Προηγμένες αρχιτεκτονικές υπολογιστών και προγραμματισμός παραλλήλων συστημάτων

LAB-11. CUDA

Κάρτες γραφικών και παραλληλισμός

- GPU: Graphics Processing Unit
- Οι σημερινές κάρτες γραφικών διαθέτουν μεγάλο βαθμό παραλληλισμού.
  - Χιλιάδες GPUs
  - GBs μνήμης (Video RAM) υψηλών επιδόσεων.
- Οι σημερινές εφαρμογές video και τα παιχνίδια έχουν μεγάλες απαιτήσεις.

Κάρτες γραφικών

- Υψηλού κόστους: NVIDIA RTX 3090
  - Υπολογιστικοί πυρήνες (CUDA Cores): 10496 @ 1.7GHz
  - Μνήμη: 24GB GDDR 6X
  - Τιμή: 2000-2500€
- Μέσου κόστους: NVIDIA RTX 3060Ti
  - Υπολογιστικοί πυρήνες (CUDA Cores): 4864 @ 1.67GHz
  - Μνήμη: 8GB GDDR 6
  - Τιμή: 600-1000€
- Χαμηλού κόστους NVIDIA GTX 1050Ti
  - Υπολογιστικοί πυρήνες (CUDA Cores): 768 @ 1.4GHz
  - Μνήμη: 4GB GDDR 5
  - Τιμή: < 200€

CUDA

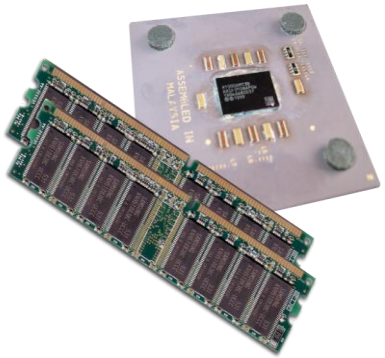
- **Ιδέα**: χρήση αυτής της μεγάλης (και φθηνής) υπολογιστικής ισχύος όχι μόνο για ανάγκες γραφικών, αλλά και για εφαρμογές γενικού σκοπού.
- **CUDA**: Βιβλιοθήκη – Σύστημα παραλληλισμού (parallelization framework) της NVIDIA για χρήση των GPUs των καρτών γραφικών σε γενικής φύσεως προβλήματα.

CUDA C/C++

- Αρχιτεκτονική:
  - Χρήση παραλληλισμού επεξεργαστών καρτών γραφικών (Graphics Processing Unit, GPU) για υπολογισμούς γενικού σκοπού.
  - Αύξηση επιδόσεων.
- CUDA C/C++:
  - Γλώσσα βασισμένη στην κλασική C/C++.
  - Εισάγει ένα σύνολο επεκτάσεων για την υποστήριξη ετερογενούς (CPU/GPU) προγραμματισμού.
  - API (Application Programming Interface) για τη διαχείριση της συσκευής, της μνήμης, κλπ.

Heterogeneous Computing

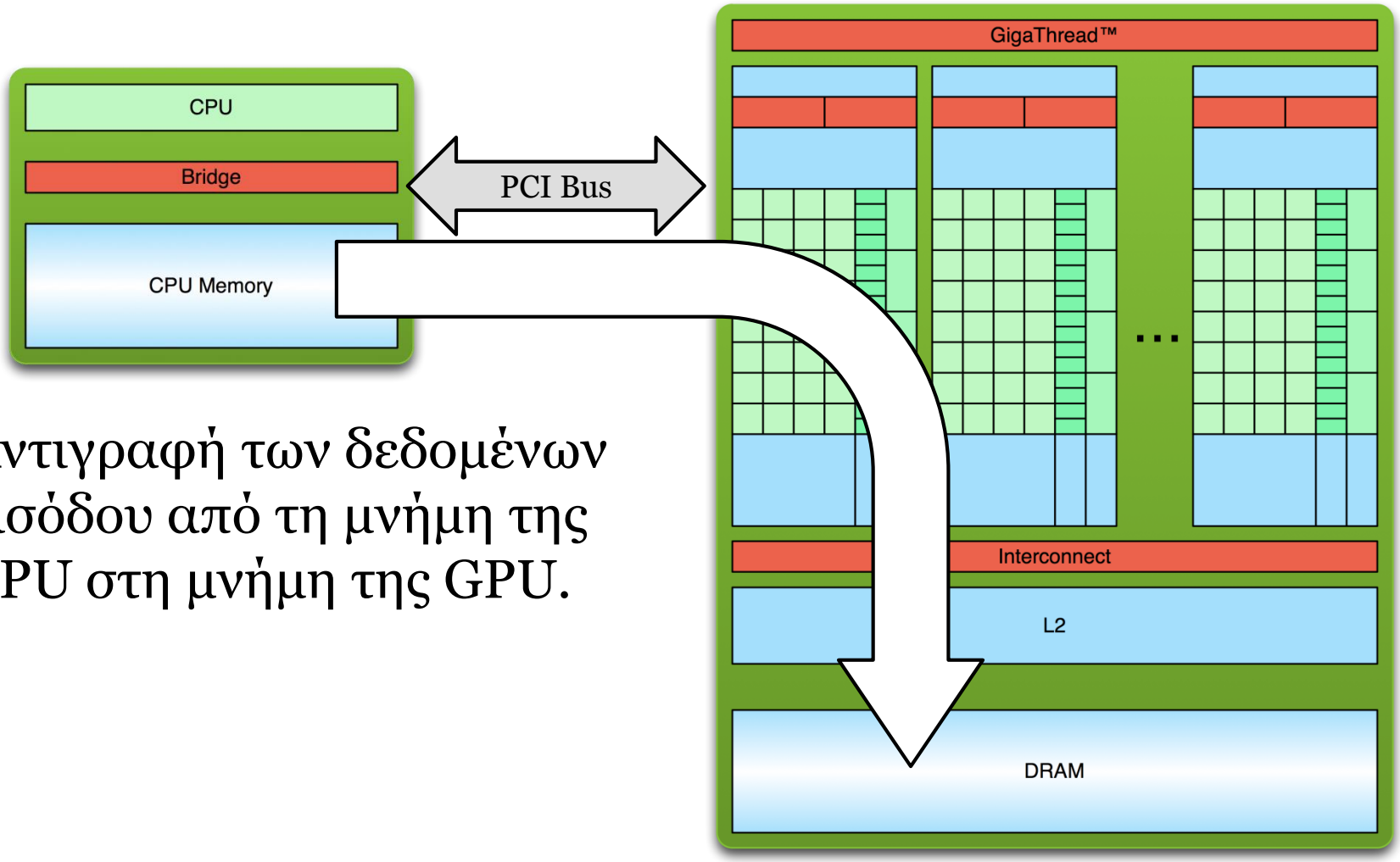
- Ετερογενές computing:
- **Host**: Η/Οι CPU και η μνήμη του υπολογιστή (host memory).
- **Device**: Η/Οι GPU και η μνήμη της/των κάρτας/ων γραφικών (device memory).



