



# **Metodi Numerici per l'Informatica**

Appunti

Carmine D'Angelo, Luca Boffa



Figure 1: Carmine D'Angelo in tutto il suo splendore

APPUNTI DI LUCA BOFFA, UNIVERSITY OF SALERNO

[HTTPS://GITHUB.COM/LUKE31999](https://github.com/Luke31999)

Questi appunti sono stati scritti da Carmine D'Angelo noto come l'infopoint del dipoartimento di Informatica. Luca Boffa ne ha fatto una riscrittura fedele riadattandole in un design più moderno. Il materiale utilizzato è stato preso dalle dispense ufficiali della Prof.ssa. Angelamaria Cardnone

*Scritti durante il primo semestre dell' anno accademico 2022/2023*

*Seconda Edizione*



# Contents

<b>1</b>	<b>Introduzione . . . . .</b>	<b>7</b>
1.1	<b>Tassonomia di Flynn</b>	<b>8</b>
1.2	<b>Principi del calcolo parallelo</b>	<b>8</b>
1.2.1	Speed-up . . . . .	8
1.2.2	Efficienza . . . . .	9
1.2.3	Granularità . . . . .	9
1.2.4	Accessi in memoria . . . . .	10
1.2.5	Carico non bilanciato . . . . .	10
1.2.6	Legge di Amdahl . . . . .	10
1.3	<b>Cenni finali</b>	<b>11</b>
<b>2</b>	<b>Somma di N numeri in parallelo . . . . .</b>	<b>13</b>
2.1	<b>Somma prima strategia di parallelizzazione</b>	<b>13</b>
2.1.1	Lavagna . . . . .	14
2.2	<b>Somma seconda strategia di parallelizzazione</b>	<b>14</b>
2.2.1	Distribuzione dei dati . . . . .	15
2.2.2	Lavagna . . . . .	15
2.3	<b>Somma terza strategia di parallelizzazione</b>	<b>16</b>
2.3.1	Lavagna . . . . .	16
2.4	<b>Legge di amdhal dimostrazione</b>	<b>17</b>
<b>3</b>	<b>Prodotto matrice-vettore . . . . .</b>	<b>19</b>
3.1	<b>Algoritmo seriale(sequenziale)</b>	<b>19</b>

<b>3.2</b>	<b>Prima strategia suddivisione in blocchi di righe</b>	<b>19</b>
3.2.1	Funzionamento algoritmo .....	20
3.2.2	Valutazione dell'algoritmo .....	20
<b>3.3</b>	<b>Seconda strategia distribuzione per blocchi di colonne</b>	<b>21</b>
3.3.1	Funzionamento algoritmo .....	21
3.3.2	Lavagna .....	21
<b>3.4</b>	<b>Topologie di processori</b>	<b>23</b>
3.4.1	Topologie .....	23
3.4.2	Parallelizzazione del problema: distribuzione a griglia cartesiana .....	23
3.4.3	Sottogrille .....	26
<b>3.5</b>	<b>Terza strategia distribuzione per schema a blocchi</b>	<b>28</b>
3.5.1	Algoritmo .....	28
3.5.2	Lavagna .....	28
<b>4</b>	<b>GPGPU (General Purpose GPU)</b> .....	<b>31</b>
<b>4.1</b>	<b>GPU</b>	<b>31</b>
4.1.1	Architettura CPU vs GPU .....	31
4.1.2	CUDA (Compute Unified Device Architecture) .....	31
4.1.3	Memorie di una GPU .....	33
4.1.4	Architetture della GPU .....	33
4.1.5	Modello di esecuzione di CUDA .....	33
4.1.6	Proprietà di CUDA .....	33
4.1.7	Latenza .....	34
4.1.8	Compute Capability .....	34
4.1.9	Come funziona uno streaming multi processor (SM) .....	36
<b>4.2</b>	<b>Somma di due Vettori</b>	<b>38</b>
4.2.1	Codice .....	38
4.2.2	Lavagna .....	40
<b>4.3</b>	<b>Esempio di capability</b>	<b>41</b>
4.3.1	Configurazione di un kernel .....	41
<b>4.4</b>	<b>Somma di due matrici</b>	<b>42</b>
4.4.1	Introduzione .....	42
4.4.2	Esempio con FERMI .....	42
4.4.3	Esempio con KEPLER .....	42
4.4.4	Lavagna .....	43
<b>4.5</b>	<b>Memorie di una GPU</b>	<b>44</b>
<b>4.6</b>	<b>Schema delle memorie</b>	<b>44</b>
4.6.1	Memoria globale .....	44
4.6.2	Memoria condivisa .....	44
4.6.3	Registri .....	44
4.6.4	Memoria locale .....	44
4.6.5	Memoria costante .....	44

4.6.6	Memoria Texture .....	45
<b>4.7</b>	<b>Shared memory e prodotto scalare</b>	<b>45</b>
4.7.1	Shared memory .....	45
4.7.2	Organizzazione della shared memory .....	46
4.7.3	Lavagna tipi di accessi al banco .....	47
<b>4.8</b>	<b>Prodotto scalare</b>	<b>48</b>
4.8.1	Kernel strategia 1 (banale) .....	48
4.8.2	Come calcolare C a partire da V .....	48
4.8.3	Riduzione .....	48
<b>4.9</b>	<b>Allocazione statica</b>	<b>49</b>
4.9.1	Esempio 1 .....	49
4.9.2	Esempio 2 .....	49
<b>4.10</b>	<b>Allocazione dinamica</b>	<b>49</b>
4.10.1	Esempio 3 .....	49
<b>4.11</b>	<b>Info su shared memory</b>	<b>50</b>
<b>4.12</b>	<b>Lavagne prodotto scalare</b>	<b>50</b>
4.12.1	Prima strategia (banale) .....	50
4.12.2	Seconda strategia .....	50
4.12.3	Terza strategia .....	54

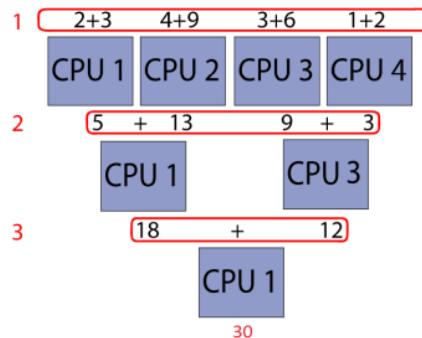


# Distributed Computing

# Parallel Computing

## 1. Introduzione

Il calcolo sequenziale risolve un problema tramite un algoritmo, le cui istruzioni sono eseguite in sequenza mentre con il calcolo parallelo il problema viene suddiviso in sequenze discrete di istruzioni che vengono eseguite una dopo l'altra.



- Parallelismo temporale (catena di montaggio, pipeline): la presenza della pipeline aumenta il numero di istruzioni contemporaneamente in esecuzione sfruttando anche i tempi morti di un processore per iniziare un nuovo ciclo computazionale. Quindi, introducendo il pipelining nel processore, aumenta il throughput (numero di istruzioni eseguite nell'unità di tempo) ma non si riduce la latenza della singola istruzione (tempo di esecuzione della singola istruzione, dal suo inizio fino al suo completamento), le pipeline devono operare in modo sincrono: questo comporta che lo stadio più lento determini la lunghezza di ogni fase della pipeline;
- Parallelismo spaziale al problema esposto: consiste nell'esecuzione contemporanea della stessa operazione su dati diversi, quindi, sono necessarie più unità aritmetico-logiche (ALU);
- Parallelismo asincrono.

## 1.1 Tassonomia di Flynn

La tassonomia di Flynn ci mostra quelle che sono le quattro architetture di elaboratori possibili:

- **SISD** (single instruction single data): un singolo set di dati una singola istruzione per volta esempio calcolatori sequenziali, nessun parallelismo possibile, le operazioni vengono eseguite sequenzialmente, su un solo dato alla volta.
- **SIMD** (single instruction multiple data): architetture composte da molte unità di elaborazione che eseguono contemporaneamente la stessa istruzione ma lavorano su insieme di dati diversi.
- **MISD** (multiple instruction single data): più flussi di istruzioni lavorano contemporaneamente su un unico flusso di dati. Non viene usata perchè è inutile.
- **MIMD** (multiple instruction multiple dat): più flussi di istruzioni lavorano contemporaneamente su set di dati diversi (il vero parallelismo) dove ho bisogno sia di più unità di elaborazione che più unità di controllo. Abbiamo due tipi di memorie MIMD:
  - **Memoria distribuita**: ogni processore ha la propria memoria e può accedere ai dati di un altro processore con lo scambio di messaggi, un problema è la comunicazione tra processori.
  - **Memoria condivisa**: tutti i processori accedono alla stessa memoria ma ci sono problemi di sincronizzazione e di sicurezza.

## 1.2 Principi del calcolo parallelo

- Valutazione di un algoritmo parallelo (Speed-Up e Efficienza);
- Trovare e sfruttare la granularità;
- Preservare la località dei dati;
- Bilanciamento del carico computazionale;
- Coordinamento e sincronizzazione;

### 1.2.1 Speed-up

Lo speed-up misura la riduzione del tempo di esecuzione rispetto all'algoritmo su di 1 processore. Dato un problema di dimensione fissata, siano  $T_s$  : tempo di esecuzione seriale (in sec) e  $T_p$  : tempo di esecuzione parallelo usando  $p$  processori (in sec):

$$S_p = \frac{T_s}{T_p} \quad (1.1)$$

dove  $T_s$  è il tempo algoritmo sequenziale e  $T_p$  è il tempo dell'algoritmo parallelo.

Si parla di *speed-up superlineare* quando  $S_p > p$ , può accadere nei seguenti casi:

1. Gestione ottimale della gerarchia di memoria (ad es. aumento della cache rispetto ad un calcolatore monoprocesso);
2. Differente ottimizzazione del codice;
3. Ordine di visita di grafi nei problemi di ricerca (suddividendo il grafo, uno dei processori potrebbe trovare prima l'oggetto della ricerca).

*Speedup assoluto*: Tempo di esecuzione sequenziale/tempo di esecuzione parallelo su  $p$  processori. *Speedup relativo*: Tempo di esecuzione parallelo su 1 processore/tempo di esecuzione parallelo su  $p$  processori Restituiscono valori diversi perchè un codice parallelo eseguito su un solo processore inserisce overhead superflui, e dunque può essere più lento del codice sequenziale.

*Speedup scalato:* Misura la scalabilità, ossia la capacità di un algoritmo parallelo di risolvere problemi «grandi», indichiamo con  $N$  la dimensione del problema, idealmente  $SS_p = 1$ :

$$SS_p = \frac{p * T_s(N)}{T_p(pN)} \quad (1.2)$$

### 1.2.2 Efficienza

L'efficienza misura quanto l'algoritmo sfrutta il parallelismo del calcolatore, ovvero le risorse che ha a disposizione.<sup>1</sup>. L'efficienza è una misura relativa al numero di processori utilizzati:

$$E_p = \frac{S_p}{p} \quad o \quad E_p = \frac{T_s}{p * T_p}$$

Idealmente,  $E_p=1$ . L'efficienza ideale sarebbe:  $E_p^{ideale} = \frac{S_p^{ideale}}{p} = 1$

#### Esempio speed-up e efficienza

Di seguito è riportata una tabella che mostra i diversi valori di speed-up e efficienza sul calcolo della somma in parallelo.

p	Sp	Ep
2	1,88	0,94
4	3,00	0,75
8	3,75	0,47

Se si rapporta lo speed-up al numero di processori, notiamo che il maggiore sfruttamento dei processori è per  $p = 2$  perché è “il più vicino” allo speed-up ideale.

Quindi da questo deduciamo che l'utilizzo di un maggior numero di processori NON è sempre una garanzia di sviluppo di algoritmi paralleli “efficaci”.

### 1.2.3 Granularità

*Parallelismo a grana fine:* il programma è suddiviso in un gran numero di piccoli compiti, assegnati ai vari processori

- il parallelismo a grana fine facilita il bilanciamento del carico;
- Il numero di processori richiesto per eseguire il programma è alto: questo aumenta l'overhead comunicazione e sincronizzazione.

*Parallelismo a grana grossa:* il programma è suddiviso in compiti di grandi dimensioni, assegnati ai vari processori.

- ciò potrebbe causare squilibri nella distribuzione del carico tra i processori, o addirittura far sì che una significativa parte del calcolo venga svolta in sequenziale;
- il vantaggio di questo tipo di parallelismo è basso numero di comunicazioni e fasi di sincronizzazione.

**Overhead:** se la quantità di computazione parallela è consistente, l'overhead è la barriera più importante che impedisce di ottenere valori ottimali di speedup.

---

<sup>1</sup>Nell'algoritmo della somma ad ogni passo di computazione il numero totale di processori attivi si dimezza

Overhead di un algoritmo parallelo:

- costo dello starting di un processo o thread;
- costo della comunicazione di dati condivisi;
- costo della sincronizzazione;
- computazione extra (ridondante).

Ciascuna di queste può essere dell'ordine di millisecondi (= milioni di flops) su alcuni sistemi. Quindi, in conclusione, un algoritmo deve avere unità di lavoro sufficientemente grandi per una veloce esecuzione parallela, cioè elevata granularità, ma non così grandi che non ci sia abbastanza lavoro da svolgere in parallelo.

#### 1.2.4 Accessi in memoria

- Le memorie più grandi sono lente, quelle veloci sono piccole;
- Le gerarchie di memoria sono grandi e veloci mediamente;
- Processori paralleli, nel complesso, hanno memorie grandi e veloci, gli accessi lenti a dati remoti sono chiamate ‘comunicazioni’;
- Un algoritmo dovrebbe eseguire la maggior parte delle istruzioni su dati locali.

#### 1.2.5 Carico non bilanciato

Un cattivo bilanciamento del carico si verifica quando ci sono alcuni processori che non eseguono operazioni a causa di insufficiente parallelismo (in quella fase) oppure compiti di dimensione diversa, come ad esempio per problemi che hanno una struttura non omogenea.

#### 1.2.6 Legge di Amdahl

Il miglioramento nelle performances che può essere realizzato mediante il parallelismo è limitato dalla frazione di calcolo sequenziale inevitabilmente presente. Sia  $\alpha$  = frazione seriale dell'algoritmo che non può essere eseguita in parallelo. Ad esempio: inizializzazione dei cicli, lettura/scrittura da unità di memoria, overhead della chiamata a funzioni. Il tempo di esecuzione parallelo è

$$T_p = \alpha T_s + (1 - \alpha) \frac{T_s}{p} \quad (1.3)$$

La legge di Amdahl dà una limitazione allo speedup in termini di  $\alpha$

$$\begin{aligned} T_p &= \alpha T_s + (1 - \alpha) \frac{T_s}{p} \text{ allora } S_p = \frac{T_s}{\alpha T_s + \frac{(1-\alpha)T_s}{p}} \\ &= \frac{1}{\alpha + \frac{(1-\alpha)}{p}} \leq \frac{1}{\alpha} \end{aligned}$$

Per esempio se  $\alpha = 10\%$ , allora il massimo speed-up è 10, anche se usiamo un numero enorme di processori.

