# CUDA – Compute Unified Device Architecture

Καλέρης Κωνσταντίνος
*Πεμπτοετής φοιτητής του τμήματος*
*Ηλεκτρολόγων Μηχανικών & Τεχνολογίας Η/Υ*
*του Πανεπιστημίου Πατρών*
ee5972@upnet.gr

Καλλέργης Γεώργιος
*Πεμπτοετής φοιτητής του τμήματος*
*Ηλεκτρολόγων Μηχανικών & Τεχνολογίας Η/Υ*
*του Πανεπιστημίου Πατρών*
ee5973@upnet.gr

## Abstract

*Σε μια εποχή που η ανάγκη για ταχύτερη επεξεργασία γίνεται ολοένα και πιο έντονη, οι κοινοί επεξεργαστές δεν επαρκούν, για τις συνεχώς αυξανόμενες υποχρεώσεις τους. Η nVidia, πρωτοπόρος στον τομέα των επεξεργαστών γραφικών, εισάγει την CUDA, μια αρχιτεκτονική που επιτρέπει την χρήση των επεξεργαστών γραφικών για συμβατικές εφαρμογές αυξημένου υπολογιστικού φορτίου. Με την νέα αυτή αρχιτεκτονική κάνει την εμφάνισή του ένα νέο προγραμματιστικό παράδειγμα γνωστό με το ακρωνύμιο SIMT (Single Instruction Multiple Threads). Με ένα μινιμαλιστικό σύνολο επεκτάσεων στην γλώσσα προγραμματισμού C η νέα αυτή αρχιτεκτονική γίνεται προσιτή σε κάθε προγραμματιστή σε ελάχιστο χρόνο. Η τεράστια υπολογιστική ισχύς των επεξεργαστών γραφικών σε συνδυασμό με την ικανότητα ελέγχου ροής των κοινών επεξεργαστών ανοίγουν τον δρόμο για μια νέα εποχή στον τομέα των υπολογιστών.*

## 1. Εισαγωγή

Η nVidia, θέλοντας να εκμεταλλευθεί την υπολογιστική ισχύ των επεξεργαστών γραφικών (*GPU*), σε συνδυασμό με την συνεχή ανάπτυξη του παράλληλου λογισμικού, εισήγαγε μια νέα αρχιτεκτονική που επιτρέπει την ανάπτυξη κοινών, μη γραφικών, εφαρμογών, τμήματα των οποίων ανατίθενται προς εκτέλεση στην *GPU*. Το ποια τμήματα της εφαρμογής θα ανατεθούν στην *GPU* έχει σχέση με την δυνατότητα παραλληλοποίησής τους και επιλέγονται από τον προγραμματιστή.

Η CUDA, προκειμένου να παραμείνει όσο το δυνατόν πιο προσιτή στους προγραμματιστές, δημιουργήθηκε ως μια επέκταση της ήδη πολύ δημοφιλούς γλώσσας C χρησιμοποιώντας σε γενικές γραμμές συντακτικό και έννοιες προγραμματισμού ίδιες με αυτήν. Σε αντίθεση με την C, όπου η εκτέλεση είναι συνήθως μονονηματική (*single-threaded execution*), η CUDA μας δίνει την δυνατότητα να ορίσουμε συναρτήσεις (ονόματι *kernels*) οι οποίες εκτελούνται παράλληλα από έναν καθορισμένο αριθμό νημάτων (*multi-threaded execution*).

Η χρήση της CUDA έχει δείξει μεγάλες βελτιώσεις σε πακέτα υπάρχοντος λογισμικού που τροποποιήθηκαν έτσι ώστε τμήματά τους να τρέχουν στην *GPU*, πχ. Matlab, Celestia, Photoshop CS 4. Στην παρούσα εργασία, θα παρουσιάσουμε ένα πρόγραμμα που εκτελεί πρόσθεση και πολλαπλασιασμό πινάκων σε CUDA και σε C και θα συγκρίνουμε τους αντίστοιχους χρόνους εκτέλεσης. Επίσης θα μελετήσουμε την επίδραση του μεγέθους των πινάκων στην ταχύτητα εκτέλεσης. Τέλος θα υπολογίσουμε τους χρόνους επικοινωνίας της κάρτας γραφικών με την CPU, προκειμένου να μπορούμε να εκτιμήσουμε τις περιπτώσεις για τις οποίες η CUDA συμφέρει έναντι της συμβατικής σειριακής εκτέλεσης.

## 2. Προγραμματιστικό Μοντέλο

Τα νήματα οργανώνονται σε μονοδιάστατες, δισδιάστατες ή τρισδιάστατες δομές που ονομάζονται *thread blocks*. Τα *thread blocks* με την σειρά τους οργανώνονται σε μονοδιάστατες ή δισδιάστατες δομές, τα *grid*. Κάθε νήμα, εντός ενός *thread block*, κατέχει έναν αριθμό που το χαρακτηρίζει μονοσήμαντα. Αυτός είναι ο δείκτης του (*index*) και είναι διαθέσιμος σε κάποιον *kernel* μέσω της μεταβλητής **threadIdx** (βλ. Ενότητα 4). Η μεταβλητή αυτή είναι ένα διάνυσμα τριών στοιχείων, μοναδικό για κάθε νήμα, που περιέχει το *index* του στο εκάστοτε *thread block* που ανήκει. Ομοίως κάθε *thread block* έχει έναν μονοδιάστατο ή δισδιάστατο δείκτη

(**blockIdx** βλ. Ενότητα 4) που το χαρακτηρίζει με μοναδικό τρόπο εντός του *grid*.

