GPUs: οργάνωση & προγραμματισμός

Λ8

Συστήματα & Λογισμικό Υψηλών Επιδόσεων

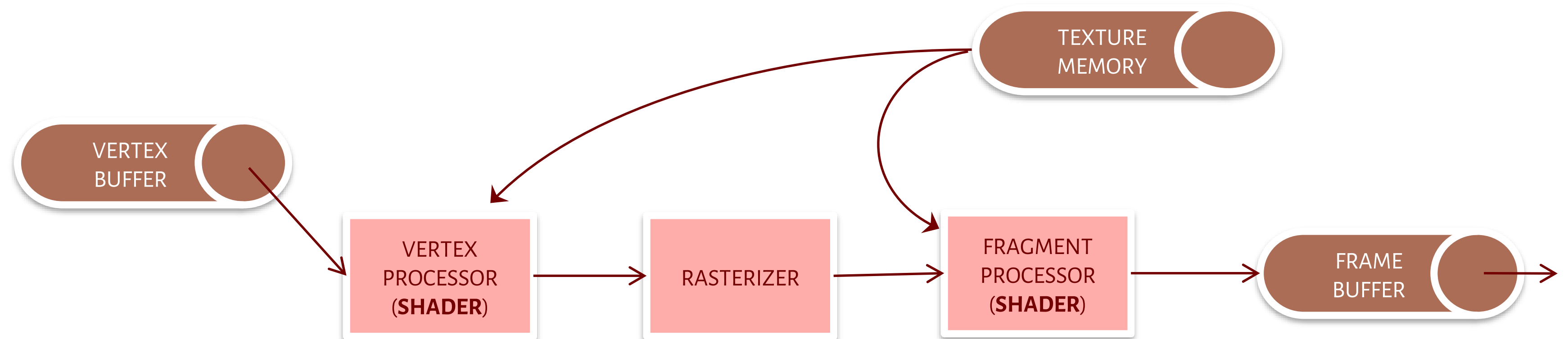
Ετερογένεια

- Τα σύγχρονα συστήματα είναι πλέον **ετερογενή**.
- Π.χ. παντού υπάρχει και ένας συν-επεξεργαστής (co-processor) ή ένας επιταχυντής (accelerator) ή μία κάρτα γραφικών (GPU)
- Ειδικά οι GPU είναι πλέον GPGPU (General-purpose GPU)
  - Μπορούν να εκτελέσουν γενικότερου σκοπού προγράμματα, πέρα από αυτά της γραφικής επεξεργασίας
  - «Κατεβάζουμε» κώδικα στην GPU
  - Τον εκτελεί η GPU (δεν ζωγραφίζει κάτι στην οθόνη προφανώς)
  - Παίρνουμε τα αποτελέσματα
- Σε αρκετούς τύπους υπολογισμών οι GPUs και γενικότερα οι επιταχυντές *είναι ταχύτεροι από τους κεντρικούς επεξεργαστές*

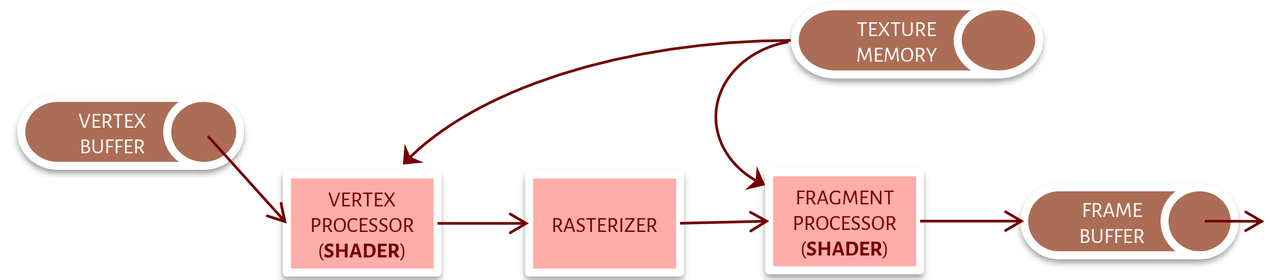
GPUs: οργάνωση

GPU history: Graphics pipeline

- Προκειμένου να αναπαραχθεί μία 3Δ σκηνή στην οθόνη, υπάρχει μία σειρά επεξεργασιών που απαιτούνται, που ονομάζονται συνολικά *graphics pipeline*.
- Κάποιες από (σχεδόν όλες) τις επεξεργασίες αυτές τις υλοποιούν στο hardware οι κάρτες γραφικών (GPUs)
- Χονδρικό, απλοποιημένο διάγραμμα:

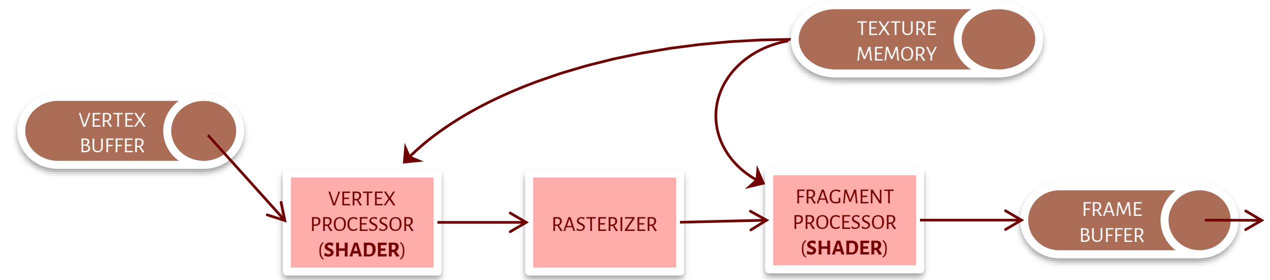


GPU history: Λειτουργία



- Η σκηνή περιγράφεται από κορυφές (vertices) στον 3Δ χώρο
- Το πρώτο στάδιο παίρνει κορυφές, τις επεξεργάζεται/μετασχηματίζει/φωτίζει/δίνει υφές και παράγει σύνολο τριγώνων. Επίσης κάνει προβολή τη σκηνής από τον 3Δ χώρο σε απεικόνιση στον 2Δ χώρο [vertex shaders]
- Το δεύτερο στάδιο χρησιμοποιεί μια «κάνναβο» ώστε ο συνεχής χώρος να ψηφιοποιηθεί σε τετραγωνίδια (fragments) – κάθε fragment τελικά θα αποτυπωθεί σε 1 pixel στην οθόνη.
- Το τρίτο στάδιο επεξεργάζεται/μετασχηματίζει/φωτίζει/δίνει υφές/υπολογίζει ορατότητες στα fragments και παράγει την έξοδο ως 1 pixel / fragment με συγκεκριμένο τελικό χρώμα. [fragment shaders]

