Παράλληλος Προγραμματισμός σε Επεξεργαστές Γραφικών

Συστήματα Παράλληλης Επεξεργασίας
9ο εξάμηνο, ΣΗΜΜΥ

Εργαστήριο Υπολογιστικών Συστημάτων
(CSLab)

National Technical University of Athens
CSLab

Ιανουάριος 2020

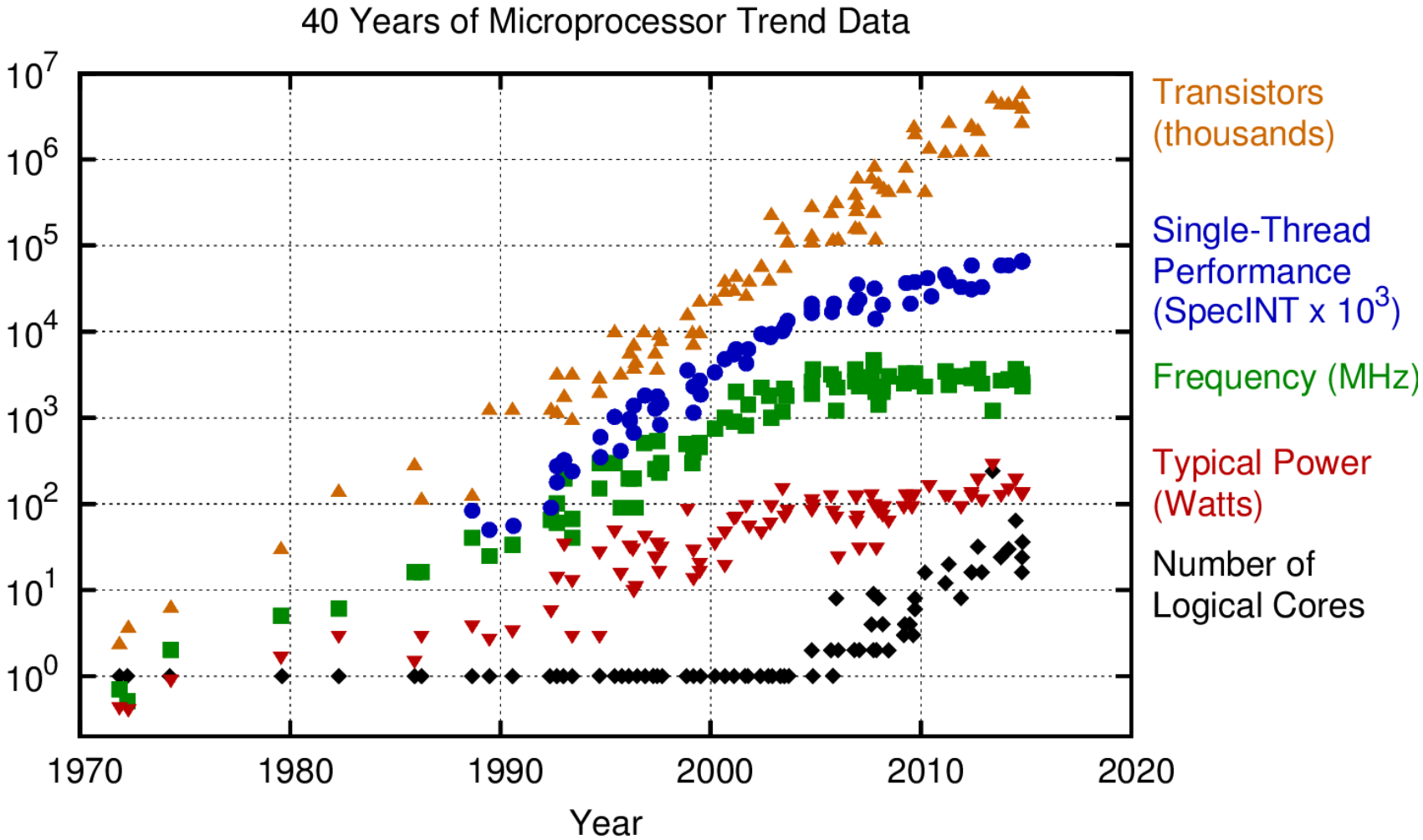
1. Εισαγωγικά

2. GPUs
   - Αρχιτεκτονική GPUs
   - Προγραμματισμός GPUs

3. Παράλληλος προγραμματισμός σε CUDA GPUs
   - CUDA parallel computing platform
   - CUDA programming model
   - CUDA C/C++
   - Παράδειγμα 1: άθροισμα διανυσμάτων
   - Ζητήματα επίδοσης
   - Παράδειγμα 2: εσωτερικό γινόμενο διανυσμάτων

4. Χρήσιμα links

Moore's law



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

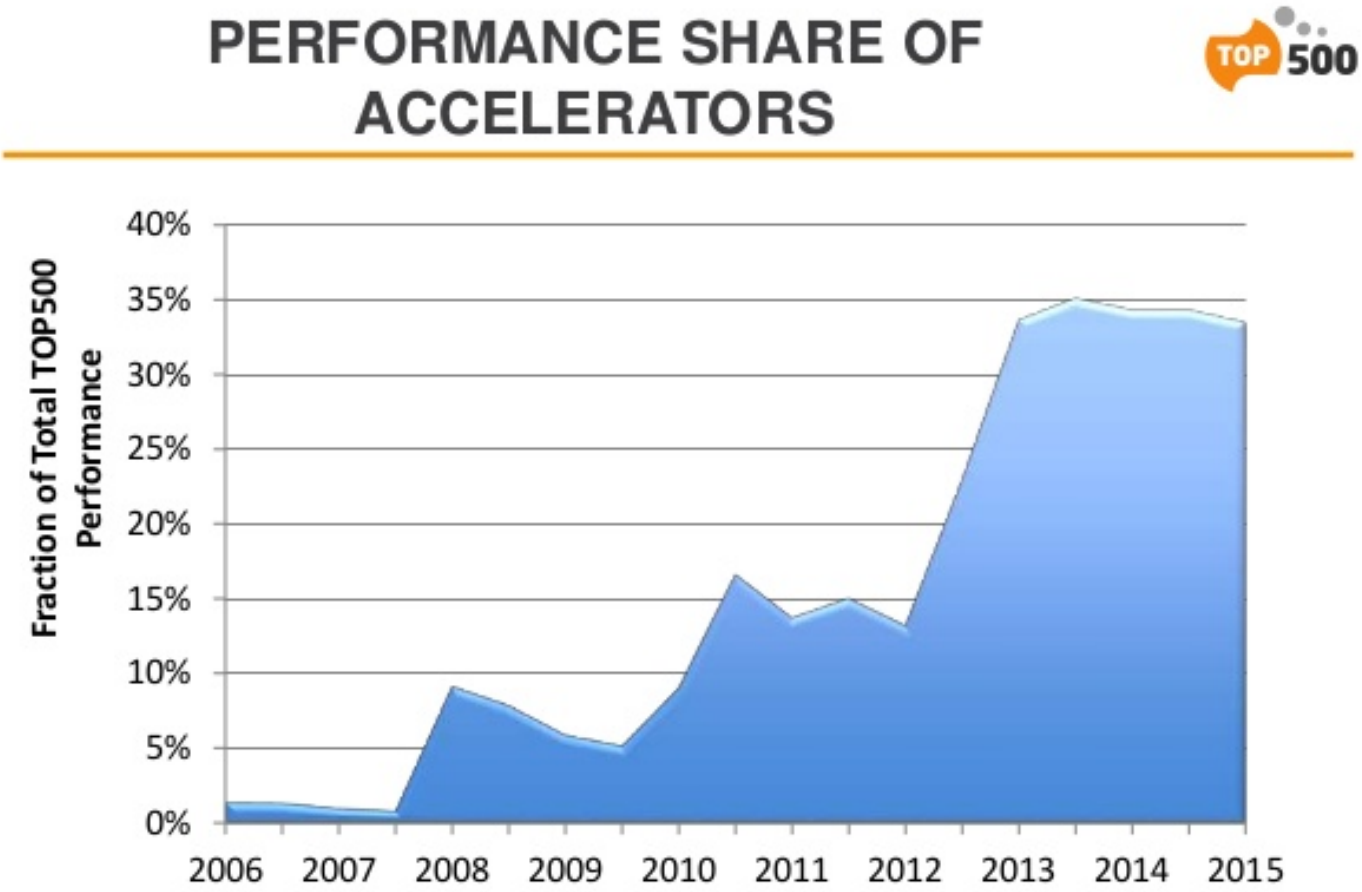
The end of Moore's law (?)

