GPU Computing

Laurea **Magistrale in Informatica** - AA 2024/25

Docente **G. Grossi**

Lezione 3 – Il modello di esecuzione CUDA

Sommario

- ✓ Panoramica su thread e block
- ✓ Modello SIMT: esecuzione di thread, warp e blocchi
- ✓ Divergenza nei warp
- ✓ Sincronizzazione a livello di warp e blocco
- ✓ Parallel pattern: parallel reduction
- ✓ Operazioni atomiche: istogramma

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

Mapping logico – fisico dei thread

Esecuzione

- ✓ Mapping **gerarchia** di **thread** -> **gerarchia** di **processori** sulla GPU (**array SMs**)
- ✓ 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**

SIMT architecture

Modello di esecuzione

Warp: architettura SIMT

- ✓ **Architettura SIMT**: Single Instruction Multiple Thread
- ✓ **Warp = 32 thread** (con ID consecutivi!)
- ✓ **Warp =** scheduling units in SM
- ✓ 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**, e può seguire **cammini distinti** di esecuzione delle istruzioni (parallelismo a livello thread)
- ✓ I thread che compongono un **warp** iniziano assieme alla stesso indirizzo del programma

Metafora

32… un numero magico!

- ✓ Il numero 32 ha origine dall'**hardware** ed è fondamentale nella programmazione CUDA per la **scalabilità** e **trasparenza**
- ✓ 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** (più flessibile, ma potenzialmente più inefficiente nella gestione dei thread!)

Warp Organization

- ✓ **Warp (32 threads) → Half-Warp (16 threads) → Quarter-Warp (8 threads)**
- ✓ **Thread Assignment**: Threads in a warp are grouped by consecutive, **increasing thread IDs**
- ✓ **SIMT vs SIMD**

Feature	SIMT (Single Instruction, Multiple Threads)	SIMD (Single Instruction, Multiple Data)
Thread control	Each thread has its own execution state	Operates on vectors exposed to software
Execution model	**Warp-based scheduling**	**Explicit vectorization**
Branching	**Warp-level divergence handling**	**Manual management of divergence**
Programming model	**Thread-level parallelism**	**Vector-based parallelism**

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
```

Warps in multi-dimensional thread blocks

- ✓ The **thread blocks** are first **linearized into 1D** in **row major order**
- ✓ In **x**-dimension **first**, **y**-dimension **next**, and **z**-dimension **last**

$$T_{row,\,col} = T_{y,\,x}$$

Logical 2D organization

Placing 2D threads into linear order

Scheduling dei blocchi

- ✓ Se si hanno **1000 block** con ognuno **128 thread** – come vengono eseguiti?
- ✓ Probabilmente **8-12 block** sono allocati allo **stesso SM**
- ✓ ogni blocco ha **4 warp -> 48 warp** in esecuzione su **ogni SM**
- ✓ ad ogni colpo di clock lo **scheduler dei warp** decide quale warp è il **prossimo** ad andare in **esecuzione** sceglie tra quelli che:
  - **non sono in attesa** di dati dalla **device memory** (mem latency)
  - **non stanno completando** un'istruzione precedente (pipeline delay)
- ✓ **Dettagli trasparenti** al programmatore…. serve solo garantire un elevato num di warp in esecuzione!

Warp Scheduling

- ✓ Warps are **scheduled** onto **execution** units (SMs)
- ✓ Threads execute the **same instruction** per **cycle**
- ✓ If branch **divergence** occurs:
  - **Different** paths execute **sequentially**
  - **Inactive** threads are **disabled**
- ✓ **Active:** Threads executing the current instruction in a warp are **active**,
- ✓ **Inactive Threads:** due to **early exit**, **branch divergence**, or being part of a **smaller thread block**

The GPU Architecture

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

- ✓ 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 costo zero!)

Registri

- ✓ 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)
- ✓ I dati nei registri sono **privati** ai thread (scope) e ogni multiprocessor ha un insieme di 32-bit register che sono **partizionati** tra i **warp**

