```
for (k = 0 ; k < n ; k++) {
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            if (m[i][k] + m[k][j] < m[i][j]) {
                m[i][j] = m[i][k] + m[k][j];
            }
        }
    }
}
```

NB

$$\begin{cases} ((i=0) \wedge (k=0 \vee j=0)) \rightarrow \infty \\ ((k=0) \wedge (i=0 \vee j=0)) \rightarrow \infty \\ ((j=0) \wedge (i=0 \vee k=0)) \rightarrow \infty \end{cases}$$

k =	0	1	2	3	4
i =	0	1	2	3	4
j	0	1	2	3	4

m[i][k] + m[k][j] < m[i][j]
m[2][0] + m[0][1] < m[2][1]
10 + 3 < ∞

k =	0	1	2	3	4
i =	0	1	2	3	4
j	0	1	2	3	4

10 + 4 < ∞

⋮

k =	0	1	2	3	4
i =	0	1	2	3	4
j	0	1	2	3	4

13 + 8 < ∞

⚠ gestire concorrenza: questo valore non è garantito essere già presente in un ambiente parallelo!

The **5×5** matrix represents the distances between nodes. Each cell gives the length of a path of the node to the left of the cell to the nodes with the label that is shown on top of the table. The entry ∞ indicates that the path length has not yet been computed.

The algorithm can now start!

If you hover the cursor above a cell, you can see the current path in the graph on the left.

The original matrix looks as follows:

	a	b	c	h	i
a	0	3	∞	∞	4
b	∞	0	8	∞	∞
c	10	∞	0	4	∞
h	2	∞	∞	0	∞
i	∞	2	∞	∞	0

13

	a	b	c	h	i
a	0	3	∞	∞	4
b	∞	0	8	∞	∞
c	10	∞	0	4	∞
h	2	∞	∞	0	∞
i	∞	2	∞	∞	0

14

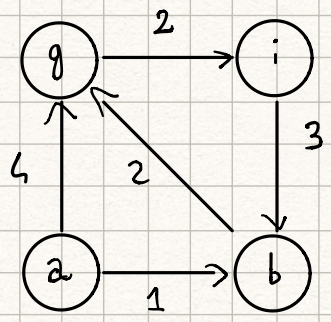
	a	b	c	h	i
a	0	3	∞	∞	4
b	∞	0	8	∞	∞
c	10	13	0	4	∞
h	2	∞	∞	0	∞
i	∞	2	∞	∞	0

Nota [i][J] non si ripeteranno mai se non quando K passa al valore successivo

" [k][J] " " " " " " i " "

```
for (k = 0; k < n; k++) {
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            if (m[i][k] + m[k][j] < m[i][j]) {
                m[i][j] = m[i][k] + m[k][j];
            }
        }
    }
}
```

	a	b	g	i
a	∞	1	4	∞
b	∞	∞	2	∞
g	∞	∞	∞	2
i	∞	3	∞	∞



	a	b	g	i
a	∞	1	3	5
b	∞	∞	2	4
g	∞	5	∞	2
i	∞	3	5	∞

K	i	J
0	1	
0		

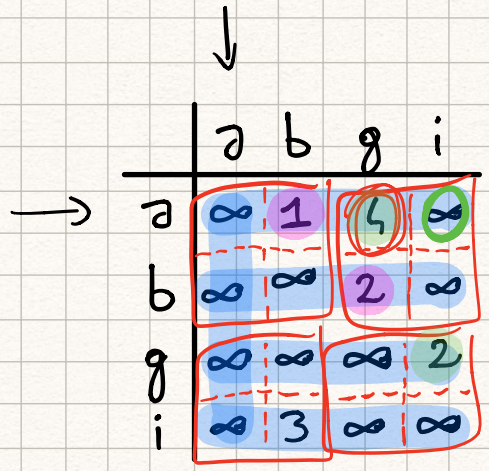
[0][1] + [1][2] < [0][2] => [0][2] = 3

1 + 2 < 4

[0][1] [1][2]

[0][2] [2][3]

3 + 2 < ∞



k 1 th[0][0] b[0][1] → [0][2] ✓

row col

< 4

k 2 th[0][1] b[0][1] → [0][3]

< ∞

K	i	j	
0	0	0	
0	0	1	X
0	0	2	X
0	0	3	X
0	1	0	X
0	1	1	
0	1	2	X
0	1	3	X
0	2	0	✓
0	2	1	X
0	2	2	
0	2	3	X
0	3	0	X
0	3	1	X
0	3	2	X
0	3	3	

Il k è un indice che permette di tenere traccia del nodo intermedio.