Se la dimensione n del problema è fissata, al crescere del numero di processori p:

$$S_p = \frac{1}{\alpha + \frac{(1-\alpha)}{p}} \leq \frac{1}{\alpha} \text{ per } p : \rightarrow \infty$$

Quindi se la dimensione del problema resta invariata, al crescere del numero di processori lo speed-up resta illimitato.

Per migliorare la legge di Amdahl dobbiamo aggiungere l'overhead quindi diventerà:

$$S_p = \frac{T_s}{\alpha T_s + \frac{(1-\alpha)T_s}{p} + T_{overhead}} \rightarrow \frac{1}{\alpha + \frac{T_{overhead}}{T_s}} \text{ per } p : \rightarrow \infty$$

Tuttavia, se  $p$  è fissato, al crescere della dimensione  $n$  del problema, la parte sequenziale  $\alpha \rightarrow 0$ , da cui:

$$S_p = \frac{1}{\alpha + \frac{(1-\alpha)}{p}} \rightarrow p \text{ per } n : \rightarrow \infty$$

Aumentando la dimensione  $n$  del problema si possono ottenere speed up ottimali MA non è possibile aumentare in maniera indefinita  $n$ : le risorse (hardware) sono limitate!

**Seconda legge di Amdahl** Aumentando il numero  $p$  di processori e mantenendo fissata la dimensione  $n$  del problema si riesce ad utilizzare in maniera efficiente l'ambiente di calcolo parallelo, se  $p \leq p_m$ .

Aumentando la dimensione  $n$  del problema e mantenendo fisso il numero di  $p$  processori le prestazioni dell'algoritmo parallelo non degradano se  $n \leq n_{max}$  superato questo valore potrei avere problemi con la memoria hardware!

### 1.3 Cenni finali

- **Complessità di Tempo  $T(n)$** : numero di operazione eseguite dall'algoritmo. È importante notare come il numero complessivo di operazioni determina anche il numero dei passi temporali, ovvero del tempo di esecuzione. In un ambiente parallelo, il numero delle operazioni non è legato al numero di passi temporali (posso fare più operazioni in un solo passo temporale).
- Indichiamo con  $T(P)$  il numero di passi temporali su  $P$  processori. Se moltiplico  $T(n)$  o  $T(p)$  con il tempo per singola operazione  $T_{calc}$  ottengo il tempo di esecuzione.
- **Complessità di Spazio  $S(n)$** : numero di variabili utilizzate dall'algoritmo





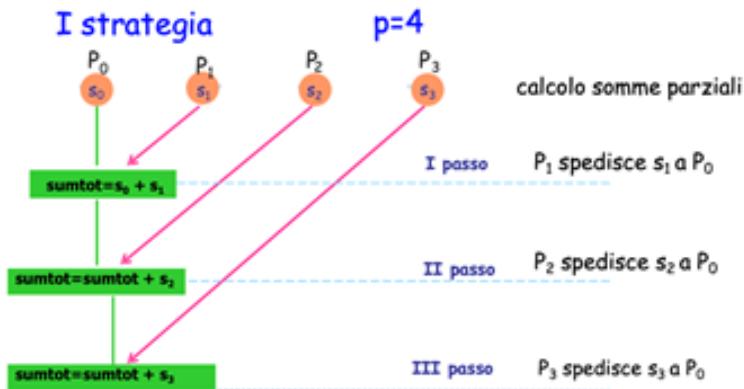
## 2. Somma di N numeri in parallelo

Problema: Vogliamo calcolare la somma di  $N$  numeri ( $a_0 + a_1 + \dots + a_{N-1}$ ). Sappiamo che su di un calcolatore mono-processore la somma è calcolata eseguendo le  $n-1$  addizioni una per volta secondo un ordine prestabilito.

### 2.1 Somma prima strategia di parallelizzazione

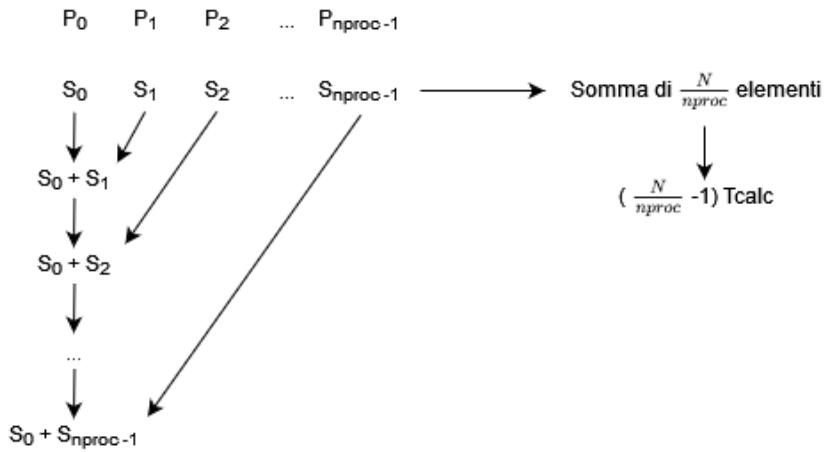
Abbiamo che ogni processore inizialmente calcola la propria somma parziale e ad ogni passo ciascun processore invia tale valore ad un unico processore prestabilito.

Tale processore contiene la somma totale.



### 2.1.1 Lavagna

Abbiamo  $N \rightarrow$  numeri e  $N_{proc} \rightarrow$  processori



In totale abbiamo  $n_{proc} - 1$  passi, questo a costo  $(n_{proc} - 1)(1 T_{calc} + 1 T_{com})$ .

Quindi possiamo dire che:

$$T_P^I = \left(\frac{N}{N_{proc}} - 1\right) T_{calc} + (n_{proc} - 1)(T_{calc} + T_{com})$$

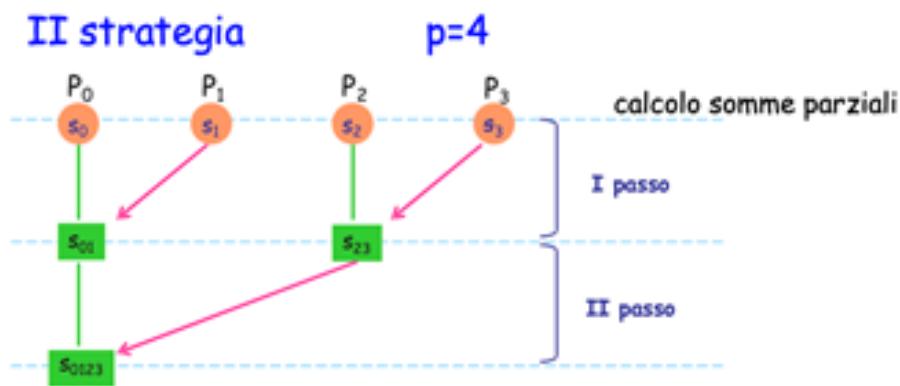
$$T_S = N - 1 T_{calc}$$

$$S_P = \frac{T_S}{T_P} = \frac{N-1 T_{calc}}{\left(\frac{N}{N_{proc}} - 1\right) T_{calc} + (n_{proc} - 1)(T_{calc} + T_{com})}$$

## 2.2 Somma seconda strategia di parallelizzazione

Abbiamo che ogni processore inizialmente calcola la propria somma parziale e ad ogni passo coppie distinte di processori comunicano contemporaneamente: in ogni coppia, un processore invia all'altro la propria somma parziale, che provvede all'aggiornamento della somma.

Il risultato è in un unico processore prestabilito.



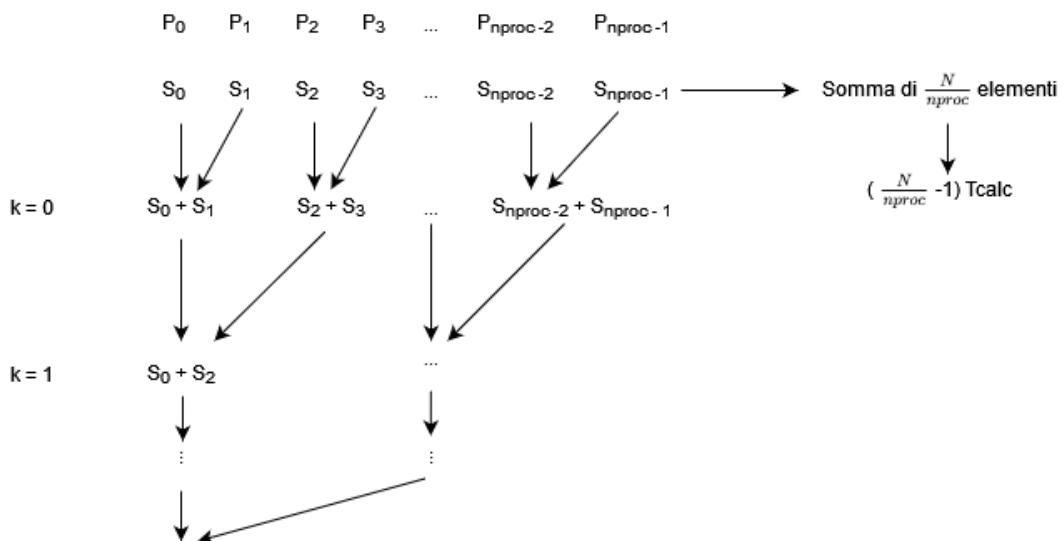
### 2.2.1 Distribuzione dei dati

Per la Distribuzione dei dati abbiamo 2 casi:

1. N multiplo di nproc,  $nloc = \frac{N}{nproc}$  con  $nloc = nlocal$ 
  - $P_0 : nloc = nloc$  dati  $a_0, a_1, \dots, a_{nloc-1}$
  - $P_1 : nloc = nloc$  dati  $a_0, a_1, \dots, a_{2nloc-1}$
  - ...
  - $P_{nproc-1} : nloc = nloc$  dati  $a_{N-nloc}, a_1, \dots, a_{N-1}$
2. N non è multiplo di nproc , $nloc = \frac{N}{nproc}$  con  $nloc = nlocal$  e  $r = resto(N, nproc)$ 
  - $P_0 : nloc = nlocgen + 1$  dati
  - $P_1 : nloc = nlocgen + 1$  dati
  - ...
  - $P_{r-1} : nloc = nlocgen + 1$  dati
  - $P_r : nloc = nlocgen + 1$  dati

### 2.2.2 Lavagna

Abbiamo  $N$  e  $Nproc = 2^P$  Ad ogni passo abbiamo che  $DIST = 2^k$  e :



- Se  $RESTO(menum, 2^{k+1}) = 0$  ricevo da menu + DIST e sommaloc = sommaloc + sommaricevuta
- Altrimenti se  $RESTO(menum, 2^{k+1}) = 2^k$  invia a  $P_{menu} - DIST$

Solo  $P_0$  ha il risultato finale.

Se vedo nproc come  $2^k$  e visto che ad ogni passo dimezzo il numero di processori usati ho  $\log_2 nproc$  passi.

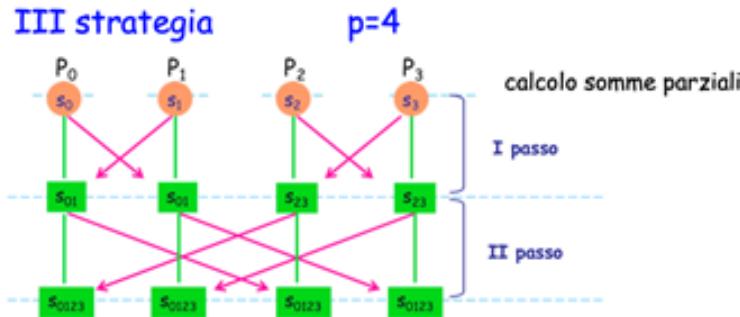
$$T_P^{II} = \left( \frac{N}{N_{proc}} - 1 \right) T_{calc} + \log_2 nproc (T_{calc} + T_{com})$$

$$S_P = \frac{T_S}{T_P} = \frac{\frac{N}{N_{proc}} - 1}{\left( \frac{N}{N_{proc}} - 1 \right) T_{calc} + \log_2 nproc (T_{calc} + T_{com})}$$

## 2.3 Somma terza strategia di parallelizzazione

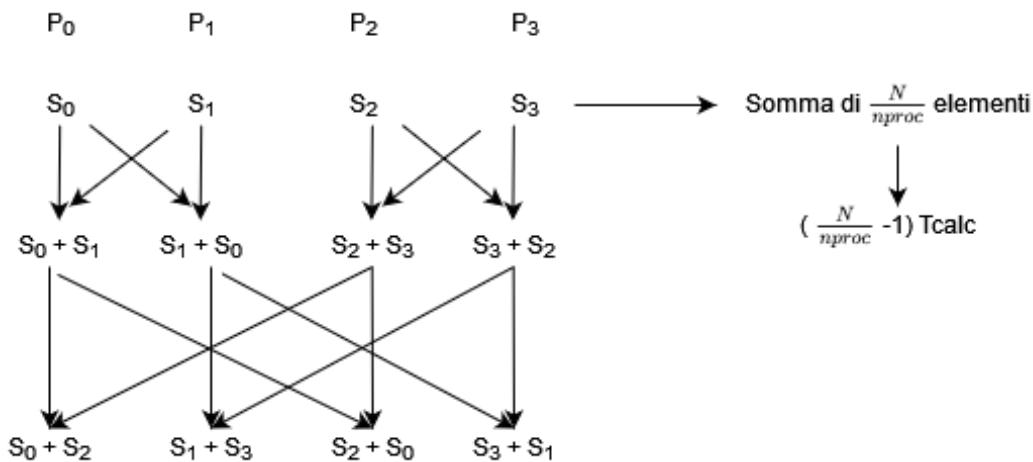
Abbiamo che ogni processore inizialmente calcola la propria somma parziale e ad ogni passo coppie distinte di processori comunicano contemporaneamente, in ogni coppia i processori si scambiano le proprie somme parziali.

Il risultato è in tutti i processori.



### 2.3.1 Lavagna

Esempio con  $N = 16$  e  $nproc = 4$



Tutti i processi hanno il risultato finale. Mi calcolo i k passi da fare con  $\log_2 nproc$ .

