vect_add OK
stencil_1D OK
matrix mul OK
matrix transpose OK

Stencil Shared memory

```
__ shared __ int Smem[1024];
int index = BlockDim.x * BlockId.x + threadId.x
if index < N :
    Smem[ threadId.x + Radius ] = input[ index + Radius ]
else
    Smem[ threadId.x + Radius ] = 0
__ Synch threads ()
if threadId.x < Radius
    Smem[ threadId.x ] = input[ index ]
if threadId.x ≥ blockDim.x - Radius
    Smem[ threadId.x + Radius*2 ] = input[ index + Radius*2 ]
__ synchthreads ()
int sum = 0
for int i = 0 ; i < Radius*2 ; i++
        sum += Smem[ threadId.x + i ]
output[index + Radius] = sum
if index ≤ N || index ≥ N - Radius
        output[index] = 0
```

Vec Add

```
int global_id = blockId.x * blockDim.x + threadId.x
if global_id < N
    output[global_id] = vec_A[global_id] + vec_B[global_id]
```

Stencil

```
int global_id = blockId.x * blockDim.x + threadId.x
if global_id >= Radius || global_id <= N - Radius
    int sum = 0
    for int i = -Radius, i < Radius; i++
        sum += input[global_id - i]
    output[global_id] = sum
```

Matrix Mul

```
int Row = BlockId.y * BlockDim.y + threadId.y
int Col = BlockId.x * BlockDim.x + threadId.x
int Pvalue = 0
for int K = 0; K < N; K++
    Pvalue += matrix_A[Row*N + K] * matrix_B[Col + K*N]
output[Row*N + Col] = Pvalue
```

matrix transpose

```
int Row = blockId.y * BlockDim.y + threadId.y
int Col = blockId.x * BlockDim.x + threadId.x
if Row < N && Col < N
    res[Row*N + Col] = input[Col*N + Row]
```

```
__shared__ int sMem[1024]

int index = BlockId.x * BlockDim.x + threadId.x

if index < N :
    sMem[threadId.x + Radius] = input[index + Radius]
else :
    sMem[threadId.x + Radius] = 0

__synchthreads()

if threadId.x < Radius :
    sMem[threadId.x] = input[index]

if threadId.x >= BlockDim.x - Radius
    sMem[threadId.x + 2* Radius] = input[index + 2* Radius]

__synchthreads()

int sum = 0

for int i = 0 ; i < Radius * 2 ; i++
    sum += smem[threadId.x + i]

output[ Index + Radius] = sum

if index > N - Radius ∧ index < N
    output[index] = 0
```

shm Matrix Mul

```
__shared__ int smemA[TILE][TILE]
__shared__ int smemB[TILE][TILE]
int bx = BlockId.x      int by = BlockId.y
int tx = ThreadId.x     int ty = ThreadId.y
int Row = by * TILE + ty
int Col = bx * TILE + tx
int Pvalue = 0
for int m = 0, m < width/tile, width++ {
    smemA[ty][tx] = inputA[Row*N + m*tile + tx]
    smemB[ty][tx] = inputB[Col + (m*tile+ty)*N]
    __syncthreads()
    for int k = 0, k < Tile, k++{
        Pvalue += smemA[ty][k] * smemB[k][tx]
    }
    __syncthreads()
}
output[Row*N + Col] = Pvalue
```

shm Matrix Transpose

```
__ shmem __ int smem [BlockDim] [Block Dim]
int Raw = Block Id.x * Block Dim.x + thread Id.x
int Col = Block Id.y * Block Dim.y + threadId.y
if Raw < N ∧ Col < N :
    smem [ threadId.y ] [thread Id.x] = input [ Raw * N + Col ]
    __ synchthreads ()
    Col = block Id.x * Block Dim.x + threadId.x
    Raw = block Id.y * Block Dim.y + thread Id.y
    d_matrix_out [Raw * N + Col] = smem [threadId.x ][thread Id.y ]
```

```
__shared__ int memA [TILE][TICE]

__shared__ int memB [TILE][TICE]

int ty = threadId.y ; int tx = threadId.x ;

int by = BlockId.y ; int bx = BlockId.x ;

int Col = bx * TILE .x + tx ;
int Raw = by * TICE + ty ;
int Pvalue = 0
for (int K=0 ; K < width / TILE ; K++) {

    smemA[ty][tx] = matA [Raw * N + K * TILE + tx]

    smemB[ty][tx] = matB [Col + (K*Tile + ty) * N]

    __syncthreads()

    for (int m=0 ; m < TILE ; m++) {

        Pvalue += smemA[ty][m] * smemB[m][tx]

    }

    __syncthreads()

}

output [Raw*N + Col] = Pvalue
```

```
__shared__ int smem[1024];
int global_id = BlockId.x * BlockDim.x + threadId.x
smem[threadId.x] = input[global_id]
__syncthreads()
for (int i = 1; i <= BlockDim.x; i*=2) {
        index = threadId.x * i*2

        if ( index < BlockDim.x ) {
                smem[threadId.x] += smem[threadId + i]
        }
        __syncthreads()
}
if (threadId.x == 0)
    output[BlockId.x] = smem[0]
}
```

Esercizio 2

Abbiamo da fare il confronto tra due architetture hardware:

- Shared memory: è un'architettura hardware che dispone di una memoria singola accessibile da tutti i processori qualunque sia la sua locazione fisica. I vantaggi della shared memory sono la rapidità di accesso ai dati vista la vicinanza di questa memoria ai processori e la semplicità di programmazione infatti tutto lo spazio degli indirizzi è unico e accessibile da qualunque processore.
  Gli svantaggi invece sono la scalabilità, infatti con l'aumentare del numero dei processori il traffico per accedere alla memoria aumenta. Inoltre possono peggiorare le performance per gestire le race condition ed è difficile per il programmatore che ne è responsabile mantenere un eventuale code coerency.

