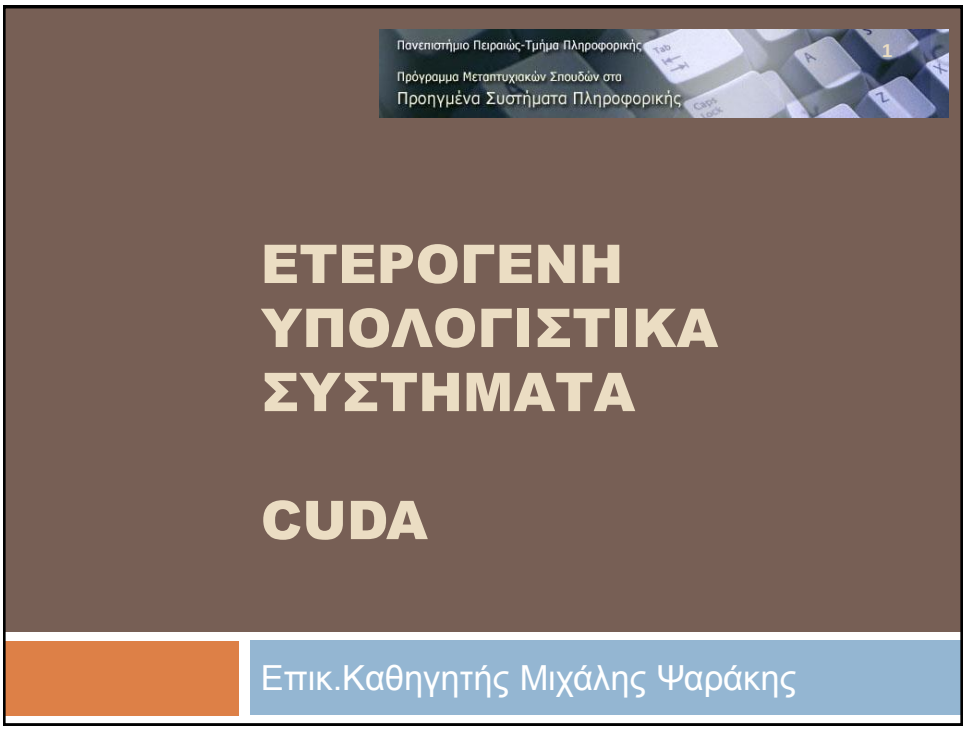


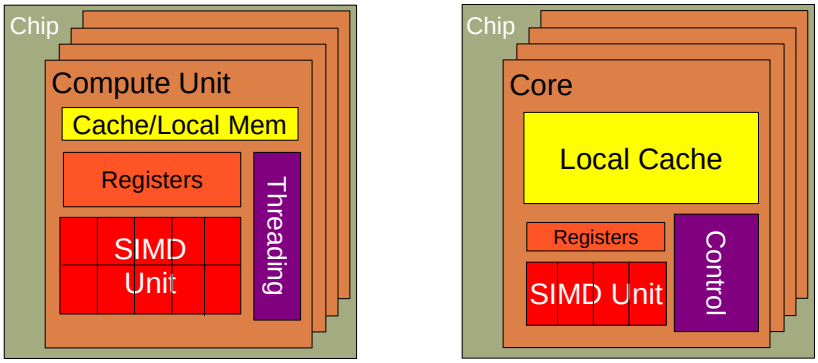
Ενότητα 1

- Εισαγωγή στην CUDA
- Παράδειγμα: πρόσθεση διανυσμάτων

CPU και GPU έχουν διαφορετική σχεδιαστική φιλοσοφία

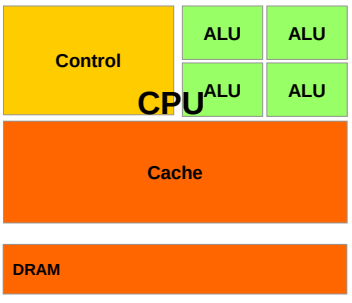
GPU
Throughput Oriented Cores

CPU
Latency Oriented Cores



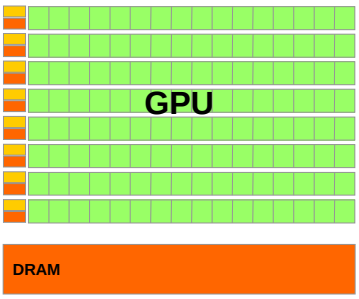
CPUs: Έμφαση στην καθυστέρηση (latency)

- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency
- Powerful ALU
  - Reduced operation latency



GPUs: έμφαση στην ικανότητα διεκπεραίωσης (throughput)

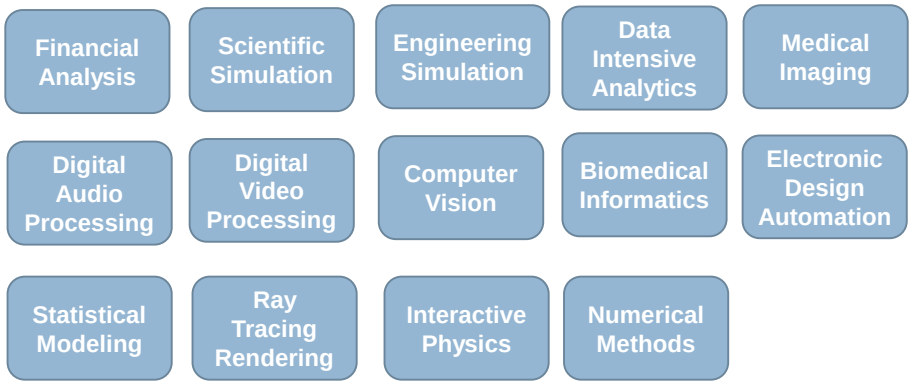
- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies



Ετερογενής υπολογιστική: χρήση CPU και GPU

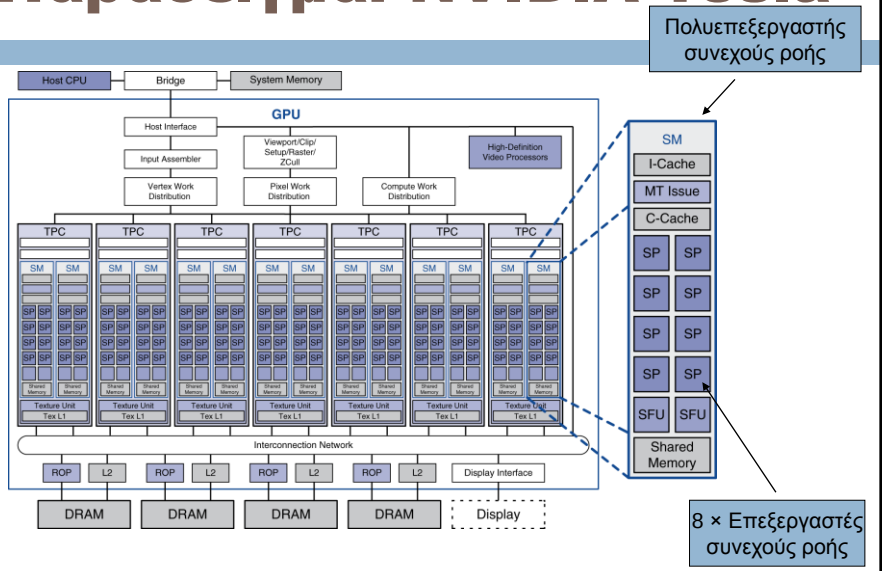
- CPUs for sequential parts where latency matters
  - CPUs can be 10+X faster than GPUs for sequential code
- GPUs for parallel parts where throughput wins
  - GPUs can be 10+X faster than CPUs for parallel code

Η ετερογενής παράλληλη υπολογιστική είναι μεταδοτική



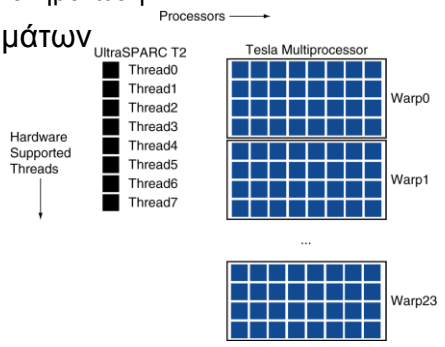
- 280 submissions to GPU Computing Gems
  - 110 articles included in two volumes

Παράδειγμα: NVIDIA Tesla



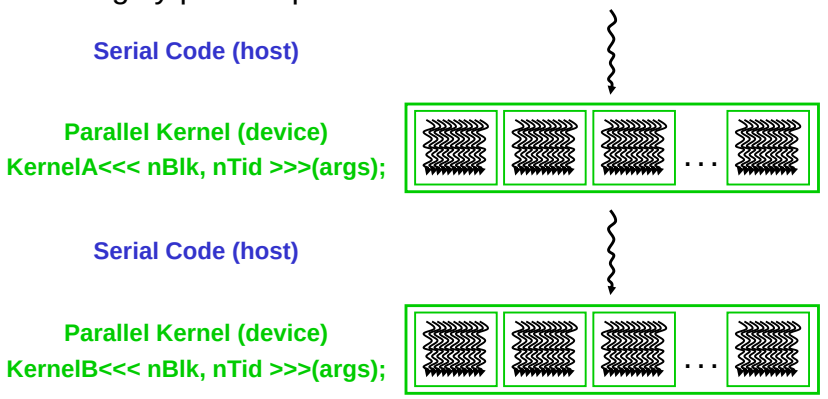
Παράδειγμα: NVIDIA Tesla

- Επεξεργαστές συνεχούς ροής (Streaming Processors – SP)
  - Μονάδες FP απλής ακρίβειας και ακέραιες μονάδες
  - Κάθε SP υποστηρίζει λεπτή πολυνημάτωση
- Στημόνι (warp): μπλοκ 32 νημάτων
  - Εκτελούνται παράλληλα
    - 8 SPs × 4 κύκλους ρολογιού
  - Υποστήριξη υλικού για 24 στημόνια
    - Καταχωρητές, PCs, …



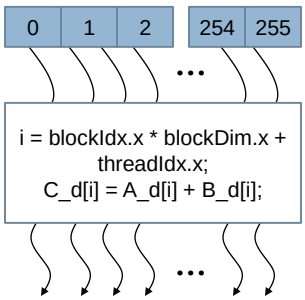
CUDA C – Μοντέλο εκτέλεσης

- Integrated host+device app C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD kernel C code



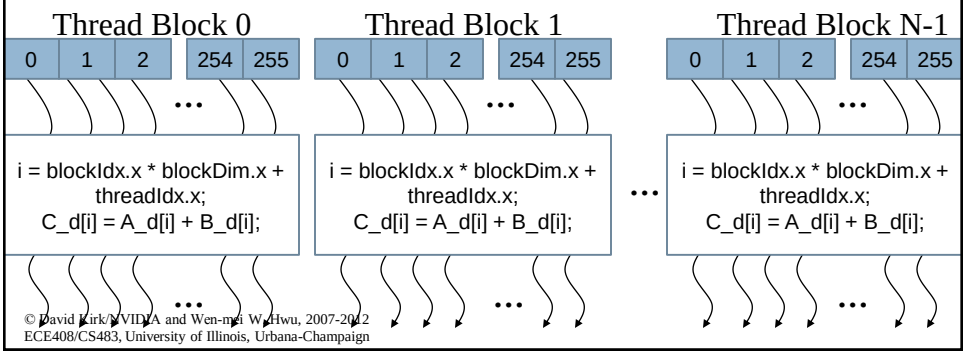
Πλέγμα παράλληλων νημάτων

- A CUDA kernel is executed by a grid (array) of threads
  - All threads in a grid run the same kernel code (SPMD)
  - Each thread has an index that it uses to compute memory addresses and make control decisions



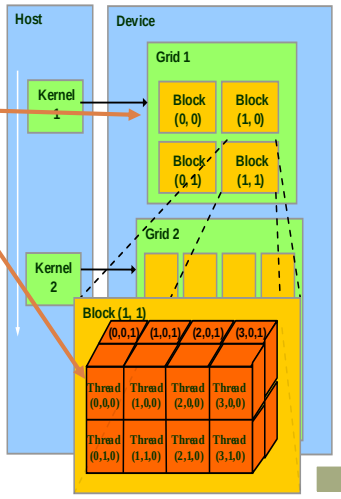
Μπλοκ νημάτων: Επεκτασιμότητα

- Divide thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks cannot cooperate

