Πανεπιστήμιο Δυτικής Μακεδονίας
Τμήμα Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Συστήματα Παράλληλης & Κατανεμημένης Επεξεργασίας

Ενότητα 7: GPU Processing (CUDA)

Δρ. Μηνάς Δασυγένης
mdasyg@ieee.org
Εργαστήριο Ρομποτικής, Ενσωματωμένων και Ολοκληρωμένων Συστημάτων
http://arch.ece.uowm.gr/mdasyg

Άδειες Χρήσης

- Το παρόν εκπαιδευτικό υλικό υπόκειται σε άδειες χρήσης Creative Commons.
- Για εκπαιδευτικό υλικό, όπως εικόνες, που υπόκειται σε άλλου τύπου άδειας χρήσης, η άδεια χρήσης αναφέρεται ρητώς.

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ψηφιακά Μαθήματα στο Πανεπιστήμιο Δυτικής Μακεδονίας**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

Σκοπός της Ενότητας

- Η κατανόηση της παραλληλοποίησης μιας εφαρμογής για εκτέλεση στους παράλληλους επεξεργαστές της κάρτας γραφικών.

Όροι

- **Τι είναι GPGPU;**
  - Προγραμματισμός γενικής σκοπού (General-Purpose) σε μια μονάδα επεξεργασίας γραφικών (Graphics Processing Unit).
  - Χρησιμοποιεί hardware γραφικών για μη γραφικούς υπολογισμούς.
- **Τι είναι CUDA ;**
  - **C**ompute **U**nified **D**evice **A**rchitecture.
  - Αρχιτεκτονική λογισμικού για διαχείριση δεδομένων σε παράλληλο προγραμματισμό.

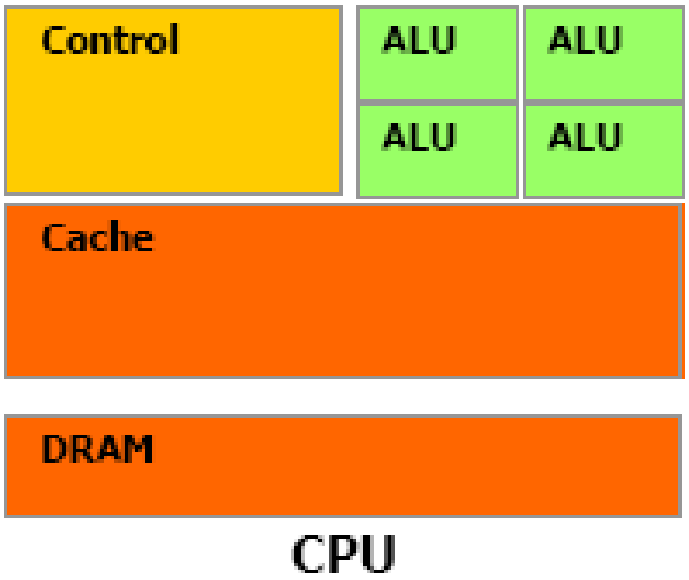
Year	GPU Card	FLOPS
2000	GeForce 2 Ultra	1 GFLOPS
2001	GeForce 3 Ti 500	1.6 GFLOPS
2002	GeForce 4 Ti 4600	4.8 GFLOPS
2003	Radeon 9800 XT	15.6 GFLOPS
2004	GeForce 6800 Ultra	36 GFLOPS
2005	Radeon X1800 XT	83.2 GFLOPS
2006	GeForce 8800 GTX	345.6 GFLOPS
2007	Radeon HD 2900 XT	475.2 GFLOPS
2008	GeForce GTX 280	933 GFLOPS
2009	Radeon HD 5870	2.72 TFLOPS
2010	GeForce GTX 480	1.35 TFLOPS
2011	Radeon HD 6990	5.1 TFLOPS
2012	GeForce GTX 690	5.62 TFLOPS
2013	GeForce GTX Titan Black	5.1 TFLOPS
2014	Radeon R9 Fury X	8.6 TFLOPS
2015	GeForce GTX Titan X (Maxwell)	6.14 TFLOPS
2016	GeForce GTX Titan X (Pascal)	11 TFLOPS
2017	GeForce GTX Titan V (Volta)	15 TFLOPS
2018	Radeon VII (Vega)	13.8 TFLOPS
2019	GeForce RTX Titan (Turing)	16.3 TFLOPS
2020	GeForce RTX Titan (Ampere)	34.1 TFLOPS
2021	Radeon RX7900 XTX (RDNA2)	51.2 TFLOPS
2023	Radeon RX7900 XTX (RDNA3)	58 TFLOPS

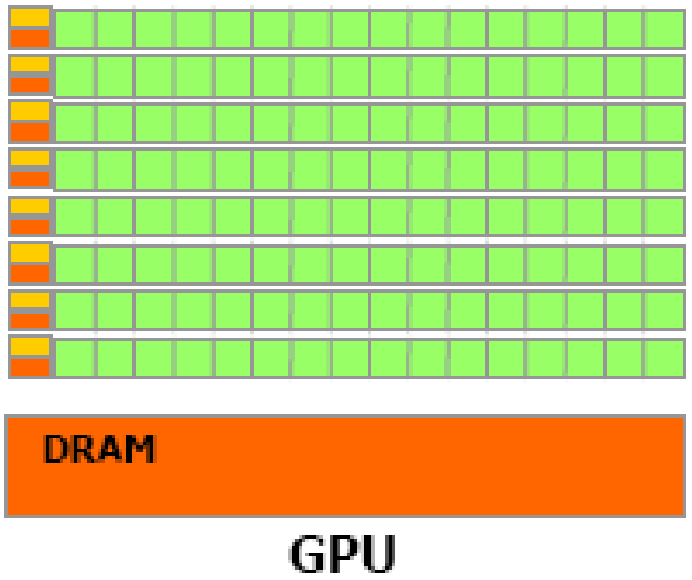
CPU εναντίον GPU

- **CPU:**
  - Γρήγορες caches.
  - Προσαρμοστικότητα διακλάδωσης.
  - Υψηλή απόδοση.
- **GPU:**
  - Πολλαπλές ALUs.
  - Γρήγορη onboard μνήμη.
  - Υψηλή ρυθμαπόδοση (throughput) σε παράλληλες εργασίες.
  - Εκτελεί το πρόγραμμα σε κάθε κομμάτι/κορυφή.

- Τα CPUs είναι εξαιρετικά για παραλληλισμό εργασιών.
- Τα GPUs είναι εξαιρετικά για παραλληλισμό δεδομένων.

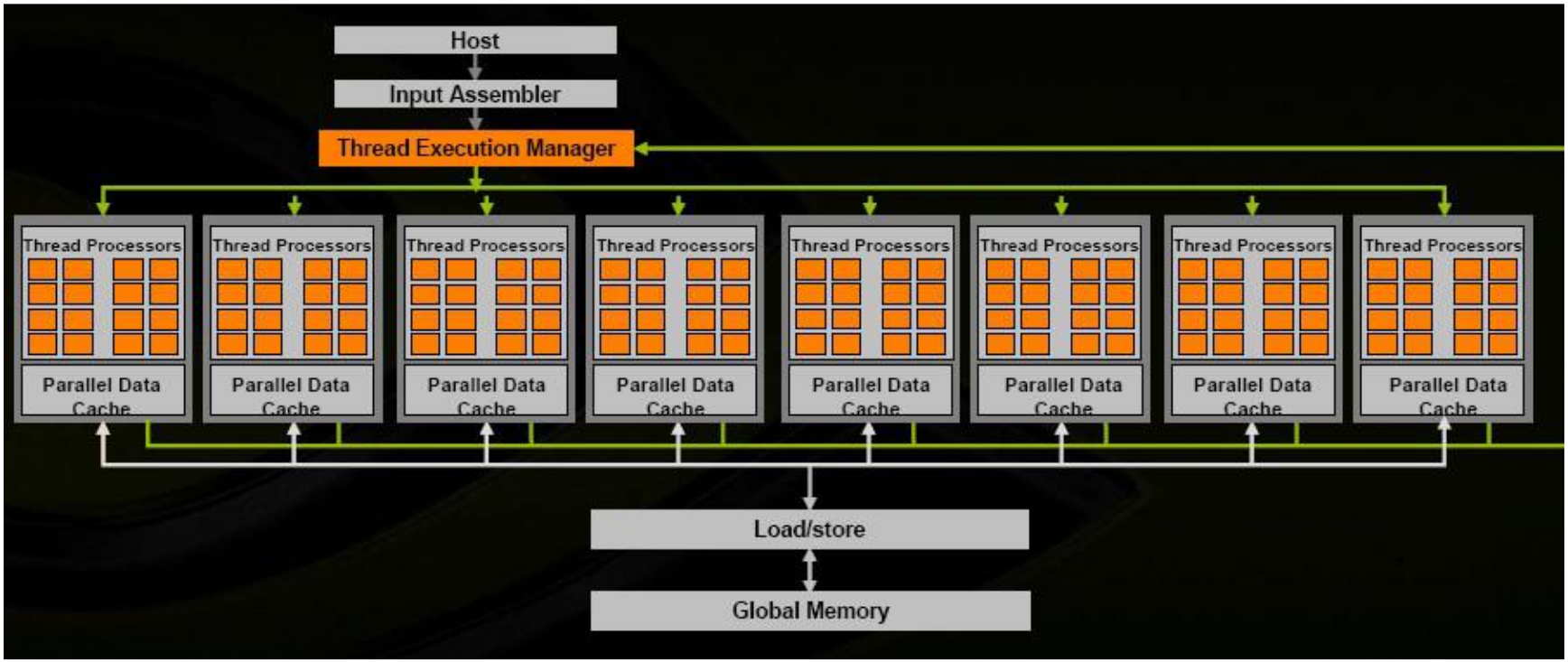
CPU vs. GPU - Hardware





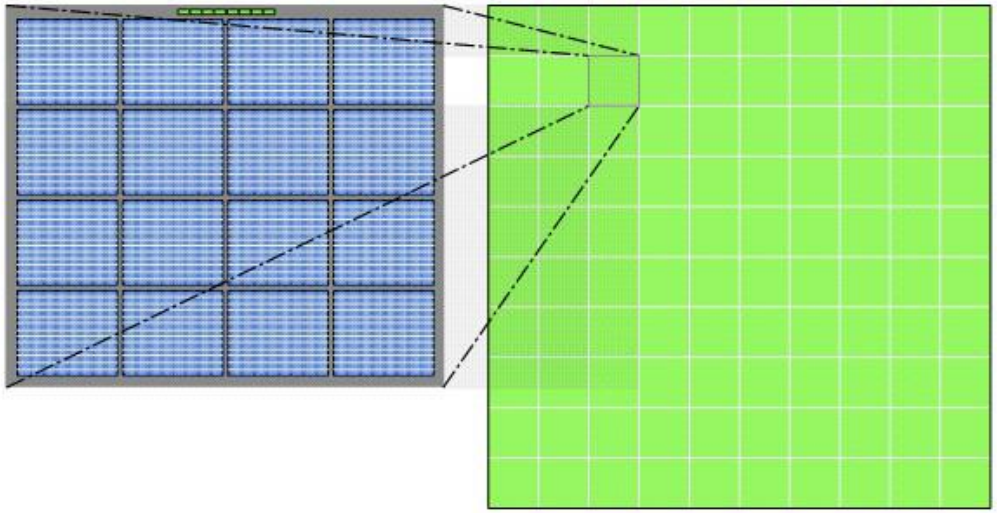
- Περισσότερα transistors αφιερώνονται στην επεξεργασία δεδομένων.

Η Αρχιτεκτονική GPU



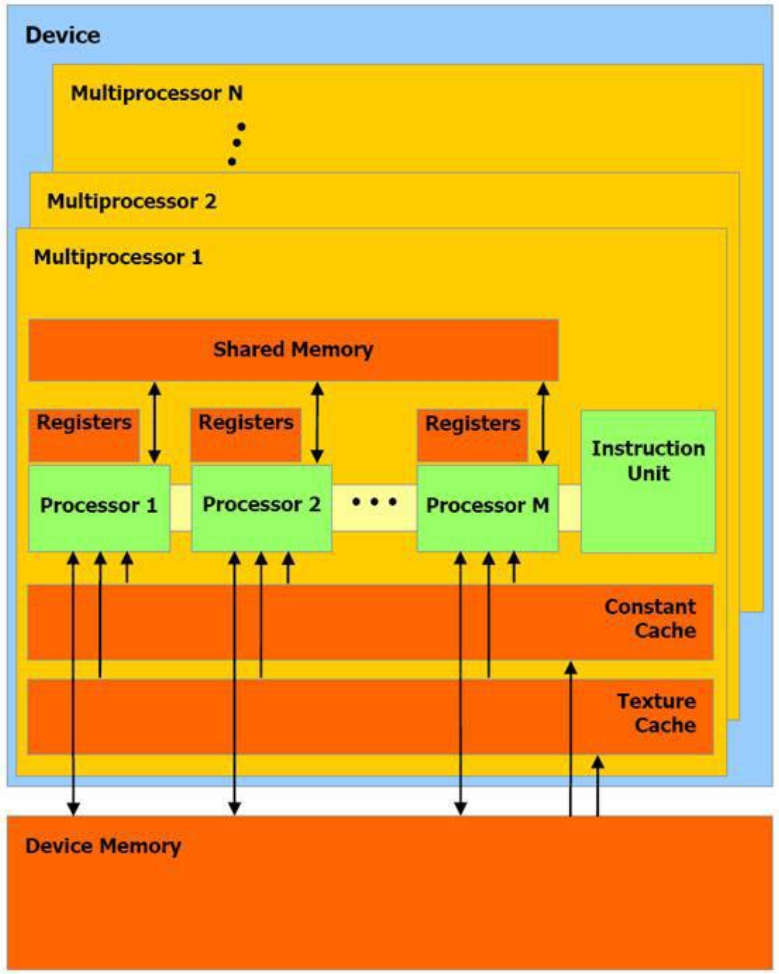
Εργασίες της CUDA

- Παρέχει την ικανότητας εκτέλεσης κώδικα στη GPU.
- Διαχείριση πόρων.
- Διαμέριση των δεδομένων για να χωρέσουν στους πυρήνες.
- Προγραμματίζει blocks σε πυρήνες.



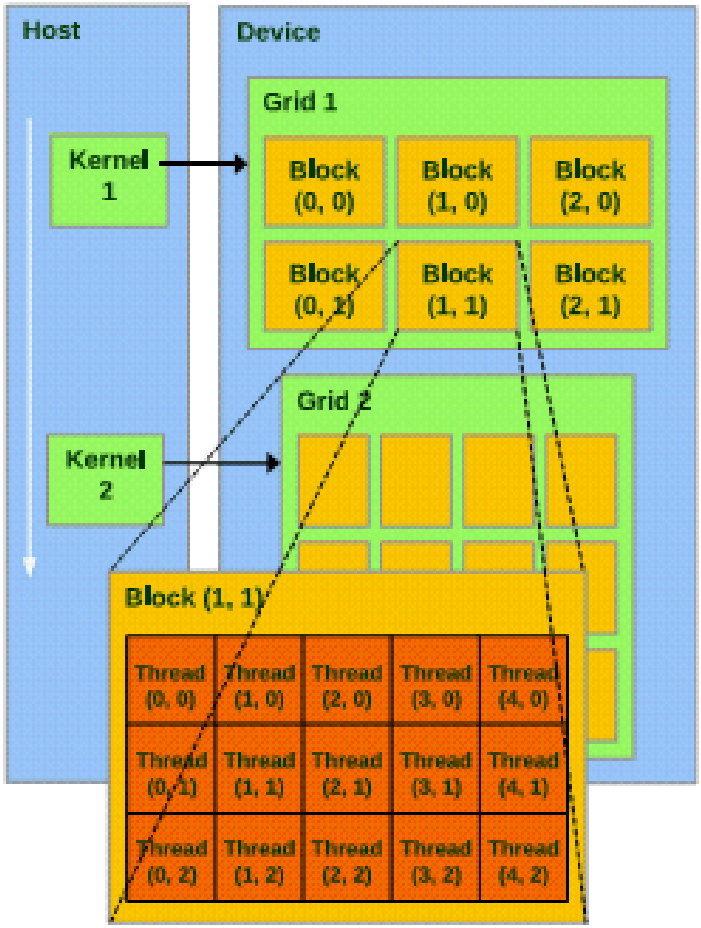
Αρχιτεκτονική Μνήμης

- Η GPU έχει τα παρακάτω είδη μνήμης:
  - Constant Memory.
  - Texture Memory.
  - Device Memory.



