


# Παράλληλα Συστήματα

Χειμερινό εξάμηνο 2024–2025

CUDA #1



# CUDA – Εισαγωγή

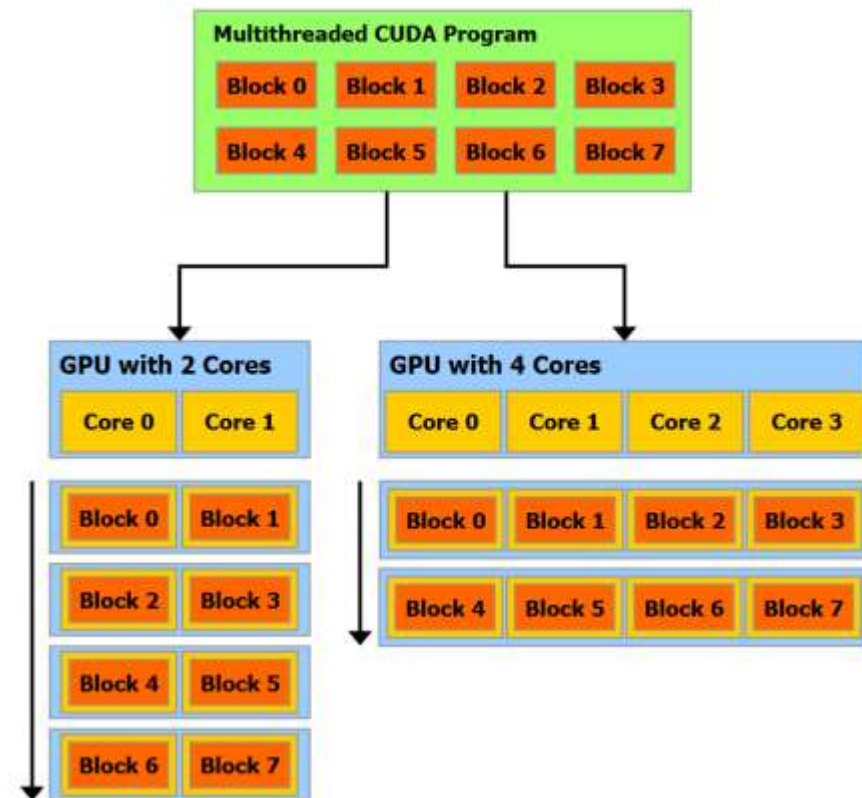
- ▶ Compute Unified Device Architecture
  - ▶ Η CUDA είναι μια πλατφόρμα παράλληλου προγραμματισμού που αναπτύχθηκε από την Nvidia το 2007
  - ▶ Κύριος στόχος της CUDA είναι να δώσει στον προγραμματιστή τη δυνατότητα να χρησιμοποιήσει την τεράστια υπολογιστική ισχύ μιας κάρτας γραφικών
- 

# GPU & Streaming Multiprocessors

- ▶ Μια GPU της NVIDIA έχει «κτιστεί» γύρω από ένα σύνολο από scalable streaming multiprocessors ή πιο απλά SM.
- ▶ Ένα SM μέσα σε μια GPU είναι υπεύθυνο για την παράλληλη (concurrent) εκτέλεση ομάδων από νήματα (threads).
- ▶ Όταν μια ομάδα νημάτων ανατίθεται σε ένα SM, αυτά παραμένουν μέχρι το τέλος της ζωής τους.
- ▶ Κάθε SM αποτελείται από ένα σύνολο πυρήνων, διαμοιραζόμενη μνήμη (shared memory), καταχωρητές, μονάδα load/store και μια μονάδα χρονοπρογραμματισμού (scheduler)

# Scalable προγραμματιστικό μοντέλο

- ▶ Ένα πολυνημαντικό (multithreaded) πρόγραμμα χωρίζεται σε blocks από threads που εκτελούνται ανεξάρτητα το ένα από το άλλο έτσι ώστε μια GPU με περισσότερα cores να μπορεί αυτόματα να εκτελέσει το πρόγραμμα σε λιγότερο χρόνο από ότι μια GPU με λιγότερα cores.