- Distributed memory: è un architettura hardware in cui ciascun processore dispone della propria memoria locale e può accedere a quelle degli altri processori comunicando con essi seguendo le varie opportunità messe a disposizione. I vantaggi sono la possibilità di scalare in maniera semplice poiché basta aggiungere un processore con la sua memoria. Non è inoltre necessario gestire la code coerency in quanto ogni processore è indipendente
  Gli svantaggi sono la difficoltà di trasferire queste strutture in strutture adatte alla shared memory. Inoltre il programmatore ha tutte le responsabilità di gestire la comunicazione tra i vari processori per raggiungere le altre memorie

Esercizio 3

$$speedup = \frac{1}{(1-x) + \frac{x}{100}}$$

$$80 = \frac{1}{(1-x) + \frac{x}{100}}$$

$$80\left((1-x) + \frac{x}{100}\right) = 1$$

$$80 - 80x + \frac{80}{100}x = 1$$

$$\frac{396}{5}x = 79$$

$$x = 0,997$$

99.7 % parallelizzabile

0.3 % per rimanere sequenziale

```
__shared__ int smem[1024]

int ind = BlockId.x * BlockDim.x + threadId.x

if (ind < BlockId.x - 1) {
    smem[threadId.x + 1] = input[ind]
} else
    smem[0] = 0
}

__syncthreads()

for (offset = 1; offset < BlockDim.x; offset *= 2) {
    if (threadId.x >= offset) {
        LeftValue = smem[threadId.x - offset]
        RightValue = smem[threadId.x]
    }
    __syncthreads()
    if (threadId.x >= offset) {
        smem[threadId.x] = LeftValue + RightValue
    __syncthreads
}

output[ind] = smem[threadId.x
```

```
__shared__ int smem[1024];
int ind = BlockId.x * BlockDim.x + threadId.x
smem[threadId.x] = input[ind]
__syncthreads()
for (int i = 1; i < BlockDim.x; i *= 2){
    int offset = threadId.x * (i * 2)
    if (offset < BlockDim.x )
        smem[offset] += smem[offset + i]
    __syncthreads()
}
if (threadId.x == 0)
    output[BlockId.x] = smem[0]
```

```
__shared__ int smem[1024];

int ind = blockIdx.x * blockDim.x + threadIdx.x;

if (threadIdx.x < blockDim.x - 1)
    smem[threadIdx.x+1] = input[ind];

else
    smem[0] = 0;

__syncthreads();

for (int offset = 1; offset <= blockDim.x; offset *= 2) {

        int leftValue; rightValue;

        if (threadIdx.x >= offset)

            leftValue = smem[threadIdx.x - offset];
            rightValue = smem[threadIdx.x];

        __syncthreads();

        if (threadIdx.x >= offset)

            smem[threadIdx.x] = leftValue + rightValue;

        __syncthreads;

}

output[ind] = smem[threadIdx.x];
```

```
__shared__ int smem [1024]

int ind = BlockId.x * BlockDim.x + threadId.x

if threadId.x < BlockDim.x - 1

    smem [threadId.x + 1] = input [index]

else

    smem [0] = 0

__syncthreads()

for (int offset = 1 ; offset <= BlockDim.x ; offset *= 2) {

    int LeftValue ; int RightValue

    if (threadId.x >= offset)

        LeftValue = smem [threadId.x - offset]
        RightValue = smem [threadId.x]

    __syncthreads()

    if (threadId.x >= offset)

        smem [threadId.x] = LeftValue + RightValue

    __syncthreads()

}

output [ind] = smem [threadId.x]
```

(3) $$\text{Speedup} = \frac{1}{1 - \text{Frac}_{\text{parallel}} + \frac{\text{Frac}_{\text{parallel}}}{\text{Speedup}_{\text{parallel}}}}$$

$$80 = \frac{1}{(1-x) + \frac{x}{100}}$$

$$80 - 80x + \frac{80}{100}x = 1$$

$$79 = 80x - \frac{80}{100}$$

$$79 = \frac{8000 - 80x}{100}$$

$$79 = \frac{7920}{100}x$$

$$x = 0{,}997$$

Il 99.7% del codice deve essere parallelo
Lo 0.3% del codice deve essere sequenziale

(4)

1024 thread → 32 warp

1° iterazione → 512 thread lavorano
2° iterazione → 256 thread ''
3° '' → 128 '' ''
4° '' → 64 '' ''
5° '' → 32 '' ''

