Ottimizzazioni 1

Corso di sviluppo Nvidia CUDA™

Davide Barbieri

Panoramica Lezione

- Principali versioni dell'hardware CUDA
  - Tesla
  - Fermi
- Accesso veloce alla memoria globale
- Accesso veloce alla memoria condivisa

Ottimizzazioni CUDA

- Ottimizzazione degli accessi in memoria
  - Pattern specifici per il tipo di architettura
- Ottimizzazione a livello di thread
  - Configurazione del blocco di thread
  - Occupazione dei multiprocessori
- Ottimizzazione a livello di istruzione
  - *Instruction-Level Parallelism*
- **Un kernel non ottimizzato può perdere ordini di grandezza in prestazioni rispetto ad un kernel ottimizzato**

Compute Capability

➲ *Compute Capability:*

- **Numero che indica le capacità computazionali di un dispositivo**
  - Caratteristiche Architetturali
    - Numero di core
    - Memoria
    - Cache L1, L2
  - Features come:
    - Standard *floating point*
    - Primitive di sincronizzazione

  - **Major number** identifica il tipo di architettura CUDA
    - 1.x ***Tesla***
    - 2.x ***Fermi***
    - 3.x ***Kepler*** (?)
    - 4.x ***Maxwell*** (?)

  - Al **minor number** corrisponde il miglioramento incrementale della stessa

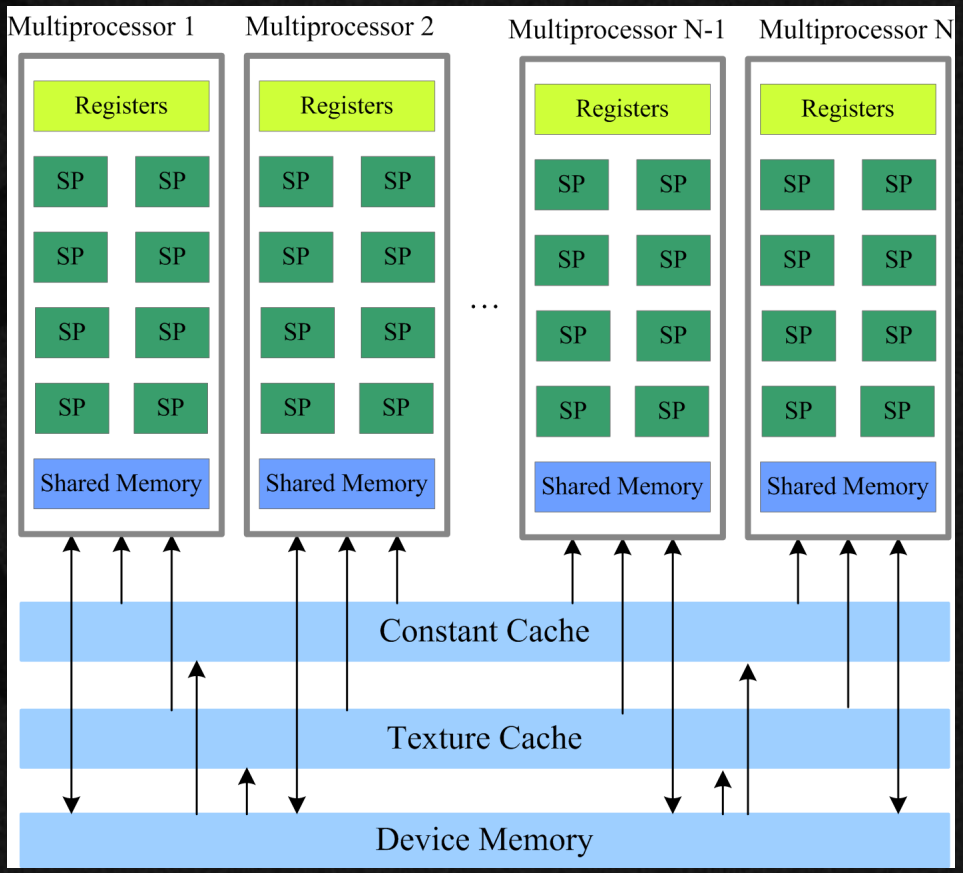

Tesla

Tesla

- Non è il nome ufficiale per le GPU con c.c. 1.*
  - Nome dato alle soluzioni senza uscita video (solo GPGPU)
  - Nelle presentazioni Nvidia, l'architettura viene chiamata così
- 1.0
  - Prima versione
- 1.1
  - **G80**
  - Operazioni atomiche in memoria globale
- 1.2
  - Operazioni atomiche in memoria condivisa
  - Accesso a memoria globale più flessibile
- 1.3
  - **GT200**
  - Doppia precisione

C.C. 1.0 - 1.1 (G80)

- **8 CUDA core per SM**
  - Operazioni *floating point* (solo precisione singola)
  - Operazioni su interi
- **2 Unità per funzioni *speciali***
  - *Operazioni 1/x, Sin(x), Sqrt(x), ...*
- ***8192** registri a 32 bit* per *SM*
  - *Divisi tra blocchi di thread schedulati sullo stesso SM*
- *16 KB di memoria condivisa*
- La GPU esegue un solo kernel alla volta



- 1 *Warp Scheduler*
  - Seleziona ogni 4 cicli un'istruzione appartenente ad un ***warp attivo*** sul multiprocessore
  - Warp attivo = warp appartenente a blocco che ha ottenuto le risorse sullo SM
    - Pronto per essere schedulato

- Istruzioni hanno throughput differenti
  - Operazioni come **ADD**, **MUL** e **MAD** (**MUL** + **ADD**) eseguite sugli ***8 core***

    1 warp (32 thread) : 8 core → issue di un warp ogni ***4 cicli***
  - Operazioni come **DIV**, **SQRT** eseguite sui 2 *Spec. Func. Units*

    *32 thread : 2 → issue di un warp ogni **16 cicli***

- *Ogni SM riesce a tenere attivi 24 warp*
  - *24 warp = 24*32 = 768 thread*

GT200

- Regole meno rigide per accessi coalizzati
- 16384 registri a 32 bit
- Doppia precisione
  - 1 unità double per SM
    → 1/8 di throughput rispetto a float
- *Ogni SM riesce a tenere attivi 32 warp*

Tesla

- Il modello floating-point è simile a IEEE754
  - Alcune differenze
- Non supporta i numeri denormalizzati
  e.g. $(-1)^s \times 0.[\text{MANTISSA}] \times 2^{-126}$
  - Vengono approssimati a 0
- Non sono supporta le eccezioni
- Spesso potrete notare differenze tra i risultati CPU e GPU
  - La CPU x86 può utilizzare come buffer intermedi registri a 32, 64 e 80 bit
    - FPU control word: registro a 16 bit per configurare l'FPU
      - fstcw / fldcw
      - 80 bit → 32 bit
    - Troncamento avviene solo alla fine di una serie di operazioni
    - Le estensioni x86 SSE funzionano diversamente