- Υπάρχουν φυσικοί περιορισμοί στο πόσα τρανζίστορ μπορούμε να χωρέσουμε σε ένα ολοκληρωμένο κύκλωμα
- Η περαιτέρω μείωση των διαστάσεων των τρανζίστορ αυξάνει σε κόστος καθώς πλησιάζουμε τα φυσικά όρια της εκάστοτε τεχνολογίας ημιαγωγού (πυρίτιο)
- Περαιτέρω βελτίωση της επίδοσης μπορεί να έρθει μέσα από επεξεργαστές ειδικού σκοπού
- Άκρως ετερογενή υπολογιστικά συστήματα

Επιταχυντές

- Αρχιτεκτονικές εξειδικευμένες στην επιτάχυνση κάποιας οικογένειας εφαρμογών
  - Κρυπτογραφία
  - Γραφικά
  - Όραση υπολογιστών
  - Κρυπτο-νομίσματα
  - Internet of Things
  - Μηχανική μάθηση
  - ...
- Συνήθως χρησιμοποιούνται ως συνεπεξεργαστές σε κλασσικές CPUs
  - Η χρήση τους μπορεί να είναι διαφανής
  - Ή να απαιτεί αλλαγή των προγραμμάτων

Επιταχυντές στο TOP500



Προκλήσεις

- **Applicability**: είναι ο αλγόριθμός μου κατάλληλος για τον επιταχυντή;
- **Programmability**: πόσο δύσκολο είναι να προγραμματίσω τον επιταχυντή;
- **Portability**: μεταφέρεται εύκολα ο αλγόριθμος μου σε διαφορετικούς επιταχυντές;
- **Availability**: υπάρχουν επιταχυντές στη διάθεσή μου;
- **Scalability**: κλιμακώνεται εύκολα ο αλγόριθμός μου σε πολλούς επιταχυντές;

GPUs

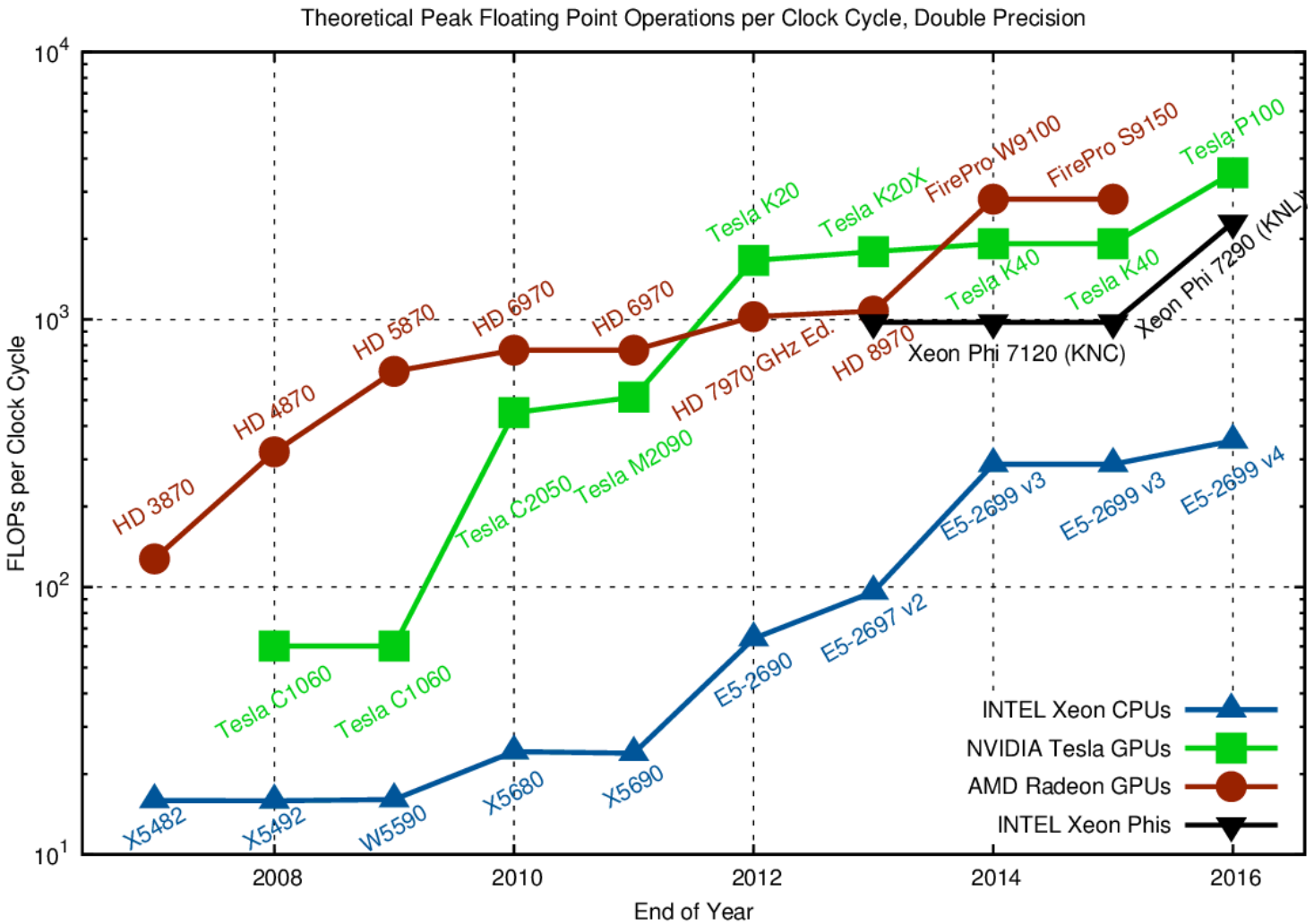
- Επιταχυντές για την επεξεργασία γραφικών (**G**raphics **P**rocessing **U**nits, **GPU**s)
  - Embarrassingly parallel εφαρμογές (visualization, gaming, κ.λπ.)
  - Αρχικά, σχεδιάστηκαν μόνο για γραφικά (hardwired logic)
- Επωφελήθηκαν από το νόμο του Moore και εξελίχθηκαν σε πιο ευέλικτους και προγραμματίσιμους επεξεργαστές
- Οι σύχρονες GPUs μπορούν να εκτελέσουν ένα μεγάλο εύρος υπολογιστικά απαιτητικών εφαρμογών (General Purpose GPUs)
  - Υψηλός παραλληλισμός
  - Πολλά δεδομένα
  - Πολλοί υπολογισμοί ανά δεδομένο (arithmetic intensity)

GPUs στο TOP500

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	DOE/SC/Oak Ridge National Laboratory United States	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM	2,397,824	143,500.0	200,794.9	9,783
2	DOE/NNSA/LLNL United States	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM / NVIDIA / Mellanox	1,572,480	94,640.0	125,712.0	7,438
3	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
4	National Super Computer Center in Guangzhou China	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 NUDT	4,981,760	61,444.5	100,678.7	18,482
5	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 Cray Inc.	387,872	21,230.0	27,154.3	2,384

Επίδοση

CPUs vs GPUs - Υπολογισμοί διπλής ακρίβειας



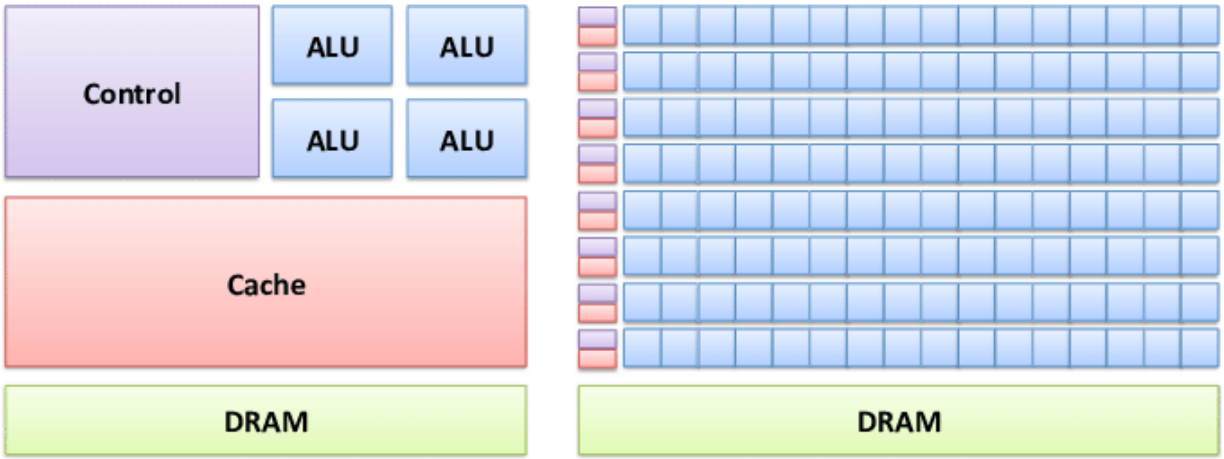
Αρχιτεκτονική

Φιλοσοφία σχεδίασης

- Επιτάχυνση data-parallel compute-intensive εφαρμογών
- Βασικές σχεδιαστικές επιλογές
  - Υψηλός αριθμός πυρήνων
  - Fine-grained hardware multithreading (γιατί;)
  - Thread scheduling logic (γιατί;)

Αρχιτεκτονική

CPUs vs GPUs



- CPU
  - Λίγοι και σύνθετοι υπολογιστικοί πυρήνες
  - Βαθιά ιεραρχία κρυφής μνήμης
  - Κύρια μνήμη μεγάλης χωρητικότητας και χαμηλού εύρους ζώνης
  - Διαχείρηση νημάτων εκτέλεσης από το λειτουργικό σύστημα
- GPU
  - Περισσότεροι και απλοί υπολογιστικοί πυρήνες
  - Λίγα επίπεδα κρυφής μνήμης
  - Κύρια μνήμη περιορισμένης χωρητικότητας και υψηλού εύρους ζώνης
  - Διαχείρηση νημάτων εκτέλεσης από το υλικό

Flynn's Taxonomy

CPUs vs GPUs

CPUs

MIMD + SIMD

- SIMD implementation:
  - "explicit" με επεκτάσεις του ISA (π.χ. MMX, SSE, AVX extensions για x86)
  - ο προγραμματιστής (ή ο μεταγλωττιστής) προσαρμόζει ρητά τον κώδικα