Host

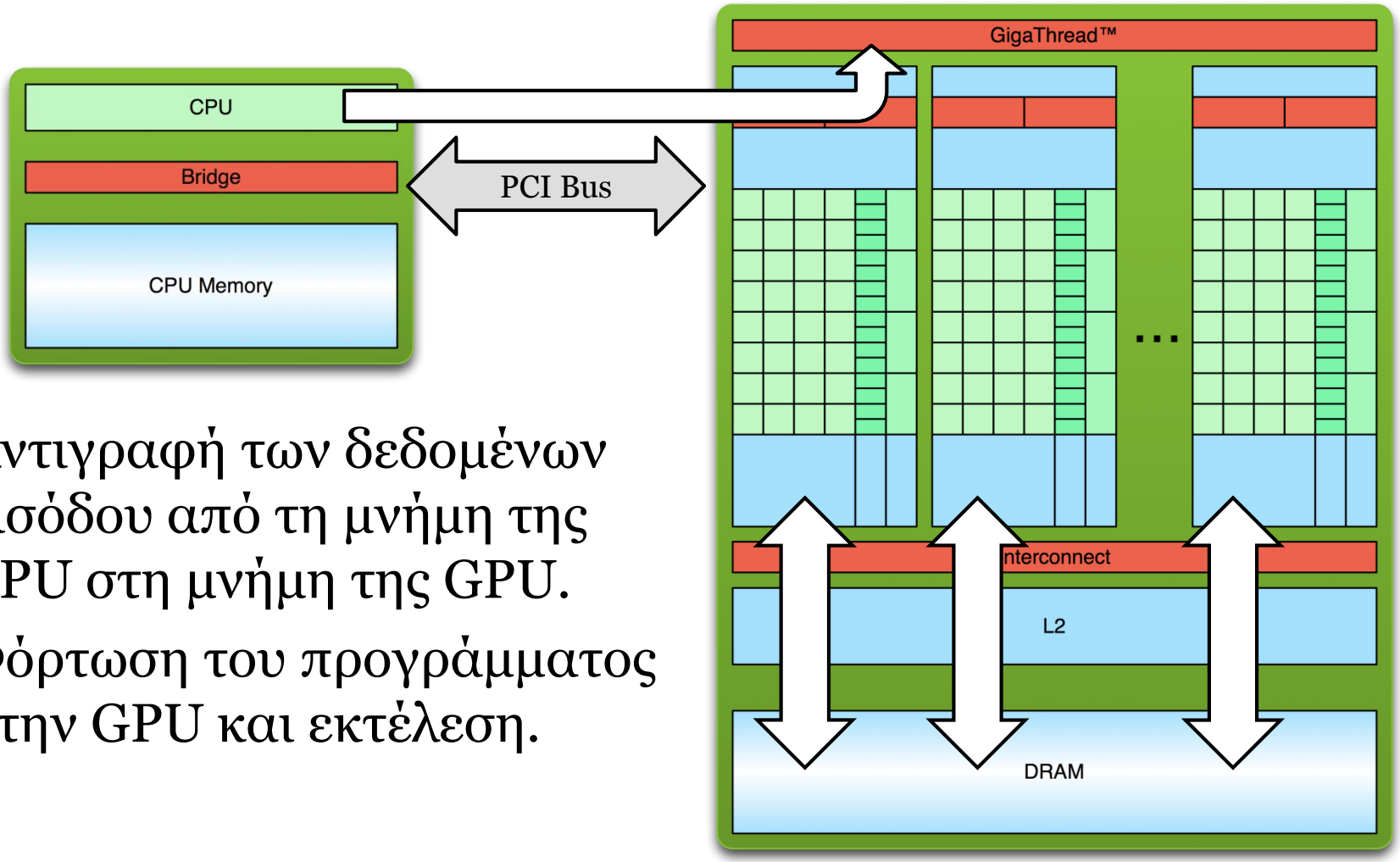
Device

Ροή επεξεργασίας (1)