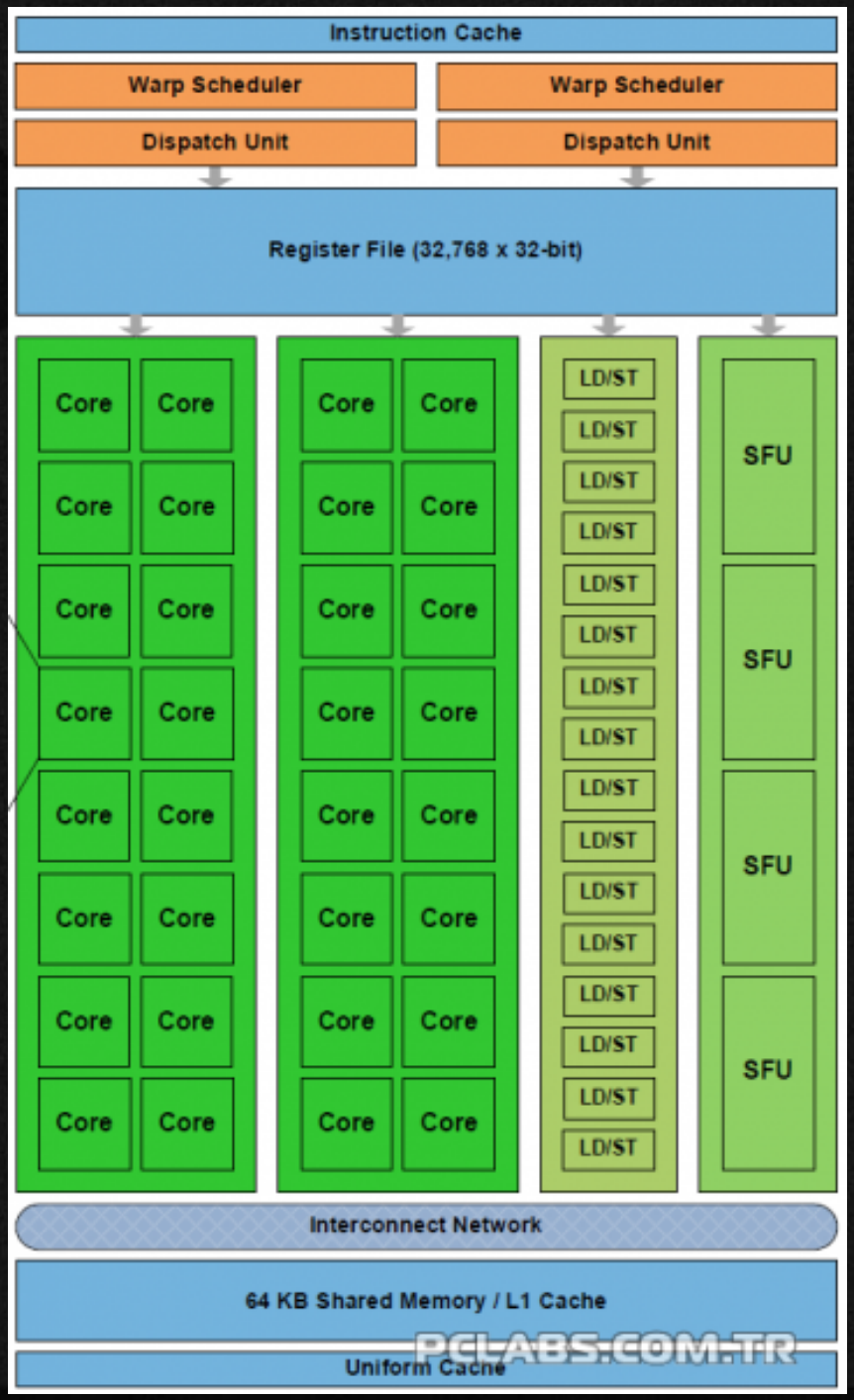

Fermi

Fermi

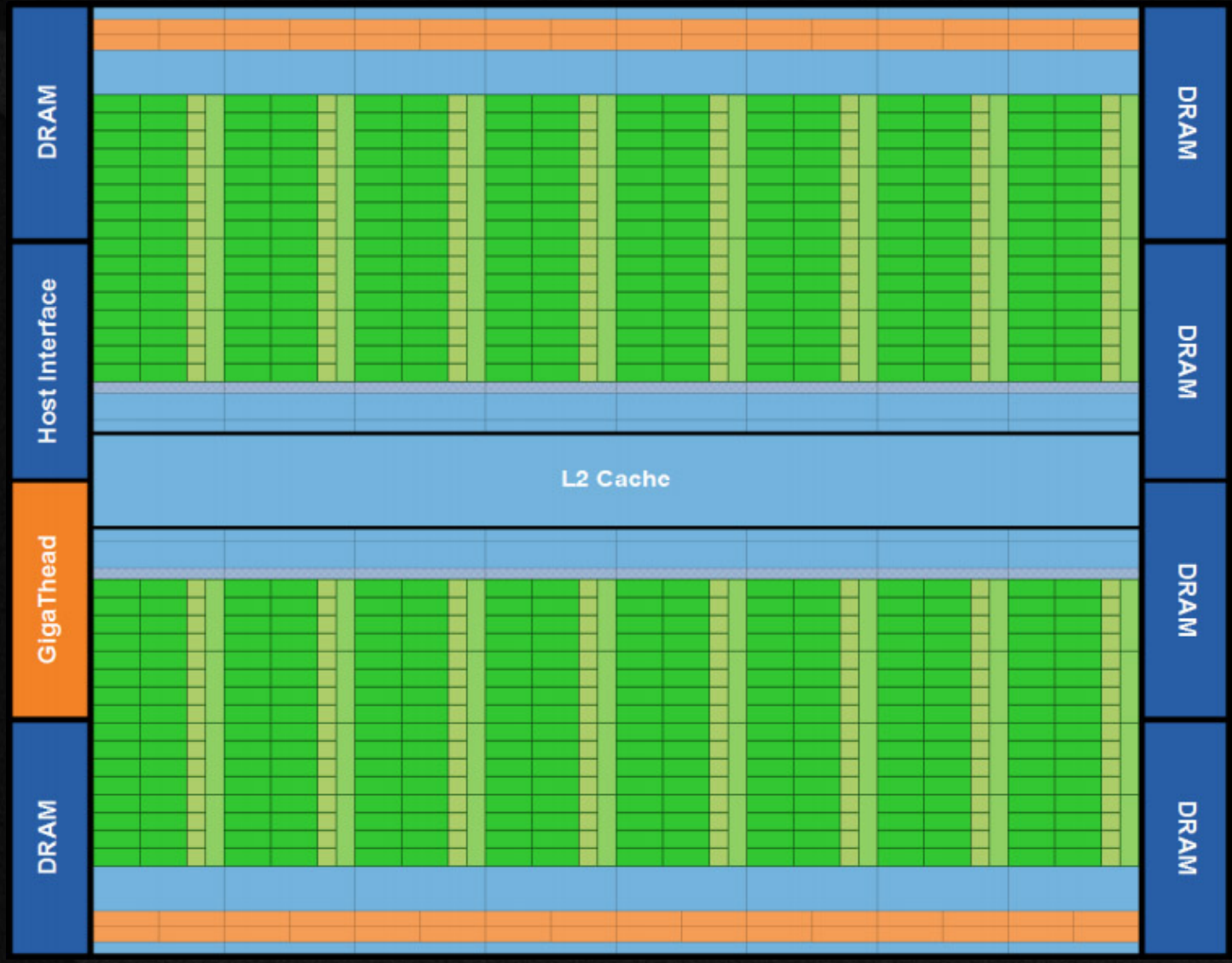
- Compute Capability 2.0
  - **GF100**
- Compute Capability 2.1
  - **GF104**

GF100

- 32 core per SM
  - Divisi in due metà da 16 core
- 32768 registri a 32 bit
- 4 special function unit
- 16 unità load/store
- ***Cache L1*** / Memoria condivisa *configurabili* per SM
  - 16 KB Cache L1
    e 48 KB Memoria Condivisa
  - 48 KB Cache L1
    e 16 KB Memoria Condivisa
- *Ogni SM riesce a tenere attivi 48 warp*



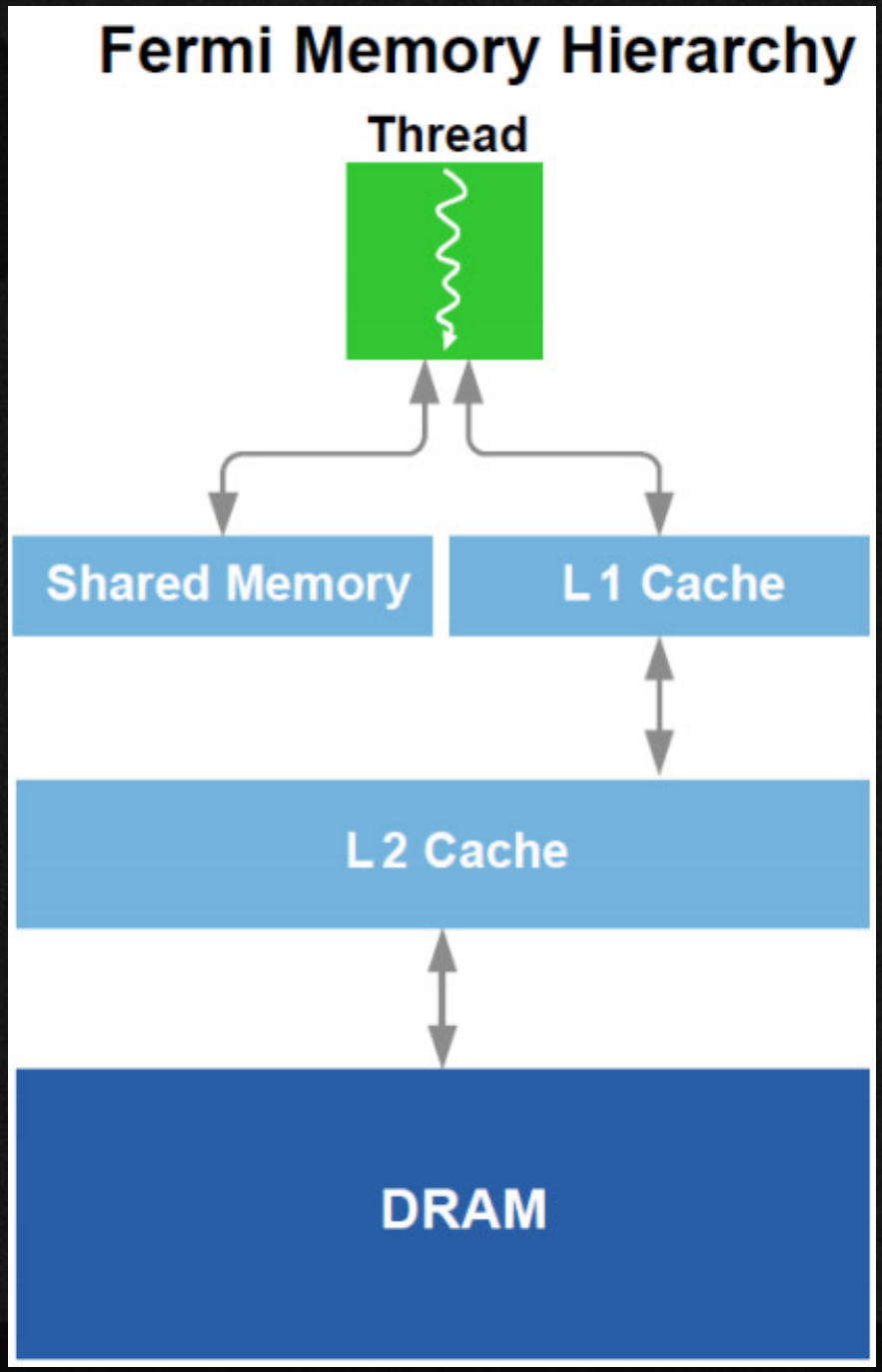
- Cache L2
  - 768 KB
  - Una per GPU
- *Indirizzi a 64-bit*