1 0 1
1 0 2
1 0 3
1 1 0

1 + ∞
1 + 2 < 4 ✓ [0][2] = 3
1 + ∞
∞ +

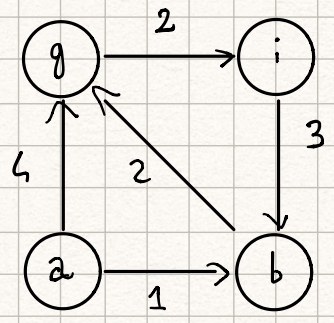
×
×

1 1 2
1 1 3
1 2 0
1 2 1

1 2 3
1 3 0
1 3 1
1 3 2

CONGETTURA: non si ripeterà mai lo stesso [i] [j] per lo stesso k

	a	b	g	i
a	∞	1	4	∞
b	∞	∞	2	∞
g	∞	∞	∞	2
i	∞	3	∞	∞



2 0 0
2 0 1
→ 2 0 2
2 0 3
2 1 0

2 1 2
2 1 3
2 2 0
2 2 1

2 2 3
2 3 0
2 3 1
2 3 2

b → g => 2
g → a => ∞

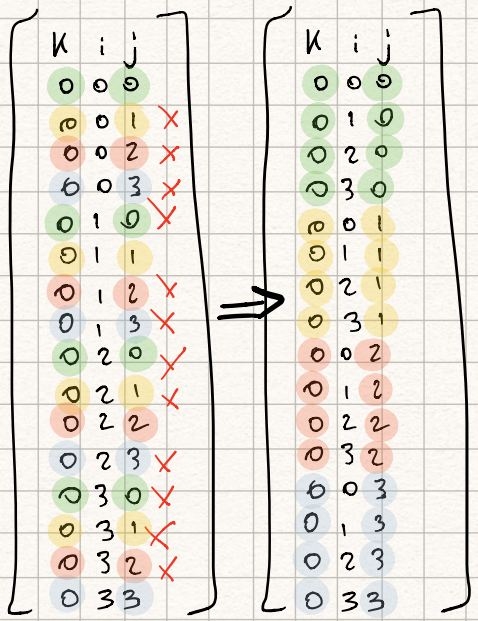
- Ad ogni iterazione il k è lo stesso per tutte le thread e per tutti i blocchi
- Ogni volta che cambio la i si ripetono gli stessi [K][J]

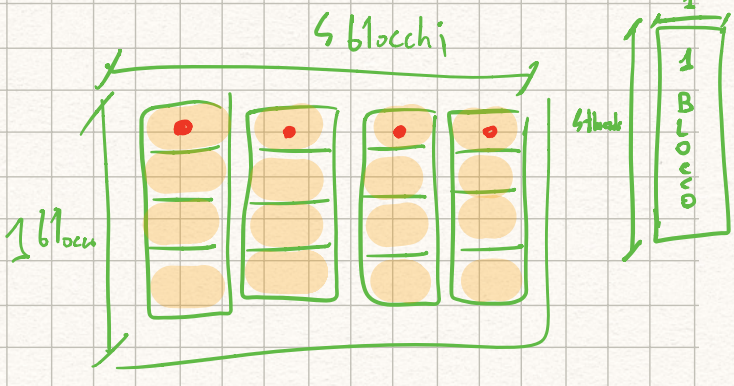
↖ possibile uso di memoria condivisa

3 0 1
→ 3 0 2
3 0 3
3 1 0

3 1 2
3 1 3
3 2 0
3 2 1

3 2 3
3 3 0
3 3 1
3 3 2





Ogni blocco fa le seguenti thread:
th[0][0] → k_J_value = N[2*n+2] =
th[0][1]
th[0][2]
th[0][3]

BLOCCHI

x	y
[0]	[0]
[1]	[0]
[2]	[0]
[3]	[0]

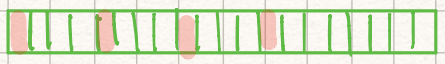
THREADS

x	y
[0]	[0]
[0]	[1]
[0]	[2]
[0]	[3]

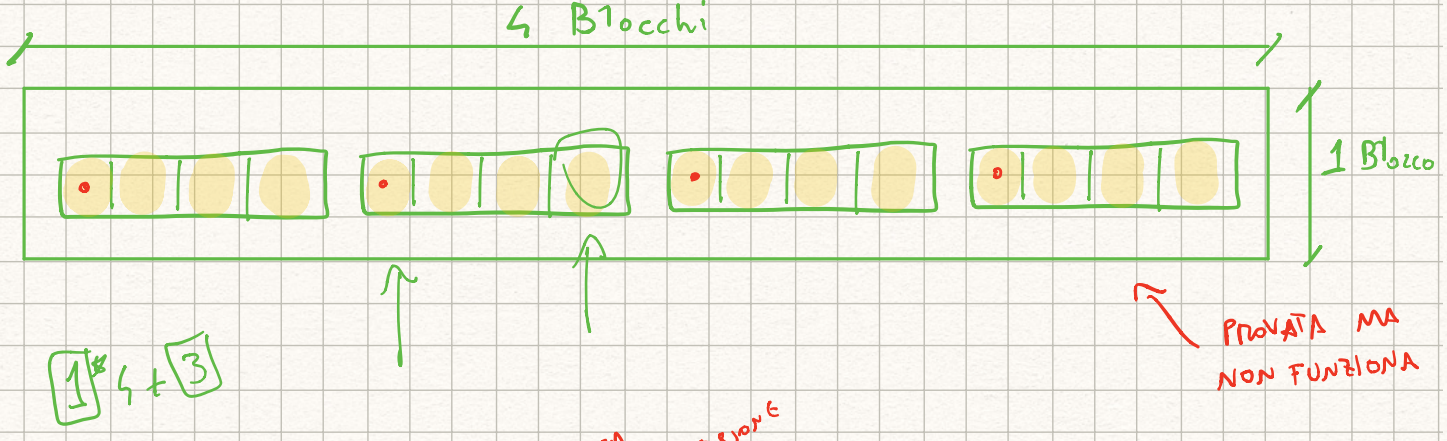
```
i = by * bDim + ty
J = bx
```

i	J	
0	0	Valori calcolati dal blocco 0
1	0	
2	0	
3	0	
0	1	Valori calcolati dal blocco 1
1	1	
2	1	
3	1	
0	2	Valori calcolati dal blocco 2
1	2	
2	2	
3	2	
0	3	Valori calcolati dal blocco 3
1	3	
2	3	
3	3	

```
tx + ty * BlockDim / warpSize = warpID
0 + 0  /32 = 0
0 + 4  /32
0 + 8  /32
0 + 12 /32
```



=> WARP DIVERGENCE + ACCESSI A MEMORIA NON COALIZZATI



BLOCCHI

x	y
[0]	[0]
[0]	[1]
[0]	[2]
[0]	[3]

