità | Forma | Esempio |

| --- | --- | --- | --- |

| 0 | Scalare | [] | Numero |

| 1 | Vettore | [N₀] | Lista di numeri |

| 2 | Matrice | [N₀, N₁] | Matrice |

| 3 | Tensore 3-D | [N₀, N₁, N₂] | Immagine |

| 4 | Tensore 4-D | [N₀, N₁, N₂, N₃] | Batch di immagini |

Ad esempio, per un batch di immagini:

- N₀ - numero di immagini
- N₁ - altezza
- N₂ - larghezza
- N₃ - numero di canali.

```

# TF 1.x
# Aggiunge al grafo un tensore scalare di valore 5, di nome x e tipo float32
x = tf.Variable(5, dtype=tf.float32, name='x')
# Definisce un nuovo nodo, un tensore, che nella fase di esecuzione conterrà
# il rango di x
rank_x = tf.rank(x)

```

Esecuzione

In TF 1.x, per eseguire un grafo, bisogna creare una sessione (`Session`) e vincolare il grafo alla sessione appena creata. Dopodiché, si demanda alla sessione di eseguire il grafo.

In TF 1.x si può creare una `Session` all'interno di uno statement `with`, che si occuperà di gestire il ciclo di vita della sessione (apertura, chiusura, rilascio risorse).

All'avvio di una `Session` TF 1.x, è necessario inizializzare (metodo `run` della sessione) le variabili locali e globali, dopodiché si può eseguire `run` con qualsiasi nodo del grafo.

In TF 2.x è stato eliminato il bisogno delle `Session`. Chiamando operazioni sulle variabili, queste vengono direttamente processate grazie alle Eager Execution.

Per riassumere le differenze:

- TF 1.x → 2 fasi
 - Creazione grafo → statico
 - Esecuzione grafo (con `Session`)
 - è una versione mantenuta ormai solo per questioni di **legacy**
- TF 2.x → 1 sola fase
 - Esecuzione senza sessione esplicita.

Keras

- **Front-end** di “*alto livello*” che viene con TensorFlow
- Funzioni per implementare molti componenti usati nelle NN
- Compatibile con più back-end.

Creazione di un modello sequenziale con `keras.Sequential`

```

from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential{
    [

```

```
keras.Input(shape=input_shape)

# 32 filtri con kernel 3x3 e funzione di attivazione ReLU
layers.Conv2D(32, kernel_size=(3,3),activation="relu"),

layers.MaxPooling2D(pool_size=(2, 2)),

# Altri livelli...

layers.Flatten(),
layers.Dropout(0.5),
layers.Dense(num_classes, activation="softmax"),
]

}
```

Addestramento del modello tramite `fit`

```
batch_size = 128
epochs = 15

model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])

model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
validation_split=0.1)
```

Valutazione del modello tramite la `evaluate`

```
score = model.evaluate(x_test, y_test, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

PyTorch

Si tratta di un framework attualmente supportato da **Meta AI**, proposto nel 2016 in sostituzione a Torch. PyTorch ha esteso il suo predecessore fornendo il supporto a Python, a differenza di Torch che utilizzava il linguaggio di scripting Lua. PyTorch è installabile via `pip`.

Supporta la creazione di un grafo dinamico, anticipando quello che poi sarebbe diventato TensorFlow 2.x.

Anche PyTorch supporta una libreria per dispositivi mobili chiamata `torchscript`

Il grafo computazionale di PyTorch è del tutto analogo a quello di TensorFlow; consente quindi di seguire il flusso dei dati nelle due direzioni di forward step e back-propagation.

Anche qui, la struttura dati fondamentale è il `Tensor`

```
a = torch.Tensor([3])
b = torch.Tensor([4])

print(a + b) # Tensor([7.])

a.shape # Torch.Size([1])
```

A differenza di TensorFlow, **non** si ha a disposizione un frontend come Keras, ma sono predisposti dei metodi di `torch.nn` per realizzare i vari layers

```
nn.Conv2D(in_channels, out_channels, kernel_size)
nn.AvgPool2D(kernels_size)
nn.MaxPool2D(kernels_size)
nn.Linear(in_features, out_features)
```

Nonostante l'assenza di un frontend di alto livello, PyTorch permette di utilizzare le sue API ad un livello più alto rispetto ai metodi nativi di TensorFlow senza Keras. Ad esempio, `nn.Conv2D` non chiede all'utente di inizializzare o gestire i pesi dei kernel.

Per incapsulare una rete PyTorch si estende l'oggetto `Module`

```
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Layer di Normalizzazione

Ciò che non è mostrato da Keras, da PyTorch viene palesato. Alcuni livelli si comportano diversamente in base alla fase in cui ci si trova (training, testing, evaluating, prediction); per esempio, alcuni layers che si occupano di normalizzare l'output del layer precedente (per far ricadere tutti i risultati all'interno di una distribuzione di probabilità omogenea) accumulano delle statistiche durante la fase di training, li aggiustano e li mantengono così quando il training è finito.

Grazie a questi livelli è possibile chiamare i metodi `train()` ed `eval()` sullo stesso modello.