GPUs

SIMD

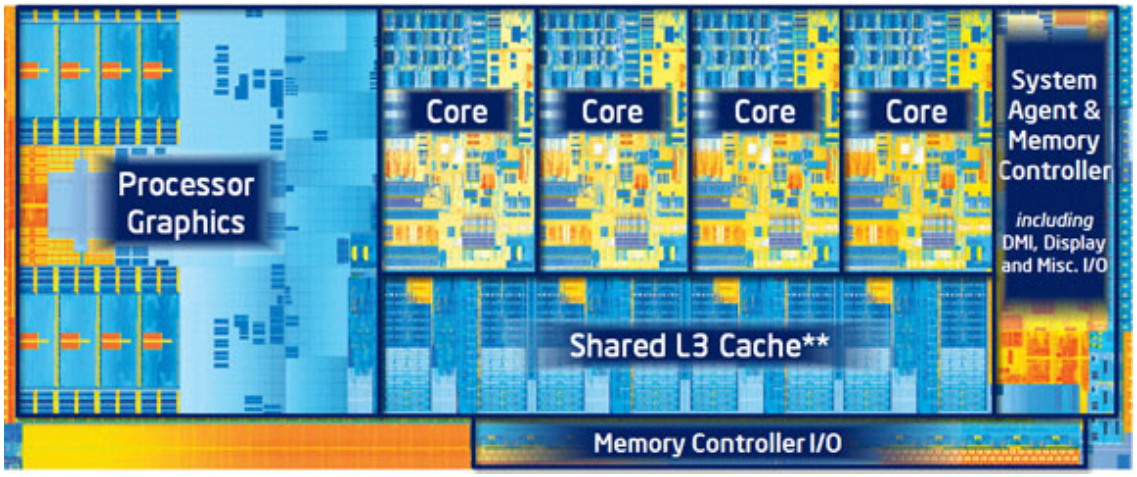
- SIMD implementation:
  - "implicit" με διαχείριση από το υλικό
  - ο προγραμματιστής (ή ο μεταγλωττιστής) παράγει σειριακό κώδικα

Είδη GPUs

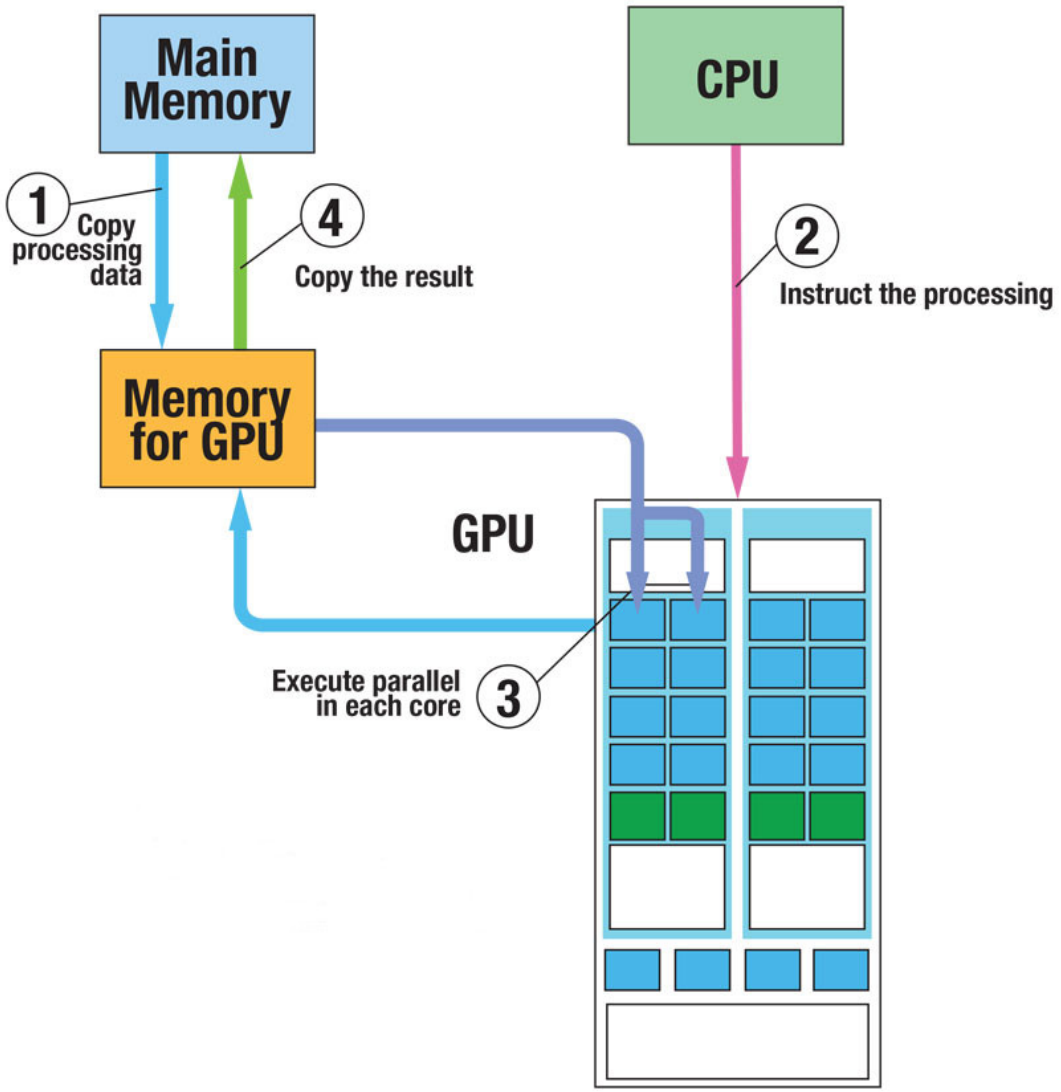
- Διακριτές: συνδέονται με τη CPU μέσω καναλιού
  - Δική τους μνήμη
    - Μεγαλύτερο εύρος ζώνης
    - Περιορισμένη χωρητικότητα
  - Περισσότερη υπολογιστική δύναμη
  - Υψηλότερη κατανάλωση ισχύος

Είδη GPUs

- Ολοκληρωμένες στο ίδιο τσιπ με τη CPU
  - Μοιράζονται την ίδια μνήμη με τη CPU
    - Μικρότερο εύρος ζώνης
    - Μεγαλύτερη χωρητικότητα
    - Ανταγωνισμός για τη μνήμη
  - Λιγότερη υπολογιστική δύναμη
  - Μικρότερη κατανάλωση ισχύος



Εκτέλεση υπολογισμών σε διακριτές GPUs

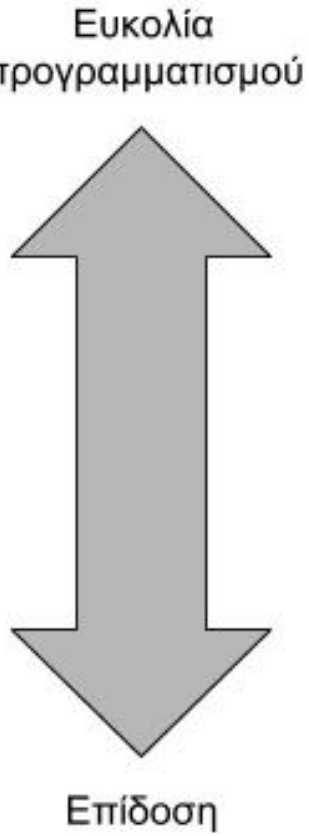


Βασικά βήματα:

1. Αντιγράφουμε τα δεδομένα που θέλουμε να επεξεργαστούμε από τη μνήμη της CPU στη μνήμη της GPU
2. Η CPU αναθέτει τον υπολογισμό στην GPU
3. Ο υπολογισμός εκτελείται στην GPU
4. Αντιγραφή των αποτελεσμάτων του υπολογισμού από τη μνήμη της GPU στη μνήμη της CPU

Πώς χρησιμοποιώ μία GPU;

- Με έτοιμες εφαρμογές που υποστηρίζουν GPUs
  - TensorFlow, PyTorch
- Με βιβλιοθήκες βελτιστοποιημένες για GPUs
  - cuBLAS, cuSPARSE, cuDNN
- Με διεπαφές (APIs) που βασίζονται σε οδηγίες (directives) προς το μεταγγλωτιστή:
  - OpenACC
  - OpenMP
- Με επέκταση γλώσσας προγραμματισμού:
  - OpenCL (C/C++)
  - **NVIDIA CUDA** (C/C++, Fortran)

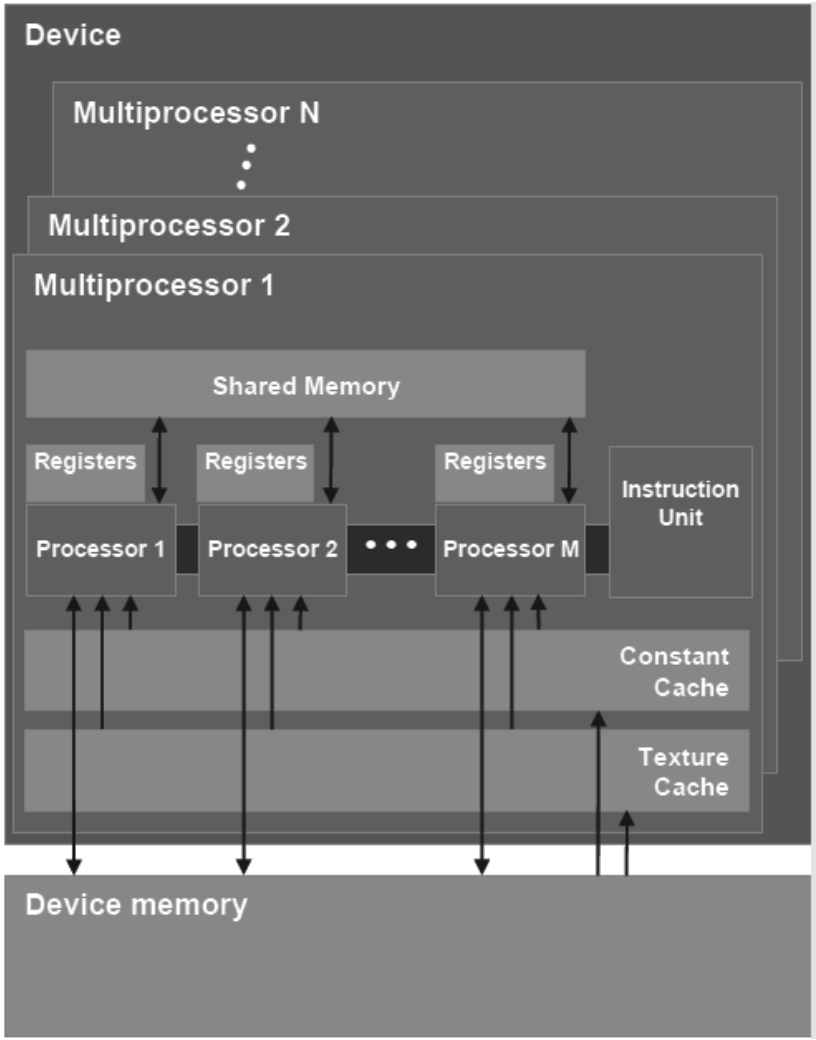


Compute Unified Device Architecture (CUDA)