Όλα τα νήματα εκτελούν τον ίδιο κώδικα. Η επιτάχυνση έγκειται στο ότι το κάθε *thread* εκτελεί τον κώδικα πάνω σε διαφορετικά δεδομένα. Αυτό σημαίνει ότι όταν έχουμε να κάνουμε, πχ. 100 προσθέσεις ανεξάρτητες μεταξύ τους, μπορούμε να αναθέσουμε μια πρόσθεση σε κάθε πυρήνα. Ο συνολικός χρόνος για τις 100 προσθέσεις, για την περίπτωση όπου έχουμε 100 πυρήνες διαθέσιμους, είναι ίσος με τον χρόνο μιας πρόσθεσης. Άρα το συγκεκριμένο κομμάτι κώδικα επιταχύνεται κατά 100 φορές, και ο συνολικός χρόνος εκτέλεσης για ολόκληρο το πρόγραμμα μειώνεται κατά ένα ποσοστό που δίνεται από τον νόμο του Amdahl.

Ο κώδικας του πυρήνα περιέχει παραμέτρους οι οποίες καθορίζουν την θέση των δεδομένων μέσα στην δομή που τα περιέχει, στα οποία θα επιδράσει το κάθε *thread* (παράμετροι δεδομένων). Οι παράμετροι δεδομένων πρέπει να παίρνουν διαφορετικές τιμές για κάθε *thread*, ώστε όλα τα δεδομένα να υποστούν την κατάλληλη επεξεργασία. Για να το πετύχουμε αυτό, γράφουμε τις παραμέτρους δεδομένων ως συνάρτηση της θέσης του *thread* μέσα στο *grid*, η οποία είναι μοναδική για κάθε *thread*. Έτσι, η κάθε παράμετρος γράφεται με την βοήθεια των **threadIdx**, **blockIdx**, **blockDim** και **gridDim**. Οι δυο τελευταίες περιέχουν τις διαστάσεις του κάθε *block* και κάθε *grid* αντίστοιχα.

# 3. Αρχιτεκτονική Επεξεργαστή Γραφικών

## 3.1. Επεξεργαστικοί Πόροι

Όπως το software, έτσι και οι πόροι του hardware είναι δομημένοι ιεραρχικά σε ένα σύνολο από επίπεδα. Για την ακριβή επεξήγηση αυτής της ιεραρχίας είναι απαραίτητο να γνωρίζουμε τον τρόπο με τον οποίο η κάρτα επεξεργάζεται τα γραφικά, κάτι που ξεφεύγει από το αντικείμενο αυτής της εργασίας. Παρ' όλα αυτά, καθώς στην δόμηση του υλικού αρχίζει να λαμβάνεται υπ' όψιν και ο ρόλος της GPU ως *co-processor*, μπορούμε να δώσουμε μια γενική εικόνα για τον τρόπο με τον οποίο εκτελούνται οι *kernels*. Ως αναφορά θα έχουμε την κάρτα geForce GTX280.

Από την σκοπιά της CUDA, στο υψηλότερο σημείο της ιεραρχίας είναι οι 30 *multiprocessors*. Ένας *multiprocessor* αποτελείται από 8 υπολογιστικές μονάδες (*processing cores*) που σημαίνει ότι στο σύνολο έχουμε 240 μονάδες επεξεργασίας. Ο *multiprocessor* είναι υπεύθυνος για να κατανέμει τα *threads* στους *processors*, να τα συγχρονίζει, να φορτώνει και να αποκωδικοποιεί τις εντολές και να τις προωθεί για εκτέλεση.

*Threads* του ιδίου *block* ανατίθενται πάντα στον ίδιο *multiprocessor*. Για να εκτελεστούν, ομαδοποιούνται σε *warps*, τα οποία είναι πακέτα των 32 *threads*. Κάθε warp εκτελείται στον *multiprocessor* με την λογική SIMT (*Single Instruction Multiple Data*). Σύμφωνα με τον ορισμό της nVidia, τα *threads* ενός *warp* εκτελούν ταυτόχρονα όλα την ίδια εντολή (αυτό είναι πρακτικά αδύνατο με 8 *processors*, αλλά πετυχαίνεται με switching χωρίς σημαντικό *overhead*). Μόλις ολοκληρωθεί η εκτέλεση της εντολής, ο *multiprocessor* προωθεί την επόμενη κοινή εντολή (γνωστό μοντέλο *SIMD*). Σε περίπτωση που ο κώδικας του *kernel* περιέχει διακλαδώσεις που δεν ακολουθούνται με τον ίδιο τρόπο από όλα τα *threads*, τότε το κάθε *branch* εκτελείται σειριακά, μέχρι και πάλι όλα τα *threads* να συγκλίνουν στην κοινή τους ροή. Αυτή είναι η λογική της εκτέλεσης SIMT που εισάγει η nVidia στην αρχιτεκτονική CUDA.

Παρόλα αυτά υπάρχουν κάποιοι περιορισμοί. Δεν μπορούμε να έχουμε περισσότερα από συνολικά 32 *warps* ενεργά σε κάθε πολυεπεξεργαστή. Δεν μπορούμε να έχουμε περισσότερα από 8 *blocks* ενεργά σε κάθε *multiprocessor*, ακόμα και αν ο αριθμός των *threads* δεν είναι περιοριστικός. Για την geForce GTX280 αυτό σημαίνει ότι το σύνολο των ενεργών *threads* ανά *multiprocessor* είναι 1024.

