

GPU Computing

Laurea Magistrale in Informatica - AA 2019/20

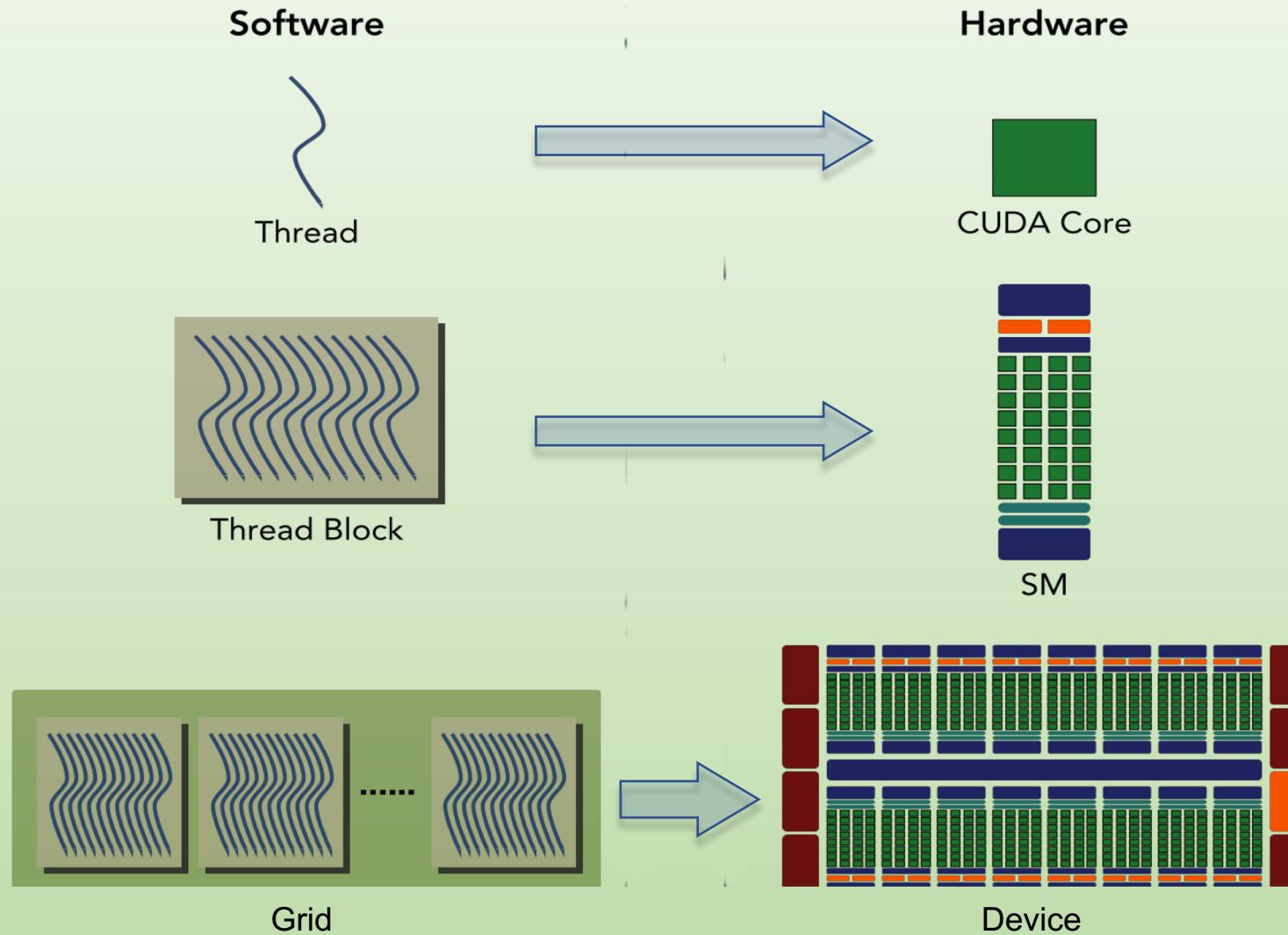
Docente **G. Grossi**

Lezione 3 – Il modello di esecuzione CUDA

Sommario

- ✓ Elementi dell'architettura GPU
- ✓ Modello di esecuzione di thread, warp e blocchi
- ✓ divergenza nei warp
- ✓ sincronizzazione a livello di blocco
- ✓ Il problema parallel reduction

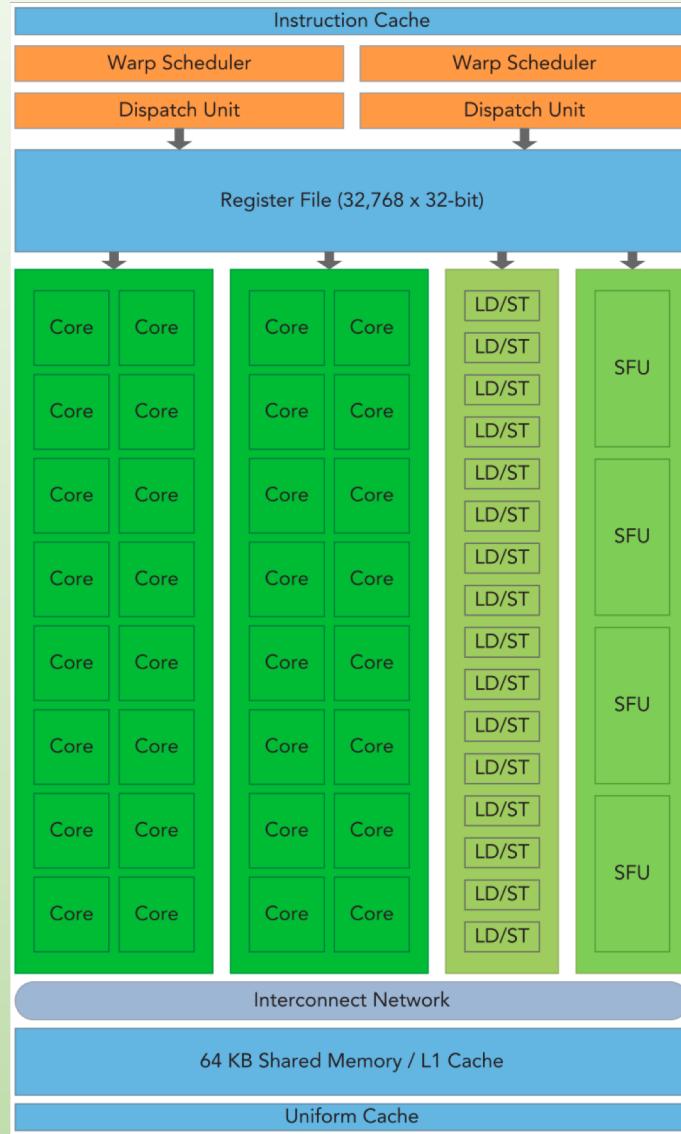
Mapping logico – fisico dei thread



Streaming Multiprocessors (SM)

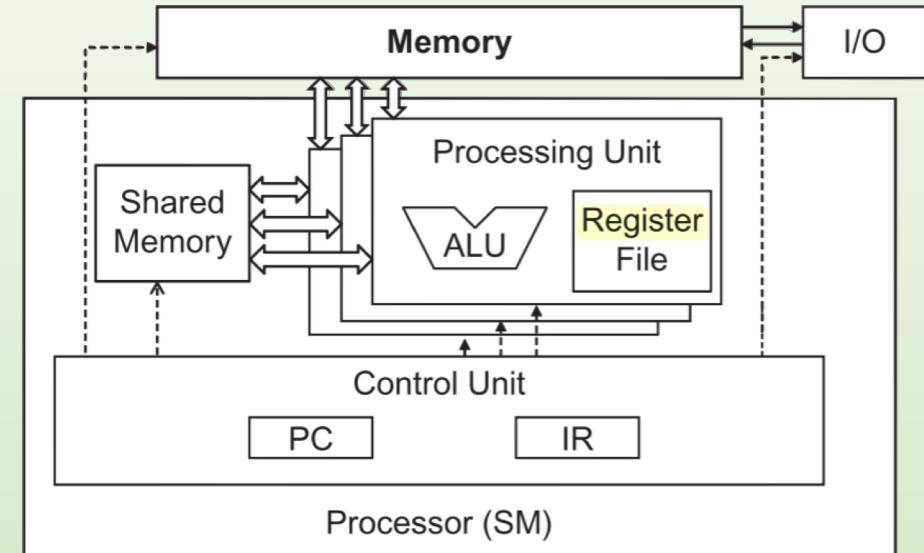
- ✓ L'architettura della GPU è disegnata come un array scalabile di **Streaming Multiprocessors (SM)**
- ✓ Il **parallelismo hardware** della GPU è ottenuto **replicando questo elemento base**
- ✓ Qui sono illustrate le componenti chiave dell'architettura **SM Fermi** (la prima!)

- CUDA Cores
- Shared Memory/L1 Cache
- Register File
- Load/Store Units
- Special Function Units
- Warp Scheduler



HW multi-threading

- ✓ SIMT architecture – warp level
- ✓ Ogni warp ha una **contesto di esecuzione** (a runtime) che risulta trasparente al programmatore e consta di:
 - **Program counters**
 - **Registri** a 32-bit ripartiti tra thread
 - **Shared memory** ripartita tra blocchi
- ✓ I registri sono usati per le variabili locali automatiche scalari (che non sono array quindi) e le coordinate dei thread (per es. **Row**, **Col** e **val** nel kernel per il prodotto di matrici)
- ✓ La shared memory permette di **condividere** memoria tra thread del blocco che possono così scambiare tra loro informazioni (vedi prossime lezioni...)
- ✓ I dati nei registri sono **privati** ai thread (scope) e il lifetime è quello del kernel
- ✓ Il contesto x ogni warp è mantenuto **on-chip** per la durata del warp (switch tra contesti ha costo zero!)