```
__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;
  }
}
```

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 numero 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 per le architetture Fermi e Kepler 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			

Limiti imposti dalle risorse

✓ Architetture recenti

GPU features	Nvidia Tesla P100	Nvidia Tesla V100	Nvidia A100
GPU codename	GP100	GV100	GA100
GPU architecture	Nvidia Pascal	Nvidia Volta	Nvidia Ampere
Compute capability	6.0	7.0	8.0
Threads / warp	32	32	32
Max warps / SM	64	64	64
Max threads / SM	2048	2048	2048
Max thread blocks / SM	32	32	32
Max 32-bit registers / SM	65536	65536	65536
Max registers / block	65536	65536	65536
Max registers / thread	255	255	255
Max thread block size	1024	1024	1024
FP32 cores / SM	64	64	64
Ratio of SM registers to FP32 cores	1024	1024	1024
Shared Memory Size / SM	64 KB	Configurable up to 96 KB	Configurable up to 164 KB

Latency hiding

Il grado di parallelismo a livello di thread è utile a massimizzare l'uso 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 prog. model and HW structure

Independent thread scheduling

Barriera di sincronizzazione e divergenza (inefficienza) nei branch

Evolution of Warp Scheduling

- ✓ **Pre-Volta** (Before NVIDIA Volta, cc 7.0)
  - **Shared program counter** per warp
  - **Active mask** tracks executing threads
  - **Limited flexibility**, potential for deadlocks in synchronization
- ✓ **Post-Volta** (NVIDIA Volta & Beyond)
  - **Independent Thread Scheduling** introduced
  - Each thread has its **own program counter and call stack**
  - Allows **fine-grained thread scheduling** and **sub-warp divergence management**
- ✓ A key design goal was to **simplify GPU programming** and enhance **efficiency for fine-grained parallel algorithms**

Volta SIMT scheduling

- ✓ Volta (CC 7.0) **independent thread scheduling** architecture block diagram compared to Pascal (< CC 7.0)
- ✓ Per-thread scheduling **program counter** (PC) and **call stack** (S)

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;
}
```

Esempio: divergenza nei warp

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

```
/*
* Kernel with warp divergence
*/
__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; // thread pari
   else
      b = 1; // thread dispari
   c[tid] = a + b;
}
```

Caso peggiore

- ✓ Le prestazioni **decrescono** con l'aumentare del **tasso** di **divergenza** nei warp!
- ✓ Il parallelismo (la sua efficienza) in questo caso è peggiorato di **32 volte**!

#paths = 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;
   }
}
```

Barriere di sincronizzazione

- ✓ Se presente una primitiva di sincronizzazione deve essere eseguito da **tutti** i **thread** del **blocco**
- ✓ Thread in **differenti blocchi** non possono sincronizzarsi (tranne nel caso di cooperative groups!)
- ✓ Senza meccanismi di sincronizzazione si possono verificare **race condition** in cui diversi thread accedono simultaneamente alla stessa locazione di memoria dando luogo a comportamenti impredicibili
- ✓ Attenzione al branch **if-then-else**: thread su branch distinti non possono sincronizzarsi (**deadlock**! pre-volta)

Sincronizzazione a più livelli

- ✓ **System-level**: attesa che venga completato un dato task su entrambi host e device

Segnatura

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

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

- ✓ **Block-level**: attesa che tutti i thread in un blocco raggiungano lo stesso punto di esecuzione

Segnatura

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

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

- ✓ **Warp-level**: attesa che tutti i thread in un warp raggiungano lo stesso punto di esecuzione

Segnatura

```
__device__ void __syncwarp(mask);
```

- ➢ Sincronizza i thread all'interno di un warp: attende fino a che tutti raggiungono il punto di sincronizzazione (riconverge)

Warp divergence (pre volta)

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

- ✓ La **CPU** include complessi meccanismi hardware per effettuare **branch prediction**
- ✓ Se la predizione non è corretta la **CPU** può rimanere bloccata per un **elevato numeri di cicli di clock** per resettare la pipeline

- ✓ La **GPU** sono dispositivi più semplici **senza** HW per strategie di **branch prediction**
- ✓ In linea di principio **tutti** i **thread** in un warp devono eseguire la **medesima istruzione**
- ✓ **Warp divergence:** un gruppo di thread (nel warp) prende un differente path

Deadlock

- ✓ Quite easy to get a **deadlock between threads** within a warp
- ✓ This program **will hang** since either **A or B will never be executed**, but each branch is waiting on an event that occurs in the other

```
if(threadIdx.x & 0x4)
{
    A();
    waitOnB();
}
else
{
    B();
    waitOnA();
}
C();
```

Divergenza e deadlock

- ✓ La divergenza può causare deadlock!

```
//some device functions F1 & F2

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

deadlock

- ✓ la prima metà dei thread nei warp esegue il ramo `if`, poi attende la seconda metà che raggiunga

`__syncthreads`

- ✓ la seconda metà non entra nel ramo `then`… quindi la prima metà aspetterà per sempre!