## 3.2. Οργάνωση μνήμης

Τα νήματα της CUDA έχουν πρόσβαση σε δεδομένα από μια πληθώρα θέσεων μνήμης. Κάθε νήμα έχει την δική του τοπική μνήμη στην οποία μόνο το ίδιο έχει πρόσβαση. Αυτή είναι οι καταχωρητές του πολυεπεξεργαστή στον οποίο το νήμα έχει ανατεθεί προς εκτέλεση. Σε ένα υψηλότερο επίπεδο όλα τα νήματα ενός *thread block* έχουν πρόσβαση σε μία κοινή μνήμη (*shared memory*) μέσω της οποίας μπορούν και να επικοινωνήσουν. Υψηλότερα από όλα τα άλλα είδη μνήμης υπάρχει η *global* μνήμη στην οποία έχουν πρόσβαση όλα τα νήματα ανεξάρτητα από το *block* στο οποίο ανήκουν. Ενημερωτικά αναφέρουμε την ύπαρξη των *constant* και *texture* μνημών οι οποίες προσφέρονται μόνο για ανάγνωση από τα νήματα. Η εγγραφή τους είναι δυνατή μόνο από την πλευρά της CPU (*host*).

# 4. Προγραμματισμός Εφαρμογής

## 4.1. Διεπαφή προγραμματισμού

Για να είναι δυνατός ο προγραμματισμός της εφαρμογής η nVidia παρέχει μία διεπαφή προγραμματισμού (*API*) η οποία περιέχει συναρτήσεις, που βοηθούν στην εκτέλεση χρήσιμων λειτουργιών στην κάρτα γραφικών και μεταβλητές που είναι απαραίτητες για τον καθορισμό των δεδομένων στα οποία επιδρά κάθε νήμα.

Οι συναρτήσεις που θα χρησιμοποιηθούν στην παρούσα εφαρμογή αφορούν στην δέσμευση,

αποδέσμευση μνήμης στην κάρτα γραφικών, στην δημιουργία, χρησιμοποίηση και καταστροφή *events* για μέτρηση χρονικών διαστημάτων εκτέλεσης καθώς και στην μεταφορά δεδομένων από και προς την κάρτα γραφικών.

Οι συνάρτηση δέσμευσης χώρου είναι παρόμοια με αυτή της C. Η συνάρτηση που χρησιμοποιούμε είναι η **cudaMalloc** η οποία δέχεται σαν ορίσματα της έναν δείκτη στον οποίο επιστρέφει την διεύθυνση της δεσμευθήσας περιοχής μνήμης στην κάρτα και έναν ακέραιο αριθμό που δηλώνει το μέγεθος της μνήμης προς δέσμευση. Με την παρακάτω δήλωση δεσμεύουμε στην κάρτα γραφικών έναν πίνακα ακεραίων δέκα στοιχείων.

```
int *matrix;
cudaMalloc((void**)&matrix, 10 * sizeof(int));
```

Για την αποδέσμευση του παραπάνω χώρου χρησιμοποιούμε την συνάρτηση **cudaFree** δίνοντας της σαν μοναδικό όρισμα τον δείκτη προς την περιοχή που θέλουμε να αποδεσμεύσουμε.

```
cudaFree(matrix);
```

Οι συναρτήσεις δημιουργίας *event* για την χρονομέτρηση χρησιμοποιούνται όπως φαίνεται στο παρακάτω δείγμα κώδικα

```
//Ορισμός μεταβλητών event
cudaEvent_t start, stop;
//Δημιουργία event
cudaEventCreate(&start);
cudaEventCreate(&stop);
//Εναρξη μέτρησης
cudaEventRecord(start, 0);
<<<Κώδικας προς χρονομέτρηση>>>
//Λήξη μέτρησης
cudaEventRecord(stop, 0); cudaEventSynchronize(stop);
float time;
//Λήψη χρόνου
cudaEventElapsedTime(&time, start, stop);
//Καταστροφή των event
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Με τον τρόπο αυτό ουσιαστικά χρησιμοποιούμε το ρολόι του επεξεργαστή γραφικών σαν χρονόμετρο ακρίβειας 0,5μsec! Η συνάρτηση **cudaEventElapsedTime** μας επιστρέφει σε μία μεταβλητή float (πρώτο όρισμα), σε κλίμακα millisecond, τον χρόνο που πέρασε από την έγερση του πρώτου *event* (δεύτερο όρισμα) μέχρι την έγερση του δεύτερου *event* (τρίτο όρισμα). Σημειώνουμε ότι το πρώτο όρισμα της **cudaEventElapsedTime** είναι δείκτης στην θέση μνήμης που θα αποθηκευθεί το αποτέλεσμα της χρονομέτρησης γι' αυτό και είναι απαραίτητο εδώ το σύμβολο **&**.

Τέλος για την μεταφορά των δεδομένων γίνεται χρήση της συνάρτησης **cudaMemcpy**. Η συνάρτηση αυτή λαμβάνει σαν δεδομένα εισόδου έναν δείκτη στην μνήμη προέλευσης, έναν δείκτη στην μνήμη προορισμού, έναν ακέραιο που δίνει το μέγεθος των δεδομένων προς μεταφορά και τέλος έναν ακέραιο αριθμό που καθορίζει την φορά της μεταφοράς. Ο τελευταίος αυτός αριθμός για διευκόλυνση του προγραμματιστή έχει δηλωθεί σαν enumeration στην CUDA και εμφανίζεται ως cudaMemcpyHostToDevice, cudaMemcpyDeviceToDevice ή cudaMemcpyDeviceToHost. Πρέπει να σημειωθεί ότι ο δείκτης που καθορίζει την θέση μνήμης στην κάρτα γραφικών είναι απαραίτητο να έχει προκύψει από ανάθεση μέσω της **cudaMalloc** ή από ανάθεση μέσω αλλού δείκτη ο οποίος έχει δημιουργηθεί με χρήση της **cudaMalloc**. Έτσι με τον παρακάτω κώδικα δεσμεύουμε χώρο στην κάρτα γραφικών, μεταφέρουμε σε αυτήν ακέραια δεδομένα από την RAM του υπολογιστή, στην συνέχεια τα μεταφέρουμε σε άλλη θέση στην κάρτα και τέλος τα μεταφέρουμε πίσω στην μνήμη RAM του *υπολογιστή*.