Alla 5° iterazione lavora solo un warp di thread
Le altre 31 sono ferme

(10)

Senza shared

$$\frac{\#\ \text{computation}}{\#\ \text{comunication}} = \frac{1}{2} = \frac{1}{2}$$

Con shared

$$\frac{\#\ \text{computation}}{\#\ \text{comunication}} = \frac{\text{nthread} * 1}{\text{nthread} * 2} = \frac{1}{2}$$

Senza shared

$$\frac{\#\ \text{computation}}{\#\ \text{comunicafich}} = \frac{2}{2} = 1$$

Con shared

$$\frac{\#\ \text{computation}}{\#\ \text{comunication}} = \frac{256 * (2 * 16)}{256 * 2} = 16$$

↳ riduzione di bandwith 16

① $\frac{4,90}{1,43} = 3.42$

② $\frac{5,69}{1,35} = 4,21$

③ $\frac{1,43}{0,695} = 2,06$ $\qquad \frac{1,35}{0,96} = 1,40$

$$\frac{1}{1 - 0,4 + \frac{0,4}{10}} = 1,56 \text{ speedup}$$

$$\frac{1}{0,5 + \frac{0,5}{5}} = 1,66 \quad \text{speedup} = \frac{5}{3} \text{ speedup}$$

$$\text{cost} = \frac{5}{3} + \frac{2}{3} = \frac{7}{3} \text{ costo}$$

Ho più costo che speedup introdotto, non ne vale la pena

$$\frac{1}{1 - x + \frac{x}{100}} = 80$$

$$80 - 80x + \frac{80}{100}x = 1$$

$$\frac{8000 - 80}{100}x = 79$$

$$x = \frac{79}{79{,}2} = 0{,}997$$

$$x = 0{,}03$$

```
__shared__ int smem[1024]

int index = BlockId.x * BlockDim.x + threadId.x

if (threadId.x < BlockDim.x - 1)

    smem[threadId.x + 1] = input[index]

else

    smem[0] = 0

__syncthreads()

for (int offset = 1; offset < BlockDim.x; offset *= 2) {

    int leftvalue, Rightvalue;

    if (threadId.x >= offset )

        leftvalue = smem[threadId.x - offset]
        Rightvalue = smem[threadId.x]

    __syncthreads()

    if (threadId.x >= offset)

        smem[threadId.x] = leftvalue + Rightvalue

    __syncthreads()

}

output[index] = smem[threadId.x]
```

```
__shared__ int smem[1024]

int ind = BlockId.x * BlockDim.x + threadId.x

smem[threadId.x] = input[ind]

__syncthreads()

int step=1

for (int limit = BlockDim.x/2; limit>0; limit /=2) {

    if (threadId.x < limit) {

        RightValue = (threadId.x + 1) * (step*2) - 1
        LeftValue = RightValue - step
        smem[RightValue] += smem[LeftValue]
    }

    step *=2
    __syncthreads()

}

if (threadId.x == 0)

    smem[BlockDim.x - 1] = 0

__syncthreads()

step= BlockDim.x/2

for (int limit = 1; limit < BlockDim.x; limit*=2) {

    RightValue = (threads+1)* (step*2) - 1
    LeftValue = RightValue - step

    int tmp = smem[LeftValue]

    smem[LeftValue] = smem[RightValue]

    smem[RightValue] += tmp

  }
  step /= 2

  __syncthreads()

}

output[ind] = smem[threadId.x]
```

```
__shared__ int smem[2048];
int index = blockIdx.x * blockDim.x + threadIdx.x;
smem[threadIdx.x] = input[index];
__syncthreads();
int step = 1;
for (int limit = blockIdx.x/2; limit > 0; limit /= 2) {
    if (threadIdx.x < limit)
        RightValue = (threadIdx.x + 1) * (step*2) - 1;
        LeftValue = RightValue - step;
        smem[RightValue] += smem[LeftValue];
    step *= 2;
    __syncthreads();
}
if (threadIdx.x == 0)
    smem[blockDim.x - 1] = 0;
__syncthreads();
int step = blockDim/2;
for (int limit = 1; limit <= blockDim.x/2; limit *= 2) {
    if (threadIdx.x < limit)
        RightValue = (threadIdx.x + 1) * (i*2) - 1;
        LeftValue = RightValue - step;
        int tmp = smem[LeftValue];
        smem[LeftValue] = smem[RightValue];
        smem[RightValue] += tmp;
    step /= 2;
    __syncthreads();
}
output[index] = input[threadIdx.x];
```

```
__shared__ int smem[1024]

int index = BlockId.x * BlockDim.x + threadId.x

smem[threadId.x] = input[index]

__syncthreads()

int step = 1

for (int limit = BlockDim.x / 2 ; limit > 0 ; limit /= 2) {

    if (threadId.x < limit)

        RightValue = (threadId.x + 1) * (i * 2) - 1

        LeftValue = RightValue - step

        smem[RightValue] += smem[LeftValue]

    step *= 2
    __syncthreads()
}

if (threadId.x == 0)

    smem[BlockId.x - 1] = 0

__syncthreads

int step = BlockId.x / 2

for (int limit = 1 ; limit <= BlockDim.x ; limit *= 2) {

        if (threadId.x < limit)

            RightValue = (threadId.x + 1) * (i * 2) - 1

            LeftValue = RightValue - step

            int tmp = smem[LeftValue]

            smem[LeftValue] = smem[RightValue]

            smem[RightValue] += tmp

        step /= 2
        __syncthreads()
}
output[index] = smem[threadId.x]
```

```
__shared__ int smem[1024]

int index = BlockId.x * BlockDim.x + threadId.x

smem[threadId.x] = input[index]

__syncthreads()

int step = 1

for (int limit = BlockId.x /2 ; limit > 0; limit /= 2) {

        if (threadId.x < limit)

            int RightValue = (threadId.x +1) *(i*2) - 1
            int LeftValue = RightValue - step
            smem[RightValue] += smem[LeftValue]

        step *= 2
        __syncthreads()

}

if (threadId.x == 0)

    smem[BlockDim.x - 1] = 0

__syncthreads

int step = BlockDim.x /2

for (int limit = 1 ; limit <= BlockDim.x /2 ; limit *= 2)

        if (limit < step)

            int RightValue = (threadId.x +1) * (i*2) - 1

            int LeftValue = RightValue - step

            int tmp = smem[LeftValue]

            smem[LeftValue] = smem[RightValue]

            smem[RightValue] += tmp

        step /2
        __syncthreads()
}
output[index] = smem[threadId.x]
```