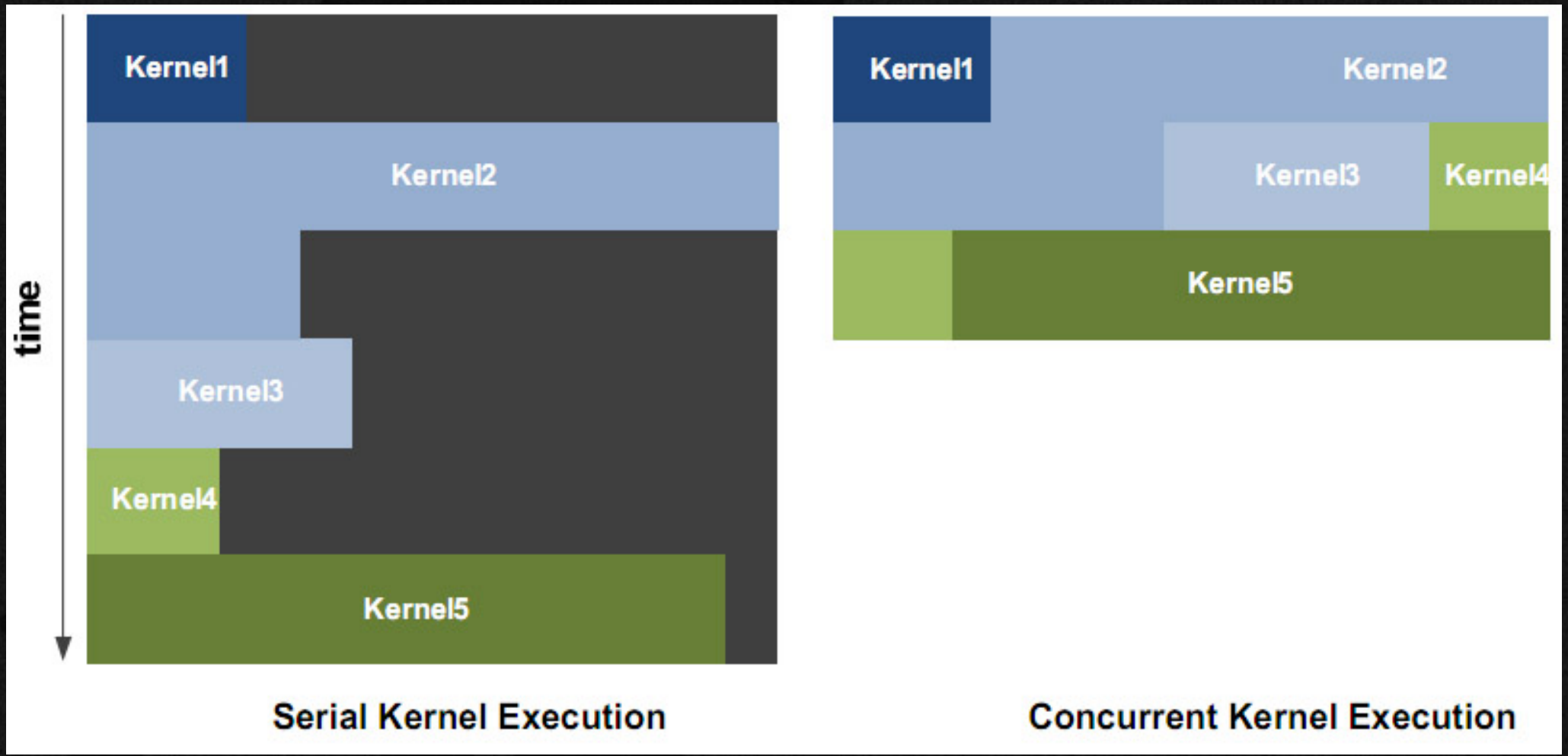
- Ram in modalità ***ECC*** (*Error-Correcting Code*)
  - 1 errore → *Errore corretto*
  - 2 errori → *Errore rilevato e segnalato*

Gerarchia di Cache

- Cache L1 per SM
  - Fino a 48 KB
- Cache L2 condivisa
  - 768 KB
- Le cache funzionano a linee di 128 Byte
- La cache L1 può essere disabilitata
  - Si cambia modalità di accesso: blocchi di 32B anziché 128B
  - La differenza è che si risparmia l'*overfetch* in caso di accessi *scattered* (*distanti* fra loro)

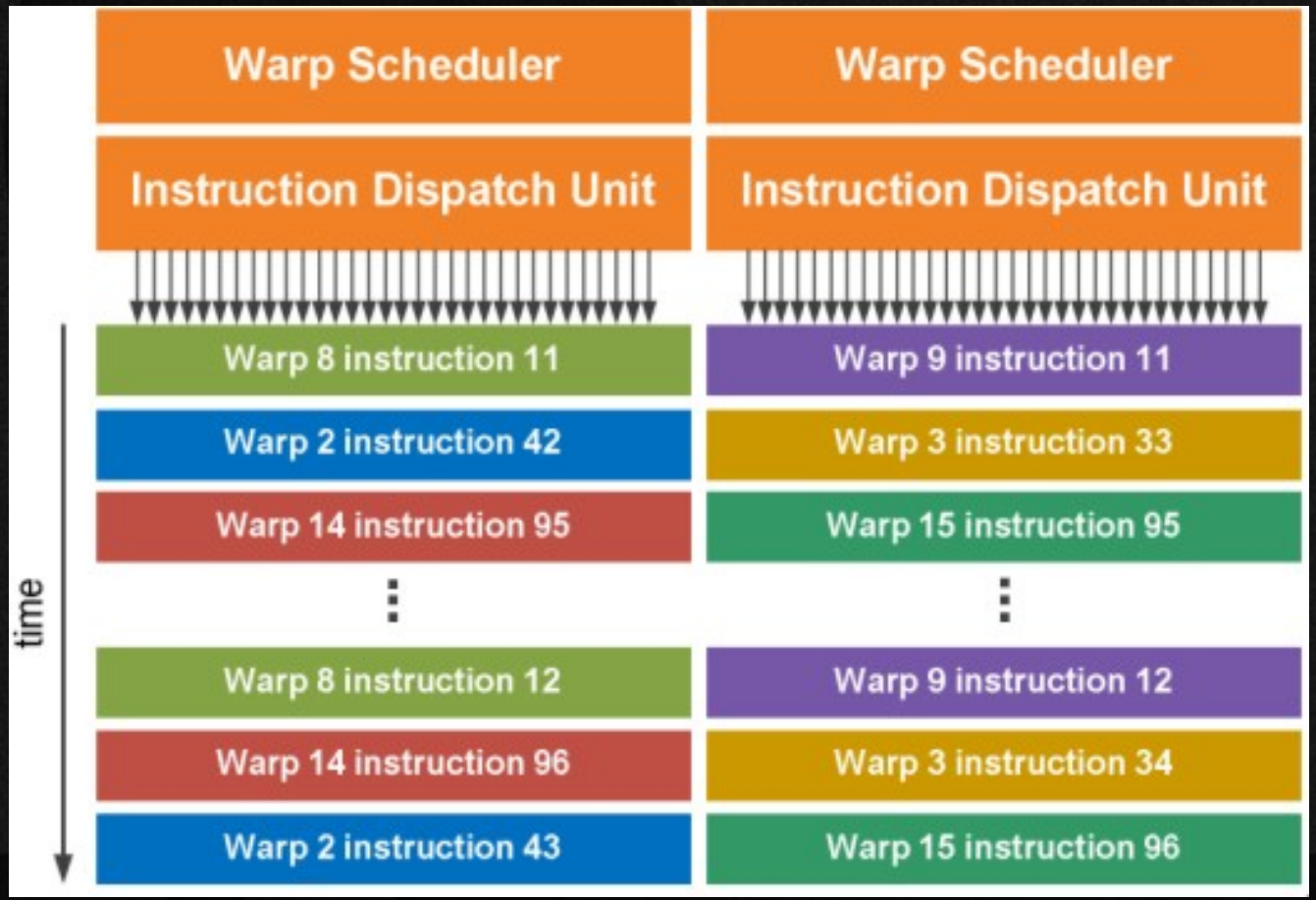


- Più kernel eseguiti in **concorrenza**
  - Fino a 16
  - La bassa efficienza di un kernel può essere coperta da altri kernel



