Ιόνιο Πανεπιστήμιο – Τμήμα Πληροφορικής
Παράλληλος Προγραμματισμός
2024-25

Προγραμματισμός σε GPUs

(Το προγραμματιστικό μοντέλο CUDA)

https://mixstef.github.io/courses/parprog/

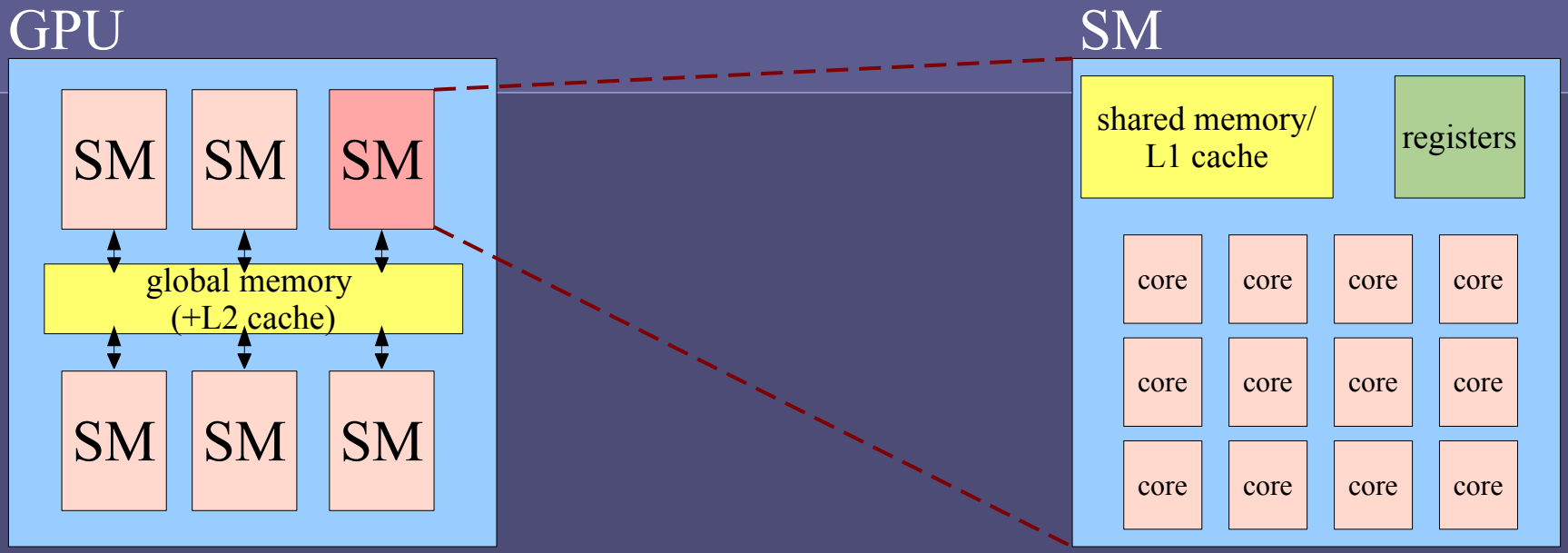
Το εργαστήριο του μαθήματος χρησιμοποιεί υπολογιστικούς πόρους AWS Cloud χρηματοδοτούμενους από το ΕΔΥΤΕ

Μ.Στεφανιδάκης

Προγραμματισμός σε GPUs: ιδιαιτερότητες

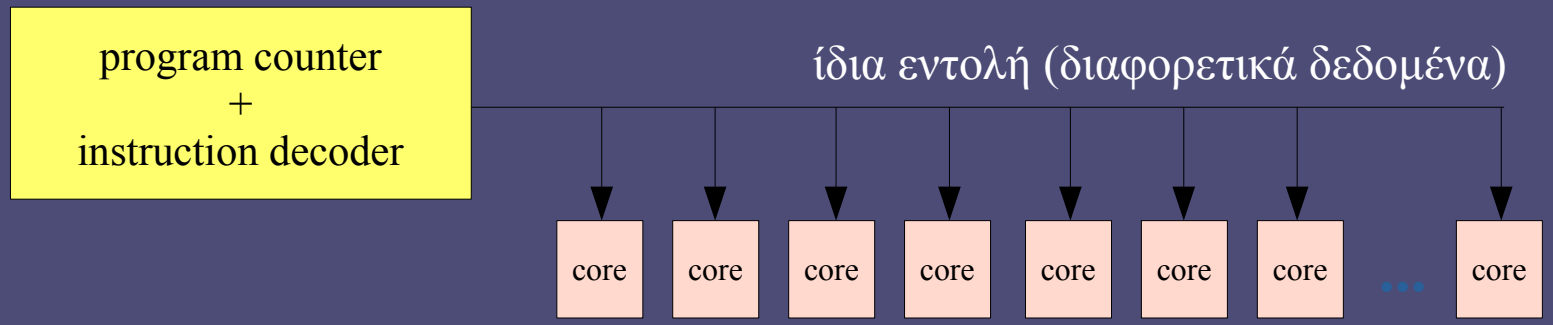
- **Πού εκτελείται το πρόγραμμα;**
  - Ένα μέρος στη CPU (**host**)
  - και ένα μέρος σε κάποια GPU/accelerator (**device**)
- **Που βρίσκονται τα δεδομένα;**
  - Στην **κύρια μνήμη** του υπολογιστή
  - ή/και σε **κάποια μνήμη** της GPU
    - Μια τυπική GPU έχει διάφορα είδη μνήμης
  - Στις μνήμες της GPU **δεν έχει πρόσβαση** (συνήθως) η CPU
    - Και το αντίστροφο
  - Συνεπώς προκύπτει η ανάγκη **μεταφοράς δεδομένων** μεταξύ διαφορετικών μνημών (CPU ↔ GPU)
    - Πρέπει να συνυπολογίζεται το **κόστος μεταφοράς** στη συνολική απόδοση