THREADS

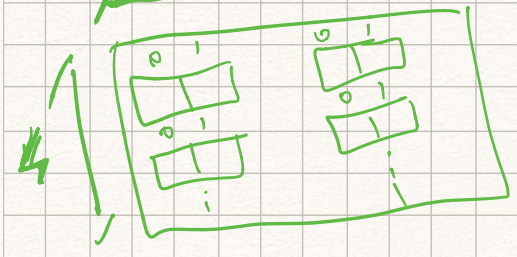
x	y
[0]	[0]
[1]	[0]
[2]	[0]
[3]	[0]

QUESTA È LA VERSIONE IMPLEMENTATA

NON SI PUÓ FARE PERCHÉ LA "i" NON VARIA NELLO STESSO BLOCCO, QUINDI GLI STESSI [K][J] SI RIPETONO SOLO IN BLOCCHI DIVERSI

i	J	
0	0	Valori calcolati dal blocco 0
0	1	
0	2	
0	3	
1	0	Valori calcolati dal blocco 1
1	1	
1	2	
1	3	
2	0	

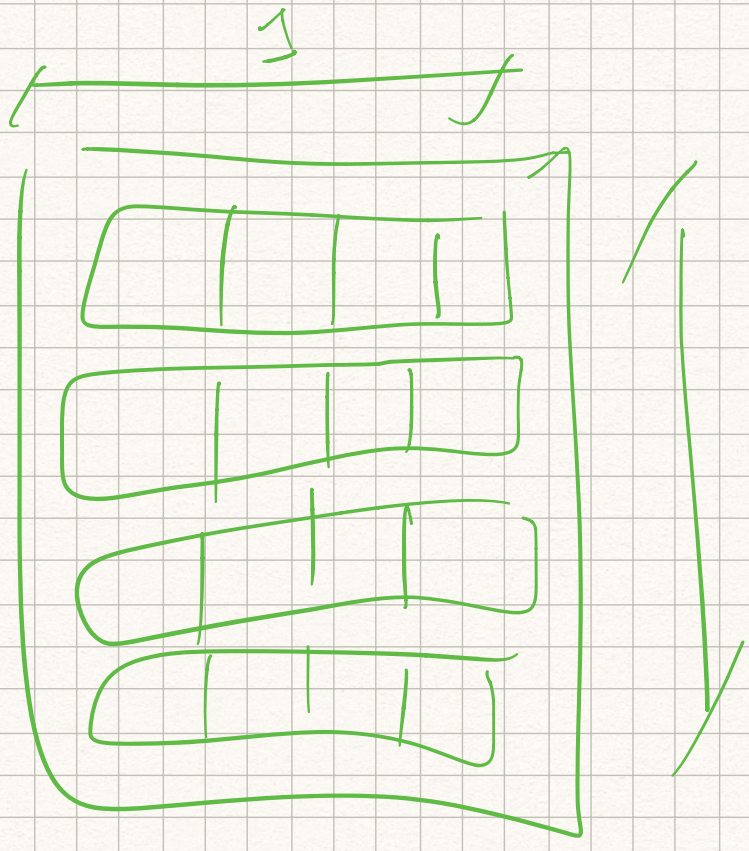
MA AVRÓ GLI STESSI [i][k] AVENDO COME VARIANTE NELLO STESSO BLOCCO LA "J"