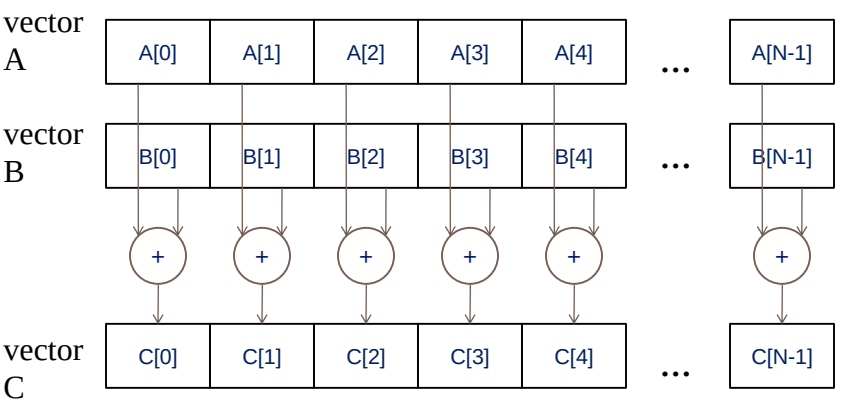


blockIdx και threadIdx

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
  - threadIdx: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data



Πρόσθεση διανυσμάτων– εννοιολογική άποψη

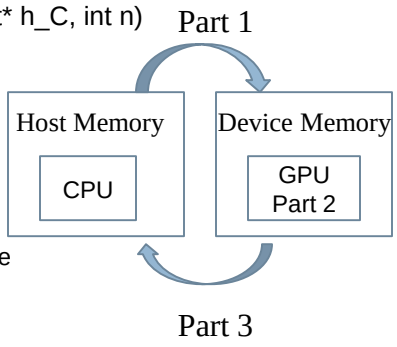


Πρόσθεση διανυσμάτων – Παραδοσιακός κώδικας C

```
// Compute vector sum C = A+B
void vecAdd(float* A, float* B, float* C, int n)
{
  for (i = 0; i < n; i++)
    C[i] = A[i] + B[i];
}

int main()
{
    // Memory allocation for A_h, B_h, and C_h
    // I/O to read A_h and B_h, N elements
    …
    vecAdd(A_h, B_h, C_h, N);
}
```

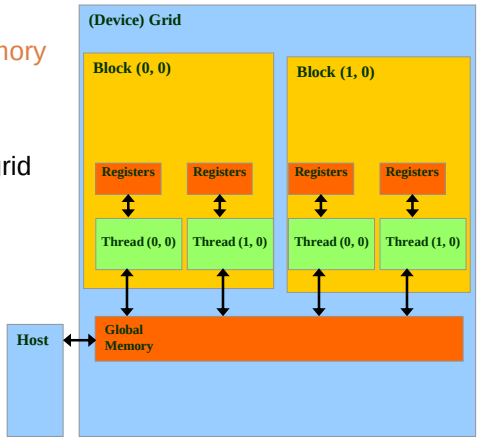

Ετερογενής υπολογιστική: vecAdd CUDA Host Code

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  int size = n* sizeof(float);
  float* d_A, d_B, d_C;
  …
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

2. // Kernel launch code – to have the device
   // to perform the actual vector addition

3. // copy C from the device memory
   // Free device vectors
}
```



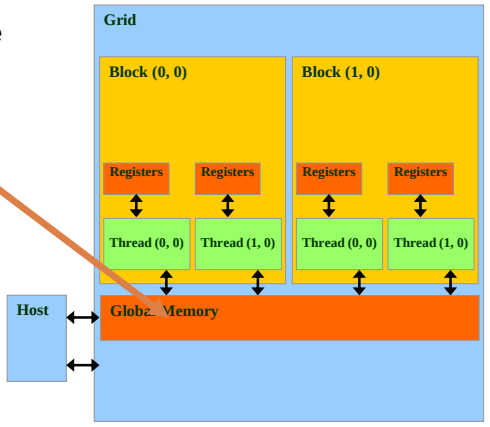
Μερική άποψη των μνημών στην CUDA

- Device code can:
  - R/W per-thread registers
  - R/W all-shared global memory
- Host code can
  - Transfer data to/from per grid global memory



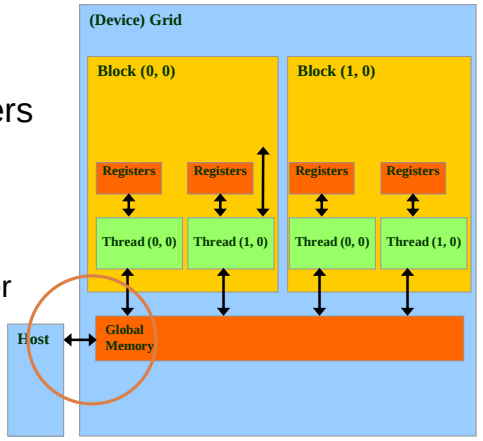
Συναρτήσεις API για την διαχείριση της μνήμης της συσκευής

- cudaMalloc()
  - Allocates object in the device global memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
    - **Pointer** to freed object



Συναρτήσεις API για μεταφορά δεδομένων μεταξύ Host & Device

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer




```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  int size = n * sizeof(float);
   float* d_A, d_B, d_C;

1. // Transfer A and B to device memory
   cudaMalloc((void **) &d_A, size);
   cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
   cudaMalloc((void **) &B_d, size);
   cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Allocate device memory for
    cudaMalloc((void **) &d_C, size);

2.  // Kernel invocation code – to be shown later
   …
3.   // Transfer C from device to host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
     // Free device memory for A, B, C
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

Παράδειγμα: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}

int vectAdd(float* h_A, float* h_B, float* h_C, int n)
{
        // d_A, d_B, d_C allocations and copies omitted
    // Run ceil(n/256.0) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
}
```


Παράδειγμα: Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddkernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}
```

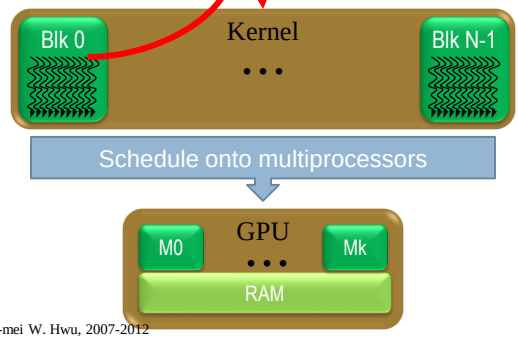
Host Code

```
int vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  // d_A, d_B, d_C allocations and copies omitted
  // Run ceil(n/256.0) blocks of 256 threads each
   vecAddKernnel<<<ceil(n/256.0),256>>>(d_A, d_B, d_C, n);
```

Η εκτέλεση του Kernel

```
__host__
Void vecAdd()
{
  dim3 DimGrid = (ceil(n/256.0),1,1);{
  dim3 DimBlock = (256,1,1);
 vecAddKernel<<<DimGrid,DimBlock>>>
(A_d,B_d,C_d,n);
}
```

```
__global__
void vecAddKernel(float *A_d,
     float *B_d, float *C_d, int n)
{
  int i = blockIdx.x * blockDim.x
      + threadIdx.x;

  if( i<n ) C_d[i] = A_d[i]+B_d[i];
}
```



Δήλωση συναρτήσεων CUDA

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Each "__" consists of two underscore characters
  - A kernel function must return `void`
- `__device__` and `__host__` can be used together

Υπολογιστικές δυνατότητες

Technical specifications	Compute capability (version)							
	1.0	1.1	1.2	1.3	2.x	3.0	3.5	5.0
Maximum dimensionality of grid of thread blocks	2				3			
Maximum x-, y-, or z-dimension of a grid of thread blocks	65535						2^{31}-1	
Maximum dimensionality of thread block	3							
Maximum x- or y-dimension of a block	512				1024			
Maximum z-dimension of a block	64							
Maximum number of threads per block	512				1024			
Warp size	32							
Maximum number of resident blocks per multiprocessor	8						16	32
Maximum number of resident warps per multiprocessor	24		32		48		64	
Maximum number of resident threads per multiprocessor	768		1024		1536		2048	
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K		64 K	
Maximum number of 32-bit registers per thread	128				63		255	
Maximum amount of shared memory per multiprocessor	16 KB				48 KB			64 KB

Ενότητα 2

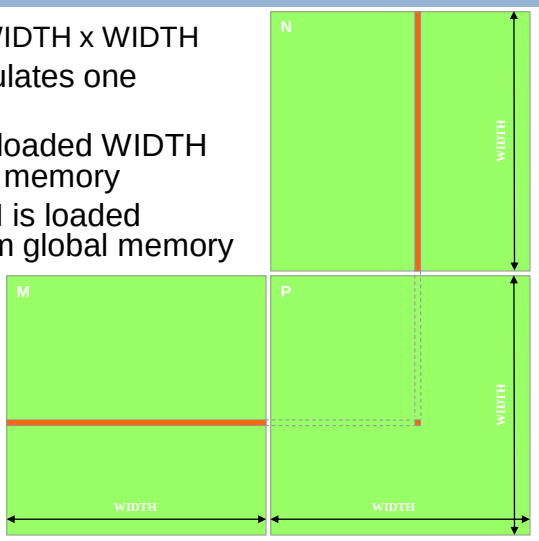
- Τα νήματα της CUDA
- Παράδειγμα: πολλαπλασιασμός πινάκων

Ένα απλό παράδειγμα: πολλαπλασιασμός πινάκων

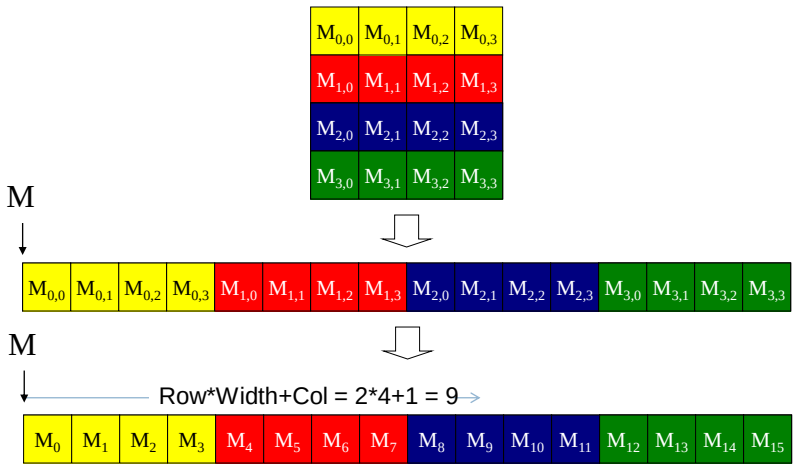
- A simple illustration of the basic features of memory and thread management in CUDA programs
  - Thread index usage
  - Memory layout
  - Register usage
  - Assume square matrix for simplicity
  - Leave shared memory usage until later

Πολλαπλασιασμός πινάκων

- P = M * N of size WIDTH x WIDTH
  - Each thread calculates one element of P
  - Each row of M is loaded WIDTH times from global memory
  - Each column of N is loaded WIDTH times from global memory

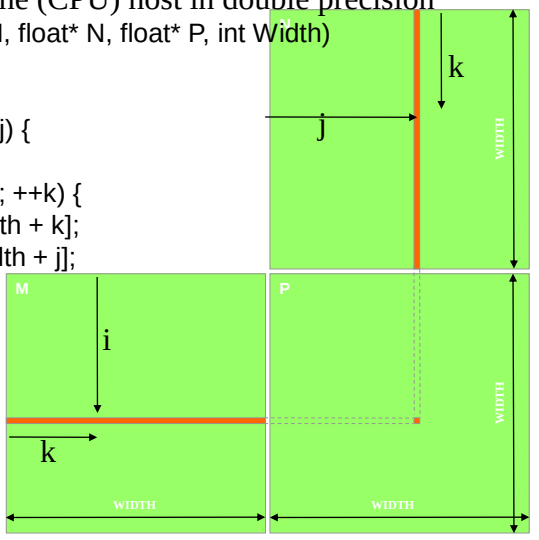


Διάταξη Row-Major στην C/C++



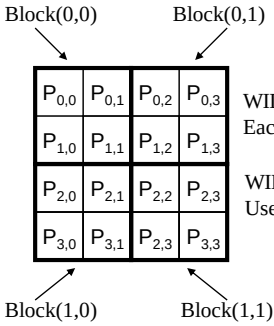
Πολλαπλασιασμός πινάκων Μια απλή έκδοση host C code

```
// Matrix multiplication on the (CPU) host in double 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 Kernel – ένα μικρό παράδειγμα

- Have each 2D thread block to compute a $(TILE\_WIDTH)^2$ sub-matrix (tile) of the result matrix
  - Each has $(TILE\_WIDTH)^2$ threads
- Generate a 2D Grid of $(WIDTH/TILE\_WIDTH)^2$ blocks



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

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

Ένα ελαφρώς 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

Ένα ελαφρώς μεγαλύτερο παράδειγμα, συν.

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

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

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

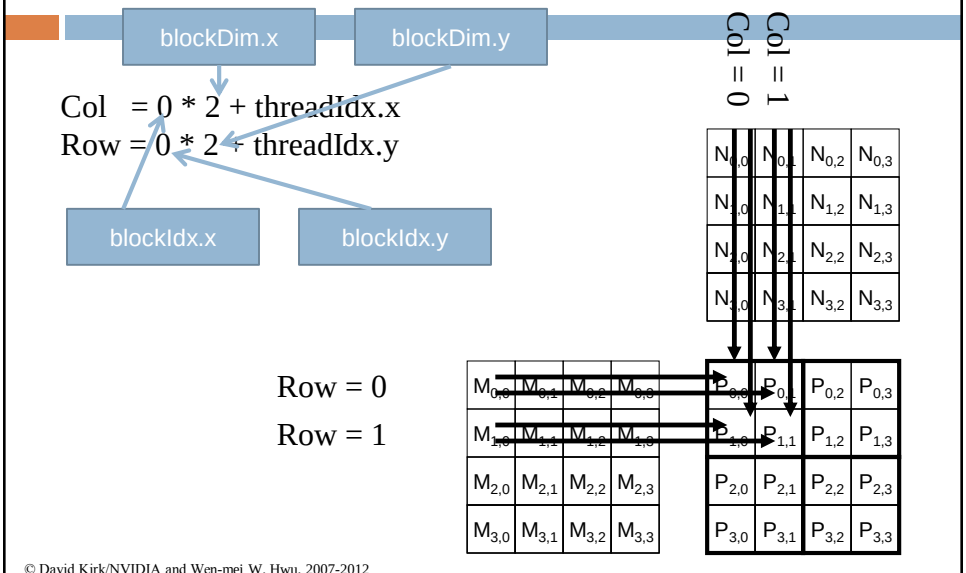
Κλήση Kernel (κώδικας Host)