Η οργάνωση μιας τυπικής GPU



- Μια GPU αποτελείται από Streaming Multiprocessors (SMs)
  - Πρόσβαση όλων των SM σε global memory της GPU
- Κάθε SM διαθέτει
  - Έναν αριθμό υπολογιστικών cores
    - Με απλή λογική λειτουργίας, τα cores είναι σχεδιασμένα να εκτελούν μαζικά πράξεις (ίδια πράξη σε μεγάλες ομάδες cores)
  - Ένα σύνολο καταχωρητών (registers)
  - Μια γρήγορη κοινή μνήμη (shared memory)

GPUs και threads



- Στις GPUs η έννοια του hardware thread είναι διαφορετική απ' ό,τι σε μια CPU
  - Διαθέσιμος (μαζικά) μεγάλος αριθμός hardware threads (cores)
  - Παράλληλη εκτέλεση της ίδιας εντολής από πολλά threads την ίδια στιγμή (μοντέλο SIMT)
    - Οι πόροι (state) των εκτελούμενων threads βρίσκονται συνεχώς στους καταχωρητές του SM
  - Κάθε SM χρονοδρομολογεί διαδοχικά ομάδες threads με τον ίδιο program counter (PC) στα cores
    - Οι ομάδες αυτές ονομάζονται warps στην ορολογία CUDA και αποτελούνται από 32 threads με την τρέχουσα τεχνολογία

Ιδιαιτερότητες εκτέλεσης SIMT

- Τι συμβαίνει στις διακλαδώσεις;

```
if cond {
  A;
}
else {
  B;
}
```

  - Η ομάδα των threads με ίδιο PC (warp) πρέπει να εκτελεστεί δύο φορές
    - Την πρώτη φορά με ενεργοποιημένα μόνο τα threads που εκτελούν τον κώδικα A
    - και τη δεύτερη φορά με ενεργοποιημένα τα threads που εκτελούν το B

Ιδιαιτερότητες εκτέλεσης SIMT (2)

- Αν ένα thread της ομάδας δεν μπορεί να προχωρήσει (π.χ. αναμονή για πόρους);
  - Όλη η ομάδα των threads πρέπει να περιμένει
  - Το SM επιλέγει άλλο warp από threads που είναι έτοιμο να εκτελεστεί
    - Γρήγορη εναλλαγή, «κρύψιμο» της καθυστέρησης
  - Προσοχή σε barriers: πρέπει να εκτελούνται εκτός διακλαδώσεων (από όλα τα threads του warp)
  - Προσοχή σε locks: Αν συναγωνίζονται threads του ίδιου warp θα υπάρξει deadlock
- Νεώτερες αρχιτεκτονικές GPU επιτρέπουν ένα ανεξάρτητο PC για κάθε thread του warp
  - Προτιμούν όμως την «ομαδοποίηση» των threads σε κοινό PC, όσο είναι δυνατό, για μέγιστη απόδοση

Το προγραμματιστικό μοντέλο CUDA

- Ένας τρόπος «αφαιρετικής» περιγραφής του hardware (threads, warps, cores, SMs) σε ένα προγραμματιστικό interface
  - Πλαισιώνεται από τις βιβλιοθήκες και τα εργαλεία που το υλοποιούν
- Ο στόχος: να καλυφθούν με ενιαίο τρόπο GPUs με διαφορετικά χαρακτηριστικά και δυνατότητες
  - Η γνώση όμως των ιδιαιτεροτήτων του hardware εκτέλεσης είναι σε μεγάλο βαθμό αναγκαία
    - Ο προγραμματισμός σε CUDA δεν είναι εύκολος εάν θέλουμε να επιτύχουμε τη μέγιστη απόδοση...