Pre-volta execution model

- ✓ Pascal warp uses a **single program counter** shared amongst all **32 threads**, combined with an **active mask** that specifies which threads of the **warp are active** at any given time
- ✓ **divergent execution paths** leave some threads inactive → **serializing execution** for different portions of the warp

- ✓ The original **mask is stored** until the **warp reconverges** at the end of the **divergent section**

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 più 'complesso' il compilatore adotta tecniche di **warp voting** per verificare se tutti i warp seguono la stessa strada

VOLTA (SM70) WARP EXECUTION MODEL

`__syncwarp` is used to explicitly force synchronization, reconvergence

```
if (threadIdx.x < 4) {
    A;
    __syncwarp();
    B;
} else {
    X;
    __syncwarp();
    Y;
}
__syncwarp();
```

Syncwarp

```
__device__ void __syncwarp(mask);
```

- ✓ Will cause the executing thread to **wait** until all **warp lanes** named in **mask** have executed a **`__syncwarp()`** (with the same mask) before **resuming execution**
- ✓ All **non-exited** threads named in mask must execute a corresponding **`__syncwarp()`** with the **same mask**
- ✓ It is similar to the **`__syncthreads()`** primitive but at **finer granularity**

Summary of SIMT Architecture

- ✓ The **Single Instruction, Multiple Threads (SIMT)** architecture enables efficient parallel processing by managing execution in **warps**, which are groups of **32 threads** that execute together. Each thread has its own **instruction address counter** and **register state**, allowing independent execution despite starting at the same address.
- ✓ When a **multiprocessor** receives thread blocks, it partitions them into warps and assigns them to a **warp scheduler**. Within a warp, all threads execute the same instruction per cycle; however, if a conditional branch causes divergence, the warp serially executes each path, disabling inactive threads. This affects performance but only occurs within a warp—warps themselves execute independently
- ✓ **SIMT** is similar to **SIMD (Single Instruction, Multiple Data)** but abstracts hardware-level vectorization, allowing programmers to write **scalar and data-parallel** code without explicit vector management. Performance optimization, however, requires minimizing warp divergence, much like optimizing for cache efficiency in traditional CPU programming.
- ✓ Before **NVIDIA Volta**, warps shared a single **program counter**, limiting fine-grained thread synchronization and risking deadlocks in multi-threaded algorithms. **Volta and later architectures** introduced **Independent Thread Scheduling**, where execution state is maintained per thread. This allows **greater concurrency**, **sub-warp granularity divergence**, and improved efficiency, but also requires revisiting older **warp-synchronous** code for compatibility.

Warp shuffle

- ✓ Allows for **intra-warp communication**
- ✓ Various supported **movement patterns**:
  - `__shfl_sync()` : copy from lane ID (arbitrary pattern)
  - `__shfl_xor_sync()` : copy from calculated lane ID (calculated pattern)
  - `__shfl_up_sync()` : copy from delta/offset lower lane
  - `__shfl_down_sync()` : copy from delta/offset higher lane

- ✓ Both **source** and **destination** threads in the warp must "**participate**"
- ✓ Sync "**mask**" used to identify and **reconverge** needed threads

Shuffle up

```
T __shfl_up_sync(unsigned mask, T var, unsigned int delta);
```

- ✓ `mask` controls which threads are involved — usually set to `-1`or `0xffffffff`, equivalent to all 1's
- ✓ `Var` is a local register variable (**`int, unsigned int, long long, unsigned long long, float or double`**)
- ✓ `Delta` is the offset within the warp – if the appropriate thread does not exist (i.e. it's off the end of the warp) then the value is taken from the current thread

```
T __shfl_down_sync(unsigned mask, T var, unsigned int delta);
```

- ✓ Defined similarly

To sum all the elements in a warp

```
__global__ void warpReduce(int arg) {
    int laneId = threadIdx.x;
    int value = 1; // value of 1 for all threads

    // Use DOWN mode to perform reduction
    for (int delta= warpSize/2; delta>0; delta/=2)
        value += __shfl_down_sync(0xffffffff, value, delta);

    // "value" now contains the sum across all threads
    printf("Thread %d final value = %d\n", laneId, value);
}
```

```
Thread 0 final value = 32
Thread 1 final value = 32
Thread 2 final value = 32
. . .
Thread 30 final value = 32
Thread 31 final value = 32
```