```
// Setup the execution configuration
// TILE_WIDTH is a #define constant
    dim3 dimGrid(Width/TILE_WIDTH, Width/TILE_WIDTH, 1);
   dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);


// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

Συνάρτηση Kernel

```
// Matrix multiplication kernel – per thread code

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

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
```

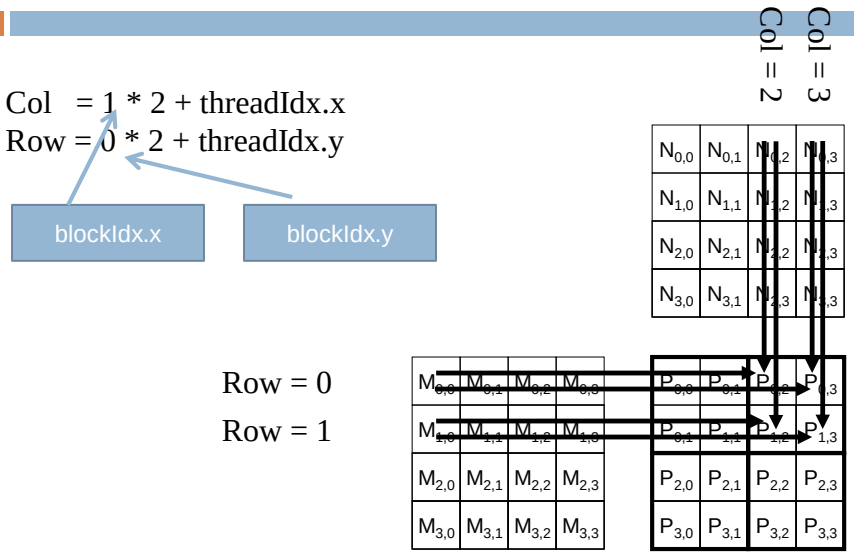

Μπλοκ(0,0) σε μια διαμόρφωση με TILE_WIDTH = 2



Μπλοκ(0,1)

Col = 1 * 2 + threadIdx.x
Row = 0 * 2 + threadIdx.y

blockIdx.x blockIdx.y



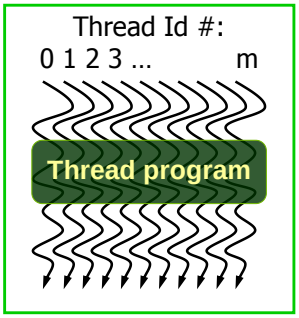
Ένας απλός Kernel πολλαπλασιασμού πινάκων

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

  if ((Row < Width) && (Col < Width)) {
    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;
   }
}
```

Μπλοκ νημάτων στην CUDA

- All threads in a block execute the same kernel program (SPMD)
- Programmer declares block:
  - Block size 1 to **1024** concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- Threads have thread index numbers within block
  - Kernel code uses thread index and block index to select work and address shared data
- Threads in the same block share data and synchronize while doing their share of the work
- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocks!

CUDA Thread Block



Courtesy: John Nickolls, NVIDIA

Ιστορία της παραλληλίας

- 1st gen - Instructions are executed sequentially in program order, one at a time.
- Example:

Cycle	1	2	3	4	5	6
Instruction1	Fetch	Decode	Execute	Memory		
Instruction2					Fetch	Decode

Ιστορία - διοχέτευση

- 2nd gen - Instructions are executed sequentially, in program order, in an assembly line fashion. (pipeline)
- Example:

Cycle	1	2	3	4	5	6
Instruction1	Fetch	Decode	Execute	Memory		
Instruction2		Fetch	Decode	Execute	Memory	
Instruction3			Fetch	Decode	Execute	Memory

Ιστορία – παραλληλία επιπέδου εντολής

- 3rd gen - Instructions are executed in parallel
- Example code 1:

```
c = b + a;
d = c + e;
```

Non-parallelizable

- Example code 2:

```
a = b + c;
d = e + f;
```

Parallelizable

Παραλληλία επιπέδου εντολής, συν.

- Two forms of ILP:
  - Superscalar: At runtime, fetch, decode, and execute multiple instructions at a time. Execution may be out of order

Cycle	1	2	3	4	5
Instruction1	Fetch	Decode	Execute	Memory	
Instruction2	Fetch	Decode	Execute	Memory	
Instruction3		Fetch	Decode	Execute	Memory
Instruction4		Fetch	Decode	Execute	Memory

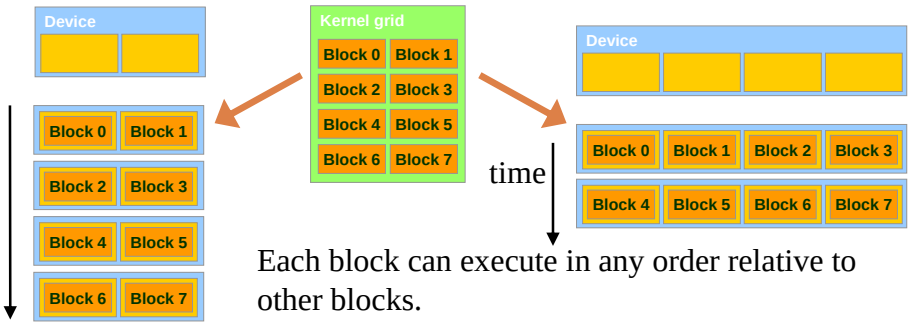
  - VLIW: At compile time, pack multiple, independent instructions in one large instruction and process the large instructions as the atomic units.

Ιστορία – πολυνημάτωση και πολυπύρηνοι επεξεργαστές

- 4th gen – Multi-threading: multiple threads are executed in an alternating or simultaneous manner on the same processor/core
- 5th gen - Multi-Core: Multiple threads are executed simultaneously on multiple processors

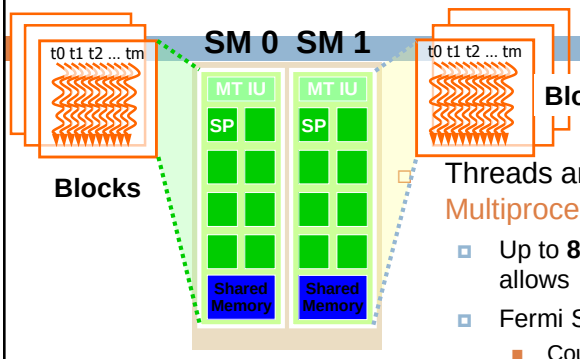
Διαφανής επεκτασιμότητα

- Hardware is free to assign blocks to any processor at any time
  - A kernel scales across any number of parallel processors



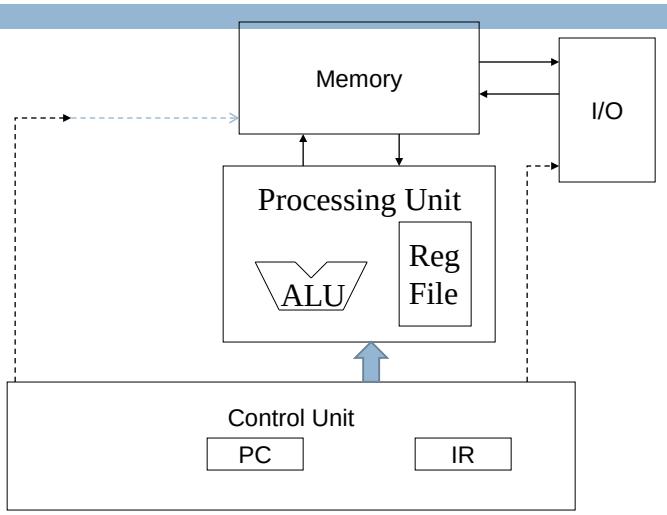
Each block can execute in any order relative to other blocks.

Παράδειγμα: εκτέλεση μπλοκ νημάτων

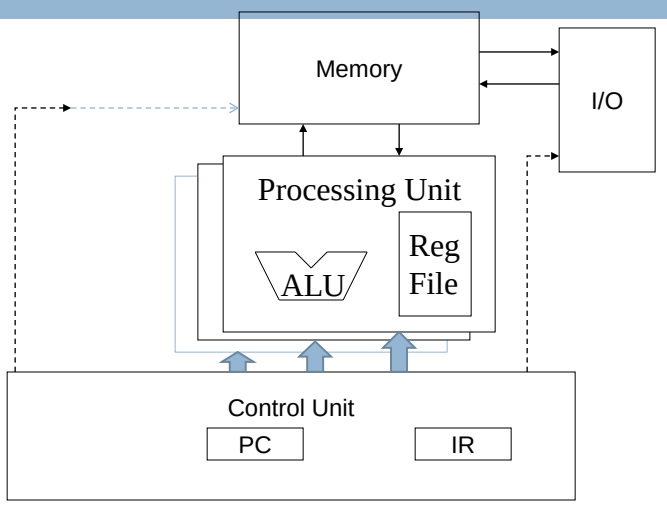


- Threads are assigned to Streaming Multiprocessors in block granularity
  - Up to **8** blocks to each SM as resource allows
  - Fermi SM can take up to **1536** threads
    - Could be 256 (threads/block) * 6 blocks
    - Or 512 (threads/block) * 3 blocks, etc.
- Threads run concurrently
  - SM maintains thread/block id #s
  - SM manages/schedules thread execution

Μοντέλο Von-Neumann

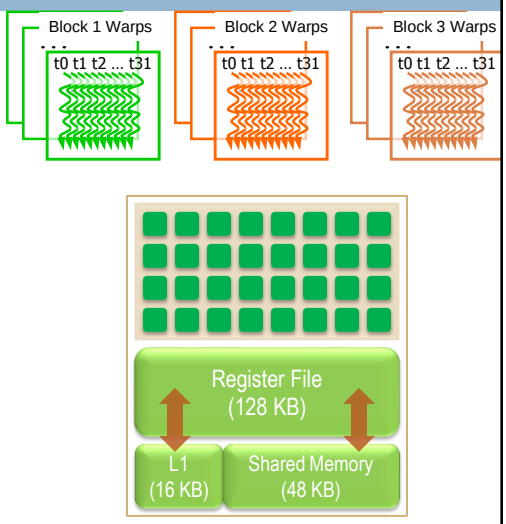


Μοντέλο Von-Neumann με SIMD μονάδες



Παράδειγμα: δρομολόγηση νημάτων

- Each Block is executed as 32-thread Warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps



Πως διαμερίζονται τα νήματα των μπλοκ

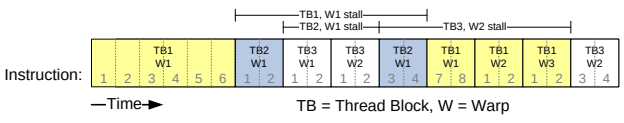
- Thread blocks are partitioned into warps
  - Thread IDs within a warp are consecutive and increasing
  - Warp 0 starts with Thread ID 0
- Partitioning is always the same
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation
  - (Covered next)
- **However, DO NOT rely on any ordering between warps**
  - If there are any dependencies between threads, you must __syncthreads() to get correct results (more later).

Εντολές ελέγχου ροής

- Main performance concern with branching is divergence
  - Threads within a single warp take different paths
  - Different execution paths are serialized in current GPUs
    - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
- A common case: avoid divergence when branch condition is a function of thread ID
  - Example with divergence:
    - `If (threadIdx.x > 2) { }`
    - This creates two different control paths for threads in a block
    - Branch granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
  - Example without divergence:
    - `If (threadIdx.x / WARP_SIZE > 2) { }`
    - Also creates two different control paths for threads in a block
    - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path

Παράδειγμα: δρομολόγηση νημάτων, συν.

- SM implements zero-overhead warp scheduling
  - At any time, 1 or 2 of the warps is executed by SM
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected



Μέγεθος μπλοκ

- For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 blocks?
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 1536 threads, there are 24 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!
  - For 16X16, we have 256 threads per Block. Since each SM can take up to 1536 threads, it can take up to 6 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we would have 1024 threads per Block. Only one block can fit into an SM for Fermi. Using only 2/3 of the thread capacity of an SM. Also, this works for CUDA 3.0 and beyond but too large for some early CUDA versions.

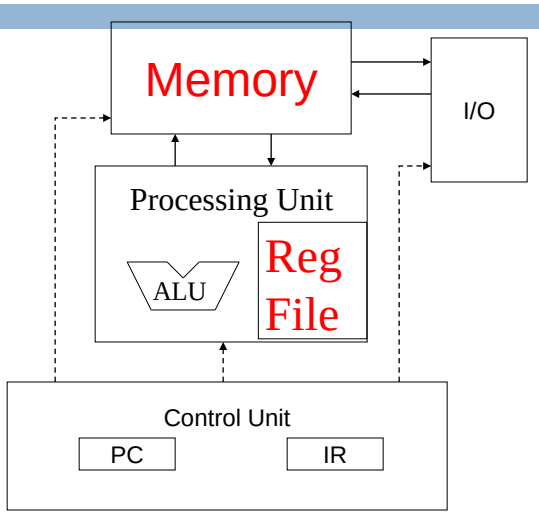
Ενότητα 3

- Μνήμες στην CUDA

Στόχοι

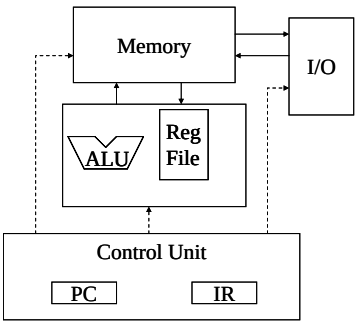
- Να μάθουμε να χρησιμοποιούμε αποδοτικά τα επίπεδα ιεραρχίας μνήμης της CUDA
  - Registers, shared memory, global memory
  - Tiled algorithms and barrier synchronization

Το μοντέλο Von-Neumann



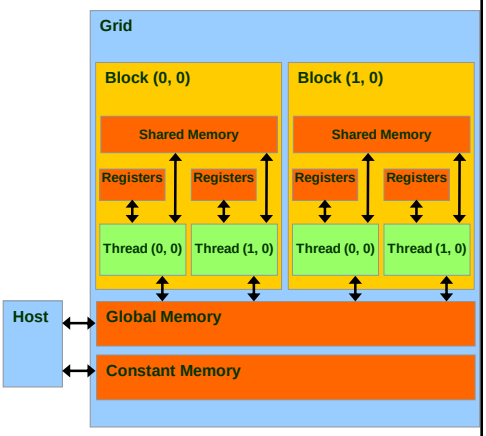
Καταχωρητές vs. Μνήμης

- Registers are “free”
  - No additional memory access instruction
  - Very fast to use, however, there are very few of them
- Memory is expensive (slow), but very large



Άποψη του προγραμματιστή για τις μνήμες της CUDA

- Each thread can:
  - Read/write per-thread **registers (~1 cycle)**
  - Read/write per-block **shared memory (~5 cycles)**
  - Read/write per-grid **global memory (~500 cycles)**
  - Read/only per-grid **constant memory (~5 cycles with caching)**



Κοινόχρηστη μνήμη στην CUDA

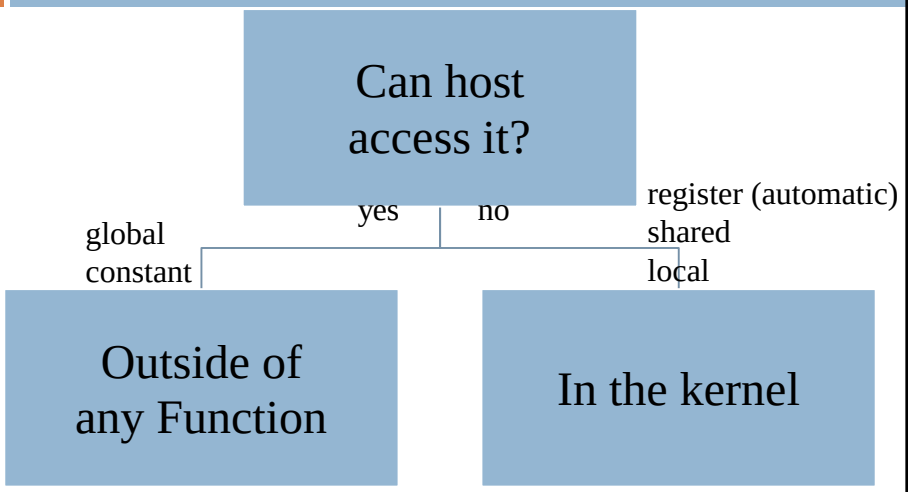
- A special type of memory whose contents are explicitly declared and used in the source code
  - Located in the processor
  - Accessed at much higher speed (in both latency and throughput)
  - Still accessed by memory access instructions
  - Commonly referred to as scratchpad memory in computer architecture

Προσδιοριστικά τύπου μεταβλητών 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

- `__device__` is optional when used with `__shared__`, or `__constant__`
- Automatic variables without any qualifier reside in a register
  - Except per-thread arrays that reside in global memory

Που δηλώνουμε μεταβλητές;



Δήλωση κοινόχρηστης 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];
```

