

Παράλληλα Συστήματα

Χειμερινό εξάμηνο 2024–2025

CUDA #3



Παράδειγμα: vecadd1.cu

- ▶ Size of vectors (n) = 1.000.000
- ▶ blockSize = 1024
- ▶ gridSize =
(int)ceil((float)n/blockSize);
- ▶ Event recording???
- ▶ vecAdd<<<gridSize, blockSize>>>

CUDA kernel launch with 977 blocks of 1024 threads
final result: 1.000000
Time for the kernel: 0.341376 ms


```
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}
```

Απόδοση & χρόνος

- ▶ Πως καταλαβαίνουμε πως μια οποιαδήποτε αλλαγή στον κώδικά μας επηρεάζει την απόδοση του προγράμματος;
 - Ποια έκδοση τρέχει πιο γρήγορα;
- ▶ Χρήση των timers του λειτουργικού συστήματος
 - Λανθάνων χρόνος (latency) και διακύμανση (variation) από διάφορες πηγές όπως πχ. Χρονοπρογραμματισμός νημάτων από το λειτουργικό σύστημα, ύπαρξη CPU timers υψηλής ακρίβειας, κλπ)
 - Όσο τρέχει ο πυρήνας της GPU, ενδέχεται να γίνονται ασύγχρονα υπολογισμοί στον host
- ▶ Ο μόνος τρόπος να μετρήσουμε αυτούς τους υπολογισμούς στον host είναι χρησιμοποιώντας κάποιον μηχανισμό συγχρονισμού της CPU ή του λειτουργικού συστήματος.
 - Έτσι, για τη μέτρηση του χρόνου που δαπανά η GPU πάνω σε μια εργασία, θα πρέπει να χρησιμοποιήσουμε το CUDA event API.

Γεγονότα (events)

- ▶ Ένα γεγονός στην CUDA είναι στην πράξη μια χρονοσφραγίδα της GPU (GPU time stamp) που καταγράφεται σε μια προκαθορισμένη χρονική στιγμή
 - ▶ Επειδή η GPU καταγράφει τη σφραγίδα, παρακάμπτει πολλά προβλήματα που θα αντιμετωπίζαμε αν χρησιμοποιούσαμε CPU timers
 - ▶ Σχετικά εύκολη διαδικασία
- 

Διαδικασία καταγραφής [1]

- ▶ Δημιουργία γεγονότων αρχής και τέλους
 - `cudaEvent_t start, stop;`
 - `cudaEventCreate(&start);`
 - `cudaEventCreate(&stop);`
- ▶ Εκκίνηση καταγραφής αρχής
 - `cudaEventRecord(start, 0);`
- ▶ >>> Γεγονός προς καταγραφή <<<
- ▶ Εκκίνηση καταγραφής τέλους
 - `cudaEventRecord(stop, 0);`

Διαδικασία καταγραφής [2]

- ▶ Υπολογισμός χρονικής διάρκειας μεταξύ δύο γεγονότων
 - `cudaEventElapsedTime(&elapsedTime,start,stop);`
- ▶ Χρήση (πχ. εκτύπωση) του χρόνου αυτού
 - `printf ("Time for the kernel: %f ms\n", elapsedTime);`
- ▶ Απελευθέρωση δεσμευμένης μνήμης
 - `cudaEventDestroy(start)`
 - `cudaEventDestroy(stop)`

Ανάγκη συγχρονισμού

- ▶ Πρόβλημα! Κάποιες κλήσεις στην CUDA C είναι ασύγχρονες
 - Η GPU ξεκινά να εκτελεί τον κώδικά μας, αλλά η CPU συνεχίζει εκτελώντας την επόμενη γραμμή του προγράμματός μας πριν προλάβει να τελειώσει η GPU
- ▶ Μια κλήση `cudaEventRecord()` σημαίνει ότι μπαίνουν εντολές στην ουρά εργασιών προς εκτέλεση της GPU για καταγραφή του τρέχοντα χρόνου
 - Αυτό έχει σαν αποτέλεσμα ότι το γεγονός μας δεν θα καταγραφεί μέχρι η GPU να τελειώσει με όλες τις δουλειές της πριν την κλήση προς το `cudaEventRecord()`.
 - Για την περίπτωση της μέτρησης του σωστού χρόνου για το stop event, αυτό είναι ακριβώς αυτό που θέλουμε
 - Αλλά δεν μπορούμε με ασφάλεια να διαβάσουμε την τιμή του stop event μέχρι η GPU να έχει ολοκληρώσει την πρότερη εργασία της και να έχει καταγράψει το stop event.
- ▶ Ευτυχώς υπάρχει η δυνατότητα να ζητήσουμε από την CPU να συγχρονιστεί με το γεγονός χρησιμοποιώντας την κλήση `cudaEventSynchronize()`
 - `cudaEventSynchronize(stop);`

Παράδειγμα: vecadd2.cu

- ▶ Size of vectors (n) = 1.000.000
- ▶ blockSize = 1024
- ▶ gridSize = 10
- ▶ Event recording: Ακριβώς πριν και μετά το vecadd

CUDA kernel launch with 10 blocks of 1024 threads

final result: 1.000000

Time for the kernel: 0.336864 ms

```
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int k, id = blockIdx.x*blockDim.x+threadIdx.x;

    // Make sure we do not go out of bounds
    for (k = id; k < n; k += blockDim.x*gridDim.x) {
        c[k] = a[k] + b[k];
    }
}
```


Παράδειγμα: vecadd3.cu

- ▶ Ίδιο με το vecadd2.cu εκτός από τις θέσεις καταγραφής
 - `cudaEventRecord(start,0)` πριν την αντιγραφή των host vectors στην device
 - `cudaEventSynchronize(stop)` μετά την αντιγραφή του array πίσω στον host

CUDA kernel launch with 10 blocks of 1024 threads
final result: 1.000000
Time for the kernel: 6.608128 ms

Παράδειγμα: vecadd4.cu

- ▶ Σχεδόν ίδιο με vecadd2.cu και vecadd3.cu
- ▶ Διαφορές με vecadd2.cu
 - Αντί για πίνακα 1.000.000 στοιχείων έχουμε μόνο 50.000
- ▶ Διαφορές με vecadd3.cu
 - Αντί για πίνακα 1.000.000 στοιχείων έχουμε μόνο 50.000
 - Οι θέσεις καταγραφής είναι ακριβώς πριν και μετά την κλήση πυρήνα vecAdd

CUDA kernel launch with 10 blocks of 1024 threads
final result: 1.000000
Time for the kernel: 0.014816 ms

Παράδειγμα: vecadd5.cu

- ▶ Σχεδόν ίδιο με το vecadd4.cu
 - Αντί για vector 50.000 στοιχείων έχουμε 1.024.000
 - Διαφορές στην vecAdd

```
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int t, k, id = blockIdx.x*blockDim.x+threadIdx.x;
    t=blockDim.x*gridDim.x;
    // Make sure we do not go out of bounds
    for (k = id*(n/t); k < (id+1)*(n/t); k++)
    {
        c[k] = a[k] + b[k];
    }
}
```

CUDA kernel launch with 10 blocks
of 1024 threads
final result: 1.000000
Time for the kernel: 2.985888 ms

Παράδειγμα: vecadd6.cu

- ▶ 'Ιδιο με το vecadd5.cu
 - Εκτός από το for στην vecAdd

```
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int t, k, id = blockIdx.x*blockDim.x+threadIdx.x;
    t=blockDim.x*gridDim.x;
    // Make sure we do not go out of bounds
    for (k = id; k < n; k += t)
    {
        c[k] = a[k] + b[k];
    }
}
```

CUDA kernel launch with 10
blocks of 1024 threads
final result: 1.000000
Time for the kernel: 0.336448 ms

Άσκηση: vad2.cu

- ▶ Να γραφτεί πρόγραμμα CUDA που:
 - Ο host θα ζητάει από το χρήστη
 - Τον αριθμό N των στοιχείων των πινάκων A και B
 - Τις (ακέραιες) τιμές των στοιχείων των πινάκων A και B
 - Η device θα υπολογίζει παράλληλα το άθροισμα κάθε στοιχείου του A με το αντίστοιχο στοιχείο του πίνακα B , αποθηκεύοντας τη νέα τιμή του στον πίνακα A
 - $A[i] = A[i] + B[i] \quad i=\{0, 1, \dots, N\}$
 - Ο host θα εκτυπώνει στην οθόνη τον πίνακα A (που προκύπτει) με τις νέες του τιμές

Ασκ: vad2.cu (υπόδειξη)

```
#include <stdio.h>
```

```
__global__ void add(int *a, int *b)
{
    . . .
}
```

```
int main(void) {
    int *a, *b, size_ab;
    int *d_a, *d_b;
    int size = sizeof(int);
    int i;

    // Ανάγνωση μεγέθους πινάκων A και B από το χρήστη

    // Δέσμευση μνήμης για τους πίνακες στον host

    // Ανάγνωση των στοιχείων των δύο πινάκων από τον χρήστη

    // Δέσμευση μνήμης μέσω cudaMalloc και μεταφορά στην device μέσω cudaMemcpy

    // Κλήση συνάρτησης πυρήνα (kernel launch) για υπολογισμό αθροίσματος πινάκων

    // Μεταφορά αποτελέσματος από το device πίσω στον host

    // Εκτύπωση αποτελέσματος (νέα περιεχόμενα του πίνακα A)

    // Απελευθέρωση δεσμευμένης μνήμης

    return 0;
}
```

```
$ ./vad2
```

Size of arrays a and b:5

A Element 0=1

A Element 1=2

A Element 2=3

A Element 3=4

A Element 4=5

B Element 0=6

B Element 1=7

B Element 2=8

B Element 3=9

B Element 4=10

A[0]=7

A[1]=9

A[2]=11

A[3]=13

A[4]=15

Άσκηση: lvs.cu

- ▶ Να φτιαχτεί πρόγραμμα CUDA που:
 - Θα παίρνει 3 ορίσματα
 - Μέγεθος των πινάκων (από 1 έως 1.000.000 στοιχεία)
 - Ακέραιος αριθμός
 - Αριθμός νημάτων (1 έως 1024)
 - Θα αρχικοποιεί πίνακα input_h
 - Κάθε στοιχείο ίσο με 1
 - Θα αρχικοποιεί πίνακα output_h
 - Κάθε στοιχείο ίσο με 0
 - Θα καλεί συνάρτηση πυρήνα που θα πολλαπλασιάζει κάθε στοιχείο του πίνακα input_h με τον ακέραιο αριθμό
 - Θα εκτυπώνει δειγματοληπτικά ορισμένα στοιχεία του τελικού πίνακα που θα προκύπτει μετά τον πολλαπλασιασμό

