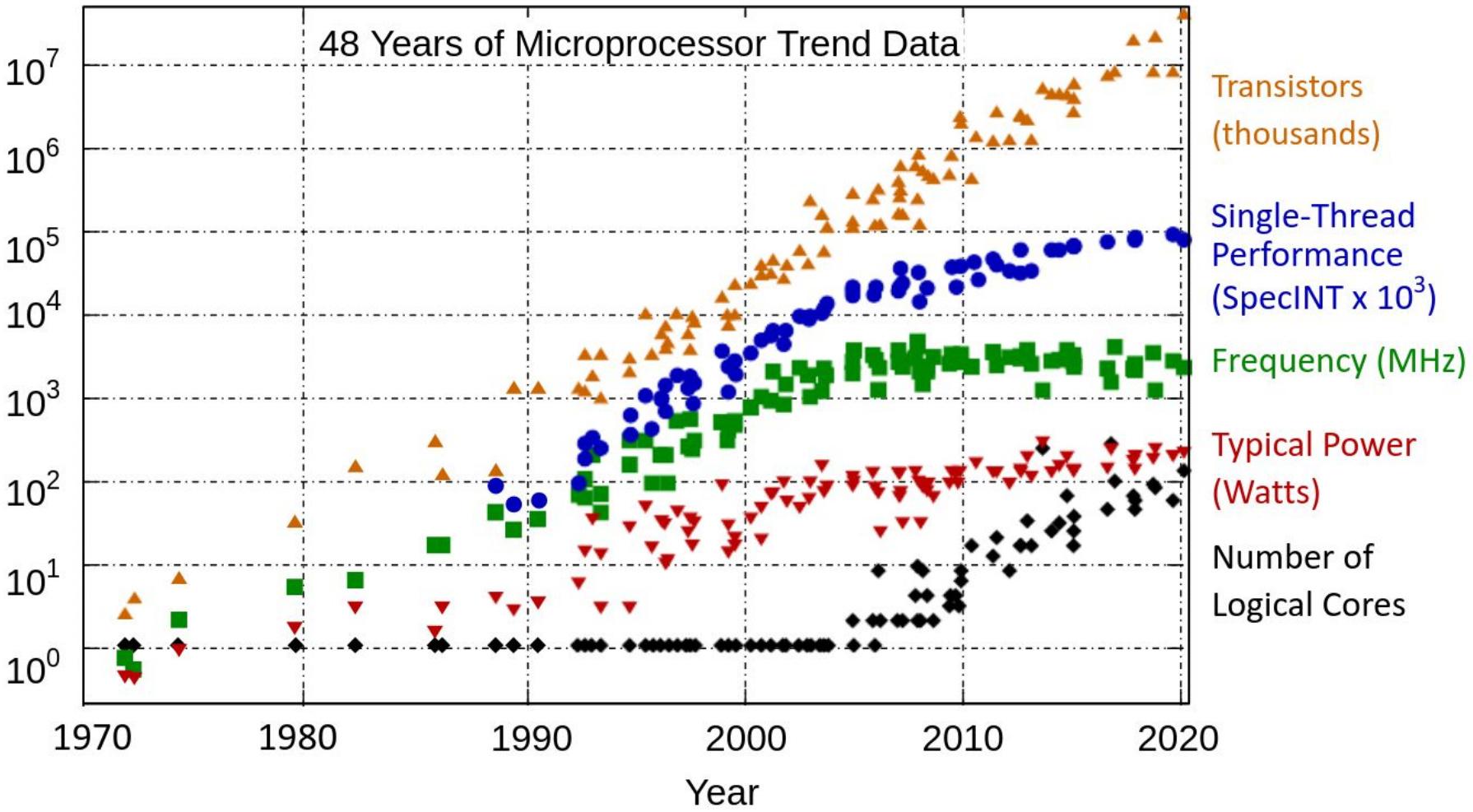
Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχ. και Μηχανικών Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Παράλληλος Προγραμματισμός σε Επεξεργαστές Γραφικών

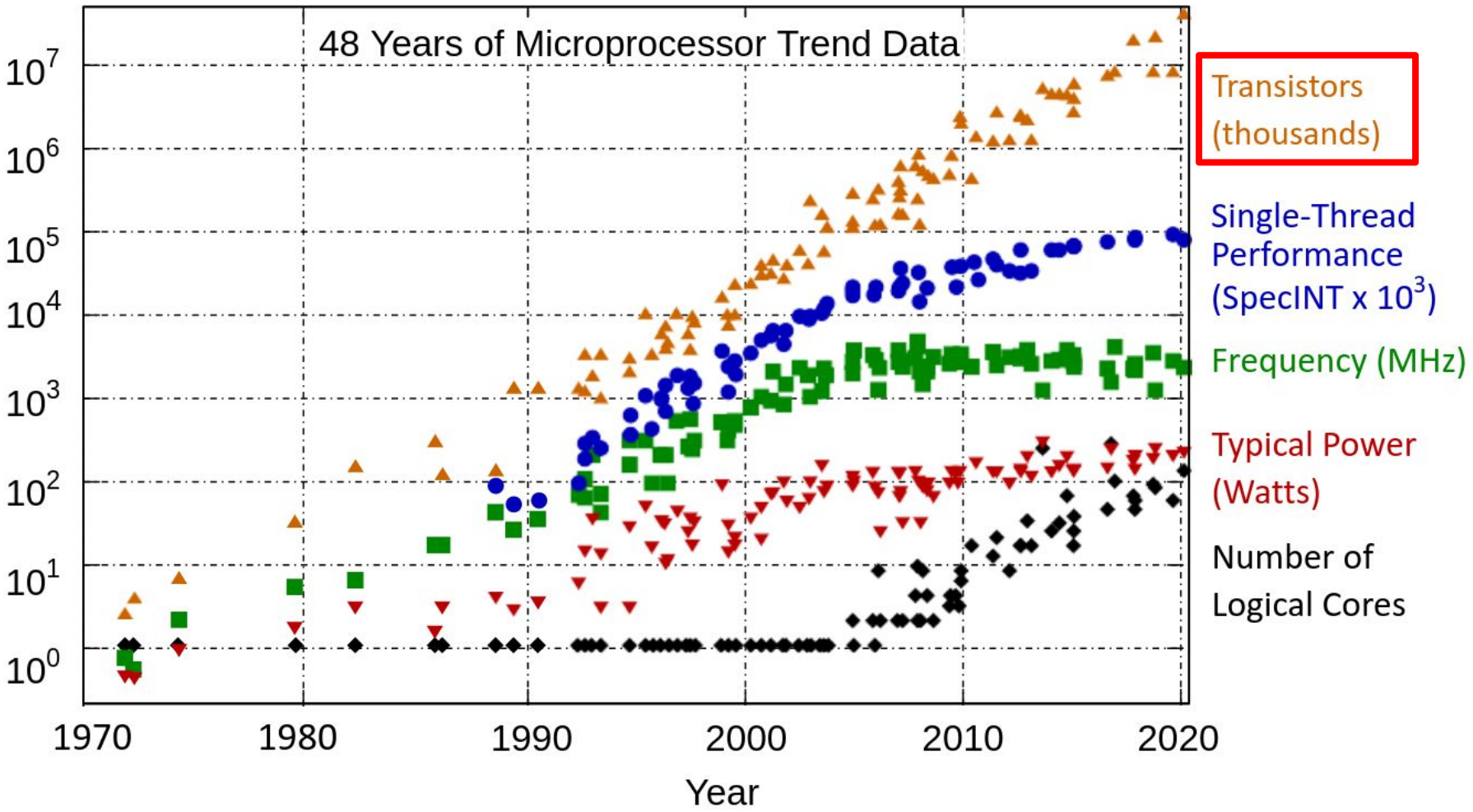
Συστήματα Παράλληλης Επεξεργασίας
9ο Εξάμηνο

Εισαγωγή

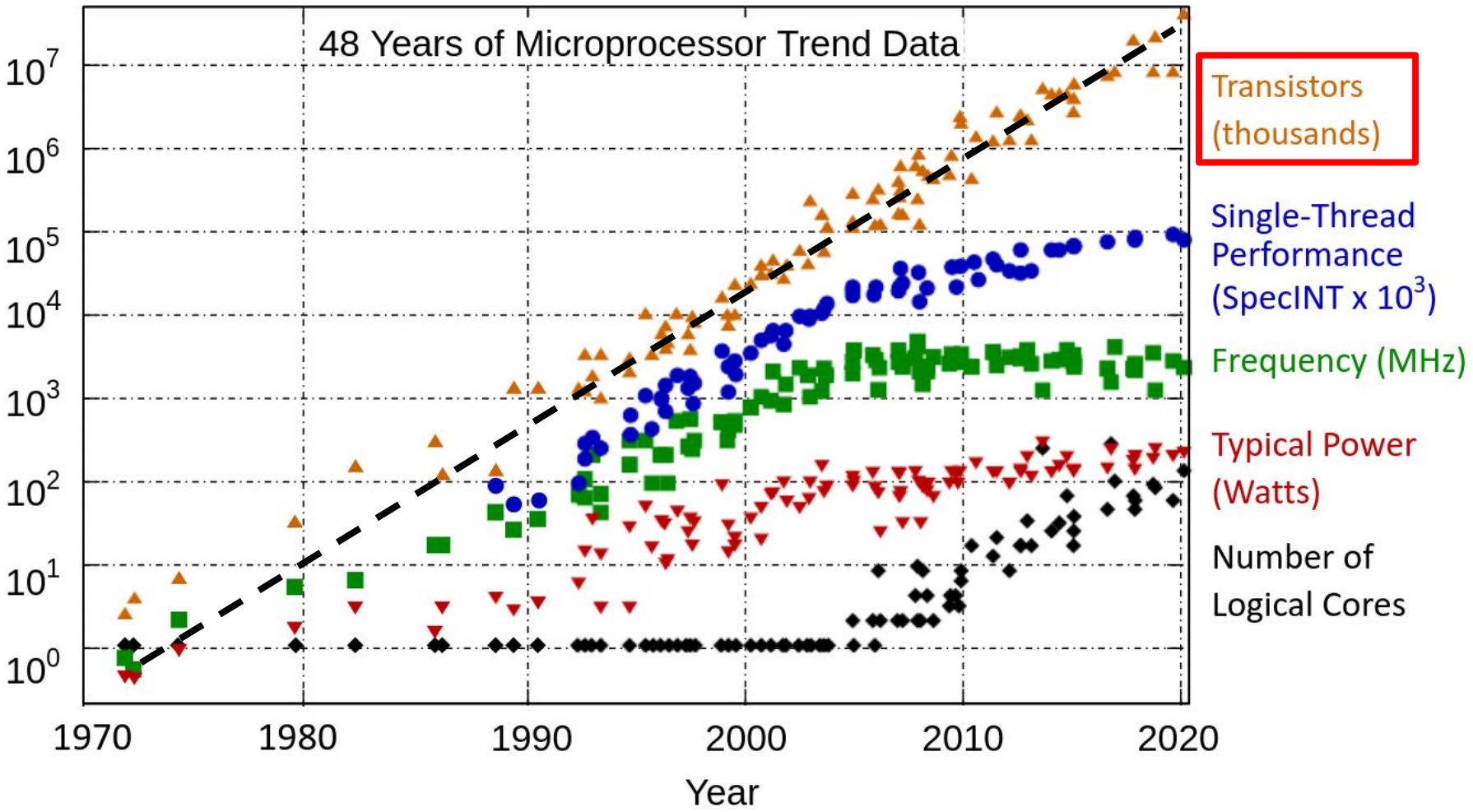
Εισαγωγή
Moore's law



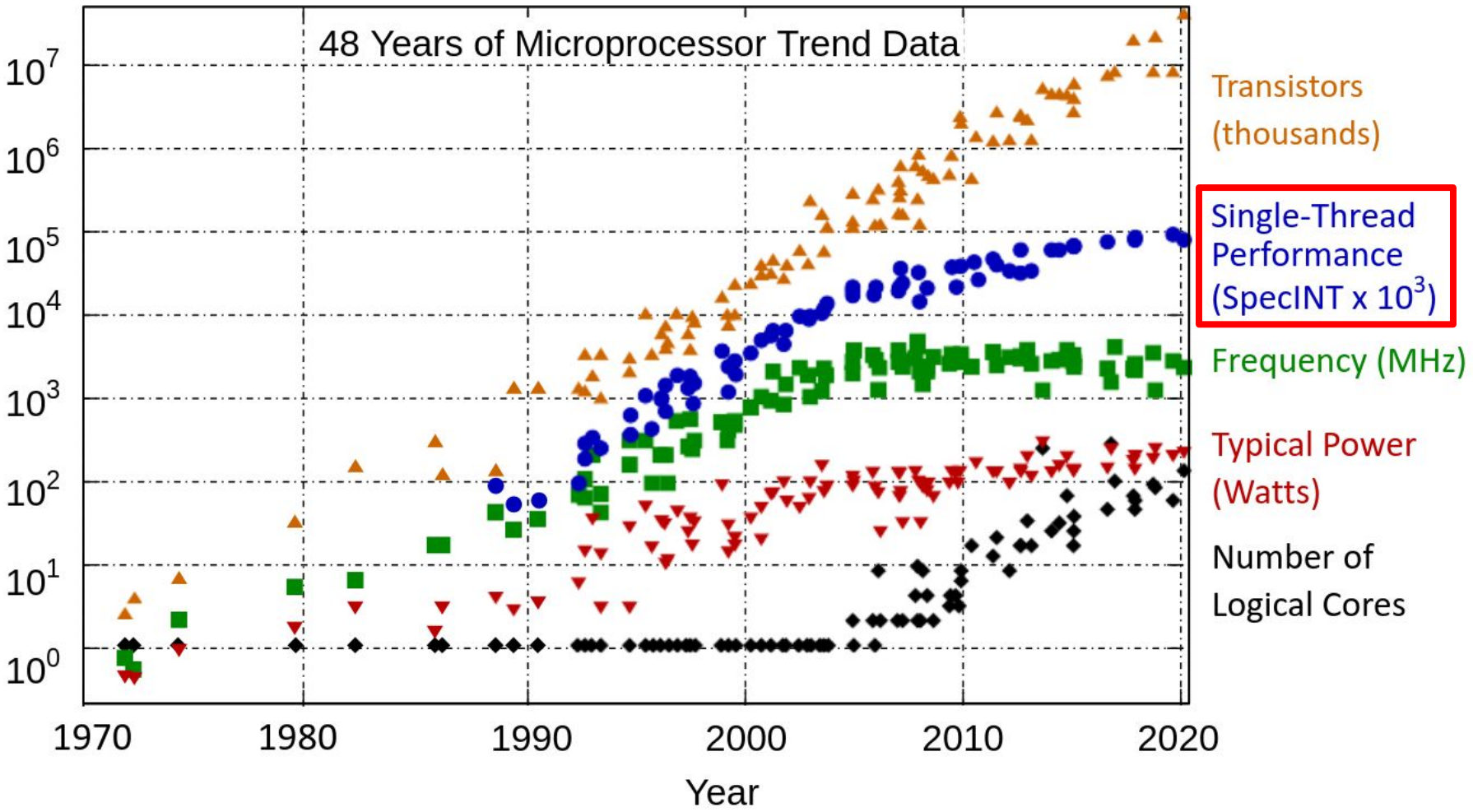
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten New plot and data collected for 2010-2020 by K. Rupp



