



Programmazione GPU

Una panoramica sul calcolo GPGPU





- **Task :**

Trovare le imperfezioni su di una tessitura di dimensione variabile

- **Strumenti Utilizzati :**

Descrittore Local Binary Pattern

- **Metodologie Applicate :**

Implementazione LBP,

Parallelizzazione dell'algoritmo tramite GPU



- **GPGPU =**

General Purpose computing on
Graphic Processing Units.

- *Ovvero* : trasferire determinati calcoli da CPU a dispositivo GPU
- Tecnica sempre più usata per calcolo scientifico in molti settori:



Economia



Astronomia



IA



Bioinformatica

[Analisi immagini
e video](#)



- Ingenti investimenti nel corso degli ultimi anni.
- **Strumento in evoluzione**, architetture nuove ogni anno.
- Sono definite come :
'macchine per calcolo in virgola mobile'

Funzioni :

- Possono accelerare determinati processi tramite parallelizzazione
es.: disegnare a schermo un'immagine.
- Specific Purpose, ma con delle particolarità
- Hardware disponibile sul mercato è potente ed a basso costo

Breve storia GPU



- **1980s** : accelerano funzioni 2D di disegno, sono usate per applicare trasformazioni su immagini e video
([Industrial Light & Magic](#) : Motion control photography)



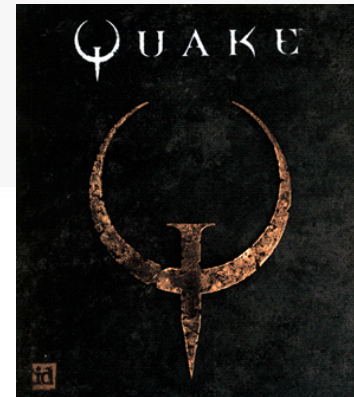
Breve storia GPU



- **1990s** : funzioni grafiche integrate su chip di Intel, prime schede video 3D per personal computer ([3Dfx](#) : Voodoo)

1996 : Primo esperimento di *Id Software* in collaborazione con **3Dfx**:

prima applicazione capace di sfruttare l'accelerazione hardware della GPU tramite OpenGL

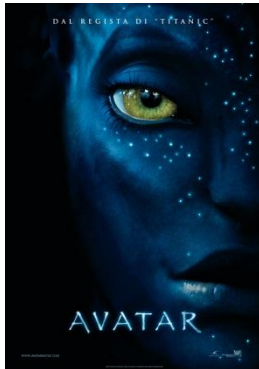


- Il mercato videoludico diventa una delle più importanti industrie americane.

Situazione attuale GPU



- **2000-2005: L'industria videoludica fattura più del cinema.**
Si pensa al raggiungimento del fotorealismo, alle GPU viene richiesto di aumentare il numero di poligoni a schermo.
Le GPU sono strumenti di calcolo in virgola mobile altamente parallelizzati.
- **2005-oggi:** Investimenti massivi nel settore GPU



Avatar: costo 300 mln \$,
incassi : (prima settimana) 77 mln \$

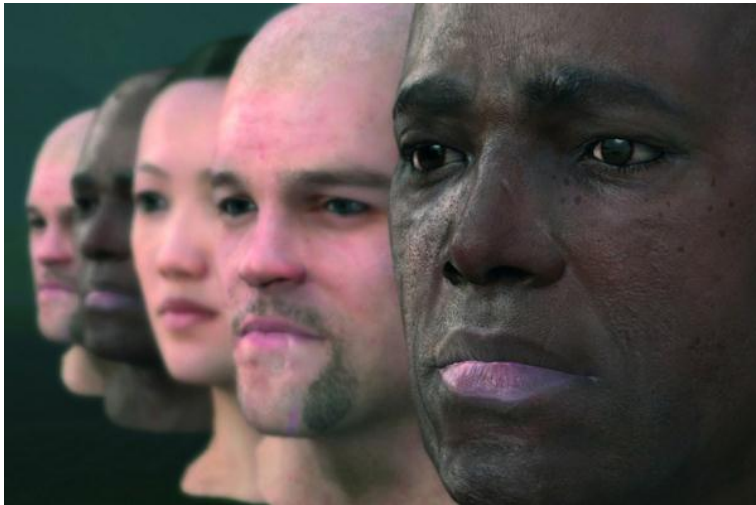


Call of Duty M.W. 3: costo 100 mln \$,
incassi : (prima settimana) 738 mln \$

Situazione attuale GPU



- ❑ il **realismo** consiste nel riuscire a simulare un mondo reale, tramite lo studio delle luci, la fisica dei corpi, le animazioni facciali e di movimento, intelligenza artificiale, visione stereoscopica, ...
- ❑ Alle GPU è richiesto una maggiore flessibilità di calcolo, non si richiede più di aumentare soltanto numero di poligoni



CryEngine 2



MotionScan

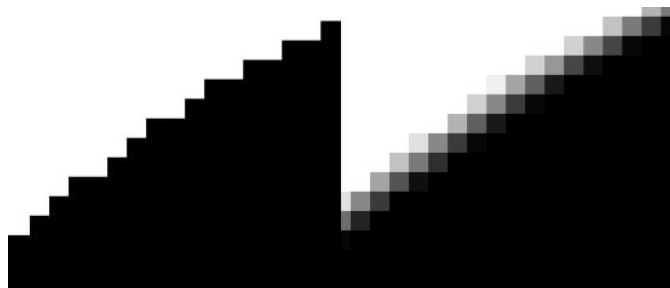
Tecniche di Analisi di Immagini



- Alcune delle tecniche viste a lezione sono implementate dalle GPU nei moderni videogiochi nella fase di **Pre o Post Processing**:



Trasformazioni geometriche di immagini:
(Doom le utilizza per simulare il 3D)



Anti-Aliasing

L'antialiasing ammorbidisce le linee smussandone i bordi e migliorando l'immagine.

Tecniche di Analisi di Immagini



MotionBlur : Applicazione di uno speciale filtro di blur che coinvolge la scena circostante al punto di fuga.

Tecniche di Analisi di Immagini



Filtro Anisotropico: accresce la qualità delle immagini in cui sono presenti tessiture inclinate rispetto al punto di osservazione

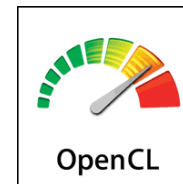
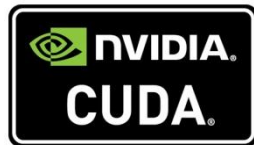
Situazione attuale GPU



- Utilizzo delle GPU era relegato alle interfacce:



- Le ultime tendenze dell'industria videoludica hanno permesso l'evoluzione delle GPU.
- Specific Purpose (calcoli floating point) → General Purpose.
- Dal 2007 è possibile iniziarle ad usare per calcolo scientifico:
Alcuni linguaggi per creare applicativi per la GPU vengono rilasciati:



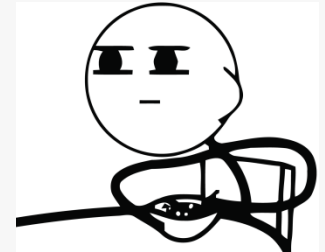
- La maggior parte dei centri di ricerca utilizza queste tecnologie al giorno d'oggi (si parla di GPU revolution, ma perchè?).