# Πυρήνας (Kernel)

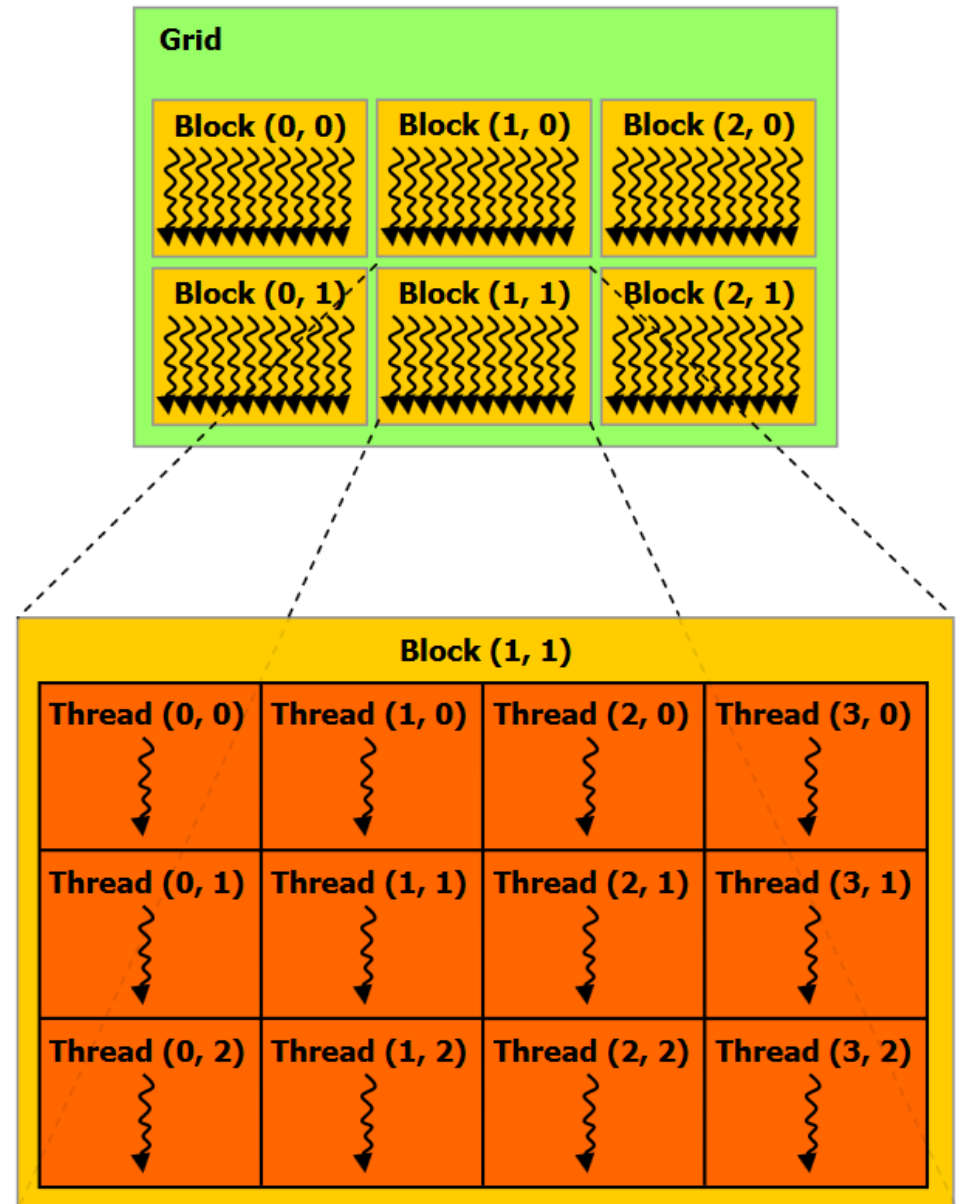
- ▶ Η CUDA C επεκτείνει τη C επιτρέποντας στον προγραμματιστή να ορίσει συναρτήσεις C που ονομάζονται πυρήνες (kernels) και όταν κληθούν εκτελούνται N φορές παράλληλα από N διαφορετικά νήματα CUDA σε αντίθεση με την μοναδική εκτέλεση μιας κανονικής συνάρτησης C.
- ▶ Κάθε νήμα που εκτελεί τον kernel λαμβάνει ένα μοναδικό thread ID που είναι προσβάσιμο από τον πυρήνα χρησιμοποιώντας μια συγκεκριμένη, προκαθορισμένη μεταβλητή

# Ιεραρχία νημάτων CUDA

- ▶ Η ιεραρχία νημάτων CUDA thread αποτελείται από ένα πλέγμα (grid) με thread blocks.
- ▶ Thread block: Ένα thread block είναι ένα σύνολο από παράλληλα εκτελούμενα νήματα που βρίσκονται στο ίδιο SM – μοιράζονται τους πόρους αυτού του SM και συνεργάζονται μεταξύ τους χρησιμοποιώντας διαφορετικούς μηχανισμούς υλικού (hardware mechanisms). Κάθε thread block έχει ένα block ID μέσα στο grid του. Ένα thread block μπορεί να είναι 1D, 2D ή και 3D.
  - Παρέχει έναν φυσικό τρόπο υπολογισμού στοιχείων σε μορφή διανύσματος, πίνακα ή όγκου
- ▶ Grid: Ένα πλέγμα είναι ένα array από thread blocks που εκκινούνται από τον πυρήνα, διαβάζουν είσοδο από από την global memory και γράφουν αποτελέσματα στην global memory με δυνατότητες συγχρονισμού. Ένα grid περιγράφεται από τον χρήστη και μπορεί να είναι 1D, 2D ή και 3D.

# Grid από thread blocks

- ▶ Υπάρχει ένα όριο στον αριθμό των threads σε ένα block, αφού όλα τα threads ενός block πρέπει να βρίσκονται στο ίδιο processor core και πρέπει να μοιράζονται τους περιορισμένους πόρους μνήμης του πυρήνα αυτού.
- ▶ Στις τρέχουσες GPUs, ένα thread block μπορεί να περιέχει μέχρι και 1024 threads.
- ▶ Ένας kernel μπορεί να εκτελεστεί από πολλαπλά, ισομεγέθη thread blocks





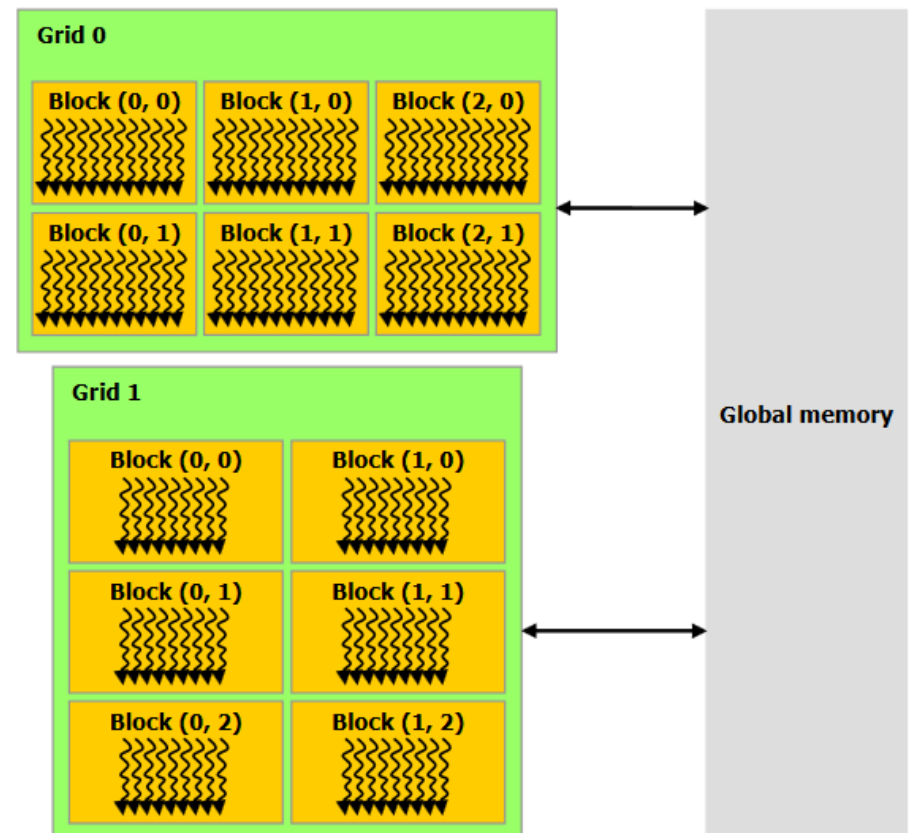
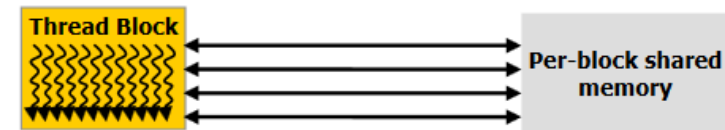
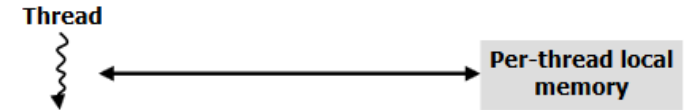
# Thread blocks