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 2010-2020 by K. Rupp

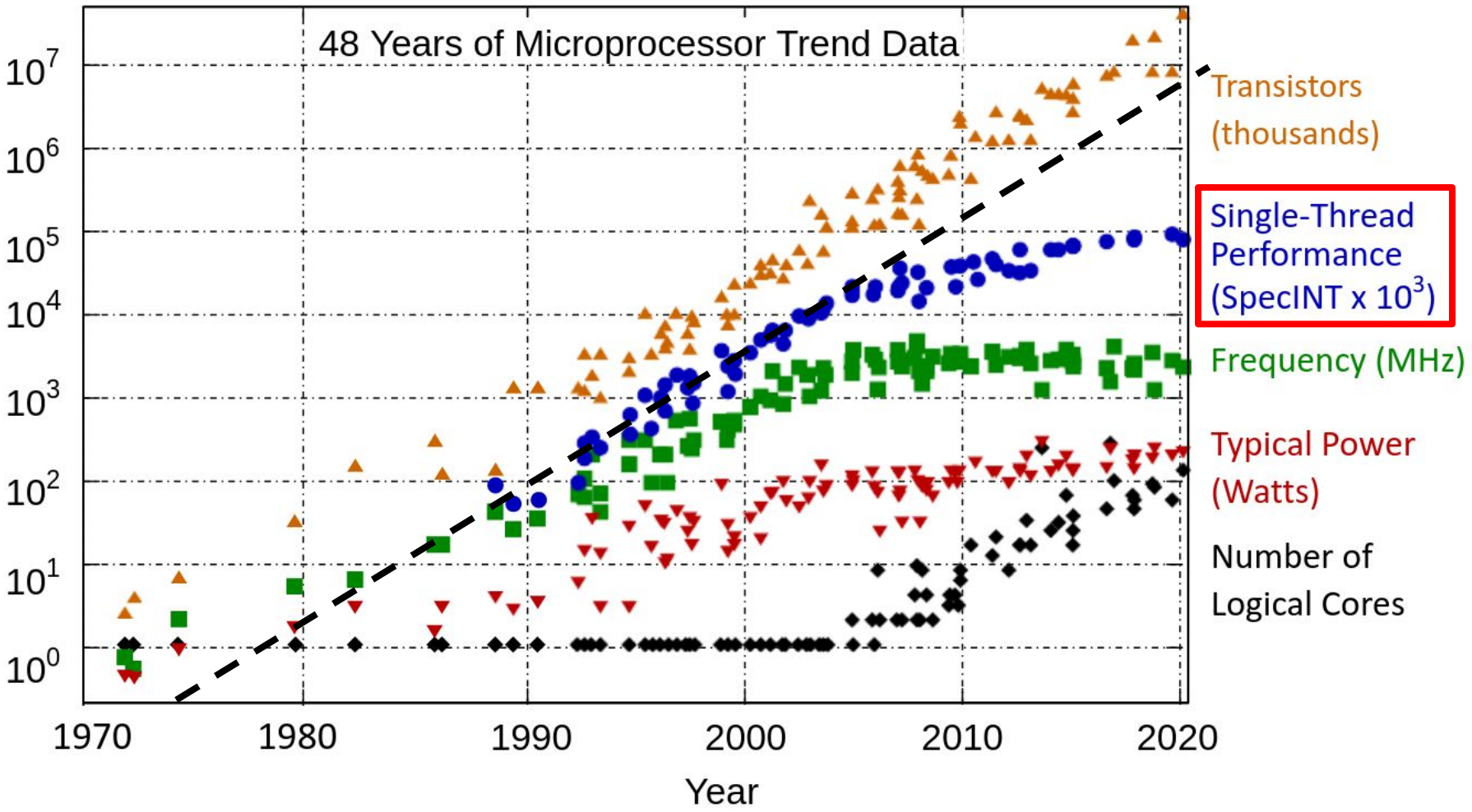


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten New plot and data collected for 2010-2020 by K. Rupp



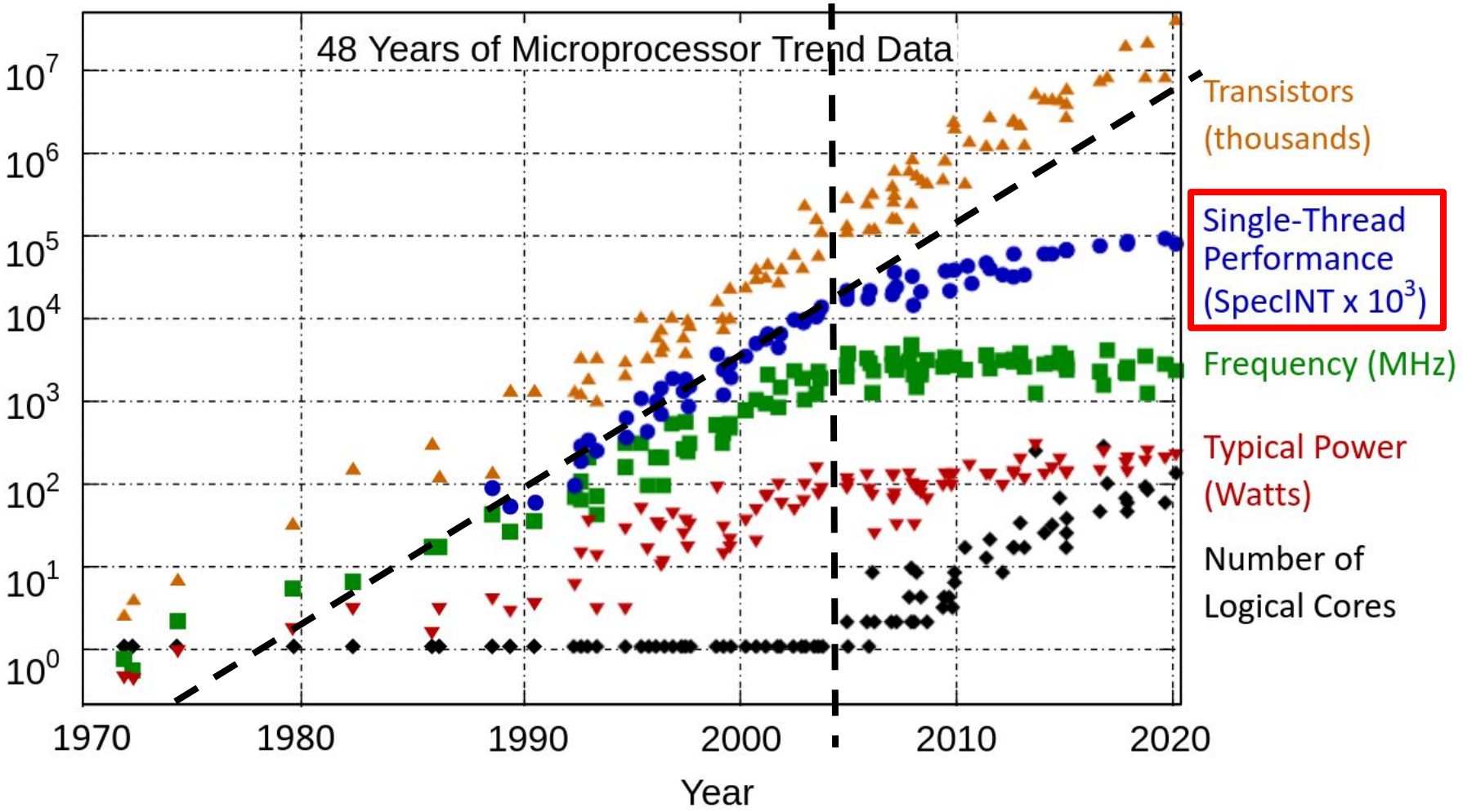
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten New plot and data collected for 2010-2020 by K. Rupp

Εισαγωγή
Moore's law



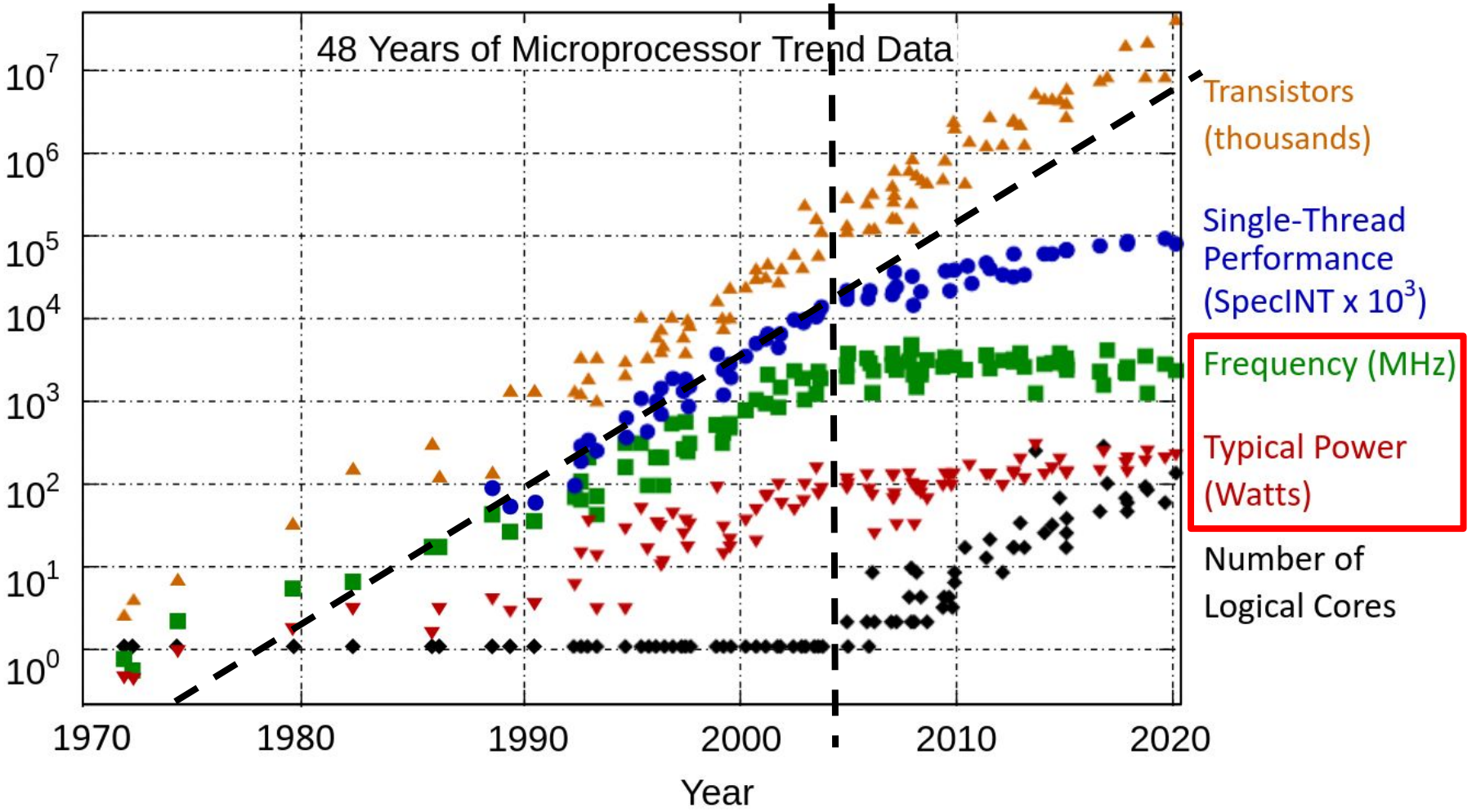
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten New plot and data collected for 2010-2020 by K. Rupp