Ad ogni passo ho  $\log_2 nproc$  processi che lavorano.

Se RESTO(menum,  $2^{k+1}$ ) <  $2^k$  invio e ricevo da  $P_{menu} + DIST$

Se RESTO(menum,  $2^{k+1}$ )  $\geq 2^k$  invio e ricevo da  $P_{menu} - DIST$

$T_P^{III} = (\frac{N}{N_{proc}} - 1) T_{calc} + \log_2 nproc (T_{calc} + T_{com})$  è uguale alla seconda strategia

## 2.4 Legge di amdhald dimostrazione

La legge di amdhald da una limitazione allo speed-up in termini di  $\alpha$ , dove  $\alpha$  è la parte non parallelizzabile.

$$T_P = \begin{cases} \alpha T_S & \text{parte non parallelizzabile} \\ (1 - \alpha) T_S & \text{parte parallelizzabile} \end{cases} \quad . \quad (2.1)$$

Quindi  $T_P = \alpha T_S + \frac{(1-\alpha)T_S}{nproc}$

$$S_P = \frac{T_S}{T_P} = \frac{1}{\alpha T_S + \frac{(1-\alpha)T_S}{p}} = \frac{1}{\alpha + \frac{1-\alpha}{p}} \leq \frac{1}{\alpha}$$

Notiamo che:  $\frac{1}{\alpha T_S + \frac{(1-\alpha)T_S}{p}} = \begin{cases} p > \infty \text{ allora } \frac{1}{\alpha + \frac{1-\alpha}{p}} = 0 = \frac{1}{\alpha} \\ n > \infty \text{ allora } \frac{1}{\alpha + \frac{1-\alpha}{p}} = p \end{cases} \quad (1) \quad (2.2)$

- (1) = parte non parallelizzabile  $\alpha$
- $n > \infty$  inciderà pochissimo
- $(1 - \alpha)$  è più grande

Legge di amdhald generalizzata:  $T_S = \alpha_1 T_S + \alpha_2 T_S + \dots + \alpha_p T_S$

Ogni  $\alpha_n T_S$  con  $n = 1, 2, \dots, p$  rappresenta i processori utilizzati.



```
shivam@shivam-VirtualBox:~$ mpicc sum_array_mpi.c -o obj
shivam@shivam-VirtualBox:~$ mpirun -np 4 ./obj
Enter number of elements: 20
sum calculated by root process: 15
Sum calculated by process 3: 90
Partial sum returned from process 3: 90
Sum calculated by process 2: 65
Sum calculated by process 1: 40
Partial sum returned from process 1: 40
Partial sum returned from process 2: 65
Sum of array is : 210
shivam@shivam-VirtualBox:~$
```

## 3. Prodotto matrice-vettore

Progettazione di un algoritmo parallelo per architettura MIMD a memoria distribuita per il calcolo del prodotto di una matrice (densa) A per un vettore x:  $y = Ax, A \in R^{m \times n}, x \in R^n, y \in R^m$ .

### 3.1 Algoritmo seriale(sequenziale)

ogni elemento di ogni riga si moltiplica per ogni elemento del vettore, sommandolo al successivo per riga.

---

```
for(int i = 0; i < n; i++){
    for(int j = 0; i < m; j++){
        y[i] = a[i][j] + y[i];
    }
}
```

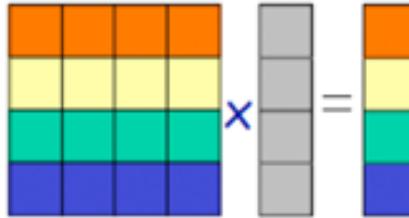
---

Costo: 1 moltiplicazione + 1 addizione (= 2tcalc) per ogni  $i=1, \dots, m$  e per ogni  $j=1, \dots, n$ . L'algoritmo sequenziale richiede  $m \cdot n$  moltiplicazione e  $m \cdot n$  addizioni  $T_S = 2mn t_{\text{calc}}$ .

### 3.2 Prima strategia suddivisione in blocchi di righe

Suddividiamo la matrice A in blocchi di righe. ogni processo ottiene una o più righe della matrice e l'intero vettore, effettua la moltiplicazione e la somma di ogni elemento della riga per ogni elemento del vettore e restituisce una o più celle di quello che sarà il vettore finale.

$$\begin{aligned}
 y_1 &= a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 \\
 y_2 &= a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 \\
 y_3 &= a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 \\
 y_4 &= a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4
 \end{aligned}$$



La matrice A viene distribuita in blocchi di righe fra i processori. Il vettore x è fornito interamente a tutti i processori. Se la decomposizione dei dati avviene mediante un partizionamento in blocchi di righe della matrice allora il vettore prodotto finale (il vettore y) viene calcolato in parallelo, in blocchi distribuiti fra i processori.

### 3.2.1 Funzionamento algoritmo

1. Inizializzazione/acquisizione dei dati:
  - Il processo root (quello con identificativo me=0) inizializza (oppure prende in input) la matrice A ed il vettore x;
  - Il processo root calcola la dimensione locale local\_m=m/nproc.
2. Distribuzione dei dati, il processo root distribuisce a tutti i processi:
  - m, n e local\_m broadcast;
  - la sottomatrice local\_A della matrice A scatter per righe), se stesso incluso;
  - l'intero vettore x broadcast.
3. Calcolo dei prodotti parziali:
  - Ciascun processo calcola il prodotto parziale local\_y = local\_A x.
4. Combinazione dei risultati:
  - Il processo root raccoglie gli elementi local\_y calcolati da ogni processo nel vettore risultato y gather.
5. Stampa del risultato finale: il solo processo root stampa il vettore y.

### 3.2.2 Valutazione dell'algoritmo

Sia p il numero di processori  $p \leq m$ , ogni processo esegue un prodotto mat-vet tra una matrice  $(m/p)$  e un vettore di lunghezza n:

$$\begin{aligned}
 T_P &= \frac{2mn}{nproc} T_{calc} \\
 S_P &= \frac{T_S}{T_P} = \frac{2mnT_{calc}}{\left(\frac{2mn}{nproc}\right)T_{calc}} = p \\
 E_P &= \frac{S_P}{p} = 1
 \end{aligned}$$

**NOTA:** le fasi di comunicazione riguardano unicamente la distribuzione iniziale dei dati e la raccolta finale dei risultati e quindi non sono state considerate nella valutazione dell'algoritmo.

### 3.3 Seconda strategia distribuzione per blocchi di colonne

Ogni processo ottiene una o più colonne della matrice e un o più celle del vettore, moltiplica ogni elemento della colonna per il rispettivo elemento del vettore, creando un vettore di moltiplicazioni/somme parziali lungo quanto la lunghezza della colonna, che poi verrà sommato agli altri vettori parziali per ottenere quello risultante finale.

$$y_1 = a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4$$

$$y_2 = a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4$$

$$y_3 = a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4$$

$$y_4 = a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4$$

$A_j$  = colonna j-sima della matrice A

$$y = A_1x_1 + A_2x_2 + A_3x_3 + A_4x_4 = r_0 + r_1 + r_2 + r_3$$

#### 3.3.1 Funzionamento algoritmo

1. Inizializzazione/acquisizione dei dati:
  - Il processo root (con identificativo 0) inizializza (o legge da file) la matrice A ed il vettore v;
  - Il processo root calcola la dimensione locale local\_n=n/nproc.
2. Distribuzione dei dati il processo root distribuisce a tutti i processi:
  - m, n e local\_n (broadcast);
  - la sottomatrice local\_A della matrice A (scatter per colonne), se stesso incluso;
  - il sottovettore local\_x del vettore x (scatter), se stesso incluso.
3. Calcolo dei prodotti parziali:
  - Ciascun processo calcola un vettore local\_y=local\_A\*local\_x .
4. Combinazione dei risultati:
  - Mediante un'operazione di riduzione\*, vengono sommati in parallelo tutti i vettori local\_y, ottenendo il vettore risultato y, e salvati dal processo root (o da tutti i processi).
5. Stampa del risultato finale: il solo processo root stampa il vettore y.

#### 3.3.2 Lavagna

Supponiamo nproc = 2.  $A(mx n) = \begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix}$   $x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$   $w = [w_0 \quad w_1 \quad w_2 \quad w_3]$  la dimensione di w = m.

$P_0$  avrà:  $\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \\ A_{20} & A_{21} \\ A_{30} & A_{31} \end{bmatrix}$  size:  $m \times \frac{n}{nproc}$  e  $\begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$  size:  $\frac{n}{nproc}$

$P_1$  avrà:  $\begin{bmatrix} A_{02} & A_{03} \\ A_{12} & A_{13} \\ A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix}$   $\begin{bmatrix} x_2 \\ x_3 \end{bmatrix}$

Ogni processo fa una moltiplicazione matrice-vettore tra una matrice  $m \times \frac{n}{nproc}$  e un vettore  $\frac{n}{nproc}$  producendo un vettore w di dimensione m, questi vettori w saranno poi sommati con una strategia di somma o una reduce. Prima della somma abbiamo:

- $T_P = 2m \frac{n}{nproc} T_{calc}$

- Se uso la prima strategia per la somma:

$$T_P^I = 2m \frac{n}{nproc} T_{calc} + (nproc - 1)m(T_{calc} + T_{com}), m$$
 indica una somma per ogni riga

- Se usiamo la seconda o terza strategia:

$$\begin{aligned}
 T_P^{II} &= 2m \frac{n}{nproc} T_{calc} + (\log_2 nproc)m(T_{calc} + T_{com}) \\
 \cdot S_P &= \frac{T_S}{T_P} = \frac{2mnT_{calc}}{\frac{2mn}{nproc} + (\log_2 nproc)m(T_{calc} + T_{com})} = \\
 &= \frac{1}{\frac{1}{nproc} + \frac{(\log_2 nproc)}{2n}} \quad \text{abbiamo due casi possibili:} \\
 \left\{ \begin{array}{l} n \text{ fissato } \lim_{nproc \rightarrow \infty} \frac{1}{\frac{1}{nproc} + \frac{\infty}{2n} = \frac{1}{\infty}} = 0 \\ nproc \text{ fissato } \lim_{n \rightarrow \infty} \frac{1}{\frac{1}{nproc} + \frac{\log_2 nproc}{2m} = \frac{1}{\frac{1}{nproc}}} = nproc \end{array} \right. \quad (3.1)
 \end{aligned}$$

nproc = speed-up ideale

### 3.4 Topologie di processori

La disposizione più naturale per i processori è una griglia bidimensionale di  $p \times q$  processori:

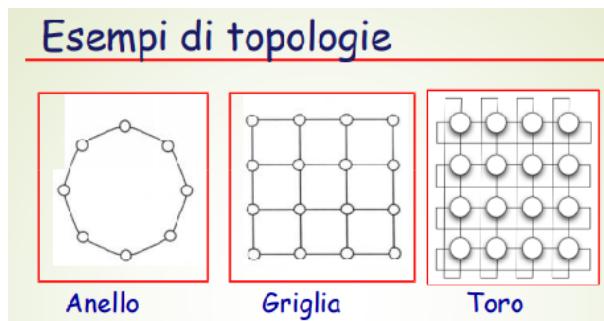
$$\begin{bmatrix} P_{00} & P_{01} & \dots & P_{0,q-1} \\ P_{10} & P_{11} & \dots & P_{1,q-1} \\ \dots & \dots & \dots & \dots \\ P_{p-1,0} & P_{p-1,1} & \dots & P_{p-1,q-1} \end{bmatrix}. \text{ Alcuni termini importanti:}$$

- Contesto (communicator): insieme di processi che possono comunicare reciprocamente attraverso lo scambio di messaggi;
- Una topologia è la geometria “virtuale” in cui si immaginano disposti i processori. La topologia “virtuale” in cui sono disposti i processori può non avere alcun nesso con la disposizione “reale” degli stessi! Infatti è struttura imposta sui processi che fanno parte di un contesto al fine di indirizzare specifici schemi di comunicazione.

#### 3.4.1 Topologie

La topologia di comunicazione identifica gli schemi (principali) di comunicazione che avvengono in un codice parallelo. Le topologie virtuali sono uno strumento utile per: risparmiare tempo, evitare errori, strutturare il codice e sono utili quando gli schemi di comunicazione seguono una o più strutture precise.

- Topologia lineare: Di default, MPI assegna a ciascun processo in un contesto un identificativo da 0 a  $n-1$ : topologia lineare ( $P_0 -> P_1 -> \dots -> P_{n-1}$ ). Con adeguate funzioni, MPI supporta anche topologie cartesiane e a grafo (ridefinizione del contesto).
- Topologia cartesiana: Ogni processo è identificato da coordinate cartesiane ed è connesso ai vicini da una griglia virtuale. Ai bordi ci può essere o no periodicità e i processi sono identificati dalle loro coordinate.
- altre topologie:



#### 3.4.2 Parallelizzazione del problema: distribuzione a griglia cartesiana

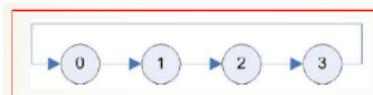
Si può dividere il dominio computazionale a “blocchi”. Questo tipo di suddivisione del dominio computazionale è preferibile perché:

- È compatibile con la geometria del problema;
- Permette una maggiore granularità;
- Riduce la comunicazione.

Però è più complessa da trattare (inizialmente).

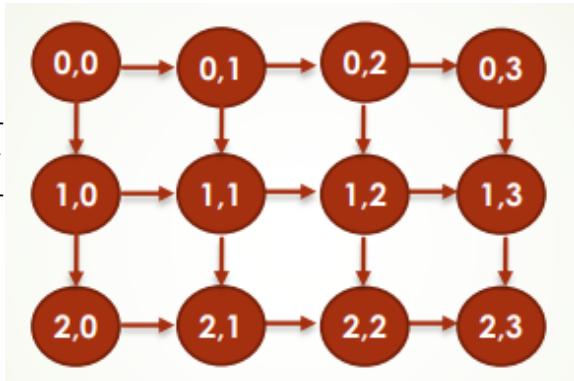
### Circular shift

topologia cartesiana 1D (conviene quella ad anello).



