Sistemi di Elaborazione Accelerata

Ingegneria Informatica M

Matteo Fontolan

2025/2026

Disclaimer & Info

Questo documento é una raccolta **personale** basata sulle lezioni e/o sui materiali didattici forniti dai docenti e/o raccolti negli anni dagli studenti.

> ❗ **Disclaimer**
>
> *L'autore non si assume alcuna responsabilità per eventuali errori, omissioni o inesattezze contenuti in questo documento. Il materiale è fornito «così com'è» a solo scopo di supporto allo studio.*

Se vuoi **aggiungere** materiale utile o **segnalarmi** errori o correzioni, fallo qui.

Spero che queste risorse ti siano utili.
Buono studio e in bocca al lupo! 🤞

Se ti va di offrirmi una cioccolata🍫 puoi farlo qui.

Bento

Indice

1. Modulo 1 - SIMD **6**
1.1. Introduzione 6
1.1.1. Pipelining 6
1.1.2. Paradigma SIMD 6
1.1.3. x86 7
1.1.4. ARM 8
1.1.5. RISC-V 8
1.1.6. Dati e Memoria 8
1.1.7. Contatiri di performance 11
1.1.8. Autovettorializzazione del codice 11
1.1.9. Dati e predisposizione al calcolo vettoriale 11
1.1.10. Istruzioni logiche, shift, shuffling 12
1.1.11. Packing/Unpacking 14
1.1.12. cvt (Passaggio tra tipi di dato) 15
1.1.13. Blending 16
1.1.14. Copia con inserimento di un valore 17
1.1.15. Confronto 17
1.1.16. Maschere per Assegnamenti Condizionati 19
1.1.17. Minimi e Massimi 22
1.1.18. Istruzioni Aritmetiche con Byte 23
1.1.19. ABS 27
1.1.20. Add e Sub orizzontali 27
1.1.21. SAD (Sum of Absolute Differences) 28
1.1.22. Somme varie (esempi) 29
1.1.23. Media (abg) e Cambio di Segno 33
1.1.24. Moltiplicazioni 34
1.2. Memorie e Cache 38
1.2.1. Introduzione 38
1.2.2. SRAM vs DRAM 38
1.2.3. Layout di una memoria DRAM 40
1.2.4. Cache 43
1.2.5. Casi di studio relative all'uso (efficiente) di cache 50
1.3. Principi di acquisizione di immagini e 3D sensing 59
1.4. Numeri reali (Floating e Fixed Point e nuovi tipi) 73
1.5. FPGA (Field Programmable Gate Array) 112

2. Modulo 2 - CUDA **124**
2.0 Introduzione 124
2.0.1. Un po' di storia... 124
2.0.2. GPU 125
2.0.3. CPU vs GPU 126
2.0.4. Architetture eterogenee 127
2.0.5. CUDA 128
2.1. Modello di programmazione CUDA 131
2.1.1. Livelli di astrazione 132
2.1.2. Gestione della Memoria (accenni) 133
2.1.3. Organizzazione dei Thread 136
2.1.4. Lancio di kernel 138
2.1.5. Mapping e dimensionamento 138
2.1.6. Analisi delle prestazioni 140
2.2. Modello di Esecuzione CUDA 143

2.2.1. Architettura Hardware GPU ... 143
2.2.2. Organizzazione e Gestione dei Thread ... 148
2.2.3. Modello di Esecuzione SIMT e Warp ... 149
2.2.4. Sincronizzazione e Comunicazione ... 157
2.2.5. Ottimizzazione delle risorse ... 159
2.2.6. Parallelismo avanzato ... 161
2.3. Modello di Memoria CUDA ... 163
2.3.1. Modelli di Performance ... 163
2.3.2. Gerarchia di Memoria CUDA ... 166
2.3.3. Gestione della Memoria Host-Device ... 172
2.4. Global Memory ... 176
2.4.1. Shared Memory (SMEM) ... 182
2.5. Librerie CUDA ed Applicazioni ... 184
2.5.1. Introduzione alle Librerie CUDA ... 184
2.5.2. Utilizzo delle Librerie CUDA ... 188
2.5.3. Utilizzo delle Librerie CUDA ... 190
2.5.4. Applicazioni ... 198

3. Riferimenti ... 204

1. Modulo 1 - SIMD

1.1. Introduzione

Il paradigma di elaborazione storico tipico di un microprocessore é single core (scalare). Questo prevede l'esecuzione sequenziale di un'istruzione e l'elaborazione dei dati previsti dall'istruzione stessa (un'istruzione per clock, a meno di letture/scritture). Tale paradigma può essere definito **SISD** (Single Instruction, Single Data). Alcuni processori, definiti **superscalari**, sono in grado di eseguire contemporaneamente più istruzioni.

L'architettura "storica" si definisce scalare in quanto ogni operazione avviene su dati scalari (nel senso matematico), infatti, ad esempio, l'operazione `ADD` *somma due numeri, scalari, tra loro, ponendo il risultato in un registro.*

1.1.1. Pipelining

Pipelining

Il **pipelining** é una strategia per **aumentare il throughput** mantenendo inalterata la latenza. In particolare prevede l' esecuzione contemporanea di **più istruzioni** in **fasi differenti** di elaborazione. Il numero di istruzioni dipende dalla lunghezza della pipeline (ossia dal numero di "fasi" che può ospitare contemporaneamente).

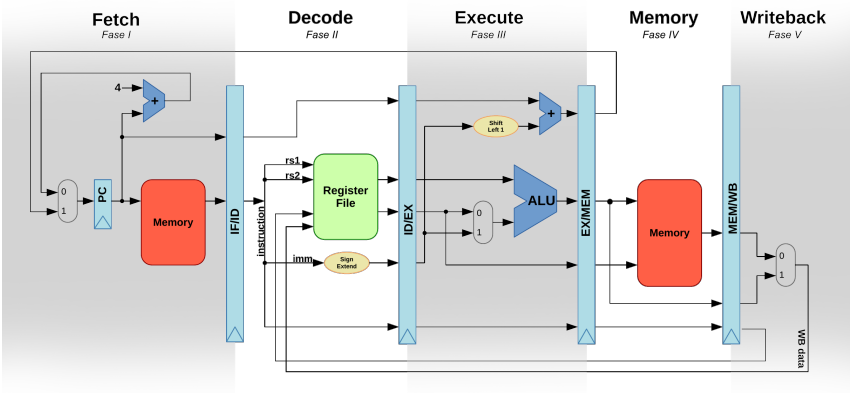


Figura 1: Pipeline con 5 fasi (RISC-V).

Il pipelining però genera problematiche nuove rispetto all'esecuzione sequenziale, comunque gestibili tramite opportune soluzioni (come le dipendenze tra dati elaborati in fasi diverse, cui si può porre rimedio tramite forwarding).
Questa strategia necessita della cache per evitare di stallare la pipeline.

Esistono alternative al pipelining, come l'**Out Of Order Execution**.

1.1.2. Paradigma SIMD

SIMD

Il paradigma **SIMD** (**Single Instruction, Multiple Data**), prevede che una **singola istruzione** elabori **dati multipli** (⇒ stessa operazione su dati diversi) memorizzati in **registri speciali/estesi** residenti nella CPU. In questo caso si tratta di registri e valori vettoriali ("array di scalari").

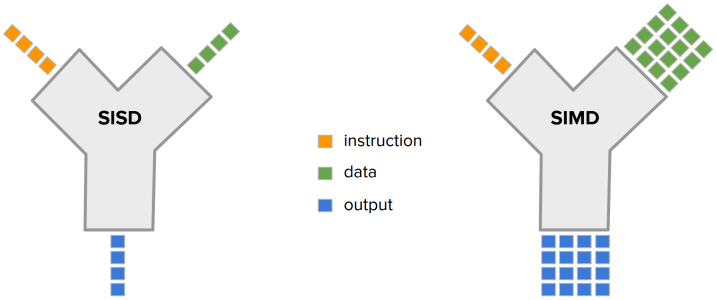
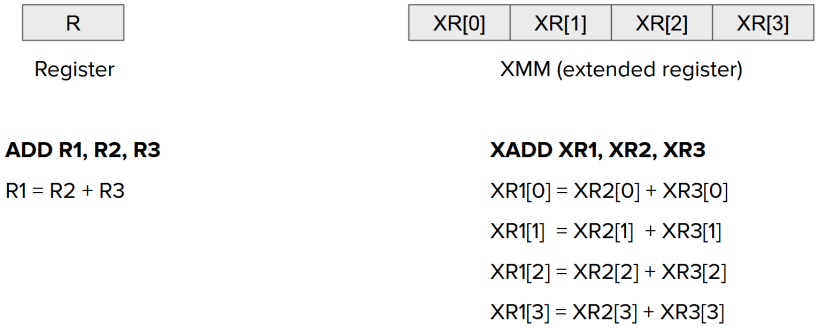


Figura 2: SISD vs SIMD.

Considerando una somma nel caso SISD vs SIMD, nel caso SIMD si eseguono più somme contemporaneamente (per l'equivalente in SISD si dovrebbero fare `len(XR)` `ADD`).



How SIMD?: Il trucco dell'esecuzione contemporanea di operazioni sul contenuto dei registri estesi sta nell'avere a disposizione fisicamente hardware replicato in grado di operare su vettori (quindi ad esempio N ALU).

PRO:
- **Parallelismo** incrementato.
- Utilizzabile con strategie per aumentare il throughput (come il pipelining).
- Modesto numero di modifiche hardware (rispetto a SISD).
- Supportato da quasi tutte le ISA maggiori (x86, ARM, RISC-V), ma ciascuna ha spesso set differenti (SIMD).

Nota: Le ISA stesse tipicamente supportano diversi set di istruzioni SIMD con funzionalità simili, in quanto la tendenza é di supportare tutte le estensoni precedenti (per compatibilità). Talvolta comunque alcuni set sono deprecati.

CONTRO:
- Poco flessibile.

1.1.3. x86

Il paradigma SIMD é stato introdotto negli anni '60, ma ha iniziato a diffondersi attorno agli anni '90 (nell'architettura **x86**) con la crescente richiesta di operazioni intensive su dati multimediali (audio, video, immagini).
Negli anni si sono susseguite diverse estensioni, aumentando via via la dimensione dei registri (da 64b a 256/512b) e i tipi di dato supportati.

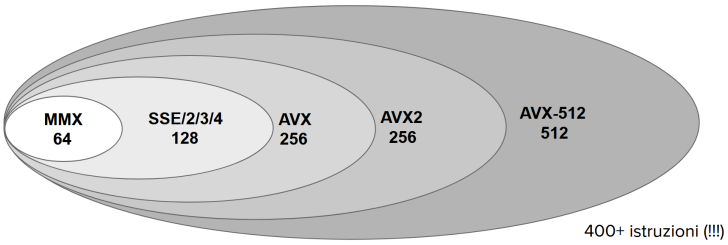


Figura 4: Estensioni Intel x86 evolutesi nel tempo (1996-2016).

I tipi di dato oggi supportati dall'architettura x86 sono **byte** (8 bit - unsigned e signed), **interi** a 16 bit, interi a 32 bit, **single precision floating-point** (32 bit), **double precision floating-point** (64 bit).

👉 ***Nota***: Non tutte le istruzioni o estensioni supportano tutti i tipi di dato, non c'é molta ortogonalità nell'ISA x86.

1.1.4. ARM

Anche l'architettura **ARM** supporta operazioni SIMD, tramite set di istruzioni NEON. Inizialmente i registri (estesi) erano da 32b o 64b, oggi da 128b.
Sono supportati interi a 8 bit, 16 bit, 32 bit e 64 bit signed e unsigned e floating-point a 16 e 32 bit.

Esistono estensioni per HPC (SVE e SVE2) con supporto a dati di dimensioni variabili (128-2048b con step di 128b).

1.1.5. RISC-V

L'ISA RISC-V prevede un'estensione SIMD, denominata RISC-V Vector Extension (RVV), in cui sono definiti 32 registri di dimensioni pari ad una potenza di 2 non prefissata (configurabile dai progettisti della CPU).
I dati possono essere di taglia pari a una potenza di 2, con valore ≥ 8.

1.1.6. Dati e Memoria

Abbiamo detto che le estensioni e le operazioni SIMD supportate non sono uniformi tra loro, tuttavia sono generalmente simili a livello di implementazione.

Le operazioni SIMD sono disponibili come istruzioni in linguaggio **assembly**. É possibile scrivere codice con linguaggi di alto livello, inserendo porzioni SIMD in codice assemply. Tuttavia, esiste un'alternativa: l'uso di **intrinsics**.

> 📖 **Intrinsics**
>
> Gli **intrinsics** sono **funzioni online** che corrispondono a singole o sequenze di **istruzioni SIMD** in linguaggio **assembly**.

👉 ***Nota***: Il vantaggio é codice più leggibile e semplice da scrivere. Tuttavia si perde il controllo del dell'uso diretto delle istruzioni assembli (in alcune circostanze).

Da ora e nel seguito si farà riferimento principalmente ad SSE (ed evoluzioni).

SSE sta per Streaming SIMD Extensions. È un set di istruzioni introdotto da Intel nel 1999 che estende l'architettura x86 per supportare il paradigma SIMD.

⚠️ Tipi di dato in SSE

In SSE i **registri** (chiamati **XMM**) sono da **128b** e sono supportati i seguenti tipi di dati:

- `__m128i`, interi
  - 16x8 bit – **epi8** (signed), **epu8** (unsigned).
  - 8x16 bit – **epi16** (signed), **epu16** (unsigned).
  - 4x32 bit – **epi32** (signed), **epu32** (unsigned).
  - 2x64 bit – **epi64** (signed).
- `__m128`, float
  - 4x32 bit – tipicamente denominati **ps** (floating-point a singola precisione).
- `__m128d`, double
  - 2x64 bit – tipicamente denominati **pd** (floating-point a doppia precisione)

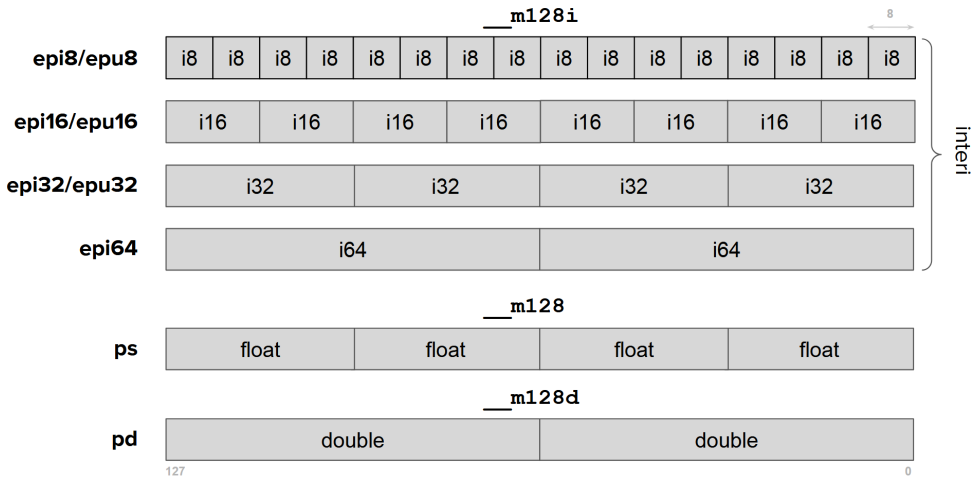


Figura 5: Tipi di dato SSE e registri (128b).

Come si può vedere in Figura 5 il numero di elementi (sempre dello stesso tipo) contenuti in un registro corrisponde esattamente a $\frac{\text{Dimensione registro (128b)}}{\text{Dimensione tipo}}$).

Le transazioni di **lettura** verso la memoria sonno supportate in dimensione pari a quella del registro (esteso), ossia $\mathbf{128b = 16B}$. La firma dipende dal tipo di dato da leggere:

```
__m128i _mm_load_si128 (__m128i const* mem_addr)    // interi (32b o 16b) e byte
__m128 _mm_load_ps (float const* mem_addr)          // float sp
__m128d _mm_load_pd (double const* mem_addr)        // float dp
```

Prende in ingresso il registro contenente l'indirizzo di memoria e restituisce il risultato del tipo opportuno.

Le transazioni di **scrittura** in memoria verso posizioni contigue, analogamente hanno dimensione pari **sempre a** $\mathbf{128b = 16B}$. Le firme similmente sono:

```
void _mm_store_si128 (__m128i* mem_addr, __m128i a) // interi (32b o 18b) e byte
void _mm_store_ps (float* mem_addr, __m128 a)       // float sp
void _mm_store_pd (double* mem_addr, __m128d a)     // float dp
```

Prende in ingresso il registro contenente l'indirizzo di memoria e il registro contenente i dati da scrivere (del tipo opportuno).

👉 ***Nota***: Le firme sono del tutto analoghe, cambia unicamente il tipo di dato (a parità di operazione) e il nome dell'operazione (a partià di tipo di dato).

⚠️ Trasferimenti da/verso memoria

Ogni **trasferimento** di dati tra la memoria e i registri estesi (o viceversa) avviene **attraverso** delle memorie **cache**.
Le cache sono organizzate in **linee** (di dimensione pari ad una potenza di 2, dipende dall'hardware). Ogni linea contiene una **porzione di memoria** del sistema. Per questo motivo é fondamentale leggere e scrivere ad **indirizzi allineati**.

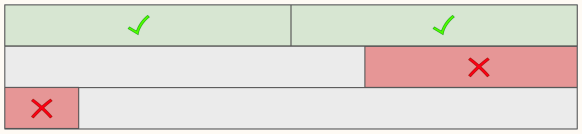


Figura 6: Cache e lines - Transazioni allineate ✅ e non ❌.

Pertanto, i dati vanno memorizzati in memoria ad indirizzi allineati con la taglia dei registri estesi e sarebbe buona cosa fare lo stesso per le scritture.

👉 ***Nota***: Alcune estensioni SIMD non consentono accessi disallineati, tuttavia, anche in questi casi, si potrebbe incorrere in penalizzazioni (maggior tempo di esecuzione). In altri casi esistono istruzioni per accedere a dati non allineati (come `__m128i _mm_loadu_si128 (__m128i const* mem_addr)` e `void _mm_storeu_si128 (__m128i* mem_addr, __m128i a)`).

📝 Esempio: C++

In C++ ad esempio é possibile forzare l'allineamento ad indirizzi multipli (per esempio) di 16B con `__attribute__` e `aligned()`:

```
int A[8] __attribute__((aligned(16))); // 16 byte (128 bit) aligned
```

In SSE sono presenti anche funzioni specifiche per **allocare** e **deallocare** dinamicamente memoria, specificando il tipo di allineamento desiderato.

```
void* _mm_malloc (size_t size, size_t align)
void _mm_free (void * mem_addr)
```

SSE permette anche di **invalidare** in tutti i livelli di una cache la **linea contenente un undirizzo** di memoria (potrebbe avere un impatto molto negativo) con:

```
void _mm_clflush (void const* p)
```

É anche possibile suggerire (non essendo una vera istruzione) al compilatore di eseguire un **prefetch** di dati nella cache in accordo a determinate strategie parametro `i=0, 1, 2, 3`):
`i` *indica alla CPU in quale livello della Cache mettere i dati e per quanto tempo si prevede che rimangano utili.*

```
void _mm_prefetch (char const* p, int i)
```

Oltre a quanto visto, sono disponibili delle istruzioni (stream) che eseguono operazioni con la memoria **senza modificare** il contenuto delle **cache** (a meno che il dato non sia già presente, a quel punto aggiorna il valore in cache). Un esempio é:

```
__m128i _mm_stream_load_si128 (void* mem_addr)
```

Carica 128 bit di dati (interi) senza modificare il contenuto della cache, a meno che il dato non sia già presente.

👉 ***Nota***: L'istruzione richiede che i dati siano allineati con la taglia del registro (16B), altrimenti lancia un'eccezione.

1.1.7. Contatiri di performance

Considerando sistemi **multitasking**, il tempo di elaborazione é soggetto a fluttuazioni dovute alla gestione/schedulazione variabile dei task. Nei sistemi moderni esistono **contatori di clock CPU** che si aggiornano unicamente quando il processo é in esecuzione, permettendo quindi il calcolo dei tempi in maniera più precisa (in base al clock/frequenza del processore).

In x86, un contatore a 64 bit é accessibile mediante

```
unsigned __int64 __rdtsc();
```

1.1.8. Autovettorializzazione del codice

I compilatori moderni possono generare un codice eseguibile che sfrutta il paradigma SIMD e altre forme di ottimizzazione identificando flussi di elaborazione parallela.
Per esempio con il comando:

```
g++ -O3 -msse2 -ftree-vectorize code.cpp -o code_ex
```

Il grado di ottimizzazione con `g++` (o `gcc`) si può impostare con il flag `-O` seguito da un numero (0-3). A cui si può aggiungere `-msse2 -ftree-vectorize -ftree-loop-linear`.

1.1.9. Dati e predisposizione al calcolo vettoriale

I dati da elaborare non sono in origine sempre memorizzati in modo tale da rendere efficiente l'utilizzo di operazioni parallele tra registri estesi. A causa di ciò tutte le estensioni SIMD prevedono la possibilità di manipolare direttamente i dati per renderli elaborabili parallelamente.

Il modo più semplice (ma poco efficiente) é inserire manualmente i dati in un registro esteso, ad esempio:

```
__m128i _mm_set_epi32 (int e0, int e1, int e2, int e3)
```

Inserisce nel registro destinazione a 128 bit i quattro valori interi a 32 bit e0, e1, e2, e3.

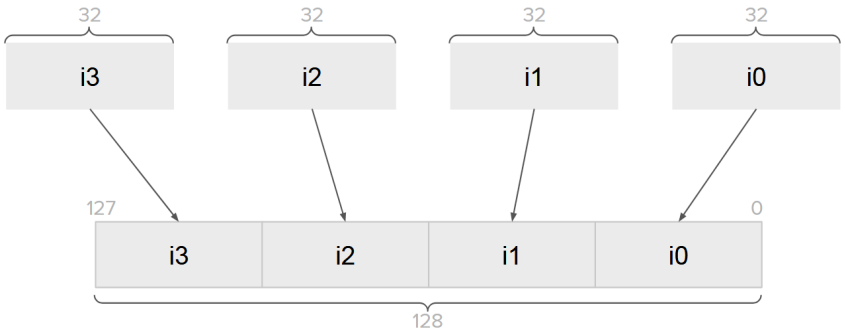


Figura 7: Inserimento manuale dei valori nel registro esteso (l'ordine non é casuale).

👉 ***Nota***: Esistono equivalenti istruzioni per gli altri tipi di dato (supportati dall'ISA x86).

É possibile, da un registro esteso, estrarre anche un singolo valore, per mezzo di un valore immediato presente nell'istruzione. Per esempio (per interi e byte):

```
int _mm_extract_epi32 (__m128i a, const int imm8)
```

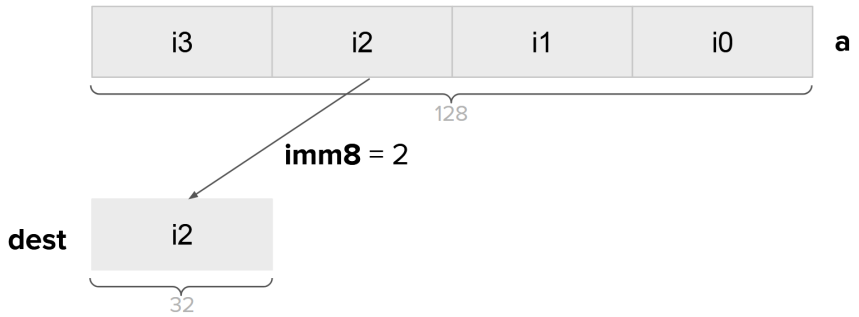


Figura 8: Estrazione di un valore da un registro esteso.

👉 ***Nota*** : Esistono equivalenti istruzioni per gli altri tipi di dato (supportati dall'ISA x86).

1.1.10. Istruzioni logiche, shift, shuffling

Le istruzioni SIMD supportano operazione logiche bit a bit (bitwise) **AND**, **OR**, **XOR**:

```
__m128i _mm_and_si128 (__m128i a, __m128i b)
__m128i _mm_or_si128 (__m128i a, __m128i b)
__m128i _mm_xor_si128 (__m128i a, __m128i b)
```

👉 ***Nota*** : Esistono equivalenti istruzioni per gli altri tipi di dato (supportati dall'ISA x86).

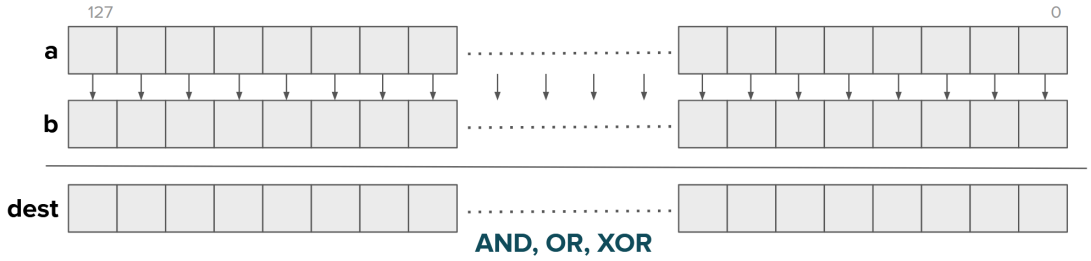


Figura 9: Operazioni logiche bitwise.

Nelle operazioni logiche bitwise, viene eseguita l'operazione scelta tra il bit `i` *-esimo dei due elementi e il risultato é posto nell'* `i` *-esima posizione dell'elemento di destinazione.*

Analogamente per l'operazione **ANDNOT**, che prevede la negazione del primo elemento, in AND con il secondo:

```
__m128i _mm_andnot_si128 (__m128i a, __m128i b)
```

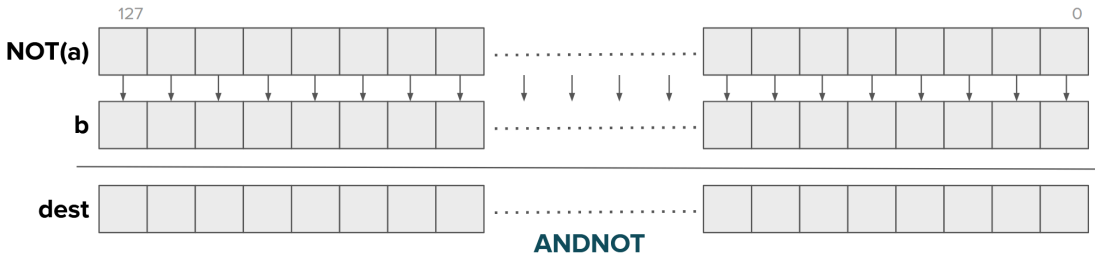


Figura 10: NOTAND.

Esistono inoltre le operazioni di **shift Left Logic** (SLL), **Right Logic** (SRL) **Right Arithmetic** (SRA)m eseguite su tutti gli elementi del registro con gli stessi parametri. L'entità dello shift é indicata con un immediato o un altro registro esteso. Sono supportati solo interi a 16, 32 e 6 bit.Ad esempio:

```
__m128i _mm_slli_epi16 (__m128i a, int imm8)        // con immediato per offset
__m128i _mm_sll_epi16 (__m128i a, __m128i count)    // con registro per offset
```

Viene seguito uno shift a sinistra di ciascun elemento del vettore di interi a 16 bit a di un numero di bit indicato da imm8/count (per count si guardano gli ultimi bit).

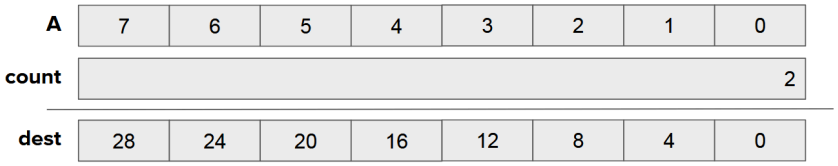


Figura 11: Shift Left con registro per offset (pultimi bit a destra).

Esiste anche la possibilità di eseguire lo **shift** dell'intero registro a 128 bit di una quantità di BYTE pari al valore dell'immediato count:

```
__m128i _mm_slli_si128 (__m128i a, int imm8)
```

Esistono poi istruzioni di shuffling, ossia per smistare dati da un registro sorgente ad uno destinatario in posizioni differenti, sulla base di un maschera.

```
__m128i _mm_shuffle_epi32 (__m128i a, int imm8)
```

👉 ***Nota*** : Esistono equivalenti istruzioni per gli altri tipi di dato (supportati dall'ISA x86).

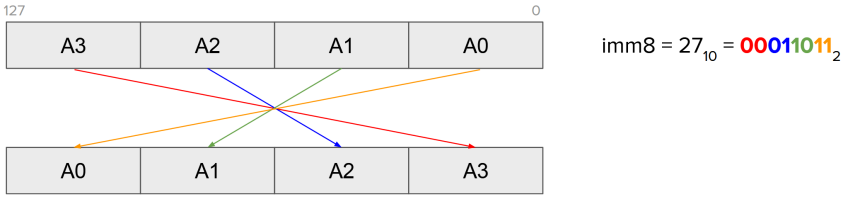


Figura 12: Esempio di shuffling di interi.

1.1.11. Packing/Unpacking

Istruzioni di packing unsigned

- Istruzioni che *impacchettano* in un registro il contenuto di due registri effettuando il casting (con saturazione*) a un tipo di dato di dimensione interiore (32→16, 16→8)

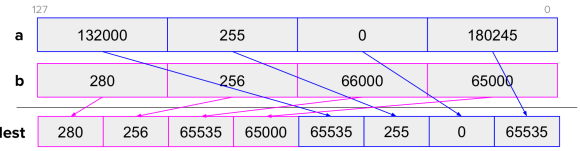
__m128i _mm_packus_epi16 (__m128i a, __m128i b)
__m128i _mm_packus_epi32 (__m128i a, __m128i b)

unsigned

*"con saturazione" significa che se il valore codificato supera il limite rappresentabile dal tipo a cui si fa il cast (che ha dimensione minore) allora si usa il tetto (il massimo rappresentabile con i bit a disposizione).
Normalmente verrebbero tolti semplicemente i bit in eccesso (a prescindere dal valore).

=> perdita d'informazione (potenziale)

- Esempio: dest = _mm_packus_epi32 (a, b);



Prendendo 132 000 (32 bit)
- con saturazione (16 bit) → max (unsigned) = 65 536 < 132 000 => 65 536 = valore max
- normalmente (16 bit) → $132\,000_{10} = 0010\ \underbrace{0000\ 0011\ 1010\ 0000_2}_{16\ bit} = 2^5+2^3+2^8+2^9 = 928$

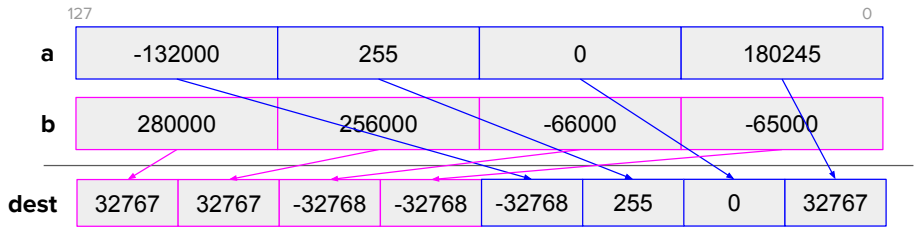
48

Istruzioni di packing signed

- Istruzioni di *impacchettamento* (32→16, 16→8) con saturazione analoghe alle precedenti ma per dati signed

__m128i _mm_packs_epi16 (__m128i a, __m128i b)
__m128i _mm_packs_epi32 (__m128i a, __m128i b)

- Esempio: dest = _mm_packs_epi32 (a, b);



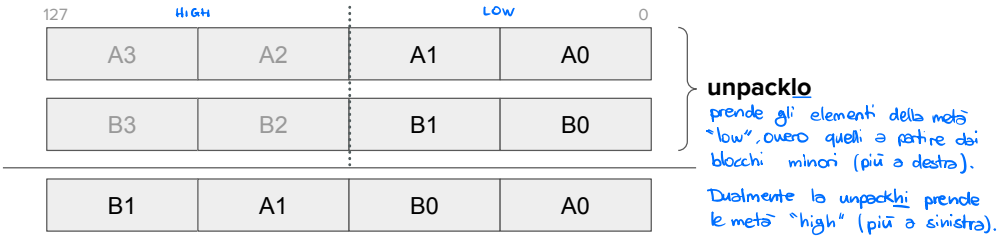
49

Istruzioni di unpacking

- Questo gruppo di istruzioni consentono di *combinare* il contenuto di due registri attraverso *interleaving*
- Disponibili per tutti i tipi di dato intero (8, 16, 32, 64 bit)
- Nel caso di dati a 32 bit:

__m128i _mm_unpacklo_epi32 (__m128i a, __m128i b)

__m128i _mm_unpackhi_epi32 (__m128i a, __m128i b)



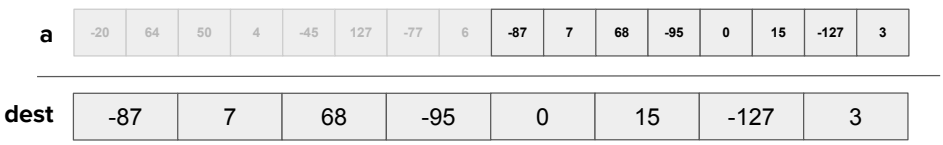
1.1.12. cvt (Passaggio tra tipi di dato)

Conversione tra tipi di dati: istruzioni cvt

- Talvolta è necessario passare da una rappresentazione ad un'altra (eg, int8→ int16)
- Già vista la compattazione per ridurre la dimensione dei dati mediante *packing*
- Il passaggio da un tipo di dato a uno, tipicamente di dimensione superiore, è gestito attraverso la classe di istruzioni cvt
  ↳ avviene senza perdita di informazione

Esempio, la conversione da valori a 8 bit signed a 16 bit signed può essere eseguita mediante l'istruzione seguente (che agisce sugli 8 byte meno significativi):

__m128i _mm_cvtepi8_epi16 (__m128i a)



1.1.13. Blending

Istruzioni di blending 1/2

- Questo gruppo di istruzioni consentono di *combinare* contenuto di due registri in funzione di un valore immediato (`blend`) o un registro (`blendv`)
- Disponibili per i tipi di dati interi a 8 [redacted] e 16 bit [redacted]
- Nel caso dell'istruzione `blend` (solo per int16):

__m128i _mm_blend_epi16 (__m128i a, __m128i b, const int imm8)

imm8 = 11001010 → come una maschera: 0 = elemento 1° registro, 1 = elemento 2° registro

A7	A6	A5	A4	A3	A2	A1	A0	**a**
B7	B6	B5	B4	B3	B2	B1	B0	**b**
B7	B6	A5	A4	B3	A2	B1	A0	**dest**

52

Istruzioni di blending 2/2

- Questo gruppo di istruzioni consentono di *combinare* contenuto di due registri in funzione di un valore immediato (`blend`) o un registro (`blendv`)
- Disponibili per i tipi di dati interi a 8 (`blendv`) e 16 bit (`blend`)
- Nel caso dell'istruzione `blendv` (solo per byte):

__m128i _mm_blendv_epi8 (__m128i a, __m128i b, __m128i mask)

→ si usa il bit più significativo (più a sinistra) per la scelta del registro (0 = 1°, 1 = 2°).

HEX

80	00	80	80	80	00	00	80	00	00	80	80	00	00	00	00	**mask**
A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	**a**
B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0	**b**
B15	A14	B13	B12	B11	A10	A9	B8	A7	A6	B5	B4	A3	A2	A1	A0	**dest**

53

1.1.14. Copia con inserimento di un valore

Copia con inserimento di un valore

Questa istruzione consente di copiare un registro in un altro modificando un singolo elemento del registro sorgente

__m128i _mm_insert_epi16 (__m128i a, int i, const int imm8)

Copia il registro sorgente **a** nel registro destinazione inserendo il valore **i** nella posizione specificata dal valore dell'immediato **imm8;** l'intero i può rappresentare un valore a 8, 16, 32 o 64 bit

Esempio: **dest** = _mm_insert_epi16 (**a**, 68, 3)

16

A	87	7	1245	95	-42	-15	127	3
dest	87	7	1245	95	**68**	-15	127	3

54

1.1.15. Confronto

Istruzioni di confronto 1/3

- Le istruzioni di confronto sono fondamentali nell'esecuzione scalare di un qualsiasi codice al fine di poter compiere operazioni differenti sui dati in base all'esito del confronto
- Nel caso SIMD non è possibile avere flussi di elaborazioni diverse in funzione dei multipli confronti effettuati sui dati e si sfruttano delle maschere per poter compiere o meno delle operazione sui dati
- A tal fine, sono confrontati tra loro elementi posti nelle stesse posizioni all'interno dei registri estesi generando una maschera
- Se il confronto è stato soddisfatto la maschera corrisponde a una serie di 1 per quell'elemento mentre una seri di 0 in caso contrario
- Le maschere ottenute possono essere utilizzate per filtrare e/o elaborare i dati in funzione dell'esito del confronto

55

Istruzioni di confronto 2/3

- Sono previste tre tipologie di confronti: uguale (EQ), maggiore (GT) o minore (LT)
  - EQ: per valori interi a 8, 16, 32, 64 bit
  - GT: per valori interi signed a 8, 16, 32, 64 bit
  - LT: per valori interi signed a 8, 16, 32 bit

```
__m128i _mm_cmpeq_epi8 (__m128i a, __m128i b)
```

```
__m128i _mm_cmpgt_epi8 (__m128i a, __m128i b)
```

```
__m128i _mm_cmplt_epi8 (__m128i a, __m128i b)
```

56

Istruzioni di confronto 3/3

Esempio: dest = _mm_cmpgt_epi16(A,B);

16

A	42	-765	68	870	0	1556	-7234	3030
B	87	7	87	95	0	-15	127	3
dest	0000	0000	0000	65535	0000	65535	0000	65535

I due valori 0000h e FFFFh (65535_{10}) generati dalle istruzioni di confronto si prestano ad essere utilizzati, ma non necessariamente, con istruzioni di tipo logico

57

1.1.16. Maschere per Assegnamenti Condizionati

Istruzioni di confronto senza branch/salti?

- Come appena visto, esistono istruzioni per eseguire confronti ma non istruzioni di salto condizionato (*branch*)
- Questo era prevedibile essendo possibile un solo flusso di esecuzione delle istruzioni
- Per eseguire operazioni diverse sui dati all'interno di un unico flusso di esecuzione si utilizzano delle *maschere* (eg, l'output di una istruzione di confronto)
- Mediante le maschere è possibile filtrare dati che soddisfano (o non soddisfano) determinate condizioni

Esercizio: si consideri l'esempio della slide precedente, come è possibile inserire in un registro solo i valori minimi per ogni linea tra i due registri estesi **A** (chiamato in seguito **prev_mins**) e **B** (chiamato in seguito **new_data**) contenenti ciascuno 8 valori interi signed a 16 bit?

58

Utilizzo di maschere per assegnamenti condizionati 1/4

- Con riferimento all'esercizio proposto nella slide precedente, una strategia per selezionare i valori minimi tra due vettori (**prev_mins** e **new_data**) potrebbe essere la seguente, partendo da una istruzione di confronto (**`__m128i _mm_cmplt_epi16`**)

16

prev_mins	180	-65	94	870	0	1556	-7234	3030
new_data	42	-765	68	870	0	54	-234	30
mask	FFFF	FFFF	FFFF	0000	0000	FFFF	0000	FFFF

< ? (CMPLT)

_mm_cmplt_epi16(new_data, prev_mins)

- La maschera indica con 0000 le linee che non soddisfano la condizione CMPLT mentre con FFFF il caso contrario (attenzione al mix, voluto, tra valori decimali ed esadecimali)

59

Utilizzo di maschere per assegnamenti condizionati 2/4

- Una volta ottenuta la maschera è possibile azzerare in **new_data** i valori che non concorrono all'aggiornamento dei valori contenuti in **prev_mins** e mantenere inalterati gli altri valori; per questa operazione è sufficiente un AND tra **mask** e **new_data**

16

mask	FFFF	FFFF	FFFF	0000	0000	FFFF	0000	FFFF	**AND**
new_data	42	-765	68	870	0	54	-234	30	
temp_1	42	-765	68	0000	0000	54	0000	30	_mm_and_si128

- I valori che non sono 0000 in **temp_1** saranno quelli che, successivamente, sostituiranno i corrispondenti valori per quella linea nel registro **prev_mins**

60

Utilizzo di maschere per assegnamenti condizionati 3/4

- Analogamente è possibile azzerare in **prev_mins** i valori che non risultano più minimi e mantenere inalterati gli altri valori; per questa operazione è sufficiente un ANDNOT tra **mask** e **prev_mins**; si ricorda che ANDNOT prima nega il primo operando (**mask**) e poi esegue AND del risultato con **prev_mins**

16

not(mask)	0000	0000	0000	FFFF	FFFF	0000	FFFF	0000	**ANDNOT**
prev_mins	180	-65	94	870	0	1556	-7234	3030	
temp_2	0000	0000	0000	870	0	0000	-7234	0000	_mm_pandnot_si128

- I valori non nulli in **temp_2** sono quelli che, successivamente, non saranno sostituiti dai corrispondenti valori per quella linea nel registro **prev_mins**

61

Utilizzo di maschere per assegnamenti condizionati 4/4

- L'ultimo passo consiste nell'aggiornare il valore di **prev_mins** *fondendo* i valori *sopravvissuti* in **temp_1** e **temp_2** con un semplice OR per ottenere, in **prev_mins**, i valori minimi per ogni linea tra i due registri (**prev_mins** e **new_data**)

16

temp_1	42	-765	68	0000	0000	54	0000	30
temp_2	0000	0000	0000	870	0	0000	-7234	0000
prev_mins	42	-765	68	870	0	54	-7234	30

OR

_mm_por_si128

- Sarebbe stato possibile usare altre istruzioni (eg, istruzioni di *blending*)? In questo caso no, non è disponibile l'istruzione *blendv* per interi a 16 bit

62

Utilizzo di maschere per assegnamenti condizionati in memoria

- Solo nel caso di **byte** è possibile fare scritture in memoria condizionate da una maschera
- Solo i byte in **a** per i quali il valore più significativo della corrispondente maschera è 1 sono scritti in memoria all'indirizzo **mem_addr**; gli altri byte in memoria rimangono invariati (simbolo "-" nella figura)
- La latenza è 6 clock e l'indirizzo non deve essere necessariamente allineato

void _mm_maskmoveu_si128 (__m128i a, __m128i mask, char* mem_addr)

mask	80h	20h	FFh	04h	00h	00h	20h	06h	80h	00h	60h	05h	00h	15h	07h	03h
a	20	64	50	4	-45	127	-77	6	-87	7	68	-95	0	15	-127	3
mem_addr	20	-	50	-	-	-	-	-	-87	-	-	-	-	-	-	-

63

1.1.17. Minimi e Massimi

Ricerca di minimi e massimi 1/2

- Le operazioni dell'esempio precedente sono molto comuni
- Per questa ragione, esistono istruzioni specifiche per l'identificazione di MIN e MAX
- Sono applicabili a registri estesi contenenti tipi di dato interi – signed (epi) e unsigned (epu) – a 8,16, e 32 bit

```
__m128i _mm_max_epi8 (__m128i a, __m128i b)
__m128i _mm_max_epi16 (__m128i a, __m128i b)
__m128i _mm_max_epi32 (__m128i a, __m128i b)

__m128i _mm_max_epu8 (__m128i a, __m128i b)
__m128i _mm_max_epu16 (__m128i a, __m128i b)
__m128i _mm_max_epu32 (__m128i a, __m128i b)
```

64

Ricerca di minimi e massimi 2/2

- Esclusivamente per interi unsigned a 16 bit, esiste la possibilità di identificare il MIN e la sua posizione all'interno di un registro mediante una singola istruzione (minpos)
- Il registro destinazione contiene il valore minimo, bit [15..0], e la sua posizione, bit [18..16]

```
__m128i _mm_minpos_epu16 (__m128i a)
```

Esempio: dest = _mm_minpos_epu16 (a)

								16
a	87	7	87	95	2	15	127	3
dest	0000	0000	0000	0000	0000	0000	0003	2

valore minimo — posizione

65

Esercizi

Esercizio 1: Scrivere il codice per individuare il valore minimo all'interno di un vettore di byte signed utilizzando:

i) **codice scalare**

ii) **maschere e operazioni logiche**

iii) **istruzioni di blending (se possibile)**

iv) **istruzioni per ricerca minimi (se possibile)**

Confrontare le prestazioni mediante *performance counters*

Esercizio 2: Individuare il valore minimo e relativo indice all'interno di un vettore di byte signed

Suggerimento: utilizzare un registro per gli indici (o iterazione loop e indici fissi)

66

1.1.18. Istruzioni Aritmetiche con Byte

Istruzioni aritmetiche con byte: addizione

Nell'ISA SSE sono presenti le seguenti istruzioni aritmetiche per dati di tipo BYTE:

__m128i _mm_add_epi8 (__m128i a, __m128i b) ⇝ somma "tradizionale" (con overflow)

Somma ciascuna coppia a[i] e b[i] di elementi presenti nei registri **a** e **b**

dest[i] = a[i] + b[i] i ∈ [0, 15]

Gli operandi possono essere BYTE signed o unsigned

__m128i _mm_adds_epi8 (__m128i a, __m128i b) ⇝ somma con saturazione

Somma con saturazione ciascuna coppia a[i] e b[i] di elementi presenti in **a** e **b**

dest[i] = saturation(a[i] + b[i]) i ∈ [0, 15]

Gli operandi possono essere solo BYTE signed

85

Istruzioni aritmetiche con byte: addizione tra dati unsigned

__m128i _mm_add_epi8 (__m128i a, __m128i b)

a	200	64	250	4	45	128	77	6	87	7	68	195	0	255	128	3	+
b	100	68	210	34	145	120	68	26	87	71	32	106	30	53	128	103	=
a + b	44	132	204	38	190	248	145	32	174	78	100	45	30	52	0	106	

I casi di overflow sono evidenziati in rosso

86

Istruzioni aritmetiche con byte: addizione tra dati signed

__m128i _mm_add_epi8 (__m128i a, __m128i b)

a	-20	64	50	4	-45	127	-77	6	-87	7	68	-95	0	15	-127	3	+
b	100	68	10	34	-127	-128	68	26	87	71	32	106	-30	53	-127	-103	=
a + b	80	-124	60	38	84	-1	-9	32	0	78	100	11	-30	68	2	-100	

I casi di overflow sono evidenziati in rosso

87

Istruzioni aritmetiche con byte: addizione con saturazione

__m128i _mm_adds_epi8 (__m128i a, __m128i b)

a	-20	64	50	4	-45	127	-77	6	-87	7	68	-95	0	15	-127	3	+
b	100	68	10	34	-127	-128	68	26	87	71	32	106	-30	53	-127	-103	=
Sat(a + b)	80	127	60	38	-128	-1	-9	32	0	78	100	11	-30	68	-128	-100	

Non si verificano mai situazioni di overflow (si satura, evidenziato in blu)

Le addizioni con saturazione sono supportate solo con dati BYTE (epi8) e INT16 (ep16) di tipo signed

88

Istruzioni aritmetiche con byte: sottrazione tra dati unsigned

__m128i _mm_sub_epi8 (__m128i a, __m128i b)

a	-200	64	250	4	45	128	77	6	87	7	68	195	0	255	128	3	-
b	100	68	210	34	145	120	68	26	87	71	32	106	30	53	128	103	=
a - b	100	252	40	-226	156	8	9	236	0	192	36	89	226	202	0	156	

I casi di overflow sono evidenziati in rosso

89

Istruzioni aritmetiche con byte: sottrazione tra dati signed

__m128i _mm_sub_epi8 (__m128i a, __m128i b)

a	-20	64	50	4	-45	127	-77	6	-87	7	68	-95	0	15	-127	3	-
b	100	68	10	34	-127	-128	68	26	87	71	32	106	-30	53	-127	-103	=
a - b	-120	-4	40	-30	82	-1	111	-20	82	-64	36	55	30	-38	0	106	

I casi di overflow sono evidenziati in rosso

90

Istruzioni aritmetiche con byte: sottrazione con saturazione

__m128i _mm_subs_epi8 (__m128i a, __m128i b)

a	-20	64	50	4	-45	127	-77	6	-87	7	68	-95	0	15	-127	3	-
b	100	68	10	34	-127	-128	68	26	87	71	32	106	-30	53	-127	-103	=
Sat(a - b)	-120	-4	40	-30	82	127	-128	-20	-128	-64	36	-128	30	-38	0	106	

Non si verificano mai situazioni di overflow

Le addizioni con saturazione sono supportate solo con dati BYTE (epi8) e INT16 (ep16) di tipo signed

91

1.1.19. ABS

Absolute value (ABS)

- Calcolano il valore assoluto degli elementi di un registro contenente valori signed
- Disponibile per interi a 8, 16 e 32 bit

```
__m128i _mm_abs_epi8 (__m128i a)
__m128i _mm_abs_epi16 (__m128i a)
__m128i _mm_abs_epi32 (__m128i a)
```

325678	-254	-563434	3643

325678	254	563434	3643

⇝ Dato che i valori sono signed basta togliere il bit più significativo (MSB), che in complemento a due "identifica" il segno

92

1.1.20. Add e Sub orizzontali

Add e Sub *orizzontali* senza saturazione

- Oltre alle canoniche operazioni di somma e sottrazione *verticale*, è possibile eseguire anche le stesse *orizzontalmente* tra interi signed a 16 e 32 bit contenuti in due registri

```
__m128i _mm_hadd_epi16 (__m128i a, __m128i b)
__m128i _mm_hadd_epi32 (__m128i a, __m128i b)
__m128i _mm_hsub_epi16 (__m128i a, __m128i b)
__m128i _mm_hsub_epi32 (__m128i a, __m128i b)
```

Esempio:

dest = _mm_hadd_epi32 (a, b)

somma valori (dello stesso reg.) due a due (sempre partendo da destra.

a	34	-254	-13	143
b	8	25	-56	36
dest	33	-20	-220	140

93

Add e Sub *orizzontali* con saturazione

- Le operazioni di somma e sottrazione orizzontale con saturazione esistono solo per interi signed a 16 bit

```
__m128i _mm_hadds_epi16 (__m128i a, __m128i b)
__m128i _mm_hsubs_epi16 (__m128i a, __m128i b)
```

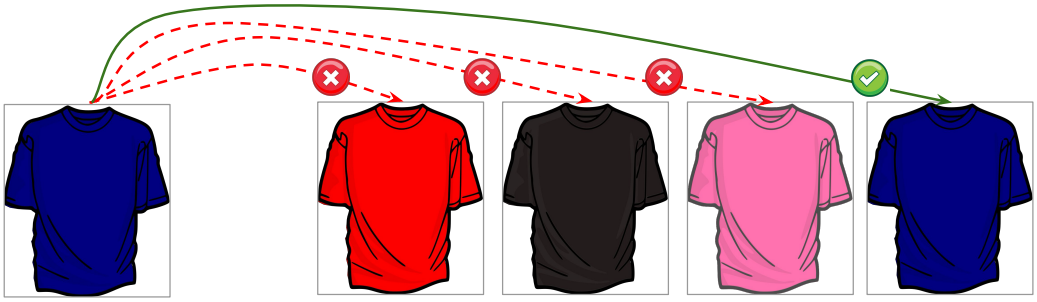
- Le istruzioni di somma e sottrazione verticale possono essere utilizzate anche con registri a 64 bit (__m64)

94

1.1.21. SAD (Sum of Absolute Differences)

Ulteriori istruzioni aritmetiche per SAD

- Esistono nel set di istruzioni SSE altre istruzioni aritmetiche con interi
- Tra queste, ci sono istruzioni che consentono di eseguire SAD (Sum of Absolute Differences), un'operazione spesso utilizzata nell'ambito della computer vision
- SAD può essere utile, per esempio, per confrontare se due immagini o patch sono *simili* oppure no



95

1.1.22. Somme varie (esempi)

Somma degli elementi di un array di byte unsigned

Problema: calcolare la somma di tutti gli elementi di un array di byte unsigned di dimensione pari a `length`

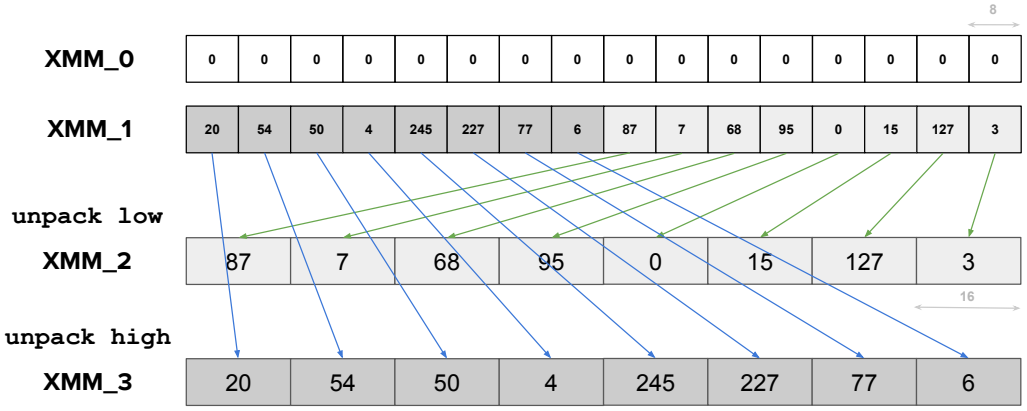
```
unsigned char A[length];
unsigned int sum=0;

for (i=0; i<length; i++)
    sum = sum + A[i];
```

- E' fondamentale prestare attenzione a possibili condizioni di overflow
- Inoltre, dovendo addizionare valori a 8 bit, per prima cosa è necessario estendere la dimensione del risultato (la somma di due valori a 8 bit è codificata con 9 bit)
- Per l'estensione si possono utilizzare le istruzioni `unpack` (pagina successiva)

96

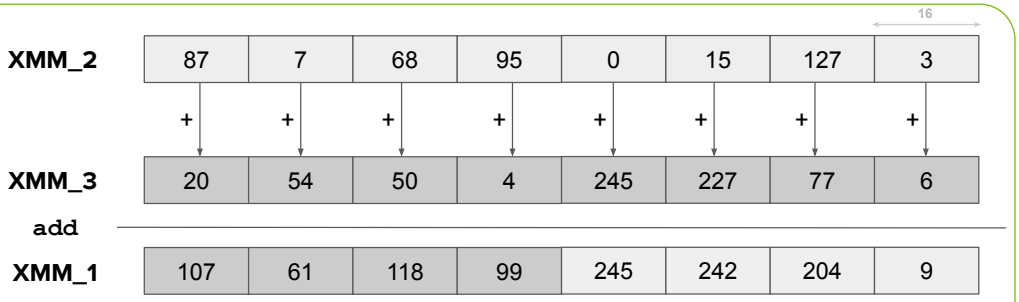
Somma degli elementi di un array di byte unsigned: 8 → 16 bit

- Non essendo sufficienti 8 bit per la somma di due byte, è necessario eseguire le addizioni con dati (almeno) convertiti a 16 bit
- Le istruzioni `unpack` consentono questo utilizzando un registro contenente tutti zero



97

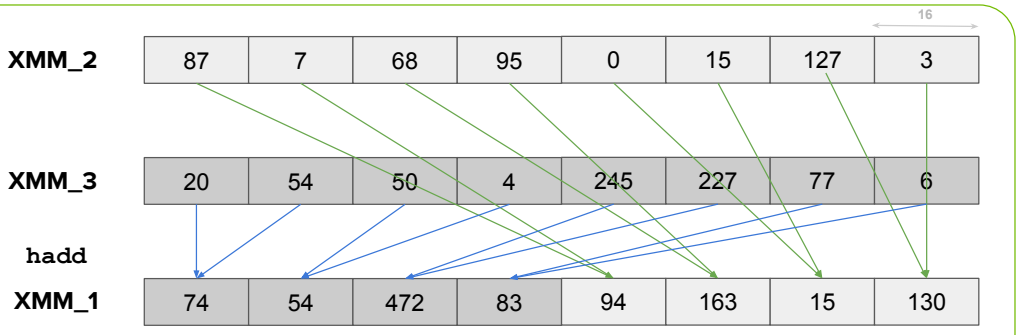
Somma degli elementi di un array di byte unsigned (con `add`)



- Una volta estesi i byte a 16 bit, con le istruzioni `unpack` è possibile eseguire come passo iniziale l'addizione tra coppie di elementi a 16 bit *verticali* mediante l'istruzione `add` (in figura) prima di aggiornare un registro usato come accumulatore delle somme parziali
- In alternativa, è anche possibile eseguire l'addizione orizzontale tra elementi a 16 bit utilizzando l'istruzione `hadd` (come mostrato nella pagina successiva)

98

Somma degli elementi di un array di byte unsigned (con `hadd`)



- Il risultato dell'addizione (`add` o `hadd`) tra due elementi è un valore a 9 bit (<=255*2), contenuto nei 16 bit (x8) di XMM_1, che può aggiornare un accumulatore (eg, a 16 bit)
- Tuttavia, con accumulo a 16 bit si può verificare overflow se `length` > 2048 byte
- Infatti, con dati 16 bit unsigned, si verifica overflow con più di 128 addizioni (i.e., con 129 elementi la somma può raggiungere il valore 129x(255+255) = 65790 > 65535)

99

Somma degli elementi di un array di byte unsigned e overflow

- Se il numero di elementi dell'array è minore o uguale a 2048 è possibile ignorare il problema e procedere in sicurezza con l'accumulo di multipli (x8) valori a 16 bit
- In caso contrario, esistono almeno due soluzioni:
  - si eseguono addizioni in gruppi (max 2048 element)i e, al termine di ciascun gruppo di operazioni, si utilizza un accumulatore di maggiori dimensioni (eg, a 32 bit)
  - si sommano tutti gli elementi con un unico ciclo aumentando la dimensione dei dati (eg, passando a 32 bit, con gli stessi limiti appena evidenziati) per evitare problemi di overflow dopo 128 addizioni (ie. dopo aver sommato più di 128x16 elementi)

La prima soluzione ha un costo maggiore nella gestione dei loop multipli (eg, modulo 2048 elementi) ma, rispetto alla seconda, esegue complessivamente meno istruzioni di unpacking e addizioni di dati di maggiori dimensioni che riducono il grado di parallelismo ottenibile

100

Somma degli elementi di un array di byte unsigned: `hadd` vs `add`

- Dall'analisi delle specifiche delle istruzioni, per la specifica finalità dell'esempio, risulta vantaggioso l'utilizzo della `add` (minore latenza e CPI) vs `hadd`
- Inoltre, sebbene irrilevante sommando due byte, il risultato della `hadd` è signed

```
__m128i _mm_hadd_epi16 (__m128i a, __m128i b)
```

Synopsis

```
__m128i _mm_hadd_epi16 (__m128i a, __m128i b)
#include <tmmintrin.h>
Instruction: phaddw xmm, xmm
CPUID Flags: SSSE3
```

Description

Horizontally add adjacent pairs of 16-bit integers in `a` and `b`, and pack the signed 16-bit results in `dst`.

Operation

```
dst[15:0] := a[31:16] + a[15:0]
dst[31:16] := a[63:48] + a[47:32]
dst[47:32] := a[95:80] + a[79:64]
dst[63:48] := a[127:112] + a[111:96]
dst[79:64] := b[31:16] + b[15:0]
dst[95:80] := b[63:48] + b[47:32]
dst[111:96] := b[95:80] + b[79:64]
dst[127:112] := b[127:112] + b[111:96]
```

Latency and Throughput

Architecture	Latency	Throughput (CPI)
Alderlake	2	1
Icelake Xeon	-	1.07
Sapphire Rapids	2	1
Skylake	3	2

```
__m128i _mm_add_epi16 (__m128i a, __m128i b)
```

Synopsis

```
__m128i _mm_add_epi16 (__m128i a, __m128i b)
#include <emmintrin.h>
Instruction: paddw xmm, xmm
CPUID Flags: SSE2
```

Description

Add packed 16-bit integers in `a` and `b`, and store the results in `dst`.