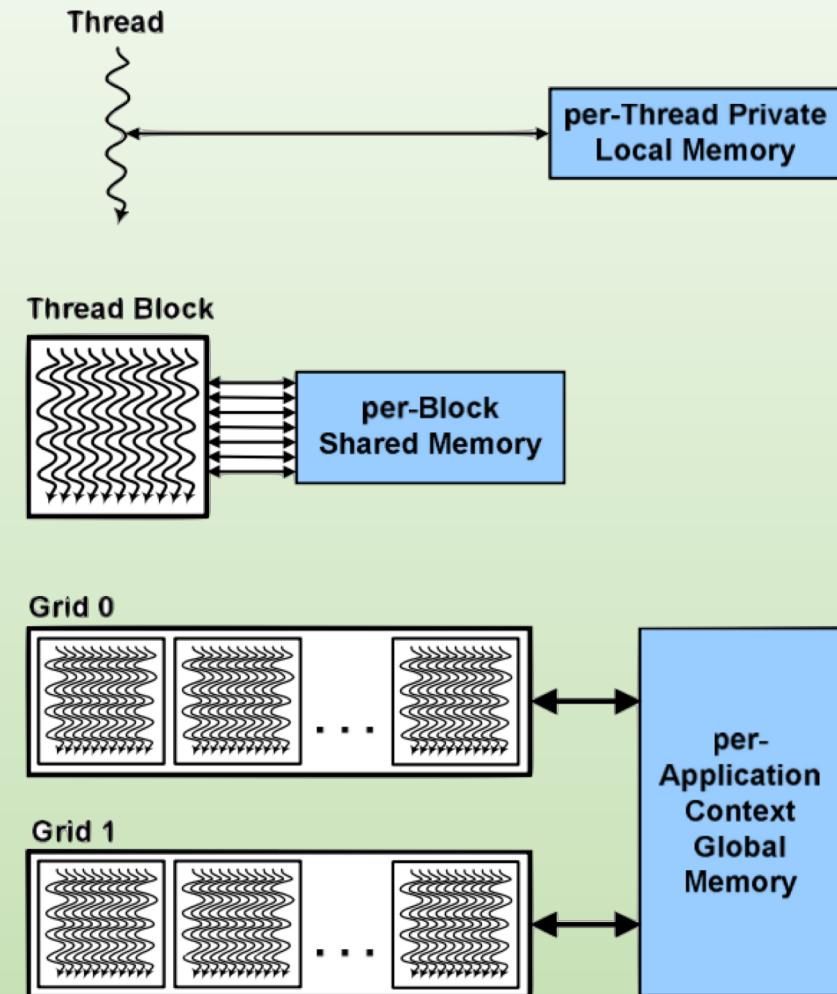


```
__global__ void matrix_prod(float* A, float* B, float* C) {
    // indici di riga e colonna
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    // each thread computes an entry of the product matrix
    if ((Row < N) && (Col < M)) {
        float val = 0;
        for (int k = 0; k < M; k++)
            val += A[Row * M + k] * B[k * M + Col];
        C[Row * M + Col] = val;
    }
}
```

Panoramica

- ✓ Ogni **thread** viene eseguito su un CUDA core ed ha un suo spazio privato di **memoria** per **registri, chiamate di funzioni e variabili C automatiche**
- ✓ Un **thread block** è un gruppo di thread eseguiti **concorrentemente** che può cooperare attraverso **barriere di sincronizzazione**
- ✓ Un thread block usa **shared memory** per la comunicazione **inter-thread** e condivisione dati
- ✓ Una **grid** è un **array di thread block** che eseguono tutti lo stesso **kernel**, legge e scrive in global memory e sincronizza le chiamate di kernel tra loro dipendenti

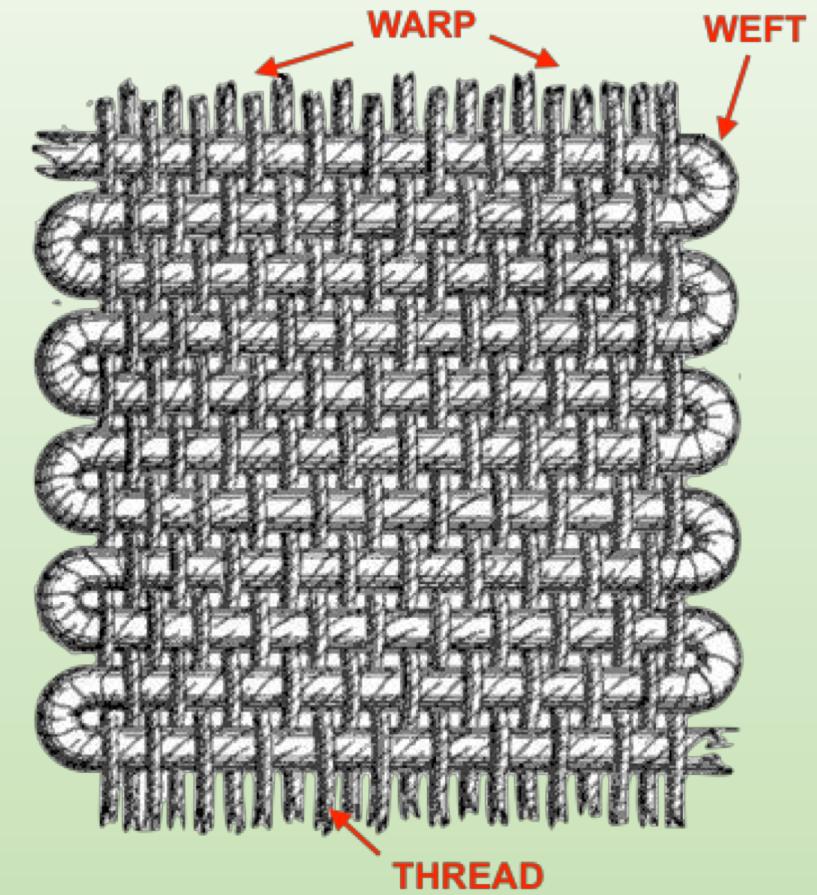


Esecuzione

- ✓ Viene mappata la **gerarchia di thread** nella **gerarchia di processori** sulla GPU
- ✓ La GPU esegue uno o più **kernel**
- ✓ Uno streaming multiprocessor (**SM**) esegue uno più **thread block**
- ✓ Un thread **block** è **schedulato** solo su un SM
- ✓ I **core** e altre unità di esecuzione nello SM eseguono i thread
- ✓ Gli SM eseguono i thread a gruppi di **32 thread** chiamati **warp**

Warp: architettura SIMT

- ✓ Numero magico: **warp = 32 thread (consecutivi)**
- ✓ Idealmente tutti i thread in un warp eseguono in **parallelo** allo stesso tempo (modello SIMD)
- ✓ Ogni thread ha il suo **program counter** e **register state**, ed esegue l'istruzione corrente su dati assegnati
- ✓ Ogni thread può seguire **cammini distinti** di esecuzione delle istruzioni (parallelismo a livello thread)
- ✓ I thread che compongono un **warp** iniziano assieme alla stessa indirizzo del programma, ma ognuno è libero di seguire path propri indipendentemente da altri thread, seguendo flussi distinti
- ✓ **Architettura SIMT:** Single Instruction Multiple Thread
Una differenza chiave è che SIMD espone l'intero set di dati (vettore) all'istruzione che deve essere eseguita, mentre SIMT specifica il comportamento di esecuzione e branching di un singolo thread



32... un numero magico!

- ✓ Il numero 32 ha origine dall'**hardware** ed è fondamentale nella programmazione CUDA
- ✓ E' l'**unità minima di esecuzione** che permette grande efficienza nell'uso della GPU
- ✓ Ha un forte impatto sulle **prestazioni** degli algoritmi sviluppati
- ✓ Concettualmente ha un comportamento modello **SIMD**
- ✓ In pratica assume modello **SIMT** (da evitare!)

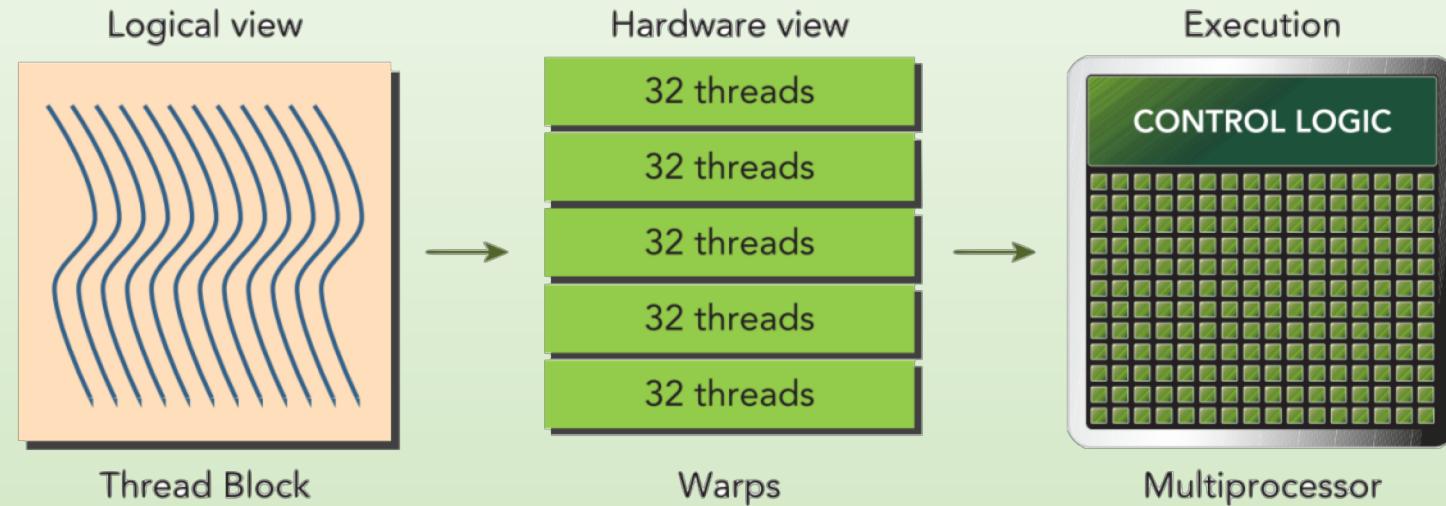
Vista logica/HW dei warp

Prospettiva logica

- ✓ I blocchi di thread possono essere configurati con ID nelle 3 dimensioni: 1D, 2D o 3D

Prospettiva hardware

- ✓ Tutti i thread sono organizzati in una sola dimensione con un ID progressivo



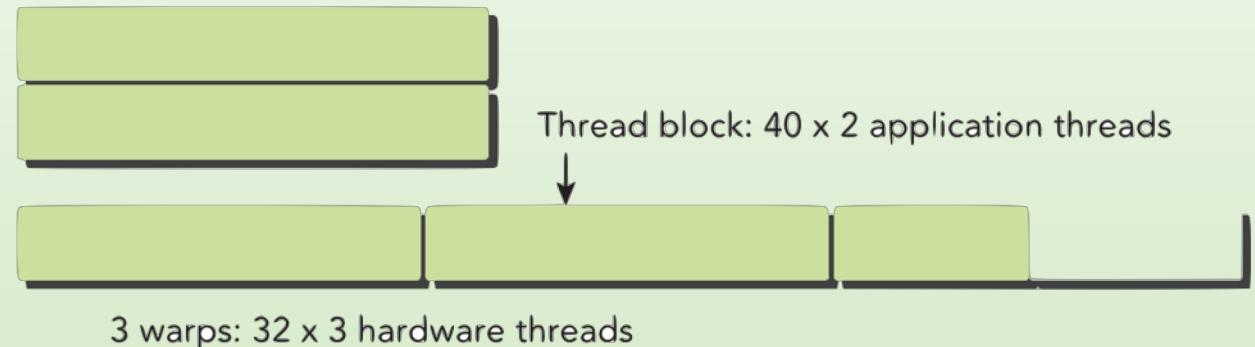
Esempio

- ✓ un blocco con 128 thread verrà suddiviso in 4 warp: warp0, warp1, warp2, warp3

```
Warp 0: thread 0, thread 1, thread 2, ... thread 31
Warp 1: thread 32, thread 33, thread 34, ... thread 63
Warp 2: thread 64, thread 65, thread 66, ... thread 95
Warp 3: thread 96, thread 97, thread 98, ... thread 127
```

Esempio: numero di warp

- ✓ **Vista logica:** un blocco di thread bidimensionale con 40 thread nella dimensione x e 2 thread nella dimensione y
- ✓ **Vista HW:** verranno allocati 3 warp HW per il blocco, con un totale di 96 thread per supportarne 80 SW



Nota che l'ultimo semi-warp (16 thread) è inattivo e comunque consumano risorse anche quelli inattivi, registri, memoria, etc.