### Topologia cartesiana 2D

Ad ogni processo è associata una coppia di indici: coordinate nello spazio 2D. Funzione per la creazione di una griglia di processori (periodica e non): MPI\_Cart\_create. Definizione di tipo Communicator: MPI\_Comm comm\_grid.



```
/* Scopo: definizione di una topologia a griglia bidimensionale nproc=row*col
 */
int main(int argc, char **argv) {
    int menum,nproc,menum_grid,row,col;
    int dim,*ndim,reorder,*period;
    int coordinate[2];

    MPI_Comm comm_grid; /* definizione di tipo communicator */
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&menum);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc); /* Numero di righe della griglia di
        processo */

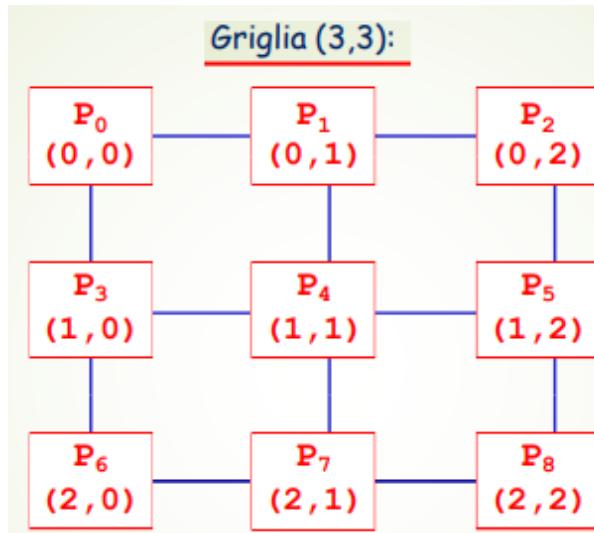
    if (menum == 0) {
        printf("Numero di righe della griglia ");
        scanf("%d",&row);
        /* Spedizione di row da parte di 0 a tutti i processori */
        MPI_Bcast(&row,1,MPI_INT,0,MPI_COMM_WORLD);
        dim = 2; /* Numero di dimensioni della griglia */
        col = nproc/row; /* Numero di colonne della griglia */
        /* vettore contenente le lunghezze di ciascuna dimensione*/
        ndim = (int*)calloc(dim,sizeof(int));
        ndim[0] = row;
        ndim[1] = col;
        /* vettore contenente la periodicita' delle dimensioni */
        period = (int*)calloc(dim,sizeof(int));
        period[0] = period[1] = 0;
        reorder = 0;
    }
}
```

```

/* Definizione della griglia bidimensionale */
MPI_Cart_create(MPI_COMM_WORLD, dim, ndim, period, reorder, &comm_grid);
MPI_Comm_rank(comm_grid, &menu_grid); /* id nella griglia */
/* Definizione delle coordinate di ciascun processo nella griglia bidimensionale
   */ MPI_Cart_coords(comm_grid, menu_grid, dim, coordinate);
/* Stampa delle coordinate */
printf("Processore %d coordinate nella griglia (%d,%d) \n", menu,*coordinate,
      *(coordinate+1)); MPI_Finalize();
return 0; }

```

Nel programma: MPI\_Cart\_Create(MPI\_COMM\_WORLD,dim,ndim,period,reorder,&comm\_grid); Ogni processo dell'ambiente MPI\_COMM\_WORLD definisce la griglia denominata comm\_grid , di dimensione 2 (dim) e non periodica lungo le due componenti (period[i]=0, i=0,1). Il numero di righe e di colonne della griglia sono memorizzati rispettivamente nella prima e nella seconda componente del vettore ndim. Gli identificativi dei processi non possono essere riordinati (reorder=0). Sintassi:



MPI\_Cart\_Create(MPI\_COMM Comm\_old, int dim, int \*ndim, int \*period, int reorder, MPI\_COMM New Comm);

- comm\_old: contesto di input
- dim: numero di dimensioni della griglia
- \*ndim: vettore di dimensione dim. La i-esima dimensione ha lunghezza ndim[i].
- \*period: vettore di dimensione dim. Se period[i]=1, la i-esima dimensione della griglia è periodica; non lo è se period[i]=0.
- reorder: permesso di riordinare gli identificativi (1=sì; 0=no)
- \*new\_comm: contesto di output associato alla griglia

**Note su MPI\_Cart:** Se reorder=1, MPI potrebbe riassegnare gli id dei processi del nuovo contesto (per potenziale guadagno nelle prestazioni, grazie a vicinanza fisica dei processi). Se reorder =0, l'id del processo nel nuovo contesto è identico a quello che aveva nel vecchio contesto

. Se la dimensione complessiva della griglia: è più piccola dei processi disponibili, i processi rimasti fuori avranno in output MPI\_COMM\_NULL, se è più grande, la chiamata a MPI\_Cart produrrà un errore.

Nel programma: MPI\_Cart\_coords (comm\_grid, menum, dim, coordinate);

Ogni processo menum calcola le proprie dim (2) coordinate (coordinate[i], i=0,1) nel contesto comm\_grid.

Sintassi: MPI\_Cart\_coords(MPI\_Comm comm\_grid, int menum\_grid, int dim, int \*coordinate);  
Operazione collettiva che restituisce a ciascun processo di comm\_grid con identificativo menum\_grid, le sue coordinate all'interno della griglia predefinita. coordinate è un vettore di dimensione dim, i cui elementi rappresentano le coordinate del processo all'interno della griglia, output.

Funzione che restituisce l'identificativo (rank) date le coordinate: MPI\_Cart\_rank(MPI\_Comm comm\_grid,int icoords, int rank); è un'operazione collettiva che restituisce a ciascun processo di comm\_grid, date le coordinate icoords, l'identificativo rank associato all'interno della griglia predefinita.

Abbiamo due tipi di periodicità:

- period[0] = 1 e period[1] = 0: periodicità sulle colonne;
- period[0] = 0 e period[1] = 1: periodicità sulle righe;

**Effetti della periodicità:** Se imponiamo la periodicità, ogni riferimento prima della prima o dopo l'ultima componente di ogni riga/colonna si riavvolgerà ciclicamente. Se non c'è periodicità, ogni riferimento all'infuori del range definito, produrrà un identificativo negativo (MPI\_PROC\_NULL, che è -1).

### 3.4.3 Sottogriglie

Spesso si vuole operare su una parte di una topologia, si divide la topologia in sottotopologie che formano griglie di dimensioni minori e ogni sottogriglia genera un nuovo contesto in cui eseguire operazioni collettive.

Funzione per la creazione di una sottogriglia di processori: MPI\_Cart\_sub(MPI\_Comm comm\_grid, int\* rdims, MPI\_Comm\* new\_griglia);

- comm\_grid: contesto della topologia;
- int\* rdims :vettore di dimensione dim (dimensione della topologia):
  - se rdims[i]=1: la i-ma dimensione varia;
  - se rdims[i]=0, la i-ma dimensione è fissata
- new\_griglia: nuovo contesto della sottogriglia

Il numero di sottotopologie è il prodotto del numero di processi relativi alle dimensioni eliminate. Esempio:

---

```
/* Crea una topologia 2D cartesiana */
MPI_Cart_create(MPI_COMM_WORLD, ndim, dims, period, reorder, &comm2D);
MPI_Comm_rank(comm2D, &id2D);
MPI_Cart_coords(comm2D, id2D, ndim, coords2D); /* crea le sottogriglie di righe*/
belongs[0] = 0;
belongs[1] = 1; /*dim. Variabile*/
MPI_Cart_sub(comm2D, belongs, &commrow); /* crea le sottogriglie di colonne*/
belongs[0] = 1; /* dim. Variabile*/
belongs[1] = 0;
MPI_Cart_sub(comm2D, belongs, &commcol);
```

---

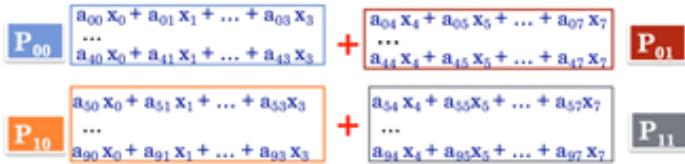
#### Note su MPI\_Cart\_sub:

- Le sottogriglie hanno dimensione pari alla dimensione della griglia di origine, lungo la

- direzione considerata;
- Ogni sottogriglia è un contesto, che eredita le proprietà della griglia cartesiana, in particolare è ancora una griglia cartesiana ( => ogni processo ha sia id che coordinate);
- MPI\_Cart\_sub è una routine collettiva. È richiamata da tutti i processi del contesto di origine;
- Restituisce il contesto a cui il processo chiamante appartiene.

### 3.5 Terza strategia distribuzione per schema a blocchi

Ogni processo ottiene una sottomatrice e un sotto vettore, ne calcola il prodotto, che poi viene sommato o concatenato a quello degli altri.



#### 3.5.1 Algoritmo

Le operazioni da effettuare sono le seguenti:

1. Inizializzazione/acquisizione dei dati:
  - Il processo root (con identificativo (0,0)) inizializza (o legge da file) la matrice A ed il vettore v;
2.  $P_{00}$  distribuisce il vettore x suddetto a tutti i processi della sua riga;
3. I processi della 1° riga che hanno il pezzo di x che gli serve e lo mandano i broadcast alla propria colonna;
4.  $P_{00}$  distribuisce i pezzi di matrice grandi  $n * \frac{m}{p}$  ai processi della 1° colonna della griglia;
5. i processi della 1° colonna distribuiscono pezzi di matrice grandi  $\frac{m}{p} * \frac{n}{q}$  alla propria riga;
6. ogni processo fa un mat-vet di  $\underbrace{\frac{m}{p} \frac{n}{q}}_A \underbrace{\frac{m}{p}}_x$ ;
7. ogni processo della 1° colonna somma i risultati parziali della propria riga;
8. Ogni processo della 1° colonna avrà un pezzo di w che verranno unita da  $P_{00}$  (gather).

#### 3.5.2 Lavagna

$$A(mxn) = \begin{bmatrix} a_{00} & \dots & a_{0,n-1} \\ \dots & \dots & \dots \\ a_{m-1,0} & \dots & a_{m-1,n-1} \end{bmatrix} x = \begin{bmatrix} x_0 \\ \dots \\ x_{n-1} \end{bmatrix} w = [w_0 \ \dots \ w_{m-1}]$$

A ha dimensione  $p \times q$  processi, x ha dimensione n e w ha dimensione m.

Divido la matrice a blocchi di righe e colonne, ogni processo effettuerà una moltiplicazione matrice-vettore tra una matrice  $\frac{m}{p} * \frac{n}{q}$  e un vettore  $\frac{n}{q}$  e successivamente ogni processo della prima colonna della griglia somma tutti i vettori parziali grandi  $\frac{m}{p}$  con una delle strategie per la somma in parallelo.

- $T_P^I = 2\frac{m}{p}\frac{n}{q}T_{calc} + (q-1)\frac{m}{p}(T_{cal} + T_{com})$
  - $T_P^{II} = 2\frac{m}{p}\frac{n}{q}T_{calc} + (\log_2 q)\frac{m}{p}(T_{cal} + T_{com})$
- $\frac{m}{p}$  perchè ogni processore in contemporanea somma su  $\frac{m}{p}$  righe.

Otteniamo così  $w_0$  parziali che saranno uniti da  $P_{00}$  con una gather.

- $S_{pq}^I = \frac{T_S}{T_{pq}^I} = \frac{2mn}{\frac{2mn}{pq} + (q-1)\frac{m}{p}} = \frac{1}{\frac{1}{pq} + \frac{(q-1)}{2np}}$
- $E_{pq}^I = \frac{S_{pq}^I}{pq} = \frac{1}{pq} * \frac{1}{\frac{1}{pq} + \frac{(q-1)}{2np}} = \frac{1}{1 + \frac{q(q-1)}{2n}}$   
se  $n- > \infty$  allora  $\frac{1}{1+0}=1$   
se  $q- > \infty$  allora  $\frac{1}{1+\infty}=0$
- $S_{pq}^{II} = \frac{2mn}{\frac{2mn}{pq} + (\log_2 q)\frac{m}{p}} = \frac{1}{\frac{1}{pq} + \frac{(\log_2 q)}{2np}}$
- $E_{pq}^{II} = \frac{1}{pq} * \frac{1}{\frac{1}{pq} + \frac{(\log_2 q)}{2np}} = \frac{1}{1 + \frac{q(\log_2 q)}{2n}}$   
se  $n- > \infty$  allora  $\frac{1}{1+0}=1$   
se  $q- > \infty$  allora  $\frac{1}{\infty}=0$



```
shivam@shivam-VirtualBox:~$ mpicc sum_array_mpi.c -o obj
shivam@shivam-VirtualBox:~$ mpirun -np 4 ./obj
Enter number of elements: 20
sum calculated by root process: 15
Sum calculated by process 3: 90
Partial sum returned from process 3: 90
Sum calculated by process 2: 65
Sum calculated by process 1: 40
Partial sum returned from process 1: 40
Partial sum returned from process 2: 65
Sum of array is : 210
shivam@shivam-VirtualBox:~$
```

## 4. GPGPU (General Purpose GPU)

### 4.1 GPU

La GPU(Graphics Processing Unit) è un microprocessore altamente parallelo (many-core) dotato di memoria privata ad alta banda ed è specializzata per le operazioni di rendering grafico 3D. La GPU è vista come un coprocessore matematico con una propria area di memoria (device memory).

#### 4.1.1 Architettura CPU vs GPU

Le GPU sono processori specializzati per problemi che possono essere classificati come intense data-parallel computations:

- lo stesso algoritmo è eseguito su molti elementi differenti in parallelo;
- controllo di flusso molto semplice (control unit ridotta);
- limitata località spaziale (parallelismo a granularità fine) e alta intensità aritmetica che nasconde le latenze di load/store (cache ridotta);
- i thread GPU non hanno costo di attivazione/disattivazione quindi sono leggeri;
- la GPU richiede migliaia di threads per la piena efficienza.

Le applicazioni con un'elevata logica di controllo del processo di calcolo vengono eseguite efficientemente in modo sequenziale dalle tradizionali CPU, ma con pessime prestazioni dall'architettura parallela delle GPU (es. gestione dei database, compressione dei dati, algoritmi ricorsivi).

#### 4.1.2 CUDA (Compute Unified Device Architecture)

Il cuore di cuda è basato su un set di istruzioni chiamato PTX (Parallel Thread Execution). La GPU è divisa in due parti:

- data-parallel: definisce una funzione kernel che viene eseguita identicamente da tutti i thread sul dispositivo.
- computational-intensive.

Un'applicazione CUDA combina parti sequenziali (a carico dell'host) e parti parallele, dette kernel (a carico del device).