Ορολογία CUDA

- Kernel
  - Μια συνάρτηση που περιγράφει τι θα εκτελεστεί στη GPU από κάθε ένα thread
- Thread
  - Η μικρότερη μονάδα εκτέλεσης του κώδικα του kernel
- Block (Cooperative Thread Array – CTA)
  - Μια ομάδα threads που εκτελούν τον ίδιο kernel στο ίδιο SM
    - Αν υπάρχουν ελεύθεροι πόροι (hardware), το SM μπορεί να εκτελεί και άλλα blocks παράλληλα
- Grid
  - Το σύνολο των blocks που εκτελούν τον kernel, όχι κατ' ανάγκη ταυτόχρονα
    - Σε πρόσφατες GPU υπάρχει και το cluster, μεταξύ block και grid

CUDA block

- Μια ομάδα από threads που εκτελούνται ταυτόχρονα σε ένα SM
  - Τα threads αυτά μοιράζονται τους καταχωρητές (registers) και την κοινή μνήμη (shared memory) του SM
    - Κάθε thread έχει δεσμευμένο το δικό του μέρος από τους παραπάνω πόρους όσο διαρκεί η εκτέλεση του block
    - Συνεπώς οι εναλλαγές στην εκτέλεση μεταξύ warps είναι εξαιρετικά γρήγορες
- Τα threads ενός block μπορούν να συγχρονιστούν μεταξύ τους
  - Αντιθέτως, δεν υπάρχει τρόπος να συγχρονιστούν διαφορετικά blocks μεταξύ τους
  - Ούτε μπορούμε να υποθέσουμε με ποια σειρά θα εκτελεστούν τα blocks στα SMs

Δήλωση ενός kernel

- Χρήση του keyword __global__ σε μια δήλωση συνάρτησης

```
// a kernel function - must return void
__global__ void add(int a,int b,int *c) {

    *c = a+b;
}
```

- CUDA C/C++
  - Η κλασσική C/C++ με προσθήκη επεκτάσεων (extensions) της CUDA
  - Ο compiler driver της CUDA (nvcc), στέλνει την κλασσική C/C++ στον gcc (ή αντίστοιχο) ενώ χειρίζεται διαφορετικά τις επεκτάσεις της CUDA

Λειτουργία του nvcc

- Η κλασσική C/C++ στέλνεται στον αντίστοιχο μεταγλωττιστή του host (gcc ή παρόμοιο)
- Ο κώδικας που προορίζεται για τη GPU μετασχηματίζεται σε δύο φάσεις:
  - Δημιουργία κώδικα PTX: είναι μια μορφή assembly (text) που περιγράφει μια γενική «οικογένεια» GPU (virtual architecture)
  - Δημιουργία δυαδικού κώδικα: η μορφή PTX μετατρέπεται σε δυαδικό κώδικα για μια συγκεκριμένη GPU (real architecture)
- Το τελικό εκτελέσιμο στον host (fatbinary) περιέχει το πρόγραμμα C/C++ και ενσωματώνει κάποιες από τις μορφές του κώδικα της GPU

Γιατί και PTX και εκτελέσιμος κώδικας;

- Σε αντίθεση με τις CPU, η αρχιτεκτονική και το σύνολο εντολών στις GPU αλλάζει συχνά (~ ανά 2-3 χρόνια)
- Θα πρέπει να εξασφαλιστεί η μέγιστη δυνατή συμβατότητα παλαιότερου κώδικα
  - Κώδικας PTX: γενική μορφή κώδικα, δεν χρησιμοποιεί συγκεκριμένο σύνολο εντολών μηχανής
    - Διαλέγουμε «οικογένεια» GPU ανάλογα με τις ζητούμενες δυνατότητες
    - Μπορεί να μετατραπεί σε πραγματικές εντολές μηχανής λίγο πριν την εκτέλεση (JIT) σε συγκεκριμένη GPU
    - Συμβατότητα με την επιλεγμένη «οικογένεια» GPU και κάθε νεώτερη

Γιατί και PTX και εκτελέσιμος κώδικας; (2)

- Για να αποφύγουμε την καθυστέρηση της μετατροπής σε πραγματική γλώσσα μηχανής πριν την εκτέλεση μπορούμε να προσθέσουμε στο fatbinary και εκδοχές (versions) του προγράμματος για συγκεκριμένες GPU
  - Μπορούν να χρησιμοποιηθούν μόνο για τις GPU αυτές
- Διαδικασία εκτέλεσης
  - Εάν υπάρχει διαθέσιμο εκτελέσιμο πρόγραμμα για την GPU, τότε χρησιμοποιείται αυτό
  - Αλλιώς, χρησιμοποιείται ο κώδικας PTX για να παραχθεί επιτόπου το εκτελέσιμο πρόγραμμα της GPU

CUDA kernel → PTX → SASS

CUDA source

```
// a kernel function - must return void
__global__ void add(int a,int b,int *c) {

    *c = a+b;
}
```