- ▶ Όλα τα νήματα ενός block εκτελούν την ίδια συνάρτηση πυρήνα
- ▶ Τα νήματα έχουν δείκτες (indices) εντός του block
  - Ο κώδικας πυρήνα χρησιμοποιεί τους δείκτες νημάτων και block για να επιλέξει τμήμα υπολογισμών και να διευθυνσιοδοτήσει μνήμη
- ▶ Τα νήματα του ίδιου block μπορούν να μοιραστούν δεδομένα και να συγχρονιστούν/συνεργαστούν μεταξύ τους μέσω κοινής μνήμης, ατομικών εντολών και φραγμάτων
- ▶ Τα νήματα διαφορετικών block ΔΕΝ μπορούν να συνεργαστούν
  - Κάθε block μπορεί να εκτελεστεί με οποιαδήποτε σειρά σε σχέση με τα άλλα block



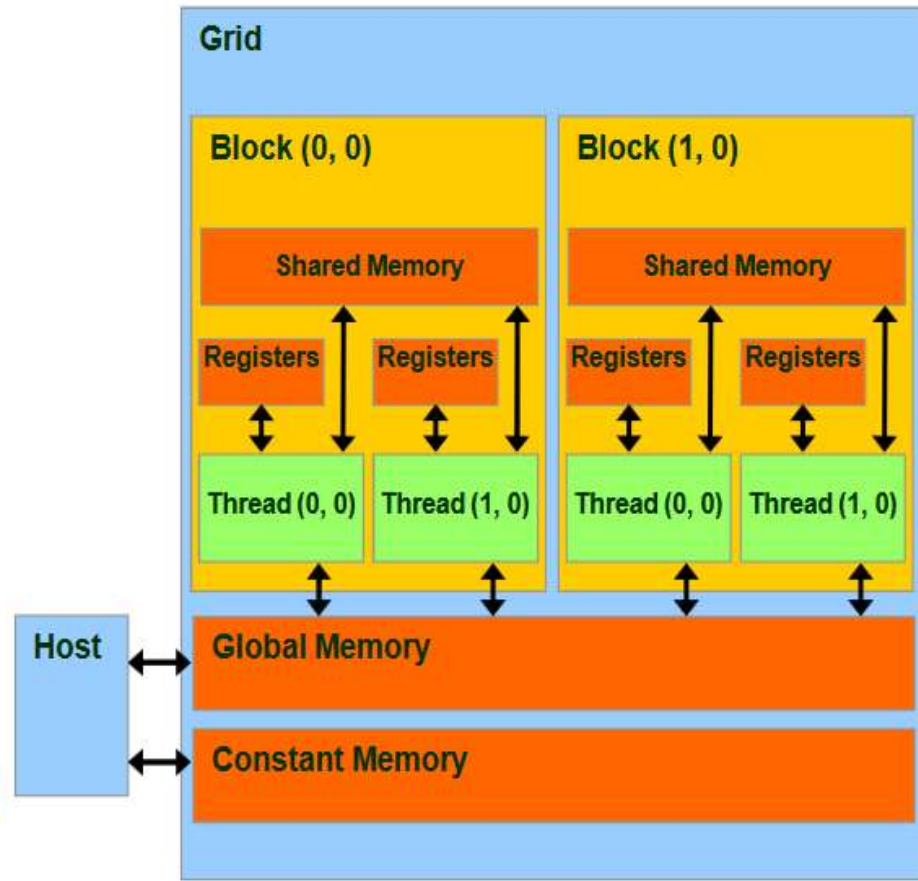
# Ιεραρχία μνήμης

- ▶ Τα CUDA νήματα μπορούν να προσπελάσουν δεδομένα από πολλαπλά memory spaces κατά την διάρκεια της εκτέλεσής τους
- ▶ Κάθε νήμα έχει private local memory
- ▶ Κάθε thread block έχει shared memory που είναι ορατή σε όλα τα νήματα αυτού του block και έχει το ίδιο χρόνο ζωής με το ίδιο το block.
- ▶ Όλα τα νήματα έχουν πρόσβαση στην ίδια global memory.
- ▶ Υπάρχουν επίσης δύο επιπρόσθετα read-only memory spaces που είναι προσβάσιμα από όλα τα νήματα:
  - constant memory space
  - texture memory space
- ▶ Τα global, constant και texture memory spaces παραμένουν σε διαφορετικές εκκινήσεις πυρήνα (kernel launches) από την ίδια εφαρμογή.