- ✓ Numero di warp determinati per blocco:

$$\text{warpPerBlock} = \left\lceil \frac{\text{threadPerBlock}}{\text{warpSize}} \right\rceil = \left\lceil \frac{40}{32} \right\rceil = 2$$

Thread, Warp status

- ✓ Un blocco è **attivo** dopo che le risorse di computazione (registri e shared mem.) gli sono assegnate... i warp contenuti sono anch'essi attivi
- ✓ Un **warp attivo** può essere di tre tipi:
 - **Selezionato**: in esecuzione su un dato path (preso in carico dallo scheduler di warp)
 - **Bloccato**: non pronto per l'esecuzione
 - **Candidato**: eleggibile se 32 core sono liberi e tutti gli argomenti della prossima istruzione sono disponibili
- ✓ Thread è nello stato di
 - **attivo**: quando appartiene al path in esecuzione del warp corrente
 - **inattivo**: quando non nel path in exec, uscito prima di altri dal warp, ultimo di blocchi non multipli di 32
- ✓ Il numero di **warp** per un **thread block** è determinato come segue:

$$\# \text{Warps} \times \text{Block} = \left\lceil \frac{\#\text{Threads} \times \text{Block}}{\text{warpSize}} \right\rceil \quad (< 32)$$

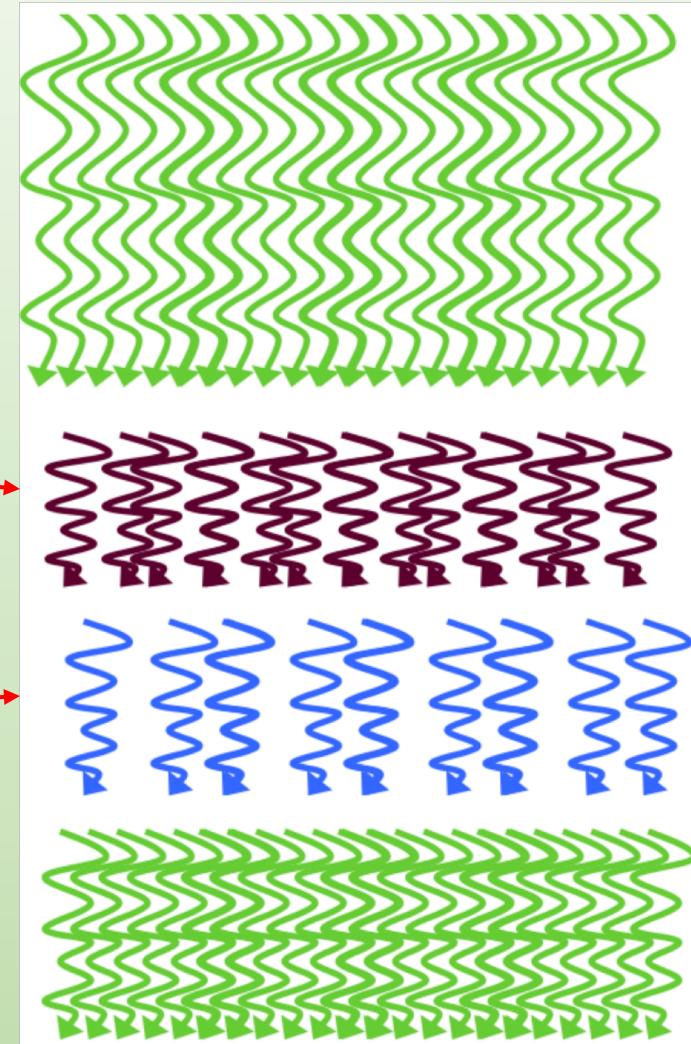
Warp divergence

```
if (cond)
{
    ...
} else {
    ...
}
```

- ✓ La CPU include complessi meccanismi hardware per effettuare **branch prediction**, cioè predire a ogni step condizionale quale branch il flusso di controllo intraprenderà
- ✓ Se la predizione non è corretta la CPU può rimanere bloccata per un **elevato numero di cicli di clock** per resettare la pipeline
- ✓ La GPU sono dispositivi più semplici **senza** meccanismo di complessi **branch prediction**
- ✓ Poiché (in teoria) tutti i thread in un warp devono eseguire la medesima istruzione nello stesso ciclo, se un gruppo di thread, all'interno dello stesso warp, prende un differente path, allora siamo in presenza di quello che viene chiamato **warp divergence**

Divergenza ed esecuzione

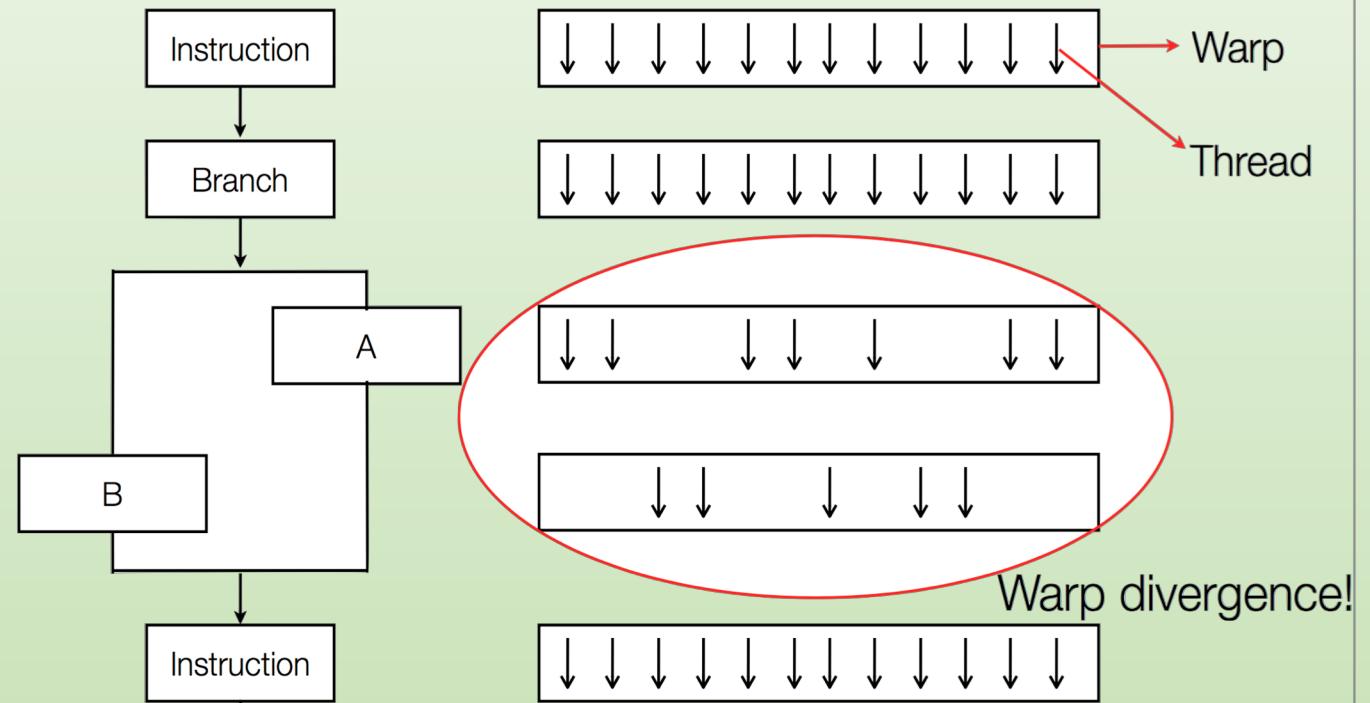
```
__global__ void kernel(int* x, int* y) {  
    int i = threadIdx.x + blockDim.x * blockIdx.x;  
    int t;  
    bool b = f(x[i]);  
    if( b ) {  
        // evaluate g(x)  
        t = g(x[i]);  
    }  
    else {  
        // evaluate h(x)  
        t = h(x[i]));  
    }  
    y[i] = t;  
}
```



Serializzazione nella warp divergence

- ✓ Se I thread di un warp divergono, il warp **esegue serialmente** ogni branch path, disabilitando I thread che non appartengono a quel dato path
- ✓ La divergenza nei warp introduce una significativa **degradazione** delle prestazioni... con un numero elevato di branch condizionali, la perdita di parallelismo potrebbe essere insostenibile (fino a 32 volte!)
- ✓ Si noti che un il fenomeno della divergenza occorre solo **all'interno** di un warp
- ✓ I passi condizionali in **differenti** warp non causano divergenza

Esempio di serializzazione di warp:



Caso peggiore

Le prestazioni **decrescono** con il grado di **divergenza** nei warps

Warp size!

```
__global__ void dv(int* x) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    switch (i % 32) {
        case 0 :
            x[i] = a(x[i]);
            break;
        case 1 :
            x[i] = b(x[i]);
            break;
        . . .
        case 31:
            x[i] = v(x[i]);
            break;
    }
}
```

Esempio: divergenza nei warp

Warp divergence:
16 thread su branch (a = 2)
16 thread su branch (b = 1)

```
/*
 * Kernel con divergenza dei warp
 */
__global__ void pari_dispari_1(int *c) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int a = 0, b = 0;

    if (tid % 2 == 0)
        a = 2;           // warp pari
    else
        b = 1;           // warp dispari
    c[tid] = a + b;
}
```

ESERCITAZIONE:

1. E' possibile eliminare la divergenza dei warp in questo caso? In quali modi?
2. Misurare la differenza in tempo con la soluzione trovata priva di divergenza con
`nvprof --metrics branch_efficiency ./app_name`

Branch efficiency

- ✓ Misurare l'efficienza in termini di **warp divergence**:

```
$ nvprof --metrics branch_efficiency ./Application
```

Invocations	Metric Name	Metric Description	Min	Max	Avg
Device "Tesla M2090 (0)"					
Kernel: evenODD2(int*, int)					
1	branch_efficiency	Branch Efficiency	99.83%	99.83%	99.83%
Kernel: evenODD1(int*)					
1	branch_efficiency	Branch Efficiency	100.00%	100.00%	100.00%

- ✓ Impedire l'**ottimizzazione** del compilatore pars **-g** e **-G**:

```
$ nvcc -g -G application.cu -o Application
```

Invocations	Metric Name	Metric Description	Min	Max	Avg
Device "Tesla M2090 (0)"					
Kernel: evenODD2(int*, int)					
1	branch_efficiency	Branch Efficiency	99.83%	99.83%	99.83%
Kernel: evenODD1(int*)					
1	branch_efficiency	Branch Efficiency	83.33%	83.33%	83.33%

Branch efficiency: numero

$$\text{\#Branch Efficiency} = 100 \times \left[\frac{\text{\#Branches} - \text{\#DivergentBranches}}{\text{\#Branches}} \right]$$

- ✓ Il compilatore interviene per ovviare la divergenza nei branch
- ✓ Usa fare sostituzioni di path condizionali a patto che il corpo del branch sia piccolo

```
$ nvprof --events branch,divergent_branch ./Application
```

Invocations	Event Name	Min	Max	Avg	Total
Device "GeForce GT 650M (0)"					
Kernel: evenODD1(int*, int)					
1	branch	229376	229376	229376	229376
1	divergent_branch	32768	32768	32768	32768
Kernel: evenODD2(int*, int)					
1	branch	327680	327680	327680	327680
1	divergent_branch	0	0	0	0

Esempio: eliminazione della divergenza

APPROCCIO: usare
granularità dei warp e non
quella dei thread!
Tutto si svolge a livello di
BLOCCO!