Αρχιτεκτονική Μνήμης (συνεχεία..)

- Διαμερισμός των δεδομένων σε μικρότερα blocks ώστε να μπορούν να επεξεργαστούν από έναν πυρήνα.
- Μέχρι και 512 νήματα σε ένα block.
- Όλα τα blocks ορίζουν ένα πλέγμα.
- Όλα τα blocks εκτελούν το ίδιο πρόγραμμα (kernel).
- Τα blocks είναι ανεξάρτητα.
- Μόνο ένας kernel κάθε φορά.



CUDA: Επέκταση της C

- Προσδιορισμός Συναρτήσεων.
- Προσδιορισμός Μεταβλητών.
- Ενσωματωμένες λέξεις – κλειδιά.
- Εγγενές (Intrinsics).
- Κλήσεις Συναρτήσεων.

Προσδιορισμός Συναρτήσεων (1/2)

Συναρτήσεις: device , global , host:

__global__ void filter(int *in, int *out) { ... }

- Default (προεπιλογή): Host.
- Χωρίς δείκτες συνάρτησης.
- Χωρίς αναδρομή.
- Χωρίς στατικές μεταβλητές.
- Χωρίς μεταβλητό αριθμό ορισμάτων.
- Χωρίς επιστρεφόμενη τιμή.

Προσδιορισμός Συναρτήσεων (2/2)

- Μεταβλητές: device , constant , shared.

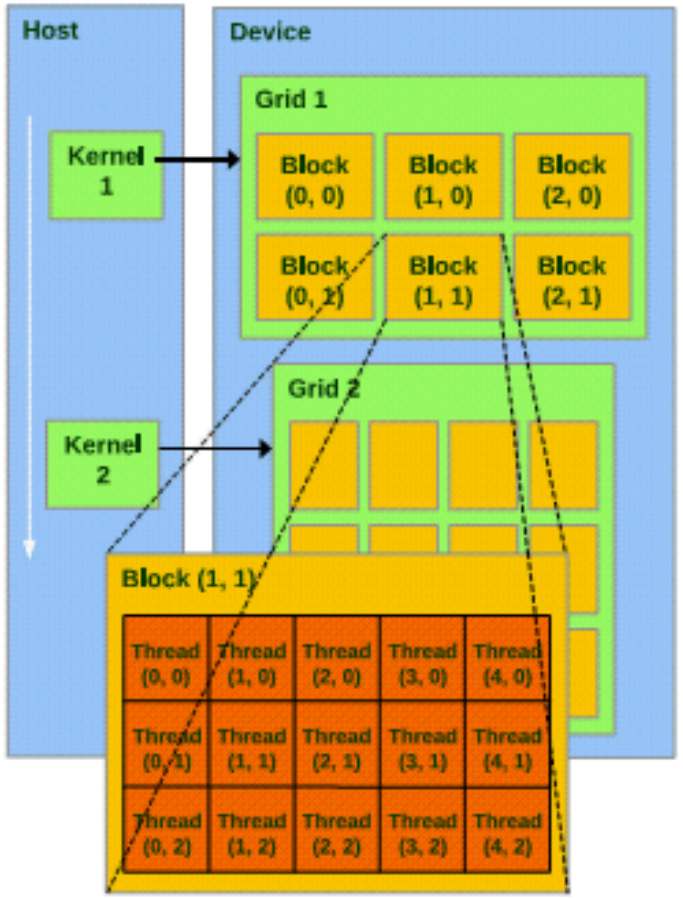
__constant__ float matrix [10] = {1 .0f, ... };

__shared__ int [32][2];

- Default: Μεταβλητές που διαμένουν στους καταχωρητές.

Ενσωματωμένες Μεταβλητές

- Διαθέσιμες μέσα στο κώδικα του Kernel.
- **Ευρετήριο νημάτων μέσα στο τρέχων νήμα:** threadIdx.x, threadIdx.y, threadIdx.z.
- **Ευρετήριο blocks μέσα στο πλέγμα:** blockIdx.x, blockIdx.y.
- **Διάσταση πλέγματος,block:** gridDim.x, gridDim.y, blockDim.x, blockDim.y, blockDim.z.
- **Μέγεθος στρέβλωσης:** warpSize.



Εγγενές - Intrinsics

- void __syncthreads();
  - Συγχρονίζει όλα τα νήματα του τρέχοντος block.
  - Η χρησιμοποίηση της σε conditional κώδικα μπορεί να οδηγήσει σε αδιέξοδα.
  - Υπάρχουν εγγενή για τις περισσότερες μαθηματικές συναρτήσεις.
- π.χ.,

__sinf(x), __cosf(x), __expf(x), ...

- Συναρτήσεις Πλοκής (texture).

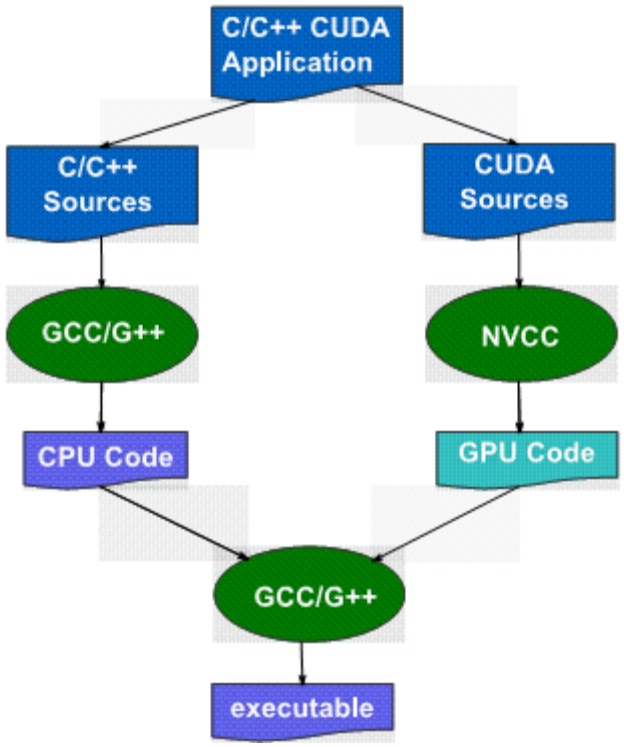
Κλήση Συναρτήσεων

- Ξεκίνημα των παραμέτρων:
  - Διάσταση πλέγματος (μέχρι και 2D).
  - Διάσταση block (μέχρι και 3D).
  - Προαιρετικά: stream ID.
  - Προαιρετικά: Μέγεθος κοινής μνήμης.

kernel<<<grid, block, stream, shared_mem>>>();

```
__global__ void filter(int *in, int *out);
...
dim3 grid(16, 16);
dim3 block(16, 16);
filter <<< grid, block, 0, 0 >>> (in, out);
filter <<< grid, block >>> (in, out);
```

Το μονοπάτι του compiler



- **gcc/g++**
  - μεταγλωττιστής για host κώδικα.
- **nvcc**
  - μεταγλωττιστής για κώδικα μηχανής.
- **gcc/g++**
  - μεταγλωττιστής για ενώσεις.
- **Εργαλεία απόσφαλμάτωσης:**
  - gdb,valgrind,cudavisualprofiler.

Διαχείριση Μνήμης

- Ο Host διαχειρίζεται τη μνήμη της GPU.
  - cudaMalloc(void **pointer, size_t size);
  - cudaMemset(void *pointer, int value, size_t count);
  - cudaFree(void *pointer);
- Memcopy για τη GPU:
  - cudaMemcpy(void *dst, void *src, size_t size, cudaMemcpyKind).
    - cudaMemcpyKind:
      - cudaMemcpyHostToDevice.
      - cudaMemcpyDeviceToHost.
      - cudaMemcpyDeviceToDevice.

Μέτρηση Χρόνου (1/2)

Η Αρχικοποίηση biases το χρόνο εκτέλεσης:

- Μη μετράτε την πρώτη εκκίνηση του Kernel!
- **Το SDK παρέχει timer:**

```
int timer =0;
cutCreateTimer (& timer );
cutStartTimer (timer );
...
cutStopTimer (timer );
cutGetTimerValue ( timer );
cutDeleteTimer (timer );
```

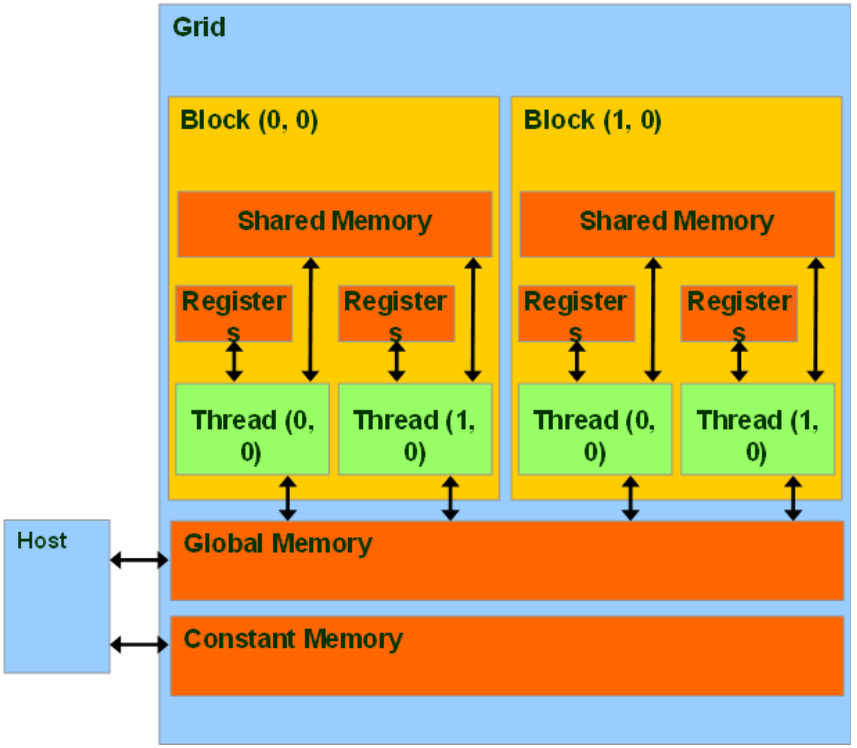
Μέτρηση Χρόνου (2/2)

- **Χρήση γεγονότων για ασύγχρονες συναρτήσεις:**

```
cudaEvent_t start_event, stop_event ;
cutilSafeCall ( cudaEventCreate (& start_event ));
cutilSafeCall ( cudaEventCreate (& stop_event ));
cudaEventRecord (start_event, 0); // record in stream-0, to ensure that all previous CUDA calls have completed
...
cudaEventRecord (stop_event, 0);
cudaEventSynchronize ( stop_event ); // block until the event is actually recorded
cudaEventElapsedTime (& time_memcpy, start_event, stop_event );
```

Τυπική αρχιτεκτονική για Μνήμες CUDA

- Κάθε νήμα μπορεί να κάνει:
  - Read/write ανά νήμα σε **registers.**
  - Read/write ανά νήμα σε **local memory.**
  - Read/write ανά block σε **shared memory.**
  - Read/write ανά πλέγμα σε **global memory.**
  - Read/only ανά πλέγμα σε **constant memory.**



Ορισμοί τύπων μεταβλητών CUDA

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

Η **__device__** είναι προαιρετική όταν χρησιμοποιείται με **__local__**, **__shared__**,ή **__constant__**

- **Αυτόματες μεταβλητές** χωρίς κανένα περιορισμό διαμένουν σε έναν καταχωρητή.
  - **Εξαιρούνται οι πίνακες** που εδρεύουν στην τοπική μνήμη.

Περιορισμοί τύπου Μεταβλητών

- Οι Δείκτες (pointers) μπορούν να δείχνουν μόνο σε μνήμη εκχωρημένη ή ορισμένη μέσα στην καθολική μνήμη:
  - Κατανεμημένη στον Host και περασμένη στον Kernel:
    - **__global__ void KernelFunc(float* ptr)**
- Παίρνεται σαν τη διεύθυνση μιας καθολικής μεταβλητής:
  - **float* ptr = &GlobalVar;**

Μια κοινή Στρατηγική Προγραμματισμού

- Η καθολική μνήμη βρίσκεται στη μνήμη συσκευής (DRAM)
  - έχει αρκετά πιο αργή προσπέλαση από ότι η κοινή μνήμη.
- Έτσι, ένας αποδοτικός τρόπος εκτέλεσης υπολογισμών στη συσκευή είναι το να κάνουμε tile τα δεδομένα για να επωφεληθούμε της γρήγορης κοινής μνήμης.
- Διαμερισμός των δεδομένων σε υποσύνολα που χωράνε στην κοινή μνήμη.
- Χειριζόμαστε κάθε υποσύνολο δεδομένων με ένα block νημάτων με το να:
  - Φορτώνουμε το υποσύνολο από την καθολική μνήμη στην κοινή μνήμη, χρησιμοποιώντας πολλαπλά νήματα για να εκμεταλλευτούμε παραλληλισμό επιπέδου μνήμης.
  - Εκτελούμε τον υπολογισμό στο υποσύνολο από την κοινή μνήμη ώστε κάθε νήμα να μπορεί να περάσει πολλές φορές (multi-pass) αποτελεσματικά πάνω από οποιοδήποτε στοιχείο δεδομένων.
  - Αντιγράφουμε τα αποτελέσματα από την κοινή στην καθολική μνήμη.

Μια κοινή Στρατηγική Προγραμματισμού (Cont.)

- Η σταθερή μνήμη εδρεύει επίσης μέσα στη μνήμη συσκευής (DRAM):
  - Έχει αρκετά πιο αργή προσπέλαση από ότι η κοινή μνήμη.
  - Αλλά είναι… cached!
  - Υψηλής απόδοσης προσπέλαση για δεδομένα μόνο προς ανάγνωση.
- Προσεκτικός διαχωρισμός των δεδομένων σύμφωνα με το πρότυπο προσπέλασης:
  - R/Only → σταθερή μνήμη (πολύ γρήγορη αν είναι μέσα στην cache).
  - R/W διαμοιραζόμενα μεταξύ των blocks → κοινή μνήμη (πολύ γρήγορη).
  - R/W μέσα σε κάθε νήμα → καταχωρητές (πολύ γρήγορα).
  - R/W είσοδοι/αποτελέσματα → καθολική μνήμη (πολύ αργή)

Ατομικές Λειτουργίες Ακεραίων της GPU

- Ατομικές λειτουργίες σε ακεραίους στην καθολική μνήμη:
  - **Συνδετικές λειτουργίες σε signed/unsigned ints:** atomicAdd(), atomicSub(), atomicExch(), atomicMin(), atomicMax(), atomicInc(), atomicDec, atomicCAS().
  - Ακόμη, μερικές λειτουργίες σε bits
- Προυποθέτει hardware με υπολογιστική ικανότητα 1.1 και παραπάνω.

Προγραμματισμός για Παραλληλισμό Δεδομένων

- Σκεφτείτε τη CPU σαν ένα συνεπεξεργαστή μαζικού νήματος.
- Γράφει “kernel” συναρτήσεις που εκτελούνται στη μηχανή:
  - επεξεργασία πολλαπλών στοιχείων δεδομένων παράλληλα.
- Κρατήστε το απασχολημένο! -> μαζική νηματοποίηση (massive threading).
- Κρατήστε τα δεδομένα σας κοντά! -> τοπική μνήμη.