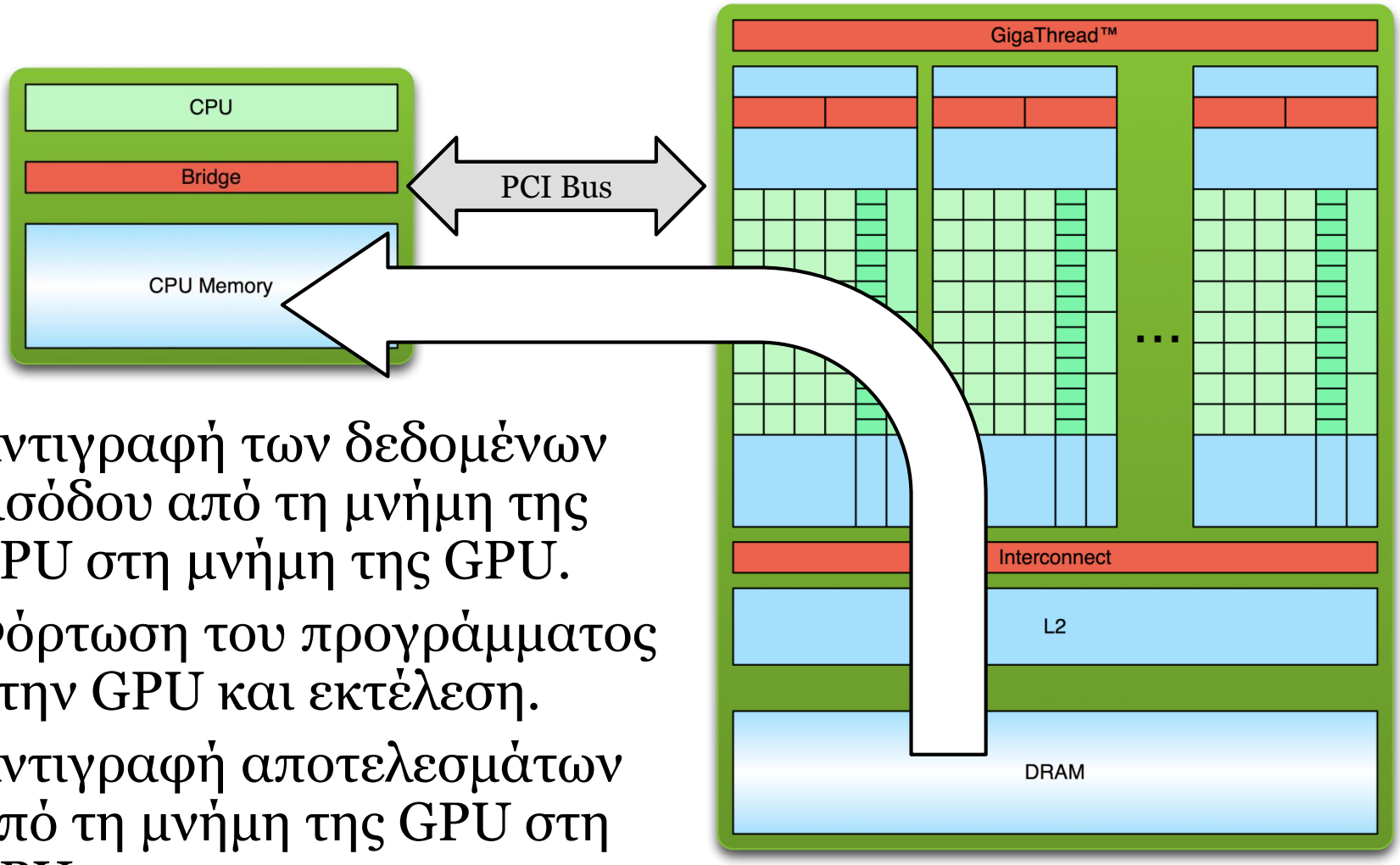
1. Αντιγραφή των δεδομένων εισόδου από τη μνήμη της CPU στη μνήμη της GPU.

Ροή επεξεργασίας (2)



1. Αντιγραφή των δεδομένων εισόδου από τη μνήμη της CPU στη μνήμη της GPU.
2. Φόρτωση του προγράμματος στην GPU και εκτέλεση.

Ροή επεξεργασίας (3)



1. Αντιγραφή των δεδομένων εισόδου από τη μνήμη της CPU στη μνήμη της GPU.
2. Φόρτωση του προγράμματος στην GPU και εκτέλεση.
3. Αντιγραφή αποτελεσμάτων από τη μνήμη της GPU στη CPU.

Hello World (host only)

- Κλασική C που εκτελείται στον host.

```
int main (int argc, char ** argv) {
        printf("Hello World!\n");
        return 0;
}
```

- Ο compiler της NVIDIA (nvcc) μπορεί να χρησιμοποιηθεί για να μεταγλωττίσει προγράμματα C/C++ ακόμα και αυτά δεν περιέχουν καθόλου device κώδικα.

