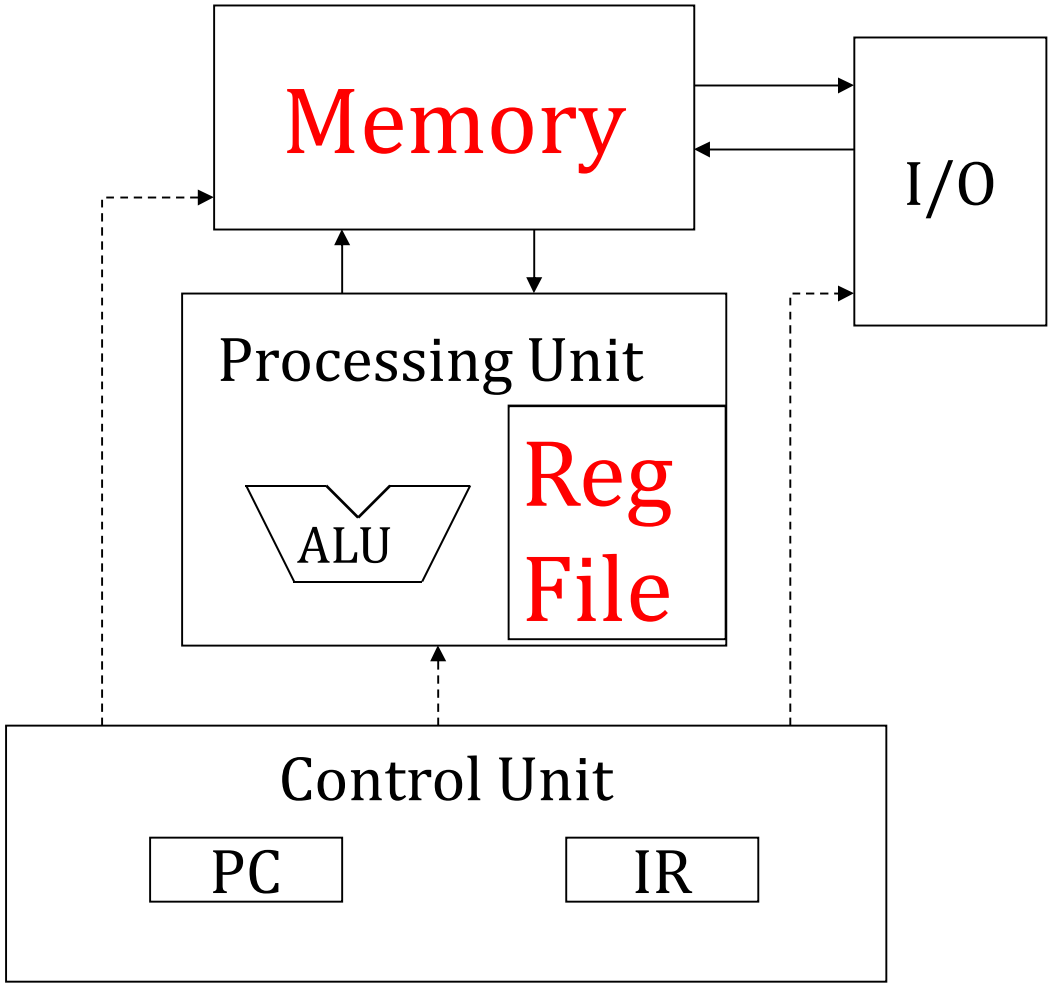
ΠΑΡΑΛΛΗΛΑ ΣΥΣΤΗΜΑΤΑ

Μάθημα #9

CUDA (Μοντέλο Μνήμης)

Το μοντέλο Von-Neumann



Επιστροφή στην εκτέλεση προγράμματος

- Κάθε εντολή πρέπει να προσκομιστεί από την μνήμη, να αποκωδικοποιηθεί και μετά να εκτελεστεί
- Υπάρχουν τριών ειδών εντολές: Εκτέλεση πράξης, Μεταφορά δεδομένων και Ελέγχου ροής του προγράμματος
- Ένα παράδειγμα εκτέλεσης των σταδίων μιας εντολής είναι το παρακάτω:

Fetch | Decode | Execute | Memory

Εντολές εκτέλεσης πράξης

- Παράδειγμα εντολής:

  ADD R1, R2, R3

- Στάδια εκτέλεσης εντολής:

  Fetch | Decode | Execute | Memory

Εντολές μεταφοράς δεδομένων

- Παραδείγματα εντολών:

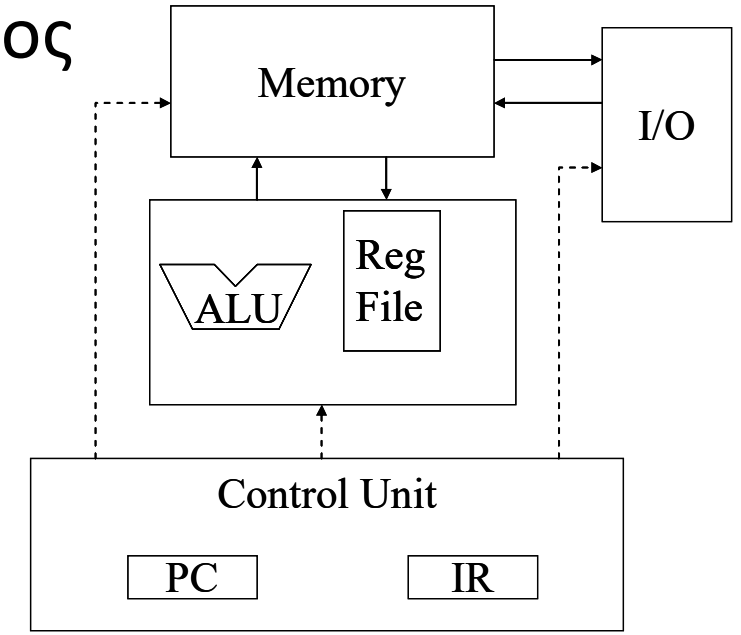
```
LDR   R1, R2, #2
STR   R1, R2, #2
```

- Στάδια εκτέλεσης εντολής:

Fetch | Decode | Execute | Memory

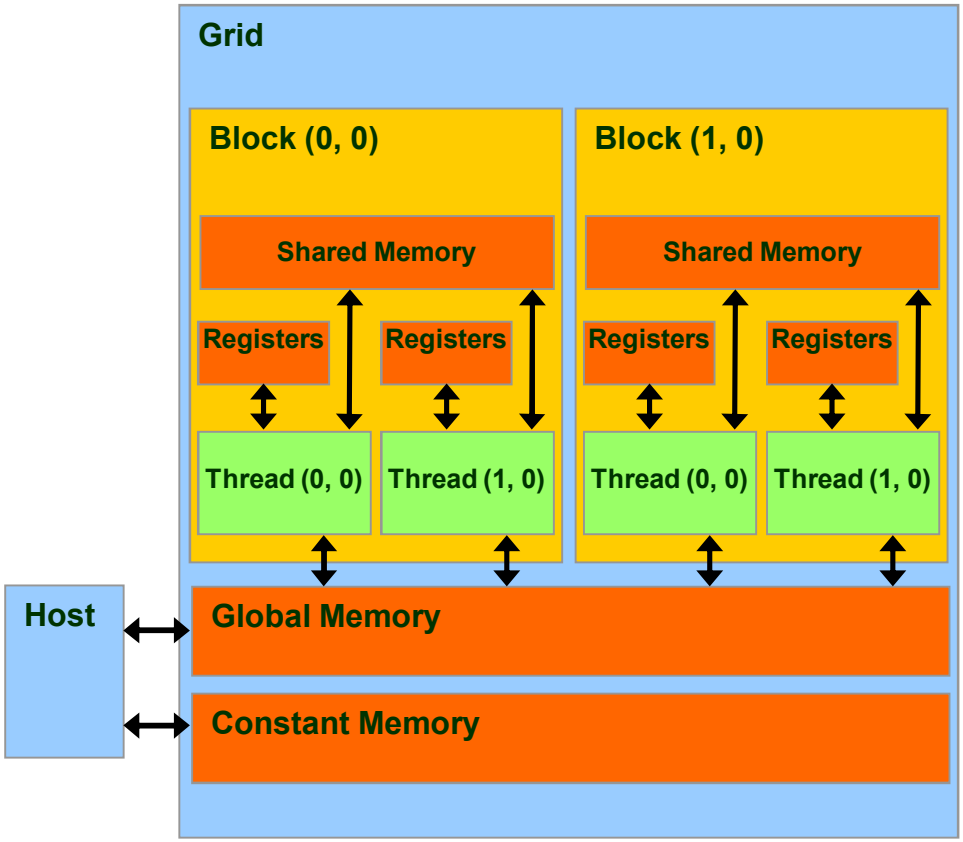
Καταχωρητές vs Μνήμης

- Η χρήση καταχωρητών είναι «δωρεάν»
  - Δεν χρειάζονται επιπλέον εντολές πρόσβασης στη μνήμη
  - Πολύ γρήγοροι στην χρήση, όμως πολύ λίγοι
- Η μνήμη είναι ακριβή (αργή) στην προσπέλαση της, όμως είναι μεγάλη σε μέγεθος



Η ιεραρχία της μνήμης από την σκοπιά του προγραμματιστή

- 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)



Η κοινή μνήμη στην CUDA

- Ειδικός τύπος μνήμης της οποίας τα περιεχόμενα και η προσπέλαση ορίζονται ρητά στον πηγαίο κώδικα
  - Βρίσκεται στον επεξεργαστή
  - Προσπελαύνεται πολύ ταχύτερα από την καθολική (κύρια) μνήμη
  - Προσπελαύνεται όπως και η καθολική μνήμη με εντολές προσπέλασης δεδομένων
  - Αναφέρεται συνήθως ως “local” ή “scratchpad memory” στην αρχιτεκτονική υπολογιστών

