ΠΑΝΕΠΙΣΤΗΜΙΟ
ΔΥΤΙΚΗΣ ΜΑΚΕΔΟΝΙΑΣ

Συστήματα Παράλληλης και Κατανεμημένης Επεξεργασίας

Ενότητα: ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ No:13

Δρ. Μηνάς Δασυγένης

mdasyg@ieee.org

Τμήμα Μηχανικών Πληροφορικής και Τηλεπικοινωνιών

Εργαστήριο Ψηφιακών Συστημάτων και Αρχιτεκτονικής Υπολογιστών

http://arch.icte.uowm.gr/mdasyg

Άδειες Χρήσης

- Το παρόν εκπαιδευτικό υλικό υπόκειται σε άδειες χρήσης Creative Commons.
- Για εκπαιδευτικό υλικό, όπως εικόνες, που υπόκειται σε άλλου τύπου άδειας χρήσης, η άδεια χρήσης αναφέρεται ρητώς.

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ψηφιακά Μαθήματα του Πανεπιστημίου Δυτικής Μακεδονίας**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Περιεχόμενα

1. Σκοπός της άσκησης

- Ανάπτυξη Παράλληλων εφαρμογών σε GPU επεξεργαστές (CUDA)[1] #1.

2. Παραδοτέα

(A) 4 ερωτήσεις

(C) 4 ασκήσεις

Με τον όρο CUDA, η Nvidia αναφέρεται σε μία αρχιτεκτονική παράλληλου υπολογισμού *(Compute Unified Device Architecture)*, η οποία έχει ως στόχο να εκμεταλλευθεί την εξαιρετική δύναμη των επεξεργαστών γραφικών της εταιρείας, για εργασίες που δεν σχετίζονται με την απεικόνιση γραφικών.

Σε πιο τεχνικό επίπεδο, θα λέγαμε ότι η αρχιτεκτονική CUDA αποτελεί μία μικρή επέκταση της γλώσσας C, βάσει της οποίας οι προγραμματιστές αποκτούν πρόσβαση στις δυνατότητες της κάρτας γραφικών και έτσι μπορούν να εκμεταλλευθούν την ισχύ της για την επίλυση διαφόρων προβλημάτων, συνήθως επιστημονικής φύσης. Αξίζει να σημειωθεί ότι η Nvidia δίνει τη δυνατότητα στους προγραμματιστές να εκμεταλλευθούν παράλληλα και τον κεντρικό επεξεργαστή του συστήματος *(όπως γίνεται άλλωστε με κάθε γλώσσα προγραμματισμού)* και έτσι να μπορούν να μοιράσουν τον φόρτο εργασίας του προγράμματός τους, τόσο στη CPU *(Central Processing Unit)*, όσο και στη GPU *(Graphics Processing Unit).*

Τι είδους εφαρμογές όμως είναι αυτές που θα μπορούν να εκμεταλλευτούν τη χρήση της αρχιτεκτονικής CUDA αλλά και της ισχύος των chips που φιλοξενούνται στις αντίστοιχες κάρτες γραφικών; Μπορείτε να πάρετε μία πρώτη γεύση στο site, αλλά σε γενικές γραμμές οι εν λόγω εφαρμογές έχουν επιστημονικό χαρακτήρα και απαιτούν πολύ μεγάλη επεξεργαστική ισχύ. Για παράδειγμα, η ανάλυση πρωτεϊνών στα αμινοξέα από τα οποία αποτελούνται, ο υπολογισμός καιρικών φαινομένων, η κίνηση υγρών, η ανάλυση χρηματιστηριακών συναλλαγών, οι προσομοιώσεις του ανθρώπινου εγκεφάλου, η κίνηση ουράνιων σωμάτων κ.λπ. είναι μερικές από τις περιπτώσεις στις οποίες η αρχιτεκτονική CUDA μπορεί να βρει εφαρμογή.

Θα πρέπει να σημειώσουμε πάντως, ότι είναι πολύ διαφορετικό το να δημιουργήσει κανείς μία εφαρμογή με τη βοήθεια της αρχιτεκτονικής CUDA, και πολύ διαφορετικό με το απλά να συμμετάσχει σε ένα CUDA project. Στην πρώτη περίπτωση θα πρέπει να διαθέτει αυξημένες γνώσεις προγραμματισμού καθώς φυσικά και αντίστοιχες γνώσεις επί του αντικειμένου που θέλει να μελετήσει, ενώ στη δεύτερη περίπτωση χρειάζεται απλά την κατάλληλη εφαρμογή και ένα σύστημα με κάρτα γραφικών που είναι συμβατή με την αρχιτεκτονική CUDA.

[1] Για τη δημιουργία του εργαστηριακού φυλλαδίου, βοήθησε ο φοιτητής Κεχαγιάς Απόστολος.

3. Απαιτήσεις συστήματος για χρήση της CUDA

- CUDA-enabled GPU
- Windows, Linux ή Mac
- Device driver
- CUDA software (διαθέσιμο δωρεάν από http://www.nvidia.com/cuda)
- Microsoft Visual Studio 2005 or 2008, ή Microsoft Visual C++ Express ή gcc

4. Εγκατάσταση Βιβλιοθηκών

(τα βήματα της εγκατάστασης έχουν γίνει στο εργαστήριο, οπότε καταγράφονται μόνο για λόγους πληρότητας)

Μεταβείτε στην παρακάτω σελίδα:
http://developer.nvidia.com/object/cuda_3_2_downloads.html
Βρίσκουμε τα παρακάτω αρχεία και τα κατεβάζουμε. Υπάρχουν διαθέσιμά για Windows linux και mac.