```
int *data_device, *data_device_2;
int *data_host;
//Δέσμευση μνήμης στην κάρτα
cudaMalloc((void**)&data_device, size);
//Δέσμευση επιπλέον μνήμης στην κάρτα
cudaMalloc((void**)&data_device_2, size);
//Μεταφορά από την RAM στην κάρτα
cudaMemcpy(data_device,data_host,cudaMemcpyHostToDevice);
//Μεταφορά από την κάρτα στην κάρτα
cudaMemcpy(data_device_2,data_device,cudaMemcpyDeviceToDevice);
//Μεταφορά από την κάρτα στην RAM
cudaMemcpy(data_host,data_device_2,cudaMemcpyDeviceToHost);
```

Για απλοποίηση θεωρούμε ότι η μεταβλητή size περιέχει το μέγεθος των δεδομένων προς μεταφορά και ότι η μεταβλητή data_host είναι δείκτης στα δεδομένα τα οποία μεταφέρουμε. Στην παρούσα εφαρμογή αρχικά δεσμεύουμε τον απαραίτητο χώρο στην κάρτα, μεταφέρουμε εκεί τα δεδομένα προς επεξεργασία, τα επεξεργαζόμαστε με έναν *kernel* και στην συνέχεια μεταφέρουμε το αποτέλεσμα πίσω στην κεντρική μνήμη του υπολογιστή.

### 4.2. Πυρήνες Προγράμματος

Για τον πυρήνα του πολλαπλασιασμού, οι πίνακες αντιμετωπίστηκαν ως δισδιάστατοι προκειμένου να εφαρμοστούν σε αυτούς απ' ευθείας οι ιδιότητες του πολλαπλασιασμού πινάκων. Με βάση την παραπάνω προσέγγιση, υπάρχουν δύο παράμετροι επιλογής δεδομένων από τις οποίες σχηματίζεται το *Thread ID* του κάθε *thread*, η παράμετρος **i** και η παράμετρος **j**, που αντιστοιχούν στην γραμμή και την στήλη του πίνακα που περιέχει το αποτέλεσμα. Τα *blocks* και το *grid* είναι δισδιάστατα.

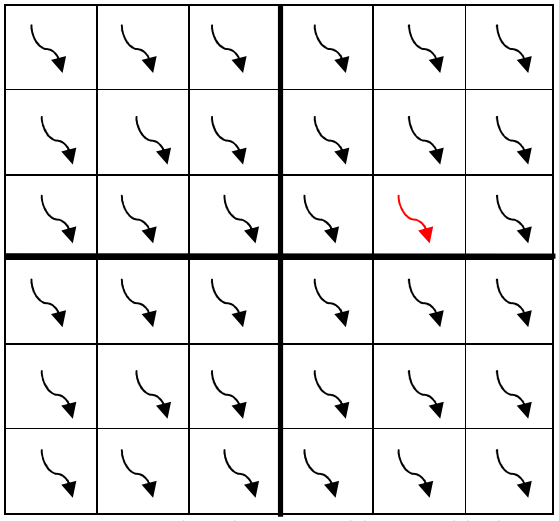
Με τον παραπάνω σχεδιασμό του *grid* και των *blocks*, το *grid* έχει ακριβώς την δομή του πίνακα του αποτελέσματος (αν εξαιρέσουμε τα πιθανώς επιπρόσθετα *threads* που δεν παίζουν ρόλο στο αποτέλεσμα).

Ο κώδικας του πυρήνα φαίνεται παρακάτω

```
__global__ void CudaMul(float *A, float *B, float *C, int
noc0, int noc1, int nor0)
{

 int k;
 int i = blockDim.x*blockIdx.x + threadIdx.x;
 int j = blockDim.y*blockIdx.y + threadIdx.y;

