

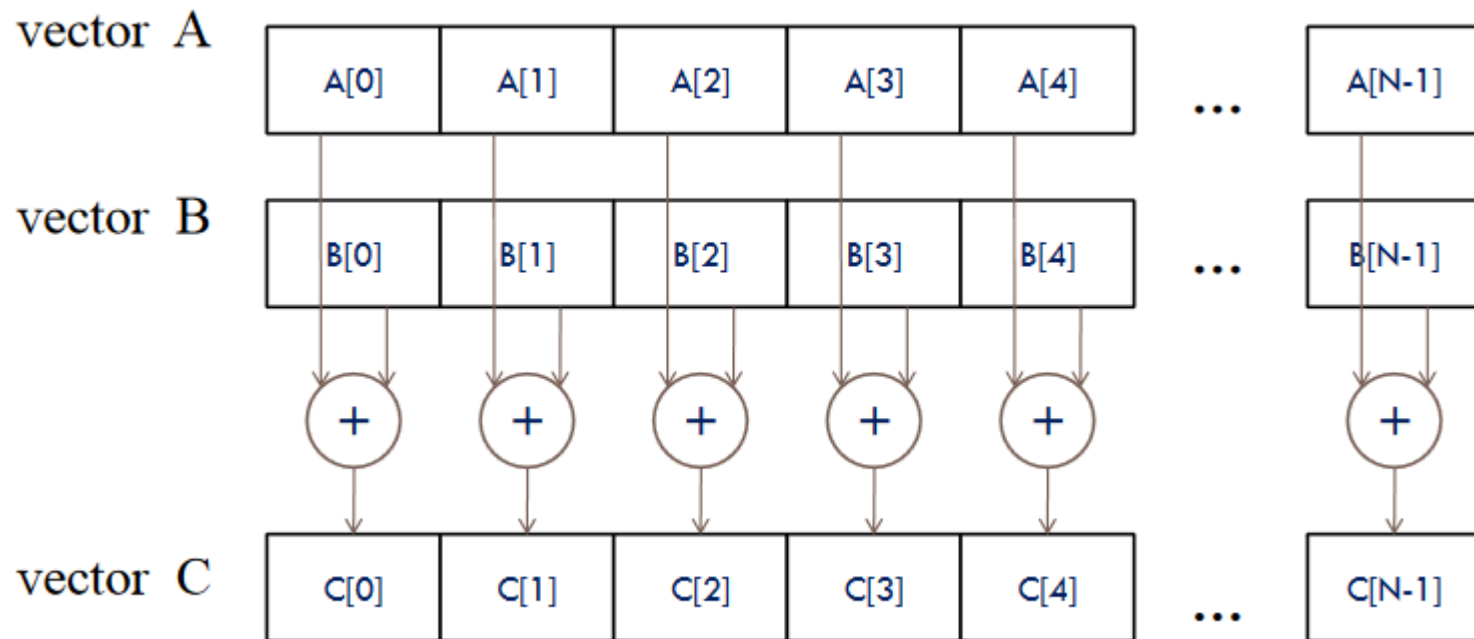
Παράλληλα Συστήματα

Χειμερινό εξάμηνο 2024–2025

CUDA #2



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

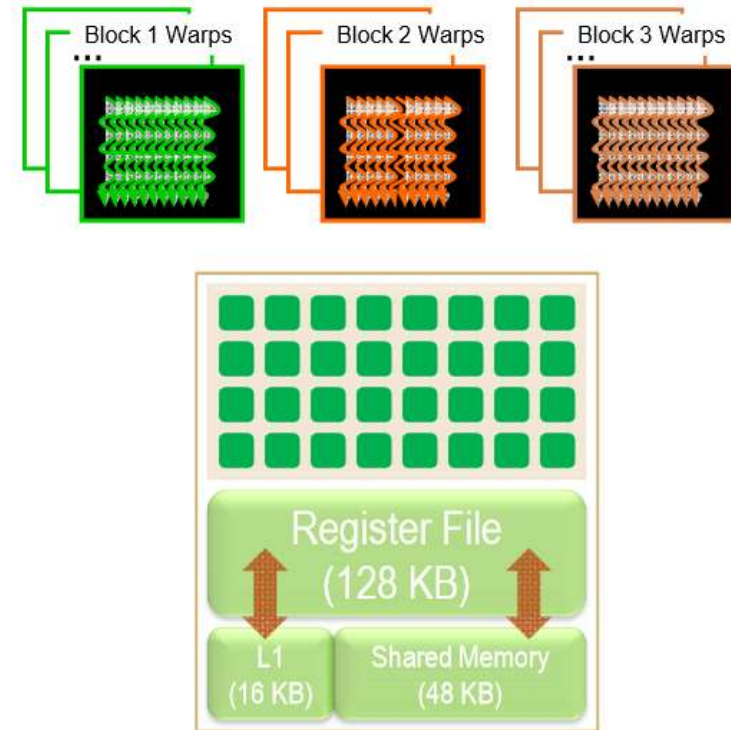
Άσκηση: algpu.cu

- ▶ 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

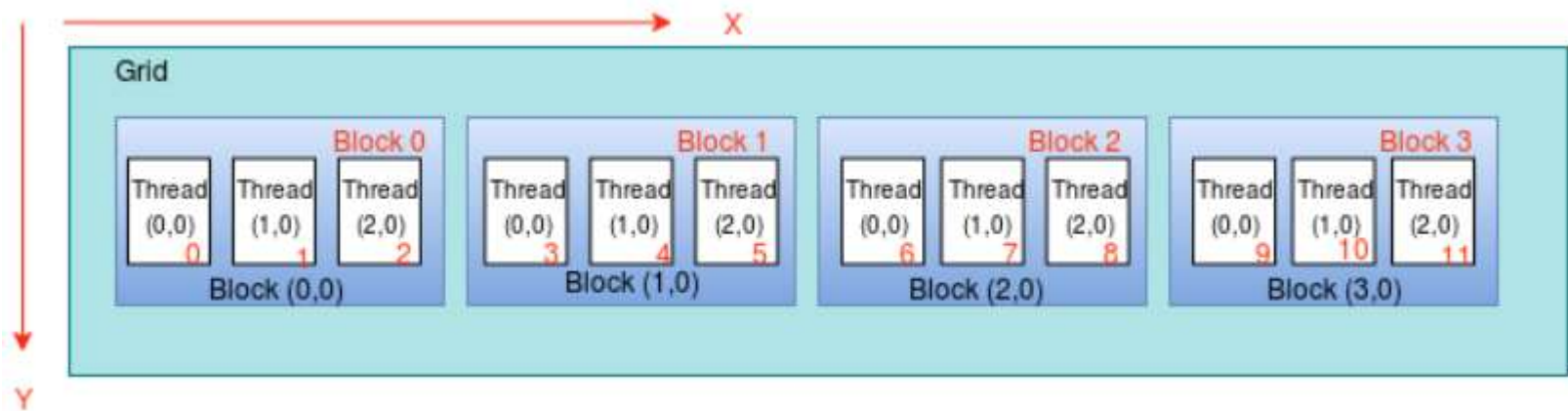
- ▶ Τα SM υλοποιούν χρονοπρογραμματισμό των warp με μηδενική επιβάρυνση
 - Κάθε χρονική στιγμή 1 ή 2 warp εκτελούνται από ένα SM
 - Τα warp των οποίων η επόμενη προς εκτέλεση εντολή έχει τα δεδομένα της έτοιμα προς χρήση μπορεί να επιλεγεί προς εκτέλεση
 - Τα έτοιμα προς εκτέλεση warp επιλέγονται για 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.