Windows

- Developer Drivers for Win (263.06)
- CUDA Toolkit
- GPU Computing SDK code samples
- Compiler

Linux

- Developer Drivers for Linux (260.19.21)
- CUDA Toolkit
- GPU Computing SDK code samples

Mac

- Developer Drivers for MacOS
- CUDA Toolkit
- GPU Computing SDK code samples

Το πακέτο GPU Computing SDK code samples περιέχει κάποια παραδείγματα χρήσης όποτε είναι προαιρετικό, αλλά θα το χρησιμοποιήσουμε στην συνέχεια για να δούμε αν έχουν γίνει όλα σωστά στην εγκατάσταση και να δούμε κάποια παραδείγματα.

5. Εγκατάσταση

5.1 Linux

Αλλάζουμε το directory που έχουμε κατεβάσει τα αρχεία.

```
cd ~/Downloads
```

βεβαιώνουμε ότι τα αρχεία είναι εκτελέσιμα:

```
chmod +x devdriver_*.run
chmod +x cudatoolkit_*.run
chmod +x gpucomputingsdk_*.run
```

Για να κάνουμε install τον devdriver πρέπει να σταματήσουμε τον xserver

```
sudo /etc/init.d/gdm stop
```

ή

```
sudo /etc/init.d/kdm stop
```

ή

```
sudo /etc/init.d/xdm stop
```

Σε αυτό το σημείο μπορεί να ζητηθεί ξανά να γίνει login.

```
sudo ./devdriver_*.run
sudo  ./cudatoolkit_*.run
 ./gpucomputingsdk_*.run
```

αφού τελειώσουμε με επιτυχία τα παραπάνω, δίνουμε

```
startx
```

προσθέτουμε στο path τις απαραίτητες βιβλιοθήκες και εκτελέσιμα δίνοντας

```
echo "
PATH=\$PATH:/usr/local/cuda/bin
LD_LIBRARY_PATH=\$LD_LIBRARY_PATH:/usr/local/cuda/lib
export PATH
export LD_LIBRARY_PATH" >> ~/.profile
```

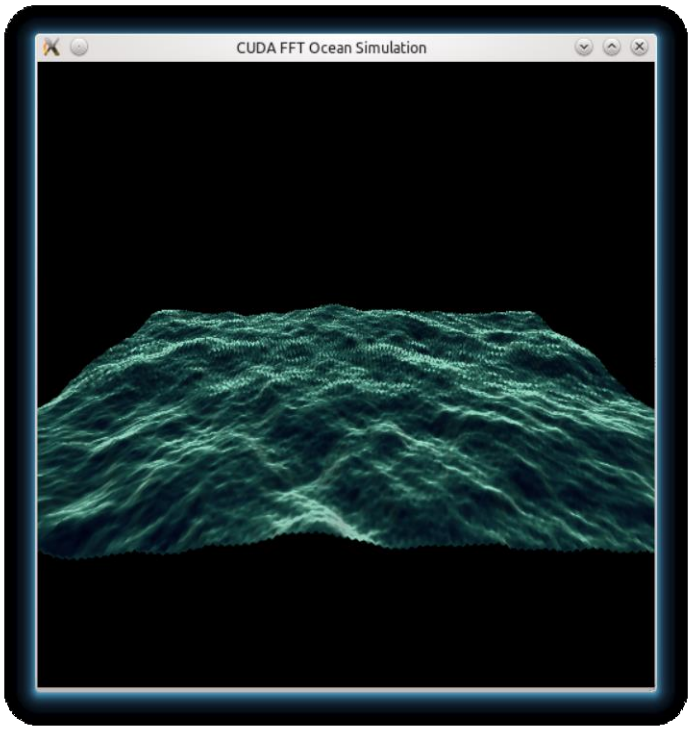
κάνουμε logout και ξανά login για να ενεργοποιηθούν οι ρυθμίσεις
αλλάζουμε directory στα παραδείγματα

```
cd ~/NVIDIA_GPU_Computing_SDK/C
```

και κάνουμε compile δίνοντας

```
make[2]
```

[2] Το compile μπορεί να αποτύχει γιατί υπάρχει περίπτωση στην διανομή σας να μην υπάρχουν οι απαραίτητες βιβλιοθήκες. Σε Ubuntu για παράδειγμα είναι απαιτούμενα τα build-essential, freeglut3-dev, libxmu-dev



Εικόνα: running oceanFFT sample

Τα προγράμματα έχουν κατάληξη .cu και αντί του gcc χρησιμοποιούμε το nvcc για compile.

π.χ

```
nvcc -o my_cuda my_cuda.cu
```

5.2 Windows

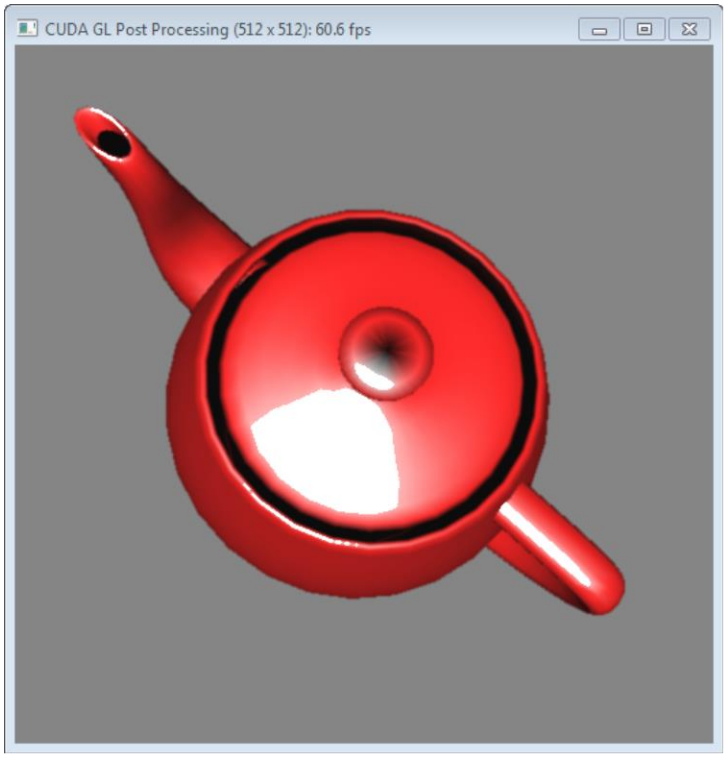
Για Windows θα πρέπει να χρησιμοποιήσουμε τον compiler του Visual Studio 2008 ή 2005 γιατί ο compiler του 2010 δεν υποστηρίζεται πλήρως ακόμη. Επίσης δεν είναι απαραίτητο να έχουμε εγκατεστημένο το Visual Studio. Αρκεί ο command line compiler *(cl)* που κατεβάσαμε προηγουμένως.