CPU vs GPU



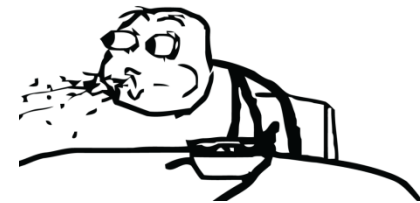
Confronti specifiche:

	Intel Core i7 3960x	GTX 690
Frequenza max	3.9 GHz	1 GHz
Memoria Cache	15 Mb	4 Kb
Memoria Globale Max	64 Gb	2 Gb
Numero di operazioni	Migliaia	Poche decine



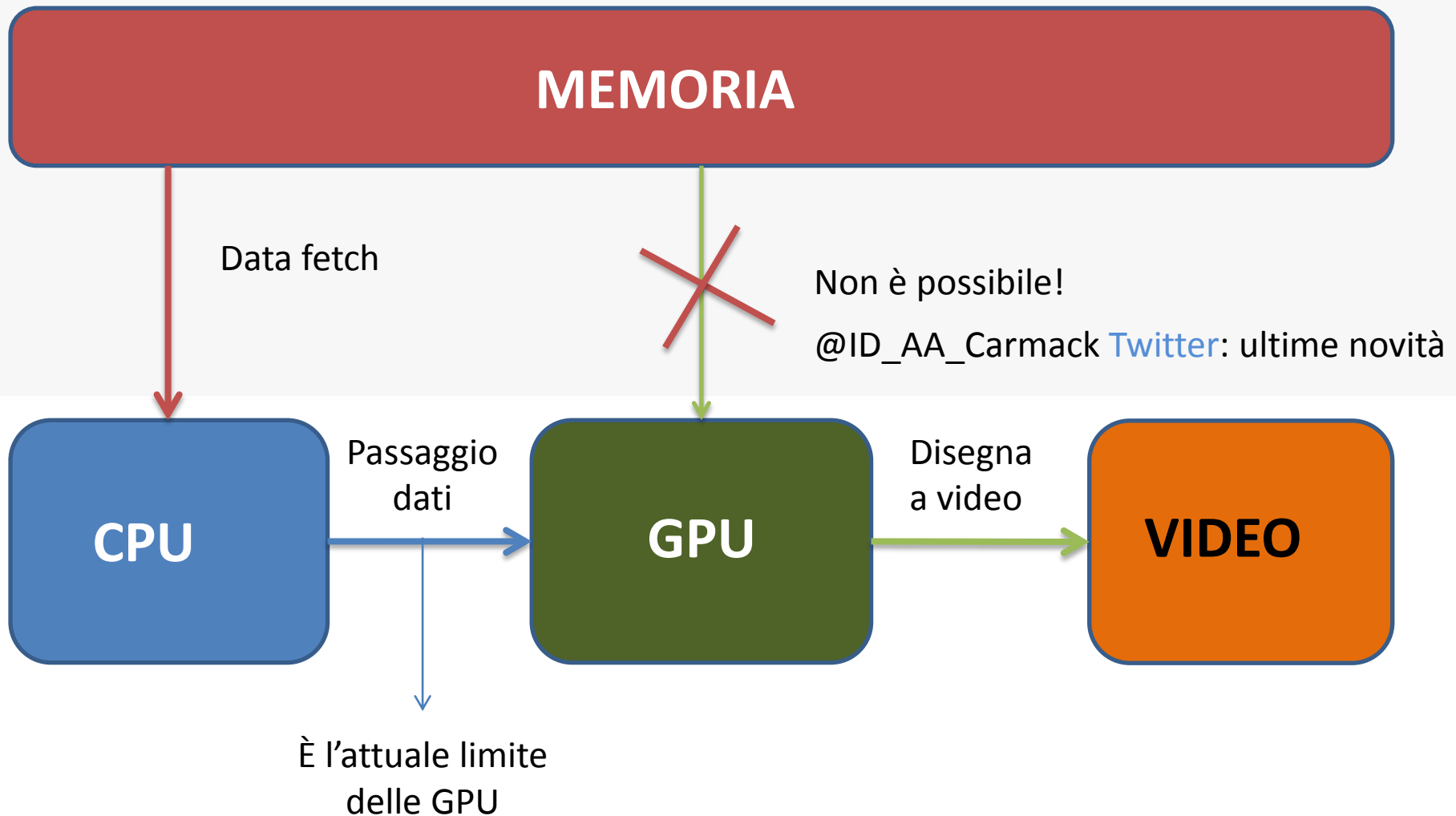
Vantaggi GPU

	Intel Core i7 3960x	GTX 690
# Cores	6	36'864
Max # Threads	12	Più di 1'000'000
Bandwith Memoria	51 Gb/s	384 Gb/s



Le GPU presentano un hardware particolare...

Schema Funzionamento GPU



Perché GPU Nvidia?



- CUDA = **Compute Unified Device Architecture**
Al momento il miglior sistema di controllo per Massively Parallel Processors.
- CUDA-C , offre un completo ambiente di Sviluppo, numerose librerie sviluppate da terze parti.



- Numerosi Tools e risorse ([Nvidia Developer Zone](https://developer.nvidia.com))
- È molto performante rispetto alle alternative (OpenCL)
- Schede Nvidia hanno processori orientati più verso il General Purpose (supportano via hardware Phisyx)
- C, C++, Java, Fortran, Python, Matlab, ...



1. **Concetti di programmazione concorrente.**
2. Dettagliata **conoscenza dell'hardware utilizzato** per creare applicativi efficienti.