Προσδιοριστικά τύπων δεδομένων στην CUDA

Variable declaration	Memory	Scope	Lifetime
`int LocalVar;`	register	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	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

Που πρέπει να ορίζονται οι μεταβλητές;

global
constant

NAI

OXI

register (automatic)
shared
local

Παράδειγμα ορισμού κοινής μνήμης

```
__global__ void MatrixMulKernel(float* d_M, float* d_N,
                                float* d_P, int Width)
{

  1.  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
  2.  __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];
```

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

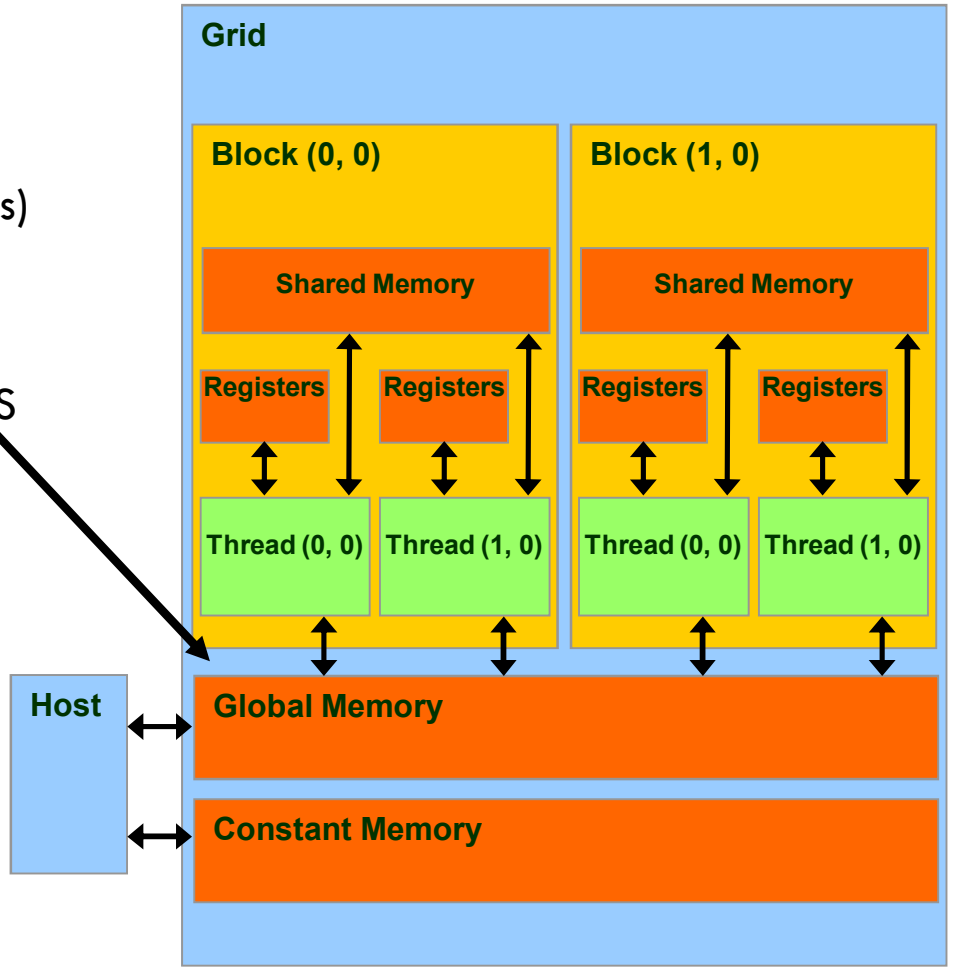
Πολλαπλασιασμός πινάκων με χρήση κοινής μνήμης

Βασικός αλγόριθμος πολλαπλασιασμού μητρώων

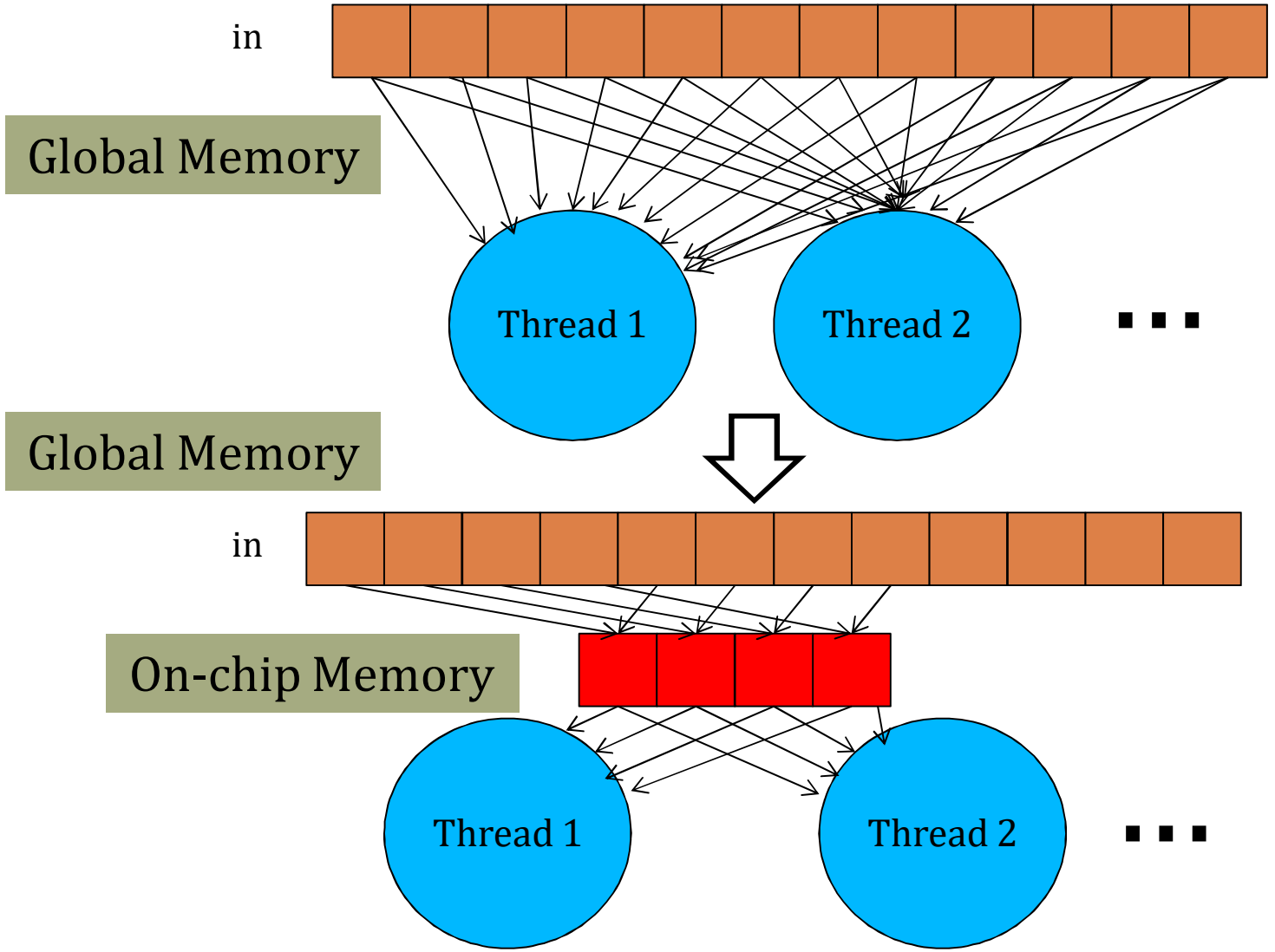
```
__global__ void MatrixMulKernel(float* d_M, float* d_N,
                                float* d_P, int Width)
{
  // Calculate the row index of the Pd element and M
  int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
  // Calculate the column idenx of Pd and N
  int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

  float Pvalue = 0;
  // each thread computes one element of the block sub-matrix
  for (int k = 0; k < Width; k++)
    Pvalue += d_M[Row*Width+k]* d_N[k*Width+Col];

  d_P[Row*Width+Col] = Pvalue;
}
```

Ζητήματα απόδοσης στην αρχιτεκτονική Fermi

- Όλα τα νήματα προσπελαύνουν την καθολική μνήμη για τα στοιχεία των πινάκων εισόδου
  - Δύο προσπελάσεις μνήμης (8 bytes) ανά πολλαπλασιασμό και πρόσθεση αριθμών κινητής υποδιαστολής
  - 4B/s του bandwidth μνήμης/FLOPS
  - 4*1,000 = 4,000 GB/s απαίτηση για επίτευξη μέγιστης απόδοσης
  - 150 GB/s διαθέσιμα περιορίζουν την απόδοση σε 37.5 GFLOPS
- Ο πραγματικός κώδικας επιτυγχάνει περίπου 25 GFLOPS
- Πρέπει να περιοριστούν δραστικά οι προσπελάσεις στην μνήμη για να φτάσουμε κοντά στο μέγιστο των 1,000 GFLOPS