Για χρήση με το Visual Studio χρειάζεται και το parallel Nsight *(είναι free αλλά χρειάζεται register)*.[3]

Εγκαθιστούμε τον driver, το cuda toolkit τον compiler και το gpu computing code samples όπως όλες τις windows εφαρμογές *(next, next, ok, finish)*.

Εδώ κάνουμε restart για να φορτωθεί ο καινούριος driver. Αν έχουν γίνει όλα σωστά θα πρέπει να λειτουργούν τα samples που εγκαταστήσαμε. Αν τρέξουμε το NVIDIA GPU Computing SDK 3.2 Browser βλέπουμε όλα τα διαθέσιμα παραδείγματα.

[3] περισσότερες πληροφορίες http://forums.nvidia.com/index.php?showtopic=184539



Εικόνα : running openGL sample

Πλέον μπορούμε να κάνουμε compile μέσω command line.

π.χ

```
nvcc -o my_win_cuda my_win_cuda.cu
```

ή μέσω Visual Studio αν έχουμε ακολουθήσει τις παραπάνω οδηγίες.

5.3 Mac

Με παρόμοιες διαδικασίες γίνεται η εγκατάσταση σε MAC.

Links:

- Forum: http://forums.nvidia.com/index.php?s=45e13dc70535606c5898aa318916d78b&showforum= 75
- Install guide: http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/CUDA_Getting_Started_2.2_Mac_OS.pdf

Σημείωση: Μπορείτε να χρησιμοποιήσετε τα μηχανήματα CUDA με το ιδρυματικό σας λογαριασμό. Συνδεθείτε στο φοιτητικό server και ενεργοποιήσετε το account σας στα μηχανήματα CUDA του Τμήματος με την εντολή “cuda_enable”. Στη συνέχεια συνδεθείτε στις IP που αναγράφονται κατά την εκτέλεση του cuda_enable με τον

ιδρυματικό σας κωδικό. Επίσης, μπορείτε να εκτελέσετε το `ssh-keygen` *(πατάτε συνεχώς enter)* στο zafora, ώστε να δημιουργήσετε δυο κλειδιά, και να δώσετε: `mv ~/.ssh/id_rsa.pub ~/.ssh/authorized_keys` ώστε να συνδέεστε χωρίς κωδικό στα CUDA από το zafora, δεδομένου ότι όλα τα μηχανήματα είναι συνδεδεμένα στο ίδιο NFS User Home Directory.

6. Το πρώτο πρόγραμμα: Hello World

Το κλασσικό πρόγραμμα hello world δεν έχει κάποια ιδιαίτερη αξία για την παραλληλία αλλά είναι σημαντικό για να ξέρουμε ότι όντως μπορούμε να κάνουμε compile.

```
#include <stdio.h>
int main(int argc, char **argv)
{
      printf("Hello World\n");
      return 0;
}
```

6.1 Το παραδοτέο Α1

(A1) Κάντε compile στο μηχάνημα CUDA τον παραπάνω κώδικα και δώστε το screenshot εκτέλεσης, αφού χρησιμοποιήσετε το όνομα **a1.c** .

Αφού κάνουμε compile και βεβαιωθούμε ότι λειτουργεί σωστά πρέπει να σκεφτούμε πως θα μπορούσε το hello world να παραλληλοποιηθεί. Μια σκέψη για το πώς μπορεί να γίνει αυτό, φαίνεται παρακάτω.

```
#include <stdio.h>
#include <stdlib.h>
#define N 13
__global__ void hello(char* c)
{
      char msg[N] = "Hello World!";
      int idx = blockIdx.x;
            c[idx] = msg[idx];
}
int main(int argc, char **argv)
{
      int i;
      char *d_msg;
      char *msg;
      size_t hello_size;
      hello_size = N*sizeof(char);
      msg = (char *)malloc(hello_size);
      cudaMalloc((void **)&d_msg,hello_size);
      hello<<<13,1>>>(d_msg);
      cudaMemcpy(msg,d_msg,hello_size,cudaMemcpyDeviceToHost);
```

```
        for(i=0;i<N-1;i++) printf("%c",*(msg+i));
        printf("\n");
        cudaFree(d_msg);
        free(msg);
        return 0;
}
```

***Να σημειώσετε ότι: ***Σε περίπτωση που κατά την εκτέλεση του κώδικα σας αναφερθεί το μήνυμα λάθους *(ή κάτι παρόμοιο)*: `"error while loading shared libraries: libcudart.so.5.0: cannot open shared object file: No such file or directory"` τότε αυτό σημαίνει ότι δεν έχετε τροποποιήσει το LD_LIBRARY_PATH σύμφωνα με τις οδηγίες που αναφέρθηκαν προηγουμένως. Μπορείτε να το λύσετε με δυο τρόπους:

- είτε τροποποιείτε το LD_LIBRARY_PATH όπως προηγουμένως, ή
- πριν από κάθε εκτέλεση χρησιμοποιείτε την ειδική μεταβλητή περιβάλλοντος LD_PRELOAD στην οποία προ-φορτώνετε τις βιβλιοθήκες που θα χρειαστούν. Για παράδειγμα, αν θέλουμε να εκτελέσουμε το a.out με προ-φόρτωση βιβλιοθηκών θα δώσουμε:

```
LD_PRELOAD="/usr/local/cuda-5.0/lib/libcudart.so.5.0"  ./a.out
```

ενώ αν έχουμε περισσότερες βιβλιοθήκες, θα δώσουμε:

```
LD_PRELOAD="/usr/local/cuda-5.0/lib/libcudart.so.5.0 /lib/anotherlib.so " ./a.out
```

Επίσης, **σε περίπτωση που έχετε πρόσβαση στο MTL** θα πρέπει να υποβάλλετε την εργασία μέσω του κατάλληλου PBS αρχείου ζητώντας τον αντίστοιχο πόρο (**gpus=1**), γιατί αν απλώς το εκτελέσετε δε θα γίνει κάτι, αφού το MTL δεν έχει CUDA GPU κάρτα.