$$y = \frac{1}{(1-x)+\frac{x}{20}}$$

$$2 = \frac{1}{(1-x)+\frac{x}{20}}$$

$$y = \frac{1}{(1-x)+\frac{x}{20}}$$

$$2 - 2x + \frac{2x}{20} = 1$$

$$y = \frac{1}{(1-x)+\frac{x}{20}} = \frac{1}{1-\frac{19}{20}x}$$

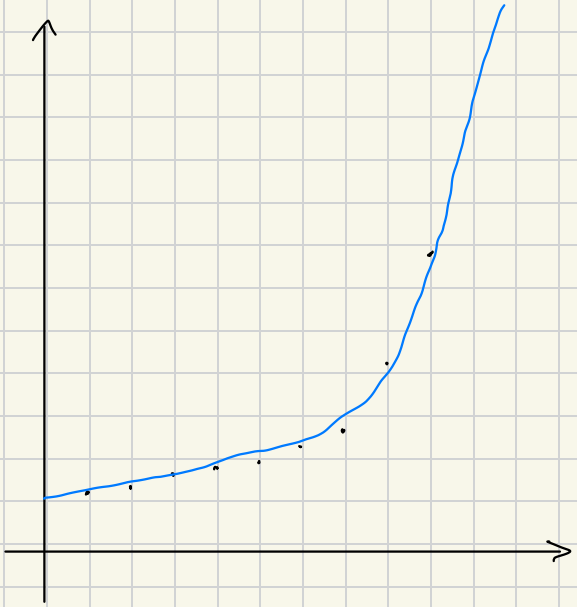
$$1 = 2x - \frac{2x}{20}$$

$$1 = \frac{40-2x}{20}$$

$$1 = \frac{38}{20}x$$

$$x = \frac{20}{38}$$

$$x = 0{,}52$$



$$10 = \frac{1}{(1-x)+\frac{x}{20}}$$

$$1 = 10 - 10x + \frac{x}{2}$$

$$9 = \frac{19}{2}x$$

$$x = \frac{9 \cdot 2}{19} = 0{,}95$$

$$x = \frac{1}{(1-0{,}7)+\frac{0{,}7}{40}}$$

$$x = \frac{1}{0{,}3+\frac{0{,}7}{40}} = 3.14$$

$$3.14 = \frac{1}{(1-x)+\frac{x}{20}}$$

$$3.14 - 3.14x + \frac{3.14}{20}x = 1$$

$$2.14 = 2.98x$$

$$x = 0{,}718$$

$$\frac{1}{1-0,9+\frac{0,9}{5}} = 3,57$$

$$\frac{1}{1-1+\frac{1}{1}} = 1$$ Speedup 3,57

$CPI_{orig} = 0,25 * 4 + 0,75 * 1,33 = 1 + 1 = 2$

$CPI_{mod_1} = 2 - 0,02\,(20 - 2) = 1,64$

$CPI_{mod_2} = 2 - 0,25\,(4 - 2,5) = 1,625$

↑ meglio

$0,43 * 1 + 0,21 * 2 + 0,12 * 2 + 0,24 * 2 = 1,57$

$$\frac{(0,43 - (0,43 - 0,25)) * 1 + (0,21 - (0,43 * 0,25)) * 2 + (0,43 * 0,25) * 2 + 0,12 * 2 + 0,24 * 3}{1 - (0,43 * 0,25)}$$