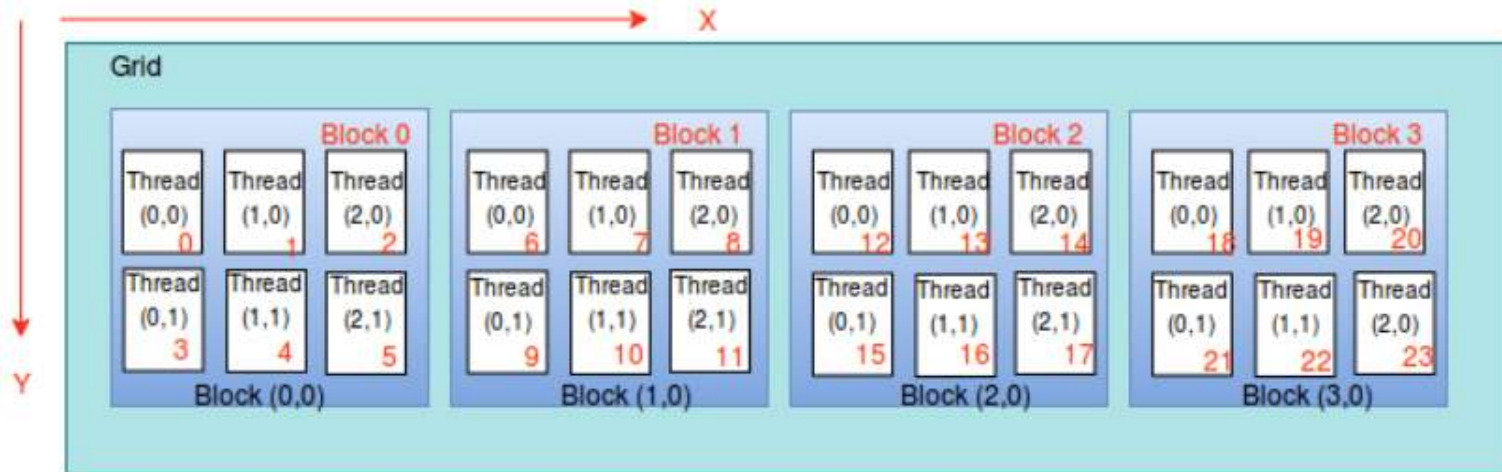
1D Grid από 1D Blocks

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

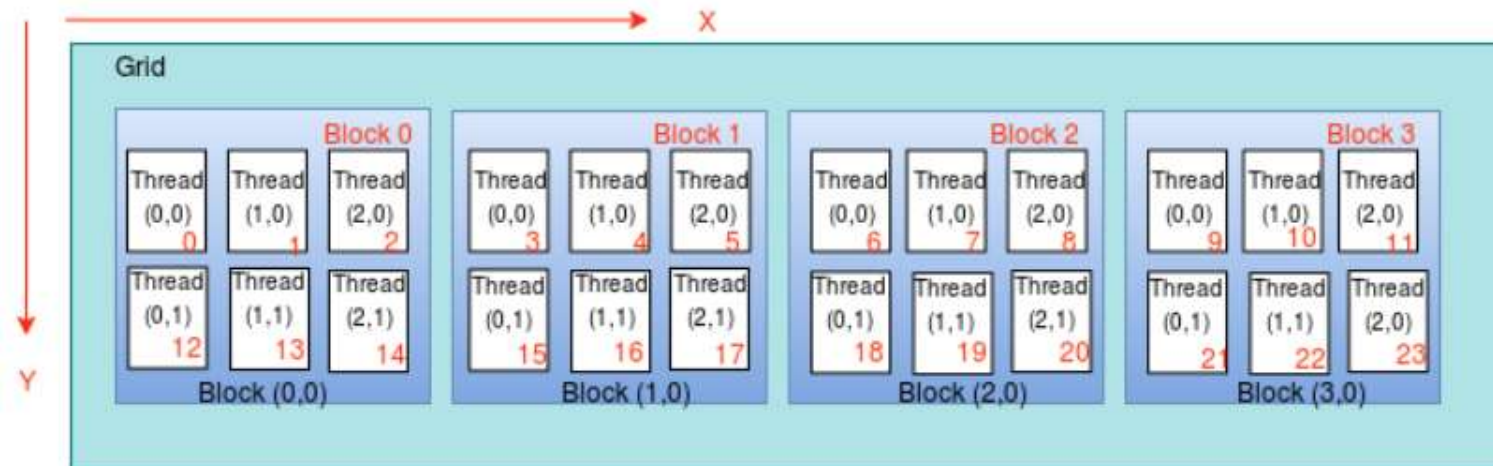


1D Grid από 2D Blocks

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

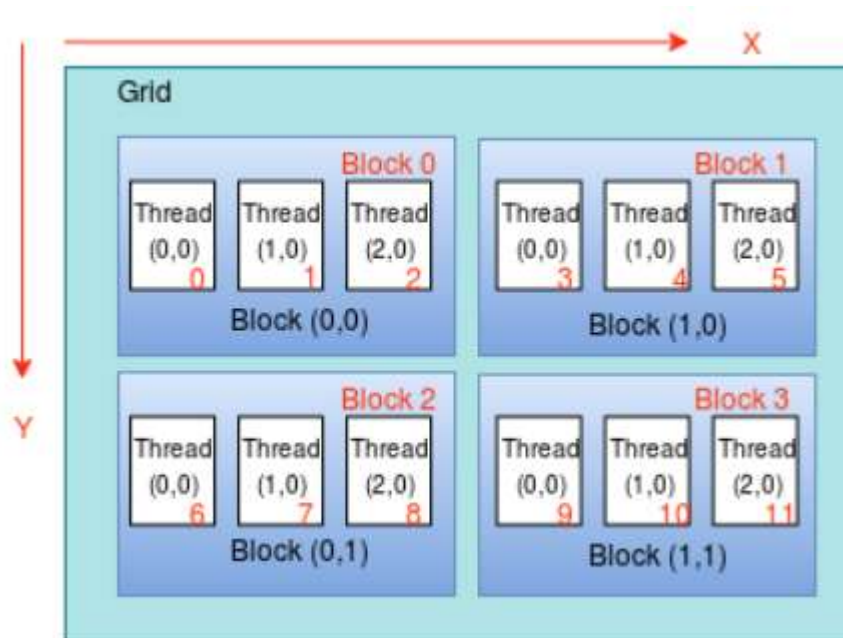


- ▶ $threadId = (gridDim.x * blockDim.x * threadIdx.y) + (blockDim.x * blockIdx.x) + threadIdx.x$



2D Grid από 1D Blocks

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



2D Grid από 2D Blocks [1]

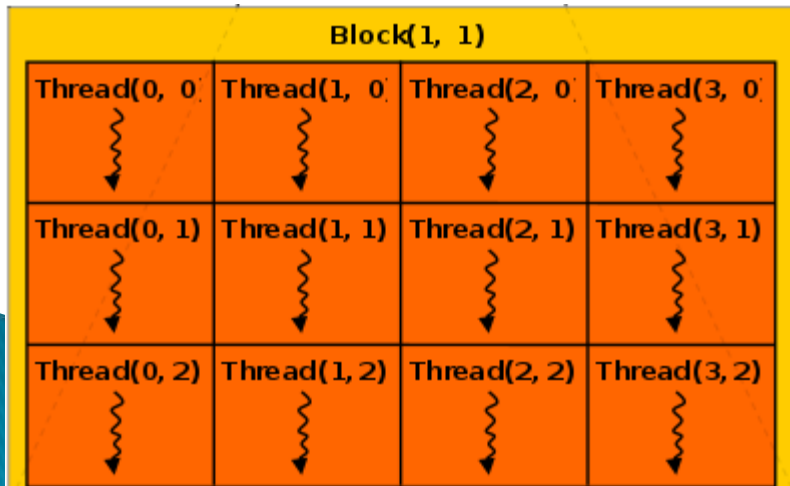
- ▶ $blockId = (gridDim.x * blockIdx.y) + blockIdx.x$
- ▶ $threadId = (blockId * (blockDim.x * blockDim.y)) + (threadIdx.y * blockDim.x) + threadIdx.x$