  if(i < noc1 && j < nor0)
  {
      C[j*noc1 + i] = 0;
      for(k = 0;  k < noc0;  k++)
      {
          C[j*noc1 + i] +=  A[j*noc0 + k] * B[k*noc1 + i];
      }
  }
}
```

Ο παραπάνω κώδικας εκτελεί τον πολλαπλασιασμό $C = A \cdot B$. Σημειώνεται ότι τα ορίσματα $noc0$, $noc1$, $nor0$ είναι ο αριθμός των στηλών του πρώτου και δεύτερου πίνακα και ο αριθμός των γραμμών του πρώτου πίνακα.

Στο κάθε *thread* ανατίθεται ο υπολογισμός ενός στοιχείου του πίνακα του αποτελέσματος. Αυτό σημαίνει ότι το κάθε *thread* εκτελεί ένα σύνολο από $noc0$ (αριθμός στηλών του πρώτου πίνακα) πολλαπλασιασμούς και $noc0-1$ προσθέσεις, οι οποίες γίνονται σειριακά για κάθε *thread* στο βρόχο *for*. Η παράμετρος **k** για τον πίνακα $A$ επιλέγει στοιχεία της ίδιας γραμμής, διαδοχικά για κάθε μια στήλη, ενώ για τον πίνακα $B$ επιλέγει στοιχεία της ίδιας στήλης, διαδοχικά για κάθε γραμμή. Ανάλογα με την θέση του στο *grid*, το κάθε *thread* υπολογίζει ένα στοιχείο του $C[j][i]$. Για παράδειγμα, στο *grid* που φαίνεται στο Σχήμα 1, το οποίο είναι διαστάσεων 2x2 και αποτελείται από τέσσερα *blocks* 3x3, το σημειωμένο *thread* έχει $i = blockDim.x \cdot blockIdx.x + threadIdx.x = 3 \cdot 1 + 1 = 4$ και $j = blockDim.y \cdot blockIdx.y + threadIdx.y = 3 \cdot 0 + 2 = 2$, άρα υπολογίζει το στοιχείο $C[2][4]$ του πίνακα αποτελέσματος. Να σημειωθεί ότι η αρίθμηση των δεικτών ξεκινά από το μηδέν.



**Σχήμα 1**. Θέση thread(2,1) σε grid 2x2 με block 3x3

Για να δημιουργήσουμε το *grid* και να τρέξουμε τον *kernel* εκτελούμε τις παρακάτω εντολές στο κυρίως πρόγραμμα:

```
dim3 dimBlock(16,16);
dim3 dimGrid(ceil(nor[0]/float(16)),
ceil(noc[1]/float(16)));
CudaMul<<<dimGrid, dimBlock>>>(matrix_d[0],
matrix_d[1], result_d, noc[0], noc[1], nor[0]);
```

Με αυτές τις εντολές δημιουργούμε ένα *grid* του οποίου οι διαστάσεις είναι **ceil(nor[0]/float(16))*ceil(noc[1]/float(16))** και το κάθε block έχει διαστάσεις **16*16.** Στην περίπτωση που η διαίρεση **noc[1]/float(16)** ή **nor[0]/float(16)** δεν γίνεται ακριβώς, δημιουργούνται περισσότερα *threads* απ' όσα είναι απαραίτητα. Γι' αυτό τον λόγο χρησιμοποιούμε το **if**, προκειμένου τα επιπλέον *threads* να μην εκτελέσουν τον κώδικα και να τερματίσουν κατευθείαν.

Ο πυρήνας της πρόσθεσης είναι πολύ πιο απλός λόγω της απλότητας της ίδιας της πράξης, αλλά και διότι χρησιμοποιήθηκε μονοδιάστατο *grid* και *block*. Οι πίνακες στην πρόσθεση αντιμετωπίστηκαν ως μονοδιάστατοι και έτσι η μόνη εργασία του κάθε νήματος είναι να προσθέσει δυο στοιχεία των πινάκων που προστίθενται και να αποθηκεύσει το αποτέλεσμα στον τελικό πίνακα.

Ο κώδικας του πυρήνα της πρόσθεσης είναι ο ακόλουθος:

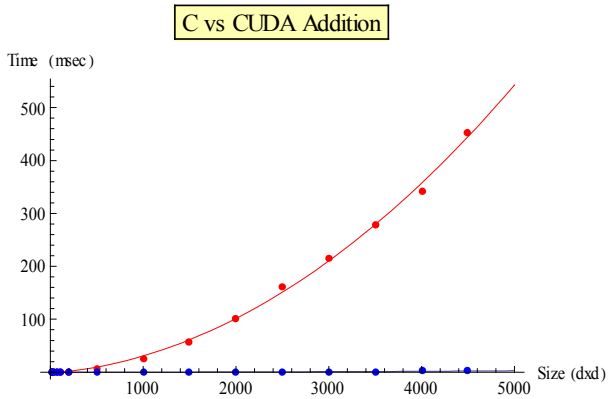
```
dim3 dimBlock(THREADS_PER_BLOCK);
dim3 dimGrid(ceil((noc[0]*nor[0])/(float)
THREADS_PER_BLOCK));
CudaAdd<<<dimGrid, dimBlock>>>(matrix_d[0],
matrix_d[1], result_d, nor[0], noc[0]);