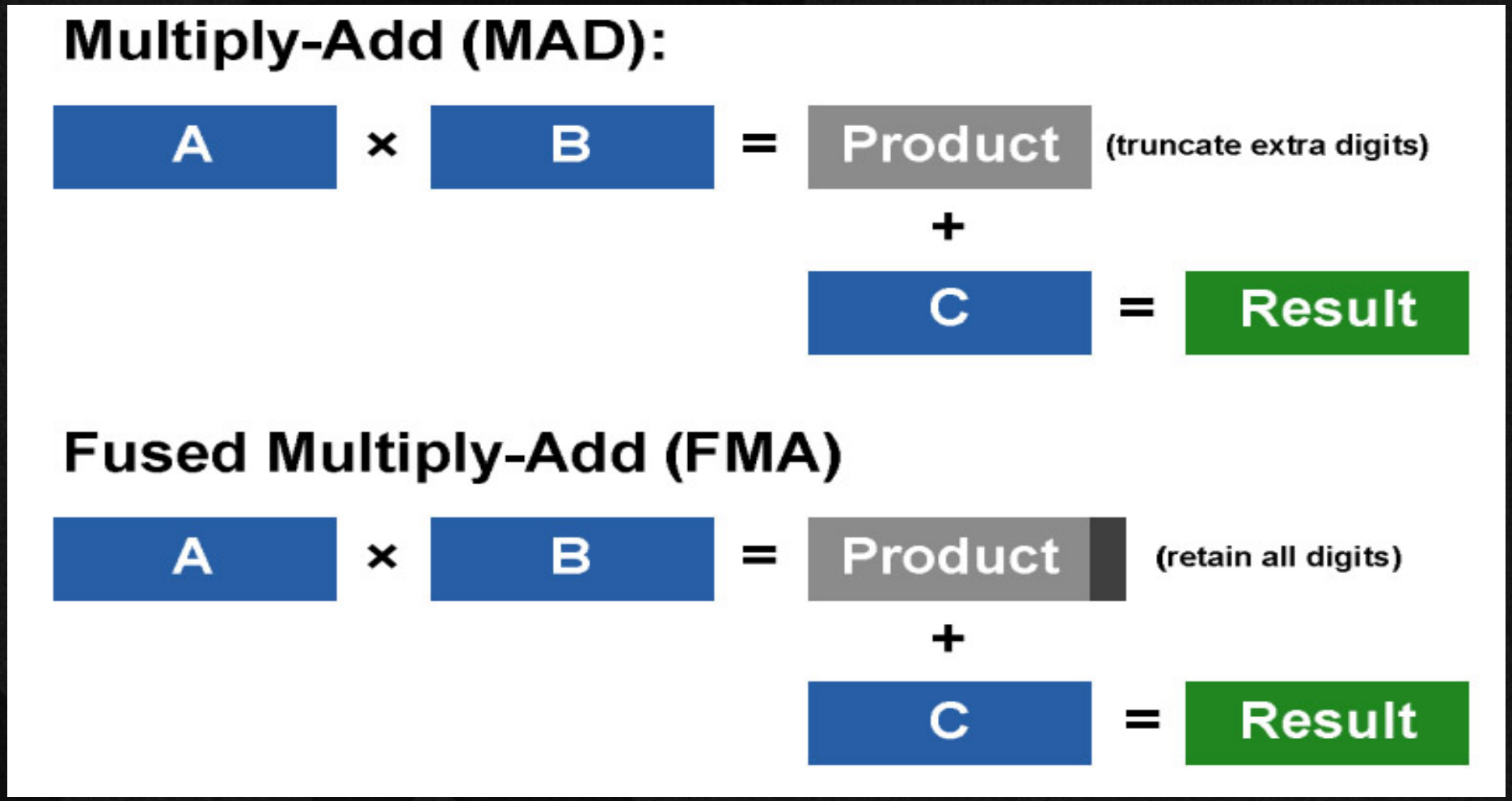
GF100

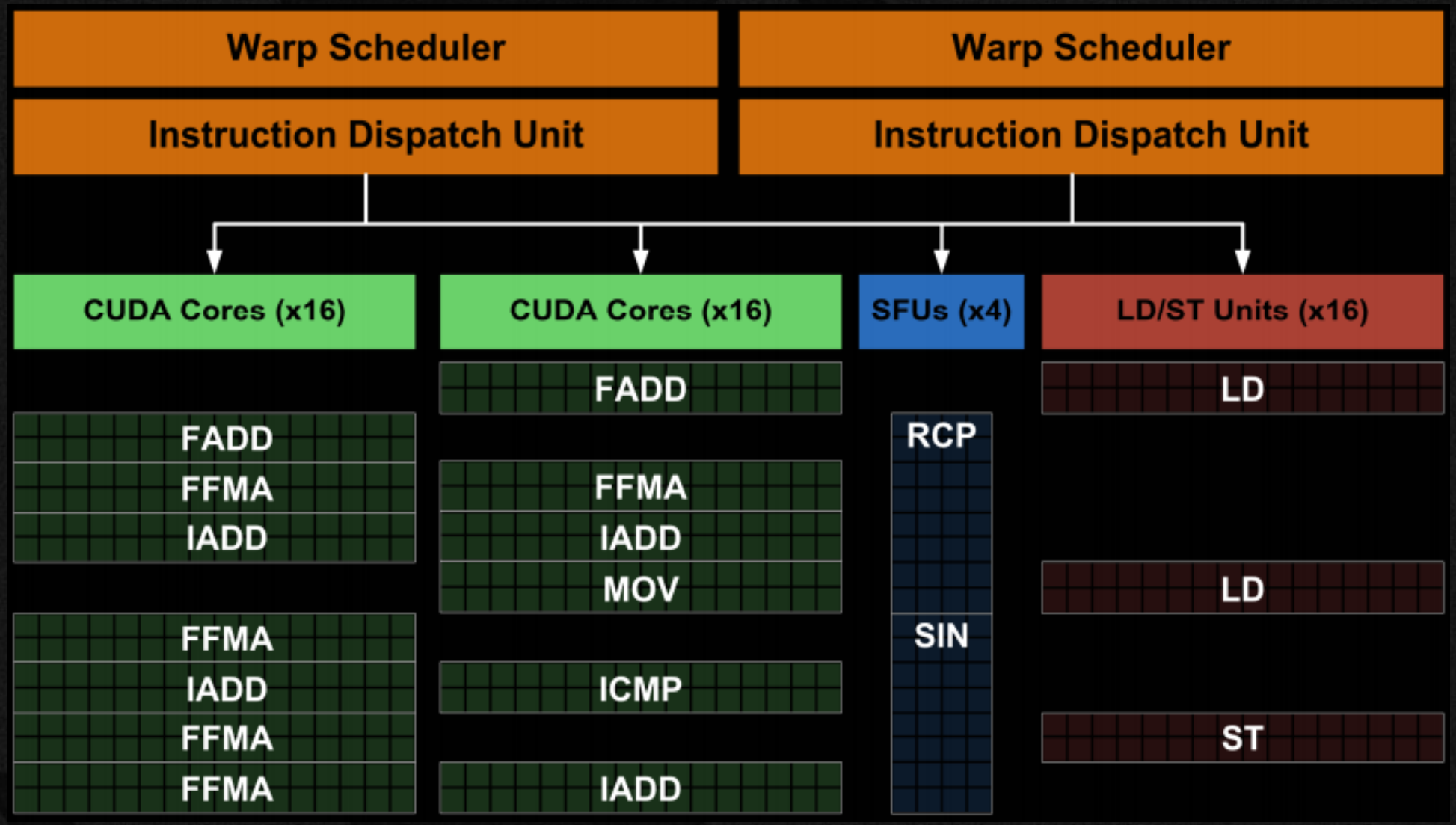
- 2 warp scheduler
  - Un'istruzione di un warp schedulata su 16 core in 2 cicli
  - Il primo scheduler si occupa dei warp con id dispari, l'altro i warp con id pari
  - Se l'istruzione è a doppia precisione, è attivo un solo warp scheduler alla volta