Για τον GPU υπολογισμό χρειαζόμαστε να

- Εκχωρήσουμε τη μνήμη που θα χρησιμοποιηθεί για τον υπολογισμό (δήλωση και εκχώρηση μεταβλητής).
- Διαβάσουμε τα δεδομένα τα οποία θα υπολογίσουμε (input).
- Συγκεκριμενοποιήσουμε τον υπολογισμό που θα εκτελεστεί.
- Γράψιμο των αποτελεσμάτων στην κατάλληλη μηχανή των αποτελεσμάτων (output).

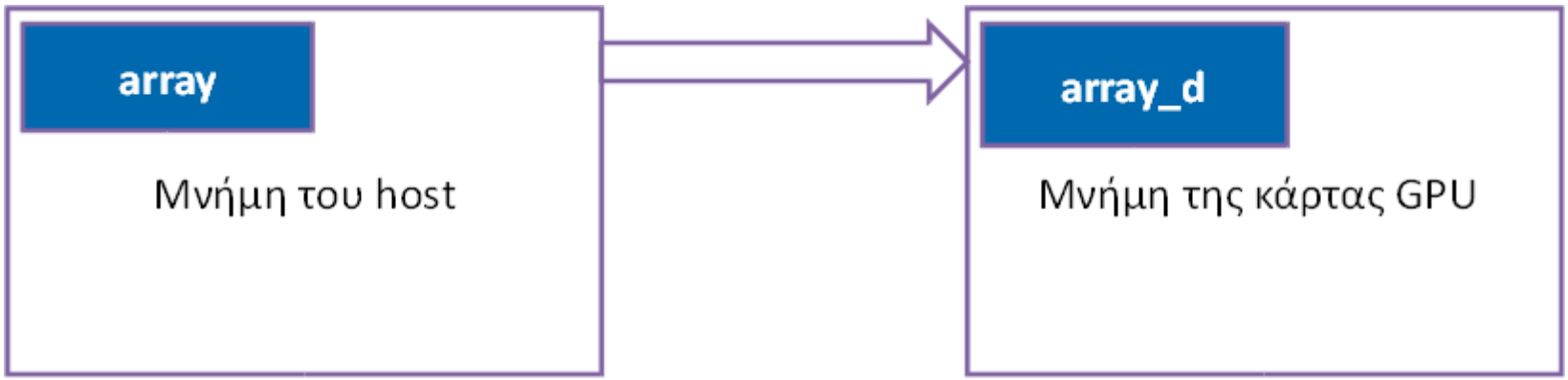
Μια GPU είναι ένας εξειδικευμένος υπολογιστής

- Χρειάζεται να **εκχωρήσουμε** χώρο στη μνήμη της κάρτας γραφικών για τις μεταβλητές.
- Η κάρτα γραφικών δεν έχει I/O συσκευές, έτσι χρειάζεται να **αντιγράψουμε τα εισαγόμενα δεδομένα** από τη μνήμη του host υπολογιστή, στη μνήμη της κάρτας γραφικών, χρησιμοποιώντας τη μεταβλητή που εκχωρήσαμε προηγουμένως.
- Χρειάζεται να προσδιορίσουμε τον κώδικα προς **εκτέλεση**.
- **Αντιγραφή των αποτελεσμάτων πίσω** στη μνήμη του υπολογιστή host.

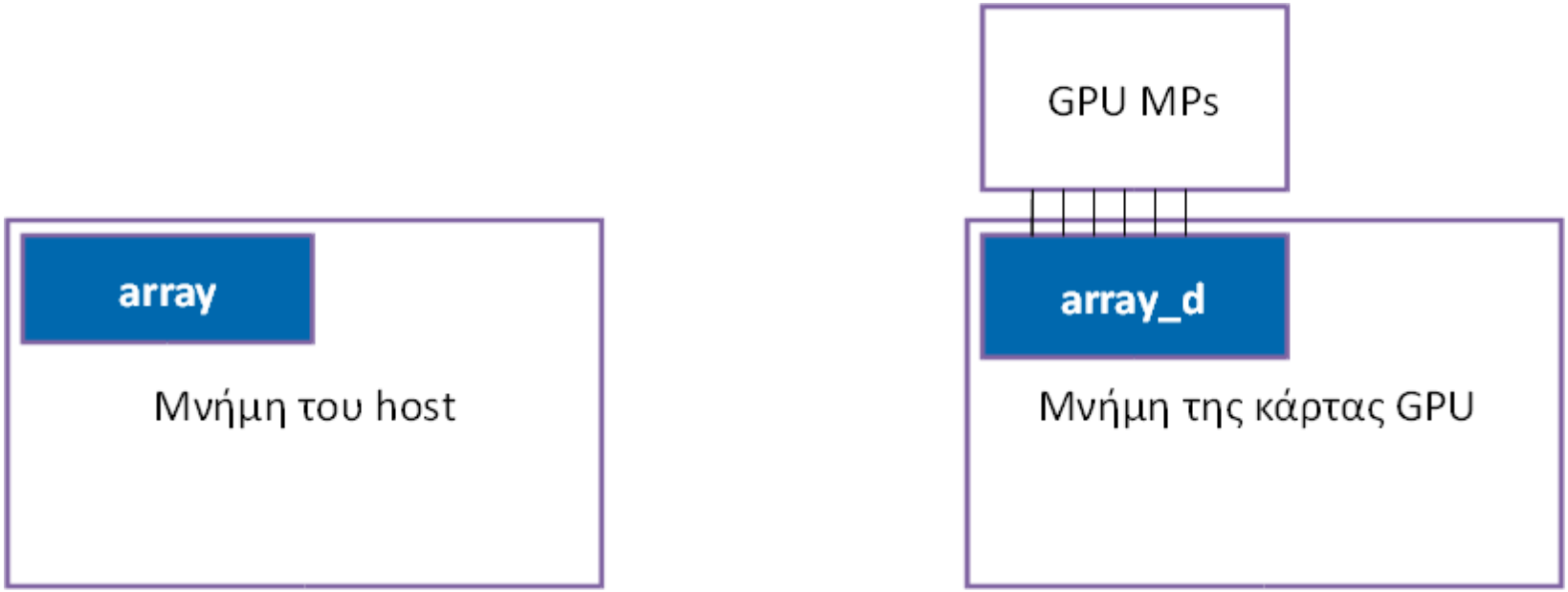
Αρχικά η μνήμη GPU είναι άδεια

Κατανομή μνήμης στην κάρτα GPU

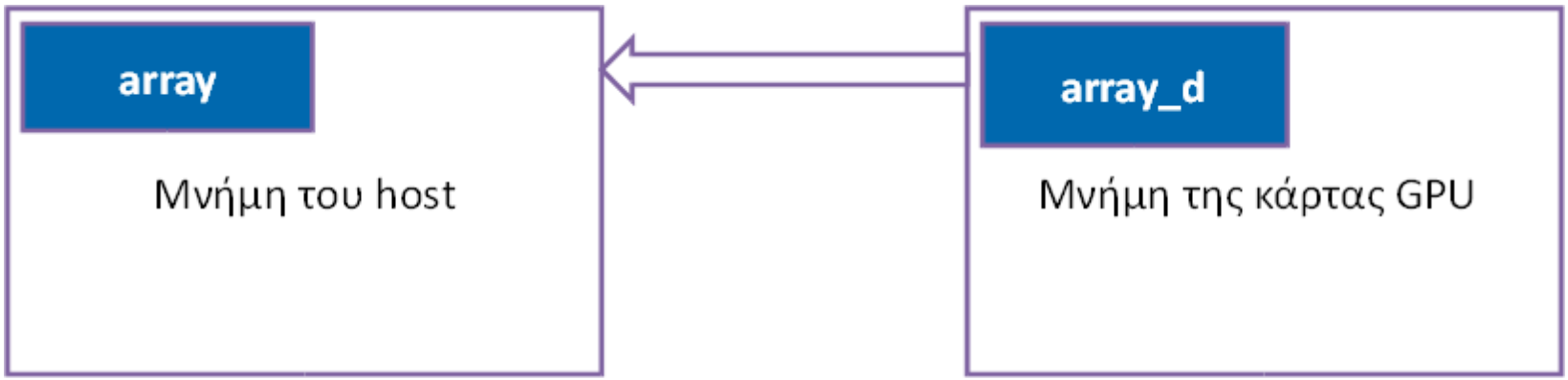
Αντιγραφή περιεχομένου από την μνήμη του host στην κάρτα μνήμης GPU



Εκτέλεση κώδικα στη GPU



Αντιγραφή των αποτελεσμάτων πίσω στη μνήμη του host



Ο Kernel

- Είναι απαραίτητο να γραφεί ο κώδικας που πρόκειται να εκτελεστεί στους stream επεξεργαστές στην κάρτα GPU.
- Αυτός ο κώδικας, που ονομάζεται kernel, θα κατεβαστεί και εκτελεστεί, συγχρόνως και σε lock-step fashion, σε μερικούς (όλους;) stream επεξεργαστές στην κάρτα GPU.
- Πώς μπορεί κάθε instance του kernel να ξέρει **ποιο κομμάτι δεδομένων βρίσκεται σε λειτουργία;**

Μέγεθος Πλέγματος και Block

- Οι προγραμματιστές πρέπει να είναι σαφής στα:
  - **Μέγεθος του πλέγματος (grid):** Το μέγεθος και το σχήμα των δεδομένων στα οποία το πρόγραμμα πρόκειται να δουλέψει.
  - **Μέγεθος του block:** Υποδηλώνει την υποπεριοχή του αυθεντικού πλέγματος που πρόκειται να απονεμηθεί σε ένα MP (ένα σύνολο από επεξεργαστές stream που μοιράζονται την τοπική μνήμη).

Ας ρίξουμε μια ματιά σε ένα απλό παράδειγμα

- Ο κώδικας έχει διαιρεθεί σε δυο αρχεία:
  - simple.c
  - simple.cu
- Το simple.c είναι συνηθισμένος κώδικας της C.
- Εκχωρεί ακεραίους σε έναν πίνακα, τον αρχικοποιεί στις τιμές που ανταποκρίνονται στα περιεχόμενα του πίνακα και τον εκτυπώνει.
- Καλεί μια συνάρτηση η οποία τροποποιεί το πίνακα.
- Ο πίνακας ξαναεκτυπώνεται.

```
#include <stdio.h>
#define SIZEOFARRAY 64
extern void fillArray(int *a,int size);
/* Το κυρίως πρόγραμμα */
int main(int argc,char *argv[])
{ /* Δήλωση του πίνακα που θα τροποποιηθεί από την GPU */
int a[SIZEOFARRAY];
int i;
/* Αρχικοποίηση του πίνακα σε 0s */
for(i=0;i < SIZEOFARRAY;i++) {
a[i]=i; }
/* Εκτύπωση του αρχικοποιημένου πίνακα */
printf("Initial state of the array:\n");
for(i = 0;i < SIZEOFARRAY;i++) {
printf("%d ",a[i]);
} printf("\n");
/* Κλήση της συνάρτησης που με τη σειρά της θα καλέσει τη συνάρτηση της, η οποία θα
συμπληρώσει τον πίνακα */
fillArray(a,SIZEOFARRAY);
/* Εκτύπωση του πίνακα μετά την κλήση της fillArray */
printf("Final state of the array:\n");
for(i = 0;i < SIZEOFARRAY;i++) {
printf("%d ",a[i]);
} printf("\n");
return 0; }
```

simple.cu

- Η simple.cu περιέχει δύο συναρτήσεις.

1. **fillArray():** Μια συνάρτηση που θα εκτελεστεί στον host και φροντίζει για:
   - Εκχώρηση μεταβλητών στην καθολική μνήμη της GPU.
   - Αντιγραφή του πίνακα από τον host στη μνήμη της GPU.
   - Ορισμός μεγέθους πλέγματος και block.
   - Επιστράτευση του kernel που εκτελείται στην GPU.
   - Αντιγραφή των τιμών πίσω στη μνήμη του host.
   - Απελευθέρωση της μνήμης της GPU.

fillArray (Μέρος 1ο)

```
#define BLOCK_SIZE 32
extern "C" void fillArray(int *array,int arraySize){
/* Η a_d είναι ο μετρητής GPU του πίνακα ο οποίος βρίσκεται στη μνήμη του host */
int *array_d;
/* Κατανομή μνήμης στη συσκευή */
/* Η cudaMalloc κατανέμει χώρο στη μνήμη της κάρτας GPU */
result = cudaMalloc((void**)&array_d,sizeof(int)*arraySize);
/* Αντιγραφή του πίνακα στη μεταβλητή array_d εντός της συσκευής */
/* Η μνήμη του host αντιγράφεται στην ανταποκρινόμενη μεταβλητή στην GPU της καθολικής μνήμης */
result = cudaMemcpy(array_d,array,sizeof(int)*arraySize,
cudaMemcpyHostToDevice);
```

fillArray (Μέρος 2ο)

```
/* Εκτέλεση διαμόρφωσης... */
/* Δηλώνει τη διάσταση του τετραγώνου */
dim3 dimblock(BLOCK_SIZE);
/* Δηλώνει τη διάσταση του δικτύου σε blocks */
dim3 dimgrid(arraySize/BLOCK_SIZE);
/* Γνήσιος υπολογισμός: Κλήση του kernel, η συνάρτηση που είναι */
/* Εκτελεσμένο από μία και κάθε χαρακτηριστικό διεργασίας στην κάρτα GPU */
cu_fillArray<<<dimgrid,dimblock>>>(array_d);
/* Διάβασμα των αποτελεσμάτων που παίρνουμε: */
/* Αντιγραφή των αποτελεσμάτων από την GPU πίσω στη μνήμη του host */
result = cudaMemcpy(array,array_d,sizeof(int)*arraySize,cudaMemcpyDeviceToHost);
/* Ελευθέρωση της μνήμης στην κάρτα GPU */
cudaFree(array_d);
}
```

simple.cu (συνέχεια)

- Η άλλη συνάρτηση στη simple.cu είναι:

2. **cu_fillArray():**
   - Πρόκειται για τον kernel που θα εκτελεστεί σε κάθε stream επεξεργαστή στη GPU.
   - Αναγνωρίζεται σαν ένας kernel με τη χρήση της λέξης κλειδί: **__global__.**
   - Αυτή η συνάρτηση χρησιμοποιεί τις ενσωματωμένες μεταβλητές,
   - blockIdx.x, και
   - threadIdx.x

     ...για να αναγνωριστεί μια συγκεκριμένη θέση στον πίνακα.

cu_fillArray

```
__global__ void cu_fillArray(int *array_d)
{
int x;
/* blockIdx.x is a built-in variable in CUDA
   that returns the blockId in the x axis
   of the block that is executing this block of code
   threadIdx.x is another built-in variable in CUDA
   that returns the threadId in the x axis
   of the thread that is being executed by this
   stream processor in this particular block
*/
x=blockIdx.x*BLOCK_SIZE+threadIdx.x;
array_d[x]+=array_d[x];
}
```

Για τη μεταγλώττιση

- **`nvcc simple.c simple.cu –o simple.`**
- Ο μεταγλωττιστής παράγει τον κώδικα και για τον host και για την GPU.