$= 1,908$

$CPI_{orig} = 1 * 0,3 + 1 * 0,7 = 1$

$CPI_{neu} = 1,05 * 0.1 + 1 * 0.7 = 0,805$

↳ optim. Zeitraum

```
__shared__ int smem [1024]

int index = BlockId.x * BlockDim.x + threadId.x

smem [threadId.x] = input [index]

__synchthreads ()

int step = 1

for (int limit = BlockDim.x /2 ; limit > 0 ; limit /= 2) {

        if (threadId.x < limit)

            int Rightvalue = (threadId.x + 1) * (step * 2) - 1
            int Leftvalue = Rightvalue - step
            smem [Rightvalue] += smem [Leftvalue]

        step *= 2
        __synchthreads()

}

if threadId.x == 0

    smem [BlockDim.x - 1] = 0

__synchthreads()

int step = BlockDim.x /2

for (int limit = 1 ; limit ≤ BlockDim.x ; limit *= 2)

    if (threadId.x < limit)

        int Rightvalue = (threadId.x + 1) * (step * 2) - 1
        int Leftvalue = Rightvalue - step
        int tmp = smem [Leftvalue]
        smem [Leftvalue] = smem [Rightvalue]
        smem [Rightvalue] += tmp

    step /= 2
    __synchthread ()

}

output [index] = smem [threadId.x]
```

$$\frac{\#\ computation}{\#\ comunication} = \frac{36}{7}$$

$GFLOPS = \frac{100}{4} \cdot \frac{36}{7} = 128.5\ Gflops < 200\ GFlops$ di picco ← memory bound, la computazione è limitata dai trasferimenti di memoria

$GFLOPS = \frac{2500}{4} \cdot \frac{36}{7} = 3214\ Gflops > 300\ GFlops$ di picco ← compute bound la computazione è limitata dalla potenza del Kernel

$$\frac{109\,970\,178}{0{,}094 \cdot 10^9} = 1{,}17\ GFLOPS$$

3 $$80 = \frac{1}{1 - x + \frac{x}{100}}$$

$$80 - 80x + \frac{80}{100}x = 1$$

$$79 = \frac{7920}{100}x$$

$$x = \frac{79}{79{,}2} = 0{,}997$$

$x = 99{,}7\%$ codice necessariamente parallelo

codice sequenziale $0{,}3\%$

4

$CPI_{avg} = 0{,}25 * 4 + 0{,}75 * 1{,}33 \approx 2$

$CPI_1 = 2 - 0{,}02\,(20 - 2) = 1{,}64$

$CPI_2 = 2 - 0{,}25\,(4 - 2{,}5) = 1{,}625$

meglio secondo ho CPI più basso

Usando l' algoritmo di reduce assumendo che abbia la miglior implementazione per evitare la divergenza in un warp ho:

1024 thread all'inizio di cui solo 512 superano il controllo

step 2 512 ⟶ 256

256 → 128

128 → 64

64 → 32

perciò alla 5 iterazione avrò un warp attivo e 31 warp inattivi

```
__shared__ int smem[1024]

int index = BlockId.x * BlockDim.x + threadId.x

smem[threadId.x] = input[index]

__syncthreads()

int step = 1

for (int limit = BlockDim.x / 2 ; limit > 0 ; limit /= 2) {

    if ( threadId.x < limit ) {

        int RightValue = (threadId.x + 1) * (step * 2) - 1
        int LeftValue = RightValue - step
        smem[RightValue] += smem[RightValue]
    }
    step *= 2
    __syncthreads()
}

if (threadId.x == 0)

    smem[BlockDim.x - 1] = 0

__syncthreads()

int step = BlockDim.x / 2

for (int limit = 1 ; limit < BlockDim.x / 2 ; limit *= 2) {

    if (threadId.x < limit) {

        int RightValue = (threadId.x + 1) * (step*2) - 1
        int LeftValue = RightValue - step
        int tmp = smem[LeftValue]
        smem[LeftValue] = smem[RightValue]
        smem[RightValue] += tmp
    }

    step /= 2
    __syncthreads()

}

output[index] = smem[threadId.x]
```

3.

$$80 = \frac{1}{(1-x) + \frac{x}{100}}$$

$$80 - 80x + \frac{80}{100}x = 1$$

$$\frac{7920}{100}x = 79$$

$$x = \frac{79}{79{,}2}$$

$x = 0{,}997$ codice parallelo

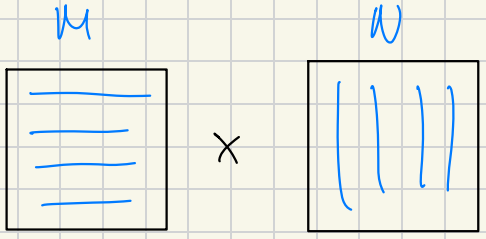
0,3 % codice che può rimanere seq

4

$CPI_{orig} = 0{,}25 * 4 + 0{,}75 * 1{,}33 = 1 + 1 = 2$

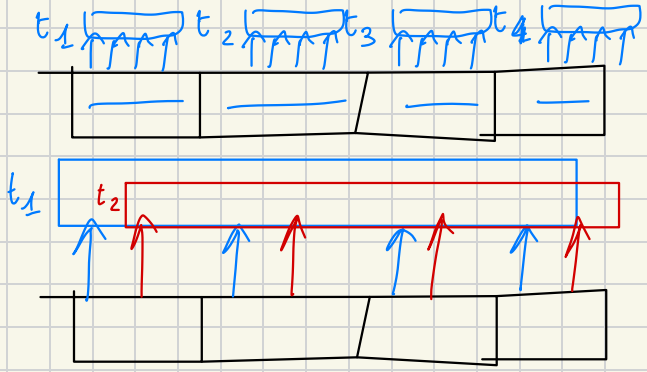
$CPI_{1^\circ mod} = 2 - 0{,}02(20 - 2) = 1{,}64$