GPU history: Λειτουργία



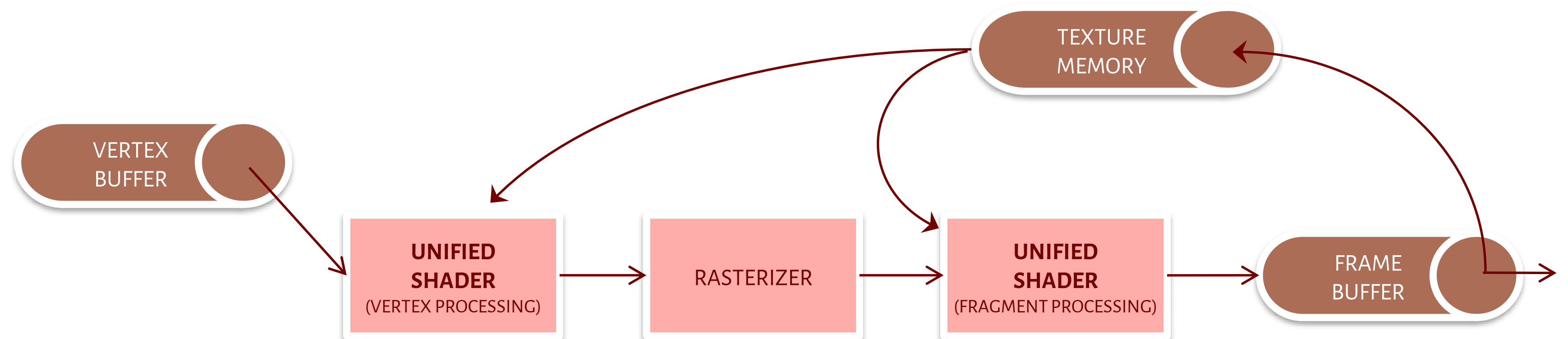
- Αρχικά: *fixed-function pipeline*
  - Οι λειτουργίες και το υλικό σε κάθε στάδιο ήταν προκαθορισμένα (π.χ. για φωτισμό της σκηνής) – οι shaders ήταν κυκλώματα ειδικού σκοπού.
  - Το πολύ-πολύ να υπήρχε μικρή επιλογή από fixed λειτουργίες
- Μετέπειτα: *programmable pipeline*
  - Οι shaders έγιναν μίνι επεξεργαστές και μπορούσαν να εκτελέσουν οποιαδήποτε ακολουθία εντολών επάνω σε μια κορυφή (vertex shader) ή σε ένα fragment (fragment/pixel shader).
  - Ο προγραμματιστής γράφει κώδικα (π.χ. σε HLSL, GLSL, Cg), τον «κατεβάζει» σε έναν shader και στη συνέχεια ο κώδικας εκτελείται για κάθε δεδομένο που περνάει από αυτόν.

GPU history: Λειτουργία

- Μεγέθη δεδομένων
  - Κορυφές: εκατομμύρια, floating point
  - Pixels: δισεκατομμύρια, fixed point
- Μοιραία, οι shaders έγιναν **πολλοί** σε κάθε στάδιο:
  - πολλαπλές μικρές και απλές CPU (ALU βασικά) που μπορούσαν να κάνουν πράξεις
  - εκτελούν τον ίδιο «κατεβασμένο» κώδικα ταυτόχρονα σε διαφορετικά vertices/pixels (SIMD!!)
  - παράλληλα => ταχύτερα
- Και, τελικά, φτάσαμε στην **ενοποίηση** (unified shaders) όπου έχουμε ένα σύνολο από ίδιες μικρές CPU που μπορούν να χρησιμοποιηθούν σε οποιοδήποτε στάδιο του pipeline (δηλ. είτε ως vertex είτε ως fragment shaders)
  - Συν κάποιος scheduler που επιλέγει ποια θα χρησιμοποιηθεί που και πότε

GPU history:
Η βασική ιδέα για γενική χρήση (GP) μίας GPU

- GPGPUs:
  - Κατέβασμα κώδικα (μη-γραφικής) εφαρμογής στους shaders
  - Τροφοδοσία με δεδομένα (ως «κορυφές» ή/και υφές)
  - Αποτελέσματα στον frame buffer
  - Με αντιγραφή των αποτελεσμάτων στη μνήμη υφών, μπορεί να ανατροφοδοτηθούν τα αποτελέσματα στους shaders για περαιτέρω επεξεργασία κ.ο.κ.