wid è l'indice di warp
all'interno del blocco

- Uso i warp a coppie (64 thread) per calcolare l'indice **i** del vettore c:
- Il primo warp calcola gli indici pari **0,2,4,...,62**
 - Il secondo warp calcola gli indici dispari **1,3,5,...,63**
 - Il pari e dispari della coppia si ha con **wid%2**
 - La formula che rende questo è **2*(tid%warpSize)**
 - L'offset viene calcolato modulo 64 (2 warpSize): **(tid/64)*64**

```
/*
 * Kernel con divergenza dei warp risolta (solo se N > 64)
 */
__global__ void pari_dispari_2(int *c, int N) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int a = 0, b = 0, i;

    int wid = tid / warpSize; // indice dei warp wid = 0,1,2,3, ...
    if (!(wid % 2))
        a = 2; // Branch1: thread tid = 0-31, 64-95, ...
    else
        b = 1; // Branch2: thread tid = 32-63, 96-127, ...

    // right index
    if (!(wid%2)) // even
        i = 2*(tid%32) + (tid/64)*64;
    else // odd
        i = 2*(tid%32)+1 + (tid/64)*64;
    if (i<N)
        c[i] = a + b;
}
```

Ottimizzazione nvcc x divergenza

- ✓ CUDA introduce i **predicati** che sono istruzioni che vengono eseguite se e solo se la **flag** è **true**

```
p: a=b+c;      //computed only if p is true
```

- ✓ Esempio:

```
if (x < 0.0)
    z = x - 2.0;
else
    z = sqrt(x);
```

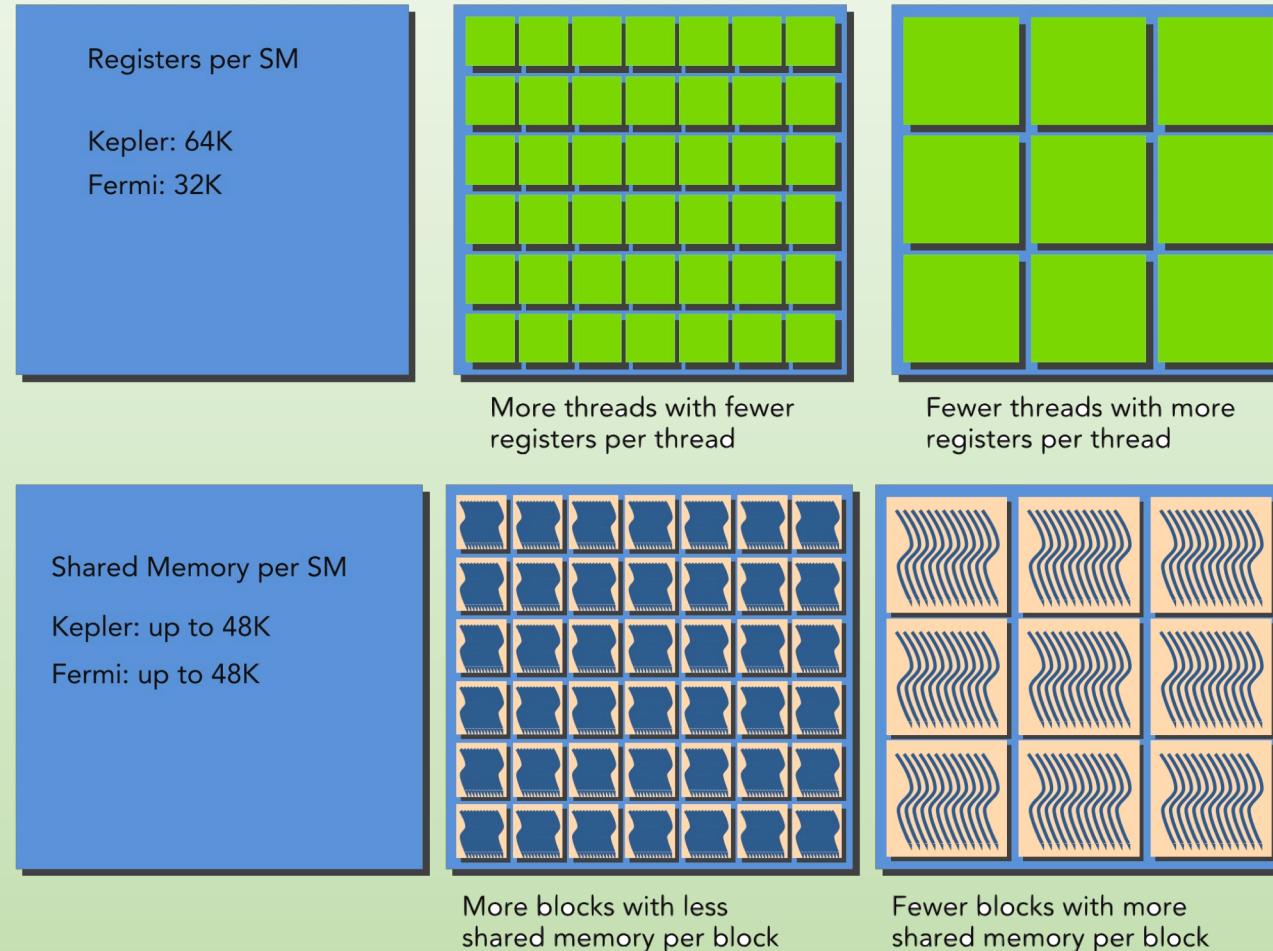
- ✓ Tutti i **thread** calcolano il **predicato logico** e le due **istruzioni** del predicato:

```
p = (x<0.0);
p: z = x-2.0;      // single instruction
!p: z = sqrt(x);
```

- ✓ In situazioni di branching ‘complesso’ il compilatore adotta tecniche di **warp voting** per verificare se tutti i warp seguono la stessa strada... l’esito della votazione dà adito a strategie diverse per ovviare all’inefficienza, per es. usando meccanismi estesi basati su predicati

Ripartizione risorse

- ✓ Ogni SM ha un insieme di **registri a 32-bit** in un **register file** che sono ripartiti tra i **thread**
- ✓ Ogni SM ha una fissata quantità di **shared memory** che è ripartita tra i **thread block**
- ✓ Il numero di **thread block** and **warp** che risiedono simultaneamente su un SM per un dato **kernel** dipende dal numeri di registri e dalla quantità di memoria shared disponibili **sull'SM** e da quelle richieste dal kernel



Limiti imposti dalle risorse

- ✓ Il numero di blocchi e warp che possono essere elaborati insieme su un SM per un dato kernel dipende:
 - dalla quantità di **registri** e di **shared memory** usata dal **kernel**
 - dalla quantità di **registri** e **shared memory** resi disponibile dallo **SM**
- ✓ Esempi di vincoli sono riportati in tabella:

TECHNICAL SPECIFICATIONS	COMPUTE CAPABILITY			
	2.0	2.1	3.0	3.5
Maximum number of threads per block	1,024			
Maximum number of concurrent blocks per multiprocessor	8		16	
Maximum number of concurrent warps per multiprocessor	48		64	
Maximum number of concurrent threads per multiprocessor	1,536		2,048	
Number of 32-bit registers per multiprocessor	32 K		64 K	
Maximum number of 32-bit registers per thread	63		255	
Maximum amount of shared memory per multiprocessor	48 K			

Misurare i warp attivi

OCCUPANCY

Il tasso tra warp attivi e numero massimo di warp per SM:

$$\text{Occupancy} = \frac{\text{warp attivi}}{\text{max numero warp}}$$

```
$ nvprof --metrics achieved_occupancy ./Application
```

Invocations	Metric Name	Metric Description	Min	Max	Avg
Device "GeForce GT 650M (0)"					
Kernel: matrix_prod(float*, float*, float*)	achieved_occupancy	Achieved Occupancy	0.879916	0.879916	0.879916

Esempio su Tesla M2070

```
$ ./sumMatrix 32 32sumMatrixOnGPU2D <<< (512,512), (32,32) >>> elapsed 60 ms
$ ./sumMatrix 32 16sumMatrixOnGPU2D <<< (512,1024), (32,16) >>> elapsed 38 ms
$ ./sumMatrix 16 32sumMatrixOnGPU2D <<< (1024,512), (16,32) >>> elapsed 51 ms
$ ./sumMatrix 16 16sumMatrixOnGPU2D <<< (1024,1024),(16,16) >>> elapsed 46 ms
```

```
$ nvprof --metrics achieved_occupancy ./sumMatrix 32 32 sumMatrixOnGPU2D <<<(512,512), (32,32)>>> Achieved Occupancy 0.501071
$ nvprof --metrics achieved_occupancy ./sumMatrix 32 16 sumMatrixOnGPU2D <<<(512,1024), (32,16)>>> Achieved Occupancy 0.736900
$ nvprof --metrics achieved_occupancy ./sumMatrix 16 32 sumMatrixOnGPU2D <<<(1024,512), (16,32)>>> Achieved Occupancy 0.766037
$ nvprof --metrics achieved_occupancy ./sumMatrix 16 16 sumMatrixOnGPU2D <<<(1024,1024),(16,16)>>> Achieved Occupancy 0.810691
```

Latency hiding

Il grado di parallelismo a livello di thread utile a massimizzare l'utilizzo delle unità funzionali di un SM dipende dal numero di warp residenti e attivi nel tempo

LATENZA:

- ✓ Definita come il **numero di cicli** necessari al completamento di un'istruzione
- ✓ Per massimizzare il **throughput** occorre che lo scheduler abbia sempre warp eleggibili ad ogni ciclo di clock
- ✓ Si ha così **latency hiding** intercambiando la computazione tra warp
- ✓ Classificazione dei tipi di istruzione che inducono latenza:
 - **Istruzioni aritmetiche**: tempo necessario per la terminazione dell'operazione (add, mult, ...)
 - **Istruzioni di memoria**: tempo necessario al dato per giungere a destinazione (load, store)
- ✓ Le latenze sono dell'ordine: **10-20** cicli per operazioni aritmetiche, **400-800** cicli per accesso a global memory!