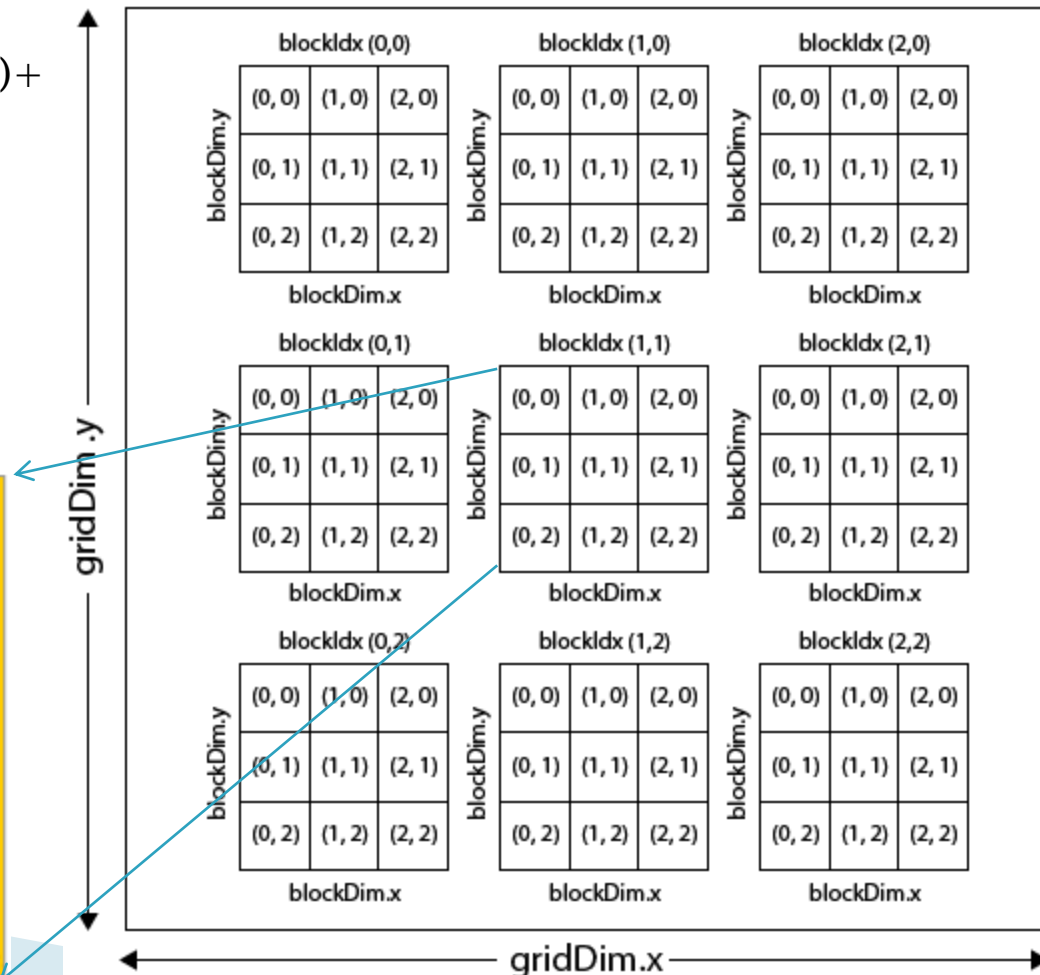
2D Grid από 2D Blocks [2]

$\text{blockId} = (\text{gridDim.x} * \text{blockIdx.y}) + \text{blockIdx.x}$

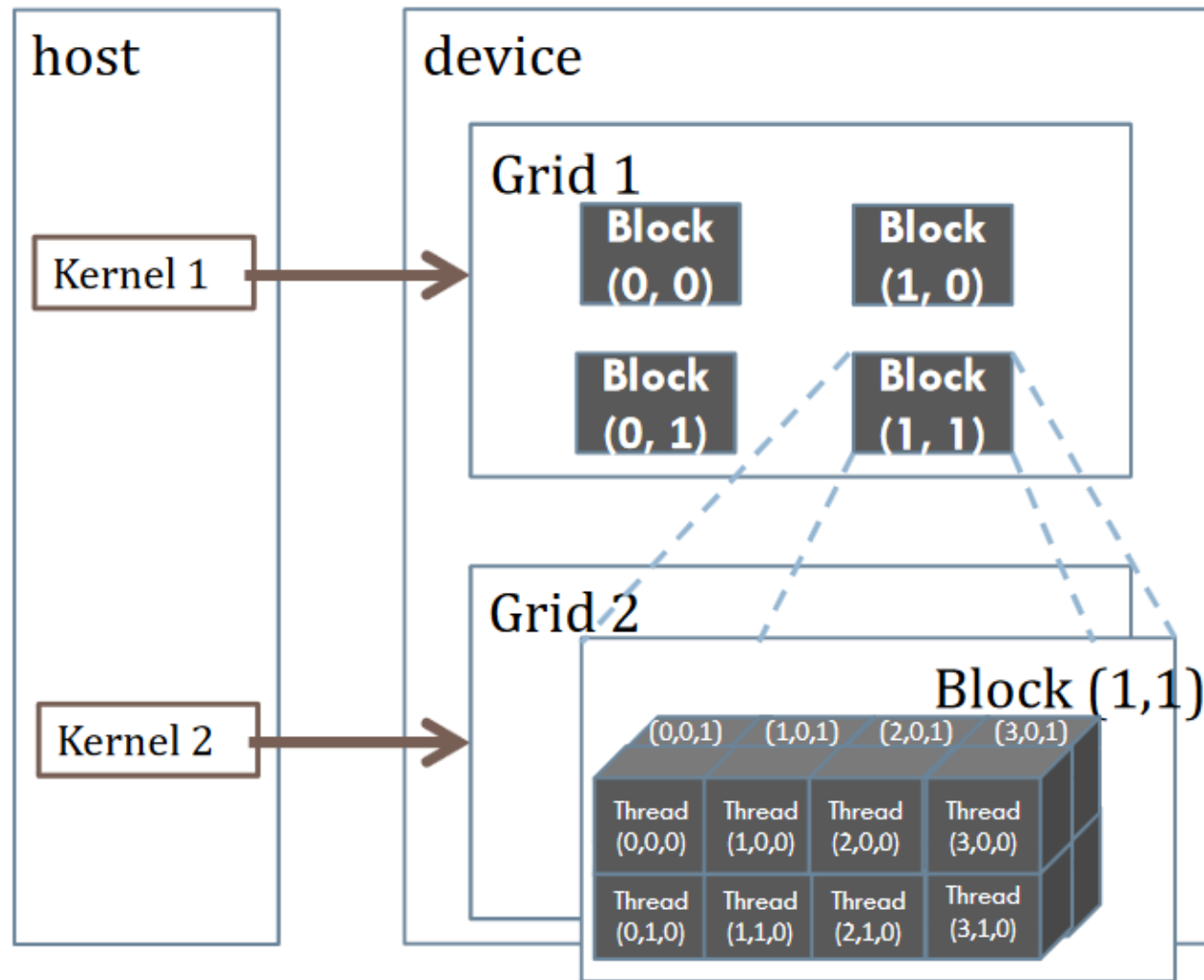
$\text{threadId} = \text{blockId} * (\text{blockDim.x} * \text{blockDim.y}) + (\text{threadIdx.y} * \text{blockDim.x}) + \text{threadIdx.x}$