Operation

```
FOR j := 0 to 7
        i := j*16
        dst[i+15:i] := a[i+15:i] + b[i+15:i]
ENDFOR
```

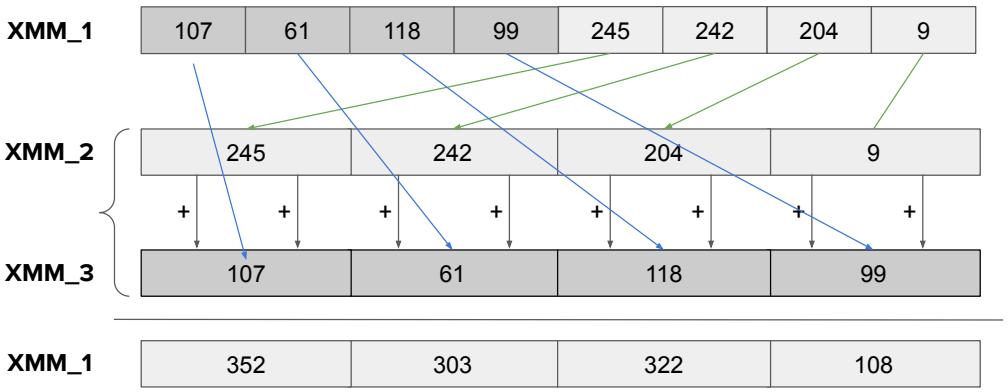
Latency and Throughput

Architecture	Latency	Throughput (CPI)
Alderlake	1	0.333333333
Icelake Xeon	1	0.33
Sapphire Rapids	1	0.333333333
Skylake	1	0.33

101

Somma degli elementi di un array: addizioni a 32 bit

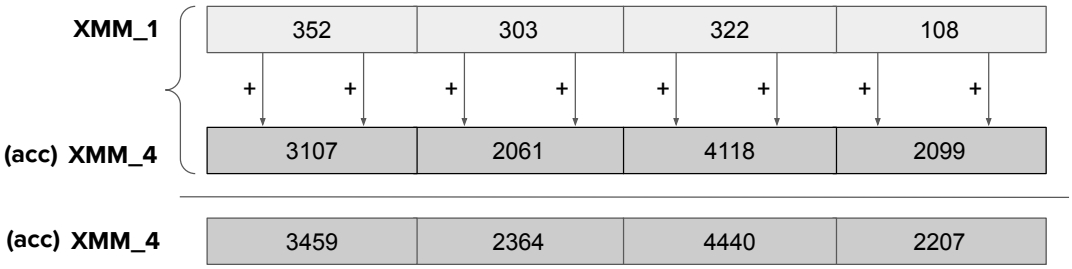
- Per gestire l'overflow è possibile estendere (unpack) le somme parziali da 16 a 32 bit, riducendo così il parallelismo, e poi aggiornare un accumulatore con elementi a 32 bit
- L'overflow non è eliminato ma si verifica con array maggiori di $2^{32}/2^{10}$ (x4) byte



102

Somma degli elementi di un array: accumulo a 32 bit

- Una volta eseguite le addizioni tra 4 coppie di dati a 32 bit è possibile procedere con la stessa operazione per aggiornare le 4 somme a 32 bit in un registro accumulatore



- Terminate tutte le somme è possibile sommare i 4 valori a 32 bit presenti nel registro accumulatore per ottenere il risultato desiderato
- Questo può essere fatto con istruzioni `unpack` e `add` e/o `extract`

103

1.1.23. Media (abg) e Cambio di Segno

Media (approssimata): Average

- Esegue un media approssimata sommando due valori corrispondenti nei registri **a** e **b** e successivamente eseguendo uno shift a destra di un bit del risultato
  = divisione per 2
- Disponibile solo per valori unsigned a 8 e 16 bit
- La latenza è 1 clock

__m128i _mm_avg_epu8 (__m128i a, __m128i b)

__m128i _mm_avg_epu16 (__m128i a, __m128i b)

16

a	17	7	3	15	0	15	7	3
b	7	3	8	5	4	15	12	4
dest	12	5	5	10	2	15	9	3

104

Cambiamento di segno condizionato: Sign 1/2

- Dati due registri estesi **a** e **b**, ogni elemento **i** del registro destinazione **dest** è:

  desti = 0 se il corrispondente elemento b_i**=0**
  desti = a_i se il corrispondente elemento b_i**>0**
  desti = -a_i se il corrispondente elemento b_i**<0**

  ⇝ a può essere negativo
  (=> -a in realtà è positivo)

- Disponibile per interi con segno a 8, 16 e 32 bit

__m128i _mm_sign_epi8 (__m128i a, __m128i b)

__m128i _mm_sign_epi16 (__m128i a, __m128i b)

__m128i _mm_sign_epi32 (__m128i a, __m128i b)

105

Cambiamento di segno condizionato: Sign 2/2

- Esempio:

```
dest = _mm_sign_epi32(a,b);
```

a	-1362	-3034	32	4567
b	-56	0	322	-2108
dest	1362	0	32	-4567

106

1.1.24. Moltiplicazioni

Moltiplicazioni tra interi: Mul

- La moltiplicazione tra interi richiede per il risultato un numero di bit doppio
- Nel caso sia necessario eseguire moltiplicazioni tra interi, esistono alcune istruzioni
- Mul, per tipi signed e unsigned a 32 bit, esegue moltiplicazioni complete a 32 bit

__m128i _mm_mul_epu32 (__m128i a, __m128i b)

__m128i _mm_mul_epi32 (__m128i a, __m128i b)

32

a	-4	3	-60000	1	_mm_mul_epi32 (a, b)
b	6	5	4	2	
dest	-240000		2		risultato a 64 bit

107

Moltiplicazioni tra interi: Mullo

- E' anche possibile eseguire le moltiplicazioni tra tutte le linee corrispondenti
- Tuttavia, servirebbe un registro di dimensione doppia per memorizzare i risultati
- Per questo motivo, esistono istruzioni che forniscono il risultato con un numero di bit pari alla dimensione del dato di origine troncando la parte più significativa

__m128i _mm_mullo_epi16 (__m128i a, __m128i b)
__m128i _mm_mullo_epi32 (__m128i a, __m128i b)

32

a	-4	3	-60000	1
b	6	5	4	2
dest	-24	15	-240000	2

_mm_mullo_epi32 (a, b)

per ogni linea, il risultato contiene solo i 32 bit meno significativi di a_i*b_i

108

Moltiplicazioni tra interi: Mulhi

- Nel caso delle istruzioni *mulhi*, solo per interi a 16 bit, del risultato a 32 bit della moltiplicazione sono memorizzati nel registro destinazione solo i 16 bit più significativi

__m128i _mm_mulhi_epu16 (__m128i a, __m128i b)
__m128i _mm_mulhi_epi16 (__m128i a, __m128i b)

16

a	42	65	2	18	60000	0	2	60000
b	42	7	8	200	2	54	3	4
dest	0	0	0	0	1	0	0	3

_mm_mulhi_epu16 (a, b)

per ogni linea, il risultato contiene solo i 16 bit più significativi di a_i*b_i

109

Moltiplicazioni tra interi: Mulhrs

- Mulhrs (Multiply High Round Scale) è un'istruzione piuttosto complessa
- Per ogni linea a 16 bit nei registri estesi:
  1. moltiplica due interi con segno a 16 bit, generando un risultato a 32 bit
  2. esegue uno shift a destra del risultato mantenendo solo i 18 bit più significativi
  3. somma al valore a 18 bit il valore 1 per eseguire un arrotondamento
  4. memorizza i 16 bit [16..1] del risultato nella linea corrispondente del registro destinazione
- La latenza è pari a 5 clock

__m128i _mm_mulhrs_epi16 (__m128i a, __m128i b)

110

Moltiplicazioni tra interi: Mulhrs

Moltiplicazione convenzionale a 16 bit: 2000 **x** 1000 **=** 2000000

Con `Mulhrs`, il risultato è 61 (0000 0000 0011 1101):

0000 0111 1101 0000	**x**
0000 0011 1110 1000	**=**
0000 0000 0001 1110 1000 0100 1000 0000	**>> 14**
0000 0000 0001 1110 10	**+1**
0**000 0000 0001 1110 1**1	**=**
0000 0000 0011 1101	**61_{10}**

111

Esercizio

Esercizio: Calcolare il prodotto scalare tra vettori di 4 elementi (byte signed) contenuti all'interno di due array, confrontando mediante l'utilizzo di *performance counters*, al variare delle dimensioni degli array e del grado di ottimizzazione del compilatore:

i) **codice scalare**

ii) **codice vettoriale**

112

Per quanto riguarda `mulhrs`) serve a risolvere un problema molto specifico ma fondamentale nel calcolo numerico digitale (DSP - Digital Signal Processing): mantenere la «scala» dei numeri dopo una moltiplicazione.

In termini semplici: serve a moltiplicare due numeri trattandoli come se fossero frazioni (tra -1 e 1) invece che come numeri interi molto grandi.

Utilizzo concreto di mulhrs (extra)

Immaginando di usare i numeri interi a 16 bit (che vanno da -32.768 a $+32.767$) per rappresentare l'intensità di un segnale audio o la luminosità di un pixel.

Spesso in informatica si usa la logica «Fixed Point» (Q15):

- Si immagina che il valore massimo 32.767 corrisponda a 1.0 (100% intensità).
- Si immagina che 16.384 corrisponda a 0.5 (50% intensità).

Se si intende calcolare "il 50% del 50%" (ovvero $0.5 \times 0.5 = 0.25$), usando una normale moltiplicazione intera:

$$16.384 \times 16.384 = 268.435.456$$

Questo numero è molto elevato e non entra in 16 bit (overflow), in più non ha senso nella scala originale (dove il massimo era circa 32.000). L'istruzione mulhrs (Multiply High Round Scale) esegue la moltiplicazione e riporta immediatamente il risultato nella scala "corretta".

Matematicamente:

$$\text{Risultato} = \frac{(A \times B) + \text{Arrotondamento}}{2^{15}}$$

Tornando all'esempio di prima:

$$16.384 \times 16.384 = 268.435.456$$
$$\frac{2^{15}}{32.768} = \frac{268.435.456}{32.768} = 8.192$$

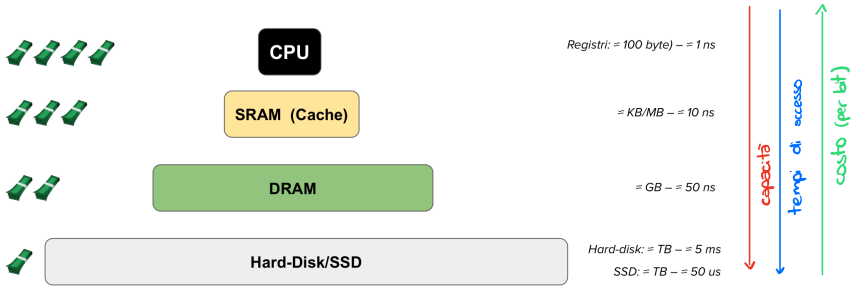
8.192 nella nostra scala corrisponde esattamente a 0.25.

1.2. Memorie e Cache

1.2.1. Introduzione

Gerarchia delle memorie

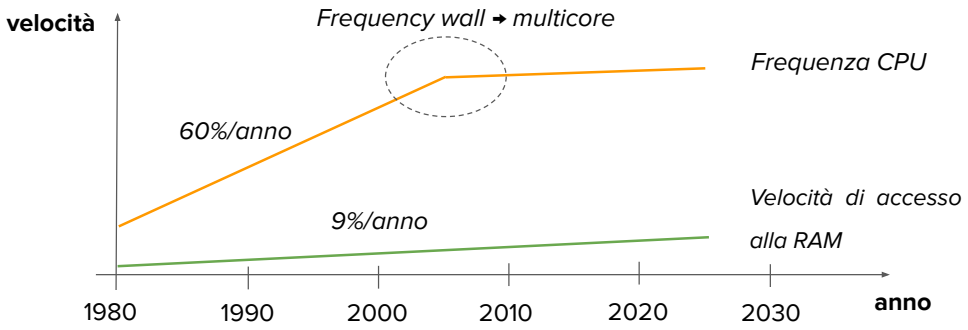
- In un sistema di elaborazione sono presenti varie tecnologie di memorie con diverse caratteristiche quali: capacità, tempo di accesso,e costo per bit



2

CPU vs Memoria: confronto sulle prestazioni

- Idealmente, sarebbe auspicabile che la CPU potesse accedere alla memoria alla stessa velocità con la quale necessità di accedere ai dati
- Tuttavia, il divario tra la frequenza delle CPU e il tempo di accesso alla memoria (eg, DRAM) è elevato e aumentato significativamente per un lungo periodo (molti dati)



3

1.2.2. SRAM vs DRAM

La differenza tra SRAM e DRAM é stata trattata anche in *questo approfondimento* nel modulo su CUDA.

Static RAM (SRAM) e Dynamic RAM (DRAM) 1/2

- Esistono due tecnologie per realizzare dispositivi di memoria RAM
  - Static RAM SRAM
    - L'elemento base per la memorizzazione di un bit è un latch (eg, 6 transistor)

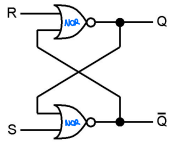


Tabella della verità

S	R	Q	
0	0	Q	Hold
0	1	0	Reset
1	0	1	Set
1	1	0	Non valido

si può realizzare (nello stesso modo) anche a NAND

  - Dynamic RAM DRAM
    - L'elemento base per la memorizzazione di un bit è un condensatore (e 1 transistor)

Condensatore di piccola capacità, poche cariche, si scarica velocemente → richiede refresh

4

Static RAM (SRAM) e Dynamic RAM (DRAM) 2/2

- Le DRAM hanno una capacità molto maggiore a parità di area (eg, 1/6 transistor)

SRAM *Area/bit*

DRAM *Area/bit*

- Per questo motivo, il costo per bit è favorevole alle DRAM
- Tuttavia, le DRAM sono più lente nell'accesso ai dati rispetto alle SRAM (eg, 10x)
- Per quanto riguarda il consumo per bit: le DRAM risultano avvantaggiate nonostante sia indispensabile accedere a ciascun bit di informazione più volte al secondo per non perdere l'informazione memorizzata
- Tale periodico *refresh* è realizzato mediante un opportuno dispositivo al fine rigenerare continuamente la carica in ciascun condensatore

5

Considerazioni sulle memorie

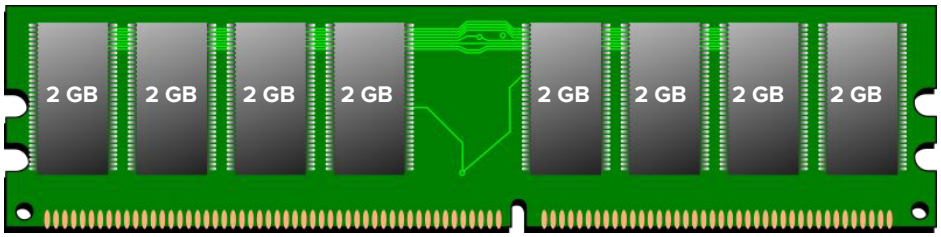
- Idealmente, sarebbe desiderabile avere memorie capienti (come le DRAM) e veloci (come le SRAM)
- Tuttavia, le capienti DRAM, a causa dell'elevata latenza (eg, 50+ ns), hanno un tempo di accesso considerevole che induce notevoli ritardi (cicli di WAIT)
- Risulta quindi necessaria una strategia che consenta di superare questa limitazione
- Tale strategia si basa sull'utilizzo di una veloce memoria cache (SRAM) posta tra la CPU e la DRAM al fine di (cercare di) compensare l'elevato tempo di accesso delle DRAM
- Prima di descrivere come funziona una cache bene analizzare come sono organizzate le DRAM

6

1.2.3. Layout di una memoria DRAM

Layout di una memoria (eg, DRAM)

- Le moderne CPU hanno ampi bus dati (eg, 64 bit) e più di 32 linee di indirizzo (eg, 36)
- I moduli di memoria sono composti da multipli dispositivi a 8 bit al fine di incrementare la capacità di trasferimento dei dati
- Esempio, una ipotetica memoria da 16 GB potrebbe essere composta da 8 moduli di memoria (ciascuno con bus dati a 8 bit) al fine di eseguire trasferimenti a 64 bit

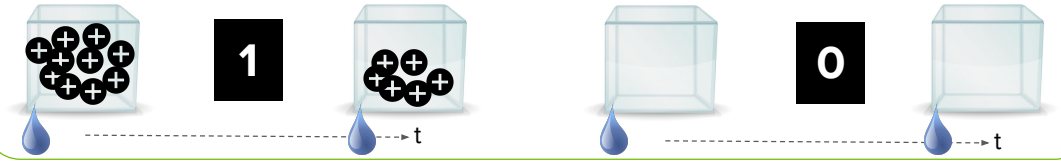


https://youtu.be/7J7X7aZvMXQ?si=5QPrKxwYVjURjbAt

7

Elemento di memoria in una DRAM

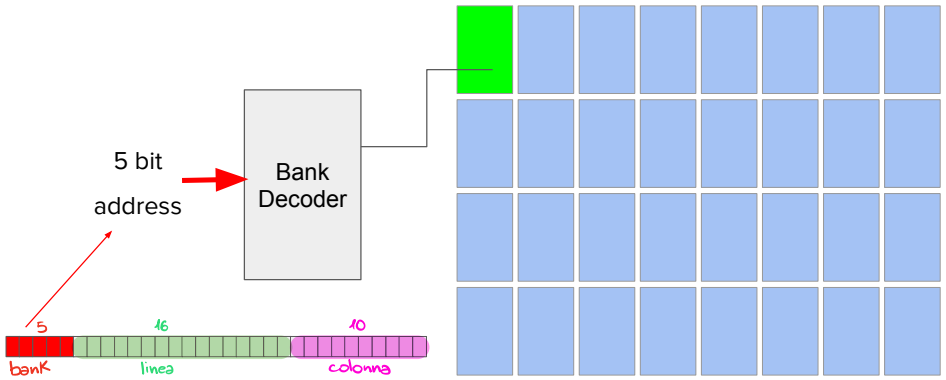
- Come anticipato, l'elemento di memoria unitario (bit) in una DDR è composto da un transistor un condensatore
- Questa strategia consente di avere, a parità di area, una capacità molto più elevata rispetto a una SRAM
- Tuttavia, una SRAM risulta avere un tempo di accesso molto inferiore vs DRAM
- L'informazione memorizzata in una DRAM è la quantità di carica nel condensatore
- Tale carica, tende a svanire nel tempo ed è necessario un continuo *refresh* di ogni bit della memoria che viene realizzato attraverso un apposito circuito



8

Organizzazione di una DRAM (2 GB) 1/4

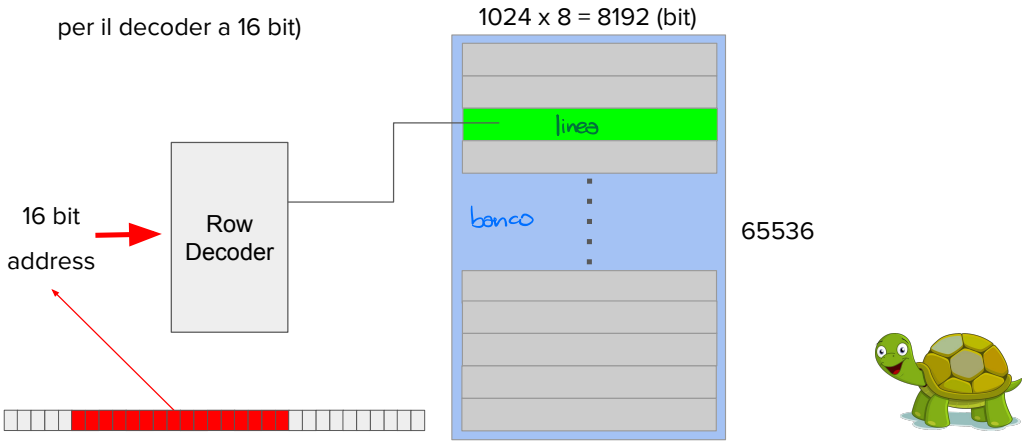
- La memoria da 2 GB è organizzata in banchi (bank), ad esempio 32
- Ogni banco è identificato mediante i **5 bit** più significativi dell'indirizzo a **31 bit**
- Al suo interno la decodifica (di **II livello**) è organizzata nel modo seguente



9

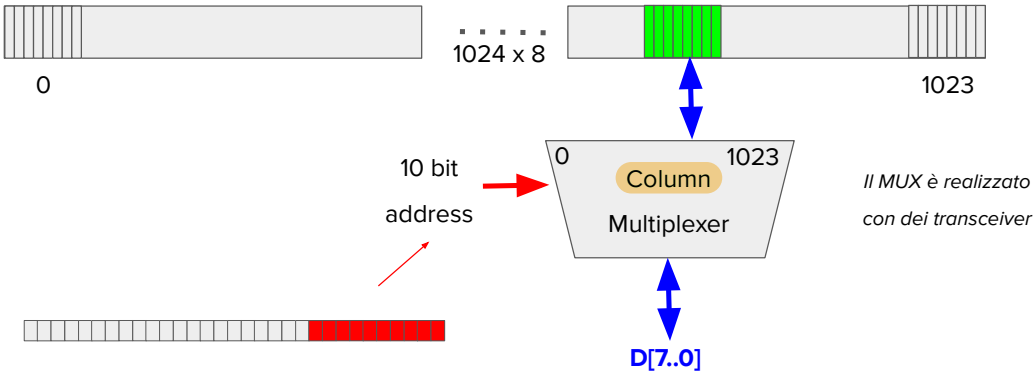
Organizzazione di una DRAM (2 GB) 2/4

- Ogni banco è organizzato in righe (16 bit successivi dell'indirizzo a 31 bit)
- Individuare e predisporre l'accesso ai dati nella linea è la fase più lenta (non solo per il decoder a 16 bit)



Organizzazione di una DRAM (2 GB) 3/4

- Individuata la linea, i dati (8 bit/modulo) possono essere acceduti rapidamente
- Inoltre, individuata la linea, è possibile eseguire veloci accessi in sequenza (*burst*)
- Gli accessi burst sono fondamentali per riempire le memorie cache (*line-fill*)



Organizzazione di una DRAM (2 GB) 4/4

- E' importante osservare che, la priorità, è reperire il byte/dato richiesto (in verde) e inviarlo alla CPU il prima possibile, al fine di consentire alla CPU di proseguire il proprio lavoro

- Successivamente, in *background* e senza l'intervento diretto della CPU, sarà eseguito un line-fill degli altri dati appartenenti alla line di cache nella quale è contenuto il byte/dato verde
- Questa strategia consente di rifornire la CPU con i dati richiesti appena possibile, in modo da non rimamere bloccata ulteriormente in attesa di quel dato, e, in background, di riempire una linea di cache sfuttando il principio di località discusso in seguito ipotizzando l'utilizzo di dati in prossimità a breve

12

1.2.4. Cache

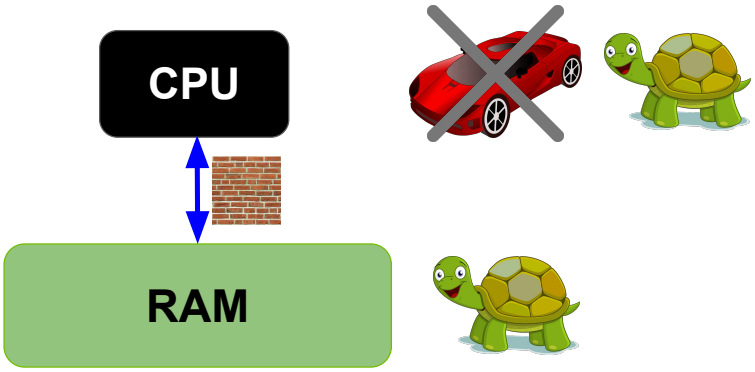
Introduzione alle memorie cache

- Con riferimento alle memorie DDR, una volta che è stata individuata la linea (row open), è possibile accedere a tutti i dati in essa contenuti molto rapidamente (poiché le fasi precedenti non sono più necessarie)
- Questo, consente di eseguire veloci accessi *burst* a dati multipli senza dover eseguire nuovamente la decodifica di tutti gli indirizzi (ma solo di quelli che identificano i dati desiderati all'interno della linea generabili in modo automatico)
- Tale caratteristica è fondamentale per eseguire velocemente riempimenti di linee intere (line-fill) nelle cache
- In seguito, vedremo come sia possibile sfruttare questa caratteristica e la tipica modalità di accesso ai dati per velocizzare mediante il *caching* gli accessi alla lenta memoria centrale DRAM mediante l'utilizzo di più veloci SRAM

13

Memory wall

- Una CPU che non può rapidamente accedere ai dati deve rimanere in attesa (WAIT)
- Una memoria *lenta* riduce significativamente l'efficienza delle CPU



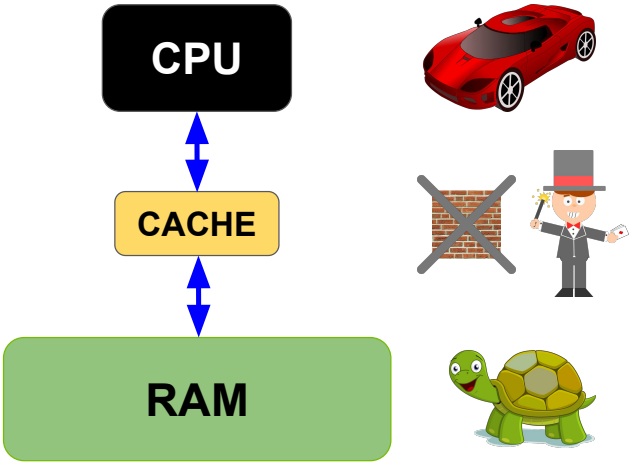
14

Cache

- Le cache sono memorie SRAM piccole e veloci (vs DRAM) poste tra la CPU e la RAM

tecnologia: SRAM

tecnologia: DRAM



15

Cache: principio di località spaziale e temporale

Al fine di emulare il comportamento ideale di una memoria, si sfruttano due osservazioni per inserire nella memoria cache un sottoinsieme dei dati presenti in memoria RAM:

- la **località spaziale**: se si accede ad un dato ad un determinato indirizzo di memoria, è presumibile che si acceda anche a dati contigui o in prossimità (e.g. prelievo codice, accesso ad elementi consecutivi di un array, etc)
- la **località temporale**: se si accede ad un determinato dato in memoria, è presumibile che, successivamente, si acceda al medesimo dato (e.g. codice di un loop, funzioni chiamate frequentemente, accesso ad elementi non consecutivi di un array, etc)

16

Accesso alla memoria in presenza di cache

- Una cache è una memoria associativa (contiene indirizzi e dati) organizzata per linee; in ciascuna linea è presente l'indirizzo (unico associato alla linea) e i relativi dati
- Con le cache è anche presente un dispositivo, il controller della memoria, che esamina (indirizzi, segnali di controllo) le richieste di accesso alla memoria
- La richiesta è inviata a tutti i dispositivi di memoria, incluse le cache
- Se tale indirizzo è presente all'interno della cache (Hit), si esegue rapidamente l'accesso da tale dispositivo
- In caso contrario (Miss), si accede alla più lenta memoria di livello inferiore accedendo all'indirizzo richiesto; tuttavia, ipotizzando che sia valido il principio di località, si esegue un accesso burst per leggere un'intera linea della cache (line-fill)
- In caso di Hit, la velocità di accesso ai dati è quello della memoria veloce (ie, cache)

17

Organizzazione di una cache in linee

- Una cache è una memoria associativa organizzata per linee di lunghezza costante
- In ogni linea è presente l'indirizzo dei dati (contigui) presenti, esclusi i bit meno significativi che consentono di individuare ciascun byte

18

Cache Fully-associative 1/3

- La presenza o meno di un indirizzo è determinato confrontando l'indirizzo richiesto con quello presente in ciascuna linea
- Questo confronto, per essere veloce, deve avvenire contemporaneamente

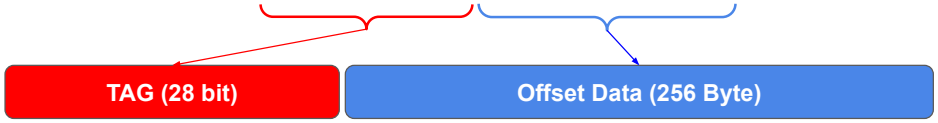
Indirizzo cercato (serve un comparatore per ogni linea)

19

Cache Fully-associative 2/3

- Ogni linea contiene 2^K byte; pertanto, nella comparazione dovranno essere esaminati tutti gli indirizzi esclusi i K bit meno significativi ⇝ questi K bit sono usati (tutti) per indirizzare i singoli byte nella riga.
- L'indirizzo completo, escludendo tali bit meno significativi, è denominato **TAG**
- Esempio, CPU con 36 bit di indirizzo, cache da 128K con 512 linee da 256 byte (K=8)
  - Il TAG è composto dai 36-8 = 28 bit più significativi dell'indirizzo
  - Sono necessari 512 comparatori a 28 bit per confrontare i TAG
  - Ogni byte è localizzato (**Offset**) all'interno della linea mediante i K=8 bit meno significativi dell'indirizzo a 36 bit
  - Indirizzo completo: **A35**,**A34**,....,**A8**,**A7**,**A6**,**A5**,**A4**,**A3**,**A2**,**A1**,**A0**

↳ Ogni byte è indirizzabile
⇒ $256 = 2^8$
⇒ 8 bit (LSB) sono usati per "indirizzare" i singoli byte della linea



20

Cache Fully-associative 3/3

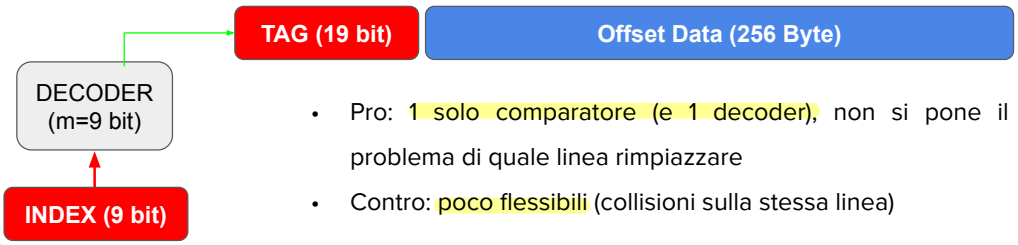
- In una cache *fully associative*, ogni indirizzo viene collocato in una linea libera senza altri vincoli (ie, ogni indirizzo può finire in qualsiasi linea di cache)
- Per questa ragione serve un comparatore (veloce e costoso) per ogni linea
- Cosa accade se non ci sono più linee libere? Per il principio di località (temporale), si dovrebbe eliminare la linea che da più tempo è stata inserita nella cache
- Tuttavia, tenere traccia di questo aspetto (counter + altri comparatori?) e l'elevato numero di comparatori per il confronto con ogni linea le rende poco utilizzate in favore di altre strategie come:
  - Direct-mapped (i bit meno significativi del TAG vincolano univocamente la linea)
  - Set associative, come le precedenti ma sono presenti più linee per ogni possibile opzione aumentando la flessibilità

cache fully associative
↳ cioè sono raggruppate le linee sulla base dei loro indirizzi ⇒ è possibile scorrere solo il gruppo di linee in cui potrebbe essere il tag
alla ricerca del tag
set

21

Cache Direct-mapped

- Gli m bit meno significativi del TAG, denominati INDEX, determinano univocamente la linea corrispondente nella cache
- Indirizzi diversi con lo stesso INDEX collidono → ossia competono. Se un nuovo dato destinato ad un INDEX già occupato viene richiesto dalla CPU ⇒ il nuovo rimpiazza il vecchio
- Processore con 36 bit di indirizzo, 512 linee di cache da 256 byte (cache da 128 K)
- In tal caso, l'INDEX è composto dai m=9 bit meno significativi del TAG (19 bit) → $2^9 = 512$ (linee della cache)
- La decodifica dell'INDEX individua univocamente la linea di cache associata



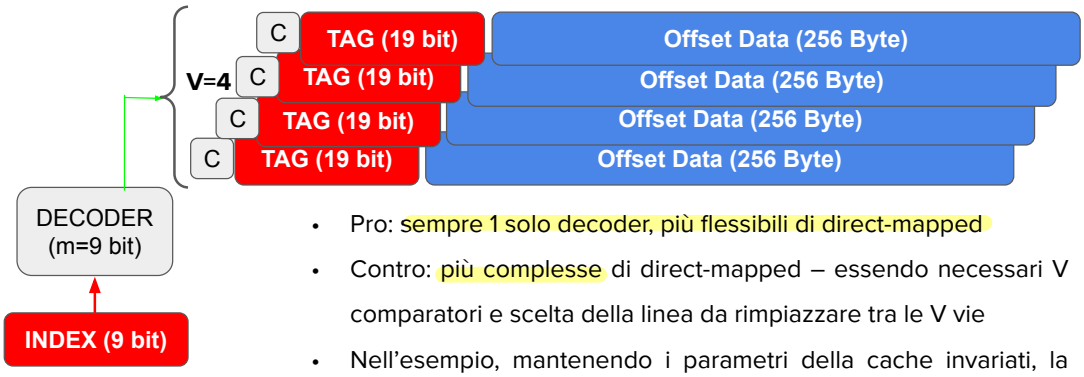
- Pro: 1 solo comparatore (e 1 decoder), non si pone il problema di quale linea rimpiazzare
- Contro: poco flessibili (collisioni sulla stessa linea)

22

Cache Set Associative a V vie

- dalle direct mapped prende l'associazione (analoga) set = index
- dalle fully associative prende il controllo (parallelo) da eseguire per le linee (vie) una volta individuato il set.

- Le cache Set-associative combinano le caratteristiche delle due strategie precedenti
- Come le direct-mapped l'INDEX individua univocamente un set
- Tuttavia, per ogni entry (set) esistono più vie/linee, in totale V (ie, 4 nell'esempio)



- Pro: sempre 1 solo decoder, più flessibili di direct-mapped
- Contro: più complesse di direct-mapped – essendo necessari V comparatori e scelta della linea da rimpiazzare tra le V vie
- Nell'esempio, mantenendo i parametri della cache invariati, la dimensione totale della cache aumenta di un fattore V (ie, 4)

23

Scelta della linea e altre informazioni

- La scelta della linea da rimpiazzare (problema inesistente nel caso directly-mapped) può essere eseguito in accordo a varie strategie. Tra le quali:
  - **Random**: semplice ma sorprendentemente efficace
  - **LRU** (Least Recently Used): basata su un contatore
  - ...
- In realtà, ogni linea di cache contiene anche altre informazioni:
  - **Dirty bit**: indica se la linea è stata modificata, in caso non sia così non è necessario modificare i dati corrispondenti nel livello gerarchico più basso (RAM) nel caso sia necessario rimpiazzare quella linea (perché la RAM non viene aggiornata subito, alla modifica di un dato in cache.)
  - **Invalid bit**: indica se la linea contiene dati validi, utile per evitare di utilizzare dati mai inseriti nella cache e quindi non significativi => 1 = dato utile (caricato dalla RAM). 0 = ciarpame o dato non aggiornato.

24

Parametri che impattano sulla frequenza di miss (le 3 "C")

Compulsory

Il primo accesso a qualunque blocco (linea) di memoria (cold start o first-reference misses)

Capacity

La cache non è in grado di contenere tutti i blocchi necessari all'esecuzione di un programma ovvero il *working set* è maggiore della capacità della intera cache. Ne deriva che un blocco viene scartato per essere poi richiamato subito dopo

Conflict

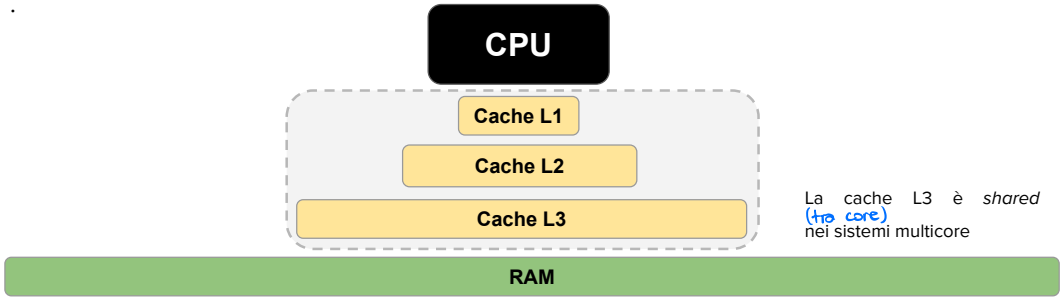
Le restrizioni sulla allocazione in cache (se non fully associative): index uguale per un numero di blocchi contemporaneamente utilizzati superiore alla associatività (*collision misses*). Di fatto si è obbligati a scartare blocchi che potrebbero servire immediatamente dopo

25

Cache a più livelli

- In realtà, in un sistema reale sono presenti all'interno del chip più livelli di caching
- Il livello superiore, più veloce, contiene un sottoinsieme delle informazioni presenti nei livelli inferiori, più lenti
- Possono sorgere problemi di coerenza tra i dati (gestiti con appropriate strategie)

.



La cache L3 è *shared* (tra core) nei sistemi multicore

26

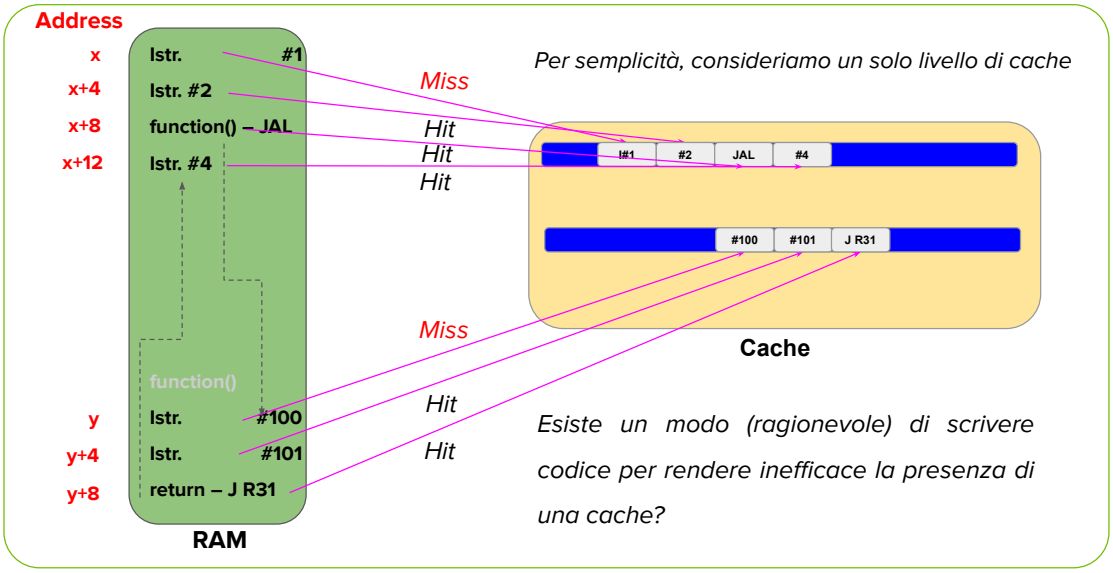
1.2.5. Casi di studio relative all'uso (efficiente) di cache

Caso di studio I: *fetch* del codice 1/2

- Il codice che deve essere eseguito da un sistema di elaborazione è letto dalla memoria a indirizzi consecutivi
- Questo regolare comportamento varia solo in presenza di salti (condizionati o incondizionati come chiamate a funzioni), interrupt o altre circostanze che portano a eseguire codice a indirizzi diversi
- Tuttavia, anche nel caso precedente (eg, chiamata a una funzione), si cambierà l'indirizzo nel quale effettuare il fetch (ie, modifica del PC) si procederà con la lettura del codice della funzione a indirizzi consecutivi
- Per le ragioni evidenziate, il principio di località (spaziale e temporale) risulta valido nell'accesso al codice
- TIpicamente, esiste una cache dedicata al codice separata da quella dei dati

27

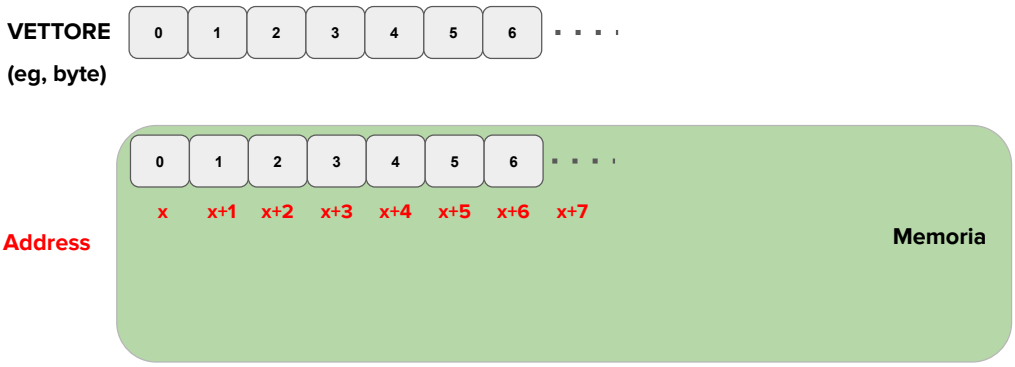
Caso di studio I: *fetch* del codice 2/2



28

Come sono memorizzati i vettori in memoria

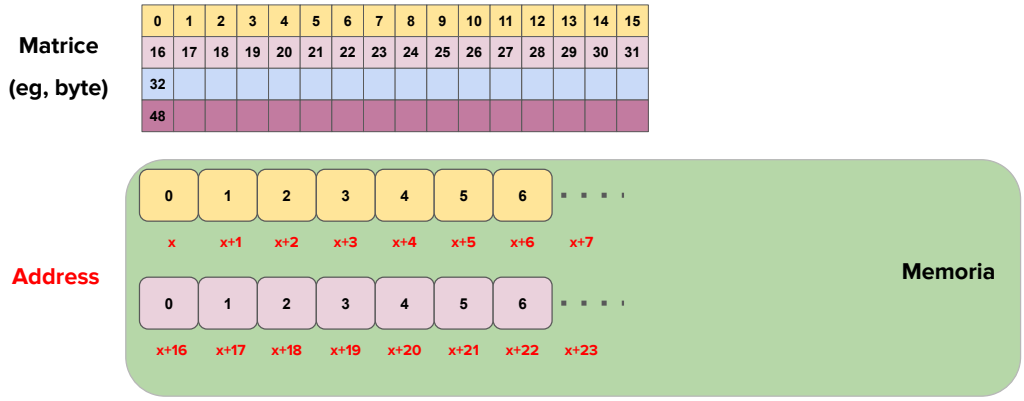
- Elementi consecutivi di un vettore sono memorizzati a indirizzi consecutivi di memoria
- L'allineamento dei dati in memoria è cruciale per ottenere le migliori prestazioni
- In alcuni casi, eg DLX, è indispensabile fare accessi allineati alla memoria



29

Come sono memorizzati le matrici in memoria

- Nel caso di matrici: righe: partendo dalla prima riga, gli elementi di ciascuna riga sono memorizzati a indirizzi consecutivi (*raw major order*)

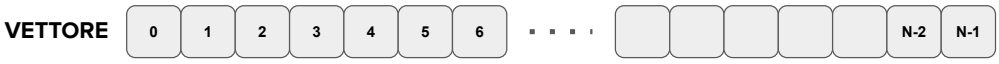


30

Caso di studio II: accesso a un vettore 1/2

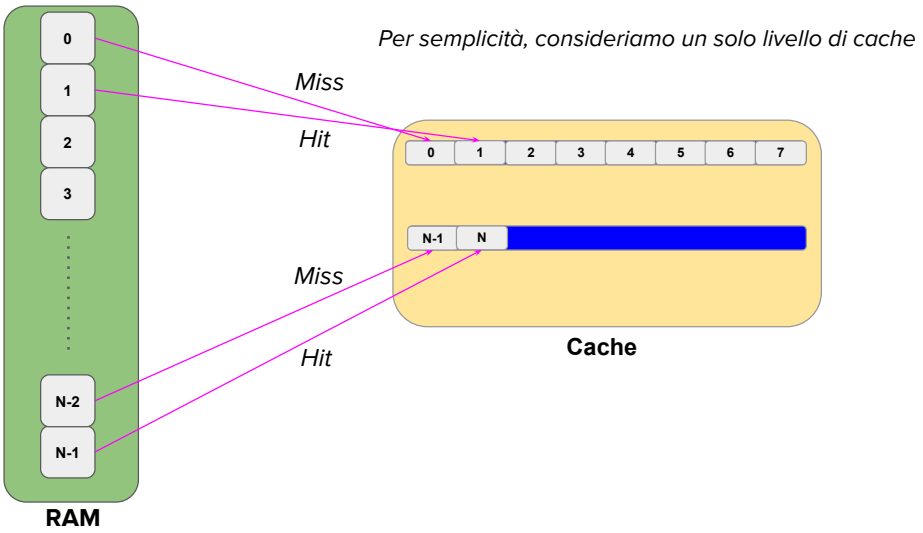
- Spesso, l'accesso a un vettore avviene facendo accessi a elementi consecutivi
- Se questo è il caso, le cache sono molto efficaci
- Si verifica un *miss*, con conseguente line-fill, solo quando si accede a un elemento non presente nella cache
- Nel caso di vettori sarebbe facile scrivere del codice che rende meno efficace lo sfruttamento della cache. Come?

.



31

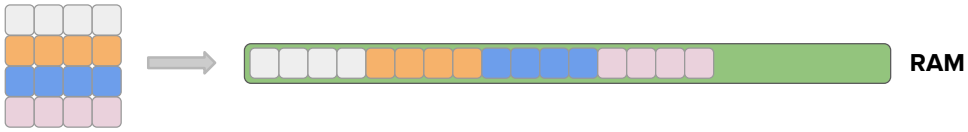
Caso di studio II: accesso a un vettore 2/2



32

Caso di studio III: accesso a una matrice 1/3

- L'accesso a una matrice è ancora più critico e potrebbe facilmente portare a un utilizzo non efficace della cache
- Una matrice è organizzata in memoria come un vettore composto dalla concatenazione di righe consecutive

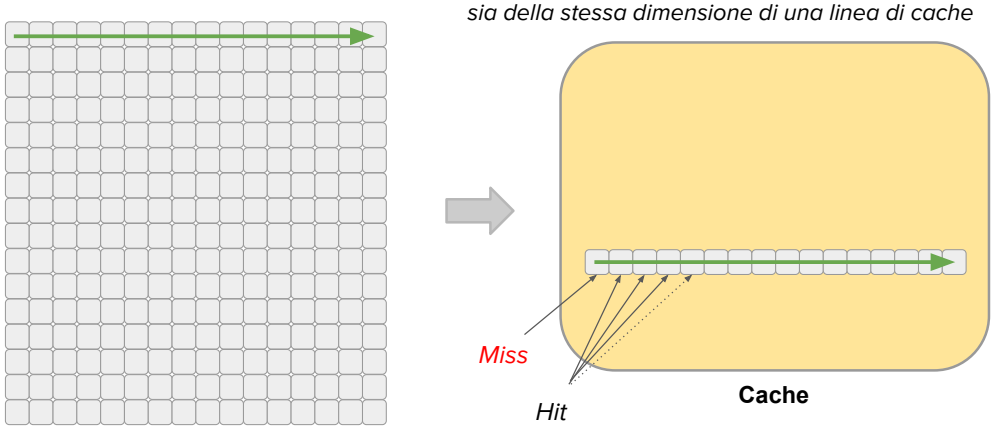


- Gli accessi a elementi nella stesso riga potrebbero* essere compatibili con il principio di località
- Al contrario, accessi a elementi di una stessa colonna, potrebbero* risultare non ottimali per una cache

33

Caso di studio III: accesso a una matrice (per righe) 2/3

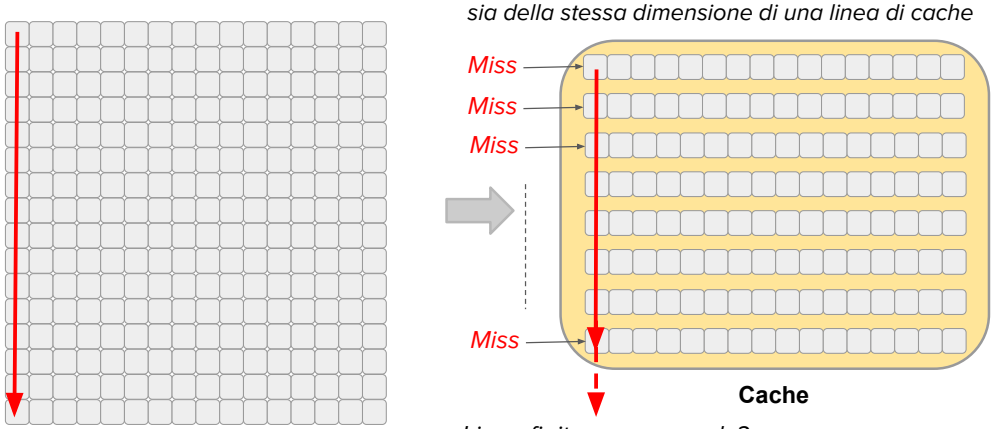
Per semplicità, nell'esempio, si assume che una riga sia della stessa dimensione di una linea di cache



34

Caso di studio III: accesso a una matrice (per colonne) 3/3

Per semplicità, nell'esempio, si assume che una riga sia della stessa dimensione di una linea di cache



Linee finite: cosa accade?

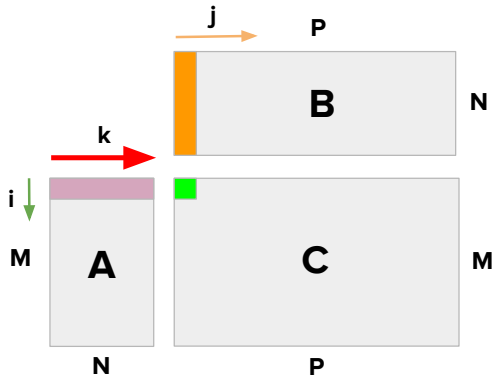
35

Caso di studio IV: moltiplicazione tra matrici 1/9

- La moltiplicazione tra matrici è essenziale in molteplici contesti (eg, uno tra tutti AI)
- Tuttavia, può risultare critica,per le cache quando le matrici hanno grandi dimensioni
- Date due matrici **A**[MxN] e **B** [§NxP], il prodotto **C**[MxN] risulta:

$$\mathbf{C}[i][j] = \sum \mathbf{A}[i][k] \cdot \mathbf{B}[k][j]$$

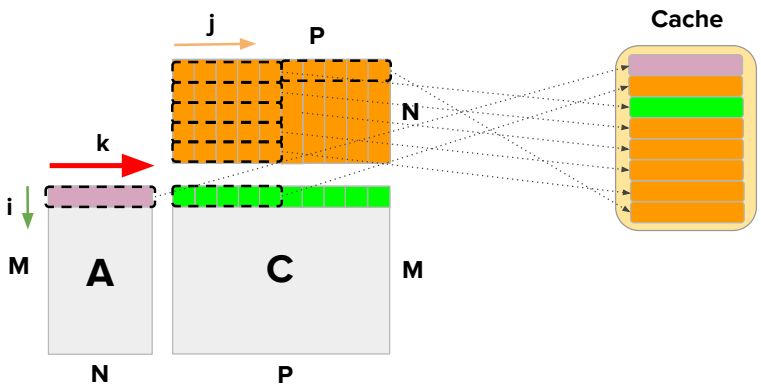
```
for (int i=0;i<M;i++)
 for (int j=0;j<P;j++)
  for (int k=0;k<N;k++)
    C[i][j] += A[i][k]*B[k][j];
```



36

Caso di studio IV: moltiplicazione tra matrici 2/9

- Con riferimento al codice precedente, si verificano continue miss (*capacity*) nel caso non si riesca a memorizzare interamente la matrice B nelle linee di cache



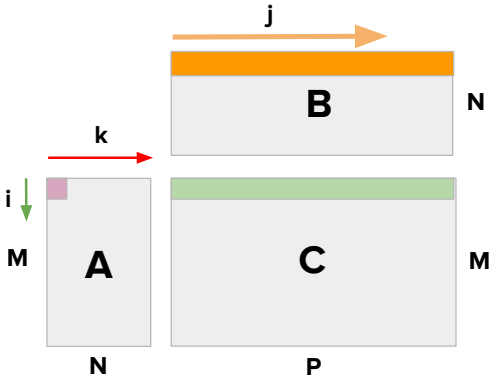
37

Caso di studio IV: moltiplicazione tra matrici 3/9

- Invertendo i due loop più interni (k e j) si calcolano i valori di C progressivamente con una maggiore efficacia nell'utilizzo della cache

```
for (int i=0;i<M;i++)
for (int k=0;k<N;k++)
for (int j=0;j<P;j++)
    C[i][j] += A[i][k]*B[k][j];
```

Perché si fanno più iterazioni lit leggendo gli elementi consecutivi delle righe e una volta per ciclo esterno gli elementi di colonna => più efficiente

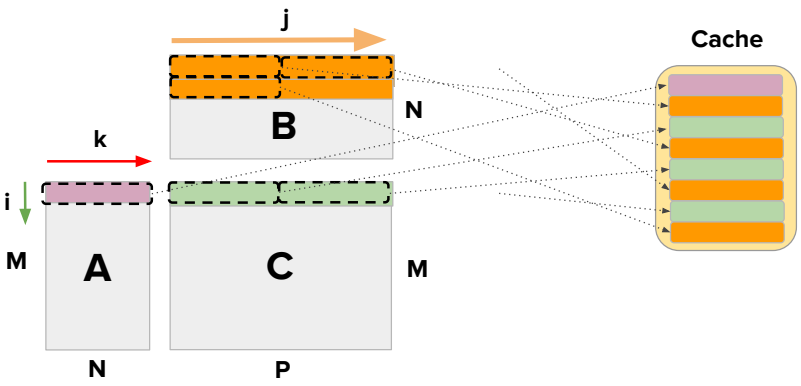


38

Caso di studio IV: moltiplicazione tra matrici 4/9

la matrice è più piccola

- In questo caso, il *working set* è ridotto e ciò migliora l'impatto delle cache



39

Caso di studio IV: moltiplicazione tra matrici 5/9

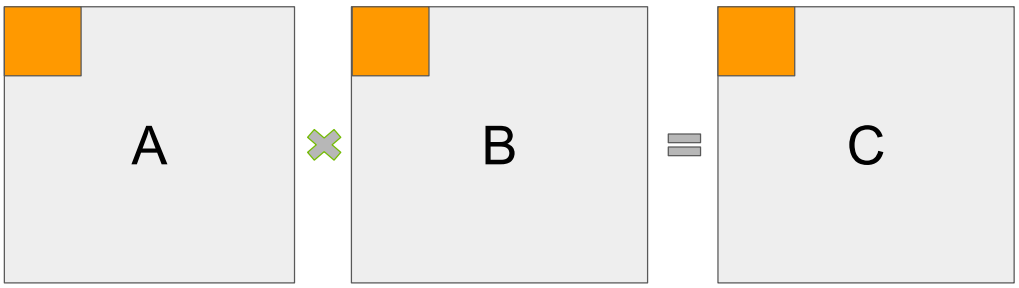
- Benchmark con un MacBook Air M2 – gcc optimization -O0 (ie, no optimization)
- Con vari ordini tra i tre loop utilizzati per la moltiplicazione tra matrici risultano i seguenti tempi di elaborazione :

Loop order	time [sec]
IJK	5.1
IKJ	3.6
KIJ	3.6
KJI	3.7
JKI	3.7
JIK	5.1

40

Caso di studio IV: moltiplicazione tra matrici 6/9

- Una strategia ancora più efficace nella moltiplicazione tra matrici consiste nel ridurre ulteriormente le dimensioni del *working set*
- Mediante una strategia denominata *loop tiling* o *loop blocking* si eseguono calcoli parziali su blocchi di matrici di dimensione limitata che possono sfruttare la meglio la cache



41

Caso di studio IV: moltiplicazione tra matrici 7/9

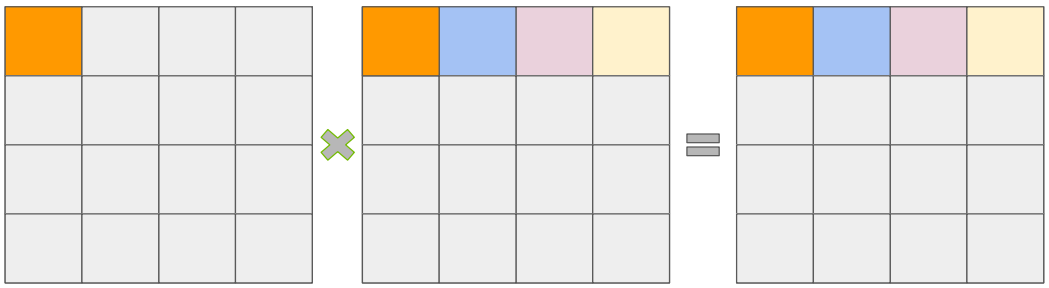
- Esempio di codice che esegue l'elaborazione dividendo le matrici in *tile*/*block*

```
for (int iB=0;iB<M;iB=iB+BLOCK_SIZE_i)
    for (int jB=0;jB<P;jB=jB+BLOCK_SIZE_j)
        for (int kB=0;kB<N;kB=kB+BLOCK_SIZE_k)
            for (int i=iB;i<iB+BLOCK_SIZE_i;i++)
                for (int j=jB;j<jB+BLOCK_SIZE_j;j++)
                    for (int k=kB;k<kB+BLOCK_SIZE_k;k++)
                        C[i][j] += A[i][k]*B[k][j];
```

42

Caso di studio IV: moltiplicazione tra matrici 8/9

- I tre loop sono suddivisi in blocchi di dimensioni tali da rendere più efficace l'utilizzo della/e cache riducendo il working set al solo calcolo all'interno del blocco
- Per esempio, mediante la progressione seguente:



43

Caso di studio IV: moltiplicazione tra matrici 9/9

- MacBook Air con processore Apple M2 – test eseguiti con gcc (optimization -O0)
  - 16 GB RAM
  - Data Cache* L1: 128 KB, linee da 128 byte, set associativa a 8 vie, L2 16 MB shared
- Test eseguito con matrici di interi: **A**=[512x1024], **B**[1024x512] ➔ **C**[512x512]
  - Peggior risultato: ≃ **1.96 s**, configurazione blocchi [i,j,k] = [1,1,1]
  - Miglior risultato: ≃ **0.54 s**, configurazione blocchi [i,j,k] = [128,512,32]
  - Speed-up ≃ **3.63**
- Il *tiling* consente di accedere ai dati in modo più *cache-friendly* anche se questo è solo un aspetto da considerare in determinati contesti
- Per esempio, con SIMD i dati nella matrice B non sono disposti in modo ottimale per questo paradigma di elaborazione

* Dati non ufficiali, valori stimati e resi disponibili da utenti online

44

1.3. Principi di acquisizione di immagini e 3D sensing

Introduction

Cameras are widespread sensing devices used in countless application fields, such as robotics, augmented reality, and autonomous driving, to name a few.

Moreover, the massive processing required by image-based algorithms and networks, in most cases, demands parallel computation techniques.
This issue is further exacerbated when the *framerate* (i.e., the number of images acquired for each second) gets higher (e.g., 30+ frames per second (FPS)).