# Η ιεραρχία της μνήμης από την σκοπιά του προγραμματιστή

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



# Μοντέλο εκτέλεσης

- ▶ Τα νήματα στην CUDA εκτελούνται σε μια φυσικά ξεχωριστή συσκευή που λειτουργεί σαν συνεπεξεργαστής προς τον host που εκτελεί το πρόγραμμα C.
- ▶ Το προγραμματιστικό μοντέλο της CUDA θεωρεί ότι τόσο ο host όσο και η συσκευή έχουν δικές τους, ξεχωριστές μνήμες: host memory και device memory αντίστοιχα.
- ▶ Ένα πρόγραμμα διαχειρίζεται τα global, constant και texture memory spaces που είναι ορατά στους kernels μέσω κλήσεων προς την CUDA runtime. Αυτό περιλαμβάνει εκχωρήσεις (allocations) σε device memory καθώς και μεταφορά δεδομένων μεταξύ host και device memory

## C Program Sequential Execution

Serial code

Parallel kernel  
Kernel0<<<>>>()

Serial code

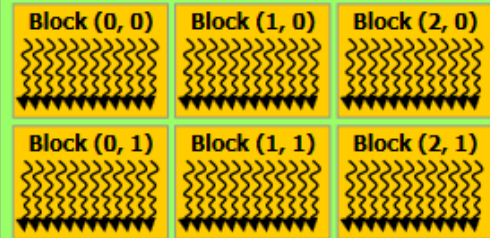
Parallel kernel  
Kernel1<<<>>>()

Host



Device

Grid 0

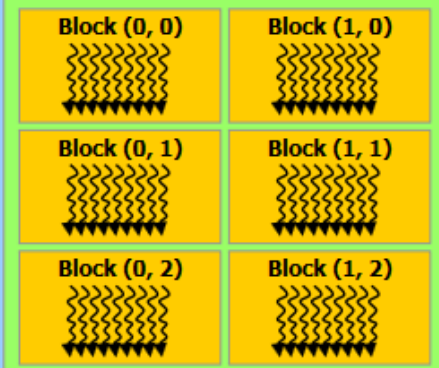


Host



Device

Grid 1



# Ενσωματωμένες μεταβλητές CUDA (built-in variables)

- ▶ `blockIdx.x`, `blockIdx.y`, `blockIdx.z`
  - Ενσωματωμένες (built-in) μεταβλητές που επιστρέφουν το block ID στον άξονα x, y και z του block που εκτελεί το συγκεκριμένο κομμάτι κώδικα
- ▶ `threadIdx.x`, `threadIdx.y`, `threadIdx.z`
  - Ενσωματωμένες συναρτήσεις που επιστρέφουν το thread ID στον άξονα x, y και z του νήματος που εκτελείται από τον SM (stream processor) σε αυτό το συγκεκριμένο block.
- ▶ `blockDim.x`, `blockDim.y`, `blockDim.z`
  - Ενσωματωμένες συναρτήσεις που επιστρέφουν την αντίστοιχη διάσταση του block (δηλαδή τον αριθμό των νημάτων ενός block στον κάθε άξονα)

# Υπολογισμός του Global Thread ID

- ▶ Έστω μονοδιάστατο grid και μονοδιάστατο block
- ▶ 4 block στο grid και 8 νήματα ανά block
  - $\text{gridDim.x} = 4 * 1$   $\text{blockDim.x} = 8 * 1$
  - Global Thread ID:
    - $\text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x} = 3 * 8 + 2 = 26$

## Global Thread ID

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
threadIdx.x								threadIdx.x								threadIdx.x								threadIdx.x							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
blockIdx.x = 0								blockIdx.x = 1								blockIdx.x = 2								blockIdx.x = 3							

# Παράδειγμα: hello.cu

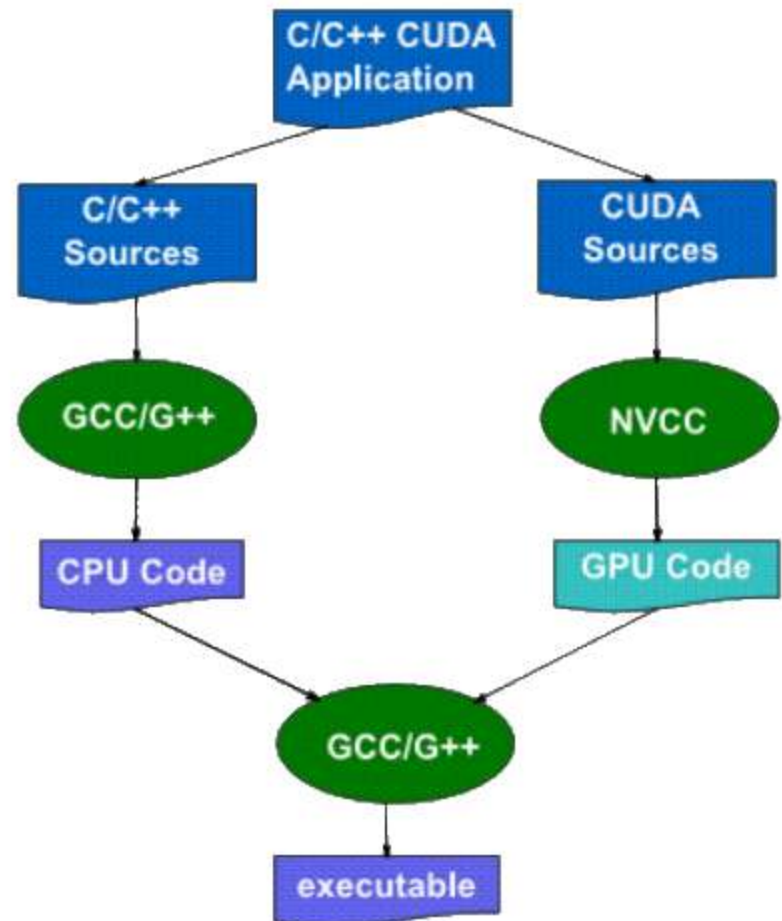
```
int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```

- ▶ Standard C που τρέχει στον host
- ▶ Ο NVidia compiler μπορεί να χρησιμοποιηθεί για να μεταγλωττίσει προγράμματα χωρίς device κώδικα



# Διαδικασία παραγωγής εκτελέσιμου αρχείου

- ▶ `nvcc -o hello hello.cu`
- ▶ `./hello`
  - Hello, world





# Παράδειγμα: hello2.cu

- ▶ Το CUDA C/C++ keyword `__global__` υποδηλώνει μια συνάρτηση που:
  - Τρέχει στην device
  - Καλείται από τον κώδικα του host
- ▶ Το `nvcc` διαχωρίζει τον πηγαίο κώδικα σε δύο συστατικά (components):
  - Host
  - Device
- ▶ Οι device συναρτήσεις (πχ. `mykernel()`) επεξεργάζονται από τον NVIDIA compiler
- ▶ Οι host συναρτήσεις (πχ. `main()`) επεξεργάζονται από τον standard host compiler (`gcc`)
- ▶ Τα `<<< >>>` δείχνουν μια κλήση από host σε device
  - Ονομάζεται και “kernel launch”