Il parallelismo su cui si basa è SPMD e SIMD.

```

int main(int argc, char *argv[]) {
    int i;
    const int N = 1000;
    double u[N], v[N], z[N];

    initVector (u, N, 1.0);
    initVector (v, N, 2.0);
    initVector (z, N, 0.0);

    printVector (u, N);
    printVector (v, N);

    // z = u + v
    for (i=0; i<N; i++)
        z[i] = u[i] + v[i];

    printVector (z, N);

    return 0;
}

void gpuVectAdd( const double *u,
                 const double *v, double *z)
{
    // use GPU thread id as index
    i = getThreadIdx();
    z[i] = u[i] + v[i];
}

```

```

int main(int argc, char *argv[]) {
    ...
    // z = u + v
    {
        // run on GPU with N threads
        gpuVectAdd(u, v, z);
        // close threads
    }
    ...
}

```

Quando si esegue un kernel CUDA sul device GPU bisogna specificare il numero di thread che devono essere lanciati, i thread sono raggruppati in blocchi che sono identificati da un set di coordinate cartesiane 1D, 2D, 3D.

I blocchi di thread costituiscono gli elementi di una griglia, ogni blocco all'interno della griglia può essere identificato mediante un set di coordinate cartesiane bidimensionale.

In ogni CUDA kernel è possibile utilizzare le seguenti variabili per identificare le coordinate del thread corrente e del blocco:

- **threadIdx**: coordinate del thread all'interno del blocco;
- **blockIdx**: coordinate del blocco all'interno della griglia;
- **blockDim**: dimensione del blocco all'interno del thread;
- **gridDim**: dimensione della griglia all'interno del blocco.

La scelta di come calcolare l'indice varia in base alla dimensione della griglia.

A una dimensione abbiamo diverse possibilità:

- $\text{index} = \text{threadIdx.x} + \text{blockIdx.x} * \text{blockDim.x}$  (utilizzato dalla prof) ;
- $\text{index} = \text{numBlocks} * (\text{N} + \text{numThreads} - 1) / \text{numThreads.x}$ ;
- $\text{index} = (\text{blockIdx \% x} - 1) * \text{blockDim \% x} + \text{threadIdx \% x}$ .

A due dimensioni abbiamo:

---

```

i = blockIdx.x * blockDim.x + threadIdx.x;
j = blockIdx.y * blockDim.y + threadIdx.y;
index = j * gridDim.x * blockDim.x + i;

```

---

#### 4.1.3 Memorie di una GPU

Per ottenere prestazioni ottimali è fondamentale fare un buon uso delle memorie:

- **Global memory:** memoria ad accesso lento, condivisa da tutti i blocchi thread;
- **Shared memory:** memoria ad accesso veloce, condivisa da tutti i thread di un blocco;
- **Registri:** i dati all'interno dei registri sono visibili solo al thread che li ha scritti e vengono rimossi alla fine del thread.
- **Local memory:** simile ad un registro ma è più lento, vengono memorizzate variabili di dimensione maggiore (array);
- **Constant memory:** è una memoria molto efficiente quando più thread devono accedere allo stesso valore contemporaneamente;
- **Texture memory:** Visibile da ogni thread, si sola lettura

#### 4.1.4 Architetture della GPU

##### NVIDIA Fermi

È composta da 16 SM (Streaming processor) composto da:32/48 CUDA core ognuno con una sua ALU sia per interi sia per floating point completamente pipelined, ha registri di 32 bit, 16 unità load/store. Ha anche 4-6 GB di memoria con ECC, ha due cache, L1 configurabile per Sm e L2 comune a tutti gli SM. 2 controller indipendenti per il trasferimento dati da/verso host. Uno scheduler globale per distribuire i blocchi thread ad ogni scheduler degli SM.

##### NVIDIA Kepler

Compsta da 192 core CUDA, 4 warp scheduler, 32 unità load/store, i registri sono a 32 bit.

#### 4.1.5 Modello di esecuzione di CUDA

Al lancio di un kernel CUDA ogni blocco della griglia è assegnato ciclicamente a uno SM. Il numero massimo di blocchi in carico a ciascun SM dipende dalle caratteristiche hardware del multiprocessore (memoria shared e registri) e da quante risorse richiede il kernel da eseguire; eventuali blocchi non assegnati vengono allocati non appena un SM completa un blocco (non è possibile fare alcuna ipotesi sull'ordine di esecuzione dei blocchi! (nessuna sincronizzazione!))

I blocchi, una volta assegnati, non migrano, i thread in ciascun blocco sono raggruppati in gruppi di 32 thread di indice consecutivo (detti warp). Lo scheduler seleziona da uno dei blocchi a suo carico dei warp pronti per l'esecuzione e le istruzioni vengono dispacciate per warp, ogni CUDA core elabora uno dei 32 thread del warp.

#### 4.1.6 Proprietà di CUDA

##### Scabilità Trasparente

Un kernel CUDA scala su un qualsiasi numero di Streaming Multiprocessor e lo stesso codice può girare al meglio su architetture diverse (non essendo ottimizzato le prestazioni fanno cagare).

##### Gestione dei Registri

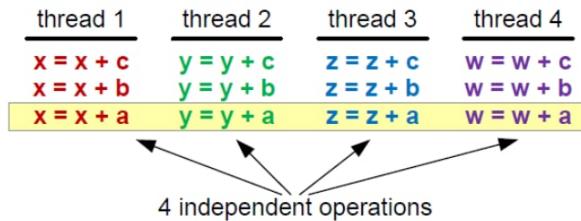
I registri sono partizionati dinamicamente tra tutti i blocchi assegnati ad un SM. I thread di un blocco accedono solo ai registri che gli sono stati assegnati. Lo zero-overload schedule è realizzato grazie al fatto che le informazioni di content switch dei warp rimangono inalterate nei registri del blocco.

#### 4.1.7 Latenza

Una latenza è il numero necessario di cicli affinché un'istruzione venga completata. Possiamo limitarla saturando la pipeline di calcolo e la bandwidth. Questo si ottiene fornendo allo scheduler un numero sufficiente di operazioni da svolgere. Due possibili paradigmi:

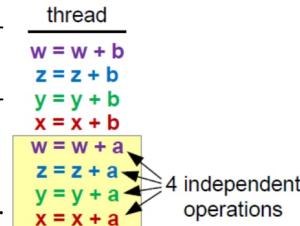
##### Thread-Level Parallelism (TLP)

Generalmente si consiglia di fornire molti thread per SM in modo da garantire il parallelismo minimo per coprire le latenze aritmetiche.



##### Instruction-Level Parallelism (ILP)

Abbiamo che ogni processore: Una seconda alternativa è sfruttare il parallelismo interno alle istruzioni dello stesso thread, fornendo allo scheduler più operazioni indipendenti da accodare nella pipeline. Lo scheduler non passa a un nuovo thread se ci sono altre istruzioni da poter istanziare.



#### 4.1.8 Compute Capability

La compute capability specifica il set di istruzioni e features che si vuole supportare per la generazione del codice PTX infatti il codice PTX, pur descrivendo una macchina virtuale, viene esteso e arricchito nel tempo con nuove istruzioni per il nuovo hardware. È identificata da dal codice “compute\_Xy”:

- (X): identifica l'architettura base del chip
- (y): identifica varianti con più o meno features

Infine specifica il set di istruzioni disponibili in compilazione del PTX code.

compute capability	feature support
compute_10	very basic features
compute_13	basic + double precision + atomics
compute_20	FERMI architecture (double precision)
compute_30	KEPLER K10 architecture (single precision)
compute_35	KEPLER K20, K20X, K40 architectures

Technical specifications	Compute capability (version)															
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0 (7.2?)
Maximum number of resident grids per device (concurrent kernel execution)	t.b.d.		16		4		32		32	16	128	32	16		128	
Maximum dimensionality of grid of thread blocks	2									3						
Maximum x-dimension of a grid of thread blocks	65535									2 <sup>31</sup> - 1						
Maximum y-, or z-dimension of a grid of thread blocks										65535						
Maximum dimensionality of thread block										3						
Maximum x- or y-dimension of a block	512									1024						
Maximum z-dimension of a block										64						
Maximum number of threads per block	512									1024						
Warp size										32						
Maximum number of resident blocks per multiprocessor	8					16				32				16		
Maximum number of resident warps per multiprocessor	24	32	48							64				32		
Maximum number of resident threads per multiprocessor	768	1024	1536							2048				1024		
Number of 32-bit registers per multiprocessor	8 K	16 K	32 K		64 K	128 K				64 K						
Maximum number of 32-bit registers per thread block	N/A	32 K	64 K	32 K		64 K		32 K	64 K		32 K	64 K		64 K		
Maximum number of 32-bit registers per thread	124		63						255							
Maximum amount of shared memory per multiprocessor	16 KB		48 KB		112 KB	64 KB	96 KB		64 KB	96 KB	64 KB	96 KB (of 128)		64 KB (of 96)		
Maximum amount of shared memory per thread block					48 KB							48/96 KB		64 KB		
Number of shared memory banks	16								32							
Amount of local memory per thread	16 KB								512 KB							
Constant memory size									64 KB							
Cache working set per multiprocessor for constant memory					8 KB				4 KB				8 KB			
Cache working set per multiprocessor for texture memory	6 – 8 KB		12 KB		12 – 48 KB		24 KB	48 KB	N/A	24 KB	48 KB	24 KB	32 – 128 KB	32 – 64 KB		
Maximum width for 1D texture reference bound to a CUDA array	8192								65536							
Maximum width for 1D texture reference bound to linear memory								2 <sup>27</sup>								
Maximum width and number of layers for a 1D layered texture reference	8192 × 512								16384 × 2048							
Maximum width and height for 2D texture reference bound to a CUDA array	65536 × 32768								65536 × 65535							
Maximum width and height for 2D texture reference bound to a linear memory								65000 <sup>2</sup>								
Maximum width and height for 2D texture reference bound to a CUDA array supporting texture gather	N/A								16384 <sup>2</sup>							
Maximum width, height, and number of layers for a 2D layered texture reference	8192 × 8192 × 512								16384 × 16384 × 2048							
Maximum width, height and depth for a 3D texture reference bound to linear memory or a CUDA array	2048 <sup>3</sup>								4096 <sup>3</sup>							
Maximum width and number of layers for a cubemap layered texture reference	N/A								16384 × 2046							
Maximum number of textures that can be bound to a kernel	128								256							
Maximum width for a 1D surface reference bound to a CUDA array									65536							
Maximum width and number of layers for a 1D layered surface reference									65536 × 2048							
Maximum width and height for a 2D surface reference bound to a CUDA array									65536 × 32768							
Maximum width, height, and number of layers for a 2D layered surface reference									65536 × 32768 × 2048							
Maximum width, height, and depth for a 3D surface reference bound to a CUDA array									65536 × 32768 × 2048							
Maximum width and number of layers for a cubemap layered surface reference									32768 × 2046							
Maximum number of surfaces that can be bound to a kernel	8								16							
Maximum number of instructions per kernel	2 million								512 million							

[35]

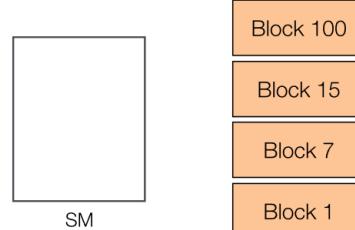
Architecture specifications	Compute capability (version)														
	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5	3.7	5.0	5.2	6.0	6.1, 6.2	7.0, 7.2	7.5
Number of ALU lanes for integer and single-precision floating-point arithmetic operations	8 <sup>[36]</sup>		32	48	192		128	64	128		64				
Number of special function units for single-precision floating-point transcendental functions	2		4	8		32		16	32		32				
Number of texture filtering units for every texture address unit or render output unit (ROP)	2		4	8	16		16		16		8 <sup>[37]</sup>				
Number of warp schedulers	1		2		4		4		2		4				
Max number of instructions issued at once by a single scheduler	1		2 <sup>[38]</sup>				2 <sup>[38]</sup>			1					
Number of tensor cores					N/A						8 <sup>[37]</sup>				
Size in KB of unified memory for data cache and shared memory per multi processor	t.b.d.										128	96 <sup>[39]</sup>			

#### 4.1.9 Come funziona uno streaming multi processor (SM)

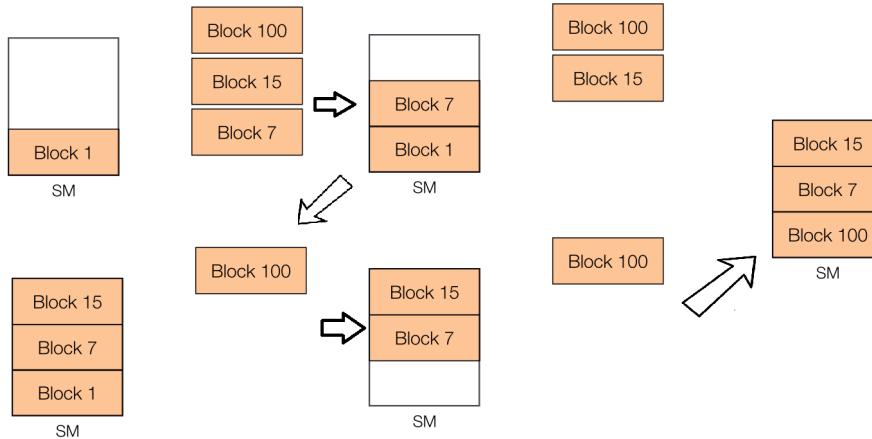
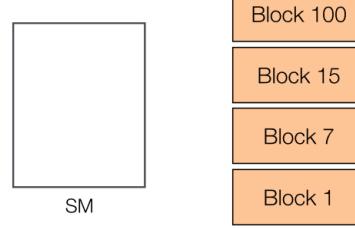
I thread CUDA sono raggruppati in blocchi di thread. I thread dello stesso blocco vengono eseguiti contemporaneamente nello stesso SM. Gli SM hanno memoria condivisa, quindi i thread all'interno di un blocco di thread possono comunicare.

La totalità dei thread di un blocco deve essere eseguita prima che ci sia spazio per schedulare un altro blocco di thread.

L'hardware pianifica i blocchi di thread sugli SM disponibili, non c'è nessuna garanzia sull'ordine di esecuzione e se un SM ha più risorse, l'hardware pianificherà più blocchi.



L'hardware pianifica i blocchi di thread sugli SM disponibili, non c'è nessuna garanzia sull'ordine di esecuzione e se un SM ha più risorse, l'hardware pianificherà più blocchi.



#### Warps

Il warp è l'unità fondamentale di esecuzione. Ogni warp può eseguire un'istruzione su 16 cuda core, 16 load/store units. Ci vogliono 2 cicli per completare un'istruzione o una load/store per tutto il warp. All'interno degli SM, i thread vengono lanciati in gruppi di 32 chiamati appunto warps. I warp condividono la parte di controllo (warp scheduler). In qualsiasi momento, viene eseguito solo un warp per SM. I thread in un warp eseguiranno la stessa istruzione. Sono usati gli Half warp per la computer capability 1.X .