- Parallel computing platform + Application programming interface
- Αναπτύχθηκε από την NVIDIA ειδικά για GPGPU computing

CUDA parallel computing platform

Γενική αρχιτεκτονική μίας NVIDIA GPU



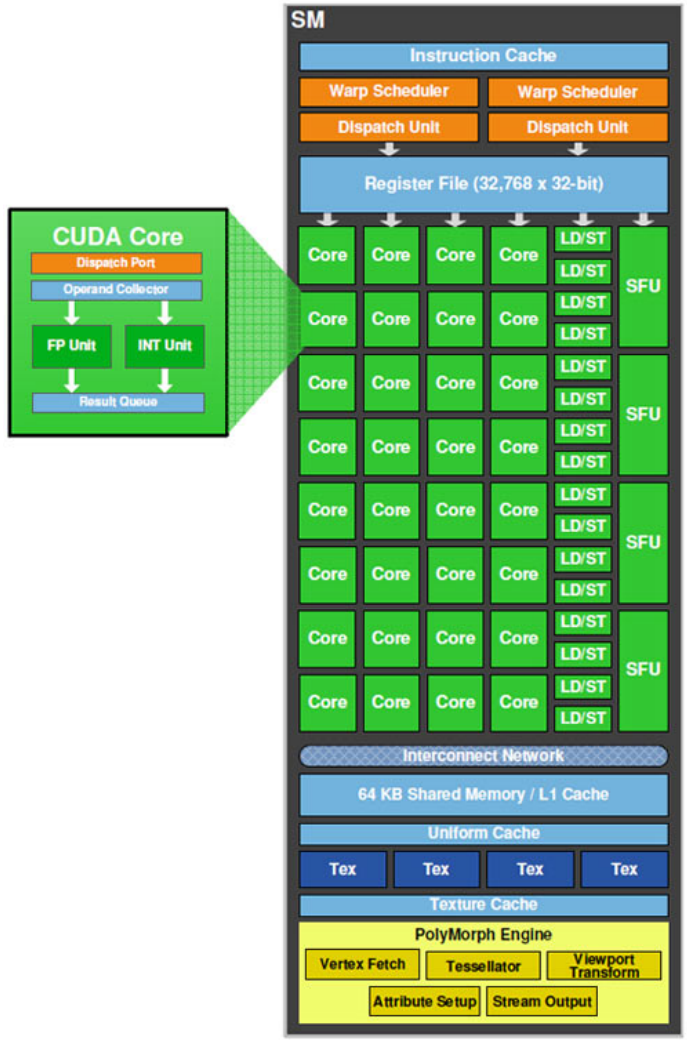
- Κάθε συσκευή GPU αποτελείται από συστοιχίες πολυεπεξεργαστών ροών (**Streaming Multiprocessors – SMs**)
- Κάθε πολυεπεξεργαστής ροών αποτελείται από:
  - Επεξεργαστές ροών (**Streaming Processors – SPs** ή αλλιώς **CUDA cores**)
  - Πολύ μεγάλο αρχείο καταχωρητών (8K – 32K)
  - Shared memory
  - Constant cache (read-only)
  - Texture cache
  - ...

Πηγή: NVIDIA CUDA Programming Guide

CUDA parallel computing platform

Βασική δομική μονάδα: Streaming Multiprocessor (SM)

- Κάθε SM αναλαμβάνει να εκτελέσει ένα ή περισσότερα μπλοκ από νήματα (thread blocks)
- Όλα τα νήματα ενός thread block μοιράζονται τους πόρους του SM
- Τα νήματα οργανώνονται περαιτέρω σε ομάδες των 32 νημάτων (warps) και δρομολογούνται από warp schedulers στα cores που το αποτελούν
- Σε κάθε κύκλο ρολογιού όλα τα νήματα ενός warp εκτελούν την ίδια εντολή
  - **S**ingle **I**nstruction **M**ultiple **T**hread (**SIMT**)
- Η χρονοδρομολόγηση των warps ενός μπλοκ έχει μηδαμινό κόστος

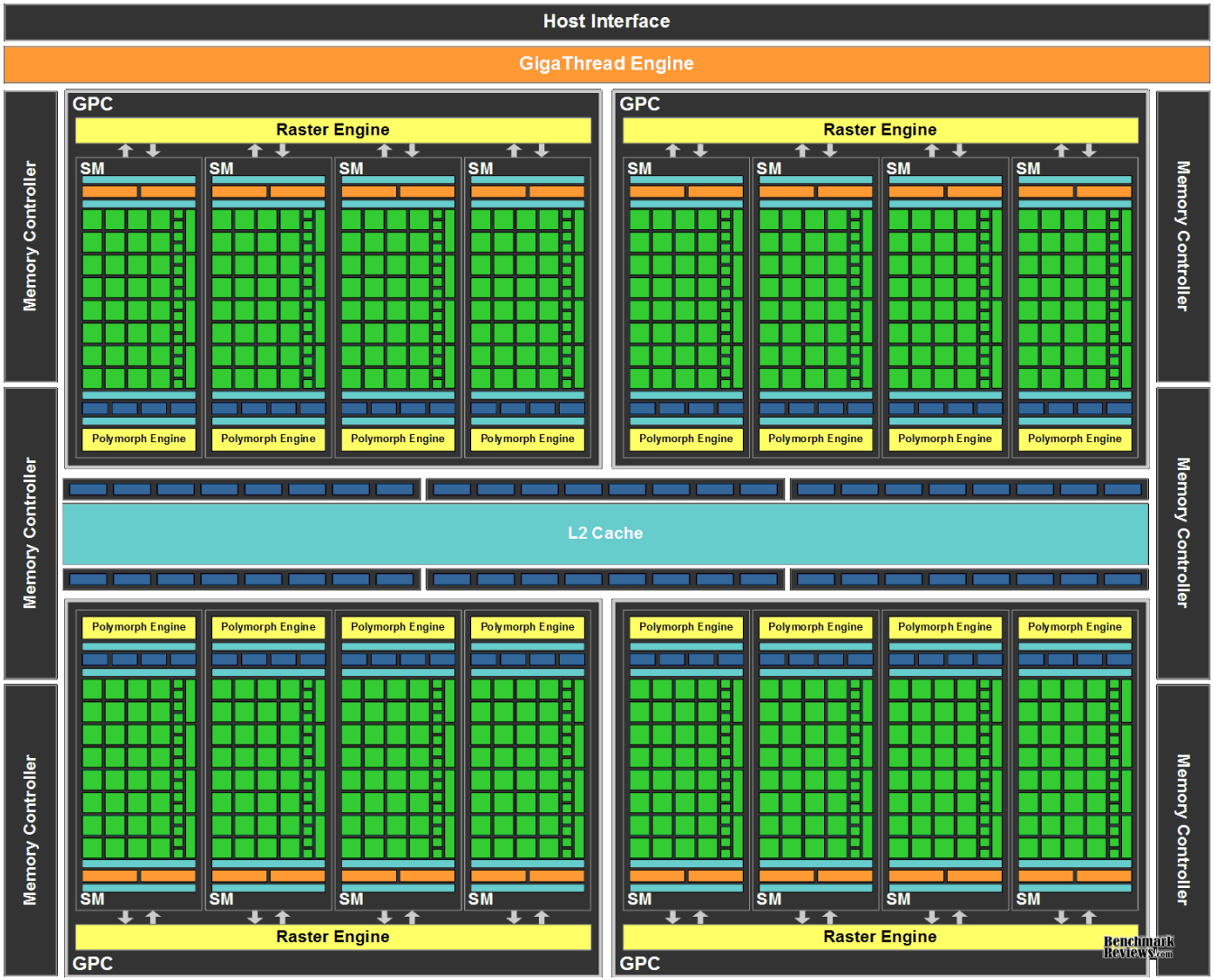


Streaming Multiprocessor (SM)

Πηγή: NVIDIA CUDA Programming Guide

CUDA parallel computing platform

Παράδειγμα: αρχιτεκτονική Fermi



Πηγή: NVIDIA CUDA Programming Guide

CUDA parallel computing platform

Δυνατότητες συσκευών

- Οι δυνατότητες κάθε συσκευής καθορίζονται από το “compute capability”
  - `<major>.<minor>`
  - Ο αριθμός major δηλώνει έκδοση της μικρο-αρχιτεκτονικής
  - Ο αριθμός minor δηλώνει βελτιώσεις της μικρο-αρχιτεκτονικής

Microarchitecture	Comp. Capability
Tesla	1.x
Fermi	2.x
Kepler	3.x
Maxwell	5.x
Pascal	6.x
Volta	7.x