6.2 Το παραδοτέο Α2

(Α2) Κάντε compile στο μηχάνημα CUDA τον παραπάνω κώδικα και δώστε το screenshot εκτέλεσης, αφού χρησιμοποιήσετε το όνομα **a2.cu**

Για να γίνει κατανοητός ο παραπάνω κώδικας πρέπει να εξηγήσουμε μερικά πράγματα. Το declaration `__global__` που βλέπουμε σημαίνει ότι είναι κώδικας ο οποίος προορίζεται για εκτέλεση στην GPU. Υπάρχουν και άλλες τέτοιες δηλώσεις όπως `__device__` και `__shared__`. Ο λόγος που χρειάζεται να το κάνουμε αυτό, είναι γιατί ο κώδικας κατά το compilation μεταφέρεται σε διαφορετικούς μεταγλωττιστές για την CPU και για την GPU.

Η συνάρτηση `cudaMalloc()` είναι αντίστοιχη με την γνωστή malloc μόνο που η cudaMalloc δεσμεύει μνήμη στην GPU. Αντίστοιχα όταν χρησιμοποιούμε την cudaMalloc θα πρέπει να χρησιμοποιήσουμε την cudaFree για να αποδεσμεύσουμε.

```
cudaError_t cudaMalloc ( void ** devPtr, size_t size )
```

Parameters:

devPtr - Pointer to allocated device memory
size - Requested allocation size in bytes

```
cudaError_t cudaFree ( void * devPtr )
```

Parameters:
devPtr - Pointer to allocated device memory

Τέλος η `cudaMemcpy()` χρησιμοποιείται για να αντιγράψουμε στοιχεία που βρίσκονται στην μνήμη της GPU.

```
cudaError_t cudaMemcpy ( void *dst, const void *src, size_t count, enum cudaMemcpyKind kind )
```

Parameters:
dst - Destination memory address
src - Source memory address
count - Size in bytes to copy
kind - Type of transfer

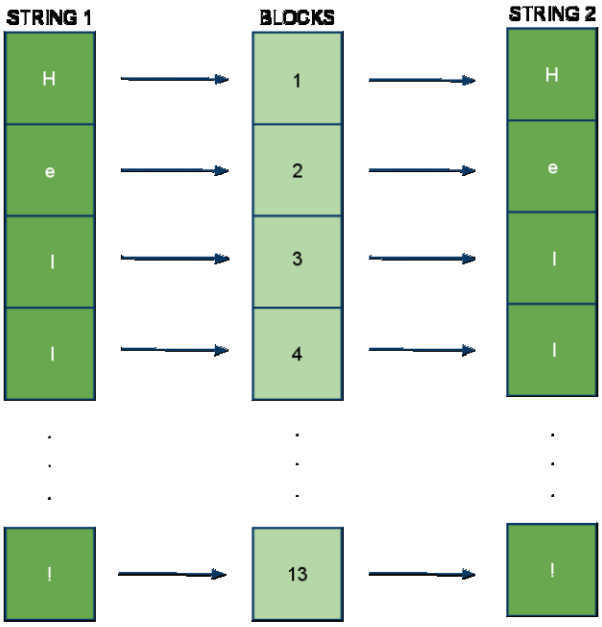
Σε αυτή την περίπτωση χρησιμοποιούμε το `cudaMemcpyDeviceToHost` αφού θέλουμε να αντιγράψουμε δεδομένα από την μνήμη της GPU στην μνήμη του υπολογιστή.

Η λογική είναι η εξής: Έχουμε ένα string το οποίο αποτελείται από ένα σύνολο χαρακτήρων N. Θέλουμε να αξιοποιήσουμε την παραλληλία οπότε θα έχουμε N επεξεργαστικές μονάδες. Κάθε μια από αυτές τις μονάδες θα έχει στην διάθεσή του το string “Hello World!”. Όμως κάθε μονάδα θα πρέπει να κάνει κάτι. Έτσι δεσμεύουμε έναν χώρο μνήμης N και κάθε μονάδα γράφει σε αυτόν τον χώρο έναν χαρακτήρα. Στην συνέχεια αντιγράφουμε το αποτέλεσμα στην μνήμη του υπολογιστή και το εμφανίζουμε.

Τα ερωτήματα που προκύπτουν είναι το πως ορίζουμε σε πόσες επεξεργαστικές μονάδες θα εκτελεστεί ο κώδικας και πως μπορούμε να διαφοροποιήσουμε τι κάνει κάθε μια από αυτές.

Εδώ αποκτά νόημα και `το hello<<<N,1>>>`. Σημαίνει ότι ο κώδικας θα εκτελεστεί σε N blocks που το κάθε ένα θα έχει 1 thread.