Es: FERMI ARCHITECTURE: massimo numero di thread attivabili 1024 (thread massimi per blocco)\* 8 (blocchi massimi per SM)\* 32 (SM) =262144 Lo SM implementa un warp scheduling con zero-overhead. L'architettura Fermi usa due warp scheduling.

Per sfruttare al massimo il parallelismo intrinseco del SM, i thread dello stesso warp devono eseguire la stessa istruzione: se questa condizione non si verifica si parla di divergenza dei threads. Se i thread dello stesso warp stanno eseguendo istruzioni diverse l'unità di controllo non può gestire tutto il warp: deve seguire le successioni di istruzioni per ogni singolo thread (o per sottoinsiemi omogenei di threads) in modo seriale (perdita del parallelismo).

### Esempio di divergenza

---

```
int sum = 0;
if (threadIdx.x < 5 {
    sum = sum +10; *
} else {
    sum = sum + 3; #
}
```

---

In un warp avremo quindi che:

- Da 0 a 4 thread eseguiranno \*;
- da 5 a 31 eseguiranno #.

Questo porta ad avere una divergenza. Come possiamo risolvere questa divergenza?

### Risoluzione della divergenza

---

```
int WARP_SIZE = 32;
int sum = 0;
if (threadIdx.x/WARP_SIZE < 5 {
    sum = sum +10; *
} else {
    sum = sum + 3; #
}
```

---

Questo comporta che:

Warp	Thread	threadIdx.x/WARP_SIZE	Eseguono
1	0...31	0	*
2	32...63	1	*
...	...	...	...
z	256...282	8	#

Table 4.1:

## 4.2 Somma di due Vettori

Si vogliono sommare due vettori sfruttando come parallelismo la gpu.

### 4.2.1 Codice

Inanzitutto identifichiamo le parti data-parallel e i dati coinvolti da trasferire.

```
int main(int argc, char *argv[]) {
    int i;

    /*****
    const int N = 1000;
    double u[N], v[N], z[N];
    *****/
    initVector (u, N, 1.0);
    initVector (v, N, 2.0);
    initVector (z, N, 0.0);
    printVector (u, N);
    printVector (v, N);

    /*****
    // z = u + v
    for (i=0; i<N; i++)
        z[i] = u[i] + v[i];
    *****/
    printVector (z, N);
    return 0;
}
```

---

```
program vectoradd
integer :: i
integer, parameter :: N=1000
real(kind(0.0d0)), dimension(N):: u,
      v, z
call initVector (u, N, 1.0)
call initVector (v, N, 2.0)
call initVector (z, N, 0.0)
call printVector (u, N)
call printVector (v, N)
! z = u + v
do i = 1,N
z(i) = u(i) + v(i)
end do
call printVector (z, N)
end program
```

---

Poi implementiamo il Kernel CUDA e scriviamo l'algoritmo per il singolo elemento.

---

```
const int N = 1000;
double u[N], v[N], z[N];
// z = u + v
for (i=0; i<N; i++)
    z[i] = u[i] + v[i];
```

---

diventa:

---

```
void gpuVectAdd (const double
                  *u, const double *v, double
                  *z, int N) {
    // index is a unique
    // identifier of each GPU
    // thread
    int index = ... ;
    if (index < N)
        z[index] = u[index] +
                    v[index];
}
```

---

La

scelta delle dimensioni della griglia è dettata dal problema e dal mapping che si utilizza nel kernel CUDA tra l'indice degli elementi da elaborare e gli identificativi dei thread CUDA.

Nel caso di un problema lineare come l'elaborazione degli elementi di un array, possiamo scegliere:

- blocchi di thread monodimensionali (1D)
- griglia monodimensionale di blocchi (1D)

Ogni thread della griglia elabora in parallelo un solo elemento del vettore per determinare la griglia in modo da generare un numero di thread sufficiente a ricoprire tutta la superficie da elaborare.

Scelta la dimensione del blocco, la dimensione della griglia si può adattare al problema, alcuni thread potrebbero uscire dal dominio (nessun problema).

---

```
--global__ void gpuVectAdd (const double
    *u, const double *v, double *z, int
    N) {
    // index is a unique identifier of
    // each GPU thread
    int index = blockIdx.x * blockDim.x
    + threadIdx.x ;
    if (index < N)
        z[index] = u[index] + v[index];
}
```

---

Modificare il codice in modo da girare sulla GPU e allocare la memoria sul device.

---

```
double *u_dev, *v_dev, *z_dev;
cudaMalloc((void **)&u_dev, N * sizeof(double));
cudaMalloc((void **)&v_dev, N * sizeof(double));
cudaMalloc((void **)&z_dev, N * sizeof(double));
```

---

trasferire i dati necessari dall'host al device

---

```
cudaMemcpy(u_dev, u, sizeof(u), cudaMemcpyHostToDevice);
cudaMemcpy(v_dev, v, sizeof(v), cudaMemcpyHostToDevice);
```

---

Porting finale del codice somma di due vettori in cuda c

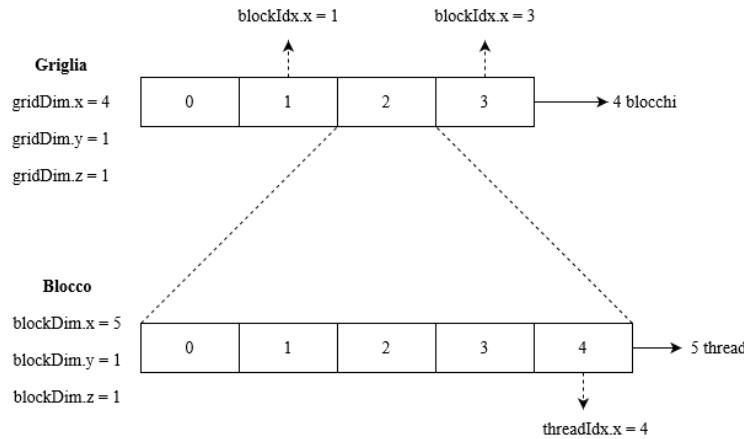
---

```
double *u_dev, *v_dev, *z_dev;
cudaMalloc((void **)&u_dev, N * sizeof(double));
cudaMalloc((void **)&v_dev, N * sizeof(double));
cudaMalloc((void **)&z_dev, N * sizeof(double));
cudaMemcpy(u_dev, u, sizeof(u), cudaMemcpyHostToDevice);
cudaMemcpy(v_dev, v, sizeof(v), cudaMemcpyHostToDevice);
dim3 numThreads( 256 ); // 128-512 are good choices
dim3 numBlocks( (N + numThreads.x - 1) / numThreads.x );
gpuVectAdd<<<numBlocks, numThreads>>>( u_dev, v_dev, z_dev, N );
cudaMemcpy(z, z_dev, N * sizeof(double), cudaMemcpyDeviceToHost);
```

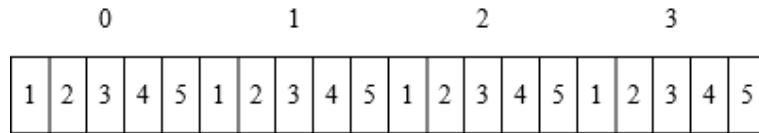
---

### 4.2.2 Lavagna

Abbiamo una griglia una griglia 1D e blocchi di 1D:



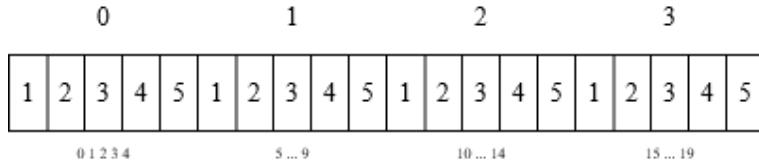
se rappresentiamo la griglia con tutti i thread ci troviamo davanti a:



Abbiamo quindi 20 thread attivabili, ora ritornando alla somma di due vettori, abbiamo che  $z = u + v$ , con  $u, v, z$  vettori di dimensione N.

Abbiamo 3 possibili casi di distribuzione dei dati:

1. Abbiamo  $N = 20$



$$\text{index} = \text{threadIdx.x} + \text{blockIdx.x} * \text{blockDim.x}$$

2.  $N > 20$ : abbiamo delle componenti che rimangono fuori non è accettabile;
3. La situazione di quest'ultimo caso è quella che capita più spesso ed è quella in cui siamo noi a scegliere i numeri dei thread da utilizzare. Quindi se  $N < 20$  e  $N = 18$

Associamo ai thread una coppia di componenti:

---

```
if(index < N)
    z(i) = u(i) + v(i)
```

---

Problema del terzo caso, con la formula da noi utilizzata potrebbero esserci dei casi in cui nessun blocco ha assegnato delle componenti e questo porta alla perdita del parallelismo.

Per risolvere questo problema segliamo in modo arbitrario la dimensione.

1. Fissiamo blockDim.x ad un valore, in questo esempio mettiamolo a 32.
2. Scelgo gridDim.x in funzione di N e blockDim

Es:

- $N = 64 \rightarrow \text{gridDim.x} = \frac{N}{\text{blockDim.X}} = \frac{64}{32}$
- $N = 70 \rightarrow \text{gridDim.x} = \frac{70}{32} = 2$  elementi fuori, per non avere elementi fuori possiamo applicare quest'altra formula:  $\text{gridDim.x} = \frac{N}{\text{blockDim}} + 1$

Quindi segliamo in modo che almeno un thread nell'ultimo blocco lavori sempre.

## 4.3 Esempio di capability

### 4.3.1 Configurazione di un kernel

Vogliamo configurare un kernel con compute capability di tipo 2.0 (dati reperibili sulla tabella 4.1.8).

Innanzitutto dobbiamo tenere conto di:

- Massimo numero di blocchi: 65535
- Massimo numero di thread: 1024
- Massimo numero blocchi in SM: 8
- Massimo numero di thread in SM : 1536
- Massimo numero di registri in SM: 32K

Abbiamo anche due limitazioni il numero di blocchi/thread da utilizzare e la capienza della memoria.

Vogliamo ottenere una configurazione ottimale, ma non tutte le configurazioni sono ottimali per via delle limitazioni che possono verificarsi. Una configurazione si dice ottimale se vengono utilizzati il massimo numero di blocchi per ogni SM e se vengono utilizzati il massimo numero di thread per ogni SM.

Il nostro esempio si baserà su di una geometria 1D.



Per prima cosa vediamo per il nostro SM quanti thread può contenere all'interno di ogni blocco:

$\frac{\# \text{thread}}{\# \text{blocchi}} = \frac{1536}{8} = 192$ . Quindi blockDim = 192 è la scelta massima di # blocchi e #thread per ogni SM e quindi sarebbe la scelta ottimale; ma per vedere se è vero dobbiamo tener conto della memoria.

Supponiamo  $\forall$  thread occorrono 30 registri,  $\forall$  SM abbiamo  $30 \times 1536$  ( $8 \times 192$ ) registri = 46 080.

Notiamo quindi che abbiamo superato il numero di registri disponibili che è 32K quindi non è una scelta ottima.

Ora supponiamo che si attivano solo 5 blocchi:

$5 \times 192 \times 30 = 28880$ , in questo caso va bene perchè non superiamo 32k.

Quindi se scegliamo blockDim = 192 e si attivano 5 blocchi  $\forall$  SM abbiamo che alla fine saranno attivi  $5 \times 192$  thread = 960 thread  $\forall$  SM < 1536.

**Un'altro esempio:** Proviamo un'altra scelta sempre con compute capability 2.0, ma stavolta con blockDim = 128.  $8 \times 128 = 1024$  thread < 1536. È ottima?  $1024 \times 30 = 30720 < 32K$  quindi è OK. Si attiveranno per ogni  $\forall$  8 blocchi e 1024 thread, la soluzione quindi possiamo dire che è buona ma non è ottima.

## 4.4 Somma di due matrici

Si vuole effettuare la somma di due matrici con il calcolo parallelo su GPU.

### 4.4.1 Introduzione

Innanzitutto qual è la dimensione ottimale di un blocco di thread (BLOCK\_DIM) su architetture Fermi e Kepler? Esistono infatti dei vincoli strutturali:

- Un blocco di CUDA thread può contenere al massimo 1024 thread;
- La griglia di blocchi non può contenere più di 65535 blocchi per Fermi o 231-1 blocchi per Kepler;
- BLOCK\_DIM deve essere scelto in modo da saturare le risorse a disposizione dello Streaming Multiprocessor (SM) e da elevare il grado di parallelismo potenziale.

Infine si deve controllare che vengano rispettati i vincoli sull'utilizzo della memoria.

### 4.4.2 Esempio con FERMI

Nell'architettura FERMI ogni SM può gestire fino a 1536 thread.

- Numero massimo di 8 blocchi residenti per SM e blocco  $8 \times 8 = 64$  thread :  $1536/64 = 24$  blocchi per occupare tutto un SM. Con un massimo di 8 blocchi per SM :  $64 \times 8 = 512$  thread per SM, su di un totale di 1536 disponibili;
- blocco  $16 \times 16 = 256$  thread :  $1536/256 = 6$  blocchi  $256 \times 6 = 1536$  thread per SM (Piena occupazione dello SM!);
- blocco  $32 \times 32 = 1024$  thread :  $1536/1024 = 1.5 \Rightarrow 1$  blocco per SM  $1024 \times 1 = 1024$  thread per SM.

Quindi possiamo dire che BLOCK\_DIM = 16 è scelta ottimale.

### 4.4.3 Esempio con KEPLER

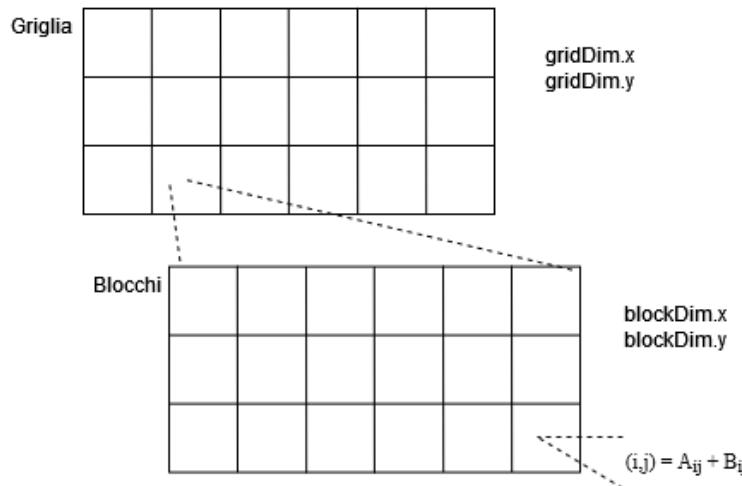
ogni SM può gestire max 2048 thread; max 16 blocchi per SM.