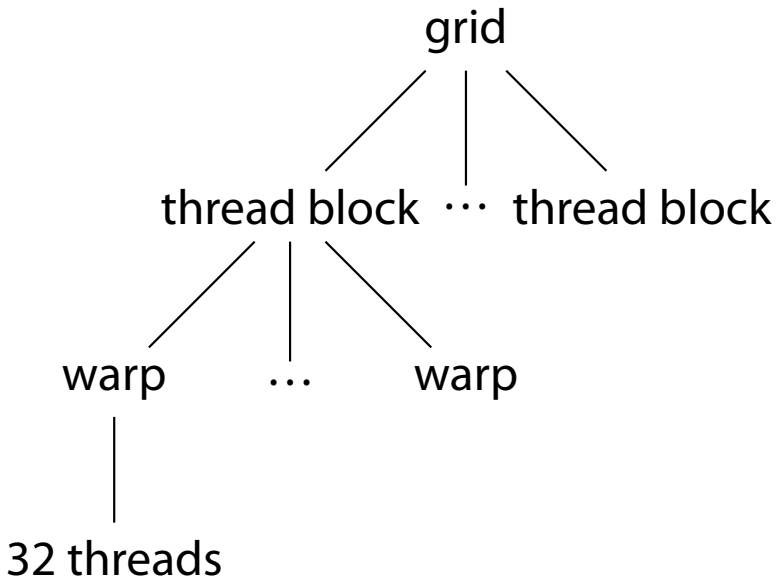
CUDA programming model

- Ετερογενές
  - Συμμετέχουν η CPU (ως host) και η GPU (ως device)
- Η εκτέλεση υπολογισμών στη GPU γίνεται μέσα από kernel launches
  - kernel = ο κώδικας που εκτελεί κάθε νήμα (σειριακός)
  - Κάθε kernel εκτελείται N φορές, όπου N ο αριθμός των νημάτων που ορίζονται από τον προγραμματιστή
  - **S**ingle **I**nstruction **M**ultiple **T**hread (**SIMT**) μοντέλο εκτέλεσης
- Κλιμακώσιμο
  - Το ίδιο εκτελέσιμο μπορεί να εκτελεστεί σε διαφορετικές κάρτες γραφικών με διαφορετικό αριθμό από SMs

CUDA programming model

Thread abstraction

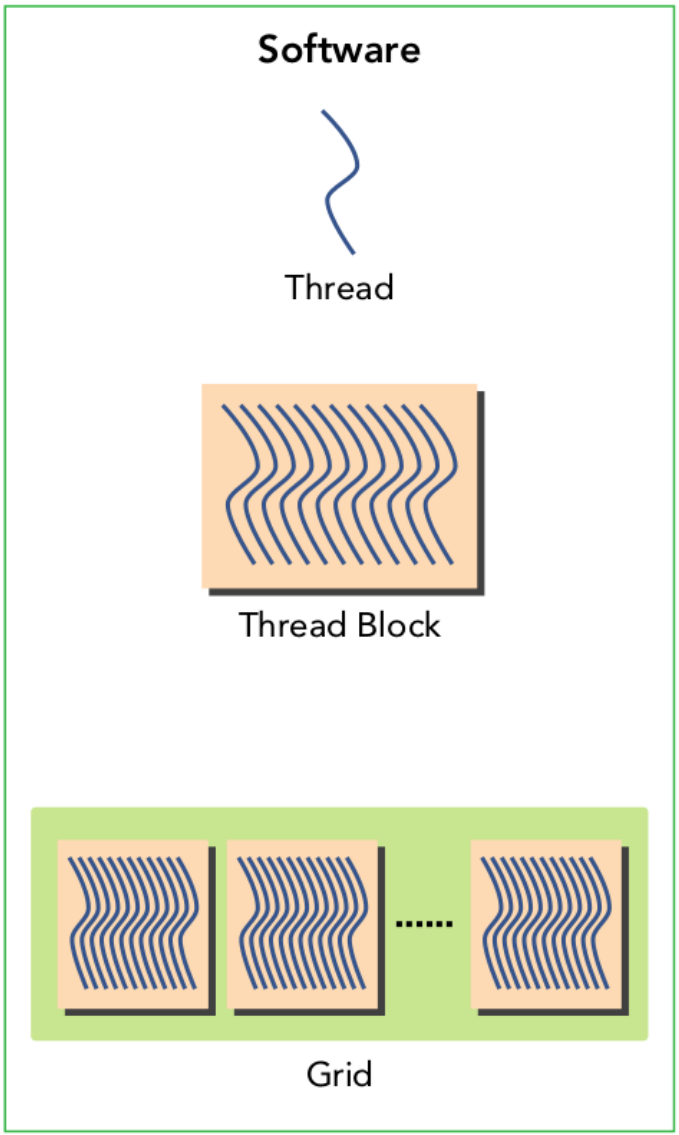
- Κάθε υπολογιστικός πυρήνας (kernel) που ανατίθεται προς εκτέλεση στην GPU οργανώνεται από τον προγραμματιστή σε ένα πλέγμα (grid) από νήματα

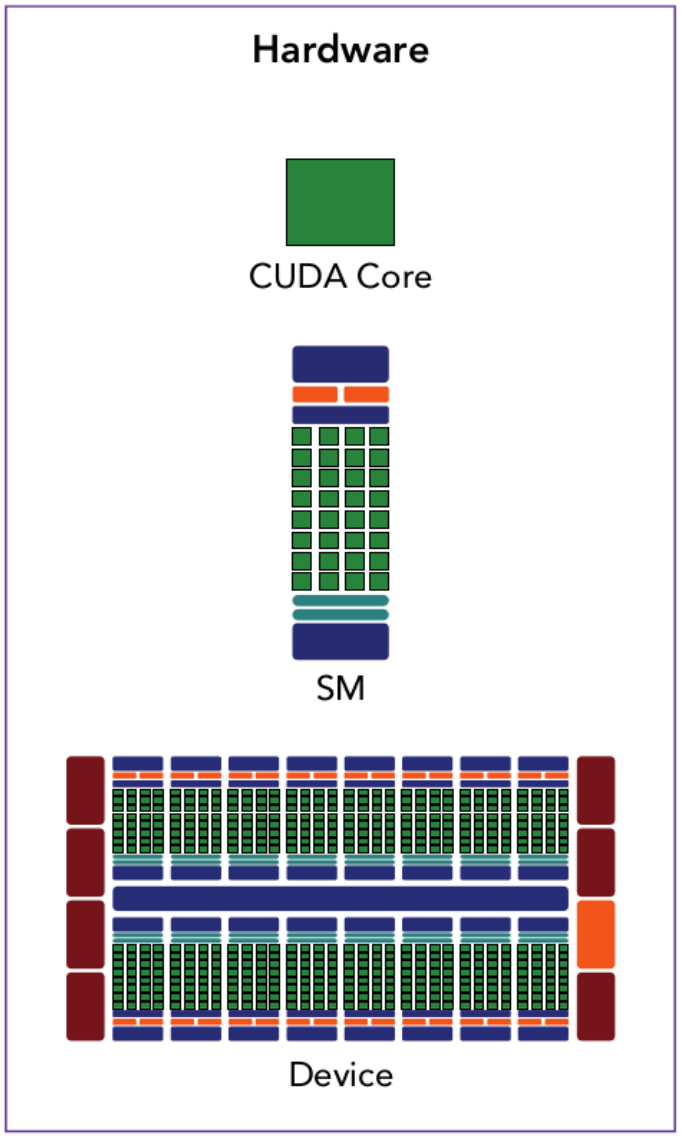


- Το grid και τα thread blocks μπορούν να είναι 1D, 2D ή 3D
- Υπάρχουν περιορισμοί στις διαστάσεις του grid και του thread block (ανάλογα με το compute capability)

CUDA programming model

Thread abstraction





CUDA programming model

Thread abstraction

- Τι αναπαριστά ένα CUDA thread;
  - ένα ανεξάρτητο νήμα εκτέλεσης
  - με δικό του program counter, καταχωρητές, processor state κ.λπ.
- Τι αναπαριστά ένα CUDA thread block;
  - ένα (data) parallel και ανεξάρτητο task
  - όλα τα thread blocks έχουν το ίδιο σημείο εισόδου, αλλά μπορούν να διαφοροποιηθούν στην εκτέλεση κώδικα
  - το πρόγραμμα οφείλει να είναι ορθό για οποιαδήποτε σειρά εκτέλεσης των threads blocks

CUDA programming model

Thread abstraction

- Σε κάθε νήμα εκτέλεσης ανατίθεται ένα μοναδικό ID εντός του thread block στο οποίο ανήκει
- Το ID κάθε νήματος προσδιορίζεται από τις «συντεταγμένες» του εντός του thread block στο οποίο ανήκει
  - Για μπλοκ (Dx, Dy), το ID του νήματος (x, y) είναι $x + yD_x$
  - Για μπλοκ (Dx, Dy, Dz), το ID του νήματος (x, y, z) είναι $x + yD_x + zD_xD_y$
- Η οργάνωση των νημάτων ενός thread block σε warps γίνεται σειριακά με βάση τα IDs των νημάτων
  - το warp 0 περιέχει τα νήματα με IDs 0-31, το warp 1 τα νήματα με IDs 32-63, κ.ο.κ.
  - αν ο αριθμός νημάτων του thread block δεν είναι πολλαπλάσιο του warp size (32) τότε το τελευταίο warp θα συμπληρωθεί με νήματα τα οποία θα είναι «μαρκαρισμένα» ως ανενεργά
- Τα IDs χρησιμοποιούνται συνήθως για την ανάθεση δεδομένων σε νήματα

CUDA programming model

Συγχρονισμός

- Εντός ενός thread block (ή SM):
  - Memory fences
  - Barriers
  - Atomic operations (compute capability >= 1.1)
- Καθολικά:
  - Μεταξύ διαδοχικών κλήσεων πυρήνων
  - Atomic operations (compute capability >= 1.1)

CUDA programming model

Πρόσβαση στην ιεραρχία μνήμης

- Η ιεραρχία μνήμης εκτίθεται στο Π.Μ. και μπορεί να την χειριστεί ο προγραμματιστής

Διαμοιρασμός της ιεραρχίας	
Καθολική μνήμη	Όλα τα blocks
Μνήμη σταθερών	Όλα τα blocks
Μνήμη textures	Όλα τα blocks
Μοιραζόμενη μνήμη	Όλα τα νήματα ενός block
Register file	Όλα τα νήματα ενός block