→ lab3

Warp divergency

Parallel reduction

Somma di elementi di un vettore in parallelo: sincronizzazione + divergenza

Reduction

- ✓ Un'operazione molto comune che va sotto il nome di **reduction** è la **somma** degli **elementi** di **array** grandi dimensioni:
  - calcolo della **media** in generale (es. Monte Carlo simulation, RMS, …)
  - calcolo del **prodotto interno** tra vettori in algebra lineare
  - operazioni analoghe come il calcolo del **minimo** e del **massimo**

$$(x_1, x_2, \dots, x_n) \quad \mapsto \quad s = \sum_{i=1}^{n} x_i$$

- ✓ Le proprietà richieste per un **operatore** $\oplus$ di **reduction** sono:
  - **commutativa** $a \oplus b = b \oplus a$
  - **associativa** $a \oplus (b \oplus c) = (a \oplus b) \oplus c$
- ✓ Queste due proprietà: gli elementi possono essere **riordinati** e **combinati** in **qualsiasi modo**

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. **Sommare** tra loro i risultati **parziali** ottenuti

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

Coppie di elementi equispaziati (stride)

Coppie di elementi contigui (sequenz.)

log n + 1
step (=4)

Somma (strided) ricorsiva di un vettore

Dimezzamento del passo

Somme parziali: dimezzamento dell'array

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 <-> somma ricorsiva:

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)

Schema di indici con stride

Approccio sequenziale

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];
}
```

Il problema della divergenza

Alto livello di divergenza: warp sono molto inefficienti

```
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();
}
```

- ✓ Nella **I iterazione** solo un **quarto** dei thread è attivo…. `tid= 0,2,4,…,30,…`
- ✓ Nella **II iterazione** solo un **quarto** dei thread è attivo… `tid= 0,4,8,…,32,…`
- ✓ Nella **III iterazione** solo un **ottavo** dei thread è attivo…. `tid= 0,8,16,24,32,…`

Approccio sequenziale

Schema sequenziale senza divergenza!

divergenza eliminata!

```
// 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];
   // synchronize within threadblock
   __syncthreads();
}
```

Note:

- ✓ Eliminazione della **`if`** clausola in tutti i thread!
- ✓ La locazione delle somme parziali non è cambiata, ma l'indice dei thread è diverso

Divergenza in parallel reduction

```
// 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];
}
```

```
// 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];
}
```

Schema interlacciato

- ✓ Lo schema interlacciato considera lo striding degli elementi dell'array
- ✓ Lo stride parte a metà di block size e si riduce di metà ad ogni iterazione
- ✓ Migliora gli accessi a memoria (load/store pattern)… vedi accessi a memoria ottimizzati!

Operazioni atomiche

Calcolo istogramma immagini con operazioni atomiche

Atomic vs non-atomic operations

- ✓ **Non-Atomic Writes:** When multiple threads in a warp write to the same memory location **without atomicity**, the number of serialized writes depends on the **device's compute capability**, and the **final write is undefined**

- ✓ **Atomic Operations:** If multiple threads perform an **atomic read-modify-write** on the same memory location, all operations **occur and are serialized**, but the **execution order is unpredictable**

Key Concepts of Atomic Operations

- ✓ A **read-modify-write** operation performed by a **single hardware instruction** on a memory **location** ***address***
  - Read the old value, calculate a new value, and write the new value to the location
- ✓ The **hardware** ensures that **no other threads** can perform another read-modify-write operation on the same location until the **current atomic** operation is **complete**
  - Any **other threads** that attempt to perform an atomic operation on the same location will typically be **held** in a **queue**
  - All **threads** perform their atomic operations **serially** on the same location

Data Race in Parallel Thread Execution

```
thread1:  old      ← count[x]        thread2:  old      ← count[x]
          new      ← old + 1                   new      ← old + 1
          mem[x]   ← new                       mem[x]   ← new
```

- ✓ Both `old` and `new` are per-thread register variables
- ✓ Question 1: If `count[x]` was initially `0`, what would the value of `mem[x]` be after threads 1 and 2 have completed?
- ✓ Question 2: What does each thread get in their `old` variable?
- ✓ Unfortunately, the answers may vary according to the relative execution timing between the two threads, which is referred to as a **data race**

Operazioni atomiche

- ✓ Che cosa accadrebbe se più thread effettuassero **l'aggiornamento** (cioè l'incremento) con **kernel `incr`**?
- ✓ Siamo di fronte a una **race condition**!

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

- ✓ Le **operazioni atomiche in CUDA** eseguono (solo) operazioni matematiche ma **senza interruzione** (**locked**) da parte di altri thread