1) Programmazione Concorrente



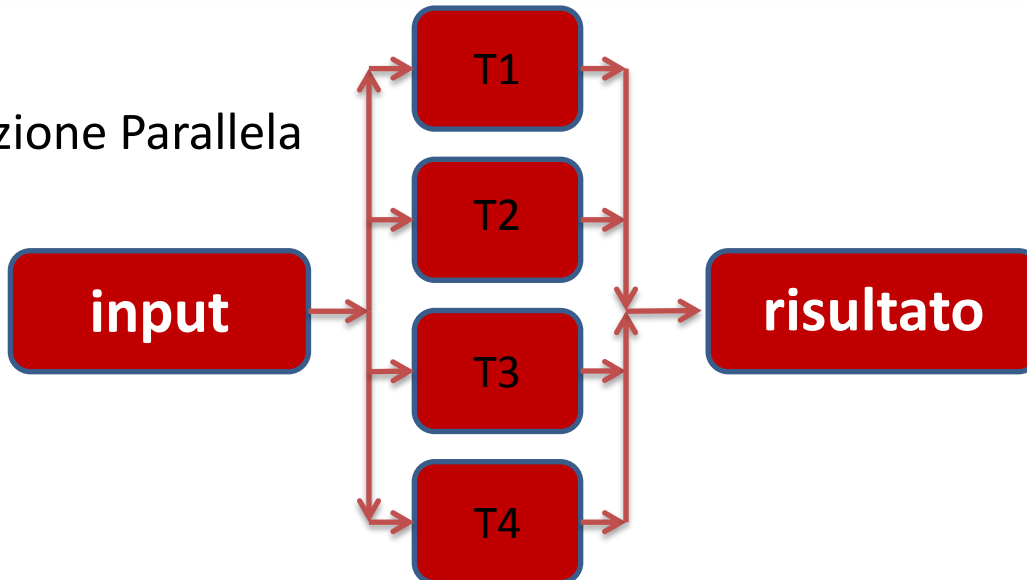
- Esecuzione tradizionale: modello pipeline



Tempo :




- Esecuzione Parallela



I threads sono solitamente disponibili in numero limitato

1) Programmazione Concorrente



- Non tutti i task sono parallelizzabili:
 - ❖ Ruotare un'immagine ?
 - ❖ Applicare un filtro BLUR ad un'immagine?
 - ❖ Simulare il comportamento del sistema solare?
 - ❖ Contare fino a 10?
- Il grado di parallelismo di un'attività è ricavabile da analisi del Task Dependency Graph (o empiricamente). Il livello di parallelizzazione può variare da
 - ❖ Non parallelizzabile (avvio del sistema)
 - ❖ Embarassingly Parallel (somma di due vettori)

1) Programmazione Concorrente



- Per dimostrare l'efficacia della parallelizzazione di un processo, si possono tenere in conto diverse misure.
- Una delle più utili è lo speed-up, dove si misura il rapporto fra il tempo di esecuzione dell'algoritmo in parallelo con l'algoritmo seriale.

$$S_{\text{speedUp}} = \frac{T_{\text{seriale}}}{T_{\text{parallelo}}}$$

1) Programmazione Concorrente



- La programmazione su GPU affronta un particolare tipo di programmazione parallela dove **i thread disponibili possono essere considerati infiniti**
- Infiniti = l'utente teoricamente può allocare più di *4'000 miliardi di threads* indipendenti.
- In questa configurazione, spesso, la soluzione parallela greedy può essere la migliore. Analizziamo alcune tecniche applicabili



- Ruotare un'immagine (esecuzione SERIALE) :

```
For i=0:img.size.x
    For j=0:img.size().y
        img(i,j).coord = newCoord(i,j);
    end
end
```

- Ruotare un'immagine (esecuzione PARALLELA GPU):
prendo un numero di thread = pixel Immagine

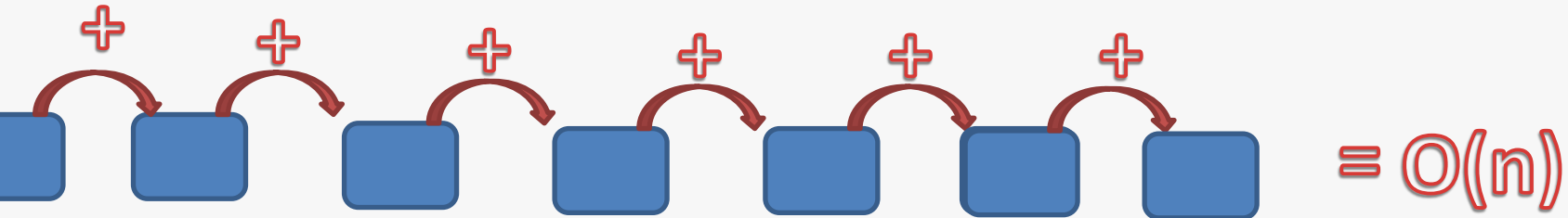
```
Img(thread.Id.x, thread.Id.y) =  
    newCoord(thread.Id.x, thread.Id.y)
```