CUDA C/C++

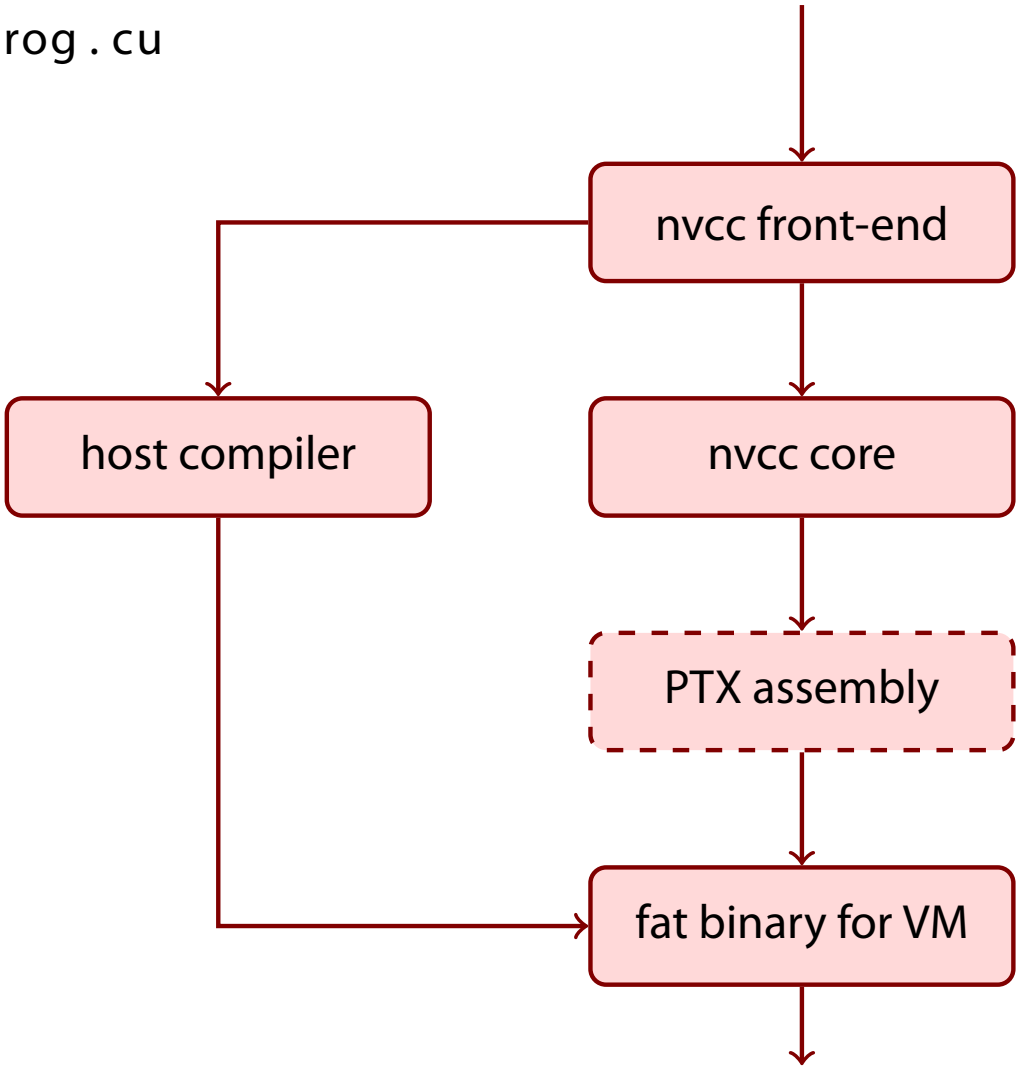
- Επέκταση της C με επιπλέον τύπους και προσδιοριστές
- Χρησιμοποιεί το front-end της C++
  - Πολυμορφισμός
  - Παράμετροι προεπιλογής (default parameters)
  - Υπερφόρτωση τελεστών
  - Χώροι ονομάτων (namespaces)
  - Πρότυπα συναρτήσεων (function templates)
- Αρχεία πηγαίου κώδικα με κατάληξη `.cu`
- Μεταγλωττιστής: `nvcc`
  - Κώδικας που πρόκειται να τρέξει στην CPU (host code)
  - Κώδικας που πρόκειται να τρέξει στην GPU (device code)
  - Χρησιμοποιεί τον μεταγλωττιστή του συστήματος για την παραγωγή κώδικα για την CPU

CUDA C/C++

Διαδικασία μεταγλώττισης και εκτέλεσης

```
$ nvcc –o my_gpu_prog my_prog.cu
```

+ Just-In-Time μεταγλώττιση του πυρήνα GPU



CUDA C/C++

Διαχωρισμός κώδικα host/device

- Με χρήση προσδιοριστών (qualifiers) συναρτήσεων
- __device__
  - Εκτελείται στην **συσκευή**
  - Μπορεί να κληθεί μόνο από κώδικα της **συσκευής**
- __host__
  - Εκτελείται στο **host**
  - Μπορεί να κληθεί μόνο από κώδικα του **host**
- __global__
  - Εκτελείται στην **συσκευή**
  - Συνήθως αναφέρεται ως GPU kernel

CUDA C/C++

Διαχωρισμός κώδικα host/device – Παράδειγμα

```
__device__ int get_thread_id()
{
    return (...);
}

__global__ void my_gpu_kernel(const float *input, float *result)
{
    int tid = get_thread_id();
    ...
}

int main()
{
    float *my_input, *my_result;
    ...
    my_gpu_kernel<<<G, B>>>(my_input, my_result);
}
```

CUDA C/C++

Προσδιοριστές μεταβλητών

- __device__
  - Βρίσκεται στην καθολική μνήμη
  - Προσβάσιμη από όλα τα blocks
- __constant__
  - Βρίσκεται στην μνήμη σταθερών
  - Προσβάσιμη από όλα τα blocks
- __shared__
  - Βρίσκεται στην μοιραζόμενη μνήμη του SM
  - Προσβάσιμη από όλα τα νήματα ενός thread block
- Π.χ., πίνακας 256 floats στην μοιραζόμενη μνήμη:

```
__shared__ float array[256];
```

CUDA C/C++

Περιβάλλον εκτέλεσης και οργάνωση νημάτων – Κλήση πυρήνων

- Κατά την κλήση ενός πυρήνα για την GPU θα πρέπει να προσδιορίζονται πάντα το μέγεθος του block και του grid
- Παράδειγμα δημιουργίας μονοδιάστατου block 32 νημάτων και αντίστοιχου grid:

```
int main(...)
{
    ...
    dim3 block(32);
    dim3 grid(N/32);
    my_gpu_kernel<<<grid, block>>>(...);
    ...
}
```

Η κλήση του πυρήνα μπορεί να αποτύχει, εάν δεν επαρκούν οι πόροι της συσκευής (registers, shared memory)

CUDA C/C++

Περιβάλλον εκτέλεσης και οργάνωση νημάτων – Προσδιορισμός νήματος

- Το ID κάθε νήματος προσδιορίζεται από τις «συντεταγμένες» του εντός του block στο οποίο ανήκει
- Η CUDA C/C++ ορίζει τις εξής μεταβλητές:
  - **uint3 threadIdx**: οι συντεταγμένες του νήματος εντός του τρέχοντος block
  - **uint3 blockIdx**: οι συντεταγμένες του block εντός του grid
  - **dim3 blockDim**: οι διαστάσεις του block
  - **dim3 gridDim**: οι διαστάσεις του grid
- Π.χ., το ID του τρέχοντος νήματος ενός μονοδιάστατου block σε ένα μονοδιάστατο grid είναι
  - local_tid = threadIdx.x , εντός του block
  - global_tid = threadIdx.x + blockDim.x∗blockIdx.x , εντός του grid

CUDA C/C++

Πρόσβαση στην ιεραρχία μνήμης

- Μνήμη σταθερών
  - Με χρήση του προσδιοριστή __constant__
- Μοιραζόμενη μνήμη
  - Με χρήση του προσδιοριστή __shared__
- Μνήμη textures
  - Με χρήση ειδικών δομών που ονομάζονται texture references
- Τοπική μνήμη
  - Με χρήση του προσδιοριστή **.local** στην PTX assembly

CUDA C/C++

Μοιραζόμενη μνήμη

- Το μέγεθος της μοιραζόμενης μνήμης μπορεί να καθοριστεί και κατά την κλήση μίας __global__ συνάρτησης
- Κώδικας πυρήνα

```
__global__ my_gpu_kernel(...)
{
        extern __shared__ float shmem_buff[];
        ...
}
```

- Κλήση του πυρήνα

```
int main()
{
        ...
        shmem_size = 32*sizeof(float);
        my_gpu_kernel<<<G,B,shmem_size>>>(...);
        ...
}
```

CUDA C/C++

Συγχρονισμός

- Memory fences
  - Εγγυάται ότι όλες οι προσβάσεις στην μνήμη (global ή shared) μέχρι αυτό το σημείο έχουν πραγματοποιηθεί
  - __threadfence(), __threadfence_block()
- Barriers
  - Εγγυάται ότι όλα τα νήματα του block έχουν φτάσει σε κάποιο σημείο και όλες οι προσβάσεις που έχουν γίνει στην μνήμη (global ή shared) έχουν πραγματοποιηθεί
  - __syncthreads()
- Ατομικές συναρτήσεις
  - Αναφορές στην μνήμη (global, shared)
  - atomicAdd(), atomicInc() κ.ο.κ.