Εισαγωγή
Moore's law

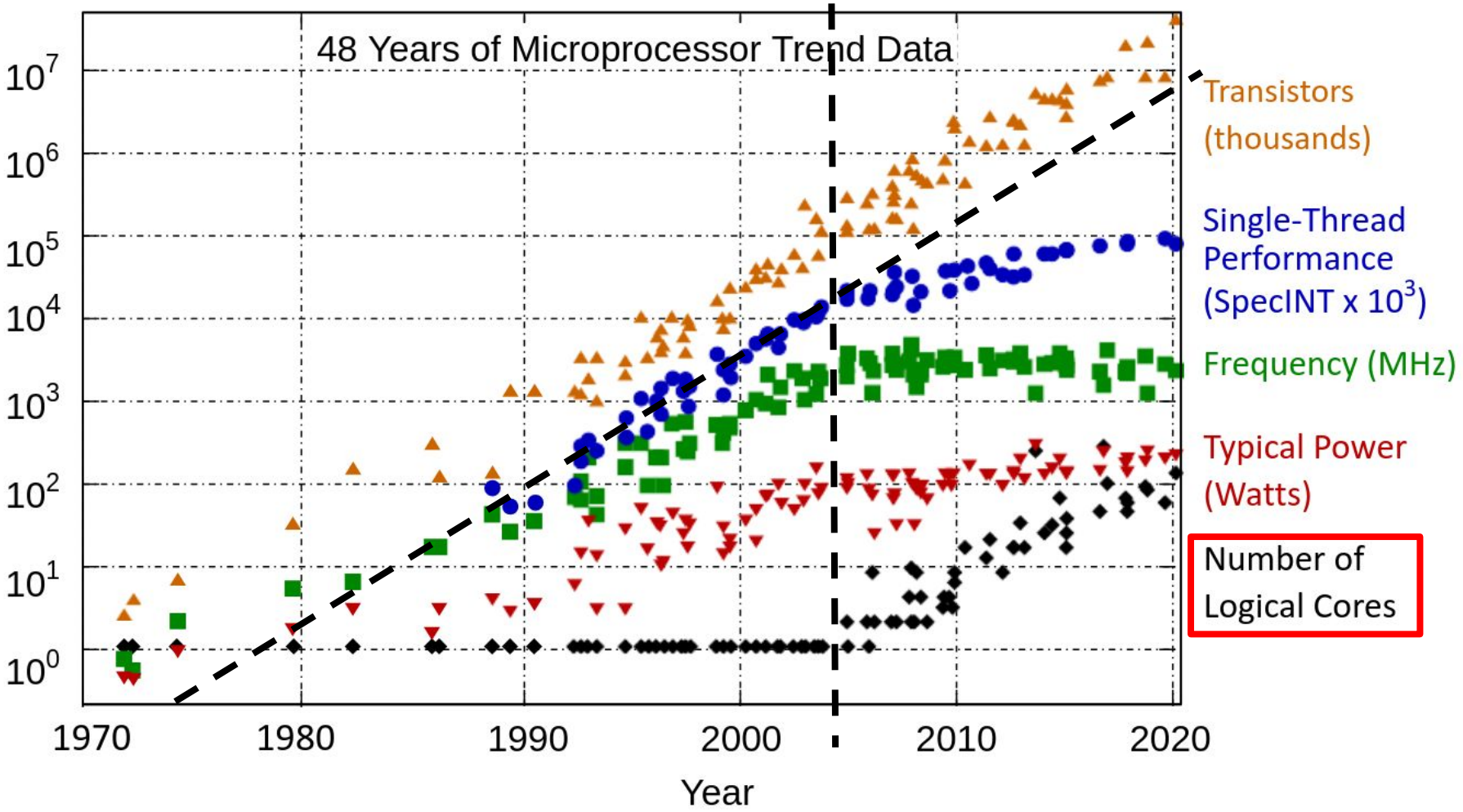


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten New plot and data collected for 2010-2020 by K. Rupp

Εισαγωγή
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 2010-2020 by K. Rupp



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 2010-2020 by K. Rupp

Εισαγωγή
The end of Moore's law (?)

- Υπάρχουν **φυσικοί** περιορισμοί στην αύξηση της συχνότητας.
  - Από το πόσα τρανζίστορ χωράνε σε ένα ολοκληρωμένο κύκλωμα...
    - Περαιτέρω μείωση διαστάσεων → τεράστιο **κόστος.**
    - Πλησιάζουμε τα φυσικά όρια των ημιαγωγών (πυριτίου).

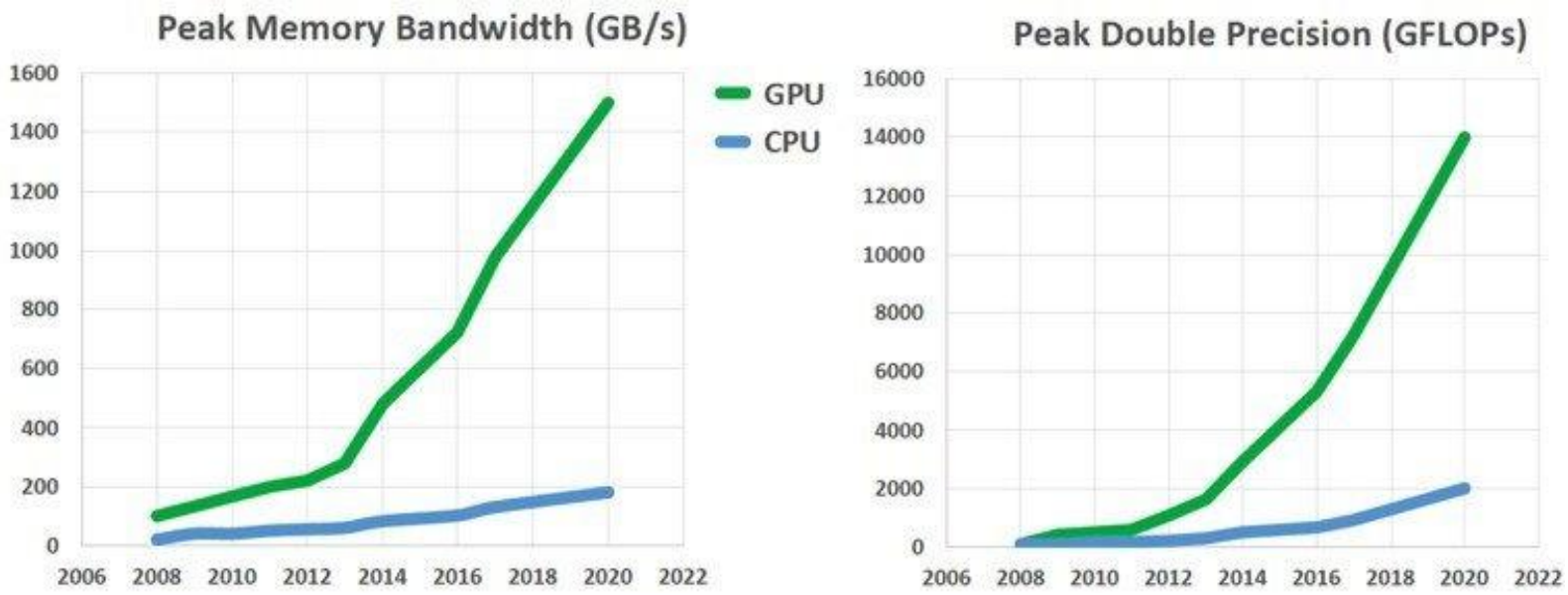
- Περαιτέρω βελτίωση της επίδοσης με επεξεργαστές **ειδικού σκοπού.**
  - Επιταχυντές (specialization).
  - Ετερογενή υπολογιστικά συστήματα (variety).

- Αρχιτεκτονικές εξειδικευμένες στην ***επιτάχυνση*** οικογενειών εφαρμογών:
  - Κρυπτογραφία
  - Γραφικά
  - Όραση υπολογιστών
  - Κρυπτο-νομίσματα
  - Internet of Things
  - Μηχανική μάθηση
  - …
- Συνήθως χρησιμοποιούνται ως ***συνεπεξεργαστές*** σε κλασικές CPUs.
  - Αλλαγή αλγοριθμικής λογικής η/και προγραμματιστικού μοντέλου.

Επιταχυντές Επεξεργασίας γραφικών (GPUs)

- Φιλοσοφία σχεδίασης: Embarrassingly parallel εφαρμογές.
  - Υψηλό παραλληλισμό (καθόλου / ελάχιστες εξαρτήσεις μεταξύ πράξεων).
  - Πολλά δεδομένα.
  - Πολλούς υπολογισμούς ανά δεδομένο (arithmetic intensity).
- Αρχικά, έτρεχαν **μόνο** γραφικά (non-programmable, hardwired logic).
  - π.χ. πέρασμα pixel εικόνας από φίλτρο, 3D object rotation, …
- Αλλά, δεν είναι μόνο τα γραφικά embarrassingly parallel…
  - Και ο νόμος του Moore καραδοκεί…
- Γιατί όχι GPUs γιά άλλες (ή όλες τις) υπολογιστικά απαιτητικές εφαρμογές?
  - Γέννηση των προγραμματίσιμων GPU (General Purpose GPU - GPGPU)

Εισαγωγή
Επίδοση CPUs vs GPUs

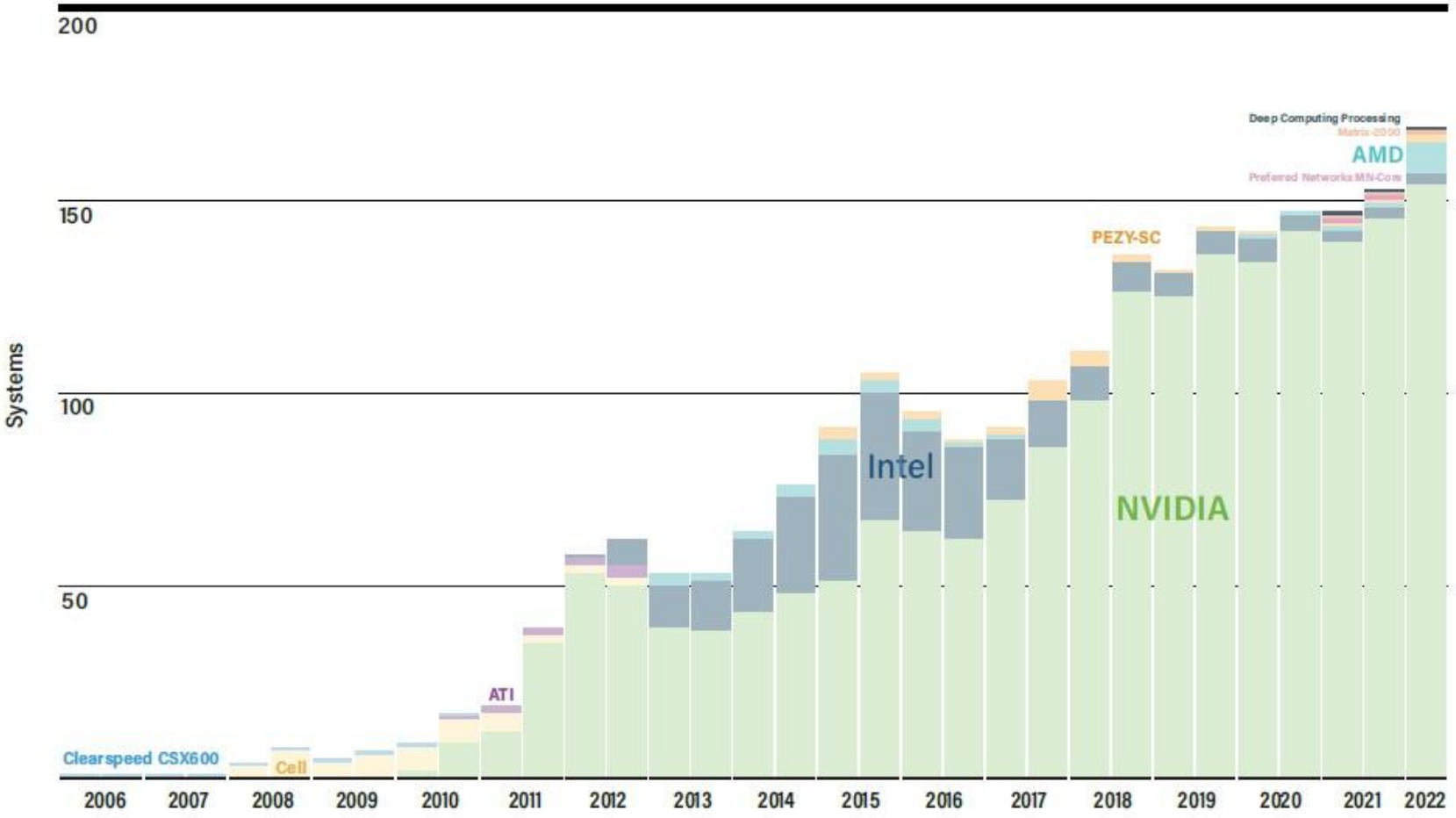


- Οι CPUs περιορίζονται λόγω του νόμου του Moore.
- Η επίδοση των GPU αντίθετα συνεχίζει να εξελίσσεται ραγδαία.
- Σε συνδυασμό με την σημαντική αύξηση του programmability τους.

Η επιστημονική κοινότητα άρχισε να χρησιμοποιεί GPUs για άλλες εφαρμογές.

Εισαγωγή

Επιταχυντές στο TOP500



- Ραγδαία αύξηση HPC συστημάτων με επιταχυντές τα τελευταία χρόνια.
- Η πλειοψηφία αυτών χρησιμοποιεί Nvidia GPUs.

Εισαγωγή
Η κορυφή του TOP500 (Nov 2025)

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**El Capitan** - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE DOE/NNSA/LLNL United States	11,340,000	1,809.00	2,821.10	29,685
2	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS, HPE DOE/SC/Oak Ridge National Laboratory United States	9,066,176	1,353.00	2,055.72	24,607
3	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
4	**JUPITER Booster** - BullSequana XH3000, GH Superchip 72C 3GHz, NVIDIA GH200 Superchip, Quad-Rail NVIDIA InfiniBand NDR200, RedHat Enterprise Linux, EVIDEN EuroHPC/FZJ Germany	4,801,344	1,000.00	1,226.28	15,794
5	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	

- Η εικόνα στην κορυφή αλλάζει σιγά σιγά… με νέους παίκτες (AMD, Intel).