- $8 \times 8 = 64$  thread :  $2048/64 = 32$  blocchi per occupare tutto un SM, con un massimo di 16 blocchi per SM :  $64 \times 16 = 1024$  thread per SM su un totale di 2048 disponibili;
- $16 \times 16 = 256$  thread :  $2048/256 = 8$  blocchi e  $256 \times 8 = 2048$  thread per SM (Piena occupazione dello SM!);
- $32 \times 32 = 1024$  thread :  $2048/1024 = 2$  blocchi e  $1024 \times 2 = 2048$  thread per SM (Piena occupazione dello SM ma minore parallelismo potenziale).

Quindi possiamo dire che BLOCK\_DIM = 16 è scelta ottimale.

#### 4.4.4 Lavagna

Supponiamo di avere Due Mtrici A,B enteambe di dimensione n x n e di volerle sommare: A + B = C. Quindi in questo caso non utliziamo Geometrie 1D ma 2D:



Ogni thread sommerà  $C_{i,j} = A_{i,j} + B_{i,j}$ .

Per comodità anche in questo esercizio usiamo array monodimensionali. La relazione tra l'indice dell'array monodimensionale dei dati del problema e le coordinate del thread è la seguente:

- $i = blockIdx.x * blockDim.x + threadIdx.x;$
- $j = blockIdx.y * blockDim.y + threadIdx.y;$
- $index = j * gridDim.x * blockDim.x + i;$

Altro esempio: Sia A matrice M x N, supponiamo di voler assegnare un elemento di A a ciascun thread, la corrispondenza è: A(row, col) con:

- $\text{row} = blockIdx.y * blockDim.y + threadIdx.y;$
- $\text{col} = blockIdx.x * blockDim.x + threadIdx.x.$

## 4.5 Memorie di una GPU

Un buon uso delle memorie è fondamentale per ottenere buone prestazioni nel calcolo su GPU

## 4.6 Schema delle memorie

### 4.6.1 Memoria globale

La memoria globale è la memoria principale della GPU, è accessibile da tutti i thread sia in lettura che in scrittura. È il canale di comunicazione con l'host (cudaMemcpy opera qui). Questa tipologia di memoria è dell'ordine dei Gb ma è lenta. In questa memoria sono presenti tutti gli argomenti del kernel.

### 4.6.2 Memoria condivisa

E' la memoria che offre le migliori prestazioni, per cui è consigliabile massimizzarne l'uso ed è un canale di comunicazione tra tutti i thread di un blocco. Le variabili della memoria condivisa vivono solo nel kernel e si dichiarano nel seguente modo:

```
__shared__ float variable;
```

La grandezza è nell'ordine dei KB ma è veloce.

### 4.6.3 Registri

Nei registri vengono memorizzate tutte le variabili locali del kernel. E' la memoria con i migliori tempi di accesso.

```
float variable;
```

Il registro è un'area di memoria privata di ogni thread ed è veloce.

### 4.6.4 Memoria locale

Memoria usata nel kernel, per memorizzare: array o altre variabili, qualora si superi la memoria dei registri. E' una memoria lenta.

### 4.6.5 Memoria costante

Memoria da usare quando si deve accedere (in lettura) molte volte ad una stessa variabile, in quanto riduce i tempi di attesa

```
__ constant__ float variable;
```

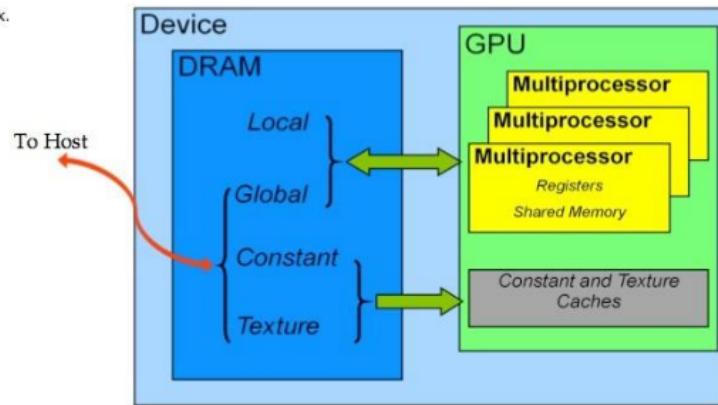
E' una memoria di sola lettura visibile da tutti i thread della griglia e dall'host, è molto veloce.

#### 4.6.6 Memoria Texture

E' una memoria veloce di sola lettura visibile a tutti i thread ottimizzata per la località spaziale 2D.

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

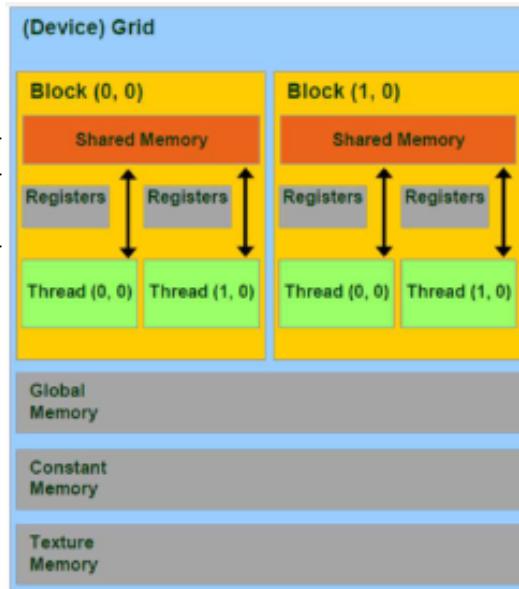
<sup>†</sup>Cached only on devices of compute capability 2.x.



## 4.7 Shared memory e prodotto scalare

### 4.7.1 Shared memory

La memoria condivisa (Shared Memory) è una memoria veloce residente su ciascuno Streaming Multiprocessor. È accessibile in lettura e scrittura dai soli thread del blocco: canale di comunicazione tra i thread di un blocco. La sua durata è limitata al kernel e non mantiene il suo stato tra il lancio di un kernel e l'altro. Bassa Latenza: 2 cicli di clock, throughput: 4 bytes per banco ogni 2 cicli infine di default ha 48 KB (Configurabile : 16/48 KB).



In esecuzione una risorsa critica che limita il parallelismo a livello device. Viene ripartita tra tutti i blocchi residenti in un SM, maggiore è la shared memory richiesta da un kernel, minore è il numero di blocchi attivi concorrenti. L'accesso è per warp e il caso migliore è una transazione x 32 thread, il caso peggiore 32 transazioni.

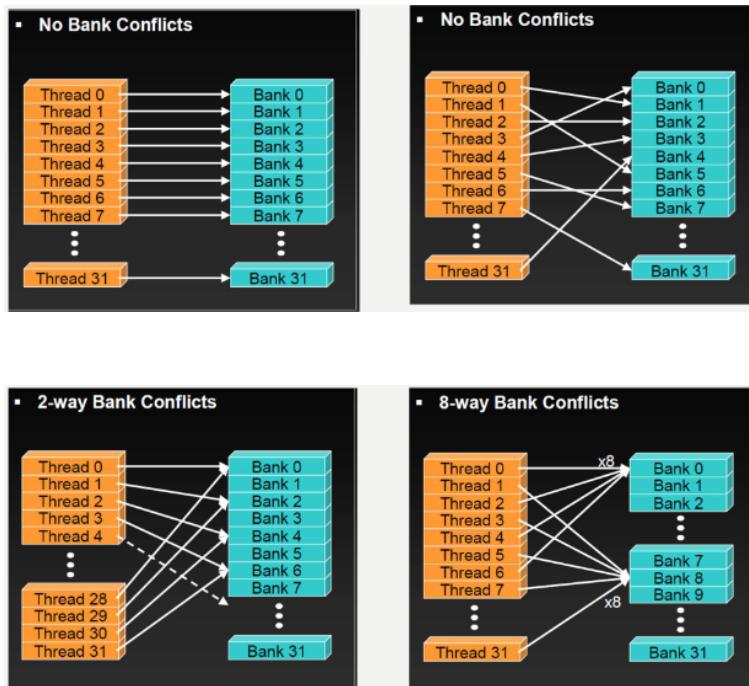
Viene utilizzata nelle seguenti modalità:

- Si caricano i dati nella memoria shared;
- Si sincronizza (se necessario);
- Si opera sui dati nella memoria shared;
- Si sincronizza (se necessario);
- Si scrivono i risultati nella memoria globale.

#### 4.7.2 Organizzazione della shared memory

La shared memory è organizzata in 32 banchi da 4-byte di ampiezza ciascuno (dette word, ciascuna può contenere 1 int, 1 float, 2 short, 4 char, 1 half double, ...). I dati vengono distribuiti ciclicamente su banchi successivi ogni 4-byte, e gli accessi alla shared memory avvengono per warp. Abbiamo diversi tipi di accessi:

- Multicast : se n thread del warp accedono allo stesso elemento, l'accesso è eseguito in una singola transazione;
- Broadcast : se tutti i thread del warp accedono allo stesso elemento, l'accesso è eseguito in una singola transazione;
- Bank Conflict : se due o più thread differenti (dello stesso warp) tentano di accedere a dati differenti, residenti sullo stesso banco. Ogni conflitto viene servito e risolto serialmente.



Conflitto doppio (ottuplo): due (otto) thread differenti dello stesso warp tentano di accedere a dati differenti, residenti sullo stesso banco.

### 4.7.3 Lavagna tipi di accessi al banco

Memorizziamo in un array shared di 66 elementi: `__shared__ int a[66];`  
Il banco sarà:

0	a(0)	a(32)	a(64)		
1	a(1)	a(33)	a(65)		
...	...	...			
...	...	...			
31	a(31)	a(63)			

Table 4.2:

Ricordando che siamo sempre all'interno del kernel abbiamo che:

- Broadcast:  $x = a(2) + 5$ ; Tutti accedono alla stessa variabile: OK;
- Multicast:

---

```
if(threadIdx.x < 5)
    x = a(5) +1; // thread 0,1,2,3,4 accedono alla stessa variabile:
                    OK
```

---

- Bank conflict:  $x = a(2 * threadIdx.x) + 10$ ; Supponiamo che il banco seguente sia del warp 1:

Thread	Variabile	Banco
0	a(0)	0
1	a(2)	2
...	...	...
15	a(30)	30
16	a(32)	0
...	...	...
30	a(60)	28
31	a(62)	30

Table 4.3:

I conflitto li abbiamo sui thread 0 e 16 perchè accedono a elementi diversi ma presenti nello stesso banco. Questo succede anche ai thread 15 e 31.

#### Osservazioni

- Latency hiding: il ritardo che intercorre tra la richiesta avanzata dai thread alla Shared memory e l'ottenimento effettivo dei dati non è in generale un problema anche in caso di bank conflict: se vi sono molti thread in esecuzione, lo scheduler passa a un altro warp in attesa che quelli sospesi completino il trasferimento dei dati dalla shared memory. In questo modo il ritardo potrebbe essere irrilevante;
- Efficienza massima: Il modo più semplice per avere prestazioni elevate è quello di fare in modo che un warp acceda a word consecutive in memoria shared;
- Caching: con lo scheduling efficace, anche in presenza di conflitti, le prestazioni sono di gran lunga migliori rispetto a quelle in cui il cammino che i dati percorrono passa attraverso la cache L2 o peggio, la global memory.

## 4.8 Prodotto scalare

Implementare in cuda il prodotto scalare fra due array di float,  $c = a \cdot b$ , dove  $a$  e  $b$  sono array di uguali dimensioni e con  $N$  elementi ( $N$  costante definita opportunamente o letta dall'utente). Preso un vettore  $v$  come segue:

$$v = (a_1 b_1, a_2 b_2, \dots, a_n b_n)$$

abbiamo

$$c = \sum_{i=0}^n a_i$$

### 4.8.1 Kernel strategia 1 (banale)

---

```
--global__ void dotProdGPU(float *v, float *a, float *b, int N){
    //global index
    int idx=threadIdx.x+blockIdx.x*blockDim.x;
    if idx< N
        v[idx]=a[idx]*b[idx];
}
```

---

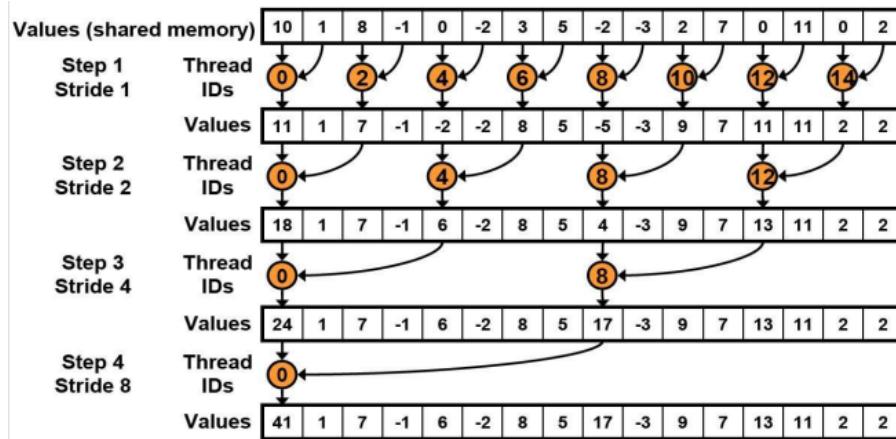
### 4.8.2 Come calcolare C a partire da V

Possiamo calcolare C banalmente in maniera seriale sull'host ma così facendo abbiamo un basso parallelismo, un uso solo della memoria globale (lenta) e inoltre dobbiamo trasferire V. Invece è possibile migliorare le prestazioni sfruttando il parallelismo sulla GPU, mediante la memoria condivisa.

### 4.8.3 Riduzione

Nel calcolo parallelo, capita frequentemente di ridurre più valori, calcolati in parallelo, in un singolo valore. Tra le possibili operazioni di riduzione sono: ADD, MUL, MIN, MAX.

#### Schema di riduzione in CUDA



Come possiamo vedere i thread attivi ad ogni passo dimezzano quindi questo codice è altamente divergente ottenendo prestazioni molto deludenti.