2	1
2	2
2	3
3	0
3	1
3	2
3	3

Valori calcolati dal blocco 2

Valori calcolati dal blocco 3



i	j	
0	0	blocco 0
1	0	
2	0	
3	0	
0	1	blocco 1
1	1	
2	1	
3	1	

tx + ty * BlockDim / warpSize = warpID

0	+ 0	/ 32
1	+ 0	/ 32
2	+ 0	/ 32
3	+ 0	/ 32

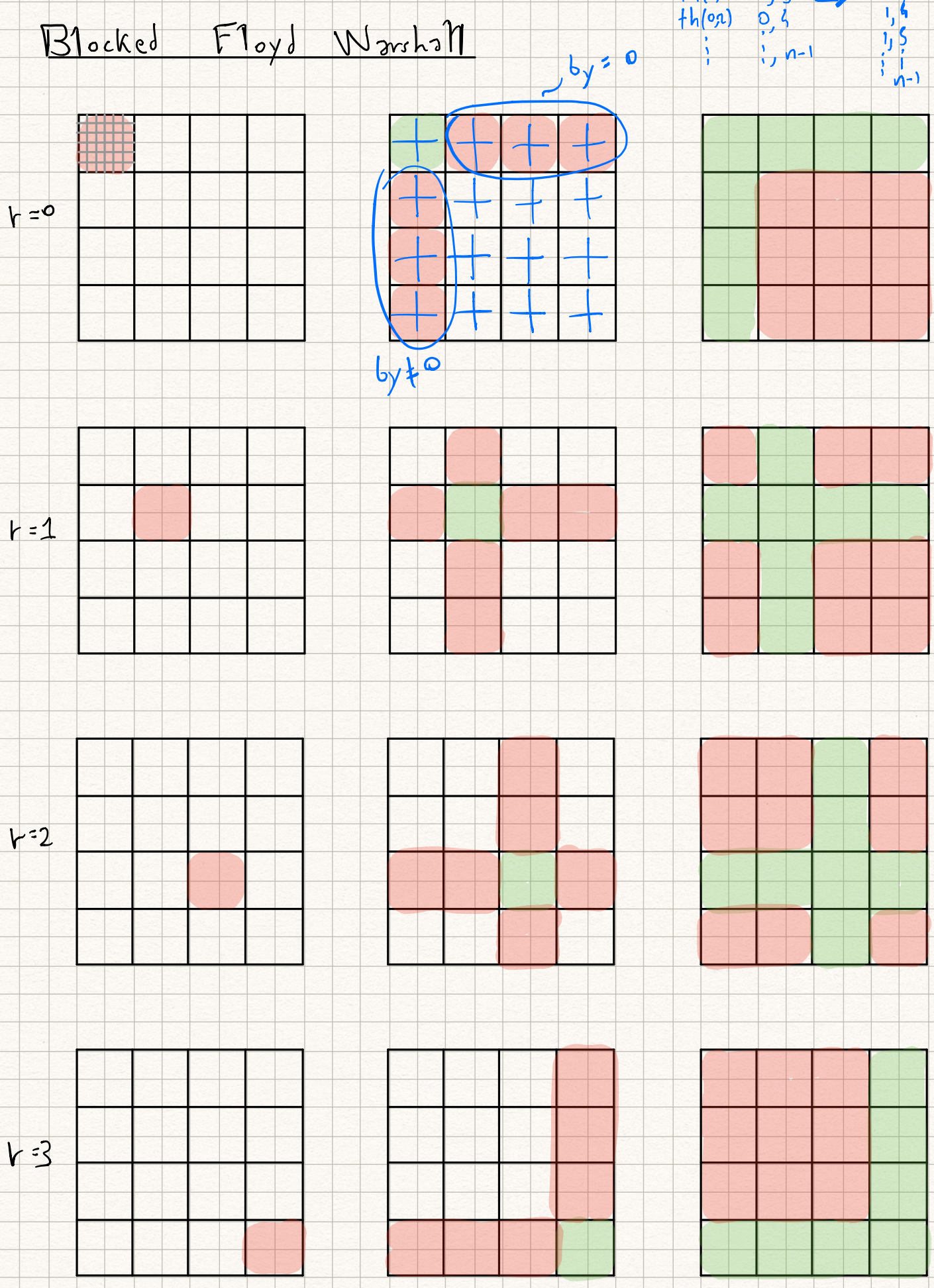
PROBLEMA

RIMANGONO GLI ACCESSI A MEMORIA NON COALIZZATI.

Notiamo quindi che con questo approccio, nonostante l'uso di memoria condivisa non si hanno grandi risultati

	ph 2,0	ph 2,1
th(0,0)	0, 2	1, 0
th(0,1)	0, 3	1, 1

Blocked Floyd Warshall



Esempio

k = 1 ← stage

phase 1:

$D^0(i,j) = A^0(i,j) = cost(i,j)$

$D^k(i,j) = A^k(i,j) = \min\{A^{k-1}(i,j),\ A^{k-1}(i,k) + A^{k-1}(k,j)\}$

$d^{(1)}(0,0) = \min(\infty,\ 1+\infty) = \infty$
$d^{(1)}(0,1) = \min(1,\ 1+\infty) = 1$
$d^{(1)}(1,0) = \min(\infty,\ \infty+\infty) = \infty$
$d^{(1)}(1,1) = \min(\infty,\ \infty+\infty) = \infty$

	a	b	g	i
a	∞	1	4	∞
b	∞	∞	2	∞
g	∞	∞	∞	2
i	∞	3	∞	∞

phase 2:

$D^0(i,j) = A^0(i,j)$

$D^k(i,j) = \min\{D^{k-1}(i,j),\ D^{k-1}(i,k) + D^{k-1}(k,j)\}$ rows

$D^k(i,j) = \min\{D^{k-1}(i,j),\ D^{k-1}(i,k) + D^{k-1}(k,j)\}$ cols

$d^{(1)}(0,2) = \min(4,\ 1+2) = 3$
$d^{(1)}(0,3) = \min(\infty,\ 1+\infty) = \infty$
$d^{(1)}(1,2) = \min(2,\ \infty+2) = 2$
$d^{(1)}(1,3) = \min(\infty,\ \infty+\infty) = \infty$ rows

$d^{(1)}(2,0) = \min(\infty,\ \infty+\infty) = \infty$
$d^{(1)}(2,1) = \min(\infty,\ \infty+\infty) = \infty$
$d^{(1)}(3,0) = \min(\infty,\ 3+\infty) = \infty$
$d^{(1)}(3,1) = \min(3,\ 3+\infty) = 3$ cols

	a	b	g	i
a	∞	1	3 (~~4~~)	∞
b	∞	∞	2	∞
g	∞	∞	∞	2
i	∞	3	∞	∞

phase 3:

$D^0(i,j) = A^0(i,j)$

$D^k(i,j) = \min\{D^{k-1}(i,j),\ D^{k-1}(i,k) + D^{k-1}(k,j)\}$

$d^{(1)}(2,2) = \min(\infty,\ \infty+2) = \infty$
$d^{(1)}(2,3) = \min(2,\ \infty+\infty) = 2$
$d^{(1)}(3,2) = \min(\infty,\ 3+2) = 5$
$d^{(1)}(3,3) = \min(\infty,\ 3+\infty) = \infty$

∞	1	3	∞
∞	∞	2	∞
∞	∞	∞	2
∞	3	~~∞~~ 5	∞

	PRIM_R	PRIM_C	
(2,2)	(0,0)	+ (0,0)	∞ + 3
(2,2)	(0,1)	+ (1,0)	∞ + 2

K = 2

Phase 1:

$D^0(i,j) = A^0(i,j) = cost(i,j)$

$D^k(i,j) = A^k(i,j) = \min\{D^{k-1}(i,j),\ D^{k-1}(i,k) + D^{k-1}(k,j)\}$

doubio su questo indice !!!