$$O(n^2) \sim O(1) \rightarrow S_{\text{speedUp}} = n^2$$

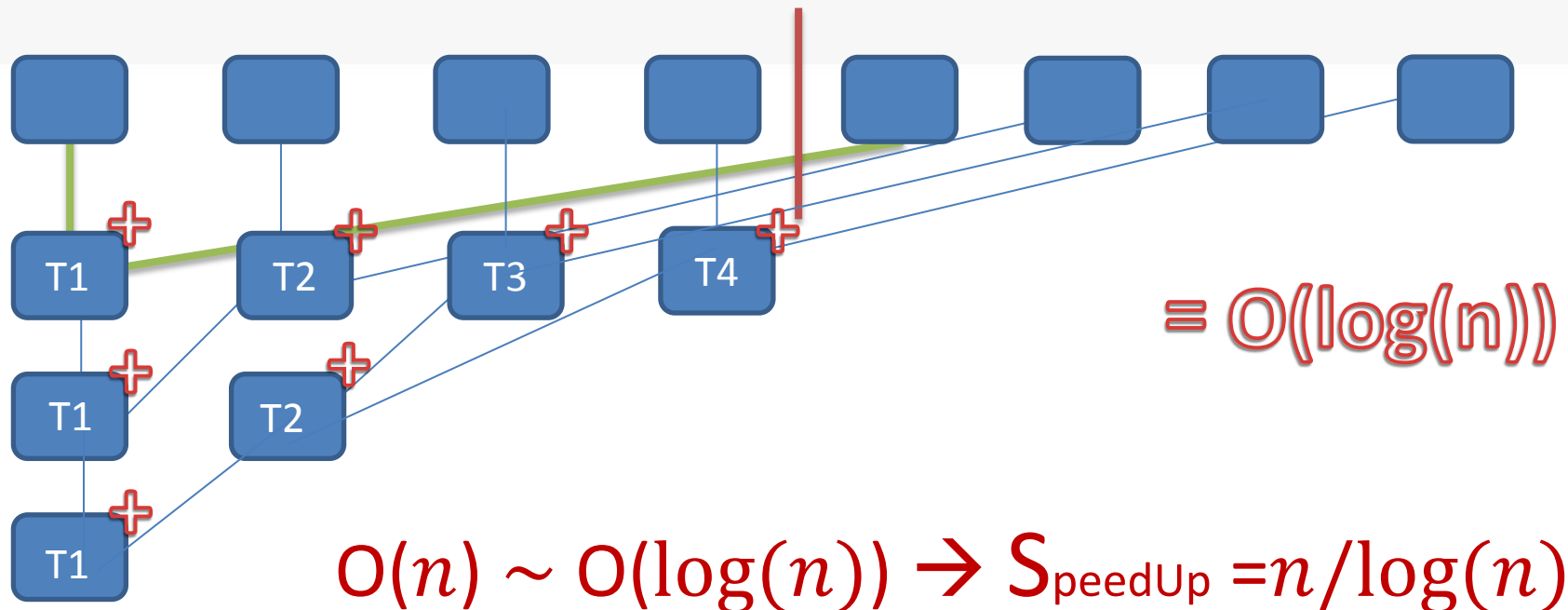
Programmazione GPU



- Somma degli elementi di un vettore (esecuzione SERIALE) :



- Somma degli elementi di un vettore (GPU)



2) Architettura delle GPU

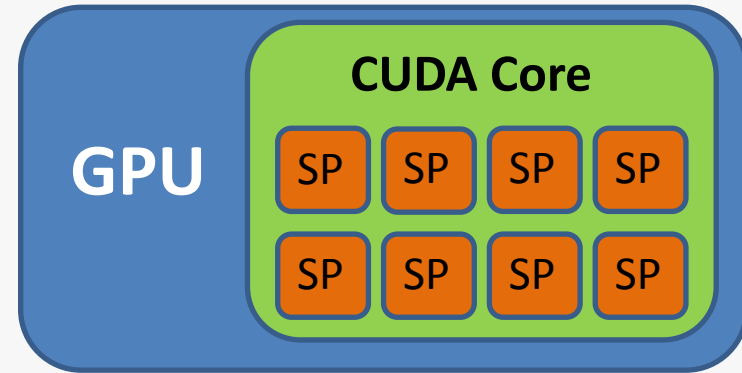


- Il passaggio dallo pseudocodice precedente alla soluzione in CUDA-C è immediato?
- No, in CUDA-C è l'hardware a comandare.
- Un algoritmo anche se altamente parallelizzabile può essere messo in ginocchio da overhead legati all'hardware.

2) Architetture Parallele:



- Le GPU Nvidia si basano su architetture parallele, chiamate CUDA cores
- Ogni singolo CUDA core contiene Single Processors al suo interno.



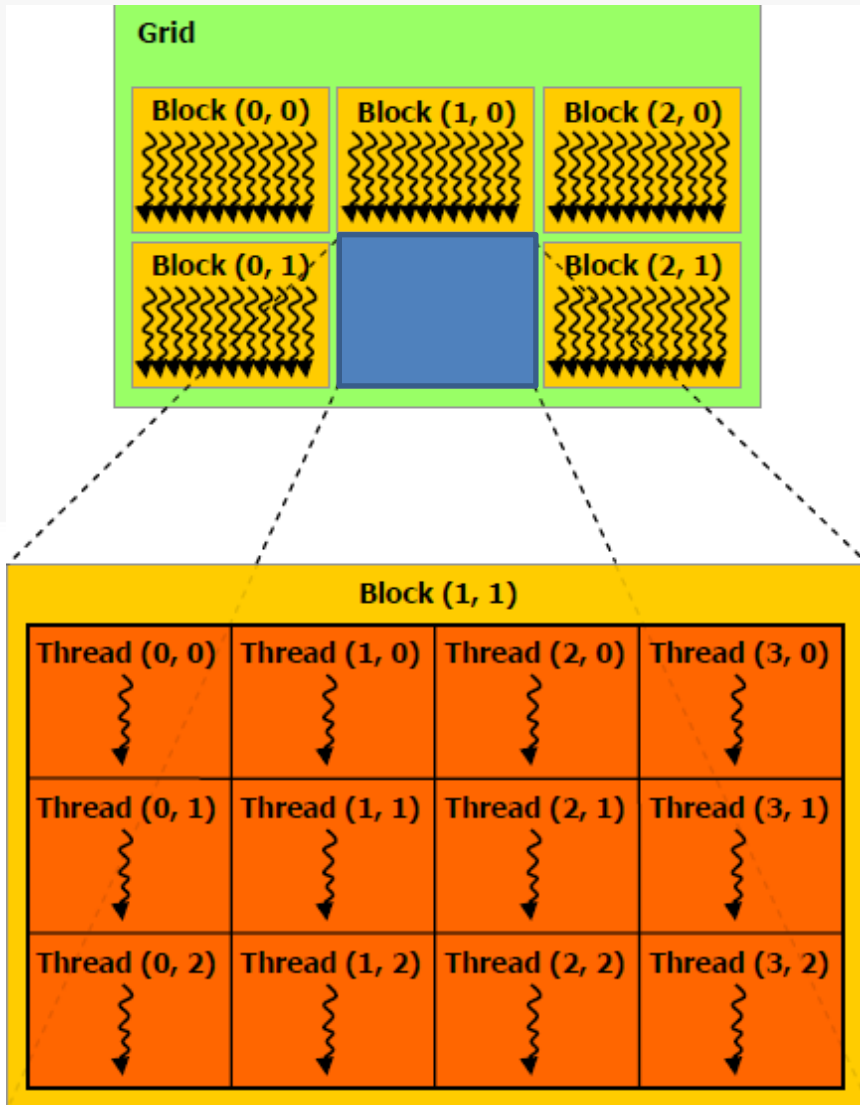
- (Single Processor) **SIMD** = **S**ingle **I**nstruction **M**ultiple **D**evices.
 - ❖ *Sincroni, Memoria condivisa, Iniz. Veloce, stessi comandi, Economici*
- (CPU) **MIMD** = **M**ultiple **I**nstructions **M**ultiple **D**evices
 - ❖ *Asincroni, Memoria privata, Iniz. Costosa, indipendenti, Costosi*



- In CUDA-C, il programming Model si fonde con conoscenza dell'hardware.
 - Concetti di **Kernel, Blocks e Threads**
- ❖ **KERNEL**: una funzione che è eseguita sulla GPU. Kernel diversi sono Asincroni. La sua visibilità può essere :
- globale*** : richiamabile dall'esterno.
 - device*** : non può essere richiamata dall'host ma solo all'interno del device
 - host*** : viene eseguita sul device chiamante.

Il Kernel è rappresentato dal codice che viene eseguito da ogni thread.