## 4.9 Allocazione statica

### 4.9.1 Esempio 1

---

```
--global__ void staticKernel(...){  
    __shared__ int s[30];  
    ...  
}
```

---

### 4.9.2 Esempio 2

---

```
Nel main  
const int n = 64;  
  
Nel kernel  
__global__ void staticKernel(...){  
    __shared__ int s[n];  
    ...  
}
```

---

## 4.10 Allocazione dinamica

### 4.10.1 Esempio 3

---

```
--global__ void dynamicKernel(...){  
    extern __shared__ int M[ ];  
    ...  
}  
  
Chiamata nel main  
  
sizeM=m*sizeof(int);  
dynamicKernel<<<grid, block, sizeM>>>(...)  
...
```

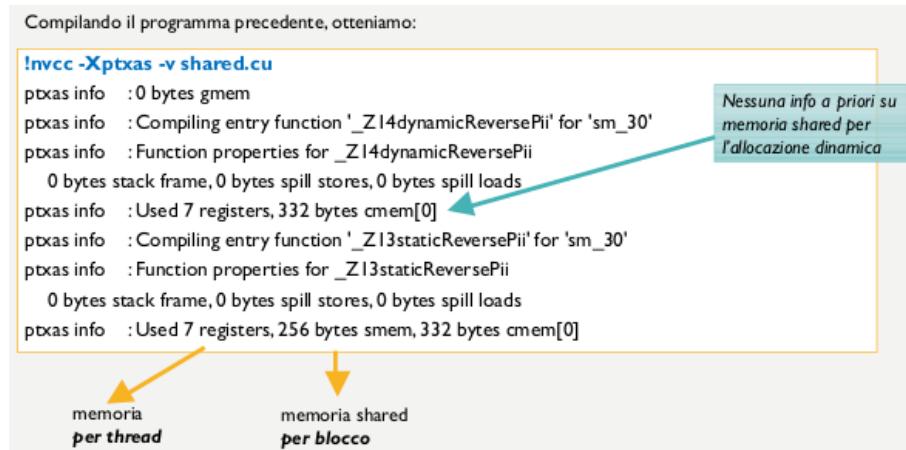
---

Nel caso di allocazione dinamica, va aggiunto un terzo parametro nella configurazione, dato dalla dimensione in byte dell'area di memoria shared (per ogni blocco di thread)

## 4.11 Info su shared memory

Compilando il programma precedente, otteniamo:

```
!nvcc -Xptxas -v shared.cu
ptxas info : 0 bytes gmem
ptxas info : Compiling entry function '_Z14dynamicReversePii' for 'sm_30'
ptxas info : Function properties for '_Z14dynamicReversePii'
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info : Used 7 registers, 332 bytes cmem[0]
ptxas info : Compiling entry function '_Z13staticReversePii' for 'sm_30'
ptxas info : Function properties for '_Z13staticReversePii'
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info : Used 7 registers, 256 bytes smem, 332 bytes cmem[0]
```



## 4.12 Lavagne prodotto scalare

### 4.12.1 Prima strategia (banale)

Abbiamo innanzi tutto due vettori:

- $a = (a_0, \dots, a_{n-1})$
- $b = (b_0, \dots, b_{n-1})$

il vettore risultante finale sarà:  $c = a * b = \sum_{i=0}^{n-1} a_i b_i = \sum_{i=0}^{n-1} v_i$

con  $v_0 = (a_0 b_0, \dots, a_{n-1} b_{n-1})$ , infine avremo che:

1. Device: calcoliamo  $v$

---

```
--global__ void dotProdGPU(float *v, float *a, float *b, int N){
    //global index
    int idx=threadIdx.x+blockIdx.x*blockDim.x;
    if idx< N
        v[idx]=a[idx]*b[idx];
}
```

---

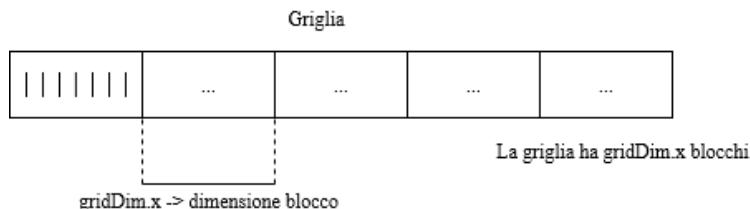
2. Host: calcoliamo  $\sum_{i=0}^{n-1} a_i b_i = \sum_{i=0}^{n-1} v_i \rightarrow c$

### 4.12.2 Seconda strategia

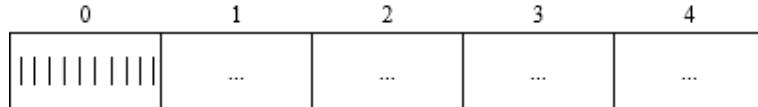
1. Device: calcoliamo  $v$

2. Device: calcoliamo  $\sum_{i=0}^{n-1} a_i b_i$

Immaginiamo di avere una griglia 1D:



**Esempio:** prendiamo come  $n = 50$  e che la griglia abbia 5 blocchi con ciascuno 10 thread:



1. Ogni thread calcola una componente di  $\mathbf{v}$

$$2. \mathbf{c} = \sum_{i=0}^{49} a_i b_i = \underbrace{\sum_{i=0}^9 a_i b_i}_{c_0} + \underbrace{\sum_{i=10}^{19} a_i b_i}_{c_1} + \underbrace{\sum_{i=20}^{29} a_i b_i}_{c_2} + \underbrace{\sum_{i=30}^{39} a_i b_i}_{c_3} + \underbrace{\sum_{i=40}^{49} a_i b_i}_{c_4}$$

Ogni sommatoria sarà assegnata quindi ad un blocco:

Blocchi	0	1	2	3	4
$\mathbf{c}$	0	1	2	3	4

Table 4.4:

Per svolgere in parallelo questi calcoli abbiamo bisogno che i thread cooperino, ma coopereranno soltanto se condividono dei dati, ed è possibile solo all'interno del blocco.

Ogni blocco esegue la somma ad esso assegnato in parallelo usando la memoria condivisa.

### In generale

$$\mathbf{c} = \sum_{i=0}^{n-1} a_i b_i = \sum_{i=0}^{blockDim.x-1} a_i b_i + \sum_{i=blockDim.x}^{2blockDim.x-1} a_i b_i + \dots$$

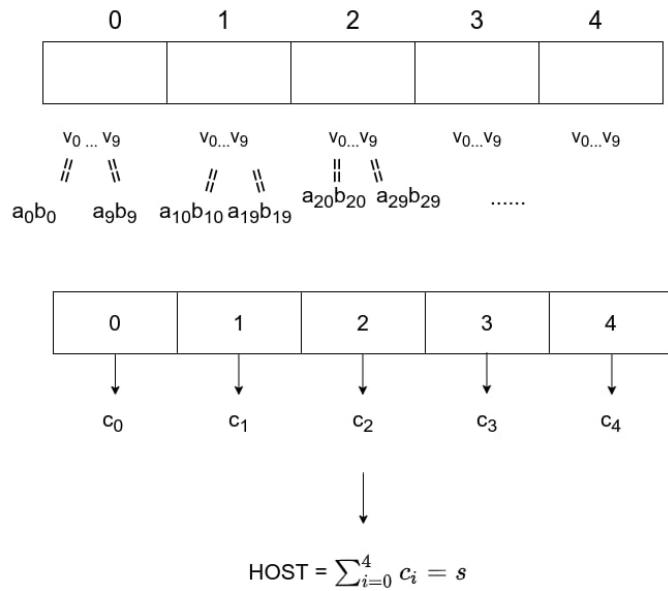
### Kernel

---

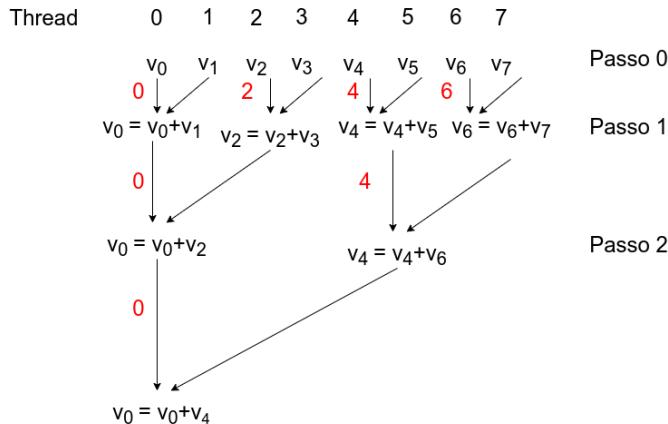
```
prodotto (a, b, c, n) // a e b input c e n saranno dati in output
__shared__ float v[blockDim.x];
index = threadIdx.x + blockIdx.x*blockDim.x;
id = threadIdx.x;
v(id) = a(index) * b(index) // a v e' associata una coppia di elementi di a e
    b
sincronizzazione
somma in parallelo  $\sum_{i=0}^{blockDim.x} v_i -> Sum$ 
sincronizzazione
c[blockIdx.x] = Sum
END
```

---

Dopo l'esecuzione del codice Kernel l'host effettua una memmove ed esegue  $\sum_{i=0}^{gridDim.x-1} c_i$



### Somma parallela all'interno del blocco



$p = \log_2 blockDim.x$  è il numero di passi.

Ad ogni passo  $K$  con  $k = 0, 1, \dots, p-1$ :

- $\text{Dist} = 2^k, id = threadIdx.x$
- Se  $\text{resto}(id, 2^{k+1}) = 0$   
 $v(id) = v(id) + v(id + \text{Dist})$
- Sincronizzazione

---

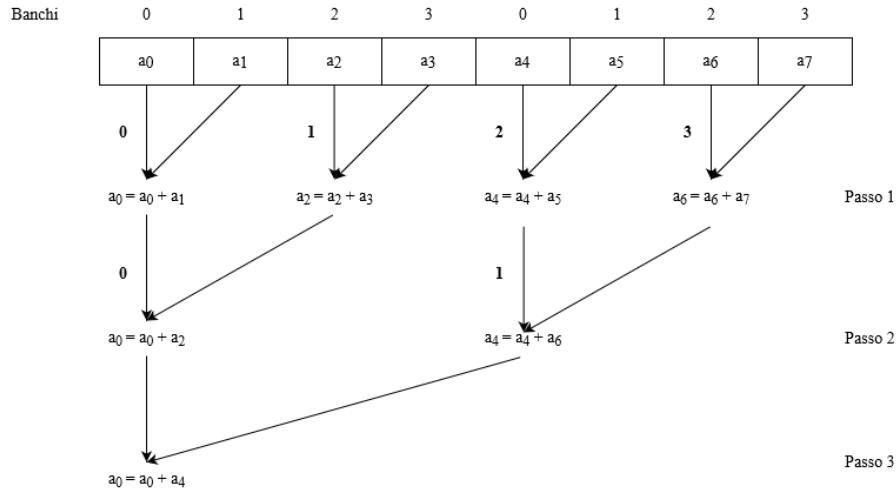
```
if(id = 0) //per via della memoria condivisa conviene farlo fare da un solo thread
    c(blockIdx.x) = v(0)
```

---

Nella seconda strategia ad ogni passo si crea della divergenza, perché abbiamo thread dello stesso warp che lavorano e altri che non eseguono nessun compito. Prestazioni non ottimali.

### Seconda strategia bank conflict

Iporizziamo di avere un Warp di 4 thread e una memoria condivisa con 4 banchi, infine supponiamo di avere un blocco composto da 8 thread:



La situazione che abbiamo è la seguente:

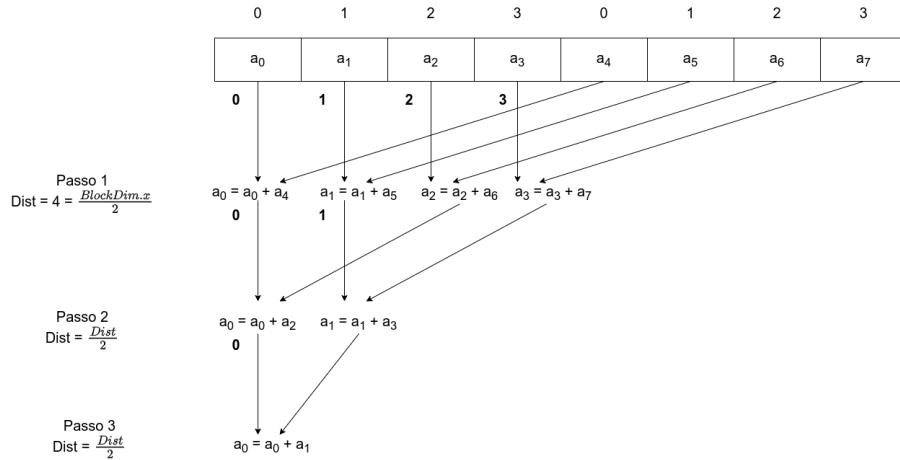
Passo	Thread	Variabile	Banco	
1	0	a(0)	0	Conflitto doppio (accedono a elementi diversi dello stesso banco)
		a(1)	1	
	2	a(4)	0	
		a(5)	1	
	1	a(2)	2	
		a(3)	3	
	3	a(6)	2	
		a(7)	3	
2	0	a(0)	0	Conflitto doppio
		a(2)	2	
	1	a(4)	0	
		a(6)	2	

Table 4.5:

Possiamo osservare quindi diversi conflitti di accesso ai banchi. Quindi presentiamo ora una nuova strategia senza divergenze (o almeno usciranno da un certo punto in poi) e senza conflitti di banco.

### 4.12.3 Terza strategia

Supponiamo sempre di avere blockDim.x = 8, 4 banchi e 4 thread.



Dallo schema notiamo che nel primo warp tutti i thread da 0 a 3 lavorano, mentre dal warp 2 in poi nessun thread lavora. Analizziamo poi l'eventuale presenza di conflitti:

Passo	Thread	Variabile	Banco	
1	0	a(0)	0	Nessun conflitto
		a(4)	0	
	1	a(1)	1	
		a(5)	1	
	2	a(2)	2	
		a(6)	2	
2	0	a(3)	3	Nessun conflitto
		a(7)	3	
	1	a(0)	0	
		a(2)	2	
		a(1)	1	
		a(3)	3	

Table 4.6:

Per la distanza poniamo inizialmente  $\text{Dist} = \frac{\text{BlocchiDim.x}}{2}$  per poi nei passi successivi fare  $\text{Dist} = \frac{\text{Dist}}{2}$ .  $p = \log_2 \text{blockDim.x}$  è il numero di passi e  $\text{blockDim.x}$  potenza di 2.

Ad ogni passo K con  $k = 0, 1, \dots, p-1$ :

- $\text{Dist} = \frac{\text{Dist}}{2}, id = \text{threadIdx.x}$
- if  $id < \text{Dist}$   
 $v(id) = v(id) + v(id + \text{Dist})$
- Sincronizzazione

```
if(id == 0) //per via della memoria condivisa conviene farlo fare da un solo thread
    c(blockIdx.x) = v(0)
```

*Wish you all the best, Andrea Hidalgo*