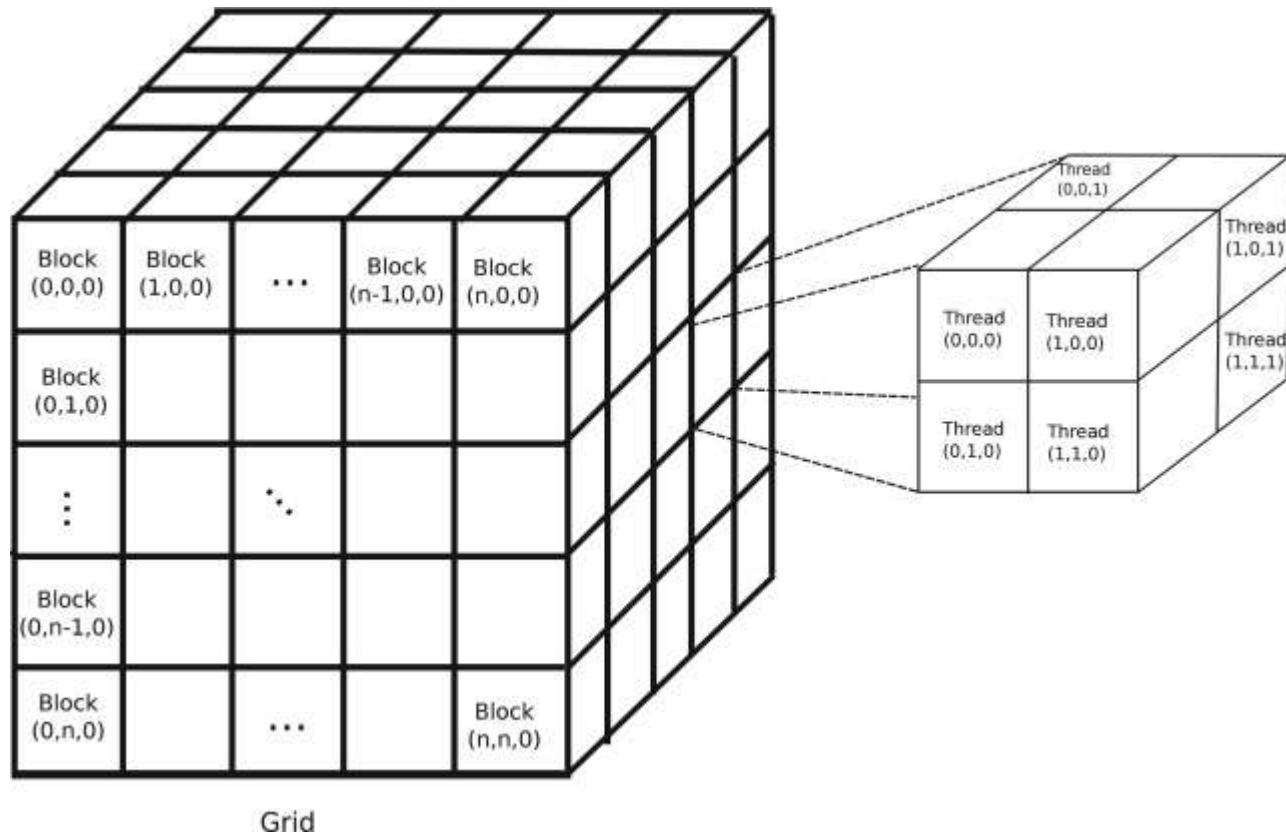
CUDA Grid



2D Grid από 3D Blocks



3D Grid από 3D Blocks



Ατομικές εντολές (γενικά)

- ▶ Πραγματοποιείται από μια εντολή μηχανής στα περιεχόμενα μιας διεύθυνση μνήμης
 - Διάβασε την παλιά τιμή, υπολόγισε την νέα τιμή και αποθήκευσε την νέα τιμή στην διεύθυνση μνήμης
- ▶ Το υλικό εξασφαλίζει πως κανένα άλλο νήμα δεν μπορεί να προσπελάσει την διεύθυνση μνήμης μέχρι να ολοκληρωθεί η ατομική εντολή
 - Κάθε άλλο νήμα που θα προσπαθήσει να προσπελάσει την διεύθυνση μνήμης θα πρέπει να αναστείλει την εκτέλεση του
 - Όλα τα νήματα τελικά εκτελούν την ατομική εντολή σειριακά

Ατομικές εντολές (στην CUDA)

- ▶ Είναι υλοποιημένες ως κλήσεις συναρτήσεων, που τελικά μεταφράζονται σε απλές εντολές μηχανής (intrinsics)
 - Atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap)
- ▶ Ατομική πρόσθεση προσημασμένου ακεραίου 32-bit
 - `int atomicAdd(int *address, int val);`
 - Διαβάζει την τρέχουσα τιμή των 32-bit στην οποία δείχνει ο δείκτης address (καθολική ή κοινή μνήμη)
 - Υπολογίζει την τιμή ($\text{old} + \text{val}$)
 - Αποθηκεύει την τιμή αυτή στην ίδια διεύθυνση μνήμης
 - Επιστρέφει ως αποτέλεσμα την προηγούμενη τιμή που υπήρχε στην διεύθυνση μνήμης

Περισσότερες ατομικές εντολές πρόσθεσης στην CUDA

- ▶ Ατομική πρόσθεση μη προσημασμένου ακεραίου 32-bit
 - `unsigned int atomicAdd(unsigned int *address, unsigned int val);`
- ▶ Ατομική πρόσθεση προσημασμένου ακεραίου 64-bit
 - `unsigned long long int atomicAdd(unsigned long long int *address, unsigned long long int val);`
- ▶ Ατομική πρόσθεση αριθμού κινητής υποδιαστολής μονής ακρίβειας (Compute capability > 2.0)
 - `float atomicAdd(float *address, float val);`

Υλοποίηση μη υπάρχουσας ατομικής εντολής με χρήση υπάρχουσας

```
__device__ void atomicAdd(float *address, float val)
{
    int *address_as_i = (int *)address;
    int old = *address_as_i, assumed;
    do {
        assumed = old;
        old = atomicCAS(address_as_i, assumed,
            __float_as_int(val + __int_as_float(assumed)));
    } while (assumed != old);
}
```

race.cu

- ▶ Πρόγραμμα με race condition
 - Που υπάρχουν θέματα;

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

__global__ void kernel(int *a_d) {
    *a_d += 1;
}

int main() {
    int a=0, *a_d;
    cudaMalloc((void**) &a_d, sizeof(int));
    cudaMemcpy(a_d, &a, sizeof(int), cudaMemcpyHostToDevice);

    kernel<<<1000,1000>>>(a_d);

    cudaMemcpy(&a, a_d, sizeof(int), cudaMemcpyDeviceToHost);