$d^{(2)}(2,2) = \min(\infty,\ 3+\infty) = \infty$
$d^{(2)}(2,3) = \min(2,\ 3+\infty) = 2$
$d^{(2)}(3,2) = \min(5,\ 2+\infty) = 5$
$d^{(2)}(3,3) = \min(\infty,\ 2+\infty) = \infty$

	a	b	g	i
a	∞	1	3	∞
b	∞	∞	2	∞
g	∞	∞	∞	2
i	∞	3	5	∞

Phase 2:

$D^0(i,j) = A^0(i,j)$

$D^k(i,j) = \min\{D^{k-1}(i,j),\ D^{2B-1}(i,k) + D^{k-1}(k,j)\}$ rows

$D^k(i,j) = \min\{D^{k-1}(i,j),\ D^{k-1}(i,k) + D^{2B-1}(k,j)\}$ cols

$d^{(2)}(2,0) = \min(\infty,\ \infty+\infty) = \infty$
$d^{(2)}(2,1) = \min(\infty,\ 2+3) = 5$
$d^{(2)}(3,0) = \min(\infty,\ 5+\infty) = \infty$
$d^{(2)}(3,1) = \min(3,\ 5+\infty) = 3$

$d^{(2)}(0,2) = \min(3,\ 3+\infty) = 3$
$d^{(2)}(0,3) = \min(\infty,\ 3+2) = 5$
$d^{(2)}(1,2) = \min(2,\ 2+\infty) = 2$
$d^{(2)}(1,3) = \min(\infty,\ 2+2) = 4$

rows

cols

	P	OWN	
(2,1),	(0,0)	+ (0,1)	∞ + ∞
(2,1),	(0,1)	+ (1,1)	2 + 3

	a	b	g	i
a	∞	1	3	~~∞~~ 5
b	∞	∞	2	~~∞~~ 4
g	∞	~~∞~~ 5	∞	2
i	∞	3	5	∞

Phase 3:

$D^0(i,j) = A^0(i,j)$

$D^k(i,j) = \min\{D^{k-1}(i,j),\ D^{2B-1}(i,k) + D^{2B-1}(k,j)\}$

$d^{(2)}(0,0) = \min(\infty,\ 3+\infty) = \infty$
$d^{(2)}(0,1) = \min(1,\ 3+\infty) = 1$
$d^{(2)}(1,0) = \min(\infty,\ 2+\infty) = \infty$
$d^{(2)}(1,1) = \min(\infty,\ 2+\infty) = \infty$

PRIM_R PRIM_C

(1,1) (1,0) + (0,1) ∞ + 5

(1,1) (1,2) + (2,1) 2 + 5

	a	b	g	i
a	∞	1	3	5
b	∞	∞	2	4
g	∞	5	∞	2
i	∞	3	5	∞

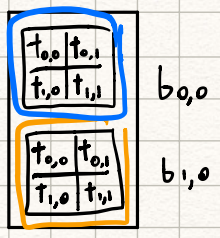
✓

STAGES = 2

$0 \leq bx < stages - 1$

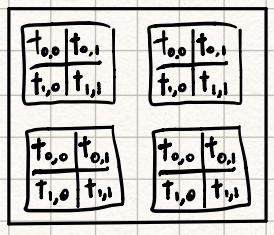
PHASE 2 → Kernel config.