__global__ void CudaAdd(float* A, float* B, float* C, int
h, int w)
{
    int i = (blockIdx.x * blockDim.x) + threadIdx.x;

    if(i < (h*w))
    {
       C[i] = A[i] + B[i];
    }
}
```

Όπου $h \cdot w = noc[0] \cdot noc[1]$ ο συνολικός αριθμός στοιχείων των πινάκων που προστίθενται.

## 5. Μετρήσεις και Αποτελέσματα

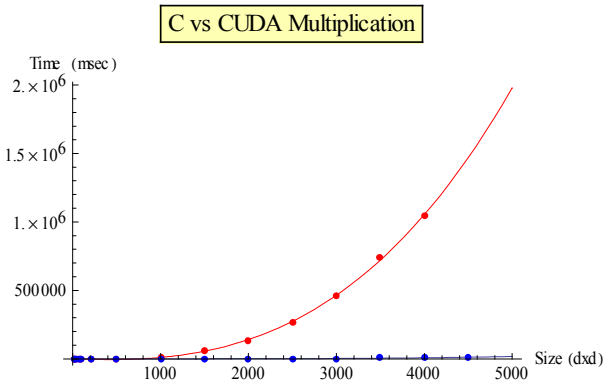
Η παραπάνω εφαρμογή αρχικά εκτελέστηκε για διαφορετικά μεγέθη τετραγωνικών πινάκων εισόδου και μετρήθηκε κάθε φορά ο απαιτούμενος χρόνος ολοκλήρωσης της εκτέλεσης. Μετρήσεις έγιναν για την πρόσθεση σε C, την πρόσθεση σε CUDA, τον πολλαπλασιασμό σε C, τον πολλαπλασιασμό σε CUDA, τον συνολικό χρόνο εκτέλεσης πρόσθεσης και πολλαπλασιασμού σε CUDA συμπεριλαμβανομένης και της μεταφοράς των δεδομένων από και προς την κάρτα καθώς και μετρήσεις χρόνου μεταφοράς δεδομένων ανάμεσα στην κεντρική μνήμη και την κάρτα.

Στο Σχήμα 2 βλέπουμε τον χρόνο εκτέλεσης της πρόσθεσης σε C (κόκκινο χρώμα) και CUDA (μπλε χρώμα) για τα διάφορα μεγέθη πινάκων.



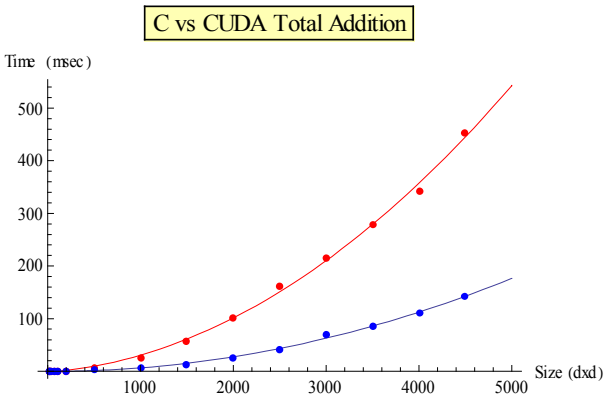

**Σχήμα 2**. Πρόσθεση σε C & CUDA

Στο Σχήμα 3 βλέπουμε τον χρόνο εκτέλεσης του πολλαπλασιασμού σε C (κόκκινο χρώμα) και CUDA (μπλε χρώμα) για τα διάφορα μεγέθη πινάκων.



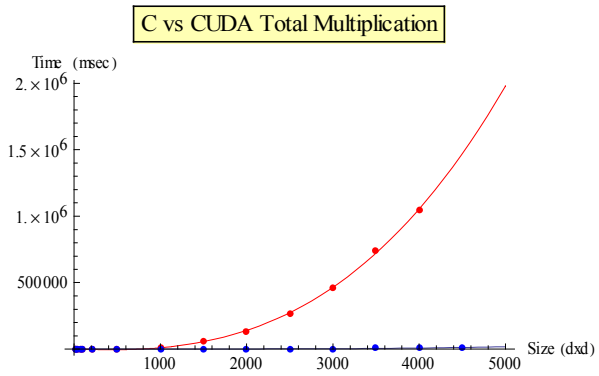

**Σχήμα 3**. Πολλαπλασιασμός σε C & CUDA

Στο Σχήμα 4 βλέπουμε τον χρόνο εκτέλεσης της πρόσθεσης σε C (κόκκινο χρώμα) και CUDA (μπλε χρώμα) για τα διάφορα μεγέθη πινάκων συμπεριλαμβάνοντας στην CUDA τον χρόνο μεταφοράς των δεδομένων από και προς την κάρτα.



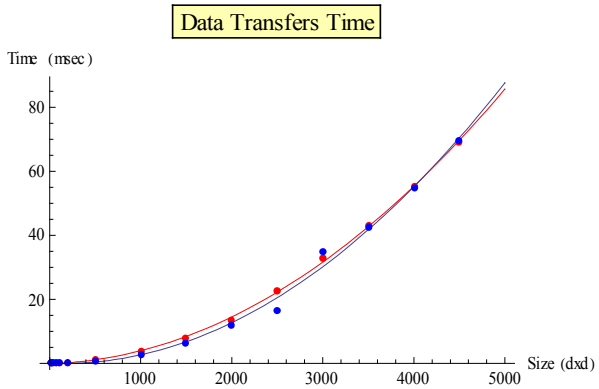

**Σχήμα 4**. Πρόσθεση σε C & CUDA (με καθυστέρηση μεταφοράς δεδομένων)

Στο Σχήμα 5 βλέπουμε τον χρόνο εκτέλεσης του πολλαπλασιασμού σε C (κόκκινο χρώμα) και CUDA (μπλε χρώμα) για τα διάφορα μεγέθη πινάκων συμπεριλαμβάνοντας στην CUDA τον χρόνο μεταφοράς των δεδομένων από και προς την κάρτα.



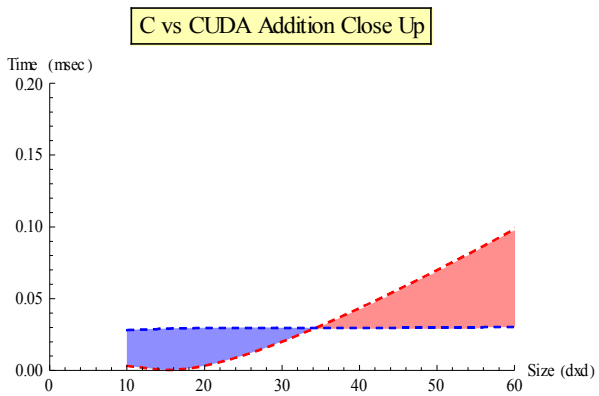

**Σχήμα 5**. Πολλαπλασιασμός σε C & CUDA (με καθυστέρηση μεταφοράς δεδομένων)

Στο Σχήμα 6 βλέπουμε τους χρόνους μεταφοράς των δεδομένων από την κεντρική μνήμη του υπολογιστή στην κάρτα (κόκκινο χρώμα) και το αντίστροφο (μπλε χρώμα). Να σημειωθεί ότι κατά την μεταφορά δεδομένων πίσω στην κεντρική μνήμη το μέγεθος είναι το μισό.

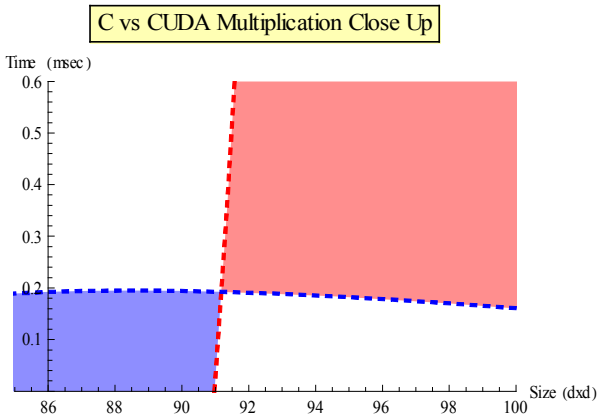