Grids e Blocks



- ❖ GRID: matrice 2D, massime dimensioni [65535, 65535] (TESLA)
Ogni elemento (blocco) ha una quantità di memoria privata ed una matrice di threads.

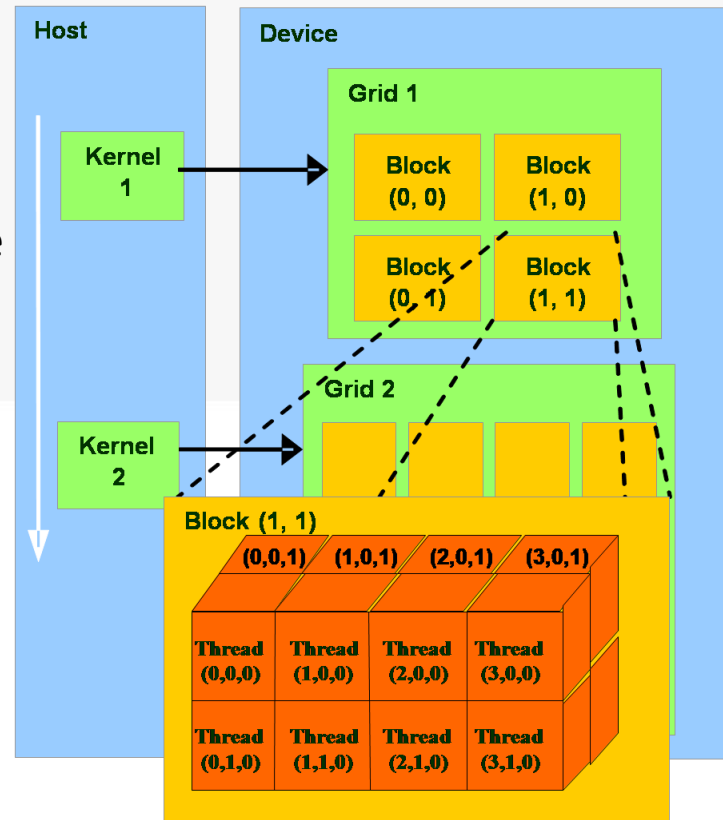
- ❖ THREAD: matrice 3D, massimo numero di elementi: 1024 (TESLA)
Ogni thread è l'unità di lavoro che esegue il codice specificato nel kernel.

I thread sono i lavoratori 'atomici'.

Quadro Generale



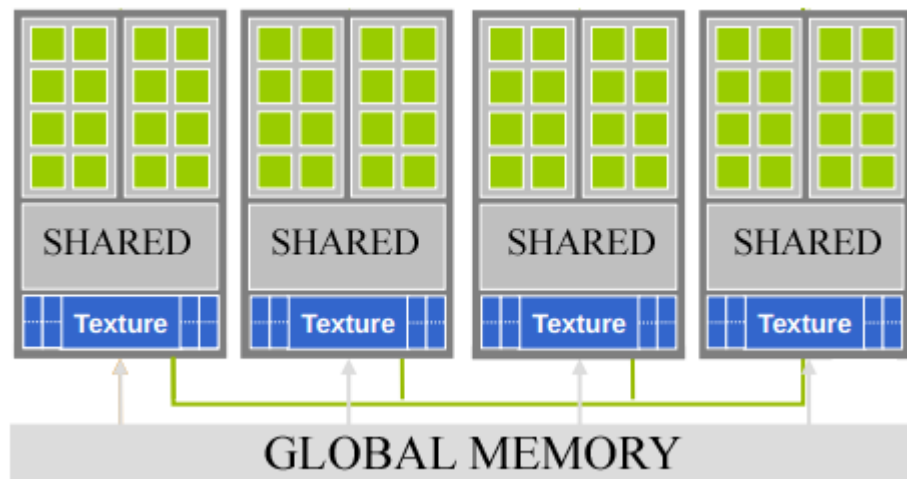
1) Chiamate asincrone a kernels si possono seguire nel codice chiamante



2) Per ogni Kernel viene istanziato 1 GRID di *Blocks* di dimensioni definite dall'utente

3) Per ogni Block viene istanziato una matrice di *Threads* dimensioni definite dall'utente

La memoria video



- La gestione della memoria è fondamentale per le performance.

NUMA : Not Uniform Memory Access

E' possibile dichiarare diversi tipi di variabili:

- **__global__** : memoria principale della GPU | **scope** : *tutti i threads* → **LENTA**
- **__shared__** : memoria del blocco | **scope** : *thread blocco* → **VELOCE**
- **__texture__** : memoria del blocco costante | **scope** : *thread blocco* → **VELOCE**
- **__constant__** : memoria principale, costante | **scope** : *tutti i thread* → **VELOCE**
- **__register__** : memoria del thread (4Kb), **scope** : *thread* → **MOLTO VELOCE**



- Trasferimenti di memoria :
Il più grosso limite delle GPU, consiste nel dover richiedere alla CPU i dati da elaborare.
Soluzione : concentrare tutti i trasferimenti di memoria, richiederne il meno possibile, lavorare con i dati in memoria.
- Accesso alla memoria globale:
è il tipo di memoria più lento delle GPU
Soluzione: accedervi il meno possibile, accesso '*coalescence*'
- Inizializzazione blocco:
Molto più costosa dell'inizializzare un thread
Soluzione: aumentare la granulosità dell'applicazione

*Più un problema è diviso in un numero grande di threads più è detto **granulare**.*

Un problema ben posto



Deve soddisfare i seguenti criteri

- Massivamente parallelo
 - Capace di suddividere il calcolo in centinaia o migliaia di task indipendenti
- Computazionalmente Intensivo
 - Il tempo di Computazione deve eccedere il tempo di trasferimento dei dati in memoria

Se questi criteri sono soddisfatti, il problema riceverà speed up.