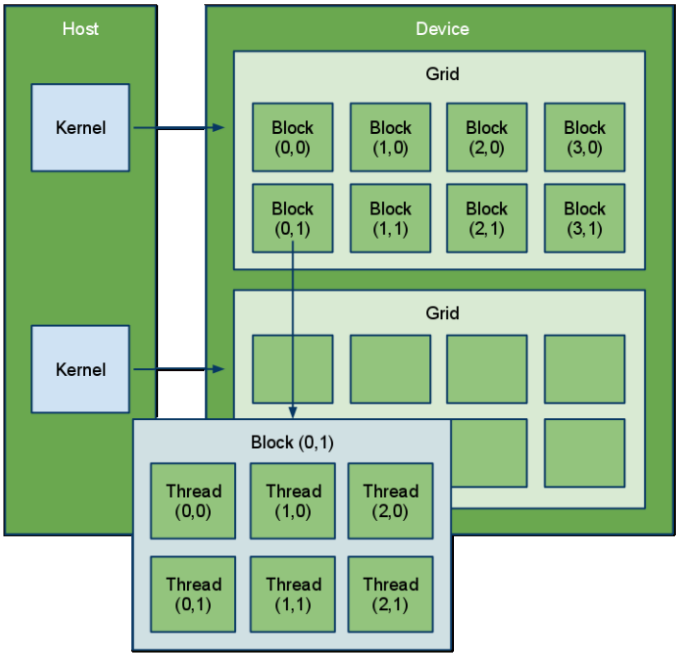
Εφόσον έχουμε ορίσει ότι θα εκτελεστούν N blocks, κατά την εκτέλεση του καθενός μπορούμε να έχουμε πρόσβαση στην global μεταβλητή `blockIdx` που δείχνει το id του. Τα blocks μπορούν να έχουν μια ή δύο διαστάσεις. Εμείς χρησιμοποιούμε μια διάσταση για ευκολία, δηλαδή blockIdx.x. Έτσι σύμφωνα με αυτό το id θα αναθέτουμε στον πίνακα την αντίστοιχη τιμή του string. Π.χ το block 0 θα πάρει το στοιχείο 0 του string (`msg[0]='H'`) και θα το γράψει στην θέση 0 του δεύτερου string (`d_msg[0]='H'`). Αντίστοιχη δουλειά θα γίνει και στα υπόλοιπα blocks.



Εικόνα: σχηματική αναπαράσταση εκτέλεσης

Εκτός από τα blocks υπάρχουν όπως είπαμε τα threads τα οποία μπορούν να ενσωματωθούν μέσα σε grids. Τα grids μπορούν να περιέχουν blocks και τα blocks μπορούν να περιέχουν threads. Τα grids μπορεί να είναι 1D ή 2D όπως και τα blocks, ενώ τα threads μπορεί να είναι 1D, 2D και 3D.

Κάποιος θα μπορούσε να αναρωτηθεί γιατί υπάρχουν δισδιάστατες και τρισδιάστατες δομές επεξεργαστικών μονάδων. Αυτό γίνεται γιατί έτσι απλοποιούνται πολύπλοκα προβλήματα στα οποία είναι απαραίτητη η ύπαρξη τους *(πχ επεξεργασία εικόνας)*. Ουσιαστικά η αντιμετώπισή τους μπορεί να είναι πολύ πιο ρεαλιστική αφού δεν απαιτείται να σκεφτούμε πως θα λύσουμε το πρόβλημα χρησιμοποιώντας μια μόνο διάσταση για ένα δισδιάστατο πρόβλημα. Ένα καλό παράδειγμα είναι οι δισδιάστατοι πίνακες οι οποίοι υπάρχουν σε κάθε γλώσσα προγραμματισμού. Τώρα ας σκεφτούμε ότι χρησιμοποιούμε μια γλώσσα που δεν υποστηρίζει αυτή την δυνατότητα, δηλαδή έχει μόνο μονοδιάστατους πίνακες. Αν θέλαμε να πολλαπλασιάσουμε δύο δισδιάστατους πίνακες θα έπρεπε να χρησιμοποιήσουμε μονοδιάστατους πίνακες, κάτι το οποίο θα έκανε πολύπλοκη την υλοποίηση.



Εικόνα: Σχηματικό παράδειγμα δομής για grids blocks και threads

Κάθε μια από αυτές τις νοητές δομές έχει ένα id με το οποίο μπορούμε να καθορίζουμε σε ποιο grid, block και thread θα κάνουμε μια εργασία. Για να ορίζουμε πιο εύκολα αυτές τις διαστάσεις υπάρχουν οι παρακάτω ενσωματωμένες μεταβλητές.

Variables:

- **dim3** `gridDim;`
  - Dimensions of the grid in blocks
- **dim3** `blockDim;`
  - Dimensions of the block in threads
- **dim3** `blockIdx;`
  - Block index within the grid
- **dim3** `threadIdx;`
  - Thread index within the block

6.3 Τα παραδοτέα C1 και C2

(C1) Άσκηση 1: Χρησιμοποιήστε τις μεταβλητές gridDim και blockDim για να αντικαταστήσετε την δήλωση N,1 στην κλήση `hello<<<N,1>>>`. (όνομα αρχείου **c2.cu**)

(C2) Άσκηση 2: Τροποποιήστε τον κώδικα έτσι ώστε να χρησιμοποιεί παραπάνω από ένα thread/block (όνομα αρχείου **hello03.cu**)

7. Υπολογισμός του π

Θα υπολογίσουμε το π χρησιμοποιώντας την παρακάτω εξίσωση.

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

Για να υπολογίσουμε τον παραπάνω τύπο θα χρησιμοποιήσουμε την αριθμητική ολοκλήρωση από 0 έως 1, η οποία καταλήγει στην παρακάτω σχέση:

$$\pi = \int_0^1 \frac{4}{1+x^2} dx = \sum_{i=0}^{N-1} \frac{4W}{1+[(i+0.5)W]^2}$$

όπου:

- N ο αριθμός των διαστημάτων στον οποίον χωρίζουμε την περιοχή από 0 έως 1 (η ακρίβεια αυξάνεται με τον αριθμό των διαστημάτων)
- W το πλάτος του κάθε διαστήματος, το οποίο ισούται με 1/N

Αν θέλαμε να προγραμματίσουμε σειριακά το παραπάνω κομμάτι θα το γράφαμε ως εξής:

```
int i, N=512;
float pi, W=1.0/N;
pi=0.0;
for (i=0;i<N;i++)
{
      pi+=4*W/(1+(i+0.5)*(i+0.5)*W*W);
}
```

7.1 Το παραδοτέο Α3

(Α3) Να αντιγράψετε τον παραπάνω κώδικα στο αρχείο **a3.c**, να το κάνετε μεταγλώπιση και να το εκτελέσετε. Δώστε το αντίστοιχο screenshot.