Καθολικός συγχρονισμός

Μόνο μεταξύ διαδοχικών κλήσεων του πυρήνα

CUDA C/C++

Runtime Environment

- Συναρτήσεις διαχείρισης μνήμης
  - cudaMalloc(), cudaFree()
  - cudaMallocHost(), cudaFreeHost()
- Συναρτήσεις επικοινωνίας με την συσκευή
  - cudaMemcpy()
- Συναρτήσεις διαχείρισης σφαλμάτων
  - cudaGetLastError ()
- Συναρτήσεις διαχείρισης νημάτων
  - cudaThreadSynchronize(), cudaThreadExit ()
- Συναρτήσεις διαχείρισης της συσκευής
- Συναρτήσεις διαχείρισης γεγονότων
  - cudaEventCreate(), cudaEventDestroy(), cudaEventElapsedTime()

Μεταγλώττιση

Συνδεση με την βιβλιοθήκη cudart, −lcudart

Άθροισμα διανυσμάτων

Host code (1/2)

```
int main()
{
    float a[N], b[N], c[N];
    float *dev_a, *dev_b, *dev_c;

    // Allocate memory on the GPU
    cudaMalloc((void**)&dev_a, N*sizeof(int));
    cudaMalloc((void**)&dev_b, N*sizeof(int));
    cudaMalloc((void**)&dev_c, N*sizeof(int));

    // Initialize data on the CPU
    ...

    // Copy the arrays 'a' and 'b' to the GPU
    cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice);
    ...
```

Άθροισμα διανυσμάτων

Host code (2/2)

```
    ...

    // Launch the kernel with N/128 thread blocks
    // of 128 threads
    dim3 block(128);
    dim3 grid((N+127)/128);
    vec_add<<<grid, block>>>(dev_a, dev_b, dev_c, N);

    // Copy the array 'c' back from the GPU to the CPU
    cudaMemcpy(c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost);

    // Free the memory allocated on the GPU
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);

    return 0;
}
```

Άθροισμα διανυσμάτων

Device code – version 1

- Ορίσαμε 1Δ grid και 1Δ block (γιατί;)
- Κάθε νήμα αναλαμβάνει τον υπολογισμό ενός στοιχείου

```
__global__ void vec_add(const float *a, const float *b,
                        float *c, int N)
{
    int tid = blockDim.x*blockIdx.x + threadIdx.x;
    if (tid >= N)
        return;

    c[tid] = a[tid] + b[tid];
}
```

Τι θα συμβεί αν το N είναι αρκετά μεγάλο ώστε τα N/128 blocks να υπερβαίνουν το επιτρεπτό όριο της διάστασης του grid;

Άθροισμα διανυσμάτων

Device code – version 2

- Ορίζουμε grid με fixed διαστάσεις μέσα στα επιτρεπτά όρια (ανεξάρτητες του N)
- Αναθέτουμε σε κάθε νήμα τον υπολογισμό περισσότερων στοιχείων

```
__global__ void vec_add(const float *a, const float *b,
                        float *c, int N)
{
    // Grid-stride loop
    for (int tid = blockDim.x*blockIdx.x + threadIdx.x;
         tid < N;
         tid += blockDim.x*gridDim.x)
    {
        c[tid] = a[tid] + b[tid];
    }
}
```

Τώρα μπορούμε να προσθέσουμε διανύσματα οποιουδήποτε μήκους, περιοριζόμενοι μόνο από το μέγεθος της RAM στην GPU

Προϋποθέσεις για υψηλή επίδοση

- Συνένωση αναφορών στην καθολική μνήμη (memory access coalescing)
- Warps χωρίς διακλαδώσεις
- Εκμετάλλευση της μοιραζόμενης μνήμης
- Μεγάλος αριθμός νημάτων
- Υψηλή χρησιμοποίηση των SMs

Προϋποθέσεις για υψηλή επίδοση

Συνένωση αναφορών στην καθολική μνήμη

- Κρύβει την καθυστέρηση πρόσβασης στην καθολική μνήμη
- Η καθολική μνήμη χωρίζεται σε τμήματα (segments) των 32, 64 ή 128 byte, ευθυγραμμισμένα (aligned) στα αντίστοιχα όρια
- Η πρόσβαση στην μνήμη γίνεται με διενέργειες (transactions) ανά τμήμα
- Με την συνένωση των αναφορών στην μνήμη επιδιώκουμε με ένα transaction να μεταφέρουμε όσο το δυνατόν περισσότερα χρήσιμα δεδομένα
- Αυστηρές προϋποθέσεις στην ακολουθία των αναφορών ανάμεσα στα νήματα και στο alignment των δεδομένων

Προϋποθέσεις για υψηλή επίδοση

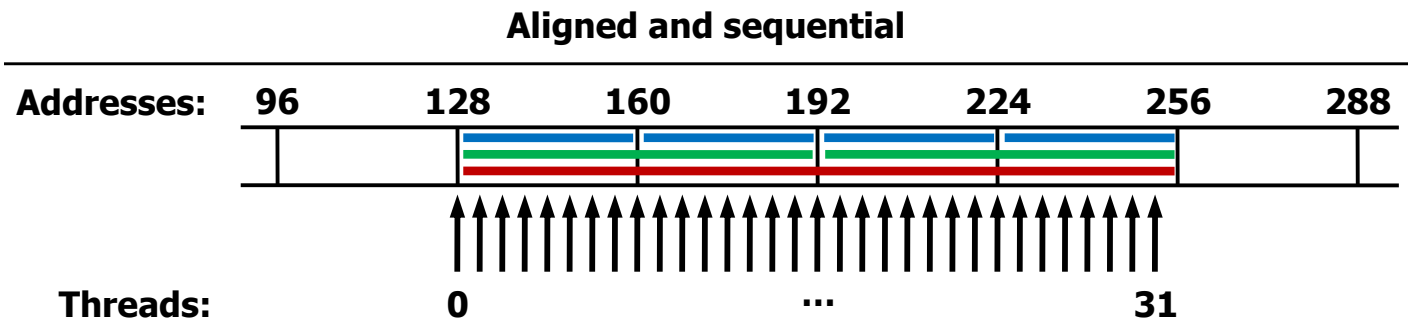
Συνένωση αναφορών στην καθολική μνήμη – Προϋποθέσεις (I)

- Τα 16 νήματα ενός half-warp θα πρέπει να προσπελαύνουν
  - είτε λέξεις 4-byte (ένα 64-byte transaction)
  - είτε λέξεις 8-byte (ένα 128-byte transaction)
  - είτε λέξεις 16-byte (δύο 128-byte transactions)
- Και οι 16 λέξεις θα πρέπει να βρίσκονται στο ίδιο memory segment
- Το k-οστό νήμα θα πρέπει να προσπελαύνει την k-οστή λέξη

Προϋποθέσεις για υψηλή επίδοση

Συνένωση αναφορών στην καθολική μνήμη – Προϋποθέσεις (II)

- Παράδειγμα πρόσβασης σε λέξεις 4 byte
  - Συνεχόμενες και ευθυγραμμισμένες



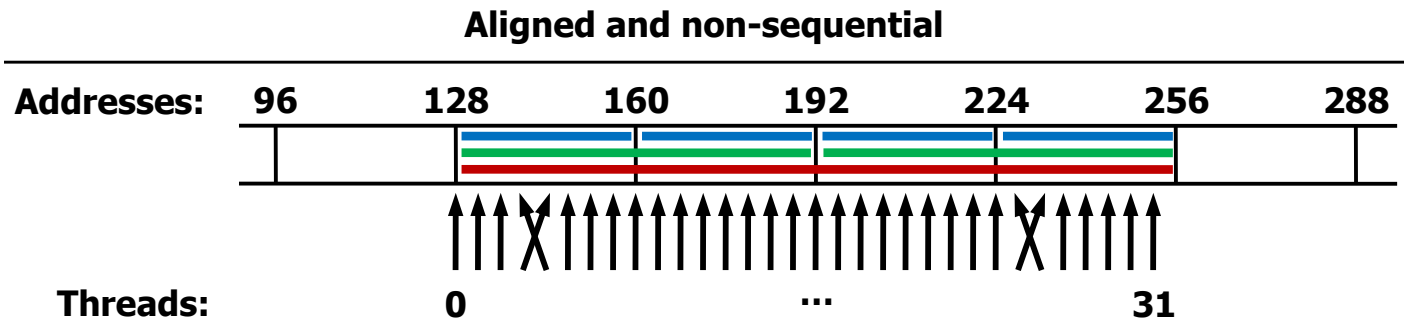
Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Πηγή: NVIDIA CUDA Programming Guide

Προϋποθέσεις για υψηλή επίδοση

Συνένωση αναφορών στην καθολική μνήμη – Προϋποθέσεις (III)

- Παράδειγμα πρόσβασης σε λέξεις 4 byte
  - Μη συνεχόμενες και ευθυγραμμισμένες



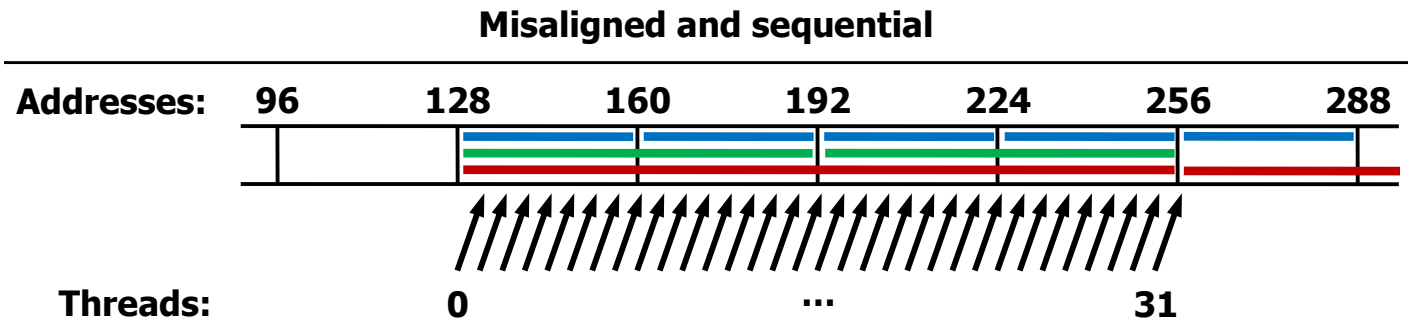
Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Πηγή: NVIDIA CUDA Programming Guide

Προϋποθέσεις για υψηλή επίδοση

Συνένωση αναφορών στην καθολική μνήμη – Προϋποθέσεις (IV)

- Παράδειγμα πρόσβασης σε λέξεις 4 byte
  - Συνεχόμενες και μη ευθυγραμμισμένες



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	7 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224 1 x 32B at 256	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256

Πηγή: NVIDIA CUDA Programming Guide

Προϋποθέσεις για υψηλή επίδοση

Warps χωρίς διακλαδώσεις

- Το SIMT μοντέλο επιτρέπει σε κάθε νήμα να ακολουθήσει διαφορετικό κλάδο, αλλά…
  - Κάθε κλάδος εκτελείται από όλα τα νήματα του warp, απενεργοποιώντας τα αντίστοιχα νήματα όταν εκτελείται ο "not taken" κλάδος τους
  - Το πλήθος των χρήσιμων υπολογισμών ανά νήμα μειώνεται σημαντικά

```
if (threadIdx.x > 2) {
    // do stuff
}
```

Divergence για το πρώτο warp κάθε thread block!

```
if (threadIdx.x/WARP_SIZE > 2) {
  // do stuff
}
```

No divergence!