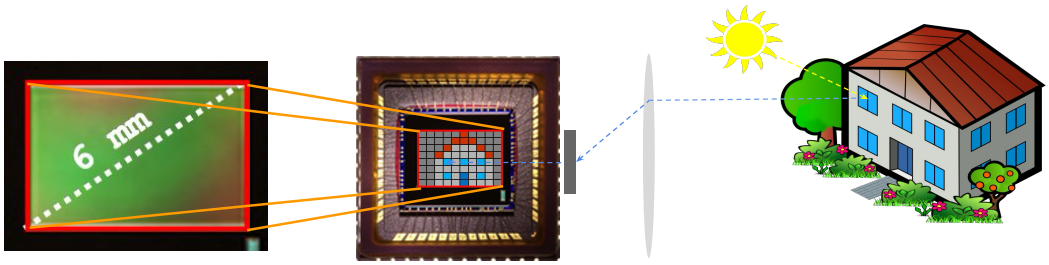
Considering the broad deployment of cameras as ubiquitous sensing devices, this lecture introduces the basic principles of image acquisition and other emerging devices

2

Camera: how this sensing device works

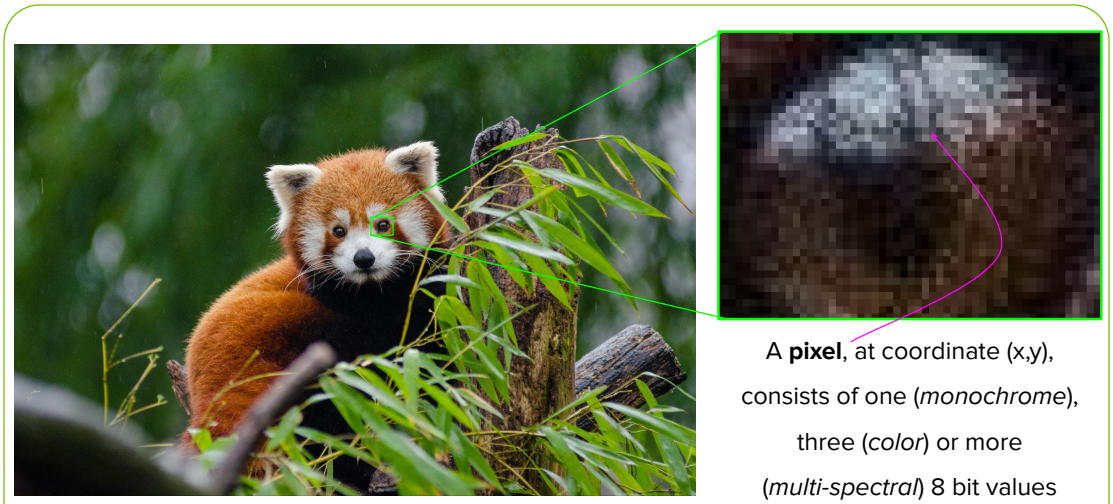
Cameras are popular sensing devices:

- It is somehow an *analog* (actual scene) to *digital* (image) converter
- For each pixel, it gathers through the lens the light emitted by each single point in the scene and converts it into a discrete value, typically, one or more values [255,0]
- Each pixel corresponds to a photosensitive area on a tiny image sensing device
- For instance, the Aptina/ON 1/3" below has a diagonal of about ≈ 6 mm



3

Camera: image and pixels



A **pixel**, at coordinate (x,y), consists of one (*monochrome*), three (*color*) or more (*multi-spectral*) 8 bit values

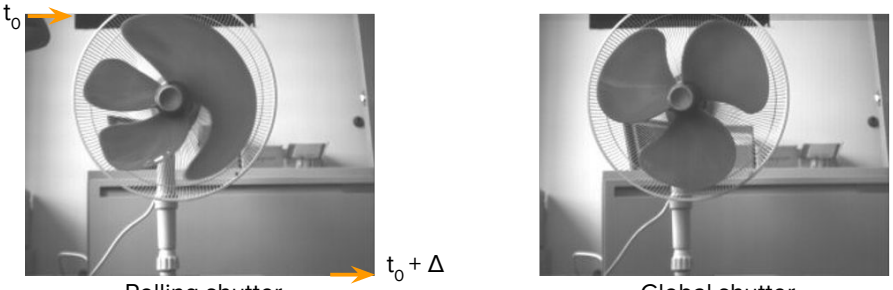
https://www.pexels.com/photo/red-panda-perching-on-tree-during-daytime-146102/

4

Camera: Rolling shutter vs Global shutter acquisition

The acquisition phase can occur according to two technologies/approaches:

- *Rolling shutter*: progressively, from the leftmost top pixel to the rightmost bottom one, generating artefacts in the presence of moving objects and/or *ego-motion*
- *Global shutter*; simultaneously for all pixels
- The latter is now mainstream, while rolling shutter is less popular yet still used

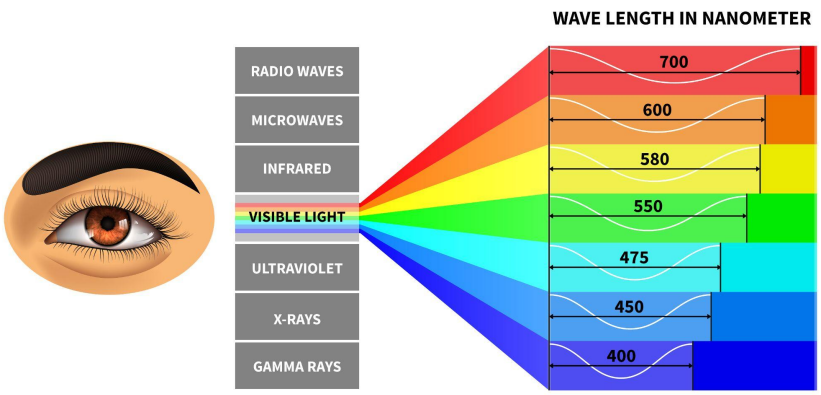


Rolling shutter https://www.flir.com/ Global shutter

5

Camera: the visible spectrum

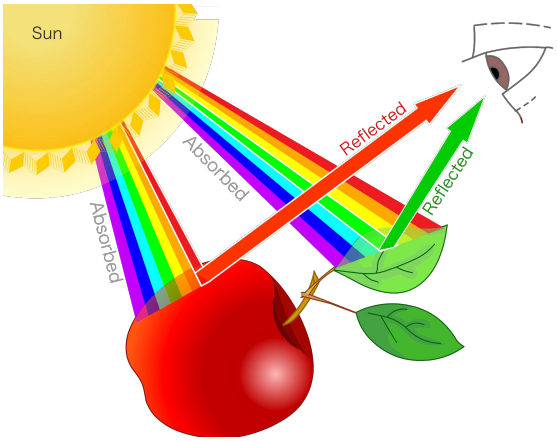
- Electromagnetic signals have a broad spectrum
- However, our eyes can perceive only a tiny subpart of it called the visible spectrum
- Specifically, the visible spectrum ranges from 400 to 700 nanometer



6

Camera: Perception in the visible spectrum

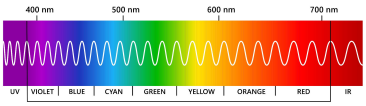
- The color perceived by our eyes is only the component of the visible spectrum reflected by a specific material (eg, red for the apple, green for the leaf)



7

Camera: Greyscale and Color image formation

Image sensors can gather through their photosensitive devices all light components (*greyscale*, or *monochrome*) or some of its bands (3 for color, 4+ for *multispectral* devices)



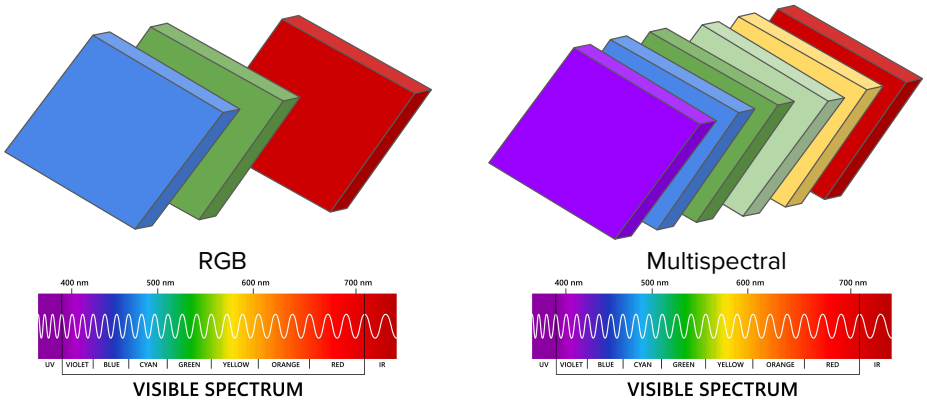
VISIBLE SPECTRUM

http://digital.ni.com/public.nsf/allkb/8AAAE9435D30CEB886256F80007A39AD

8

Camera: Color and Multispectral image formation

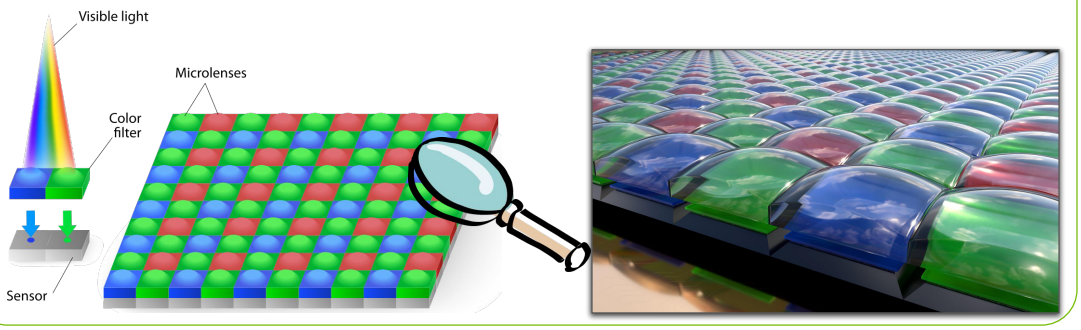
- A multispectral sensing device can gather much more information from the sensed scene than a conventional color camera
- Helpful in many fields such as agriculture, medical imaging, quality inspection, etc



9

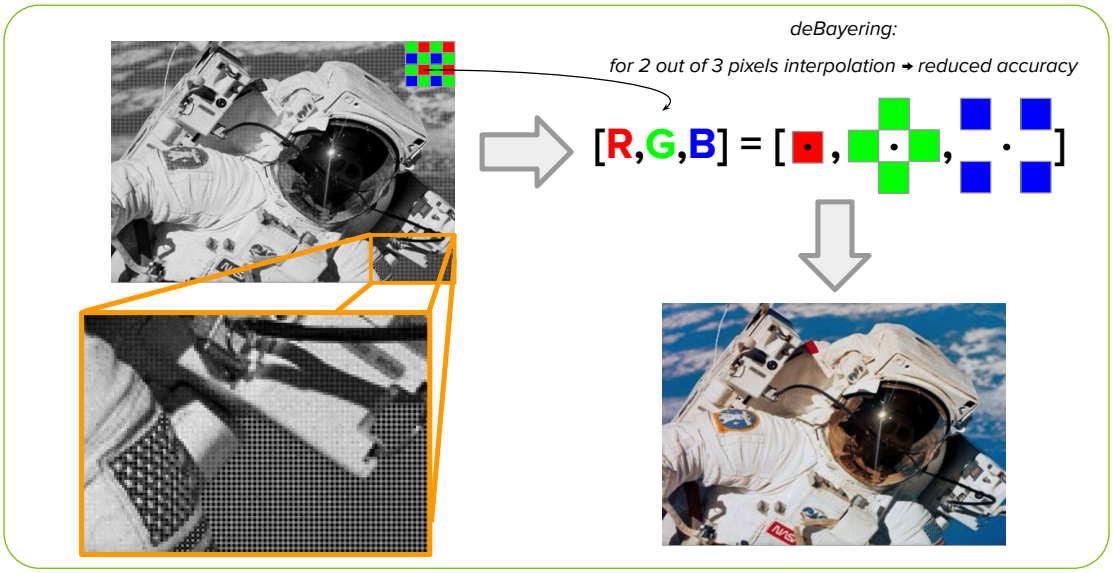
Camera: Bayer filter to sense RGB bands

- Rather than using a single sensing device for each band, in most cases, a single image sensor with microlens is used to gather (approximate) information from multiple portions of the spectrum using color filters for Red, Green and Blue
- The Bayer filter relies on this strategy; hence, 2 out of 3 components of each pixel are always obtained through interpolation of neighbours points with a cyclic scheme



10

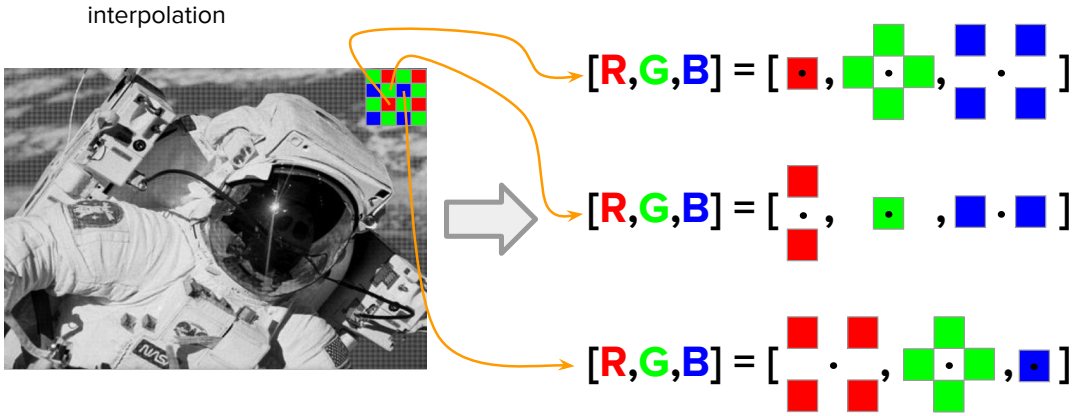
Camera: Bayer filter and conversion from raw data to RGB



11

Camera: Bayer filter and interpolation (ie, deBayering)

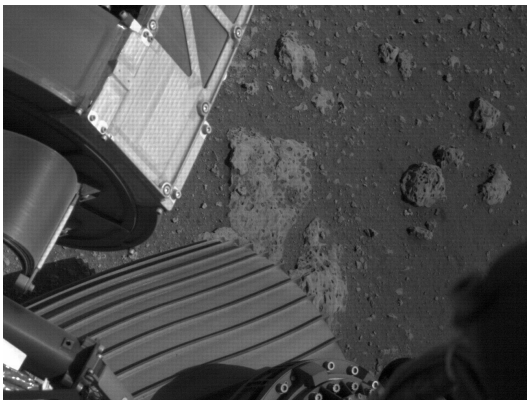
- Below, the three recurrent schemes to infer RGB for each pixel through deBayering
- One of the three RGB values is actually sensed, the other two are obtained through interpolation

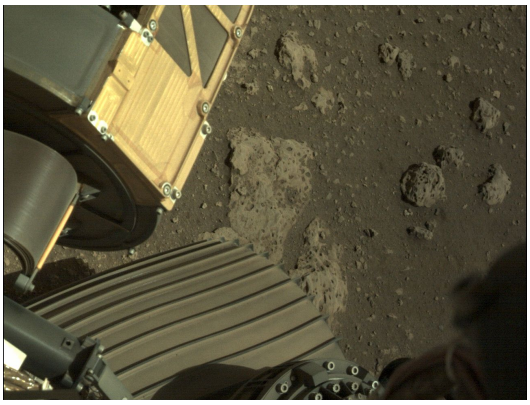


12

Camera: Bayer image example 1/2

- Example of images acquired by rover NASA Perseverance on Mars
- Many images available at: https://mars.nasa.gov/mars2020/multimedia/raw-images/

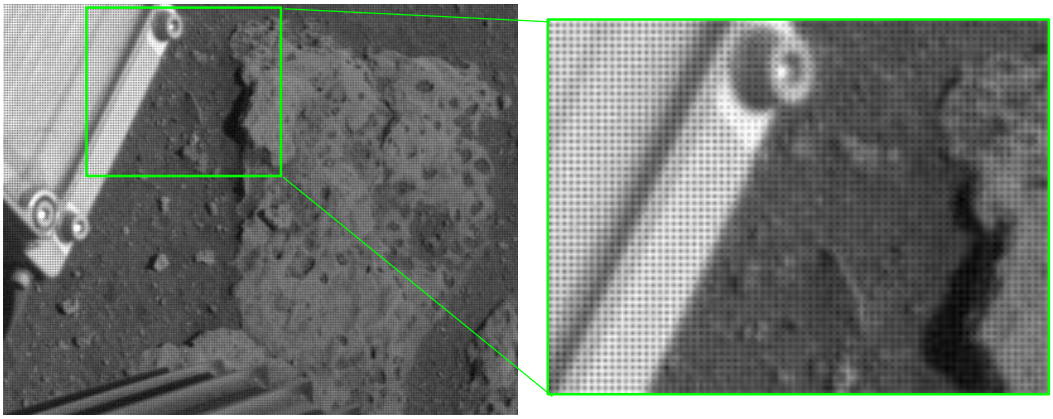




13

Camera: Bayer image example 2/2

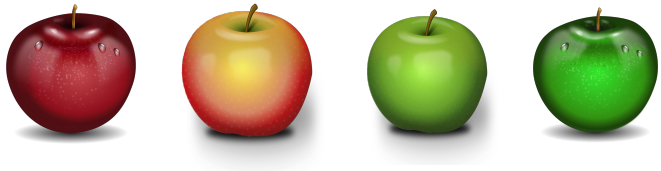
- Zooming allows to better perceive the effect of the Bayer pattern onto the raw image



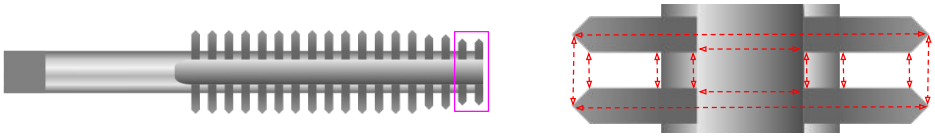
14

Camera: color or monochrome images?

In many applications, the color cue is crucial, for instance, when sorting items like fruit or vegetables according to their ripeness



On the other hand, color is obtained through interpolation; hence, for applications like *metrology,* a monochrome camera might be a more appropriate choice



15

Exercise: DeBayering

Exercise: describe how to perform image deBayering (demosaicing) using only integer data and arithmetic/logic operations.

For this purpose, we need to average two or four pixels, each in the range [255,0]:

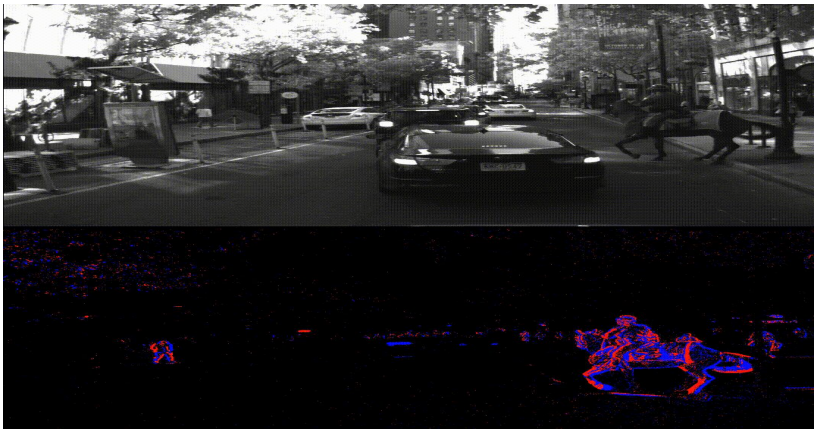
i) two pixels: 67, 188

ii) four pixels: 67, 188, 245, 177

16

Event cameras 1/4

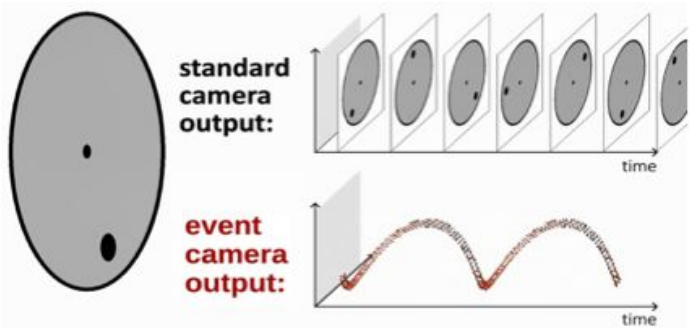
- In recent years, *event cameras* emerged as a novel image sensing paradigm
- These devices continuously capture brightness change in the sensed scene



17

Event cameras 2/4

- The different behaviour of conventional and event cameras is described by this video:



https://www.youtube.com/watch?v=LauQ6LWTkxM&t=30s

18

Event cameras 3/4

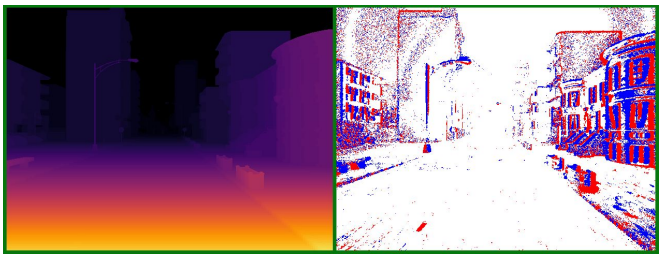
- The event stream consists of a binary information: brightness increase and decrease
- Compared to conventional imaging devices:
  - information is gathered and delivered at microsecond resolution
  - not affected by motion blur
  - much higher dynamic range
  - low power

https://rpg.ifi.uzh.ch/docs/scaramuzza/2019.07.11_Scaramuzza_Event_Cameras_Tutorial.pdf

19

Event cameras 4/4

- Despite the much lower data density, learning based methods enabled to face several downstream tasks, such as depth perception and pose estimation, potentially at microsecond resolution

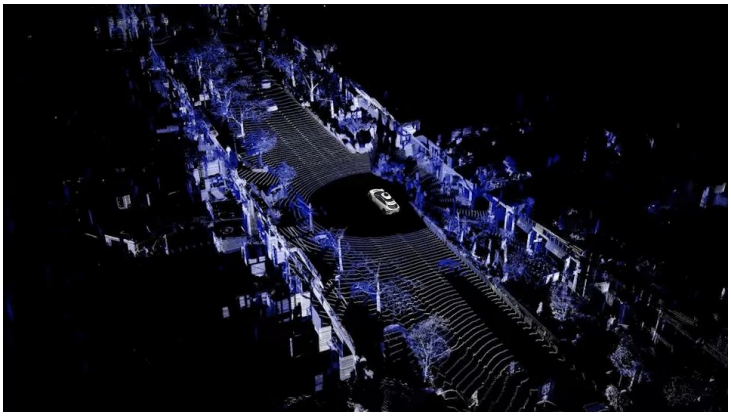


- The only constraints are:
  - the sparse data available vs traditional cameras
  - the computational requirements

20

3D sensing

- 3D sensing aims at inferring the geometry of the sensed scene to obtain a *point cloud*
- It is pivotal for autonomous navigation, augmented reality and robotics, etc

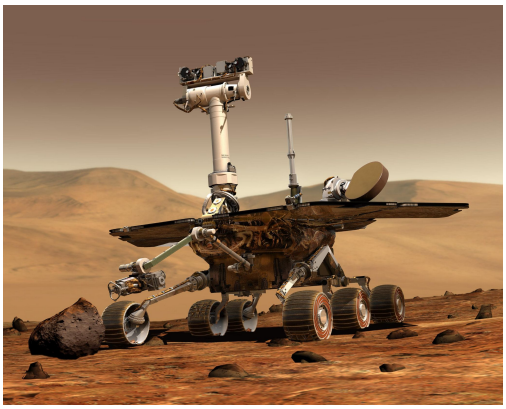


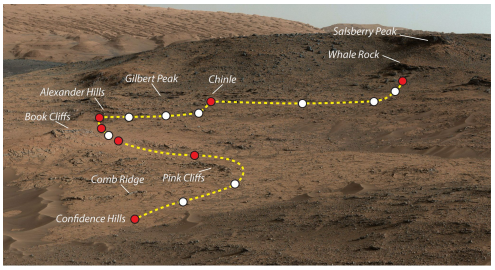
https://waymo.com/

21

3D sensing with cameras (standard and event)

Conventional and RGB cameras enable 3D reconstruction through different strategies (eg, stereo vision, monocular depth perception, etc)

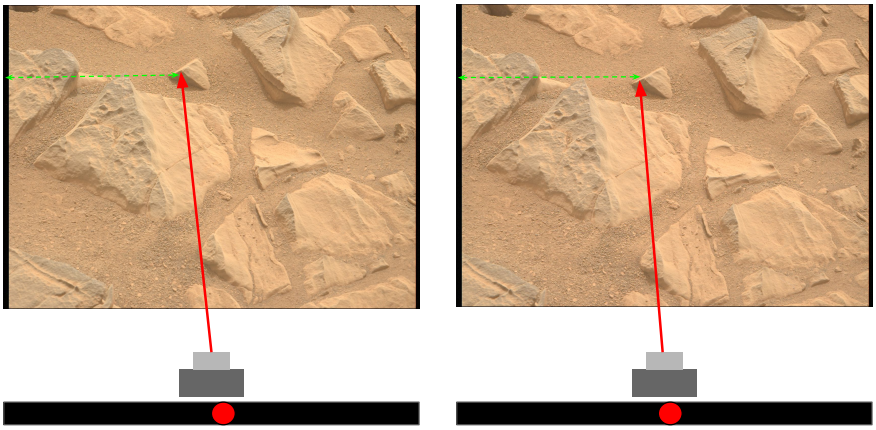




22

3D sensing through stereo vision 1/2

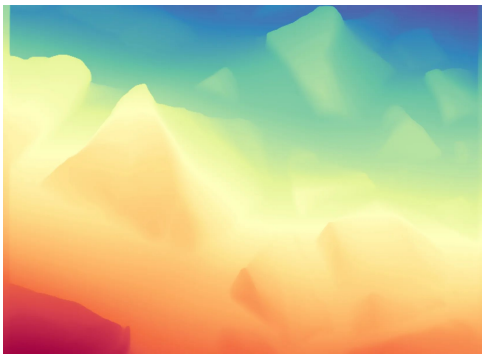
- Cameras enable 3D reconstruction through different strategies (eg, stereo vision)



23

3D sensing through stereo vision 2/2

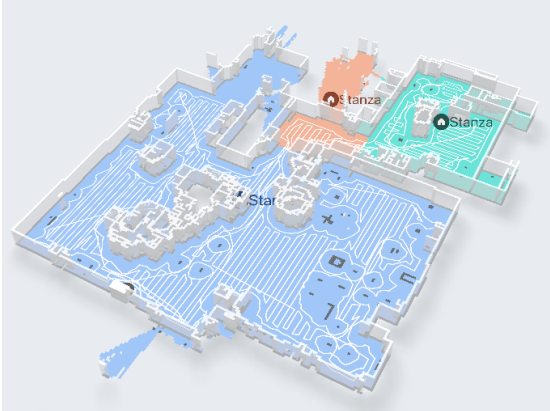
Once located corresponding points, a point cloud can be obtained from the disparity map shown at the right



24

3D sensing in consumer devices

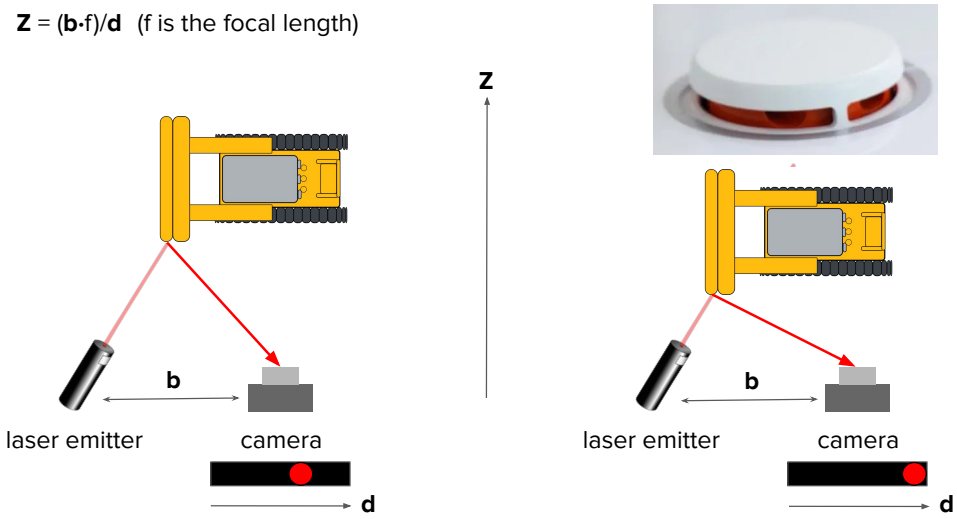
A well known deployment of 3D sensing in consumer devices: vacuum cleaners



25

Depth perception through active stereo triangulation

- **Z** = (**b**·f)/**d** (f is the focal length)



26

Depth sensors

- Although cameras are cheap and have high resolution, they have some limitations
- 3D geometry of a scene can be obtained even deploying specialized sensors
- Most of them rely on the time-of-flight principle
- Used in notable application fields

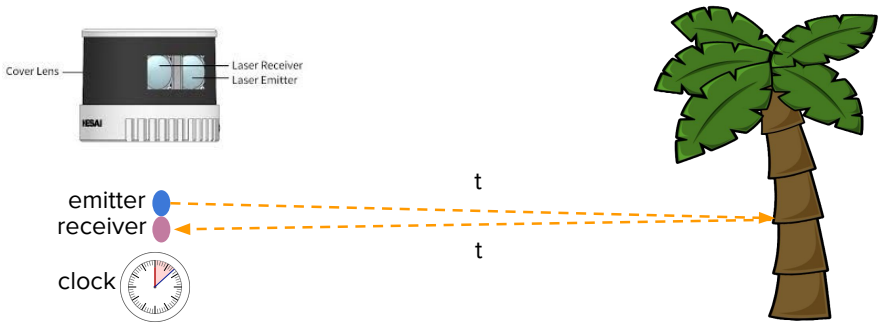
https://apple.com/

https://waymo.com/

27

Depth perception through time-of-flight

- The speed of light is about 300000 m/s and it takes about 3.34 ns to travel for 1 meter
- Distance can be estimated measuring the time to bounce from emitter to receiver
- Distance **Z = v·t**
- Emitter and receiver can be designed according to different technologies
- For instance, laser emitter and receiver



28

1.4. Numeri reali (Floating e Fixed Point e nuovi tipi)

Introduction

- Before dealing with strategies to manage real numeric values, this section introduces/recalls some basic concepts about precision and accuracy
- Then, we'll introduce the two primary methodologies for encoding and processing data representing real numeric values: fixed-point and floating-point
- For the latter, we'll also introduce widely used yet not standardized formats and issues related to the deployment of floating-point data to increase efficiency and memory footprint
- Finally, we'll consider techniques to perform computations involving computations with real numbers using only basic arithmetic and logic operations with integer data

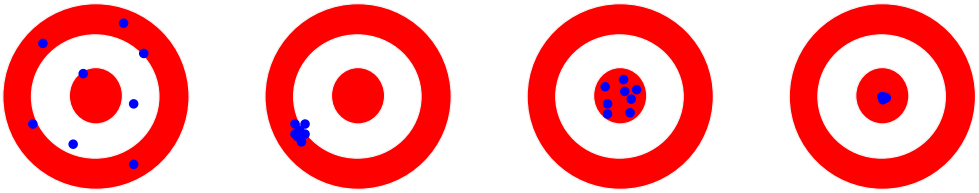
2

Precision and Accuracy

Before dealing with real numbers encoding, let's recall ***precision*** and ***accuracy***

- Precision quantifies how well a measuring device can obtain consistent values of the same entity across different measurements (*repeatability*) Precision = consistenza tra diverse misurazioni (ripetibilità)
- Another relevant concept is accuracy; it quantifies how well we can approximate the actual value with a specific data encoding or the capability of a particular sensing device

Accuracy = quanto una misura si avvicina al valore reale.

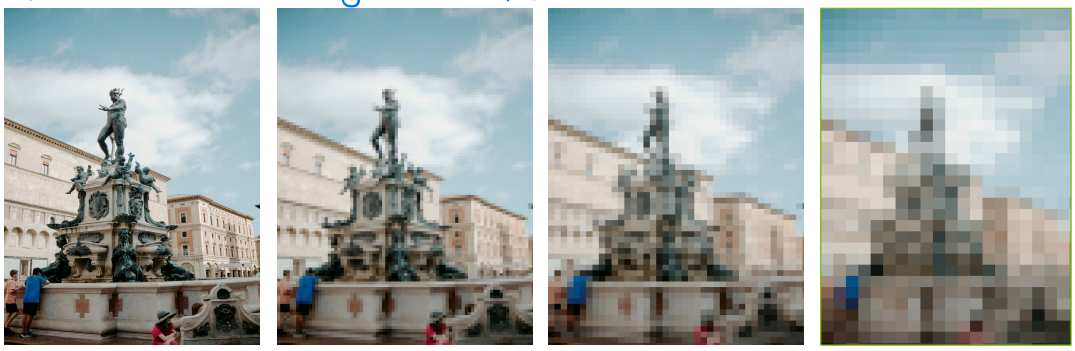


Low Precision ✗	High Precision ✓	Low Precision ✗	High Precision ✓
Low accuracy ✗	Low accuracy ✗	High accuracy ✓	High accuracy ✓

3

Camera: Accuracy (Resolution)

Per le telecamere: accuracy = risoluzione (larghezza x altezza e DPI)



Different imaging sensors can frame the same scene with various amounts of details (accuracy) according to their resolution (ie, the smaller the pixel, the higher the accuracy)

Credits: Picture by Emre Akyol – https://www.pexels.com/photo/fountain-of-neptune-in-bologna-17647393/

4

Camera: Precision (Repeatability)

However, the same imaging sensor can frame a scene from the same position with the same settings, generating different (unprecise) outcomes for the same pixel over time

Credits: Picture by Emre Akyol – https://www.pexels.com/photo/fountain-of-neptune-in-bologna-17647393/

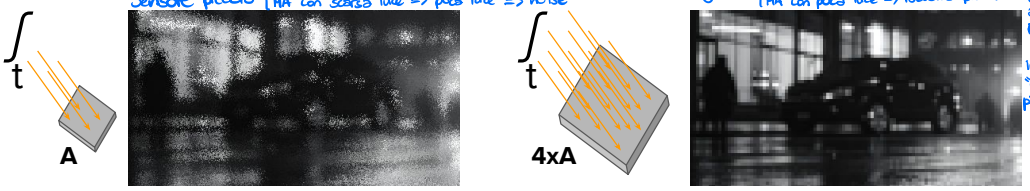
5

Camera: how pixel size can affect precision and accuracy

- The spatial resolution (ie, the number of photosensitive elements present in the imaging sensor), which sets the accuracy, is often considered the crucial parameter
- However, the camera acquisition process relies on the acquisition of light photons
- The larger is the pixel, the higher is the light gathered through the photosensitive device
- Thus, in low light conditions, smaller sensing devices would receive fewer photons than a device with larger ones, which may lead to a noisier image, reducing the *precision* (ie, increasing the noise) despite the potentially higher *accuracy* (ie, smaller pixels)

Sensore piccolo { => teoricamente più accurato (più pixel)
MA con scarsa luce => poca luce => noise

sensore grande { => teoricamente meno preciso (meno pixel)
MA con poca luce => raccolta più luce (rispetto a sens. più piccolo)
⇓
immagine "migliore", più accurata



https://stockcake.com/i/rainy-city-lights_461935_96232

6

Dealing with real numbers

The need for real numbers

- Integer data types and computations are helpful, but managing real numbers is pivotal for many scientific purposes and other tasks, such as gaming
- Most applications, including AI, heavily rely on real data, encompassing integers, fractions, and irrationals. Hence, a representation of such data (even if in an approximate format) and the ability to perform mathematical computations is crucial
- Nonetheless, performing computation using real data type (eg, with floating-point encoding) is generally expensive and requires many bits to encode such data
- Therefore, we also introduce a less popular yet helpful data representation that can speed up computations in some relevant circumstances.

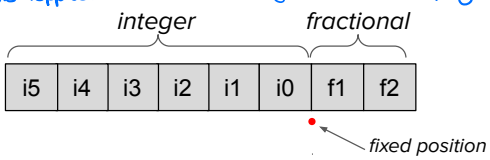
7

Fixed-point: introduction 1/2

Fixed-point representation assumes that we can encode numbers with a constant number of digits for the integer and fractional part with the **decimal point in a fixed position**

Posizione della virgola, nella rappresentazione binaria, fissa (decisa dal programmatore)

→ trade-off tra precisione decimale (più bit per la parte decimale) e possibilità di rappresentare numeri più elevati (più bit per la parte intera)



For instance, 0.75 and 1.5 can be encoded (unsigned, 6+2 digits) as:

$$000000.11_2 = 0\cdot2^5 + 0\cdot2^4 + 0\cdot2^3 + 0\cdot2^2 + 0\cdot2^1 + 0\cdot2^0 + 1\cdot2^{-1} + 1\cdot2^{-2} = 0 + ½ + ¼ = 0.75_{10}$$

$$000001.10_2 = 0\cdot2^5 + 0\cdot2^4 + 0\cdot2^3 + 0\cdot2^2 + 0\cdot2^1 + 1\cdot2^0 + 1\cdot2^{-1} + 0\cdot2^{-2} = 1 + ½ = 1.5_{10}$$

For unsigned values, the data range with 6+2 digits is [0, +63.75]

For signed values, the data range with 6+2 digits is [-32.0, +31.75]

8

Fixed-point: introduction 2/2

- Not surprisingly, with this specific encoding (two fractional digits), 0.20_{10} would be approximated to 0.25_{10} (ie, 0.01_2), and 0.1_{10} would be approximated to 0.0_{10}, being zero the closest number
- The major limitation of fixed-point encoding is the limited data range that spans approximately (signed numbers) $[-2^K, = +2^K]$ with k integer digits
- For these reasons, encoding real numbers with a *floating* decimal point position is the preferred solution, even if approximation issues persist
- Does it mean that fixed-point numbers are useless? In some circumstances, they can be valuable in replacing expensive computations with real numbers with more lightweight integer operations, as will be discussed later
- Moreover, compared to floating-point, it allows a constant accuracy within its range, although such range is much smaller using the same number of bits

9

Floating-point: introduction 1/2

- Despite the advantages previously outlined, fixed-point data is seldom deployed in practice, mainly due to its limited dynamic range
- A more effective and ubiquitous method to represent real values relies on the floating-point paradigm
- First of all, we recall that a number is said in *normalized scientific notation* if it contains a single integer digit, different from zero, and a limited (eg, ½) or unlimited number (eg, ⅓) of decimal digits
- These are some examples (using base 10):
  - -2.3443434 normalized scientific format
  - $7.26752 \cdot 10^{23}$ normalized scientific format
  - 0.3310 not in normalized scientific format
  - $16.4510 \cdot 10^{-3}$ not in normalized scientific format

10

Floating-point: introduction 2/2

In contrast to fixed-point, for which the decimal point is in a fixed position, it can move in floating-point according to an exponent term embedded into the same encoding format

Hence, with floating-point, a numeric value N **in scientific format*** is encoded as:

ovvero essere normalizzata (in bin)

$$\mathbf{N = (-1)^{S} \cdot VAL_{BASE} \cdot BASE^{EXP}}$$

BASE = base numerica della rappresentazione (binario = 2, dec. = 10, ...)
VAL = mantissa = cifre effettive del numero ("parte significativa"), deve iniziare con 1
EXP = esponente, corrisponde movimento della virgola (>0 => numero elevato, <0 => numero piccolo)

Es. $1{,}234 = (+1) \cdot 1{,}234 \cdot 10^{0}$

with **S** the sign (a single digit set to 1 or 0), **VAL** the numeric value encoded according to parameters **BASE** and **EXP**. The two latter terms, **VAL** and **EXP**, are encoded with a certain number of digits called ***mantissa*** and ***exponent***.

For instance, the *float* format uses BASE=2, 23+1 bit for VAL and 8 bit for EXP

* not true for **subnormals** (or denormalized) values

11

Floating-point: encoding approximation issues 1/3

Let's consider the numerical value $\mathbf{10.75}_{10}$

We can represent it in the scientific format as: $\mathbf{1.075}_{10}\cdot\mathbf{10}^{1}$

With a binary encoding, the same number is: $\mathbf{1010.11}_{2}\cdot\mathbf{2}^{0}$

Hence, its representation in scientific format with 6 bit is: $\mathbf{1.01011}_{2}\cdot\mathbf{2}^{3}$ = $(1 + 1/4 + 1/16 + 1/32)\cdot 2^{3}$

Similarly, $\mathbf{0.125}_{10}$ in scientific format is $\mathbf{1.25}_{10}\cdot\mathbf{10}^{-1}$

Its binary version is $\mathbf{0.001}_{2}$ in scientific format is $\mathbf{1.00000}_{2}\cdot\mathbf{2}^{-3}$

Regardless of the representation, the **number of available digits is limited**, and despite some *lucky* cases, like those reported before, many numerical values cannot be encoded without approximation/rounding. Indeed, increasing the number of binary digits can yield a better approximation but not solve the problem as discussed next.

12

Floating-point: encoding approximation issues 2/3

Let's consider the numerical value $\mathbf{1.21}_{10}$, already in scientific format using a decimal encoding and a single fractional decimal digit. Its (approximated) binary representation with 8 fractional binary digits is:

$\mathbf{1.00110101}_{2} = 1 + 0\cdot2^{-1} + 0\cdot2^{-2} + 1\cdot2^{-3} + 1\cdot2^{-4} + 0\cdot2^{-5} + 1\cdot2^{-6} + 0\cdot2^{-7} + 1\cdot2^{-8} = \mathbf{1.20703125}_{10} \neq \mathbf{1.21}_{10}$

or (with a better approximation/rounding):

$\mathbf{1.00110110}_{2} = 1 + 0\cdot2^{-1} + 0\cdot2^{-2} + 1\cdot2^{-3} + 1\cdot2^{-4} + 0\cdot2^{-5} + 1\cdot2^{-6} + 1\cdot2^{-7} + 0\cdot2^{-8} = \mathbf{1.21093750}_{10} \neq \mathbf{1.21}_{10}$

1.0011 1111 10

→ i due numeri (successivi) non rappresentano correttamente il nostro target (uno è < e uno >) => rappresentazione approssimativa

We can increase the number of digits to get better-approximated results; however, we can only represent it with errors even by increasing the number of fractional digits to 23 or more. Of course, the higher the number of digits, the better the approximation, but the error persists when we can't represent a number with a finite number of power of 2 terms.

13

Floating-point: encoding approximation issues 3/3

Extending the number of fractional binary digits from 8 to 23 bit, the numerical value $\mathbf{1.21}_{10}$ is:

$\mathbf{1.00110101110\ 000101001000}_2 = 1.21000003814697265625_{10} \neq \mathbf{1.21}_{10}$

↳ Anche aumentando il numero di bit la rappresentazione rimane lacunante.

With **8** fractional bits, the absolute approximation error is:

$\mathbf{1.21}_{10} - \mathbf{1.21093750}_{10} = -0.00093750_{10}$

errore ridotto MA presente

With **23** fractional bits, the absolute approximation error is:

$\mathbf{1.21}_{10} - \mathbf{1.21000003814697265625}_{10} = -0.00000003814697265625_{10}$

We get a much better approximation with 23 fractional bits, but we cannot encode exactly a *simple* numerical value like 1.21_{10}. This evidence is the first critical aspect to consider when using floating-point: **most numbers are just a rounded approximation to the closest available encodable representation.**

14

Floating-point: some encoding examples, positive exponent

ULP = distanza di un numero rappresentabile dal computer e il suo successivo.

- fixed point → ULP costante
- floating point → ULP variabile (in base a EXP)

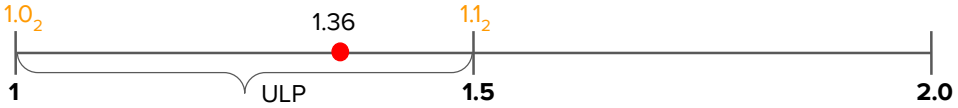
Let's consider $\mathbf{2.00}_{10}$; in binary format, $\mathbf{10.0}_2$. In scientific format, with **3 fractional digits**, $\mathbf{2.00}_{10}$ is: $\mathbf{1.00}\mathbf{0}_2 \cdot \mathbf{2^1}$. The red digit represents the *Unit in the Last Place* (**ULP**), which affects accuracy.

- Hence, the largest number we can represent in scientific format with 3 fractional digits and **exponent 1** is: $\mathbf{1.11}\mathbf{1}_2 \cdot \mathbf{2^1} = \mathbf{1.875}_{10} \cdot \mathbf{2^1} = \mathbf{3.75}_{10}$ (equal to 2^2-ULP$\cdot 2^1$)
- With the next **exponent 2**, we can encode values from $\mathbf{4}_{10}$ ($\mathbf{100.0}_2$), in scientific format $\mathbf{1.00}\mathbf{0}_2 \cdot \mathbf{2^2}$, to $\mathbf{1.11}\mathbf{1}_2 \cdot \mathbf{2^2} = \mathbf{1.875}_{10} \cdot \mathbf{2^2} = \mathbf{7.5}_{10}$ (equal to 2^3-ULP$\cdot 2^2$)
- With the next **exponent 3**, we can encode values from $\mathbf{8}_{10}$ ($\mathbf{1000.0}_2$), in scientific format is $\mathbf{1.00}\mathbf{0}_2 \cdot \mathbf{2^3}$, to $\mathbf{1.11}\mathbf{1}_2 \cdot \mathbf{2^3} = \mathbf{1.875}_{10} \cdot \mathbf{2^3} = \mathbf{15.0}_{10}$ (equal to 2^4-ULP$\cdot 2^3$)
- Interestingly, with **exponent 4** (values ranging from $\mathbf{1.00}\mathbf{0}_2 \cdot \mathbf{2^4}$ (**16**) to $\mathbf{1.11}\mathbf{1}_2 \cdot \mathbf{2^4} = \mathbf{1.875}_{10} \cdot \mathbf{2^4} =$ **30** (equal to 2^5-ULP$\cdot 2^4$)) even integer values (ie, 17, 19, 21, 23, 25, 27, 29, 31) cannot be encoded without rounding error ($|½ \cdot \text{ULP} \cdot 2^4| = 1$)
- Hence. accuracy can be improved by increasing the mantissa (ie, number of fractional bits)

15

Floating-point vs Fixed-point and rounding error 1/10

Let's consider the numerical value **1.36**$_{10}$ and a single bit for the fractional part



Should we encode it as $1{,}0_2$ or 1.1_2?

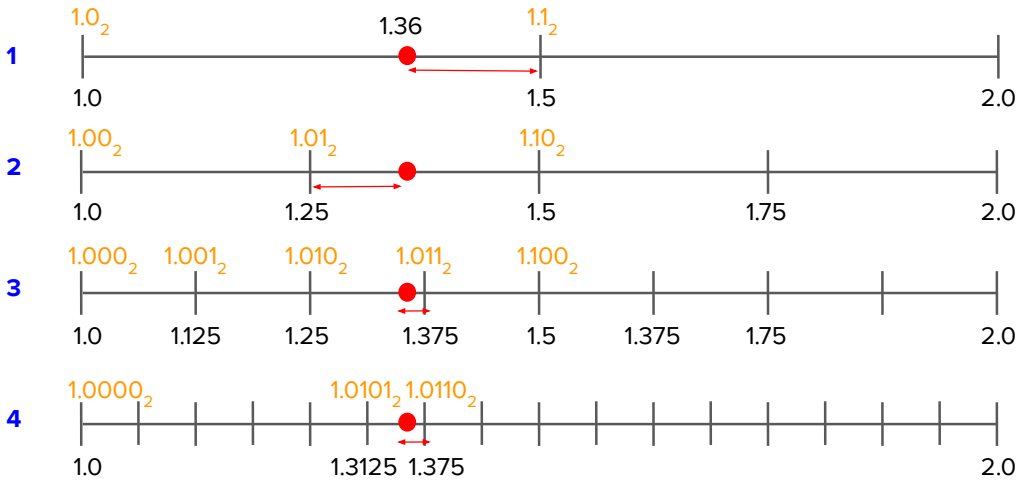
Of course, looking at the picture, we'd say 1.1, since 1.36 is closer to 1.5 than 1.0 and consequently the relative error is lower for 1.5 (ie, |1.5-1.36|/1.36 ≃ 0.10 < 0.26 ≃ |1.0-1.36|/1.36).

Given a specific binary encoding, the error should be as low as possible (eg, smaller than half of the weight given by the ULP (Unit in the Last Place) that for a single bit for the fractional part is ½·0.5. In the example, similar considerations apply to fixed and floating-point representations.

16

Floating-point vs Fixed-point and rounding error 2/10

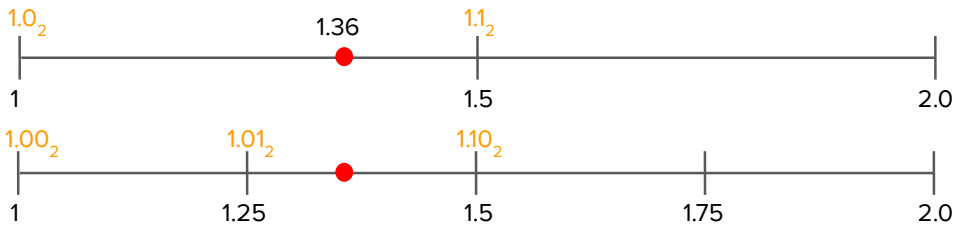
Let's consider again the numerical value **1.36**$_{10}$ with different numbers of fractional digits:



17

Floating-point vs Fixed-point and rounding error 3/10

The most significant error is half the weight assigned to the ULP (0.25 using 1+1 digits and 0.125 using 1+2 digits).



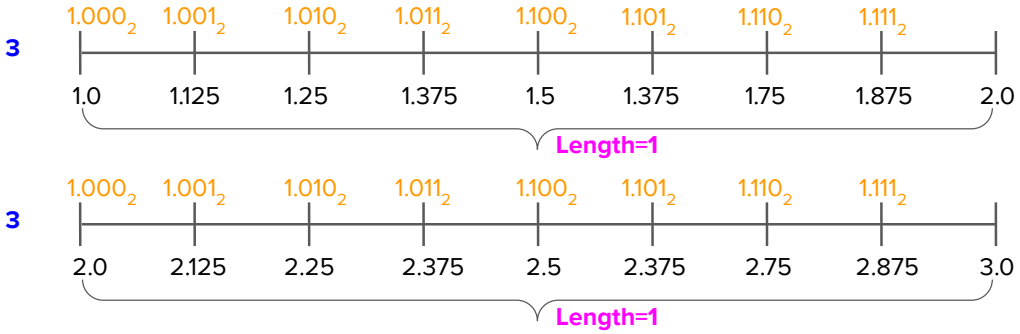
But, what happens when the value is in between (eg, 1.375, using 2 decimal digits?)

The same considerations apply to fixed-point and floating-point. However, there is a major difference in behaviour between the two, as discussed next.

18

Floating-point vs Fixed-point and rounding error 4/10

With **fixed-point**, intervals of the same length are always split into portions of the same size. For instance, with 3 fractional bits, intervals of the same length (eg, [1,2] and [2,3]) are always split into 8 portions of the same size, as depicted below.



Consequently, the rounding error is constant regardless of the considered interval range.
perché le unità sono suddivise in (# bit decimali) parti uguali

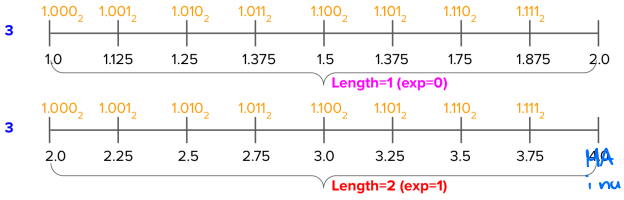
19

Floating-point vs Fixed-point and rounding error 5/10

In contrast, with **floating-point**, an interval is always split into the same number of portions, but the length of the intervals changes according to the exponential value.

With binary encoding and 3 bits for fractionals, we can encode numeric values falling between* [1, 1.875] and [2, 3.75], respectively, with exponent 0 and 1.

il numero di "decimali" rappresentabile è lo stesso



a causa del modo in cui sono rappresentati meri in floating point la granularità dipende dall'esponente: mantissa ed esponente hanno a disposizione sempre lo stesso numero di bit, MA la quantità di numeri coperti all'aumentare dell'esponente aumenta esponenzialmente (con EXP=1 $[2^1, 2^2]=[2,4]$, ma con EXP=9 $[2^9, 2^{10}]=[512, 1024]$)

The impact of ULP increases with the magnitude of the number according to the exponent

20

Per capire meglio...

In floating point il numero di bit di mantissa (VAL) ed esponente (EXP) non cambia,
MA la granularità dei numeri rappresentabili dipende dall'esponente.

↓

perché aumentando o diminuendo l'esponente
è possibile coprire intervalli più o meno grandi.

esempio:

→ con EXP=0 va coperto l'intervallo $[2^0, 2^1] = [1, 2]$

⇝ con 3 bit di mantissa si ha $ULP = \frac{2-1}{2^3} = \frac{1}{8} = 0.125$

→ con EXP=10 va coperto $[2^{10}, 2^{11}] = [1024, 2048]$ ← intervallo molto più grande

⇝ con 3 bit di mantissa si ha $ULP = \frac{2048-1024}{2^3} = \frac{1024}{8} = 128$ ⇒ bassissima precisione

↓

si saltano numeri interi completamente

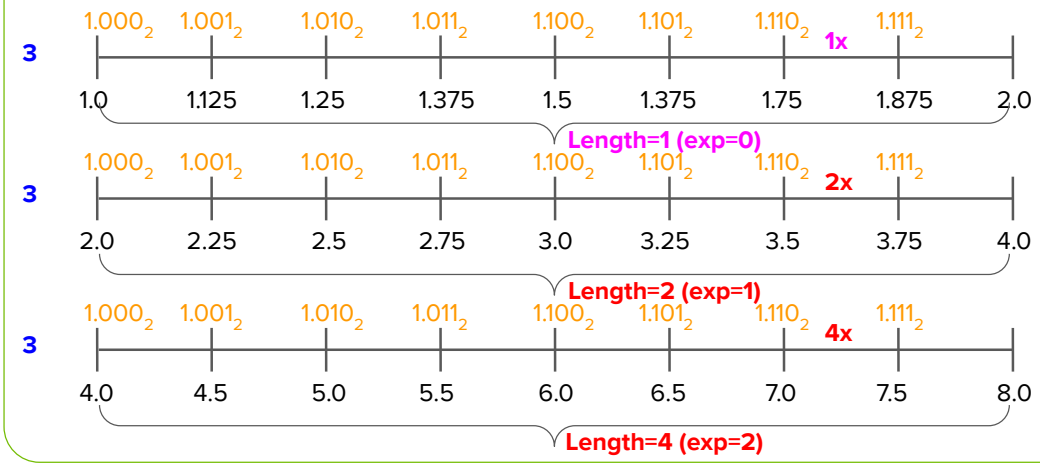
⇓

Aumentando l'esponente si coprono via via sempre meno numeri
(anche se la quantità di numeri numeri rappresentabili è la stessa, tuttavia aumenta, esponenzialmente la dimensione dell'intervallo da coprire).
Viceversa, diminuendo l'esponente aumenta la granularità dei valori rappresentabili
(perché diminuiscono le dimensioni dell'intervallo da coprire).

Possiamo dire che l'esponente altera il peso dei bit della mantissa e in maniera inversa la precisione.
(in modo direttamente proporzionale alla sua dimensione)

Floating-point vs Fixed-point and rounding error 6/10

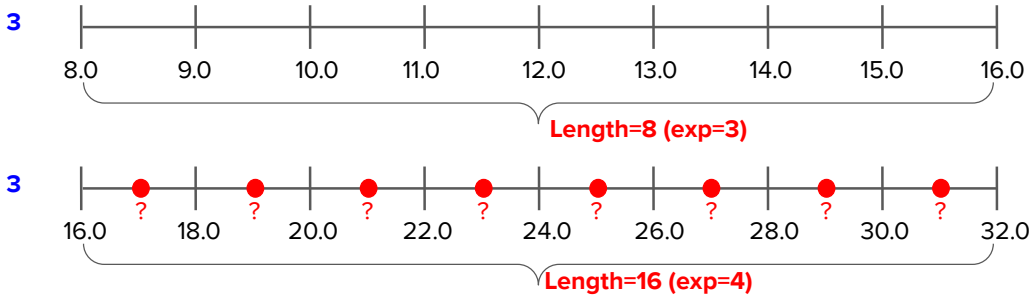
Increasing the magnitude of the numeric value further, the positive exponent gets higher (2x for each unit increase), and the issue outlined becomes even more evident.



22

Floating-point vs Fixed-point and rounding error 7/10

Increasing even further, with 3 fractional digits, we cannot encode odd integers (17, 19, 21, 23, 25, 27 and 31) without rounding when the exponent is 4 (see the graph at the bottom).



To overcome this issue, the solution consists of increasing the number of fractional bits (mantissa) at the cost of a larger data footprint and energy consumption for computations

23

Floating-point vs Fixed-point and rounding error 8/10

- Hence, the distance between two encodable FP values increases with the magnitude of the numeric value (or better, according to the increase of its positive exponent)
- Consequently, the rounding error in the worst cases is higher with larger positive exponents because the weight of ULP gets higher (2x when the exp. increases by 1)
- On the other hand, the issue just highlighted is the side effect that allows encoding much higher numeric values with floating-point vs fixed-point
- In contrast, with fixed-point, the largest rounding error is constant across all the representable ranges and equal to ½ of the constant weight of the ULP
- Regardless of the encoding type (fixed or floating-point), both allow encoding only a subset of numeric values without approximation
- With floating-point: the larger the exponent, the higher the likelihood that this issue occurs (eg, odd integers (eg, 16777217) cannot be encoded with a *float* data type)

probabilità

24

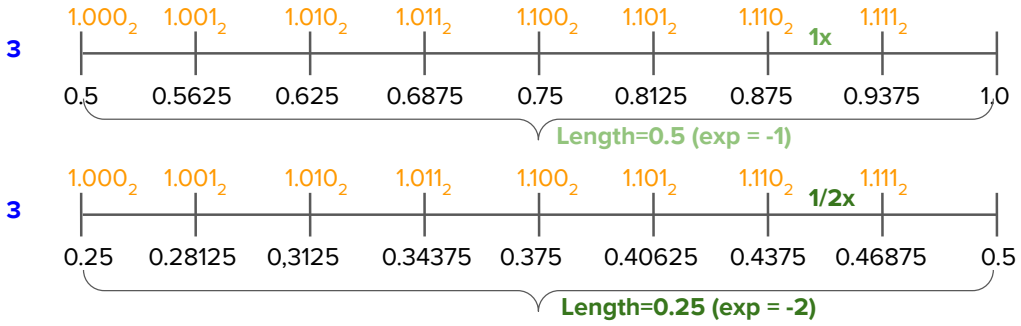
Floating-point vs Fixed-point and rounding error 9/10

- Of course, with negative exponents, the behaviour is in the opposite direction and the higher the magnitude of the negative exponent, the better the accuracy
- Intuitively, as outlined for positive exponents, by increasing the magnitude of a negative exponent (ef, from -1 to -2), a smaller portion of the range (½ in the example by increasing by 1) is divided into the same constant number of segments
- For instance, let's consider the range [0.5, 0.9375] and [0.25, 0.46875]:
  [$0.5_{10} = 0.1_2 = \mathbf{1.000 \cdot 2^{-1}}$, $0.9375_{10} = 0.1111_2 = \mathbf{1.111 \cdot 2^{-1}}$]
  [$0.25_{10} = 0.01_2 = \mathbf{1.000 \cdot 2^{-2}}$, $0.46875_{10} = 0.01111_2 = \mathbf{1.111 \cdot 2^{-2}}$]
  As outlined in the next slide, decreasing the exponent from -1 to -2, the data interval decreases from 0.5 to 0.25, but both segments are divided into the same 8 portions. Thus, the weight assigned to the ULP is, respectively, $\frac{1}{8} \cdot 2^{-1} = 0.0625$ and $\frac{1}{8} \cdot 2^{-2} = 0.03125$

25

Floating-point vs Fixed-point and rounding error 10/10

- Below, we show how increasing the magnitude of a negative exponent yields better accuracy



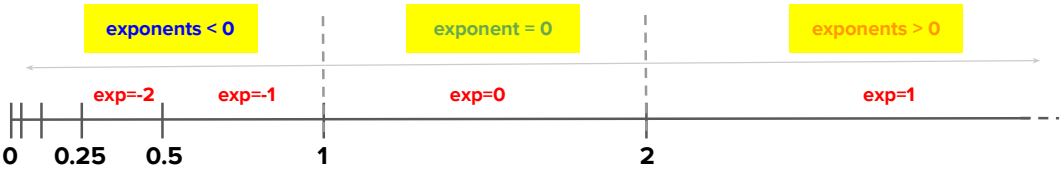
- Using the same scale, the behaviour described so far would appear like this:

26

Floating-point in a nutshell 1/3

subnormal number = {EXP=0, VAL≠0} e in questo caso la mantissa inizia per 0_2.
=> il numero sarà $0,qualcosa_2$

A summary of floating-point encoding

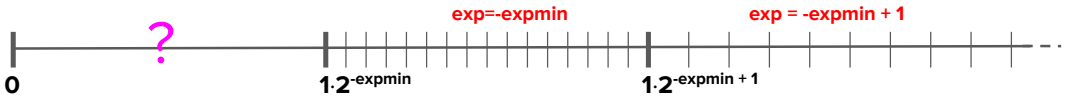


- Each segment has a different length according to the exponent and consists of a constant number of equidistant subsections according to the mantissa length
- There is a single exception to this scheme, the **subnormal numbers** discussed next, placed between 0 and the leftmost segment (ie, the one with the most negative exponents). They have the same most negative exponent, but they are not encoded in scientific notation, enabling increased accuracy around zero ~> evitando di dover troncare risultati a 0 perdendo informazioni
- The graph is specular for negative numbers (Sign=-1)

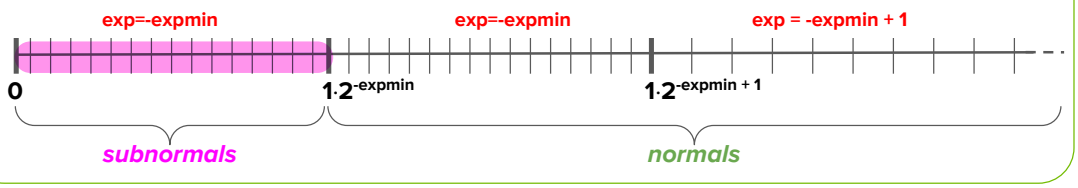
27

Floating-point in a nutshell 2/3

When a negative exponent is equal to the minimum value **-expmin**, using the scientific notation would yield a gap between] **0** and **1·2**$^{\text{-expmin}}$ [, as depicted below:



To avoid this weird behaviour (*flush to zero*) and gain accuracy near zero, numeric values between] **0** and **1·2**$^{\text{-expmin}}$ [are not represented in scientific notation (ie, they are encoded as **0.x·2**$^{\text{-expmin}}$). For this reason, they are called **denormal** (or **subnormal** numbers), leading to:



28

Floating-point in a nutshell 3/3 – GRS

- With a single notable exceptions (ie, the E4M3 minifloat introduced later), floating point reserves codes for **NaN** (Not a Number), **Inf** (Infinite) and **0**
- NaN examples: Inf - Inf, Inf/Inf, Infx0, 0/0
- Inf examples: 1/0, 2^{Inf}
- NaN, Inf and 0 can be positive or negative (ie, ±NaN, ±Inf and ±0)
- The range for positive and negative number is the same, the only difference is the sign
- For rounding and during computations the IEEE standard defines three extra bits: Guard, Round and Sticky (referred to as GRS) → servono per mantenere accuratezza nei calcoli intermedi.
- Guard and Round are the two additional bits at the right of the least significant bit used with the chosen encoding representation (LSB (destra))
  G: protegge precisione con shift (di fatto, il bit shiftato fuori).
  R: serve a capire come approssimare (assieme a G).
  Di fatto i bit che "escono" (es. shift dx) dalla mantissa vanno in GR.
  | MANTISSA | G | R | S |
  ← in memoria (mantissa con LSB a destra).
- The Sticky bit is a third bit (at the right of Round bit) which *remembers* (eg, during right shifts) if at least one bit among those at its right is 1
  S: risultato dei bit scartati (perché non entrano) in OR logico => se almeno uno =1 segnala che oltre alla mantissa (e GR) c'è altro da considerare (S).

29

Floating-point: additions 1/2

- The approximation issues highlighted start at the beginning, during the encoding phase and propagate through arithmetic operations
- Let's consider the addition of two numerical values in scientific notation encoded with 8 fractional bits: **1.21 + 1.22 = 2.43**

1.00110110_2 + **1.2109375**

1.00111000_2 = **1.2187500**

—----------

10.01101110_2

In scientific notation: **$1.0011\ 0111_2 \cdot 2^1$** = **2.4296875_{10}**

https://www.rapidtables.com/convert/number/binary-to-decimal.html?x=100110111

30

Floating-point: additions 2/2

Let's consider again the addition of two numerical values in scientific notation encoded with 8 fractional bits: **1.21 + 0.02** = $1.21 \cdot 10^0 + 2 \cdot 10^{-2} = 1.23 \cdot 10^0$

0.02 in scientific format is: $1.0100\ 0111_2 \cdot 2^{-6} = 0.01953125_{10}$

The first phase consists of aligning the two exponents to the largest one (0 in this case):

1.00110110 $000000_2 \cdot 2^0$ + **1.2109375**

0.00000101 $000111_2 \cdot 2^0$ = **0.01953125** (0.01995849609375 with extra bits)

—----------

1.00111011 $000111_2 \cdot 2^0$

In scientific notation: **$1.00111011_2 \cdot 2^0$** = **1.23046875_{10}**

Adding large values with smaller ones can induce errors due to the exponents' alignment

31

Floating-point: multiplication

Let's consider the multiplication of two numerical values in scientific notation encoded with 8 fractional bits: **1.21 x 0.02** = $1.21 \cdot 10^0 \times 2 \cdot 10^{-2} = 0.0242 \cdot 10^0 = 2.42 \cdot 10^{-2}$

0.02 in scientific format is: $1.0100\ 0111_2 \cdot 2^{-6} = 0.01953125$

The exponent of the results is the sum of the two exponents (0-6=-6 in this case), and the magnitude is the product of the two *significands* (1 + fractional part):

1.00110110_2 x **1.2109375**

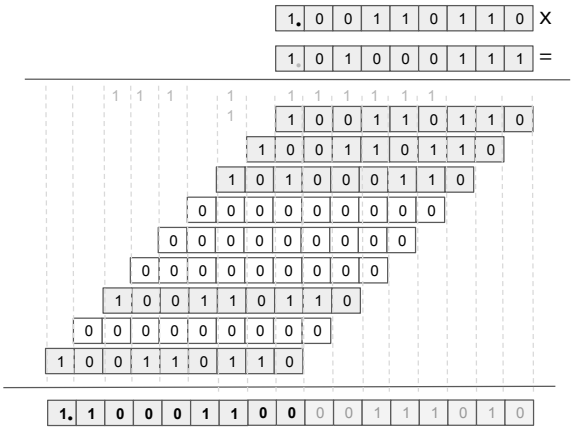
$1.0100\ 0111_2$ = **0.01953125** (0.01995849609375 with extra bits, previous slide)

1.10001100 00111010$\cdot 2^{-6}$ (details in the next slide)

The result has 2x the number of binary digits than the operands

32

Floating-point: multiplication details



1.1000 1100 0011 1010$\cdot 2^{-6}$ = **0.024183750152588**

1.10001100$\cdot 2^{-6}$ = **0.024169921875**

https://www.rapidtables.com/convert/number/binary-to-decimal.html?x=100110111

33

Multiply and Accumulate (MAC) and Fused MAC (FMA)

A multiplication followed by an addition, **Z = AxB + C**, is a typical sequence of operations performed in many application fields (eg, AI, computer graphics, image/signal processing).