Βασική ιδέα για blocking στην κοινή μνήμη



Βασική ιδέα για Blocking/Tiling

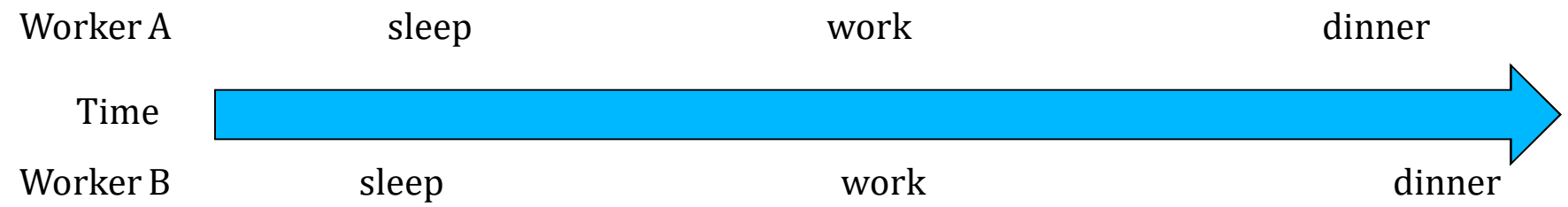
- Σε ένα μποτιλιαρισμένο σύστημα, η δραστική μείωση των κινούμενων οχημάτων οδηγεί σε μειωμένους χρόνους μετακίνησης για τα υπολειπόμενα οχήματα
  - Carpooling για επιβάτες
  - Blocking/Tiling για προσπελάσεις στην καθολική μνήμη
    - οδηγοί = νήματα
    - οχήματα = δεδομένα

Μερικοί τύποι υπολογισμών είναι ευκολότεροι στην εφαρμογή blocking/tiling

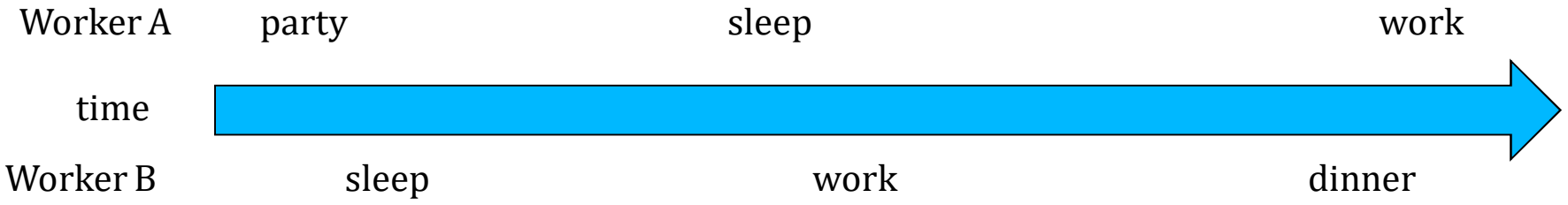
- Μερικοί τύποι carpooling μπορεί να είναι ευκολότεροι από άλλους
  - Πιο αποδοτικοί αν οι γείτονες είναι συμφοιτητές ή συνάδελφοι
  - Μερικοί τύποι οχημάτων μπορεί να είναι πιο κατάλληλοι για carpooling
- Αντίστοιχες διαφοροποιήσεις υπάρχουν κατά την εφαρμογή του blocking/tiling

Το carpooling χρειάζεται συγχρονισμό

- Καλό – όταν οι συνεπιβάτες έχουν αντίστοιχα ωράρια

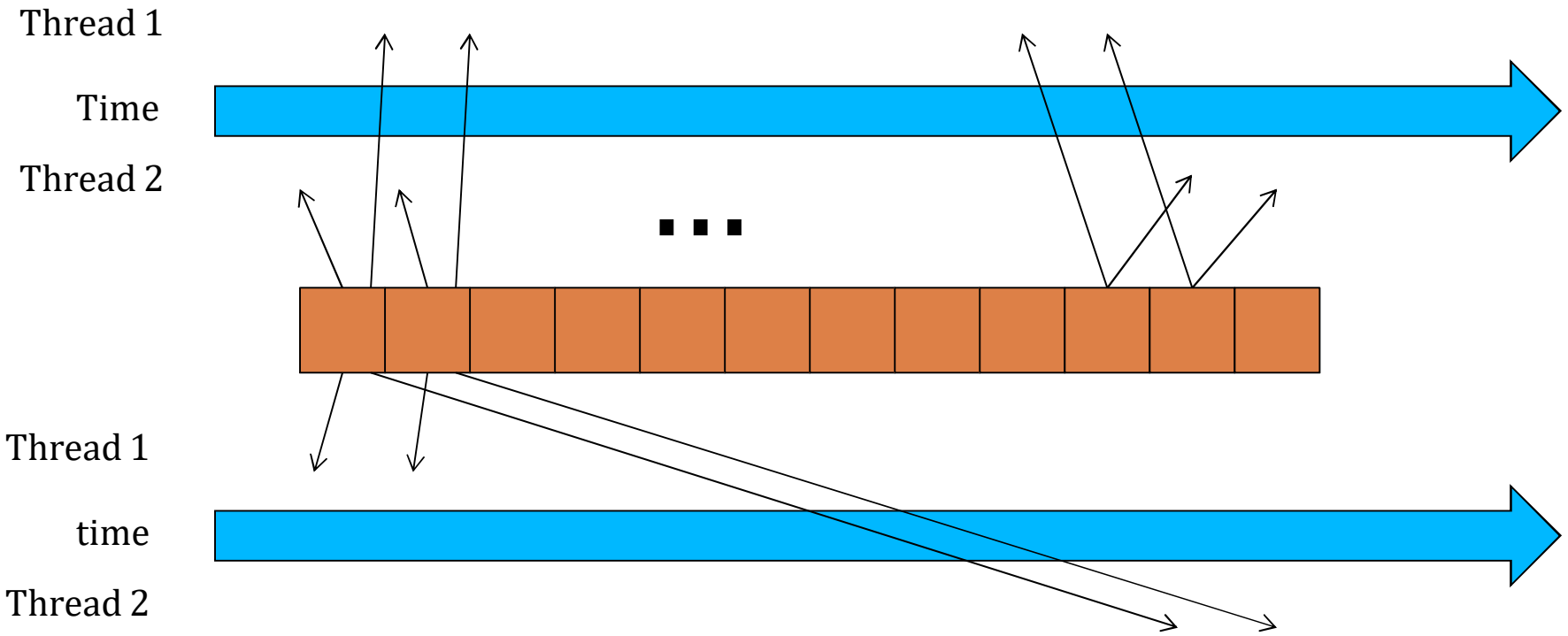


- Κακό – όταν οι συνεπιβάτες έχουν διαφορετικά ωράρια



Ίδια περίπτωση στο Blocking/Tiling

- Καλό – όταν τα νήματα κάνουν προσπελάσεις με μικρές χρονικές διαφορές



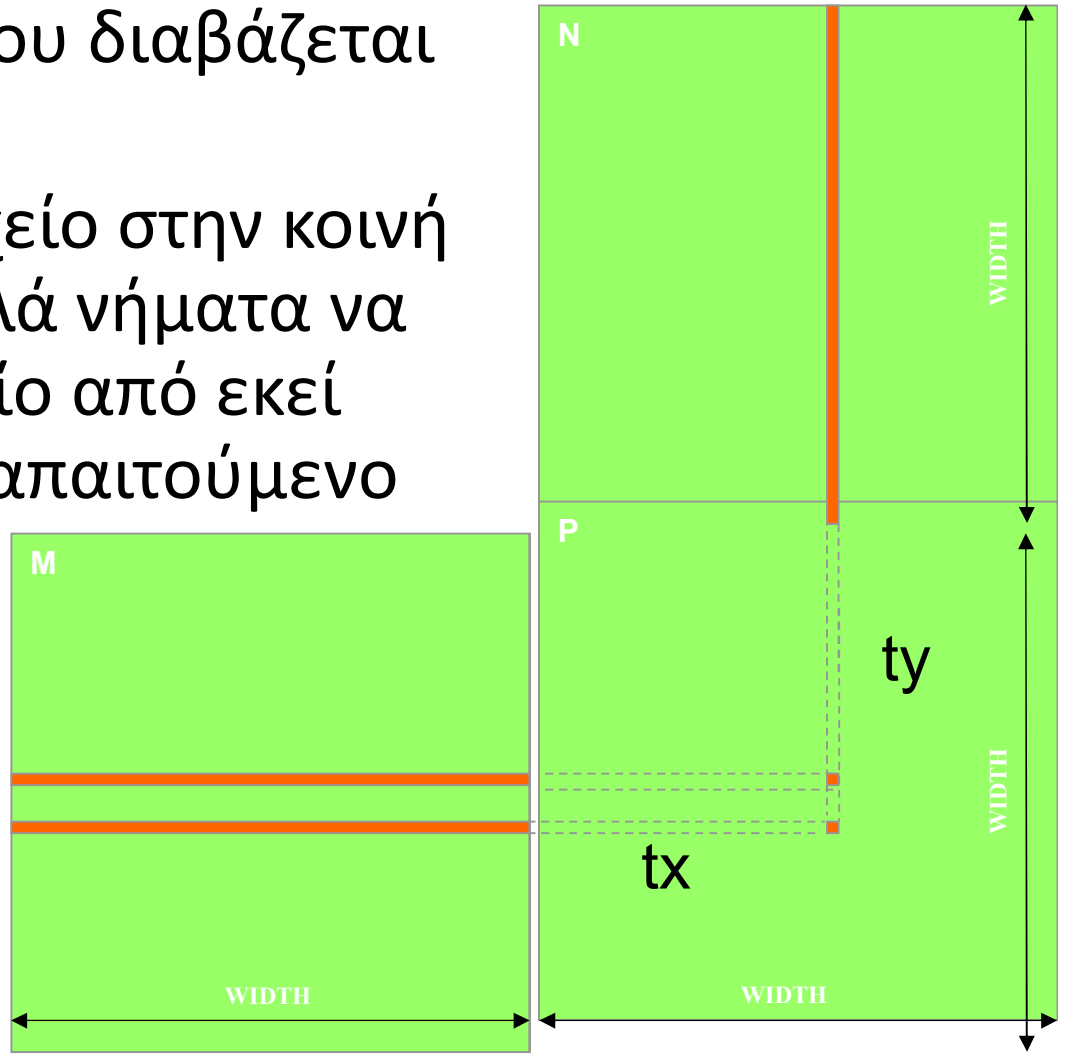
- Κακό – όταν τα νήματα κάνουν προσπελάσεις με μεγάλες χρονικές διαφορές