Παρατηρήστε ότι ο παραπάνω αλγόριθμος υπολογίζει σε N βήματα το παραπάνω άθροισμα. Θα μπορούσαμε λοιπόν να υπολογίζουμε το άθροισμα με το να χρησιμοποιήσουμε έναν αριθμό από blocks όπου θα χωρίσουμε τα N βήματα σε blocks και κάθε block θα υπολογίζει το άθροισμα στο διάστημα που της αντιστοιχεί.

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>     //optional

#define NUMBLOCKS 3   //number of blocks we will use

__global__ void calc(float *pi, int *N)
{
      int start;  //starting point for each blockIdx.x
      int end;    //ending point
      int i;
      int div;   //dividing the problem into smaller ones
      int idx = blockIdx.x;//getting the id of the current
blockIdx.x running
      float W = 1.0/(*N);
      pi[idx] = 0;
      div = *N/(NUMBLOCKS);

      start = idx*div;      // 0*div 1*div .. NUMBLOCKS*div
      if(idx==NUMBLOCKS-1)
      {
            end =*N ;  //the last block will compute from
idx*div - N
      }
      else
      {
            end = (idx+1)*div;//the rest from idx*div -
idx*(div+1)
      }
      for (i=start;i<end;i++)
      {
            pi[idx]+=4*W/(1+(i+0.5)*(i+0.5)*W*W);
            //each cell will contain a temp sum
      }
}

int main(int argc, char** argv)
{
      int i;
      int N;
      int *dev_N;
      float *pi;
            float *dev_pi;
      float pi_sum = 0;

      //get N from the user
      N = atoi(argv[1]);
```

```
	if(argv[1])
	{

	//allocate an array that will contain our temp sums in
host
	pi = (float *)malloc(NUMBLOCKS*sizeof(float));

	//allocate the same array in device(gpu)
	cudaMalloc((void**)&dev_pi,NUMBLOCKS*sizeof(float));

	//allocate some space for N in device and copy the value
	cudaMalloc((void**)&dev_N,sizeof(int));
	cudaMemcpy(dev_N, &N,
sizeof(int),cudaMemcpyHostToDevice);

	//executing in device
	calc<<<NUMBLOCKS,1>>>(dev_pi,dev_N);

	//copy the array of tmp sums to host
	cudaMemcpy(pi, dev_pi,
		NUMBLOCKS*sizeof(float),cudaMemcpyDeviceToHost);

	//adding the tmp sums into a total one
	for(i=0;i<NUMBLOCKS;i++)
	{
		printf("block:%d sum:%f\n",i,pi[i]);
		pi_sum += pi[i];
	}

	printf("Pi:%.20f\n",pi_sum);