```
#include <stdio.h>

__global__ void mykernel(void)
{
}

int main(void)
{
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

```
$ nvcc -o hello2 hello2.cu
$ ./hello2
Hello World!
```

# Kernel (κλήση της mykernel)

- ▶ `mykernel<<< N, T >>>();`
  - N blocks
  - T threads
- ▶ `mykernel<<< 1, 1 >>>();`
  - 1 block
  - 1 thread
- ▶ Η κλήση σε μια συνάρτηση πυρήνα είναι ασύγχρονη (από την CUDA 1.0 και μετά)

# Διαχείριση μνήμης [1]

- ▶ Οι device pointers δείχνουν σε GPU memory
  - Μπορούν να περαστούν από/προς host code
  - Δεν μπορούν να γίνουν dereference σε host code
- ▶ Οι host pointers δείχνουν σε CPU memory
  - Μπορούν να περαστούν από/προς device code
  - Δεν μπορούν να γίνουν dereference σε device code
- ▶ Συναρτήσεις
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - `malloc()`, `free()`, `memcpy()`

# Διαχείριση μνήμης [2]

## ▶ cudaMalloc()

- Δεσμεύει αντικείμενο στην καθολική μνήμη του device

- Δύο παράμετροι

- `void ** devPtr`

Δείκτης προς το δεσμευμένο αντικείμενο (στην device memory)

- `size_t size`

Μέγεθος αντικειμένου σε bytes

## ▶ cudaFree()

- Ελευθερώνει αντικείμενο από την καθολική μνήμη του device

- Μία παράμετρος

- `void * devPtr`

Δείκτης προς αντικείμενο

# Διαχείριση μνήμης [3]

- ▶ `cudaMemcpy()`
  - Μεταφορά δεδομένων
  - 4 παράμετροι
    - `void * dst` Προορισμός (Δείκτης στο αντικείμενο-προορισμός)
    - `const void * src` Πηγή (Δείκτης στο αντικείμενο-πηγή)
    - `size_t count` Μέγεθος προς αντιγραφή (πλήθος σε bytes)
    - `enum cudaMemcpyKind kind` Κατεύθυνση μεταφοράς
      - `cudaMemcpyHostToHost`
      - `cudaMemcpyHostToDevice`
      - `cudaMemcpyDeviceToHost`
      - `cudaMemcpyDeviceToDevice`
  - Η μεταφορά προς το device είναι ασύγχρονη
    - Το πρόγραμμα στον host συνεχίζει άμεσα την εκτέλεση του

# Άσκηση: incint.cu

## ▶ Πρόγραμμα CUDA

- Ο host θα ορίζει έναν ακέραιο  $a$  και θα τον αρχικοποιεί
- Η device θα λαμβάνει τον ακέραιο, θα τον προσυζάνει κατά 1 και θα τον επιστρέφει στον host

## ▶ Σημειώσεις

- Χρήση `cudaMalloc` για δέσμευση μνήμης, `cudaMemcpy` για αντιγραφή μνήμης από και προς την device και `cudaFree` για απελευθέρωση της μνήμης
- Το πρόγραμμα θα είναι πρακτικά σειριακό

# Δήλωση συναρτήσεων

- ▶ Η `__global__` ορίζει μια συνάρτηση πυρήνα
  - Κάθε «`__`» αποτελείται από δύο underscore
  - Μια συνάρτηση πυρήνα πρέπει να είναι τύπου `void`
- ▶ Υπάρχουν επίσης οι συναρτήσεις `__device__` και `__host__`

	Executed on the:	Only callable from the:
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host



# Άσκηση: sdc.cu

- ▶ Υπολογισμός αθροίσματος  $3 + 4$  χρησιμοποιώντας και τη συνάρτηση `addem` στην `device` που καλείται από την `add`

- `$ nvcc -o sdc sdc.cu`
- `$ ./sdc`
- $3 + 4 = 7$

```
#include <stdio.h>

__device__ int addem(...) {
    ...
}

__global__ void add(...) {
    // Κλήση της addem(...);
}

int main() {
    int c;
    int *dev_c;
    cudaMalloc(...);
    add ...
    cudaMemcpy(...);
    printf("3 + 4 = %d\n", c);
    cudaFree(...);
    return 0;
}
```

# Άσκηση: addint.cu

- ▶ Πρόσθεση δύο αριθμών (a και b) στην device
  - Πχ. `int a, b;`
  - `a=2; b=7;`
- ▶ Ο host θα καλεί την `add` (kernel launch) η οποία θα εκτελεστεί στην device
- ▶ Το αποτέλεσμα (c) θα επιστρέφεται στον host

# Άσκηση: addint.cu (υπόδειξη)

- ▶ Πρόσθεση δύο αριθμών στην device

```
__global__ void add(int *a, int *b, int *c)
{
    *c = *a + *b;
}
```

- ▶ Η συνάρτηση πυρήνα add() θα κληθεί από τον host και θα εκτελεστεί στην device
- ▶ Συνεπώς οι μεταβλητές (δείκτες) a, b, c θα πρέπει να δείχνουν στη μνήμη της device