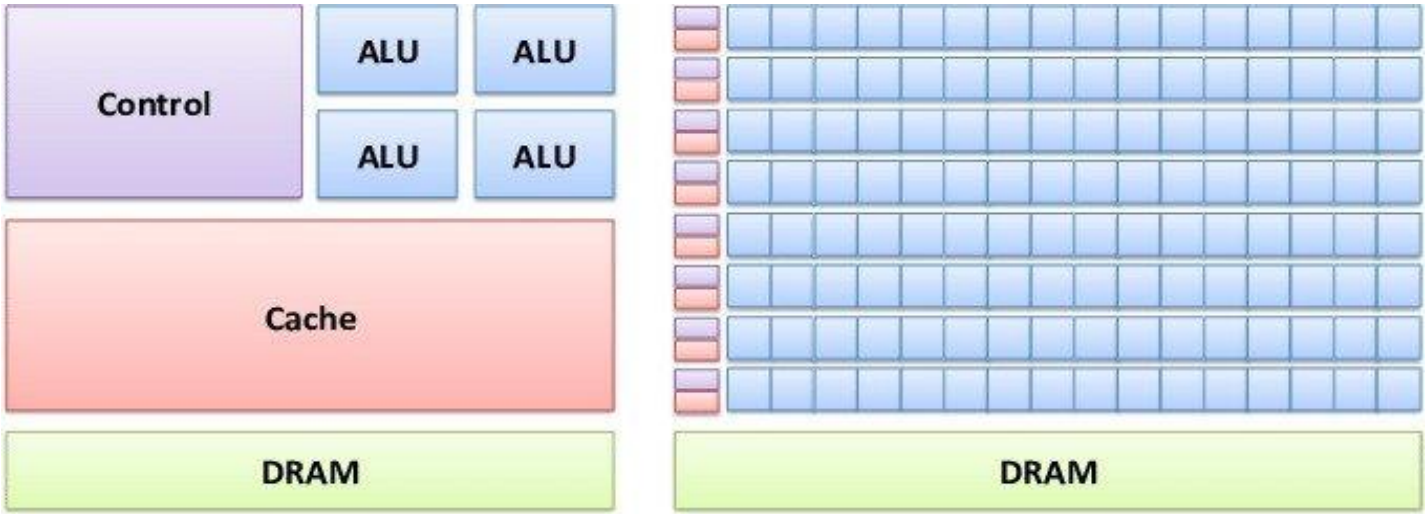
Αρχιτεκτονική GPU

Αρχιτεκτονική GPU
Φιλοσοφία σχεδίασης

- Στόχος: Επιτάχυνση ***data-parallel*** + ***compute-intensive*** εφαρμογών.
  - Χαμηλότερο **κόστος**, υψηλότερη **ενεργειακή απόδοση**.
- Πώς? Ολά απαιτούν χώρο στο υλικό (chip) → Tradeoffs.
  - ***Latency-oriented*** (CPU) vs ***Throughput-oriented*** (GPUs).
- Λυση: Μείωση *πολυπλοκότητας* → αύξηση του *παραλληλισμού*.
  - Υψηλός αριθμός πιο ***απλών πυρήνων*** (ποσότητα > ποιότητα).
  - Φτήνος παραλληλισμός μέσω ***hardware thread scheduling***.
  - Μείωση ελέγχου και κρυφής μνήμης για ***επιπλέον πυρήνες***.
  - ***Υψηλό εύρος μνήμης*** για κάλυψη του κενού.

Αρχιτεκτονική GPU
CPUs vs GPUs: Hardware

	CPU	GPU
Υπολογιστικοί πυρήνες	Λίγοι και σύνθετοι	Περισσότεροι και απλοί
Κρυφή μνήμη	Περίπλοκη ιεραρχία	Λίγα επίπεδα, απλή χρήση
Διαχείριση νημάτων	Λειτουργικό (αργή)	Υλικό (~μηδενικό κόστος)
Extra: κύρια μνήμη	Χαμηλό εύρος ζώνης Μεγάλη χωρητικότητα	Υψηλό εύρος ζώνης Περιορισμένη χωρητικότητα



Αρχιτεκτονική GPU
CPUs vs GPUs: Flynn's Taxonomy

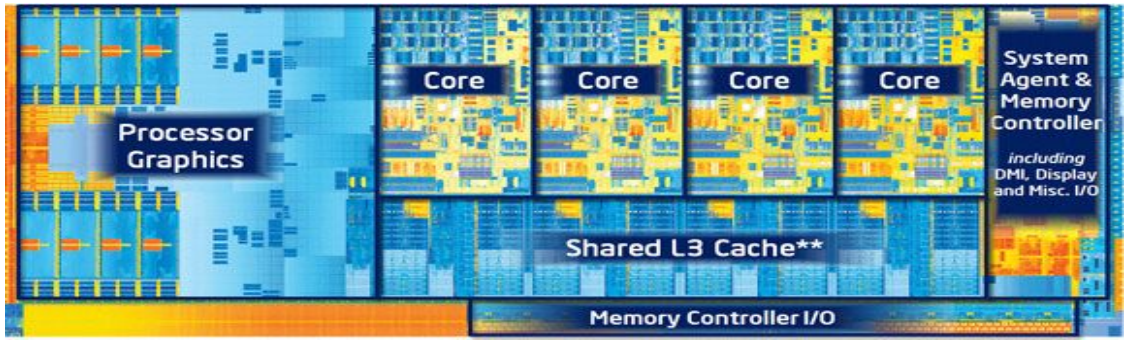
CPUs
MIMD + SIMD

- SIMD implementation:
  - ”explicit” με επεκτάσεις ISA (π.χ. AVX για x86, NANO/SVE για ARM).
  - Ο προγραμματιστής (ή ο μεταγλωττιστής) προσαρμόζει τον κώδικα.

GPUs
SIMD

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

- Ενσωματωμένες: Στο ίδιο τσιπ με τη CPU.
  - Λιγότερη υπολογιστική δύναμη (όχι πάντα). ×
  - Μοιράζονται την ίδια μνήμη (DRAM) με τη CPU.
    - Μεγαλύτερη χωρητικότητα. ✓
    - Μικρότερο εύρος ζώνης. ×
    - ανταγωνισμός για τη μνήμη. ×
  - Μικρότερη κατανάλωση ισχύος. ✓
  - Κομμάτι του υλικού χωρίς δυνατότητα επέκτασης. ×

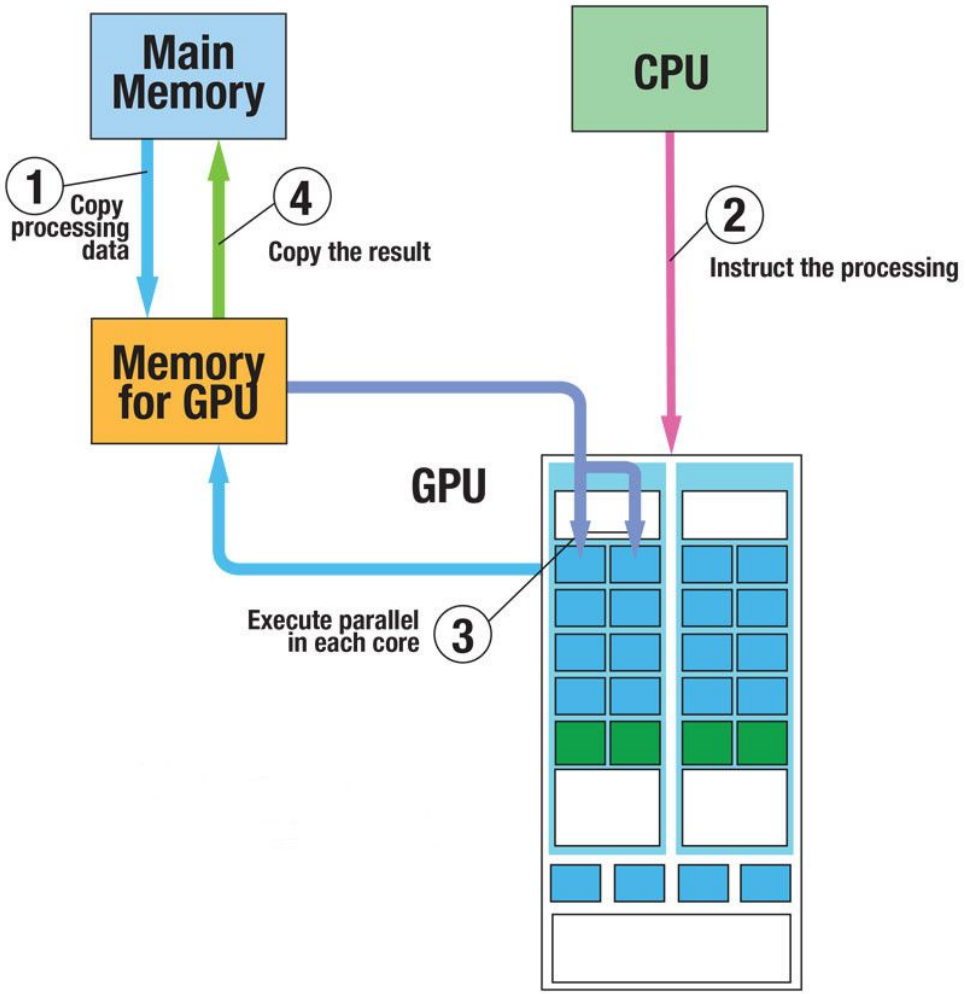


- Διακριτές: συνδέονται με τη CPU μέσω καναλιού.
  - Περισσότερη υπολογιστική δύναμη. ✓
  - Δική τους μνήμη:
    - Μεγαλύτερο εύρος ζώνης. ✓
    - Περιορισμένη χωρητικότητα. ×
  - Υψηλότερη κατανάλωση ισχύος. ×
  - Δυνατότητα επέκτασης και χρήσης πολλαπλών ανά σύστημα. ✓

Προγραμματισμός GPU

Προγραμματισμός GPU

Εκτέλεση σε διακριτές GPUs

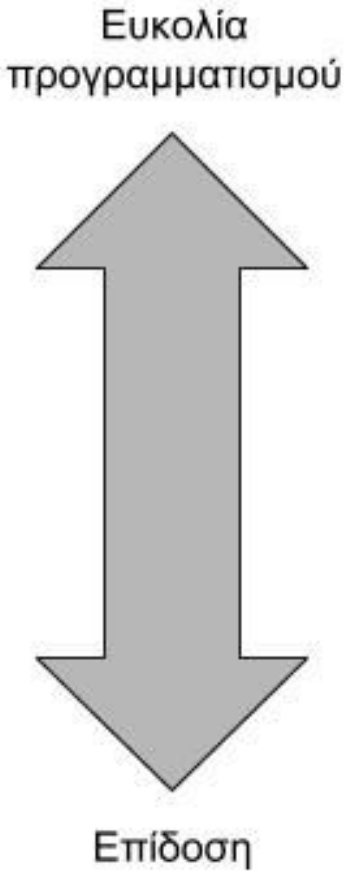


Βασικά βήματα:

1. Αντιγραφή δεδομένων εισόδου (***μνήμη CPU → μνήμη GPU***).
2. Η CPU αναθέτει τον υπολογισμό στην GPU (***kernel launch***).
3. Εκτέλεση ***kernel(s)*** στην GPU.
4. Αντιγραφή δεδομένων εξόδου (***μνήμη GPU → μνήμη CPU***).

Προγραμματισμός GPU
Πώς χρησιμοποιώ μία GPU;

- Με εφαρμογές που υποστηρίζουν GPUs (high-level frameworks):
  - TensorFlow, PyTorch
- Με βιβλιοθήκες:
  - cuBLAS, cuSPARSE, cuDNN
- Με διεπαφές (APIs) που βασίζονται σε οδηγίες (directives) προς το μεταγλωττιστή:
  - OpenACC
  - OpenMP (4.0+)
- Με επέκταση γλωσσών προγραμματισμού:
  - OpenCL (C/C++)
  - **NVIDIA CUDA** (C/C++, Fortran)



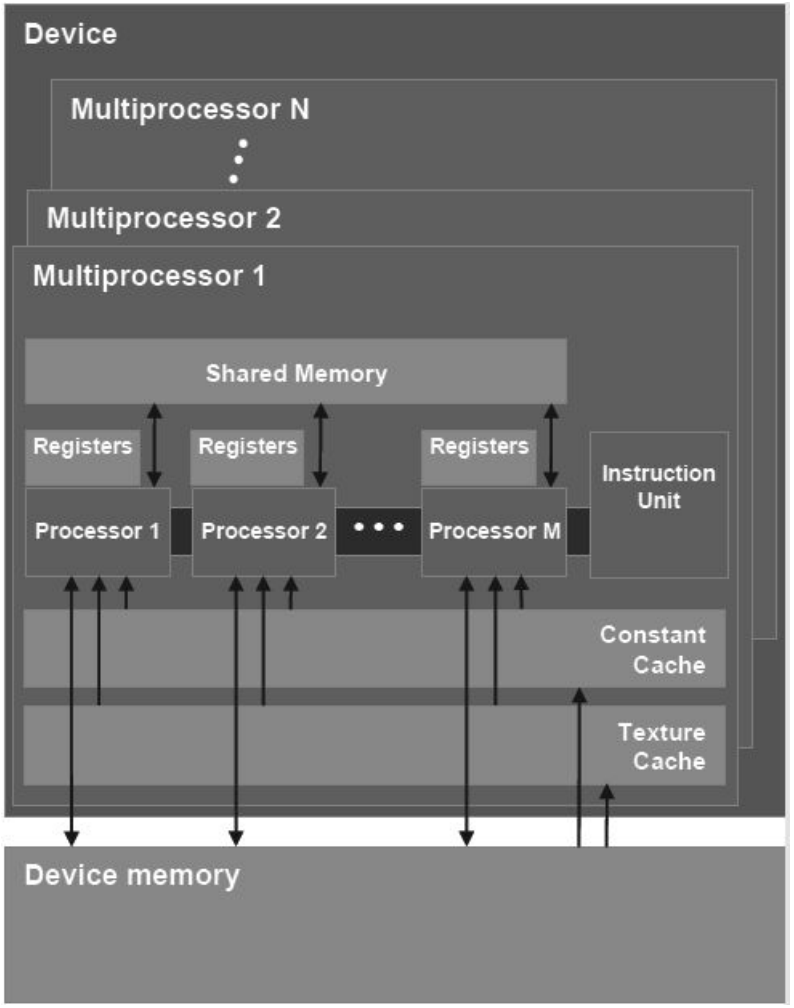
Προγραμματισμός GPU
Compute Unified Device Architecture (CUDA)

- Αναπτύχθηκε από την NVIDIA ειδικά για GPGPU computing.
- Το πιο διαδεδομένο εργαλείο προγραμματισμού για (NVIDIA) GPUs.
- *Διαχωρίζει:*
  - ***Application programming interface/model.***
    - API με **αντιστοιχίες** διαχωρισμού δεδομένων - αρχιτεκτονικής.
    - Περιγράφει **κανόνες** γιά την αξιοποίηση NVIDIA GPUs
    - **CUDA C/C++** : Υλοποίηση CUDA API για προγραμματιστές
  - ***Parallel computing platform***.
    - **Γενικευμένη** μορφή αρχιτεκτονικής (γιά οποιαδήποτε GPU).
      - Γιατί? Πραγματική μορφή → κλειστή (proprietary).