CUDA Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click): 2.0
1.b) Select Shared Memory Size Config (bytes) 49152

2.) Enter your resource usage:
Threads Per Block 512
Registers Per Thread 48
Shared Memory Per Block (bytes) 4096
(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:
Active Threads per Multiprocessor 512
Active Warps per Multiprocessor 16
Active Thread Blocks per Multiprocessor 1
Occupancy of each Multiprocessor 33%

Physical Limits for GPU Compute Capability: 2.0
Threads per Warp 32
Max Warps per Multiprocessor 48
Max Thread Blocks per Multiprocessor 8
Max Threads per Multiprocessor 1536
Maximum Thread Block Size 1024
Registers per Multiprocessor 32768
Max Registers per Thread Block 32768
Max Registers per Thread 63
Shared Memory per Multiprocessor (bytes) 49152
Max Shared Memory per Block 49152
Register allocation unit size 64
Register allocation granularity warp
Shared Memory allocation unit size 128
Warp allocation granularity 2

Allocated Resources
Warp (Threads Per Block / Threads Per Warp) 16 Limit Per SM 48 = Allocatable Blocks Per SM 3
Registers (Warp limit per SM due to per-warp reg. count) 16 20 1
Shared Memory (Bytes) 4096 49152 12

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor Blocks/SM * Warps/Block = Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor
Limited by Registers per Multiprocessor 1 16 16
Limited by Shared Memory per Multiprocessor 12
Note: Occupancy limiter is shown in orange
Physical Max Warps/SM = 48
Occupancy = 16 / 48 = 33%

Impact of Varying Block Size

Impact of Varying Register Count Per Thread

Calculator Help GPU Data Copyright & License +

Ottimizzare la configurazione

Linee guida per calibrare grid e block size

1. Il numero di thread per blocco deve essere un multiplo di warpSize (32)
2. Evitare blockSize troppo piccoli: partire con almeno 128 o 256 thread per blocco
3. Aggiustare il blockSize in accordo alla richiesta di risorse del kernel
4. Mantenere il numero di blocchi molto più grande del numero di SM per esporre sufficiente parallelismo al device
5. Condurre esperimenti e misure per trovare la migliore configurazione di esecuzione e uso delle risorse per il problema dato

Sincronizzazione

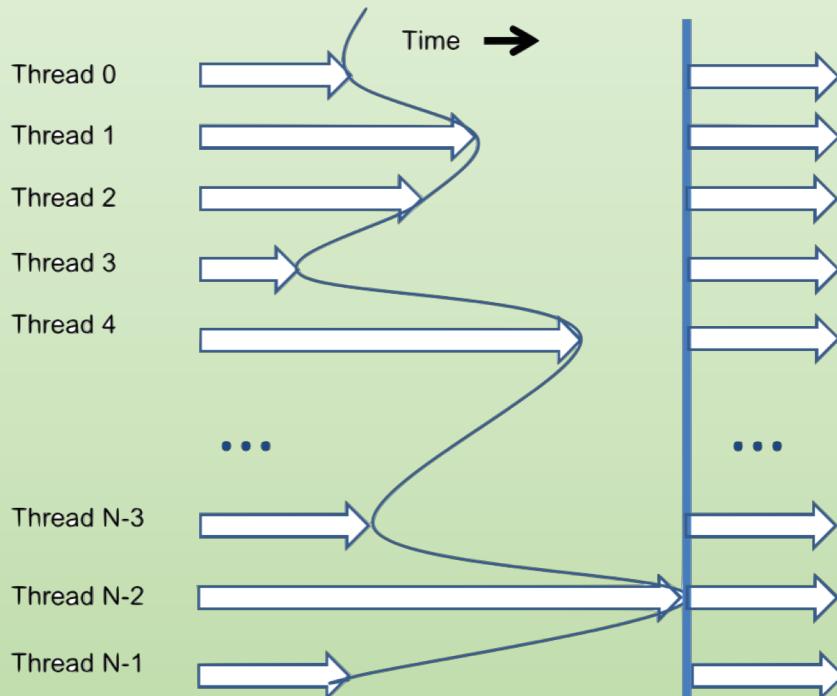
La **sincronizzazione** può essere eseguita a due livelli:

- ✓ **System-level**: attesa che venga completato un dato task su entrambi host e device
- ✓ **Block-level**: attesa che tutti i thread in un blocco raggiungano lo stesso punto di esecuzione sul device

Segnatura->

```
cudaError_t cudaDeviceSynchronize(void);
```

- blocca l'applicazione su host finché tutte le operazioni CUDA (copie, kernel, etc) siano completate



Segnatura->

```
__device__ void __syncthreads(void);
```

- Sincronizza i thread all'interno di un blocco: attende fino a che tutti raggiungono il punto di sincronizzazione

NOTE:

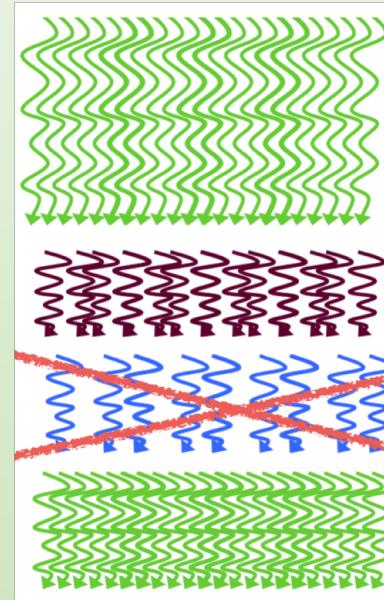
- ✓ Se presente deve essere eseguito da **tutti i thread** del blocco
- ✓ Attenzione al branch **if-then-else**: thread su branch distinti non possono sincronizzarsi (**attesa infinita!**)
- ✓ Thread in **differenti blocchi** non possono sincronizzarsi (i blocchi sono indipendenti anche sotto questo profilo!)

Divergenza e deadlock

- ✓ La divergenza può causare deadlock!

```
//some device functions F1 & F2

if (threadIdx.x < 16) {
    F1();
    __syncthreads();
}
else if (threadIdx.x >= 16) {
    F2();
    __syncthreads();
}
```



- ✓ la prima metà dei thread nei warp esegue il ramo **if**, poi attende la seconda metà che raggiunga **__syncthread()**
- ✓ la seconda metà non entra nel ramo **then**... quindi la prima metà aspetterà per sempre!

Il problema parallel reduction

Sequenziale:

```
int sum = 0;  
for (int i = 0; i < N; i++)  
    sum += array[i];
```

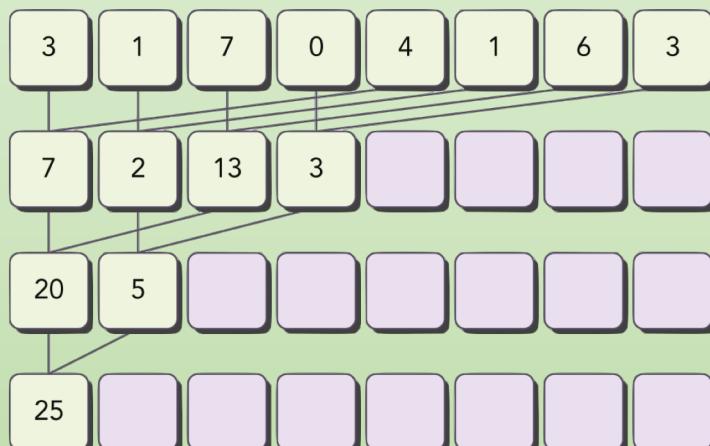
Parallelo?

1. Suddividere il vettore in parti più piccoli
2. Attivare thread per la somma parziale sui pezzi
3. Addizionare tra loro le somme parziali ottenute

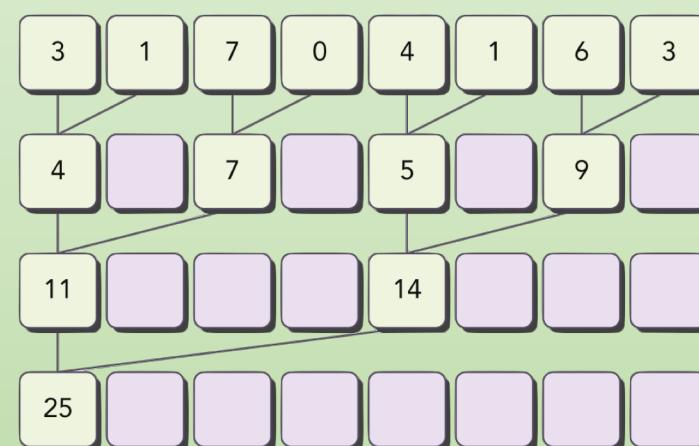
Approccio:

Somme parziali memorizzate **in-place** nel vettore stesso:

Copie di elementi equispaziati



Copie di elementi contigui



Somma (seq.) ricorsiva di un vettore

Dimezzamento del passo

Chiamata ricorsiva nel codice sequenziale

```
int recursiveReduce(int *data, int const size) {  
    // terminazione  
    if (size == 1)  
        return data[0];  
  
    // rinnova lo stride  
    int const stride = size / 2;  
  
    // riduzione in-loco  
    for (int i = 0; i < stride; i++)  
        data[i] += data[i + stride];  
  
    // chiamata ricorsiva  
    return recursiveReduce(data, stride);  
}
```

Strategia parallela:

1. Ad ogni passo un numero di thread pari alla metà degli elementi dell'array effettua le somme parziali nella prima metà di elementi (**passo di riduzione**)
2. Il numero di thread attivi deve dimezzarsi ad ogni passo (**rinnovare lo stride**)
3. Occorre sincronizzare il comportamento dei thread affinché tutti i thread al passo t abbiano terminato il compito prima di andare al passo successivo $t+1$ (analogo della **chiamata ricorsiva**)

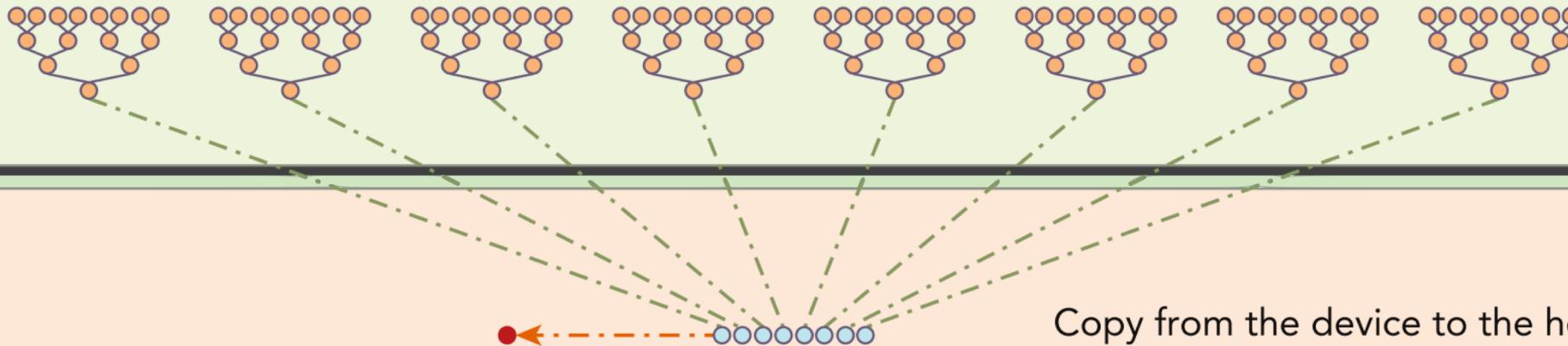
Somma parziali su blocchi + sincronizzazione

Schema dicotomico:

- ✓ **Soluzione locale:** somma parziale sincrona sui blocchi della grid ottenuta per riduzione parallela
- ✓ **Vincolo:** la dimensione di blocco deve essere necessariamente una **potenza di 2**
- ✓ **Svantaggio:** è una soluzione che introduce divergenza crescente a livello di warp (inutilizzo dei thread)