```
__global__ void incr(int *ptr) {
   atomicAdd(ptr, 1);
}
```

Atomic AND

- ✓ Atomic AND: performs the following in one atomic operation

```
__global__ void AtomicAnd(int *address, int val) {
   int old = *address;
   *address = val & old;
   return old;
}
```

Atomic Arithmetic Operations in CUDA

- ✓ Performed by calling **functions** that are translated into **single instructions**
  - ✓ Atomic **add**, **sub**, **inc**, **dec**, **min**, **max**, **exch** (exchange), **CAS** (compare and swap)
- ✓ **Atomic Add**
  - ✓ `int atomicAdd(int* address, int val);`
  - ✓ reads the 32-bit word **old** from the location pointed to by **address** in global or shared memory, computes (**old + val**), and stores the result back to memory at the **same address**. These three operations are performed in one atomic transaction. The function **returns old**.

indivisible
READ-MODIFY-WRITE

```
LD R2, a[i]          (thread independent)
LD R1, c             (READ)
ADD R3, R1, R2       (MODIFY)
ST R3, c             (WRITE)
```

Becomes one indivisible operation/instruction:

RED.E.ADD.F32.FTZ.RN [c], R2;

Operazioni atomiche

✓ 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… esempio:

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

More Atomic Adds in CUDA

✓ Unsigned 32-bit integer atomic add

- `unsigned int atomicAdd(unsigned int* address, unsigned int val);`

✓ Unsigned 64-bit integer atomic add

- `unsigned long long int atomicAdd(unsigned long long int* address, unsigned long long int val);`

✓ Single-precision floating-point atomic add (Compute capability 2.x+)

- `float atomicAdd(float* address, float val);`

✓ Double-precision floating-point atomic add (Compute capability 6.x+)

- `double atomicAdd(double* address, double val);`

✓ 16-bit floating-point atomic add (Compute capability 7.x+)

- `__half atomicAdd(__half* address, __half val);`

Esempi d'uso

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

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

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

Es. istogramma di testo

- ✓ Define the **bins** as **four-letter sections** of the alphabet: a-d, e-h, i-l, n-p, …
- ✓ For each **character** in an input string, **increment** the appropriate **bin counter**

Istogramma dei colori nelle immagini

- ✓ Nell'elaborazione di immagini e in fotografia **l'istogramma del colore** è una rappresentazione della **distribuzione** (frequenza assoluta/relativa) **tonale** dei **colori** in un'immagine
- ✓ **L'istogramma** del colore può essere costruito per ogni tipo di **spazio** colore, come **RGB** o **HSV**

→ lab3

Calcolo istogramma in parallelo

Moltiplicazione MQDB

Ottimizzazione prodotto matrici

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 hoverhead
- ✓ **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

```
/*
* Kernel per il prodotto di due matrici A e B (dims MxK, KxN): C = A * B
*/
__global__ void matrix_prod(float* A, float* B, float* C, int K, nt 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 < M) && (Col < N)) {
      float val = 0;
      for (int k = 0; k < K; k++)
         val += A[Row * M + k] * B[k * N + Col];
      C[Row * M + Col] = val;
   }
}
```

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*N+Col) della mat B

Scrittura dei risultati in global memory del device

Lettura dei dati in global memory del device

Prodotto di MQDB su GPU

- ✓ Un esempio con **k=3** è dato in figura
- ✓ Se **D={d_1,…,d_k}** insieme delle dimensioni dei blocchi t.c. $\sum d_i = n$
- ✓ Vale inoltre che il prodotto di matrici con parametri **⟨n,D⟩**, è ancora
- ✓ **Input**: due matrici MQDB ***A*** e ***B*** di tipo float descritte da un intero ***n*** e un array ***d[k]***
- ✓ **Output**: la matrice prodotto $C = A * B$ 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***

Riferimenti bibliografici

Testi generali:

1. J. Cheng, M. Grossman, T. Mckercher, Professional Cuda C Programming, Wrox Pr Inc. (1^ ed), 2014 (cap 3)

   (free available https://www.cs.utexas.edu/~rossbach/cs380p/papers/cuda-programming.pdf)

2. D. B. Kirk, W. W. Hwu, Programming Massively Parallel Processors, Morgan Kaufmann (3^ ed), 2013 (cap 3-4)

NVIDIA docs:

1. CUDA Programming: *http://docs.nvidia.com/cuda/cuda-c-programming-guide/*
2. CUDA C Best Practices Guide: *http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/*