# Άσκηση: addint3.cu

## ▶ Να γραφτεί πρόγραμμα CUDA που:

- Ο host θα ζητάει από το χρήστη
  - Τον αριθμό των στοιχείων του πίνακα A
  - Τις (ακέραιες) τιμές των στοιχείων του πίνακα A
  - Έναν ακέραιο αριθμό B
- Η device θα υπολογίζει (σειριακά) το άθροισμα κάθε στοιχείου του A με τον αριθμό B, αποθηκεύοντας τη νέα τιμή του στοιχείου στον πίνακα
- Ο host θα εκτυπώνει στην οθόνη τον πίνακα A με τις νέες του τιμές

```
• $ nvcc -o addint3 addint3.cu
• $ ./addint3
Size of array a:5
Element 0=5
Element 1=10
Element 2=15
Element 3=20
Element 4=25
Give integer b:3
• A[0]=8
• A[1]=13
• A[2]=18
• A[3]=23
• A[4]=28
```

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

```
int main(int argc, char *argv[])
{
```

```
    int N, i;
    int *input_h, *output_h;
    int *vector_d;
```

```
    if (argc != 2) {
        printf("Usage: ./arg <Size of vectors upto 1024>\n");
        exit(1);
    }
```

```
    N = atoi(argv[1]);
    if (N < 1 || N > 1024) exit(1);
```

```
    input_h = (int *)malloc(N * sizeof(int));
    output_h = (int *)malloc(N * sizeof(int));
    for (i = 0; i < N; i++) {
        input_h[i] = 1;
        output_h[i] = 0;
    }
```

```
    cudaMalloc(&vector_d, N * sizeof(int));
    cudaMemcpy(vector_d, input_h, N * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(output_h, vector_d, N * sizeof(int), cudaMemcpyDeviceToHost);
```

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

```
}
```

## Παράδειγμα arg.cu

- ▶ \$ nvcc -o arg arg.cu
- ▶ \$ ./arg

Usage: ./arg <Size of  
vectors upto 1024>

- ▶ ./arg 1024

output\_h[ 0] = 1

output\_h[511] = 1

output\_h[1023] = 1

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

# Παράδειγμα: err.cu

```
int main(int argc, char *argv[])
{
```

```
    int N, i;
    int *input_h, *output_h;
    int *vector_d;
    cudaError_t err;
    if (argc != 2) {
        printf("Usage: ./err <Size of vectors upto 1024>\n");
        exit(1);
    }
```

```
    N = atoi(argv[1]);
```

```
    if (N < 1 || N > 1024) { printf("Error: Size of vectors between 1 and 1024!\n"); exit(1); }
```

```
    input_h = (int *)malloc(N * sizeof(int));
```

```
    if (input_h == NULL) { printf("Could not allocate memory for input vector on host.\n"); exit(1); }
```

```
    output_h = (int *)malloc(N * sizeof(int));
```

```
    if (output_h == NULL) { printf("Could not allocate memory for output vector on host.\n"); exit(1); }
```

```
    for (i = 0; i < N; i++) {
```

```
        input_h[i] = 1;
```

```
        output_h[i] = 0;
```

```
    }
```

```
    err = cudaMalloc(&vector_d, N * sizeof(int));
```

```
    if (err != cudaSuccess) { printf("Could not allocate memory for vector on the device.\n"); exit(1); }
```

```
    err = cudaMemcpy(vector_d, input_h, N * sizeof(int), cudaMemcpyHostToDevice);
```

```
    if (err != cudaSuccess) { printf("Could not copy input vector to device.\n"); exit(1); }
```

```
    err = cudaMemcpy(output_h, vector_d, N * sizeof(int), cudaMemcpyDeviceToHost);
```

```
    if (err != cudaSuccess) { printf("Could not copy vector from device to output vector on host.\n"); exit(1); }
```

```
    printf("output_h[%4d] = %d\n", 0, output_h[0]);
```

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

```
    printf("output_h[%4d] = %d\n", N - 1, output_h[N - 1]);
```

```
}
```

```
▶ $ nvcc -o err err.cu
```

```
▶ $ ./err
```

```
Usage: ./err <Size of vectors upto 1024>
```

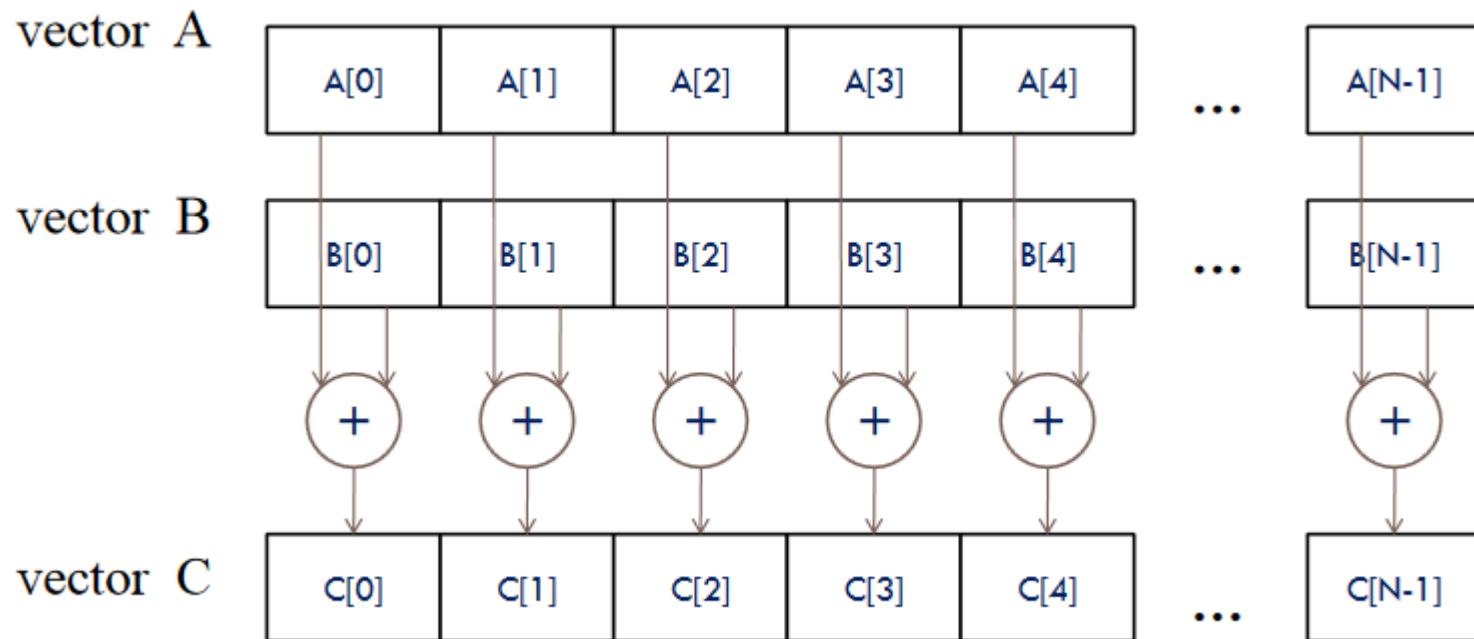
```
▶ ./err 1024
```

```
output_h[ 0] = 1
```

```
output_h[511] = 1
```

```
output_h[1023] = 1
```