	//eco programming
	cudaFree(dev_pi);
	cudaFree(dev_N);
	free(pi);
	}
	else
	{
		printf("usage: <n> <b> | Number, blocks to use\n");
	}
	return 0;
}
```

Σε περίπτωση που ο compiler σας ενημερώσει ότι δεν υποστηρίζονται οι πραγματικοί αριθμοί διπλής ακρίβειας (**`warning: Double is not supported. Demoting to float`**), θα πρέπει να χρησιμοποιήσετε μια νεότερη αρχιτεκτονική συμβολομετάφρασης και όχι την 1.1 που είναι η προεπιλογή. Δηλαδή, θα πρέπει να χρησιμοποιήσετε στο **nvcc** την παράμετρο
-arch=sm_20 που υποδηλώνει ότι απαιτούμε τη δημιουργία κώδικα για κάρτες γραφικών αρχιτεκτονικής 2.0 και ανώτερης. Η συνέπεια της παραμέτρου είναι ότι ο συγκεκριμένος κώδικας δε θα μπορεί να εκτελεστεί σε κάρτες γραφικών που έχουν αρχιτεκτονικές δυνατότητες μικρότερες από 2.0.

Οι κάρτες γραφικών που έχουμε στο εργαστήριο είναι GTX 760, με αρχιτεκτονικές δυνατότητες έκδοσης 3.0. Στη διεύθυνση, https://developer.nvidia.com/cuda- gpus μπορείτε να δείτε τις αρχιτεκτονικές δυνατότητες *(ή compute capability)* όλων των CUDA καρτών.

Η απλή εκτέλεση του ανωτέρω κώδικα ενδέχεται να μην είναι επιτυχής και να τερματίσει με **`"segmentation fault"`**. Διαβάστε τον κώδικα προσεκτικά ή χρησιμοποιήστε το gdb για να βρείτε σε ποια γραμμή εμφανίζεται το segmentation fault, και θα καταλάβετε πως πρέπει να εκτελεστεί ο συγκεκριμένος κώδικας.

(A4) Κάντε compile στο μηχάνημα CUDA τον παραπάνω κώδικα, αφού δώσετε το όνομα **pcalc.cu**.

Έχουμε ένα define που ορίζουμε σε πόσα block θα εκτελέσουμε τον κώδικα μας. Ο χρήστης δίνει τον αριθμό των διαστημάτων για το οποίο θα υπολογίσουμε το π και το περνάμε σαν παράμετρο στην συνάρτηση που θα εκτελεστεί στην GPU. Κάθε block υπολογίζει διαφορετικό διάστημα που προκύπτει από την διαίρεση του N με τον αριθμό των blocks. Τέλος, για να είμαστε σίγουροι ότι δεν θα εκτελέσουμε τον υπολογισμό για μικρότερο διάστημα δίνουμε στο τελευταίο block σαν άνω όριο το N.

Κάτι που πρέπει να προσέξουμε σε αυτό το παράδειγμα είναι ότι οι δηλώσεις define είναι ορατές και στον κώδικα που εκτελείται στην GPU(NUMBLOCKS). Θα μπορούσαμε να είχαμε ορίσει και το N έτσι αλλά θα έπρεπε κάθε φορά που θέλουμε να το αλλάξουμε να κάνουμε compile τον κώδικα. Έτσι δεσμεύουμε μνήμη στην GPU και ταυτόχρονα αντιγράφουμε την τιμή του host σε εκείνη την θέση μνήμης[4]. Δεν φτάνει όμως μόνο αυτό. Πρέπει να περαστεί και σαν παράμετρος στην συνάρτηση γιατί αλλιώς δεν είναι ορατή η τιμή από την συνάρτηση.

Στο παραπάνω παράδειγμα χρησιμοποιούμε ένα thread/block. Αυτό δεν είναι καλή τακτική. Τα threads που βρίσκονται στο ίδιο block μπορούν να συνεργάζονται και να έχουν πρόσβαση σε κάποια κοινή μνήμη. Επίσης μπορούμε να χρησιμοποιήσουμε την συνάρτηση **`__syncthreads()`** η οποία λειτουργεί σαν barrier. Δηλαδή βάζει ένα φράγμα στην εκτέλεση των threads ώστε να φτάσουν όλα στο ίδιο σημείο επεξεργασίας. Μόνο όταν φτάσουν όλα τα threads σε εκείνο το σημείο συνεχίζεται η εκτέλεση του υπόλοιπου κώδικα. Γενικά ισχύει ότι η υπάρχει μνήμη ιδιωτική για κάθε thread, διαμοιραζόμενη μνήμη μεταξύ των threads που βρίσκονται στο ίδιο block και η κοινή μνήμη.

[4] Κάθε συνάρτηση που προορίζεται για εκτέλεση στην GPU πρέπει να παίρνει ορίσματα τα οποία υπάρχουν στην μνήμη της. Αν προσπαθήσουμε να βάλουμε ορίσματα που βρίσκονται στην κανονική μνήμη θα πάρουμε μήνυμα λάθους.

7.2 Το παραδοτέο C3

(C3) Κατασκευάστε κώδικα για CUDA που να κάνει την διανυσματική πράξη S=AX+Y, όπου X πίνακας ακεραίων 1024 θέσεων, Y πίνακας ακεραίων 1024 θέσεων και A ένας αριθμός. Τοποθετήστε τυχαίες τιμές στους πίνακες X και Y. Εκτυπώστε τους πίνακες S, X, Y στο τέλος. Επίσης, θα εκτυπώνετε το χρόνο εκτέλεσης της πράξη. Ονοματίστε το αρχείο **saxy.cu**

8. Cuda και MPI

Η Κίνα πλέον κατέχει την κορυφαία θέση στον κατάλογο με τους μεγαλύτερους και πιο γρήγορους υπολογιστές του κόσμου, χάρη στο νέο της υπερ-υπολογιστή Tianhe-1, ο οποίος έχει την δυνατότητα να κάνει περισσότερους από 2.500 τρισεκατομμύρια υπολογισμούς ανά δευτερόλεπτο.

Αυτός ο κινεζικός ισχυρισμός επιβεβαιώθηκε από τον διεθνή οργανισμό «Top 500» που συντάσσει τη λίστα με τα 500 ισχυρότερα υπολογιστικά μηχανήματα του κόσμου. Σύμφωνα με το BBC, για να φτάσει αυτή την ασύλληπτη ταχύτητα, ο υπολογιστής εφοδιάζεται με περισσότερους από 7.000 επεξεργαστές γραφικών της Nvidia και 14.000 επεξεργαστές της Intel.

Το Πανεπιστήμιο Δυτικής Μακεδονίας δεν μπορούσε να αντέξει κάποια άλλη θέση εκτός από την πρώτη και έτσι αποφάσισε να φτιάξει τον δικό του υπερυπολογιστή ο οποίος θα είχε 10.000 επεξεργαστές γραφικών της Nvidia και 40.000 επεξεργαστές της Intel. Μάλιστα αυτός ο υπερυπολογιστής συναρμολογήθηκε σε χρόνο ρεκόρ ώστε να προλάβει την αλλαγή στο top 500. Όμως επειδή βρισκόμαστε στην Ελλάδα και επειδή έπρεπε να είμαστε σίγουροι ότι κατά την παραλαβή του υπερυπολογιστή όλα είναι σωστά, έπρεπε να ελέγξουμε τις κάρτες γραφικών. Ήταν όντως αυτές για τις οποίες είχε πληρώσει το Πανεπιστήμιο; Προφανώς αυτό δεν μπορούσαμε να το κάνουμε ανοίγοντας τον κάθε ένα υπολογιστή ξεχωριστά για να βλέπουμε αν όντως περιέχει την σωστή κάρτα γραφικών. Κάτι τέτοιο θα έπαιρνε μέρες. Επίσης υπήρχε ο κίνδυνος να χαλάσουμε κάτι στο ήδη στημένο μας cluster.

Χρειαζόμαστε έναν τρόπο ώστε να εκτελέσουμε κώδικα σε πολλούς υπολογιστές. Αυτό θα το κάνουμε χρησιμοποιώντας το openMPI. Όλα τα υπόλοιπα θα γίνουν χρησιμοποιώντας την CUDA.

Πρώτα χρειαζόμαστε μια συνάρτηση η οποία να παίρνει τις πληροφορίες για τις κάρτες γραφικών για έναν υπολογιστή. Αργότερα θα χρησιμοποιήσουμε αυτή την συνάρτηση μέσα σε ένα πρόγραμμα MPI ώστε να εκτελεστεί σε όσους υπολογιστές επιθυμούμε. Επειδή υπάρχει μια ιδιαιτερότητα για να χρησιμοποιήσουμε αυτές τις δύο βιβλιοθήκες *(χρειάζεται να κάνουμε compile με nvcc και mpicc)* θα πρέπει να κάνουμε link το ένα εκτελέσιμο που θα προκύψει από το compile με nvcc στο compile με mpicc.