Strategia blocco x blocco + copia delle somme parziali su host

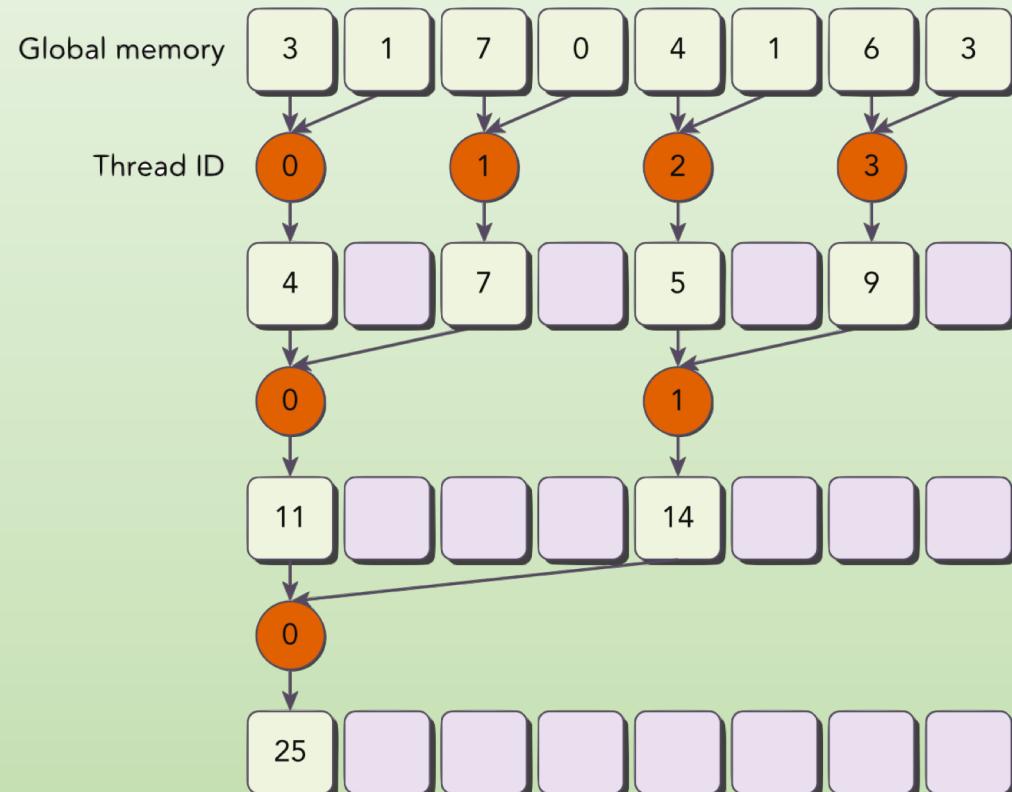
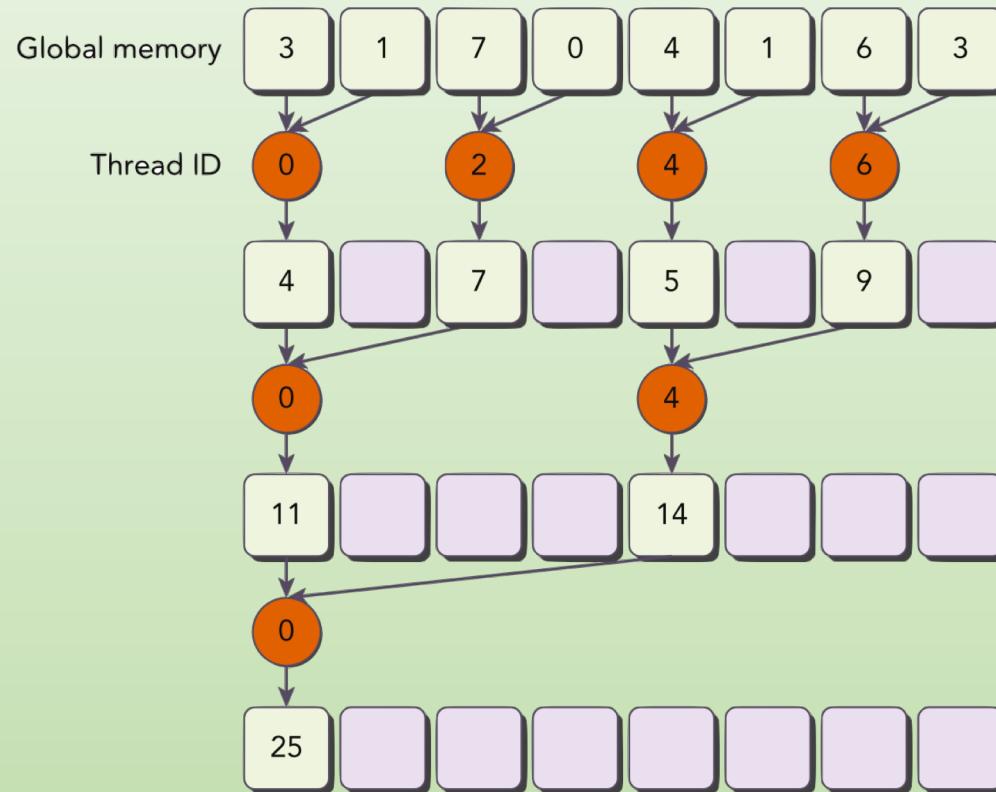
Reduce at the device side with a large amount of blocks in parallel.



Reduce at the host side in sequentially.

Schemi di parallel reduction

Due dei possibili approcci alla risoluzione del problema: quale più efficiente?



CUDA zone...

Modello di esecuzione CUDA

Somma via riduzione parallela

Ad ogni passo un numero di thread pari alla metà degli elementi ancora da sommare effettua le somme parziali nella prima metà di elementi (**riduzione in-place**)

Occorre sincronizzare il comportamento dei thread affinché tutti i thread al passo t abbiano terminato il compito prima di andare al passo successivo $t+1$

```
/* Block by block parallel implementation with divergence */
__global__ void block_parallel_reduce(int *array_in, int *array_out, unsigned int n) {
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    // boundary check
    if (idx >= n)
        return;
    // convert global data pointer to the local pointer of this block
    int *array_off = array_in + blockIdx.x * blockDim.x;
    // in-place reduction in global memory
    for (int stride = 1; stride < blockDim.x; stride *= 2) {
        if ((tid % (2 * stride)) == 0)
            array_off[tid] += array_off[tid + stride];
        // synchronize within threadblock
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0)
        array_out[blockIdx.x] = array_off[0];
}
```

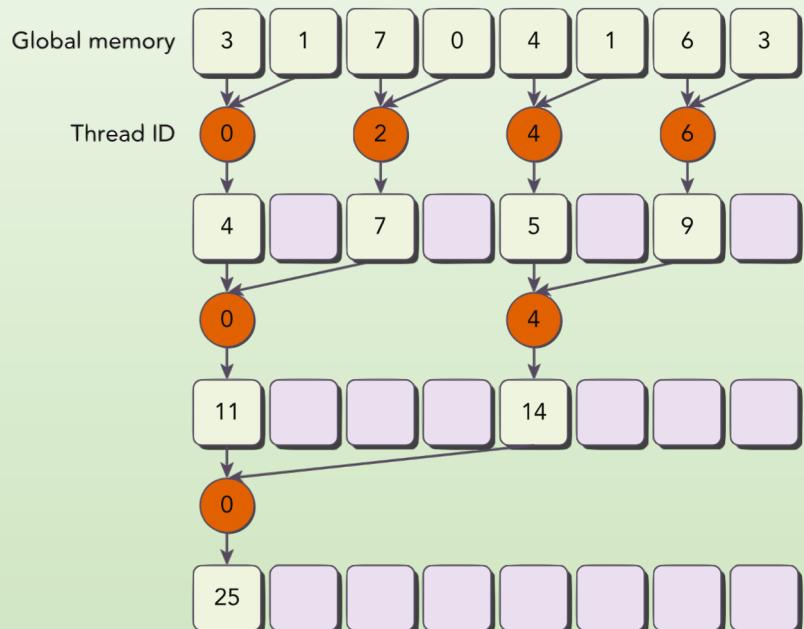
Esercitazione

Progettare un kernel per la somma di vettori di grandi dimensioni:

- Usare lo schema che suddivide in blocchi (sincronizzazione)
- Sommare su ogni blocco con parallel reduction
- Prestare attenzione alla divergenza!

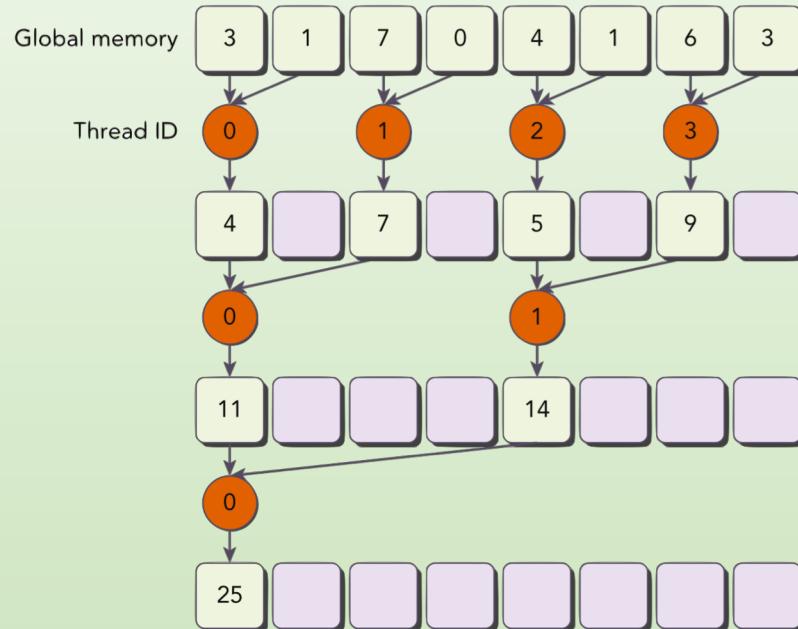
Divergenza in parallel reduction

Schema inefficiente x troppa divergenza



```
// in-place reduction in global memory
for (int stride = 1; stride < blockDim.x; stride *= 2) {
    if ((tid % (2 * stride)) == 0)
        idata[tid] += idata[tid + stride];
}
```

Schema efficiente perché riduce (quasi) a zero la divergenza



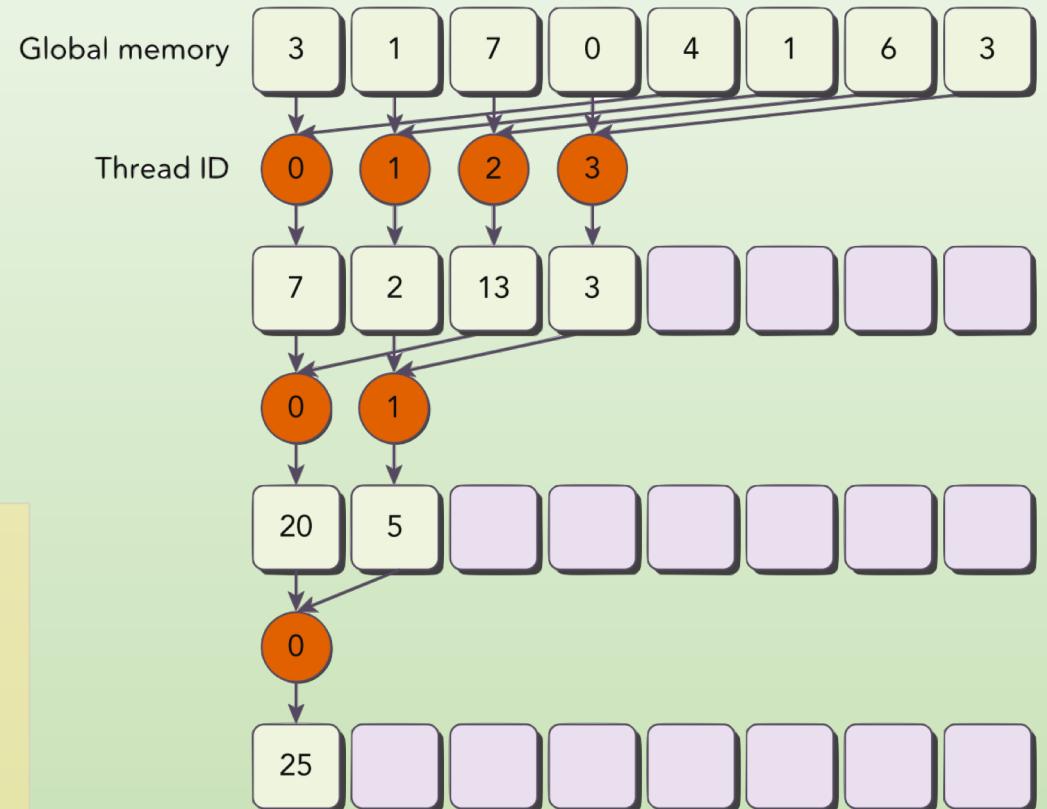
```
// in-place reduction in global memory
for (int stride = 1; stride < blockDim.x; stride *= 2) {
    // convert tid into local array index
    int index = 2 * stride * tid;
    if (index < blockDim.x)
        idata[index] += idata[index + stride];
}
```