Παράδειγμα: Πρόσθεση Διανυσμάτων

- CPU κώδικας:

```
void vector_add (float *iA, float
*iB, float* oC, int width)
{
int i;
for (i=0; i<width ; i++) {
oC[i] = iA[i] + iB[i];}
}
```

Παράδειγμα: Πρόσθεση Διανυσμάτων (GPU)

```
// include CUDA and SDK headers - CUDA 2.1
#include <cutil_inline.h>
// include CUDA and SDK headers - CUDA 2.0
#include <cuda.h>
#include <cutil.h>
// include kernels
#include "vector_add_kernel.cu"
int main( int argc, char** argv) {
int dev;
// CUDA 2.1
dev = cutGetMaxGflopsDeviceId();
cudaSetDevice(dev);
// CUDA 2.0
CUT_DEVICE_INIT(argc, argv);
}
```

Παράδειγμα: Πρόσθεση Διανυσμάτων (GPU) Διαχείριση μνήμης

```
// allocate device memory
int *device_idata_A, * device_idata_B, *device_odata_C;
cudaMalloc ((void**) & device_idata_A, mem_size );
cudaMalloc ((void**) & device_idata_B, mem_size );
cudaMalloc ((void**) & device_odata_C, mem_size );
// copy host memory to device
cudaMemcpy (device_idata_A, host_idata_A, mem_size,
cudaMemcpyHostToDevice);
cudaMemcpy (device_idata_B, host_idata_B, mem_size,
cudaMemcpyHostToDevice);
...
// copy result from device to host
cudaMemcpy (host_odata_C, device_odata_C, mem_size,
cudaMemcpyDeviceToHost);
// free memory
cudaFree (device_idata_A);
cudaFree (device_idata_B);
cudaFree (device_odata_C);
```

Παράδειγμα: Πρόσθεση Διανυσμάτων (GPU) Εκκίνηση Kernel

```
// setup execution parameters
dim3 grid(1, 1);
dim3 threads( num_elements, 1);
// execute the kernel
vec_add<<< grid, threads >>>( device_idata_A, device_idata_B,
device_odata_C);
cudaThreadSynchronize();


__global__ void vec_add (float *iA, float *iB, float* oC) {
int idx = threadIdx.x + blockDim.x * blockId.x ;
oC[idx] = iA[idx] + iB[idx ];
}
```

Τι είναι αυτά τα blockIds και threadIds;

- Με μια μικρή τροποποίηση στον κώδικα, μπορούμε να εκτυπώσουμε τα blockIds και threadIds.
- Θα χρησιμοποιήσουμε δύο πίνακες αντί για έναν.
  - Έναν για τα blockIds.
  - Έναν για τα threadIds.
- Ο κώδικας στον kernel:

```
x=blockIdx.x*BLOCK_SIZE+threadIdx.x;
block_d[x] = blockIdx.x;
thread_d[x] = threadIdx.x;
```

Αυτό μπορεί να επεκταθεί σε δύο διαστάσεις

- Δείτε τα αρχεία:
  - blockAndThread2D.c
  - blockAndThread2D.cu
- Η ουσία είναι στον kernel:
  - x = blockIdx.x*BLOCK_SIZE+threadIdx.x;
  - y = blockIdx.y*BLOCK_SIZE+threadIdx.y;
  - pos = x*sizeOfArray+y;
  - block_dX[pos] = blockIdx.x;
- Μεταγλώττιση και εκτέλεση του blockAndThread2D:
  - nvcc blockAndThread2D.c blockAndThread2D.cu
    - -o blockAndThread2D
  - ./blockAndThread2D

blockAndThread2D.c & blockAndThread2D.cu

- Κατεβάστε τα αρχεία από το site:
  - http://sc08.sc-education.org/conference/pdg/mon/cuda/
- ή εναλλακτικά:
  - Ψάξτε στο google για blockAndThread2D.c
  - Κάντε compile, εκτελέστε τα και παρατηρήστε την έξοδο.

Όταν καλείται ο kernel

```
dim3 dimblock(BLOCK_SIZE,BLOCK_SIZE);
nBlocks = arraySize/BLOCK_SIZE;
dim3 dimgrid(nBlocks,nBlocks);
cu_fillArray<<<dimgrid,dimblock>>>
(… params…);
```

Ένα άλλο παράδειγμα: saxpy

- SAXPY (Scalar Alpha X Plus Y).
- Μια γνωστή πράξη στη γραμμική άλγεβρα.
- CUDA: Επανάληψη του βρόχου -> νήμα.

Παραδοσιακός Ακολουθιακός Κώδικας

```
void saxpy_serial
(int n, float alpha,float *x,float *y)
{
for(int i = 0;i < n;i++)
y[i] = alpha*x[i] + y[i];
}
```

Κώδικας CUDA

```
__global__ void saxpy_parallel
(int n, float alpha,float *x,float *y)
{
int i = blockIdx.x*blockDim.x+threadIdx.x;
if (i<n)
y[i] = alpha*x[i] + y[i];
}
```

Έχοντας τους πολυεπεξεργαστές στο μυαλό μας…

- Κάθε πολυεπεξεργαστής hardware έχει την ικανότητα να επεξεργαστεί ενεργά πολλαπλά blocks σε μία φορά.
- Το πόσα εξαρτάται από τον αριθμό των καταχωρητών ανά νήμα και πόση κοινή μνήμη ανά block απαιτείται από έναν δοθέντα kernel.
- Τα blocks που υπόκεινται σε επεξεργασία από έναν πολυεπεξεργαστή σε κάποια στιγμή αναφέρονται ως “ενεργά”.
- Εάν ένα block είναι πολύ μεγάλο, τότε δε θα χωρέσει στους πόρους ενός MP.

“Warps”

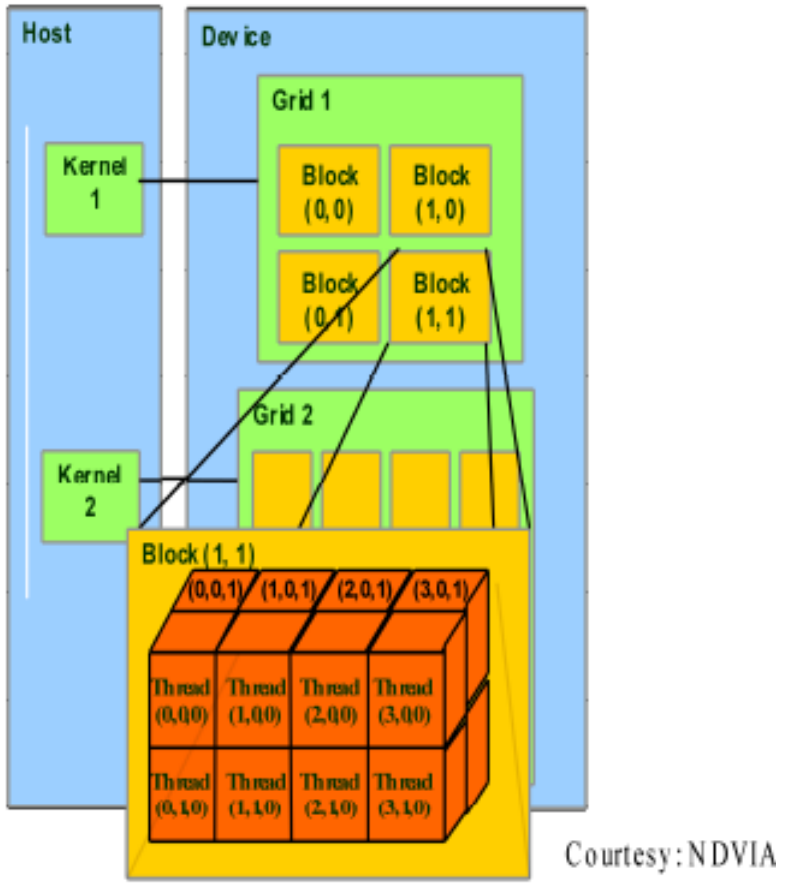
- Κάθε ενεργό block χωρίζεται σε ομάδες νημάτων SIMD ("Single Instruction Multiple Data") που ονομάζονται "warps".
- Κάθε warp περιέχει τον ίδιο αριθμό νημάτων, που ονομάζεται "warp size", τα οποία εκτελούνται από πολυεπεξεργαστές με το τρόπο των SIMD.
- Σε δηλώσεις “if” ή “while” (μεταβίβαση ελέγχου) τα νήματα ίσους αποκλίνουν.
- Χρήση: __syncthreads()

Γρήγορη επισκόπηση Ορολογίας

- ***Thread (Νήμα)*** : Συντρέχων κώδικας και σχετική μορφή εκτελεσμένη σε συσκευή CUDA (σε παραλληλία με άλλα νήματα).
  - Η μονάδα παραλληλισμού στη CUDA.
- ***Warp***: ένα σύνολο νημάτων που εκτελούνται φυσικά σε παραλληλία με το G80/GT200.
- ***Block***: ένα σύνολο από νήματα που εκτελούνται μαζί και διαμορφώνουν τη μονάδα εκχώρησης πόρων.
- ***Grid (Πλέγμα)***: ένα σύνολο από blocks νημάτων τα οποία πρέπει να ολοκληρωθούν όλα πριν την εφαρμογή της επόμενης κλήσης προγράμματος kernel.

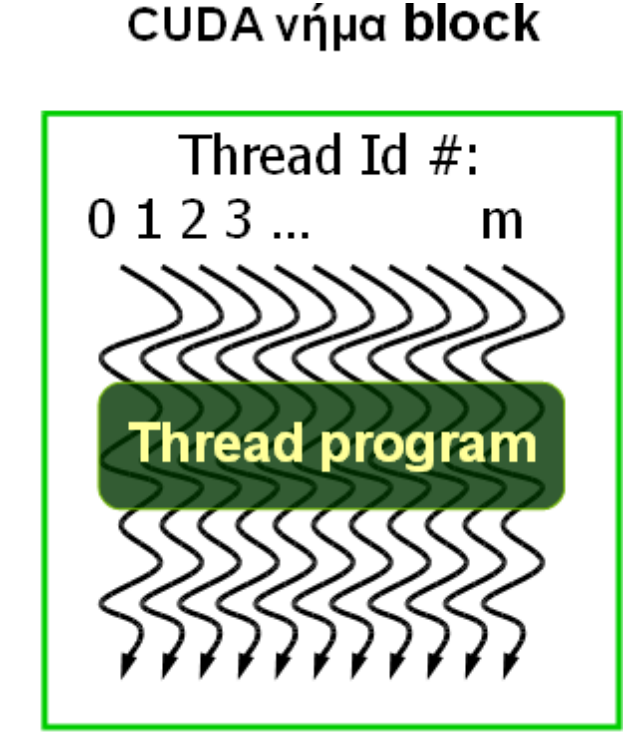
Πλέγματα και Blocks

- Ένας kernel εκτελείται ως ένα πλέγμα από blocks νημάτων.
  - Όλα τα νήματα μοιράζονται το χώρο της καθολικής μνήμης.
- Ένα block νημάτων είναι ένα σύνολο από νήματα που μπορούν να συνεργάζονται το ένα με το άλλο:
  - Συγχρονίζοντας την εκτέλεσή τους, χρησιμοποιώντας ένα barrier.
  - Διαμοιράζοντας αποτελεσματικά τη μνήμη μέσω μιας χαμηλής καθυστέρησης κοινή μνήμη.
  - Δύο νήματα από δύο διαφορετικά blocks δεν μπορούν να συνεργαστούν.



CUDA, Block νημάτων: Ανασκόπηση

- Ένας προγραμματιστής δηλώνει block (νημάτων):
  - Μέγεθος block 1 εώς 512 ταυτόχρονα νήματα.
  - Σχήμα block **1D**, **2D**, or **3D**.
  - Διαστάσεις block σε νήματα.
- Όλα τα νήματα σε ένα block εκτελούν το ίδιο πρόγραμμα νήματος.
- Τα νήματα μοιράζονται δεδομένα και συγχρονίζονται καθώς κάνουν το διαμοιρασμό της εργασίας.
- Τα νήματα έχουν **thread id** αριθμούς μέσα στα blocks.
- Το πρόγραμμα νήματος χρησιμοποιεί **thread id** για να επιλέξει την εργασία και τη διεύθυνση των κοινών δεδομένων.

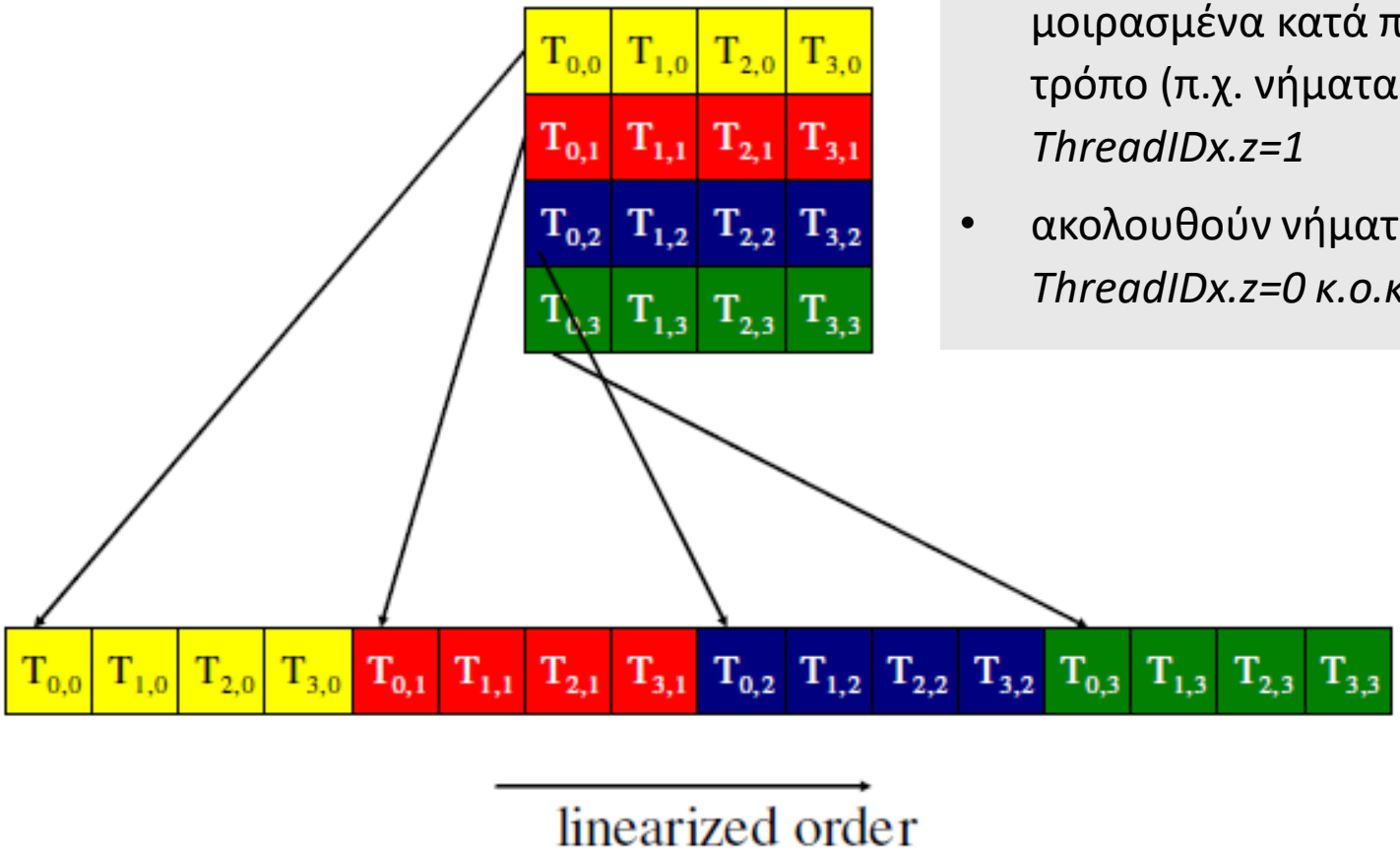


Courtesy: John Nickolls, NVIDIA

Πώς διαμερίζονται τα blocks νημάτων; (1/2)

- Τα blocks νημάτων διαμερίζονται σε warps.
- Τα ID νήματος μέσα σε ένα warp είναι διαδοχικά και αυξανόμενα. Αν το block είναι 1D τότε το threadID= threadIdx.x
  - Warp 0: thread 0, …thread 31;
  - Warp 1: thread 32 ,…, thread 63
  - Warp n: thread 32*n ,…., thread 32(n+1)-1
  - Για blocks με # από νήματα που δεν είναι πολλαπλάσια του 32:
    - Το τελευταίο warp γεμίζεται με επιπλέον νήματα για να συμπληρωθούν τα 32 νήματα.
    - π.χ ένα block: 48 νήματα - 2 warps, το warp 1 θα παραγεμιστεί με 16 επιπλέον νήματα.