The conventional approach, referred to as **MAC**, relies on two distinct phases/operations:

i) Multiply ; TEMP = AxB (1st rounding)

ii) Accumulate ; Z = TEMP + C (2nd rounding)

In contrast, Fused Multiply Add (**FMA**) performs the same operations in a single step:

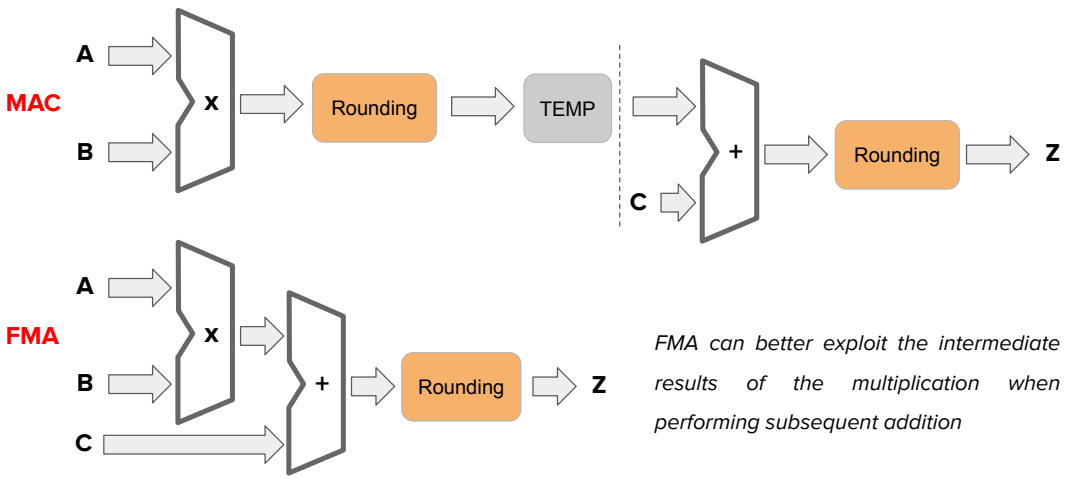
i) Multiply and Accumulate ; Z = AxB + C (single rounding)

Although FMA requires more demanding hardware, it performs a single rounding, is generally faster than MAC and is available in all modern architectures (eg, CPUs, GPUs)

34

MAC vs FMA: hardware design

- Difference between MAC and FMA from a hardware perspective



FMA can better exploit the intermediate results of the multiplication when performing subsequent addition

35

MAC vs FMA: example

Let's consider the following example:

A = $1.1011_2 \cdot 2^0 = 1.6875_{10}$, **B** = $1.1101_2 \cdot 2^0 = 1.8125_{10}$, **C** = $1.0100_2 \cdot 2^{-2} = 0.3125_{10}$

Z = AxB + C = $1.6875_{10} x 1.8125_{10} + 0.3125_{10} = 3.37109375_{10}$

Using **MAC**:

TEMP=AxB=$1.1011_2 x 1.1101_2 = 11.00001111_2 = 1.10001111_2 \cdot 2^1 \simeq 1.1000_2 \cdot 2^1 \simeq 3.00_{10}$ (**1st rounding**)

Z=TEMP+C=$1.1000_2 \cdot 2^1 + 1.0100_2 \cdot 2^{-2} = 1.1000_2 \cdot 2^1 + 0.0010100_2 \cdot 2^1 = 1.1010_2 \cdot 2^1 \simeq 3.25_{10}$ (**2nd rounding**)

Using **FMA**:

AxB=$1.1011_2 x 1.1101_2 = 11.00001111_2 = 1.100001111_2 \cdot 2^1 = 1.529296875_{10} \cdot 2^1 = 3.05859375_{10}$

AxB + C = $1.100001111_2 \cdot 2^1 + 1.0100_2 \cdot 2^{-2} = 1.100001111_2 \cdot 2^1 + 0.0010100_2 \cdot 2^1 = 1.101011111_2 \cdot 2^1 =$

$= 1.685546875_{10} \cdot 2^1 = 3.37109375_{10} \simeq 1.1011_2 \cdot 2^1 = 1.6875_{10} \cdot 2^1 = 3.375_{10}$ (**Single rounding**)

36

Floating-point pitfalls

If we add 1.11_{10} to 1.21_{10}, we do not obtain the expected result 2.32_{10} due to approximations:

```
double temp_1 = 1.11, temp_2 = 1.21, temp_3 = 2.32;
if ( (temp_1 + temp_2) != temp_3 ) is true…
```

Hence, rather than testing equality it is wiser to evaluate the difference between quantities when dealing with floating-point numbers:

```
if ( fabs((temp_1 + temp_2) - temp_3) < epsilon )
```

However, pay attention when subtracting two very close numbers because the difference can be zero due to approximations.

Moreover, for the same reason, take care even when adding a huge number with a tiny one since the latter can *vanish*

37

Floating-point: associative and distributive properties

Unfortunately, due to approximation encoding and rounding, the associative property does not (always) hold:

(A + B) + C != A + (B + C)

Moreover, even the distributive property is not always satisfied for the same reason;

Ax(B + C) != AxB + BxC

For the above reasons, take extra care when handling floating-point data, even using the large double format.

38

Floating-point and *fast-math* optimizations

- Compilers have flags to generate optimized codes, such as `-O3` in GCC
- Most compilers also have flags to speed up computations when dealing with math operations and, consequently, with floating points computations
- For instance, the `-ffastmath` flag can significantly speed up computations involving floating-point data
- However, among other things, it does so neglecting the existence of subnormal numbers by flushing to zero all these values for efficiency purposes
- Other simplifications occur; thus, take extra care before applying this kind of optimization, as described in detail in Simon Byrne's post:

  https://simonbyrne.github.io/notes/fastmath/

39

Floating-point encoding: IEEE 754 1/7

- A binary number is expressed in scientific format if it has a single integer digit set to 1 (the other option with the single bit set to 0 is not allowed* for the scientific format)
- Thus, a binary number expressed in normalized scientific format has this form:
  - $(-1)^S \cdot 1.xxxxxx \cdot 2^{yyyy}$, with S a single bit, xxxxxx and yyyy binary digits
- In this notation, the first digit (ie, the integer part) is always* 1 and thus omitted in the encoding
- The fractional part xxxxxx is the *mantissa* **M**
- The yyyy binary digits form the *exponent* **E** (discussed next)
- A single bit **S** encodes the *sign* (S=0 positive sign, S=1, negative sign)
- Consequently, we can represent* a binary value with floating-point notation as: $(-1)^S \cdot 1.M \cdot 2^{E-BIAS}$, with **BIAS** a predefined polarization value for each representation

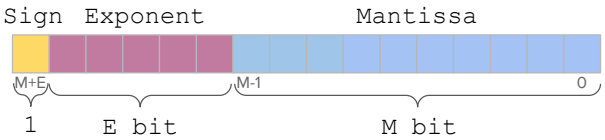
* exceptions for subnormal numbers apply (discussed later)

40

Floating-point encoding: IEEE 754 2/7

*la fractional part binaria e decimale non coincidono necessariamente (normalizzate).
Avendo 21_{10}: • dec → $21_{10} = 2{,}1 \cdot 10^{0}_{10}$
• bin → $21_{10} = 10101_2 = 1{,}\underbrace{0101}_{\text{MANTISSA}} \cdot 2^{\overset{\odot}{4}\,\text{EXP}}_2$

- Standards are crucial to enable seamless interoperability among different architecture, manufacturers, compilers, etc
- Purposely, IEEE 754 defines the standard to represent floating-point numbers using the following scheme:



- Since the integer part is always one*, it is omitted in the encoding, and the mantissa – also referred to as *significand* – is devoted to encoding the fractional part only. (binaria)*
- Hence, the encoded numeric value (excluding sign and exponent and subnormal numbers) corresponds to: **1.**

41

Floating-point encoding: IEEE 754 3/7

Considering numbers in scientific notation and a format like that defined in the previous slide (sign 1 bit, exponent 5 bit, mantissa 10 bit plus 1 implicit, and setting BIAS = 15): FP16

value = $(-1)^S \times 2^{\left(\sum_{k=0}^{4}(b_{10+i}\cdot 2^k)-15\right)} \times \left(1+\sum_{i=1}^{10} b_{10-i}\cdot 2^{-i}\right)$

$BIAS = \frac{2^{\#\text{bit Exponent}}}{2} - 1$

es. FP16 $BIAS = \frac{2^5}{2} - 1 = \frac{32}{2} - 1 = 15$

Thus, the following binary digits encodes value 23.265625

per semplificare i confronti

Sign	Exponent					Mantissa									
0	1	0	0	1	1	0	1	1	1	0	1	0	0	0	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Exponent = EXP − BIAS ⇝ BIAS costante per usare esponenti unsigned (invece che compl. a 2)

$(-1)^0 \times 2^{(19-15)} \times (1 + 0\cdot 2^{-1} + 1\cdot 2^{-2} + 1\cdot 2^{-3} + 1\cdot 2^{-4} + 0\cdot 2^{-5} + 1\cdot 2^{-6} + 0\cdot 2^{-7} + 0\cdot 2^{-8} + 0\cdot 2^{-9} + 1\cdot 2^{-10}) =$

$= 1 \times 2^{(4)} \times (1 + 1/4 + 1/8 + 1/16 + 1/64 + 1/1024) = 1 \times 16 \times 1.45410156 = 23.265625$

This specific data representation is referred to by IEEE 754 as a *half-precision floating-point* or *floating-point 16* (FP16) since it uses a 16-bit encoding

42

Floating-point encoding: IEEE 754 4/7

FP16 also allows encoding *subnormal* – numbers around zero (±0.x) which are not in scientific format – using an exponent 00000 and a mantissa different from all zeros:

-	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

subnormal, "-" means 0 or 1

A subnormal number has always exponent $\mathbf{2^{1-BIAS}}$; hence, with BIAS=15, it corresponds to:

value = $(-1)^S \times 2^{-14} \times \left(0+\sum_{i=1}^{10} b_{10-i}\cdot 2^{-i}\right)$

FP16 encodes zero(s) – indeed, there are two zero – with the following codes:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Zero (+0)

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Zero (-0)

43

Floating-point encoding: IEEE 754 5/7

Additionally, In FP16 some configurations are reserved to represent specific cases such as +∞ (eg, 1.0/0.0)and -∞ (eg, -1.0/0.0):

0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	**+∞**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	**-∞**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

Finally, FP16 encodes the result of unpredictable operations, referred to as NaN (*Not a Number*), with all exponent bits set to 1 and a mantissa different from all zeros:

0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	**Nan**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	**-Nan**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

44

Floating-point encoding: IEEE 754 6/7

In FP16, considering the reserved configurations 0 and 31, the exponent E for numbers in scientific/normalized format can assume any integer value between 1 and 30:

Exponent E										Exponent E				
0	0	0	0	1	· · · · · · · ·					1	1	1	1	0
1					2,3,4,5,6...					30				

Hence, being in FP16 the exponent polarized according to a BIAS 15 (ie, 2^{E-15}), the exponential factor spans from 2^{-14} to 2^{+15}. Consequently, the dynamic range spans from

-	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- *means 0 (+) or 1 (-)*

$\pm 65504 = 1 \times 2^{15} \times (1+¼+⅛+1/16+1/32+1/64+1/128+1/256+1/512+1/1024) = 2^{15} \times 1.9990234375$ to

-	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- *means 0 (+) or 1 (-)*

$\pm 0.00006103515625 = 1 \times 2^{-14} \times (1+0) = 2^{-14}$

45

Floating-point encoding: IEEE 754 7/7

- Subnormal numbers can be either positive (S=0) or negative (S=1)

-	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- *means 0 (+) or 1 (-)*

- Although not in scientific format, subnormal numbers allow to encode values around zero ranging from $[\simeq -1x2^{-14}$ to $\simeq +1x2^{-14}]$
- Specifically,
  $[-(2^{-1}+2^{-2}+2^{-3}+2^{-4}+2^{-5}+2^{-6}+2^{-7}+2^{-8}+2^{-9}+2^{-10})x2^{-14}, +(2^{-1}+2^{-2}+2^{-3}+2^{-4}+2^{-5}+2^{-6}+2^{-7}+2^{-8}+2^{-9}+2^{-10})x2^{-14}]$
  $[-1023/1024x2^{-14}, +1023/1024x2^{-14}]$

1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

46

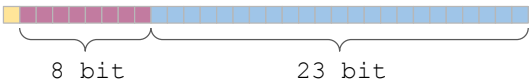
IEEE 754: data formats

- To facilitate interoperability, IEEE released a first draft to standardize floating-point data and computations in 1985
- It was updated in 2008 and 2019
- Among other things, currently, it supports the following data types:
  - **FP16** (Half, or binary16) : 16 bit, S=1, E=5, M=10, BIAS=15
  - **FP32** (Float, or binary32): : 32 bit, S=1, E=8, M=23, BIAS=127
  - **FP64** (Double, or binary64) : 64 bit, S=1, E=11, M=52, BIAS=1023
  - **FP128** (Quadruple, or binary128) : 128 bit, S=1, E=15, M=112, BIAS=16383
  - **FP256** (Octupule, or binary256) : 256 bit, S=1, E=19, M=236, BIAS=262143

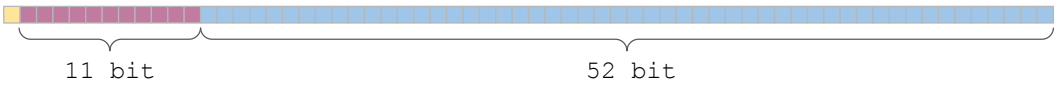
47

IEEE 754: Float and Double

- In addition to the FP16 format previously described, IEEE 754 defines the following data formats used in most programming languages/compilers: `float` and `double`, respectively, using a 32 and 64 bits (dalla dimensione doppia origina 'doublé')
- **Float** (32 bit) – *single precision*: sign 1 bit, exp. 8 bit, mantissa 23 bit (+1), and BIAS = 127. Range [min, max] is: [$1.17549435 \cdot 10^{-38}$, $3.40282347 \cdot 10^{+38}$]



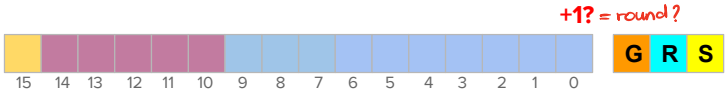
- **Double** (64 bit) – *double precision*: sign 1 bit, exp. 11 bit, mantissa 52 bit (+1), and BIAS = 1023. Range [min, max] is: [$2.2250738585072014 \cdot 10^{-308}$, $1.7976931348623158 \cdot 10^{+308}$]



48

IEEE 754: Guard, Round and Sticky bits for rounding

- IEEE 754 performs rounding using three additional hidden bits (i.e., in addition to the specific format used) during encoding and computations
- These bits are called Guard (**G**), Round (**R**), and Sticky (**S**) | **G** | **R** | **S** |
- They are somehow concatenated to the right end to decide whether the +1 is needed:



+1? = round?

Arrotondamento se 2+ bit "1" tra:
- LSB (mantissa)
- G, R, S

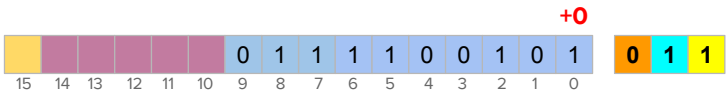
- The S bit is set to 1 (for instance, when shifting right) if at least one of the bits at its right was 1, while G and R are two extra bits which increase the mantissa length by 2
- If **G=0**, the mantissa doesn't change, regardless of R and S (GRS = 0xx) | **0** | **x** | **x** |
- If **G=1**, **R=1**, **S=x** or **G=1**, **R=0**, **S=1**, then round up (+1) | **1** | **1** | **x** | | **1** | **0** | **1** |
- If **G=1**, **R=0**, **S=0**, we are in between. If the last bit of the mantissa is 0, do nothing else add +1 to the mantissa. In the IEEE terminology, *round to even* | ? | | **1** | **0** | **0** |

49

IEEE 754: rounding examples 1/2

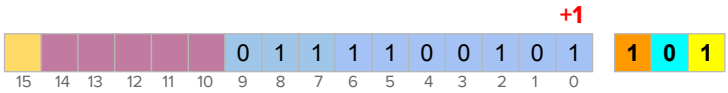
With FP16, suppose that the result of a multiplication is 1.0111100101011010·2^{-2}

GRS bits can summarize the additional bits 01011010 as G=0, R=1, S=1 (the OR between the last 6 bits 011010). Consequently, the mantissa remains unchanged:



With FP16, suppose that the result of a multiplication is 1.0111100101100110 10·2^{-2}

GRS bit can summarize the additional bits 10011010 as G=1, R=0, S=1 (the OR between the last 6 bits 011010); consequently, the mantissa requires a +1 increment:

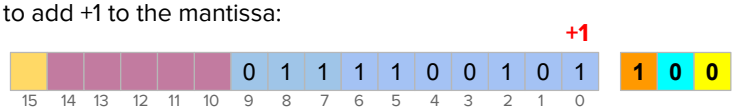


50

IEEE 754: rounding examples 2/2

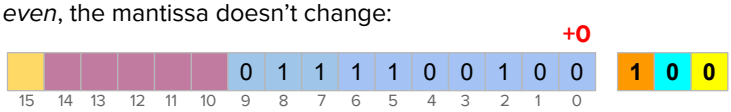
With FP16, suppose that the result of a multiplication is 1.0111100101100000000·2^{-2}

GRS bits can summarize the additional bits 10000000 as G=1, R=0, S=0 (the OR between the last 6 bits 0000000). Nonetheless, since the value in the mantissa is odd, the *round to even* requires to add +1 to the mantissa:



Finally, with FP16, suppose that the result of a multiplication is 1.0111100100100000000·2^{-2}

GRS bits can summarize the additional bits 10011010 as G=1, R=0, S=0 (the OR between the last 6 bits 0000000). There is a tie, but the last bit of the mantissa is even, and according to *round to even*, the mantissa doesn't change:



51

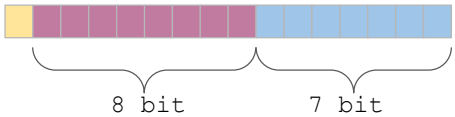
Other floating-points formats

- With the widespread diffusion of AI, some companies have defined other formats; they are typically more compact and better tailored for their purposes
- The main reason concerns efficiency and power consumption that generally *scales as the square of the mantissa length during arithmetic operations* (source, NVidia)
- Another reason for more compact representations is the reduced memory footprint
- In the following slides, we'll introduce the following popular formats, although not (yet) IEEE standard:
  - **BFloat**
  - **Tensore-Core32**
  - **Minifloat E5M2**
  - **Minifloat E4M3**

52

Brain Float (Google Brain) 1/2

- Brain Float (BFloat) is a floating-point proposed by Google for its devices
- It's gaining interest since it reduces data storage and speeds up AI computations
- To better fit AI needs, it trades dynamic range for precision using a 16 bit encoding
- Specifically, it is a more compact 16 bit version of IEEE 754 FP32; it preserves the dynamic range (ie, same E=8 and BIAS=127) but has a shortened Mantissa (from 23 to 7 bits), reducing accuracy (or precision, meaning with this the reduced number of bits):

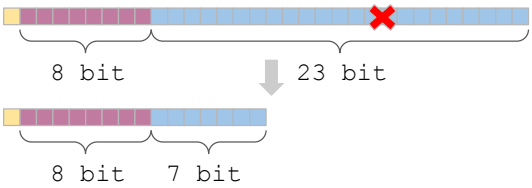


- Data range equivalent to FP32 but with reduced precision
- Adopted by Google (TPUs), Intel (AVX-512), NVidia (CUDA), Apple (CPUs), etc

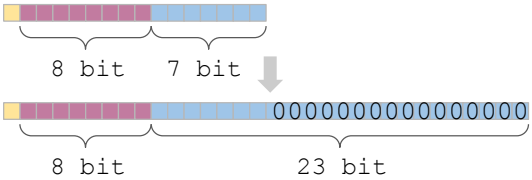
53

Brain Float 16 (Google Brain) 2/2

- Features: Fast conversion from float (FP32) to BFloat (eg, a simple truncation of M, from 23 to 7 bit, or better with rounding) and from BFloat to FP32 (ie, padding missing mantissa bits with zeros):
  - From Float to BFloat:



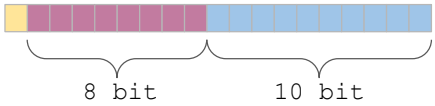
  - From BFloat to Float:



54

TensorFloat-32 (NVIDIA) 1/2

- NVidia has introduced for its GPUS the new floating-point format TF32, enabling a good trade-off between performance and accuracy, fitting very well with *tensor cores*
- It has the same exponent E of a float (FP32) and BFloat, thus using the same BIAS, but the mantissa is only 10 bit wide. Specifically, TF32 encodes numbers with 1-bit sign S, exponent E=8 bit, mantissa M=10 bit and BIAS = 127.

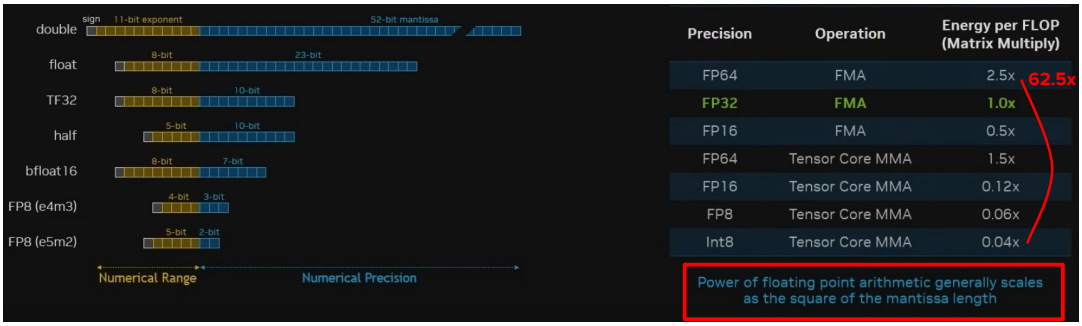


- Data range equivalent to FP32 with reduced precision (but better than BFloat)
- As for BFloat, TF32 enables a fast conversion from and to FP32 (and BFloat)

55

TensorFloat-32 (NVIDIA) 2/2

- NVidia claims significant benefits in terms of reduced power consumption vs FP32
- The reduced precision might not represent a crucial issue for some relevant tasks in AI
- The following screenshot, taken from an NVidia talk, reports Energy/FLOPS for Multiply-Add operations using different data formats on NVidia GPUs

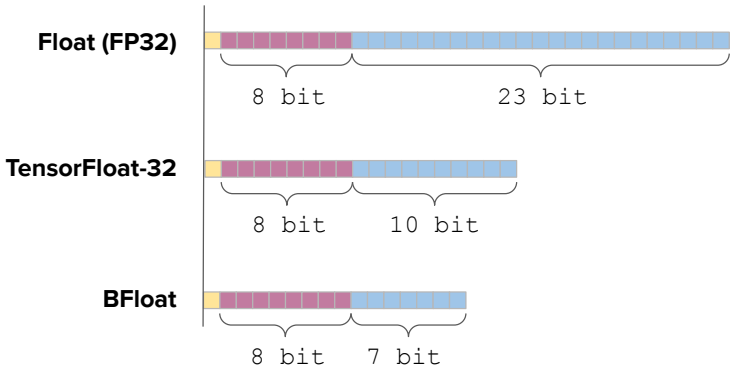


MMA: Matrix Multiply-Accumulate (like D = A * B + C) – FMA: Fused Multiply-Add

56

Float vs TensorFloat-32 vs BFloat

Below, we summarize the different encoding of the three widely used floating point formats Float, TensorFloat-32 and BFloat, all using an 8-bit exponent E and the same BIAS=127 but a different number of bits for the mantissa:



57

Minifloat data FP8

- AI can be surprisingly effective even with reduced accuracy
- Consequently, encoding data with less than 16 bits can be a viable solution in some cases to reduce data storage and computations using mixed-type data
- Below, the 8-bit encoding of two minifloat data types (not yet included in the IEEE 754 standard) named **E5M2** (E=5, M=2, BIAS=15) and **E4M3** (E=4, M=3, BIAS=7)

- As usual, they follow the IEEE 754 specifications but with some limitations (eg, E4M3)
- Major players use these data types (NVidia, ARM, Intel, AMD, ..)

58

Minifloat: E5M2 1/3

- *Full-fledged* floating point with S=1, E=5, M=2 and BIAS=15
- It represents subnormal numbers (E=0, M≠0), infinities (E= 11111, M=0), NaNs (E= 11111, M≠0) and two zeros (00000000 and 10000000, respectively, for +0 and -0)
- The dynamic range spans from $\pm 2^{-14}\cdot(0+\frac{1}{4})$ to $\pm 2^{15}\cdot(1+\frac{3}{4})$

Since E= 11111 is reserved for infinities and NaNs, and E=00000 for subnormal numbers, the exponent spans $[11110_2, 00001_2] = [30_{10}, 1_{10}]$ for values in scientific notation

Therefore, considering the BIAS=15, the actual exponents span the range $[15_{10}, -14_{10}]$, with -14 (ie, 1-BIAS) for subnormal numbers

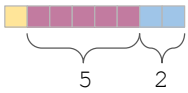
The 2-bit used for the mantissa allows to encode the following values in scientific format: 1, 1 + 1/4 , 1+ ½, 1+½+ ¼ = 1, 1.25, 1.5, 1.75 and for subnormals 1/4 , ½, ½+ ¼ = 0.25, 0.5, 0.75

59

Minifloat: E5M2 2/3

- *Full-fledged* floating point
- It encodes NaN and Inf

E5M2



E5M2

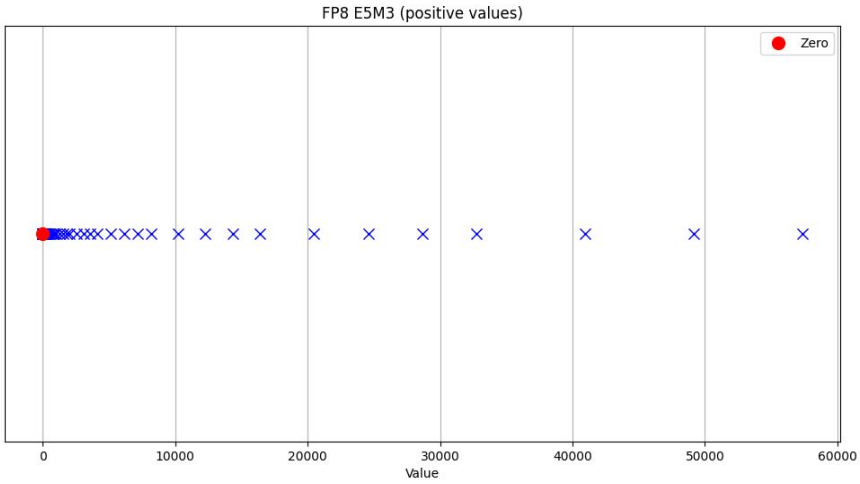
Binary	Hex	Value	Notes
0 00000 00	0	0	
0 00000 01	1	$2^{-14} \times (0 + \frac{1}{4}) \approx 0.000015258$	Smallest positive subnormal number
0 00000 11	3	$2^{-14} \times (0 + \frac{3}{4}) \approx 0.000045776$	Largest subnormal number
0 00001 00	4	$2^{-14} \times (1 + \frac{0}{4}) \approx 0.000061035$	Smallest positive normal number
0 00110 00	18	$2^{-9} \times (1 + \frac{0}{4}) = 0.001953125$	E4M3 - Smallest positive subnormal number
0 01101 01	35	$2^{-2} \times (1 + \frac{1}{4}) = 0.3125$	Nearest value to 1/3
0 01110 11	37	$2^{-1} \times (1 + \frac{3}{4}) = 0.875$	Largest number less than one
0 01111 00	38	$2^{0} \times (1 + \frac{0}{4}) = 1$	One
0 01111 01	39	$2^{0} \times (1 + \frac{1}{4}) = 1.25$	Smallest number larger than one
0 01111 10	3a	$2^{0} \times (1 + \frac{2}{4}) = 1.5$	Nearest value to sqrt(2) $= 1.414$
0 10000 10	42	$2^{1} \times (1 + \frac{2}{4}) = 3.0$	Nearest value to PI $= 3.141$
0 10111 11	5f	$2^{8} \times (1 + \frac{3}{4}) = 448$	E4M3 - Largest number
0 11110 11	7b	$2^{15} \times (1 + \frac{3}{4}) = 57344$	Largest normal number
0 11111 00	7c	inf	Infinity
0 11111 01	7d	NaN	First of 6 NaN representations
1 00000 00	80	0	
1 10000 00	c0	-2	
1 11111 00	fc	$-inf$	Negative Infinity
1 11111 11	ff	NaN	Last of 6 NaN representations

https://lambdalabs.com/blog/nvidia-hopper-h100-and-fp8-support

60

Minifloat: E5M2 3/3

The plot below reports the positive values encodable with the E5M2 format

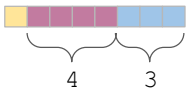


61

Minifloat FP8: E4M3

- Compared to E5M2, it has a smaller mantissa and larger exponent
- Notably, it does not encode Inf/NAn; it uses Inf/NAn code (E=1111) to extend the range

E4M3

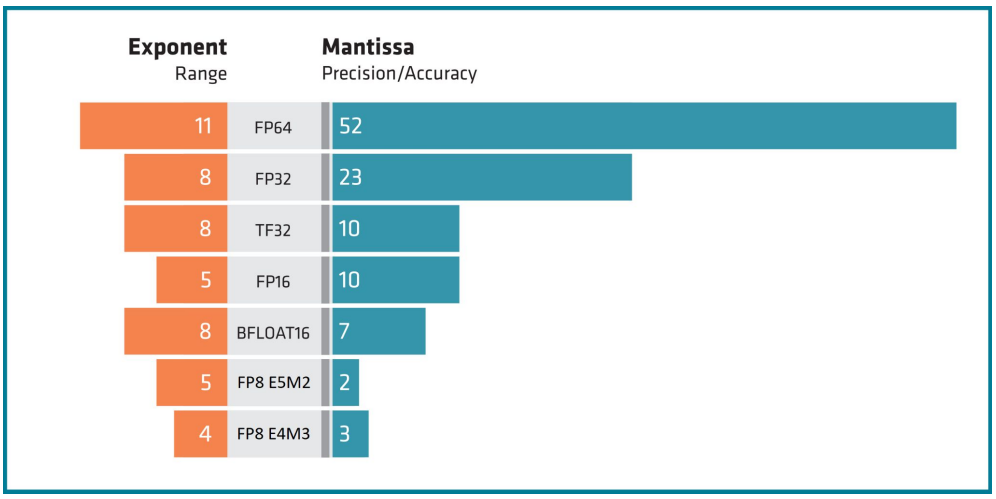


E4M3

Binary	Hex	Value	Notes
0 0000 000	0	0	
0 0000 001	1	$2^{-6} \times (0 + \frac{1}{8}) = 0.001953125$	Smallest positive subnormal number
0 0000 111	7	$2^{-6} \times (0 + \frac{7}{8}) = 0.013671875$	Largest subnormal number
0 0001 000	8	$2^{-6} \times (1 + \frac{0}{8}) = 0.015625$	Smallest positive normal number
0 0101 011	2b	$2^{-2} \times (1 + \frac{3}{8}) = 0.34375$	Nearest value to 1/3
0 0110 111	37	$2^{-1} \times (1 + \frac{7}{8}) = 0.9375$	Largest number less than one
0 0111 000	38	$2^{0} \times (1 + \frac{0}{8}) = 1$	One
0 0111 001	39	$2^{0} \times (1 + \frac{1}{8}) = 1.125$	Smallest number larger than one
0 0111 100	3c	$2^{0} \times (1 + \frac{3}{8}) = 1.375$	Nearest value to sqrt(2) = 1.414
0 1000 101	45	$2^{1} \times (1 + \frac{5}{8}) = 3.25$	Nearest value to PI = 3.141
0 1110 111	77	$2^{7} \times (1 + \frac{7}{8}) = 240$	Largest normal number without range extension
0 1111 110	7e	$2^{8} \times (1 + \frac{6}{8}) = 448$	Largest normal number
0 1111 111	7f	NaN	First of 2 NaN representations
1 0000 000	80	0	
1 1000 000	c0	-2	
1 1111 111	ff	NaN	Last of 2 NaN representations

62

Summary of floating-point data types



https://rocm.docs.amd.com/en/docs-6.0.2/about/compatibility/data-type-support.html

63

Exercise

Exercise

Implement a C function to compute, with basic arithmetic and logic operations, the numerical value of a number encoded in E5M2 floating point format. Hint: $2^1 = 1 << 1$, $2^2 = 1 << 2$, $2^{-1} = 1/ (1 << 1)$, $2^{-2} = 1/ (1 << 2)$

```
struct E5M2 {	unsigned char bit[8];
		float numeric_value;
		int raw_exponent;
		int actual_exponent;
		float fractional;
		float significand;
		bool is_negative;
		bool is_Normal;
		bool is_Subnormal;
		bool is_Zero;
		bool is_Inf;
		bool is_NaN;
		int BIAS; };
```

64

Addition is all you need? 1/2

- A recent paper (Oct. 2024), "Addition is All You Need for Energy-efficient Language Models", proposes an approximated method (referred to as L-Mul) to speed up and reduce power consumption for floating-point computation in language models
- It relies only on lightweight integer additions, replacing the expensive multiplication with an approximated term
- Given two numbers X and Y encoded in floating-point format, with m bits for mantissa and e bits for exponent, they can be expressed as: $X = (1 + Xm)\cdot 2^{Xe}$ and $Y = (1 + Ym)\cdot 2^{Ye}$
- Their product results:

$$(1 + Xm)\cdot 2^{Xe}\cdot(1 + Ym)\cdot 2^{Ye} = (1 + Xm + Ym + Xm\cdot Ym)\cdot 2^{Xe+Ye} \approx (1 + Xm + Ym + 2^{-L(m)})\cdot 2^{Xe+Ye}$$

$$\text{with } L(m) = \begin{cases} m, \text{ if } m \leq 3 \\ 3, \text{ if with } m = 4 \\ 4, \text{ if } m > 4 \end{cases}$$

https://arxiv.org/abs/2410.00907

65

Addition is all you need? 2/2

- Author claims:
  - *L-Mul is more precise (accurate???) than fp8 e4m3 multiplications but uses less computation resource than fp e5m2*
  - *massive power reduction with marginal or null performance drop in some relevant use cases reported in the paper*
- Independently of these claims, the outlined strategy yields a raw significant accuracy drop compared to the conventional multiplication approach
- Regardless of the effectiveness of this specific method, reducing power consumption with floating-point arithmetic is a vivid topic with significant environmental implications (" It is estimated that Google's AI service could consume as much electricity as Ireland (29.3 TWh per year) in the worst-case scenario (de Vries, 2023).")

66

Exercise

Exercise

Given two numbers encoded in the floating point format E5M2, evaluate the difference between the conventional and approximated methods proposed in the paper "Addition is All You Need for Energy-efficient Language Models".

Hint: use the function implemented in the previous exercise.

67

Computational issues and memory footprint

- Neural networks typically consist of many (million) weights encoded as floating-point data
- Such an amount of data yields computational issues, often related to multiply and add/matrix multiplications and convolutions; to this aim, parallel computational strategy and architecture can allow to cope with it
- However, such models also have massive memory footprints, challenging the memory available even for devices equipped with powerful computing platforms
- Therefore, there are techniques to face these problems, such as:
  - Data quantization to reduce data size
  - Weight sharing or Palletization to reduce the amount of data

68

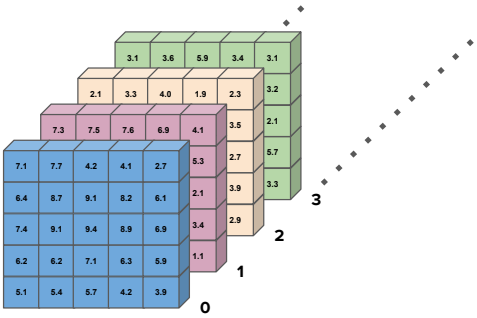
Reducing data size

- A straightforward strategy to reduce computations and memory footprint consists of reducing the data size
- Starting from an existing network model using 32-bit floating-point data, moving to a smaller data representation can improve both issues
- For instance, switching from FP32 to E5M2 would decrease the footprint by 1/4
- Of course, the resulting network is not equivalent to the original one and post-quantization fine-tuning is required to achieve reasonable performance
- A further yet more challenging improvement from the computational point of view would consist of switching to integer data
- An even further step forward is represented by Logic Gate Networks (LGNs), performing high-speed computations with conventional gates (eg, XOR, NAND, AND) https://arxiv.org/pdf/2411.04732

69

Weight-sharing (or Palettization) 1/3

- The massive amount of data in network models is primarily needed to store the floating-point weights needed to perform computations (eg, large amounts of different convolutions, each one with different kernel weights)

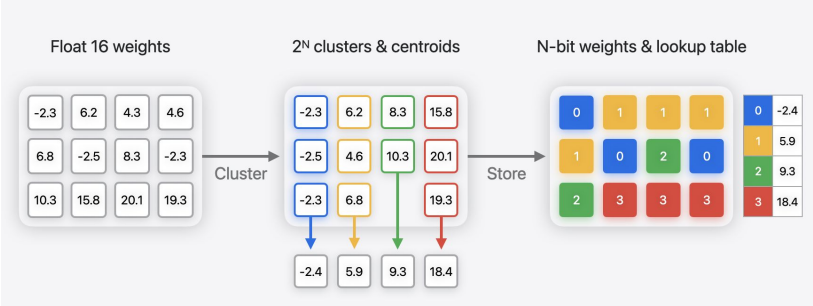


- Weight sharing aims to reduce the memory footprint using a smaller yet representative number of weights indexed through a Look-Up-Tables (LUT)

70

Weight-sharing (or Palettization) 2/3

- The key idea is to replace multiple weights with their average values (centroids) using quantization techniques and Look-Up-Tables to index centroids
- For instance, the weights below could be clustered in 4 classes and to each centroid assigned an index [0,3]; overall, 4 weights (centroids) and 2 bit for each element

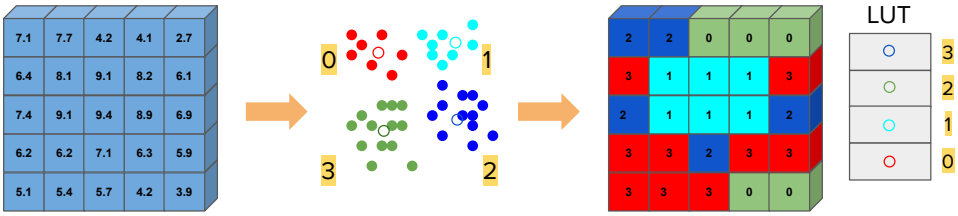


https://apple.github.io/coremltools/docs-guides/source/opt-palettization-overview.html

71

Weight-sharing (or Palettization) 3/3

- Given the number of classes, K-means is a well-known vector quantization technique suited for this purpose; it clusters the input values around each centroid



- In the figure, with FP 32 weights, the kernel requires 25x32 bit; after quantization in 4 classes, the memory footprint is 25x2 + 4x32 bit – reduced by a factor 800/178 ≃ 4.5
- The lower the number of clusters, the lower the memory footprint and accuracy
- Thus, fine-tuning is required to recover the loss in accuracy after palettization

72

Una rete neurale moderna può avere milioni di "pesi" (w). Se ogni peso è un float a 32 bit occupa tantissimo spazio (rispetto ad un intero) in memoria RAM, é lento (da scaricare dalla RAM alla Cache della CPU/GPU) e consuma molta energia.
Tuttavia, si è notato che in una rete neurale molti pesi hanno valori molto **simili** tra loro.

La soluzione é la **palettizzazione** (da *palette*, di colori).
Il termine viene dal mondo delle immagini (come i file GIF). Invece di usare milioni di colori, ne vengono scelti solo 256 (per esempio, la palette) e per ogni pixel viene scritto solo il numero del colore (da 0 a 255). Un po' come la quantizzazione dei segnali analogici (conversione analogico-digitale).

1. **Clustering**: Si prendono tutti i pesi e si raggruppano in poche classi (numero molto minore del numero di pesi).
2. Per ogni classe si calcola un valore medio, il **centroide**. Questo valore sarà il "rappresentante" e verrà salvato in una tabella (LUT), per evitare di rifare i conti più volte.
3. Nella matrice dei pesi originale, viene cancellato il numero floating point e sostituito con l'indice (la posizione) nella tabella.

Quindi il modello occupa meno spazio e i calcoli sono più veloci (perché i dati sono di dimensioni minori), al costo di perdita di **accuratezza**.
Per recuperare l'accuratezza persa, dopo aver "palettizzato" i pesi, si fa **fine-tuning** della rete per permetterle di abituarsi ai nuovi valori approssimati.

Fixed-point arithmetic: use case

So far, we have considered floating-point as the unique method to deal with real values, but can we perform these computations using only integer data/operations?

Such an approach might have significant advantages:

- floating-point modules/features could not be available (eg, microcontrollers) or highly resource-demanding (eg, FPGA)
- computations using floating-point often require longer latency vs integer

For instance, we can add two numbers like 0.75_{10} and 1.5_{10}, performing the following integer operations: $0.75 \rightarrow 75 \cdot 10^{-2}$, $1.50 \rightarrow 150 \cdot 10^{-2}$ and consequently their sum (up to a scale factor) is: $0.75_{10} + 1.5_{10} = 225 \cdot 10^{-2}{}_{10}$

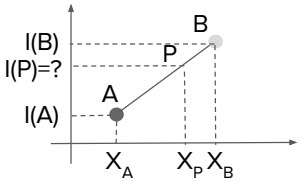
Integer operations were feasible by simply scaling up numbers by a constant factor (ie, 10^2)

73

Example: linear interpolation (real)

Example

- Given two points A and B, located at X_A and X_B and with intensity magnitude I(A), I(B), compute the weighted color intensity I(P) for point P located at $X_P = 5$



$$I(P) = I(A)\frac{X_B - X_P}{X_B - X_A} + I(B)\frac{X_P - X_A}{X_B - X_A}$$

Assuming I(A)=20, and I(B)=200: I(P)= 20·(0.25) + 200·(0.75) = 155

74

Example: linear interpolation (integer) 1/3

Can we perform this computation using only integer operations?

Assuming, for simplicity, that I(A) and I(B) are integers and point P lies only at discrete positions, the weights $[w_A,w_B)$ are: [1,0], [0.75,025], [0.5,0.5], [0.25,0.75], [0,1]

For instance, if P=5, I(A)=20, and I(B)=200, by scaling up weights by a factor of 10^2, we get:

$I^*(P)= 20\cdot(25) + 200\cdot(75) = 500 + 15000 = 15500 = 155\cdot10^2$ (only integer operations).

Finally, we obtain the expected result (ie, 155) by scaling down (ie, *dividing* by a factor of 10^2).

However, the final division is unnecessary since it suffices a simple truncation – that might yield approximation, although not in this case – of the two last decimal digits.

75

Example: linear interpolation (integer) 2/3

- Issue 1: Let's consider what happens with P=5, I(A)=23, and I(B)=178

$I^*(P)= 23\cdot(25) + 178\cdot(75) = 575 + 13350 = 13925 = 139.25\cdot10^2$ (only integer operations)

In this case, truncating the last two digits leads to an acceptable approximation in most cases (e.g., with images, the pixel intensity is a discrete value).

Moreover, by analyzing the last two digits, we can perform the appropriate rounding (in this case, 139.25 ≃ 139 equivalent to C function round()).

- Issue 2: All operations occur with integer values, but weighting factors need conventional *real* division

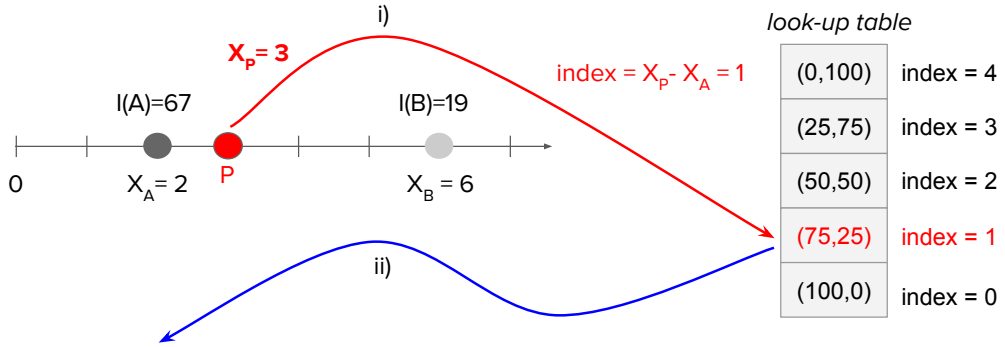
That's true, but weighting factors can be computed offline, scaled up by the chosen factor, and stored in a look-up table (LUT) easily index (e.g., in this case, using the index [XP-XA]).

The advantage is twofold: all integer operations and no weight computation at run-time.

76

Example: linear interpolation (integer) 3/3

The overall process involving only integer operations can be summarized as follows:



$I^*(P) = 67 \cdot (75) + 19 \cdot (25) = 5025 + 475 = 5500 = 55 \cdot 10^2$ (only integer operations)

I(P) = 55 (truncating last two digits and rounding)

Of course, we can use binary numbers as well and scaling by power of 2

77

1.5. FPGA (Field Programmable Gate Array)

Cenni storici

- '70s: gates, SSI, MSI, LSI
- Successivamente: dispositivi programmabili (a livello di funzioni logiche, connessioni) anche sul campo come EPROM, PAL, PLA, PLD (studiati nel corso di Reti Logiche)
- Anni 80/90: introduzione di dispositivi logici ad elevata scala di integrazione di FPGA (e CPLD), programmabili sul campo mediante linguaggi HDL

2

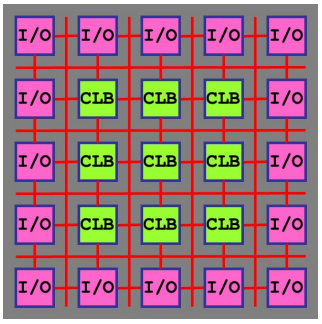
Caratteristiche dei dispositivi FPGA

- FPGA: composti da blocchi logici configurabili (Configurable Logic Blocks – CLB) che possono essere interconnessi tra loro in modo trasparente all'utente utilizzando linguaggi di programmazione di alto livello (e.g. HDL o HLS)
- Programmabili (e riprogrammabili) sul campo
- Relativamente poco costosi, rapido time to market, supportati da linguaggi di alto livello (C/C++) o Hardware Description Language (VHDL o Verilog)
- Ideali per lo sviluppo rapido di prototipi (e.g. usati per lo sviluppo di microprocessori)
- Ideali per dispositivi a basso consumo
- Consentono un elevato livello di astrazione -> è possibile implementare in hardware non solo reti logiche tradizionale ma anche algoritmi
- E' possibile integrare cores (sia soft che hard)
- FPGA vs ASIC ($$) – in entrambi i casi possono essere utilizzati linguaggi HDL

3

Moduli principali dei dispositivi FPGA

Una FPGA consiste in un insieme di Configurable Logic Blocks (CLB) che possono essere connessi tra loro. La funzione dei singoli CLB e delle connessioni viene impostata dal progettista mediante programmazione ("sul campo"). Tale programmazione può essere ripetuta più volte (teoricamente "infinite").



- Numero di CLB molto elevato (migliaia)
- Generalmente è anche disponibile della RAM interna - Block RAM (centinaia di KByte)
- Alcuni blocchi sono dedicati all'I/O
- Disponibili (decine) di sommatori, moltiplicatori

4

Tecnologie che caratterizzano i dispositivi FPGA

- Esistono diversi produttori di FPGA e differenti tecnologie
- FPGA di produttori diversi si differenziano principalmente per due aspetti principali:
  - **Tecnologia utilizzata per le connessioni**
    - Fusibili
    - Memorie flash
    - Memorie SRAM
  - **Struttura dei CLB**

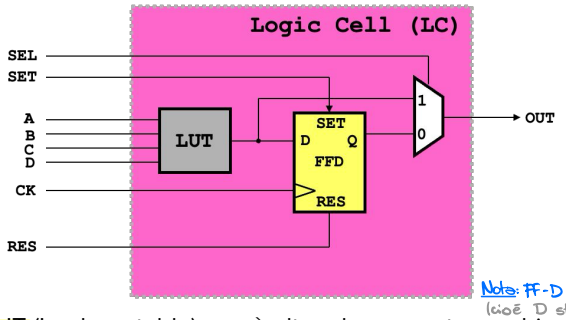
    Sebbene un'analisi dettagliata dei blocchi logici configurabili (CLB) di una tipica FPGA esuli dagli obiettivi di questo corso, è molto interessante capire come tali blocchi sono organizzati

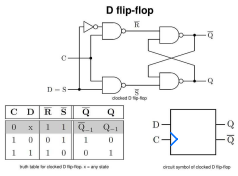
5

Logic Cell come elemento base dei CLB

- Lo schema seguente mostra, in forma semplificata, una rete denominata Logic Cell (LC) che potrebbe essere alla base di un ipotetico CLB

LUT + FF-D + MUX





FF-D funziona come un latch ma è clock-triggered
=> espone il valore in ingresso al fronte di salita del clock

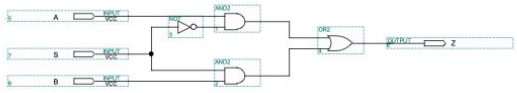
Nota: FF-D deve rispettare setup time e hold time.
(cioè D stabile prima del fronte e dopo, per un t_s/t_h)

- Il blocco LUT (Look-up-table) non è altro che una rete combinatoria programmabile (vedi pg successiva)
- Il blocco LUT, può essere anche riprogrammato per agire come uno shift-register o un una memoria (distributed RAM)

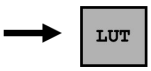
6

Funzioni combinatorie e Look-Up-Table (LUT)

- Una LUT può essere utilizzata per realizzare una qualsiasi rete combinatoria



S	A	B	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

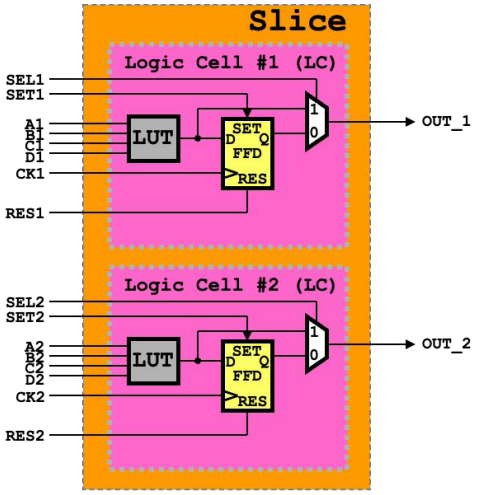


- Oppure per essere configurata come un convenzionale elemento di memoria (distributed RAM)

7

Slices come combinazione di Logic Cells

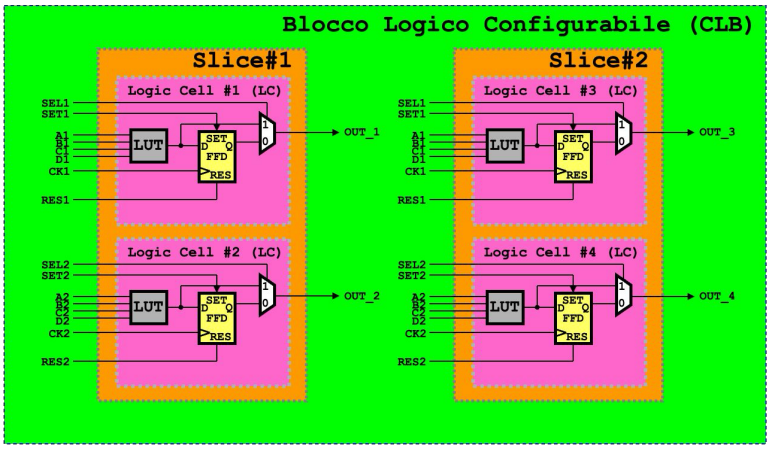
- Tipicamente più Logic Cell sono raggruppati per formare degli Slice



8

CLB come combinazione di Slice

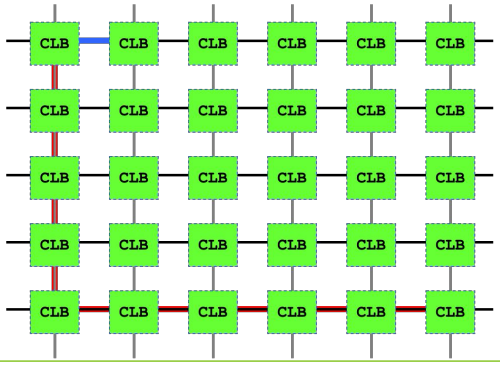
- Infine, più Slice sono raggruppati per formare un CLB (Configurable Logic Block)



9

CLB e connessioni

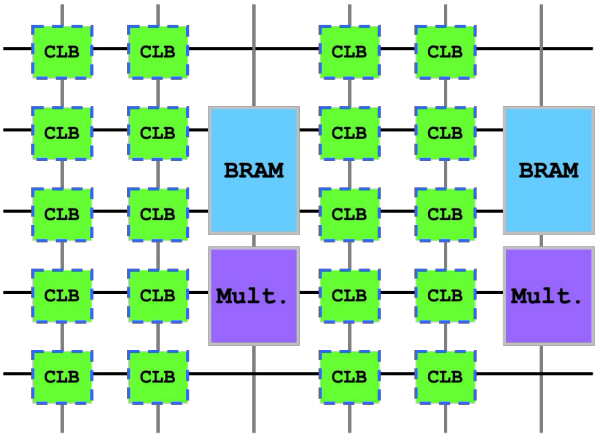
- All'interno della FPGA esistono delle connessioni configurabili tra i vari CLB, per i clock, etc
- Le connessioni sono molte ma non infinite; inoltre, sorgono problematiche di routing/ritardi causati da differenti percorsi (e.g. rosso e blu)



10

CLB e moduli addizionali

- In realtà, tra i vari CLB sono presenti altre unità funzionali (Block RAM, moltiplicatori,...)



11

Esempio di FPGA reale

Spartan-6 FPGA Feature Summary

Table 1: **Spartan-6 FPGA Feature Summary by Device**

Device	Logic Cells(1)	Configurable Logic Blocks (CLBs)			DSP48A1 Slices(3)	Block RAM Blocks		CMTs(5)	Memory Controller Blocks (Max)(6)	Endpoint Blocks for PCI Express	Maximum GTP Transceivers	Total I/O Banks	Max User I/O
		Slices(2)	Flip-Flops	Max Distributed RAM (Kb)		18 Kb(4)	Max (Kb)						
XC6SLX4	3,840	600	4,800	75	8	12	216	2	0	0	0	4	132
XC6SLX9	9,152	1,430	11,440	90	16	32	576	2	2	0	0	4	200
XC6SLX16	14,579	2,278	18,224	136	32	32	576	2	2	0	0	4	232
XC6SLX25	24,051	3,758	30,064	229	38	52	936	2	2	0	0	4	266
XC6SLX45	43,661	6,822	54,576	401	58	116	2,088	4	2	0	0	4	358
XC6SLX75	74,637	11,662	93,296	692	132	172	3,096	6	4	0	0	6	408
XC6SLX100	101,261	15,822	126,576	976	180	268	4,824	6	4	0	0	6	480
XC6SLX150	147,443	23,038	184,304	1,355	180	268	4,824	6	4	0	0	6	576
XC6SLX25T	24,051	3,758	30,064	229	38	52	936	2	2	1	2	4	250
XC6SLX45T	43,661	6,822	54,576	401	58	116	2,088	4	2	1	4	4	296
XC6SLX75T	74,637	11,662	93,296	692	132	172	3,096	6	4	1	8	6	348
XC6SLX100T	101,261	15,822	126,576	976	180	268	4,824	6	4	1	8	6	498
XC6SLX150T	147,443	23,038	184,304	1,355	180	268	4,824	6	4	1	8	6	540

Notes:

1. Spartan-6 FPGA logic cell ratings reflect the increased logic cell capability offered by the new 6-input LUT architecture.
2. Each Spartan-6 FPGA slice contains four LUTs and eight flip-flops.
3. Each DSP48A1 slice contains an 18 x 18 multiplier, an adder, and an accumulator.
4. Block RAMs are fundamentally 18 Kb in size. Each block can also be used as two independent 9 Kb blocks.
5. Each CMT contains two DCMs and one PLL.
6. Memory Controller Blocks are not supported in the -3N speed grade.

12

FPGA e prestazioni

- Le frequenze di funzionamento delle FPGA non sono molto elevate se confrontate a quelle dei processori (alcune centinaia di MHz, spesso più basse).
- Come è possibile essere competitivi con frequenze di funzionamento così ridotte?
- Configurando la FPGA per sfruttare al massimo il parallelismo
- Una nota (molto) positiva di questo approccio; riducendo la frequenza si riduce l'assorbimento di potenza ma con elevato parallelismo si ottengono eccellenti prestazioni

13

FPGA: linguaggi di programmazione/progettazione

- Come configurare una FPGA connettendo i vari CLB, etc al fine di realizzare un determinata logica?
- Per questa finalità (sintesi) si utilizzano i linguaggi HDL o tool di sintesi ad alto livello (HLS) e compilatori che generano i dati di configurazione (bitstream)
- Il bitstream è un sorta di software che configura la FPGA per svolgere un determinato compito/rete
- Il bitstream è generato a partire dal codice HDL (o dallo schematico, anche se oggi non più) o da tool di sintesi ad alto livello (quasi) senza l'intervento del progettista/programmatore (a meno che non desideri esplicitamente intervenire ma tipicamente questo non accade)

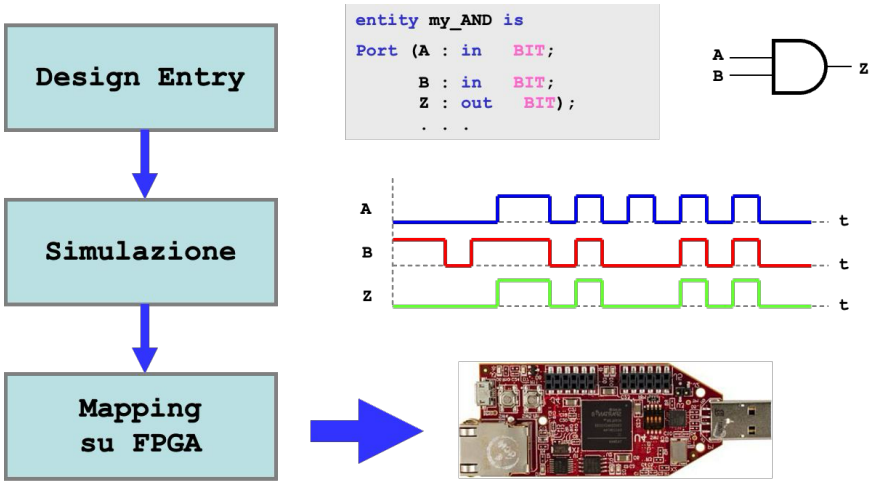
14

Progettazione con linguaggi HDL 1/2

- Linguaggi ad alto livello finalizzati alla simulazione di circuiti digitali ma sempre più utilizzate (anche) per la sintesi
- L'obiettivo è quello di ridurre i tempi e i costi di sviluppo, consentendo la gestione di progetti di grandi dimensioni, etc
- li linguaggio HDL (e.g. VHDL o Verilog) consentono di modellare il comportamento dell'hardware, ovvero: intrinseco parallelismo e ritardi
- Tipicamente utilizzati per programmare gli FPGA
- Utilizzati anche per realizzare ASIC, processori, etc
- Standard IEEE

15

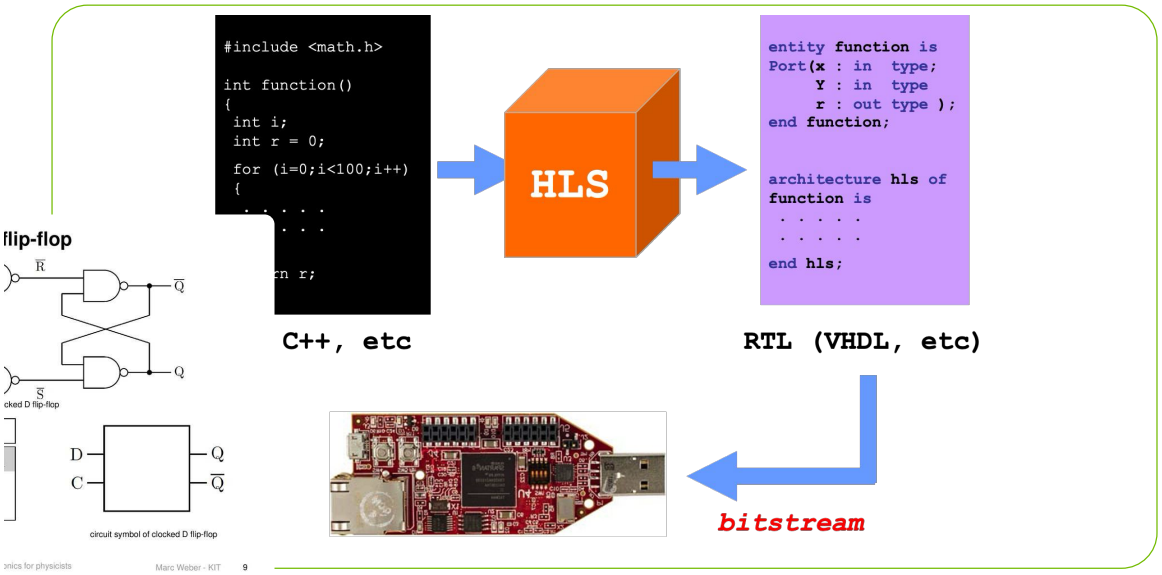
Progettazione con linguaggi HDL 2/2



Tutte e fasi sono eseguite al calcolatore

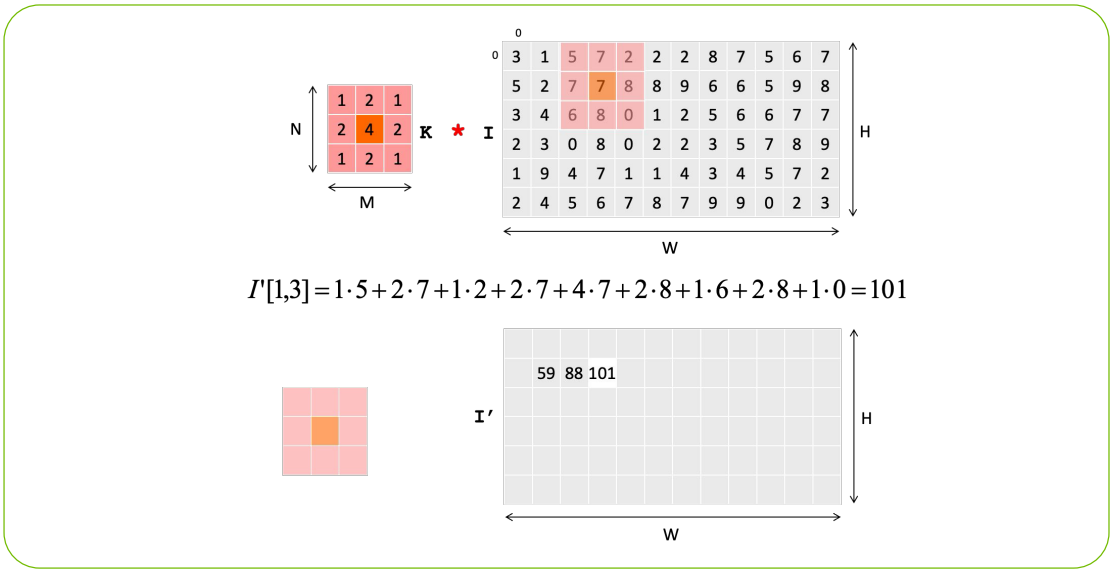
16

Progettazione con strumenti di High Level Synthesis (HLS)



17

Filtro di convoluzione con elaborazione *streaming* con FPGA



18

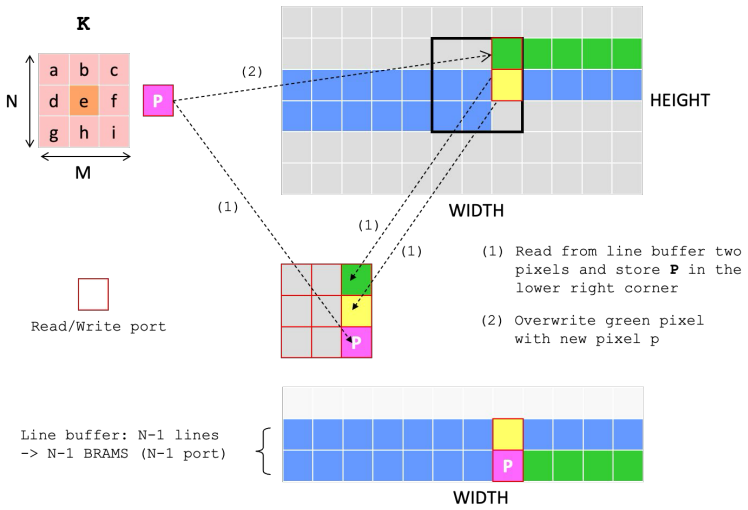
Struttura base codice HLS

- Utilizzando HLS, è possibile descrivere la convoluzione nel modo seguente
- Le #pr agma servono per guidare il compilatore (C -> VHDL/Verilog)

```
#include "ap_int.h"
typedef ap_uint<8> pixel;

void Filter(pixel input_img[640*480], pixel output_img[640*480])
{
#pragma HLS INTERFACE axis port=out_img
#pragma HLS INTERFACE axis port=in_img
    . . .
Loop_row: for (int row = 0; row < HEIGHT + (N-1)/2; row++)
    Loop_col: for int col = 0; col < WIDTH + (M-1)/2; col++)
    { #pragma HLS PIPELINE II=1

        // filter code here

    }
}
```

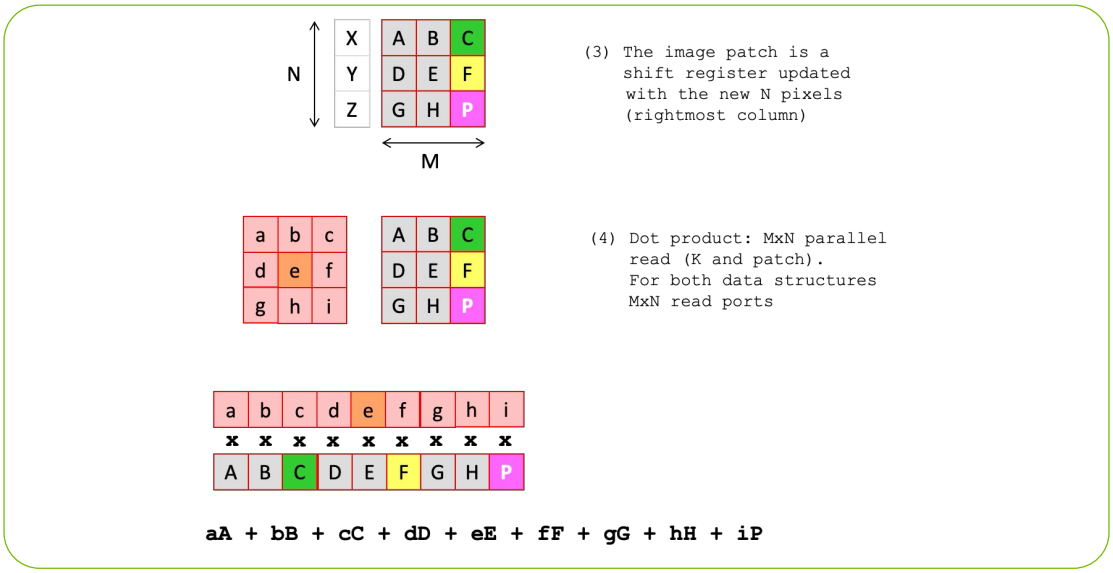
19

Data structures: line buffer (BRAM)



20

Data structures: image patch e kernel (FFD)

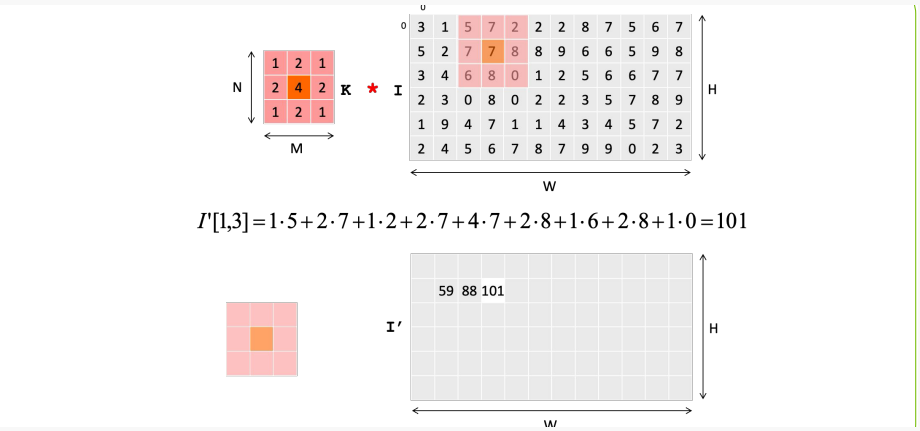


21

Spiegazione Esempio BRAM FPGA

L'esempio trattato (*Slide 20*) è un caso classico di ottimizzazione dell'hardware per il calcolo in streaming. La convoluzione 2D è la base per moltissime elaborazioni di immagini (come sfocatura, sharpening, edge-detection). Si ottiene moltiplicando una finestra di pixel dell'immagine `I` per una matrice filtro `K` (costante) e sommando tutti i prodotti (dot product). Il risultato diventa il valore del pixel corrispondente (al pixel centrale della patch) nell'immagine di output.