CUDA parallel computing platform

CUDA parallel computing platform

Γενικευμένη αρχιτεκτονική NVIDIA GPU



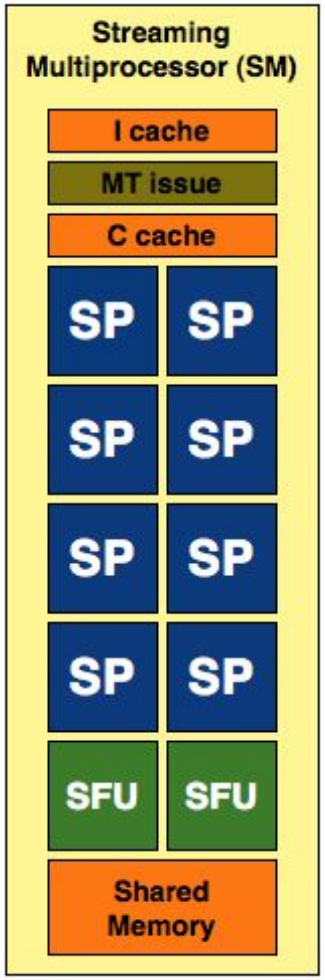
Πηγή: NVIDIA CUDA Programming Guide

- Οι GPUs αποτελούνται από πολυεπεξεργαστές ροών **(Streaming Multiprocessors – SMs)**
- Κάθε ***SM*** αποτελείται από:
  - Επεξεργαστές ροών (**Streaming Processors – SPs** ή **CUDA cores**)
  - Πολύ μεγάλο αρχείο καταχωρητών (8K – 32K)
  - Shared memory
  - Constant cache (read-only)
  - Texture cache

CUDA parallel computing platform

Βασική δομική μονάδα: Streaming Multiprocessor (SM)

- Κάθε *SM* αναλαμβάνει να εκτελέσει ένα ή περισσότερα μπλοκ νημάτων (***thread blocks***).
  - Όλα τα νήματα ενός thread block μοιράζονται τους πόρους του *SM* που το εκτελεί.
- Τα νήματα μετά οργανώνονται σε 32-άδες (***warps***).
  - Τα *warps* δρομολογούνται από τους *warp schedulers* στα *cuda cores* του *SM*.
- Σε κάθε κύκλο ρολογιού όλα τα νήματα ενός *warp* εκτελούν **την ίδια εντολή**.
  - **S**ingle **I**nstruction **M**ultiple **T**hread (***SIMT***)
- Η δρομολόγηση των *warps* έχει "μηδενικό" κοστος.



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/Turing	7.x
Ampere/Ada Lovelace	8.x
Hopper	9.x

CUDA parallel computing platform

Παράδειγμα: αρχιτεκτονική Volta

Πηγή: NVIDIA CUDA Programming Guide.

CUDA programming model

CUDA programming model

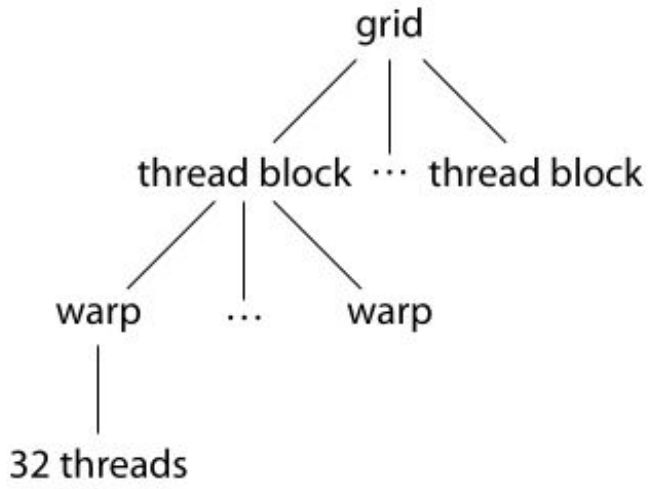
Βασικό μοντέλο εκτέλεσης

- 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 = αριθμός νημάτων).**
      - **S**ingle **I**nstruction **M**ultiple **T**hread (**SIMT**) μοντέλο εκτέλεσης.
- Κλιμακώσιμο/Μεταφέρσιμο
  - Το ίδιο εκτελέσιμο μπορεί να εκτελεστεί σε διαφορετικές κάρτες γραφικών με διαφορετικό αριθμό από SMs.

CUDA programming model

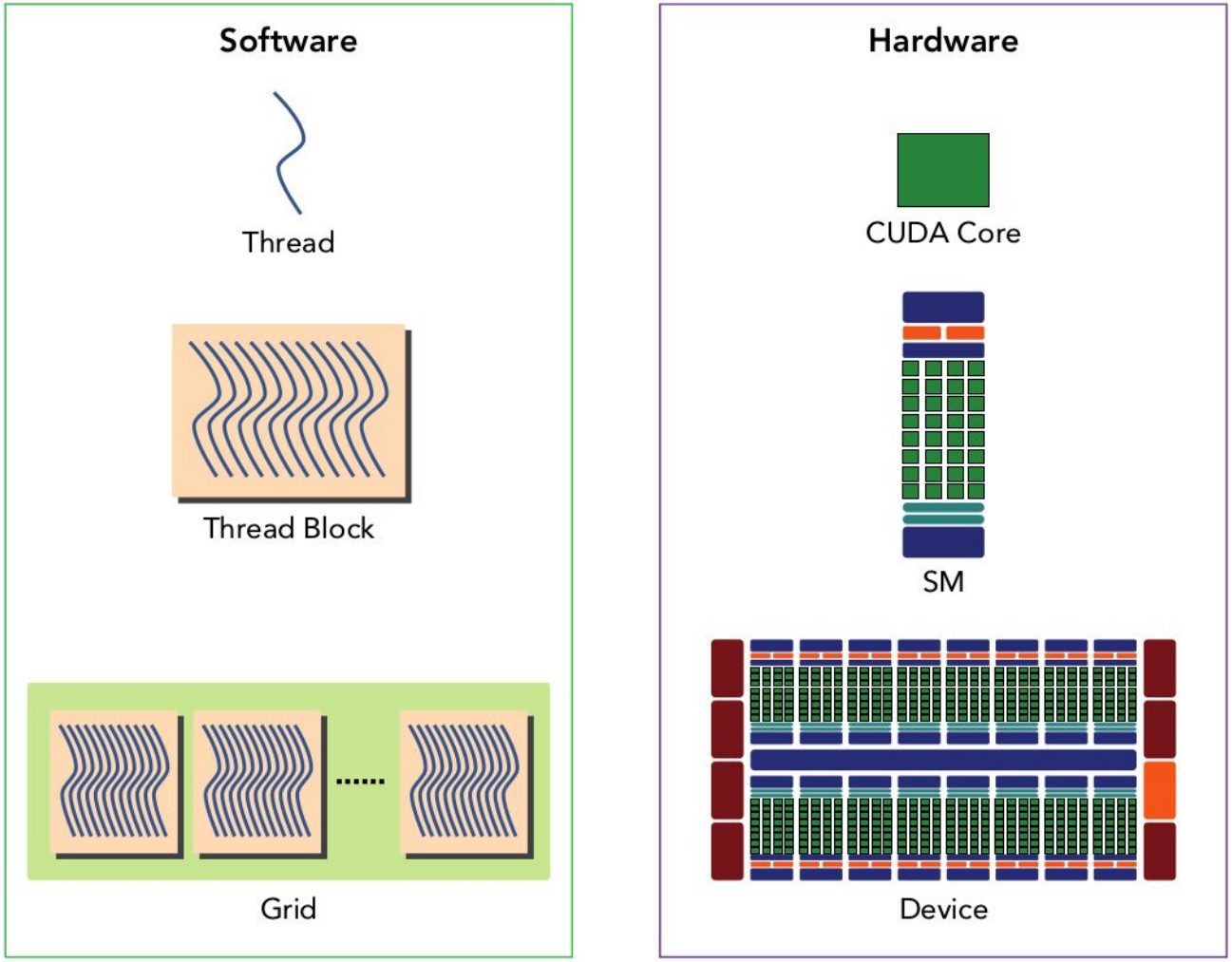
Thread abstraction

- Τα threads οργανώνονται κάτω από ένα **ιεραρχικό σχήμα**.
- Σε κάθε *kernel,* ο προγραμματιστής ορίζει ένα **πλέγμα** (***grid***) από **μπλοκς νημάτων** (***thread blocks***), τα οποία μπορούν να είναι ***1D, 2D ή 3D***.
- Τα thread blocks οργανώνονται αυτόματα σε 32ες από threads: ***warps***.
  - ➢ Μέγεθος *thread block* πολλαπλάσιο του 32.
- 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 (ίδιος κώδικας)*.
  - Μπορούν να *διαφοροποιηθούν* στην εκτέλεση κώδικα *(ifs, loops etc)*.
  - Το πρόγραμμα οφείλει να είναι ορθό για οποιαδήποτε σειρά εκτέλεσης των threads block (πιο συγκεκριμένα, των warps).

CUDA programming model

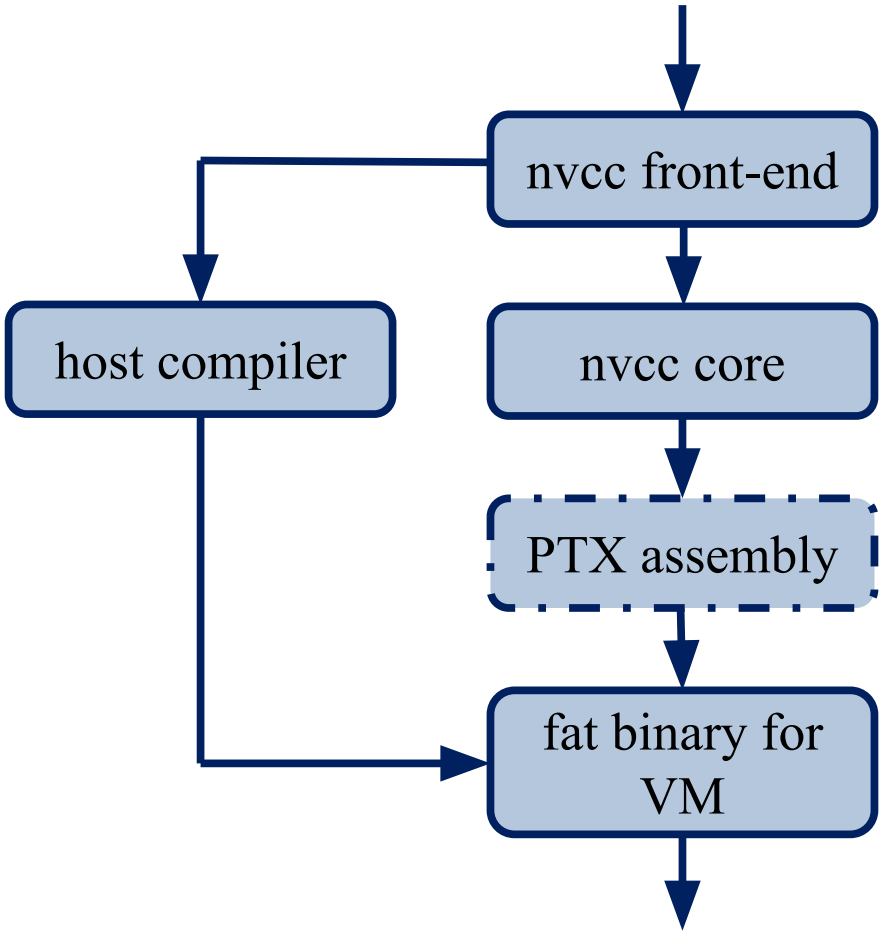
Διαχωρισμός ιεραρχίας μνήμης

Επίπεδο	**Ορατότητα**	**Χρήση**
Καθολική μνήμη (global)	Ολα τα blocks	Με συναρτήσεις διαχείρισης μνήμης σε *host code* και δομές (πίνακες κ.ο.κ.) σε *device code*.
Μνήμη σταθερών (constant)	Ολα τα blocks	Με χρήση του προσδιοριστή __constant__ σε *device code*.
Μνήμη textures	Ολα τα blocks	Με χρήση ειδικών δομών που ονομάζονται **texture references.**
Μοιραζόμενη μνήμη (shared)	Ολα τα νήματα **ενός** block	Με χρήση του προσδιοριστή __shared__ σε *device code*.
Register file	Ολα τα νήματα **ενός** block	Με τον ορισμό οποιασδήποτε μεταβλητής σε *device code*.

CUDA C/C++

- Επέκταση της C/C++ με επιπλέον τύπους και προσδιοριστές
- Χρησιμοποιεί το front-end της C++
- 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.

- nvcc -o prog_name gpu_prog.cu



Μεταγλώττιση
Σύνδεση με την βιβλιοθήκη cudart, −lcudart

- 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.**

Διαχωρισμός κώδικα host/device - Παράδειγμα

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

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

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

- Στην κλήση ενός *kernel* ορίζονται το μέγεθος του block και του grid
  - **dimX** : CUDA type για X διαστάσεις -> [uint,uint,...] X times
    - *dim1 var_name(uint)*, *dim2 var_name(uint, uint)* κ.ο.κ.
- Παράδειγμα δημιουργίας 1D block (*32 threads*) και grid για N(ήματα):

```
int main(…)
{ …
dim1 block(32);
dim1 grid( (N + 31)/32);
my_gpu_kernel <<<grid , block>>>(kernel_input_args);
…
}
```

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

Προσδιορισμός νήματος εντός thread block (*local TID*)

- Το CUDA C/C++ ορίζει τις εξής μεταβλητές:
  - **uint3 threadIdx[.x|.y|.z]**: συντεταγμένες του νήματος εντός του block.
  - **dim3 blockDim[.x|.y|.z]:** διαστάσεις του block.
  - Διαθέσιμες σε κάθε συνάρτηση ***__device__*** και ***__global__***.
- Το τοπικό *ID* ενός thread (***local TID***) προσδιορίζεται από τις συντεταγμένες του εντός του thread block.
  - Για *1D* block(*blockDim.x*) το *local TID* του νήματος (x) = ***threadIdx.x***
  - Για *2D* block(*blockDim.x*, *blockDim.y*), το *local TID* του νήματος (x, y) = ***threadIdx.x + threadIdx.y * blockDim.x***
- Το *local TID* χρησιμοποιείται για εργασίες **εσωτερικά** του thread block.
  - reductions, χρήση shared memory, κ.ο.κ.