Συνοπτική περιγραφή της τεχνικής

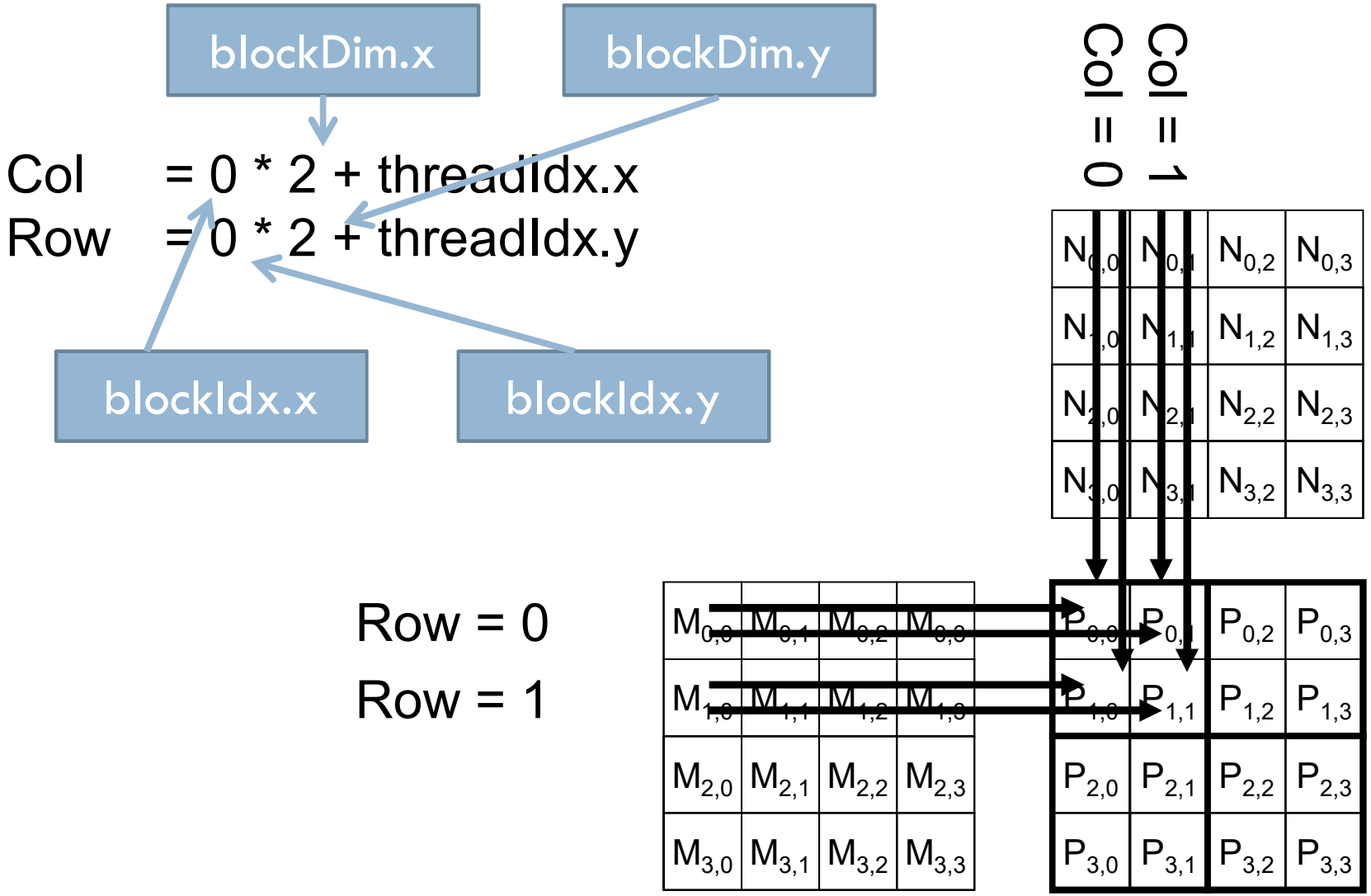
- Βρες ένα block/tile της καθολικής μνήμης το οποίο προσπελαύνεται από πολλά νήματα
- Αντέγραψε το block/tile από την καθολική στην κοινή μνήμη
- Βάλε τα νήματα να προσπελαύνουν τα δεδομένα που χρειάζονται από την κοινή μνήμη
- Πήγαινε στο επόμενο block/tile