Άσκηση: lvs.cu (υπόδειξη)

```
int main(int argc, char *argv[])
{
    int      N, num, i, blocks, threads;
    int      *input_h, *output_h;
    int      *vector_d;

    if (argc != 4) {
        // Usage instructions
    }

    N = atoi(argv[1]);
    num = atoi(argv[2]);
    threads = atoi(argv[3]);

    . . . .

}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
```

```
__global__ void multiply(int *vector, int num, int N)
{
    . . . .
}
```

```
$ ./lvs 1000000 25 1024
output_h[ 0] = 25
output_h[499999] = 25
output_h[999999] = 25
```


Άσκηση mn.cu

- ▶ Να φτιαχτεί πρόγραμμα CUDA που:
 - Θα παίρνει 2 ορίσματα
 - Μέγεθος (διάσταση N) των πινάκων (N από 1 έως 20.000)
 - Αριθμός νημάτων (1 έως 1024)
 - Θα αρχικοποιεί μονοδιάστατο πίνακα input_h (κάθε στοιχείο ίσο με 1) και τον μονοδιάστατο πίνακα output_h (κάθε στοιχείο ίσο με 0)
 - Θα αρχικοποιεί NxN πίνακα matrix_h
 - $matrix_h[i * N + j] = i \% 100;$
 - Θα καλεί συνάρτηση πυρήνα που θα πολλαπλασιάζει τον πίνακα matrix με τον input
 - Θα εκτυπώνει δειγματοληπτικά ορισμένα στοιχεία του τελικού πίνακα που θα προκύπτει μετά τον πολλαπλασιασμό

Άσκηση: mv.cu (υπόδειξη)

```
int main(int argc, char *argv[])
{
    int      N, i, j, fill, blocks, threads;
    int      *input_h, *output_h, *matrix_h; /* Pointers for vectors on the host. */
    int      *input_d, *output_d, *matrix_d; /* Pointer for vector on the device. */

    if (argc != 3) { ... }
    N = atoi(argv[1]);
    threads = atoi(argv[2]);
    // malloc for host pointers
    for (i = 0; i < N; i++) { input_h[i] = 1; output_h[i] = 0; }
    for (i = 0; i < N; i++) { ... }
        fill = i % 100;
        for (j = 0; j < N; j++) {
            matrix_h[i * N + j] = fill;
        }
    }

    // cudaMalloc, cudaMemcpy & kernel launch
    ....

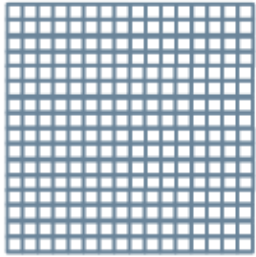
    // Sample output
    printf("output_h[%4d] = %d\n", 0, output_h[0]);
    printf("output_h[%4d] = %d\n", (N - 1) / 2, output_h[(N - 1) / 2]);
    printf("output_h[%4d] = %d\n", N - 1, output_h[N - 1]);
}
```

```
#include <stdio.h>
#include <stdlib.h>

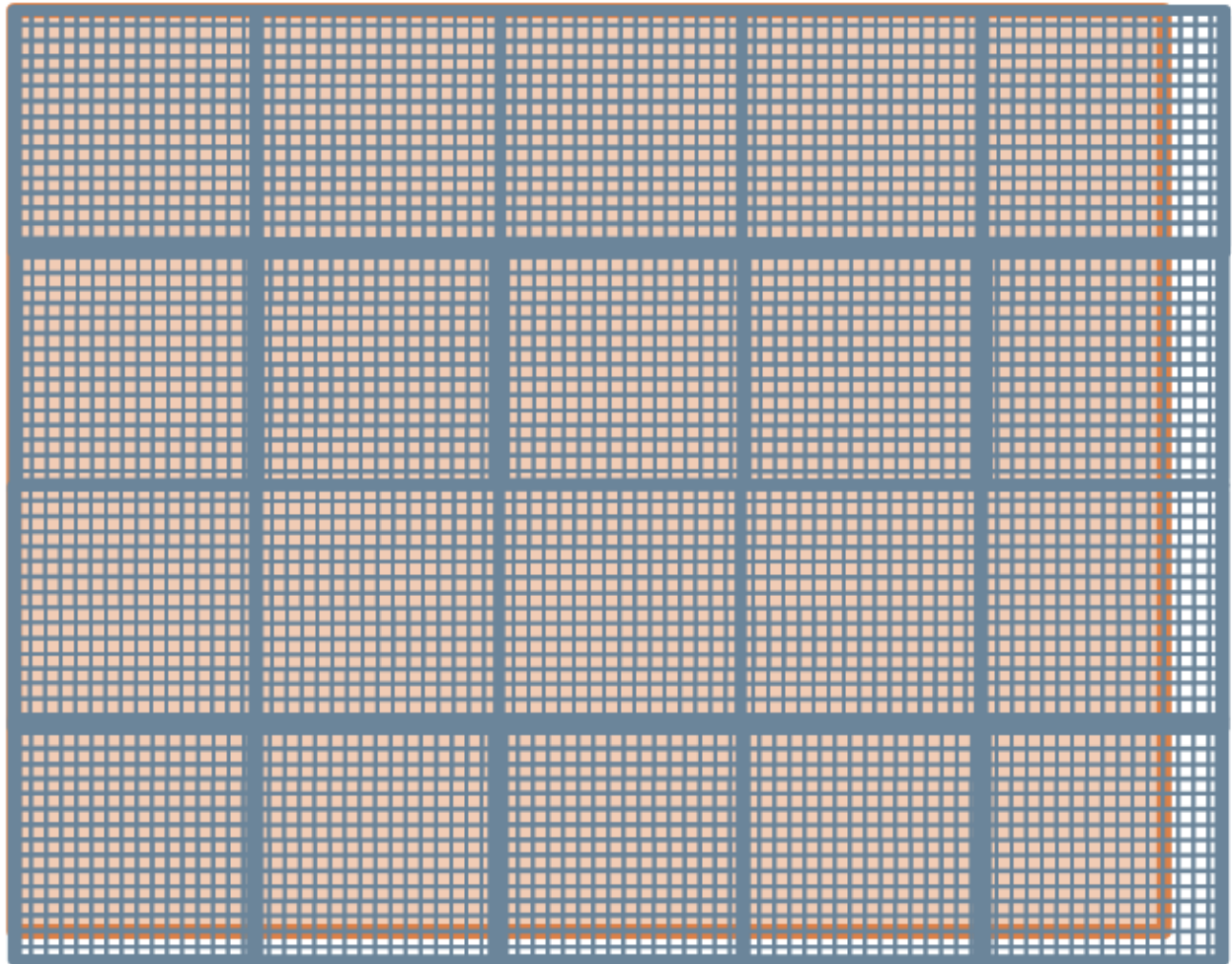
__global__ void multiply(int *matrix, int *input, int *output, int N)
{
    ....
}
```

```
$ ./mv 20000 1024
output_h[  0] = 0
output_h[9999] = 1980000
output_h[19999] = 1980000
```

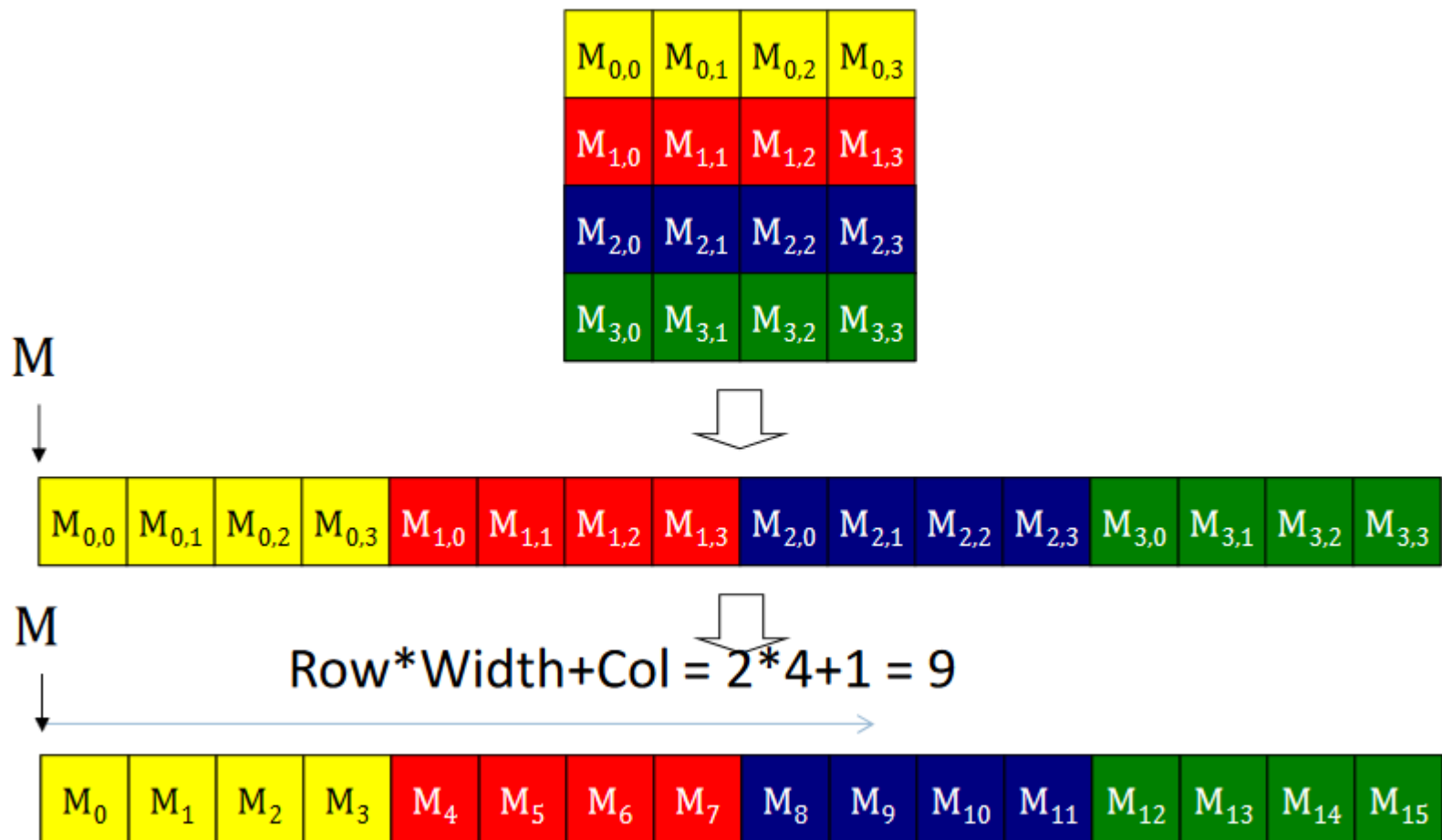
Επεξεργασία εικόνας με χρήση δισδιάστατου πίνακα νημάτων