CUDA C/C++

Προσδιορισμός θέσης νήματος στο πλέγμα (*global TID*)

- Το CUDA C/C++ επίσης ορίζει τις εξής μεταβλητές:
  - **uint3 blockIdx[.x|.y|.z]**: συντεταγμένες κάθε block εντός του grid.
  - **dim3 gridDim[.x|.y|.z]:** διαστάσεις του grid.
  - Διαθέσιμες σε κάθε συνάρτηση ***__device__*** και ***__global__***.
- Το καθολικό *ID* ενός thread (***global TID***) προσδιορίζεται συνδυάζοντας τις συντεταγμένες του block του στο grid με το *local TID* του.
  - Π.χ., το *global TID* ενός νήματος σε μονοδιάστατο block και grid είναι:
  - *global TID* = ***local_tid + blockDim.x * blockIdx.x***
- Το *global TID* χρησιμοποιείται για την **ανάθεση δεδομένων** στα νήματα.
  - Το αντίστοιχο του N σε ένα for loop…

- Το μέγεθος της μοιραζόμενης μνήμης μπορεί να καθοριστεί:
  - κατά τον ορισμό μεταβλητών μέσα σε __device__ και __global__ συναρτήσεις.
  - Hard-coded: ανεξάρτητο αριθμού νημάτων

```
__global__ void gpu_kernel(...)
{
    …
    __shared__ float array1 [256];
    __shared__ double array2 [42];
    …
}
```

- Αλλά και κατά την κλήση μίας __global__ συνάρτησης :
  - Ως τρίτη παράμετρος κατά την κλήση του πυρήνα

```
__global__ void gpu_kernel(...)
{ ...
   extern  __shared__  float shmem_buff [ ] ;
  …
}
```

```
int main ( ) {
      shmem_sz = 32 * sizeof ( float ) ;
      gpu_kernel <<<G , B , shmem_sz >>> (...) ;
      …
}
```

- Barrier: Όμοια με OpenMP, εγγυάται ότι:
  - Όλα τα νήματα του block έχουν φτάσει σε κάποιο σημείο.
  - Όλες οι προσβάσεις στη μνήμη (global/shared) έχουν ολοκληρωθεί.
  - ***__syncthreads()***
- Memory fence:
  - Εγγυάται ότι όλες οι προσβάσεις στην (global ή shared) μνήμη μέχρι αυτό το σημείο έχουν πραγματοποιηθεί.
  - ***__threadfence(), __threadfence_block()***
- Ατομικές συναρτήσεις:
  - ***atomicAdd(addr, value)***

- Ατομικές συναρτήσεις:
  - ***atomicAdd(addr, value)***
  - Πολύ ακριβό!!! Ποτέ για πολλές τιμές (π.χ. για όλα τα threads)
- Καθολικός συγχρονισμός
  - ***cudaDeviceSynchronize()***
  - Μόνο μεταξύ διαδοχικών κλήσεων πυρήνων.
  - **Πρέπει** να οριστεί από τον προγραμματιστή.

Σημαντικό: Γιατί μπορώ να κάνω καθολικό atomic αλλά όχι barrier?

- Συναρτήσεις διαχείρισης μνήμης
  - cudaMalloc(), cudaFree()
  - cudaMallocHost(), cudaFreeHost()
- Συναρτήσεις μεταφοράς δεδομένων με την συσκευή
  - cudaMemcpy(), cudaMemcpy2D(), cudaMemcpyAsync()
- Συναρτήσεις διαχείρισης σφαλμάτων
  - cudaGetLastError()
- Συναρτήσεις επιλογής/διαχείρισης της συσκευής
  - cudaGetDevice(), cudaSetDevice()

Παράδειγμα 1: άθροισμα διανυσμάτων

Άθροισμα διανυσμάτων

Host code (1/2)

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

    // Initialize data on the CPU
    ...

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

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

Άθροισμα διανυσμάτων

Host code (2/2)

```
    // Launch the kernel with N/128 thread blocks of 128 threads
    dim1 block(128);
    dim1 grid( (N+127)/128 );
    vec_add<<< grid, block >>>(dev_a, dev_b, dev_c, N);

    // Synchronize device! Kernel launch is async.
    cudaDeviceSynchronize();

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

```
__device__ int get_GTID(){
    return blockDim.x*blockIdx.x + threadIdx.x;  // 1D grid and block
}
```

```
__global__ void vec_add(const float *a, const float *b, float *c, int N) {
    int tid = get_GTID();
    if (tid >= N)
        return;
    c[tid] = a[tid] + b[tid];  // 1 addition per thread
}
```

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 (επιτρεπτές) fixed διαστάσεις (ανεξάρτητες του N)
dim1 block(128);
dim1 grid(84);
vec_add<<< grid, block >>>(dev_a, dev_b, dev_c, N);
```

```
__global__ void vec_add(const float *a, const float *b, float *c, int N) {
    // Αναθέτουμε σε κάθε νήμα περισσότερα στοιχεία (Grid-stride loop)
    for ( int tid =get_GTID(); tid < N;  tid += blockdim.x*gridDim.x)
        c[tid] = a[tid] + b[tid];
}
```

Τώρα μπορούμε να προσθέσουμε διανύσματα οποιουδήποτε μήκους, περιορισμένοι μόνο από το μέγεθος της μνήμης στην GPU.

Ζητήματα επίδοσης

Ζητήματα επίδοσης

Προϋποθέσεις για υψηλή επίδοση

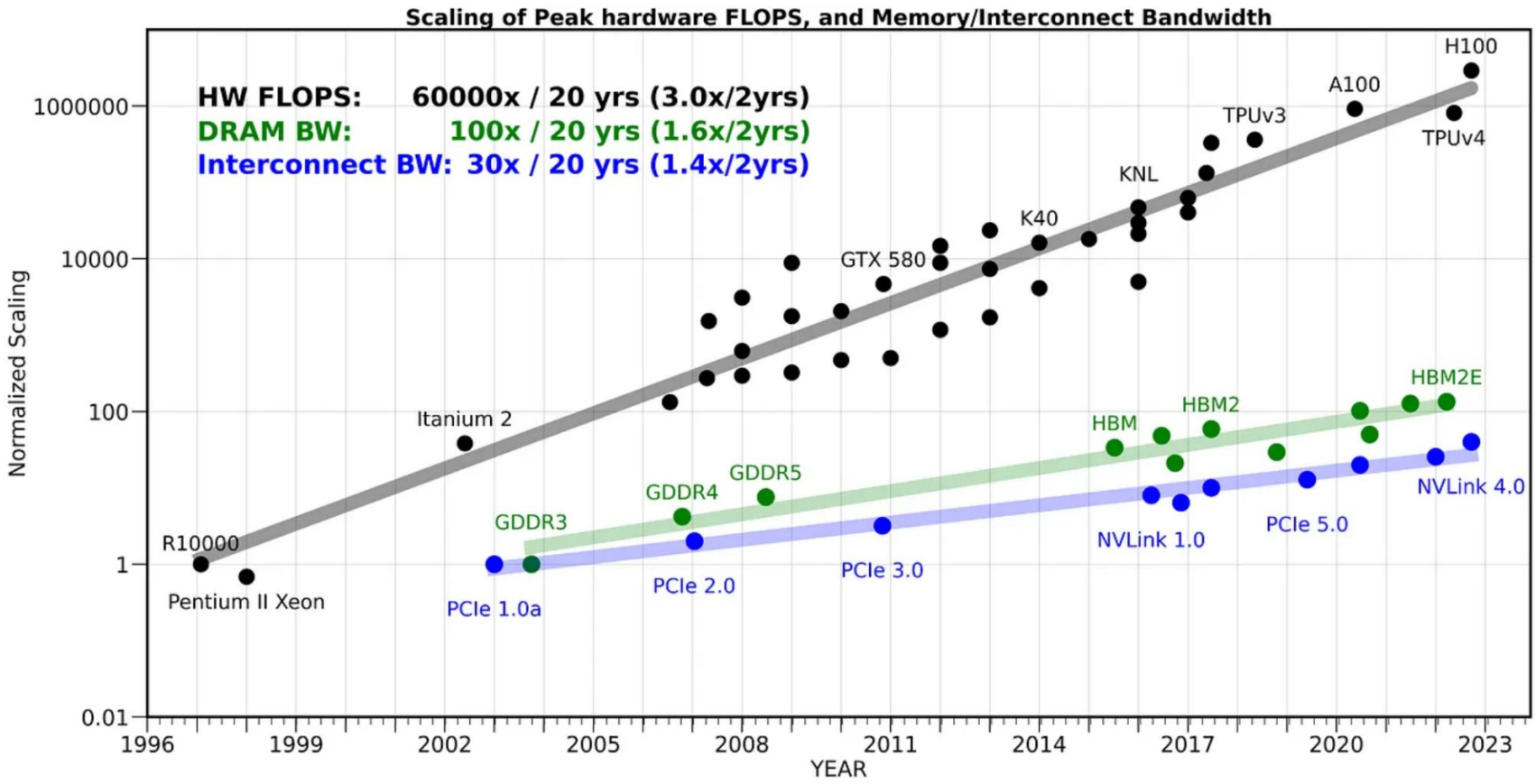
- Problem-wise:
  - Κατάλληλο πρόβλημα.
    - Υψηλός παραλληλισμός, υπολογιστικά βαρύ.
    - **Προσοχή** στη μεταφορά δεδομένων.
  - Μεγάλο μέγεθος προβλήματος.
    - Ακόμα και για *τέλεια* προβλήματα: ***επίδοση => πολλά νήματα***.
- Implementation-wise:
  - Υψηλή αξιοποίηση των SMs (high occupancy).
  - Warps χωρίς διακλαδώσεις (divergence).
  - Συνένωση αναφορών στην καθολική μνήμη (memory access coalescing).
  - Εκμετάλλευση της μοιραζόμενης μνήμης (shared memory).

Προϋποθέσεις για υψηλή επίδοση
Καταλληλότητα προβλήματος

- Υπολογισμοί SIMD (SIMT), λίγες διακλαδώσεις (ή ομαδοποιημένες).
  - Όχι περίπλοκο branching (if, αναδρομές κτλ).
  - Συνεχή δεδομένα (ή blocks μεγάλου μεγέθους).
  - Simple Problem iterators, χωρίς (πολλές) μη-συνεχείς προσβάσεις.
    - *y[i] = a[i] * b[i]* ✓
    - *y[i] = a[i - 1] * b[i % 2]* ×
- Υπολογιστικά βαριά προβλήματα (e.g. mem accesses « computations).
  - Αλλιώς επίδοση = εύρος ζώνης μνήμης (ALUs under-utilization).
  - Προβλήματα που εξαρτώνται από την επίδοση της μνήμης *δεν αξιοποιούν καλά τους πόρους* του υλικού.
    - **ΠΡΟΣΟΧΗ**: Καλύτερο απο CPU != Καλό πρόβλημα για GPU.

Προϋποθέσεις για υψηλή επίδοση

Το διάκενο μεταξύ υπολογισμού, μνήμης και δικτύου



AI and Memory Wall, Amir Gholami et al.

- Η υπολογιστική ισχύς αυξάνεται (όλο και πιο) δυσανάλογα με τα χαρακτηριστικά της μνήμης και του δικτύου διασύνδεσης.

Προϋποθέσεις για υψηλή επίδοση

Προσοχή στη μεταφορά δεδομένων

- Τα προβλήματα που απαιτούν επικοινωνία με τη CPU περιορίζονται από το δίκτυο, ακόμα και αν έχουν πολλούς υπολογισμούς.
  - Επειδή Interconnect bandwidth <<<< GPU FLOP performance.
  - Ολοένα και πιο σημαντικό πρόβλημα με την αύξηση του διακένου.
- **Παράδειγμα**: Το σύστημα *silver1* με τη GPU *Tesla-V100* πού θα γίνει η άσκηση έχει: ***BW ~ 9.8 Gbytes/s*** vs ***FP64 Perf ~ 7.2 TFlops/s***.
  - Στο χρόνο που απαιτεί 1 double να 'φτάσει' στην GPU, η GPU μπορεί να εκτελέσει ~***6000 πράξεις***.
  - ***Ή αλλιώς***: Πρέπει να κάνουμε **> *6000*** πράξεις σε κάθε δεδομένο που μεταφέρεται στην GPU, για να αξίζει η μεταφορά του.