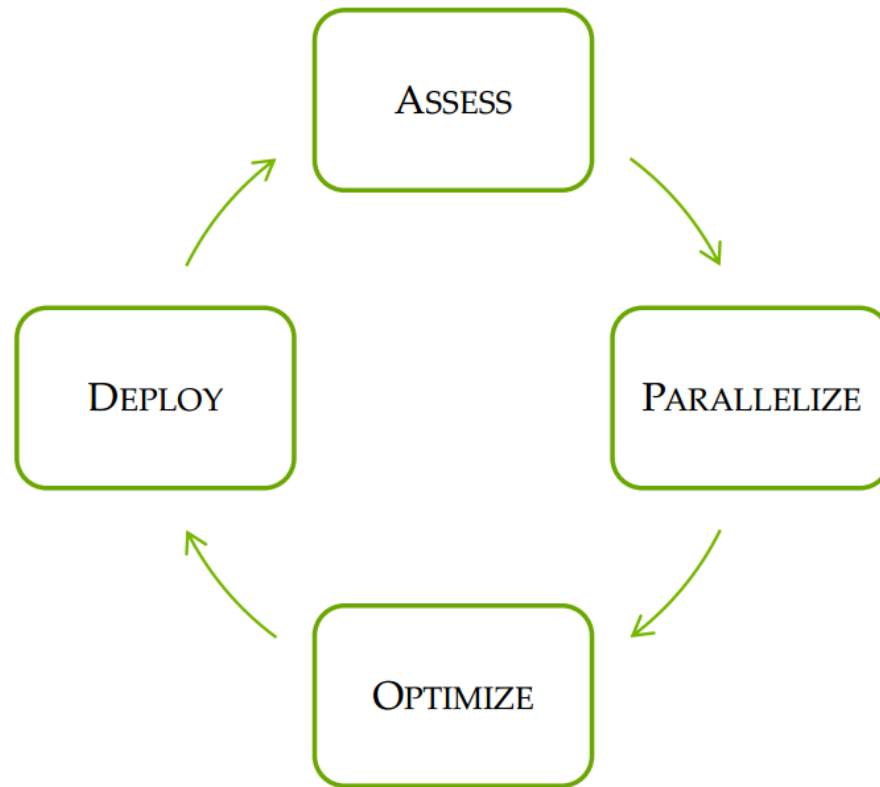
Altre Considerazioni su CUDA



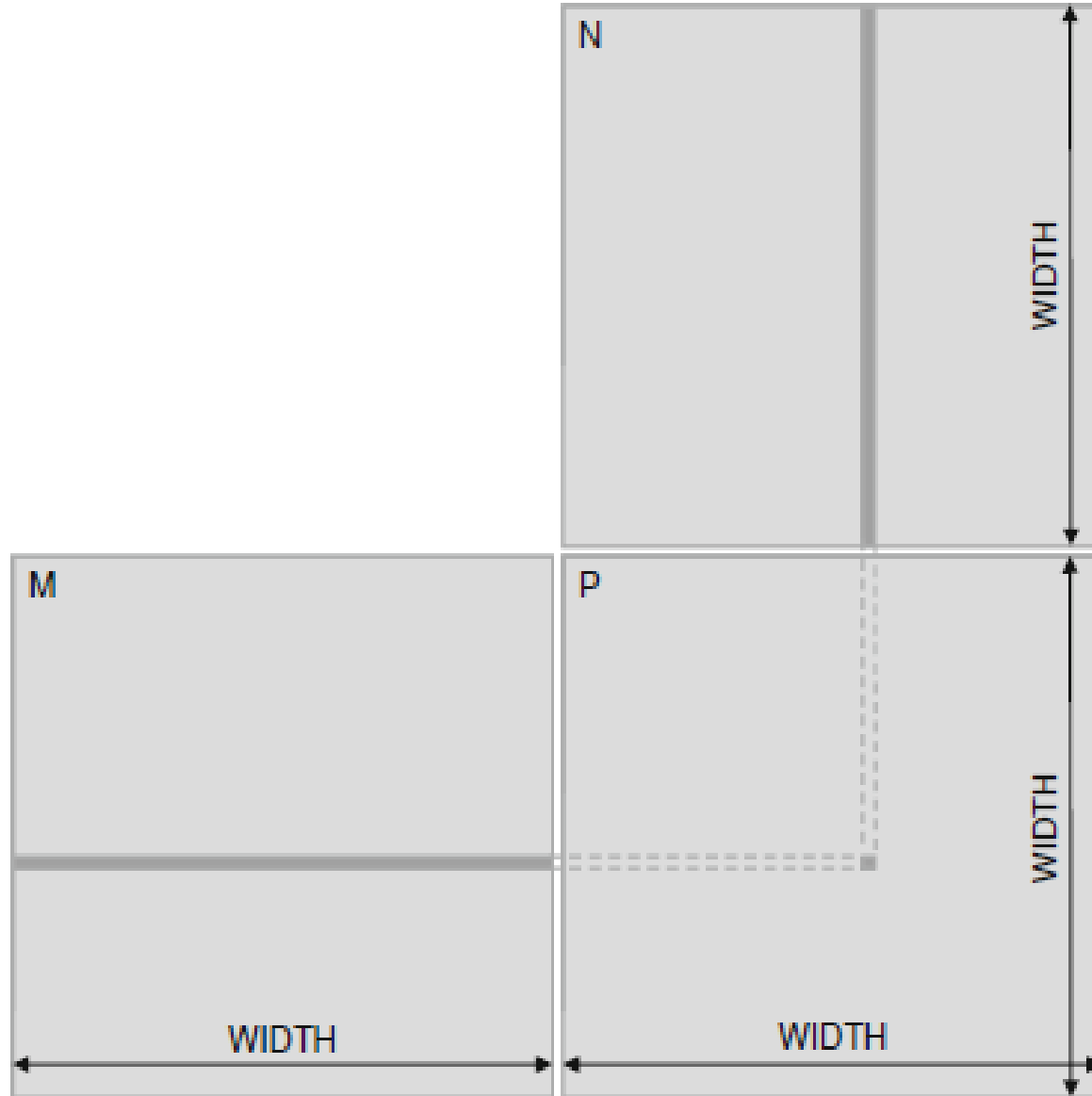
- Le GPU complementano i processori, non possono sostituirli
 - I processori sono divisi in integer o floating points, basse performance sugli integer.
 - Al momento le GPU non possono accedere alla RAM
-
- Non è possibile sfruttare tutte le caratteristiche del C. (Doppi puntatori non sono supportati)
 - Esecuzione non deterministica!
(Dipende dal carico sul bus e da altri fattori, riprovare gli esperimenti).
 - Un programma che occupa pochi core è inutile.