Πώς διαμερίζονται τα blocks νημάτων; 2D blocks νημάτων



- 3D blocks νημάτων είναι μοιρασμένα κατά παρόμοιο τρόπο (π.χ. νήματα με *ThreadIDx.z=1*
- ακολουθούν νήματα με *ThreadIDx.z=0 κ.ο.κ.*

Πώς διαμερίζονται τα blocks νημάτων; (2/2)

- Ο διαμερισμός είναι πάντα ο ίδιος.
- Έτσι μπορείτε να χρησιμοποιήσετε αυτή τη γνώση στη ροή ελέγχου.
- Το ακριβές μέγεθος των warps ίσως αλλάξει από γενιά σε γενιά συσκευών.
- **Όπως και να 'χει, ΜΗΝ βασίζεστε σε οποιαδήποτε διάταξη μεταξύ warps.**
- Αν υπάρχουν οτιδήποτε εξαρτήσεις μεταξύ των νημάτων, θα πρέπει να χρησιμοποιήσετε __syncthreads() για να πάρετε σωστά αποτελέσματα.

Εντολές Ροής Ελέγχου

- Το hardware εκτελεί μια εντολή για όλα τα νήματα στο ίδιο warp πριν πάει στην επόμενη εντολή.
  - SIMT: *single instruction, multiple thread* execution (μονή εντολή, πολλαπλή εκτέλεση νημάτων)
    - Λειτουργεί καλά όταν όλα τα νήματα στο wrap ακολουθούν το ίδιο μονοπάτι ροής ελέγχου.
- Κύρια ανησυχία απόδοσης με τη διακλάδωση είναι η απόκλιση: *π.χ νήματα μεταξύ ενός μονού warp ακολουθούν διαφορετικά μονοπάτια.*
- Διαφορετικά μονοπάτια εκτέλεσης είναι serialized.
  - Παράδειγμα: *if-then-else* εκτελείτε σε δύο φάσεις.
    - Μία για νήματα που εκτελούν το μονοπάτι *then.*
    - Και μία δεύτερη για νήματα που εκτελούν τον μονοπάτι else.
- Τα μονοπάτια ελέγχου που παίρνονται από τα νήματα των warps διαβαίνονται ένα τη φορά μέχρι να μην υπάρχουν άλλα.

Εντολές Ροής Ελέγχου (συνέχεια)

- Μια συχνή περίπτωση: αποφυγή απόκλισης όταν μια συνθήκη διακλάδωσης αποτελεί συνάρτηση του ID νήματος.
- Παράδειγμα με απόκλιση:
  - If (threadIdx.x > 2) { }
  - Αυτό δημιουργεί δύο διαφορετικά μονοπάτια ελέγχου για νήματα σε ένα block.
  - Βαθμός ανάλυσης κλάδου (branch granularity) < warp size: Τα νήματα 0 και 1 ακολουθούν διαφορετικό μονοπάτι από τα υπόλοιπα νήματα στο πρώτο warp.
- Παράδειγμα χωρίς απόκλιση:
  - If (threadIdx.x / WARP_SIZE > 2) { }
  - Επίσης, δημιουργεί δύο διαφορετικά μονοπάτια ελέγχου για νήματα σε ένα block.
  - Ο βαθμός ανάλυσης κλάδου είναι ένα ολόκληρο πολλαπλάσιο του warp size: Όλα τα νήματα σε ένα οποιοδήποτε δοσμένο warp ακολουθούν το ίδιο μονοπάτι.

Παράλληλη Μείωση (Reduction)

- Δοθέντος ενός πίνακα με τιμές, η reduction τους σε μία μοναδική τιμή παράλληλα.
- Παραδείγματα
  - Sum Reduction: Άθροισμα όλων των τιμών στον πίνακα.
  - Max Reduction: Η μέγιστη τιμή από όλες στον πίνακα.
- Τυπικά παράλληλη υλοποίηση:
  - Αναδρομικά μείωση κατά το ήμισυ του αριθμού των νημάτων, προσθέτει δύο τιμές ανά νήμα.
  - Απαιτεί log(n) βήματα για n στοιχεία, απαιτώντας n/2 νήματα.

Ένα παράδειγμα reduction διανύσματος

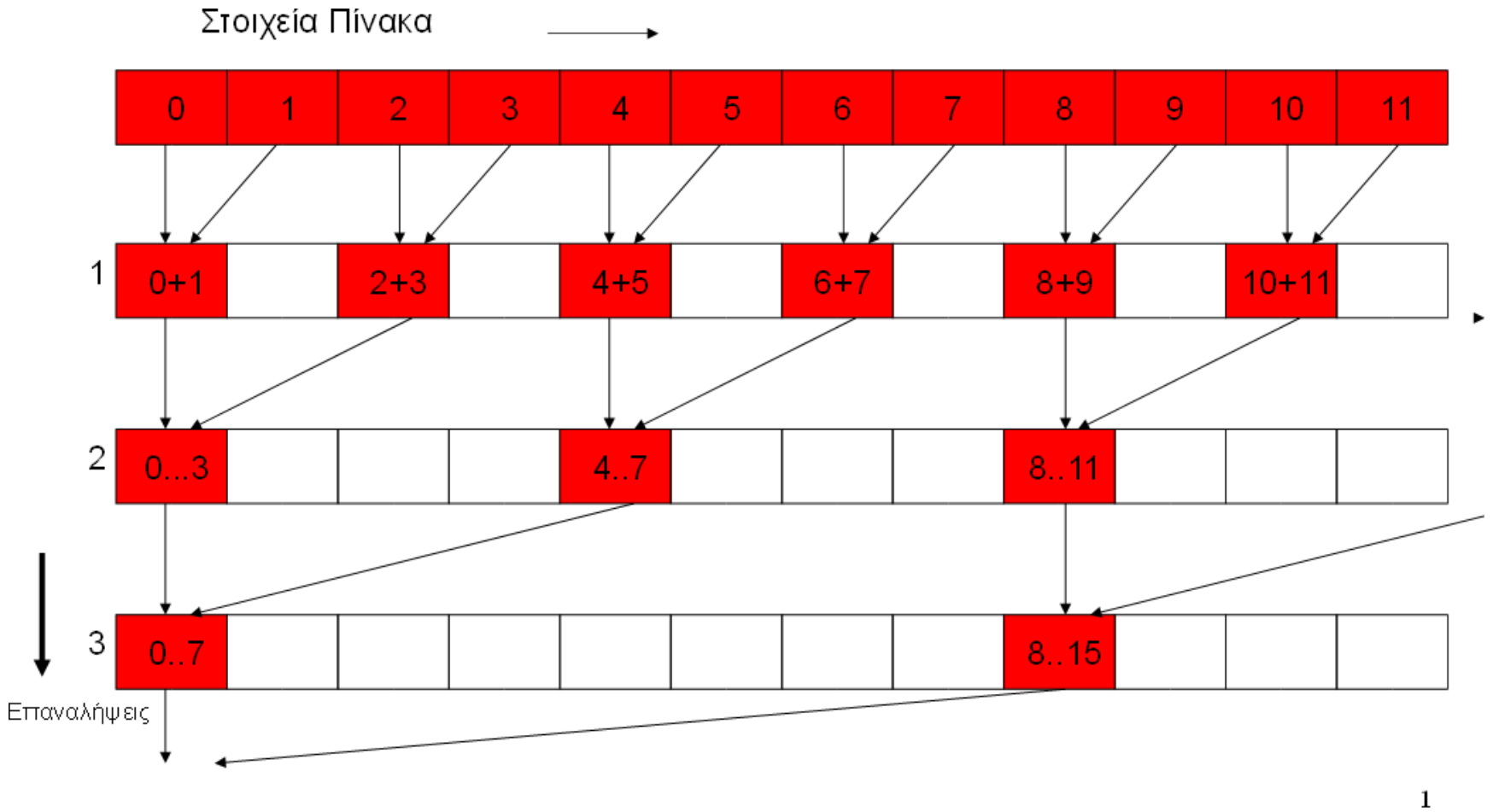
- Θεωρούμε ότι μια in-place reduction χρησιμοποιεί κοινή μνήμη.
  - Το αυθεντικό διάνυσμα βρίσκεται στη μηχανή καθολικής μνήμης.
  - Η κοινή μνήμη συνηθίζει να διατηρεί ένα διάνυσμα μερικού αθροίσματος.
  - Κάθε επανάληψη φέρνει το διάνυσμα μερικού αθροίσματος πιο κοντά στο τελικό άθροισμα.
  - Η τελική λύση θα βρίσκεται στο στοιχείο 0.

Μια απλή Εφαρμογή

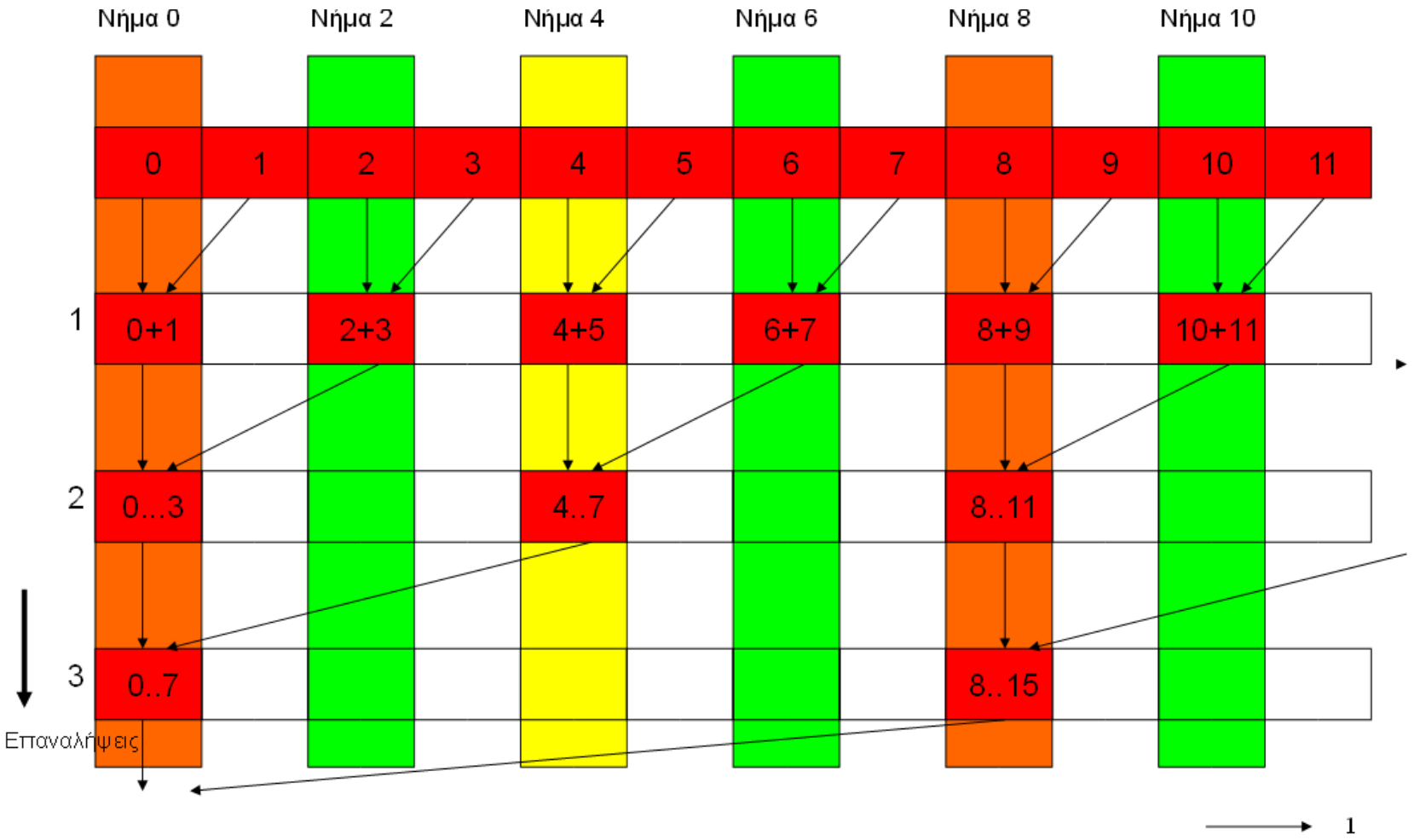
- Θεωρούμε ότι έχουμε ήδη φορτώσει έναν πίνακα μέσα:

```
__shared__ float partialSum[]
unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
stride < blockDim.x; stride *= 2)
{
__syncthreads();
if (t % (2*stride) == 0)
partialSum[t] += partialSum[t+stride];
}
```

Διανυσματική μείωση με Bank Conflicts



Διανυσματική μείωση με απόκλιση κλάδου



Μερικές Παρατηρήσεις

- Σε κάθε επανάληψη, δύο μονοπάτια ελέγχου ροής θα διασχίζονται διαδοχικά για κάθε warp.
  - Νήματα που διεξάγουν πρόσθεση και νήματα που δεν διεξάγουν.
  - Νήματα που δεν διεξάγουν πρόσθεση ίσως κοστίσουν επιπλέον κύκλους που εξαρτώνται από την υλοποίηση της απόκλισης.
- Θα εκτελούνται σε μια χρονική στιγμή όχι περισσότερα από τα μισά νήματα:
  - Όλα τα περαιτέρω Index νήματα είναι απενεργοποιημένα ακριβώς με το ξεκίνημα.
  - Κατά μέσο όρο, λιγότερο απ' το ¼ των νημάτων θα ενεργοποιηθούν για όλα τα warps με την πάροδο του χρόνου.
  - Ξεκινώντας με την 5η επανάληψη, ολόκληρα warps σε κάθε block θα απενεργοποιηθούν, φτωχή αξιοποίηση πόρων αλλά χωρίς απόκλιση.
    - Αυτό μπορεί να συνεχιστεί για λίγο, μέχρι και για 4 επιπλέον επαναλήψεις (512/32=16= 24), όπου κάθε επανάληψη έχει ένα μόνο νήμα ενεργοποιημένο μέχρι όλα τα warps να αποσυρθούν.