$CPI_{2^\circ mod} = 2 - 0{,}25(4 - 2{,}5) = 1{,}625$ ← è la soluzione migliore



↓

Row Major layout



(8) Utilizziamo un algoritmo in cui la warp divergence è ridotta al minimo

1024 di cui 512 buoano alla prima iterazione

2° 256 " " " "

3° 128 " " " "

4° 64 " " " "

5° 32 " " " "

1 warp attiva alla 5° iterazione

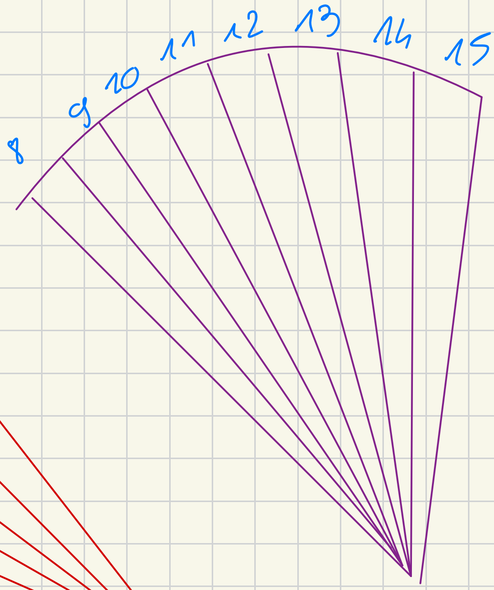
31 inattive

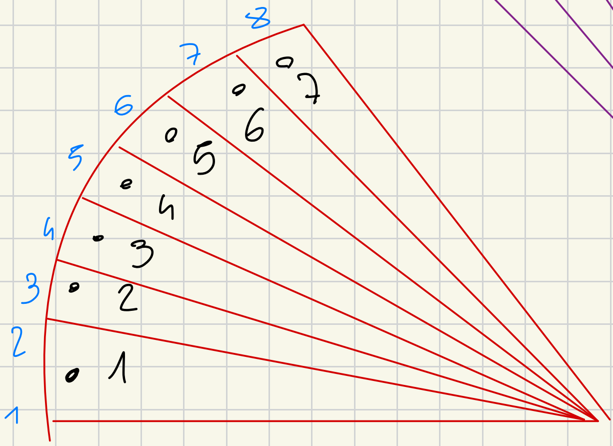
(9)

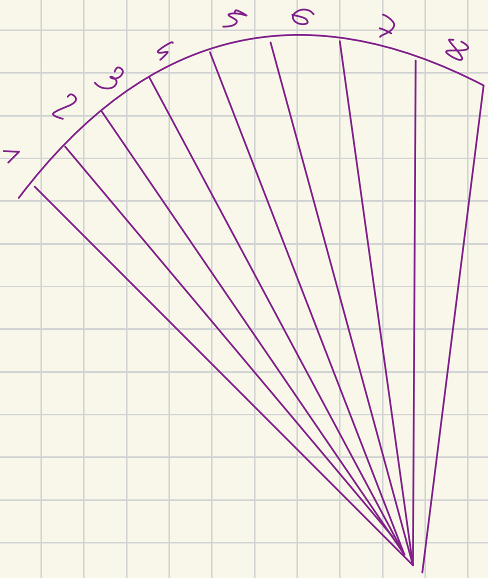
$$\frac{\#\ \text{computation}}{\#\ \text{comunication}} = \frac{36}{7}$$

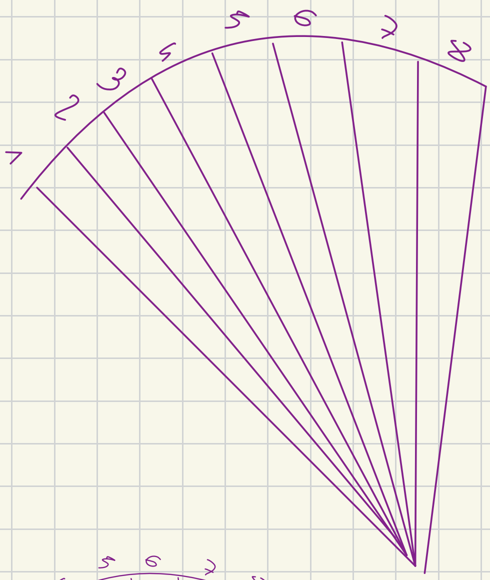
$GFLOPS = \frac{100}{4} \cdot \frac{36}{7} = 128{,}5 < 200$ GFLOPS di picco perciò la computazione è memory bound perché limitata dai trasferimenti da memoria