    printf("a = %d\n", a);
    cudaFree(a_d);
}
```

```
$ ./race
a = 132
```

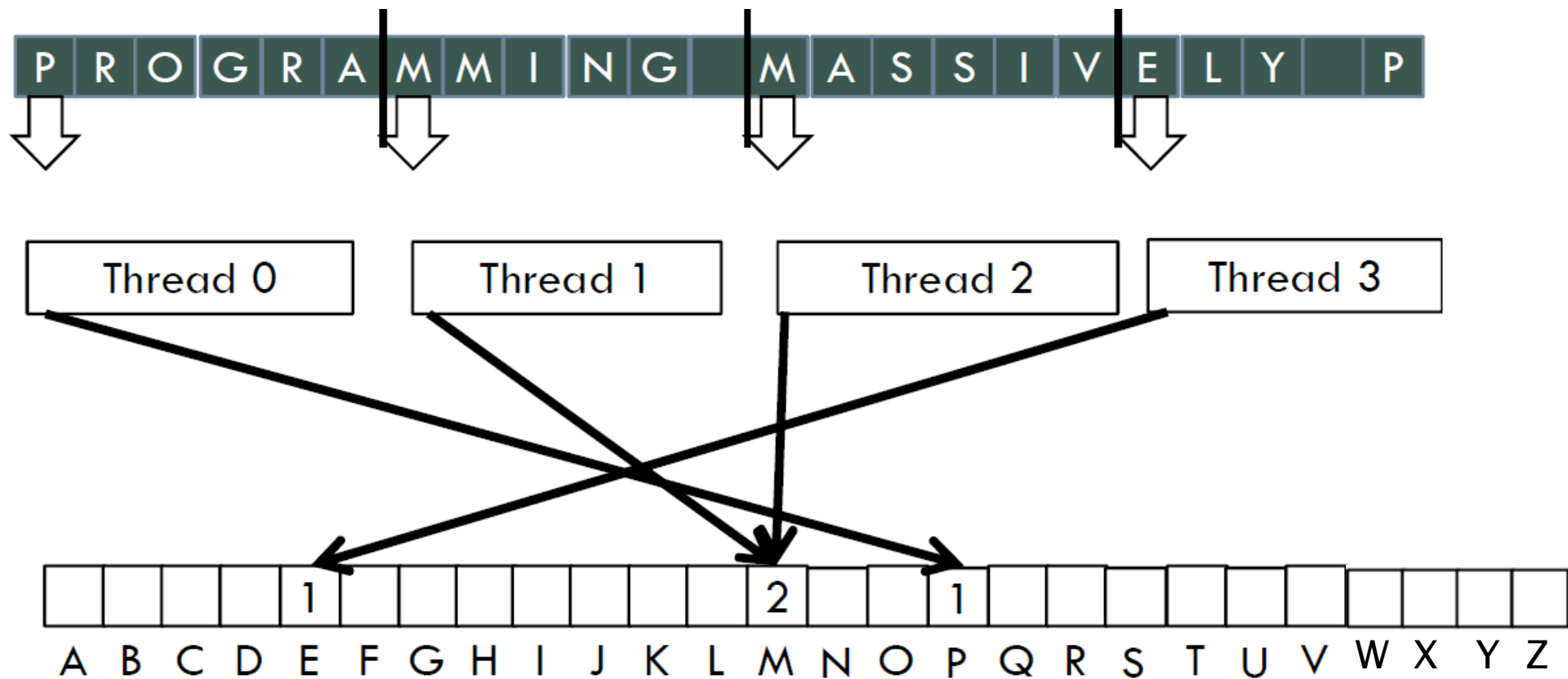
Υπολογισμός ιστογράμματος

- ▶ Μέθοδος για την εξαγωγή χρήσιμων χαρακτηριστικών και μοτίβων από μεγάλα σύνολα δεδομένων
 - Εξαγωγή χαρακτηριστικών για την αναγνώριση αντικειμένων σε εικόνες
 - Ανίχνευση απάτης σε συναλλαγές με πιστωτικές κάρτες
 - Συσχέτιση κινήσεων ουράνιων σωμάτων στην αστροφυσική
 - ...
- ▶ Βασικός αλγόριθμος
 - Χρησιμοποίησε την τιμή κάθε στοιχείου του συνόλου δεδομένων ως αναγνωριστικό ενός «δοχείου», του οποίου την τιμή θα αυξήσεις κατά ένα