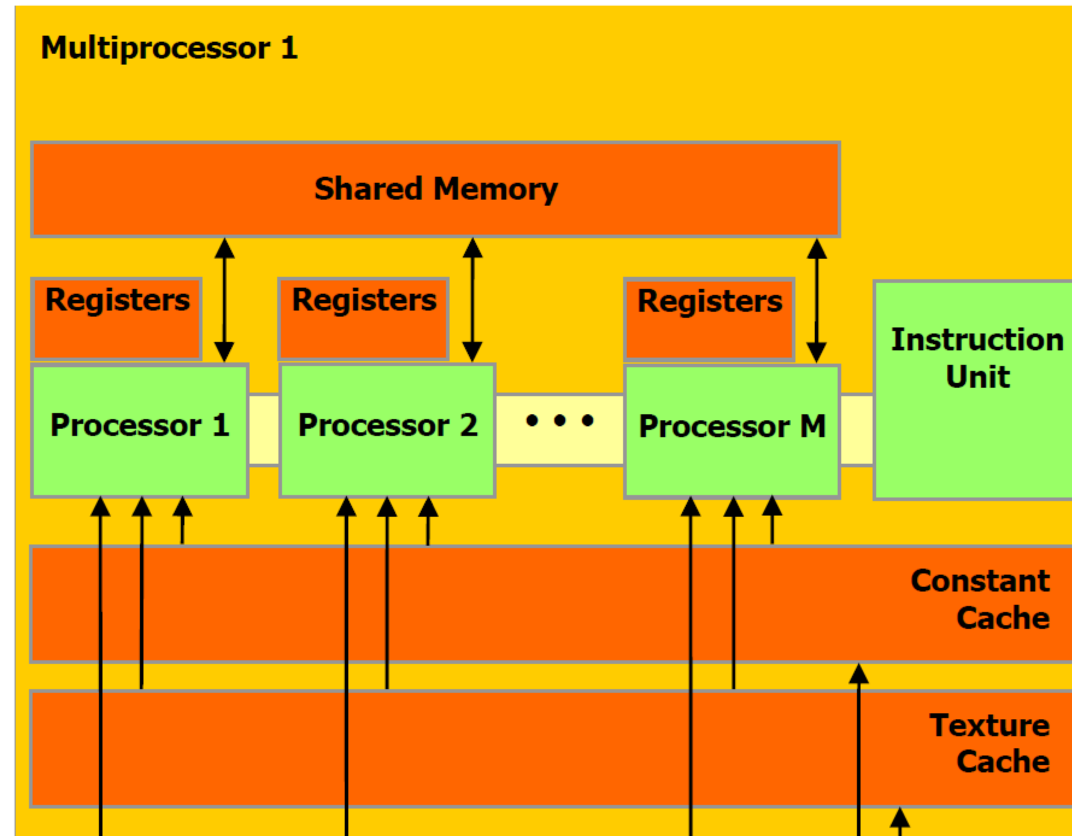


Η σημερινή αρχιτεκτονική

- Θα χρησιμοποιήσουμε την αρχιτεκτονική των CUDA GPU της NVIDIA, αλλά έτσι περίπου είναι όλες οι σύγχρονες GPU.
  - CUDA = Compute Unified Device Architecture (βασικά ενοποιημένοι shaders που μπορούν να εκτελέσουν γενικού σκοπού κώδικα)
- **“multiprocessor” ή “streaming multiprocessor” (MP ή SM):**
  - πολλά μικρά cores (=shaders, παλιότερα ονομάζονταν και “processors” ή “streaming processors”) που εκτελούν συγχρονισμένα την ίδια εντολή: *καθαρή οργάνωση SIMD*.
  - Κάθε core έχει δική του μικρή ιδιωτική μνήμη (καταχωρητές)
  - Υπάρχει επίσης μικρή (KBs) κοινόχρηστη μνήμη ανάμεσα στα core του SM
    - υπάρχουν και άλλες 2 κοινόχρηστες cache ειδικού σκοπού (read-only)

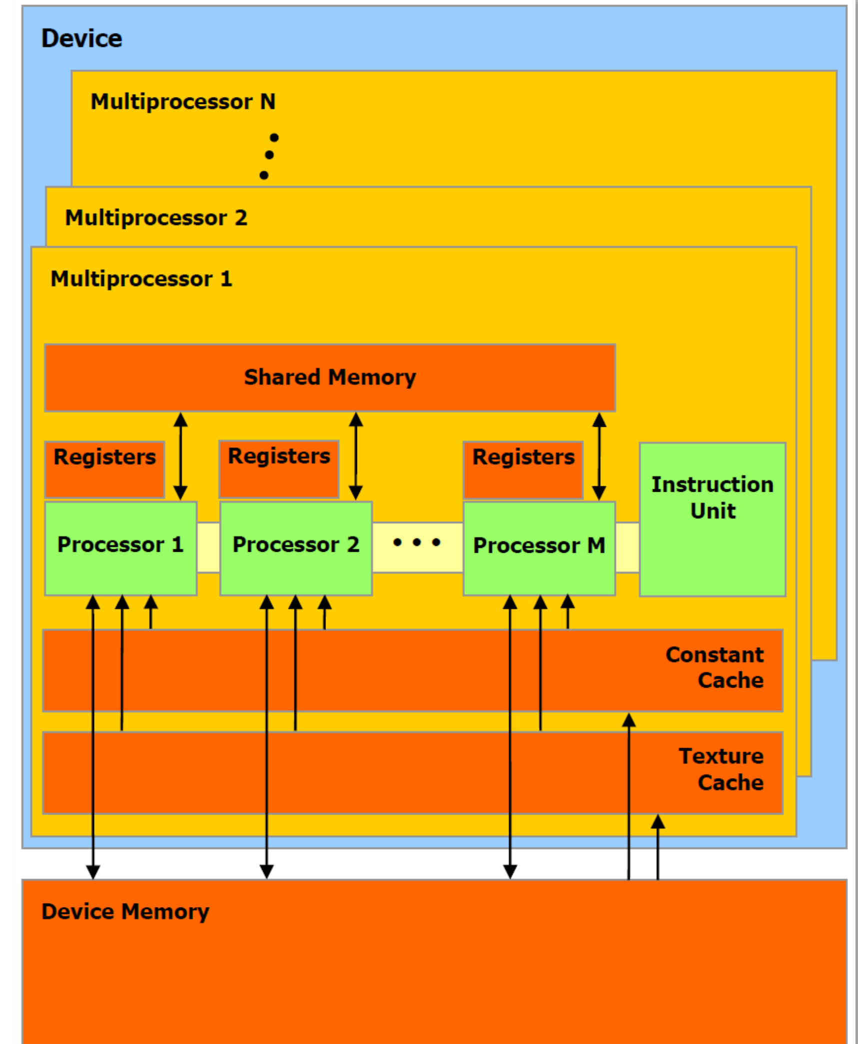
CUDA architecture: Δομή ενός SIMD "multiprocessor"

- Το Instruction Unit έχει την εντολή που εκτελούν όλοι, καθένας με δικά του δεδομένα
- Processor = πολύ απλό core

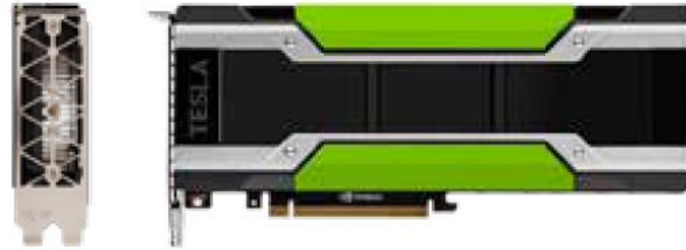


CUDA architecture: Μία πλήρης GPU

- GPU = πολλά multiprocessors + μνήμη
- Κάθε SM είναι ανεξάρτητο από το άλλο και μπορεί να εκτελεί δικό του κώδικα.
  - *Άρα SIMD οργάνωση και εκτέλεση μόνο εσωτερικά σε έναν SM.*
- Η Device Memory είναι συνήθως μεγάλη (Gbytes) και χρησιμοποιείται σχεδόν για τα πάντα
- Οι υπόλοιπες caches είναι για επιτάχυνση της προσπέλασης στην Device Memory (είναι μικρές όμως)
- Τα GPUs *δεν επενδύουν σε μνήμη/caches αλλά σε πάρα πολλά cores* - γενικά οι μνήμες τους πρέπει να θεωρούνται αργές.