Uso efficiente della memoria

Schema dicotomico:

- ✓ **Soluzione locale**: somma parziale sincrona sui blocchi della grid ottenuta per riduzione parallela
- ✓ **Vincolo**: la dimensione di blocco deve essere necessariamente una **potenza di 2**
- ✓ **Vantaggio**: è una soluzione più efficiente a causa di accesso contiguo a banchi di memoria globale (vedi succ. lezione su gerarchie di memorie!)

```
// in-place reduction in global memory
for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
    if (tid < stride)
        idata[tid] += idata[tid + stride];
    __syncthreads();
}
```



Misurare il tempo di esec. del kernel in CPU

Inizio tempo misurato in CPU:
non molto preciso meglio
usare gli eventi di
sincronizzazione gestiti con API
opportune

Necessità di sincronizzare
altrimenti host procede senza
attendere la fine del kernel:
chiamata asincrona host-
device

Oppuntivo catturare eventuali
errori durante le computazioni del
kernel (qui può essere omesso – ci
pensa `cudaDeviceSynchronize()`)

- ✓ Uso di `cudaDeviceSynchronize`: blocca la CPU finché il device non ha terminato tutti i task precedentemente lanciati (kernel + memcpy)
- ✓ Restituisce errore se un task lanciato in precedenza ha fallito

```
#include "common/CPU_time.h"
.

.

// block by block parallel implementation with divergence
double iStart, iElaps;
iStart = seconds();
block_parallel_reduce1<<<grid, block>>>(d_a, d_b, n);
CHECK(cudaDeviceSynchronize());
iElaps = seconds() - iStart;
printf("block_parallel_reduce1 elapsed %f sec \n", iElaps);
CHECK(cudaGetLastError());
```

Esercitazione

Progettare mediante procedura dicotomica:

- Un kernel che ricerchi il massimo valore in un array
- Un kernel che effettui il prodotto interno tra due array
- Un kernel per il calcolo della media degli elementi di un array

Operazioni atomiche

- ✓ Eseguono operazioni matematiche ma **senza interruzione** da parte di altri thread
- ✓ Che cosa accadrebbe se più thread effettuassero l'aggiornamento (cioè l'incremento) qui a lato?
- ✓ Per evitare errori potrei usare l'operazione atomica di seguito

```
__global__ void incr(int *ptr) {  
    int temp = *ptr;  
    temp = temp + 1;  
    *ptr = temp;  
}
```

Segnatura-> `int atomicAdd(int *M, int V);`

- Esegue atomicamente la somma garantendone il successo

Uso-> `__global__ void incr(int *ptr) {
 int temp = atomicAdd(ptr, 1);
}`

- ✓ Le operazioni basilari sono:
 - **Matematiche**: add, subtract, maximum, minimum, increment, and decrement
 - **Bitwise**: AND, bitwise OR, bitwise XOR
 - **Swap**: scambiano valore in memoria con un nuovo valore (vedi esempio sotto)

```
__global__ void check_threshold(int *arr, int threshold, int *flag) {  
    if (arr[blockIdx.x * blockDim.x + threadIdx.x] > threshold)  
        atomicExch(flag, 1);  
}
```

Esempi d'uso

```
__global__ void testAtomic(int *g_odata) {
    // access thread id
    const unsigned int tid = blockDim.x * blockIdx.x + threadIdx.x;

    // Arithmetic atomic instructions
    atomicAdd(&g_odata[0], 10);           // Atomic addition
    atomicSub(&g_odata[1], 10);           // Atomic subtraction
    atomicMax(&g_odata[3], tid);         // Atomic maximum
    atomicMin(&g_odata[4], tid);         // Atomic minimum
    atomicInc((unsigned int *)&g_odata[5], 17); // Atomic increment
    atomicDec((unsigned int *)&g_odata[6], 137); // Atomic decrement

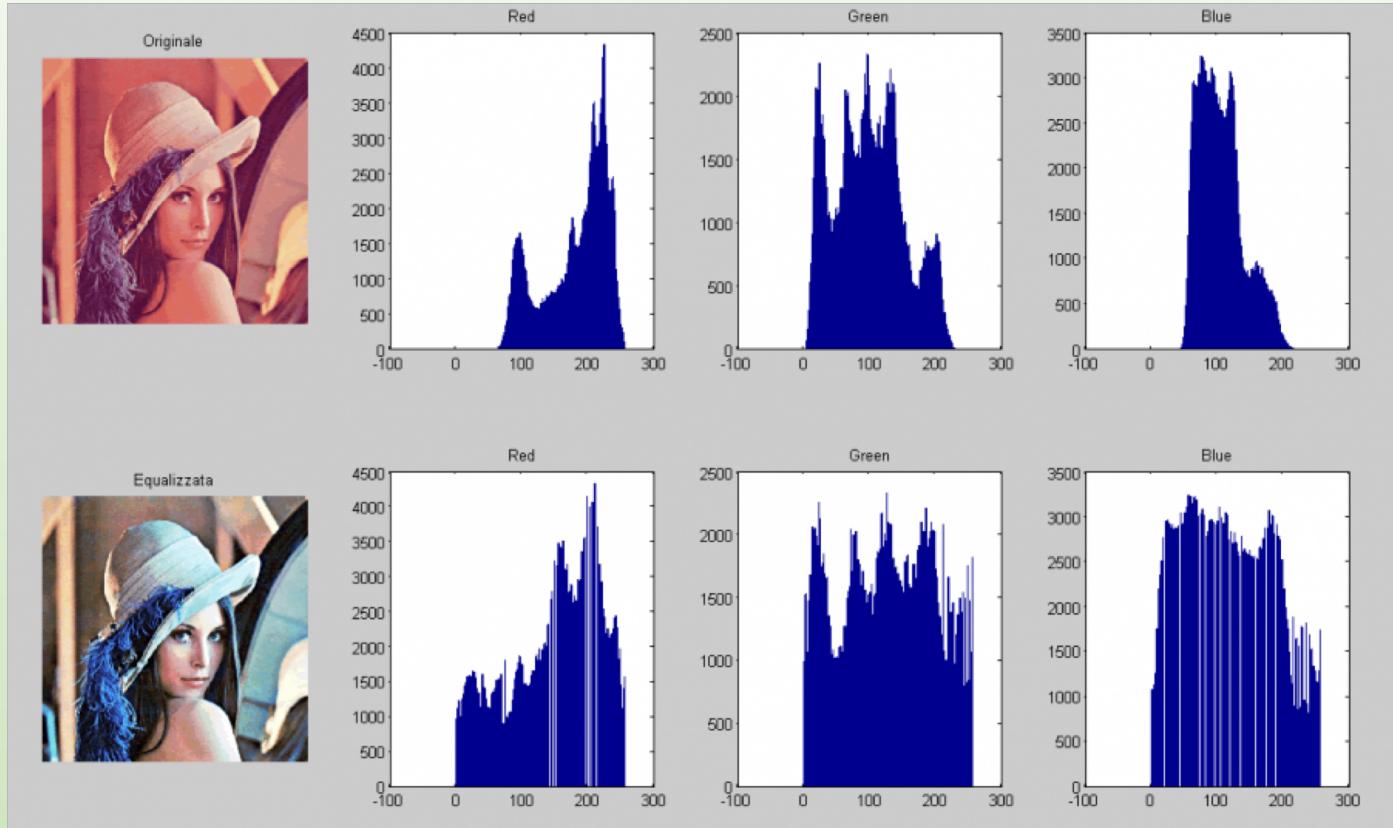
    // Arithmetic atomic instructions
    atomicExch(&g_odata[2], tid);        // Atomic exchange
    atomicCAS(&g_odata[7], tid-1, tid);   // Atomic compare-and-swap

    // Bitwise atomic instructions
    atomicAnd(&g_odata[8], 2*tid+7);      // Atomic AND
    atomicOr(&g_odata[9], 1 << tid);     // Atomic OR
    atomicXor(&g_odata[10], tid)          // Atomic XOR
}
```

Esercitazione

Istogramma di immagini

- ✓ Nell'elaborazione di immagini e in fotografia, l'istogramma del colore è una rappresentazione della distribuzione dei colori in un'immagine, è quindi una rappresentazione grafica della distribuzione tonale di un'immagine
- ✓ L'istogramma del colore può essere costruito per ogni tipo di spazio colore, sebbene il termine è più spesso usato per spazi tridimensionali come RGB o HSV
- ✓ Sugg. Usare le operazioni atomiche nello sviluppo CUDA per calcolo frequenze!



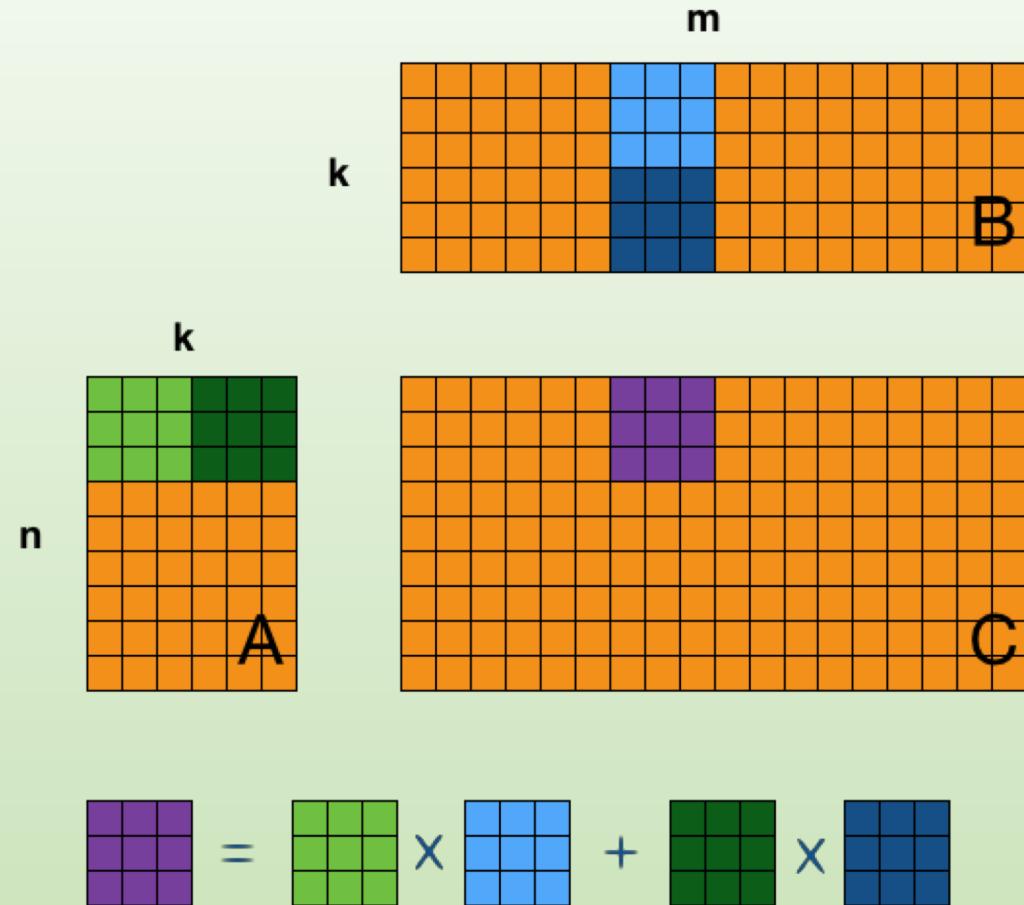
Istogramma

Progettare un kernel che realizzi l'istogramma di un'immagine a livelli di grigio o sui tre canali RGB