Ελλείψεις κατά την υλοποίηση

- Θεωρούμε ότι έχουμε ήδη φορτώσει έναν πίνακα μέσα:

```
__shared__ float partialSum[]

  unsigned int t = threadIdx.x;
  for (unsigned int stride = 1;
    stride < blockDim.x;  stride *= 2)
  {
    __syncthreads();
    if (t % (2*stride) == 0)
    partialSum[t] += partialSum[t+stride];
  }
```

BAD: Divergence due to interleaved branch decisions

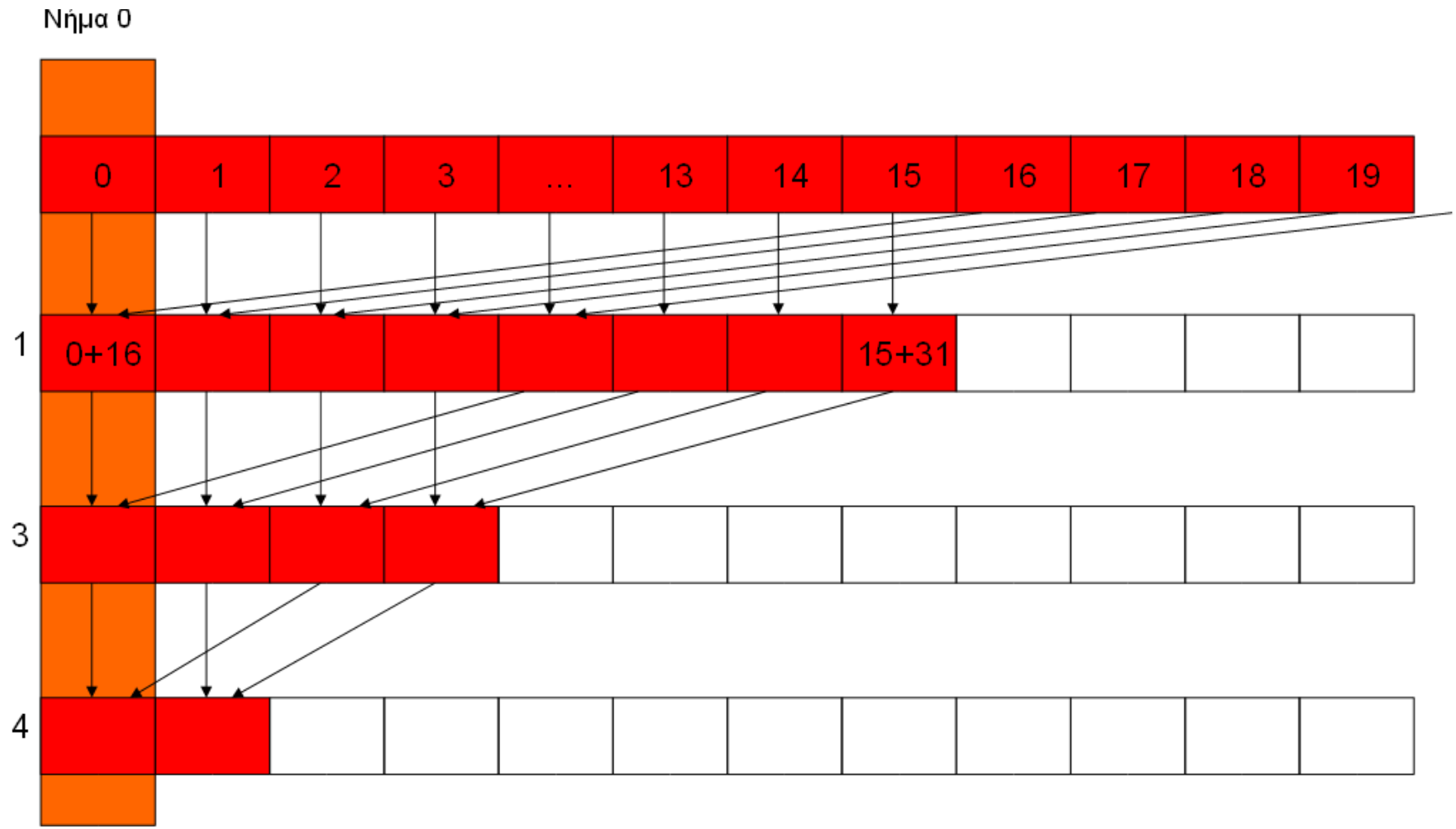
Μια καλύτερη Υλοποίηση

- Θεωρούμε ότι έχουμε ήδη φορτώσει έναν πίνακα μέσα:

```
__shared__ float partialSum[]

  unsigned int t = threadIdx.x;
  for (unsigned int stride = blockDim.x >> 1;
    stride > 1;  stride >> 1)
  {
    __syncthreads();
    if (t < stride)
    partialSum[t] += partialSum[t+stride];
  }
```

Εκτέλεση του διορθωμένου αλγορίθμου (If blockDim.x = 32)



Χωρίς απόκλιση, με τη χρήση αναθεωρημένου Αλγόριθμου

If blockDim.x = 512;

Στην 1η επανάληψη:

- Τα νήματα από 0 εώς 255 εκτελούν πρόσθεση.
- Τα νήματα από 256 εώς 511 δεν εκτελούν πρόσθεση.
- Τα Pair-wise αθροίσματα αποθηκεύονται σε στοιχεία 0 – 255 αργότερα.
- Όλα τα νήματα στα warps 1 - warp 8 εκτελούν πρόσθεση.
- Warps 9 - warp 15 παραλείπουν τις προσθέσεις.
- Όλα τα νήματα σε κάθε warp ακολουθούν το ίδιο μονοπάτι.
- Δεν υπάρχει απόκλιση νήματος!

Μερικές παρατηρήσεις σχετικά με τη νέα υλοποίηση

Επανάληψη	1	2	3	4	5	6	7	8	9
#Ενεργά warps	8	4	2	1	1	1	1	1	1
# Ενημερωμένα στοιχεία	256	128	64	32	16	8	4	2	1
Απόκλιση	No	No	No	No	Yes	Yes	Yes	Yes	Yes

- Μόνο οι τελευταίες πέντε επαναλήψεις θα έχουν απόκλιση.
- Όλα τα warps θα κλείνουν καθώς οι επαναλήψεις προχωράνε.
- Για ένα block των 512 νημάτων, θα γίνουν 4 επαναλήψεις για να τερματίσουν όλα τα warps αλλά κάθε φορά κι από ένα warp σε κάθε block.
- Καλύτερη αξιοποίηση των πόρων, θα αποσύρει πιθανώς τα warps κι έτσι και τα blocks πιο γρήγορα.

Μια πιθανή περεταίρω βελτίωση

Επανάληψη	1	2	3	4	5	6	7	8	9
#Ενεργά warps	8	4	2	1	1	1	1	1	1
#Ενημερωμένα στοιχεία	256	128	64	32	16	8	4	2	1
Απόκλιση	No	No	No	No	Yes	Yes	Yes	Yes	Yes

Για τους τελευταίους 6 βρόγχους μόνο ένα warp είναι ενεργό (π.χ tid's 0..31)

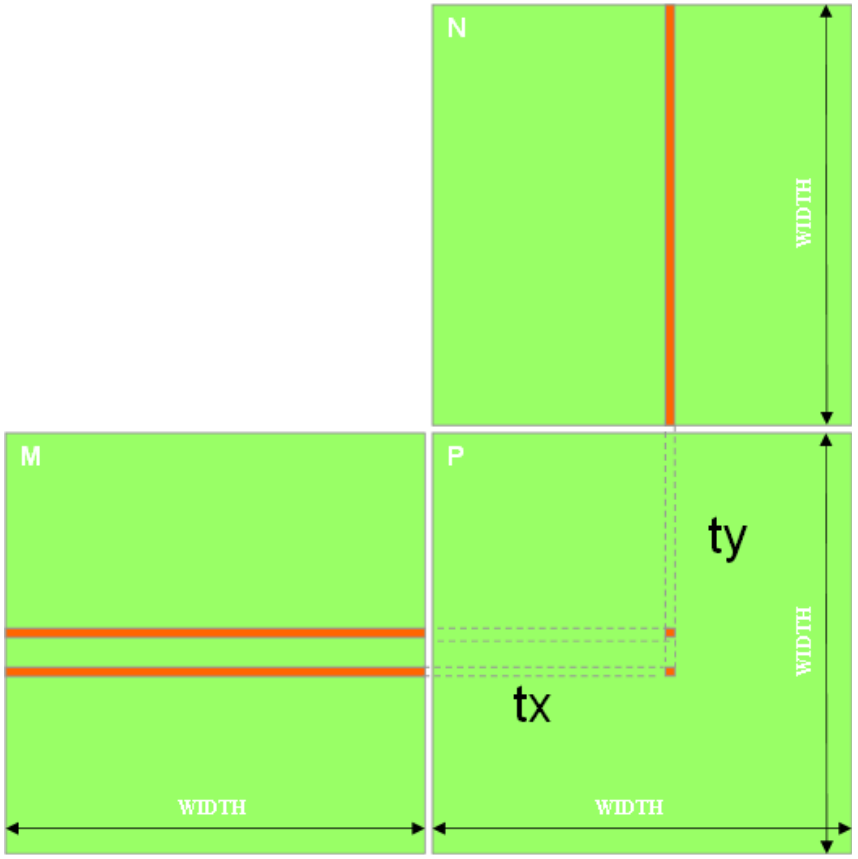
- Κοινά διαβάσματα και εγγραφές είναι *SIMD synchronous* εντός ενός warp.

Έτσι παραλείπει τα __syncthreads() και κάνει αναδίπλωση των τελευταίων 6 επαναλήψεων.

```
unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x >> 1;
stride > 1; stride >> 1)
{
__syncthreads();
if (t < stride)
partialSum[t] += partialSum[t+stride];
} (συνεχίζεται)
```

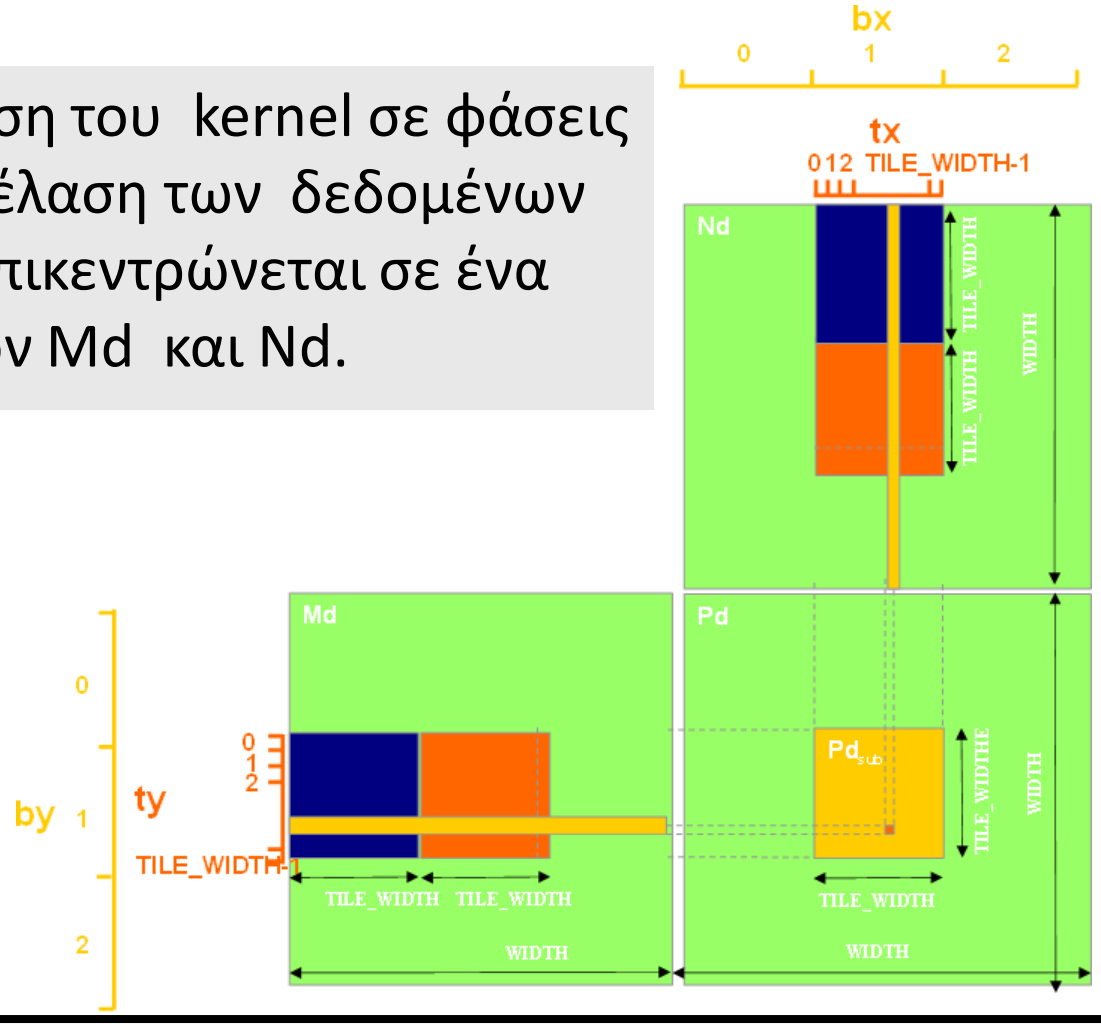
Χρήση της Κοινής μνήμης για την επαναχρησιμοποίηση των δεδομένων της Καθολικής μνήμης

- Κάθε στοιχείο εισόδου διαβάζεται από το πλάτος νημάτων..
- Φόρτωση κάθε στοιχείου στην κοινή μνήμη και τοποθέτηση πολλών νημάτων να χρησιμοποιούν την τοπική έκδοση για να μειώσουν το εύρος ζώνης
  - Tiled Αλγόριθμοι

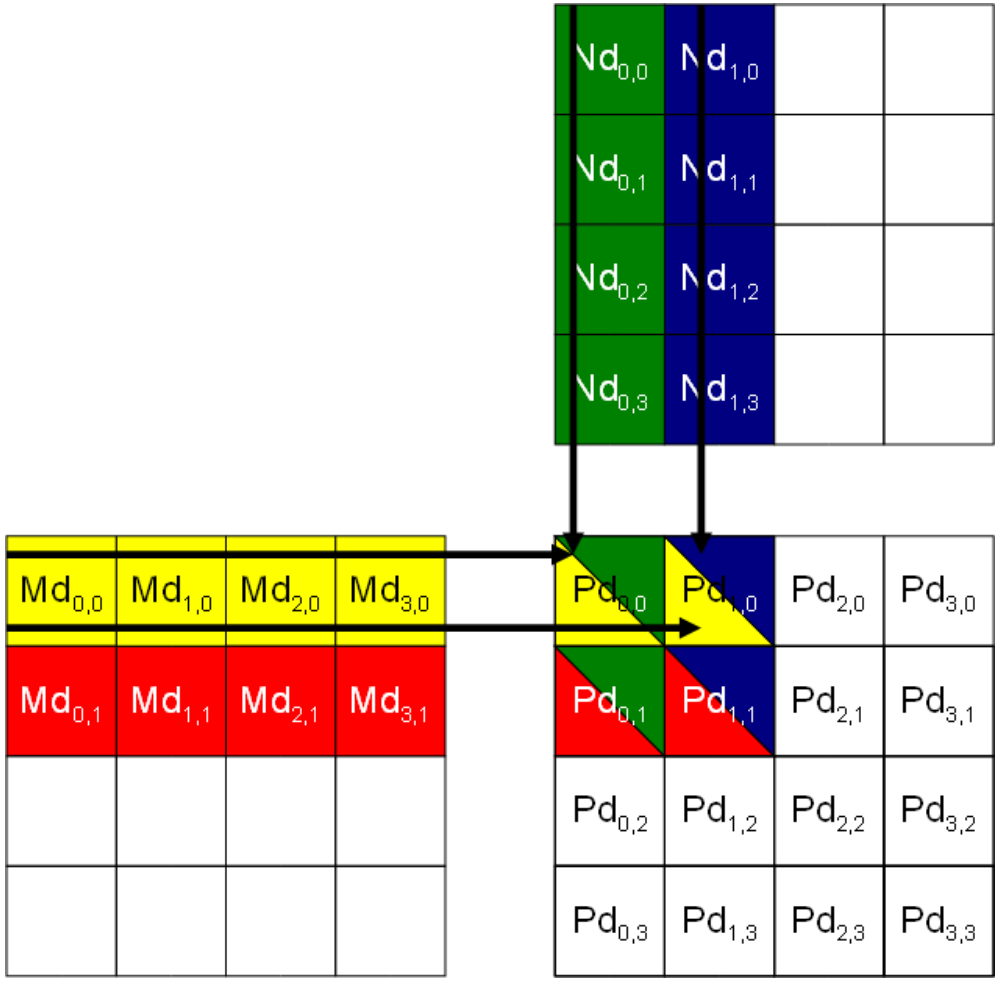


Tiled Multiply (1/2)