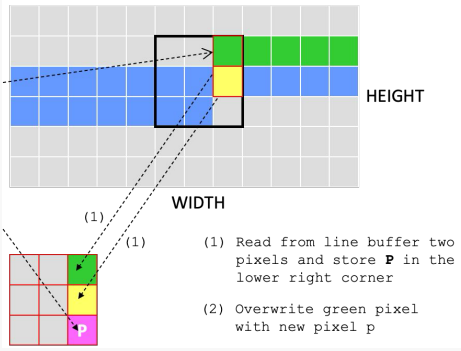
Il problema fondamentale è che in un flusso video i pixel arrivano serialmente (uno per ogni clock), ma per una convoluzione 3×3 servono 9 pixel contemporaneamente.



Si sfrutta la FPGA usando le **BRAM** (Block RAM), memorie situate tra i CLB, per creare dei Line Buffer in cui salvare i pixel del flusso.
Un Line Buffer contiene esattamente una riga intera dell'immagine.

Abbiamo detto che e immagini arrivano dalla memoria esterna riga per riga. Quando si riceve il pixel corrente (chiamiamolo `P`), esso rappresenta l'angolo in basso a destra della finestra 3×3. Ma per fare il calcolo servono anche i pixel delle due righe precedenti, che sono passati molti cicli di clock prima. Per non dover rileggere ogni volta i dati dalla RAM esterna (lenta), la FPGA usa le sue BRAM interne per salvare temporanenamente i dati (che chiameremo `Line Buffers`).

Un Line Buffer contiene esattamente una riga intera dell'immagine.
Per una finestra $N \times N$, servono $N - 1$ Line Buffer. Per una convoluzione 3×3, ne servono 2.

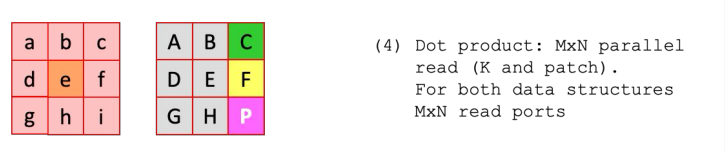


Funzionamento:

1. Quando arriva il nuovo pixel `P`, la FPGA lo salva nella BRAM.
2. Contemporaneamente, si leggono dalle BRAM i pixel che si trovano esattamente sopra `P` nelle righe precedenti. In questo modo, ad ogni ciclo di clock, otteniamo una colonna verticale di 3 pixel "allineati".

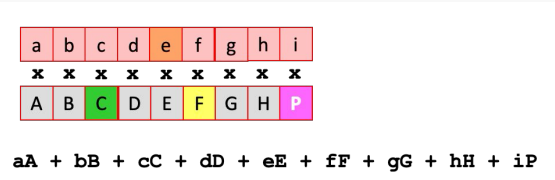
Adesso é necessario costruire la finestra interna 3×3 (quella della convoluzione). Qui passiamo dalle BRAM ai **Flip-Flop** (FFD), in cui saranno salvati i 9 elementi su cui eseguire le moltiplicazioni in parallelo.

3. Questa colonna entra in una struttura di FF-D organizzata come **Shift Register**. Ad ogni clock, i dati scorrono verso sinistra di una posizione.
4. Dopo 3 clock, i 9 FFD contengono l'intera patch di immagine 3×3 necessaria per il calcolo. I valori del filtro `K` sono invece memorizzati stabilmente in altri 9 FFD (o costanti).



Matrice filtro K *Matrice patch* I

5. I FF-D vengono usati per le moltiplicazioni (9) in parallelo e la somma di tutti i risultati (dot product) in un unico clock.



👉 ***Nota***: Grazie a questa pipeline, dopo il ritardo iniziale per riempire i buffer, (a regime) la FPGA è in grado di produrre un pixel filtrato ad ogni ciclo di clock, garantendo prestazioni altissime.

2. Modulo 2 - CUDA

2.0 Introduzione

2.0.1. Un po' di storia...

Negli anni '60-'80 la CPU gestiva tutto. Con l'aumento della richiesta di interfacce grafiche e lo sviluppo di videogiochi, la CPU è presto diventata un collo di bottiglia a causa della sua natura sequenziale nell'eseguire i propri compiti. A questo si aggiunngeva il fatto che, usando la CPU anche per la grafica, veniva di conseguenza rallentata l'elaborazionne.

L'evoluzione é partita dai **coprocessori grafici** (~1977), i quali liberano la CPU dalla gestione grafica, consentendo giochi ed interfacce più complesse. Successivamente si sono susseguite diverse schede (*MDA, CGA, EGA*) che hanno portato via via al primo predecessore delle moderne GPU. Queste schede tuttavia avevano specifiche differenti, creando problemi di compatibilità notevoli.

Nel 1987 é stato introdotto lo standard **VGA**, che ha definito le specifiche per schede video e monitor, diventando un riferimento dotato di retrocompatibilità con alcuni dei modelli precedenti (*CGA ed EGA*). Di questo standard fanno parte:

- **Controller VGA**: chip specializzato che implementa lo standard gestendo l'output grafico verso il monitor e la generazione dei segnali analogici RGB.
- **Scheda video VGA**: la scheda contenente il controller VGA e la memoria video (*256KB*), destinata a gestire la visualizzazione su monitor.
- **Connettore VGA**: connettore analogico DE-15 a 15 pin per la connessione ai monitor.

Queste schede sono state progettate inizialmente per gestire grafica 2D e basse risoluzioni.

Figura 13: Scheda e Connettore VGA.

L'avvento di VGA ha portato allo sviluppo di funzionalità 3D basilari (triangolazione, rasterizzazione, texture mapping, shading) e del **Ray Tracing**, tecnica di rendering che simula il comportamento fisico della luce. Nel Ray Tracing ogni pixel richiede il calcolo di più raggi luminosi e delle loro interazioni (riflessioni, rifrazioni ed ombre), cosa che si presta bene al parallelismo.

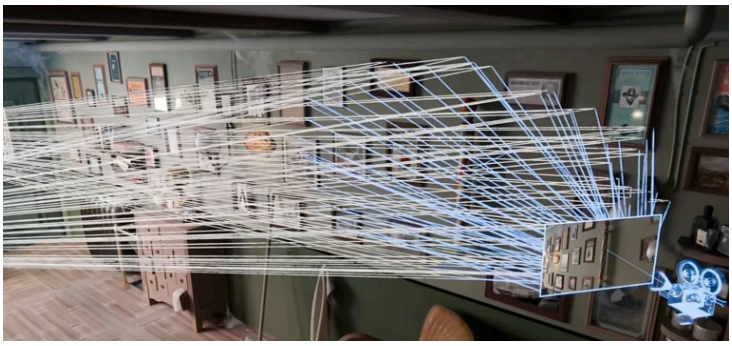


Figura 14: Visualizzazione Ray Tracing.

Figura 15: RT Off vs On

Negli anni '90 sono nati i **Graphics Accelerator**, hardware specializzato progettato per accelerare le operazioni grafiche, riducendo il carico sulla CPU. A causa della scarsa flessibilità, delle prestazioni insufficienti e della manncanza di programmabilità, anche questa tecnologia é diventata presto stretta.

In diversi ambiti si ha avuto una spinta verso l'evoluzione dell'hardware grafico, in particolare da:
- industria dei videogiochi (rendering 3D complesso ed interattivo),
- industria cinematografica (elaborazione di effetti visivi e simulazioni realistiche),
- HHPC (simulazioni scientifiche complesse e calcolo numerico intensivo),
- industria medica (ricostruzione e visualizzazione di immagini 3D da TAC e RMN),
- ...

Si é quindi giunti alle **GPU** (Graphics Processing Unit), le quali hanno segnato un punto di svolta.

2.0.2. GPU

📖 **GPU**

Le GPU sono **hardware** progettato per l'**elaborazione parallela**, ottimizzato per gestire e accelerare il **rendering grafico** e le **operazioni di calcolo intensivo**.

Il rendering grafico consiste di trasformazione di modelli 3D in immagini 2D e calcolo di luci, ombre ed effetti visivi in tempo reale.

La caratteristica principale delle GPU é il **calcolo parallelo massivo**, ossia l'esecuzione di migliaia di operazioni simultaneamente.

Sebbene siano nate per la grafica, le GPU hanno da subito mostrato potenziale per altri tipi di calcolo. L'intuizione chiave é stata sfruttare la struttura parallela, applicandolo a domini diversi da quello grafico (general purpose).

📖 **GPGPU**

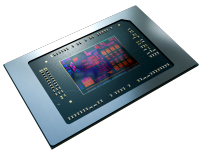
Per **GPGPU** (General Purpose computing on Graphics Processing Units), si intende l'utilizzo delle GPU per calcoli non grafici.

I primi approcci si sono rivelati complessi e limitati poiché era necessario usare **API grafiche** (OpenGl o DirectX) per presentare i calcoli general purpose come operazioni grafiche. La programmazione inoltre era complessa, limitata, senza strumenti di debug e profiling. Con la crescente domanda sono stati sviluppati **framework** e **linguaggi** specifici come **CUDA** per **NVIDIA** e OpenCL, i quali hanno semplificato notevolmente la programmazione GPGPU.

La classificazione odierna si basa su due architetture:

GPU dedicata

- collegata alla scheda madre,
- prestazioni **elevate**,
- memoria video separata (**VRAM**),
- consumo maggiore,
- costo maggiore,
- ideale per gaming e lavori intensivi,
- usata principalmente in PC da gaming, workstation, server.



GPU integrata

- incorporata nel **processore**,
- prestazioni moderata,
- condivide la **memoria di sistema**,
- consumo ridotto,
- economica (inclusa nel processore),
- sufficiente per attività quotidiane,
- comune in laptop, ultrabook e dispositivi mobili.

Nelle moderne console (ma non solo) si utilizzano **SoC** (System on Chip), ossia sullo stesso chip sono ospitate CPU, GPU e memoria primaria. In questo contesto la GPU é dedicata, ma condivide la RAM (memoria unificata) con la CPU.

Nel mercato odierno la situazione é dominata da **NVIDIA**, leader del mercato HPC, AI e gaming, la quale sfrutta tecnologie come CUDA e Tensor Cores. Altre aziende blasonate sono **Qualcomm**, leader per i dispositivi mobili, utilizza SoC Snapdragon (Adreno); **AMD**, secondo produttore di GPU, impiega ROCm, RDNA; **Intel**, new entry nel mercato GPU e **Apple** che usa GPU integrane nei chip della serie M (Metal API).

2.0.3. CPU vs GPU

CPU e GPU nascono per adempiere a scopi ben diversi, pertanto anche le loro caratteristiche sono differenti.

CPU vs GPU

Caratteristica	CPU	GPU
Core	**Pochi core** complessi (4–128)	**Centinaia/migliaia di core** semplici (es. 16384 CUDA cores)
Frequenza	2.5–5 GHz	1–2 GHz
Architettura	Ottimizzata per **bassa latenza** e **controllo complesso** (ILP, OOO, branch prediction)	Ottimizzata per alto throughput (SIMT, parallelismo massivo)
Memoria	**DDR4/DDR5** condivisa con il sistema operativo	**GDDR6/HBM** dedicata, non condivisa
Banda Memoria	Fino a 100 GB/s	Fino a 3.4 TB/s (con HBM2/3)
Cache	**Cache multi-livello** (L1, L2, L3), grandi e sofisticate per core	**Cache più piccole**; L1/shared configurabile, L2 globale
Versatilità	**Alta**: eccellente per **carichi sequenziali** e **controllo complesso**	**Limitata**: eccellente solo per **carichi altamente parallelizzabili**
Latenza	**Molto bassa** per singole operazioni	**Più alta** per singole operazioni, ottimizzata per carichi bulk
Parallelismo	**Limitato** (decina fino a centinaia di thread)	**Massivo** (migliaia di thread simultanei)
Consumo	**Moderato**: scalabile in base al carico	**Alto**: molto energivora sotto carichi intensivi
Scalabilità	Difficile oltre 64–128 core (vincoli termici, sincronizzazione)	Ottima scalabilità orizzontale (**multi-GPU, migliaia di core**)
Programmazione	**Più semplice** (general-purpose, toolchain consolidata)	**Più complessa**: richiede modelli come CUDA/OpenCL

Esempio

Un esempio concreto:

Caratteristica	AMD Ryzen 9 9950X (CPU) - 2024	NVIDIA GeForce RTX 5090 (GPU) - 2025
Architettura	Zen 5 (Granite Ridge)	Blackwell
Numero di Core	16 core / 32 thread	21,760 CUDA core
Frequenza Base	4.3 GHz	2.017 GHz
Frequenza Boost	Fino a 5.7 GHz	Fino a 2.407GHz
Cache	80 MB (L2 + L3)	98 MB L2
Memoria Supportata	DDR5-5600	32 GB GDDR7X
Larghezza di Banda	~100 GB/s (con DDR5)	1,008 GB/s
TDP (Thermal Design Power)	170W	575W
Transistor	~16.6 miliardi	92.2 miliardi
Prezzo di Lancio	$649	$1,999
Prestazioni FP32	~1.8 TFLOPS	104.8 TFLOPS

- **Verde**: Vantaggio significativo
- **Arancione**: Svantaggio relativo
- **Nero**: Aspetto Neutro

Data questa eterogeneità e i punti di forza di entrambi, si potrebbe trarre beneficio da un impiego misto.

2.0.4. Architetture eterogenee

Architettura Eterogenea

Un'**architettura eterogenea** combina unità di elaborazione diverse (per esempio CPU e GPU), le quali collaborano sfruttando le caratteristiche di ciascuna unità.

Nel nostro caso ci concentreremo su **CPU e GPU**. La prima gestisce l'ambiente, il codice ed i dati, mentre la seconda fa da co-processore, accelerando calcoli intensivi (**hardware accelerator**). CPU e GPU sono collegate tramite bus **PCI-Express** e il codice delle applicazioni é diviso in codice host (CPU) e codice device (GPU).

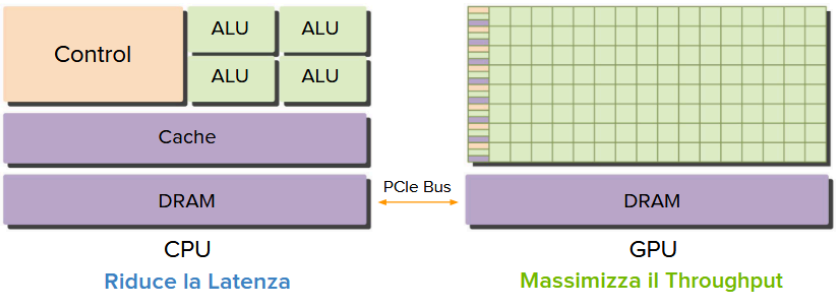


Figura 16: Architettura eterogeneca CPU+GPU.

Sistema Multi-GPU

Un **sistema multi-GPU** integra all'interno della stessa **architettura eterogenea** più GPU, permettendo di **distribuire** e **parallelizzare** il carico di lavoro sulle diverse unità di elaborazione grafica.

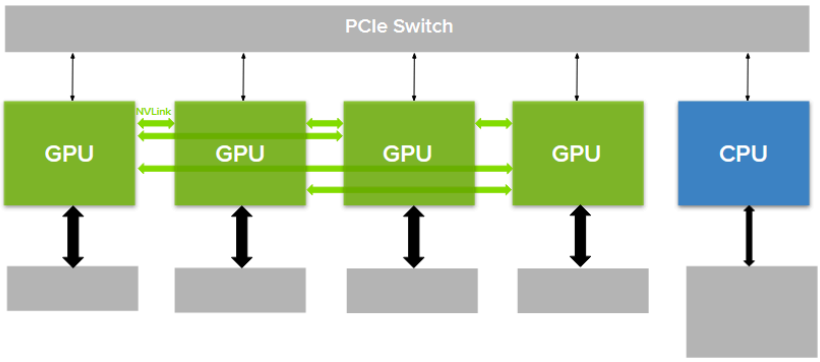


Figura 17: Architettura Multi-GPU.

Lo scope di applicabilità di CPU e GPU si differenzia sulla base di due dimensioni: livello di **parallelismo** e dimensione dei **dati**. Tipicamente la CPU ha pochi thread, contro i migliaia della GPU, quindi un basso livello di parallelismo, inoltre la CPU gestisce dati di piccole/medie (KB/MB) dimensioni, mentre la GPU di grandi/massive (GB/TB) dimensioni.

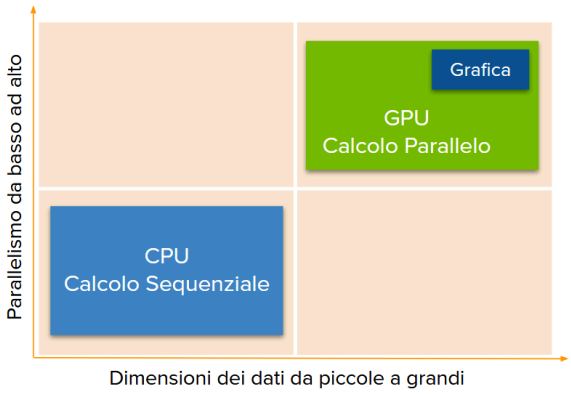


Figura 18: Trade-off CPU e GPU.

Combinado CPU e GPU si ottengono le massime prestazioni:

- Basso parallelismo e dati limitati sfruttano al massimo la CPU,
- Alto parallelismo e dati massicci impiegano al meglio la GPU.

2.0.5. CUDA

Il **codice Host** (CPU) gestisce il flusso generale del programma, si occupa delle operazioni di controllo e I/O e prepara ed invia dati ed istruzioni alla GPU.

Il **codice Device** (GPU) contiene funzioni specializzate chiamate **kernel**, esegue **calcoli intensivi** in parallelo ed elabora **grandi quantità** di dati simultaneamente.

> 📖 **CUDA**
>
> **CUDA** é una piattaforma di calcolo **parallelo general-purpose** ideato da **NVIDIA** (2007). Ma é anche un **modello di programmazione** per GPU NVIDIA (esclusivamente).
>
> CUDA é uno **standard de facto**.

Il suo obiettivo é sfruttare la potenza di calcolo parallelo delle GPU e semplificare lo sviluppo per sistemi CPU-GPU.

CUDA dispone di un vero e proprio **ecosistema**: compilatore (nvcc), profiler/debugger (Nsight), librerie ottimizzate, strumenti di sviluppo (CUDA Toolkit). Inoltre offre un ampio set di librerie ottimizzate (cuBLAS, cuDNN, Thrust, etc.) e strumenti di sviluppo per facilitare l'ottimizzazione e il debugging.

Per accedere a CUDA si possono sfruttare diverse metodologie:
- API (CUDA Runtime API, CUDA Driver API),
- Estensioni per C, C++ e Fortran,
- Binding Python (Numba, CuPy, PyCUDA),
- Integrazione con framework (come TensorFlow, Pytorch).

Il **CUDA Toolkit** è un insieme completo di **strumenti di sviluppo** fornito da NVIDIA per creare applicazioni accelerate tramite GPU. Fornisce tutti gli strumenti necessari per scrivere, compilare e ottimizzare codice CUDA. I componenti chiave del CUDA Toolkit sono:

- **Driver NVIDIA**, essenziali per CUDA, ma sostanzialmente nascosti, funzionano da ponte tra sistema operativo e GPU, gestendo l'hardware, il caricamento del codice e il trasferimento di dati tra CPU e GPU.
- **CUDA Runtime/CUDA Driver API**, a scelta, il primo fornisce un livello di astrazione più alto ed é più semplice da utilizzare, il secondo é di livello più basso.
- **Compilatore CUDA** (NVCC - NVIDIA CUDA Compiler), traduce il codice CUDA 5 scritto in linnguaggi come C, C++, Fortran in un formato eseguibile dalle GPU NVIDIA. Le fasi che segue il compilatore sono:
  - **Separazione** del codice destinato alla CPU da quello per la GPU.
  - **Compilazione** del codice GPU in un linguaggio intermedio (PTX) o direttamente in codice macchina per l'architettura GPU target (tenendo conto delle sue caratteristiche, ossia *Compute Capability* e *CUDA Version*).
  - **Linking** del codice CPU e GPU con le librerie CUDA.

  Si ottiene quindi un unico gile che contiene sia il codice GPU che quello CPU.
- **Librerie CUDA**, che forniscono implementazioni **ottimizzate e parallele** di operazioni comuni. Tra queste cuBLAS (algebra lineare), cuFFT (analisi di segnali ed elaborazione immagini), cuDNN (primitive per deep neaural networks), cuRAND (numeri casuali), cuSPARSE (operazioni su matrici sparse), Thrust (varie).
- **Strumenti di Debugging**
  - **Nsight Systems**, per **profiling di sistema** del comportamento dell'applicazione su CPU e GPU, evidenzialo eventuali bottle-neck.
  - **Nsight Compute**, per **profiling della GPU** approfondito, permette analisi di prestazioni dei kernel in dettaglio per l'identificazione di aree di ottimizzazione per la memoria e l'utilizzo dei core.
  - CUDA-GBD, per il debugging a riga di comando del codice CUDA da sorgente (sia CPU che GPU).
  - *NVIDIA Visual Profiler (NVVP), non più supportato, offre rappresentazioni grafiche timeline delle attività di CPU e GPU, le sue funzionalità sono state integrate in Nsight.*

> ⚠️ **CUDA Toolkit e CUDA Version**
>
> Il **CUDA Toolkit** viene rilasciato in **versioni numerate**. (del tipo `CUDA X.Y`) La versione installata determina la **CUDA Version** del sistema. Le diverse versioni includono aggiornamenti software, bugfix, nuovi strumenti, librerie e supporto alle nuove architetture.

Le nuove versioni del CUDA Toolkit mantengono la compatibilità con GPU più vecchie, anche se non tutte le funzionalità più recenti sono disponibili su queste architetture.

In generale, il codice é retrocompatibile, cioè quello scritto con versioni precedenti del Toolkit può essere eseguito su versioni più recenti, ma può richiedere piccoli adattamenti.

Compute Capability (CC)

La **Compute Capability** (CC) é un numero in formato `X.Y` che identifica le **caratteristiche** e le **capacità** di una GPU NVIDIA in termini di funzionalità **supportate** e **limiti hardware**. `X` indica la generazione architetturale, mentre `Y` indica una revisione della stessa architettura con miglioramenti o varianti hardware.

Relazione tra CC e CUDA Version

La **CC** é indipendente dalla **CUDA Version**, ma quest'ultima deve supportare la CC della GPU per sfruttare appieno le sue capacità. Una stessa CUDA Version può supportare più CC.

Nel susseguirsi delle generazioni, ognuna ha portato significativi avanzamenti nelle capacità di calcolo e nell'efficienza energetica. L'evoluzione riflette il passaggio da un focus su grafica e HPC a un'enfasi crescente su AI, deep learning e calcolo ad alte prestazioni.

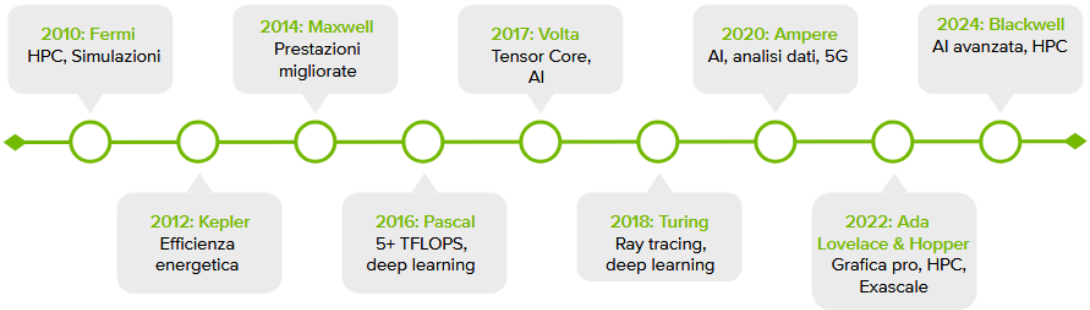


Figura 19: Evoluzione

I file di codice sorgente CUDA hanno estensione `.cu` e contengono codice host e codice device.

Il codice host é codice C/C++ eseguito dalla **CPU**. Questo gestisce la **logica dell'applicazione**, **alloca memoria** sulla GPU, **trasferisce dati** tra CPU e GPU, **lancia kernel** GPU e gestisce la sincronizzazione.

Il codice device é codice **CUDA C** eseguito dalla GPU, il quale contiene i **kernel** (funzioni parallele) ed esegue operazioni computazionali intensive in parallelo.

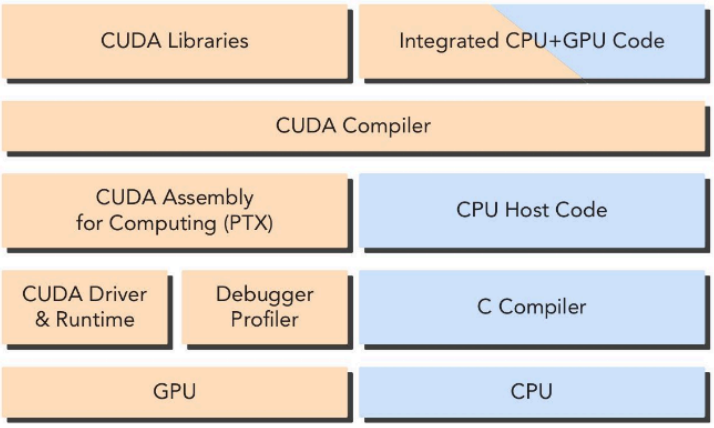


Figura 20: CUDA stack

Per quanto riguarda il **flusso di compilazione** vale quanto detto in precedenza, precisando che la compilazione del codice device, una volta ottenuto nel linguaggio intermedio PTX (Parallel Thread Execution), questo viene tradotto in codice macchina GPU (SASS-Assembly) al momento dell'esecuzione tramite compilatore **Just-In-Time**.

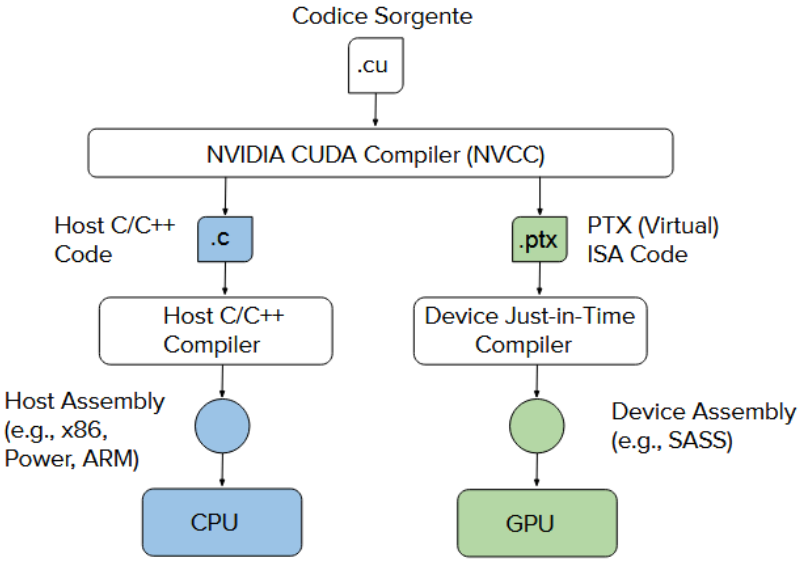


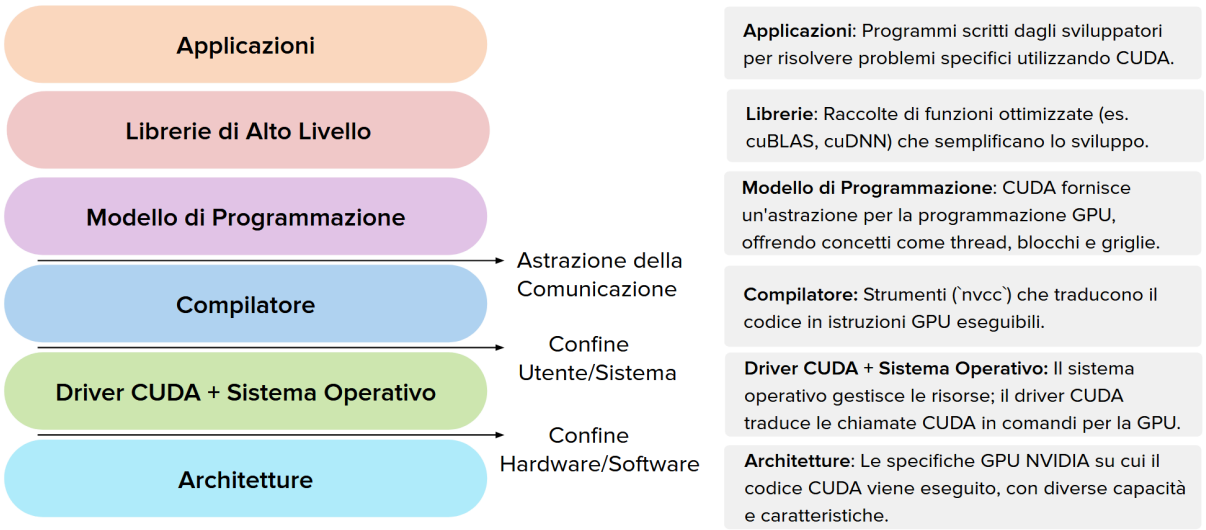
Figura 21: Compilazione codice CUDA

CUDA estende C/C++ con costrutti specifici (come `__global__`, `__shared__` e la sintassi `<<<...>>>` per lanciare kernel sulla GPU).

> ⚠ **Codice sequenziale → esecuzione parallela**
>
> Ogni kernel viene scritto come **codice sequenziale**, ma viene **eseguito in parallelo** da migliaia di thread.

2.1. Modello di programmazione CUDA

Il modello CUDA é un ecosistema stratificato per algortmi paralleli su GPU, con semplicità e controllo hardware ottimizzati, come accennato nella Sezione 2.0.5.



Modello di Programmazione

Il **modello di programmazione** definisce la **struttura** e le **regole** per sviluppare applicazioni parallele su GPU. Gli elementi fondamentali sono:

- esecuzione parallela **organizzata gerarchicamente** in **thread**, **blocchi** e **griglie**.
- **gerarchia di memoria** (globale, condivisa, locale, costante, texture), ciascuna con diverse prestazioni e scopi, per ottimizzare l'accesso ai dati.
- funzioni e librerie per gestire esecuzione, trasferimento di dati e operazioni tramite API.

Programma (vs modello di programmazione)

Il programma rappresenta l'**implementazione concreta** che specififca (in CUDA) come i thread condividono dati, accedono alla memoria, si sincronizzano e coordinano le loro attività.

2.1.1. Livelli di astrazione

Il calcolo parallelo si articola in tre livelli di astrazione: dominio, logico e hardware.

Dominio (livello)

Il focus del **dominio** é la **decomposizione** del problema, con cui viene definita la struttura parallela di alto livello.

Scopo: Ottimizzare la strategia di parallelizzazione.

Logico (livello)

Il livello **logico** tratta l'organizzazione e la gestione dei thread a stretto contatto con l'implementazione della strategia di parallelizzazione.

Scopo: Massimizzare l'efficienza del parallelismo.

Hardware (livello)

Il livello **hardware** mappa l'esecuzione sull'architettura della GPU, concentrandosi sull'ottimizzazione delle prestazioni hardware.

Scopo: Sfruttare al meglio le risorse GPU.

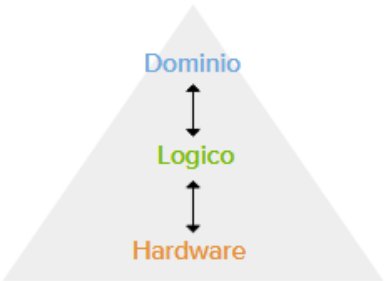


Figura 22: Livelli di Astrazione nella Programmazione Parallela CUDA

Esempio: Moltiplicazione di Matrici

- *Dominio*: Suddivisione delle matrici.

- *Logico*: Organizzazione dei thread per i calcoli.
- *Hardware*: Ottimizzazione accesso memoria ed esecuzione sui core GPU

📖 **Thread**

Un **thread** CUDA rappresenta un' **unità di esecuzione elementare** nella GPU , che esegue su un sottoinsieme dei dati. Ogni thread CUDA esegue una porzione di un programma **parallelo** , chiamato **kernel**.
I thread dello stesso kernel eseguono il **medesimo codice**, operando su **dati diversi** determinati dagli identifiatori univoci del thread stesso (*threadIdx*, *blockIdx*).
Ogni thread possiede un **proprio stato** (*PC, registri, memoria locale, ...*), pertanto anche una propria **memoria privata** (registri).

⚠️ **Sequenzialità dei thread**

Sebbene migliaia di thread vengano eseguiti concorrentemente sulla GPU, ogni singolo thread segue un percorso di esecuzione sequenziale all'interno del suo contesto.

La potenza dei thread GPU é il loro numero. Infatti, qeusti sono tipicamente migliaia, anche se più semplici di quelli CPU, pertanto possono gestire un **parallelismo massivo** con basso overhead di gestione.

La programmazione CUDA prevede sia codice device che codice host (v. Sezione 2.0.5). Le diverse sezioni di codice seriale e parallelo si alternano nello stesso file, pertanto il codice può essere eseguito in maniera alternata. Infatti, il codice host può continuare l'esecuzione mentre i kernel GPU sono in esecuzione. É possibile anche lanciare più kernel (anche diversi) nella stessa applicazione, anche in overlapping temporale.

Il flusso tipico di elaborazione CUDA prevede:

i. **Inizializzazione ed allocazione di memoria** host e device dall'host.
ii. **Trasferimento dei dati** dalla memoria host a quella device.
iii. **Esecuzione del kernel** sul device secondo la configurazione di griglia e blocchi.
iv. **Recupero dei risultati** dal device all'host.
v. **Post-elaborazione** dei risultati su CPU.
vi. **Liberazione risorse** (memoria allocata host e device).

I passi ii. e v. possono essere eseguiti più volte o in pipeline per massimizzzare l'overlap tra calcolo e trasferimento dati.

2.1.2. Gestione della Memoria (accenni)

Il modello su cui viene usato CUDA prevede che il sistema abbia host (CPU) e device (GPU) . La comunicazione tra memoria host e device avviene tramite **PCIe** (Peripheral Component INterconnect Express).

📖 **PCIe**

Il bus **PCIe** è un' interfaccia **seriale point-to-point** che sfrutta più **lane** indipendenti in parallelo per aumentare la banda. Una lane è un canale costituito di **due coppie di segnali differenziali** : una per ricevere (RX) ed una per trasmettere (TX). Pertanto è supportata la comuniczione **full duplex** (tramissione ee ricezione simultanea). Inoltre, garantisce trasferimenti a bassa latenza, ma può comunque diventare un collo di bottiglia in trasferimenti di grandi volumi.

👉 ***Nota*** : la larghezza di banda varia in base al numero di lane (x1, x2, x4, x8, x16).

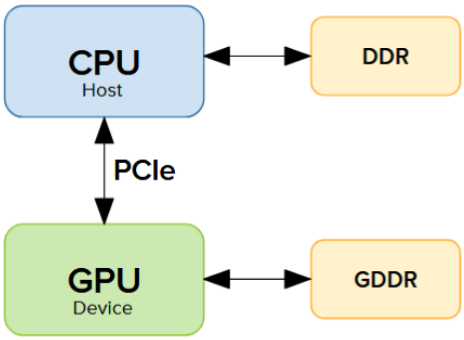


Figura 23: Schema di connessione PCIe.

⚠️ Memoria e kernel

I kernel CUDA operano sulla memoria device (GPU). Infatti non possono accedere alla memoria host.

Per accedere a dati in memoria host, questi vanno copiati in memoria device (dall'host).

Il CUDA Runtime fornisce funzioni per **allocare** e **rilasciare** memoria sul device e per **trasferire** dati **bidirezionalmente** tra memoria host e device.

Standard C	CUDA C	Funzione
`malloc`	`cudaMalloc`	**Alloca** memoria dinamica
`memcpy`	`cudaMemcpy`	**Copia** dati tra aree di memoria
`memset`	`cudaMemset`	**Inizializza** memoria a un valore specifico
`free`	`cudaFree`	**Libera** memoria allocata dinamicamente

Figura 24: Funzioni memoria principali CUDA vs standard C

In CUDA però esistono anche diversi tipi di memoria, ciascuno con caratteristiche specifiche in termini di accesso, velocità, e visibilità. Le due più importanti sono Global Memory e Shared Memory.

📖 Global Memory

La memoria globale:

- è **accessibile da tutti i thread** su tutti i blocchi,
- è **grande** (più della shared),
- persiste per tutta la durata del programma CUDA,
- è adatta a memorizzare grandi e persistenti dati.

📖 Shared Memory

La shared memory:

- è **condivisa** tra i thread di un blocco ,
- è più **rapida** ma di dimensioni limitate (della globale),
- esiste per la durata del blocco di thread,
- è utilizzata per **dati temporanei** o **intermedi**.

👉 ***Nota*** : le funzioni CUDA in Figura 24 operano principalmente sulla Global Memory.

2.1.2.1. CUDA malloc

La `cudaMalloc` è una funzione CUDA utilizzata per allocare **memoria lineare contigua** sulla GPU (device) dinamicamente. La sua firma é:

```
cudaError_t cudaMalloc(void** devPtr, size_t size)
```

👉 ***Nota***: la memoria allocata non é inizializzata.

2.1.2.2. CUDA meset

La `cudaMemset` é una funzione CUDA utilizzata per **impostare** un determinato **valore** in **un blocco di memoria** allocato sulla GPU (device). La sua firma é:

```
cudaError_t cudaMemset(void* devPtr, int value, size_t size)
```

Si usa comunemente per azzerare la memoria (`value=0`).
👉 ***Nota***: é preferibile usarla per grandi blocchi di memoria (per ridurre l'overhead).

2.1.2.3. CUDA memcpy

La `cudaMemcpy` é una funzione CUDA per il **trasferimento di dati tra memoria** host e device o all'interno dello stesso tipo di memoria. La sua firma é:

```
cudaError_t cudaMemcpy(void* dst, const void* src, size_t size, cudaMemcpyKind
kind)
```

dove `cudaMemcpyKind` é la direzione della copia:

- `cudaMemcpyHostToHost` : Da host a host.
- `cudaMemcpyHostToDevice` : Da host a device.
- `cudaMemcpyDeviceToHost` : Da device a host.
- `cudaMemcpyDeviceToDevice` : Da device a device.

👉 ***Nota***: é una **funzione sincrona**, pertanto blocca l'host fino al completamento del trasferimento.

> ⚠️ **Spazi di Memoria e Puntatori**
>
> Gli spazi di memoria CPU e GPU sono differenti, pertanto i puntatori device non devono essere dereferenziati nel codice host.

2.1.2.4. CUDA free

La `cudaFree` é una funzione CUDA utilizzata per **liberare la memoria** precedentemente allocata sul device. La sua firma é:

```
cudaError_t cudaFree(void* devPtr)
```

👉 ***Nota***: il puntatore su cui viene eseguita la free deve essere stato allocato in precedenza (e non già deallocato).
👉 ***Nota***: La deallocazione è **sincrona** e può avere overhead significativo.

> 📝 **Utilizzo funzione di memoria CUDA**
>
> ```
> const int N = 5;
> const size_t size = N * sizeof(float);
> ```

```
float h_data[N] = {1.0, 2.0, 3.0, 4.0, 5.0}; // Dati sulla CPU (Host)
float *d_data;                               // Puntatore per la GPU

// 1. Allocazione memoria sulla GPU
cudaMalloc(&d_data, size);

// 2. Inizializzazione della memoria GPU a zero (per pulizia)
cudaMemset(d_data, 0, size);

// 3. Trasferimento dati da Host a Device
cudaMemcpy(d_data, h_data, size, cudaMemcpyHostToDevice);

// ... elaborazione dati su GPU (kernel) ...

// 4. Recupero risultati (da Device a Host)
float h_results[N];
cudaMemcpy(h_results, d_data, size, cudaMemcpyDeviceToHost);

// 5. Liberazione memoria GPU allocata
cudaFree(d_data)
```

2.1.3. Organizzazione dei Thread

CUDA adotta una gerarchia a due livelli per organizzare i **thread** basata su **blocchi** di thread e **griglie** di blocchi.

Il concetto di **thread** é stato fornito nella sezione dedicata.

📖 **Blocco**

Un **blocco** (di thread) è un **gruppo di thread** eseguiti **logicamente in parallelo**. Ogni blocco ha un **id univoco**.
I blocchi sono **organizzati in una struttura 1D, 2D o 3D**. I thread di uno stesso blocco possono **sincronizzarsi** (non automaticamente) e **condividere memoria**.

👉 ***Nota*** : Thread di blocchi diversi possono sincronizzarsi unicamente tramite memoria globale o kernel successivi.

Il numero massimo totale di thread per blocco è **1024** per la maggior parte delle GPU. Pertanto il prodotto delle dimensioni del blocco non può superare questo limite.

📖 **Griglia**

Una **griglia** é un array di **blocchi di thread**, anch'essa con **struttura 1D, 2D o 3D**.
La griglia condivide lo stesso spazio di **memoria globale** e contiene tutti i **thread** che eseguono un **singolo kernel**.

⚠️ **Relazione tra thread, blocchi e griglia**

Thread $\in$ Blocco $\in$ Griglia $\equiv$ kernel

👉 ***Nota***: Posso esserci più kernel in esecuzione.

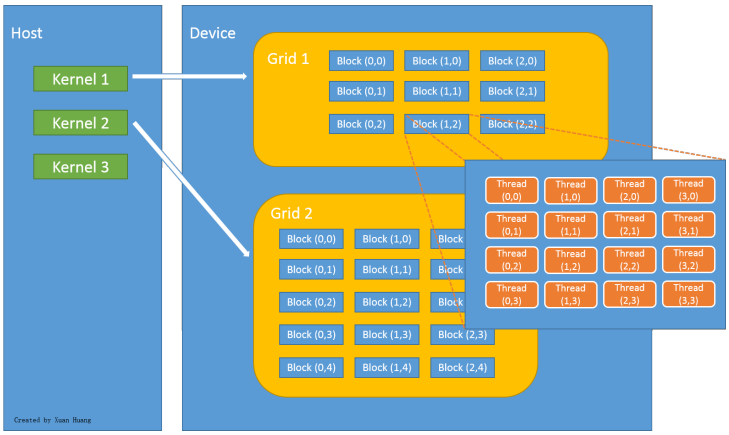


Figura 25: Gerarchia thread-blocchi-griglia-kernel

Avere una gerarchia di thread comporta diversi benefici:

- Possibilità di **scomporre problemi complessi** in unità di lavoro più piccole e gestibili (spesso rispecchiando la struttura intrinseca del problema).
- Possibilità di **adattamento delle dimensioni** di blocchi e griglia per adattare l'esecuzione alle caratteristiche specifiche di hardware e problema, ottimizzando l'uso delle risorse
- Possibilità di condividere memoria on-chip veloce tra thread di un blocco **riducendo gli acccessi alla memoria globale**.
- **Gerarchia scalabile** ⇒ **CUDA portabile** tra architetture GPU NVIDIA.
- **Sincronizzazione granulare** (a blocco o a kernel).

Come accennato in precedenza (nella definizione di Thread), ogni thread ha un **identificatore univoco**, costruito tramite le coordinate del thread all'interno della gerarchia grid-block.

- `blockIdx` = indice del blocco all'interno della griglia, possiede una coordinata per dimensione: `blockIdx.x`, `blockIdx.y`, `blockIdx.z`.
- `threadIdx` = indice del thread all'interno del blocco, possiede una coordinata per dimensione: `threadIdx.x`, `threadIdx.y`, `threadIdx.z`.

👉 ***Nota***: gli Idx sono variabili built-in di tipo `uint3` (a sua volta built-in vettoriale di CUDA con campi `x`, `y`, `z` di tipo `unsigned int`).

Analogamente sono definiti i buil-in per le dimensioni:

- `blockDim` = dimensione del blocco, con campi `x`, `y`, `z`.
- `gridDim` = dimensione della griglia, con campi `x`, `y`, `z`.

Le dimensoni massime utilizzabili dipendono tipicamente dalla compute capability del device.

👉 ***Nota***: Le dimensioni di blocchi e griglia vanno definite nel codice host, prima del lancio del kernel.

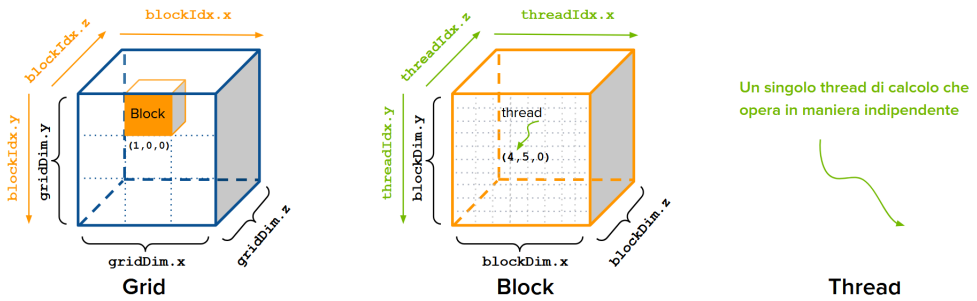


Figura 26: Coordinate thread, identificatori e dimensioni di griglia e blocco.

👉 ***Nota***: Si possono combinare dimensioni di griglia e blocco diverse tra loro (esempio griglia 2D e blocco 3D), anche se tipicamente sono uguali.

2.1.4. Lancio di kernel

`dim3` é una struttura definita in `vector_types.h`, usata per specificare le dimensioni di griglia e blocchi, che supporta le dimensioni 1, 2 e 3.
Questo tipo utilizzato per le dimensioni di griglia e blocchi al lancio di un kernel lato host.

📖 **Kernel CUDA**

Un kernel CUDA è una **funzione** che viene eseguita in parallelo sulla GPU da migliaia o milioni di thread. Il kernel rappresenta il **nucleo computazionele** di un programma CUDA.

```
kernel_name <<<gridSize,blockSize>>>(argument list);
```

📝 **Esempio: Lancio di un kernel**

```
dim3 gridDim(256);                          // Griglia di 256x1x1 blocchi.
dim3 blockDim(512, 512);`                   // Blocco di 512x512x1 threads.
kernel_name<<<gridDim, blockDim>>>(...);    // Lancio kernel
```

I **qualificatori di funzione** in CUDA sono essenziali per specificare dove una funzione verrà eseguita e da dove può essere chiamata.

Qualificatore	Esecuzione	Chiamata	Note
`__global__`	Sul Device	Dall'Host	Deve avere tipo di ritorno `void`
`__device__`	Sul Device	Solo dal Device	
`__host__`	Sull'Host	Solo dall'Host	Può essere omesso

👉 ***Nota***: Le funzioni `__global__` sono di fatto i kernel.

⚠️ **Combinazione qualificatori**

In CUDA, combinando `__host__` e `__device__`, una funzione può essere eseguita sia sulla CPU che sulla GPU, ma il codice é il medesimo.

⚠️ **`__global__` e `__device__`**

- Accesso consentito unicamente alla memoria device (no host).
- Nessun supporto agli argomenti variabili.
- Nessun supporto ai puntatori a funzione.
- Le funzioni `__global__` sono **asincrone**.

2.1.5. Mapping e dimensionamento

⚠️ **Mapping di indici per problema lineare**

Considerando un problema facilmente mappabile su blocchi e griglia monodimensionali, una strategia efficacie di mapping degli indici dei thread per la suddivisione del lavoro é:

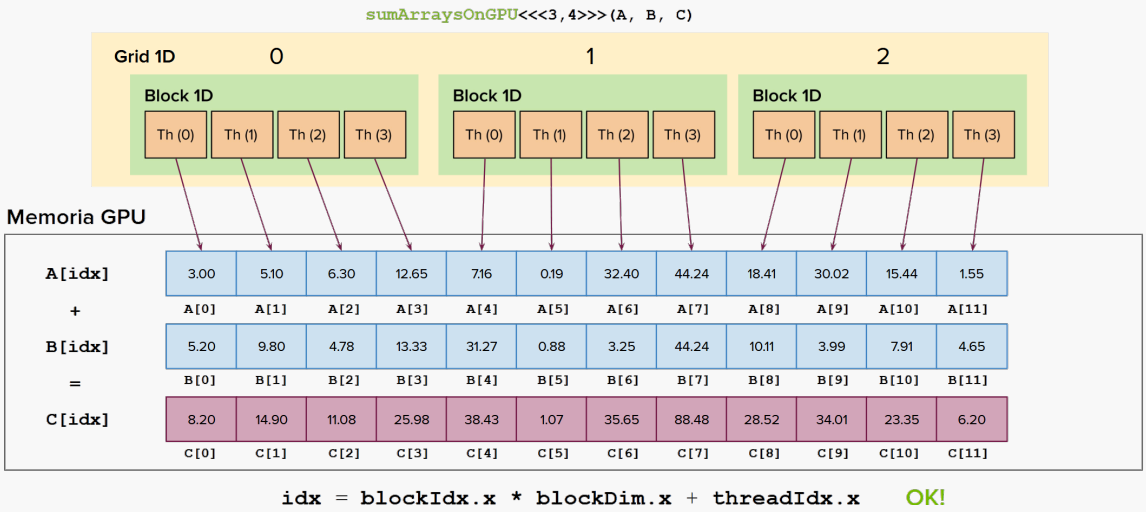
```
idx = blockIdx.x * blockDim.x + threadIdx.x;
```

In questo modo sono garantite le proprietà chiave:

- **Copertura completa** degli elementi su cui elaborare.
- **Mapping corretto** (ogni thread é associato ad un unico elemento).
- Nessuna ripetizione.
- **Parallelismo massimizzato**, infatti un compito specifico per ogni thread disponibile.
- **Bilanciamento del carico** uniforme.
- **Accessi coalescenti**, ossia thread adiacenti in un blocco accedono a elementi di memoria adiacenti, favorendo accessi coalescenti (quindi meno transizioni, ma più elevate) e migliorando l'efficienza della memoria.

⇒ Si usa l'indice globale di un thread (che lo identifica univocamente) per identificare la pozione di dati che quel thread deve elaborare.

Esempio di applicazione mapping di un problema lineare



Calcolo Indice Globale di un Thread - Metodo Coordinate

Caso generale (tridimensionale):

```
ix = threadIdx.x + blockIdx.x * blockDim.x    // indice thread lungo asse x
iy = threadIdx.y + blockIdx.y * blockDim.y    // indice thread lungo asse y
iz = threadIdx.z + blockIdx.z * blockDim.z    // indice thread lungo asse z
nx = gridDim.x * blockDim.x                   // numero thread per riga
ny = gridDim.y * blockDim.y                   // numerothread per colonna
global_idx = iz * (nx * ny) + iy * nx + ix    // indice globale
```

Nota: Nel caso di dimensioni inferiori basta togliere "z" ed eventualmente anche "y"

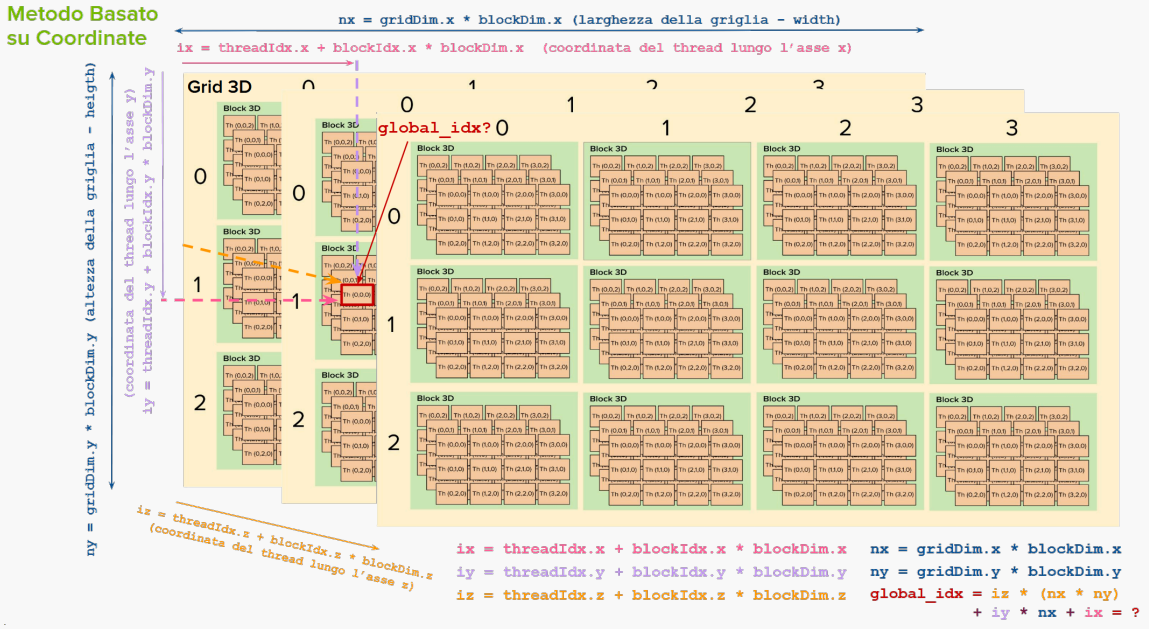
Con questo approccio si definisce **manualmente prima la dimensione del blocco** e poi si calcola automaticamente la dimensione della griglia in base ai dati (e alla dimensione del blocco).

```
int blockSize = 256; int dataSize = 1030;
dim3 blockDim(blockSize);
dim3 gridDim((dataSize + blockSize - 1) / blockSize);
```

🤔 ***Why?*** : La divisione fornisce il numero di blocchi pieni, e l'aggiunta di blockSize-1 include anche i blocchi parziali

👉 ***Nota*** : Questo perché la dimensione del blocco è legata alle **caratteristiche hardware** della GPU e la natura del problema, mentre la dimensione della griglia si adatta alla dimensione del blocco e al volume dei dati da processare.

📝 **Esempio: Mapping con coordinate**



2.1.6. Analisi delle prestazioni

Il **controllo dei kernel** CUDA mira a **confermare l'affidabilità dei calcoli** eseguiti sulla GPU. Tale controllo deve essere sistematico e con una certa tolleranza (dobuta ad esempio ai confronti in virgola mobile).

⚠️ **Macro CHECK**

Molte chiamate CUDA sono asincrone, rendendo difficile associare un errore alla specifica chiamata che lo ha causato.
Si può quindi usare la macro CHECK (da definire), per ottenere file, riga, codice e descrizione degli errori.

```
#define CHECK(call) \
{ \
    const cudaError_t error = call; \
    if (error != cudaSuccess) \
```

```
    { \
        printf("Error: %s:%d, ",__FILE__,__LINE__); \
        printf("code: %d, reason: %s\n",error,cudaGetErrorString(error)); \
        exit(1); \
    } \
}
// si utilizza passando come argomento la funzione da eseguire (e
controllare)
```

2.1.6.1. Profiling

Il profiling permette di **misurare le prestazioni**, **analizzare l'uso delle risorse** e **individuare** possibili **ottimizzazioni**.

Importanza del profiling e della misurazione dei tempi:

- Identificazione dei Colli di Bottiglia
- Analisi degli Effetti delle Modifiche implementative sul tempo di esecuzione.
- Confronto tra Implementazioni.
- Verifica se il carico di lavoro è distribuito in modo efficiente tra thread, blocchi e host-device.

Metodi per la misurazione dei tempi:

- **Timer CPU**, é semplice e diretto, utilizza funzioni di sistema per ottenere il tempo di esecuzione. Misura il tempo wall-clock visto dalla CPU. Il tempo include gli overhead.

```
#include <time.h>
double cpuSecond() {
  struct timespec ts;
  timespec_get(&ts, TIME_UTC);  // tempo corrente del sistema
  return ((double)ts.tv_sec + (double)ts.tv_nsec * 1.e-9);  // tempo in s
}
```

Pro

- **Facile** da implementare e utilizzare.
- Non richiede librerie CUDA **specifiche** per il timing.
- Funziona su **qualsiasi sistema** con supporto CUDA.
- Efficace per **kernel lunghi** e **misure approssimative**.

Contro

- **Impreciso** per kernel molto brevi (< 1 ms).
- Include **overhead** non relativo all'esecuzione del kernel (es., sistema operativo, utilizzo CPU, etc.).
- Non fornisce dettagli sulle **fasi interne** del kernel.
- Precisione influenzata dal **carico dell'host**.

👉 ***Nota***: Per misurare i tempi di esecuzione di 1 (o più) kernel é necessario eseguire la `cudaDeviceSynchronize()` prima di calcolare il tempo, affinchè il l'esecuzione del kernel sia effettivamente conclusa.