Prodotto di matrici

Scelte implementative:

- ✓ Come definire la **grid**? Quale **mapping** tra indici di thread e indici delle matrici A, B e C?
- ✓ Notare che le dimensioni rilevanti sono 2:
numero **n** di righe della mat A e numero **m** di colonne della mat B (insieme sono anche le dimensioni finali di C)
- ✓ Quale attività assegnare ai thread?
- ✓ **Possibili casi:** i pattern ripetuti sono i prodotti e le somme di singole coppie di elementi (caso 1), il prodotto interno tra due vettori (caso 2), riga di A e colonna di B
- ✓ **Caso 1:** numero molto elevato di thread pari al numero prodotti $n*m*k$ (granularità molto fine), poco lavoro del singolo thread e grande overhead
- ✓ **Caso 2:** numero ridotto di thread $n*m$ (granularità meno fine), e attività di maggior peso per il singolo thread

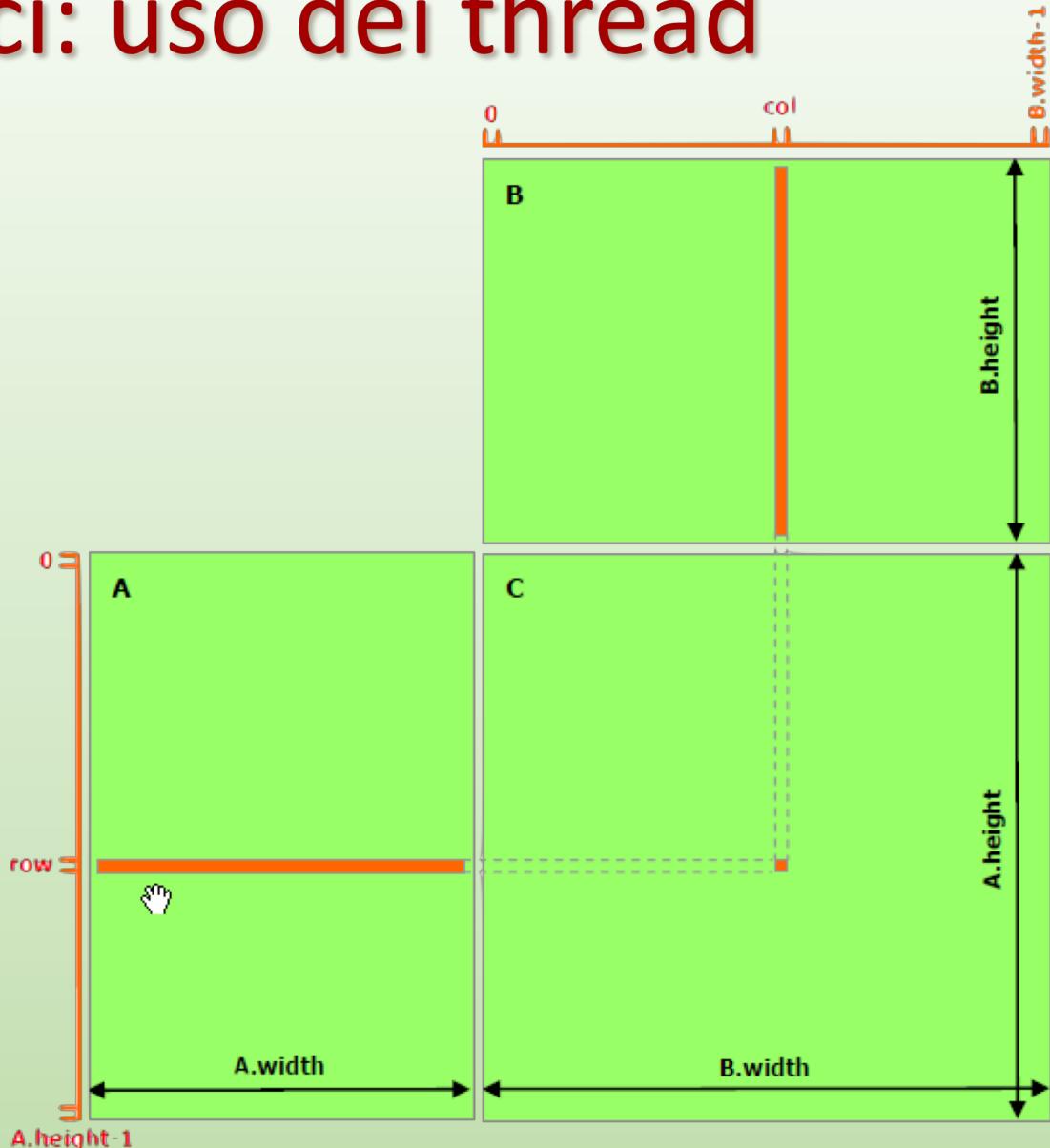


Morale...se gli SM sono tutti impegnati (ragionevole numero di warp schedulati) quindi lavorano a pieno ritmo, non dobbiamo preoccuparci di effetture task sequenziali all'interno del singolo thread... Anzi!

Moltiplicazione matrici: uso dei thread

Un possibile approccio (caso 2):

- Trasferimento delle matrici **A**, **B** e
allocazione di **C** nella memoria del device
- La grid deve coprire il numero di righe
delle matrici **A** nella dimensione **y** e il
numero di colonne della matrice **B** nella
dimensione **x**
- Un thread si preoccupa di effettuare il
prodotto riga-colonna della matrice
- Per ogni riga **i** e colonna **j**, il thread con
indice **i** nella dimensione **x** e il thread con
indice **j** nella dimensione **y** mediante un
ciclo itera la somma dei prodotti che
coinvolgono la riga e la colonna
- Salva la somma nella entry **i, j** della
matrice risultato **C** e la trasferisce in
memoria RAM della CPU



Soluzione

Passaggio argomenti: per valore o puntatori ottenuti da cudaMalloc

Mapping degli indici di thread su indici di riga e colonna della matrice risultato

Ciclo su elementi di riga (Row*M+k) della mat A e colonna (k*M+Col) della mat B

```
/*
 * Kernel per il prodotto di due matrici
 */

__global__ void matrix_prod(float* A, float* B, float* C, int N, int M) {
    // indici di riga e colonna
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    int Col = blockIdx.x * blockDim.x + threadIdx.x;
    // ogni thread calcola una entry della matrice prodotto
    if ((Row < N) && (Col < M)) {
        float val = 0;
        for (int k = 0; k < M; k++)
            val += A[Row * M + k] * B[k * M + Col];
        C[Row * M + Col] = val;
    }
}
```

Scrittura dei risultati in global memory del device

Lettura dei dati in global memory del device

Ottimizzazione profile-driven

Profiling significa analizzare le prestazioni del programma misurando:

- ✓ La **complessità** in spazio (memory) o in tempo dell'algoritmo
- ✓ L'uso di particolari **istruzioni**
- ✓ La **frequenza** e la durata delle chiamate a funzione

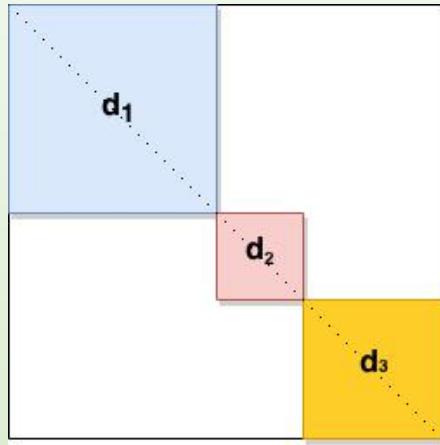
Lo sviluppo di applicazioni HPC spesso implicano 2 passi principali:

- ✓ Sviluppare codice **corretto**
- ✓ Migliorare il codice per elevare le **performance**

In CUDA:

- ✓ Una implementazione **naïve** del **kernel** in genere **non dà alte prestazioni**
- ✓ I tool di **profiling** aiutano a trovare **bottleneck** nel codice da rimuovere
- ✓ CUDA ripartisce le **risorse di calcolo** in un SM tra i blocchi di thread residenti e quando si eccedono i limiti si riduce l'utilizzo efficiente della GPU
- ✓ CUDA fornisce un'astrazione dell'architettura hardware che consente il controllo diretto di thread e concorrenza: il profiling può aiutare a misurare, visualizzare e guidare l'ottimizzazione

Challenge: prod. di mat. diag. a blocchi su GPU



- ✓ **Specifiche:** una matrice \mathbf{A} in $R^{n \times n}$ di tipo MQDB è composta da k blocchi B_1, \dots, B_k ($0 \leq k \leq n$) disposti sulla diagonale principale. Un esempio con $k=3$ è dato in figura. Se $D=\{d_1, \dots, d_k\}$ denota l'insieme delle dimensioni dei blocchi, allora deve valere: $\sum d_i = n$, pertanto una matrice siffatta è completamente descritta dalla coppia $\langle n, D \rangle$. Vale inoltre che il prodotto di matrici con parametri $\langle n, D \rangle$, è ancora una matrice dello stesso tipo
- ✓ **Input:** due matrici MQDB \mathbf{A} e \mathbf{B} di tipo float descritte da un intero n e un array $d[k]$
- ✓ **Output:** la matrice prodotto $\mathbf{C} = \mathbf{A} * \mathbf{B}$ di tipo float

- ✓ **Benchmark:** è disponibile un generatore di istanze di MQDB casuali ottenute fissando la dimensione n della matrice e il numero di blocchi-dati k
- ✓ **Test:** per il test finale utilizzare 3 istanze casuali generate a caso con params:
 - Seme $s = 0, 1, 2$
 - Dimensione matrice $n = 1000, 5000, 10000$
 - Numero blocchi-dati $k = 3, 5, 10$

Progetto:

- kernel CUDA ottimizzato per il prodotto di matrici quadrate diagonali a blocchi
- Effettuare l'analisi delle prestazioni al variare della taglia dei dati in ingresso e dei params di del kernel
- Usare ***nvprof*** per i tempi reali

...osservazioni

- ✓ **Cuda malloc & copy:** gestire con attenzione i puntatori all'interno della *struct mqdb*
- ✓ **Progettare** un kernel adeguato al caso di matrici così strutturate
- ✓ **Verificare** configurazioni diverse delle grid
- ✓ **Confronto** con la funzione sequenziale su host
- ✓ **Usare** la profilazione come strumento di controllo del funzionamento
- ✓ **Controllare** chunk di memoria allocata

```
$ deviceQuery
Device 0: "Tesla M2090"
    CUDA Driver Version / Runtime Version 8.0 / 8.0
    CUDA Capability Major/Minor version number: 2.0
    Total amount of global memory: 5375 MBytes (5636554752 bytes)
    ...
```