PTX

```
.visible .entry add(int, int, int*)(
  .param .u32 add(int, int, int*)_param_0,
  .param .u32 add(int, int, int*)_param_1,
  .param .u64 add(int, int, int*)_param_2
)
{

  ld.param.u32  %r1, [add(int, int, int*)_param_0];
  ld.param.u32  %r2, [add(int, int, int*)_param_1];
  ld.param.u64  %rd1, [add(int, int, int*)_param_2];
  cvta.to.global.u64  %rd2, %rd1;
  add.s32             %r3, %r2, %r1;
  st.global.u32       [%rd2], %r3;
  ret;
}
```

SASS (sm_52, GTX-970 κλπ)
Μορφή assembly του δυαδικού κώδικα, undocumented!

```
add(int, int, int*):
 MOV R1, c[0x0][0x20]
 MOV R0, c[0x0][0x144]
 MOV R2, c[0x0][0x148]
 MOV R3, c[0x0][0x14c]
 IADD R0, R0, c[0x0][0x140]
 STG.E [R2], R0
 NOP
 NOP
 NOP
 EXIT
.L_x_0:
 BRA `(.L_x_0)
 NOP
.L_x_1:
```

Εκτέλεση ενός kernel

- Ο προγραμματιστής ζητά την εκτέλεση ενός kernel με συγκεκριμένο αριθμό blocks ανά grid και threads ανά block
  - Τα blocks που αποτελούν το grid του kernel κατανέμονται στα SMs της GPU
- Τα SMs εκτελούν τα blocks που τους αντιστοιχούν, το ένα μετά το άλλο ή παράλληλα (ανάλογα με τους διαθέσιμους πόρους)
  - Κάθε SM χρονοδρομολογεί warps από threads του ίδιου block για εκτέλεση στα cores του

Εκκίνηση kernel από host (CPU)

- kernel_name<< blocks-per-grid , threads-per-block >>(arg, ...)

```
// this call is asynchronous - host continues execution
add<<<1,1>>>(2,7,dev_c);
```

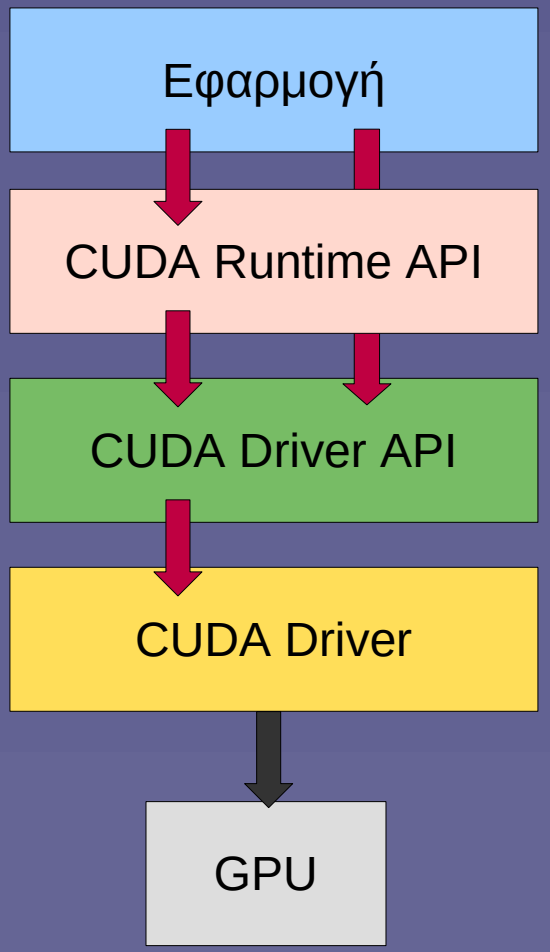
- Η εκκίνηση («κλήση») του kernel γίνεται από το κυρίως πρόγραμμα που εκτελείται στη CPU (host)
- Η κλήση είναι ασύγχρονη – η CPU συνεχίζει την εκτέλεση των επόμενων εντολών χωρίς να περιμένει την ολοκλήρωση της εκτέλεσης του kernel στην GPU

Τυπική ροή εκτέλεσης ενός kernel

- Δέσμευση μνήμης στη συσκευή GPU (device)
  - Για την υποδοχή των δεδομένων εισόδου από host
- Μεταφορά δεδομένων στη GPU
  - Στην βασική μορφή CUDA, ευθύνη του προγραμματιστή
- Εκκίνηση εκτέλεσης του kernel
  - Ανεξάρτητα και ασύγχρονα από CPU (host)
- Μεταφορά των αποτελεσμάτων πίσω στη μνήμη του υπολογιστή (host)
  - Πριν τη μεταφορά πραγματοποιείται συγχρονισμός με την ολοκλήρωση του kernel
- Αποδέσμευση μνήμης στη GPU