Mobile Deep Learning

Nell'ambito dei dispositivi mobili, non vi è interesse a cercare di avere efficienza in fase di training, dato che di solitamente viene eseguito su hardware molto performante (non su mobile). L'interesse maggiore piuttosto è che la rete neurale, una volta caricata, sia efficiente in fase di inferenza

TensorFlow Lite

Mette a disposizione un sottoinsieme delle funzionalità di TensorFlow, con una serie di tool di ottimizzazione per:

1. **Convertire un modello** TensorFlow in un formato equivalente ma ottimizzato per mobile
2. **Quantizzare** il modello per renderlo più efficiente e veloce, **minimizzando** la perdita di accuratezza

Per eseguire il porting del modello su mobile, la prima fase è la conversione della rete TF nel formato `.tflite`

```
import tensorflow as tf

converter = tf.lite.Converter.from_saved_model(saved_model_dir)
tflite_model = converter.convert()
```

L'oggetto in `tflite_model` può essere serializzato su disco con una scrittura su file

```
with open('my_model.tflite', 'wb') as f:
    f.write(tflite_model)
```

Il modello può ora essere importato su Android Studio da `New > Other > Tensorflow Lite Model` e, se il modello presenta dei metadati, è possibile avere la signature dei metodi di inferenza del modello.

PyTorch Mobile

PyTorch Mobile consente di ottimizzare per mobile un modello, rimanendo nell'ecosistema di PyTorch.

- Il primo step consiste nel convertire un modello in uno ottimizzato per mobile (`mobile_optimizer` → classe che implementa funzioni che mirano a ridurre la complessità della rete)

- Collassare più operazioni in una sola, ...
- Si salva il modello ottimizzato in un formato compatibile con il Lite Interpreter (`save_for_lite_interpreter`).

Oltre agli step descritti, è possibile opzionalmente (e consigliato) applicare ottimizzazioni quali la quantizzazione dei pesi.

Quantizzazione dei Pesi

L'ottimizzazione del modello consiste, in genere, nel convertire determinate operazioni per renderle più efficienti nella piattaforma destinataria, in questo caso un dispositivo mobile.

Una di queste ottimizzazioni è la **quantizzazione dei pesi** (ottimizzazione general-purpose) del modello, cioè ridurre la precisione di un tipo di dato.

 Si supponga di rappresentare un insieme di dati con 32 bit, con cui si ha la possibilità di esprimere 2^{32} valori, (ad esempio) nel range $\sim [4 * 10^9]$. Quantizzando questo insieme ad 8 bit, si potrebbe comunque rappresentare lo stesso **range di valori**, ma con una **granularità notevolmente inferiore**: si rappresenterebbe lo stesso range ($\sim [0, 4 * 10^9]$), ma muovendosi a “step” di N valori.

Vantaggi:

- L'occupazione di memoria della rete stessa viene diminuita di un fattore 2, 4, ecc...
- Alcune operazioni, come le fixed point, risultano molto più veloci di altre, come le floating point

Svantaggi:

- La precisione delle operazioni e quindi l'**accuratezza** finale potrebbe risentirne.

La quantizzazione viene supportata sia da TensorFlow Lite che da PyTorch. Ovviamente, visto che sono due framework di porting su mobile di modelli già addestrati e convertiti, la quantizzazione viene utilizzata **solo nel forward step**.

In particolare questa ottimizzazione viene applicata:

- Ai parametri della rete (i.e. i pesi)
- Alla rappresentazione dell'immagine che questa assume via via che attraversa gli strati della rete.

 Un modello quantizzato risulta più veloce della sua controparte solo se supporto hardware appropriato è fornito sul dispositivo target.

Quantizzare un modello in `float16` non rende la rete più veloce se l'hardware del dispositivo non ha supporto adeguato alle operazioni tra `float16`.

TensorFlow Lite offre tre tecniche di **quantizzazione post-addestramento**:

- **Dynamic range quantization**: è la più semplice, quantizza i pesi da `float32` a `int8`. A tempo di inferenza i pesi vengono opportunamente riconvertiti in `float32`. Alcuni operatori (purtroppo non tutti) detti "dinamici" possono essere quantizzati dinamicamente in base ad un range dinamico ed operare ad `int8`
- **Full Integer Quantization**: tutte le operazioni vengono convertite a `int8`. Per fare ciò, occorre calibrare il range $[min, max]$ di tutti quei tensori del modello che sono già statici (come i pesi). La stessa cosa purtroppo non si può dire di alcuni tensori, che sono intrinsecamente dinamici, come quelli dei risultati dell'inferenza, a meno che non si disponga di quello che viene detto un dataset rappresentativo.
- **Float16 quantization**: come dice il nome, viene tutto quantizzato a `float16` con un'accelerazione di 2x nella velocità di inferenza. Questa tecnica si può utilizzare **solo** quando l'hardware ci fornisce supporto alle operazioni in `float16`

Le tecniche di quantizzazione supportate da **PyTorch** sono:

- *Dynamic quantization*: i pesi vengono memorizzati in modo quantizzato e il loro livello di quantizzazione viene modificato all'occorrenza in fase di inferenza.
- *Static quantization*: una quantizzazione piuttosto tradizionale, ma con la possibilità di "fondere" le attivazioni nei livelli precedenti. Ciò ha bisogno di una rappresentazione chiara del dataset per calibrare quanto e come fondere le attivazioni. L'equivalente della Full integer quantization di TF-Lite.
- *Quantization aware training (QAT)*: in questa interessante tecnica, la quantizzazione viene decisa in fase di training, viene cioè tentata la simulazione a `int8` di alcune operazioni durante la fase di addestramento, che rimane comunque in `float32`.

Fin!