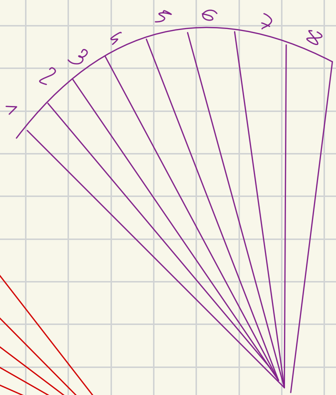
$GFLOPS = \frac{2500}{4} \cdot \frac{36}{7} = 3214 > 300$ Gflops di picco perciò la computazione è kernel bound perché limitata dalla capacità di computazione della GPU

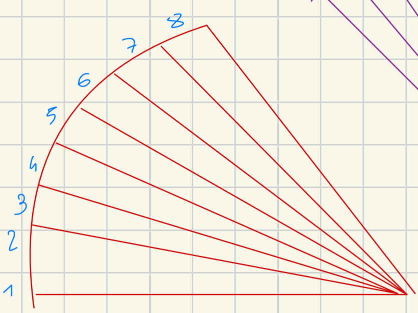

8 9 10 11 12 13 14 15

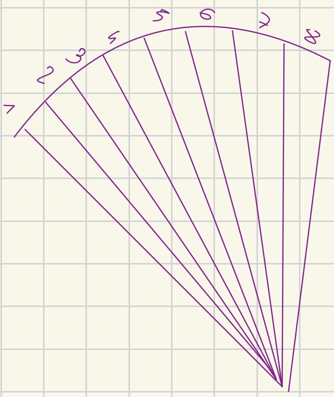

1 2 3 4 5 6 7 8
1 2 3 4 5 6 7

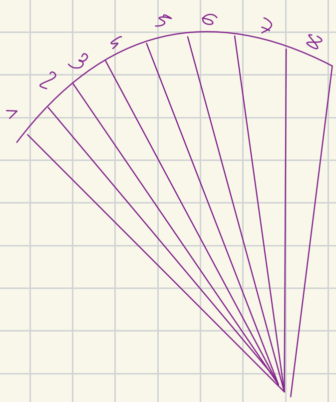

1 2 3 4 5 6 7 8


1 2 3 4 5 6 7 8


1 2 3 4 5 6 7 8


1 2 3 4 5 6 7 8


1 2 3 4 5 6 7 8


1 2 3 4 5 6 7 8

3 Mat mul ✓

8 Reduce Norm ✓

10 Prefix Naive ✓

9 Reduce No branch div ✓

5 Stencil 1D Shmem ✓

7 Transp Shmem ✓

2 Stencil ✓

11 Prefix Up Down ✓

1 Somma ✓

4 Mat Transp ✓

6 Mat Shmem

```
__shared__ int smemA[TILE][TILE]
__shared__ int smemB[TILE][TILE]

int ty = threadIdx.y ; int tx = threadIdx.x
int bx = BlockIdx.x ; int by = BlockIdx.y

int Row = by * TILE + ty
int Col = bx * TILE + tx
int Pvalue = 0
for (int m = 0 ; m < Width/TILE ; m++)

    smemA [ty] [tx] = input [Row*N + m*TILE+tx]
    smemB [ty] [tx] = input [Col + (m*TILE+Ty)*N]

    __syncthreads()

    for (int i = 0 ; i < TILE ; i++ )

        Pvalue += smemA [ty] [i] * smemB[i][tx]

    __syncthreads()

}
output [Row*N+Col] = Pvalue
```

a) 8, 9, 10, 11, 12, 13, 14, 15
16, 17, 18, 19, 20, 21, 22, 23

8 72
16 80
24 88
32 ⋮
40 ⋮
48 ⋮
56 ⋮
64

0
64
128
⋮

1
2
3
4
5
6
7
8

$$\frac{\#\ computation}{\#\ comunication} = \frac{2}{2} = 1$$

$$\frac{\#\ computation}{\#\ comunication} = \frac{\#\ num\ thread\ (16*2)}{\#\ numthreads * 2}$$