Πολιτική προγραμματισμού

- Global memory resides in device memory (DRAM) - slow access
- So, a profitable way of performing computation on the device is to tile input data to take advantage of fast shared memory:
  - Partition data into subsets that fit into shared memory
  - Handle each data subset with one thread block by:
    - Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
    - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory to global memory

Παράδειγμα

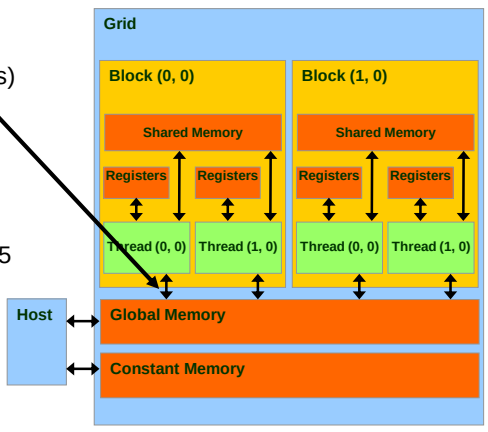
- Πολλαπλασιασμός πινάκων με χρήση κοινόχρηστης μνήμης

Βασικός kernel στον πολλαπλασιασμό πινάκων

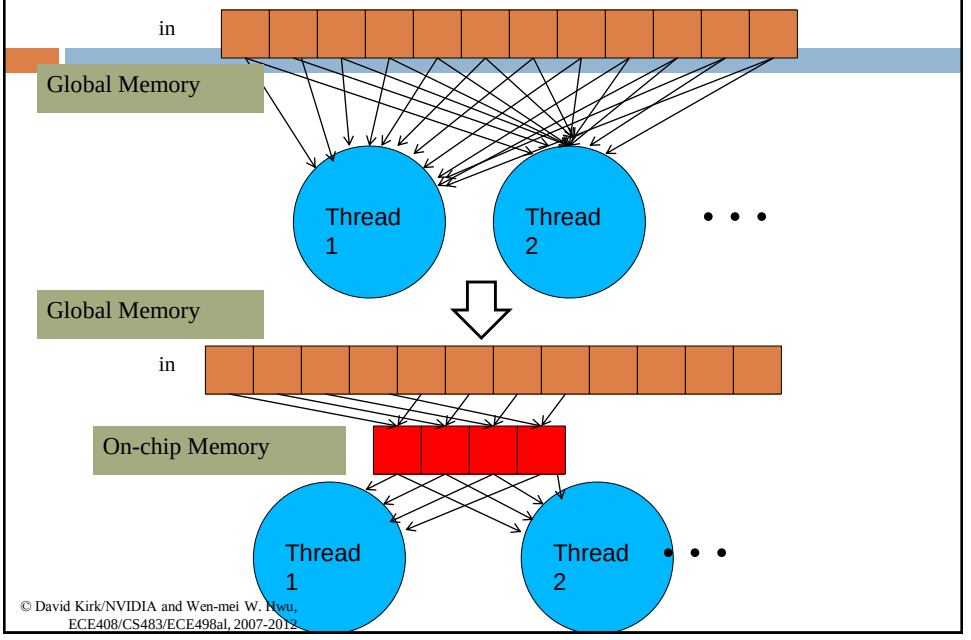
```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
// Calculate the row index of the Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column index of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

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;
}
```


Απόδοση στην αρχιτεκτονική Fermi

- All threads access global memory for their input matrix elements
  - Two memory accesses (8 bytes) per floating point multiply-add
  - 4B/s of memory bandwidth/FLOPS
  - 4*1,000 = 4,000 GB/s required to achieve peak FLOP rating
  - 150 GB/s limits the code at 37.5 GFLOPS
- The actual code runs at about 25 GFLOPS
- Need to drastically cut down memory accesses to get closer to the peak 1,000 GFLOPS

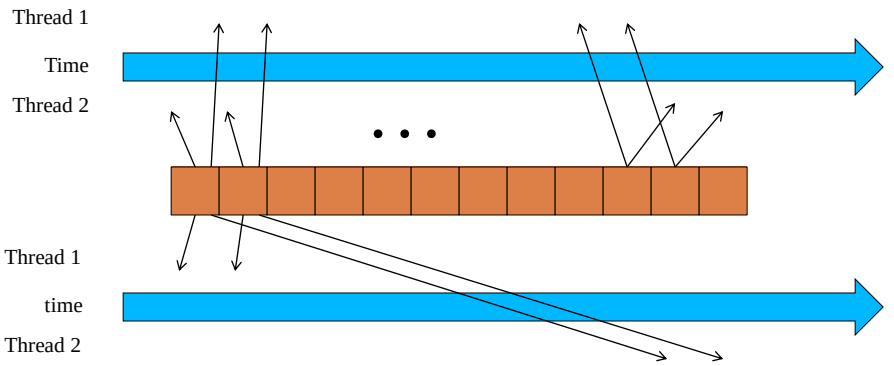


Ιδέα: Shared Memory Blocking



Το Blocking/Tiling απαιτεί συγχρονισμό

- Good – when threads have similar access timing



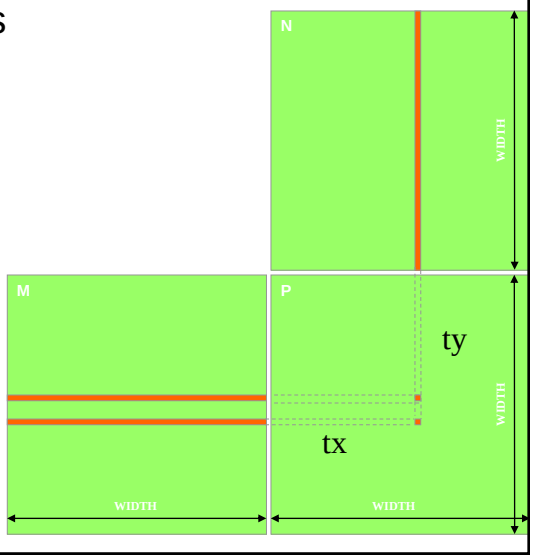
- Bad – when threads have very different timing

Σύνοψη της τεχνικής

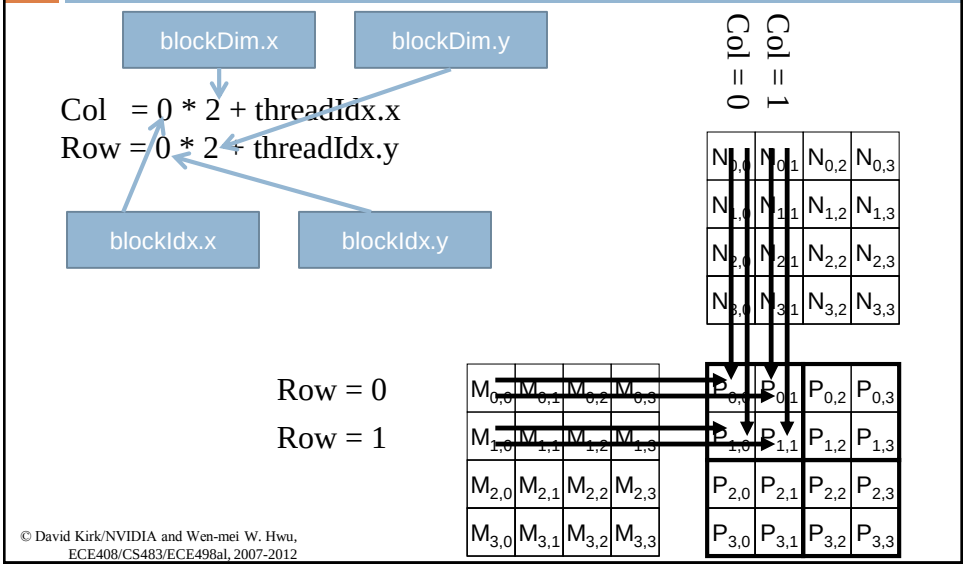
- Identify a block/tile of global memory content that are accessed by multiple threads
- Load the block/tile from global memory into on-chip memory
- Have the multiple threads to access their data from the on-chip memory
- Move on to the next block/tile