# Πρόσθεση διανυσμάτων





# Άσκηση: alcpu.cu

```
• $ ./alcpu
0 + 0 = 0
-1 + 1 = 0
-2 + 4 = 2
-3 + 9 = 6
-4 + 16 = 12
-5 + 25 = 20
-6 + 36 = 30
-7 + 49 = 42
-8 + 64 = 56
-9 + 81 = 72
```

- ▶ Γράψτε ένα σειριακό πρόγραμμα που
- ▶ Θα ορίζει και θα αρχικοποιεί τρία διανύσματα  $a$ ,  $b$  και  $c$  μήκους  $N$ 
  - Σημείωση:  $a[i] = -i$ ,  $b[i] = i * i$  και  $c[i] = 0$
- ▶ Θα καλεί την συνάρτηση `add` η οποία θα προσθέτει τα στοιχεία του διανύσματος  $a$  και του διανύσματος  $b$  στο διάνυσμα  $c$ :
  - $c[i] = a[i] + b[i]$  όπου  $i = 0, 1, \dots, N-1, N$
- ▶ Ακολουθώς θα εκτυπώνει στην οθόνη όλα τα στοιχεία όλων των διανυσμάτων

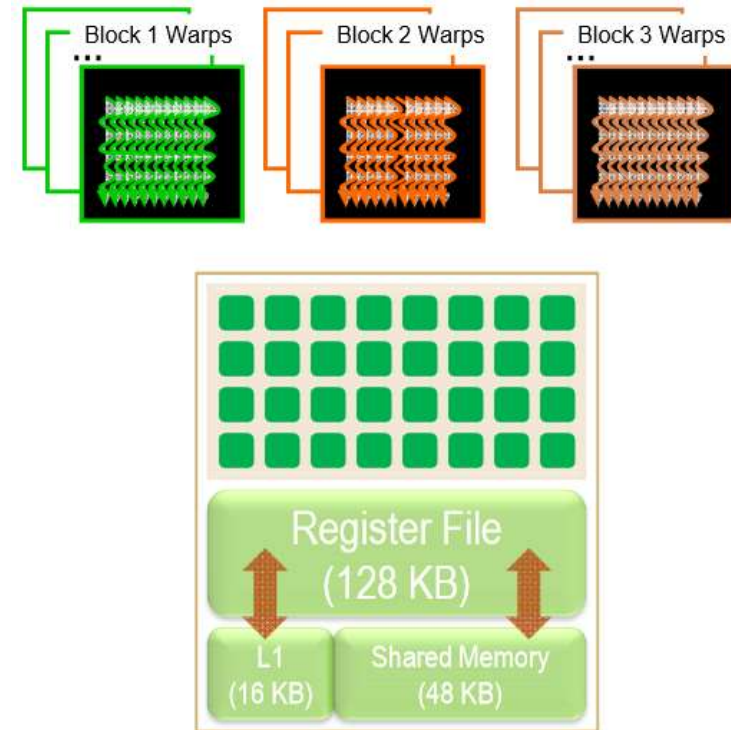
# Άσκηση: algpu.cu

- ▶ Μετατρέψτε το σειριακό πρόγραμμα alcpu.cu σε παράλληλο πρόγραμμα CUDA
- ▶ Προσοχή:
  - Ορισμός της συνάρτησης πυρήνα add με τις κατάλληλες παραμέτρους ανάμεσα στα <<< >>> ώστε να επιμεριστεί ορθά η διαδικασία αντί να γίνει η δουλειά σειριακά, όπως στο alcpu.cu
  - Σκεφτείτε καλά πως μπορεί να επιτευχθεί αυτό!!!

```
• $ ./algpu
0 + 0 = 0
-1 + 1 = 0
-2 + 4 = 2
-3 + 9 = 6
-4 + 16 = 12
-5 + 25 = 20
-6 + 36 = 30
-7 + 49 = 42
-8 + 64 = 56
-9 + 81 = 72
```

# Χρονοπρογραμματισμός νημάτων

- ▶ Τα νήματα κάθε block εκτελούνται ανά 32 σε warp
  - Απόφαση υλοποίησης στο υλικό, όχι μέρος του προγραμματιστικού μοντέλου CUDA
  - Τα warp είναι οι μονάδες χρονοπρογραμματισμού σε κάθε SM
- ▶ Αν σε ένα SM έχουν ανατεθεί 3 block και κάθε block έχει 256 νήματα, πόσα warp υπάρχουν στο SM;
  - Κάθε block αποτελείται από  $256/32 = 8$  warp
  - Συνολικά  $8 * 3 = 24$  warp