Schema generale



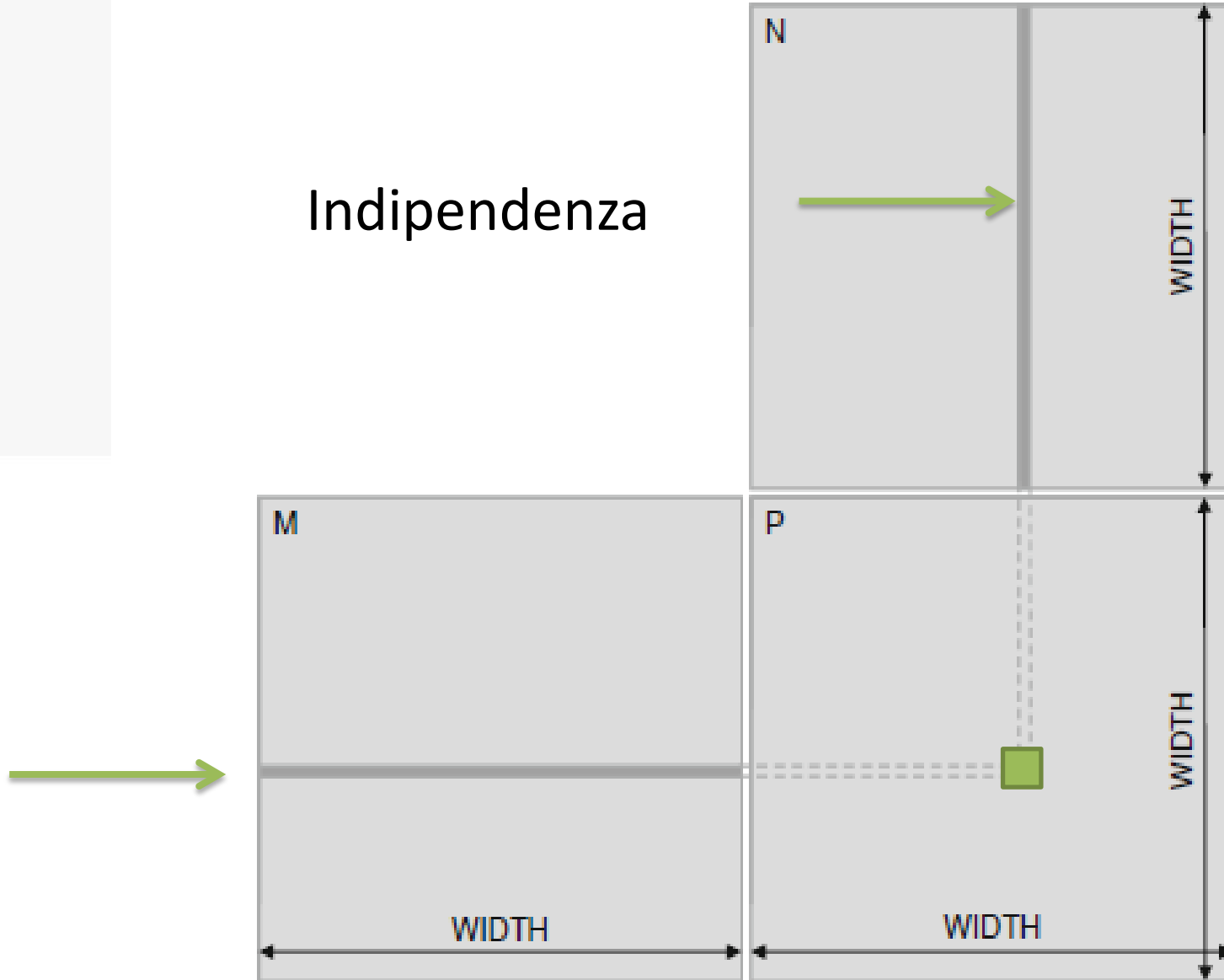
Moltiplicazione di due matrici



1) Stabile



Indipendenza



2) Parallelizzare



Grid : [M/k | N/k] Block : [$TILE_WIDTH$ | $TILE_WIDTH$]

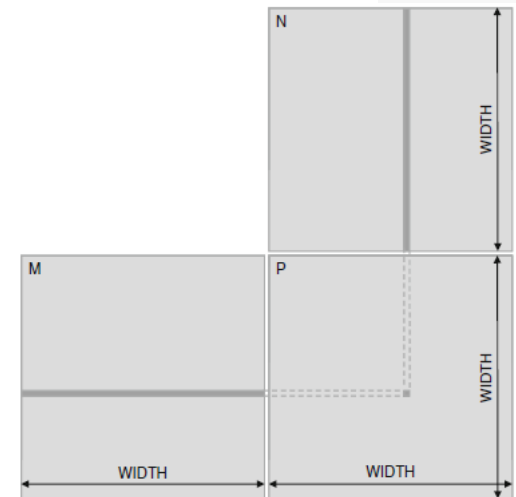
```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
```

```
    int tx = threadIdx.x; ←←←
    int ty = threadIdx.y; ←←←
```

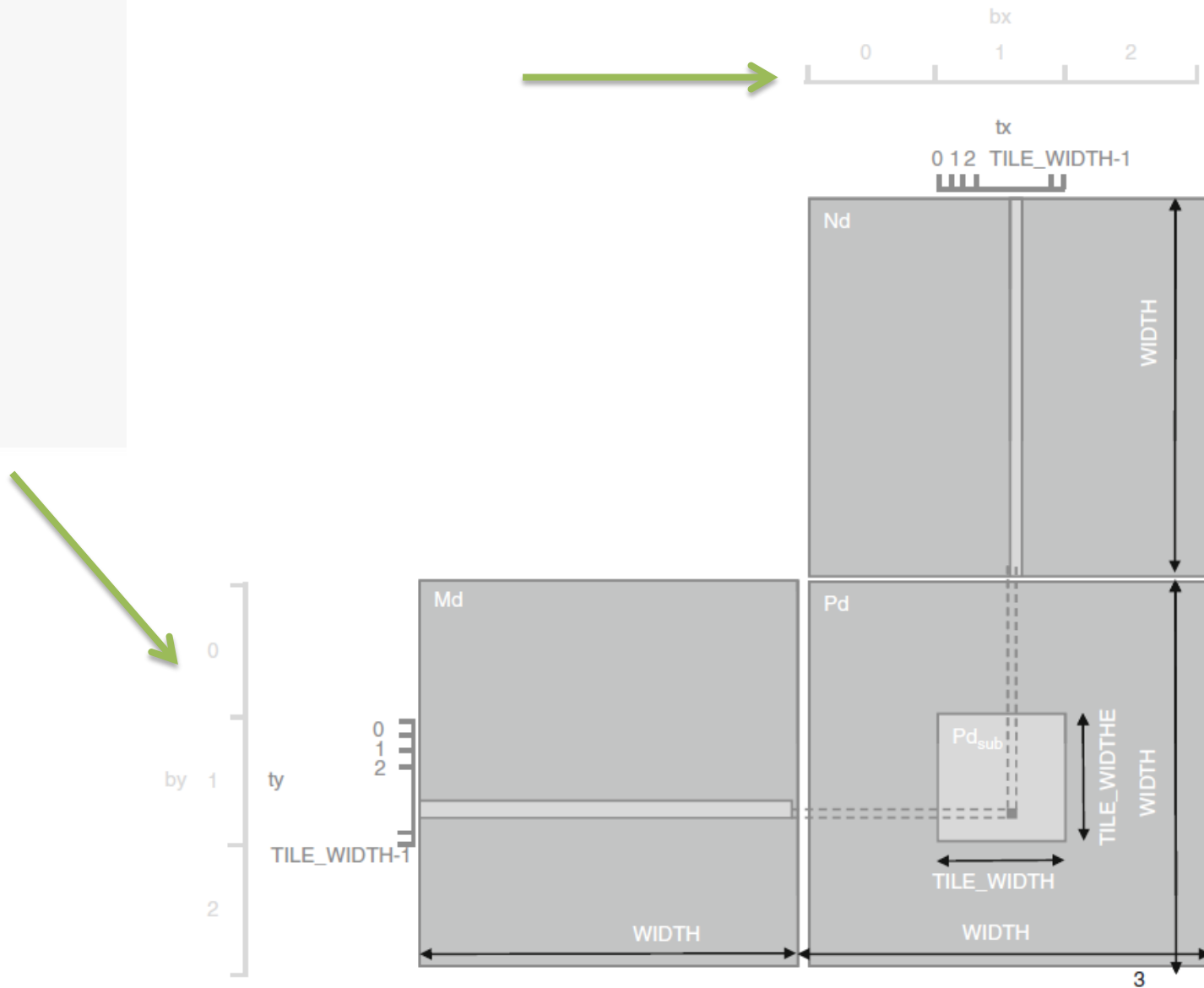
```
    float Pvalue = 0;
```

```
    for (int k = 0; k < Width; ++k)
    {
        float Mdelement = Md[ty * Width + k];
        float Ndelement = Nd[k * Width + tx];
        Pvalue += Mdelement * Ndelement;
    }
```

```
    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}
```