Παράδειγμα: NVIDIA P40

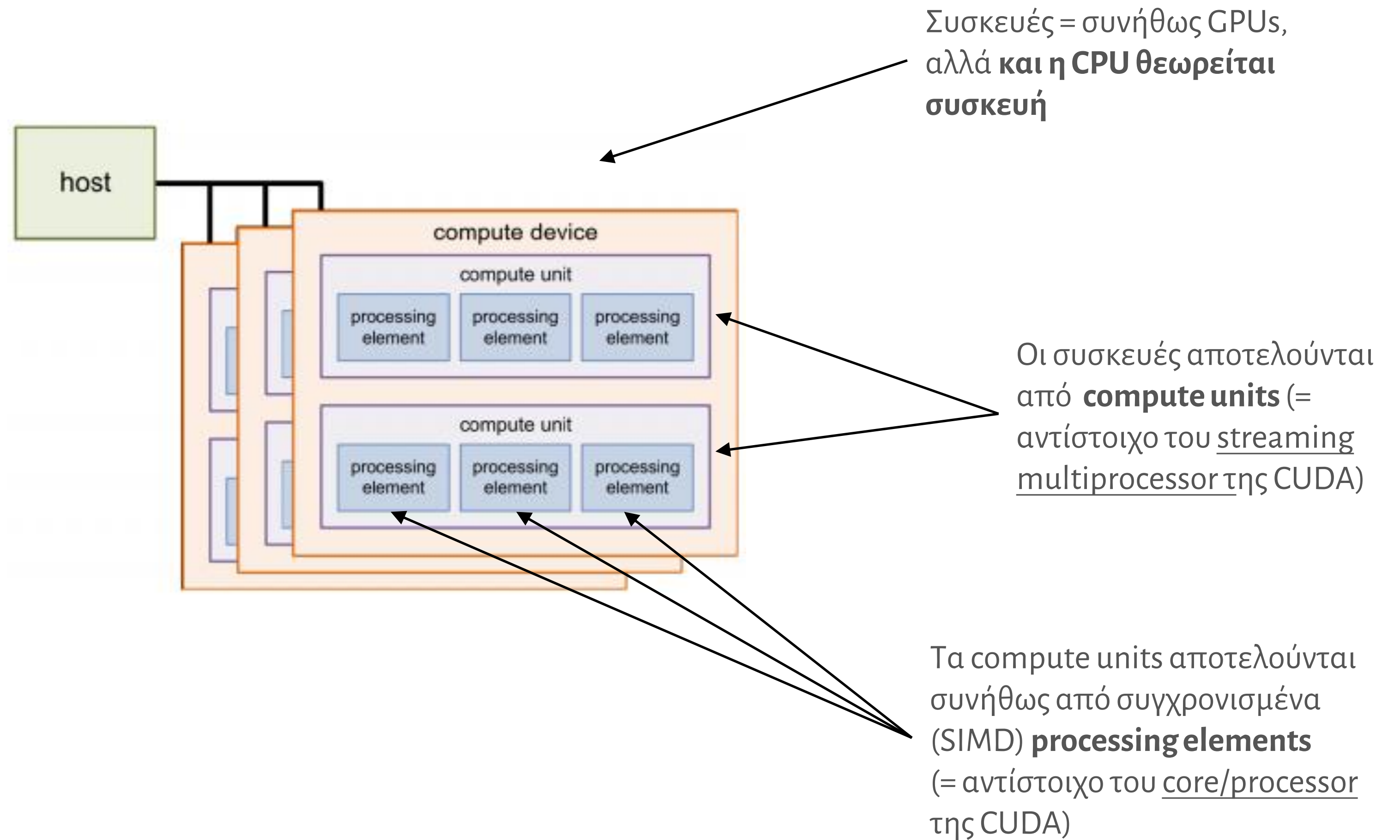


GPU	1 NVIDIA Pascal GPU
CUDA Cores	3,840
Memory Size	24 GB GDDR5
H.264 1080p30 streams	24
Max vGPU instances	24 (1 GB Profile)
vGPU Profiles	1 GB, 2 GB, 3 GB, 4 GB, 6 GB, 8 GB, 12 GB, 24 GB
Form Factor	PCIe 3.0 Dual Slot (rack servers)
Power	250 W
Thermal	Passive

30 SMs
128 cores/SM
24 GB device memory

Στο σύστημα parallax

OpenCL platform: Παρόμοια λογική, με άλλη ορολογία

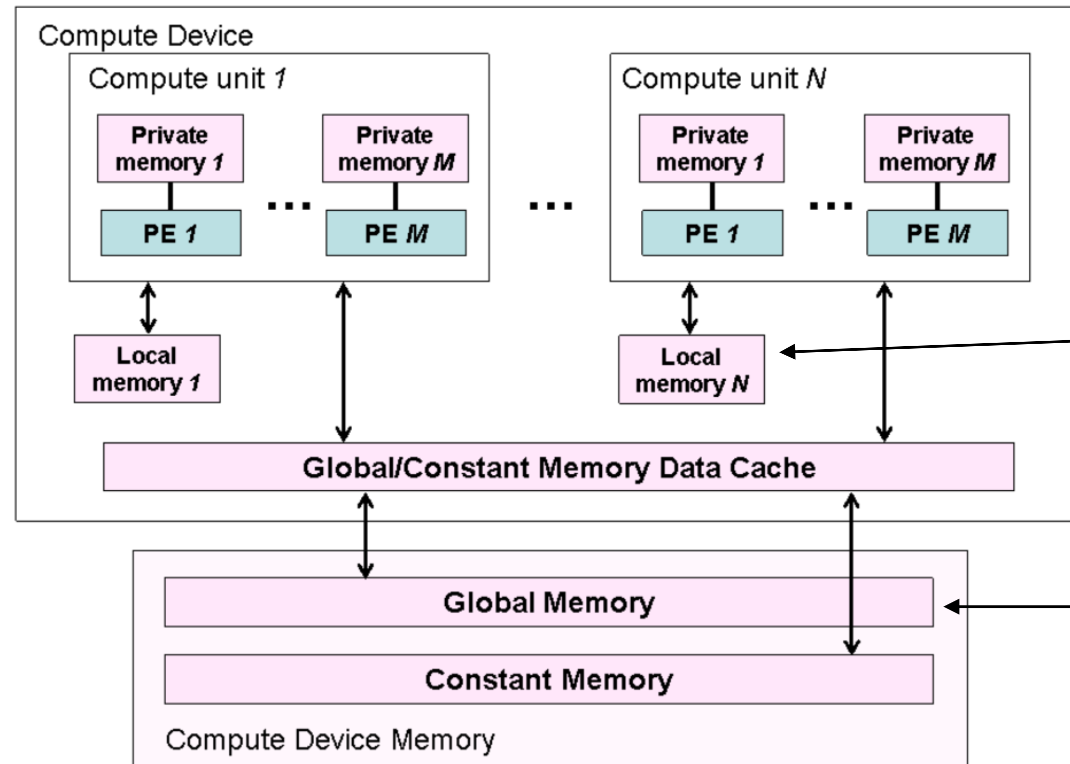


Συσκευές = συνήθως GPUs, αλλά **και η CPU θεωρείται συσκευή**

Οι συσκευές αποτελούνται από **compute units** (= αντίστοιχο του streaming multiprocessor της CUDA)