16×16 blocks



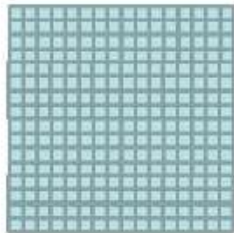
Απεικόνιση κατά γραμμές



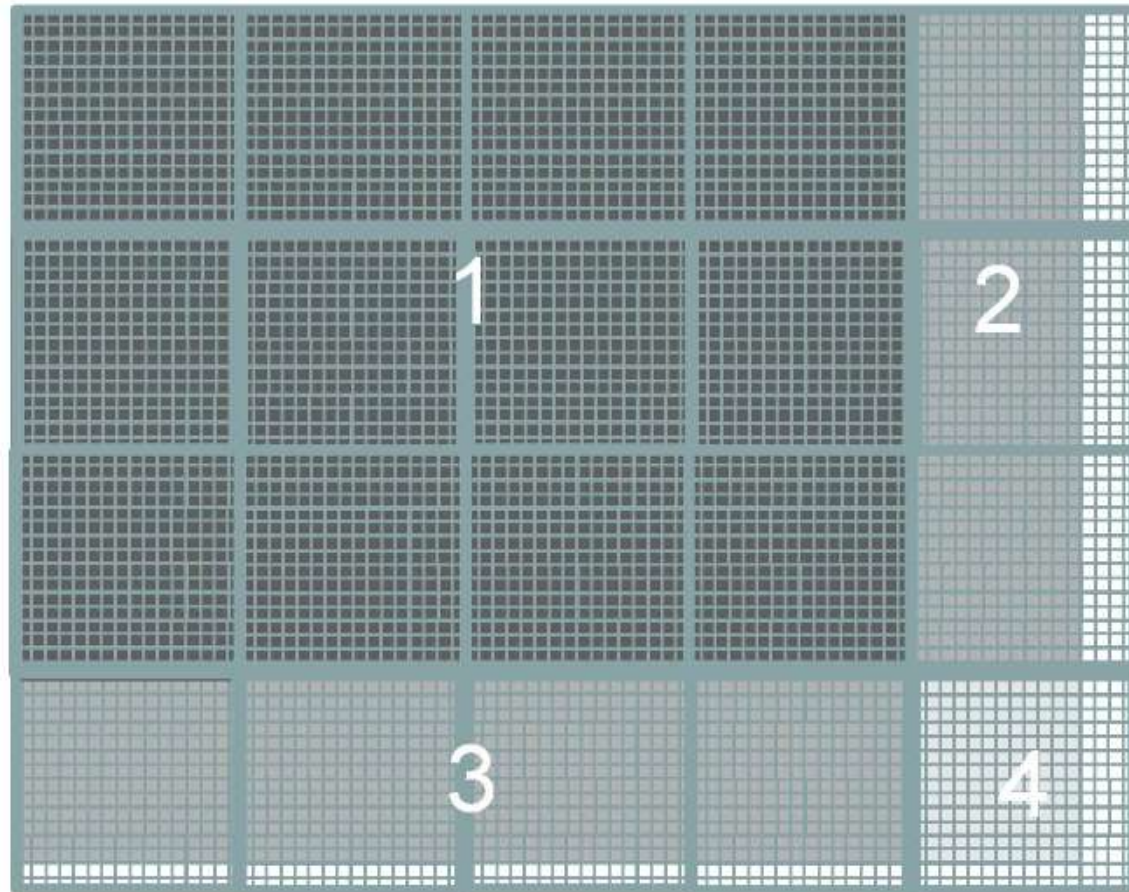
Συνάρτηση πυρήνα PictureKernel

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int n, int m)
{
    // Calculate the row # of the d_Pin and d_Pout element to process
    int Row = blockIdx.y*BlockHeight + threadIdx.y;
    // Calculate the column # of the d_Pin and d_Pout element to process
    int Col = blockIdx.x*BlockWidth + threadIdx.x;
    // Each thread computes one element of d_Pout if in range
    if ((Row < m) && (Col < n))
    {
        d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
    }
}
```

Εικόνα 76x62 με block 16x16



16x16 block



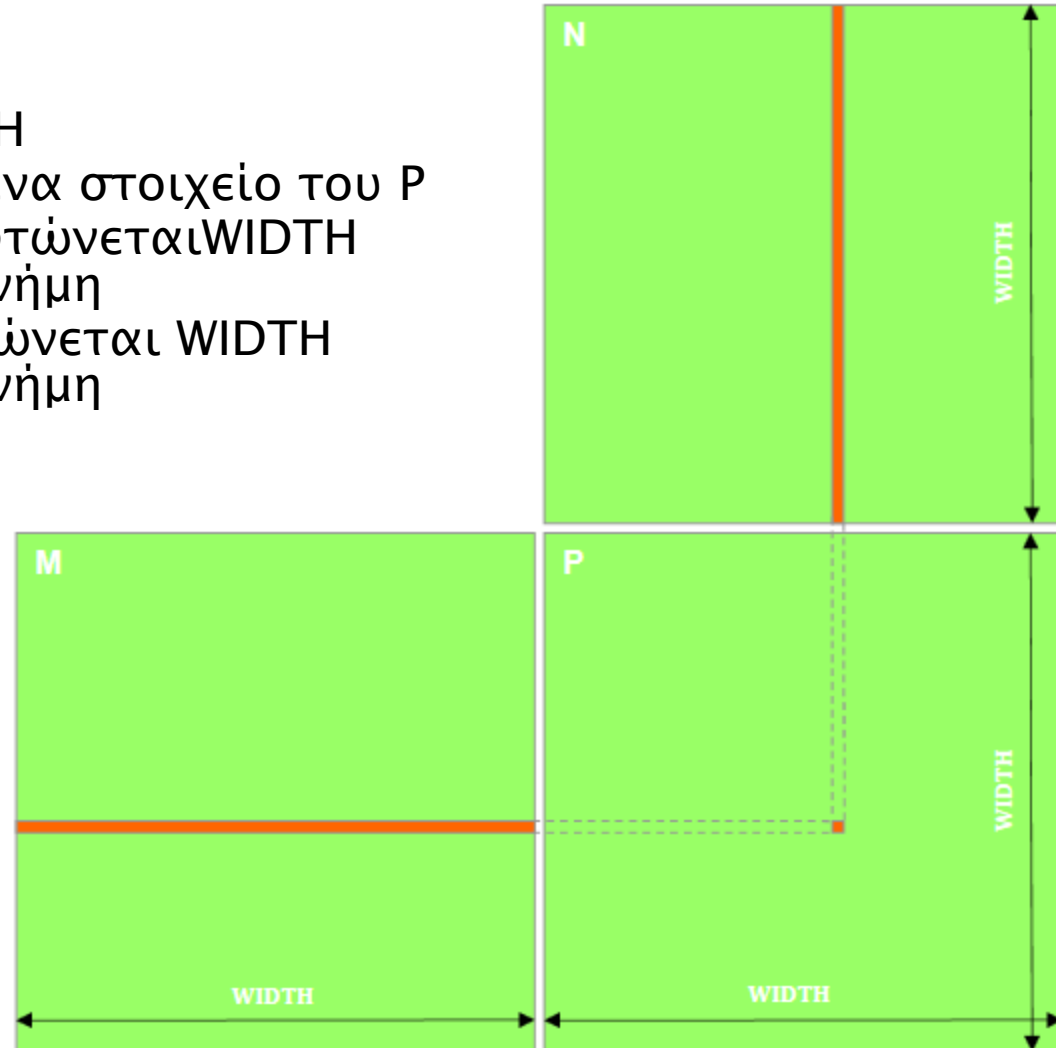
Κάλυψη εικόνας μεγέθους 76x62 με block μεγέθους 16x16

Πολλαπλασιασμός πινάκων

- ▶ Δείχνει τις βασικές δυνατότητες διαχείρισης μνήμης και νημάτων σε προγράμματα CUDA
 - Χρήση δεικτών (index) νημάτων
 - Διάταξη δεδομένων στην μνήμη
 - Χρήση καταχωρητών
- ▶ Για λόγους απλότητας
 - Υποθέτουμε τετραγωνικά μητρώα
 - Αφήνουμε την χρήση κοινής μνήμης για αργότερα

Πολ/σμος τετραγωνικών πινάκων

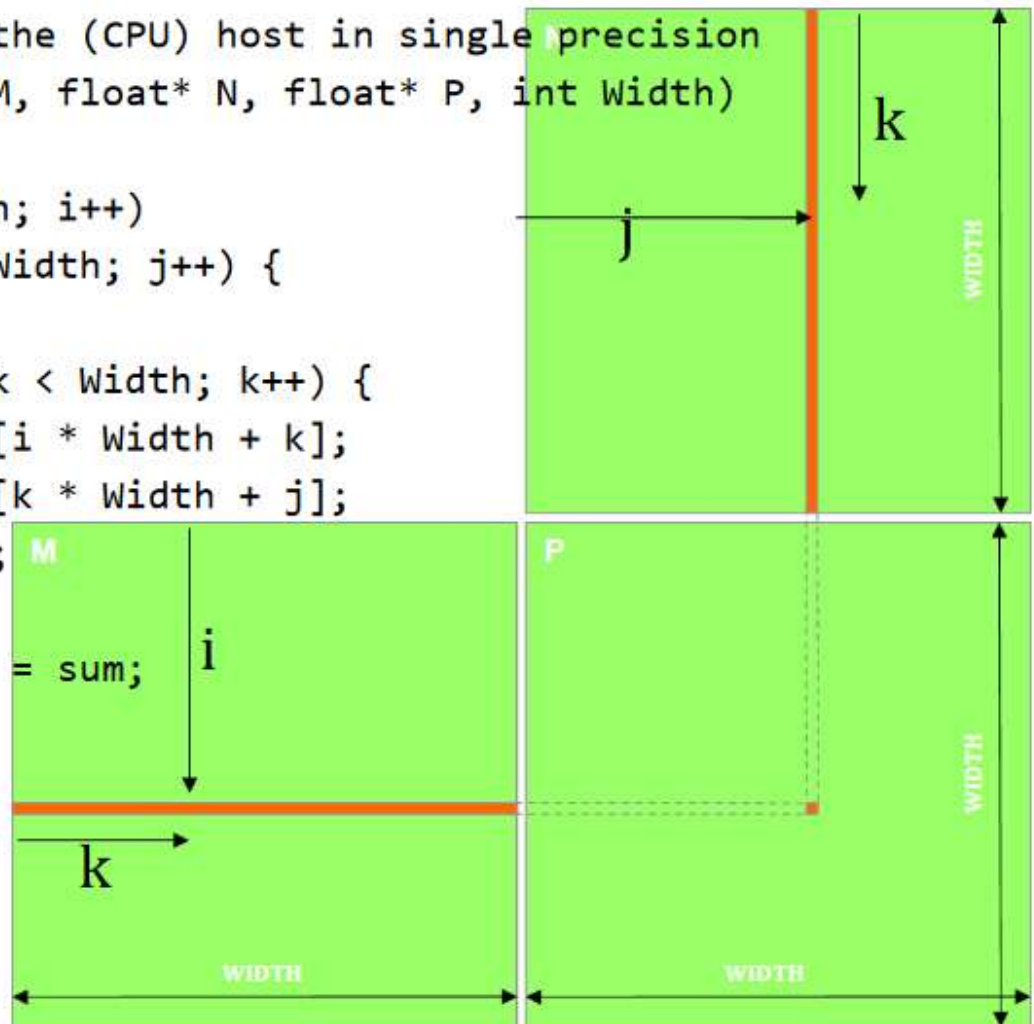
- ▶ $P = M * N$
 - Μεγέθους $WIDTH \times WIDTH$
 - Κάθε νήμα υπολογίζει ένα στοιχείο του P
 - Κάθε γραμμή του M φορτώνεται $WIDTH$ φορές από την global μνήμη
 - Κάθε στήλη του N φορτώνεται $WIDTH$ φορές από την global μνήμη



Πολλαπλασιασμός πινάκων

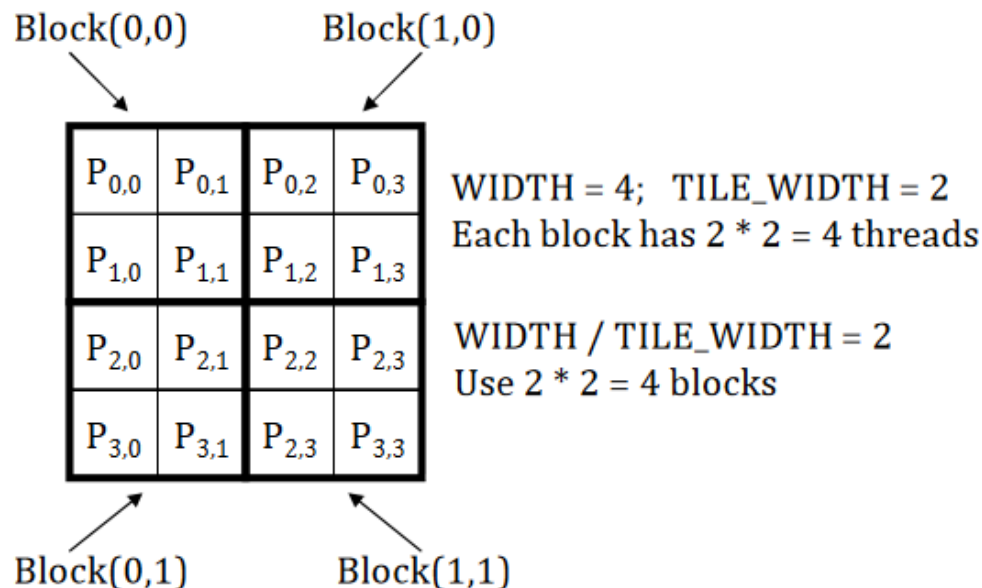
Κώδικας για CPU – mm-cpu.c