📝 Esempio: Misurazione Tempo con Timer CPU

```
double iStart = cpuSecond();                    // tempo di inizio
kernel_name <<<grid, block >>>(argument list);  // Lancip kernel
cudaDeviceSynchronize();  // Attesa completamento kernel
double iElaps = cpuSecond() - iStart;           // Calcolo tempo
```

- **NVIDIA Profiler** (solo per $5.0 \leq CC < 8.0$) o `nvprof` strumento da **riga di comando** per raccogliere informazioni sull'**attività di CPU e GPU** dell'applicazione, inclusi kernel, trasferimenti di memoria e chiamate all'API CUDA.

```
$ nvprof [nvprof_args] <application> [application_args]
```

- **NVIDIA Nsight Systems**, strumento avanzato di **profilazione** e **analisi** delle **prestazioni** a livello di sistema.
  Offre una visione globale dell'applicazione, inclusi CPU, GPU e interazioni con il sistema.
  Permette di identificare colli di bottiglia, analizzare overhead dele chiamate API, esaminare le operazioni di input/output, ottimizzare il flusso di lavoro dell'applicazione.
  Caratteristiche chiave:
  - Visualizzazione grafica delle timeline di esecuzione.
  - Analisi delle chiamate API CUDA e sincronizzazioni.
  - Monitoraggio dell'utilizzo di memoria e cache.
  - Supporto per sistemi multi-thread e multi-GPU.

```
$ nsys profile --stats=true ./application
```

- **NVIDIA Nsight Compute**, strumento di **profilazione** e **analisi** approfondita per singoli kernel CUDA. Fornisce metriche dettagliate e mirate alle prestazioni a livello di **kernel** .Permette di:
  - **Analisi** dettagliata delle metriche hardware per ogni **kernel**.
  - Visualizzazione grafica dell'utilizzo della memoria
  - Confronto side-by-side di diverse esecuzioni dei kernel.
  - Offre **report** dettagliati e suggerimenti automatici per ottimizzare il codice a livello di kernel.

Memoria - Best Practice

- **Minimizzare i trasgerimenti di memoria**
  Why? : I trasferimenti di dati tra host e device hanno un'alta latenza
  ⇒ Raggruppamento di dati in buffer più grandi per **ridurre i trasferimenti** e **sfruttare la banda**.
- **Allocazione e Deallocazione Efficiente**
  Why? : L'**allocazione** di memoria GPU tramite `cudaMalloc` é un'operazione relativamente **lenta**.
  ⇒ Allocare memoria una volta all'inizio e riutilizzarla quando possibile.
  Liberare la **memoria** (`cudaFree`) quando non più necessaria, per evitare leak e sprechi di risorse.
- **Sfruttare la Shared Memory**
  Why? : La shared memory è una memoria on-chip a **bassa latenza** accessibile a tutti i **thread di un blocco**.
  ⇒ Mettere in Shared i dati acceduti di frequente e condivisi tra i thread di un blocco per ridurre l'accesso alla memoria globale più lenta.

2.2. Modello di Esecuzione CUDA

2.2.1. Architettura Hardware GPU

In generale, un **modello di esecuzione** fornisce una **visione operativa** di come le istruzioni vengono eseguite su una specifica architettura di calcolo (nel nostro caso le GPU).

Caratteristiche:
- **Astrazione portabile** dell'architettura (Grid, Block, Thread, Warp, SM).
- Preserva concetti fondamentali tra generazioni differenti di GPU.
- Espone le **funzionalità architetturali** chiave per la programmazione CUDA.
- Descrive come kernel, griglie e blocchi vengono effettivamente **mappati** sull'hardware GPU.
- É basato sul parallelismo massivo e sul modello SIMT (Single Instruction, Multiple Thread).

É importante perché:
- Offre una **visione unificata** dell'esecuzione su diverse GPU.
- Fornisce indicazioni utili per l'ottimizzazione del codice in termini di throughput istruzioni e accessi di memoria.
- Facilita la comprensione della relazione tra il modello di programmazione e l'esecuzione effettiva.
- Permette di interpretare correttamente i risultati dei profiler CUDA, collegando i fenomeni osservati (latenze, occupancy, conflitti di memoria) alla struttura del modello di esecuzione.

📖 **Streaming Multiprocessor (SM)**

Gli **Streaming Multiprocessors** (SM) sono le **unità fondamentali di elaborazione** all'interno delle GPU.
Ogni SM contiene diverse **unità di calcolo**, **memoria condivisa** e altre **risorse essenziali** per gestire l'esecuzione concorrente e parallela di migliaia di thread.
Il parallelismo hardware delle GPU è ottenuto attraverso la **replica** di questo blocco architetturale.
Un SM é composto da:
- **CUDA Cores**: unità di elaborazione che eseguono **istruzioni aritmetico/logiche**.
- **Shared Memory/Cache L1** : memoria ad alta velocità condivisa tra i thread di un blocco.
- **Register Files** : **memoria privata** di ogni thread per dati temporanei.
- **Load/Store Units** (LD/ST): gestiscono il trasferimento dati da/verso la memoria.
- **Special Function Units** (SFU): accelerano calcoli matematici complessi (funzioni trascendenti).
- **Warp Scheduler** : seleziona thread pronti per l'esecuzione nell'SM.
- **Dispatch Unit**: assegna i thread selezionati alle unità di esecuzione.
- **Instruction Cache**: memorizza temporaneamente le istruzioni usate di frequente.

Figura 27: Architettura Streaming Multiprocessor e replica SM in GPU.

⚠️ Recap Memorie GPU

Velocità e dimensioni delle memoria vanno di pari passo. Infatti, le memorie più rapide e vicine ai Core sono anche le più piccole e "locali", allontanandosi, via via aumentano le dimensioni e diminuiscono velocità e "località".
In *Figura 28* é mostrato un recap visivo.

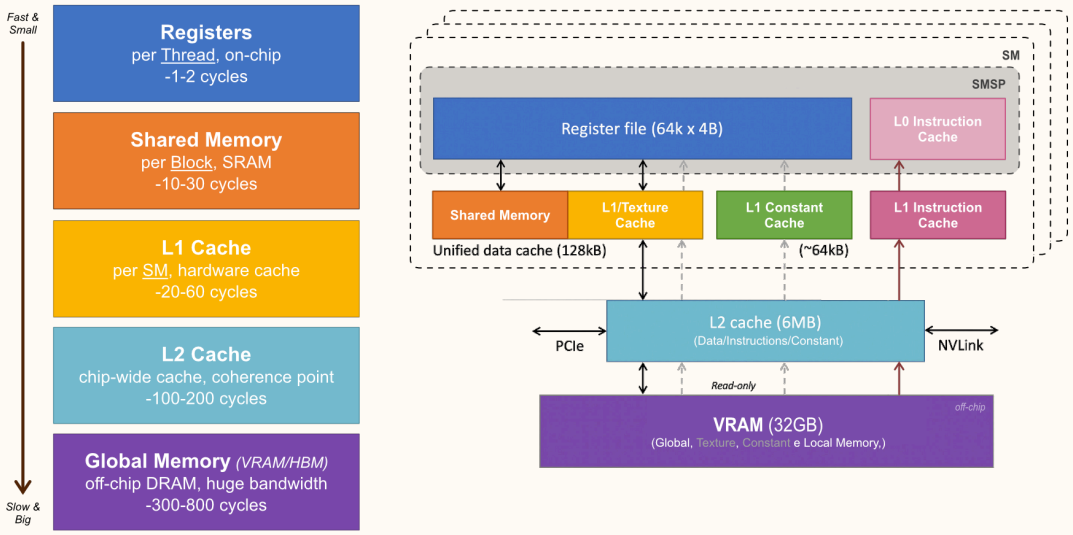


Figura 28: Gerarchia e Funzionamento Memorie GPU.

👉 ***Nota*** : Nelle GPU NVIDIA Shared Memory e Cache L1 sono spesso unificate insieme (anche se logicamente separate), mentre in altre architetture (come AMD RDNA) sono separate.
👉 ***Nota*** : Ogni SM ha una propria (e privata) **L1 Cache**, **Shared Memory** e **Constant Cache**, mentre la **L2 Cache** é **condivisa** tra tutti (o più) SM. La **Memoria Globale** (VRAM) é in ogni caso **unica** (GPU dedicata).
👉 ***Nota*** : I dati dalla memoria host che passano dal bus PCIe diretti verso la Global Memory non

passano fisicamente per la cache L2 (per questioni di overhead e di rapida saturazione), tuttavia, a livello logico, il controller della **L2 monitora il passaggio** di questi dati (punto di coerenza) per fornire dati aggiornati in caso un SM richieda informazioni situate agli indirizzi destinatari dei dati provenienti dalla memoria host.

Ulteriori approfondimenti verranno fatti in seguito (v. Sezione 2.3.2).

CUDA Core

Un **CUDA Core** è l' **unità di elaborazione di base** all'interno di un SM di una GPU NVIDIA. L'architettura e la funzionalità dei CUDA Core sono evolute nel tempo, passando da unità generiche a **unità specializzate**.

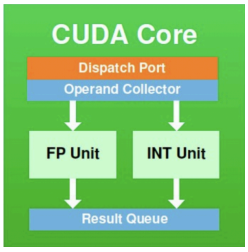


Figura 29: CUDA Core

[-Fermi]

Nelle prime architetture (fino a Fermi) erano erano unità di calcolo relativamente semplici, in grado di eseguire sia operazioni intere (INT) che in virgola mobile (FP) in un ciclo di clock (**fully pipelined**, non simultaneamente):

- **ALU** per operazioni matematiche di base (*addizioni, sottrazioni, moltiplicazioni e operazioni logiche*).
- **FPU** per operazioni in virgola mobile, supportando principalmente calcoli a precisione singola (FP32).

I CUDA Core usano **registri condivisi** a livello di Streaming Multiprocessor per memorizzare temporaneamente dati durante l'esecuzione dei thread.

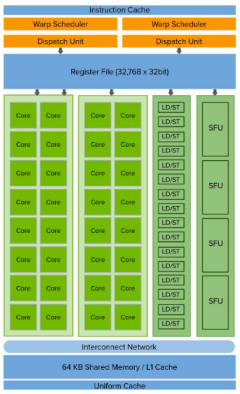
[Kepler+]

Ora i core sono specilizzati all'interno di un SM.

- Generali:
  - **Unità FP64**, per operazioni in virgola mobile a doppia precisione.
  - **Unità FP32**, per operazioni in virgola mobile a singola precisione.
  - **Unità INT**, per operazioni intere.
- AI:
  - **Tensor Core** (TC) [Ampere+], specilizzate per ottimizzare moltiplicazioni fra matrici in precisione ridotta/mistra (FP32, FP16, TF32, INT8, ...).
- Grafica:
  - **Ray Tracing Core** (RT) [Ampere+], per l'accelerazione del ray tracing.
  - **Unità di Texture**, ottimizzate per gestire texture e operazioni di filtraggio.
  - **Unità di Rasterizzazione**, per la rasterizzazione delle immagini durante il rendering.

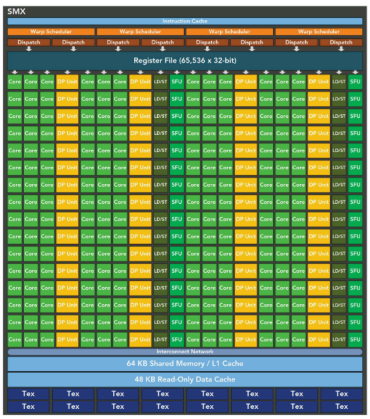
👉 ***Nota*** : Le varie unità di gestione all'interno di un SM sono più semplici rispetto ai core delle CPU, senza unità di controllo complesse, permettendo una maggiore densità di unità specializzate.

Fermi GPU (2010)



- Architettura **unificata** per grafica e calcolo.
- **Concurrent Kernel Execution**: Esecuzione simultanea di più kernel.
- Supporto **FMA** e calcolo scientifico avanzato (FP64, ECC)

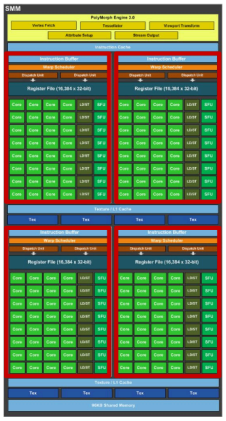
Kepler GK100X SMX (2012)



- Nuovo design **SMX** con 192 CUDA Cores → più parallelismo per SM.
- **Dynamic Parallelism**: I kernel GPU possono lanciare altri kernel senza intervento della CPU.
- **Hyper-Q**: Fino a 32 connessioni simultanee CPU-GPU per ridurre le latenze.

Maxwell SM (2014)

Streaming Multiprocessor Sub-Partition (SMSP)



- Maggiore **efficienza energetica** rispetto a Kepler.
- **Delta Color Compression** per ridurre la larghezza di banda.
- Divisione degli SM in 4 per ottimizzare l'uso delle risorse.

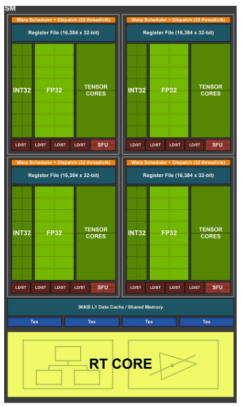
Pascal SM (2016)

- Introduzione di **NVLink** per connessioni ad alta velocità.
- Supporto **HMB2** (High Bandwidth Memory).
- **Miglioramento** della tecnologia **GPU Boost 3.0** per l'overclocking dinamico.
- Supporto nativo per **FP16** (half precision)
- **16nm FinFET**: Processo produttivo con 2x efficienza energetica rispetto a Maxwell (28nm).

Volta SM (2017)

- Introduzione dei **Tensor Core** per accelerare l'AI.
- **Independent Thread Scheduling** per un'esecuzione più flessibile dei thread.
- **Supporto FP32/INT32** simultaneo e **NVLink 2.0**

Turing (2018)

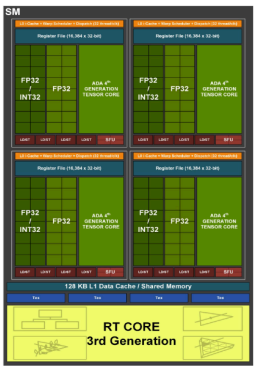


- Introduzione dei **RT Cores**: Ray-tracing in tempo reale.
- **DLSS** (Deep Learning Super Sampling)
- **Variable Rate Shading**

Ampere (2020)

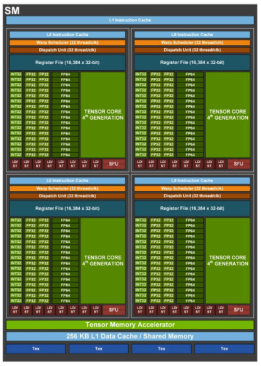
- Aumento del throughput FP32 per i CUDA Cores.
- Miglioramenti per **Tensor Cores** e **RT Cores**.
- **GPU Scheduler** ottimizzato per la gestione del carico di lavoro.

Ada Lovelace (2022)



- Incremento delle prestazioni FP32 con **più CUDA Cores**.
- Introduzione di **DLSS 3.0** con frame generation.
- **RT Cores migliorati** per ray tracing avanzato.

Hopper (2022)



- **Tensor Cores di quarta generazione** per AI.
- **Multi-Instance GPU (MIG).**
- **Thread Block Clusters**
- **FP8 Precision** per migliorare il throughput.

Blackwell (2025)

- **Tensor Core di quinta generazione** per AI accelerata
- **Seconda generazione** di **Transformer Engine**
- **RT Cores di 4° generazione**
- Supporto per **precision FP4 e MXFP6/MXFP4**

Ogni generazione ha generalmente aumentato il numero di SM e CUDA Core. L'aumento delle unità di elaborazione permette un **maggiore parallelismo**, migliorando le prestazioni complessive della GPU.

$$\text{Totale CUDA Core} = (\text{SM per GPU}) \times (\text{CUDA Core per SM})$$
$$\text{Throughput (teorico) FP32} = \text{Totale CUDA Cores FP32} \times 2^{*} \times \text{Frequenza di clock.}$$

**2 operazioni FP32/ciclo, e.g. Fused Multiply-Add*

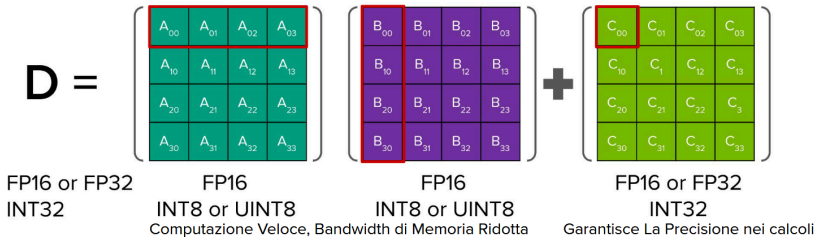
📖 Tensor Core *[Volta+]*

Un **Tensor Core** é un' unità di elaborazione specializzata per **operazioni tensoriali** (array multidimensionali).

- Esegue operazioni **matrice-matrice** in precisione mista.
- Supporta formati **FP8**, **FP16**, **FP32**, **FP64**, **INT8**, **INT4**, **BF16** e nuovi formati come **TF32** (TensorFloat-32).
- Offre un significativo **speedup** nel calcolo senza compromettere l'accuratezza.

I Tensor Core, di fatto, sono acceleratori per i calcoli di IA.

- **Fused Multiply-Add (FMA):** Un'operazione che combina una moltiplicazione e un'addizione di <u>scalari</u> in un unico passo, eseguendo $d = a \times b + c$. Un CUDA core esegue 1 FMA per ciclo di clock in FP32.
- **Matrix Multiply-Accumulate (MMA):** Operazione che calcola il prodotto di due <u>matrici</u> e somma il risultato a una terza matrice, eseguendo $D = A \times B + C$
- Per matrici $A(m \times k)$, $B(k \times n)$ e $C(m \times n)$, l'operazione produce $D(m \times n)$ e richiede $m \times n \times k$ operazioni FMA, dove ogni elemento di D necessita di k moltiplicazioni-addizioni.



Ogni Tensore Core esegue 64 FMA in un singolo ciclo di clock grazie al parallelismo interno.

👉 ***Nota***: Originariamente se le matrici non erano 4×4, allora venivano decomposte in matrici 4×4,

ma con le generazioni più recenti di GPU i TC supportando dimensioni di matrici più grandi e/o sparse con un maggiore numero di formati numerici.
👉 ***Nota*** : L'impatto delle approssimazioni é minimo sull'accuratezza finale dei modelli di deep learning, beneficiando però di minor consumo di memoria ed energia, oltre all'accelerazione dei TC.

2.2.2. Organizzazione e Gestione dei Thread

Quando un kernel viene lanciato i blocchi vengono **automaticamente e dinamicamente distribuiti** dal **GigaThread Engine** (scheduler globale) agli SM disponibili.

👉 ***Nota*** : le variabili di identificazione e dimensione (gridDim, blockIdx, blockDim, e threadIdx) sono rese disponibili ad ogni thread e condivise nello stesso SM.

Una volta assegnato a un SM, un blocco rimane vincolato a quell'SM **per tutta la durata dell'esecuzione**.

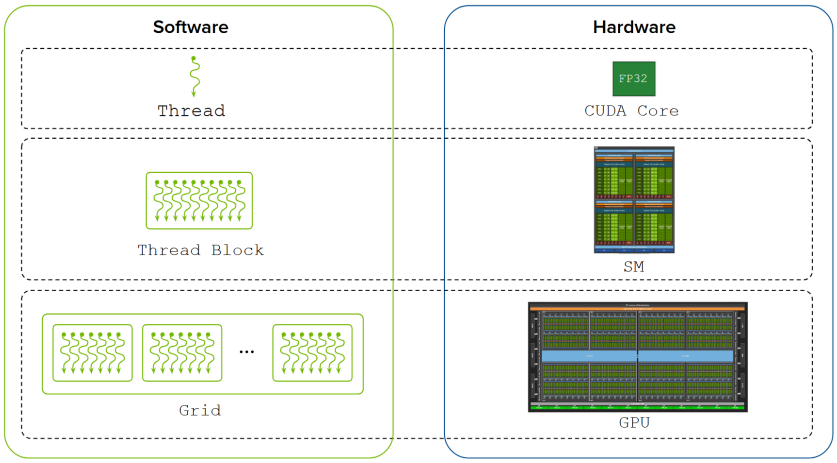


Figura 30: Corrisposndenza tra vista Logica e vista Hardware

👉 ***Nota*** : Ad un blocco é associato un unico SM, ma un SM può contenere più blocchi, fino ad un **limite massimo** imposto dalla sua CC.

👉 ***Nota*** : CUDA non garantisce l'ordine di esecuzione dei blocchi/thread , ma il Gigathread Engine smista i blocchi agli SM in base alle risorse disponibili. Infatti ogni blocco viene **elaborato in modo indipendente**.

📖 **Wave e Full Wave Capacity**

Una **Wave** rappresenta l'**insieme dei blocchi** di thread che vengono **eseguiti simultaneamente** su tutti gli SM della GPU in un dato momento.

$$\text{Numero di waves} = \lceil (\text{Blocchi totali}) \ / \ (\text{Full wave capacity}) \rceil$$

La **Full Wave Capacity**, invece, rappresenta la **capacità teorica massima** della GPU , ossia il **numero totale di blocchi** che possono essere **residenti simultaneamente** su tutti gli SM.

$$\text{Full Wave Capacity} = \text{Numero di SM} \cdot \text{Numero massimo di blocchi attivi per SM}$$

Si definisce **Partial Wave** se solo parte degli SM è occupata, oppure non tutti al massimo.

👉 ***Nota*** : Full Wave Capacity e numero di waves dipendono dall'architettura della GPU (CC) e dalle risorse richieste per ogni blocco.

Per **scalabilità** in CUDA ci si riferisce alla **capacità** di un'applicazione di **migliorare le prestazioni** proporzionalmente all'**aumento delle risorse hardware** disponibili.

Più SM disponibili ⇒ Più blocchi eseguiti contemporaneamente ⇒ Maggiore Parallelismo.

👉 ***Nota*** : Questo senza apportare nessuna modifica al codice.

2.2.3. Modello di Esecuzione SIMT e Warp

📖 **SIMD**

SIMD (Single Instruction, Multiple Data) é un **modello di esecuzione parallela** in cui una **singola istruzione** opera simultaneamente su **più elementi di dato**, utilizzando **unità vettoriali** dedicate (vector units) presenti nei core della CPU.
Per fare ciò utilizza registri vettoriali (ed hardware apposito) che possono contenere più elementi.

Limitazioni:
- Larghezza vettoriale **fissa** (nell'hardware).
- Gli elementi vettoriali in un vettore sono elaborati in **lockstep** (perfettamente sincroni).
- **Divergenza non ammessa** (niente percorsi condizionali), si usano invece maschere vettoriali.

📖 **SIMT**

SIMT (Single Instruction, Multiple Thread) é un **modello ibrido** adottato in CUDA che combina **parallelismo multi-thread** con **esecuzione SIMD-like**. Contrariamente a SIMD però«:
- non ha controllo centralizzato delle istruzioni;
- ogni thread possiede un **proprio Program Counter** (PC), **registri** e **stato indipendenti**;
- supporta la divergenza del flusso di controllo;
- l'hardware **gestisce automaticamente la divergenza**.

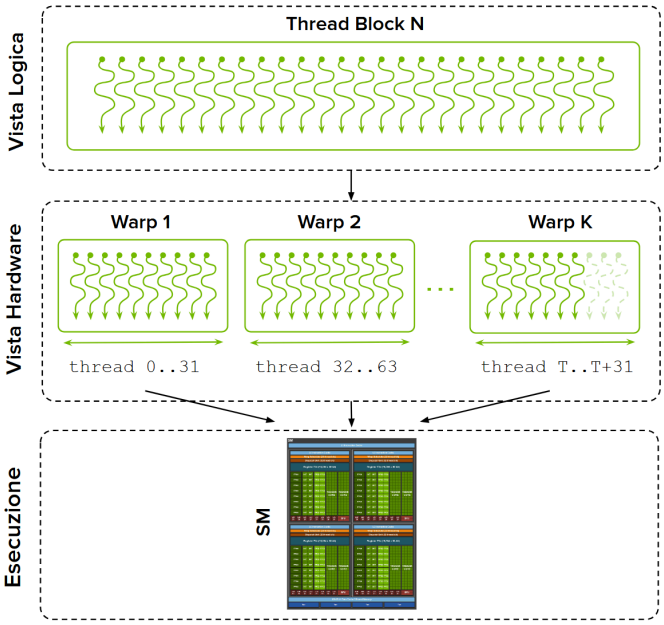
	SIMD	SIMT
Unità di Esecuzione	Un **singolo thread** controlla vettori di dimensione fissa	**Molti thread** leggeri raggruppati in warp (32 thread)
Registri	**Registri vettoriali condivisi** tra le unità di calcolo	Set completo di **registri per thread**
Flessibilità	**Bassa**: Stessa operazione per tutti gli elementi vettore	**Alta**: Ogni thread può eseguire operazioni e percorsi indipendenti
Indipendenza	**Non applicabile**, controllo centralizzato	Ogni thread mantiene il **proprio stato di esecuzione** (vedi nota*)
Branching	Gestito **esplicitamente** con **maschere** (no divergenza)	Gestito **via hardware** con **thread masking automatico**
Scalabilità	**Limitata** dalla larghezza vettoriale	**Massiva** (migliaia/milioni di thread)
Sincronizzazione	**Intrinseca** (lock-step automatico)	**Esplicita** (es., `__syncthreads`)
Utilizzo Tipico	**Estensioni CPU** (SSE, AVX, NEON)	**GPU** Computing (CUDA, OpenCL)

In CUDA i thread sono organizzati in **gruppi di 32** chiamati **warp**, che sono l'**unità di esecuzione minima di un SM**.
I thread di un warp iniziano insieme dal medesimo indirizzo del programma e possono divergere.

La **divergenza** in un warp causa l'**esecuzione seriale** dei percorsi diversi, con impatto negativo sulle prestazioni.

👉 ***Nota***: Da Volta é stato introdotto l'**indipendent thread scheduling**, che permette di assegnare ad ogni thread un PC (invece che uno per warp).

🤔 ***Why?***: *32 thread per efficienza hardware (massimizza l'utilizzo di risorse di SM, perché warp troppo piccoli sarebbero inefficienti, mentre troppo grandi complicherebbero lo schedulng e sovraccaricherebbero SM e memoria) ed efficienza della memoria (accesso coalesced in memoria con minor numero di transazioni ed evitare accessi parziali). Inoltre per flessibilità del software (granularità gestibile per i controllo della divergenza e il bilanciamento del carico) ed adattabilità/efficacia per le diverse generazioni di GPU NVIDIA.*



Quindi:
1. Lancio griglia di blocchi di thread
   ⇒ Thread distruibuiti tra gli SM disponibili.
2. Thread di un blocco divisi in **warp** (32 thread) con id consecutivi.
3. Thread di un warp eseguono la **stessa istruzione** su **dati diversi**, con possibilità di **divergenza**
   ⇒ Thread eseguiti **logicamente** in **parallelo** (ma non sempre fisicamente)
4. SM gestisce **dinamicamente** l'esecuzione dei warp.

👉 ***Nota***: Indiepndente dall'organizzazione logica (1D, 2D o 3D) di un blocco di thread, a livello hardware il layout é monodimensionale (1D). É il runtime CUDA che si occupa di linearizzare gli indici multidimensionali in ordine row-major, raggruppare i thread in warp e gestire il mapping hardware.
👉 ***Nota***: L'hardware alloca sempre un numero **discreto** di warp, calcolabile come

$$\#\text{warp} = \text{ceil}(\text{ThreadsPerBlock}/\text{warpSize})$$

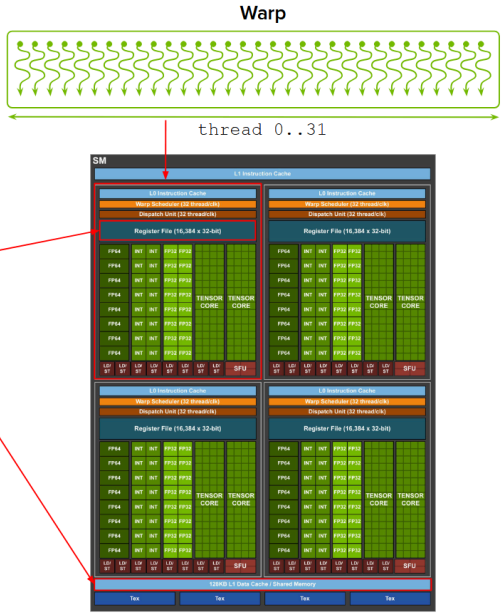
⇒ Best practice: scegliere la dimensione del blocco **multipla di 32** (per evitare warp parziali).

📖 SM Sub-Partition

Ogni SM viene suddiviso in (4) **sub-partition** **indipendenti**. Ogni sub-partition possiede le **proprie risorse** dedicate per l'esecuzione (Warp Scheduler, Dispatch Unit, registri, CUDA Cores) e gestisce un **pool di warp** concorrenti di **dimensione fissa**.

> Un warp viene quindi assegnato ad una sub-partition (non semplicemente all'SM) e ad essa rimane associato fino al suo completamento.

- Il contesto di **esecuzione locale** di un warp in un SM contiene:
  - Volta- : Per warp
    Volta+ : Per thread
    - **Program Counter (PC)**: Indica l'indirizzo della prossima istruzione da eseguire.
    - **Call Stack**: Struttura dati che memorizza le informazioni sulle chiamate di funzione, inclusi gli indirizzi di ritorno, gli argomenti, array e strutture dati più grandi.
  - **Registri**: Memoria veloce e privata per ogni thread, utilizzata per memorizzare variabili e dati temporanei.
  - **Memoria Condivisa**: Memoria veloce e condivisa tra i thread di un blocco utile per comunicare.
  - **Thread Mask**: Indica quali thread del warp sono attivi o inattivi durante l'esecuzione di un'istruzione.
  - **Stato di Esecuzione**: Informazioni sullo stato corrente del warp (es. in esecuzione/in stallo/eleggibile).
  - **Warp ID**: Identificatore che consente di distinguere i warp e calcolare l'offset nel register file per ogni thread nel warp.
- L'SM mantiene **on-chip** il contesto di ogni warp per tutta la sua durata, quindi il **cambio di contesto è senza costo**.



Un blocco di thread viene considerato **attivo** (o **residente**) quando gli vengono allocate risorse di calcolo di un SM (registri, memoria condivisa, ...). Tutti i **warp** di un blocco attivo sono a loro volta definiti **attivi** (anche se non tutti in esecuzione simultaneamente).

👉 ***Nota***: Il numero di blocchi/warp attivi in ciascun istante è limitato dalle risorse dell'SM (CC).

Un **warp** attivo può essere:

- Warp **selezionato**, se **in esecuzione attiva** su un'unità di elaborazione (usa CUDA Cores).
- Warp **in stallo**, se **in attesa di dati o risorse**, impossibilitato a proseguire l'esecuzione (latenza di memoria, dipendenze da istruzioni, sincronizzazioni, ...).
- Warp **eleggibile**/**candidato**, se pronto (ma ancora non scelto) per l'esecuzione, con tutte le risorse necessarie disponibili. Condizioni per l'eleggibilità sono:
  - **disponibilità di risorse** allocabili (unità di esecuzione disponibili),
  - **prontezza dei dati**,
  - **nessuna dipendenza bloccante**.

Un warp contiene 32 thread ma non tutti potrebbero essere logicamente attivi. Lo stato di ogni thread é tracciato mediante una **thread mask** o **maschera di attività**, di fatto un registro hardware a 32 bit. Un thread é:

- **Attivo**, se esegue l'istruzione corrente del warp (quindi contribuisce all'esecuzione SIMT attivamente).
- **Inattivo**, a causa di:
  - **divergenza**, ossia ha seguito un percorso diverso nel warp per istruzioni di controllo di flusso.
  - **terminazione**, ha completato l'esecuzione prima di altri thread (nel warp).
  - **padding**, thread usato unicamente per rendere il numero di thread del warp multiplo di 32.

👉 ***Nota***: Gli stati sono gestiti **automaticamente** dall'hardware, tramite maschere di esecuzione e la transizione tra stati é dinamica (varia durante l'esecuzione).

Warp Scheduler

Il **Warp Scheduler** é un'**unità hardware** presente in più copie all'interno di ogni SM, responsabile della **selezione** e **assegnazione** dei warp alle unità di calcolo CUDA (gestendo ordine e priorità).

L'obiettivo é massimizzare l'utilizzo delle risorse degli SM selezionando in modo efficiente i warp.
I warp scheduler selezionano i **warp eleggibili** ad ogni ciclo di clock e li inviano alle **dispatch unit**, responsabili dell'assegnazione effettiva alle unità di esecuzione. Se un warp é in stallo, si occupano di selezionare altri warp eleggibili per l'esecuzione, favorendo il **latency hiding**, in quanto nascondono le latenze eseguendo warp alternativi quando altri sono in stallo.
Il **cambio di contesto** tra warp è estremamente **rapido** (on-chip per tutta la durata del warp) grazie alla partizione delle risorse di calcolo e alla struttura hardware della GPU

👉 ***Nota***: Il numero di warp attivi su un SM è limitato dalle risorse di calcolo, mentre il numero di warp selezionati ad ogni ciclo è limitato dal numero di warp scheduler (e dal numero di warp eleggibili).

Dispatch Unit

La **Dispatch Unit** é un'**unità hardware** presente in più copie all'interno di ogni SM, responsabile di come eseguire i warp selezionati dallo scheduler. In particolare **decodifica le istruzioni** dei warp, **distribuisce i thread** alle unità di calcolo appropriate, **recupera i dati** necessari per l'esecuzione da registri e memoria e **assegna fisicamente le risorse** hardware (registri, Core, ...) ai thread.

Thread-Level Parallelism (TLP)

Il **Thread-Level Parallelism** (TLP), prevede l'**esecuzione simultanea** di più **warp** per sfruttare il parallelismo tra thread.

Quando un warp è in attesa, un altro warp viene selezionato ed eseguito, aumentando l'occupazione delle unità di calcolo.

Instruction-Level Parallelism (ILP)

L'**Instruction-Level Parallelism** (ILP), prevede l'esecuzione simultanea di **istruzioni indipendenti** all'interno dello stesso warp.

Se ci sono più istruzioni pronte da eseguire in un warp, il warp scheduler può emettere queste istruzioni in parallelo su più unità di esecuzione, massimizzando l'utilizzo delle risorse (pipelining).

👉 ***Nota***: TLP e ILP sono importanti per massimizzaree le risorse occupate (riducendo i tempi morti) e fare latency hiding delle operazioni in memoria e di calclolo.

Il **loop unrolling** é un'ottimizzazione dell'**ILP**. questa prevede la replica del corpo di un ciclo più volte nel codice, riducendo il numero di iterazioni e l'overhead delle istruzioni di controllo.

Esempio: loop unrolling

Ciclo Tradizionale

```
// 100 verifiche della condizione
for (int i = 0; i < 100; i ++) {
   a[i] = b[i] + c[i];}
```

Ciclo Unrolled (fattore 2)

```
// Solo 50 verifiche
for (int i = 0; i < 100; i += 2) {
   a[i] = b[i] + c[i];
   a[i + 1] = b[i + 1] + c[i + 1];}
```

In CUDA si può forzare il loop unrolling (o semplicemente suggerirlo):

Forzato:
`#pragma unroll <N>`

```
#pragma unroll 4
for (int i =  0; i < N; i++) {
  sum += a[i];
}
```

Replica 4 volte del corpo del ciclo.

Automatico:
`#pragma unroll`

```
#pragma unroll
for (int i =  0; i < N; i++) {
  sum += a[i];
}
```

Il compilatore decide il fattore di unrolling più opportuno.

Mean Latency

La **latenza media** è la **media delle latenze** degli elementi individuali.
La latenza di un singolo elemento è la differenza tra il suo tempo di inizio e il suo tempo di fine.

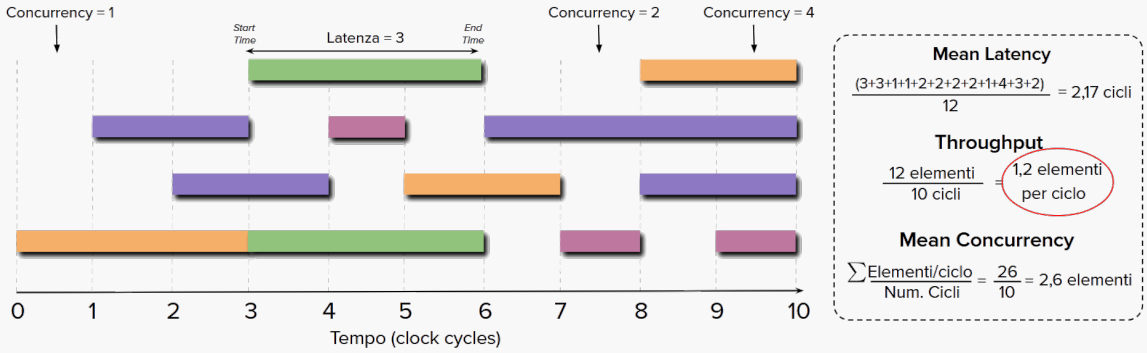
Throughput

Il **throughput** rappresenta la **velocità di elaborazione**. È definito come il numero di elementicompletati entro un dato intervallo di tempo diviso per la durata dell'intervallo stesso.

Concurrency

La **concurrency** misura quanti **elementi** vengono **processati contemporaneamente** in un determinato momento.
Si può definire sia istantaneamente che come media su un intervallo di tempo

Esempio di Mean Latency, Throughput e Concurrency



📖 Latency Hiding

Il **latency hiding** é una tecnica che permette di mascherare i tempi di attesa dovuti ad operazioni ad alta latenza (come gli accessi alla memoria globale) attraverso l'esecuzione concorrente di più warp all'interno di un SM. Si ottiene intercambiando la computazione tra warp, per massimizzare l'uso delle unità di calcolo di ogni SM.

Quando un warp è in stallo, l'SM passa immediatamente all'esecuzione di altri warp pronti. I Warp Scheduler dell'SM selezionano costantemente (ad ogni ciclo di clock) i warp pronti all'esecuzione (richiedono sempre warp eleggibili).

Il latency hiding permette di migliorare utilizzo di risorse, throughput e ridurre la latenza effettiva.

Tipi di Latenza (variano a seconda dell'architettura e dalla tipologia di operazione)

- **Latenza Aritmetica**: Tempo di completamento di operazioni matematiche (bassa, es. 4-20 cicli).
- **Latenza di Memoria**: Tempo di accesso ai dati in memoria (alta, es. 400-800 cicli per la memoria globale).

La Legge di Little (dalla teoria delle code) aiuta a calcolare quanti warp (approssimativamente) devono essere in esecuzione concorrente per ottimizzare il latency hiding e mantenere le unità di elaborazione della GPU occupate.

📖 Legge di Little

$$\textbf{Warp Richiesti} = \textbf{Latenza} \times \textbf{Throughput}$$

con:

- **Latenza** = Tempo di completamento di un'istruzione (in cicli di clock).
- **Throughput** = Numero di warp (e, quindi, di operazioni) eseguiti per ciclo di clock.
- **Warp Richiesti** = Numero di warp attivi necessari per nascondere la latenza.

👉 ***Nota*** : Si tratta di una stima, infatti latenza e throughput possono variare a seconda dell'architettura della GPU e del tipo di istruzioni. Il numero effettivo potrebbe essere leggermente diverso.

Esistono due modi principali per aumentare il parallelismo:

- **ILP (Instruction-Level Parallelism)**: Aumentare il numero di istruzioni indipendenti all'interno di un singolo thread.
- **TLP (Thread-Level Parallelism)**: Aumentare il numero di thread (e quindi di warp) che possono essere eseguiti contemporaneamente.

Esistono due modi principali per aumentare il parallelismo di memoria:

- **Maggiore Granularità:** Spostare più dati per thread (ad esempio, caricare più float per thread).
- **Più Thread Attivi:** Aumentare il numero di thread concorrenti per aumentare il numero di warp attivi.

📖 Warp Divergence

Si verifica **Warp Divergence** se un' **istruzione condizionale** porta thread diversi a percorrere **rami diversi** del codice.
In questo caso, il warp esegue serialmente ogni ramo, utilizzando una maschera di attività (calcolata automaticamente in hardware) per abilitare/disabilitare i thread.
La divergenza **termina** quando i thread **riconvergono** alla fine del costrutto condizionale.

👉 ***Nota*** : Il fenomeno della divergenza occorre solo all'interno di un warp (warp diversi dello stesso blocco non sono affetti).

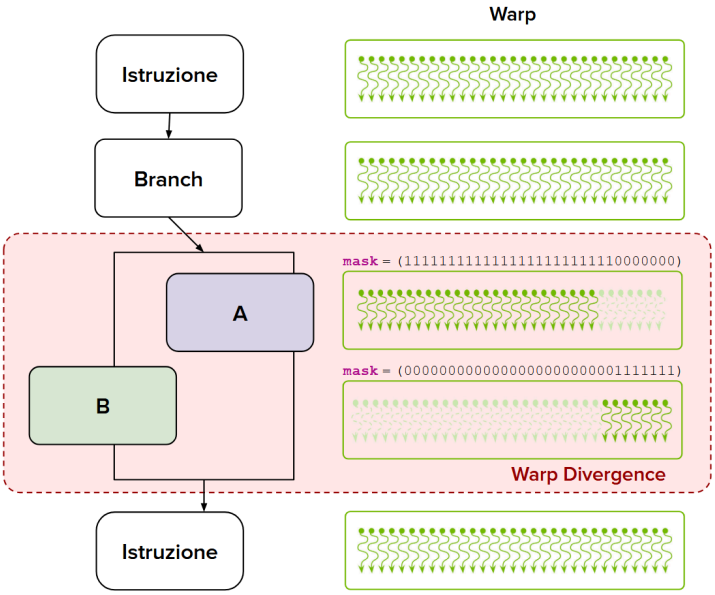


Figura 32: Divergenza warp - A e B sono eseguiti sequenzialmente (prima A, poi B, o viceversa).

La Warp Divergence può significativamente degradare le prestazioni perché i thread non vengono eseguiti in parallelo durante la divergenza (le risorse non vengono pienamente utilizzate), invece sono eseguiti i rami in maniera sequenziale.

	CPU	GPU
Esecuzione	Singoli thread o piccoli gruppi, **indipendenti** tra loro.	Warp che eseguono le **stesse istruzioni** concorrentemente.
Branch Prediction	**Hardware dedicato**, con algoritmi di predizione complessi.	**Non supportata**
Esecuzione Speculativa	Esegue istruzioni **in anticipo** basandosi sulla **branch prediction**.	**Non supportata**
Impatto della Divergenza	Mitigato dalla **branch prediction** e dall'**esecuzione speculativa.**	Causa la **warp divergence**, riducendo il parallelismo e le prestazioni.
Gestione della Divergenza	**Predizione** del ramo più probabile e **esecuzione speculativa** del codice.	**Esecuzione seriale** dei rami divergenti nel warp, **disabilitando** i thread inattivi.
Ottimizzazioni	Meno critiche, gestite in parte **dall'hardware**.	**Branch predication** (non lo vedremo) e **riorganizzazione del codice** essenziali.
Considerazioni	Il costo della predizione errata è **relativamente basso**.	Il costo della warp divergence è **elevato** a causa della perdita di parallelismo e dell'overhead di esecuzione seriale.

📖 **Branch Efficiency** *(Nsight Compute)*

La **Branch Efficiency** misura la percentuale di **branch non divergenti** rispetto al totale dei branch eseguiti da un warp.

$$\text{Branch Efficiency} = 100 \times \left(\frac{\#\text{Branches} - \#\text{DivergentBranches}}{\#\text{Branches}}\right)$$

- Bassa ⇒ elevata divergenza.
- Elevata ⇒ bassa divergenza.

Prima di Volta il warp agiva come un'unità di esecuzione coesa e sincrona (unico PC e Call Stack). La maschera di attività veniva salvata fino alla riconvergenza del warp, poi ripristinata per la riesecuzione sincrona.
👉 ***Nota***: Una volta scelto un ramo (divergente), questo deve essere completato prima di poterne iniziare un altro (unico PC).

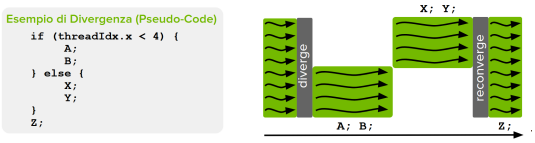


Figura 33: Warp Convergence pre-Volta.

In questo contesto divergenza ⇒ perdita di concorrenza (fino alla convergenza).
👉 ***Nota***: Problema: rischio di deadlock tra thread di un warp se questi dipendono circolarmente.

Da Volta in poi é stato introdotto l'**indipendent scheduling**, che consente **piena concorrenza** tra i thread, indipendentemente dal warp. Questo é reso possibile dal mantenimento dello stato corrente da parte di ciascun thread (PC e call stack). A questo si aggiunge un **ottimizzatore dello scheduling**, che raggruppa i thread attivi dello stesso warp in **unità SIMT**.
⇒ I thread possono divergere e riconvergere **indipendentemente** con **granularità sub-thread**.

In Volta é stata anche introdotta la funzione `__syncwarp()`, per supportare l'ITS e migliorare la gestione della divergenza dei thread. Permette di sincronizzare esplicitamente e riconvergere i thread all'interno di un warp.
Di fatto, blocca l'esecuzione del thread corrente finché tutti i thread specificati nella maschera non hanno raggiunto il punto di sincronizzazione.

```
void __syncwarp(unsigned mask=0xffffffff);
```

Utilizzando questa funzione si **evitano comportamenti non deterministici** dovuti alla divergenza intra-warp e i **thread** del warp sono **allineati** prima di comunicare o accedere a dati condivisi.
👉 ***Nota***: `__syncwarp()` é superflua se ogni thread lavora su dati privati, senza comunicare, mentre é necessaria in caso di comunicazione o dipendenza tra thread del warp.

	Pre-Volta	Post-Volta
Program Counter	Singolo *per warp*	Individuale *per thread*
Scheduling	Lockstep: tutti i thread del warp eseguono insieme	Indipendente: ogni thread può progredire autonomamente
Sincronizzazione	Implicita: i thread sono automaticamente sincronizzati	Esplicita: richiede `__syncwarp`
Divergenza	Serializzazione dei rami divergenti	Esecuzione interlacciata dei rami possibile
Deadlock Intra-Warp	Possibili in certi scenari	Largamente mitigati
Prestazioni con Divergenza	Penalità per serializzazione	Penalità simile, nessun miglioramento intrinseco
Complessità del Codice	Workaround necessari per certi algoritmi	Implementazioni più naturali possibili (ma richiede gestione esplicita)

2.2.4. Sincronizzazione e Comunicazione

Di base host e device operano in modo **asincrono**.
❗ ***Problema***: Senza sincronizzazione, l'host potrebbe tentare di utilizzare risultati non ancora pronti, o modificare dati ancora in uso dalla GPU.

Abbiamo detto che esiste la possibilità di sincronizzare i thread di un blocco in caso di necessità di operazioni di condivisione dati o coordinamento.

Inoltre, da Volta, si rende necessaria una sincronizzazione esplicita, tramite `__syncwarp()`, per garntire la coerenza delle operazioni intra-warp (a causa dell'introduzione dell'indipendent thread scheduling, mentre prima l'intero warp eseguiva la stessa istruzione).

> 📖 **Race Condition o Hazard**
>
> Una **race condition** si verifica quando più thread accedono **concorrentemente** (di cui almeno uno in scrittura) e in modo **non sincronizzato** alla **stessa locazione** di memoria, causando risultati imprevedibili ed errori.
> Tipi di Race Condition:
> - **Read-After-Write** (RAW), se un thread legge prima che un altro finisca di scrivere.
> - **Write-After-Read** (WAR), se un thread scrive dopo che un altro ha letto, invalidando il valore.
> - **Write-After-Write** (WAW) se più thread scrivono nella stessa locazione, rendendo il valore indeterminato.

👉 ***Nota***: Le race condition si verificano perchè i thread in un blocco sono logicamente paralleli ma non sempre fisicamente simultanei, quindi l'ordine di esecuzione tra thread è imprevedibile (a meno di sincronizzazione esplicita).

Per prevenire race condition all'interno di un blocco si usa `__synnchthreads()` per sincronizzare i thread e garantire la visibilità dei dati condivisi.
Per la sincronizzazione di thread di blocchi diversi non esiste un metodo diretto, l'unico sicuro é terminare il kernel ed avviarne uno nuovo.

> 📖 **Deadlock**
>
> Un **deadlock** (o **stallo**) in CUDA si verifica quando i thread di un blocco si **bloccano reciprocamente** in attesa di sincronizzazioni o risorse non raggiungibili, causando un **blocco permanente** dell'esecuzione del kernel.

Cause di deadlock sono:
- **Sincronizzazione Condizionale**: uso di `__syncthreads()` all'interno di condizioni (if, else) per cui solo una parte dei thread del blocco raggiunge il punto di sincronizzazione.
- **Dipendenze Circolari**: situazioni in cui gruppi di thread attendono reciprocamente il completamento di operazioni, creando un ciclo di dipendenze irrisolvibile.
- **Risorse Condivise**: gestione non corretta dell'accesso alla memoria condivisa o ad altre risorse comuni.

Per prevenire e gestire deadlock si ricorre a:
- **Sincronizzazione Completa**, ossia evitare sincronizzazioni nei rami condizionali divergenti, assicurandosi che tutti i thread raggiungano i punti di sincronizzazione.
- **Ristrutturazione del Codice** rimuovendo dipendenze condizionali, organizzando le operazioni a fasi da completare da parte di tutti i thread prima di passare alla successiva.
- **Indipendent Thread Scheduling** (Volta+).

⚠️ Sincronizzazione in CUDA

La sincronizzazione è il meccanismo che permette di **coordinare** l'esecuzione di thread paralleli e garantire la **correttezza** dei risultati, **evitando race condition/deadlock** e **accessi concorrenti non sicuri** alla memoria.

Esistono diversi livelli di sincronizzazione in CUDA:

- **Livello di Sistema** (Host-Device), prevede il bocco dell'applicazione host finché tutte le operazioni sul device non sono completate. Questo garantisce che il device abbia terminato l'esecuzione.

```
cudaError_t cudaDeviceSynchronize(void);
```

- **Livello di blocco** di thread, prevede la sincronizzazione di tutti i thread all'interno di un **unico blocco**. Ogni thread attende che tutti gli altri thread nel blocco raggiungano il punto di sincronizzazione. Garantisce la visibilità delle modifiche alla shared memory tra i thread del blocco.

```
__device__ void __syncthreads(void);
```

- **Livello di warp** (disponibile con ITS, CUDA 9.0+ e Volta+), prevede la sincronizzazione dei thread di un **unico warp**, garantendo la **riconvergenza** (in presenza di divergenza) ed **ottimizzando** la **cooperazione** tra i thread del warp.

```
__device__ void __syncwarp(unsigned mask=0xffffffff);
```

Oltre alla sincronizzazione, per evitare race condition per operazioni Read-Modify-Write si rende necessario l'uso di **operazioni atomiche**, le quali garantiscono l'integratià di queste operazioni.
❗ ***Problema***: Con operazioni non atomiche si può incorrere in conteggi errati (per esempio se 2+ thread leggono ed incrementano una variabile condivisa nello stesso istante, riscrivono lo stesso valore incrementato complessivamente di 1, invece di incrementarlo ciascuno), aggiornamenti di dati persi (a causa di modifiche sovrascritte tra thread) e a comportamenti non deterministici.

📖 Operazioni Atomiche

Le **operazioni atomiche** in CUDA sono operazioni del tipo **Read-Modify-Write** eseguite (solo su funzioni device) come **singola istruzione** hardware **indivisibile**.
Caratteristiche:

- **Escusività** dell'**accesso alla memoria** (esecuzione sequenziale, operata dall'hardware, dei thread che accedono alla stessa posizione di memoria).
  ⇒ **Riduzione del parallelismo effettivo**.
- **Prevenzione delle interferenze**.
- **Compatibilità con memoria globale** e **condivisa**, operando su word di 32, 64 o 128 bit.

Tipiche operazioni atomiche sono quelle **matematiche** (addizione, sottrazione, massimo, minimo, incremento, decremento), **bitwise** (operazioni bit a bit come AND, OR, XOR), **swap** (cambio valore in memoria con uno nuovo).

Operazioni Atomiche

```
// Operazioni atomiche aritmetiche
int atomicAdd(int* addr, int val);        // Somma val a *addr
int atomicSub(int* addr, int val);        // Sottrae val da *addr
int atomicMax(int* addr, int val);        // Aggiorna *addr al max tra *addr e val
int atomicMin(int* addr, int val);        // Aggiorna *addr al min tra *addr e val
unsigned int atomicInc(unsigned int* addr, unsigned int val); // Incrementa *addr, ciclato tra 0 e val
unsigned int atomicDec(unsigned int* addr, unsigned int val); // Decrementa *addr, ciclato tra 0 e val

// Operazioni atomiche di confronto
int atomicExch(int* addr, int val);          // Scambia *addr con val
int atomicCAS(int* addr, int cmp, int val);  // Confronta *addr con cmp, aggiorna *addr a val se uguali

// Operazioni atomiche bitwise
int atomicAnd(int* addr, int val);        // AND tra *addr e val, aggiorna addr
int atomicOr(int* addr, int val);         // OR tra *addr e val, aggiorna addr
int atomicXor(int* addr, int val);        // XOR tra *addr e val, aggiorna addr
```

- Leggono il valore originale dalla memoria, eseguono l'operazione e salvano il risultato **nello stesso indirizzo**, **restituendo il valore originale pre-modifica**.
- **Supporto a Tipi Estesi:** Esistono anche varianti atomiche per operazioni su tipi a 64 bit (`long long int`) e floating point (`float` e `double`), supportate su architetture recenti.

2.2.5. Ottimizzazione delle risorse

Il **Resource Partitioning** riguarda la **suddivisione** e la **gestione** delle **risorse hardware** limitate all'interno di una GPU, in particolare all'interno di ogni SM.
L'obiettivo è trovare un **equilibrio** nella distribuzione di registri e memoria condivisa tra thread e blocchi, ottimizzando l'efficienza complessiva dell'esecuzione dei kernel CUDA.

Ogni SM possiede una quantità limitata di registri (usati dai thread attivi) e shared memory (riservata e suddivisa tra i blocchi attivi), mentre il numero di blocchi di thread e warp potenzialmente residenti simultaneamente su un SM dipende dalla disponibilità delle risorse, ma anche dalle richieste specifiche del kernel per l'esecuzione.
👉 ***Nota***: Se le risorse di uno SM non permettono di eseguire almeno un blocco di thread, il kernel **fallisce**.

La questione é che tutti i blocchi in una griglia eseguono lo stesso programma usando lo stesso numero di thread, portando a 3 requisiti di risorse fondamentali (condivisi):
- **Dimensione del blocco** (numero di thread concorrenti),
- **Memoria condivisa** (dichiarata con `__shared__`, comune a ciascun thread dello stesso blocco),
- **Registri** (privati dei thread), i quali dipendono dalla complessità del programma (le variabili allocate nel kernel vengono salvate nei registri, a meno di diversa indicazione o saturazione).

👉 ***Nota***: Se si saturano la Shared Memory o il numero di thread totali, l'SM smette di accettare nuovi blocchi finchè quelli attivi non terminano.
👉 ***Nota***: Se si saturano i registri (**registry spilling**), all'occorrenza di nuovi registri, le variabili sono invece allocate in Local Memory (dentro la Global Memory ⇒ overhead).
👉 ***Nota***: Anche se si lancaisse un kernel che non usa registri, né Shared Memory, il numero di thread eseguibili é vincolato dal tetto di blocchi residenti per SM e dal numero di warp/thread residenti per SM (quindi non sono caricabili infiniti thread).

Essendo queste risorse gestibili dal programmatore, attraverso uno studio mirato é possibile migliorare il numero di warp/blocchi/thread per SM. Questo studio deve tenere conto, non solo delle esigenze del problema, ma anche dell'architettura scelta e della sua CC (i valori massimi di thread/warp/blocchi/ per SM e memorie varie per thread/SM dipendono dalla CC).

*Valori concorrenti per singolo SM

Compute Capability	Architettura	Max Thread per Blocco	Max Thread per SM*	Max Warps per SM*	Max Blocchi per SM*	Max Registri per Thread	Memoria Condivisa per SM
1.x	Tesla	512	768	24/32	8	124	16KB
2.x	Fermi	1024	1536	48	8	63	48KB
3.x	Kepler	1024	2048	64	16	255	48KB
5.x	Maxwell	1024	2048	64	32	255	64KB
6.x	Pascal	1024	2048	64	32	255	64KB
7.x	Volta/Turing	1024	1024/2048	32/64	16/32	255	96KB
8.x	Ampere/Ada	1024	1536/2048	48/64	16/24	255	164KB
9.x	Hopper	1024	2048	64	32	255	228KB
10.x/12.x	Blackwell	1024	2048/1536	64/48	32	255	128KB

Figura 34: Caratteristiche e capacità delle diverse architetture/CC.

📖 Occupancy

L'**occupancy** rappresenta il **grado di utilizzo delle risorse** di calcolo dell'**SM**. Questa é il rapporto tra i warp attivi e il numero massimo di warp supportati per SM (dipendente dalla CC):

$$\text{Occupancy } [\%] = \frac{\text{Active Warps}}{\text{Maximum Warps}}$$

- Occupancy **ottimale** = livello sufficiente per nascondere la latenza (ulteriori aumenti potrebero degradare le prestazioni a causa della riduzione di risorse per thread).
- Occupancy bassa = scarsa efficienza nell'emissione delle istruzioni, poiché non ci sono abbastanza warp eleggibili per nascondere la latenza tra istruzioni dipendenti.

👉 ***Nota***: Un' occupancy elevata non garantisce sempre prestazioni migliori. Oltre certa soglia, fattori come i pattern di accesso alla memoria e il parallelismo delle istruzioni possono diventare più rilevanti per l'ottimizzazione.

L' occupancy di un kernel CUDA si divide in **teorica**, basata sui limiti hardware, ed **effettiva**, misurata a runtime.

📖 Occupancy Teorica (Theoretical)

L'**occupancy teorica** è determinata dalla **configurazione di lancio** (numero di blocchi/thread, quantità di memoria condivisa, numero di registri per thread) e i limiti dell'SM (compute capability).

$$\text{Limite massimo warp attivi per SM} = (\text{Limite massimo blocchi attivi}) \times (\text{Warp per blocco})$$

👉 ***Nota***: È possibile aumentare il limite incrementando il numero di warp per blocco (dimensioni del blocco) o modificando i fattori limitanti (registri e/o shared memory) per aumentare i blocchi attivi per SM.

📖 Occupancy Effettiva (Achieved)

L'**occupancy effettiva** misura il **numero reale di warp attivi** durante l'esecuzione del kernel (questo numero varia durante l'esecuzione).

Il calcolo dell'occupancy effettiva (v. Nsight Compute) prevede la misurazione su ciascuno warp scheduler dei warp attivi ad ogni clock tramite **contatori di prestazioni hardware**. I conteggi sono poi sommati su tutti i warp scheduler di ogni SM (1 per SMSP) e divisi per i clock attivi dell'SM, per calcolare la **media dei warp attivi**. Infini si divide per il numero massimo di warp attivi supportati dall'SM (Maximum Warps), ottenendo l'**occupancy effettiva media** per SM.
👉 ***Nota***: Si avrà sempre

$$\text{Occupancy Teorica} \geq \text{Occupancy Effettiva}$$

Si avrà che l'occupancy teorica é molto maggiore della effettiva se il numero teorico di warp attivi non viene mantenuto durante l'attività dello SM. Questo può avvenire anche a dispetto del resource partitioning, quando si ha:

- **Carico di lavoro sbilanciato nei blocchi** (warp dello stesso blocco hanno tempi di esecuzione diversi dovuto per esempio alla divergenza).
  ⇒ Bilanciare il carico tra warp.
- **Carico di lavoro sbilanciato tra blocchi** (blocchi della griglia hanno durate diverse).
  ⇒ Lanciare più blcchi o kernel concorrenti per ridurre l'effetto.
- **Numero insufficiente di blocchi lanciati** (se la griglia ha meno blocchi del numero di SM del device, allora alcuni SM rimangono inattivi).
- **Wave parziale**.

👉 ***Nota***: L'obiettivo finale non é massimizzare l'occupancy, ma minimizzare il tempo di esecuzione del kernel.

Linee guida pratiche

- ❌ **Occupancy < 25–30 % → problema serio**
  - Non ci sono abbastanza warp per nascondere le latenze (*critico soprattutto per kernel memory-bound*).
  - La GPU risulta sottoutilizzata, con SM spesso inattivi
  - **Azione**: ridurre l'uso di registri / shared memory per thread oppure aumentare il numero di thread per blocco, se possibile.
- ✅ **Occupancy 50–80 % → generalmente buono**
  - Warp sufficienti per un buon **latency hiding** nella maggior parte dei casi
  - Risorse adeguate per ciascun thread
  - **Focus**: ottimizzare coalescing, divergenza, e accessi alla memoria
- ⚠️ **Occupancy > 90 % → non sempre vantaggiosa** (kernel memory bound 🙂 compute bound 🤔)
  - Latency hiding ottimo ma attenzione: se limita troppo le risorse il guadagno si perde.
  - Oltre una certa soglia, più occupancy potrebbe non migliorare le performance.

⚠️ **Linee Guida per dimensionamento di Griglia e Blocchi**

- Mantenere il numero di thread **multiplo** della dimensione del **warp** (32).
- **Evitare dimensioni di block piccole**, iniziare con almeno 128 o 256 thread per block.
- Regolare la **dimensione del blocco in base ai requisiti di risorse** del kernel.
- Mantenere **numero di blocchi ≫ del numero di SM** per esporre sufficiente parallelismo al dispositivo (latency hiding).
- **Condurre esperimenti** per scoprire la migliore configurazione di esecuzione e utilizzo delle risorse.

2.2.6. Parallelismo avanzato

📖 **Dynamic Parallelism**

Il **Dynamic Parallelism** (CUDA 5.0+, Kepler+, CC 3.5+) permette la **creazione** e **sincronizzazione** dinamica (on the fly) di nuovi kernel direttamente **dalla GPU**. È quindi possibile posticipare a runtime la decisione su quanti blocchi e griglie creare sul device (utile quando la quantità di lavoro nidificato è sconosciuta).

> Supporta un **approccio gerarchico** e **ricorsivo** al parallelismo **evitando** continui **passaggi** fra CPU e GPU.

Il dynamic parallelism si applica ad **algoritmi ricorsivi** (come Quick o Merge sort, in cui la profondità della ricorsione é sconosciuta), a **strutture ad albero** (elaborazione parallela nidificata irregolare) e ad **elaborazione di immagini e segnali** (decomposizione dinamica delle aree di elaborazione).

Vantaggi:
- **Flessibilità** del parallelismo in base ai dati elaborati, senza dover prevedere tutto a priori.
- **Scalabilità**, si creano nuovi blocchi e griglie solo quando necessario.
- **Efficienza**, poichè si riduce il collo di bottiglia CPU-GPU, spostando parte del controllo dell'esecuzione sulla GPU.