Τα compute units αποτελούνται συνήθως από συγχρονισμένα (SIMD) **processing elements** (= αντίστοιχο του core/processor της CUDA)

OpenCL memory: Παρόμοια λογική, με άλλη ορολογία



Local: αντίστοιχη της **shared memory** στην CUDA

Global: αντίστοιχη της **device memory** στην CUDA

GPUs: προγραμματισμός

Lower-level: OpenCL, CUDA

Παράδειγμα – OpenCL και CUDA

- Η παρακάτω συνάρτηση (“**kernel**”) θέλω να εκτελεστεί στη GPU:

```
void saxpy(int n, float a, float *b, float *c)
{
  for (int i = 0; i < n; ++i)
    c[i] = a*b[i] + c[i];
}
```

- saxpy = Single-precision A times X Plus Y

Πώς γίνεται στην πράξη η εκτέλεση

Πολύπλοκη και χρονοβόρα διαδικασία

1. Πρέπει να μεταφερθούν στην GPU τα δεδομένα (n, a, b και c)
   - Δεν αρκεί να περάσουμε δείκτες! Πρέπει να μεταφερθούν τα περιεχόμενα των διανυσμάτων (* εκτός αν υποστηρίζεται ενοποιημένη μνήμη μεταξύ host και device…)
2. Έπειτα πρέπει να εκτελεστεί ο κώδικας της συνάρτησης (kernel) στην GPU – “**offloading**”
   - Δεν έχουν ίδια γλώσσα μηχανής η CPU και η GPU…
   - Πρέπει να μεταφραστεί ξεχωριστά ο kernel είτε προκαταβολικά είτε την ώρα της εκτέλεσης (JIT)…
   - Πρέπει να μεταφερθεί ο κώδικάς του στην GPU
3. Αναμονή μέχρι να εκτελεστεί ο κώδικας και στη συνέχεια μεταφορά των αποτελεσμάτων **από** τη μνήμη της GPU

OpenCL

```c
#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>
#define VECTOR_SIZE 1024

// OpenCL kernel which is run for every work item created.
const char *saxpy =
"__kernel                               \n"
"void saxpy(float a,                    \n"
"           __global float *b,          \n"
"           __global float *c           \n"
"{                                      \n"
"    //Get the index of the work-item   \n"
"    int i = get_global_id(0);          \n"
"    c[i] = a * b[i] + c[i];            \n"
"}                                      \n";

int main(void) {
  int i;
  // Allocate space for vectors b and c in the host
  float a = 2.0f;
  float *b = (float*)malloc(sizeof(float)*VECTOR_SIZE);
  float *c = (float*)malloc(sizeof(float)*VECTOR_SIZE);
  for(i = 0; i < VECTOR_SIZE; i++) {
     b[i] = i;
     c[i] = VECTOR_SIZE - i;
  }

  // Get platform and device information
  cl_platform_id * platforms = NULL;
  cl_uint  num_platforms;
  //Set up the Platform
  cl_int clStatus = clGetPlatformIDs(0, NULL, &num_platforms);
  platforms = (cl_platform_id *) malloc(sizeof(cl_platform_id)*num_platforms);
  clStatus = clGetPlatformIDs(num_platforms, platforms, NULL);

  //Get the devices list and choose the device you want to run on
  cl_device_id  *device_list = NULL;
  cl_uint  num_devices;

  clStatus = clGetDeviceIDs( platforms[0], CL_DEVICE_TYPE_GPU, 0,NULL, &num_devices);
  device_list = (cl_device_id *) malloc(sizeof(cl_device_id)*num_devices);
  clStatus = clGetDeviceIDs( platforms[0],CL_DEVICE_TYPE_GPU, num_devices, device_list, NULL);

  // Create one OpenCL context for each device in the platform
  cl_context context;
  context = clCreateContext( NULL, num_devices, device_list, NULL, NULL, &clStatus);

   // Create a command queue
  cl_command_queue command_queue = clCreateCommandQueue(context, device_list[0], 0, &clStatus);

  // Create memory buffers on the device for each vector
  cl_mem b_clmem = clCreateBuffer(context, CL_MEM_READ_ONLY, VECTOR_SIZE * sizeof(float), NULL, &clStatus);
  cl_mem c_clmem = clCreateBuffer(context, CL_MEM_READ_WRITE,VECTOR_SIZE * sizeof(float), NULL, &clStatus);
```

The kernel

Find the device

Create context and queue

Allocate memory @ device for b, c

Copy b, c to device memory

```c
  // Copy the Buffer b and c to the device
  clStatus = clEnqueueWriteBuffer(command_queue, b_clmem, CL_TRUE, 0, VECTOR_SIZE * sizeof(float), a, 0,
                                  NULL, NULL);
  clStatus = clEnqueueWriteBuffer(command_queue, c_clmem, CL_TRUE, 0, VECTOR_SIZE * sizeof(float), b, 0,
                                  NULL, NULL);

  // Create a program from the kernel source
  cl_program program = clCreateProgramWithSource(context, 1, &saxpy_kernel, NULL, &clStatus);

  // Build the program
  clStatus = clBuildProgram(program, 1, device_list, NULL, NULL, NULL);

  // Create the OpenCL kernel
  cl_kernel kernel = clCreateKernel(program, "saxpy_kernel", &clStatus);

  // Set the arguments of the kernel
  clStatus = clSetKernelArg(kernel, 0, sizeof(float),  (void *)&a);
  clStatus = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&b_clmem);
  clStatus = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&c_clmem);

  // Execute the OpenCL kernel on the list
  size_t global_size = VECTOR_SIZE; // Process the entire lists
  size_t local_size = 64;           // Process one item at a time
  clStatus = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global_size, &local_size, 0,
                                    NULL, NULL);

  // Read the cl memory c_clmem on device to the host variable c
  clStatus = clEnqueueReadBuffer(command_queue, c_clmem, CL_TRUE, 0, VECTOR_SIZE*sizeof(float), c, 0,
                                 NULL, NULL);

  // Clean up and wait for all the commands to complete.
  clStatus = clFlush(command_queue);
  clStatus = clFinish(command_queue);

  // Display the result to the screen
  for(i = 0; i < VECTOR_SIZE; i++)
     printf("%f * %f + %f = %f\n", a, b[i], VECTOR_SIZE - i, c[i]);

  // Finally release all OpenCL allocated objects and host buffers.
  clReleaseKernel(kernel);
  clReleaseProgram(program);
  clReleaseMemObject(b_clmem);
  clReleaseMemObject(c_clmem);
  clReleaseCommandQueue(command_queue);
  clReleaseContext(context);
  free(b);
  free(c);
  free(platforms);
  free(device_list);
  return 0;
}
```