```
// Matrix multiplication on the (CPU) host in single precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; i++)
        for (int j = 0; j < Width; j++) {
            double sum = 0;
            for (int k = 0; k < Width; k++) {
                double a = M[i * Width + k];
                double b = N[k * Width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```



Συνάρτηση πυρήνα (μικρό παράδειγμα)

- ▶ Κάθε block νημάτων θα αναλάβει τον υπολογισμό ενός υπομητρώου μεγέθους $(\text{TILE_WIDTH})^2$ του τελικού μητρώου
 - Κάθε block έχει $(\text{TILE_WIDTH})^2$ νήματα
 - Δημιουργία δισδιάστατου πλέγματος block μεγέθους $(\text{WIDTH}/\text{TILE_WIDTH})^2$



Συνάρτηση πυρήνα (λίγο μεγαλύτερο παράδειγμα)

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

$WIDTH = 8$; $TILE_WIDTH = 2$
Each block has $2 * 2 = 4$ threads

$WIDTH / TILE_WIDTH = 4$
Use $4 * 4 = 16$ blocks

Συνάρτηση πυρήνα (λίγο μεγαλύτερο παράδειγμα)

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

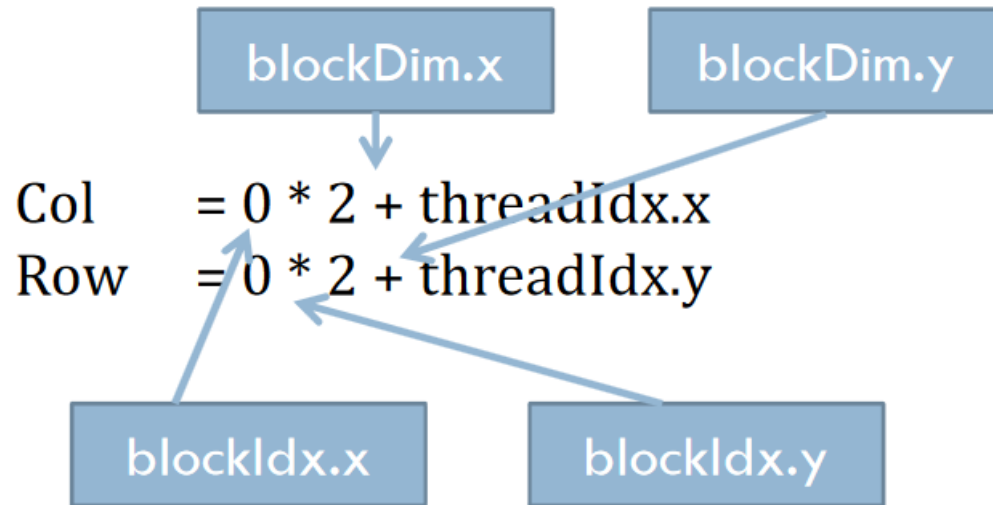
WIDTH = 8; TILE_WIDTH = 4
Each block has $4 * 4 = 16$ threads

WIDTH / TILE_WIDTH = 2
Use $2 * 2 = 4$ blocks

Πολλαπλασιασμός πινάκων

- ▶ Παράδειγμα πολλαπλασιασμού πίνακα με πίνακα
 - $\text{TILE_WIDTH} = 2$
 - 4 threads per block

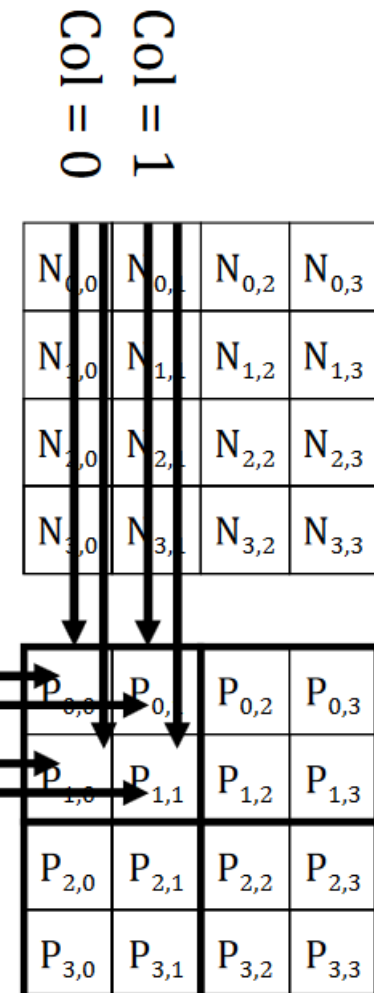
Block(0, 0) with TILE_WIDTH=2



Row = 0

Row = 1

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



Block(0, 1) with TILE_WIDTH=2

$$\begin{aligned}\text{Col} &= 1 * 2 + \text{threadIdx.x} \\ \text{Row} &= 0 * 2 + \text{threadIdx.y}\end{aligned}$$