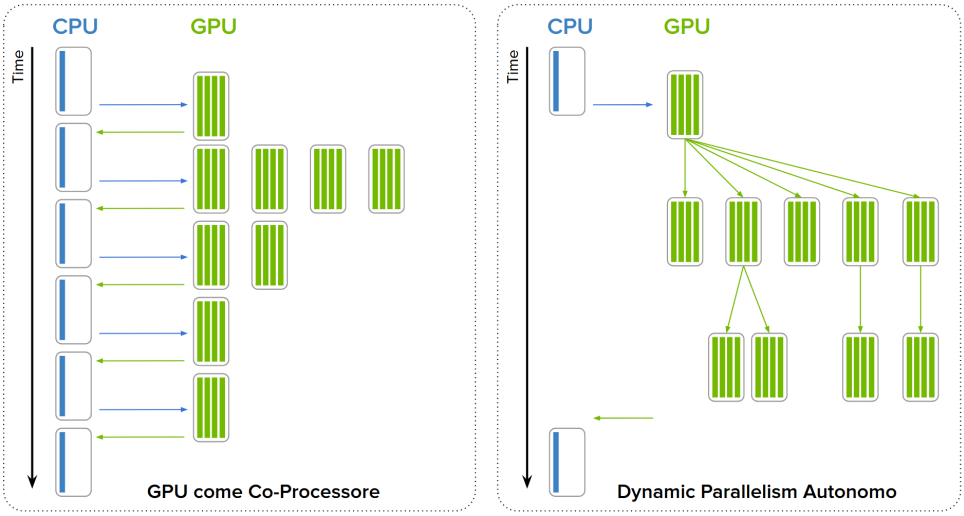


Figura 35: Lancio kernel tradizionale (host-only) VS Dynamic Parallelism

Funzionamento:

1. Un thread, un blocco di thread o una griglia (**parent**) lancia una nuova griglia (**child** grid).
   ⇒ Il child (grid) **eredita** dal kernel padre certi attributi e limiti (come configurazione cache L1/ Shared Memory e dimensione dello stack).
2. I blocchi della griglia child sono eseguiti (se possibile) **in parallelo** e in modo **indipendente** rispetto al parent.
   👉 ***Nota*** : Il parent continua l'esecuzione immediatamente dopo il lancio del child (**asincronicità**).
   👉 ***Nota*** : Affinchè il parent sia considerato **completato**, tutti i suoi child devono completare.

É importante specificare che, per quanto riguarda **visibilità** e **sincronizzazione**:

- Ogni **child** grid lanciata da un thread è **visibile** a **tutti i thread dello stesso blocco**.
  Questo permette a qualsiasi altro thread dello stesso blocco di interagire con quel lancio, anche a chi non lo ha generato. Impedendo che la perdita o il fallimento di un singolo thread lasci "orfana" una griglia child.
- Se i thread di un blocco terminano prima che tutte le loro griglie child abbiano completato, il sistema attiva automaticamente una **sincronizzazione implicita** per attendere il completamento di queste griglie.
  👉 ***Nota*** : Le risorse allocate all'SM non vengono liberate fino a che non sono terminati tutti i figli (abusandone ⇒ la GPU potrebbe saturarsi con parent che attendono e basta, portanto eventualmente al deadlock).
- Un thread può **sincronizzarsi** **esplicitamente** con le proprie griglie child e con quelle lanciate da altri thread nel suo blocco utilizzando **primitive di sincronizzazione** (`cudaDeviceSynchronize`).

👉 ***Nota*** : `cudaDeviceSynchronize` attende il completamento di tutte le griglie (kernel) precedentemente lanciate da qualsiasi thread del blocco corrente, includendo tutti i kernel discendenti (child) nella gerarchia. Se chiamata da un singolo thread, gli altri del blocco continuano l'esecuzione.
👉 ***Nota*** : `cudaDeviceSynchronize` è un'operazione computazionalmente costosa perché può causare lo **swap-out** del blocco in esecuzione e, in caso di sospensione, richiede il **trasferimento dell'intero stato del blocco** (registri, memoria condivisa, PC) nella memoria del device (che poi va ripristinato – **swap-in**.

- Quando un thread parent lancia una child grid, l'**esecuzione della griglia figlio non è garantita** immediatamente, a meno che il blocco di thread genitore non esegua una **sincronizzazione esplicita**.
  *Non è detto che il figlio parta nel millisecondo in cui viene chiamato. Se la GPU è piena, il figlio resta in coda. Eseguendo una sincronizzazione **esplicita**, il blocco padre non solo aspetta, ma «cede» il passo e assicura che il sistema di scheduling dia priorità alla risoluzione di quelle pendenze per permettere al padre di sbloccarsi.*

In sostanza le child grid sono visibili unicamente internamente al blocco di chi le ha lanciate, mai esternamente.
👉 ***Nota*** : I kernel lanciati dinamicamente possono avere una configurazione di griglia e blocco indipendente dal kernel genitore.

Per quanto riguarda le memorie, invece:

- Parent e child (grid) condividono lo **stesso spazio di memoria globale** (a cui accedono concorrentemente) e **costante**.
  👉 ***Nota*** : Accedendo alla memoria globale, la **coerenza** non é garantita tra parent e child, se non all'**avvio** del child e al suo **completamento**.
- Parent e child (grid) accedono a **diverse** zone di memoria **locale** e **shared**, in quanto la memoria locale é privata di ciascun thread e quella Shared di un blocco. Pertanto non sono visibili o coerenti tra parent e child.

👉 ***Nota*** : **Non è valido** passare un **puntatore** a memoria locale o shared come argomento quando si lancia una griglia child, tuttavia é possibile e lecito passare variabili per **copia**.

Il Dynamic Parallelism non é esente da limitazioni:

- supportato da device con CC $\geq$ 3.5.
- I kernel possono essere lanciati **solo** sullo **stesso device** (non su device fisicamente separati).
- Massima nesting depth di **24 livelli**, ma, di fatto, limitato anche dalla memoria richiesta dal **runtime** del device.
  👉 ***Nota*** : Il Runtime riserva **memoria aggiuntiva** per la sincronizzazione griglia parent-child.
- Uso di `cudaDeviceSynchronize` é deprecato da CUDA 11.6 (solo da device), rimosso per CC 9.0+.

2.3. Modello di Memoria CUDA

2.3.1. Modelli di Performance

Per ottimizzare un kernel CUDA è cruciale comprendere dove risieda il collo di bottiglia. Questa distinzione determina le strategie di ottimizzazione da adottare.

> 📖 **Kernel Memory Bound**

Un kernel è **memory bound** quando il tempo di esecuzione è **limitato** dalla **velocità di accesso alla memoria** piuttosto che dalla capacità di elaborazione dei core.
⇒ La GPU trascorre più tempo in attesa dei dati piuttosto che ad eseguire calcoli.

Cause (comuni):
- **Accessi frequenti** alla memoria con latenza elevata.
- **Banda** di memoria **insufficiente** rispetto ai requisiti del kernel.

Kernel Compute Bound

Un kernel é **compute bound** quando il **tempo di esecuzione** è **limitato dalla capacità** di **calcolo** della GPU, con sufficiente larghezza di banda per i dati.
⇒ La GPU trascorre più tempo a eseguire calcoli che in attesa dei dati.

Causa (comune) sono le operazioni aritmetiche intesnive (moltiplicazioni di matrici dense o convoluzioni), le quali richiedono elevati FLOP rispetto agli accessi in memoria.

FLOPs

I **FLOPs** inidicano le Floating Point Operations per Second e sono espressi da

$$\textbf{FLOP/s} = \frac{N_{\text{Floating Point Operations}}}{\textbf{Tempo trascorso}} \qquad \left[\frac{\text{FLOP}}{s}\right]$$

I FLOPS si utilizzano per valutare quanto un kernel sia **compute-bound**. L'unità di misura, FLOPs, può essere preceduta dal suffisso per le grandezze (T - tera, G - giga, M - mega, ...).

La **Peak Performance** della GPU rappresenta il limite teorico massimo di FLOPs.

Bandwidth

La **Bandwidth** (o **larghezza di banda**) indica la **quantità di dati** trasferiti al secondo.

$$\textbf{Bandwidth} = \frac{\textbf{Dimensione dati trasfriti}}{\textbf{Tempo trascorso}} \qquad \left[\frac{\text{Byte}}{s}\right]$$

La bandwidth si utilizza per valutare quanto un kernel sia **memory-bound**. L'unità di misura, B/s, può essere preceduta dal suffisso per le grandezze (T - tera, G - giga, M - mega, ...).

La **Peak Bandwidth** dell'hardware rappresenta il limite teorico massimo raggiungibile in termini di banda.

Ora vediamo alcune altre definizioni di metriche legate a quanto visto finora.

Memory Latency

La **Memory Latency** é il **tempo richiesto** per soddisfare una richiesta di dati dalla memoria della GPU, inclusi i ritardi di trasferimento fino ai core.

Memory Bandwidth

La **Memory Latency** é la **quantità massima di dati** che può essere trasferita tra la memoria della GPU e gli altri componenti (ad esempio, gli SM) in un'unità di tempo.

La banda si divide in **teorica** ed **effettiva**.

- La banda **teorica** coincide con la **massima larghezza di banda** raggiungibile con l'hardware disponibile.
- La banda **effettiva** é la larghezza di banda **realmente** raggiunta da un kernel in esecuzione.

$$\text{Bandwidth effettiva (GB/s)} = \frac{(\text{Byte letti} + \text{Byte scritti} \times 10^{-9})}{\text{Tempo trascorso}} \qquad \left[\frac{\text{B}}{\text{ns}}\right]$$

2.3.1.1. Roofline Model

Modello Roofline

Il **modello Roofline** è un metodo grafico utilizzato per rappresentare le **prestazioni** di un **algoritmo** (o di un kernel CUDA) in relazione alle **capacità di calcolo** e **memoria** di un sistema.

Intensità Aritmetica

L'**intensità aritmetica** (AI) misura il rapporto tra la **quantità di operazioni di calcolo** e il **volume di dati trasferiti** dalla/verso la memoria di un algoritmo/kernel:

$$\mathbf{AI} = \frac{\textbf{FLOPs}}{\textbf{Byte trasferiti}}$$

dove
- **FLOPs** = Numero di operazioni in virgola mobile o operazioni aritmetiche in generale.
- **Byte trasferiti** = Quantità di dati letti o scritti dalla memoria DRAM.

Soglia di Intensità Aritmetica

La **soglia di intensità aritmetica** dipende dall'hardware specifico ed é definita come:

$$\mathbf{AI_{ridge}} = \frac{\textbf{Theoretical Computational Peak FP64 Performance}}{\textbf{Bandwidth Peak Performance}} = \frac{\mathbf{P_{max}}}{\mathbf{BW}} \qquad \left[\frac{\text{FLOP/s}}{\text{B/s}}\right] = \left[\frac{\text{FLOP}}{\text{B}}\right]$$

dove
- **Computational Peak Performance** = Massima capacità teorica di calcolo di un device, misurata in FLOPS.
- **Bandwidth Peak Performance** = Velocità massima con cui i dati possono essere trasferiti tra GPU e memoria principale.

Intensità Aritmetica e soglia

Esiste una relazione tra l'intensità aritmetica e la soglia dell'intensità aritmetica.
- se $\boldsymbol{AI} < \boldsymbol{AI_{\text{ridge}}} \Rightarrow$ **Memory Bound**.
- se $\boldsymbol{AI} > \boldsymbol{AI_{\text{ridge}}} \Rightarrow$ **Compute Bound**.

Curve nel Diagramma

- **Bandwidth Roof**: Una linea retta inclinata che rappresenta il limite imposto dalla banda di memoria. La pendenza di questa retta è pari alla bandwidth della memoria del device (DRAM).
- **Computational Roof**: Una linea orizzontale che rappresenta il limite massimo di prestazioni computazionali in doppia precisione (FP64 Roofline). Questa è la massima velocità a cui la GPU può eseguire operazioni in FP64.

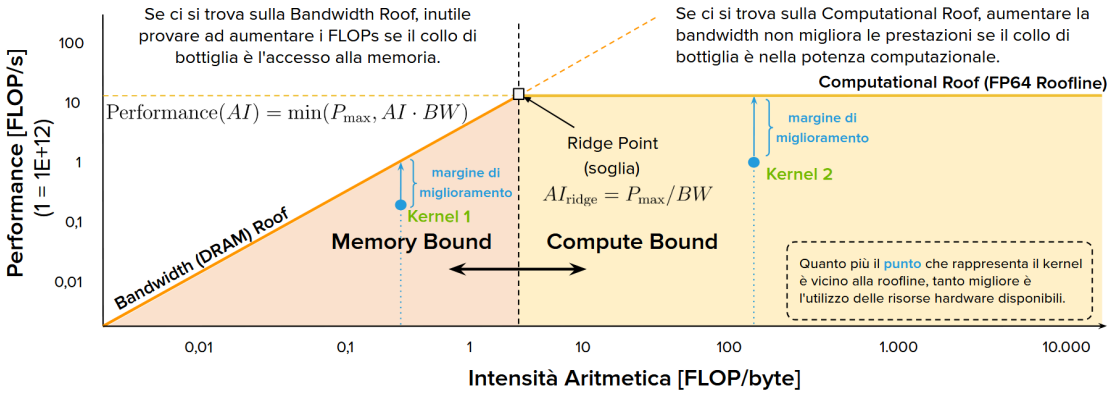


Figura 36: Diagramma Roofline

Il **margine di miglioramento** corrisponde alle ottimizzazioni realizzabili circa le **risorse** computazionali e gli **accessi** in memoria. *Tali ottimizzazioni spostano il kernel verso l'alto (nel grafico), aumentando le prestazioni effettive, e verso destra, aumentando l'intensità aritmetica, riducendo i byte trasferiti da DRAM.*

- In un contesto **memory bound** sfruttare **shared** memory e **registri** per data reuse riduce i trasferimenti da global memory, mentre ottimizzare i **pattern di accesso** (coalescenza, allineamento, stride) migliora la bandwidth effettiva, aumentando l'AI e potenzialmente cambiando il regime verso compute-bound.
- In un contesto **compute-bound**, massimizzare **occupancy**, **ridurre divergenze**, **ottimizzare ILP** e la scelta del **tipo di dato** avvicina il kernel al tetto computazionale.

2.3.2. Gerarchia di Memoria CUDA

CPU e GPU usano la **DRAM** per la memoria principale e la **SRAM** per registri/cache, mentre si usano dischi/flash per la memoria più lenta e capiente (e persistente).
Tutto questo crea l'illusione di una grande memoria rapida.

Giusto per rinfrescare la memoria (da reti logiche :D)...
*La **SRAM** é "statica" in quanto i bit rimangono memorizzati fintanto che persiste la corrente, senza bisogno di interventi esterni. Questa memoria usa un **numero elevato di transistor** per bit (tipicamente 6) e quindi anche un'area più elevata. É realizzata a **latch** che compongono uno **stato bi-stabile** (ossia per cambiare bit é necessario forzare i nodi tramite transistor di accesso). É **veloce** in quanto non c'é attesa, ma i bit (tramite transistor) commutano sostanzialmente all'istante.*

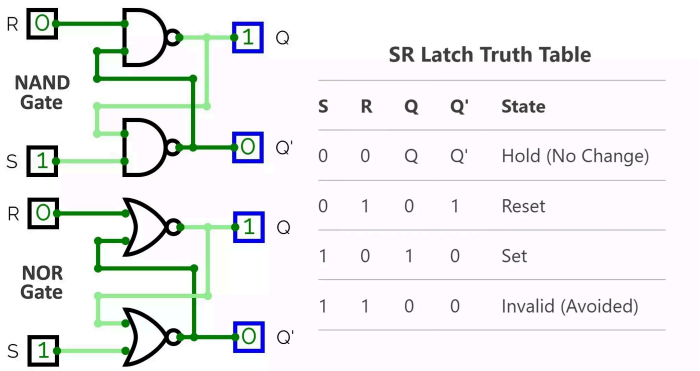


Figura 37: Latch SR (per contesto/ripasso).

*La **DRAM** é definita "dinamica" perchè i bit devono essere **costantemente "rinfrescati"**, onde evitare la perdita dell'informazione. Queste sono composte da un **unico transistor** (e un condensatore), pertanto occupano meno area e sono meno costose. Il bit rappresenta di fatto la **carica elettrica nel condensatore** (motivo per cui é necessario il refresh). Un problema delle DRAM é la **lettura distruttiva**, infatti per leggere una cella é necessario misurare la tensione del condensatore, quindi scaricarlo, per poi ricaricarlo (riscriverlo) con quanto letto. Il motivo per cui é **lenta** é legato proprio allo caricamento e scaricamento (latenza RC), che impiega un tempo discreto.*

CUDA espone **più livelli** della gerarchia rispetto ai modelli CPU, offrendo un controllo più esplicito per ottimizzare le prestazioni.

Vediamo ora le differenze tra la memoria CPU (**DDR**) e GPU (**GDDR**):

	DDR (Double Data Rate)	GDDR (Graphics Double Data Rate)
Target	CPU	GPU
Utilizzo	Sistemi operativi, applicazioni multi-tasking, database	Gaming, rendering 3D, intelligenza artificiale
Architettura	**Ottimizzata per bassa latenza** con bus a 64 bit, progettata per accessi rapidi nelle operazioni di sistema.	**Memoria ottimizzata per massimo throughput** con bus dati ampi (es. 384 bit) per garantire alta banda.
Memory Clock	Fino a 3600 MHz (DDR5)	Fino a 1500 MHz (GDDR6X)
Larghezza di banda	Fino a 100 GB/s (DDR5)	Fino a 1 TB/s (GDDR6X)
Consumo energetico	Basso consumo in idle, efficiente per carichi di lavoro variabili	Consumo elevato anche in idle, ottimizzato per prestazioni costanti
Costo per GB	Circa $10-$20 (DDR5)	Circa $30-$60 (GDDR6X)
Capacità massima per modulo	Fino a 128-256GB (DDR5)	Fino a 24-48GB (GDDR6/GDDR6X)

In realtà esistono diverse tecnologie di memoria GPU (oltre che tra le diverse generazioni) in base alla destinzione della scheda. Tipicamente le schede consumer usano memorie **GDDR**, mentre quelle server/HPC **HBM**.

	GDDR (Graphics Double Data Rate)	HBM (High Bandwidth Memory)
Esempio GPU	NVIDIA RTX 4090 Ada (GDDR6X)	NVIDIA H100, GH200 (HBM3, HBM3e)
Utilizzo	Grafica avanzata, AI su piccola scala, gaming	HPC, training AI intensivo, inferenza AI in tempo reale
Architettura	Chip singoli saldati al PCB	Moduli impilati sul die della GPU (più DRAM in uno spazio ridotto)
Bus Width	Fino a 384-bit (GDDR6X)	Fino a 5120-bit (HBM3, NVIDIA H100)
Banda Passante	~1 TB/s (GDDR6X, RTX 6000 Ada)	~2 TB/s (HBM3, NVIDIA H100), ~3 TB/s (HBM3e, NVIDIA GH200)
Latenza	Più bassa rispetto a HBM	Più alta rispetto a GDDR
Capacità Massima	Fino a 24GB (modulo GDDR6/GDDR6X)	Fino a 80-144 GB (NVIDIA H100, GH200)
Efficienza Energetica	Più alto consumo rispetto a HBM	Più efficiente, ottimizzato per HPC
Costo	Più accessibile, adatta a workstation e GPU mainstream	Costosa, ideale per acceleratori di fascia alta

Le applicazioni spesso seguono il **principio di località**, accedendo a una porzione relativamente piccola e localizzata del loro spazio di indirizzamento in un dato momento:

- **Temporale**: Dati usati di recente hanno più probabilità di essere riutilizzati a breve.
- **Spaziale**: Dati vicini a quelli usati di recente hanno più probabilità di essere necessari.

Come abbiamo visto in precedenza (v. *Recap Memorie GPU*), la **gerarchia** di memoria offre **livelli di memoria** con differenti latenze, larghezze di banda e capacità. I livelli bassi (vicini ai core) sono più rapidi (bassa latenza), costosi e di dimensioni inferiori, pertanto tipicamente vi si accede di frequente. Viceversa, i livelli più alti (più lontani dai core), hanno latenze maggiori, ma capacità più elevate, costi minori e pertanto accessi meno frequenti.

👉 ***Nota*** : Per le loro caratteristiche, i livelli più bassi, vicini ai core, sono gestiti per contenere i dati più caldi (**hot data**), ovvero quelli su cui la GPU sta lavorando, basandosi sul principio di località spaziale e temporale.

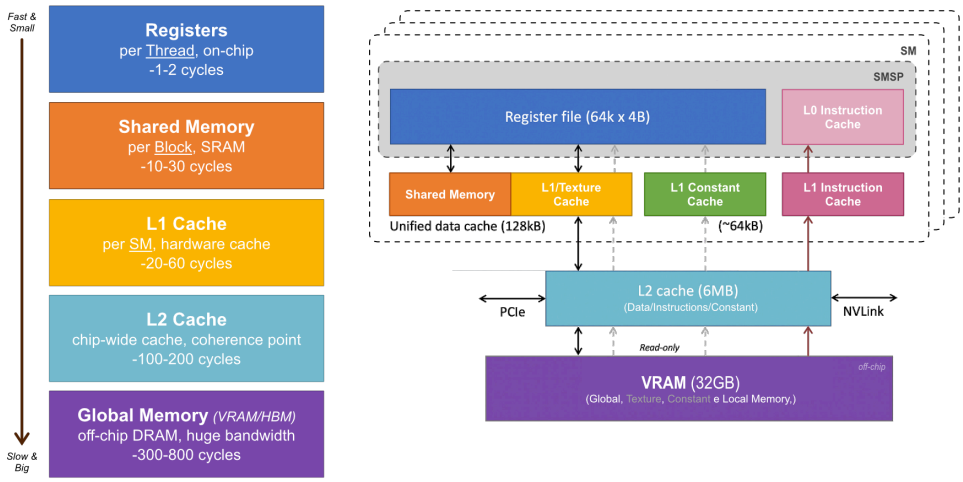
⚠️ **Gerarchia Memorie GPU - Approfondimento**

Partendo in ordine di vicinanza ai core si trovano:

1. **Registri** → Più veloci, **privati** per ogni **thread** (uso per variabili temporanee).
2. **Shared Memory** → Veloce, **condivisa** tra thread di un **blocco** (uso per comunicazione/cooperazione)
3. **Cache** (**L1**, **L2**, **Texture**, **Constant**, **Instruction**) → **Memoria interna automatica**, per ridurre i tempi di accesso ai dati usati di frequente.
4. **Local Memory** → **privata** per ogni **thread**, usata per **grandi variabili** o **register spill**.
5. **Constant Memory** → **read-only** per dati che non cambiano durante l'esecuzione del kernel.
6. **Texture Memory** → **read-only** per **accessi spazialmente coerenti** (es. immagini).
7. **Global Memory** → **grande** e **lenta**, **accesseibile a tutti** i thread e alla **CPU** (PCIe).

👉 ***Nota*** : Nell'immagine di contesto riportata di seguito, i bus PCIe ed NVLink sono "collegati" alla cache L2, in realtà si connettono alla VRAM, passando per i memory controlle che assicurano la coerenza dei dati eventualmente in corso di scrittura.

👉 ***Nota*** : In termini di programmazione possiamo distinguere le memorie tra programmabili e non. Le prime permettono un controllo esplicito sul posizionamento dei dati (diverse supportate da CUDA), mentre quelle non programmabili (come le cache L1 ed L2 CPU) non permettono nessun controll, ma posseggono una gestione automatica.



Contesto: Figura 28.

👉 ***Nota***: Come visibile dallo schema, i **Registri** e la **Instruction Cache L0** sono replicati a livello di **SM sub-partition** (SMSP), pertanto ogni SM dispone di 4 repliche di tali risorse. In maniera analoga, la **Shared Memory** e la **Cache L1** sono condivise a livello di intero **SM**. La **Cache L2** è invece l'unica risorsa globale, comune a tutti gli SM del chip.

Passiamo ora a vedere ciascuno strato nel dettaglio.

📖 Registri (GPU)

I **registri**:

- Memoria **on-chip** GPU (⇒ **massima larghezza di banda** e **minima latenza**).
- Dimensione di **32 bit** ciascuno (tipicamente
- Spazio in cui vengono allocate le variabili determinabili a compile time (anche array, se locali, molto piccoli e con indice costante).
- **Privati** per **thread** .
- **Limite massimo** per thread dipendente da CC (oggi 255).

Saturando i registri (o il limite), si ha **register spilling** ⇒ spostamento eccesso in memoria locale (lento).

`nvcc` applica delle euristiche per minimizzare l'utilizzo di registri ed evitare register spilling.

- `-Xptxas -v`, `-abi=no` : Mostra l'utilizzo delle risorse hardware (numero di registri, bytes di shared memory, etc.).
- `-maxrregcount=32` : Limita il numero massimo di registri per unità di compilazione (ignorato specificati i launch bounds specificati in `__launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor)` eventualmente specificato nella dichiarazione del kernel).

📖 Memoria Locale

La **Memoria Locale**:

- Memoria **off-chip** (**DRAM** ⇒ **alta latenza** e **bassa lunghezza di banda**).
- **Privata per thread** e **non condivisa**.
- Usata per variabili non contenibili (causa spazio) nei registri:
  - array locali referenziati con indici non determinabili a compile time;
  - grandi strutture o (grandi) array locali;
  - variabili che eccedono il limite dei registri (register-spill automatico del compilatore).

- Per CC 2.0+ i dati vengono posti in cache L1 (a livello di SM) e L2 (a livello di device).

✅ ***Tip***: Memoria in DRAM ⇒ Accessibile a tutti i thread.

📖 Shared Memory

La **Shared Memory** (SMEM):

- Memoria **on-chip** (⇒ elevata bandwidth e bassa latenza), solitamente spazio condiviso con Cache L1.
- Organizzata in **memory banks** di uguale dimensione, che permettono l'accesso simultaneo a più dati se i thread leggono da indirizzi diversi su banchi distinti (evitando **bank conflicts**).
- **Condivisa** tra thread di un **blocco**.
- **Programmabile** con controllo esplicito: variabili `__shared__` (ciclo di vita legato al blocco).
- Richiede sincronizzazione esplicita (`__syncthreads`) per prevenire **data hazards**.

La smem sarà approfondita in seguito (Sezione 2.4.1).

📖 Cache L1

La **Cache L1**:

- Memoria **on-chip** (⇒ elevata bandwidth e bassa latenza), solitamente spazio condiviso con SMEM.
- **Gestita automaticamente** dall'hardware per ridurre la latenza degli accessi alla memoria globale.
- Serve **tutti** i thread di un **SM**.
- Gestisce automaticamente la coerenza dei dati, senza bisogno di sincronizzazione esplicita.

👉 ***Nota***: La **quantità di memoria** assegnata alla cache L1 e alla memoria condivisa è **configurabile** per ogni chiamata al kernel.

📖 Memoria Costante

La **Memoria Costante**:

- Memoria di **sola lettura off-chip** (**DRAM**).
- Accessibile a **tutti** i thread.
- Una **porzione cachata** on-chip per ogni SM (tipicamente 8KB).
- **Visibile globalmente** a tutti i kernel nella **stessa unità** di compilazione (programma).
- Inizializzata da **host** con `__constant__` e **non modificabile dai kernel** (read-only).

```
cudaError_t cudaMemcpyToSymbol(const void* symbol, const void* src,
size_t count);
```

Operazione generalmente **sincrona**.

👉 ***Nota***: Se tutti i thread in un warp leggono lo **stesso indirizzo**, fa **boradcast** (prestazioni elevate). In realtà esegue comunque broadcast anche se gli indirizzi sono diversi ⇒ accessi serializzati e spreco di banda.

📖 Memoria Texture

La **Memoria Texture**:

- memoria **read-only off-chip** (DRAM).
- Accessibile a **tutti** i thread .
- Supportata da **texture cache** (8-12 KB per SM) on-chip (ottimizza i pattern d'accesso).
- Supporto hw a **filtraggio** ed **interpolazione** in virgola mobile durante la lettura dei dati.
- Ottimizzata per località spaziale 2D (accessi regolari a dati organizzati come matrici o immagini).
- Va usata con Texture Objects o variabili di tipo `texture` e relative funzioni.

👉 ***Nota*** : In assenza di località spaziale, l'uso della texture memory può risultare meno efficiente della global memory (perché si basa su pattern spaziali prevedibili).

📖 Memoria Globale

La **Memoria Globale**:

- Memoria **off-chip** (**DRAM** ⇒ grande (GB), con latenza alta).
- Accessibile a **transazioni da 32/64/128 Byte**.
- Scope e lifetime **globale**.
- Accessibile da **tutti** i thread.
- L'**allocazione** é
  - **Statica** con qualificatore `__device__` nel codice **device**.
  - **Dinamica** se allocata (**host**) con `cudaMalloc` e liberata con `cudaFree` .

👉 ***Nota*** : Per migliorare le prestazioni é bene sfruttare:
- **coalescenza**, ossia raggruppare gli accessi di thread adiacenti ad indirizzi contigui.
- **allineamento**, indirizzi di memoria allineati alla dimensione delle transazioni (32, 64, 128 B).

La memoria globale sarà ulteriormente approfondita in seguito (Sezione 2.4).

📖 Cache (GPU)

Le **cache** GPU, come quelle CPU, sono memorie **on chip non programmabili** cruciali per accelerare l'accesso ai dati. La cache viene usata per m emorizzare **temporaneamente** porzioni della **memoria principale** per accessi più veloci.

Tipi di cache:

- **L1**:
  - Più **veloce**.
  - **1×SM**.
  - Memorizza dati dalla memoria locale e globale (inclusi register-spills).
- **L2**:
  - **Unica** e **condivisa** tra SM.
  - Ponte tra L1 (più veloce) e memoria principale.
  - Memorizza dati dalla memoria locale e globale (inclusi register-spills).
- **Constant**:
  - **1×SM**.
  - Memorizza dati che non cambiano.
- **Texture**:
  - Specializzata per **dati texture** (2D/3D).
  - Supporto interpolazione e filtraggio.
  - Unificata nella L1 (nelle ultime architetture).

👉 ***Nota*** : Su alcune GPU, è possibile configurare se i dati vengono cachati solo sia in L1 che L2, o solo in L2

⚠️ Recap Caratteristiche Principali Memorie

Memoria	On/Off Chip	Cached	Accesso	Scope	Durata
Registro	On	n/a	R/W	Thread	Thread
Condivisa	On	n/a	R/W	Thread nel Blocco	Blocco
Locale	Off	†	R/W	Thread	Thread
Globale	Off	†	R/W	Tutti i Thread + Host	Allocazione Host
Costante	Off	Sì	R	Tutti i Thread + Host	Allocazione Host
Texture	Off	Sì	R	Tutti i Thread + Host	Allocazione Host

† In cache solo su dispositivi con compute capability 2.x+

⚠️ Recap Dichiarazione Variabili e Memorie (CUDA)

Memoria	On/Off Chip	Cached	Accesso	Scope	Durata
Registro	On	n/a	R/W	Thread	Thread
Condivisa	On	n/a	R/W	Thread nel Blocco	Blocco
Locale	Off	†	R/W	Thread	Thread
Globale	Off	†	R/W	Tutti i Thread + Host	Allocazione Host
Costante	Off	Sì	R	Tutti i Thread + Host	Allocazione Host
Texture	Off	Sì	R	Tutti i Thread + Host	Allocazione Host

† In cache solo su dispositivi con compute capability 2.x+

2.3.3. Gestione della Memoria Host-Device

Come in C, il programmatore deve **allocare** e **deallocare memoria** manualmente. In più, è necessario **gestire esplicitamente il trasferimento dei dati tra host e device**.

Le principali funzioni per allocare, modificare e deallocare memoria con CUDA sono già stati trattate in una *sezione del capitolo introduttivo* (e non saranno ripetute).

👉 ***Nota*** : Ricordando che i puntatori memoria host non possono essere usati nello spazio device e viceversa.

La memoria GDDR (GPU) offre un larghezza di banda teorica molto alta (2-3TB/s per HBM), ma il bus PCIe ha una larghezza di banda teorica massima molto minore (64GB/s per PCIe x16 5.0). Quindi i trasferimenti di dati tra host e device possono rappresentare facilmente un collo di bottiglia, pertanto é necessario ridurli al minimo.

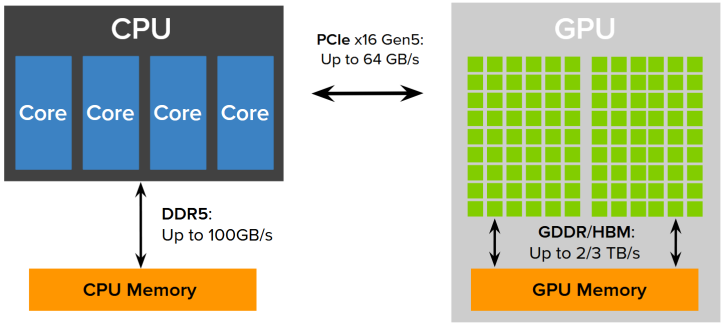


Figura 40: Banda per trasferimenti CPU-GPU.

La memoria allocata dall'host di default é **pageable**, ovvero soggetta a page fault. Per questo motivo, la GPU non può accedervi in modo sicuro (manca un controllo sui page fault).

Normalmente per un trasferimento da memoria pageable, il **driver CUDA** alloca temporaneamente memoria host pinned, quindi bloccata in RAM (page-locked o pinned), successivamente vengono **copiati** i dati dalla memoria host sorgente alla memoria pinned, infine i dati sono **trasferiti** dalla pinned alla memoria device (possibili problemi di inefficienza).

📖 **Memoria Pinned**

La **Memoria Pinned** viene allocata tramite `cudaMallocHost()` in memoria **host page-locked**, accessibile direttamente al device (Direct Memory Access - DMA). É quindi disponibile una **maggiore larghezza di banda** (rispetto alla memoria pageable) ed eliminando la necessità di copie intermedie.

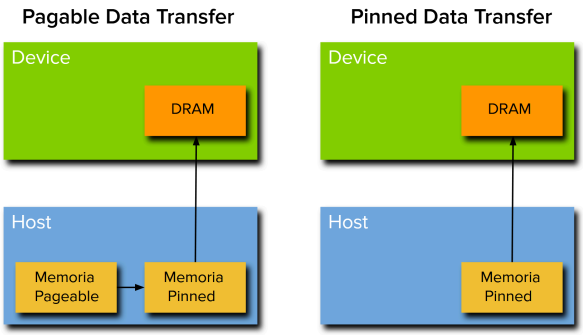


Figura 41: Trasferimento di dati - Pageable VS Pinned

👉 ***Nota***: Allocare troppa memoria pinned può degradare le prestazioni del sistema host.

2.3.3.1. CUDA Malloc Host

La `cudaMallocHost` alloca **memoria host pinned** (page-locked), che non può essere spostata dal sistema operativo (l'indirizzo fisico non cambierà) e permette trasferimenti dati ed elaborazione asincrona tra host e device.

```
cudaError_t cudaMallocHost(void **devPtr, size_t size);
```

Ad ogni allocazione va corrisposta la deallocazione con `cudaFreeHost`.

```
__host__ cudaError_t cudaFreeHost ( void* ptr );
```

📖 Memoria Zero-Copy

La **memoria Zero-Copy** è una tecnica che consente al device di **accedere direttamente** alla **memoria** dell'host e viceversa **senza** la necessità di **copiare** esplicitamente i dati tra le due memorie. È un'eccezione alla regola che l'host non può accedere direttamente alle variabili del device e viceversa.
Gli accessi avvengono **direttamente** tramite **PCIe**, con trasferimenti impliciti quando richiesti.
La memoria zero-copy é **memoria pinned** dell'host mappata nello spazio di indirizzamento del device.

👉 ***Nota***: Questa tecnica consente di utilizzare la memoria host quando quella del device é insufficiente.

👉 ***Nota***: Gli accessi alla memoria devono essere **sincronizzati** tra host e device per evitare comportamenti indefiniti.

Per creare una regione di memoria zero-copy:

```
cudaError_t cudaHostAlloc(void **pHost, size_t size, unsigned int flags);
```

dove `flags` può essere:

- `cudaHostAllocDefault` = comportamento identico a `cudaMallocHost()` (memoria pinned).
- `cudaHostAllocProtable` = memoria pinned utilizzabile da tutti i contesti CUDA.
  Normalmente la memoria pinned é legata al contesto CUDA che l'ha creata, così può essere usata da tutti i contesti/GPU presenti nel sistema.
- `cudaHostAllocWriteCombined` = memoria write-combined per trasferimenti PCIe più rapidi (dati non CPU cached).
- **`cudaHostAllocMapped`** = memoria dell'host mappata nello spazio di indirizzo del device (memoria zero-copy).

Per ottenere il puntatore device per la memoria pinned:

```
cudaError_t cudaHostGetDevicePointer(void **pDevice, void *pHost, unsigned int
flags);
```

Prende l'indirizzo host (`pHost`) e restituisce un indirizzo (`pDevice`) che può essere passato direttamente ai kernel.

📖 Unified Virtual Addressing (UVA) *[CUDA 4.0+, CC 2.0+]*

La **Unified Virtual Addressing** (UVA), o **Indirizzamento Virtuale Unificato**, è una tecnica che permette alla CPU e alla GPU di **condividere** lo stesso **spazio di indirizzamento virtuale** (la memoria fisica rimane distinta).
É il sistema di **runtime** di CUDA **gestisce automaticamente la mappatura** degli indirizzi virtuali agli indirizzi fisici nella memoria della CPU o della GPU.

👉 ***Nota***: **Non** vi è **distinzione** tra un puntatore virtuale host e uno device, tuttavia non è possibile dereferenziare un puntatore host sul device o viceversa (fa eccezione la memoria zero-copy).

👉 ***Nota***: Il parametro `cudaMemcpyKind` di `cudaMemcpy` diventa obsoleto e può essere impostato su `cudaMemcpyDefault`.

👉 ***Nota***: La memoria host zero-copy allocata con `cudaHostAlloc` ha puntatori host e device identici (non più necessario l'uso di `cudaHostGetDevicePointer`).

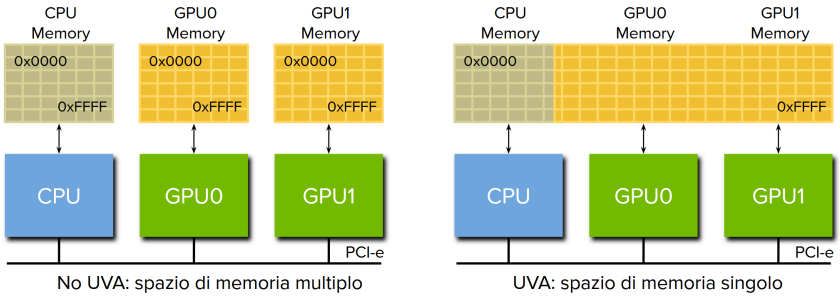


Figura 42: Indirizzamento con e senza UVA.

📖 Unified Memory (UM) *[CUDA 6.0+]*

La **Unified Memory** fornisce uno **spazio di memoria virtuale unificato** a 49 bit che permette di accedere agli stessi dati da tutti i processori del sistema usando un unico puntatore (Single-pointer-to-data).
La memoria è gestita **automaticamente** e **dinamicamente** dal **CUDA Runtime** tramite il **Page Migration Engine**, che trasferisce i dati tra host e device tramite PCIe o NVLink quando necessario (**migrazione automatica dei dati**).
Quando le GPU o CPU accedono a dati non residenti localmente, un **page faul** avvia il trasferimento automatico dei dati, gestito in modo **trasparente** dall'applicazione.

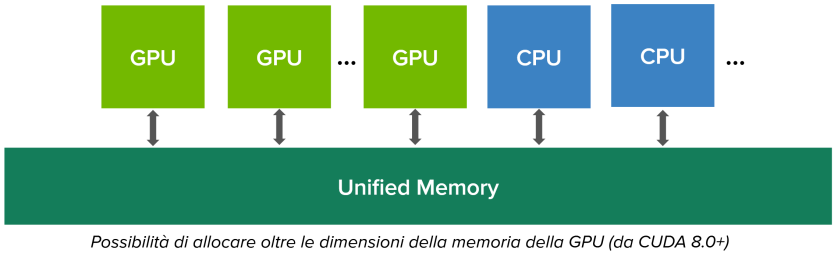


Figura 43: Spazio virtuale con Unified Memory.

📖 Memoria Gestita in UM

La **Memoria Gestita** (o **Managed Memory**) si riferisce alle **allocazioni** di **Unified Memory** che sono gestite automaticamente dal sistema sottostante e sono interoperabili con le allocazioni specifiche del device.
Si alloca staticamente con l'annotazione `__managed__` e dinamicamente con `cudaMallocManaged`:

```
cudaError_t cudaMallocManaged(void **devPtr, size_t size, unsigned int
flags=0);
```

👉 ***Nota*** : Il puntatore `devPtr` è valido e dereferenziabile sia sull'host che sui device.

Con la managed Memory e UM:
- Il sistema migra automaticamente le pagine di memoria tra host e device.
- Rimangono **compatibili** le **allocazioni specifiche del device** (es. cudaMalloc), consentendo di utilizzare entrambi i tipi di memoria all'interno dello stesso kernel (managed e unmanaged).
- Le operazioni CUDA valide per la memoria del device funzionano anche con la Memoria Gestita.

`cudaMallocManaged` alloca spazio virtuale. Le pagine fisiche vengono allocate in modo **lazy**, solo al primo utilizzo da parte di CPU o GPU, evitando allocazioni fisiche non necessarie a priori.
Il driver CUDA **migra dinamicamente** le pagine tra CPU e GPU in base agli accessi, usando euristiche (accessi recenti, device corrente, comportamento osservato) per mantenere la località e ridurre i page fault .
Se la CPU tenta di accedere al puntatore subito dopo l'allocazione (managed), riceve page fault, quindi il SO alloca la pagina in RAM. Se la GPU tenta di accedere al puntatore, riceve page fault, quindi il driver CUDA migra la pagina dalla RAM alla VRAM.

👉 ***Nota*** : La gestione autoamtica della memoria manged introduce **overhead** ed **imprevedibilità**, inoltre i page fault e le migrazioni dinaimche possono degradare le prestazioni (rispetto a trasferimenti espliciti e ben pianificati).

👉 ***Nota*** : Esiste la possibilità, per gli sviluppatori, di guidare il comportamento del driver usando:

- `cudaMemAdvice()` per suggerimenti su come accedere ai dati (lettura/scrittura), sulla loro posizione preferenziale e sul device principale che li utilizzerà (non innesca il trasferimento).
- `cudaMemPrefetchAsync()`, per migrare proattivamente i dati nella memoria del device target, riducendo page fault e preparando i dati prima dell'elaborazione (non garantisce rimangano dove richiesti).

[Pascal-]

L' **accesso** alla memoria unificata era **mutualmente esclusivo** tra GPU e CPU, in particolare la CPU non poteva accedere finchè il kernel avesse terminato il proprio lavoro. Se la CPU accede mentre un kernel é in esecuzione riceve segmentation fault o causa il crash dell'applicazione.
É necessario quindi evitare conflitti di accesso tra CPU e GPU ⇒ `cudaDeviceSynchronize()` per sospendere la CPU fino al completamento dei kernel GPU.

[Pascal+]

L' accesso diventa concorrente e possibile senza crash, ma `cudaDeviceSynchronize()` rimane necessario per **evitare race condition**.

2.4. Global Memory

Finora abbiamo detto che le istruzioni e le operazioni di memoria sono emesse ed eseguite per warp (32 thread) sulla base degli indirizzi di memoria delle operazioni stesse e dei tipi di dato trattati.

👉 ***Nota*** : Ogni richiesta é servita da **una o più transazioni** di memoria e ogni transazione é **indivisibile**.

I pattern di accesso basati sulla distribuzione degli indirizzi possono aiutare a mitigare la limitatezza della larghezza di banda.

Abbiamo trattato la *memoria globale* in precedenza, adesso aggiungiamo qualche dettaglio.

Tutti gli accessi alla memoria globale passano per la cache **L2** e molti passano anche per la **L1** (a seconda di tipo di accesso, architettura e configurazioni). Come detto in precedenza, le transizioni di memoria avvengono a blocchi di 32, 64 o 128 byte.

Accessi Allineati alla Memoria

Gli accessi alla memoria sono **allineati** se l'**indirizzo iniziale** di una transazione é **multiplo** della **dimensione della transazione** stessa.

👉 ***Nota***: La memoria allocata tramite CUDA Runtime API (come `cudaMalloc()`) è garantita essere allineata ad almeno 256 byte.

Accessi Coalescenti alla Memoria

Gli accessi alla memoria sono **coalescenti** se **tutti** i 32 thread in un warp accedono a un **blocco contiguo** di memoria, quindi l' hardware può **combinarli** in un **numero ridotto** di **transazioni** verso posizioni consecutive nella DRAM.

Accessi Allineati e Coalescenti alla Memoria

Gli accessi alla memoria sono **allineati e coalescenti** se un warp accede ad un **blocco contiguo** di memoria partendo da un **indirizzo allineato**.

👉 ***Nota***: Questo approccio ottimizza il **throughput** della **memoria** globale e **migliora** le **prestazioni** complessive del kernel. Quest oapproccio é fondamentale per ottenere le massime prestazioni (ma non sufficiente e/o sempre praticabile).

I passaggi per la cache L1 sono influenzati da CC e opzioni del compilatore (`nvcc`). Nello specifico:

- Per la cache L1:
  - CC 2.X → L1 disabilitata di default.
  - CC 3.5-5.2 → L1 disabilitata di default, usata solo per register spills.
  - CC 6.0+ → L1 abilitata di default su varianti HPC e abilitabile di default su GPU consumer da CC 6.1+.
- Per il compilatore:
  - abilitata con `-Xptxas -dlcm=ca` ⇒ le richieste passano per L1, poi per L2 e infine in DRAM (miss). Le transazioni minime sono da 128B per CC 5.2-, da 32B altrimenti.
  - disabilitata con `-Xptxas -dlcm=cg` ⇒ gli accessi passano prima per L2, poi per DRAMe le transazioni di memoria sono da 32B (ideale per accessi sparsi).

La lettura dalla Memoria Globale prevede di scorrere le memorie (abilitate) dalla più veloce alla meno, fermandosi in caso di **cache hit**, ossia di corrispondenza trovata in cache, proseguendo in caso di **cache miss**.
Il flusso é il seguente:

$$\text{L1 Cache (se abilitata)} \rightarrow \text{L2 Cache} \rightarrow \text{DRAM}$$

Se il dato non é presente nemmeno in DRAM, viene richiesto alla memoria di sistema. Ogni **accesso aggiuntivo** nella gerarchia di memoria **rallenta** le **prestazioni** e aumenta il consumo.

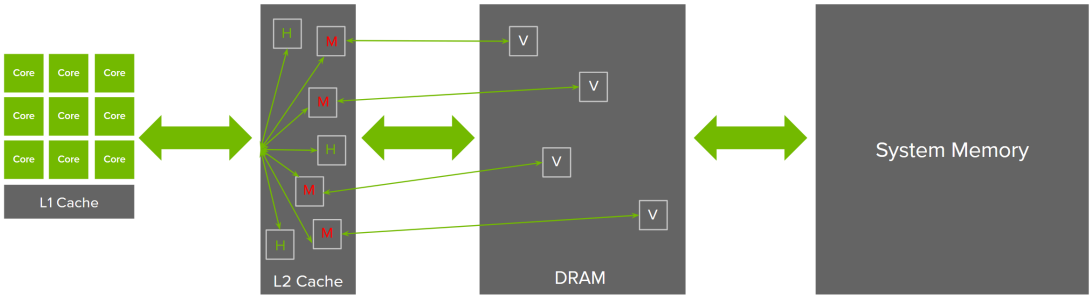


Figura 44: Flusso degli accessi alla gerarchia di memoria.

La gestione delle transazioni si é evoluta nel tempo.

[CC 5.2-] - Architetture Legacy

Con L1 abilitata (**cached load**), in caso di miss (su L1) veniva caricata l'intera linea da 128B. Questo, per accessi sparsi, causa un alto over-fetch e spreco di banda.
In caso L1 disabilitata (**uncached load**), invece, questa veniva semplicemente **bypassata** e venivano usate transazioni da 32B.

[CC 6.0+] - Architetture Moderne

Con la **Sector-Based Cache**, la cache L1 viene organizzata in **settori** da 32B. Essendo ogni linea di 128B, si hanno quindi 4 settori. Il funzionamento prevede il validamento unicamente dei settori (32B) richiesti, riducendo lo spreco.
L'API `cudaLimitMaxL2FetchGranularity` fornisce un hint al Memory Controller della GPU per ottimizzare il fetch L2, indipendente dalla struttura sector-based (32 byte) della cache L1. In pratica il caricamento in caso di miss in L2 può essere reso più o meno aggressivo.

Il pattern di accesso ai caricamenti dalla memoria può essere caratterizzato da combinazioni dei seguenti accoppiamenti:

- **Cache Load** : cache L1 abilitata.
- **Uncached Load** : bypass della L1 in favore della L2 direttamente.

- Accesso **allineato** : primo indirizzo di accesso multiplo della dimensione della transazione
- Accesso **disallineato** se non é allineato.

- Accesso **coalescente** : un warp accede a un blocco contiguo di dati.
- **Non coalescente** se non é coalescente.

Analizziamo ora diversi casi.

Cached Loads Allineato e Coalescente
Questo si configura come il caso **ideale** di accesso alla memoria. Le **transazioni** sono **da 128B** e tutti gli indirizzi richiesti dai thread di un warp ricadono in un'**unica linea di cache** ($128B = 32 \cdot 4B$).
⇒ Utilizzo bus **100%**. ✅
⇒ **Nessun dato inutilizzato** nella transazione. ✅

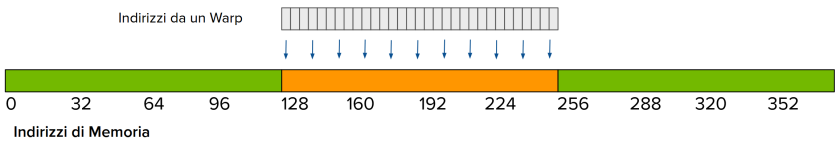


Figura 45: Accesso allineato e coalescente con cache load (CC 5.2-).

👉 ***Nota*** : Questo comportamento vale per CC 5.2-. Per CC 6.0+ le transazioni minime sono da 32 byte.

👉 ***Nota*** : Il caso in questione sottintente l'uso/richiesta di dati da 4B (per thread). Nel caso il tipo di dato richiesto fosse di dimensione inferiore si "sprecherebbe" comunque una transazione da 128B se CC < 5.2, mentre con la sector-based cache (CC 6.0+), si potrebbe eseguire una transazione anche da 32B o 64B (complessiva). Al contrario, se il dato avesse dimensioni maggiori di 4B, occorrerebbero più transazioni, per cui valgono gli stessi ragionamenti appena esposti. L'efficienza del 100% si mantiene solo in caso di saturazione completa di ogni transazione con dati "utili" (⇒ transazione multipla della dimensione complessiva dei dati richiesti dal warp).

Cached Loads Allineato ad indirizzi randomizzati

In questo caso consideriamo accessi in memoria contigui ma non consecutivi rispetto al thread ID e distribuiti randomicamente, ma comunque confinati in una singola cache line (128B ⇒ unica transazione).
⇒ Utilizzo bus **100%**. ✅
⇒ **Nessun dato inutilizzato** nella transazione. ✅

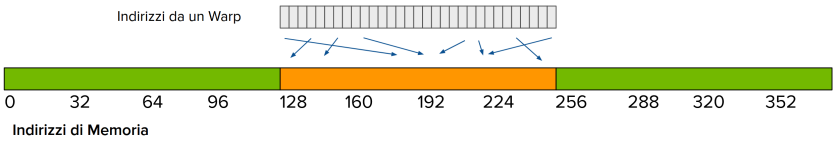


Figura 46: Accesso allineato ad indirizzi random con cache load (CC 5.2-).

👉 ***Nota*** : Valgono le medesime considrazioni del primo caso (*Nota "transazioni e CC"* e *Nota "dati < o > di 4B"*).

Cached Loads Disallineato

I thread del warp accedono a 32 elementi consecutivi ma non allineati (primo indirizzo non multiplo di 128B), pertanto gli indirizzi richiesti si estendono su due segmenti (da 128B) di memoria globale.
⇒ Necessità di più transazioni di memoria (2) e **metà** dei **byte** caricati **non verranno usati**. ☹
⇒ Utilizzo bus **50%**. ☹

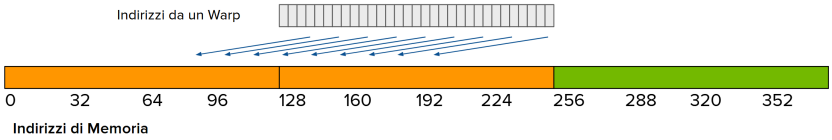


Figura 47: Accesso disallineato con cache load (CC 5.2).

👉 ***Nota*** : Valgono le medesime considrazioni del primo caso (*Nota "transazioni e CC"* e *Nota "dati < o > di 4B"*). Ovviamente in questo caso l'efficienza, soprattutto senza sector-based cache, sarà notevolmente ridotta.

Cached Loads Accesso allo Stesso Indirizzo

I thread del warp accedono allo stesso indirizzo di memoria (ovviamente ricade in una singola line cache).
⇒ **Unica transazione** di memoria (128B). ☹
Ad esempio con un intero (4B) si ha un utilizzo del ∼ 3%.
⇒ **Bassissimo** utilizzo del **bus**. ❌

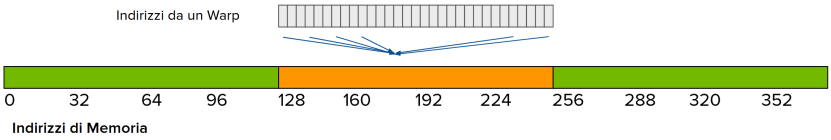


Figura 48: Accesso allo stesso indirizzo con cache load (CC 5.2).

👉 ***Nota*** : Valgono le medesime considrazioni del primo caso (*Nota "transazioni e CC"* e *Nota "dati < o > di 4B"*). Anche in questo caso ovviamente l'efficienza, soprattutto senza sector-based cache, sarà notevolmente ridotta.

Cached Loads con Accessi Sparsi
I thread del warp accedono ad indirizzi che possono estendersi su $\boldsymbol{N}$ **cache lines**, con $0 < N < 32$.
Se i thread leggono con passi non contigui (**stride** > 1) o indirizzi casuali, gli accessi diventano sparsi.
$\Rightarrow$ $\boldsymbol{N}$ **transazioni** di memoria (da 128B). ❌
$\Rightarrow$ **Basso** utilizzo del **bus** ($\frac{128B}{N \cdot 128B}$). ❌
👉 ***Nota*** : **Caso peggiore** con $N = 32$ indirizzi sparsi (scattered) richiesti nella memoria globale.

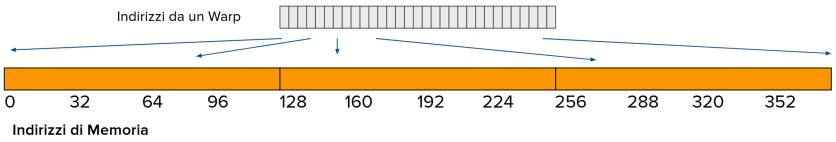


Figura 49: Accesso ad indirizzi sparsi con cached loads (CC 5.2).

👉 ***Nota*** : Valgono le medesime considrazioni del primo caso (*Nota "transazioni e CC"* e *Nota "dati < o > di 4B"*). Anche in questo caso ovviamente l'efficienza, soprattutto senza sector-based cache, sarà notevolmente ridotta.

Uncached Loads Allineato e Coalescente
La granularità dei segmenti di memoria diventa a 32B (cache L1 disabilitata).
I thread accedono in maniera allineata (primo indirizzo multiplo di 32B) e coalescente (dati contigui).
$\Rightarrow$ Richiesti **4 segmenti** da 32B $\Rightarrow$ transazioni da 32B. ✅
$\Rightarrow$ Utilizzo del bus **100%**. ✅

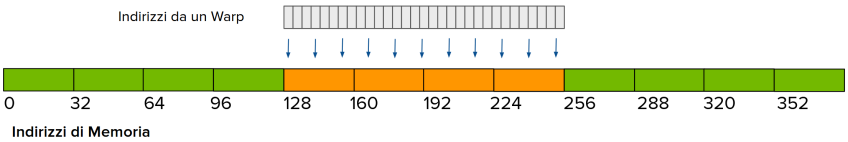


Figura 50: Accesso allineato e coalescente con uncached loads.

👉 ***Nota*** : Da Pascal (CC $\geq$ 6.0), i global loads usano transazioni minime da 32 byte, con o senza cache L1.

👉 ***Nota*** : Il caso in questione sottintente l'uso/richiesta di dati da 4B (per thread). Nel caso il tipo di dato richiesto fosse di dimensione differente, andrebbe proporzionato il numero di transazioni (sempre almeno da 32B). Eventualmente l'uso efficiente del bus potrebbe scendere.

Uncached Loads Allineato e con Indirizzi Randomizzati
I thread accedono in maniera allineata (primo indirizzo multiplo di 32B) ad indirizzi randomizzati in un range da 128B.
$\Rightarrow$ Indirizzi in **4 segmenti** (consecutivi) $\Rightarrow$ 4 transazioni da 32B. ✅
$\Rightarrow$ Utilizzo bus **100%**. ✅

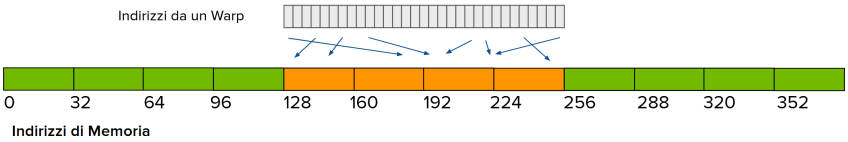


Figura 51: Accesso allineato con indirizzi randomizzati in 4 segmenti consecutivi con uncached loads.

👉 ***Nota*** : Valgono le medesime considrazioni del primo caso "uncached" (*Nota "transazioni e CC 6.0"* e *Nota "dati < o > di 4B"*).

Uncached Loads Disallineato ma Coalescente
Il warp effettua un caricamento non allineato richiedento 32 elementi consecutivi, che ricadono in massimo 5 segmenti.
$\Rightarrow$ Utilizzo del bus pari all'**80%** (128B letti vs 160B caricati). 😀

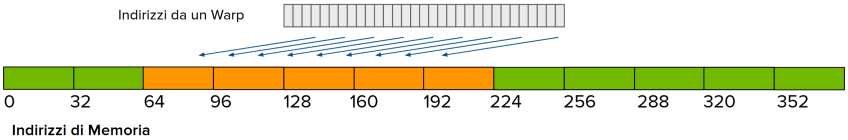


Figura 52: Accesso disallineato ma coalescente con uncached loads.

👉 ***Nota*** : I caricamenti "fine-grained" a 32 byte riducono lo spreco di banda rispetto ai cached loads da 128 byte su accessi disallineati/non coalescenti (80% vs 50% di utilizzo).

👉 ***Nota*** : Valgono le medesime considrazioni del primo caso "uncached" (*Nota "transazioni e CC 6.0"* e *Nota "dati < o > di 4B"*).

Uncached Loads Accesso allo Stesso Indirizzo
Tutti i thread in un warp richiedono lo stesso indirizzo di memoria.
⇒ ricade in un **unico segmento** da 32B.
⇒ Utilizzo del bus **12.5%** (4B richiesti vs 32B letti). ☹

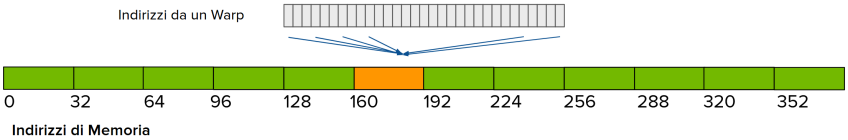


Figura 53: Accesso disallineato ma coalescente con uncached loads.

👉 ***Nota*** : Minor spreco di banda grazie ai caricamenti "fine-grained" a 32B, rispetto ai cached loads da 128B .

👉 ***Nota*** : Valgono le medesime considrazioni del primo caso "uncached" (*Nota "transazioni e CC 6.0"* e *Nota "dati < o > di 4B"*).

Uncached Loads e Accessi Sparsi
Il caso peggiore é che i thread richiedano ciascuno un diverso indirizzo sparso (scattered), quindi 32 indirizzi differenti.
⇒ fino a $\boldsymbol{N = 32}$ **transazioni** diverse. ❌
⇒ **Alto spreco** del bus ($32 \times 32B = 1024B$ letti vs 128 richiesti). ❌

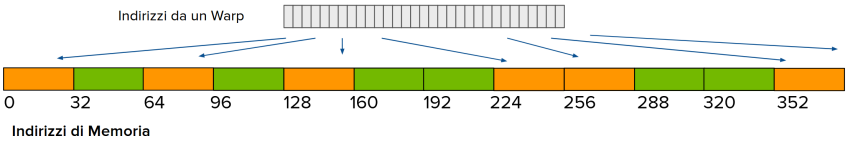


Figura 54: Accessi sparsi con uncached loads.

👉 ***Nota*** : Valgono le medesime considrazioni del primo caso "uncached" (*Nota "transazioni e CC 6.0"* e *Nota "dati < o > di 4B"*).

Le scritture verso la memoria globale hanno un comportamento differente.
Prima di volta le scritture bypassavano la cache L1, usando solo la L2.
Da Volta, viene usata la cache L1 in modalità **write-through** , che prevede la **scrittura simultanea** in cahce **L1** ed **L2** e **successivamente** nella **DRAM**.

Le **scritture** vengono eseguite a **livello di segmenti** , quindi hanno **granularità 32B** . Le **transazioni** di memoria , come detto, possono coinvolgere **1, 2 o 4 segmenti alla volta** (32B, 64B o 128B). Quindi nel caso si debbano scrivere più segmenti ad indirizzi allineati e in modo coalescente, é possibile farlo in meno transazioni, a patto che siano a "blocchi" di 1, 2 o 4 segmenti.
Intendo dire che, ad esmpio, nel caso in cui si vogliano scrivere 128B, quindi 4 segmenti, é possibile farlo con un'unica transazione di memoria da 128B, analogamente con 64B (se la scrittura é allineata e coalescente).

La scrittura ad indirizzi scattered funziona analogamente alla lettura (v. *Esempio su accessi sparsi*).

2.4.1. Shared Memory (SMEM)

In precedenza (v. *Definizione di Shared Memory*) abbiamo detto che la SMEM:

- É una **cache gestibile dal programmatore** per i dati dalla memoria globale.
- Analoga alla L1 in termini di prestazioni (e solitamente sullo stesso chip).
- Funge da **canale di comunicazione** per tutti i thread di un blocco.
- Tipicamente usata come memoria **scratch pad** (temporanea) per elaborare dati on-chip e migliorare i pattern di accesso alla global memory. Quindi permette di aumentare la banda disponibile e ridurre la latenza.

Good practice ✅ - Uso per:

- **thread diversi** dello stesso blocco accedono agli **stessi dati** (vicinato, stencil, tile di matrici…).
- **stesso dato** (residente in global memory) viene letto più volte durante l'elaborazione.
- **comunicazione/condivisione** tra thread dello stesso blocco.

Ad ogni **blocco di thread** viene **allocata una quantià fissa di SMEM** all'inizio della sua esecuzione. Questo spazio, **condiviso** da tutti i thread del blocco, rimane dedicato al blocco **per tutto il suo ciclo di vita** nell'SM.
Gli accessi alla SMEM avvengono per warp. Nel caso **migliore** con un'**unica transazione**, alla **peggio** con **32** (una per thread).