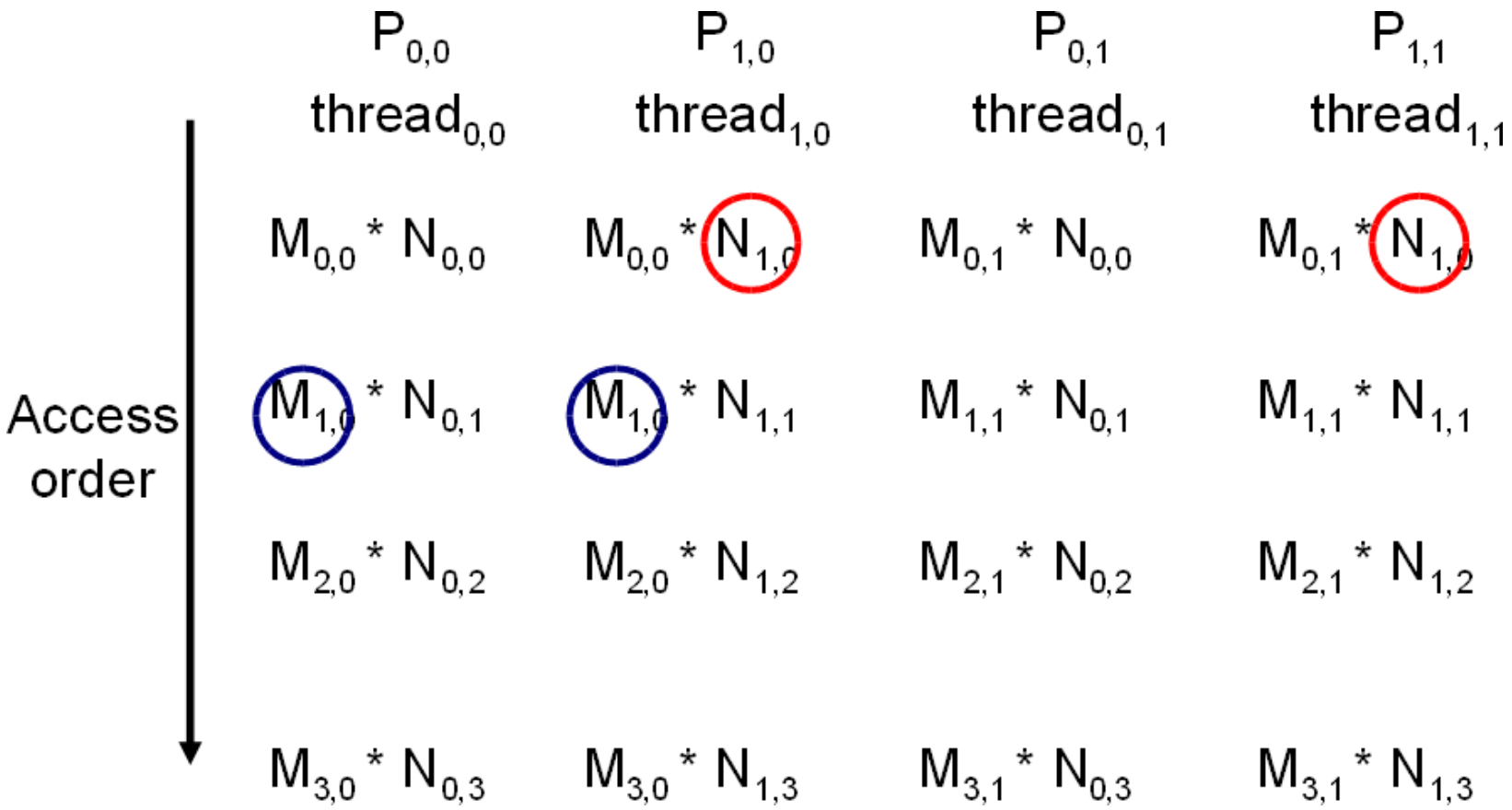
- Χωρίστε την εκτέλεση του kernel σε φάσεις έτσι ώστε η προσπέλαση των δεδομένων σε κάθε φάση να επικεντρώνεται σε ένα υποσύνολο (tile) των Md και Nd.



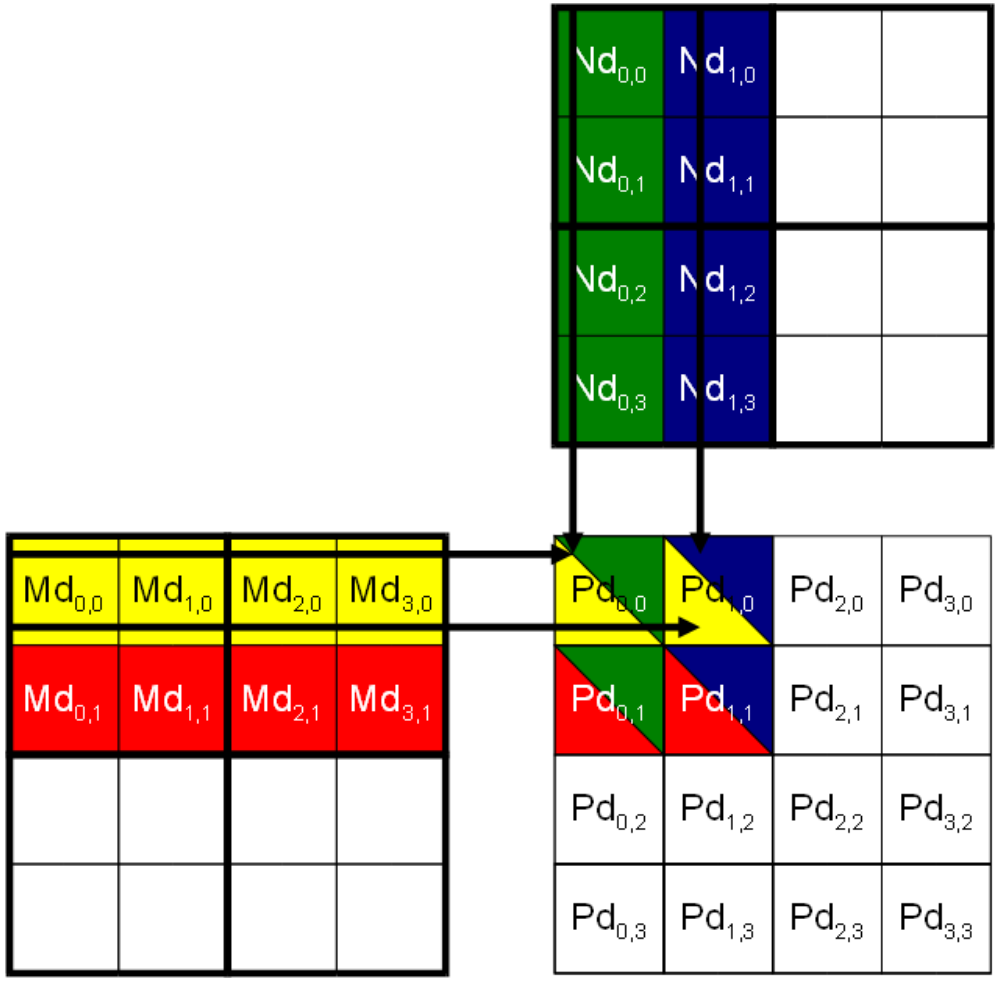
Ένα μικρό παράδειγμα



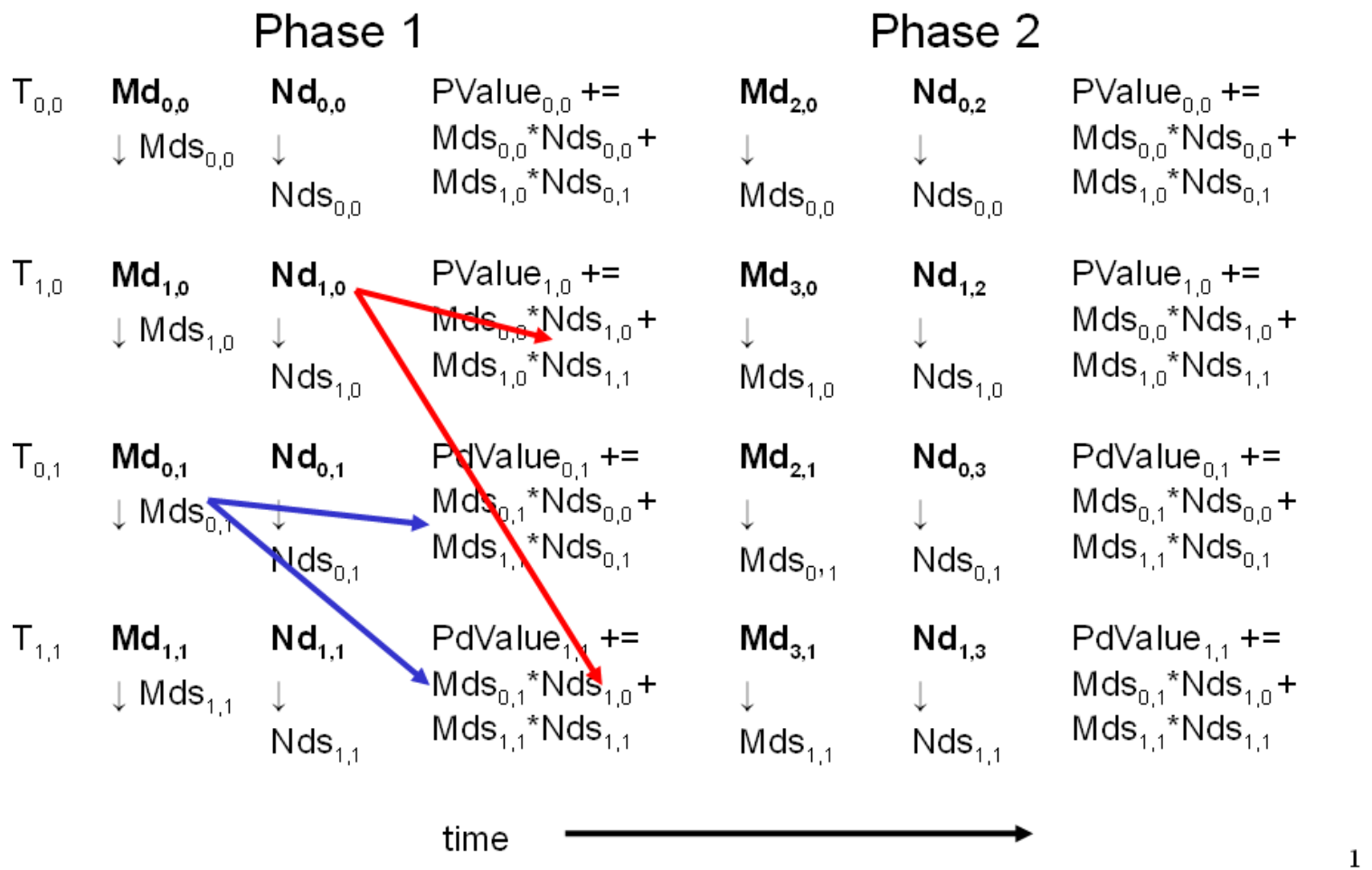
Κάθε στοιχείο Md & Nd χρησιμοποιείται ακριβώς δυο φορές στην παραγωγή ενός 2X2 tile τύπου P



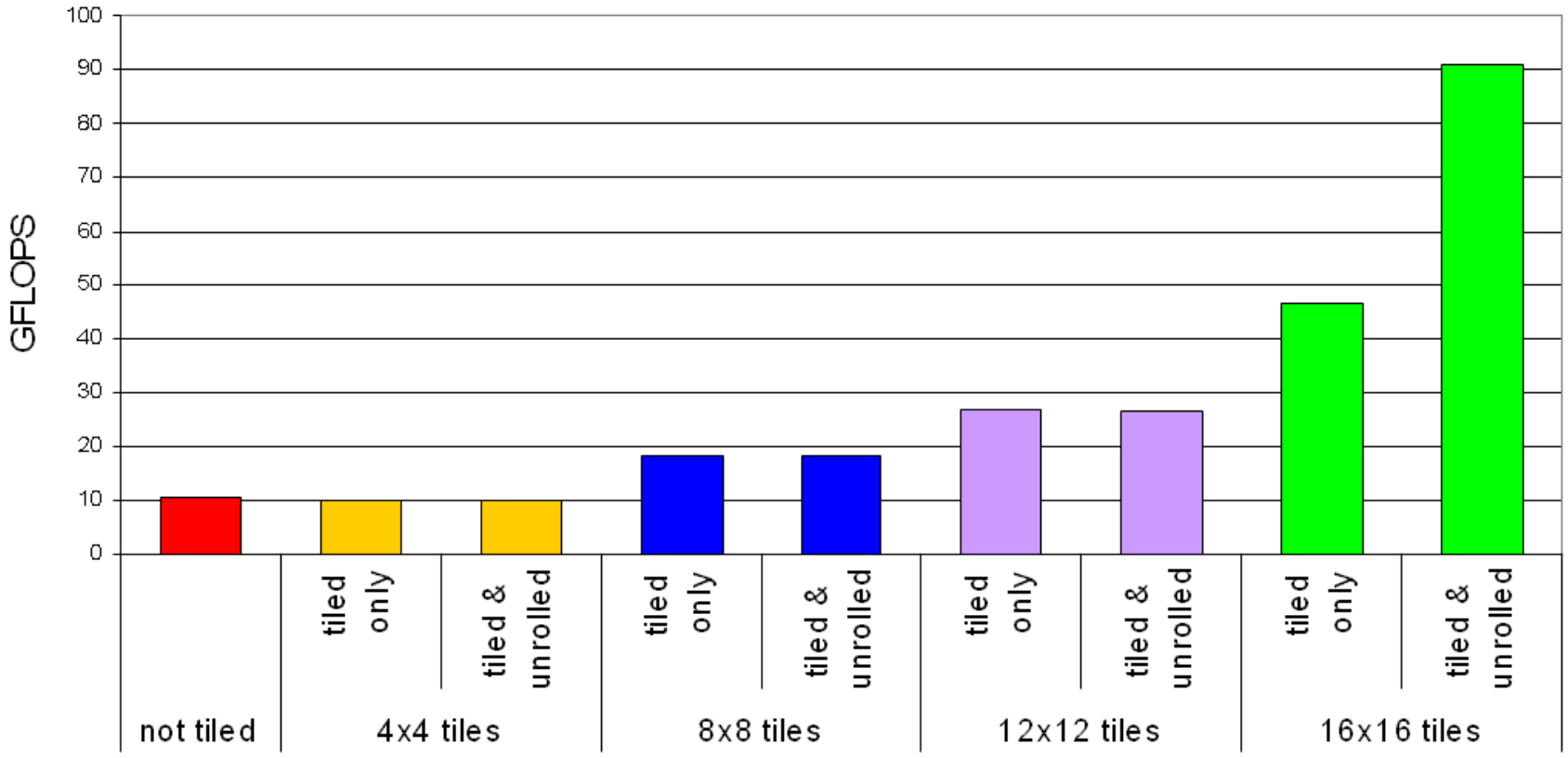
Χωρίζοντας Md και Nd σε Tiles



Κάθε φάση ενός block νημάτων χρησιμοποιεί ένα tile από Md και ένα από Nd



Tiling αποτελέσματα μεγέθους



Σύνοψη - Τυπική δομή ενός προγράμματος CUDA

- **Δήλωση καθολικών μεταβλητών**
  - **__host__**
  - **__device__… __global__, __constant__, __texture__**
- Πρωτότυπα συναρτήσεων
- __global__ void kernelOne(…)
- float handyFunction(…)
- Main ()
  - Εκχώρηση χώρου μνήμης στη συσκευή – cudaMalloc**(&d_GlblVarPtr,** bytes)
  - Μεταφορά δεδομένων από τον host στη συσκευή – cudaMemCpy**(d_GlblVarPtr, h_Gl…)**
  - Εκτέλεση ρύθμισης εγκατάστασης
  - Κλήση του kernel – kernelOne <<<εκτέλεση εγκατάστασης>>>(args…);
  - Μεταφορά αποτελεσμάτων από τη συσκευή στον host – cudaMemCpy(h_GlblVarPtr,…)
  - Προαιρετικά: Σύγκριση με χρυσή λύση (host computed)
- Kernel – void kernelOne(type args,…)
- Δήλωση μεταβλητών - __local__, __shared__
  - **Αυτόματες μεταβλητές αναθέτονται ξεκάθαρα σε καταχωρητές ή τοπική μνήμη**
  - **syncthreads()…**
- Άλλες συναρτήσεις
  - float handyFunction(int inVar…);