<<< stages - 1, z >>>



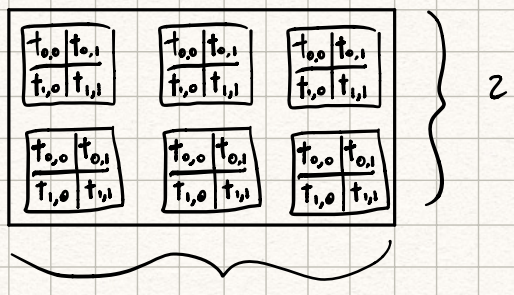
stage = K

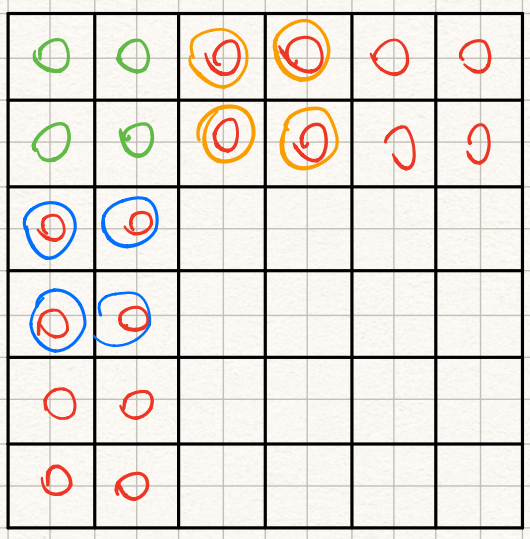
$0 \leq K < stages$

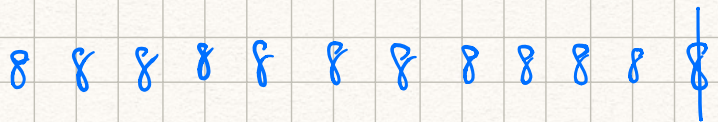


i blocchi di questa riga corrispondono alle righe di N →

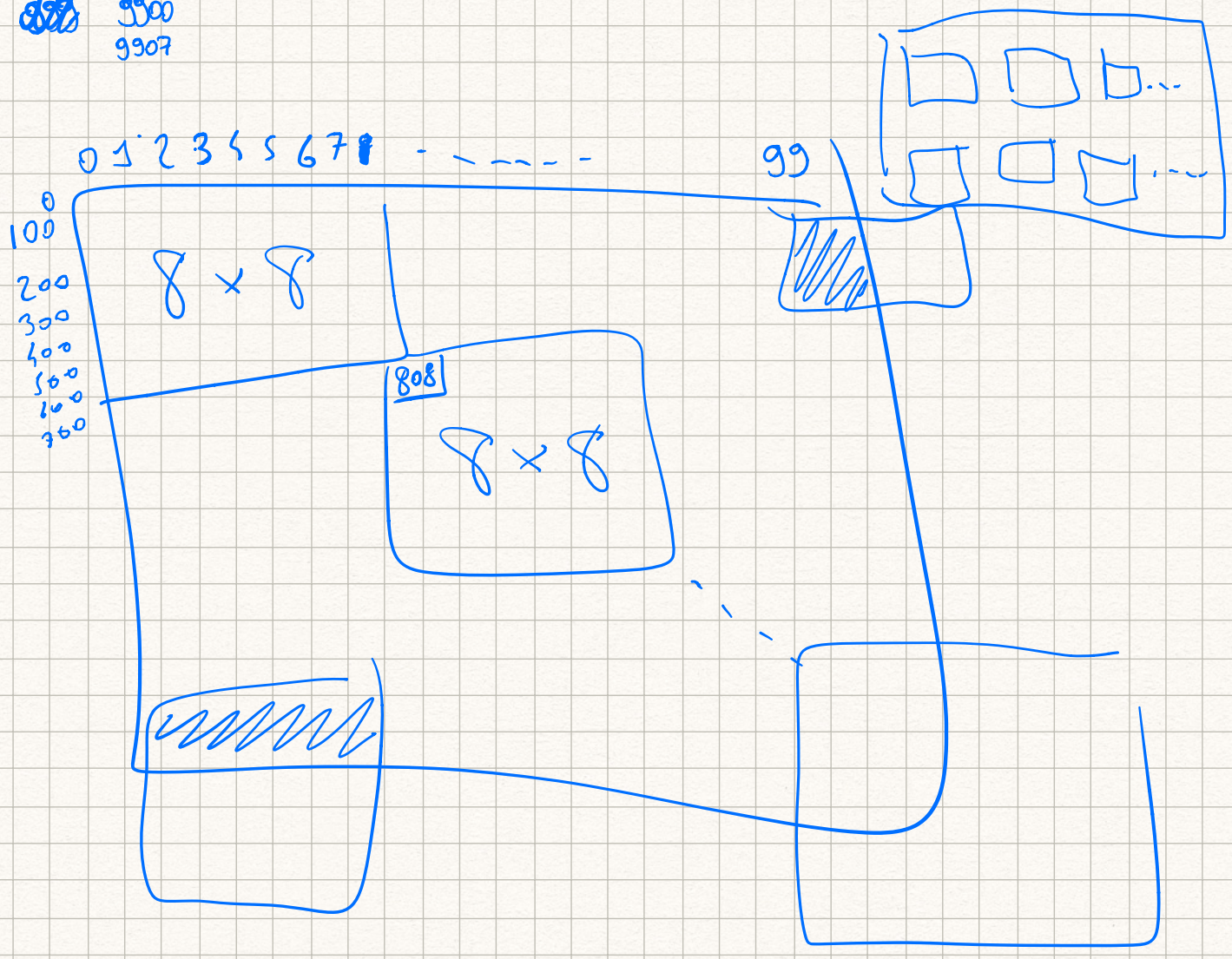
i blocchi di questa riga corrispondono alle colonne di N →







9900
9907



Progetto Architetture Avanzate A.A. 2017-2018

Parallelizzazione algoritmo All Pairs Shortest Path (Floyd-Warshall) in CUDA.

-) Codice sorgente sequenziale C/C++ consegnato dal docente (e-learning).
-) Benchmark di riferimento per i test consegnati da docente (e-learning).

Obiettivo 1:

✓ 1) Analisi codice sequenziale con profiler
✓ 2) Sviluppo kernel CUDA per esecuzione su GPU con ottimizzazione su
  ✓ a. Kernel configuration // provarne varie e tenere la migliore
  b. Brench divergence
  c. Memory coalescence
  d. Shared memory usage
  e. Load balancing
  f. Tecniche avanzate (ILP, etc.)

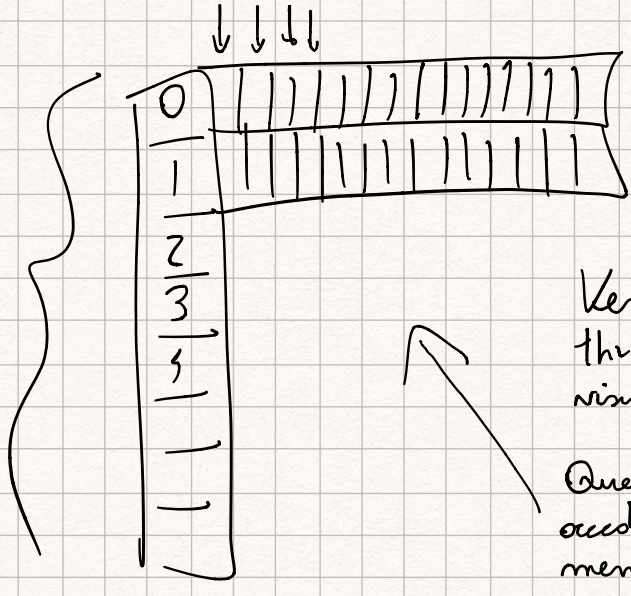
Risultati sperimentali:
Speedup rispetto sequenziale ottenuto con l'applicazione incrementale delle ottimizzazioni, con presentazione risultati profiler CUDA ad ogni step. Risultati (tempi) ottenuti su tutti i benchmark.