Ιδέα: χρησιμοποιήστε Shared Memory για τα δεδομένα της global memory

- Each input element is read by WIDTH threads.
- Load each element into Shared Memory and have several threads use the local version to reduce the memory bandwidth
  - Tiled algorithms

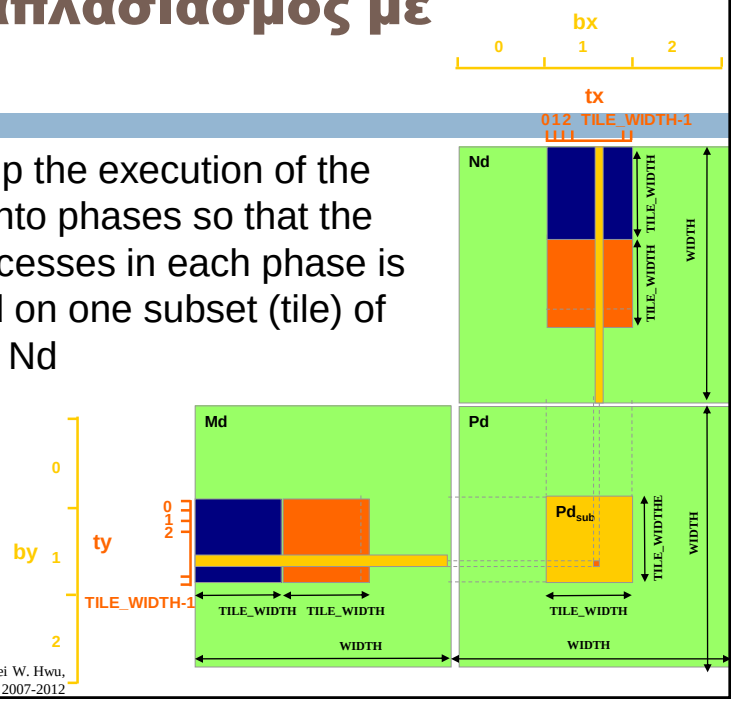


Μπλοκ (0,0) σε μια διαμόρφωση με TILE_WIDTH = 2



Πολλαπλασιασμός με tiles

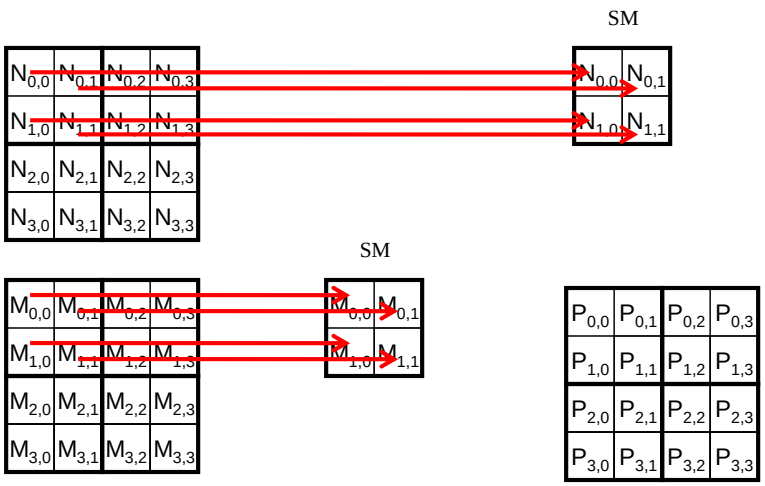
- Break up the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of Md and Nd



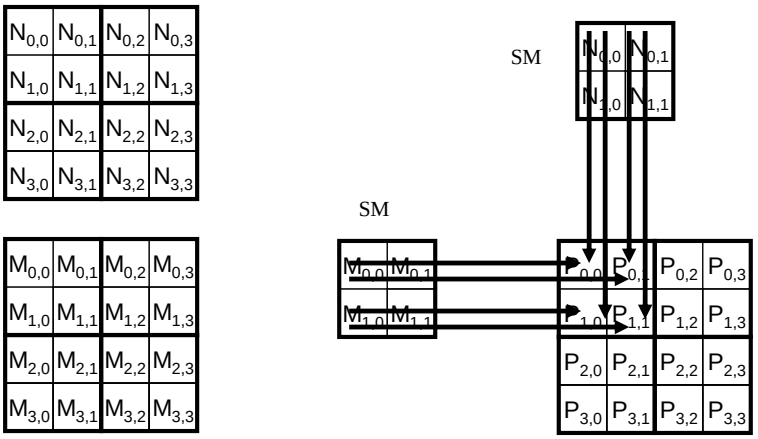
Φόρτωση ενός Tile

- All threads in a block participate
  - Each thread loads one Md element and one Nd element in based tiled code
- Assign the loaded element to each thread such that the accesses within each warp is coalesced (more later).

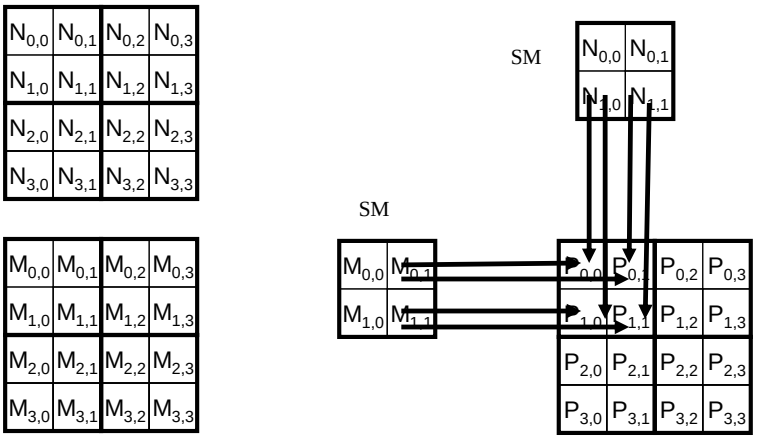
Για το μπλοκ (0,0)



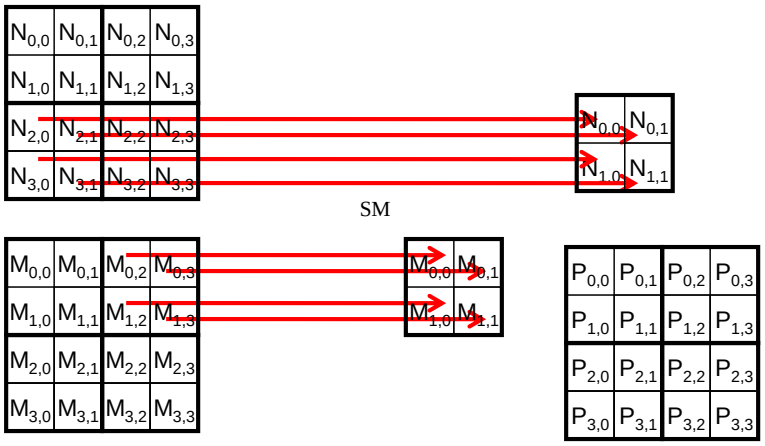
Για το μπλοκ (0,0)



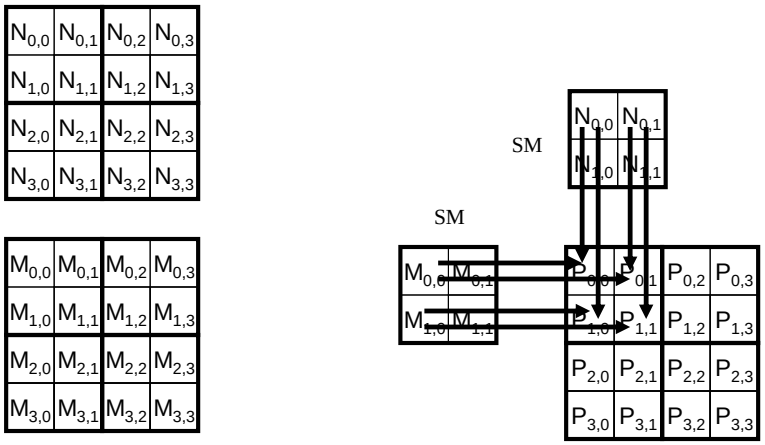
Για το μπλοκ (0,0)



Για το μπλοκ (0,0)

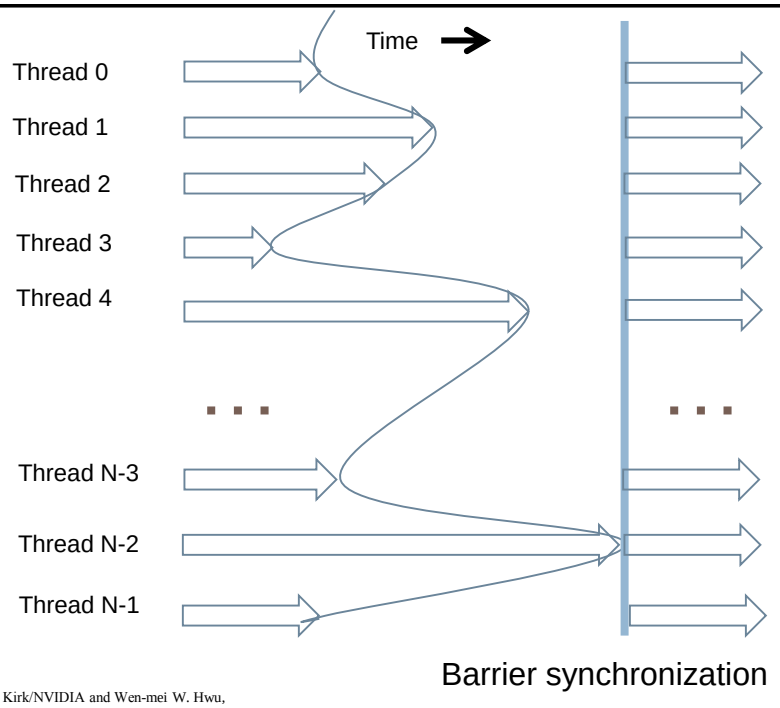


Για το μπλοκ (0,0)



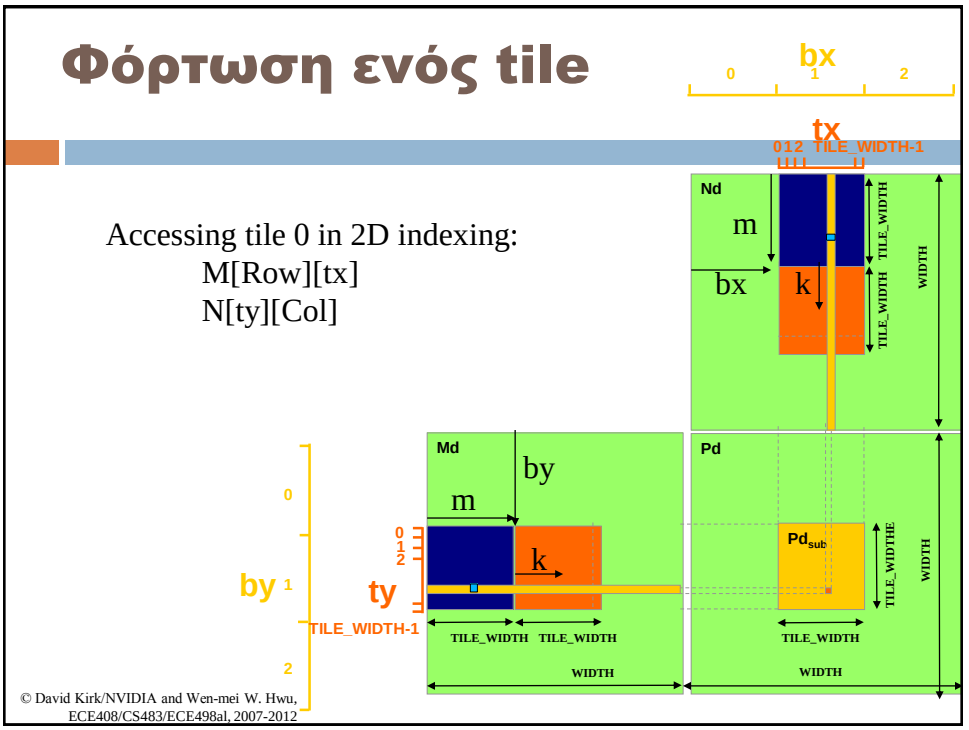
Συγχρονισμός με Barrier

- An API function call in CUDA
  - __syncthreads()
- All threads in the same block must reach the __synctrheads() before any can move on
- Best used to coordinate tiled algorithms
  - To ensure that all elements of a tile are loaded
  - To ensure that all elements of a tile are consumed