Ιδέα: Χρησιμοποίησε κοινή μνήμη για να επαναχρησιμοποιήσεις δεδομένα

- 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

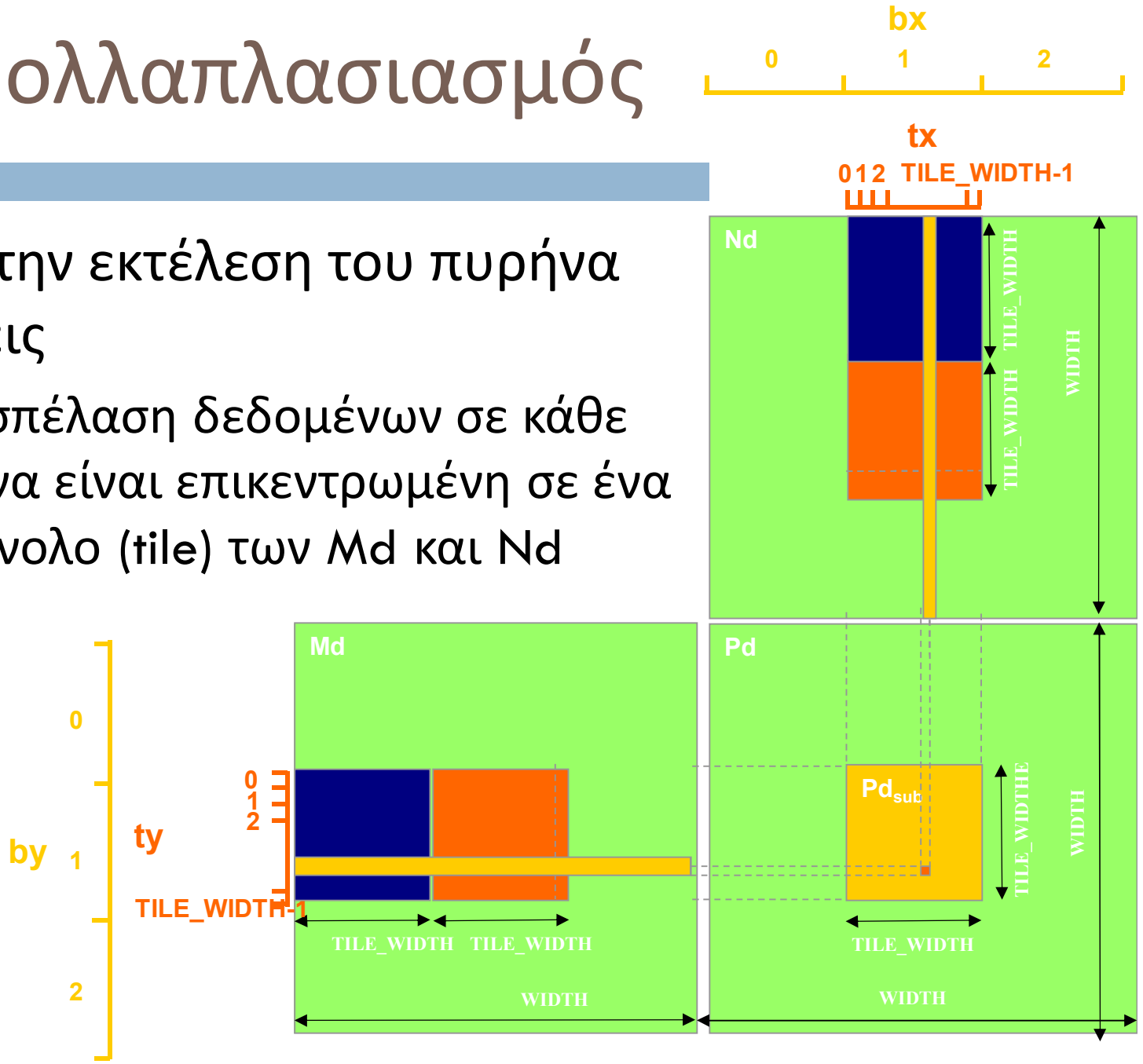


Επεξεργασία block (0, 0) για TILE_WIDTH = 2



Tiled Πολλαπλασιασμός

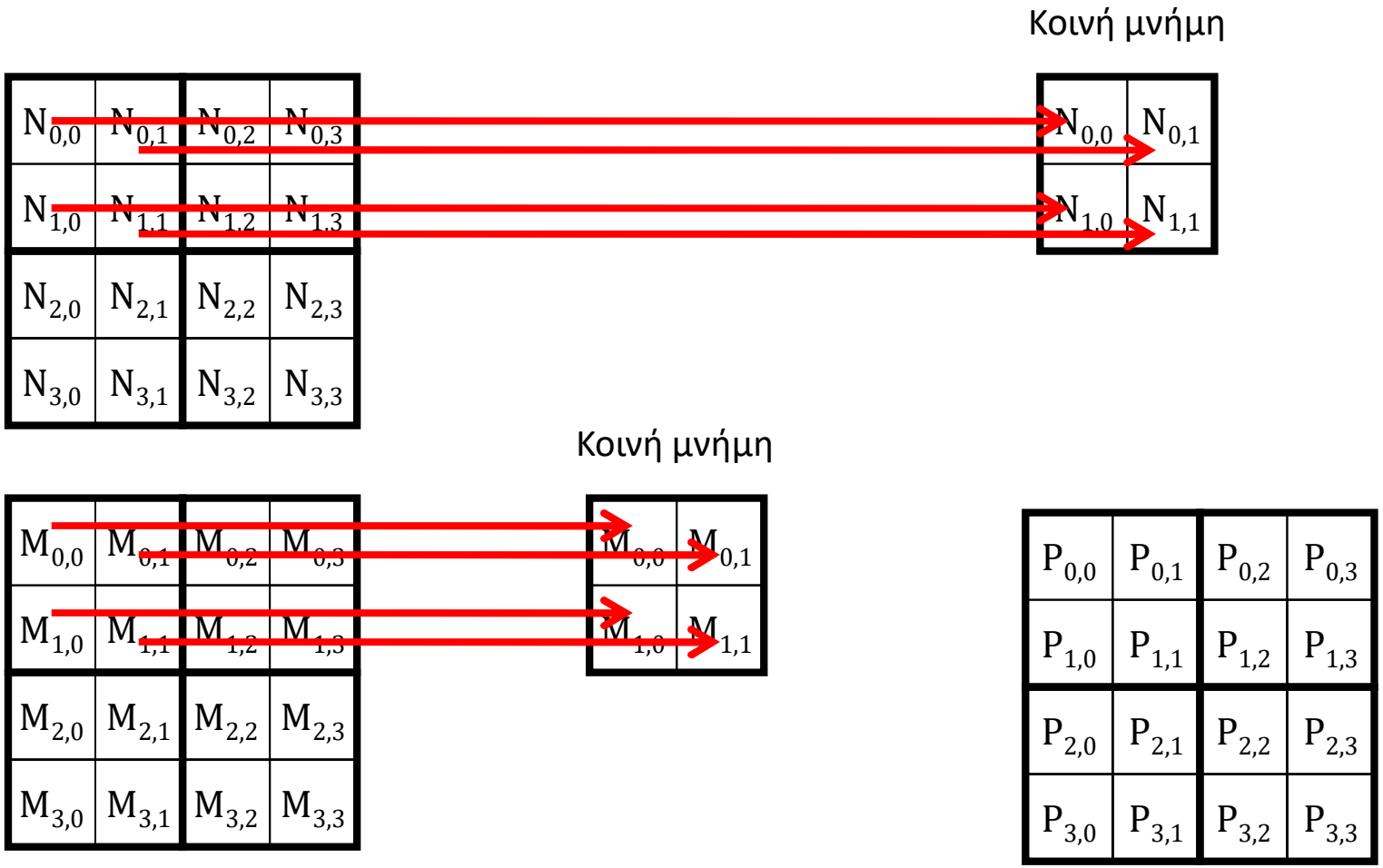
- Χώρισε την εκτέλεση του πυρήνα σε φάσεις
  - Η προσπέλαση δεδομένων σε κάθε φάση να είναι επικεντρωμένη σε ένα υποσύνολο (tile) των Md και Nd



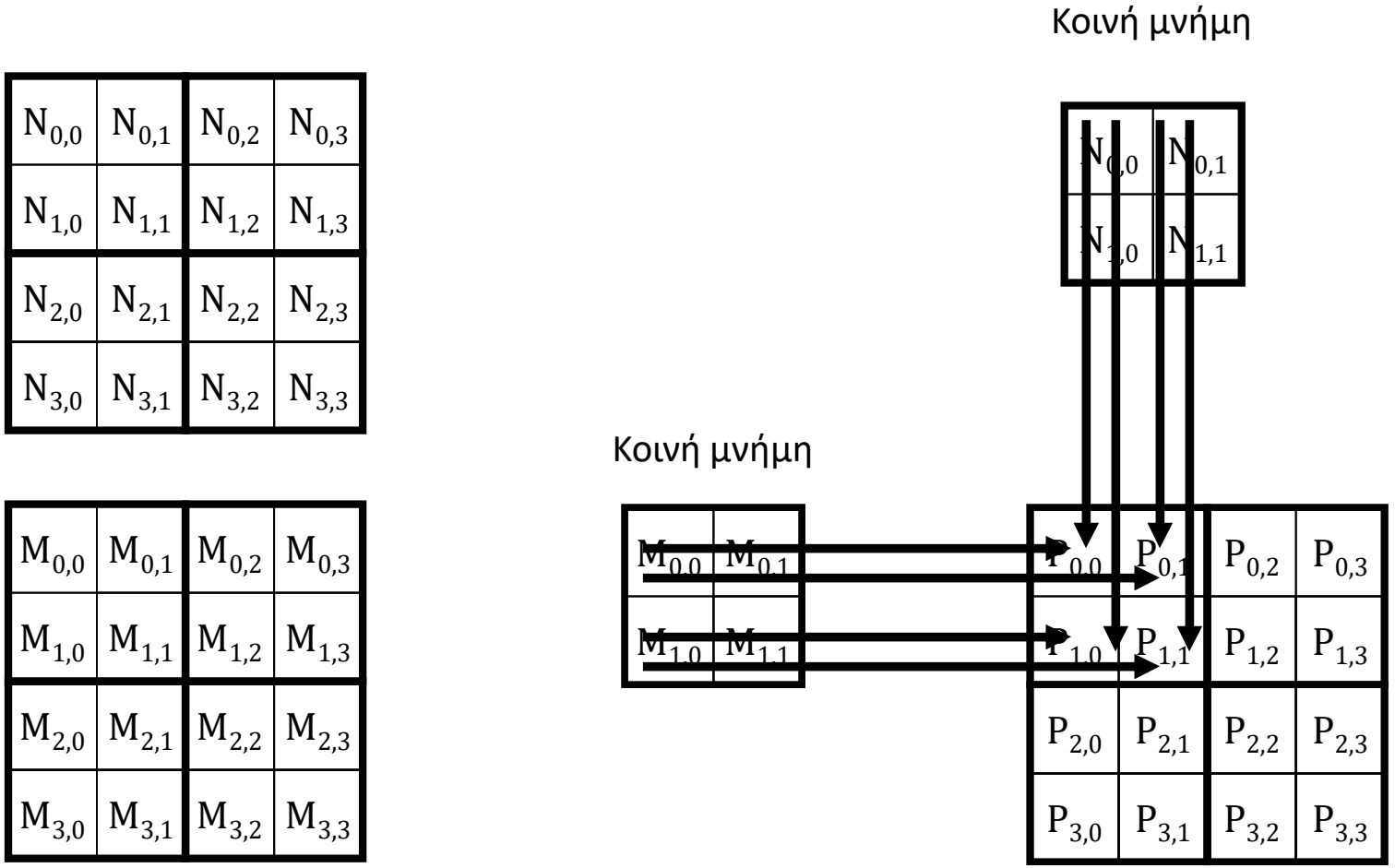
Αντιγραφή tile στην κοινή μνήμη

- Όλα τα νήματα του block συμμετέχουν
  - Κάθε νήμα αντιγράφει ένα στοιχείο του Md και ένα στοιχείο του Nd
- Αξιοποίησε το αντεγραμμένο στοιχείο σε κάθε νήμα έτσι ώστε οι προσπελάσεις μνήμης σε κάθε warp να είναι συνεχόμενες (coalesced)
  - Περισσότερα σε λίγο

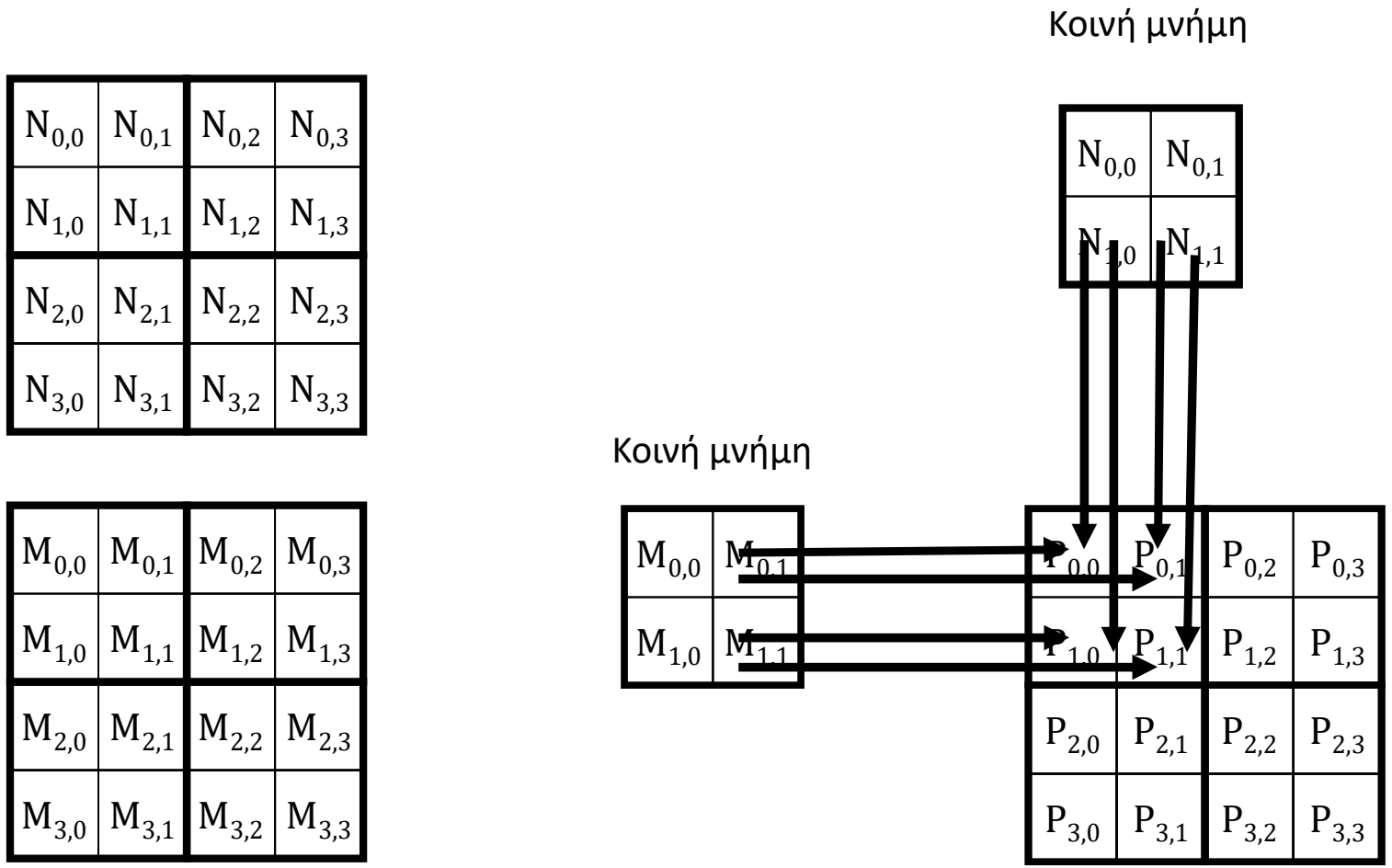
Επεξεργασία Block (0,0)