**Σχήμα 6**. Χρόνοι μεταφοράς δεδομένων πρόσθεσης

Τέλος στο Σχήμα 7 βλέπουμε τους χρόνους εκτέλεσης της πρόσθεσης σε C (κόκκινο χρώμα) και CUDA (μπλε χρώμα) για πίνακες εισόδου μικρού μεγέθους και αντίστοιχα στο Σχήμα 8 για τον πολλαπλασιασμό.

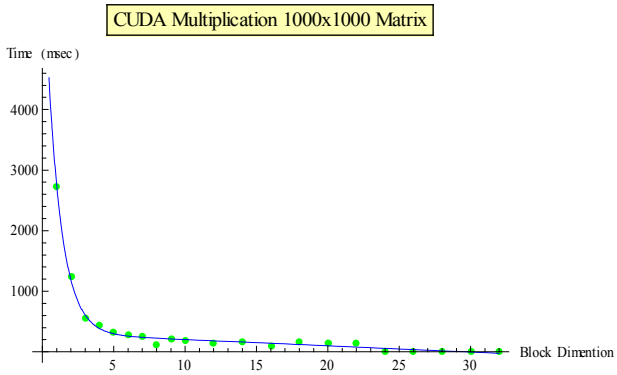



**Σχήμα 7**. Πρόσθεση σε C & CUDA



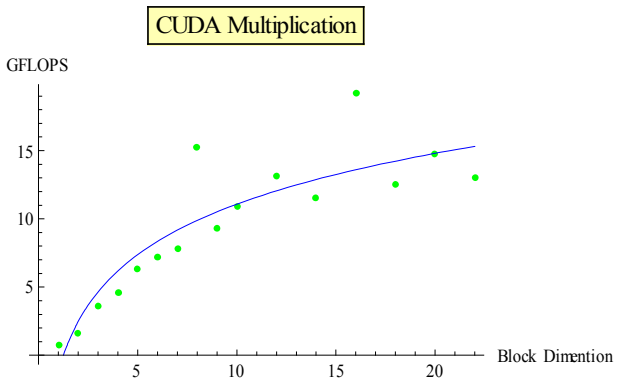

**Σχήμα 8**. Πολλαπλασιασμός σε C & CUDA

Στην συνέχεια εκτελούμε την εφαρμογή μεταβάλλοντας το μέγεθος των *block* χρησιμοποιώντας σταθερό πίνακα εισόδου διαστάσεων 1000x1000. Το Σχήμα 9 δείχνει την μεταβολή του χρόνου εκτέλεσης καθώς αυξάνει ο αριθμός των νημάτων ανά *block*.




**Σχήμα 9**. Πολλαπλασιασμός σε CUDA μεταβάλλοντας τον αριθμό νημάτων ανά *block*

Στο Σχήμα 10 φαίνεται ο αριθμός των δισεκατομμυρίων πράξεων που εκτελούνται στην μονάδα του χρόνου καθώς μεταβάλλεται η διάσταση των *block*.



**Σχήμα 10**. Πολλαπλασιασμός σε CUDA μεταβάλλοντας τον αριθμό νημάτων ανά *block*

Σημειώνουμε ότι τα *block* είναι τετράγωνα και μεταβάλουμε όμοια και τις δυο διαστάσεις τους κατά την πειραματική διαδικασία.

Τέλος υπολογίζουμε τον μέσο αριθμό πράξεων κινητής υποδιαστολής ανά δευτερόλεπτο για κάθε περίπτωση. Στην πρόσθεση ο συνολικός αριθμός πράξεων είναι $n \times n$ και στον πολλαπλασιασμό $2 \cdot n^3 - n^2$, όπου $n$ η διάσταση του τετραγωνικού πίνακα. Στον πίνακα 1 φαίνονται τα αποτελέσματα σε GFLOPS (δισεκατομμύρια πράξεις το δευτερόλεπτο).

**Πίνακας 1**. Δισεκατομμύρια πράξεις κινητής υποδιαστολής ανά δευτερόλεπτο

| Dim | C+ | CUDA+ | C* | CUDA* |
|---|---|---|---|---|
| **10** | 0,03 | 0,01 | 0,13 | 0,05 |
| **20** | 0,12 | 0,01 | 0,15 | 0,38 |
| **40** | 0,04 | 0,05 | 0,51 | 2,37 |
| **70** | 0,04 | 0,16 | 0,19 | 7,73 |
| **100** | 0,04 | 0,31 | 0,17 | 12,38 |
| **200** | 0,04 | 1,14 | 0,17 | 17,97 |
| **500** | 0,04 | 4,19 | 0,18 | 15,69 |
| **1000** | 0,04 | 7,35 | 0,13 | 19,16 |
| **1500** | 0,04 | 8,80 | 0,12 | 17,32 |
| **2000** | 0,04 | 9,52 | 0,12 | 20,37 |
| **2500** | 0,04 | 9,85 | 0,12 | 16,18 |
| **3000** | 0,04 | 10,01 | 0,12 | 17,58 |
| **3500** | 0,04 | 9,42 | 0,12 | 14,67 |
| **4000** | 0,05 | 9,29 | 0,12 | 18,76 |
| **4500** | 0,04 | 9,47 | - | 14,43 |