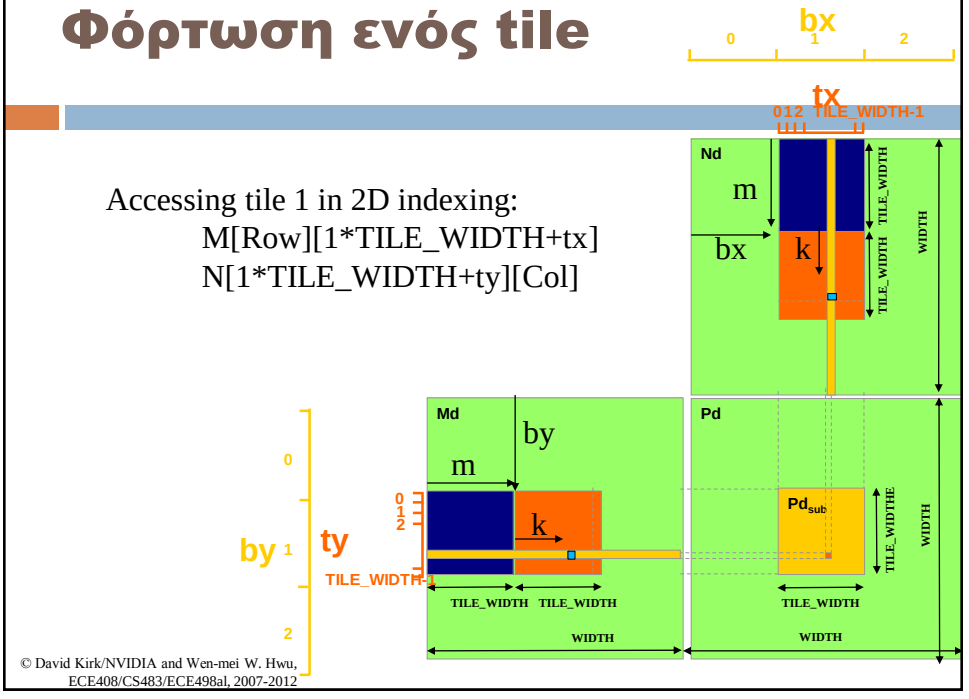
Φόρτωση ενός tile

Accessing tile 0 in 2D indexing:
M[Row][tx]
N[ty][Col]



Φόρτωση ενός tile

Accessing tile 1 in 2D indexing:
M[Row][1*TILE_WIDTH+tx]
N[1*TILE_WIDTH+ty][Col]

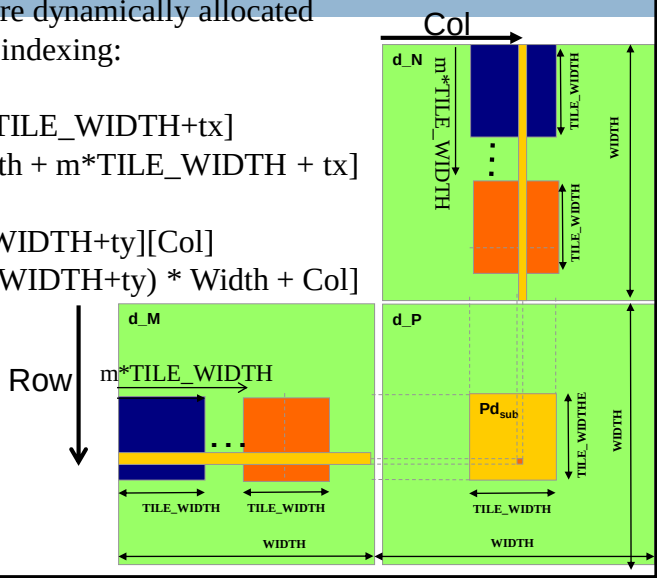


Φόρτωση του tile m

However, M and N are dynamically allocated and can only use 1D indexing:

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]



Πυρήνας πολλαπλασιασμού πινάκων 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;  int by = blockIdx.y;
4.  int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;
7.  float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
8.   for (int m = 0; m < Width/TILE_WIDTH; ++m) {
// Coolaborative loading of Md and Nd tiles into shared memory
9.     ds_M[ty][tx] = d_M[Row*Width + m*TILE_WIDTH+tx];
10.    ds_N[ty][tx] = d_N[Col+(m*TILE_WIDTH+ty)*Width];
11.    __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;
}
```

Ζητήματα μεγέθους tile

- Each thread block should have many threads
  - TILE_WIDTH of 16 gives 16*16 = 256 threads
  - TILE_WIDTH of 32 gives 32*32 = 1024 threads
- For 16, each block performs 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations.
- For 32, each block performs 2*1024 = 2048 float loads from global memory for 1024 * (2*32) = 65,536 mul/add operations

Κοινόχρηστη μνήμη και νήματα

- Each SM in Fermi has 16KB or 48KB shared memory*
  - SM size is implementation dependent!
  - For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory.
  - Can potentially have up to 8 Thread Blocks actively executing
    - This allows up to 8*512 = 4,096 pending loads. (2 per thread, 256 threads per block)
  - The next TILE_WIDTH 32 would lead to 2*32*32*4B= 8KB shared memory usage per thread block, allowing 2 or 6 thread blocks active at the same time

Χαρακτηριστικά συσκευής

- Number of devices in the system