Hello World με κώδικα device

```
__global__ void mykernel (void) { }

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

- Η δήλωση **`__global__`** υποδεικνύει συνάρτηση που:
  - εκτελείται στην κάρτα γραφικών.
  - καλείται από τον οικοδεσπότη.

nvcc, κώδικας host & device

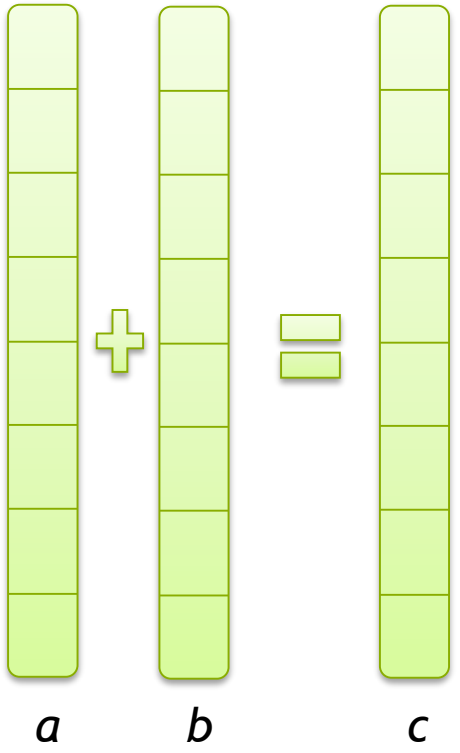
- **Ο NVIDIA compiler (nvcc) διαχωρίζει τον πηγαίο κώδικα σε κώδικα οικοδεσπότη (host code) και κώδικα συσκευής (device code).**
- Ο κώδικας device είναι αυτός που:
  - διαχειρίζεται ο nvcc (π.χ. `mykernel`).
  - εκτελείται παράλληλα στην GPU.
  - Μνήμη device = Μνήμη GPU
- Τον κώδικα host (π.χ. συνάρτηση `main`) τον διαχειρίζεται ο βασικός compiler του οικοδεσπότη (host compiler, π.χ. GCC, VC++, …).
  - Μνήμη host = Μνήμη CPU = κύρια μνήμη συστήματος

Τριπλά angle brackets

- `mykernel<<<1,1>>>();`
- Τα τριπλά angle brackets υποδεικνύουν κλήση από τον κώδικα host προς τον κώδικα device.
  - Γνωστά και ως “kernel launch”.
  - Οι παράμετροι `1,1` αφορούν στον τρόπο παραλληλισμού. Θα τους εξετάσουμε παρακάτω.

Παράλληλος προγραμματισμός CUDA

- Παράδειγμα: πρόσθεση ακεραίων.
- Αναγωγή της απλής πρόσθεσης ακεραίων σε παράλληλο υπολογισμό πολλαπλών αθροισμάτων μεταξύ δύο ακεραίων.
- Εφαρμογή: άθροισμα δύο διανυσμάτων.



Πρόσθεση αριθμών (device)

- Μία απλή συνάρτηση για την πρόσθεση δύο ακεραίων στη συσκευή.
- Όπως και προηγουμένως με την `mykernel`, ο ορισμός `__global__` υποδηλώνει ότι η `add`:
  - Θα εκτελεστεί στη συσκευή.
  - Θα κληθεί από τον host.

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

Πρόσθεση αριθμών (device)

- Παρατηρήστε τη χρήση των pointers για τα ορίσματα της add.
- Η add εκτελείται στη συσκευή, συνεπώς τα a, b και c πρέπει να δείχνουν στη μνήμη της συσκευής.
- **Πρέπει να δεσμεύσουμε μνήμη στη GPU.**

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

Host vs. Device pointers

- Οι μνήμες host και device είναι διαφορετικές οντότητες. Άρα:
  - Οι device pointers δείχνουν σε περιοχές της μνήμης GPU.
    - Μπορούν να ανταλλάσσονται μεταξύ host και device.
    - **Απαγορεύεται** στον host code να τους κάνει dereference. **Δηλαδή ο host δεν μπορεί να διαβάσει** την περιοχή της μνήμης GPU στην οποία δείχνει ένας device pointer.
  - Οι host pointers δείχνουν σε περιοχές της μνήμης CPU.
    - Μπορούν να ανταλλάσσονται μεταξύ host και device.
    - **Απαγορεύεται** στον device code να τους κάνει dereference. **Δηλαδή η συσκευή δεν μπορεί να διαβάσει** την περιοχή της κύριας μνήμης στην οποία δείχνει ένας host pointer.

CUDA Memory Management API

- Το μοντέλο CUDA παρέχει ένα απλό Application Programming Interface (API) για τη διαχείριση της μνήμης GPU.

Λειτουργία	CPU (C)	GPU (CUDA C)
Δέσμευση μνήμης	`malloc()`	`cudaMalloc()`
Αποδέσμευση μνήμης	`free()`	`cudaFree()`
Αντιγραφή μιας περιοχής μνήμης σε μία άλλη	`memcpy()`	`cudaMemcpy()`

Πρόσθεση αριθμών (device)

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

int main (int argc, char ** argv) {
        int a, b, c;                        // Αντίγραφα οικοδεσπότη (host) a, b, c
        int *d_a, *d_b, *d_c;               // Αντίγραφα συσκευής (device) a, b, c
        int size = sizeof(int);

        cudaMalloc((void **)&d_a, size);    // Δέσμευση μνήμης για το device a
        cudaMalloc((void **)&d_b, size);    // Δέσμευση μνήμης για το device b
        cudaMalloc((void **)&d_c, size);    // Δέσμευση μνήμης για το device c

        a = 2; b = 7;                       // Αρχικοποίηση τιμών host
        cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

        add<<<1,1>>>(d_a, d_b, d_c);        // Εκτέλεση της add() στην GPU
        cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);// Αντιγραφή αποτελέσματος
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);   // Αποδέσμευση μνήμης
        return 0;
 }
```

Παράλληλη εκτέλεση

- Πώς εκτελείται το παραπάνω παράδειγμα παράλληλα;

```
add<<< 1, 1 >>>();

add<<< N, 1 >>>();
```

- Αντί να εκτελέσουμε την add() μία φορά, την εκτελούμε N φορές παράλληλα.