## 6. Συμπεράσματα

Όπως φαίνεται από τις παραπάνω γραφικές παραστάσεις η CUDA είναι αρκετές φορές ποιο γρήγορη από την C κυρίως όταν η επεξεργασία αφορά σε μεγάλο όγκο δεδομένων.

Στην πρόσθεση, η μετρήσεις προσεγγίζονται στην γραφική παράσταση του Σχήματος 2 από πολυωνυμική συνάρτηση δευτέρου βαθμού. Για την C ο συντελεστής μεγιστοβάθμιου όρου είναι $1.91216 \cdot 10^{-5}$ ενώ για την CUDA είναι $1.11609 \cdot 10^{-7}$, δηλαδή δυο τάξεις μεγέθους μικρότερος. Για να κάνουμε μια αντικειμενική σύγκριση πρέπει, για την CUDA, να λάβουμε υπ όψιν και τους χρόνους μεταφοράς των δεδομένων εισόδου από την κεντρική μνήμη στην κάρτα γραφικών και των δεδομένων εξόδου από την κάρτα γραφικών στην κύρια μνήμη (Σχήμα 4). Από το Σχήμα 7 προκύπτει ότι η CUDA είναι σταθερά πιο γρήγορη από την C για πίνακες εισόδου μεγαλύτερους από $35x35$ ($5KB$ για float δεδομένα).

Στον πολλαπλασιασμό, οι μετρήσεις προσεγγίζονται στην γραφική παράσταση του Σχήματος 3 από πολυωνυμική συνάρτηση τρίτου βαθμού. Για την C ο συντελεστής μεγιστοβάθμιου όρου είναι $1.18243 \cdot 10^{-5}$ ενώ για την CUDA είναι $2.08497 \cdot 10^{-7}$, δηλαδή δυο τάξεις μεγέθους μικρότερος. Λαμβάνοντας υπ' όψιν τους χρόνους μεταφοράς (Σχήμα 8) προκύπτει ότι η CUDA είναι σταθερά πιο γρήγορη από την C για πίνακες εισόδου μεγαλύτερους από $91x91$ ($33KB$ για float δεδομένα).

Όπως έχει αναφερθεί και στην επεξήγηση του προγραμματιστικού μοντέλου, ο χρόνος διεκπεραίωσης για την CUDA εξαρτάται από τον αριθμό των *thread* ανά *block*. Για να μελετήσουμε την συμπεριφορά της CUDA ως προς αυτή την παράμετρο, εκτελέσαμε μια σειρά πολλαπλασιασμών μεταξύ πινάκων $1000 \times 1000$, αλλάζοντας κάθε φορά το **blockDim**. Για ευκολία χρησιμοποιήθηκαν μόνο τετραγωνικά *block*. Τα αποτελέσματα των μετρήσεων αυτών συνοψίζονται στις γραφικές παραστάσεις 9 και 10, που απεικονίζουν την μεταβολή του χρόνου και των GFLOPS αντίστοιχα.

Προσεγγίζοντας τις μετρήσεις με την βέλτιστη καμπύλη, βρέθηκε ότι ο χρόνος μειώνεται εκθετικά και τα GFLOPS αυξάνονται λογαριθμικά έναντι του μεγέθους της μιας διάστασης του **blockDim**.

## 7. Βιβλιογραφία - Πηγές


[1] nVidia, nVidia CUDA Compute Unified Device Architecture Programming Guide Version 2.0, nVidia, 6/7/2008.

[2] nVidia, nVidia CUDA Compute Unified Device Architecture Reference Manual Version 2.0, nVidia, June 2008.