Create kernel binary (compile)

Set kernel arguments

Execute kernel

Copy result (c) from device

Cleanup

CUDA

```
#include <stdio.h>

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

int main(void)
{
  int N = 1024;
  float a = 2.0f;
  float *b, *c, *d_b, *d_c;
  b = (float*)malloc(N*sizeof(float));
  c = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_b, N*sizeof(float));
  cudaMalloc(&d_c, N*sizeof(float));

  for (int i = 0; i < N; i++) {
    b[i] = i;
    c[i] = N-i;
  }

  cudaMemcpy(d_b, b, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_c, c, N*sizeof(float), cudaMemcpyHostToDevice);

  // Perform SAXPY on 1M elements
  saxpy<<<1024, 256>>>(N, a, d_b, d_c);

  cudaMemcpy(c, d_c, N*sizeof(float), cudaMemcpyDeviceToHost);

  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = max(maxError, abs(c[i]-(a*i+N-i)));
  printf("Max error: %f\n", maxError);

  cudaFree(d_b);
  cudaFree(d_c);
  free(b);
  free(c);
}
```

The kernel

Allocate memory @ device for b, c

Copy b, c to device memory

Create and execute kernel

Copy result (c) from device

Cleanup

- Είναι ελαφρά τροποποιημένη C
- Έχει μέσα τους kernels (συναρτήσεις `__global__`) και τον κώδικα που θα εκτελέσει ο host.
- Δέσμευση χώρου στη συσκευή (`cudaMalloc`), μεταφορά δεδομένων στη συσκευή (`cudaMemcpy`)
- Offloading (`saxpy<<< … >>>`)
- Μεταφορά αποτελεσμάτων από τη συσκευή

- Μικρότερος κώδικας από την OpenCL (η οποία προσπαθεί να είναι εντελώς γενική και να δουλεύει όχι μόνο για GPUs)
  - αν όμως υπάρχουν > 1 GPUs, χρειάζεται κώδικας να επιλεχθεί η συσκευή αλα-OpenCL
- Compile με *nvcc* (NVidia compiler)
  - παράγεται ένα κομμάτι που εκτελεί ο host και οι kernels που εκτελεί η GPU.

CUDA – προγραμματιστικό μοντέλο

- Ο kernel εκτελείται από (πολλά) **CUDA threads**
  - Όλα τα νήματα εκτελούν τον ίδιο kernel
- Τα νήματα ομαδοποιούνται σε λογικά **blocks**
- Τα blocks ομαδοποιούνται σε ένα λογικό **grid**
- Επομένως, ένας kernel τελικά εκτελείται *από ένα grid από blocks από threads*.

- Το πλήθος των block που θα έχει το grid (1024), και το πλήθος των threads που θα έχει κάθε block (256) δίνονται ως παράμετροι στο offloading:

```
saxpy<<<1024, 256>>>(N, a, d_b, d_c);
```

- Το πλήθος των νημάτων ανά block πρέπει να είναι ≤ 1024.
- Με βάση την αρχιτεκτονική των GPU της NVidia και για λόγους επιδόσεων, καλό είναι *το πλήθος των νημάτων σε κάθε block να είναι πολλαπλάσιο του 32.*

CUDA – προγραμματιστικό μοντέλο

- Το πλήθος των block που θα έχει το grid (1024), και το πλήθος των threads που θα έχει κάθε block (256) δίνονται ως παράμετροι στο offloading:

```
saxpy<<<1024, 256>>>(N, a, d_b, d_c);
```

- Γενικά, επιτρέπεται η οργάνωση των threads και των block σε 1D-3D σχήματα, δηλαδή, π.χ. τα 256 νήματα του block θα μπορούσαν να είναι ένα 3-διάστατο σύνολο από 8x8x4 νήματα και τα blocks να είναι σύνολο από 16x8x8 blocks ως εξής:

```
dim3 nblk(16,8,8), nthr(8,8,4);
saxpy<<<nblk, nthr>>>(N, a, d_b, d_c);
```

- Παρότι – πάλι – 1024 blocks των 256 threads το καθένα, παίρνω πλέον τα id τους μέσα από μία τριπλέτα, π.χ. το νήμα 11 είναι το (0,2,3).
- Το αν είναι οργανωμένα σε 1D / 2D / 3D *είναι μόνο θέμα προγραμματιστικού «βολέματος» και τίποτε άλλο* (π.χ. αν έχουμε πίνακες βολεύει η 2D οργάνωση/αρίθμηση). *Δεν επηρεάζει τις επιδόσεις.*

CUDA – προγραμματιστικό μοντέλο

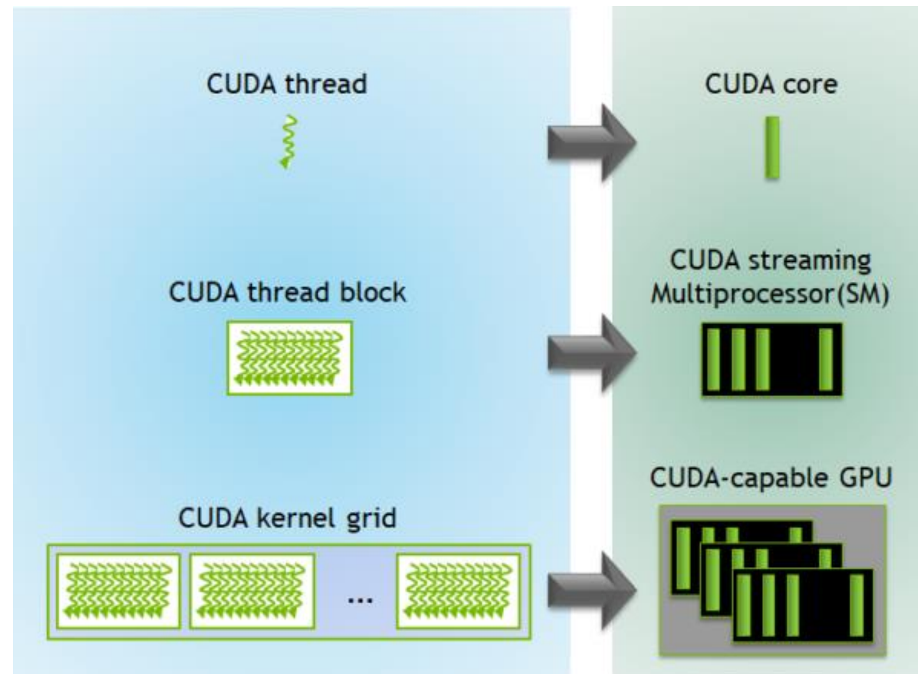
- Μέσα σε έναν kernel υπάρχει πρόσβαση στις εξής μεταβλητές, οι οποίες είναι όλες 3D:
  - gridDim – πόσα blocks υπάρχουν στην κάθε διάσταση (x/y/z) του grid
  - blockDim – πόσα threads υπάρχουν στην κάθε διάσταση (x/y/z) του block
  - blockIdx – index του block μέσα στο grid
  - threadIdx – index του thread μέσα στο block

```
__global__ void saxpy(int n, float a, float *b, float *c)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;   /* 1D */
  if (i < n) c[i] = a*b[i] + c[i];
}
```