CUDA APIs

- CUDA Runtime API
  - Υψηλού επιπέδου API, αυτό χρησιμοποιούν συνήθως οι εφαρμογές
  - Ευκολία διαχείρισης, πολλές default ρυθμίσεις
- CUDA Driver API
  - Χαμηλού επιπέδου API
  - Λεπτομερής έλεγχος λειτουργιών, πιο δύσκολο στη χρήση



Παράδειγμα

```
int c;  // host's (CPU) 'c' variable
int *dev_c;    // ptr to device's (GPU) 'c' variable

// allocate space for 'c' on device's memory
cudaMalloc((void **)&dev_c,sizeof(int)))

// call the kernel on device, 1 block/1 thread
// this call is asynchronous - host continues execution
add<<<1,1>>>(2,7,dev_c);

// transfer device's 'c' into host's 'c' - synchronous call,
waits until kernel is done
cudaMemcpy(&c,dev_c,sizeof(int),cudaMemcpyDeviceToHost);

// free memory of device's c
cudaFree(dev_c);
```

- Προσοχή: Οι συναρτήσεις της CUDA επιστρέφουν ένδειξη επιτυχίας η όχι – σε κανονικό πρόγραμμα πρέπει να ελέγχουμε εάν κάθε κλήση ήταν επιτυχής

Συναρτήσεις (CUDA Runtime API)

- `cudaError_t cudaMalloc(void** devPtr, size_t size)`
  - Δεσμεύει χώρο στην GPU (device) μεγέθους `size` και επιστρέφει έναν δείκτη στο `devPtr`
- `cudaError_t cudaFree(void* devPtr)`
  - Αποδέσμευση της μνήμης στη GPU που δείχνει το `devPtr`
- `cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind)`
  - Μεταφορά δεδομένων μεγέθους `count` από το `src` στο `dst`.
  - Το `cudaMemcpyKind` ρυθμίζει το είδος της μεταφοράς
    - Στα βασικά παραδείγματα χρησιμοποιούμε τα `cudaMemcpyHostToDevice` και `cudaMemcpyDeviceToHost`
  - Η συνάρτηση είναι σύγχρονη: θα επιστρέψει μετά την ολοκλήρωση της μεταφοράς
  - Στην GPU οι μεταφορές και η εκτέλεση των kernels είναι σειριοποιημένες με τη σειρά κλήσης (default stream)

Σφάλματα του kernel

- Δεν υπάρχει τρόπος να ειδοποιηθεί η CPU (host) για σφάλματα κατά την εκκίνηση ή εκτέλεση ενός kernel
  - Εκτελείται ασύγχρονα, σε διαφορετικό device (GPU)
- Πώς θα ελέγξουμε αν υπήρξαν σφάλματα στον kernel;
  - Σε αντίθεση με τις συναρτήσεις των CUDA APIs, οι οποίες επιστρέφουν ένα error code
- Μπορούμε να ελεγξουμε αν υπήρξαν σφάλματα μετά την ολοκλήρωση του kernel

```
// a catchall msg here, will catch kernel launch failures, too!
printf("Last error msg is: %s\n",
       cudaGetErrorString( cudaGetLastError() ));
```

Παραμετρική εκτέλεση

- Πώς γνωρίζει κάθε thread ποιο μέρος της συνολικής εργασίας θα εκτελέσει
  - Κάθε thread έχει διαθέσιμες κατά την εκτέλεση του kernel μια σειρά από μεταβλητές index (read-only)
  - Οι μεταβλητές αυτές μπορούν να οργανωθούν σε 1, 2 ή 3 διαστάσεις
    - Προγραμματιστική ευκολία, για να ταιριάζουν με το είδος (και την τοπικότητα των δεδομένων) της εφαρμογής
    - Δεν αλλάζει η υποκείμενη οργάνωση του hardware σε threads/blocks/grid

Οργάνωση σε μία διάσταση

- threadIdx.x
  - «Ποιο thread του block είμαι»
- blockIdx.x
  - «Σε ποιο block ανήκω»
- blockDim.x
  - «Πόσα threads υπάρχουν σε κάθε block»
- gridDim.x
  - «Πόσα blocks υπάρχουν στο grid»
- Σε οργανώσεις με 2 ή 3 διαστάσεις υπάρχουν επιπλέον τα .y και .z των παραπάνω