→ ogni thread fa $16*2$ operazioni

→ ogni thread fa 2 operazioni di comunicazione

$$= 16$$

Esempio 0 slide

a) $\frac{4.90}{1.43} = 3.42$

b) $\frac{5,69}{1,35} = 4.21$

c) $\frac{1,43}{0,695} = 2,05$ $\qquad \frac{1,35}{0,96} = 1,40$

1

$$Speedup = \frac{1}{1-0,4 + \frac{0,4}{10}} = 1,56$$

2

$$speedup = \frac{1}{0,5 + \frac{0,5}{5}} = 1,66 = \frac{5}{3}$$

$$costo = 5 \cdot \frac{1}{3} + \frac{2}{3} = \frac{7}{3} = 2.33$$

non ne vale la pena, il costo è maggiore della speedup

3

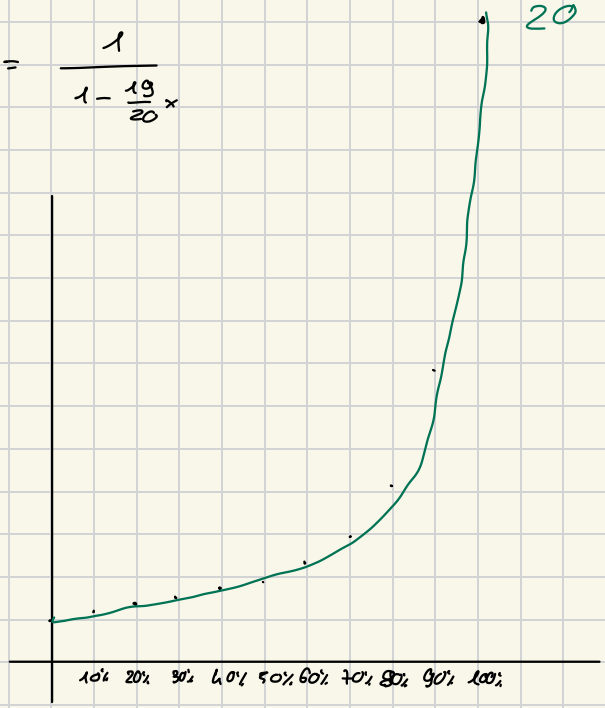
$$80 = \frac{1}{(1-x) + \frac{x}{100}}$$

$$80 - 80x + \frac{80x}{100} = 1$$

$$\frac{7920}{100} x = 79$$

$$79,2\, x = 79 \qquad x = 0,997 \qquad 99.7\%$$ deve essere parallelo, 0,3% sequenziale

(4) $Speedup = \frac{1}{(1-x) + \frac{x}{20}} = \frac{1}{\frac{20 - 20x + x}{20}} = \frac{1}{1 - \frac{19}{20}x}$



$2 = \frac{1}{1 - \frac{19}{20}x}$

$2 - \frac{19}{10}x = 1$

$\frac{19}{10}x = 1$

$x = \frac{10}{19} = 0,526$

$10 = \frac{1}{1 - \frac{19}{20}x} = 10 - \frac{19}{2}x = 1$

$9 = \frac{19}{2}x$ $\quad x = 0,95$

↳ 95% deve essere parallelo

4bis

$\frac{1}{0,3 + \frac{0,7}{40}} = 3,149$

$3,149 = \frac{1}{1 - \frac{19}{20}x}$

$3,149 - 3,149 \cdot \frac{19}{20}x = 1$

$2,99x = 2,149$

$x = \frac{2,149}{2,99} = 0,718$

71,8 % di percentuale di parallelismo

1,8 % in più non manca praticamente nulla

$$\frac{1}{0,1 + \frac{0,9}{5}} = 3.57$$

Non cache = 1

Ho uno speed up di 3.57

$CPU_{orig} = 0,25 * 4 + 0,75 * 1,33 = 2$

$CPI_{mod1} = 2 - 0,02 (20 - 2) = 1,64$

$CPI_{mod2} = 2 - 0,25 (4 - 2,5) = 1,625$

La seconda è meglio perché il CPI è più basso e quindi ci vogliono meno cicli per fare un istruzione

$0,43 + 0,42 + 0,24 + 0,48 = 1,57$

$$\frac{(0,43 - (0,43 * 0,25)) * 1 + (0,21 - (0,43 * 0,25)) * 2 + (0,43 * 0,25) * 2 + (0,12) * 2}{\frac{0,24 * 3}{(1 - 0,43 * 0,25)}}$$

$$\frac{0,13225 + 0,1205 + 0,215 * 0,24 + 0,72}{0,8925} = 1,907$$

No il CPI è aumentato quindi modificare non comporta uno speedup anzi peggiora le performances

$0,3 * 1 + 0,7 * 1 = 1$

$1,05 * 0,2 + 0,7 * 1 = 0,91$

Il secondo richiede meno CPI per fare un task quindi è meglio

$$\frac{10^9 \cdot 930178}{0,094 \cdot 10^9} = 1,17 \text{ GFLOPS}$$

```
__shared__ int smemA[TILE][TILE]
__shared__ int smemB[TILE][TILE]

int ty = threadIdx.y; int tx = threadIdx.x
int by = BlockIdx.y; int bx = BlockIdx.x

int Row = by * TILE + ty
int Col = bx * TILE + tx

int Pvalue = 0

for (int m=0; m < Width/TILE; m++) {

    smemA[ty][tx] = input[Row * N + m*TILE + tx]
    smemB[ty][tx] = input[Col + (m*TILE+ty)*N]

    __synchthreads()

    for (int i=0; i < TILE; i++)

        Pvalue += smemA[ty][i] * smemB[i][ty]

    __synchthreads()
}

output[Row * N + Col] = Pvalue
```

- 3 Mat mul ✓
- 8 Reduce Norm ✓
- 10 Prefix Naive ✓
- 9 Reduce No branch div ✓
- 5 Stencil 1D Shmem ✓
- 7 Transp Shmem ✓
- 2 Stencil ✓
- 11 Prefix Up Down ✓
- 1 Somma ✓
- 4 Mat Transp ✓
- 6 Mat Shmem