blockIdx.x

blockIdx.y

Row = 0

Row = 1

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$

Col = 2
Col = 3

Άσκηση: mm.cu

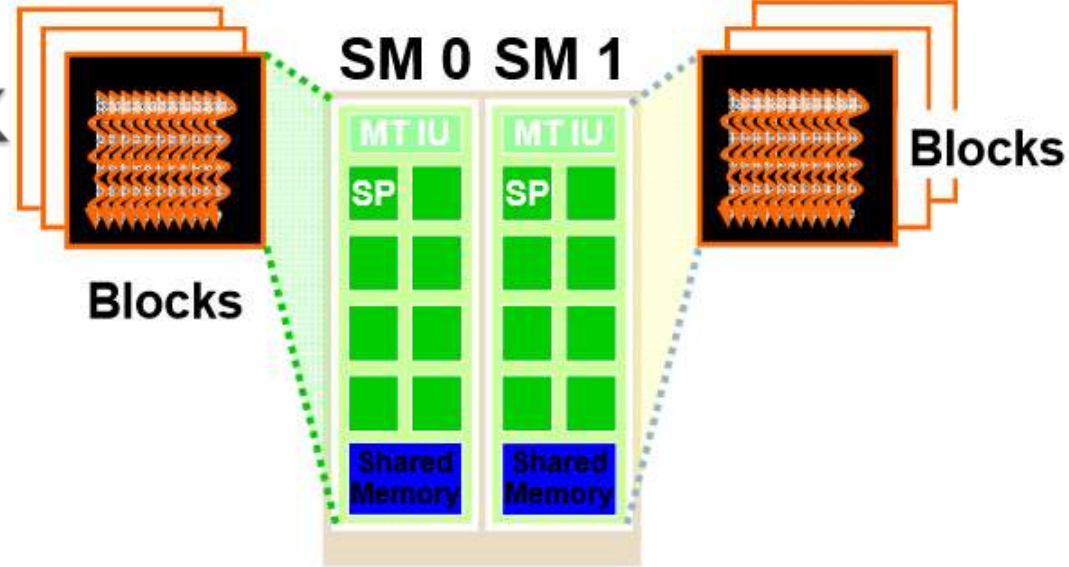
```
void MatrixMultiplication(float *M, float *N, float *P, int Width) {  
    int size = Width * Width * sizeof(float);  
    float *Md, *Nd, *Pd;  
    // Capture start time  
    // Allocate memory on the GPU  
    // Transfer M and N to device memory  
  
    // Kernel invocation code  
    dim3 dimBlock(32, 32);  
    dim3 dimGrid(Width/32, Width/32);  
    Kernel<<<dimGrid, dimBlock>>>( Md, Nd, Pd, Width);  
  
    // Transfer P from device  
    // Get stop time, and display the timing results  
    // Free the memory allocated on the GPU  
    // Destroy events to free memory  
}
```

```
#include <stdio.h>
```

```
int main(void){  
    void MatrixMultiplication(float *, float *, float *, int);  
    const int Width = 1024;  
    int size = Width * Width * sizeof(float);  
    float *M, *N, *P;  
  
    // Allocate memory on the CPU for M, N & P  
    // Initialize the matrices M and N  
  
    // Call host routine for matrix multiplication  
    MatrixMultiplication(M, N, P, Width);  
  
    // Free the memory allocated on the CPU  
  
    return 0;  
}
```

```
__global__ void Kernel(float *Md, float *Nd, float *Pd, int Width) {  
    // Calculate the column index of the Pd element  
    int x = ...  
    // Calculate the row index of the Pd element  
    int y = ...  
  
    float Pvalue = 0;  
    // Each thread computes one element of the output  
    // matrix Pd.  
    for (int k = 0; k < Width; ++k) {  
        Pvalue = . . . .  
    }  
  
    // Write back to the global memory  
    Pd[y*Width + x] = Pvalue;  
}
```


Εκτέλεση block νημάτων



- ▶ Νήματα ανατίθενται σε Streaming Multiprocessors σε μεγέθη block
 - Μέχρι 8 block σε κάθε SM, ανάλογα με την GPU
 - Fermi SM μπορεί να διαχειριστεί μέχρι 1536 νήματα
 - Ίσως $256 \text{ (νήματα/block)} * 6 \text{ blocks}$
 - Ή $512 \text{ (νήματα/block)} * 3 \text{ block}$, κλπ.
 - Τα νήματα τρέχουν ταυτόχρονα
 - SM διατηρεί δείκτες νήματος/block
 - SM διαχειρίζεται/χρονοδρομολογεί την εκτέλεση των νημάτων

Θέματα επιλογής μεγέθους block

- ▶ Για έναν πολλαπλασιασμό πινάκων που χρησιμοποιεί block, ποια είναι η καλύτερη επιλογή; Block μεγέθους 8x8, 16x16 ή 32x32;
 - Για 8x8, θα έχουμε 64 νήματα ανά block
 - Κάθε SM μπορεί να διαχειριστεί μέχρι 1536 νήματα, οπότε χρειαζόμαστε 24 block για να εκμεταλλευτούμε πλήρως το SM
 - Όμως κάθε SM μπορεί να διαχειριστεί μέχρι 8 blocks, άρα μόλις 512 νήματα θα πάνε σε κάθε SM!
 - Για 16x16, θα έχουμε 256 νήματα ανά block
 - Κάθε SM μπορεί να διαχειριστεί μέχρι 1536 νήματα, οπότε χρειαζόμαστε 6 block για να εκμεταλλευτούμε πλήρως το SM
 - Εκμεταλλευόμαστε πλήρως τις δυνατότητες του SM (εκτός αν υπάρχουν άλλα θέματα στην κατανομή των πόρων, όπως πλήθος καταχωρητών, κλπ)
 - Για 32x32, θα έχουμε 1024 νήματα ανά block
 - Κάθε SM μπορεί να διαχειριστεί μόνο ένα block
 - Εκμεταλλευόμαστε μόλις τα 2/3 της χωρητικότητας του SM σε νήματα
 - Λειτουργεί από την έκδοση 3.0 και μετά της CUDA
 - Πιθανόν πολύ μεγάλο για προηγούμενες εκδόσεις

Application Programming Interface (API)

- ▶ Η διεπαφή είναι μια επέκταση της γλώσσας προγραμματισμού C
- ▶ Αποτελείται από:
 - Επεκτάσεις της γλώσσας προγραμματισμού
 - Για την εκτέλεση τμημάτων κώδικα στο device
 - Μια βιβλιοθήκη χρόνου εκτέλεσης(runtime library) που αποτελείται από:
 - Ένα κοινό μέρος για το host και το device το οποίο προσφέρει ενσωματωμένους τύπους διανυσμάτων (vector types) και ένα υποσύνολο της βιβλιοθήκης χρόνου εκτέλεσης της C
 - Ένα τμήμα για το host για την διαχείριση ενός ή περισσότερων device από τον host
 - Ένα τμήμα για το device που προσφέρει συναρτήσεις μόνο για αυτό

Μαθηματικές συναρτήσεις

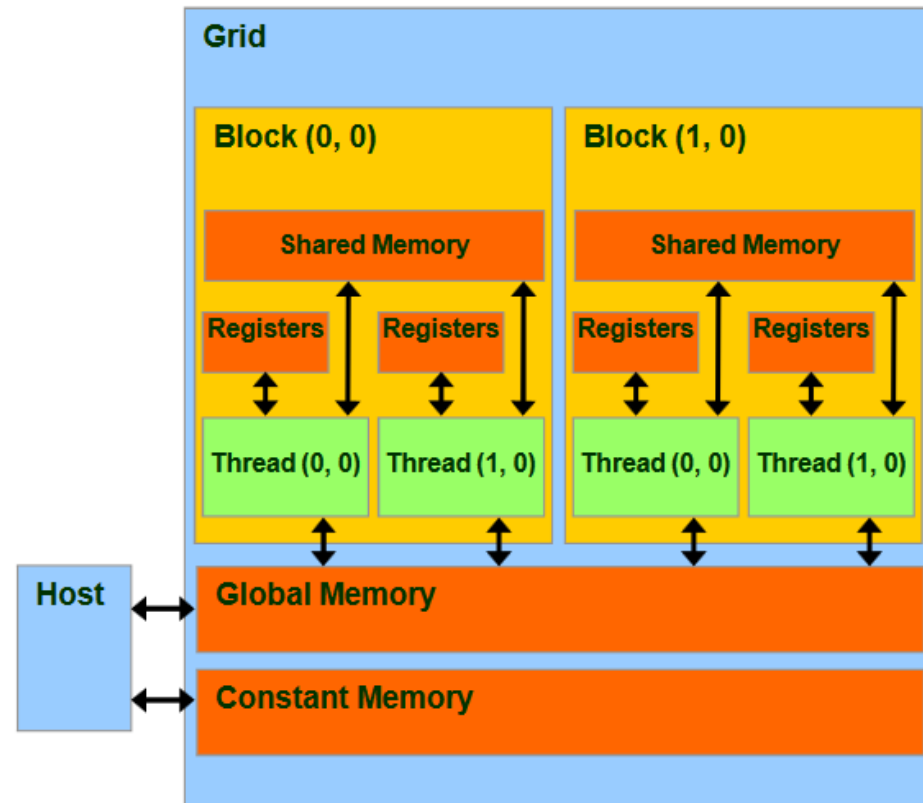
- ▶ Οι συναρτήσεις αυτές μπορούν να χρησιμοποιηθούν τόσο σε host όσο και σε device code.
 - pow, sqrt, exp, log
 - sin, cos, tan
 - ceil, floor, trunc, round
 - Κλπ.
- ▶ Όταν εκτελείται στον host, η συνάρτηση χρησιμοποιεί την αντίστοιχη υλοποίηση της βιβλιοθήκης χρόνου εκτέλεσης της C
- ▶ Οι συναρτήσεις υποστηρίζονται μόνο για μεταβλητές και όχι για διανυσματικούς τύπους δεδομένων (vector types)

Μαθηματικές συναρτήσεις

- ▶ Ενδογενείς συναρτήσεις (Intrinsic functions)
 - Συναρτήσεις που μπορούν να χρησιμοποιηθούν μονάχα στον κώδικα της device
- ▶ Αυτές οι ειδικές εκδόσεις έχουν μικρότερη ακρίβεια αλλά είναι πιο γρήγορες από τις κανονικές συναρτήσεις (standard functions).
- ▶ Έχουν το ίδιο όνομα με την προσθήκη `--` στην αρχή
 - `--pow`
 - `--log`
 - κλπ
- ▶ Είναι πιο γρήγορες γιατί κάνουν map σε λιγότερες native instructions.