```
#include <stdio.h>
extern "C"
void dev_info()
{
      int dev_count;
      int i;
      cudaDeviceProp dev_prop;
      /* Count available devices */
      cudaGetDeviceCount(&dev_count);
      if(dev_count>0)
      {
            printf("\nfound %d device(s)",dev_count);
            /* For every device */
            for(i=0;i<dev_count;i++)
            {
            /* Get device properties */
            cudaGetDeviceProperties(&dev_prop, i);
            printf("\nDevice(%d) characteristics\n",i);
            printf("------------------------\n");
            printf("GPU:  %s\n",dev_prop.name);
            printf("Memory:  %.0f Mbytes\n",
            ceil((float)dev_prop.totalGlobalMem/(1024*1024)));
            printf("Cores:  %d\n",dev_prop.multiProcessorCount);
            printf("Clock:  %.2f
Mhz\n",(float)dev_prop.clockRate/(1000*1000));
            printf("Threads/block:  %d
Max\n",dev_prop.maxThreadsPerBlock);
      printf("------------------------\n");
      }
      }
      else
      {
      printf("\nNo CUDA device found!\n");
      }
}
```

Τα νέα στοιχεία που χρησιμοποιούνται εδώ είναι δύο. Το struct cudaDeviceProp και η συνάρτηση cudaGetDeviceCount. Το struct cudaDeviceProp περιέχει διάφορες πληροφορίες για την κάρτα γραφικών ενώ η συνάρτηση cudaGetDeviceCount επιστρέφει μια τιμή με τις συσκευές που υπάρχουν στο σύστημα. Εμείς πρώτα βλέπουμε πόσες συσκευές υπάρχουν στο σύστημα και έπειτα για κάθε συσκευή τυπώνουμε κάποια στοιχεία.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
/*cuda function*/
void dev_info();
int main(int argc, char *argv[])
{
      int err;
      int rank;
      int size;
      int len;
      char name[MPI_MAX_PROCESSOR_NAME];
      err = MPI_Init(&argc, &argv);
      if(err!=MPI_SUCCESS)
      {
            printf("Initialization error\n");
            MPI_Abort(MPI_COMM_WORLD,err);
      }
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      MPI_Get_processor_name(name,&len);
      printf("rank:%d is running on node:%s",rank,name);
      dev_info();
      MPI_Finalize();
      return 0;
}
```

Θεωρώντας ότι το αρχείο που περιέχει την συνάρτηση dev_info ονομάζεται dev_info.cu και το πρόγραμμα σε MPI, mpi_cuda.c δίνεται το παρακάτω Makefile (διαβάστε τις παρατηρήσεις που ακολουθούν για την υποβολή σε cluster).

```
CUDAROOT=/usr/local/cuda-5.0
hosts=83.212.19.248,83.212.19.247
CUDA=${CUDAROOT}/bin/nvcc
MPI=/usr/bin/mpicc
RUN=LD_LIBRARY_PATH=$(CUDAROOT)/lib64 mpirun
LIBS=-lcudart -L ${CUDAROOT}/lib64 -lm
INCLUDES=-I ${CUDAROOT}/include
default: mpi_cuda.c dev_info.cu
      $(CUDA) -c dev_info.cu $(LIBS)
      $(MPI) -o mpi_cuda mpi_cuda.c dev_info.o $(LIBS)
$(INCLUDES)
execute:
ifdef N
      @$(RUN) -H ${hosts} -n $(N) mpi_cuda
else
      @$(RUN) mpi_cuda
```

```
	@echo ""
	@echo "for multiple computers you need to specify N and
hosts variable in Makefile"
	@echo "e.g make execute N=4"
endif
clean:
	rm dev_info.o
	rm mpi_cuda
```

Σημειώστε ότι στο παραπάνω hostfile στο label execute ορίζουμε εμείς συγκεκριμένα μηχανήματα, αφού το execute φέρει τη γραμμή:

```
@$(RUN) -H ${hosts} -n $(N) mpi_cuda
```

που σημαίνει ότι θα εκτελεστεί το $(RUN) το οποίο όπως βλέπετε στο Makefile είναι το …. mpirun . Στη συνέχεια υπάρχει το –H που θέτει το αρχείο με τους υπολογιστές και το –n που θέτει τον αριθμό των διεργασιών. Όμως, στο cluster MTL δε χρειάζονται αυτοί οι 2 παράμετροι. Οπότε στο αρχείο υποβολής PBS της εργασίας θα πρέπει να δοθεί η εντολή που θέτει το runtime CUDA library (/usr/local/cuda/ ή τροποποιήστε ανάλογα το path) στο LD_LIBRARY_PATH και εκτελεί με τη χρήση του mpirun το mpi_cuda που έχει δημιουργηθεί από το παραπάνω Makefile.

```
LD_LIBRARY_PATH=/usr/local/cuda/lib64 mpirun mpi_cuda
```

8.1 Το παραδοτέο C4

(C4) Κάντε compile τον παραπάνω κώδικα. Δοκιμάστε να ζητήσετε 1,2,3 κόμβους με GPU, και δώστε τα κατάλληλα αρχεία εξόδου (μαζί με το Makefile και τα πηγαία αρχεία). Ο communicator την πρώτη φορά θα έχει 1 κόμβο, μετά 2 και στην τελευταία περίπτωση 3 κόβμους (αν και οι 3 κόμβοι με τις GPU είναι online).