👉 ***Nota***: La SMEM é **gestita esplicitamente**, ma é una **risorsa limitata** e condivisa tra tutti i blocchi attivi di un SM. Un suo uso **eccessivo** può **limitare** il **parallelismo** del device (perché l'SM non possiede sufficienti risorse da allocare ad altri blocchi).

Tipicamente il flusso di utlizzo di SMEM prevede:

1. Caricamento dei dati Global → Shared.
2. Sincronizzazione post caricament con `__syncthreads()` per garantire il completamento del caricamento die dati.
3. Elaborazione dei dati (con possibile riutilizzo di dati in shared tra thread del blocco).
4. Eventuale sincronizzazione post eleaborazione (`__syncthreads()`), da usare solo se i risultati elaborati da un thread sono utilizzati da altri thread dello stesso blocco.
5. Scrittura dei risultati Shared → Global.

Metodi di allocazione:

- **Statica** (dimensione da allocare nota alla compilazione)
  - Keyword `__shared__`.
  - Visibilità **globale** se dichiarata fuori da qualsiasi kernel.
  - Visibilità **locale** se dichiarata nel kernel.
  - Supporta array 1D, 2D, 3D.
- **Dinamica** se la quantità di SMEM é specificata nella configurazione di lancio del kernel come terzo argomento (prima dell'esecuzione).
  - Keyword `extern __shared__`.
  - Supporta solo arra 1D.

  👉 ***Nota***: La quantià passata al kernel viene allocata per ogni blocco.

Nella *Definizione di Shared Memory* abbiamo detto che la SMEM é suddivisa in banks, questo per consentire accessi simulantei. Adesso approfondiamo questo aspetto.

La SMEM é organizzata **32 moduli** di memoria (**bank**) di uguale dimensione. Ogni bank (contiene e) può servire una **word** (4B o 8B, in base all'architettura). Lo **spazio di indirizzamento** della SMEM é **1D** e viene mappato fisicamente sui banchi in base alla CC.

🤔 ***Why 32?*** : Perché corrisponde al numero di thread di un warp, permettendo l'**accesso simultaneo** ad una bank per ogni thread (idealmente).

Nello scenario ideale un'operazione di lettura o scrittura (load/store) emessa da un warp accede ad un solo indirizzo per ogni banco, l'operazione è servita da una singola transazione di memoria. Mentre, in quello non ottimale, un'operazione accede a più indirizzi nello stesso banco, sono necessarie più transazioni di memoria, riducendo l'utilizzo della banda larga.

L' indirizzamento è **sequenziale**, ossia word consecutive sono mappate su banchi consecutivi, mentre i banchi stessi sono accessibili **parallelamente**.

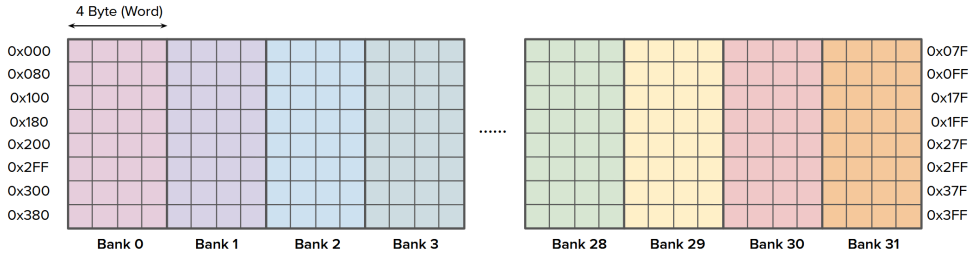


Figura 55: SMEM banks (caso semplicistico di 1KB di memoria con word da 4B).

Ogni banco ha **larghezza di banda** dipendende dalla sua CC: **4B** (32b) oppure **8B** (64b). Tale dimensione stabilisce quali indirizzi appartengano a quale bank e anche la larghezza di banda complessiva massima.

Nel caso di banchi da 4B (tipicamente architetture più recenti) é possibile trasferire 4B × 32 banchi (nel caso ideale).
Per calcolare l'indice del banco:

$$\text{indice banco} = (\text{indirizzo byte} \div 4 \text{ byte/banco}) \ \% \ 32 \text{ banchi}$$

I banchi, di fatto, posseggono una **doppia mappatura**, che mette in realzione indirizzo (byte), indice della word e indice del banco. Ogni 32 word, la mappatura dei bank si ripete ciclicamente.

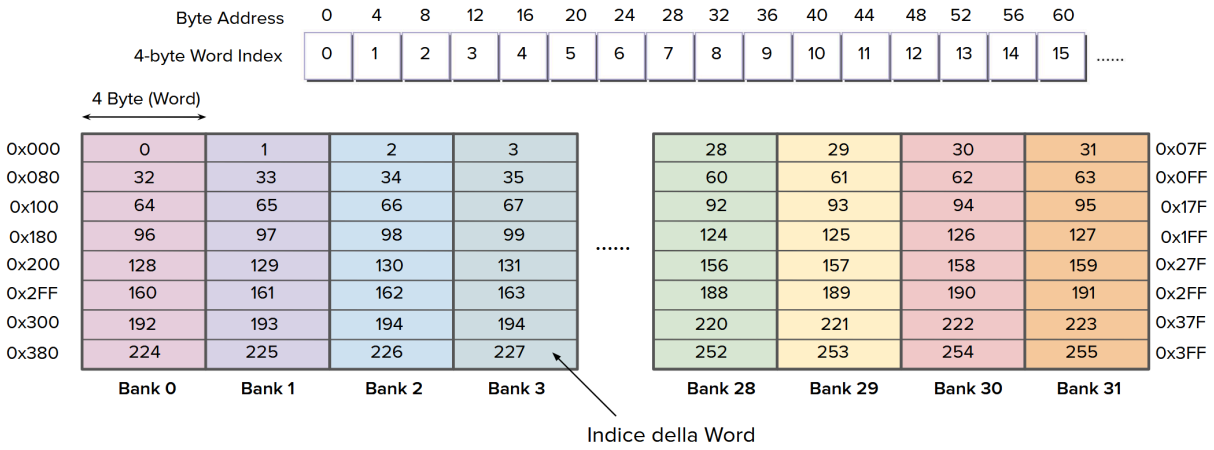


Figura 56: Doppia mappatura nella SMEM (word da 4B).

La SMEM può essere inizilizzata sia a **byte** che ad intere **word**.

Quando si alloca un array, ad esempio di float (1 float = 4B), in SMEM, i vari elementi si dispongono uno per bank consecutivamente allo stesso modo dell'indice delle word (`arr[0]` al Bank 0, `arr[1]` al Bank 1, fino a `arr[31]` al Bank 31, poi `arr[32]` torna al Bank 0).

📖 Bank Conflict

Un **bank conflict** si verifica quando **più thread** di un warp accedono ad **indirizzi diversi nello stesso memory bank**. L'hardware **divide** una **richiesta** con conflitto in più transazioni separate (**conflict-free**), riducendo la banda proporzionalmente al numero di transazioni necessarie.

👉 ***Nota***: Nessun conflitto tra thread di blocchi diversi, ma solo a livello di warp dello stesso blocco (in quanto la SMEM é allocata ai singoli blocchi).

Quindi l'accesso effettivo dei thread di un warp può essere:

- **Parallelo** per indirizzi distribuiti su bank diversi (idealmente uno per ogni bank).
  ⇒ Singola transazione per tutti/più accessi. ✅
- **Seriale** in caso di indirizzi distribuiti nello stesso bank.
  ⇒ Richiesta serializzata (N richieste, con $N \in [1, 32]$). 😐
  👉 ***Nota***: Anche in caso di conflitti prestazioni migliori rispetto alla L2 (o peggio).
- **Broadcast** se tutti gi thread leggono il medesimo (singolo) indirizzo.
  ⇒ Unica transazione (broadcast automatico). 😀
  👉 ***Nota***: Efficiente, ma basso uso della bandwidth.

Ogni banco ha dimensione 4B nelle architetture moderne e la SMEM é unificata alla L1 e configurabile dinamicamente. La capacità della SMEM dipende dalla CC, mentre la sua **configurazione** é gestita **autoamticamente dal driver CUDA**, ma é possibile forire suggerimenti per kernel specifici tramite:

```
cudaFuncSetAttribute(kernel_name,
cudaFuncAttributePreferredSharedMemoryCarveout, carveout);
```

dove `caveout` definisce la quantità di cache unificata da riservare alla SMEM per ogni kerne. Valori predefiniti sono `{cudaSharedmemCarveoutDefault(-1), cudaSharedmemCarveoutMaxL1(0), cudaSharedmemCarveoutMaxShared(100)}`. Se la percentuale richiesta non é supportata, é arrotondata al valore superiore disponibile.

Trade-off tra Memoria Condivisa e Cache L1

- La configurazione ottimale dipende dall'intensità di utilizzo della shared memory e della cache L1 nel kernel:
  - **Più Memoria Condivisa:**
    - Ideale quando i kernel fanno un uso intensivo della memoria condivisa per ridurre la latenza negli accessi alla memoria globale.
    - L'uso intensivo della shared memory può limitare l'occupancy (risorsa on-chip condivisa fra blocchi).
  - **Più Cache L1:**
    - Preferibile quando i kernel accedono frequentemente a dati globali con una buona località spaziale.
    - Utile per ridurre lo spilling dei registri nella memoria globale, migliorando le prestazioni complessive.

Differenze tra Memoria Condivisa e Cache L1

- Sebbene condividano lo **stesso hardware on-chip**, ci sono differenze fondamentali:
  - **Accesso:**
    - La memoria condivisa usa 32 banchi per l'accesso parallelo.
    - La cache L1 si basa su linee di cache (es. 128 byte) per il caricamento dei dati.
  - **Controllo:**
    - La memoria condivisa offre pieno controllo programmabile su cosa viene memorizzato e dove.
    - La cache L1 è gestita automaticamente dall'hardware, senza intervento del programmatore.

2.5. Librerie CUDA ed Applicazioni

2.5.1. Introduzione alle Librerie CUDA

Le **librerie CUDA** sono collezioni di **funzioni** e **routine** ottimizzate per GPU NVIDIA. Di fatto, sono parte integrante dell'ecosistema CUDA.

Le librerie CUDA si basano sul **runtime CUDA** e forniscono:

- **Interfaccie semplici**.
- **Funzionalità specifiche per dominio** (accesso ottimizzato a funzioni e algoritmi per settori specifici).

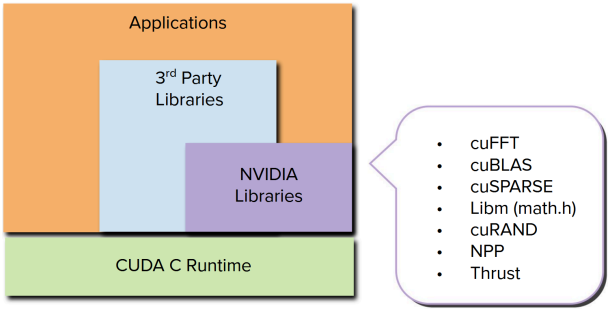


Figura 57: Librerie CUDA.

Introduzione alle Librerie CUDA

Caratteristiche Principali e Vantaggi

- **Ottimizzazione per GPU**:
  - Implementazioni altamente efficienti di algoritmi comuni (es. algebra lineare, trasformate di Fourier).
  - Sfruttano appieno le caratteristiche hardware specifiche delle GPU NVIDIA.
- **Astrazione della Complessità**
  - Offrono il miglior equilibrio tra facilità d'uso e prestazioni per molte applicazioni.
  - Nascondono i dettagli di basso livello della programmazione CUDA.
  - Forniscono API di alto livello per operazioni complesse.
- **Prestazioni Elevate**
  - Le librerie CUDA offrono prestazioni migliori rispetto alle librerie esclusive per host e spesso anche rispetto alle implementazioni custom CUDA.
  - Scalano efficacemente su diverse generazioni di GPU, garantendo compatibilità futura.
- **Versatilità**
  - Coprono una vasta gamma di domini applicativi, dalla matematica di base, all'elaborazione dei segnali digitali, all'intelligenza artificiale e molto altro.
- **Basso Carico di Manutenzione**
  - Le librerie CUDA riducono il carico di manutenzione per gli sviluppatori di software, poiché le implementazioni esistenti e mature sono già state testate e gestite da NVIDIA e dai suoi partner.

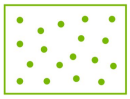
4

Librerie Matematiche in CUDA

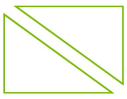
- Le **librerie matematiche** accelerate da GPU pongono le basi per **applicazioni ad alta intensità di calcolo** in aree come la dinamica molecolare, la fluidodinamica computazionale, la chimica computazionale, l'imaging medico, l'esplorazione sismica e così via.

cuBLAS
Libreria di algebra lineare di base (BLAS) accelerata da GPU.
Documentazione

cuFFT
Libreria accelerata da GPU per implementazioni della FFT
Documentazione



cuRAND
Generazione di numeri casuali accelerata da GPU.
Documentazione



cuSOLVER
Risolutori diretti densi e sparsi accelerati da GPU.
Documentazione

cuSPARSE
BLAS accelerato da GPU per matrici sparse.
Documentazione

cuDSS
Libreria di risolutori diretti sparsi accelerata da GPU.
Documentazione

CUDA Math API
API di funzioni matematiche standard accelerate da GPU.
Documentazione

cuTENSOR
Libreria di algebra lineare per tensori accelerata da GPU.
Documentazione

5

Librerie Quantisitche in CUDA

- Consentono di accelerare simulazioni e applicazioni avanzate nel **calcolo quantistico**.

cuQuantum

NVIDIA cuQuantum è un insieme di librerie altamente ottimizzate per accelerare le simulazioni di calcolo quantistico.

Documentazione

cuPQC

SDK di librerie ottimizzate per accelerare i flussi di lavoro di crittografia post-quantistica (PQC).

Documentazione

6

Librerie per Immagini e Video in CUDA

- Librerie accelerate da GPU per la **decodifica**, **codifica** ed **elaborazione di immagini e video** che utilizzano CUDA e componenti hardware specializzati delle GPU.

RAPIDS cuCIM

Accelera l'input/output (IO), la visione artificiale e l'elaborazione delle immagini di immagini n-dimensionali, in particolare immagini biomediche.

Documentazione

CV-CUDA

Libreria open-source per pre- e post-elaborazione ad alte prestazioni, accelerata da GPU, nei pipeline di visione AI.

Documentazione

NVIDIA DALI

Libreria portatile e open-source per la decodifica e l'augmented imaging di immagini e video per accelerare le applicazioni di deep learning.

Documentazione

nvJPEG

Libreria ad alte prestazioni accelerata da GPU per la decodifica JPEG.

Documentazione

NVIDIA Performance Primitives

Funzioni per l'elaborazione di immagini, video e segnali accelerate da GPU.

Documentazione

NVIDIA Video Codec

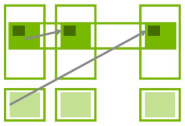
Codifica e decodifica video accelerate dall'hardware su Windows e Linux.

Documentazione

7

Librerie di Comunicazione in CUDA

- Primitive di comunicazione ottimizzate per le prestazioni su **multi-GPU** e **multi-nodo**.



NVSHMEM

Standard OpenSHMEM per la memoria GPU, con estensioni per migliorare le prestazioni sulle GPU.

Documentazione

NCCL

Libreria open-source per comunicazione rapida tra multi-GPU e multi-nodo che massimizza la larghezza di banda mantenendo bassa la latenza.

Documentazione

8

Librerie di Algoritmi Paralleli e di Litografia Computazionale

- **Librerie di Algoritmi Paralleli**: GPU-accelerate, queste librerie offrono algoritmi paralleli efficienti per operazioni in C++ e analisi di grafi in vari campi, come scienze naturali, logistica e pianificazione viaggi.
- **Libreria di Litografia Computazionale**: Progettata per affrontare le sfide della litografia computazionale su scala nanometrica.

Thrust

Libreria accelerata da GPU di algoritmi paralleli e strutture dati in C++.

Documentazione

cuLitho

Libreria con strumenti e algoritmi ottimizzati per accelerare la litografia computazionale e la produzione di semiconduttori utilizzando le GPU.

Documentazione

9

Librerie di Deep Learning in CUDA

- Librerie accelerate da GPU per **applicazioni di deep learning** che utilizzano CUDA e componenti hardware specializzati delle GPU.

TensorRT

Ottimizzatore e runtime ad alte prestazioni per l'inferenza di deep learning, pensato per il deployment in produzione.

Documentazione

cuDNN

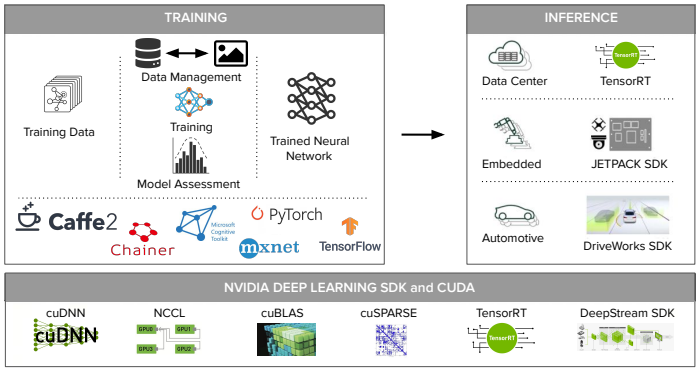
Libreria accelerata da GPU di primitive per deep neural network.

Documentazione

10

Librerie di Deep Learning in CUDA

- Librerie accelerate da GPU per **applicazioni di deep learning** che utilizzano CUDA e componenti hardware specializzati delle GPU.



11

2.5.2. Utilizzo delle Librerie CUDA

Workflow delle Librerie CUDA: Esempio con cuBLAS

Obiettivo

- cuBLAS (**CUDA Basic Linear Algebra Subprogram**): Libreria CUDA per operazioni di algebra lineare, come prodotti scalari, moltiplicazione di matrici e vettori, e prodotti di matrici, e altre operazioni di base.
- Queste operazioni sono utilizzate come blocchi costitutivi per problemi più complessi, come la risoluzione di sistemi lineari o il calcolo di autovalori e autovettori.
- Per **compatibilità con gli standard BLAS e Fortran**, cuBLAS sceglie di usare la memorizzazione **column-major**.
- **Esempio di Utilizzo**: Calcolare il prodotto di due matrici (C = A * B) in modo efficiente sfruttando la potenza di calcolo parallelo delle GPU tramite la libreria cuBLAS.

1. Includere l'Header e Inizializzare cuBLAS

- Includere l'header `<cublas_v2.h>` per accedere alle funzioni di cuBLAS e creare un **handle** per gestire le operazioni della libreria.

```
#include <cuda_runtime.h>  // Include necessario per le funzioni CUDA
#include <cublas_v2.h>     // Include necessario per le funzioni cuBLAS

cublasHandle_t handle;
cublasCreate(&handle);  // Inizializza il contesto cuBLAS
```

14

Workflow delle Librerie CUDA: Esempio con cuBLAS

2. Preparazione dei Dati

- Allocare la memoria necessaria sulla GPU per le matrici di input (A, B) e di output (C) utilizzando `cudaMalloc`.
- Copiare i dati delle matrici dalla memoria dell'host alla memoria del device con `cudaMemcpy`.

```
float *h_A, *h_B, *h_C; // Matrici su CPU (host)
float *d_A, *d_B, *d_C; // Matrici su GPU (device)
int m, n, k; // Dimensioni matrici (es. A: m x k, B: k x n, C: m x n)
// (Memoria GPU allocata con cudaMalloc)
cudaMemcpy(d_A, h_A, m * k * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, k * n * sizeof(float), cudaMemcpyHostToDevice);
```

3. Configurare ed Eseguire la Moltiplicazione

- Imposta i parametri per la moltiplicazione, come le **dimensioni delle matrici**, il **tipo di dato** (`float` in questo caso) ed **eventuali coefficienti di scaling** (α, β). **Nota**: cuBLAS usa *column-major order*.
- Eseguire la moltiplicazione delle matrici sulla GPU chiamando la funzione `cublasSgemm` (Single-precision).

```
float alpha = 1.0f, beta = 0.0f; // C = α * A * B + β * C
cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, n, m, k, &alpha, d_B, n, d_A, k, &beta, d_C, n);
```

15

Workflow delle Librerie CUDA: Esempio con cuBLAS

4. Recuperare i Dati

- Copia la matrice risultato (C) dalla memoria della GPU alla memoria dell'host utilizzando `cudaMemcpy`.

```
cudaMemcpy(h_C, d_C, m * n * sizeof(float), cudaMemcpyDeviceToHost);
```

5. Liberare le Risorse

- Libera la memoria allocata sulla GPU con `cudaFree` e distruggi l'handle cuBLAS con `cublasDestroy`.

```
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
cublasDestroy(handle);
```

Compilazione

- Per compilare un programma CUDA che utilizza cuBLAS, è necessario utilizzare `nvcc` con specifici flag:

```
nvcc program.cu -lcublas -o program
```

16

Workflow delle Librerie CUDA: Esempio con cuRAND

Obiettivo

- **Generare un insieme di numeri casuali** su GPU utilizzando la libreria cuRAND (**CUDA Random Number Generation**), che è ottimizzata per la generazione parallela di numeri casuali in grandi quantità.

1. Includere l'Header e Inizializzare cuRAND

- Includere l'header `<curand.h>` per accedere alle funzioni della libreria cuRAND.
- Creare e inizializzare uno stato del generatore di numeri casuali.

```
#include <cuda_runtime.h>  // Include necessario per le funzioni CUDA
#include <curand.h>     // Include necessario per le funzioni cuRAND

curandGenerator_t gen;
// Crea il generatore di numeri casuali
curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
// Inizializza il generatore con un seed
curandSetPseudoRandomGeneratorSeed(gen, 1234ULL);
```

17

Workflow delle Librerie CUDA: Esempio con cuRAND

2. Preparazione dei Dati

- Riservare spazio sulla GPU per contenere i numeri casuali generati.
- Specificare il numero di numeri casuali da generare.

```
float *d_data;
// Numero di numeri casuali da generare
size_t n = 100000;
// Alloca memoria sulla GPU per i numeri casuali
cudaMalloc((void**)&d_data, n * sizeof(float));
```

3. Generare i Numeri Casuali

- Usare cuRAND per generare numeri casuali uniformemente distribuiti in virgola mobile.

```
// Genera n numeri casuali uniformemente distribuiti in [0,1) su GPU
curandGenerateUniform(gen, d_data, n);
```

18

Workflow delle Librerie CUDA: Esempio con cuRAND

4. Recuperare i Dati

- Copiare i numeri casuali generati dalla GPU alla CPU per ulteriori elaborazioni o analisi.

```
// Alloca memoria sulla CPU
float *h_data = (float*)malloc(n * sizeof(float));
// Copia i dati dalla GPU alla CPU
cudaMemcpy(h_data, d_data, n * sizeof(float), cudaMemcpyDeviceToHost);
```

5. Liberare le Risorse

- Liberare la memoria GPU e distruggere il generatore cuRAND per rilasciare le risorse allocate.

```
cudaFree(d_data);  // Libera la memoria allocata sulla GPU
curandDestroyGenerator(gen);  // Distrugge il generatore di numeri casuali
```

Compilazione

- Per compilare un programma CUDA che utilizza cuRAND, è necessario utilizzare `nvcc` con specifici flag:

```
nvcc program.cu -lcurand -o program
```

19

2.5.3. Utilizzo delle Librerie CUDA

Introduzione all'uso delle GPU in Python

L'Ecosistema Python per il Calcolo Scientifico

- Python è tra i **linguaggi più usati in ambito scientifico e ingegneristico**, grazie alla sua semplicità e alle numerose librerie per il calcolo numerico e il machine learning.
- L'**ecosistema Python** include strumenti fondamentali come NumPy per il calcolo numerico, SciPy per il calcolo scientifico, Matplotlib e Jupyter per grafici e per l'esecuzione interattiva del codice.

Le Limitazioni del Python Tradizionale

- Python, essendo un **linguaggio interpretato con tipizzazione dinamica** e progettato per eseguire codice principalmente su CPU, risulta **significativamente più lento** rispetto a linguaggi compilati come C++ o Fortran.

Librerie/Framework Python per il Calcolo su GPU

- Per chi vuole scrivere **kernel GPU personalizzati**:
  - Numba, PyOpenCL, PyCUDA, CUDA Python
- **Sostituti diretti** per NumPy, SciPy, scikit-learn, ecc.:
  - CuPy, RAPIDS
- **Framework di Machine/Deep Learning**:
  - PyTorch, TensorFlow, JAX
- **Calcolo Distribuito / Multi-nodo**:
  - mpi4py+X, dask, cuNumeric

21

Introduzione a PyCUDA

Cos'è PyCUDA? (Documentazione Online)

- **PyCUDA** è una **libreria Python** che consente di eseguire codice CUDA su GPU NVIDIA direttamente da Python.
- Fornisce un'interfaccia per **scrivere kernel CUDA in C/C++** e lanciare la loro esecuzione, integrandosi con librerie Python come NumPy.

Perchè usare PyCUDA?

- **Facilità d'uso**: Sintassi Python semplice e intuitiva, ideale per chi ha familiarità con Python.
- **Integrazione Python**: Si integra perfettamente con librerie come NumPy, semplificando la gestione dei dati.
- **Compilazione dinamica**: PyCUDA permette di scrivere e compilare **kernel CUDA** (scritti in **C/C++**) al volo durante l'esecuzione del programma, senza bisogno di compilazione esterna.
- **Gestione semplificata della memoria**: Semplifica la gestione della memoria rispetto a CUDA C grazie al *garbage collector* di Python, ma l'allocazione e il trasferimento dei dati tra CPU e GPU restano manuali.
- **Debugging semplificato**: Errori CUDA gestiti come eccezioni Python, facilitando il debug.

Installazione

- PyCUDA può essere installato facilmente usando `pip`:
  `pip install pycuda`
- **Prerequisiti**: È necessario avere driver CUDA e CUDA Toolkit installati e configurati correttamente.

22

Architettura e Principi di Funzionamento di PyCUDA

Compilazione Just-in-Time (JIT):

- Il codice CUDA, **definito come stringhe** all'interno del codice Python, viene **compilato dinamicamente durante l'esecuzione** del programma (caching dei kernel compilati per evitare ricompilazioni ridondanti).

Gestione della Memoria:

- PyCUDA permette di **allocare e gestire la memoria** del device GPU **direttamente da Python**, semplificando il trasferimento dati tra host e device.

Esecuzione di Kernel:

- PyCUDA offre un'interfaccia intuitiva per lanciare kernel CUDA, definendo la configurazione di griglia, blocchi, dimensione della shared memory ed altre proprietà del kernel tramite una **sintassi Pythonica**.

API Pythonica:

- Offre sia **API di alto livello** per operazioni comuni, che **API di basso livello** per controllo granulare.
- Mantiene l'accesso alle funzionalità native CUDA pur fornendo un'interfaccia Pythonica.

Error Handling e Debugging:

- Converte automaticamente gli **errori CUDA in eccezioni Python native**, fornendo informazioni dettagliate sul contesto degli errori, inclusi problemi di compilazione, esecuzione e gestione della memoria.

Interoperabilità con Ecosistema Python e Librerie CUDA:

- Integrazione con **librerie scientifiche Python (NumPy, SciPy)** e accesso alle **librerie CUDA ottimizzate** (cuBLAS, cuFFT, cuRAND, ecc.) tramite scikit-cuda.

23

Sintassi di un Kernel in PyCUDA: Esempio

Sintassi Kernel

- **CUDA C:** Il kernel viene scritto in C e compilato prima dell'esecuzione.
- **PyCUDA:** Il kernel CUDA viene definito come stringa all'interno del programma Python, compilato dinamicamente a runtime tramite la classe **`SourceModule`** e recuperato usando il metodo **`get_function`**.

CUDA C

```
__global__ void add_arrays(float *a, float *b, float *c, int N) {
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   if (idx < N) c[idx] = a[idx] + b[idx];
}
```

PyCUDA

```
mod = SourceModule("""
__global__ void add_arrays(float *a, float *b, float *c, int N) {
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   if (idx < N) c[idx] = a[idx] + b[idx];
}
""")
func = mod.get_function("add_arrays")
```

24

Sintassi di un Kernel in PyCUDA: Esempio

Lancio di un Kernel

- **CUDA C:** Il kernel è lanciato con `add_arrays<<<grid_size, block_size>>>(args)` dove `grid_size` definisce il numero di blocchi e `block_size` il numero di thread per blocco.
- **PyCUDA:** Il kernel è lanciato con `func(d_a, d_b, d_c, block=(block_size, 1, 1), grid=(grid_size, 1))`.

CUDA C

```
int main() {
    // Allocazione memoria, copia dati, lancio kernel
    int block_size = 256;
    int grid_size = (N + block_size - 1) / block_size; // Calcolo dimensione griglia
    add_arrays<<<grid_size, block_size>>>(d_a, d_b, d_c);
}
```

PyCUDA

```
# Allocazione memoria, copia dati, lancio kernel
block_size = 256
grid_size = (N + block_size - 1) // block_size  # Calcolo dimensione griglia
func(d_a, d_b, d_c, block=(block_size, 1, 1), grid=(grid_size, 1))
```

25

Trasferimento Dati in PyCUDA: Approccio Low-Level

Compatibilità diretta con NumPy

- PyCUDA consente di **utilizzare array NumPy** per trasferire dati tra host e device.

Esempio di Trasferimento Dati

- Creazione di un array NumPy su host:

```
import numpy as np
a = np.random.randn(100).astype(np.float32)  # Array NumPy su CPU
```

- Allocazione memoria sul device + Copia dati:

```
import pycuda.driver as cuda
import pycuda.autoinit

# Allocazione memoria sul device e copia dati
d_a = cuda.mem_alloc(a.nbytes)  # Alloca memoria su GPU
cuda.memcpy_htod(d_a, a)        # Copia array da host a device
```

- Copia dati dal device all'host + Deallocazione della memoria sul device (opzionale):

```
a_host = np.empty_like(a)  # Crea array vuoto per ricevere i dati
cuda.memcpy_dtoh(a_host, d_a)  # Copia dati dal device all'host
d_a.free()  # Deallocazione esplicita (se necessario)
```

26

Trasferimento Dati in PyCUDA: Approccio High-Level

Memoria in PyCUDA

- PyCUDA semplifica notevolmente la **gestione dei dati tra host e device** tramite la classe **`gpuarray`**, che rappresenta l'equivalente GPU degli array NumPy.
- Questa classe **gestisce automaticamente allocazione e deallocazione** della memoria GPU e fornisce funzionalità familiari come operazioni su array multidimensionali e slicing.

Esempio di Trasferimento Host-Device con `gpuarray`

```
import numpy as np
import pycuda.gpuarray as gpuarray
import pycuda.autoinit

# Array su host
host_data = np.array([1, 2, 3], dtype=np.float32)

# Trasferimento dati dalla CPU alla GPU
gpu_data = gpuarray.to_gpu(host_data)

# Recupero dei risultati dalla GPU alla CPU
result = gpu_data.get()
```

27

Esempio di Codice PyCUDA - Somma di Vettori

1. Inizializzazione e Contesto GPU

- **Cos'è il Contesto?**: Il "contesto" è come una sessione di lavoro che gestisce le risorse della GPU, come la memoria e i kernel. Serve a collegare il programma Python alla GPU, permettendo di eseguire codice CUDA.

```
import numpy as np
from pycuda import driver, compiler, gpuarray

driver.init()  # Inizializza PyCUDA
device = driver.Device(0) # Seleziona la prima GPU disponibile
context = device.make_context()  # Crea un contesto CUDA per il device
```

2. Definizione e Compilazione del Kernel CUDA

- **Compilazione:** In PyCUDA, il kernel è scritto come stringa di testo in Python e viene compilato durante l'esecuzione del programma.

```
kernel_code = """
__global__ void add_arrays(float *a, float *b, float *c, int N) {
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   if (idx < N) c[idx] = a[idx] + b[idx];
}
"""
module = compiler.SourceModule(kernel_code)  # Compila il kernel CUDA
```

28

Esempio di Codice PyCUDA - Somma di Vettori

3. Allocazione Memoria su GPU e Lancio del Kernel

- **Allocazione della Memoria**: Utilizzo della classe `gpuarray` per allocare memoria sulla GPU e trasferire facilmente dati tra la GPU e l'host.

```
a_gpu = gpuarray.to_gpu(np.random.randn(N).astype(np.float32))
b_gpu = gpuarray.to_gpu(np.random.randn(N).astype(np.float32))
c_gpu = gpuarray.empty_like(a_gpu)
add_arrays = module.get_function("add_arrays")
add_arrays(a_gpu, b_gpu, c_gpu, np.int32(N), block=(block_size,1,1), grid=(grid_size, 1))
```

4. Recupero dei Risultati e Pulizia

- **Recupero dei Dati:** Il risultato dell'operazione sulla GPU viene trasferito di nuovo alla CPU con il metodo `.get()`.
- **Rilascio del Contesto**: Il contesto GPU viene rilasciato, liberando le risorse. Questo passaggio è importante per evitare memory leaks e mantenere la GPU libera per altri usi.

```
c_cpu = c_gpu.get()  # Copia il risultato dalla GPU alla CPU

context.pop()  # Rilascia il contesto GPU
```

29

Esempio di Codice PyCUDA - Somma di Vettori

Esecuzione del Profiling con Nsight Compute

- Per eseguire il profiling del programma, è possibile utilizzare il comando `ncu` direttamente dal terminale. Esempio di comando:

```
ncu --set full python vector_add.py
```

- Opzioni principali di `ncu`:
  - `--set full`: Esegue un profiling completo, includendo metriche dettagliate sulle prestazioni.
  - `python vector_add.py`: Specifica il nome dello script Python che avvia l'esecuzione dei kernel CUDA su cui fare il profiling.

30

Inclusione File CUDA Esterno in PyCUDA - Somma di Vettori

File CUDA Esterno (vector_add.cu)

- Invece di scrivere i kernel come stringhe in Python, è possibile **caricare e compilare un file CUDA esterno**, migliorando la leggibilità e il riutilizzo del codice.

File CUDA C - vector_add.cu

```
__global__ void add_arrays(float *a, float *b, float *c, int N) {
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   if (idx < N) c[idx] = a[idx] + b[idx];
}
```

Integrazione in Python (vector_add.py)

```
with open('vector_add.cu', 'r') as f:
   kernel_code = f.read()
module = compiler.SourceModule(kernel_code)
add_arrays = module.get_function("add_arrays")

# Configurazione griglia per N elementi
block_size = 256
grid_size = (N + block_size - 1) // block_size
add_arrays(a_gpu, b_gpu, c_gpu, np.int32(N),  block=(block_size,1,1), grid=(grid_size,1,1))
```

31

Confronto fra CUDA C e PyCUDA

	CUDA C	PyCUDA
Sintassi	Codice **più complesso**, ma offre **controllo preciso**.	**Sintassi semplificata**, più leggibile, ma con **meno controllo**.
Compilazione	Compilazione **statica** del codice in fasi distinte (build separata).	**Dinamica**, più flessibile ma potenzialmente meno efficiente.
Gestione della Memoria	**Manuale**, ma garantisce un controllo totale sulle risorse.	**Semplificata**, riduce il rischio di errori, ma meno flessibile.
Interfaccia host-device	**Completo controllo** su trasferimenti, allocazioni e sincronizzazioni.	**Semplificata**, ma con **meno possibilità di ottimizzazioni** avanzate.
Debugging	Più laborioso, richiede **gestione manuale degli errori**.	Gestito tramite **eccezioni Python**, semplificando il debug.
Flessibilità	Richiede **molte linee di codice** per compiti semplici.	Ideale per **prototipazione veloce** e per interfacce con altre librerie Python.
Astrazione	**Basso livello**, controllo completo su hardware e risorse.	**Livello di astrazione più alto**, automatizza molte operazioni.

32

Introduzione a CuPy

Cos'è CuPy? (Documentazione Online)

- **CuPy** è una libreria **open-source,** compatibile con NumPy, progettata per il calcolo scientifico su GPU.
- Fornisce un'interfaccia quasi identica a NumPy ma esegue i calcoli su **GPU NVIDIA tramite CUDA**.
- CuPy **sfrutta librerie del CUDA Toolkit** (come cuBLAS, cuRAND, cuSOLVER) per garantire elevate prestazioni.
- È disponibile anche il **supporto per GPU AMD tramite ROCm**, sebbene con compatibilità più limitata.

Perché usare CuPy?

- **Familiarità**: API simile a NumPy, curva di apprendimento minima.
- **Performance**: Accelerazione GPU senza scrivere codice CUDA.
- **Compatibilità**: Supporta la maggior parte delle funzioni NumPy e SciPy.
- **Facilità d'Uso**: Non richiede conoscenza diretta di CUDA.
- **Ottimizzazione Automatica**: Gestisce automaticamente memoria e trasferimenti CPU-GPU.

Funzionalità Chiave

- **Array N-dimensionali**: Supporto completo per array multi-dimensionali eseguiti su GPU.
- **Operazioni Matematiche**: Funzioni matematiche avanzate (equivalenti a Numpy e SciPy) ottimizzate per GPU.
- **Kernel Personalizzati**: Possibilità di scrivere kernel CUDA personalizzati.

Installazione

- PyCUDA può essere installato facilmente usando `pip`:

```
pip install cupy-cuda11x   # sostituire con versione CUDA appropriata
```

33

Introduzione agli Array CuPy

Core della Libreria

- La classe `cupy.ndarray` è il componente fondamentale di CuPy, offrendo un'**alternativa GPU completamente compatibile** con `numpy.ndarray`.

Inizializzazione

- Importazione delle librerie simile a NumPy:

```
import numpy as np
import cupy as cp
```

Creazione Array GPU

- Sintassi identica a NumPy ma con allocazione su GPU:

```
x_gpu = cp.array([1, 2, 3])      # Array su GPU
x_cpu = np.array([1, 2, 3])      # Array su CPU
```

Computazione

- Stesse funzioni di NumPy ma eseguite su GPU:

```
# Calcolo norma L2 (cp utilizza cuBLAS internamente)
l2_gpu = cp.linalg.norm(x_gpu)   # Esecuzione su GPU
l2_cpu = np.linalg.norm(x_cpu)   # Esecuzione su CPU
```

34

Gestione dei Dispositivi GPU in CuPy

Il Device Corrente

- Ogni operazione CuPy avviene su un "*device corrente*" (GPU predefinita).
- Esempio con **GPU default** (ID 0):

```
x = cp.array([1, 2, 3])  # Allocato su GPU 0
```

Selezione del Device (Con una sola GPU, non serve gestire manualmente i device)

- Cambio **permanente**:

```
cp.cuda.Device(1).use()  # Passa a GPU 1
```

- Cambio **temporaneo** tramite Context Manager:

```
with cp.cuda.Device(1):  # Usa GPU 1 solo in questo blocco
    x = cp.array([1, 2, 3])
```

Regole e Limitazioni

- Gli array devono essere elaborati sulla stessa GPU dove sono allocati. **Errori comuni**:

```
with cp.cuda.Device(0):
    x = cp.array([1, 2, 3])
with cp.cuda.Device(1):
    x * 2  # Errore: array su GPU 0, operazione su GPU 1
```

35

Trasferimento Dati tra CPU e GPU in CuPy

Da CPU a GPU

- **`cp.asarray()`** sposta dati su GPU corrente array numpy, liste e tuple python, qualsiasi oggetto compatibile con **`numpy.array()`**.

```
x_cpu = np.array([1, 2, 3])
x_gpu = cp.asarray(x_cpu)  # trasferimento su GPU
```

Tra GPU diverse

- Stesso metodo per trasferire tra device:

```
with cp.cuda.Device(0):
    x_gpu_0 = cp.array([1, 2, 3])  # crea un array su GPU 0
with cp.cuda.Device(1):
    x_gpu_1 = cp.asarray(x_gpu_0)  # muove l'array su GPU 1
```

Ottimizzazione Copie

- `cp.asarray()` **evita copie non necessarie se possibile**, restituendolo direttamente se già sul device corrente.
- **Per forzare la copia**: `cp.array(arr, dtype, copy=True)`
- `cp.asarray()` equivale a `cp.array(arr, dtype, copy=False)`

36

Trasferimento Dati tra CPU e GPU in CuPy

Da GPU a CPU

- Con **`asnumpy()`**:

```
x_gpu = cp.array([1, 2, 3])        # crea array su GPU
x_cpu = cp.asnumpy(x_gpu)          # sposta su CPU
```

- Con **`get()`**:

```
x_cpu = x_gpu.get()                # sposta su CPU
```

Note

- Entrambi i metodi **creano un array NumPy sulla CPU.**
- L'array originale rimane sulla GPU.

37

Codice Agnostico CPU/GPU in CuPy

`get_array_module()`

- `cupy.get_array_module()` **restituisce il modulo corretto** (Numpy o CuPy) in base agli argomenti. Esempio:

```
def softplus(x):
    xp = cp.get_array_module(x)  # alias per numpy o cupy, a seconda del tipo di x
    return xp.maximum(0, x) + xp.log1p(xp.exp(-abs(x)))
```

Conversioni tra CPU e GPU

- Uso dei metodi `asarray()` e `asnumpy()` per gestire array di tipo CPU e GPU:

```
y_cpu = np.array([4, 5, 6])
x_gpu = cp.array([1, 2, 3])

x_gpu + y_cpu              # Errore: tipi incompatibili
cp.asnumpy(x_gpu) + y_cpu  # OK: entrambi su CPU
x_gpu + cp.asarray(y_cpu)  # OK: entrambi su GPU
```

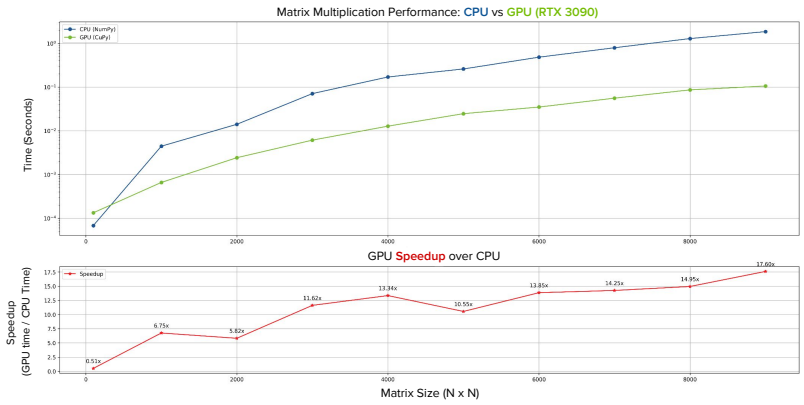
Regola Chiave

- Le operazioni richiedono array dello **stesso tipo** (entrambi CPU o GPU).

38

Moltiplicazione di Matrici con GPU (CuPy) e CPU (NumPy)

- Confronto prestazionale tra le operazioni **`cp.matmul()`** e **`np.matmul()`** per la **moltiplicazione di matrici quadrate** di dimensione crescente.



39

Definizione di Kernel in CuPy

User-Defined Kernels

- CuPy permette di definire **tre tipi di kernel CUDA** (*elementwise*, *reduction* e *raw*).
- Ci focalizziamo sui **raw kernel** che offrono il controllo completo sull'implementazione.

```
# Definizione kernel (compilazione JIT alla prima invocazione)
add_kernel = cp.RawKernel(r'''
extern "C"
__global__ void add_arrays(float *a, float *b, float *c, int N) {
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   if (idx < N) c[idx] = a[idx] + b[idx];
}
''', 'my_add')

# Utilizzo
x1 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
x2 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
y = cp.zeros((5, 5), dtype=cp.float32)
add_kernel((grid_size,), (block_size,), (x1, x2, y, N))  # griglia, blocchi, argomenti
```

40

Definizione di Kernel in CuPy

Raw Module in CuPy

- **`RawModule`** gestisce codice sorgente CUDA o binari CUDA esistenti. Si inizializza con codice sorgente CUDA o percorso del binario, offrendo accesso ai kernel tramite **`get_function()`**.

```
# Definizione modulo con più kernel
module = cp.RawModule(code=r'''
extern "C"{
   __global__ void test_sum(const float* x1, const float* x2, float* y, unsigned int N) {
       unsigned int tid = blockDim.x * blockIdx.x + threadIdx.x;
       if (tid < N) {
           y[tid] = x1[tid] + x2[tid];
       }
   }
   __global__ void test_multiply(const float* x1, const float* x2, float* y,
                                 unsigned int N) {
       unsigned int tid = blockDim.x * blockIdx.x + threadIdx.x;
       if (tid < N) {
           y[tid] = x1[tid] * x2[tid];
       }
   }
}''')
```

```
# Caricamento da file
module = cp.RawModule(path='path/to/kernels.cubin')
```

1/2

41

Definizione di Kernel in CuPy

Raw Module in CuPy

- **`RawModule`** gestisce codice CUDA esteso o binari esistenti. Si inizializza con codice sorgente CUDA o percorso del binario, offrendo accesso ai kernel tramite **`get_function()`**.

```
# Recupero e utilizzo dei kernel
ker_sum = module.get_function('test_sum')
ker_times = module.get_function('test_multiply')

# Test dei kernel
N = 10
x1 = cp.arange(N**2, dtype=cp.float32).reshape(N, N)
x2 = cp.ones((N, N), dtype=cp.float32)
y = cp.zeros((N, N), dtype=cp.float32)

ker_sum((N,), (N,), (x1, x2, y, N**2))       # y = x1 + x2
ker_times((N,), (N,), (x1, x2, y, N**2))     # y = x1 * x2
```

2/2

42

Confronto fra PyCUDA e CuPy

	PyCUDA	CuPy
Approccio Base	**Wrapper Python** per CUDA che mantiene controllo di basso livello.	**Replica NumPy su GPU** con automazione delle operazioni.
Livello di Astrazione	**Basso livello**, richiede scrittura diretta di kernel CUDA.	**Alto livello**, operazioni simili a NumPy e SciPy.
Gestione della Memoria	**Gestione perlopiù manuale** dei trasferimenti host-device.	**Gestione automatica** con memory pool.
Compilazione	**JIT** dei kernel definiti come stringhe CUDA C.	**Ibrido**: librerie pre-compilate (cuBLAS, cuFFT, ecc.) + JIT per kernel custom.
Integrazione	Richiede **codice esplicito** per interagire con NumPy.	**Compatibilità diretta** con NumPy/SciPy.
Curva Apprendimento	Richiede conoscenza CUDA C/C++.	Richiede solo conoscenza NumPy e SciPy.
Supporto Hardware	GPU NVIDIA	GPU NVIDIA, con **supporto ROCm** (AMD).
Manutenibilità	**Codice più lungo**, richiede più manutenzione.	**Codice conciso**, più facile da mantenere.

43

2.5.4. Applicazioni

Applicazioni CUDA

Perché CUDA è Importante?

- CUDA è diventata una tecnologia fondamentale in innumerevoli campi applicativi:
  - **Intelligenza Artificiale & Deep Learning**: Training e inferenza di reti neurali.
  - **Calcolo Scientifico**: Simulazioni fisiche, analisi dei dati, chimica computazionale.
  - **Crypto & Blockchain**: Mining e validazione.
  - **Media & Intrattenimento**: Rendering 3D, elaborazione video, gaming.

Opportunità nel CVLab

- Il **Computer Vision Laboratory (CVLab)** della nostra università si occupa di Computer Vision utilizzando tecniche di deep learning, con focus su:
  - **Comprensione della Scena 3D**: Monocular Depth, Multi-View/Binocular Stereo, SLAM.
  - **Analisi della Scena**: Segmentazione semantica, Object Detection, Tracking.
  - **Novel View Synthesis**: NeRF, 3D Gaussian Splatting.
  - **AI Generativa**: Generazione di immagini e video tramite Diffusion Model.

Tesi e Progetti

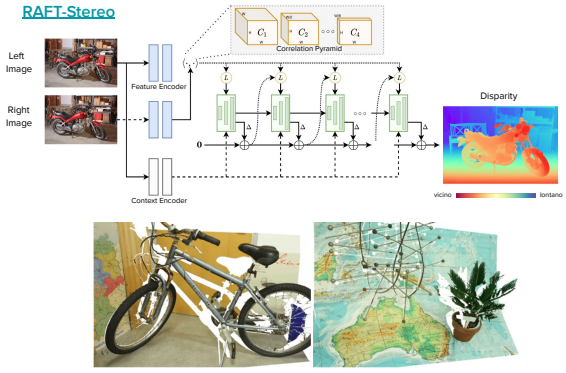
- Siamo aperti a **studenti (fortemente) interessati** a svolgere la tesi su questi temi. Le competenze CUDA acquisite nel corso sono direttamente applicabili nei nostri progetti di ricerca, dove:
  - **Ideiamo** e **implementiamo** nuove soluzioni per problemi di visione artificiale.
  - Utilizziamo **GPU NVIDIA** e **framework** basati su CUDA (PyTorch).
  - Sviluppiamo **moduli CUDA personalizzati** per ottimizzare algoritmi specifici.

45

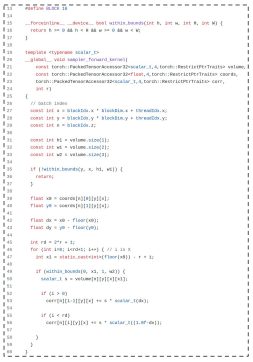
Deep Stereo Matching

- Date **due immagini rettificate** della stessa scena, la profondità viene calcolata tramite triangolazione, determinando la **disparità** come la differenza tra le coordinate orizzontali dei punti corrispondenti.

RAFT-Stereo



sampler_kernel.cu



46

Single-view Depth Estimation

- A partire da una **singola immagine**, la profondità (depth) viene stimata utilizzando **reti neurali** addestrate su dataset di coppie immagine-profondità, sfruttando indizi visivi prospettiva, dimensioni relative, ombre, texture etc.

Single Input Image | Depth Anything v2 Demo HuggingFace 🤗 | Single Input Image | Marigold Demo HuggingFace 🤗

47

Single-view Depth Estimation

- A partire da una **singola immagine**, la profondità (depth) viene stimata utilizzando **reti neurali** addestrate su dataset di coppie immagine-profondità, sfruttando indizi visivi prospettiva, dimensioni relative, ombre, texture etc.

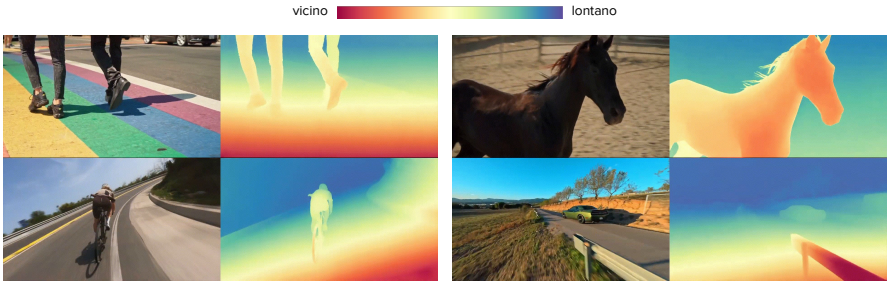
Python Code - Depth Anything v2

```python
import cv2
import torch
from depth_anything_v2.dpt import DepthAnythingV2
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'
model_configs = {
    'vits': {'encoder': 'vits', 'features': 64, 'out_channels': [48, 96, 192, 384]},
    'vitb': {'encoder': 'vitb', 'features': 128, 'out_channels': [96, 192, 384, 768]},
    'vitl': {'encoder': 'vitl', 'features': 256, 'out_channels': [256, 512, 1024, 1024]},
    'vitg': {'encoder': 'vitg', 'features': 384, 'out_channels': [1536, 1536, 1536, 1536]}
}
encoder = 'vitl' # or 'vits', 'vitb', 'vitg'
model = DepthAnythingV2(**model_configs[encoder])
model.load_state_dict(torch.load(f'checkpoints/depth_anything_v2_{encoder}.pth', map_location='cpu'))
model = model.to(DEVICE).eval()
raw_img = cv2.imread('your/image/path')
depth = model.infer_image(raw_img) # HxW raw depth map in numpy
```

48

Temporally Consistent Monocular Depth Estimation

- Estende la stima monoculare della profondità a **sequenze video**, garantendo coerenza temporale tra i frame consecutivi per produrre una ricostruzione 3D fluida e stabile nel tempo.



Monocular Input Video | Output Depth | Monocular Input Video | Output Depth

Rolling Depth

Demo HuggingFace 🤗

Novel View Synthesis

- Tecnica che permette di generare **nuove viste di una scena** a partire da una o più immagini esistenti, ricostruendo la geometria 3D e le texture per renderizzare viewpoint mai osservati.

3D Gaussian Splatting

rasterizer.cu

```
236         dim3 tile_grid((width + BLOCK_X - 1) / BLOCK_X, (height + BLOCK_Y - 1) / BLOCK_Y, 1);
237         dim3 block(BLOCK_X, BLOCK_Y, 1);
238
239         // Dynamically resize image-based auxiliary buffers during training
240         size_t img_chunk_size = required<ImageState>(width * height);
241         char* img_chunkptr = imageBuffer(img_chunk_size);
242         ImageState imgState = ImageState::fromChunk(img_chunkptr, width * height);
243
244         if (NUM_CHANNELS != 3 && colors_precomp == nullptr)
245         {
246                 throw std::runtime_error("For non-RGB, provide precomputed Gaussian colors");
247         }
248
249         // Run preprocessing per-Gaussian (transformation, bounding, conversion of SHs to RGB)
250         CHECK_CUDA(FORWARD::preprocess(
251                 P, D, M,
252                 means3D,
253                 (glm::vec3*)scales,
254                 scale_modifier,
255                 (glm::vec4*)rotations,
256                 opacities,
257                 shs,
258                 geomState.clamped,
259                 cov3D_precomp,
260                 colors_precomp,
261                 viewmatrix, projmatrix,
262                 (glm::vec3*)cam_pos,
263                 width, height,
264                 focal_x, focal_y,
265                 tan_fovx, tan_fovy,
266                 radii,
267                 geomState.means2D,
268                 geomState.depths,
269                 geomState.cov3D,
270                 geomState.rgb,
271                 geomState.conic_opacity,
272                 tile_grid,
273                 geomState.tiles_touched,
274                 prefiltered,
275                 antialiasing
276         ), debug)
```


Novel View Synthesis

- Tecnica che permette di generare **nuove viste di una scena** a partire da una o più immagini esistenti, ricostruendo la geometria 3D e le texture per renderizzare viewpoint mai osservati.

Stereo training data from image sequences collected with a single handheld camera

Synthesized Stereo Pairs

Synthesized Stereo Pairs

NeRF-Stereo / Instant-NGP

render_buffer.cu

```
401 __global__ void CUDA_EXPORT kernel_update_buffer(
402         ivec2 resolution,
403         float alpha,
404         const float* __restrict__ depth,
405         float depth_scale,
406         ivec2 image_resolution,
407         int fov_axis,
408         float zoom,
409         vec2 screen_center,
410         cudaSurfaceObject_t surface
411         ) {
412         uint32_t x = threadIdx.x + blockDim.x * blockIdx.x;
413         uint32_t y = threadIdx.y + blockDim.y * blockIdx.y;
414
415         if (x >= resolution.x || y >= resolution.y) {
416                 return;
417         }
418
419         float scale = image_resolution[fov_axis] / float(resolution[fov_axis]);
420
421         float fx = x + 0.5f;
422         float fy = y + 0.5f;
423
424         fx -= resolution.x * 0.5f; fx /= zoom; fx += screen_center.x * resolution.x;
425         fy -= resolution.y * 0.5f; fy /= zoom; fy += screen_center.y * resolution.y;
426
427         float u = (fx - resolution.x * 0.5f) * scale + image_resolution.x * 0.5f;
428         float v = (fy - resolution.y * 0.5f) * scale + image_resolution.y * 0.5f;
429
430         int srcx = floorf(u);
431         int srcy = floorf(v);
432         uint32_t srcidx = srcx + image_resolution.x * srcy;
433
434         vec4 color;
435         if (srcx >= image_resolution.x || srcy >= image_resolution.y || srcx < 0 || srcy < 0) {
436                 color = {0.0f, 0.0f, 0.0f, 0.0f};
437         } else {
438                 float depth_value = depth[srcidx] * depth_scale;
439                 vec3 c = colormap_turbo(depth_value);
440                 color = {c[0], c[1], c[2], 1.0f};
441         }
442
443         vec4 prev_color;
444         surf2Dread(*(uint4*)&prev_color, surface, x * sizeof(float4), y);
445         color = color * alpha + prev_color * (1.f-alpha);
446         surf2Dwrite(to_float4(color), surface, x * sizeof(float4), y);
447 }
```

SLAM (Simultaneous Localization and Mapping)

- Tecnica che permette a un sistema di costruire una **mappa dell'ambiente** e contemporaneamente determinare la **propria posizione all'interno di essa**, utilizzando sensori come telecamere RGB-D per l'elaborazione in tempo reale.

droid_kernels.cu

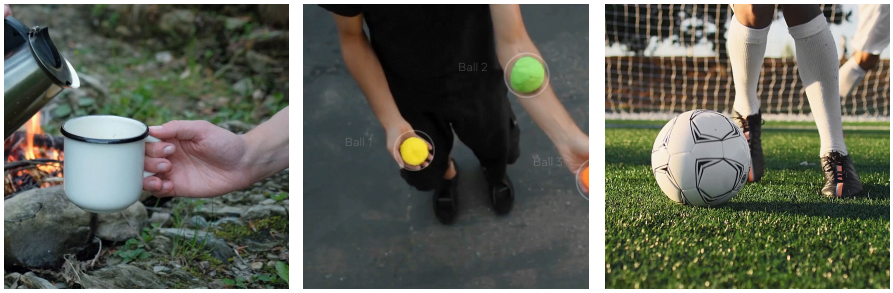
```
427   __global__ void projmap_kernel(
428       const torch::PackedTensorAccessor32<float,2,torch::RestrictPtrTraits> poses,
429       const torch::PackedTensorAccessor32<float,3,torch::RestrictPtrTraits> disps,
430       const torch::PackedTensorAccessor32<float,3,torch::RestrictPtrTraits> intrinsics,
431       const torch::PackedTensorAccessor32<long,1,torch::RestrictPtrTraits> ii,
432       const torch::PackedTensorAccessor32<long,1,torch::RestrictPtrTraits> jj,
433       torch::PackedTensorAccessor32<float,4,torch::RestrictPtrTraits> coords,
434       torch::PackedTensorAccessor32<float,4,torch::RestrictPtrTraits> valid)
435   {
436
437     const int block_id = blockIdx.x;
438     const int thread_id = threadIdx.x;
439
440     const int ht = disps.size(1);
441     const int wd = disps.size(2);
442
443     __shared__ int ix;
444     __shared__ int jx;
445
446     __shared__ float fx;
447     __shared__ float fy;
448     __shared__ float cx;
449     __shared__ float cy;
450
451     __shared__ float ti[3], tj[3], tij[3];
452     __shared__ float qi[4], qj[4], qij[4];
453
454     // load intrinsics from global memory
455     if (thread_id == 0) {
456       ix = static_cast<int>(ii[block_id]);
457       jx = static_cast<int>(jj[block_id]);
458       fx = intrinsics[0];
459       fy = intrinsics[1];
460       cx = intrinsics[2];
461       cy = intrinsics[3];
462     }
463
464     __syncthreads();
```

GLORIE-SLAM

Semantic Segmentation

- Tecnica di computer vision che mira a **suddividere un'immagine in regioni semanticamente significative**, assegnando a ogni pixel una specifica **classe** o **categoria** (es. persona, auto, strada, edificio, cielo).

Segmenting objects across images and videos



Segment Anything v2

Generative AI

- **Modelli generativi** come GAN e diffusion models vengono utilizzati per **sintetizzare e manipolare immagini realistiche** secondo specifiche richieste, mantenendo coerenza fotorealistica.

Midjourney

Generative AI

- **Modelli generativi** come GAN e diffusion models vengono utilizzati per **sintetizzare e manipolare immagini realistiche** secondo specifiche richieste, mantenendo coerenza fotorealistica.

Create Anything in 4D with Multi-View Video Diffusion Models

CAT4D

55

Generative AI

- **Modelli generativi** come GAN e diffusion models vengono utilizzati per **sintetizzare e manipolare immagini realistiche** secondo specifiche richieste, mantenendo coerenza fotorealistica.

ControlNet

Adding Conditional Control to Text-to-Image Diffusion Models

Demo HuggingFace 🤗

Control (Edges) | Original Image | **Prompt**: "*cinematic, luxury apartment, colourful, highly detailed*" | **Prompt**: "*cinematic, cyber-punk apartment out of steel and concrete, colourful, highly detailed*"

56

Generative AI

- **Modelli generativi** come GAN e diffusion models vengono utilizzati per **sintetizzare e manipolare immagini realistiche** secondo specifiche richieste, mantenendo coerenza fotorealistica.

Genie

A large-scale foundation world model

***Prompt**: The video shows a first person perspective of someone navigating difficult terrain in the middle of a volcanic area. This is a real world video shot from the perspective of a wheeled robot that needs to traverse across a terrain. The vehicle has chunky offroad tires that crunch under the blackened rock. The camera is an egocentric camera mounted to the vehicle, and you can see the front tires just on the bottom of the camera along with the body of the robot. In the distance you can see smoke and lava flowing from the volcano. There are no other visible signs of life. There are lava pools that the agent is trying to avoid and random rock formations. The sky is a vivid blue.*

***Prompt**: POV action camera of a tan house being painted by a first person agent with a paint roller*

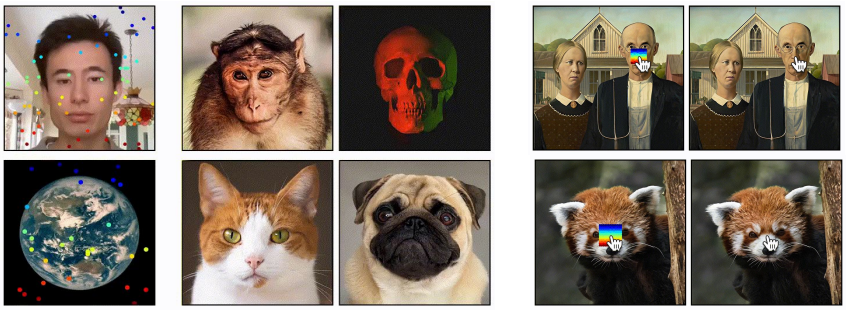
57

Generative AI

- **Modelli generativi** come GAN e diffusion models vengono utilizzati per **sintetizzare e manipolare immagini realistiche** secondo specifiche richieste, mantenendo coerenza fotorealistica.

Motion Prompting

Controlling Video Generation with Motion Trajectories



Motion Transfer

Object Control

3. Riferimenti

Tutto il materiale su cui si basano queste dispense é preso dalle slide del corso (con qualche integrazione/ spiegazione personale). I numeri dei capitoli rispecchiano la numerazione dei pacchetti di slide.