Προϋποθέσεις για υψηλή επίδοση

Υψηλή αξιοποίηση των SMs

- Τί θέλω:
  - Μέγιστο δυνατό πλήθος από ***ενεργά warp***, για κρύψιμο του latency (max occupancy).
- Περιορισμοί:
  - Κάθε cuda kernel έχει διαφορετικές απαιτήσεις:
    - Κάθε νήμα έχει απαιτήσεις σε καταχωρητές.
      - Περιορίζει τα νήματα/SM (σε 32-άδες, άρα ***warps/SM***).
    - Κάθε block έχει απαιτήσεις σε μοιραζόμενη μνήμη.
      - Περιορίζει τα ***blocks/SM***
  - Κάθε SM υποστηρίζει ένα μέγιστο αριθμό block (***max blocks/SM)***.
    - ➢ Λιγότερα, μεγαλύτερα blocks

Προϋποθέσεις για υψηλή επίδοση

Υψηλή αξιοποίηση των SMs

- Λύση: Υπολογισμός occupancy
  - CUDA Occupancy calculator
  - ***cudaOccupancyMaxPotentialBlockSize (***

    int * minGridSize, ***// επιστροφή: ελάχιστο grid size***

    int * blockSize, ***// επιστροφή: optimal block size***

    T func, ***// το cuda kernel μας***

    size_t dynamicSMemSize = 0, ***// shared memory που θέλουμε***

    int blockSizeLimit = 0)

Προϋποθέσεις για υψηλή επίδοση

Warps χωρίς διακλαδώσεις

- Το SIMT επιτρέπει στα νήματα να ακολουθήσουν διαφορετικά if/else.
  - ***Κάθε δυνατή διακλάδωση*** εκτελείται από όλα τα νήματα του warp μαζί.
    - Απενεργοποιώντας όσα εκτελούν τον "not taken" κλάδο τους.
  - Το πλήθος των *χρήσιμων υπολογισμών* μειώνεται ανάλογα.
    - ½ για 1 divergence/warp, ⅓ γιά 2 divergence/warp κ.ο.κ.

1 divergence για το πρώτο warp κάθε thread block!

```
// Thread 0 does not enter.
if (threadIdx.x > 1) {
    // do stuff
}
```

No divergence!

```
//  Threads 0-31 do not enter
if (threadIdx.x / warpSize > 1) {
    // do stuff
}
```

Προϋποθέσεις για υψηλή επίδοση

Συνένωση αναφορών στην καθολική μνήμη

- Κρύβει την καθυστέρηση πρόσβασης (latency) στην καθολική μνήμη.
- Η καθολική μνήμη χωρίζεται σε segments των 32, 64 ή 128 byte (με βάση το Compute Capability), ευθυγραμμισμένα (aligned) στη μνήμη.
- Η πρόσβαση στην μνήμη γίνεται με συναλλαγές (transactions) ανά τμήμα.
  - Ανάγνωση/εγγραφή *word* 1, 2, 4, 8, ή 16 bytes.
  - Κάθε φορά που διαβάζουμε 1 *word,* το transaction φέρνει 1 segment.
    - Το segment γίνεται διαθέσιμο στο half-warp **για την ίδια εντολή**.
- ***Αυστηρές προϋποθέσεις*** 1) στην ακολουθία των αναφορών μνήμης ανάμεσα στα νήματα και 2) στο alignment των δεδομένων.

Προϋποθέσεις για υψηλή επίδοση

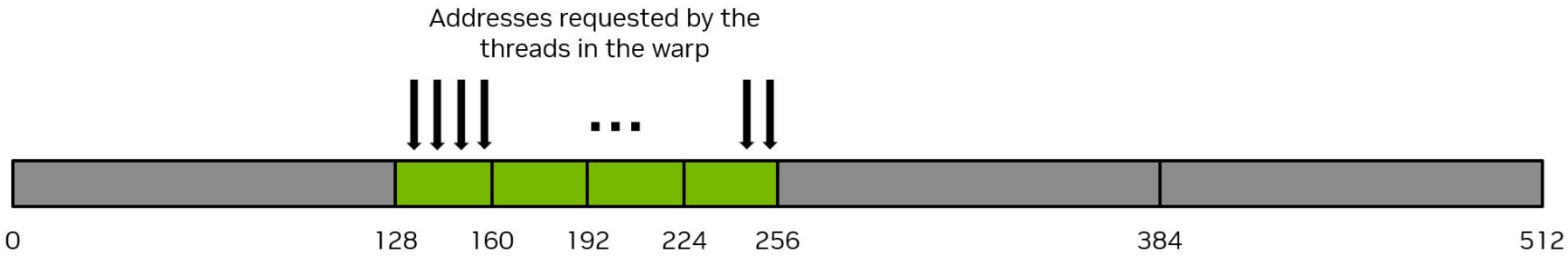
Συνένωση αναφορών στην καθολική μνήμη – Προϋποθέσεις

- Προϋποθέσεις για μέγιστο throughput (BW):
  - Τα 16 νήματα ενός half-warp θα πρέπει να προσπελαύνουν 16 λέξεις.
    - *Αλλιώς*: $partialBW = \max BW \cdot \frac{accessedWords}{16}$
  - Το k-οστό νήμα θα πρέπει να προσπελαύνει την k-οστή λέξη.
    - *Αλλιώς*: $nonseqLat = Lat \cdot \frac{cachedTransSz}{smallestTransSz}$
  - Και οι 16 λέξεις θα πρέπει να βρίσκονται στο ίδιο memory segment.
    - *Αλλιώς*: $misalignedBW = \frac{\max BW}{accessedSegments}$
- http://homepages.math.uic.edu/~jan/mcs572f16/mcs572notes/lec35.html

Προϋποθέσεις για υψηλή επίδοση

Καθολική μνήμη: Ιδανικές προσβάσεις

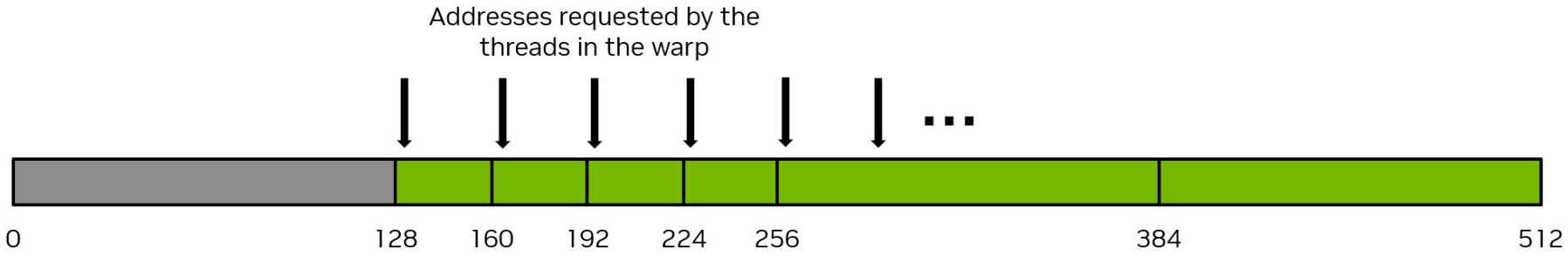
- Παράδειγμα πρόσβασης σε floats (4 byte word)
- Συνεχόμενες και ευθυγραμμισμένες



Προϋποθέσεις για υψηλή επίδοση

Καθολική μνήμη: Μη-ευθυγραμμισμένες προσβάσεις

- Χειρότερη περίπτωση: μία πρόσβαση στη μνήμη ανά 32bytes
- https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#coalesced-global-memory-access



Προϋποθέσεις για υψηλή επίδοση

Cache και μοιραζόμενη μνήμη

- On-chip memory: Πολύ γρηγορότερη πρόσβαση από την κύρια μνήμη.
  - Χρησιμοποιείται ως *hardware* ή/και *user*-managed cache.
    - Ο προγραμματιστής επιλέγει το ποσοστό του on-chip memory που θα χρησιμοποιηθεί για κάθε μια από τις δύο ανάλογα το πρόβλημα.
- Συνήθως έχουμε κάποιο ποσοστό ως user-managed cache (Shared memory)
  - Σκοπός → εκμετάλλευση της χρονικής τοπικότητας:
    - (Προ)φόρτωση δεδομένων που θα χρησιμοποιηθούν σε πολλούς υπολογισμούς και επαναχρησιμοποίηση τους.
  - Οργάνωση σε 16 ή 32 banks.
    - Διαδοχικές λέξεις σε διαδοχικά banks → παράλληλη πρόσβαση.
      - Πιό advanced : Bank alignment, bank conflicts.

Παράδειγμα 2: εσωτερικό γινόμενο διανυσμάτων

Εσωτερικό γινόμενο διανυσμάτων

Χωρίς χρήση κοινής μνήμης

```
__global__ void dot_product(float *out, const float *a, const float *b, int N){
    int tid = get_GTID();
    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, const float *a, const float *b, int N){
    int tid = get_GTID();
    if (tid >= N)
        return;
    float partial_result = a[tid] * b[tid];
    atomicAdd(out, partial_result);
}
```

Πού βρίσκεται το λάθος;

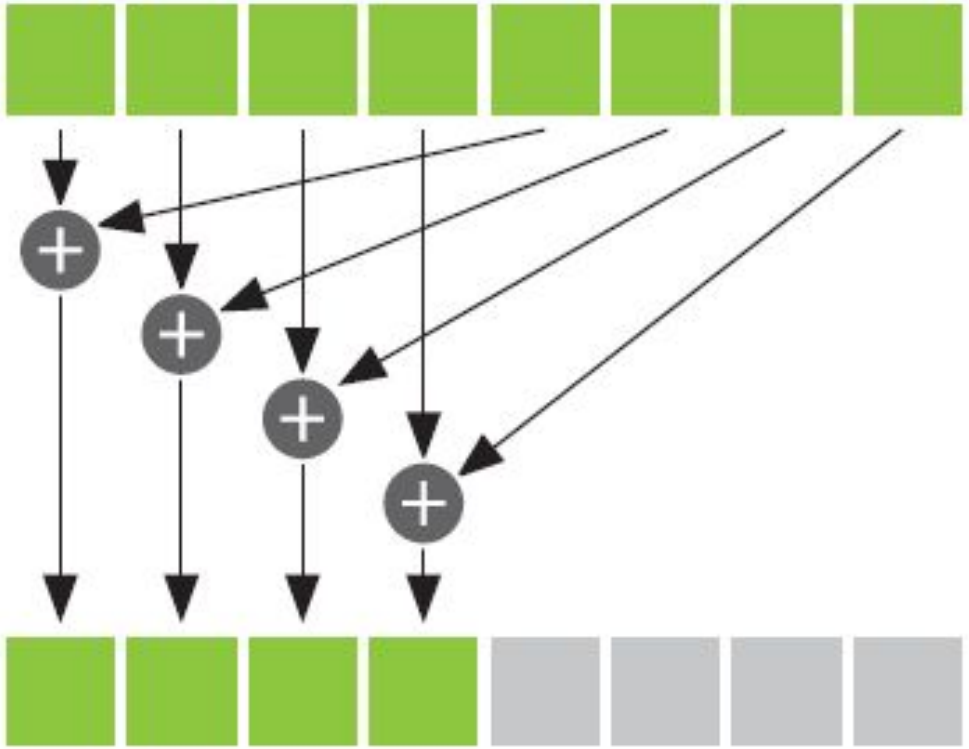
Εσωτερικό γινόμενο διανυσμάτων

Με χρήση κοινής μνήμης

```
__global__ void dot_product(float *out, const float *a, const float *b, int N){
    int tid = get_GTID(), local_tid = threadIdx.x;
    __shared__ float partials[THREADS_PER_BLOCK];
    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);
    }
}
```

Εσωτερικό γινόμενο διανυσμάτων
Reductions in CUDA

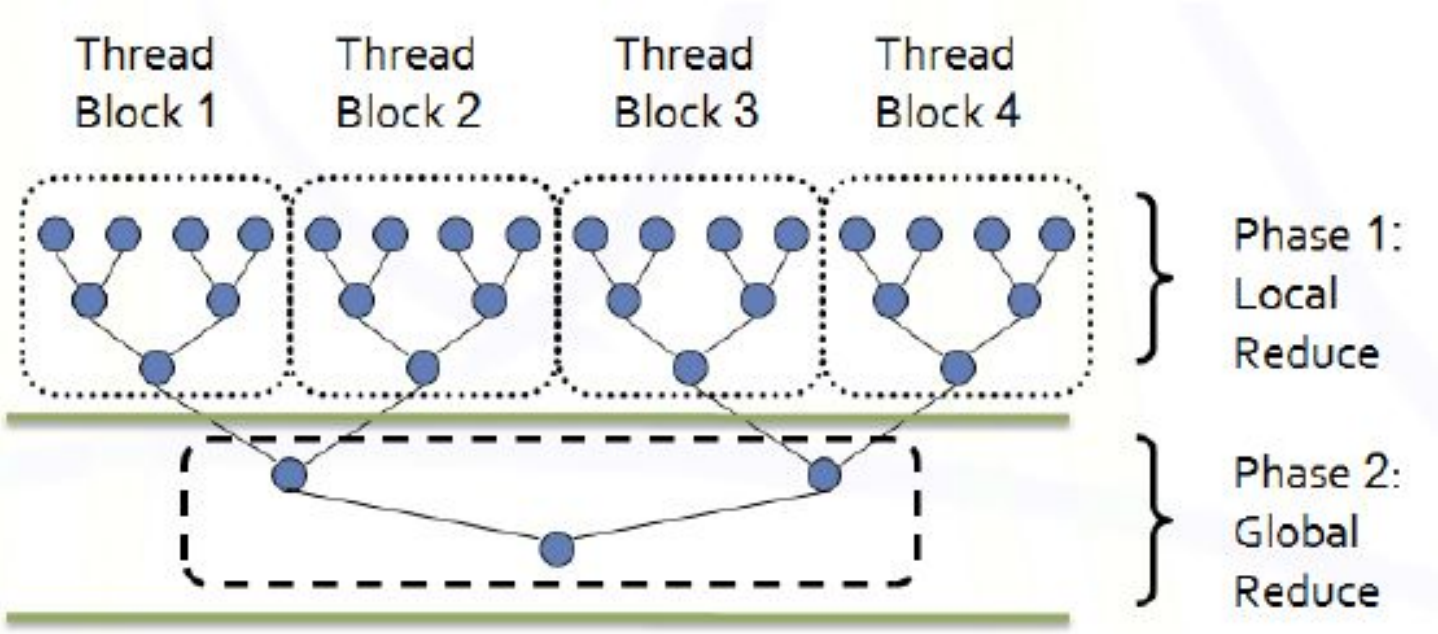
- Μπορούμε να παραλληλοποιήσουμε τον υπολογισμό του partial_result εντός του thread block!



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 κύρια μνήμη.
  - Με χρήση atomics (1/ thread block).