```
int dev_count;
cudaGetDeviceCount( &dev_count);
```

- Capability of devices

```
cudaDeviceProp          dev_prop;
for (i = 0; i < dev_count; i++) {
        cudaGetDeviceProperties( &dev_prop, i);
  // decide if device has sufficient resources and capabilities
}
```

- cudaDeviceProp is a built-in C structure type
  - dev_prop.dev_prop.maxThreadsPerBlock
  - Dev_prop.sharedMemoryPerBlock
  - …

Υπολογιστικές δυνατότητες

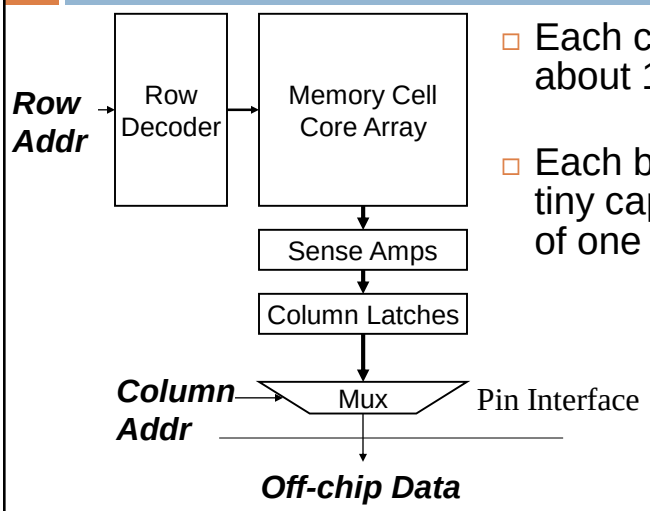
88

Technical specifications	Compute capability (version)							
	1.0	1.1	1.2	1.3	2.x	3.0	3.5	5.0
Maximum dimensionality of grid of thread blocks	2				3			
Maximum x-, y-, or z-dimension of a grid of thread blocks	65535						2^{31}-1	
Maximum dimensionality of thread block	3							
Maximum x- or y-dimension of a block	512				1024			
Maximum z-dimension of a block	64							
Maximum number of threads per block	512				1024			
Warp size	32							
Maximum number of resident blocks per multiprocessor	8						16	32
Maximum number of resident warps per multiprocessor	24		32		48		64	
Maximum number of resident threads per multiprocessor	768		1024		1536		2048	
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K		64 K	
Maximum number of 32-bit registers per thread	128				63		255	
Maximum amount of shared memory per multiprocessor	16 KB				48 KB			64 KB

Ενότητα 4

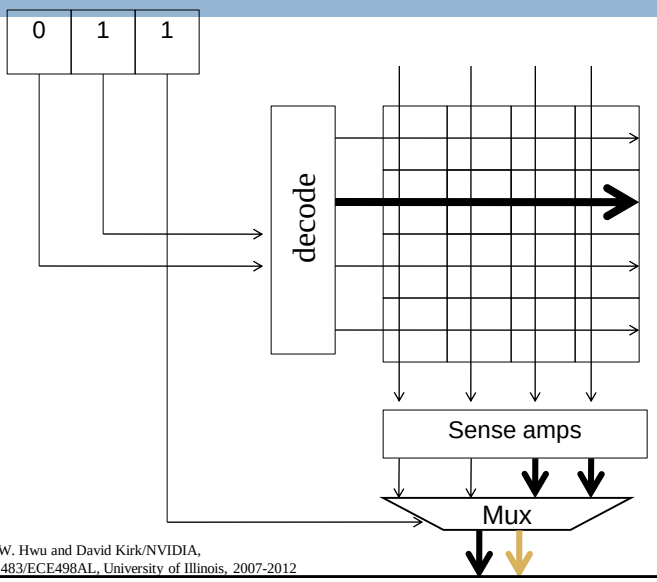
- Απόδοση DRAM
- Τεχνική συγκερασμού μνήμης (memory coalescing)

Οργάνωση DRAM



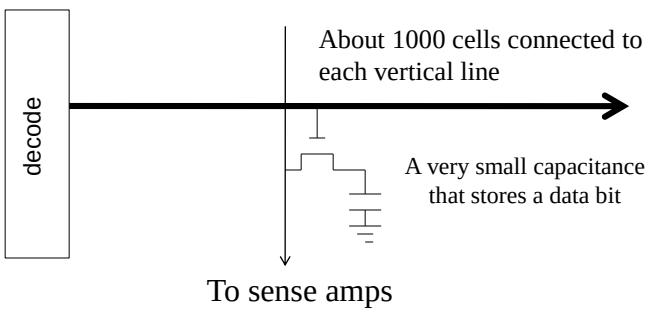
- Each core array has about 1M bits
- Each bit is stored in a tiny capacitor, made of one transistor

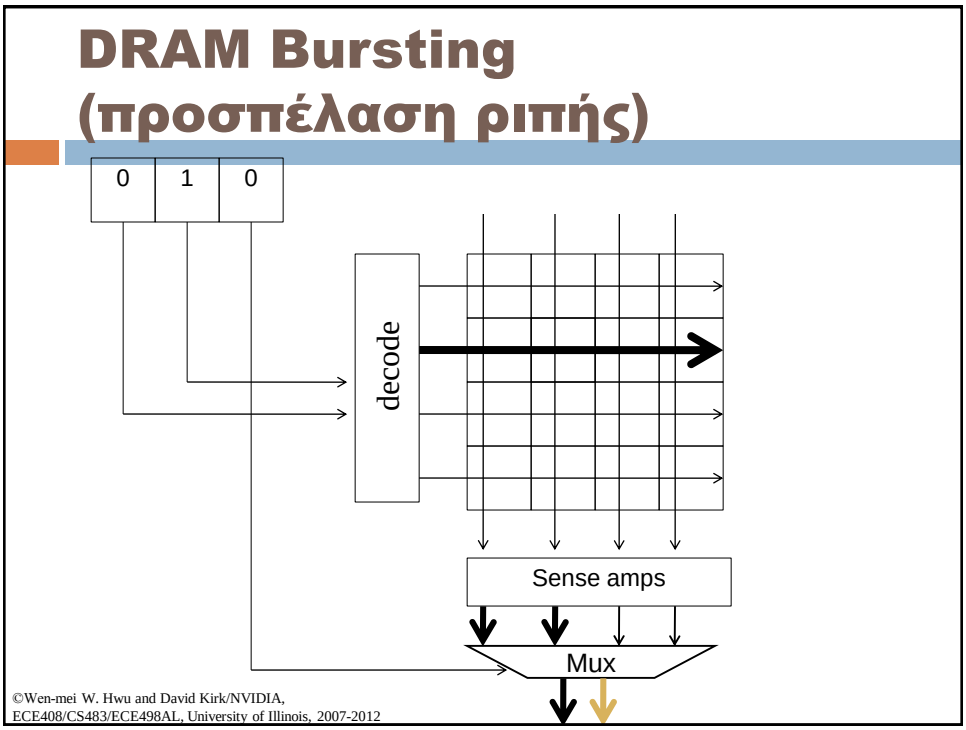
Μια μικρή σειρά (bank) DRAM (8x2 bit)

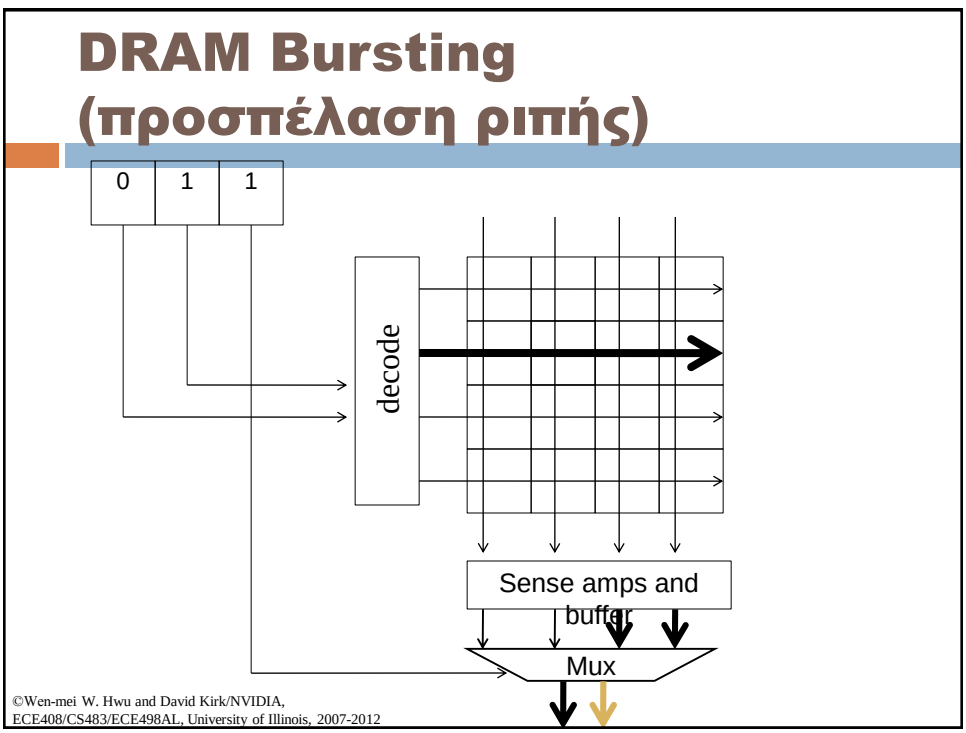


Η προσπέλαση ενός κελιού DRAM είναι αργή

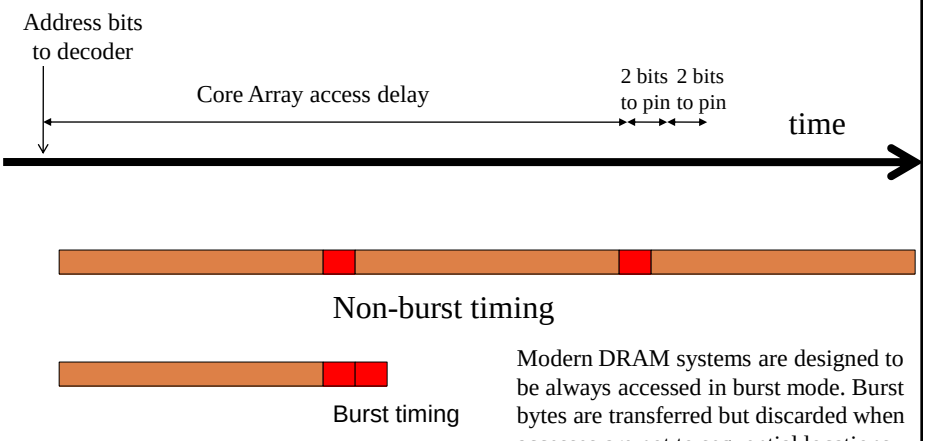
- Reading from a cell in the core array is a very slow process




DRAM Bursting
(προσπέλαση ριπής)
0
1
0
decode
Sense amps
Mux
©Wen-mei W. Hwu and David Kirk/NVIDIA,
ECE408/CS483/ECE498AL, University of Illinois, 2007-2012

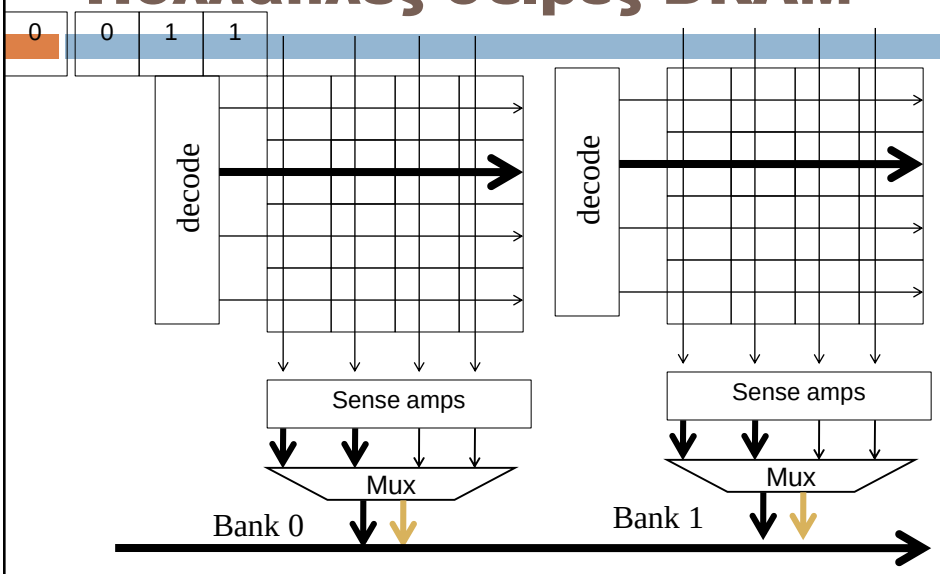

DRAM Bursting
(προσπέλαση ριπής)
0
1
1
decode
Sense amps and
buffer
Mux
©Wen-mei W. Hwu and David Kirk/NVIDIA,
ECE408/CS483/ECE498AL, University of Illinois, 2007-2012

Προσπέλαση ριπής σε μια DRAM 8x2

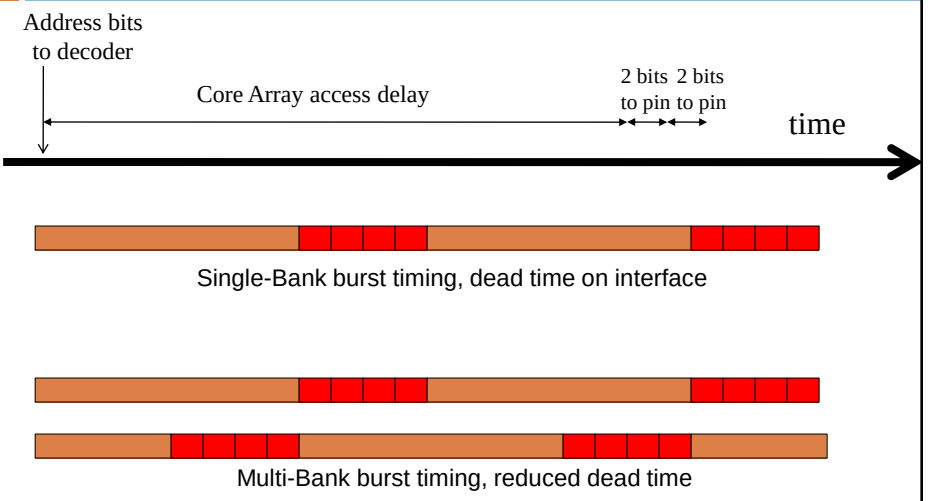


Modern DRAM systems are designed to be always accessed in burst mode. Burst bytes are transferred but discarded when accesses are not to sequential locations.

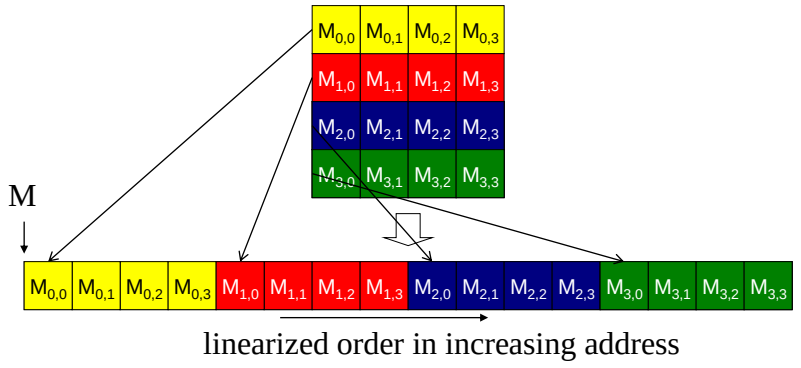
Πολλαπλές σειρές DRAM



Προσπέλαση ριπής σε μια σειρά 8x2



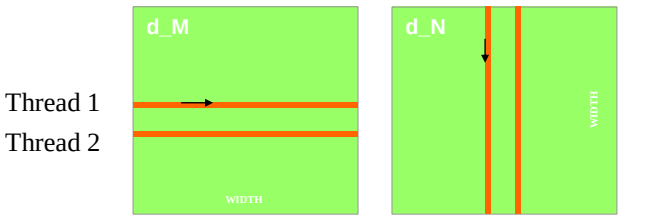
Τοποθέτηση ενός 2D πίνακα σε γραμμικό χώρο αποθήκευσης



Βασικός πυρήνας πολλαπλασιασμού πινάκων

```
__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;
}
```


Μοτίβο προσπέλασης

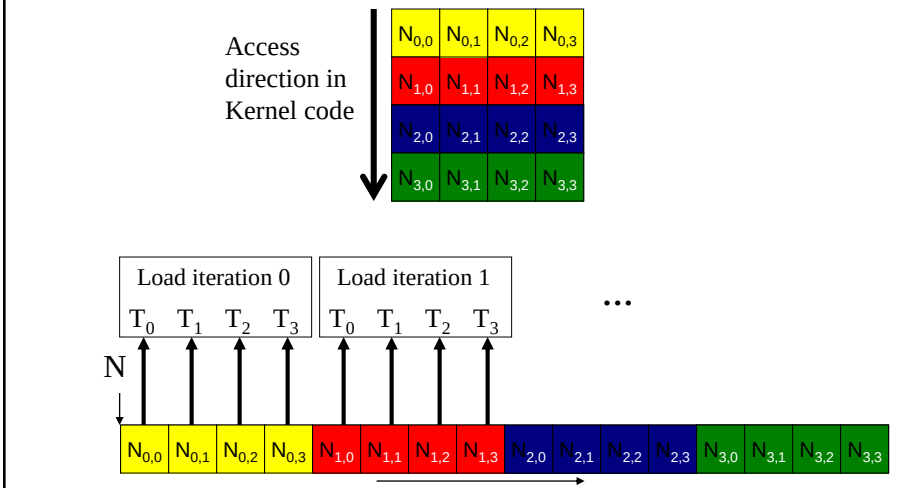


(a) (b)

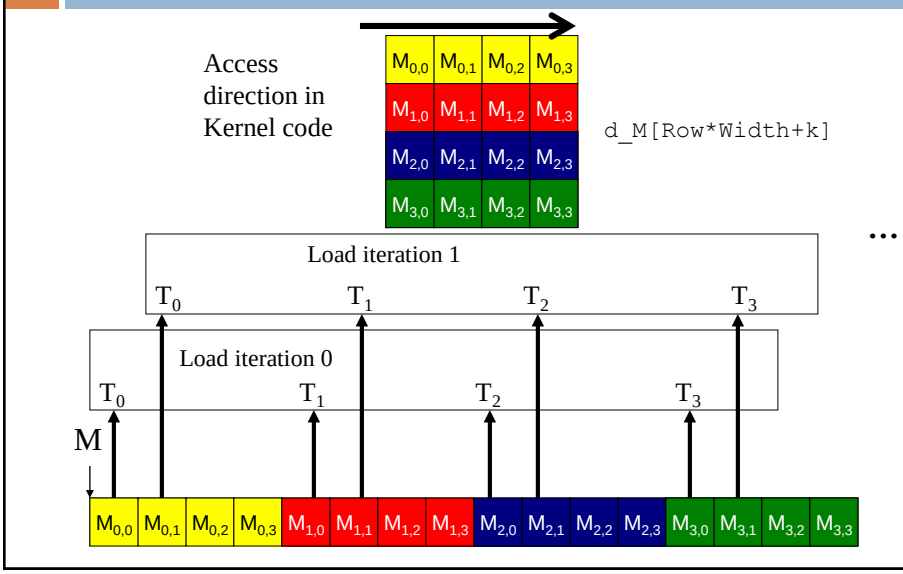
`d_M[Row*Width+k]` `d_N[k*Width+Col]`

k is loop counter in the inner product loop of the kernel code

Οι προσπελάσεις του N συγκεράζονται (coalesced)

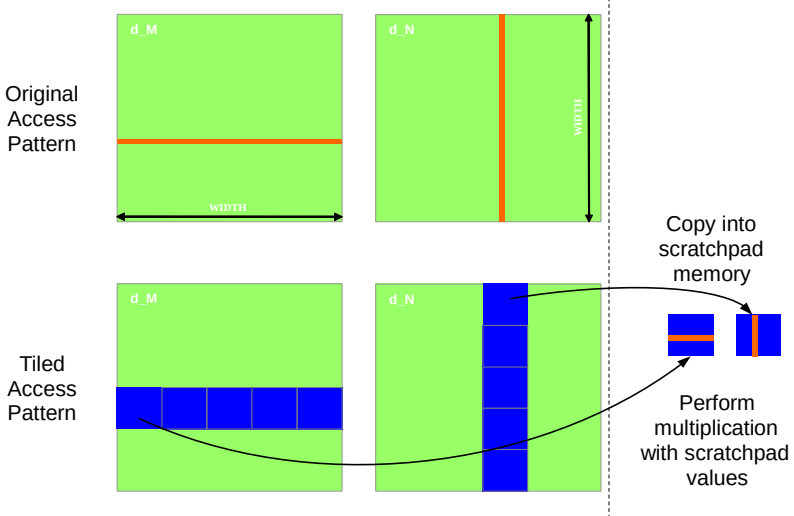


Οι προσπελάσεις του M δεν συγκεράζονται (coalesced)