Standard IEEE754-2008

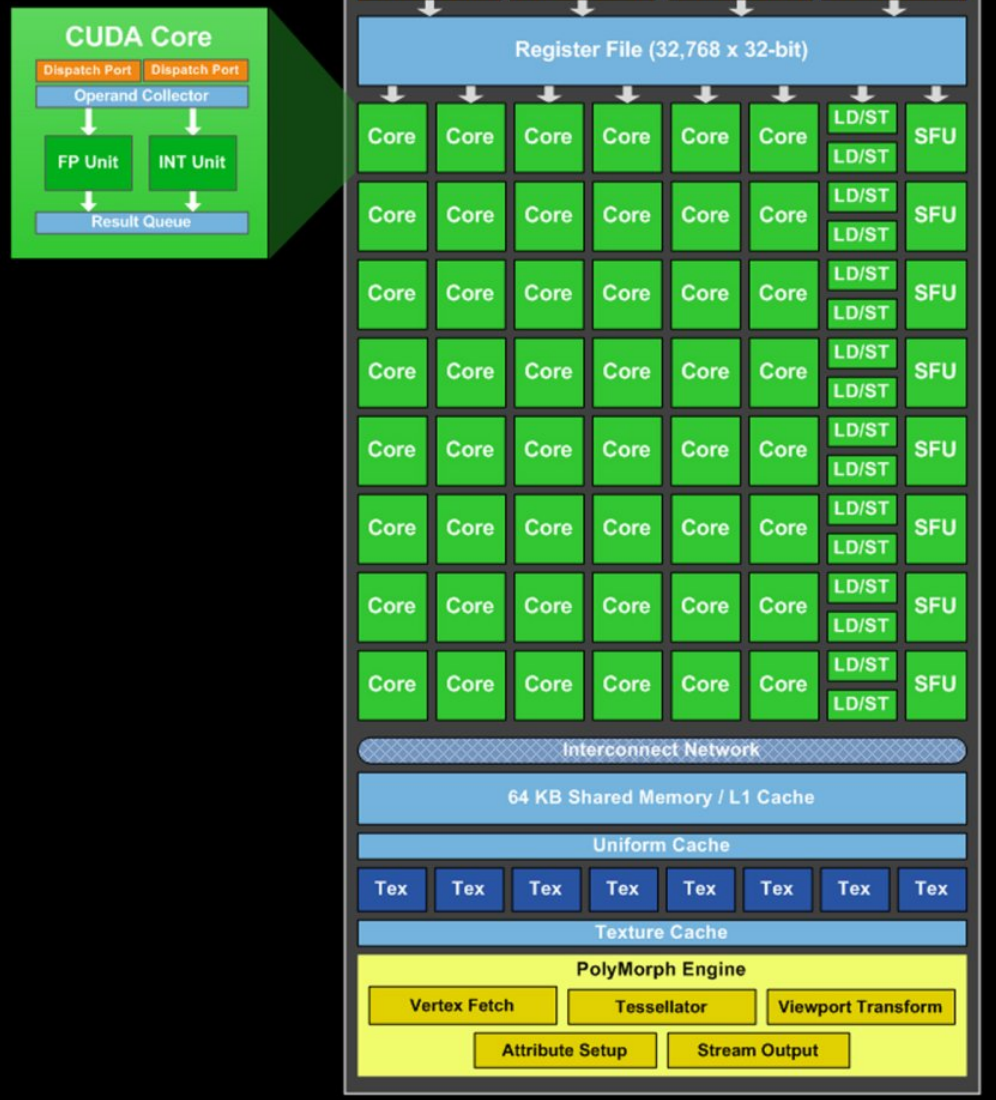
- Supporta i denormalizzati
- **FMA** (*Fused Multiply-Add*) al posto della **MAD**




Warp Scheduler
Warp Scheduler
Instruction Dispatch Unit
Instruction Dispatch Unit
CUDA Cores (x16)
CUDA Cores (x16)
SFUs (x4)
LD/ST Units (x16)
FADD
FFMA
IADD
FFMA
IADD
FFMA
FFMA
FADD
FFMA
IADD
MOV
ICMP
IADD
RCP
SIN
LD
LD
ST

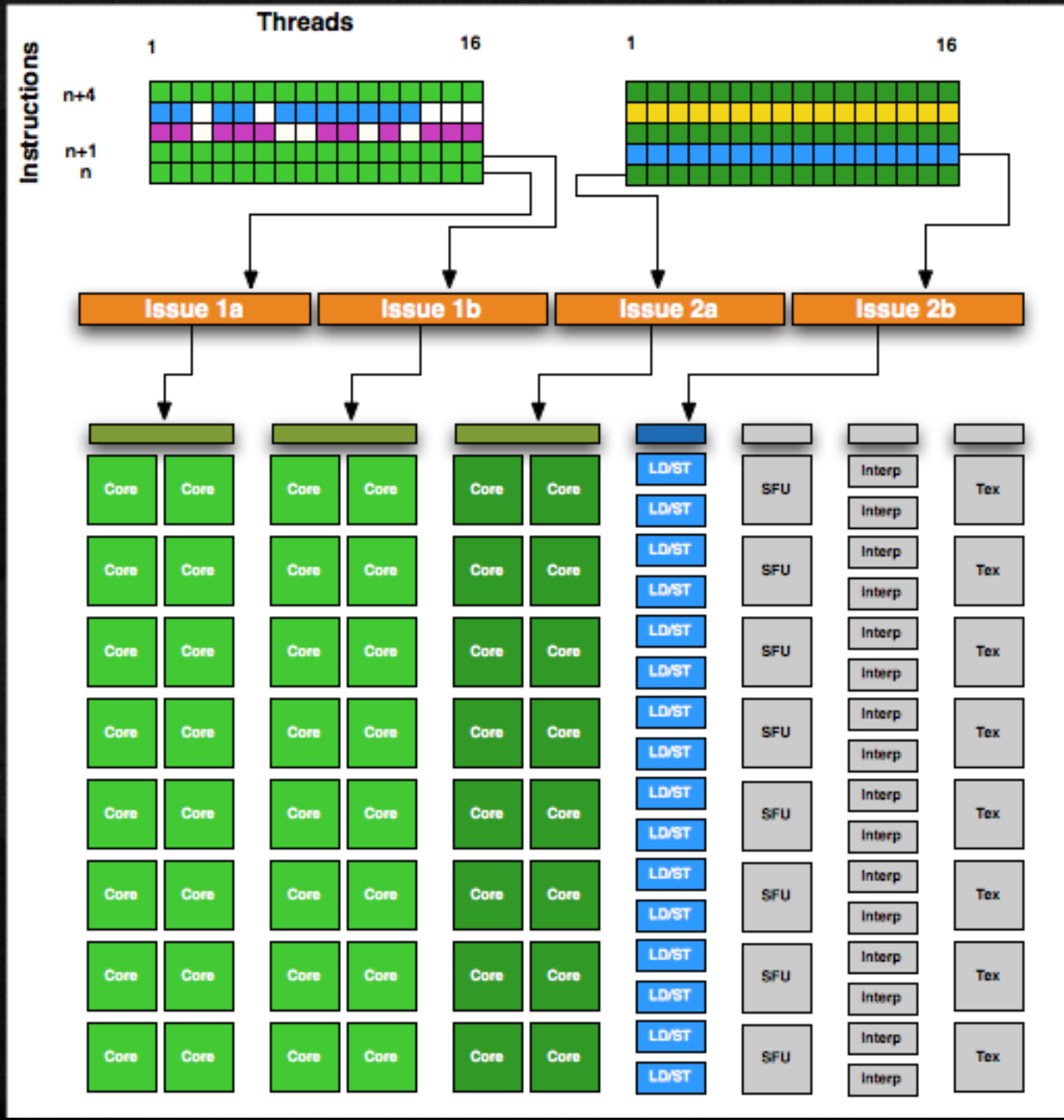
GF104

- 48 core per SM
- 8 special function unit
- 2 Warp scheduler
  - ***dual-issue***
    - *Due istruzioni indipendenti per Warp*



Dual issue

- *Ogni warp scheduler identifica 2 istruzioni **indipendenti** dello stesso warp*
- *Le schedula sulle unità disponibili*
- *Nello stesso ciclo di clock può avviare **4 istruzioni***




Ottimizzare gli
Accessi alla memoria

Memoria Globale (Tesla)