Πηγή: NVIDIA CUDA Programming Guide

Εσωτερικό γινόμενο διανυσμάτων

Βελτιωμένη λύση με χρήση κοινής μνήμης

```
__global__ void dot_product(float *out, const float *a, const 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]);
}
```

CUDA extras

CUDA extras

Ενοποιημένη Μνήμη (Unified Memory)

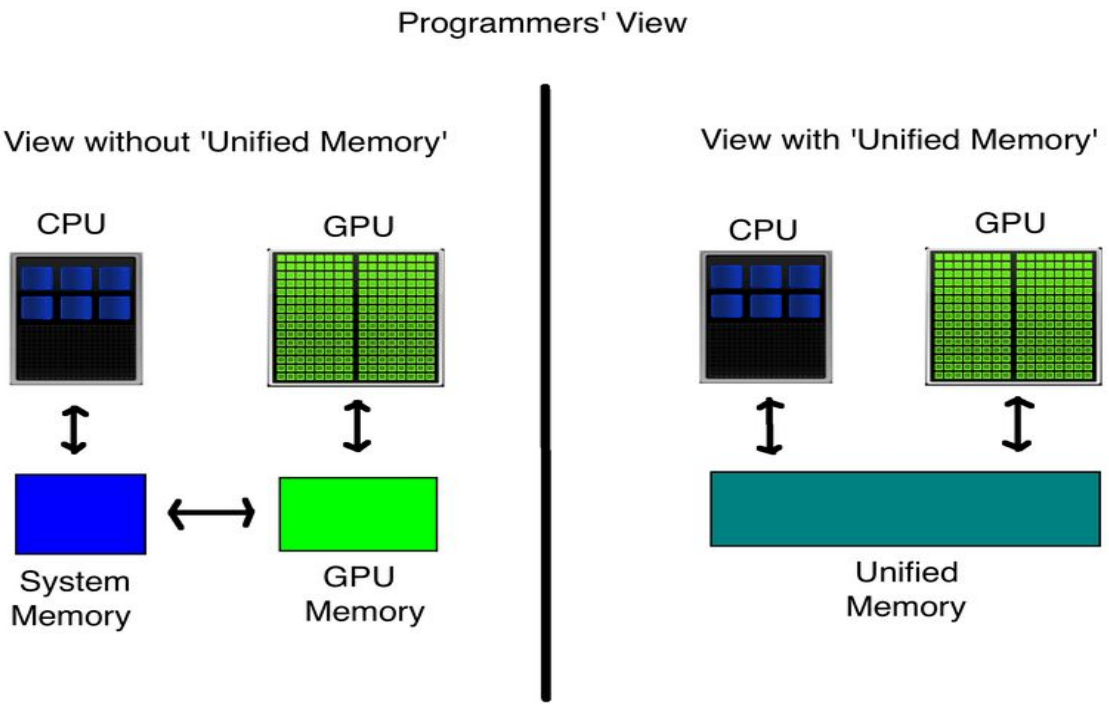
```
int main()
{
   // 1-2. Allocate - initialize a/b in host mem
   // 3. Allocate dev_a, dev_b in GPU mem
   // 4. Copy the arrays 'a' and 'b' to the GPU
   cudaMemcpy(dev_a, a, N*sizeof(float),cudaMemcpyHostToDevice);
   cudaMemcpy(dev_b, b, N*sizeof(float),cudaMemcpyHostToDevice);
   ...
   vec_add<<<grid, block>>>(dev_a, dev_b, dev_c, N);
   cudaDeviceSynchronize();

   // 5. Copy the array 'c' back from the GPU to the CPU
   cudaMemcpy(c, dev_c, N*sizeof(float),cudaMemcpyDeviceToHost);
}
```

Γιατί να επιβαρύνεται ο προγραμματιστής με τις μεταφορές δεδομένων;

CUDA extras

Ενοποιημένη Μνήμη (Unified Memory)



- Χρήση κοινού εικονικού εύρους διευθύνσεων - ο χρήστης βλέπει *μία μνήμη*.
- Το CUDA back-end υπεύθυνο για την μετατροπή διευθύνσεων.
  - Και τη μεταφορά (σελίδων) μεταξύ των δύο φυσικών μνημών.
    - **Όμως πάντα**: tradeoff → performance for programmability.

CUDA extras

Ενοποιημένη Μνήμη (Unified Memory)

```
int main() {
    float un_a[N], un_b[N], un_c[N];
    // 1. Allocate unified memory
    cudaMallocManaged((void**)&un_a, N*sizeof(float), ...);
    cudaMallocManaged((void**)&un_b, N*sizeof(float), ...);
    cudaMallocManaged((void**)&un_c, N*sizeof(float), ...);
    …
    vec_add<<<grid, block>>>(un_a, un_b, un_c, N);
    cudaDeviceSynchronize();
}
```

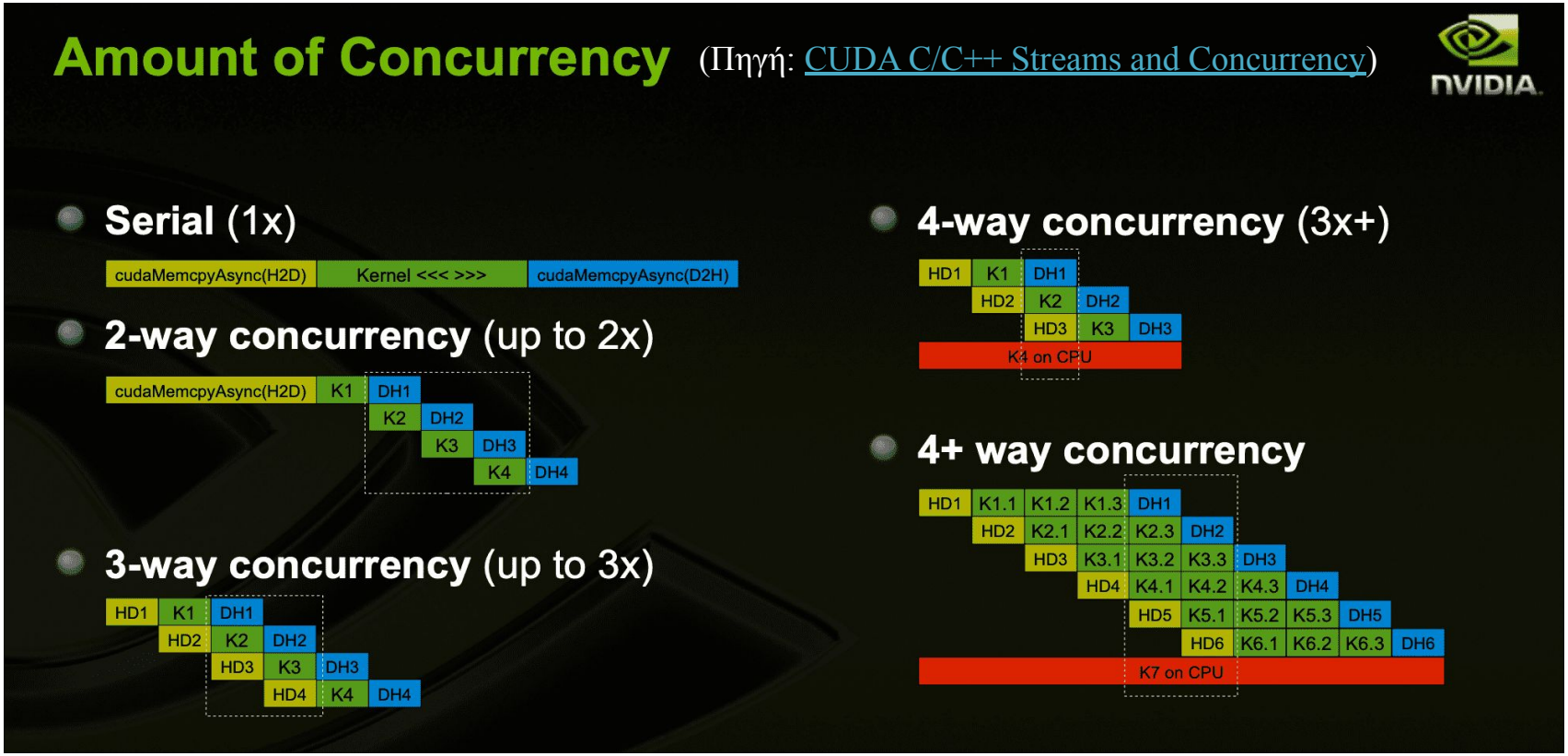
- Τα δεδομένα μεταφέρονται on-demand μεταξύ CPU-GPU.
  - Πιθανό performance bottleneck.
- Ο χρήστης μπορεί να έχει μερικό έλεγχο στην ροή τους:
  - cudaMemAdvise(), cudaMemPrefetchAsync(), … → nvidia doc

CUDA extras

Ταυτόχρονα-Παράλληλα Ρεύματα (Streams)

- Η μεταφορά δεδομένων προς/από την GPU είναι συχνά bottleneck.
  - Στην πράξη αξιοποιεί ανεξάρτητο hardware από το computation.
    - CUDA copy engines γιά μεταφορές (εξαίρεση: SM transfers).
- Το ***βασικό μοντέλο CUDA*** επιτρέπει να εκτελούμε GPU κώδικα:
  - Παράλληλα με τη CPU μέσω των ασύγχρονων kernel launch.
  - Παράλληλα με άλλες GPU, αν έχουμε πολλαπλές.
- Με χρήση ***CUDA streams*** μπορούμε να εκτελούμε ***ταυτόχρονα***:
  - Πάνω από 1 kernel/device αν περισσεύουν πόροι.
  - Ασύγχρονη μεταφορά δεδομένων Host → Device.
  - Ασύγχρονη μεταφορά δεδομένων Device → Host.
  - Ασύγχρονη μεταφορά δεδομένων Device → Device.

- Η χρήση των streams απαιτεί πρακτικά μια υλοποίηση ***software pipelining***.
- Τεμαχισμός του προβλήματος για τη δημιουργία ***υπο-προβλημάτων***.
- Τα υπο-προβλήματα δρομολογούνται *concurrently* σε διαφορετικά streams:



CUDA extras

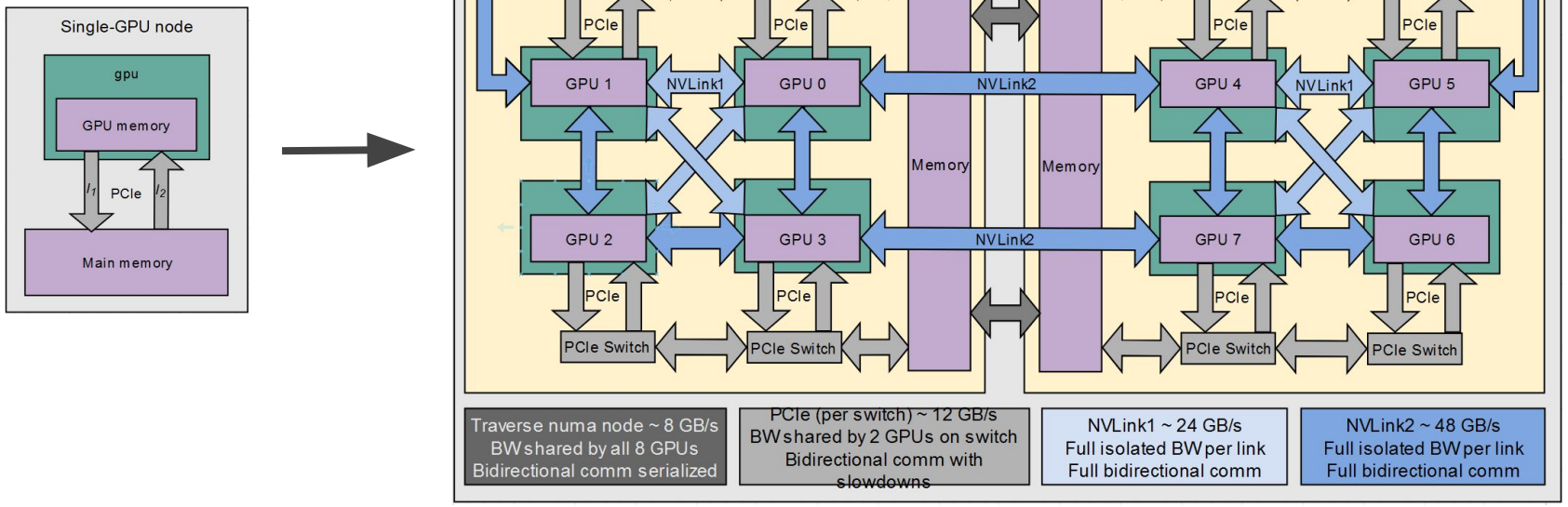
Προγραμματισμός γεγονότων (Events)

- Η χρήση διαφορετικών stream κάνει μόνο για σειριακά task dependencies.
  - Για πιο περίπλοκα workloads → Χρήση γεγονότων (*Events*)
- Κάθε event αντιστοιχεί σε μια **αλλαγή κατάστασης**.
  - Ορισμένη πάντα από τον προγραμματιστή.
    - Π.χ. έναρξη μεταφορών, τέλος μεταφορών, αρχή υπολογισμών κτλ.
- Events σε διαφορετικά streams χρησιμοποιούνται για συγχρονισμό.
  - Πρακτικά δημιουργούν (σχεδόν) οποιοδήποτε dependency task graph.
- Συναρτήσεις διαχείρισης γεγονότων:
  - cudaEventCreate(), cudaEventDestroy(), cudaEventElapsedTime()
- Για περαιτέρω μείωση του latency: ***CUDA graphs***

CUDA extras

Εκτέλεση σε πολλαπλές GPU

- Και σαν να μην έφταναν όλα αυτα…
- Τα σύγχρονα HPC συστήματα συνήθως έχουν πολλαπλές GPU *(4-8)*.
  - Πιό περίπλοκη εκτέλεση με επιπλέον ζητήματα επίδοσης:
    - Domain decomposition, scheduling, routing.



Χρήσιμα Links

Χρήσιμα links

- ***Εκτενές CUDA programming guide***
  - http://docs.nvidia.com/cuda/cuda-c-programming-guide
- ***NVIDIA Developer Zone***
  - https://developer.nvidia.com

Συζήτηση – Ερωτήσεις