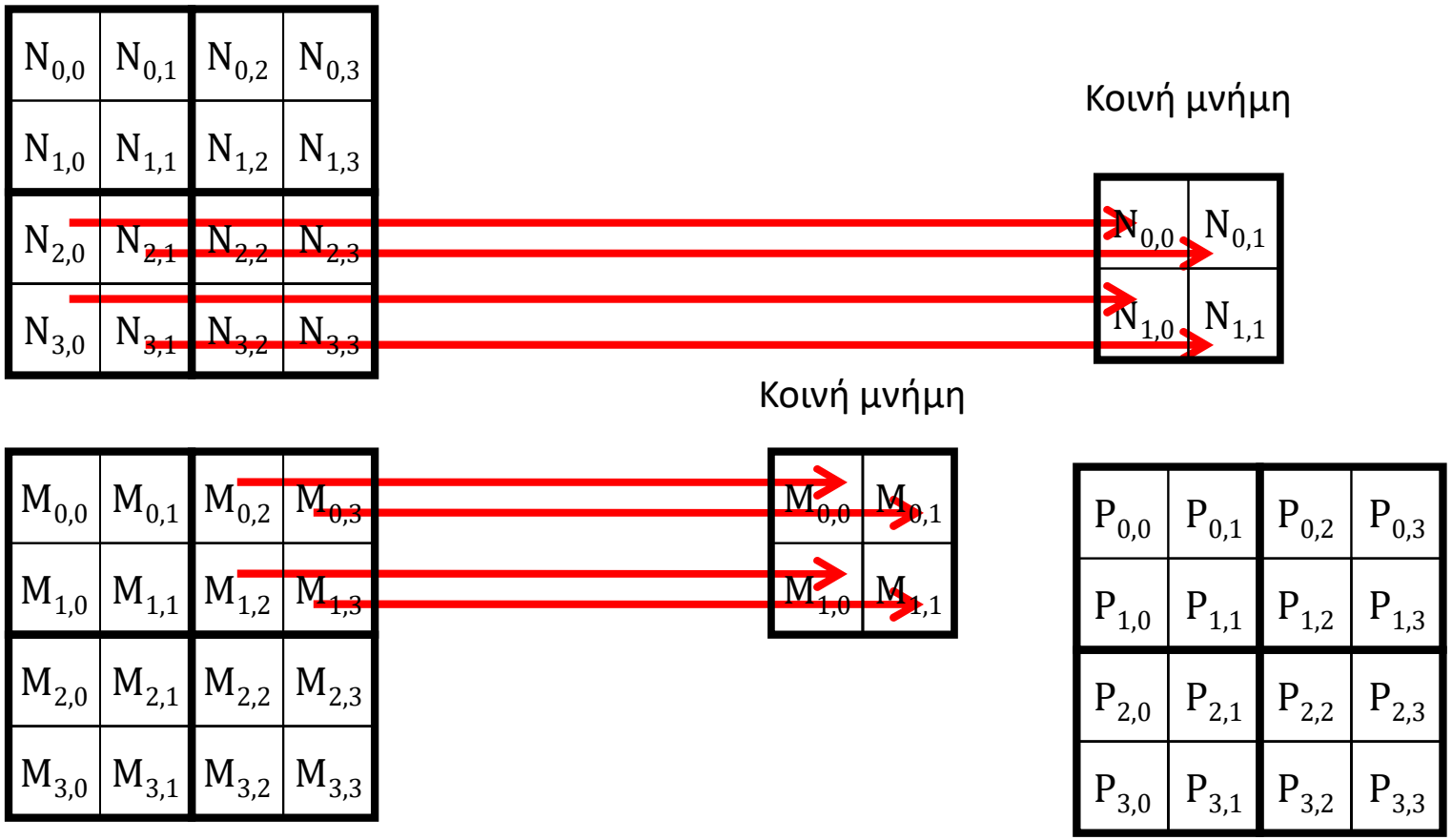
Επεξεργασία Block (0,0)



Επεξεργασία Block (0,0)



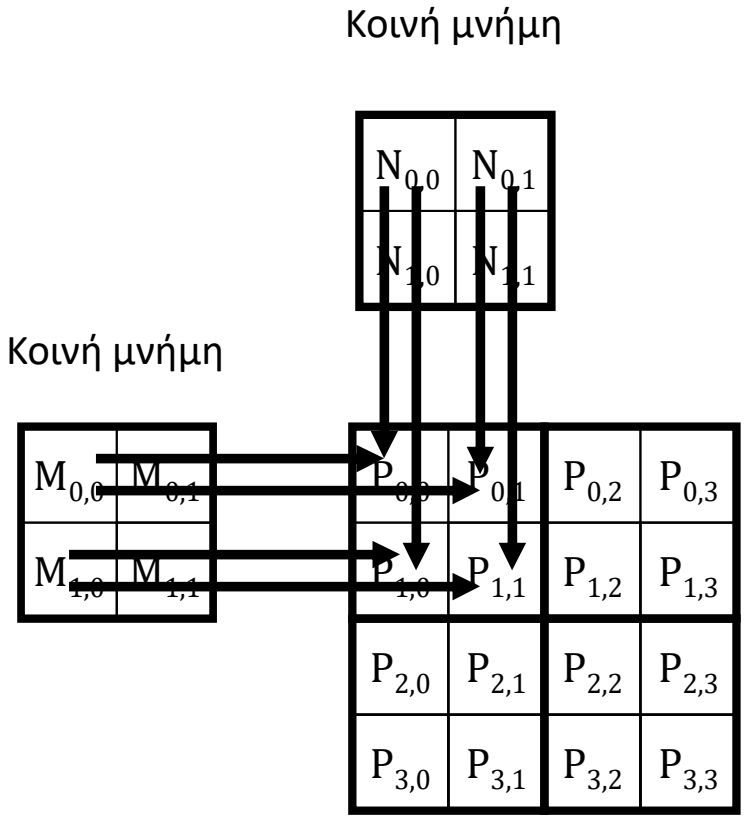
Επεξεργασία Block (0,0)



Επεξεργασία Block (0,0)

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

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



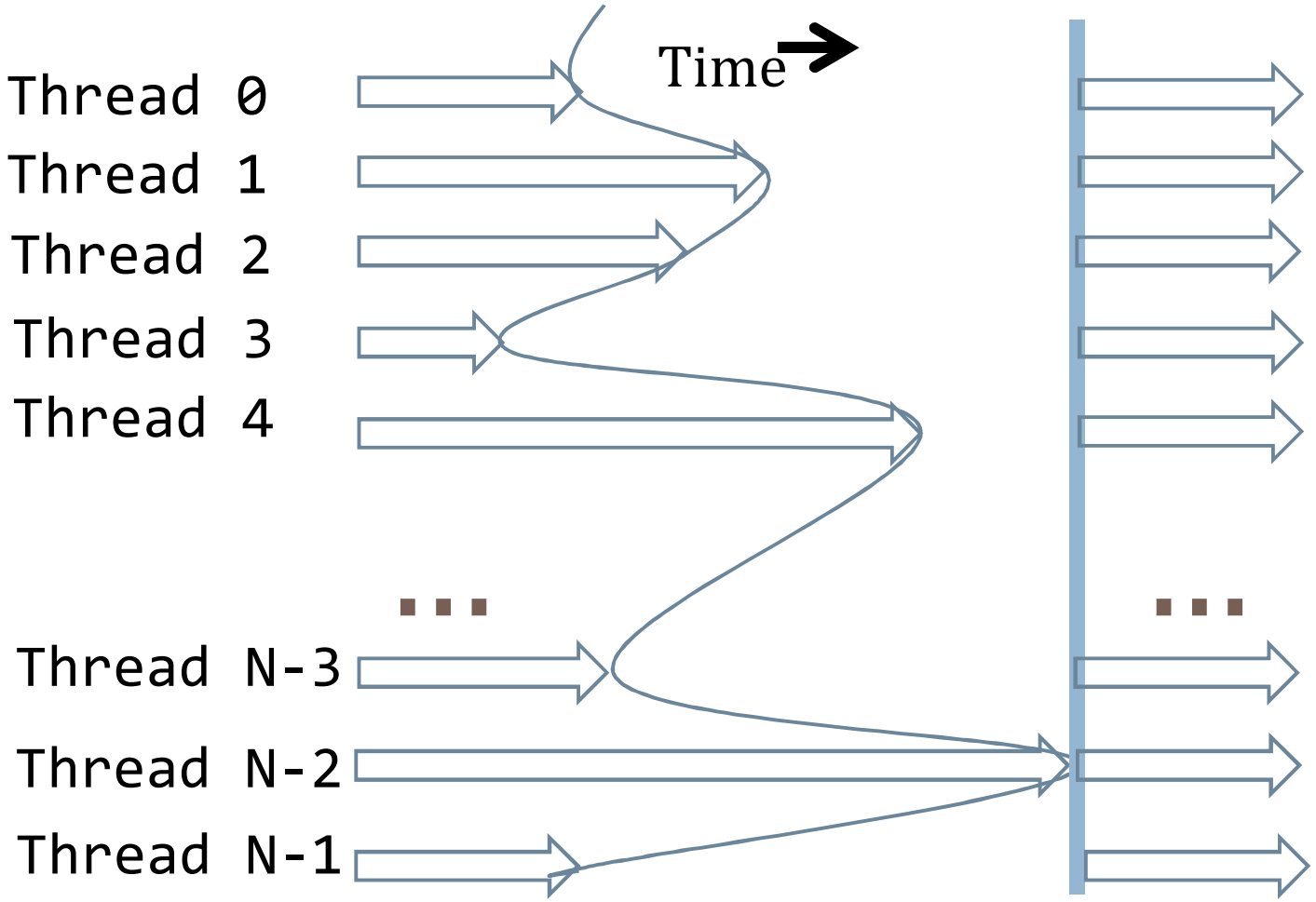
Φράγματα (Barrier)