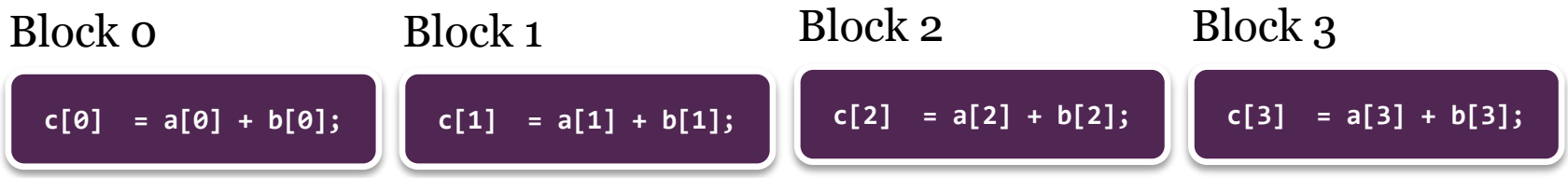
Πρόσθεση διανυσμάτων (blocks)

- Παράλληλες εκτελέσεις της `add()` μπορούν να εφαρμοστούν σε παράλληλους υπολογισμούς αθροισμάτων διανυσμάτων.
- **Ορολογία: κάθε παράλληλη κλήση της `add()` ονομάζεται block.**
- **Ένα σύνολο από blocks λέγεται grid.**
- Κάθε κλήση μπορεί να αναφέρεται στο αντίστοιχο block της (έστω `x`) μέσω της `blockIdx.x`.
- Κάνοντας χρήση της έκφρασης `blockIdx.x` για δεικτοδότηση ενός πίνακα, κάθε block χειρίζεται διαφορετικό δείκτη.

Πρόσθεση διανυσμάτων (blocks)

```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x]+b[blockIdx.x];
}
```

- Μέσα στη συσκευή, κάθε block μπορεί να εκτελείται παράλληλα.



- Πώς τροποποιείται η `main();`

Πρόσθεση διανυσμάτων (blocks)

```
#define N 512
int main (int argc, char ** argv) {
        int *a, *b, *c;                         // Αντίγραφα οικοδεσπότη (host) a, b, c
        int *d_a, *d_b, *d_c;                   // Αντίγραφα συσκευής (device) a, b, c
        int size = N * sizeof(int);

        cudaMalloc((void **)&d_a, size);        // Δέσμευση μνήμης για το device a
        cudaMalloc((void **)&d_b, size);        // Δέσμευση μνήμης για το device b
        cudaMalloc((void **)&d_c, size);        // Δέσμευση μνήμης για το device c

        // Δέσμευση μνήμης για τα αντίγραφα οικοδεσπότη a, b, c. Αρχικοποίηση με τυχαίους.
        a = (int *)malloc(size); random_ints(a, N);
        b = (int *)malloc(size); random_ints(b, N);
        c = (int *)malloc(size);

        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice); // Αντιγραφή του a στη συσκευή
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice); // Αντιγραφή του b στη συσκευή

        add<<<N,1>>>(d_a, d_b, d_c);   // Εκτέλεση της add() στην GPU με N blocks

        cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost); // Αντιγραφή αποτελεσμάτων σε host

        free(a); free(b); free(c);                        // Αποδέσμευση μνήμης
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

CUDA Threads

- **Ορολογία: κάθε block μπορεί να διαχωριστεί σε πολλαπλά παράλληλα threads.**
- Θα αλλάξουμε την `add()` ώστε να χρησιμοποιεί παράλληλα threads αντί για παράλληλα blocks.

```
__global__ void add(int *a, int *b, int *c) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

- Χρησιμοποιούμε το `threadIdx.x` αντί `blockIdx.x`.
- Απαιτείται μία μικρή αλλαγή και στη `main()`.

Πρόσθεση διανυσμάτων (threads)