(Παράρτημα) Συμβουλές για βελτίωση της επίδοσης

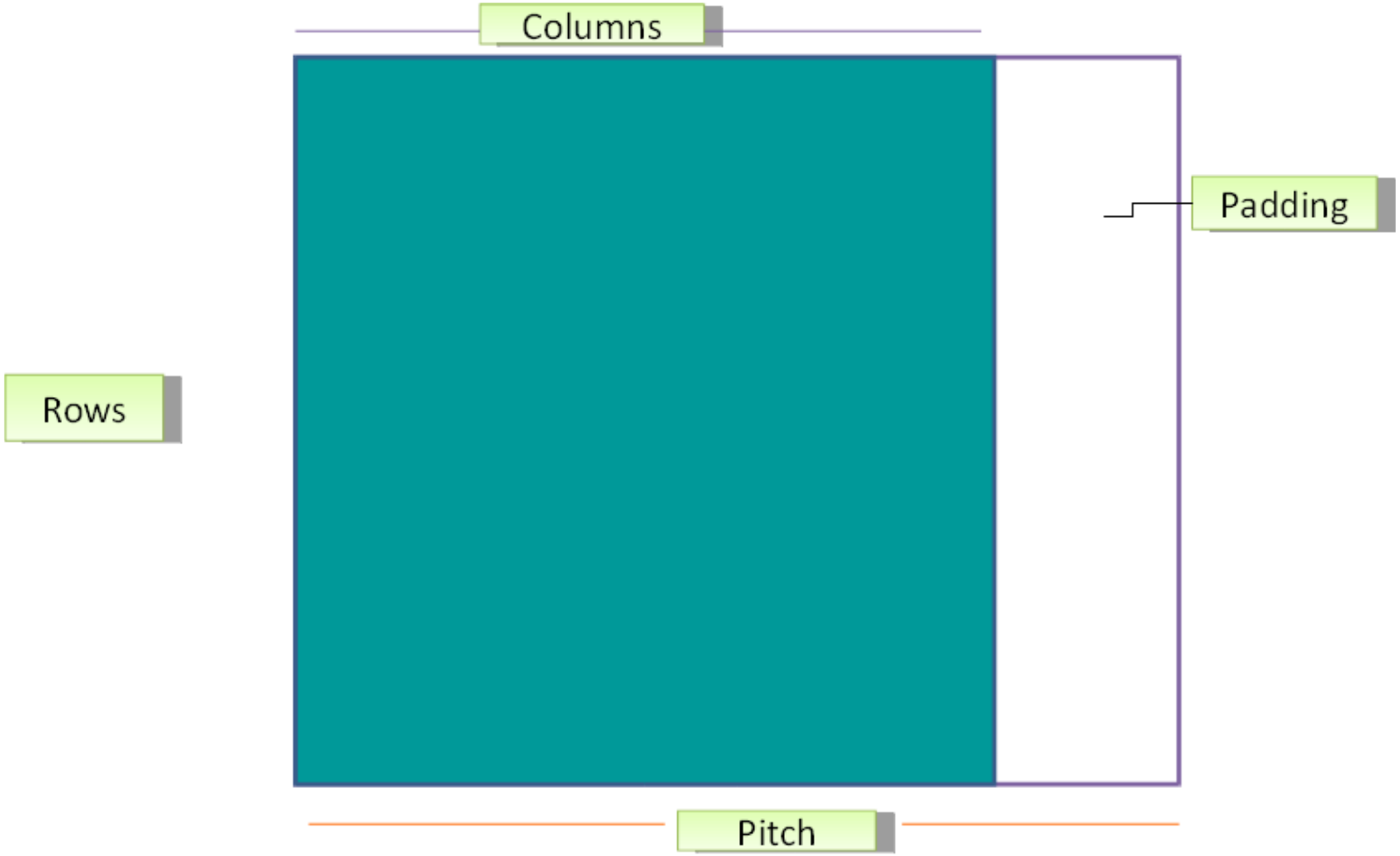
Ευθυγράμμιση Μνήμης

- Η προσπέλαση μνήμης στην GPU δουλεύει αρκετά καλύτερα εάν τα στοιχεία δεδομένων είναι ευθυγραμμισμένα σε σύνορα των 64 byte.
- Συνεπώς, η εκχώρηση δισδιάστατων πινάκων, έτσι ώστε κάθε γραμμή να ξεκινά από μια οριοθετιμένη διεύθυνση των 64 byte θα βελτιώσει την απόδοση.
- Όμως, αυτό είναι δύσκολο να γίνει για έναν προγραμματιστή.

Εκχωρώντας δισδιάστατους πίνακες με “pitch”

- Το CUDA προσφέρει ειδικές εκδόσεις:
  - Εκχώρησης μνήμης των δισδιάστατων πινάκων έτσι ώστε κάθε γραμμή να είναι γεμάτη (εάν χρειάζεται). Η συνάρτηση ορίζει το καλύτερο pitch και το επιστρέφει στο πρόγραμμα. Το όνομα της συνάρτησης είναι cudaMallocPitch().
  - Οι λειτουργίες του αντιγράφου μνήμης που λαμβάνουν υπ' όψη το pitch που είχε επιλεχθεί από την λειτουργία κατανομής μνήμης. Το όνομα της συνάρτησης είναι cudaMemcpy2D().

Pitch



Ένα απλό παράδειγμα

- Δείτε pitch.cu
- Ένας πίνακας 30 γραμμών και 10 στηλών.
- Η δουλειά χωρίζεται σε 3 blocks των 10 γραμμών:
  - Το μέγεθος block είναι 10.
  - Το μέγεθος του πλέγματος είναι 3.
- Κατεβάστε κώδικα από:
  - http://sc08.sc-education.org/conference/pdg/mon/cuda/pitch.cu

Βασικά τμήματα του κώδικα (1/2)

```
result = cudaMallocPitch(
    (void **)&devPtr,
    &pitch,
    width*sizeof(int),
    height);
```

Βασικά τμήματα του κώδικα (2/2)

```
result = cudaMemcpy2D(
    devPtr,
    pitch,
    mat,
    width*sizeof(int),
    width*sizeof(int),
    height,
    cudaMemcpyHostToDevice);
```

Μέσα στον kernel

```
__global__    void myKernel(int *devPtr,
   int pitch,
   int width,
   int height)
{
int c;
int thisRow;
thisRow = blockIdx.x * 10 + threadIdx.x;

int *row = (int *)((char *)devPtr +
              thisRow*pitch);
for(c = 0;c < width;c++)
row[c] = row[c] + 1;
}
```

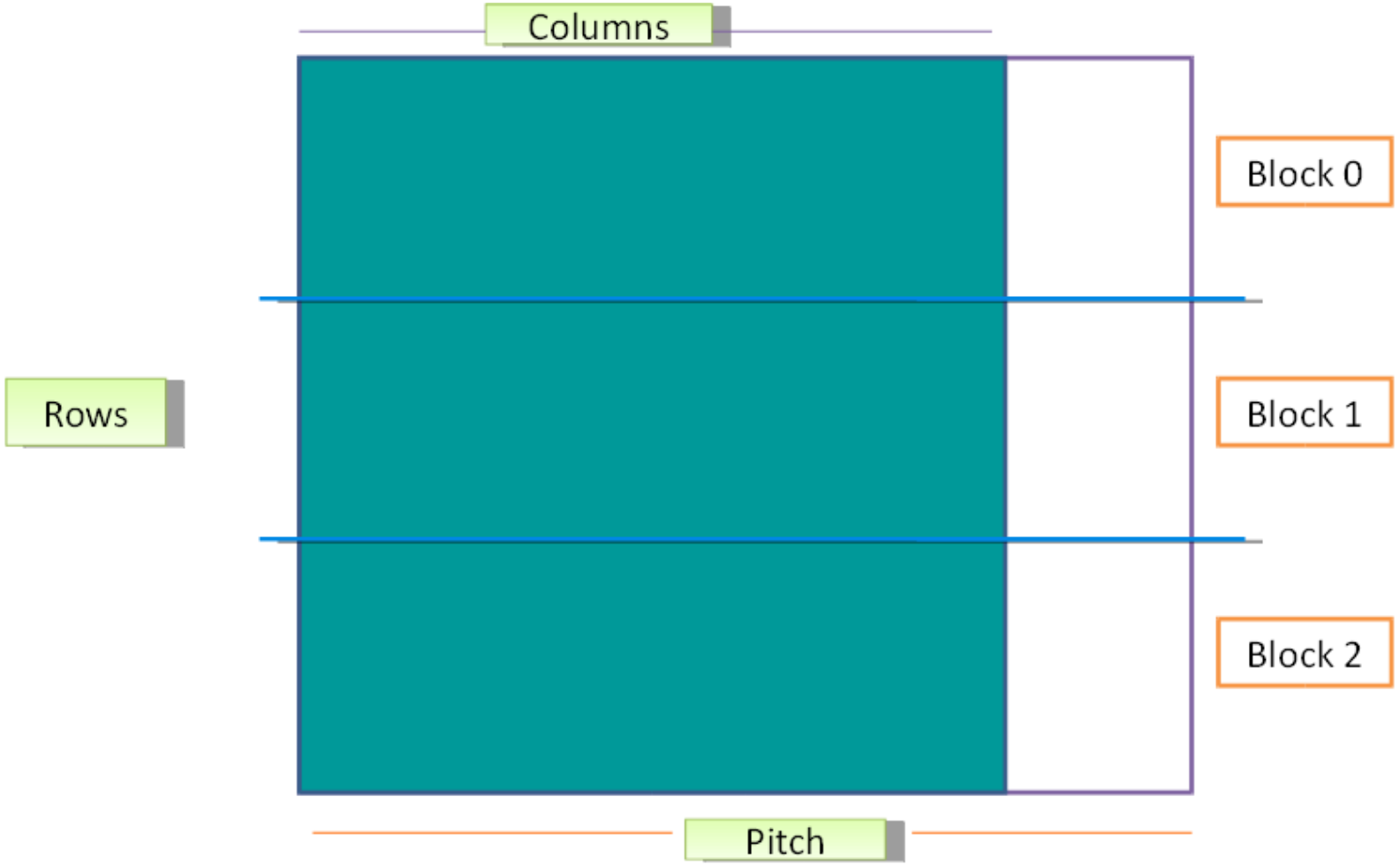
Η κλήση προς τον kernel

```
myKernel<<<3,10>>>(
devPtr,
pitch,
width,
height);
```

pitch -> Καταμερισμός εργασίας μεταξύ των γραμμών

- Παρατηρείστε πως κατά τη χρήση του pitch, διαμοιράζουμε την εργασία μέσω γραμμών.
- Αντί να χρησιμοποιήσουμε την δισδιάστατη αποσύνθεση των δισδιάστατων blocks, χωρίζουμε τον δισδιάστατο πίνακα σε blocks από γραμμές.

Καταμερισμός εργασίας μεταξύ των blocks



Συμβουλή απόδοσης: Μέγεθος block

- Σημαντικό για την απόδοση.
- Συνιστώμενη τιμή είναι το 192 ή το 256.
- Μέγιστη τιμή είναι το 512.
- Θα πρέπει να είναι πολλαπλάσιο του 32, δεδομένου ότι αυτό είναι το warp μέγεθος για τη Series 8 των GPUs, κι έτσι και το μέγεθος εκτέλεσης για πολυεπεξεργαστές.
- Περιορισμένο από τον αριθμό καταχωρητών στο MP.
- Τα Series 8 GPU MPs έχουν 8192 καταχωρητές οι οποίοι μοιράζονται μεταξύ όλων των νημάτων σε ένα MP.

Συμβουλή απόδοσης: Μέγεθος πλέγματος

- Σημαντικό για το scalability.
- Συνίσταται τιμή τουλάχιστον 100, αλλά η τιμή 1000 θα scale για πολλές γενιές hardware.
- Η πραγματική τιμή εξαρτάται στο μέγεθος των δεδομένων του προβλήματος.
- Θα πρέπει να είναι πολλαπλάσιο του αριθμού των Mps για μία ομαλή κατανομή της εργασίας (αν και δεν είναι απαραίτητο).
- Παράδειγμα: 24 blocks.
- Το πλέγμα μπορεί να λειτουργήσει αποτελεσματικά στη Series 8 (12 Mps), θα σπαταλίσει όμως πόρους στις νέες GPUs με 32MPs.

Συμβουλή απόδοσης: Απόκλιση κώδικα

- Απόκλισης εντολών ροής ελέγχου (τα νήματα ακολουθούν διαφορετικά μονοπάτια εκτέλεσης).
- Παράδειγμα: if, for, while.
- Ο αποκλίνων κώδικας αποτρέπει την SIMD να εκτελεστεί – αναγκάζει σειριακή εκτέλεση (σκοτώνει την αποδοτικότητα).
- Μια προσέγγιση είναι να προσκαλούν έναν απλούστερο kernel πολλές φορές.
- Ελεύθερη χρήση των __syncthreads()

Συμβουλή απόδοσης: Καθυστέρηση μνήμης

- 4 κύκλοι ρολογιού για κάθε διάβασμα/εγγραφή μνήμης συν επιπλέον 400-600 κύκλους για καθυστέρηση.
- Η καθυστέρηση της μνήμης μπορεί να μείνει κρυφή διατηρώντας ένα μεγάλο αριθμό νημάτων απασχολημένο.
- Κρατήστε τον αριθμό από νήματα ανά block (μέγεθος block) και τον αριθμό blocks ανά πλέγμα (μέγεθος πλέγματος) όσο μεγαλύτερο μπορείτε.
- Η σταθερή μνήμη μπορεί να χρησιμοποιηθεί για σταθερά δεδομένα (μεταβλητές που δεν αλλάζουν)
- Η σταθερή μνήμη είναι cached.

Συμβουλή απόδοσης: Αναγνώσεις μνήμης

- Η συσκευή είναι ικανή να διαβάσει ένα αριθμό των 32, 64 ή 128-bit από τη μνήμη με μια μόνο εντολή.
- Τα δεδομένα πρέπει να είναι ευθυγραμμισμένα στη μνήμη (αυτό μπορεί να επιτευχθεί χρησιμοποιώντας κλήσεις cudaMallocPitch()).
- Αν διαμορφωθεί σωστά, πολλά νήματα ενός warp μπορούν να παραλάβουν το καθένα ένα κομμάτι μνήμης με μία μόνο εντολή ανάγνωσης.

Watchdog timer

- Το λειτουργικό σύστημα GUI ίσως να έχει έναν "watchdog" timer που προκαλεί τα προγράμματα τα οποία χρησιμοποιούν τον κύριο προσαρμογέα γραφικών να λήγουν αν τρέχουν παραπάνω από τον μέγιστο επιτρεπόμενο χρόνο.
- Οι εκκινήσεις ενός προγράμματος μεμονωμένης GPU είναι περιορισμένες σε ένα χρόνο εκτέλεσης πολύ μικρότερο του μεγίστου.
- Υπέρβαση αυτού του χρονικού ορίου συνήθως προκαλεί αποτυχία εκκίνησης.
- Πιθανή λύση: τρέχουμε CUDA σε μια GPU που ΔΕΝ συνδέεται σε οθόνη.

On line πηγές

- http:/ /www.acmqueue.org/modules.php?name=Content&pa=showpage&pid=532
- http:/ /www.ddj.com/hpc-high-performance-computing/207200659
- http:/ /www.nvidia.com/object/cuda_home.html#
- http:/ /www.nvidia.com/object/cuda_learn.html

- "Computation of Voronoi diagrams using a graphics processing unit" by Igor Majdandzic et al. available through IEEE Digital Library, DOI: 10.1109/EIT.2008.4554342

Βιβλιογραφία

- **CUDA: Introduction** Christian Trefftz / Greg Wolffe, Grand Valley State University Supercomputing 2008 (Education Program).
- **Control Flow/ Thread Execution &CUDA Memories** David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009, ECE 498AL, University of Illinois, Urbana-Champaign (lect15-Cuda-control_flow.ppt, lecture5-cuda-memory-spring-2010.ppt).
- CUDA Parallel Programming TutorialRichard Membarth, University of Erlangen-Nuremberg, 2009.

Τέλος Ενότητας