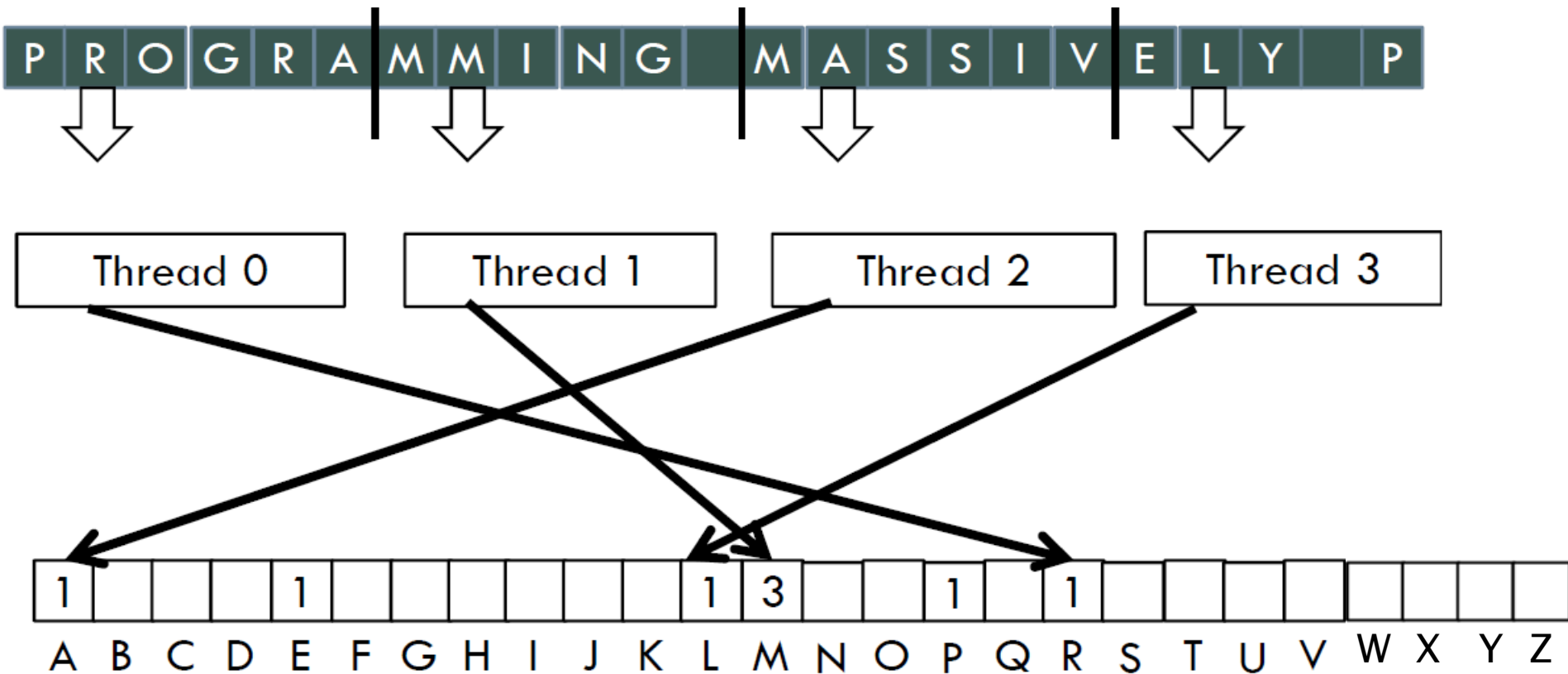
Παράδειγμα ιστογράμματος

- ▶ Για την πρόταση “Programming Massively Parallel Processors” φτιάξτε ένα ιστογράμμα για την συχνότητα εμφάνισης κάθε γράμματος
 - A(4), C(1), E(3), G(1), ...
- ▶ Πως το κάνουμε αυτό παράλληλα;
 - Ανέθεσε σε κάθε νήμα τον υπολογισμό για ένα τμήμα των δεδομένων εισόδου
 - Για κάθε χαρακτήρα, χρησιμοποίησε ατομικές εντολές για την δημιουργία του ιστογράμματος

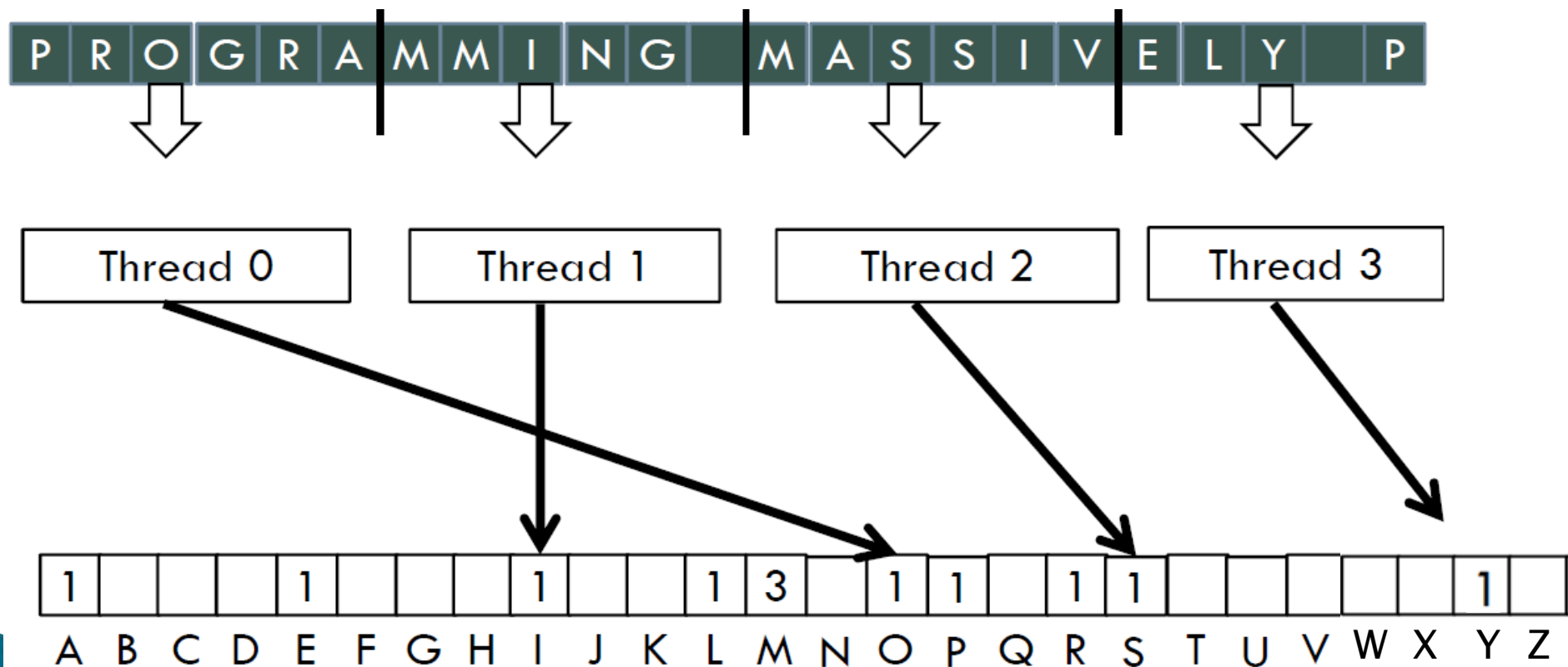
1η επανάληψη - 1ο γράμμα σε κάθε τμήμα



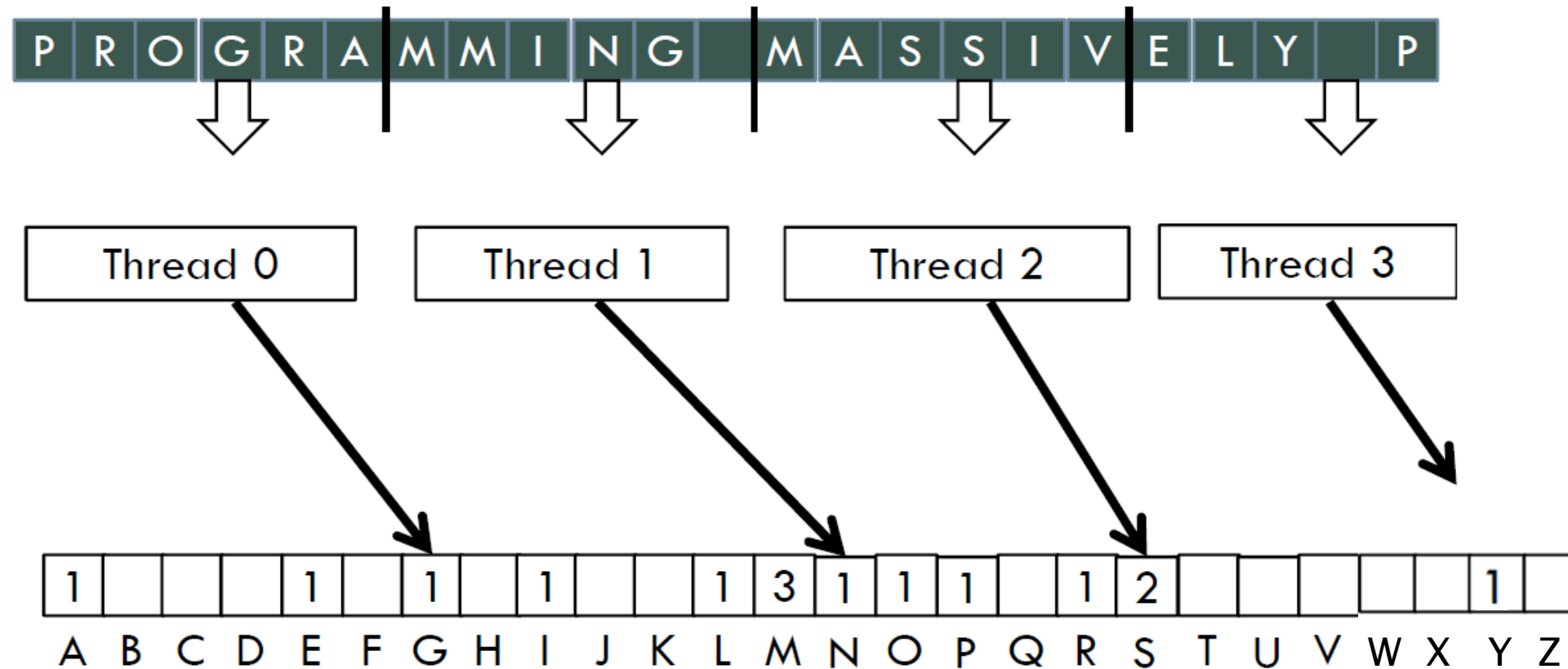