- Κλήση συνάρτησης στην διεπαφή της CUDA
  - __syncthreads()

- Όλα τα νήματα ενός block πρέπει να καλέσουν την __syncthreads() προτού μπορέσουν να συνεχίσουν την εκτέλεση τους

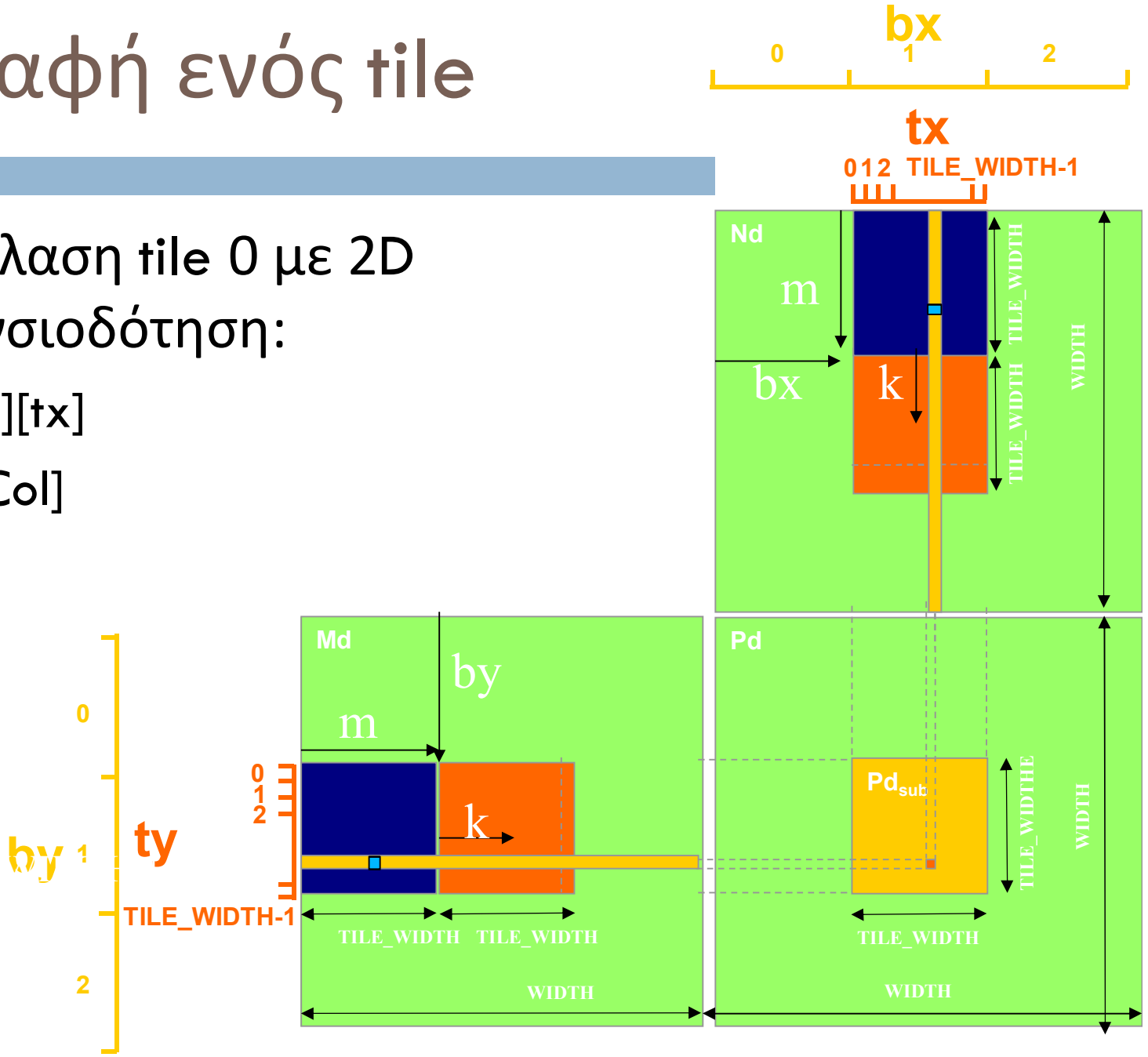
- Χρειάζεται σε αλγόριθμους που χρησιμοποιούν tiles
  - Εξασφάλιση πως όλα τα στοιχεία του tile φορτώθηκαν
  - Εξασφάλιση πως όλα τα στοιχεία του tile χρησιμοποιήθηκαν

Παράδειγμα φράγματος



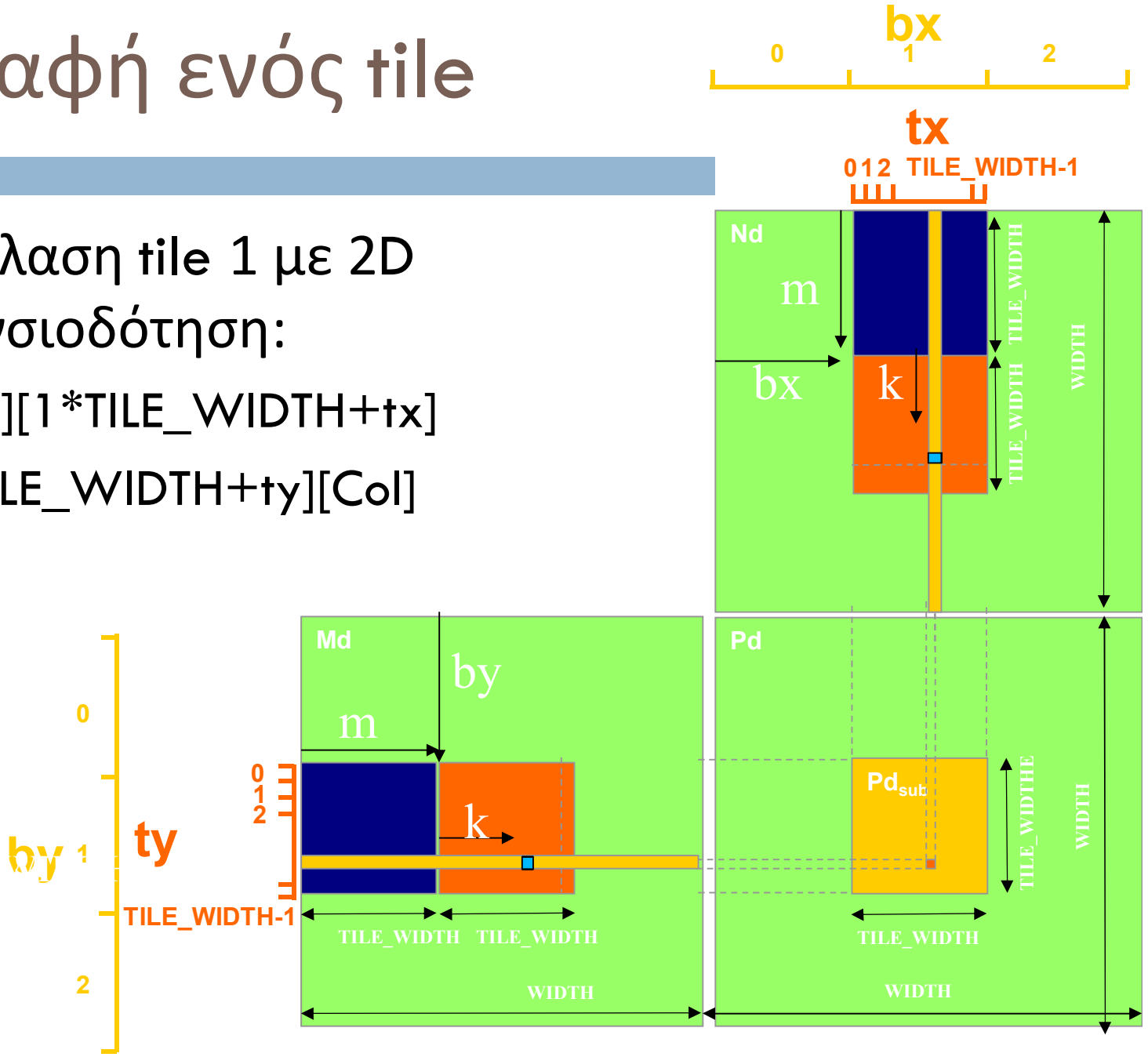
Αντιγραφή ενός tile

- Προσπέλαση tile 0 με 2D διευθυνσιοδότηση:
  - M[Row][tx]
  - N[ty][Col]