Ιεραρχία της μνήμης

- ▶ Κάθε νήμα μπορεί να:
 - Διαβάσει/Γράψει σε καταχωρητές που ανήκουν στο νήμα (per thread registers) (~1 κύκλος)
 - Διαβάσει/Γράψει σε κοινή μνήμη που ανήκει στο block (per-block shared memory) (~5 κύκλοι)
 - Διαβάσει/Γράψει σε καθολική μνήμη που ανήκει στο πλέγμα (per-grid global memory) (~500 κύκλοι)
 - Διαβάσει από constant μνήμη που ανήκει στο πλέγμα (per-grid constant memory) (~5 κύκλοι αν υπάρχει στην κρυφή μνήμη)



Κοινή μνήμη (shared memory)

- ▶ Ειδικός τύπος μνήμης της οποίας τα περιεχόμενα και η προσπέλαση ορίζονται ρητά στον πηγαίο κώδικα
 - Βρίσκεται στον επεξεργαστή
 - Προσπελάζεται πολύ ταχύτερα από την καθολική (κύρια) μνήμη
 - Προσπελάζεται όπως και η καθολική μνήμη με εντολές προσπέλασης δεδομένων
 - Αναφέρεται συνήθως ως “local” ή “scratchpad memory” στην αρχιτεκτονική υπολογιστών
 - Μια μεταβλητή στη shared memory σημαίνει ότι δημιουργείται ένα αντίγραφο της για κάθε block
 - Κάθε νήμα που ανήκει στο ίδιο block έχει πρόσβαση στο συγκεκριμένο αντίγραφο, αλλά τα νήματα σε άλλα block δεν μπορούν να το δουν ή να το τροποποιήσουν

Προσδιοριστικά τύπων δεδομένων

Variable declaration	Memory	Scope	Lifetime
<code>int LocalVar;</code>	register	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	grid	application

- ▶ Το `__device__` είναι προαιρετικό όταν χρησιμοποιείται το `__shared__` ή το `__constant__`
- ▶ Για τις αυτόματες μεταβλητές χρησιμοποιούνται καταχωρητές
 - Εκτός από τους πίνακες, οι οποίοι τοποθετούνται στην καθολική μνήμη

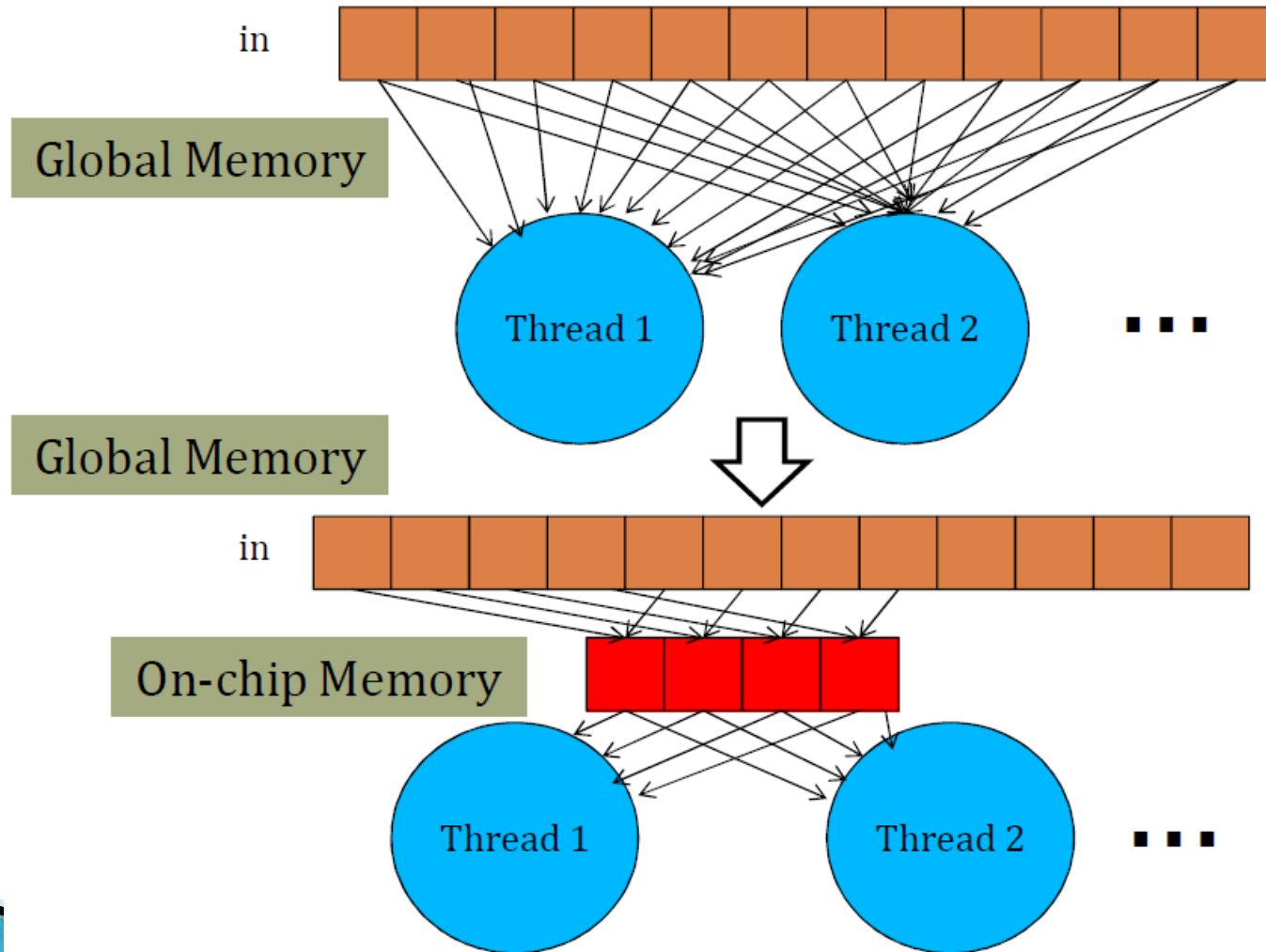
Συγχρονισμός με `__syncthreads()`

- ▶ Η κλήση αυτή εγγυάται ότι κάθε νήμα στο συγκεκριμένο block έχει ολοκληρώσει όλες τις εντολές πριν την κλήση `__syncthreads()` προτού να εκτελεστεί η πρώτη εντολή μετά από αυτό
- ▶ Θα τη δούμε πιο αναλυτικά στο παράδειγμα `sme.cu`


Στρατηγική προγραμματισμού

- ▶ Η καθολική μνήμη υλοποιείται με χρήση module μνήμης στο device (DRAM) –αργή προσπέλαση
- ▶ Μια καλύτερη στρατηγική για την πραγματοποίηση των υπολογισμών είναι να διαμοιράζουμε τα δεδομένα εισόδου σε μικρότερα τμήματα («πλακίδια» ή tiles) ώστε να εκμεταλλευόμαστε την γρηγορότερη κοινή μνήμη:
 - Διαμοιρασμός δεδομένων σε υποσύνολα που χωράνε στην κοινή μνήμη
 - Διαχείριση κάθε υποσυνόλου με ένα block νημάτων:
 - Αντιγραφή υποσυνόλου από την καθολική στην κοινή μνήμη, χρησιμοποιώντας πολλαπλά νήματα ώστε να αξιοποιηθεί ο παραλληλισμός στο επίπεδο της μνήμης
 - Πραγματοποίηση υπολογισμών στο υποσύνολο δεδομένων χρησιμοποιώντας την κοινή μνήμη
 - Κάθε νήμα μπορεί (και πρέπει!) να χρησιμοποιήσει περισσότερες φορές κάθε στοιχείο
 - Αντιγραφή αποτελεσμάτων από την κοινή στην καθολική μνήμη

Blocking στην κοινή μνήμη

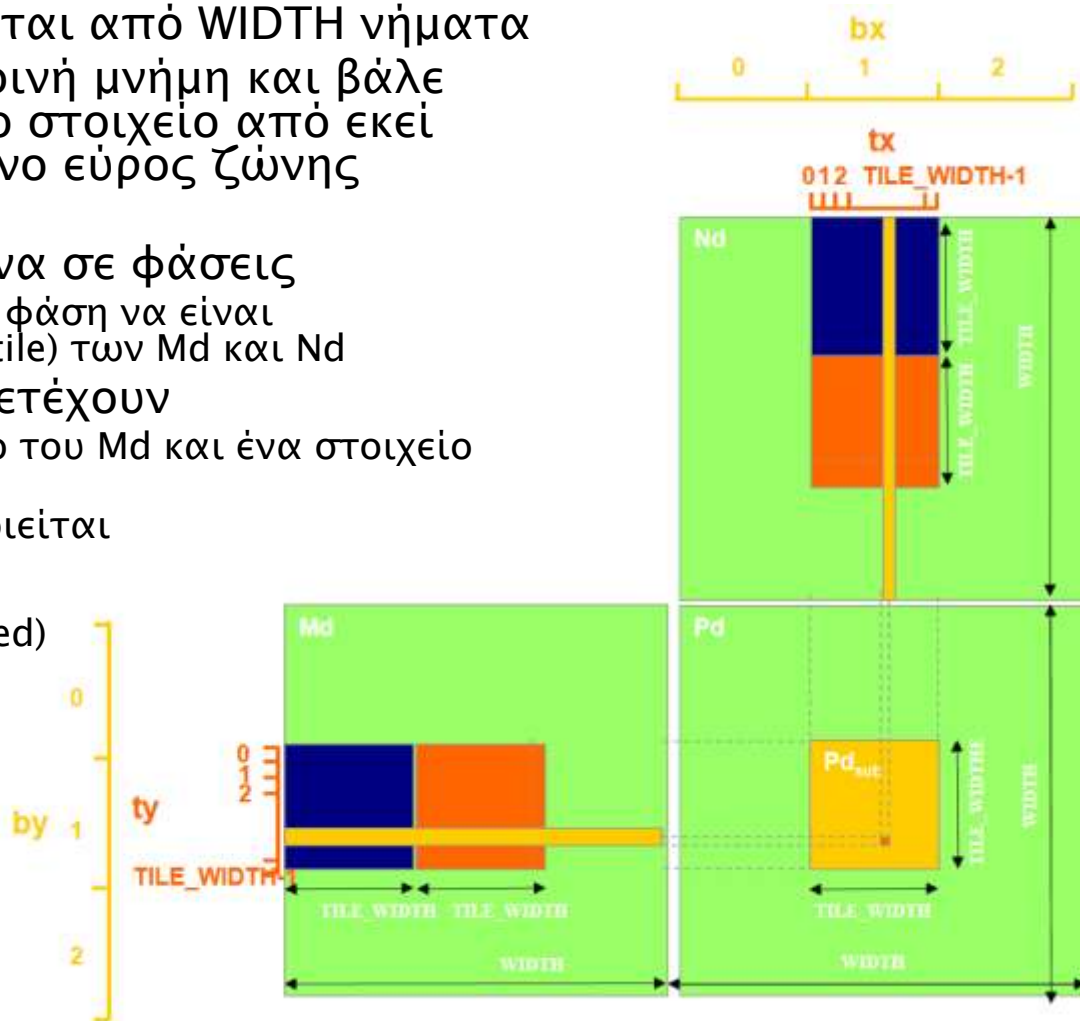


Τεχνική blocking/tiling

- ▶ Βρες ένα block/tile της καθολικής μνήμης το οποίο προσπελαύνεται από πολλά νήματα
 - ▶ Αντέγραψε το block/tile από την καθολική στην κοινή μνήμη
 - ▶ Βάλε τα νήματα να προσπελαύνουν τα δεδομένα που χρειάζονται από την κοινή μνήμη
 - ▶ Πήγαινε στο επόμενο block/tile
- 