Περιορισμοί διαστάσεων

- /usr/local/cuda-12.2/extras/demo_suite/deviceQuery

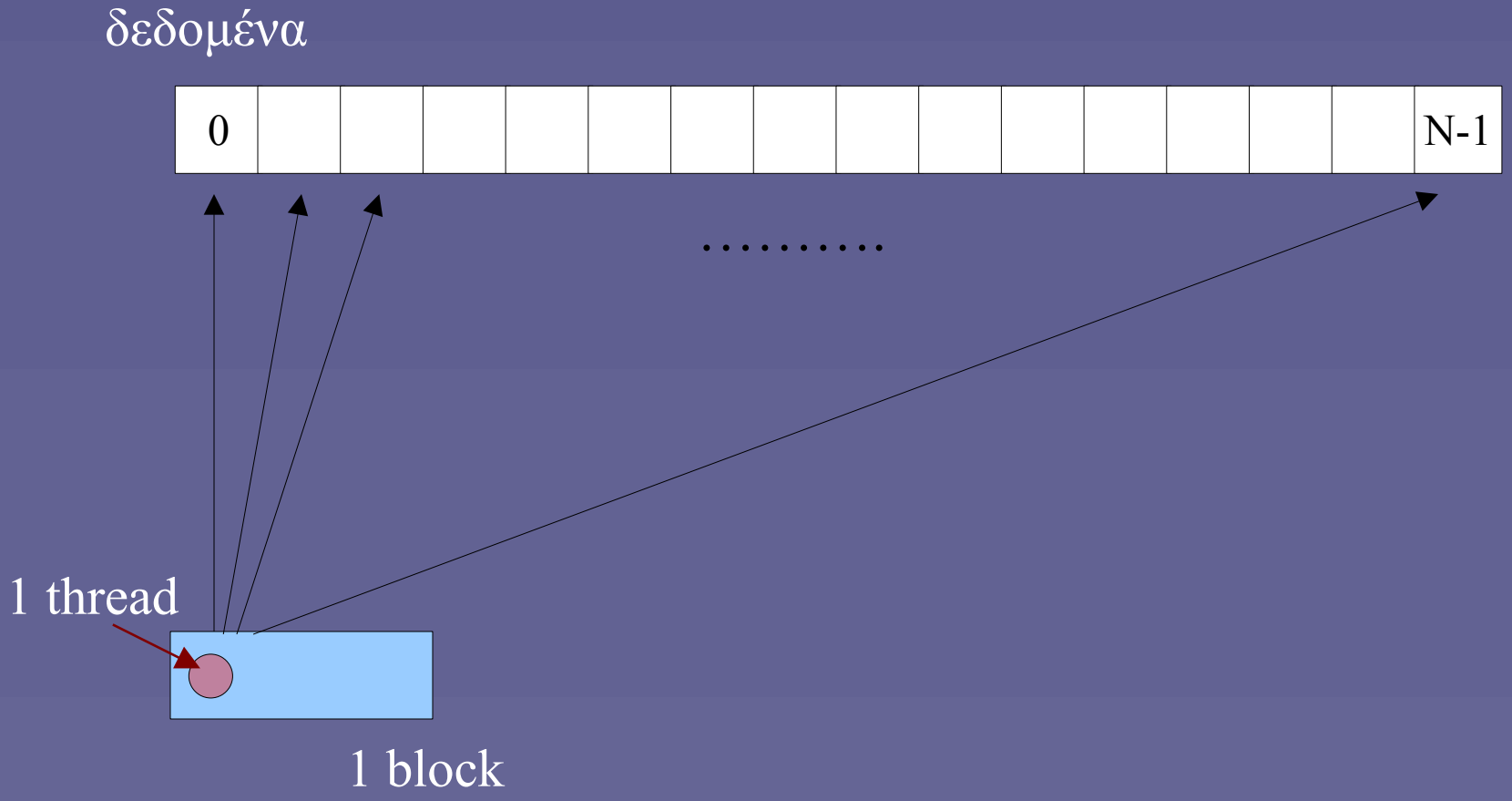
```
  Maximum number of threads per multiprocessor:  2048

  Maximum number of threads per block:           1024

  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)

  Max dimension size of a grid size    (x,y,z): (2147483647,
65535, 65535)
```

- Κάθε μοντέλο GPU έχει περιορισμούς στον μέγιστο αριθμό threads και blocks ανά διάσταση, όπως και στο πόσα threads/block ή blocks/grid μπορούν να εκτελεστούν

Παραδείγματα κατανομής εργασίας

- Έστω ότι εκτελούμε απλό μετασχηματισμό (map) σε Ν στοιχεία εισόδου
  - Πώς κατανέμω την εργασία;
- Λύση(;) #1: Ένα και μοναδικό thread εκτελεί τα πάντα
  - Το γνωστό loop `for(i=0;i<N;i++)`
  - Δεν είναι λύση στην πραγματικότητα (αδικαιολόγητη σπατάλη των διαθέσιμων πόρων της GPU...)