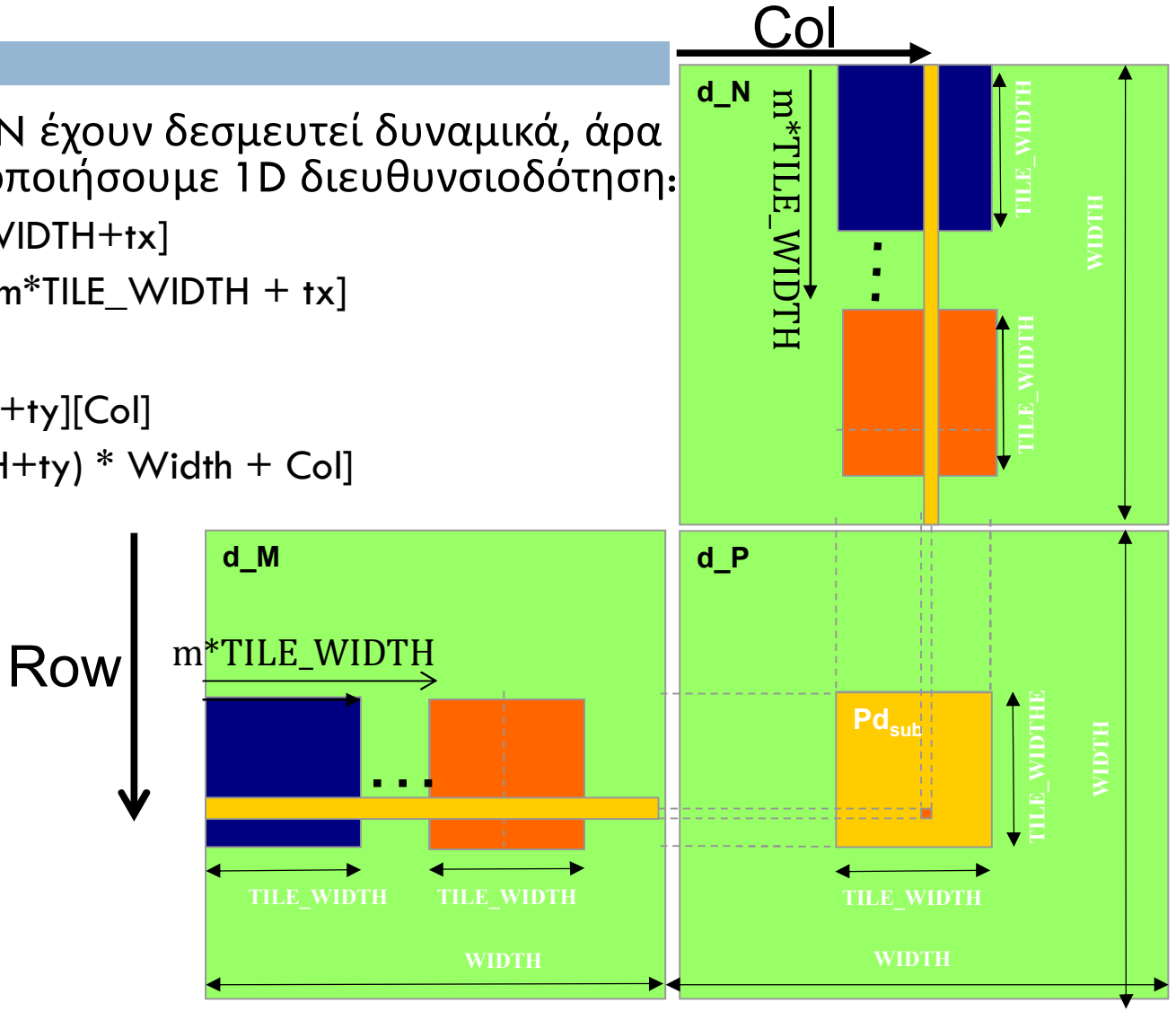
Αντιγραφή ενός tile

- Προσπέλαση tile 1 με 2D διευθυνσιοδότηση:
  - M[Row][1*TILE_WIDTH+tx]
  - N[1*TILE_WIDTH+ty][Col]



Αντιγραφή του tile m

- Τα μητρώα M και N έχουν δεσμευτεί δυναμικά, άρα πρέπει να χρησιμοποιήσουμε 1D διευθυνσιοδότηση:
  - M[Row][m*TILE_WIDTH+tx]
  - M[Row*Width + m*TILE_WIDTH + tx]
  - N[m*TILE_WIDTH+ty][Col]
  - N[(m*TILE_WIDTH+ty) * Width + Col]



Πολλαπλασιασμός μητρώων με tiles

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
1.  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

3.  int bx = blockIdx.x;
4.  int by = blockIdx.y;
5.  int tx = threadIdx.x;
6.  int ty = threadIdx.y;

    // Identify the row and column of the Pd element to work on
7.  int Row = by * TILE_WIDTH + ty;
8.  int Col = bx * TILE_WIDTH + tx;
9.  float Pvalue = 0;
```

Πολλαπλασιασμός μητρώων με tiles

```
    // Loop over the Md and Nd tiles required to compute the Pd element
10. for (int m = 0; m < Width/TILE_WIDTH; m++) {
      // Coolaborative loading of Md and Nd tiles into shared memory
11.   ds_M[ty][tx] = d_M[Row*Width + m*TILE_WIDTH+tx];
      ds_N[ty][tx] = d_N[Col+(m*TILE_WIDTH+ty)*Width];
      __syncthreads();
12.   for (int k = 0; k < TILE_WIDTH; k++)
13.     Pvalue += ds_M[ty][k] * ds_N[k][tx];
14.   __synchthreads();
15. }
16. d_P[Row*Width+Col] = Pvalue;
}
```

Σύγκριση με την αρχική συνάρτηση πυρήνα

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
  // Calculate the row index of the Pd element and M
  int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
  // Calculate the column idenx of Pd and N
  int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

  float Pvalue = 0;
  // each thread computes one element of the block sub-matrix
  for (int k = 0; k < Width; ++k)
    Pvalue += d_M[Row*Width+k]* d_N[k*Width+Col];

  d_P[Row*Width+Col] = Pvalue;
}
```

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

Κοινή μνήμη και νήματα

- Κάθε SM στην αρχιτεκτονική Fermi έχει 16KB ή 48KB κοινής μνήμης
  - Καθορίζεται μαζί με την L1, σύνολο 64KB
  - Το μέγεθος σε κάθε SM εξαρτάται από την υλοποίηση!
- Για TILE_WIDTH = 16, κάθε νήμα χρησιμοποιεί 2*256*4B = 2KB κοινής μνήμης
  - Μπορούμε να έχουμε δυνητικά μέχρι 8 block νημάτων ενεργά
    - Οδηγεί μέχρι 8*512 = 4096 εκκρεμείς αιτήσεις για δεδομένα (2 ανά νήμα, 256 νήματα σε κάθε block)
- Για TILE_WIDTH = 32, κάθε νήμα χρησιμοποιεί 2*32*32*4B= 8KB κοινής μνήμης
  - Μπορούμε να έχουμε δυνητικά μέχρι 2 ή 6 block νημάτων ενεργά
- Χρησιμοποιώντας TILE_WIDTH = 16 μειώνουμε το πλήθος των προσπελάσεων στην καθολική μνήμη 16 φορές
  - Το διαθέσιμο εύρος ζώνης των 86.4GB/s μπορεί να υποστηρίξει τώρα (86.4/4)*16 = 347.6 GFLOPS!

Πλήθος/Δυνατότητες device

- Πλήθος device στο σύστημα
  - `int dev_count;`
  - `cudaGetDeviceCount(&dev_count);`
- Δυνατότητες κάθε device
  - `cudaDeviceProp  dev_prop;`
  - `for (i = 0; i < dev_count; i++) {`
  - `  cudaGetDeviceProperties( &dev_prop, i);`
  - `// decide if device has sufficient resources/capabilities`
  - `}`
- Η cudaDeviceProp είναι μια απλή δομή της C
  - `dev_prop.dev_prop.maxThreadsPerBlock`
  - `dev_prop.sharedMemoryPerBlock`
  - …

Ανακεφαλαίωση – Τυπική δομή ενός προγράμματος CUDA

- [redacted]
- Πρότυπα συναρτήσεων
  - __global__ void kernelOne(…)
  - float handyFunction(…)
- main ()
  - δέσμευση μνήμης στο device – cudaMalloc([redacted] bytes)
  - μεταφορά δεδομένων από host στο device – cudaMemCpy([redacted]h_Gl…)
  - αρχικοποίηση περιβάλλοντος εκτέλεσης
  - κλήση συνάρτησης πυρήνα – kernelOne<<<execution configuration>>>(args…)
  - μεταφορά δεδομένων από device στο host – cudaMemCpy(h_GlblVarPtr,…)
  - προαιρετικά: σύγκριση αποτελεσμάτων με λύση που έχει υπολογιστεί στον host

Επανέλαβε όσες φορές χρειάζεται

- Συνάρτηση πυρήνα – void kernelOne(type args,…)
  - [redacted]
  - __syncthreads()…
- Άλλες συναρτήσεις
  - float handyFunction(int inVar…);

Περισσότερες ερωτήσεις; Διαβάστε το Κεφάλαιο 5!