Κοινή μνήμη για επαναχρησιμοποίηση δεδομένων

- ▶ Κάθε στοιχείο εισόδου διαβάζεται από WIDTH νήματα
- ▶ Φόρτωσε κάθε στοιχείο στην κοινή μνήμη και βάλε πολλά νήματα να διαβάζουν το στοιχείο από εκεί ώστε να μειωθεί το απαιτούμενο εύρος ζώνης
 - Tiled αλγόριθμοι
- ▶ Χώρισε την εκτέλεση του πυρήνα σε φάσεις
 - Η προσπέλαση δεδομένων σε κάθε φάση να είναι επικεντρωμένη σε ένα υποσύνολο(tile) των Md και Nd
- ▶ Όλα τα νήματα του block συμμετέχουν
 - Κάθε νήμα αντιγράφει ένα στοιχείο του Md και ένα στοιχείο του Nd
 - Το αντεγγραμμένο στοιχείο αξιοποιείται σε κάθε νήμα έτσι ώστε οι προσπελάσεις μνήμης σε κάθε warp να είναι συνεχόμενες (coalesced)



Επεξεργασία Block(0,0) [1]

Κοινή μνήμη

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$N_{0,0}$	$N_{0,1}$
$N_{1,0}$	$N_{1,1}$

Κοινή μνήμη

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

$M_{0,0}$	$M_{0,1}$
$M_{1,0}$	$M_{1,1}$

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$

Επεξεργασία Block(0,0) [2]

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

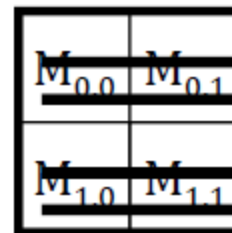
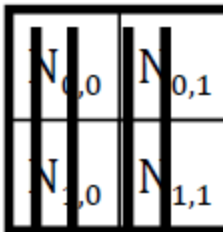
Κοινή μνήμη

$N_{0,0}$	$N_{0,1}$
$N_{1,0}$	$N_{1,1}$

Κοινή μνήμη

$M_{0,0}$	$M_{0,1}$
$M_{1,0}$	$M_{1,1}$

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$



Επεξεργασία Block(0,0) [3]

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

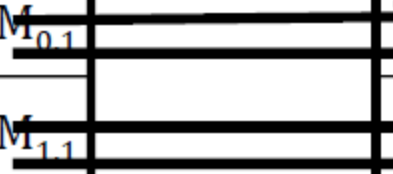
Κοινή μνήμη

$N_{0,0}$	$N_{0,1}$
$N_{1,0}$	$N_{1,1}$

Κοινή μνήμη

$M_{0,0}$	$M_{0,1}$
$M_{1,0}$	$M_{1,1}$

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$



Επεξεργασία Block(0,0) [4]

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

Κοινή μνήμη

$N_{0,0}$	$N_{0,1}$
$N_{1,0}$	$N_{1,1}$

Κοινή μνήμη

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

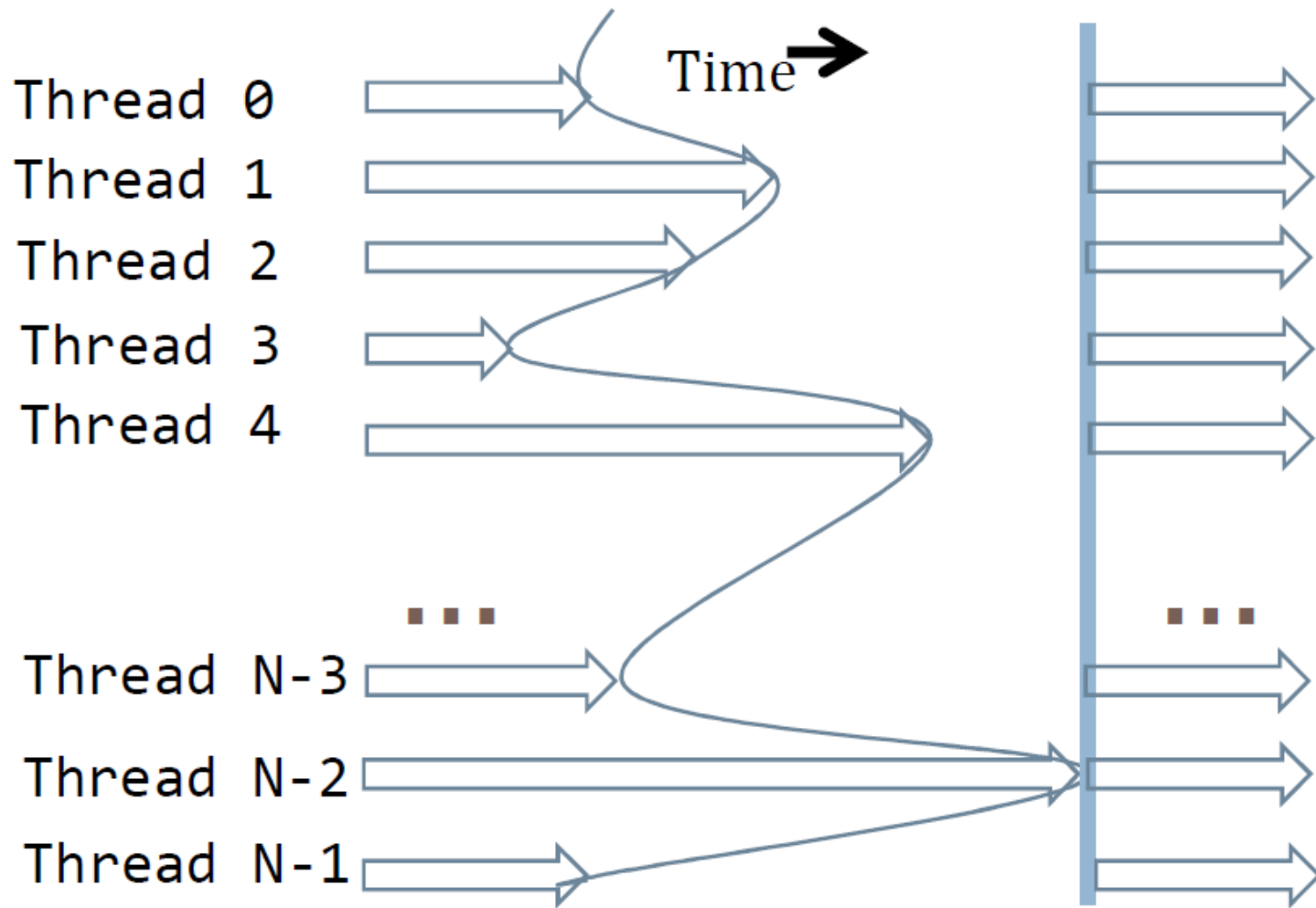
$M_{0,0}$	$M_{0,1}$
$M_{1,0}$	$M_{1,1}$

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$

Φράγματα (barriers)

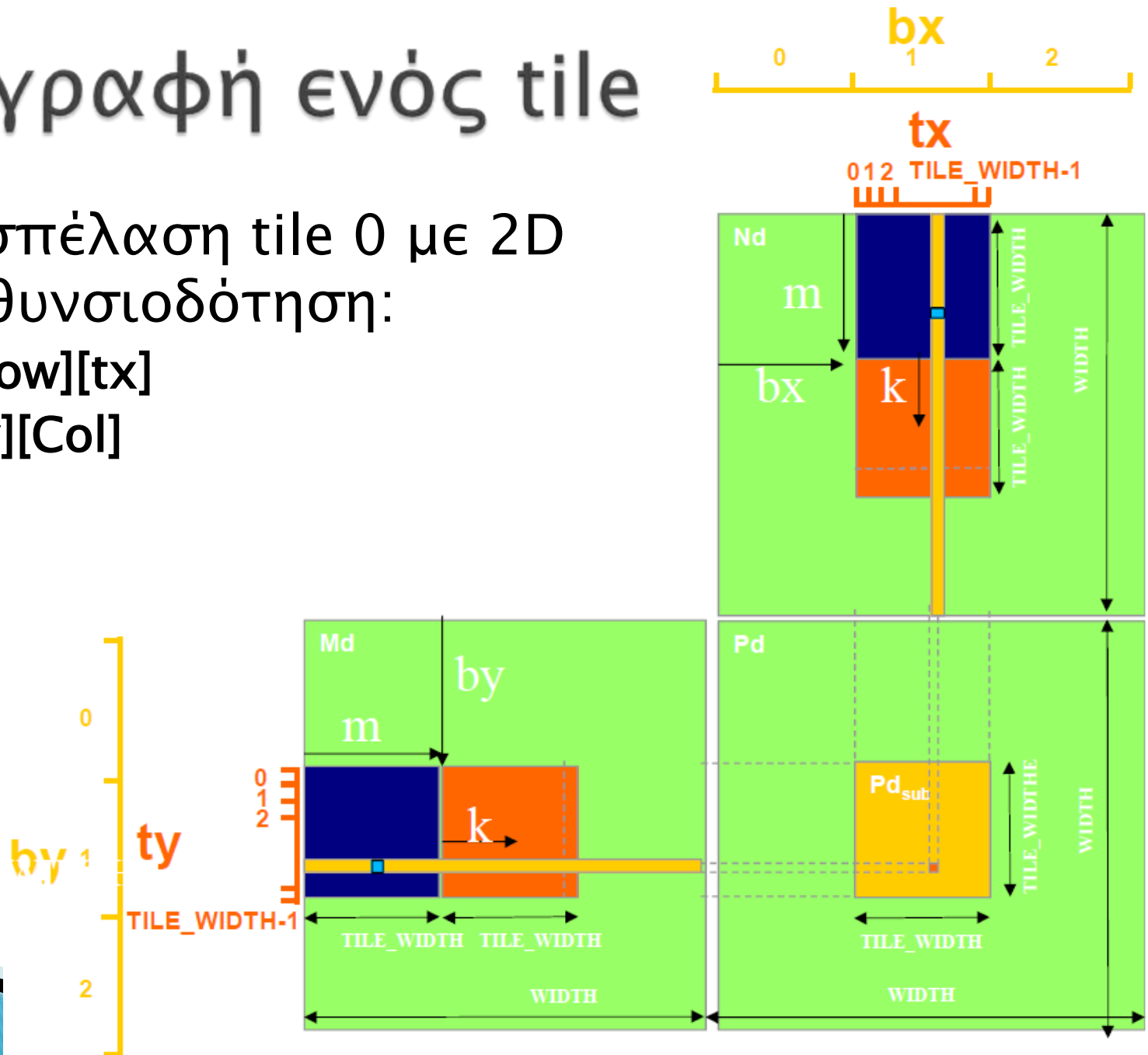
- ▶ Κλήση συνάρτησης στην διεπαφή της CUDA
 - `__syncthreads()`
- ▶ Όλα τα νήματα ενός block πρέπει να καλέσουν την `__syncthreads()` προτού μπορέσουν να συνεχίσουν την εκτέλεση τους
- ▶ Χρειάζεται σε αλγόριθμους που χρησιμοποιούν tiles
 - Εξασφάλιση πως όλα τα στοιχεία του tile φορτώθηκαν
 - Εξασφάλιση πως όλα τα στοιχεία του tile χρησιμοποιήθηκαν