```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
1.   __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2.   __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
3.   int bx = blockIdx.x;  int by = blockIdx.y;
4.   int tx = threadIdx.x; int ty = threadIdx.y;
// Identify the row and column of the d_P element to work on
5.   int Row = by * TILE_WIDTH + ty;
6.   int Col = bx * TILE_WIDTH + tx;

7.    float Pvalue = 0;
// Loop over the d_M and d_N tiles required to compute the d_P element
8.    for (int m = 0; m < Width/TILE_WIDTH; ++m) {

// Coolaborative loading of d_M and d_N tiles into shared memory
9.      Mds[tx][ty] = d_M[Row*Width + m*TILE_WIDTH+tx];
10.     Nds[tx][ty] = d_N[(m*TILE_WIDTH+ty)*Width + Col];
11.     __syncthreads();
12.   for (int k = 0; k < TILE_WIDTH; ++k)
13.       Pvalue += Mds[tx][k] * Nds[k][ty];
14.    __synchthreads();
   }
15.   d_P[Row*Width+Col] = Pvalue;
}
```


Χρήση της κοινόχρηστης μνήμης για συγκερασμό

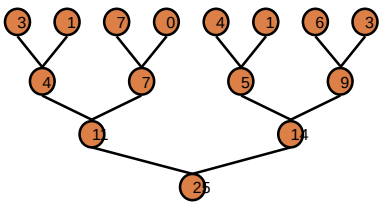


Ενότητα 5

- Μέθοδος μείωσης (reduction method)

Παράλληλη μείωση

- Tree-based approach used within each thread block



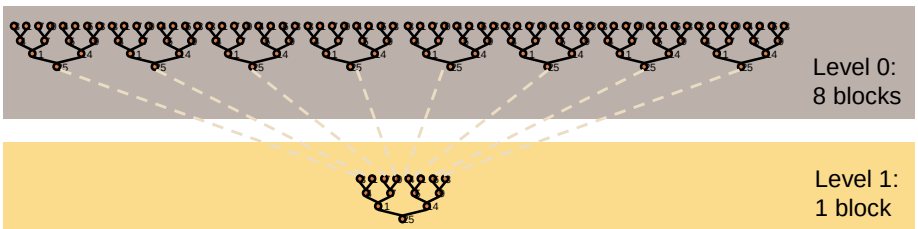
- Need to be able to use multiple thread blocks
  - To process very large arrays
  - To keep all multiprocessors on the GPU busy
  - Each thread block reduces a portion of the array
- But how do we communicate partial results between thread blocks?

Πρόβλημα: Συνολικός συγχρονισμός

- If we could synchronize across all thread blocks, could easily reduce very large arrays, right?
  - Global sync after each block produces its result
  - Once all blocks reach sync, continue recursively
- But CUDA has no global synchronization. Why?
  - Expensive to build in hardware for GPUs with high processor count
  - Would force programmer to run fewer blocks (no more than # multiprocessors * # resident blocks / multiprocessor) to avoid deadlock, which may reduce overall efficiency
- Solution: decompose into multiple kernels
  - Kernel launch serves as a global synchronization point
  - Kernel launch has negligible HW overhead, low SW overhead

Λύση: Μείωση σε επίπεδο πυρήνα (Kernel)

- Avoid global sync by decomposing computation into multiple kernel invocations

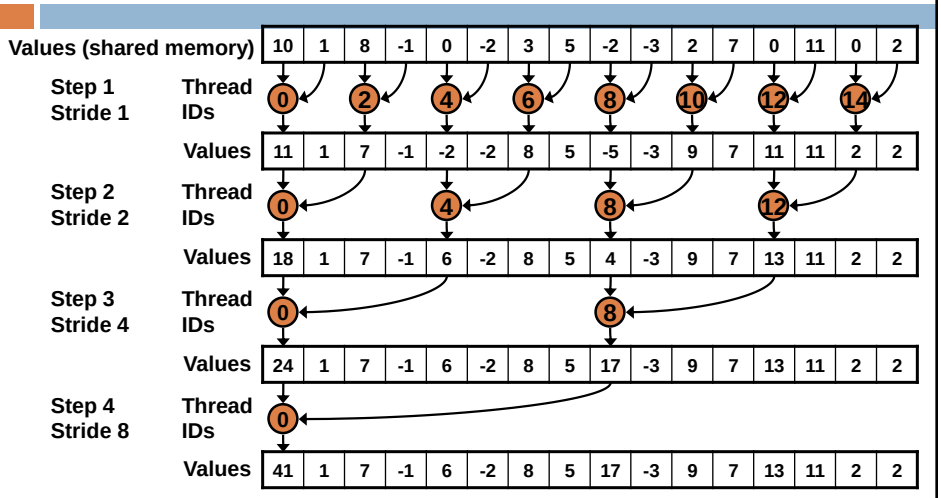


- In the case of reductions, code for all levels is the same
  - Recursive kernel invocation

Στόχος βελτιστοποίησης

- We should strive to reach GPU peak performance
- Choose the right metric:
  - GFLOP/s: for compute-bound kernels
  - Bandwidth: for memory-bound kernels
- Reductions have very low arithmetic intensity
  - 1 flop per element loaded (bandwidth-optimal)
- Therefore we should strive for peak bandwidth
- G80 GPU for example (multiple optimizations)
  - 384-bit memory interface, 900 MHz DDR
  - 384 * 1800 / 8 = 86.4 GB/s

Παράλληλη μείωση: Interleaved Addressing (πλεκτή διευθυνσιοδότηση)



Λύση #1: Interleaved Addressing (Base)

```
__global__ void reduce1(int *g_idata, int *g_odata) {
__device__ __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for (unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Λύση #1: Interleaved Addressing

```
__global__ void reduce1(int *g_idata, int *g_odata) {
__device__ __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for (unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

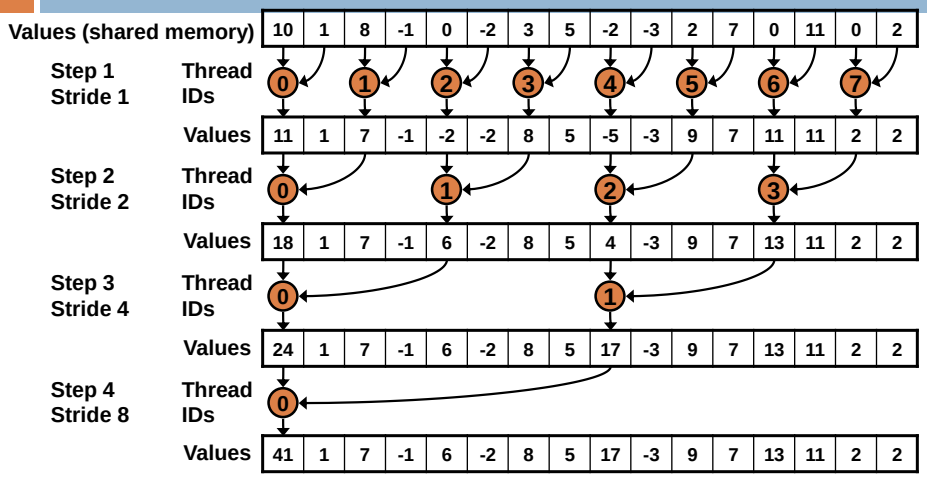
Problem: highly divergent warps are very inefficient, and % operator is very slow

Απόδοση για μείωση 4M στοιχείων

	Time (2^{22} ints)	Bandwidth
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s

Note: Block Size = 128 threads for all tests

Παράλληλη μείωση: Non-sequential Addresses



New Problem: Shared Memory Bank Conflicts

Λύση #2: Interleaved Addressing

Just replace divergent branch in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

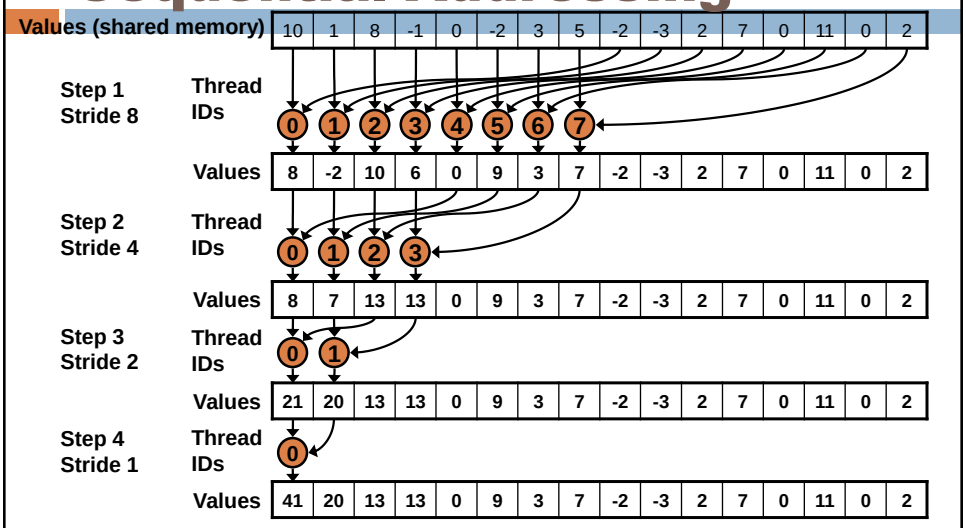
With strided index and non-divergent branch:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

Απόδοση για μείωση 4M στοιχείων

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x

Παράλληλη μείωση: Sequential Addressing



Sequential addressing is conflict free

Λύση #3: Sequential Addressing

Just replace strided indexing in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

With reversed loop and threadID-based indexing:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

Απόδοση για μείωση 4M στοιχείων

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x

Ανενεργά νήματα

Problem:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

Half of the threads are idle on first loop iteration!

This is wasteful…

Λύση #4: First add during load

Halve the number of blocks, and replace single load:

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

With two loads and first add of the reduction:

```
// perform first level of reduction,
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

Απόδοση για μείωση 4M στοιχείων

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x

Εμπόδιο απόδοσης οι εντολές

- At 17 GB/s, we're far from bandwidth bound
  - And we know reduction has low arithmetic intensity
- Therefore a likely bottleneck is instruction overhead
  - Ancillary instructions that are not loads, stores, or arithmetic for the core computation
  - In other words: address arithmetic and loop overhead
- Strategy: unroll loops

Ξετύλιγμα του τελευταίου στημονιού

- As reduction proceeds, # "active" threads decreases
  - When s <= 32, we have only one warp left
- Instructions are SIMD synchronous within a warp
- That means when s <= 32:
  - We don't need to __syncthreads()
  - We don't need "if (tid < s)" because it doesn't save any work
- Let's unroll the last 6 iterations of the inner loop

Λύση #5: Unroll the Last Warp

```
for (unsigned int s=blockDim.x/2; s>32; s>>=1)
{
   if (tid < s)
      sdata[tid] += sdata[tid + s];
   __syncthreads();
}

if (tid < 32)
{
   sdata[tid] += sdata[tid + 32];
   sdata[tid] += sdata[tid + 16];
   sdata[tid] += sdata[tid +  8];
   sdata[tid] += sdata[tid +  4];
   sdata[tid] += sdata[tid +  2];
   sdata[tid] += sdata[tid +  1];
}
```

Note: This saves useless work in *all* warps, not just the last one!
Without unrolling, all warps execute every iteration of the for loop and if statement

Απόδοση για μείωση 4M στοιχείων

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x

Περίληψη μείωσης

- Understand CUDA performance characteristics
  - Memory coalescing
  - Divergent branching
  - Bank conflicts
  - Latency hiding
- Use peak performance metrics to guide optimization
- Know how to identify type of bottleneck
  - e.g. memory, core computation, or instruction overhead
- Optimize your algorithm, *then* unroll loops