- Le richieste di load/store vengono divise in 2 richieste indipendenti
  - Si parla di ***mezzo warp***
    - L'accesso dei primi 16 thread non influenza quello dei secondi 16
- Ogni *mezzo warp* deve seguire delle **regole** precise per ottenere la massima efficienza negli accessi
  - Quando gli accessi seguono queste regole si dicono ***coalizzati o coalescenti*** *(coalesced)*
- Garantire accessi *coalizzati* è una delle priorità maggiori nell'ottimizzazione di un kernel
- Vedremo pattern di accesso più tardi

Accessi Coalizzati (c.c.1.0-1.1)

- Regole molto rigide

- Ogni thread può accedere ad una parola di:
  - 4 Byte
    - Le 16 parole devono stare in un segmento da 64 Byte
  - 8 Byte
    - Le 16 parole devono stare in un segmento da 128 Byte
  - 16 Byte
    - Prime 8 parole in un segmento da 128 Byte,
    - Seconde 8 nel segmento da 128 Byte successivo
    - Perché non 256B? (perché l'allineamento è a 128B non a 256B)
  - Il *k-esimo* thread deve accedere alla *k-esima* parola
  - Se una di queste regole non è rispettata
    → 16 transazioni da 32 Byte

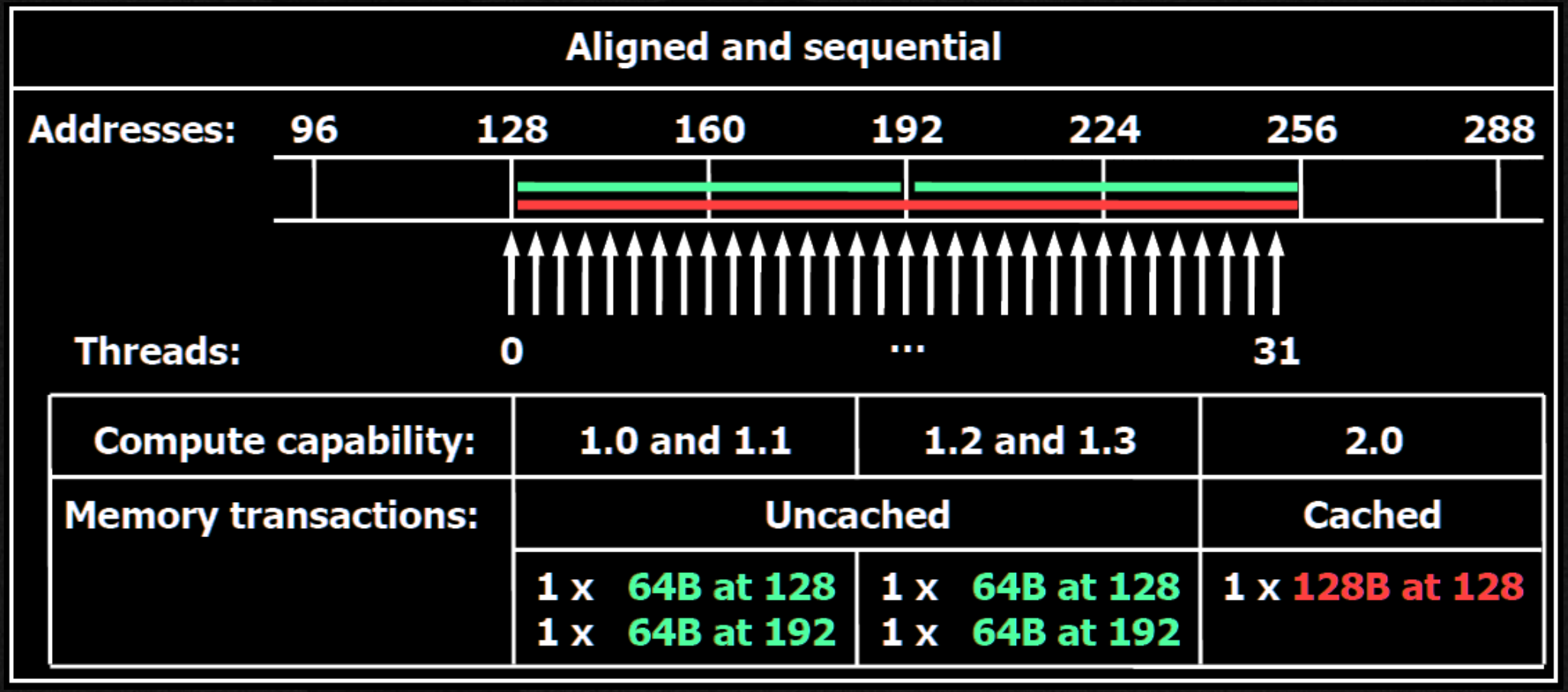
Accessi Coalizzati (c.c.1.2-1.3)

- Per questa serie di GPU, regole meno stringenti
- È possibile leggere anche parole da 1 o 2 Byte
- Niente limitazione su ordine accessi
- È sufficiente che i 16 thread accedano lo stesso segmento di 32 Byte, 64 Byte o 128 Byte
  - L'hardware automaticamente trova il segmento più piccolo che li contiene
- Se la regola non è rispettata:
  - Una transazione per ogni segmento da 128 Byte acceduto
  - Grandezza di ogni transazione ridotta a 32B o 64B se copre tutte le parole

Accessi Memoria Globale (Fermi)

- Gli accessi avvengono per linee di cache
  - 128 Byte per linea
  - L'hardware seleziona il numero minimo di linee di cache
- Le richieste avvengono per warp
  - Non più per mezzo warp

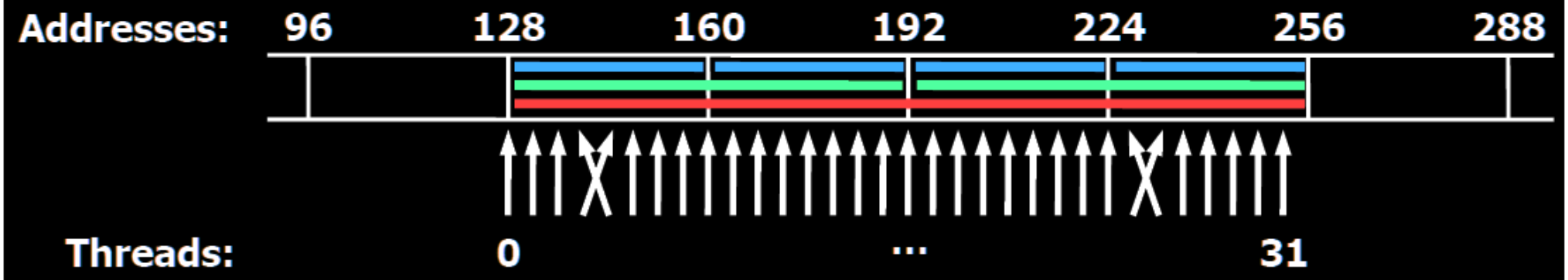
Accessi allineati e sequenziali



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Accessi allineati e non sequenziali

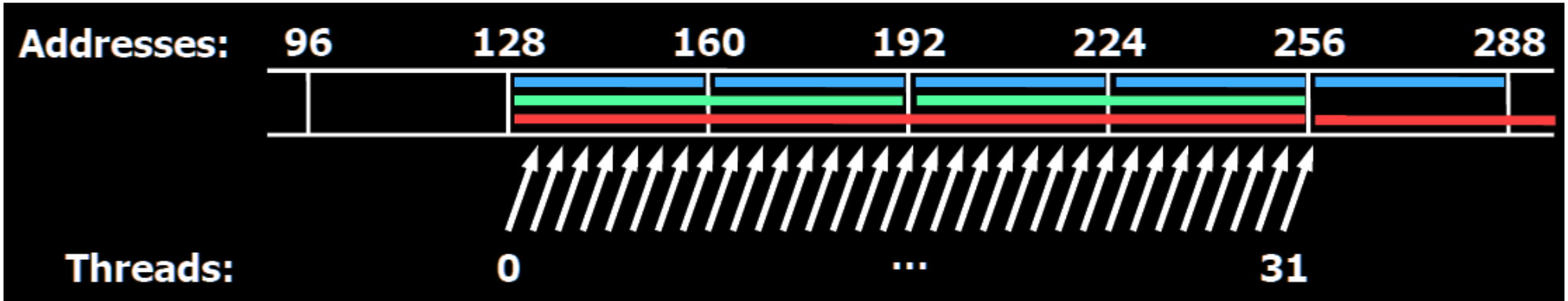
Aligned and non-sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Accessi non allineati e sequenziali

Misaligned and sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	7 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224 1 x 32B at 256	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256