# Χρονοπρογραμματισμός νήματων

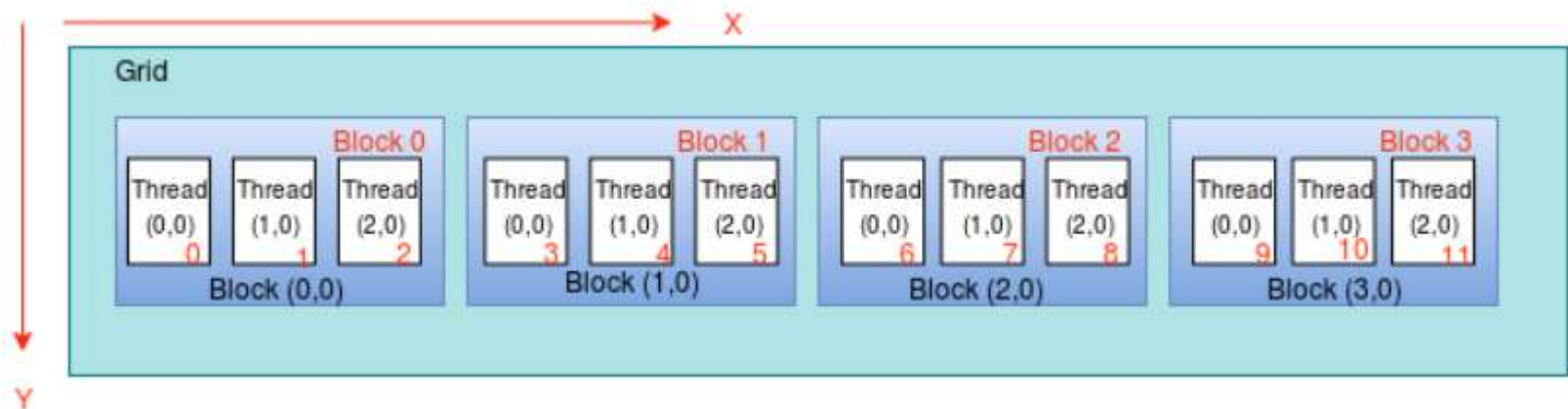
- ▶ Τα SM υλοποιούν χρονοπρογραμματισμό των warp με μηδενική επιβάρυνση
  - Κάθε χρονική στιγμή 1 ή 2 warp εκτελούνται από ένα SM
  - Τα warp των οποίων η επόμενη προς εκτέλεση εντολή έχει τα δεδομένα της έτοιμα προς χρήση μπορεί να επιλεγεί προς εκτέλεση
  - Τα έτοιμα προς εκτέλεση warp επιλέγονται για εκτέλεση με μια πολιτική χρονοδρομολόγησης βασισμένη σε προτεραιότητες
  - Όλα τα νήματα ενός warp που επιλέχθηκε προς εκτέλεση, εκτελούν τις ίδιες εντολές

# Τύπος δεδομένων dim3

- ▶ `dim3 DimGrid(256, 1, 1);`
- ▶ `dim3 DimBlock(16, 1, 1);`
- ▶ `Kernel<<<DimGrid,DimBlock>>>(...);`
- ▶ Ο τύπος `dim3` είναι ένα integer vector βασισμένος στο `uint3` που χρησιμοποιείται για τον ορισμό διαστάσεων
  - Έχει τρία μέλη (components): `.x .y .z`
- ▶ Ορίζοντας μια μεταβλητή τύπου `dim3`, κάθε μέλος που δεν έχει δηλωθεί αρχικοποιείται με την τιμή 1.

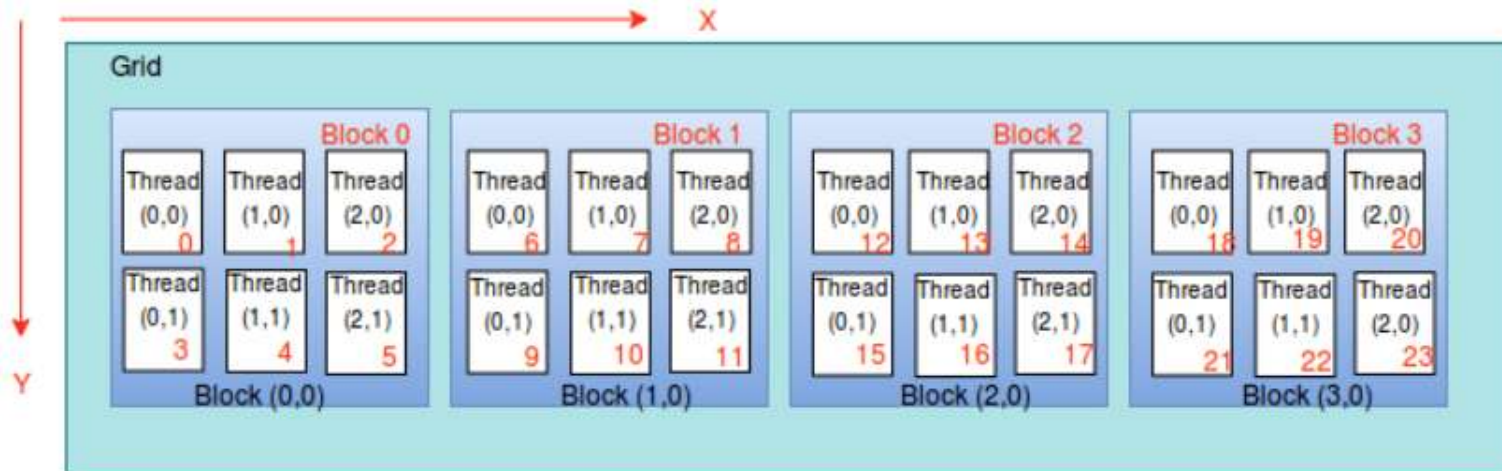
# 1D Grid από 1D Blocks

- ▶  $threadId = (blockIdx.x * blockDim.x) + threadIdx.x$



# 1D Grid από 2D Blocks

- ▶  $threadId = (blockIdx.x * blockDim.x * blockDim.y) + (threadIdx.y * blockDim.x) + threadIdx.x$



- ▶  $threadId = (gridDim.x * blockDim.x * threadIdx.y) + (blockDim.x * blockIdx.x) + threadIdx.x$