2η επανάληψη - 2ο γράμμα σε κάθε τμήμα



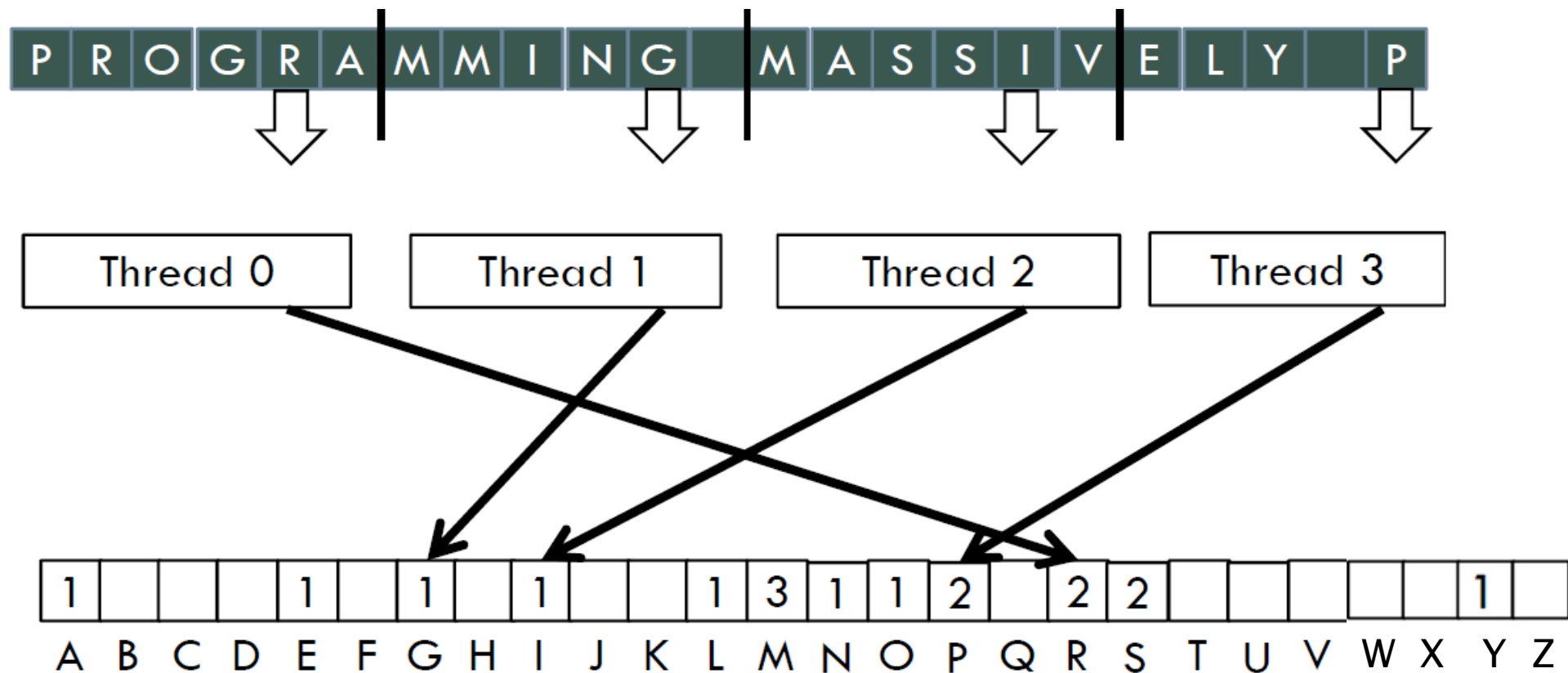
3η επανάληψη - 3ο γράμμα σε κάθε τμήμα



4η επανάληψη - 4ο γράμμα σε κάθε τμήμα

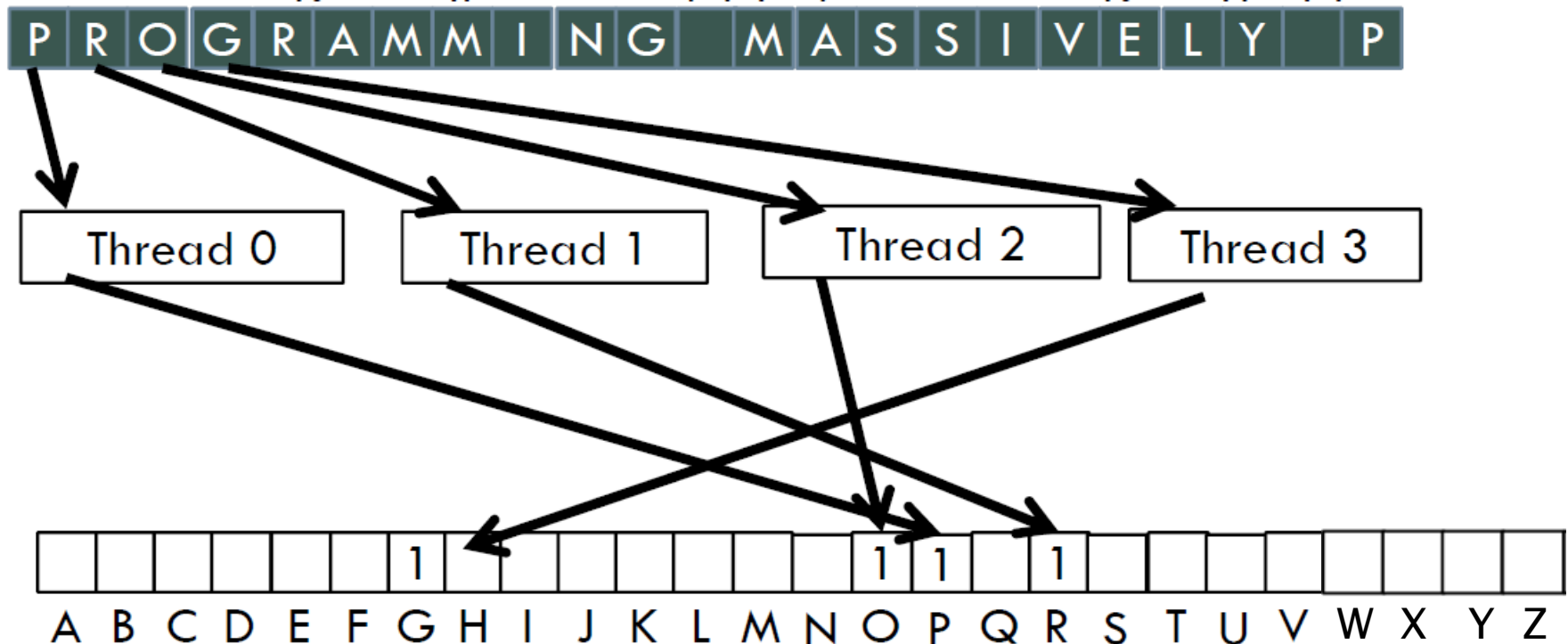


5η επανάληψη - 5ο γράμμα σε κάθε τμήμα



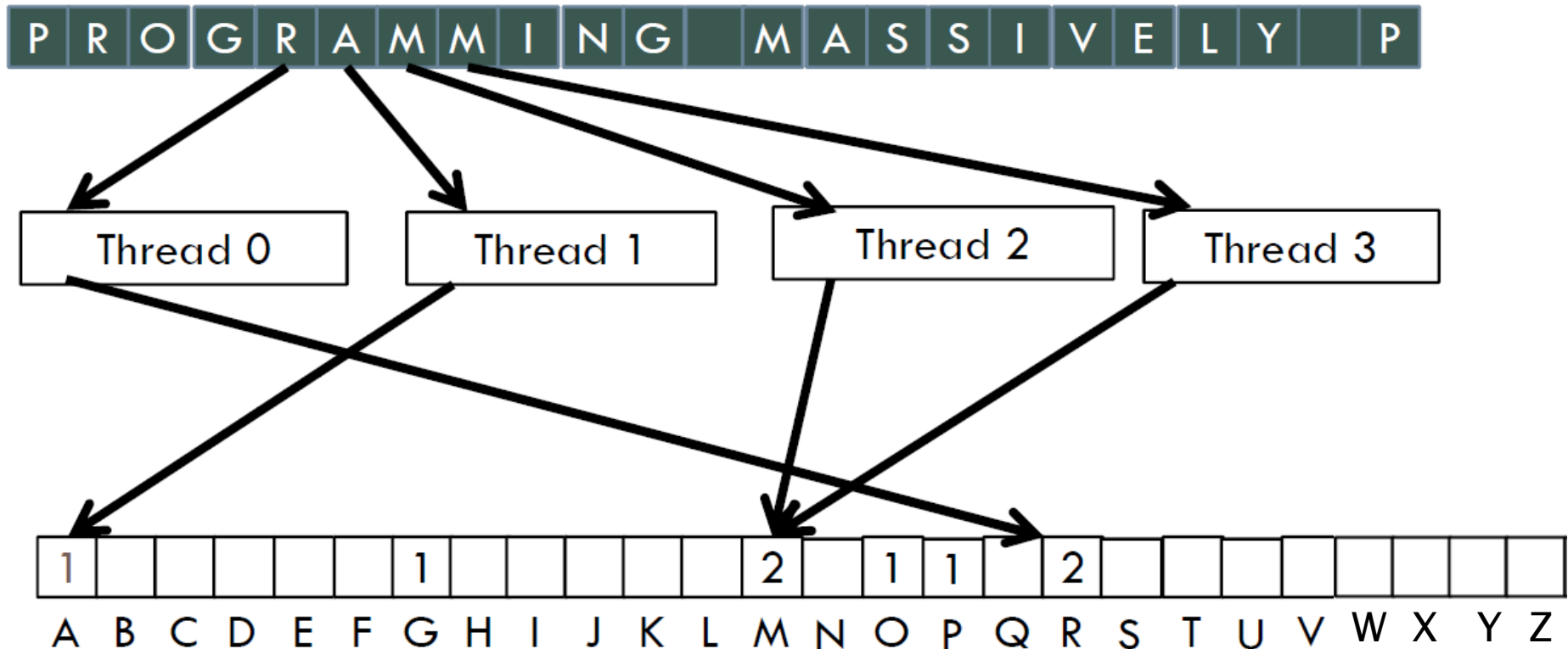
Που υπάρχει πρόβλημα στον αλγόριθμο;

- ▶ Διαβάζει είσοδο από μη συνεχόμενες θέσεις μνήμης
 - Ανάθεση στοιχείων στα νήματα σε βήματα (strides)
 - Διαδοχικά νήματα επεξεργάζονται διαδοχικά γράμματα



2η επανάληψη

- ▶ Όλα τα νήματα προχωράνε στο επόμενο τμήμα των δεδομένων εισόδου



hist_cpu.c