Προϋποθέσεις για υψηλή επίδοση

Εκμετάλλευση της μοιραζόμενης μνήμης

- Η πρόσβαση στην μοιραζόμενη μνήμη έχει μηδενικό κόστος
- Οργάνωση σε 16 ή 32 banks
  - Διαδοχικές λέξεις ανατίθενται σε διαδοχικά banks
- Εκμετάλλευση της χρονικής τοπικότητας των προγραμμάτων
- Δύο φάσεις:
  1. (Προ)φόρτωση των δεδομένων προς υπολογισμό
  2. Υπολογισμοί με χρήση της μοιραζόμενης μνήμης

Περιορισμοί

- Περιορισμένο μέγεθος (16–48 KiB) που μοιράζονται όλα τα νήματα ενός block
- Μπορεί να οδηγήσει σε χαμηλή χρησιμοποιήση των SMs
- Προσοχή στα bank conflicts

Προϋποθέσεις για υψηλή επίδοση

Υψηλή χρησιμοποιήση των SMs

- Κάθε SM υποστηρίζει ένα μέγιστο αριθμό ενεργών block (ή warps)
  - Ένα warp είναι ενεργό όταν μπορεί να δρομολογηθεί
- Περισσότερα ενεργά νήματα μπορούν να κρύψουν τις καθυστερήσεις από την πρόσβαση στην καθολική μνήμη
- Η χρησιμοποιήση των SMs εξαρτάται από…
  - τις απαιτήσεις κάθε νήματος σε καταχωρητές και μοιραζόμενη μνήμη,
  - το πλήθος των νημάτων ανά block

Προϋποθέσεις για υψηλή επίδοση

Υψηλή χρησιμοποιήση των SMs – Παράδειγμα

- Υποθέτουμε GPU με compute capability 1.0:

Register file	8192
Shared memory	16384
Active blocks/SM	8
Active warps/SM	24

- Υποθέτουμε πυρήνα με

Block size	128
Register usage/thread	9
Shared memory usage/block	1024

- Πόσα ενεργά warps υπάρχουν ανά SM?
- CUDA Occupancy calculator

Εσωτερικό γινόμενο διανυσμάτων

Χωρίς χρήση κοινής μνήμης

```
__global__
void dot_product(float *out, float *a, float *b, int n)
{
    int tid = blockDim.x*blockIdx.x + threadIdx.x;
    if (tid >= n) return;

    float partial_result = a[tid] * b[tid];
    *out += partial_result;
}
```

Πού βρίσκεται το λάθος;

Εσωτερικό γινόμενο διανυσμάτων

Χωρίς χρήση κοινής μνήμης

```
__global__
void dot_product(float *out, float *a, float *b, int n)
{
    int tid = blockDim.x*blockIdx.x + threadIdx.x;
    if (tid >= n) return;

    float partial_result = a[tid] * b[tid];
    *out += partial_result;
}
```

Πού βρίσκεται το λάθος;

- `atomicAdd(out, partial_result)`

Εσωτερικό γινόμενο διανυσμάτων

Με χρήση κοινής μνήμης

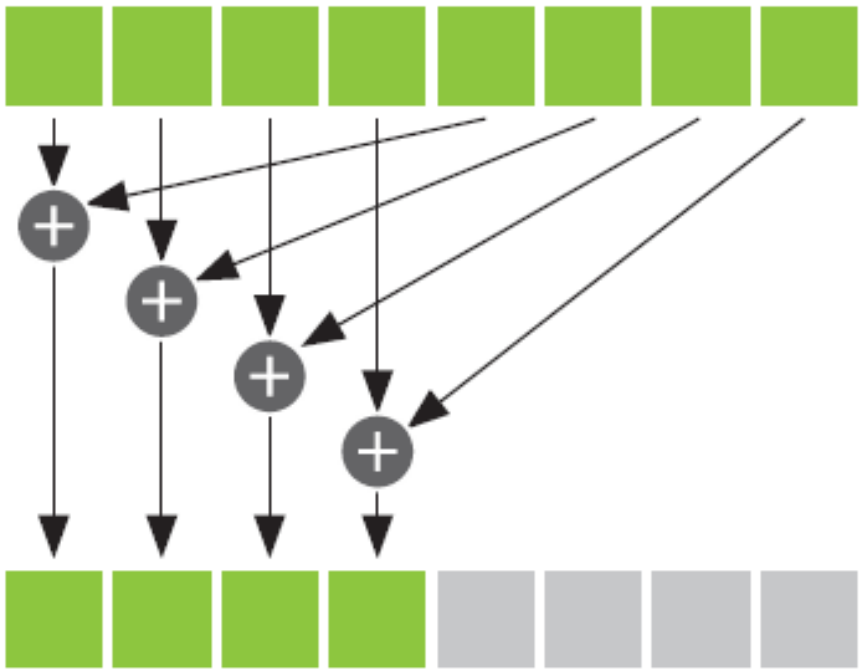
```
...
    __shared__ float partials[THREADS_PER_BLOCK];
    int tid = blockDim.x*blockIdx.x + threadIdx.x;
    int local_tid = threadIdx.x;
    if (tid >= n) return;
    partials[local_tid] = a[tid] * b[tid];
    __syncthreads();
    if (local_tid == 0) {
        float partial_result = 0;
        for (size_t i = 0; i < blockDim.x; ++i)
            partial_result += partials[i];
        atomicAdd(out, partial_result);
    }
```

Μπορούμε να παραλληλοποιήσουμε τον υπολογισμό του `partial_result`;

Εσωτερικό γινόμενο διανυσμάτων

Reductions in CUDA

Παραλληλοποίηση του `partial_result` εντός του thread block!

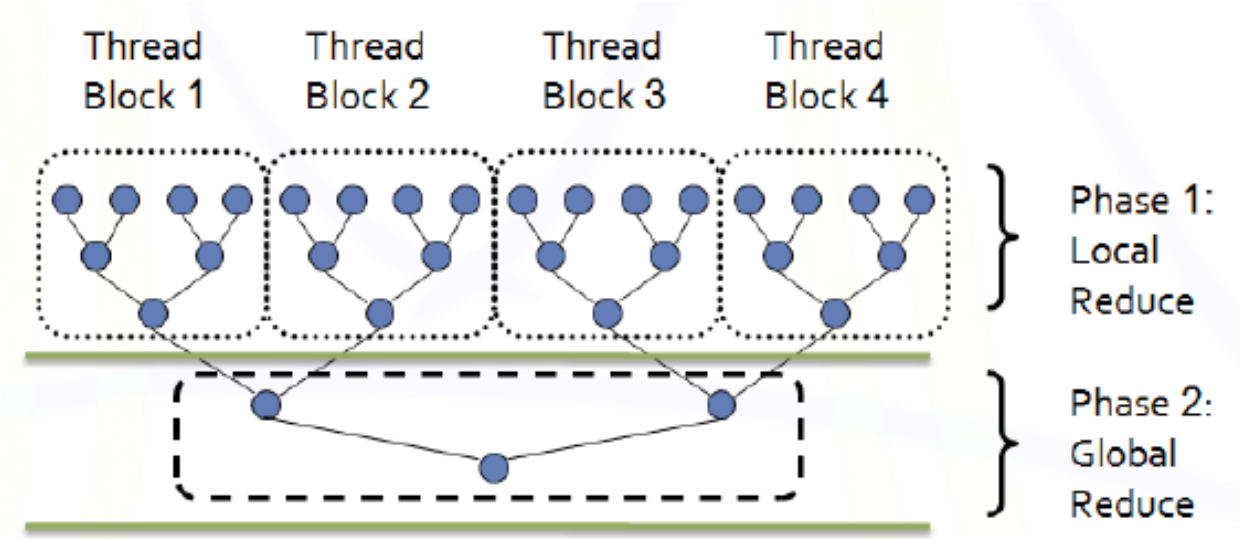


Πηγή: NVIDIA CUDA Programming Guide

Εσωτερικό γινόμενο διανυσμάτων

Reductions in CUDA

- **Phase 1:** reduction εντός των thread blocks με χρήση της κοινής μνήμης
- **Phase 2:** reduction ανάμεσα στα thread blocks στην κυρια μνήμη με χρήση atomics



Πηγή: NVIDIA CUDA Programming Guide

Εσωτερικό γινόμενο διανυσμάτων

Βελτιωμένη λύση με χρήση κοινής μνήμης

```
__global__
void dot_product(float *out, float *a, float *b, int n)
{
    ...
    __syncthreads();
    int i = blockDim.x / 2;
    while (i != 0) {
        if (local_tid < i)
            partials[local_tid] += partials[local_tid + i];
        __syncthreads();
        i /= 2;
    }
    if (local_tid == 0) {
        atomicAdd(out, partials[0]);
    }
}
```

Χρήσιμα links

- www.gpgpu.org
- www.gpucomputing.net
- NVIDIA Developer Zone,
  - https://developer.nvidia.com
  - http://docs.nvidia.com/cuda/cuda-c-programming-guide

Συζήτηση – Ερωτήσεις