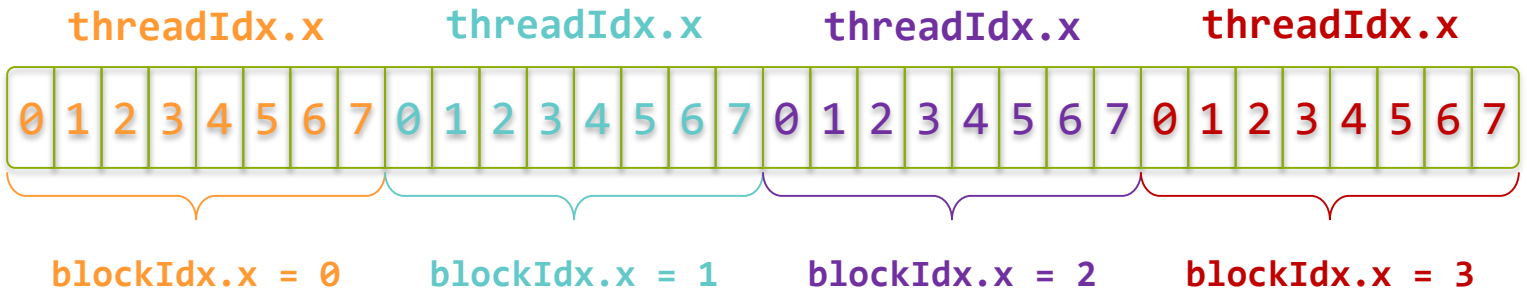
```
#define N 512
int main (int argc, char ** argv) {
        int *a, *b, *c;                         // Αντίγραφα οικοδεσπότη (host) a, b, c
        int *d_a, *d_b, *d_c;                   // Αντίγραφα συσκευής (device) a, b, c
        int size = N * sizeof(int);

        cudaMalloc((void **)&d_a, size);        // Δέσμευση μνήμης για το device a
        cudaMalloc((void **)&d_b, size);        // Δέσμευση μνήμης για το device b
        cudaMalloc((void **)&d_c, size);        // Δέσμευση μνήμης για το device c

        // Δέσμευση μνήμης για τα αντίγραφα οικοδεσπότη a, b, c. Αρχικοποίηση με τυχαίους.
        a = (int *)malloc(size); random_ints(a, N);
        b = (int *)malloc(size); random_ints(b, N);
        c = (int *)malloc(size);

        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice); // Αντιγραφή του a στη συσκευή
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice); // Αντιγραφή του b στη συσκευή

        add<<<1,N>>>(d_a, d_b, d_c);     // Εκτέλεση της add() στην GPU με N threads

        cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost); // Αντιγραφή αποτελεσμάτων σε host

        free(a); free(b); free(c);                        // Αποδέσμευση μνήμης
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

Συνδυασμός Blocks και Threads

- Τα παραπάνω παραδείγματα αφορούσαν στον υπολογισμό του αθροίσματος δύο διανυσμάτων κάνοντας χρήση:
  - Πολλαπλών block με 1 thread το καθένα.
  - Ενός block με πολλαπλά threads.
- Το CUDA παρέχει ακόμα μία δυνατότητα:
  - Χρήση πολλαπλών block με πολλά threads το καθένα.

Δεικτοδότηση πινάκων (indexing)

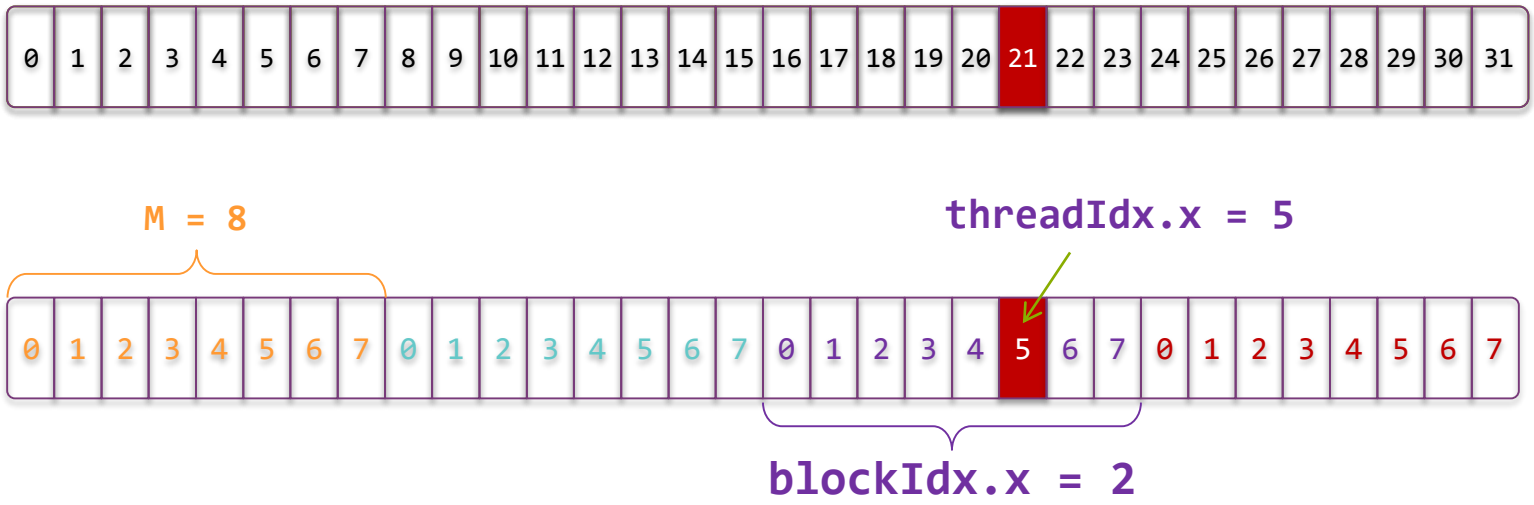
- Περισσότερο πολύπλοκο από την απλή χρήση των `blockIdx.x` και `threadIdx.x`.
- Έστω πίνακας που κατανέμεται σε N=4 blocks με M=8 threads το καθένα.



- Array Index = M * Block Index + Thread Index
- `int index = M * blockIdx.x + threadIdx.x`

Δεικτοδότηση πινάκων (παράδειγμα)

- Ποιο thread θα αναλάβει την εκτέλεση του κόκκινου στοιχείου?




```
int index = M * blockIdx.x + threadIdx.x =
          = 8 * 2 + 5 = 21
```

Πρόσθεση διανυσμάτων (blocks & threads)

- Θα κάνουμε χρήση της έκφρασης `blockDim.x` που επιστρέφει το μέγεθος του block.
- Η προηγούμενη έκφραση για το index θα γίνει:
- `int index = blockDim.x * blockIdx.x + threadIdx.x.`
- Συνδυαστική έκδοση της `add()` για παράλληλη χρήση threads και blocks.