1) Dopo una prima analisi con Valgrind è possibile notare che il carico di lavoro maggiore si ha all'interno della funzione che implementa l'algoritmo Floyd-Warshall.
In particolare su tutti i benchmark il carico si concentra sulle istruzioni condizionali all'interno del ciclo più interno!
Dopo una fase di correttezza il codice è stato leggermente ottimizzato risparmiando N iterazioni per tutte le volte che la matrice[i][k] ha un valore pari ad INF; e lasciando invariato il grafo tutte le volte che i == j, ossia tutte le volte che si sta per computare il cammino di costo minimo da un nodo a se stesso.

2) La prima versione parallela del Floyd-Warshall vuole focalizzarsi sulla funzionalità dell'algoritmo, senza dare troppa importanza alle performance.

Provando questa versione Naive sui vari benchmark, si può già notare un buon miglioramento in termini di performance, in particolare si ha un miglior rendimento quando il Kernel presenta la seguente configurazione.

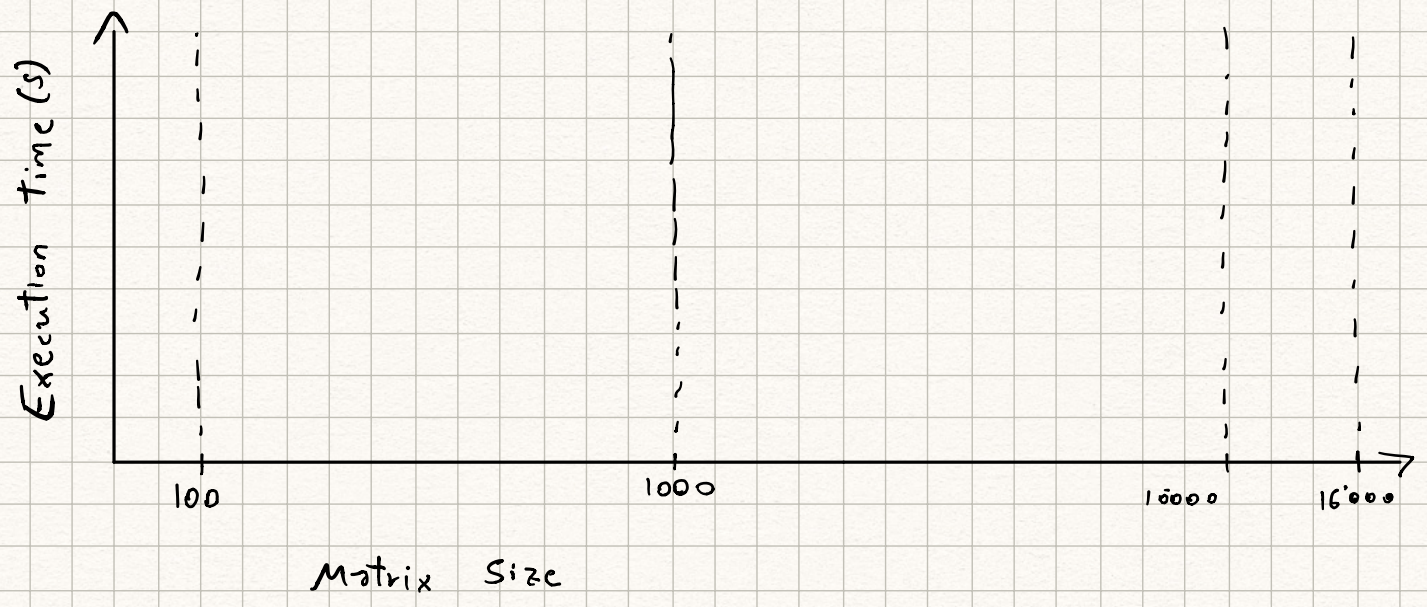
[MIGLIORE CONFIGURAZIONE KERNEL] 