3) Ottimizzare

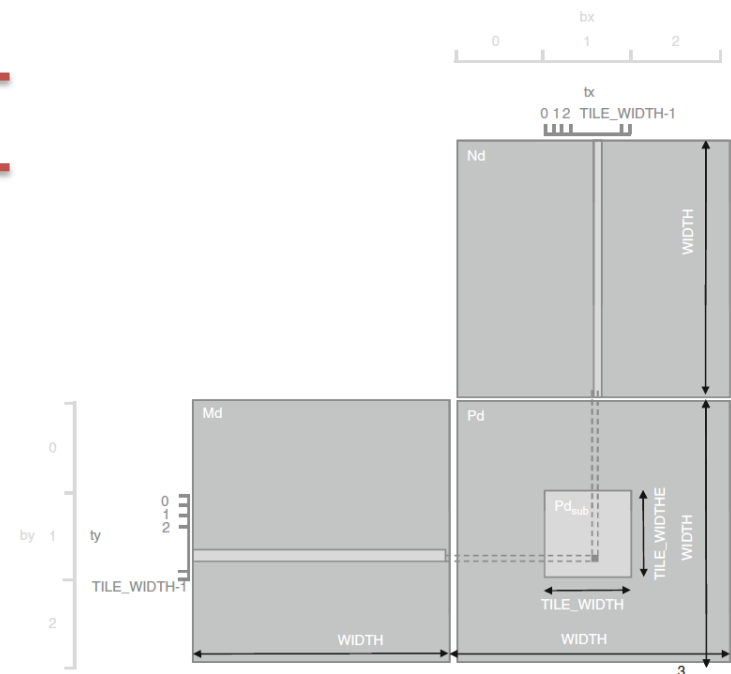


4) Concludere



Grid : [M/k | N/k] Block : [$TILE_WIDTH$ | $TILE_WIDTH$]

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
    int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;
    float Pvalue = 0;
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];
    Pd[Row*Width+Col] = Pvalue;
}
```



Kernel migliore → meno codice

Lo zen e l'arte dell'ottimizzazione



- Costo degli operatori: quello che non ti aspetti
 - op. ***bitwise, confronti, incrementi***, ... = 4 cicli di clock
 - op. *******, ***+***, ***-***, ***:***, ... = 16 cicli di clock
 - fn. ***sen/cos/log*** = 32 cicli di clock
- Scrivere ++k invece di k=k+1 rende il programma 4 volte più veloce
- Single o Double?
- Per ogni bit risparmiato in un'operazione, si incrementa la velocità (`__mul24`, `__mul16`, ...). Cast estremamente costoso
- Accesso Coalesced : salva tempo.
- Accedere ad una matrice per colonne invece che per righe aumenta le prestazioni di circa 6 volte.



- Matlab si interfaccia con CUDA tramite una libreria denominata il parallelToolBox.
- Il parallelToolBox è integrato dalla versione 2010 in poi di MatLab.
- I requisiti richiedono una GPU NVIDIA con capacità computazionali 1.3 o superiori
- Per l'installazione vedere il relativo documento

Utilizzare GPU su MatLab



- Le opzioni per utilizzare il supporto alle GPU su MatLab sono 3:

Facilità d'uso

1) Utilizzare gpuArray con funzioni pre-costruite

2) Funzioni personalizzate sugli elementi del gpuArray

3) Utilizzo di Kernel generati da CUDA e files .PTX

Controllo e Prestazioni

1) gpuArray



- Per allocare un'array in memoria basta dichiararlo nel seguente modo:

```
>> array = gpuArray(0:20);
```

- E' possibile eseguire dei calcoli con l'array in memoria utilizzando le funzioni già costruite di MatLab (che supportano la gpu)
- Per ottenere i risultati in forma leggibile da un umano, è necessario richiedere l'array alla gpu tramite il comando

```
>> array_answer = gather(array);
```

2) Utilizzo di funzioni personalizzate

- È necessario allocare uno o più array sulla GPU

```
>> array_1 = gpuArray(0:20);  
>> array_2 = gpuArray(0:20);
```

- Per poi richiamare la funzione

```
>> result= arrayfun(@myFunction,array_1,array_2);
```

- La funzione myFunction è definita dall'utente nel file myFunction.m e viene eseguita per ogni membro degli arrays.

3) CUDA kernels in MatLab



- È necessario creare un kernel, utilizzando il linguaggio CUDA-C.
- Successivamente è necessario compilare il file .cu con il compilatore nvcc utilizzando l'opzione -ptx.
- Una volta generato il file .ptx è possibile caricare in MatLab il kernel come un oggetto utilizzando:

```
>>k = parallel.gpu.CUDAKernel('myfun.ptx', 'myfun.cu');
```

- Una volta caricato l'oggetto kernel è possibile modificarne i parametri:

```
>> k.ThreadBlockSize = [24 24];  
>> k.GridSize = [45650 45650];
```

- Per eseguire il Kernel digitare

```
>>o = feval(k, ones(N, 1), ones(N, 1));
```

Fine parte teorica



Ci si vede all'esercitazione!