```
__global__ void add(int *a, int *b, int *c) {
    int index = blockDim.x * blockIdx.x + threadIdx.x;
    c[index] = a[index] + b[index];
}
```

Πρόσθεση διανυσμάτων (blocks & threads)

```
#define N (2048 * 2048)
#define THREADS_PER_BLOCK 512
int main (int argc, char ** argv) {
        int *a, *b, *c;                         // Αντίγραφα οικοδεσπότη (host) a, b, c
        int *d_a, *d_b, *d_c;                   // Αντίγραφα συσκευής (device) a, b, c
        int size = N * sizeof(int);

        cudaMalloc((void **)&d_a, size);        // Δέσμευση μνήμης για το device a
        cudaMalloc((void **)&d_b, size);        // Δέσμευση μνήμης για το device b
        cudaMalloc((void **)&d_c, size);        // Δέσμευση μνήμης για το device c

        // Δέσμευση μνήμης για τα αντίγραφα οικοδεσπότη a, b, c. Αρχικοποίηση με τυχαίους.
        a = (int *)malloc(size); random_ints(a, N);
        b = (int *)malloc(size); random_ints(b, N);
        c = (int *)malloc(size);

        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);  // Αντιγραφή του a στη συσκευή
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);  // Αντιγραφή του b στη συσκευή

        add<<< N/THREADS_PER_BLOCK,THREADS_PER_BLOCK >>>(d_a, d_b, d_c);

        cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);  // Αντιγραφή αποτελεσμάτων σε host

        free(a); free(b); free(c);                         // Αποδέσμευση μνήμης
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

Διαχείριση διανυσμάτων αυθαίρετου μήκους

- Συνήθως, τα περισσότερα προβλήματα δεν είναι «συμβατά» με τα πολλαπλάσια του blockDim.x που χρησιμοποιήσαμε προηγουμένως.
- Αποφυγή προσπάθειας πρόσβασης πέρα από τα όρια των πινάκων.

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = blockDim.x * blockIdx.x + threadIdx.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

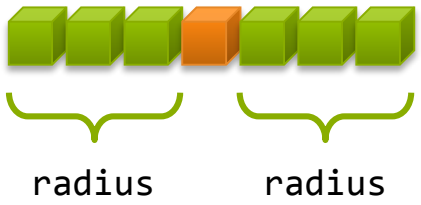
- Διαφορετικός τρόπος εκτέλεσης της kernel function.
- **`add<<<(N + M-1)/M, M>>>(d_a, d_b, d_c, N);`**

Γιατί threads;

- Από τη στιγμή που υπάρχουν τα blocks, φαίνονται αχρείαστα.
- Επιπλέον, συμβάλλουν στην αύξηση της πολυπλοκότητας του κώδικα.
- Όμως, σε αντίθεση με τα παράλληλα blocks τα threads διαθέτουν μηχανισμούς:
  - Επικοινωνίας.
  - Συγχρονισμού.

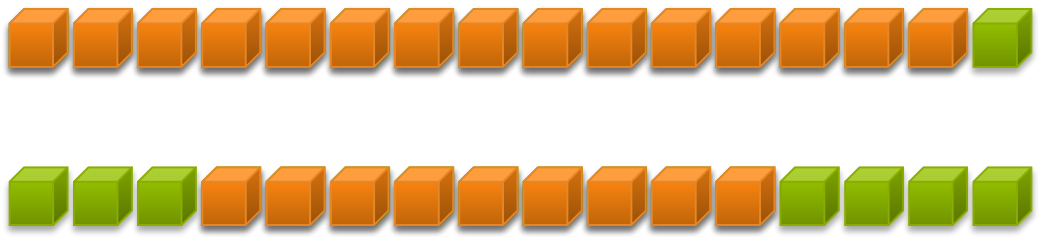
Μονοδιάστατο παράθυρο (1D stencil)

- Έστω μονοδιάστατος πίνακας στον οποίο εφαρμόζεται παράθυρο (1D stencil) μήκους `radius`.
- Κάθε στοιχείο εξόδου θα είναι το άθροισμα των στοιχείων του πίνακα μέσα σε αυτό το `radius`.
- Π.χ. για `radius=3`, κάθε στοιχείο εξόδου θα είναι το άθροισμα 7 στοιχείων εισόδου.



Υλοποίηση με CUDA (blocks & threads)

- Κάθε thread επεξεργάζεται ένα στοιχείο εξόδου.
  - `blockDim.x` στοιχεία ανά block.
- Τα στοιχεία εισόδου διαβάζονται πολλές φορές.
  - Με `radius=3`, κάθε στοιχείο εισόδου διαβάζεται 7 φορές.