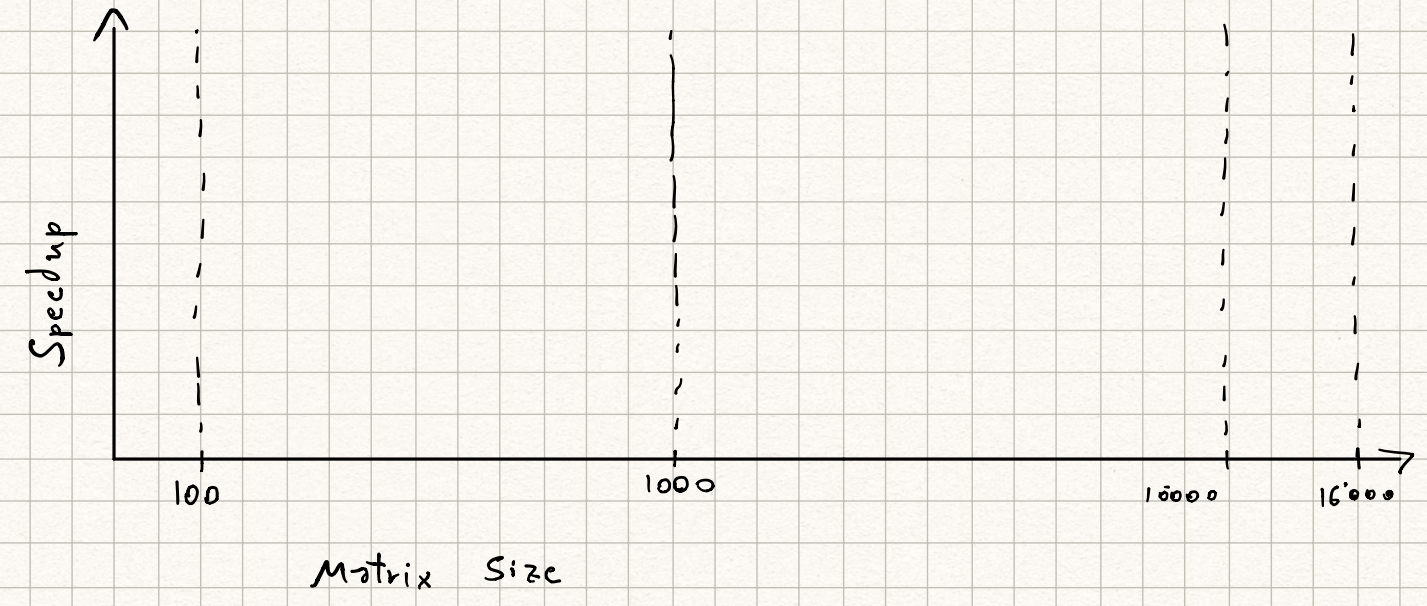


I PRIMI 32 WARPS FORMANO IL PRIMO BLOCCO

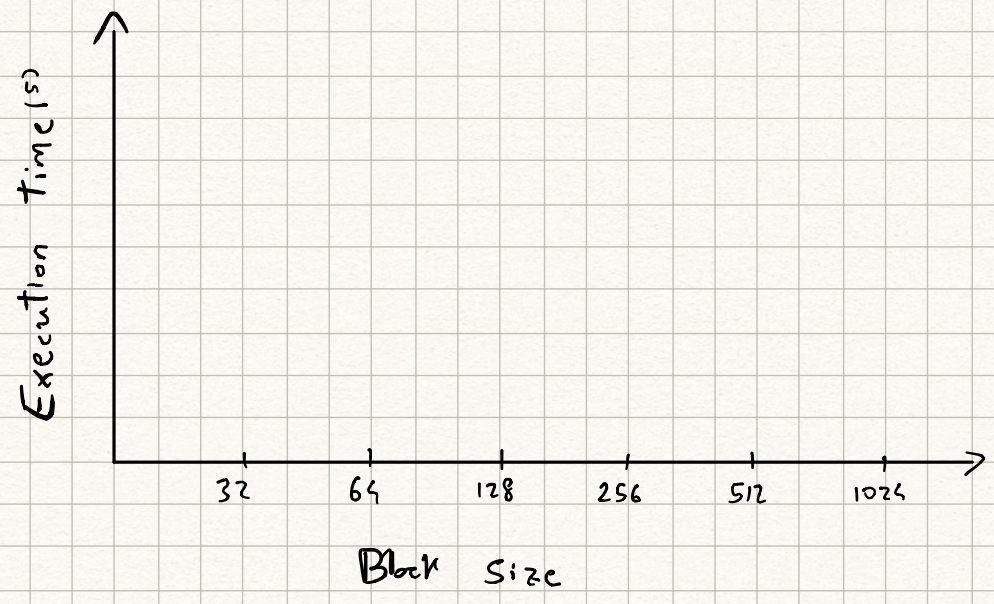
Verificare numero e quali thread sono attivi con visual studio!

Questo perché le thread accedono sequenzialmente in memoria, ma non è detto che vengano eseguite in maniera sequenziale (thread dello stesso warp)

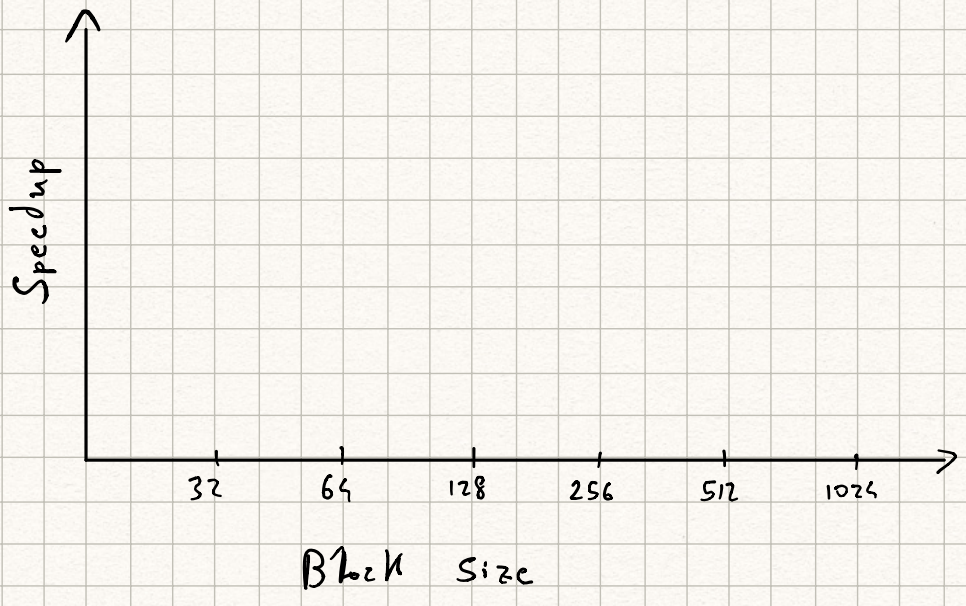




Questi 2 grafici sarebbe bene da mettere nei risultati finali.



// Questi grafici vanno bene da mettere nella sezione CUDA.
Prendiamo come riferimento il C-64 = 10.000 nodi



th	time	speed-up
32	74 ms	17x
64	52 ms	25x
128	52 ms	25x
256	53 ms	24x
512	53 ms	24x
1024	57 ms	23x

1000 nodi

$S = 10$	$10 + 1$	11
	$10 + 2$	12
	$10 + 3$	13
	$10 + 4$	14

$S = 10$	$S = 10 + 1$	11
	$S = 11 + 2$	13