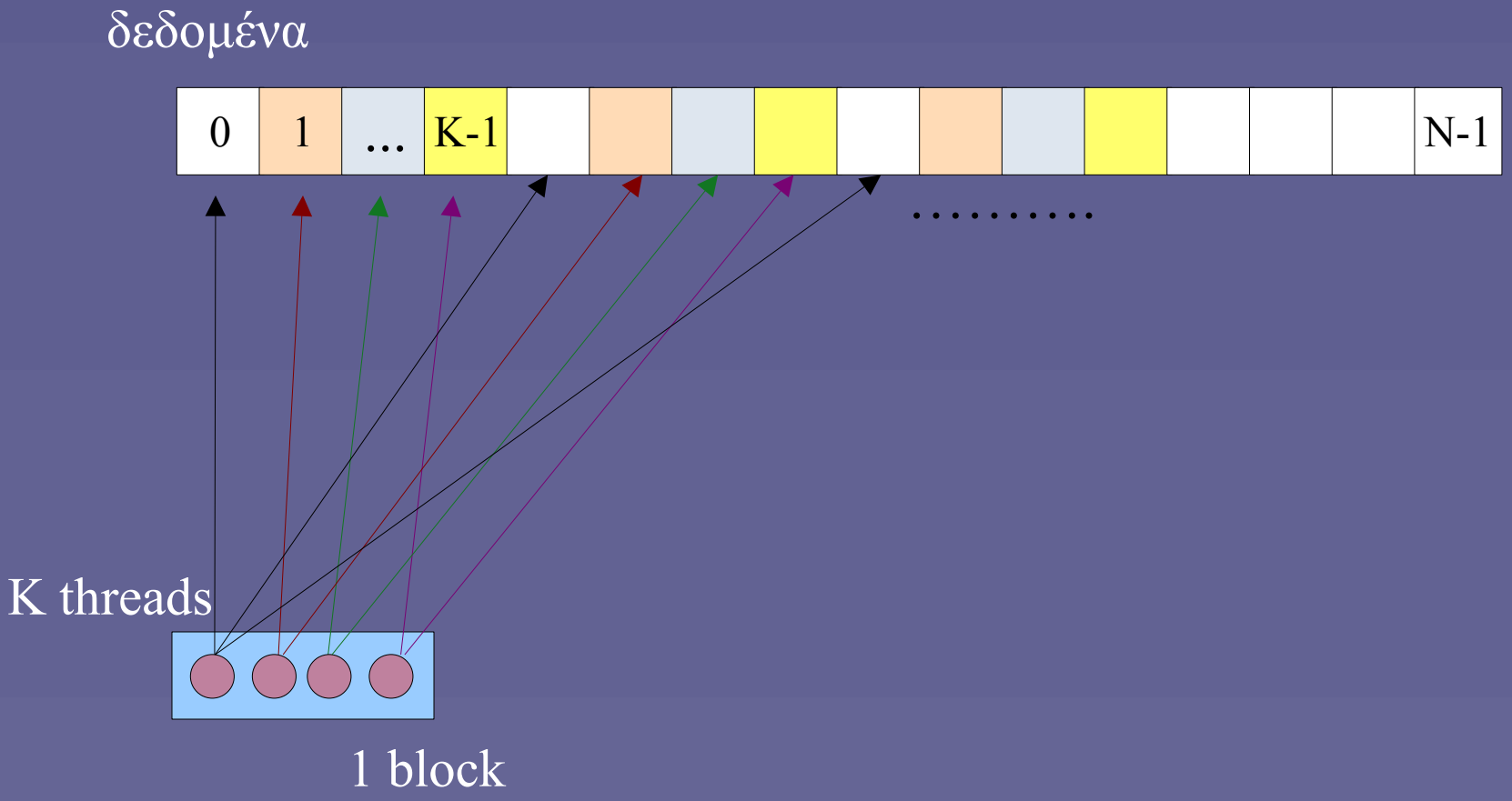
Λύση(;) #1



Παραδείγματα κατανομής εργασίας (2)

- Λύση(;) #2: Ένα block με K threads
  - Πόσο θα είναι το K;
    - Συνήθως ένα (μικρό) πολλαπλάσιο του 32 (warp size)
    - Π.χ. με K = 256 θα έχουμε 8 warps προς εκτέλεση, ικανά να «κρύψουν» καθυστερήσεις σε κάποιο/α από αυτά
  - Πώς καλύπτουμε τα N στοιχεία εισόδου;
    - Striding
    - Το thread0 χειρίζεται τα στοιχεία 0, 256, 512, …
    - Το thread1 χειρίζεται τα στοιχεία 1, 257, 513, …
    - Το thread2 χειρίζεται τα στοιχεία 2, 258, 514, …
  - Η ομοιομορφία στην προσπέλαση είναι απαραίτητη για την υψηλή απόδοση
  - Όμως με ένα ενεργό block, χρησιμοποιούμε μόνο ένα SM...