Παράδειγμα φράγματος



Αντιγραφή ενός tile

- ▶ Προσπέλαση tile 0 με 2D διευθυνσιοδότηση:
 - $M[\text{Row}][\text{tx}]$
 - $N[\text{ty}][\text{Col}]$



Αντιγραφή ενός tile

- Προσπέλαση tile 1 με 2D διευθυνσιοδότηση:

- $M[\text{Row}][1 * \text{TILE_WIDTH} + tx]$
- $N[1 * \text{TILE_WIDTH} + ty][\text{Col}]$

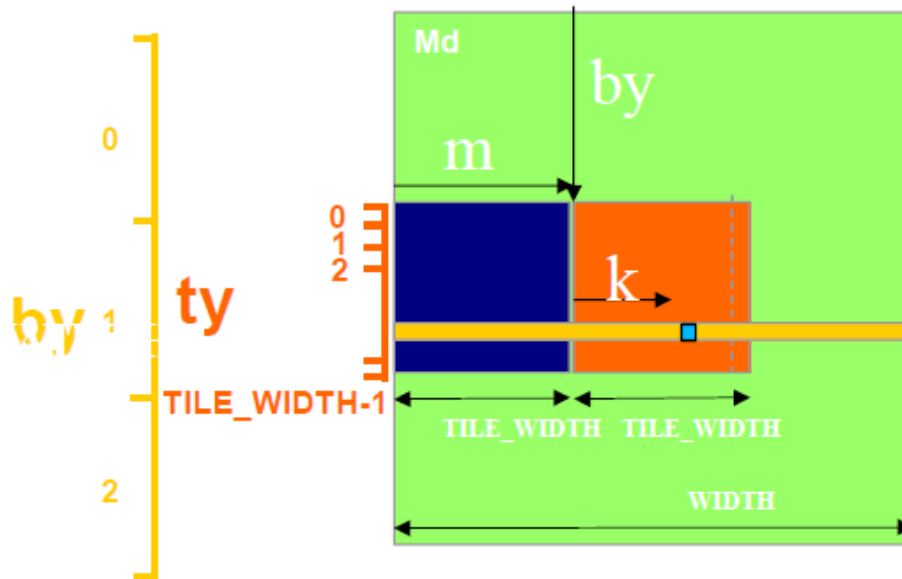
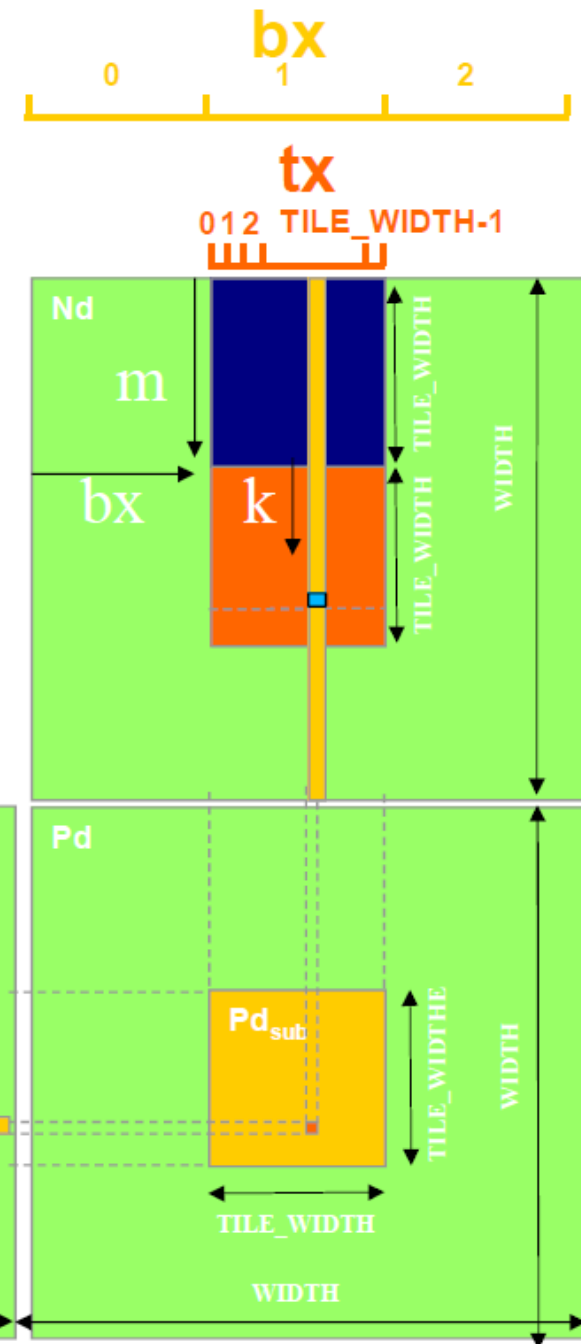
Τα μητρώα M και N έχουν δεσμευτεί δυναμικά, άρα πρέπει να χρησιμοποιήσουμε 1D διευθυνσιοδότηση:

$M[\text{Row}][m * \text{TILE_WIDTH} + tx]$

$M[\text{Row} * \text{Width} + m * \text{TILE_WIDTH} + tx]$

$N[m * \text{TILE_WIDTH} + ty][\text{Col}]$

$N[(m * \text{TILE_WIDTH} + ty) * \text{Width} + \text{Col}]$



Πολλαπλασιασμός πινάκων με χρήση κοινής μνήμης mm_shared.cu

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // declare cache in the shared memory
    __shared__ float Mds[blockD][blockD];
    __shared__ float Nds[blockD][blockD];

    // keep track of column index of the Pd element using thread index
    int x = . . . .
    // keep track of row index of the Pd element using thread index
    int y = . . . .

    float Pvalue = 0;
    // Loop over the Md and Nd block dimension required to compute the Pd
    element
    for (int m = 0; m < Width/blockD; m++){
        // collaboratively loading of Md and Nd blocks into shared memory
        Mds[threadIdx.y][threadIdx.x] = Md[y * Width + (m * blockD + threadIdx.x)];
        Nds[threadIdx.y][threadIdx.x] = Nd[(m * blockD + threadIdx.y) * Width + x];
        __syncthreads();
        // keep track of the running sum
        for (int k = 0; k < blockD; k++)
            Pvalue += Mds[threadIdx.y][k] * Nds[k][threadIdx.x];
        __syncthreads();
    }
    // write back to the global memory
    Pd[y * Width + x] = Pvalue;
}
```

```
void MatrixMultiplication(float *M, float *N, float *P, int
Width) {
    int size = Width * Width * sizeof(float);
    float *Md, *Nd, *Pd;
    // capture start time
    // allocate memory on the GPU
    // transfer M and N to device memory
    dim3 dimBlock(blockD, blockD);
    dim3 dimGrid(Width/blockD, Width/blockD);
    MatrixMulKernel<<<dimGrid, dimBlock>>>( Md, Nd,
Pd, Width);
    // transfer P from device
    // get stop time, and display the timing results
    // free the memory allocated on the GPU
    // destroy events to free memory
}
```

```
#include <stdio.h>
#define blockD 32

main(void){
    void MatrixMultiplication(float *, float *, float *,
int);
    const int Width = 1024;
    int size = Width * Width * sizeof(float);
    float *M, *N, *P;
    // allocate memory on the CPU
    for (int y=0; y<Width; y++) {
        for (int x=0; x<Width; x++){ . . . . //
initialize M, N
        }
    }
    MatrixMultiplication(M, N, P, Width);
    // free the memory allocated on the CPU
    return 0;
}
```

Καθορισμός μεγέθους tile

- ▶ Κάθε block νημάτων πρέπει να έχει το δυνατόν περισσότερα νήματα
 - $TILE_WIDTH = 16$ δίνει $16 * 16 = 256$ νήματα
 - $TILE_WIDTH = 32$ δίνει $32 * 32 = 1024$ νήματα
- ▶ Για 16, κάθε block πραγματοποιεί $2 * 256 = 512$ μεταφορές float από την καθολική μνήμη και πραγματοποιεί $256 * (2 * 16) = 8192$ πράξεις
 - 16 πράξεις/μεταφορά
- ▶ Για 32, κάθε block πραγματοποιεί $2 * 1024 = 2048$ μεταφορές float από την καθολική μνήμη και πραγματοποιεί $1024 * (2 * 32) = 65536$ πράξεις
 - 32 πράξεις/μεταφορά