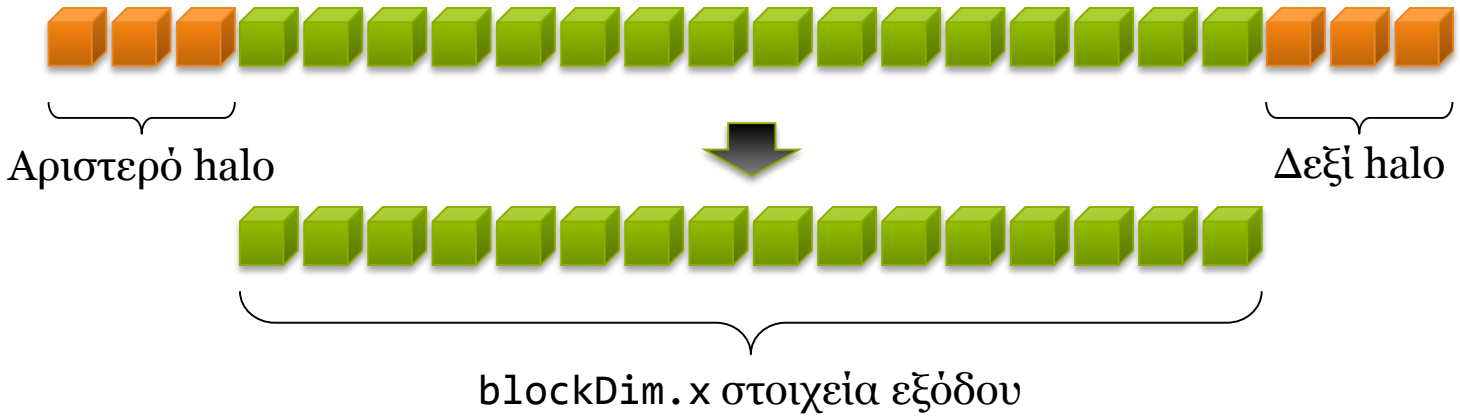


Διαμοίραση δεδομένων μεταξύ threads

- Μέσα σε ένα block, τα threads μοιράζονται δεδομένα μέσω ενός μηχανισμού κοινόχρηστης μνήμης.
- Πολύ γρήγορη on-chip memory, χειριζόμενη από το χρήστη.
- Ορίζεται με χρήση του keyword `__shared__` και δεσμεύεται ανά block.
- Τα δεδομένα ενός block δεν είναι ορατά από threads άλλων blocks.

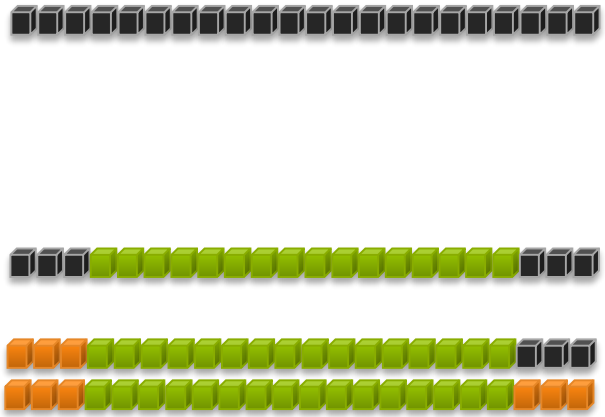
Υλοποίηση με κοινόχρηστη μνήμη

- Τοποθέτηση (cache) των δεδομένων στην κοινή μνήμη.
- Ανάγνωση `blockDim.x + 2 * radius` στοιχείων εισόδου από την κύρια μνήμη στην κοινόχρηστη.
- Υπολογισμός `blockDim.x` στοιχείων εξόδου.
- Εγγραφή `blockDim.x` στοιχείων στην κύρια μνήμη.
- Κάθε block χρειάζεται μία πλεονασματική ακτίνα (halo) σε αριστερά και δεξιά των στοιχείων εξόδου.



Stencil kernel

```
__global__ void stencil_1d (int *in, int *out) {
    // Κοινόχρηστη μνήμη για τα threads αυτού του block.
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];

    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Διάβασμα των στοιχείων εισόδου στην κοινόχρηστη μνήμη
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Συγχρονισμός των threads αυτού του block
    __syncthreads();

    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Αποθήκευση αποτελέσματος
    out[gindex] = result;
}
```



Συγχρονισμός threads

- Συνάρτηση __syncthreads().
- Συγχρονίζει όλα τα threads μέσα στο block.
  - Χρήσιμη για την αποφυγή καταστάσεων Read-After-Write, Write-After-Read και Write-After-Write.
- Όλα τα threads πρέπει να φτάσουν στο barrier που τίθεται.
- Σε κώδικα με συνθήκες, οι συνθήκες πρέπει να είναι ομοιόμορφες σε όλο το block.

Συντονισμός host/device

- Οι εκτελέσεις συναρτήσεων CUDA είναι ασύγχρονες.
- Ο έλεγχος επιστρέφει αμέσως στην CPU.
  - Η CPU πρέπει να συντονιστεί πριν λάβει τα αποτελέσματα της εκτέλεσης.

Εντολή CUDA	Λειτουργία
`cudaMemcpy()`	Η CPU μπλοκάρεται μέχρι να ολοκληρωθεί η αντιγραφή. Η αντιγραφή ξεκινά μόλις ολοκληρωθούν όλες οι κλήσεις CUDA.
`cudaMemcpyAsync()`	Ασύγχρονη, η CPU δεν μπλοκάρεται.
`cudaDeviceSynchronize()`	Η CPU μπλοκάρεται μέχρι να ολοκληρωθούν όλες οι κλήσεις CUDA.

Αναφορά σφαλμάτων

- Όλες οι κλήσεις στο CUDA API επιστρέφουν κωδικό σφάλματος του τύπου `cudaError_t`.
- Αυτό μεταφράζεται σε σφάλμα:
  - είτε την κλήση του API,
  - είτε σε προηγούμενες ασύγχρονες λειτουργίες.
- Λήψη κωδικού του τελευταίου σφάλματος:
  - `cudaError_t cudaGetLastError(void)`
- Λήψη περιγραφής σφάλματος:
  - **char** *cudaGetErrorString(cudaError_t)
  - printf("%s\n", cudaGetErrorString(cudaGetLastError()));

Διαχείριση συσκευής

- Η εφαρμογή μπορεί να υποβάλλει ερώτημα και να επιλέξει GPU:
  - `cudaGetDeviceCount(int *count)`
  - `cudaSetDevice(int device)`
  - `cudaGetDevice(int *device)`
  - `cudaGetDeviceProperties(cudaDeviceProp *prop, int device)`
- Πολλά threads μπορούν να μοιράζονται μία συσκευή.
- Ένα thread μπορεί να διαχειρίζεται πολλές συσκευές.
  - `cudaSetDevice(i)` επιλογή τρέχουσας συσκευής.
  - `cudaMemcpy(…)` peer-to-peer αντιγραφές μνήμης.