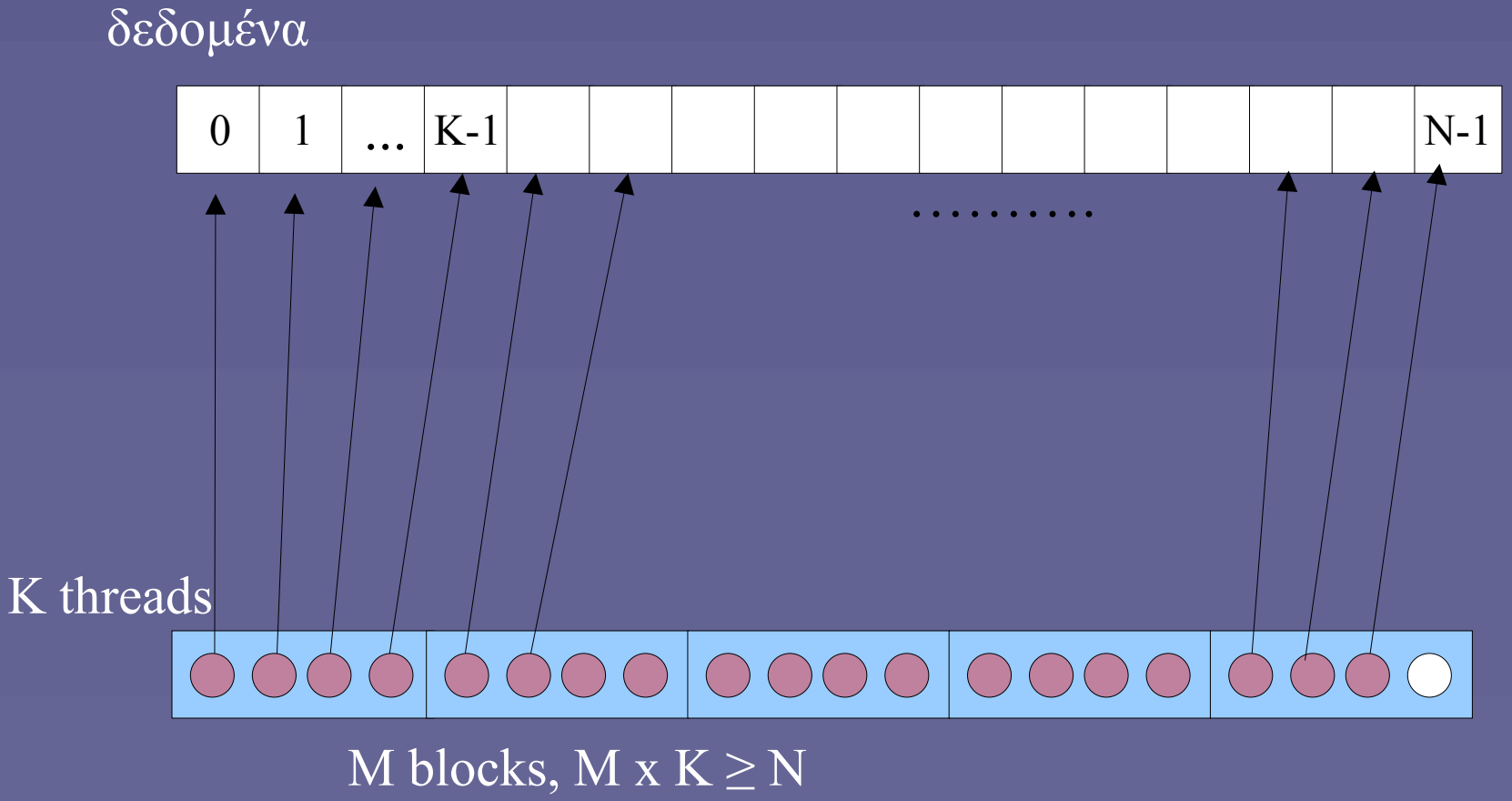
Λύση(;) #2



Παραδείγματα κατανομής εργασίας (3)

- Λύση #3: M blocks με K threads το καθένα
  - Το M πρέπει να είναι τέτοιο ώστε M x K ≥ N
    - Κάθε thread υπολογίζει ένα (ή κανένα) στοιχείο μόνο!
  - Υπολογισμός του M
    - Το γνωστό (N + K - 1) / K
    - Κρατώντας το K = 256
  - Ποιο στοιχείο χειρίζεται κάθε thread;
    - i = αριθμός-block * μέγεθος-block + αριθμός thread

Λύση #3



Παραδείγματα κατανομής εργασίας (4)

- Λύση #4: M blocks με K threads το καθένα
  - Το M είναι σταθερό, π.χ. 32 φορές τον αρθμό των SM
    - Κάθε thread υπολογίζει πολλαπλά στοιχεία

```
int threads = 256;
int devId;
cudaGetDevice(&devId);
int numSM;
cudaDeviceGetAttribute(&numSM, cudaDevAttrMultiProcessorCount,
devId);
int blocks = numSM*32; // as a multiple of SMs in GPU
```

  - Ποια στοιχεία χειρίζεται κάθε thread;
    - start = αριθμός-block * μέγεθος-block + αριθμός thread
  - Επανάληψη με stride = συνολικό αριθμό threads στο grid
    - δηλ. μέγεθος block (σε threads) * μέγεθος grid (σε blocks)

Λύση #4