```
#include <stdio.h>
#define SIZE (100*1024*1024)

void* big_random_block( int size ) {
    unsigned char *data = (unsigned char*)malloc( size );
    for (int i=0; i<size; i++)
        data[i] = rand();
    return data;
}
```

```
$ ./hist_cpu
```

```
Time to generate: 236.1 ms
```

```
Histogram Sum: 104857600
```

```
int main( void ) {
    unsigned char *buffer =
        (unsigned char*)big_random_block( SIZE );

    // capture the start time
    clock_t      start, stop;
    start = clock();
    unsigned int  histo[256];
    for (int i=0; i<256; i++)
        histo[i] = 0;
    for (int i=0; i<SIZE; i++)
        histo[buffer[i]]++;
    stop = clock();
    float  elapsedTime = (float)(stop - start) /
        (float)CLOCKS_PER_SEC * 1000.0f;
    printf( "Time to generate: %3.1f ms\n", elapsedTime );

    long histoCount = 0;
    for (int i=0; i<256; i++) {
        histoCount += histo[i];
    }
    printf( "Histogram Sum: %ld\n", histoCount );

    free( buffer );
    return 0;
}
```

Συνάρτηση πυρήνα για ιστογράμμο

- ▶ Η συνάρτηση πυρήνα θα παίρνει ως παράμετρο έναν δείκτη προς τα δεδομένα εισόδου
- ▶ Κάθε νήμα επεξεργάζεται τα δεδομένα αυτά σε βήματα (strided)

```
__global__ void histo_kernel(unsigned char *buffer, long size, unsigned int *histo)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    // stride is total number of threads
    int stride = blockDim.x * gridDim.x;
    // All threads handle blockDim.x * gridDim.x consecutive elements
    while (i < size) {
        atomicAdd( &(histo[buffer[i]]), 1);
        i += stride;
    }
}
```

Άσκηση: Υπολογισμός ιστογράμματος με GPU (υπόδειγμα: κώδικας CPU + kernel)

```
#include <stdio.h>
#define SIZE (100*1024*1024)

void* big_random_block( int size ) {
    unsigned char *data = (unsigned char*)malloc( size );
    for (int i=0; i<size; i++)
        data[i] = rand();
    return data;
}
```

```
__global__ void histo_kernel(unsigned char *buffer, long
size, unsigned int *histo)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    // stride is total number of threads
    int stride = blockDim.x * gridDim.x;
    // All threads handle blockDim.x * gridDim.x
    // consecutive elements

    while (i < size) {
        atomicAdd( &(histo[buffer[i]]), 1);
        i += stride;
    }
}
```

```
int main( void ) {
    unsigned char *buffer =
        (unsigned char*)big_random_block( SIZE );

    // capture the start time
    clock_t start, stop;
    start = clock();
    unsigned int histo[256];
    for (int i=0; i<256; i++)
        histo[i] = 0;
    for (int i=0; i<SIZE; i++)
        histo[buffer[i]]++;
    stop = clock();
    float elapsedTime = (float)(stop - start) /
        (float)CLOCKS_PER_SEC * 1000.0f;
    printf( "Time to generate: %3.1f ms\n", elapsedTime );

    long histoCount = 0;
    for (int i=0; i<256; i++) {
        histoCount += histo[i];
    }
    printf( "Histogram Sum: %ld\n", histoCount );

    free( buffer );
    return 0;
}
```

Παράδειγμα: vecadd1.cu

- ▶ Size of vectors (n) = 1.000.000
- ▶ blockSize = 1024
- ▶ gridSize =
(int)ceil((float)n/blockSize);
- ▶ Event recording???
- ▶ vecAdd<<<gridSize, blockSize>>>

CUDA kernel launch with 977 blocks of
1024 threads
final result: 1.000000
Time for the kernel: 0.341376 ms


```
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}
```


Απόδοση & χρόνος

- ▶ Πως καταλαβαίνουμε πως μια οποιαδήποτε αλλαγή στον κώδικά μας επηρεάζει την απόδοση του προγράμματος;
 - Ποια έκδοση τρέχει πιο γρήγορα;
- ▶ Χρήση των timers του λειτουργικού συστήματος
 - Λανθάνων χρόνος (latency) και διακύμανση (variation) από διάφορες πηγές όπως πχ. Χρονοπρογραμματισμός νημάτων από το λειτουργικό σύστημα, ύπαρξη CPU timers υψηλής ακρίβειας, κλπ)
 - Όσο τρέχει ο πυρήνας της GPU, ενδέχεται να γίνονται ασύγχρονα υπολογισμοί στον host
- ▶ Ο μόνος τρόπος να μετρήσουμε αυτούς τους υπολογισμούς στον host είναι χρησιμοποιώντας κάποιον μηχανισμό συγχρονισμού της CPU ή του λειτουργικού συστήματος.
 - Έτσι, για τη μέτρηση του χρόνου που δαπανά η GPU πάνω σε μια εργασία, θα πρέπει να χρησιμοποιήσουμε το CUDA event API.

Γεγονότα (events)

- ▶ Ένα γεγονός στην CUDA είναι στην πράξη μια χρονοσφραγίδα της GPU (GPU time stamp) που καταγράφεται σε μια προκαθορισμένη χρονική στιγμή
 - ▶ Επειδή η GPU καταγράφει τη σφραγίδα, παρακάμπτει πολλά προβλήματα που θα αντιμετωπίζαμε αν χρησιμοποιούσαμε CPU timers
 - ▶ Σχετικά εύκολη διαδικασία
- 

Διαδικασία καταγραφής [1]

- ▶ Δημιουργία γεγονότων αρχής και τέλους
 - `cudaEvent_t start, stop;`
 - `cudaEventCreate(&start);`
 - `cudaEventCreate(&stop);`
- ▶ Εκκίνηση καταγραφής αρχής
 - `cudaEventRecord(start, 0);`
- ▶ >>> Γεγονός προς καταγραφή <<<
- ▶ Εκκίνηση καταγραφής τέλους
 - `cudaEventRecord(stop, 0);`

Διαδικασία καταγραφής [2]

- ▶ Υπολογισμός χρονικής διάρκειας μεταξύ δύο γεγονότων
 - `cudaEventElapsedTime(&elapsedTime,start,stop);`
- ▶ Χρήση (πχ. εκτύπωση) του χρόνου αυτού
 - `printf ("Time for the kernel: %f ms\n", elapsedTime);`
- ▶ Απελευθέρωση δεσμευμένης μνήμης
 - `cudaEventDestroy(start)`
 - `cudaEventDestroy(stop)`

Ανάγκη συγχρονισμού

- ▶ Πρόβλημα! Κάποιες κλήσεις στην CUDA C είναι ασύγχρονες
 - Η GPU ξεκινά να εκτελεί τον κώδικά μας, αλλά η CPU συνεχίζει εκτελώντας την επόμενη γραμμή του προγράμματός μας πριν προλάβει να τελειώσει η GPU
- ▶ Μια κλήση `cudaEventRecord()` σημαίνει ότι μπαίνουν εντολές στην ουρά εργασιών προς εκτέλεση της GPU για καταγραφή του τρέχοντα χρόνου
 - Αυτό έχει σαν αποτέλεσμα ότι το γεγονός μας δεν θα καταγραφεί μέχρι η GPU να τελειώσει με όλες τις δουλειές της πριν την κλήση προς το `cudaEventRecord()`.
 - Για την περίπτωση της μέτρησης του σωστού χρόνου για το stop event, αυτό είναι ακριβώς αυτό που θέλουμε
 - Αλλά δεν μπορούμε να ασφάλεια να διαβάσουμε την τιμή του stop event μέχρι η GPU να έχει ολοκληρώσει την πρότερη εργασία της και να έχει καταγράψει το stop event.
- ▶ Ευτυχώς υπάρχει η δυνατότητα να ζητήσουμε από την CPU να συγχρονιστεί με το γεγονός χρησιμοποιώντας την κλήση `cudaEventSynchronize()`
 - `cudaEventSynchronize(stop);`

Παράδειγμα: vecadd2.cu

- ▶ Size of vectors (n) = 1.000.000
- ▶ blockSize = 1024
- ▶ gridSize = 10
- ▶ Event recording: Ακριβώς πριν και μετά το vecadd

CUDA kernel launch with 10 blocks of 1024 threads

final result: 1.000000

Time for the kernel: 0.336864 ms

```
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int k, id = blockIdx.x*blockDim.x+threadIdx.x;

    // Make sure we do not go out of bounds
    for (k = id; k < n; k += blockDim.x*gridDim.x) {
        c[k] = a[k] + b[k];
    }
}
```

Παράδειγμα: vecadd3.cu

- ▶ Ίδιο με το vecadd2.cu εκτός από τις θέσεις καταγραφής
 - `cudaEventRecord(start,0)` πριν την αντιγραφή των host vectors στην device
 - `cudaEventSynchronize(stop)` μετά την αντιγραφή του array πίσω στον host

CUDA kernel launch with 10 blocks of 1024 threads
final result: 1.000000
Time for the kernel: 6.608128 ms