- Έτσι, π.χ. στη γενική περίπτωση αν θέλαμε να βρούμε το (global) thread ID ενός νήματος όταν ο kernel κλήθηκε με 3D σχήμα, θα είχαμε:

```
int globalThreadID = (threadIdx.z*blockDim.y + threadIdx.y)*blockDim.x
                        + threadIdx.x;
```

CUDA – μοντέλο εκτέλεσης



- Ένα CUDA *thread* εκτελείται από ένα *core*
- Ένα προγραμματιστικό *block* ανατίθεται σε ένα *SM* για εκτέλεση.
- Τα SMs εκτελούν τα νήματα του block ανά 32άδες ("warps").
- Μέσα σε ένα warp τα νήματα εκτελούνται συγχρονισμένα (SIMD εκτέλεση)
  - Προσοχή στα branches!! (*επόμενη διαφάνεια*)

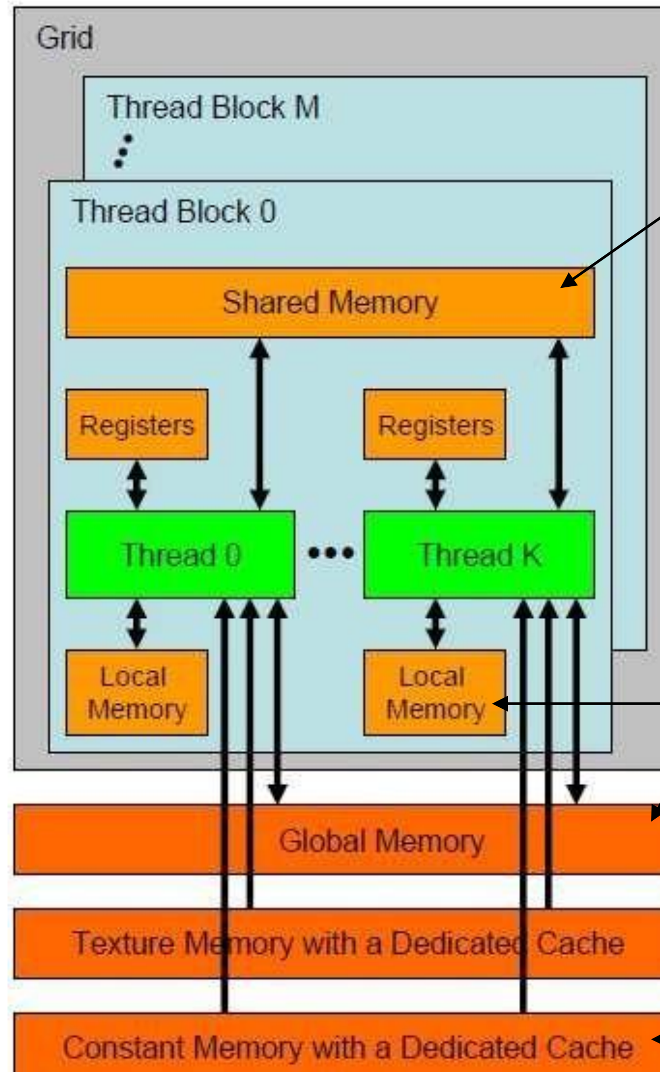
CUDA – μοντέλο εκτέλεσης

- Αφού όλα τα cores εκτελούν την ίδια εντολή (SIMD), πώς εκτελείται ο παρακάτω κώδικας στην GPU;

```
...                     /* Κώδικας μέσα στον kernel */
if (<condition>)
   <A>
else
   <B>
...
```

- Απάντηση:
  - Δυστυχώς **σειριοποιούνται** οι περιοχές <A> και <B>
  - Όσα cores στο <condition> βγάζουν FALSE **απενεργοποιούνται** – και όλα τα άλλα εκτελούν το <A>
  - Στην συνέχεια **απενεργοποιούνται** όσα cores έχουν <condition> TRUE – και όλα τα άλλα εκτελούν το <B>
  - Άρα *μείωση των επιδόσεων* – πρέπει να αποφεύγονται τα πολλά branches και όταν συμβαίνουν να είναι έτσι διαμορφωμένα ώστε να μην συμβαίνουν συχνά μέσα στο ίδιο warp (αποφυγή "**warp divergence**").

Μοντέλο μνήμης για τους kernels



Παρέχεται από την shared memory του SM. Π.χ. δήλωση:

```
__shared__ int x;
```

Παρέχεται από την device memory της κάρτας. Εκεί μπαίνουν π.χ. τα ορίσματα του kernel. Π.χ. δήλωση:

```
__device__ int x;
```

Παρέχεται από την device memory της κάρτας επίσης (=> αργή!!!).

Εκεί μπαίνουν αυτόματα τοπικές μεταβλητές ενός νήματος που δεν χωρούν στους registers.

Παρέχεται από την device memory του SM (όμως cached). Π.χ. δήλωση:

```
__constant__ int x;
```

Παρένθεση: ορολογία OpenCL

CUDA	OpenCL
Thread	Work item
Block	Work group
Grid	NDRange
Thread index	Local ID
Thread ID	Global ID
Warp (32)	Wavefront (32/64 για AMD, 8/16/32 για Intel)

GPUs: προγραμματισμός

Higher-level: OpenMP

Με OpenMP

- Λίγο πιο απλά ☺

```
void saxpy(int n, float a, float *b, float *c)
{
  #pragma omp target map(to: a,b[0:n]) map(tofrom: c[0:n])
  for (int i = 0; i < n; ++i)
    c[i] = a*b[i] + c[i];
}
```