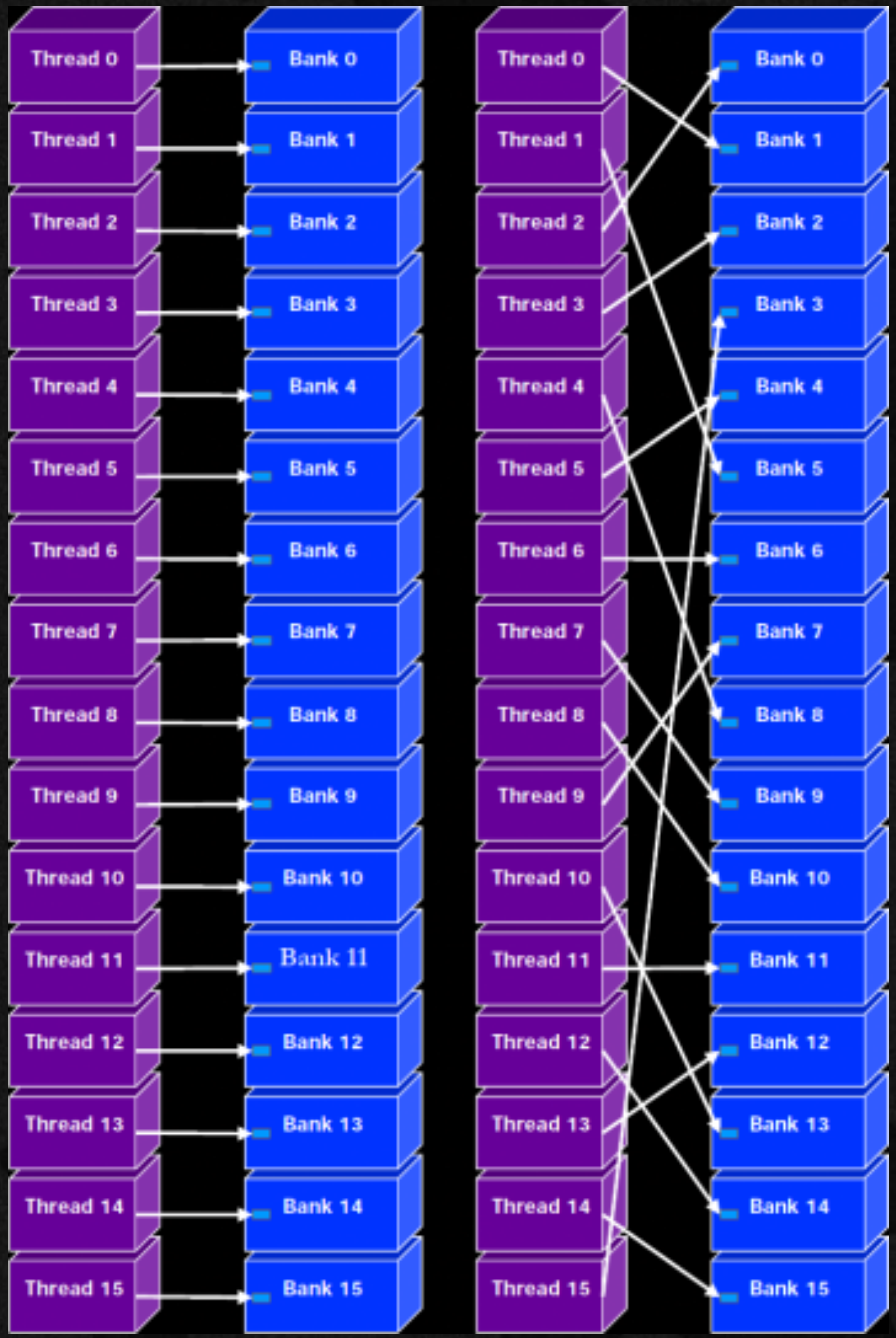
Memoria Condivisa

- Divisa in banchi

- Tesla
  - 16 banchi da 4 Byte
  - Acceduti in mezzo warp
    - Un accesso della prima metà di un warp non va in conflitto con uno della seconda metà
  - Ogni elemento letto → un banco diverso
    - Anche se accedono allo stesso elemento
    - Altrimenti: una transazione in più per ogni conflitto

- Fermi
  - 32 banchi da 4 Byte
  - Acceduti da un warp intero
    - Le due metà potrebbero avere conflitti
  - Conflitti solo se si accede a
    - Blocchi da 4 Byte differenti dello stesso banco
    - Elementi nello stesso blocco da 4 Byte non causano conflitto

Memoria Condivisa (Tesla)



- Esempi con nessun conflitto di banco
- Esempio 1:

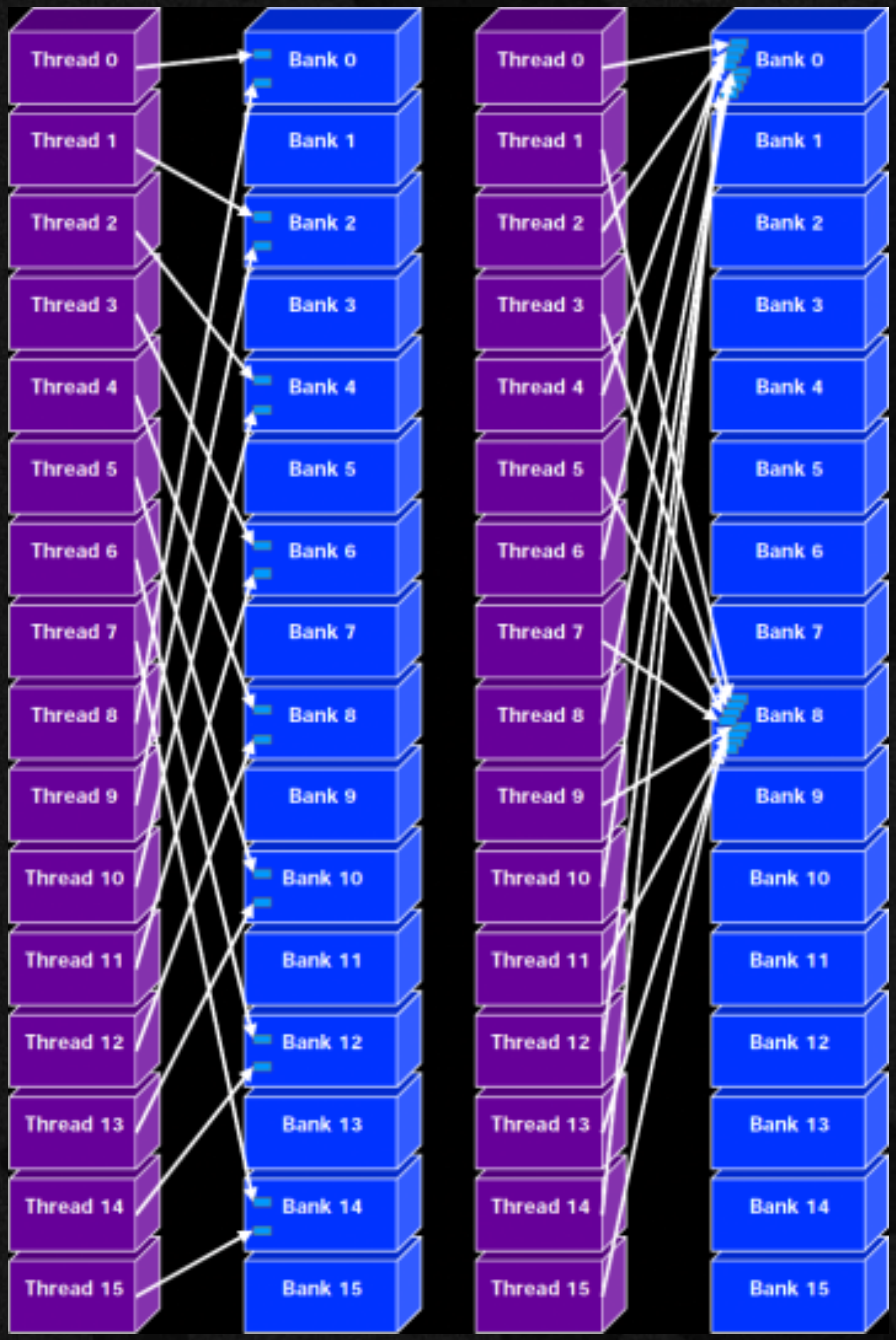
  float shared[32];

  x = shared[tId];
- Esempio 2:

  *float shared[k*32];*

  *x = shared[**k***tId];*
- ***k*** dispari

Memoria Condivisa (Tesla)



- Esempi conflitti di banco
- Esempio 1:

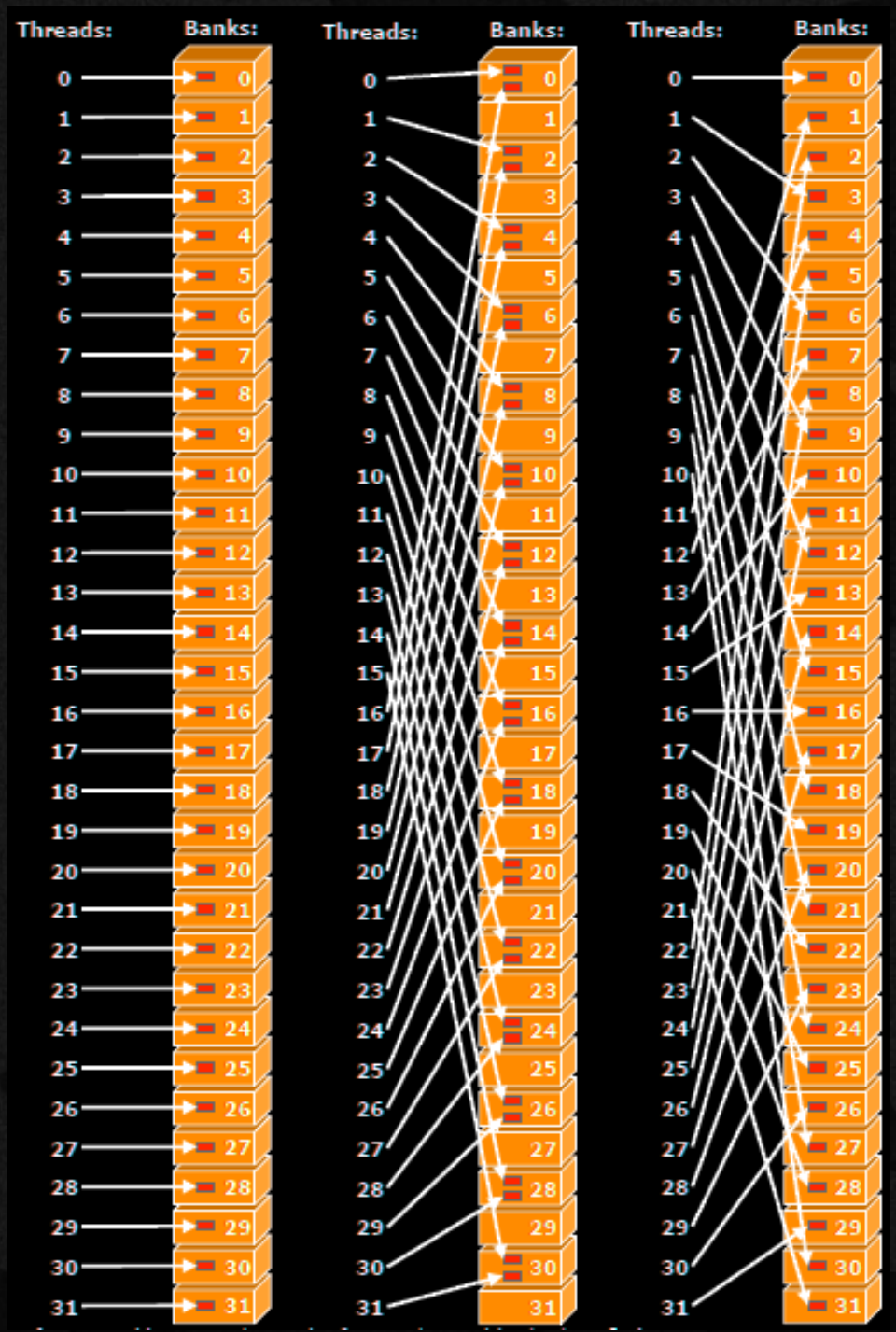
  float shared[64];

  *x = shared[2*tId];*
  - 2 transazioni
- Esempio 2:

  float shared[4];

  x = shared[tId/8];
  - *8 transazioni*

Memoria Condivisa (Fermi)

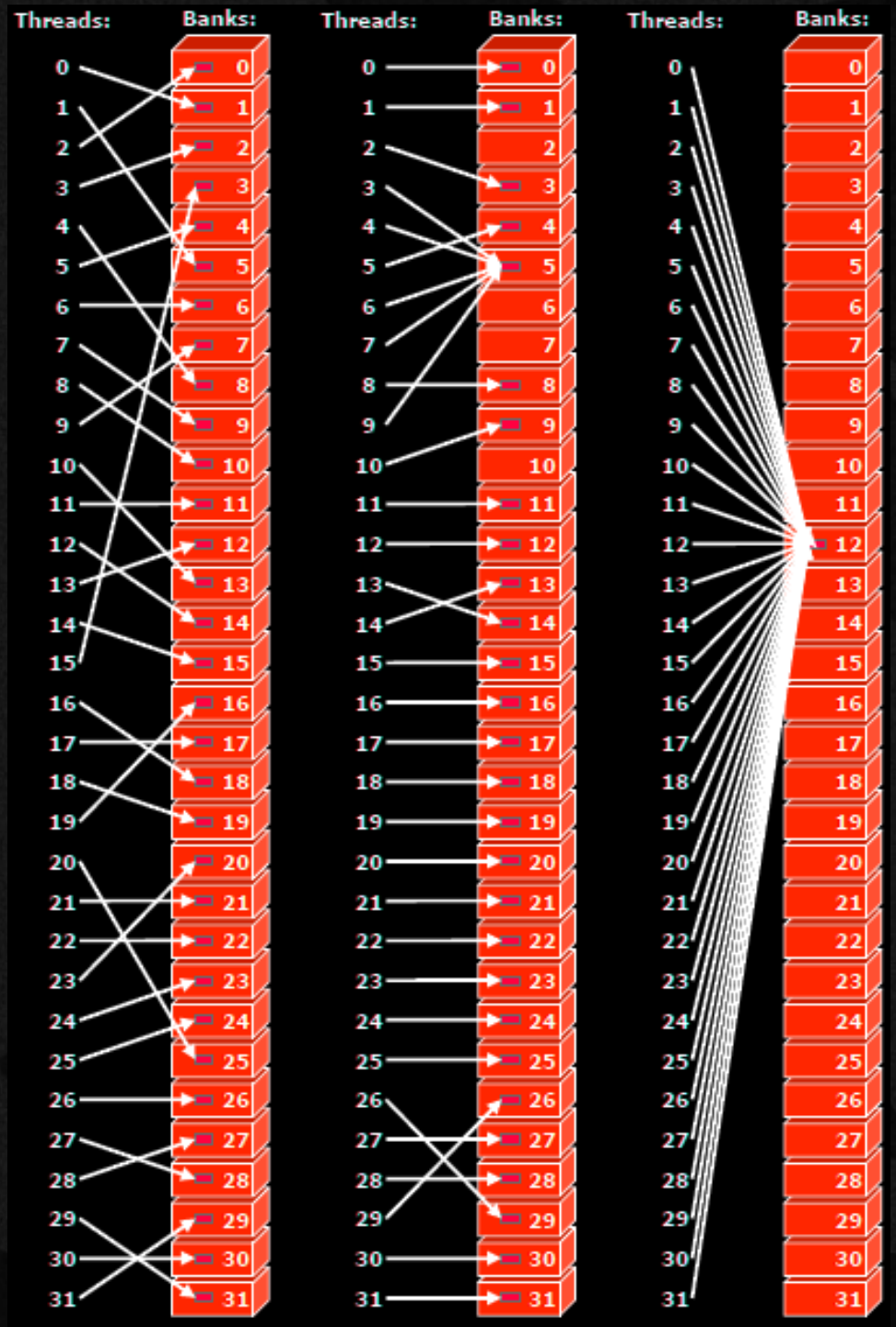


- Esempio 1:
  float shared[32];
  x = shared[tId];

- Esempio 2:
  float shared[64];
  *x = shared[2*tId];*
  - 2 transazioni
- x = shared[(2*tId) % 64]
  - 1 transazione

- Esempio 3:
  float shared[96];
  *x = shared[3*tId];*

Memoria Condivisa (Fermi)



- Esempio 1:
  - Accesso non sequenziale senza conflitti di banco
  - 1 sola transazione
- Esempio 2:
  - Accesso allo stesso blocco da 128B → nessun conflitto
- Esempio 3:

```
float shared;
x = shared;
```

  - 1 sola transazione

```
float shared[32*32];
x = shared[tId*32];
```

  - 32 transazioni

Threads per blocco

- Scegliete un numero di threads per blocco multiplo del warp (32 thread)
  - Facilita accessi alla memoria efficienti
  - Evita inutili warp *sotto-popolati*
- Su Fermi i warp pari vengono gestiti da un *warp scheduler,* quelli dispari da un altro
  - Scegliete dimensioni multiple di 2*32 = 64
- Per un kernel con collo di bottiglia su istruzioni ALU (ovvero che *nasconde* gli accessi alla memoria)
  - 64 o 128 spesso è la scelta migliore