- Και μπορώ να χρησιμοποιήσω και τα πολλά cores των GPU:

```
void saxpy(int n, float a, float *b, float *c)
{
  #pragma omp target map(to: a,b[0:n]) map(tofrom: c[0:n])
    #pragma omp parallel for
    for (int i = 0; i < n; ++i)
      c[i] = a*b[i] + c[i];
}
```

map

- Για αντιστοίχιση/μεταφορά των δεδομένων μεταξύ host και device
  - `map(to:x,y)` --> *ο host μόνο στέλνει στη συσκευή τα x,y*
  - `map(from:x,y)` --> *ο host μόνο λαμβάνει από τη συσκευή τα x,y*
  - `map(tofrom:x,y)` --> *και τα δύο*
  - `map(alloc:x,y)` --> *τα x,y δεν στέλνονται και δεν λαμβάνονται*
- Τα δεδομένα μεταφέρονται πριν «κατέβει» ο κώδικας ("offloading" στη συσκευή) αλλά και μετά το πέρας της εκτέλεσης
- Αν η συσκευή και ο host έχουν κοινή μνήμη, πιθανώς δεν απαιτούνται μεταφορές δεδομένων

Αποφυγή πολλαπλών μεταφορών

- Αν έχουμε πολλαπλούς kernels και κάποια δεδομένα που χειρίζονται είναι κοινά, μπορούμε να αποφύγουμε τις πολλαπλές μεταφορές με την οδηγία `target data`

```
#pragma omp target data map(to: a,b[0:n])
{
  printf("lauching 1st kernel\n");  /* Host code */
  #pragma omp target map(tofrom: c[0:n])
  for (int i = 0; i < n; ++i)
    c[i] = a*b[i] + c[i];

  printf("lauching 2nd kernel\n");  /* Host code */
  /* a and b are already there! */
  #pragma omp target map(tofrom: d[0:n])
  for (int i = 0; i < n; ++i)
    d[i] = a*b[i] + d[i];
}
```

Άλλες λειτουργίες

- Οδηγία `declare target`
  - Ορίζει global μεταβλητές και συναρτήσεις που πρέπει να υπάρχουν και στη συσκευή
- Οδηγία `target update`
  - Ο host μπορεί να ενημερώνει ρητά κάποιες μεταβλητές από/προς τη συσκευή, εκτός περιοχών `target`.
- Οδηγία `teams`
  - Χωρίζει τα cores της συσκευής σε ομάδες και θέτει 1 σε κάθε ομάδα ως αρχηγό
- Οδηγία `distribute`
  - Μοιράζει τις επαναλήψεις ενός for στους αρχηγούς των ομάδων
  - Αν μετά ακολουθεί και οδηγία parallel for, οι επαναλήψεις που πέφτουν σε κάθε ομάδα, εκτελούνται παράλληλα από όλα τα νήματα της ομάδας
- κλπ

Updates

- Αν έχουμε πολλαπλούς kernels και κάποια δεδομένα που χειρίζονται είναι κοινά, μπορούμε να αποφύγουμε τις πολλαπλές μεταφορές με την οδηγία `target data`

```
#pragma omp target data map(to: a,b[0:n])
{
  #pragma omp target map(tofrom: c[0:n])
  for (int i = 0; i < n; ++i)
    c[i] = a*b[i] + c[i];

  a *= 2;                                /* larger a */
  #pragma omp target update to(a)   /* send new value to device */

  /* a and b are already there! */
  #pragma omp target map(tofrom: d[0:n])
  for (int i = 0; i < n; ++i)
    d[i] = a*b[i] + d[i];
}
```

target teams distribute parallel for

- Όταν στοχεύουμε σε συσκευές GPU, για τη διαμοίραση επαναλήψεων συνηθίζεται η χρήση της οδηγίας `target teams distribute parallel for`
- Η υπο-οδηγία `target teams distribute` αρχικά θα δημιουργήσει έναν αριθμό από ομάδες και θα διαμοιράσει τις επαναλήψεις στους αρχηγούς
  - Οι ομάδες ισοδυναμούν με τα “CUDA blocks”
- Η υπο-οδηγία `parallel for` θα δημιουργήσει ένα πλήθος νημάτων ανά ομάδα και τα νήματα θα εκτελέσουν παράλληλα τις επαναλήψεις των αρχηγών τους
  - Τα νήματα των ομάδων ισοδυναμούν με τα νήματα των CUDA blocks

Μεμονωμένες, εμφωλευμένες οδηγίες

```
#pragma omp target map(to: a,b[0:n]) map(tofrom: c[0:n])
{
  #pragma omp teams
  {
    #pragma omp distribute
    {
      #pragma omp parallel for
      for (int i = 0; i < n; ++i)
         c[i] = a*b[i] + c[i];
    }
  }
}
```

Συνδυασμένη οδηγία
α) Ομάδες - νήματα

Δημιουργεί ένα πλήθος CUDA blocks

Δημιουργεί ένα πλήθος CUDA threads

```
#pragma omp target teams distribute parallel for \
  map(to: a,b[0:n]) map(tofrom: c[0:n])
  for (int i = 0; i < n; ++i)
    c[i] = a*b[i] + c[i];
```

Συνδυασμένη οδηγία
β) Επαναλήψεις

1ος διαμοιρασμός: Αρχηγοί των blocks

2ος διαμοιρασμός: Νήματα του κάθε block

```
#pragma omp target teams distribute parallel for \
  map(to: a,b[0:n]) map(tofrom: c[0:n])
  for (int i = 0; i < n; ++i)
    c[i] = a*b[i] + c[i];
```

Compilers και high-level offloading

- Υποστήριξη από GCC, Clang/LLVM
  - Πρέπει να έχει γίνει build ο compiler με ειδικά flags (οι συνήθεις εγκαταστάσεις τους δεν παρέχουν offloading)
  - Απαιτούνται ειδικά flags κατά τη μετάφραση του προγράμματος του χρήστη
    - GCC: `-fopenmp -foffload=<target> -foffload-options=…`
      (target = `nvptx-none` ή `amdgcn-amdhsa`)
    - Clang: : `-fopenmp -fopenmp-targets=<target>` …
      (target = `nvptx64-nvidia-cuda` ή `amdgcn-amd-amdhsa`)
  - Για NVIDIA GPUs και πιο πρόσφατα για AMD GPUs
- Ο OMPi παρέχει ενσωματωμένη υποστήριξη NVIDIA GPUs (AMD οσονούπω)
- Πρόσφατα ο νέος μεταφραστής *nvc* (όχι ο *nvcc*) της NVIDIA υποστηρίζει OpenMP offloading σε νεότερες GPUs της
- Υποστήριξη **OpenACC** – κι αυτό με οδηγίες #pragma παρόμοιες με το OpenMP όμως όχι τόσο γενικό όσο το OpenMP, με πλήρη «εξειδίκευση» σε GPUs και όχι τόσο για την CPU.
  - Πιο απλές οδηγίες για offloading
  - Προσφέρει ευκολίες πλήρως αυτόματης παραλληλοποίησης κάποιων loops
  - Υποστηρίζεται βασικά από κάποιες εταιρείες (κυρίως NVIDIA)