Τμήμα Μηχανικών Ηλεκτρονικών Υπολογιστών και Πληροφορικής, Πανεπιστήμιο Πατρών

Project

# *Λογισμικό και προγραμματισμός συστημάτων υψηλής επίδοσης*

*Σπύρος Καυκιάς 5317*
*Ταλάκ Μπαιράμ 5362*

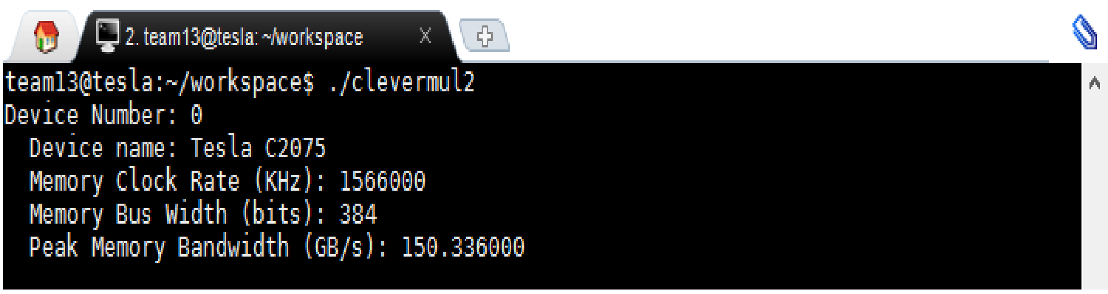
## *Εισαγωγή*

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 παραλληλοποίησης τους και επιλέγονται από τον προγραμματιστή.

Η CUDA[1] είναι μια πλατφόρμα παράλληλου προγραμματισμού και ταυτόχρονα ένα προγραμματιστικό μοντέλο διεπαφής *(API)* που έχει δημιουργήσει η nVidia. Επιτρέπει στους μηχανικούς λογισμικού να χρησιμοποιήσουν τις δυνατότητες των *GPU* για να επιλύσουν προβλήματα γενικής χρήσεως *(GPGPU[2])*. Προκειμένου η CUDA να παραμείνει όσο το δυνατόν πιο προσιτή στους προγραμματιστές, δημιουργήθηκε ως μια επέκταση της 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, η CUDA μας δίνει την δυνατότητα να ορίσουμε συναρτήσεις *(kernels)* οι οποίες εκτελούνται παράλληλα από εναν καθορισμένο αριθμό νημάτων *(multi-threaded execution)*.

Στην παρούσα εργασία θα ασχοληθούμε με ένα πρόβλημα γραμμικής άλγεβρας, με τον υπολογισμό του γινομένου του ανάστροφου ενός μητρώου με το αρχικό μητρώο. Δηλαδή, το γινόμενο $A^T \cdot A$. Φυσικά ο υπολογισμός αυτός συνήθως αποτελεί τμήμα μόνο του συνολικού προγράμματος, ωστόσο, μπορεί να απαιτεί αρκετά σημαντικό ποσοστό από τον συνολικό χρόνο εκτέλεσης της εφαρμογής. Κατά συνέπεια μια αποδοτική υλοποίηση είναι απαραίτητη. Θα υλοποιήσουμε και θα μετρήσουμε την απόδοση της προαναφερθείσας πράξης στο προγραμματιστικό μοντέλο *CUDA*.

Παρακάτω παρουσιάζουμε κάποια βασικά στοιχεία της GPU που χρησιμοποιήθηκε για να υλοποιήσουμε την εργασία.

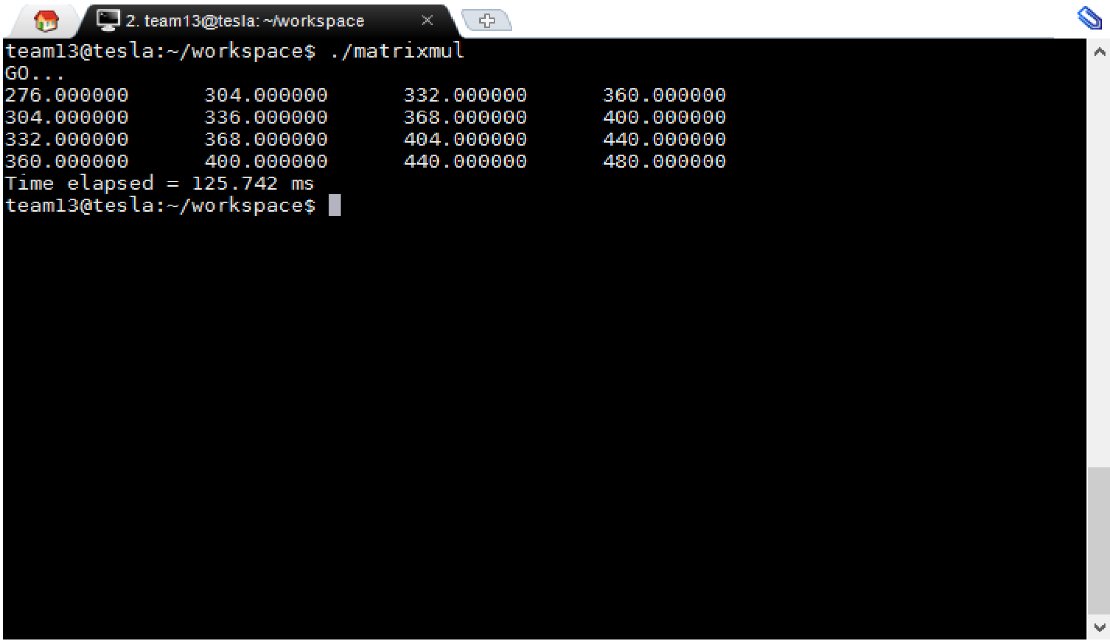



*Εικόνα 1: Χαρακτηριστικά GPU*

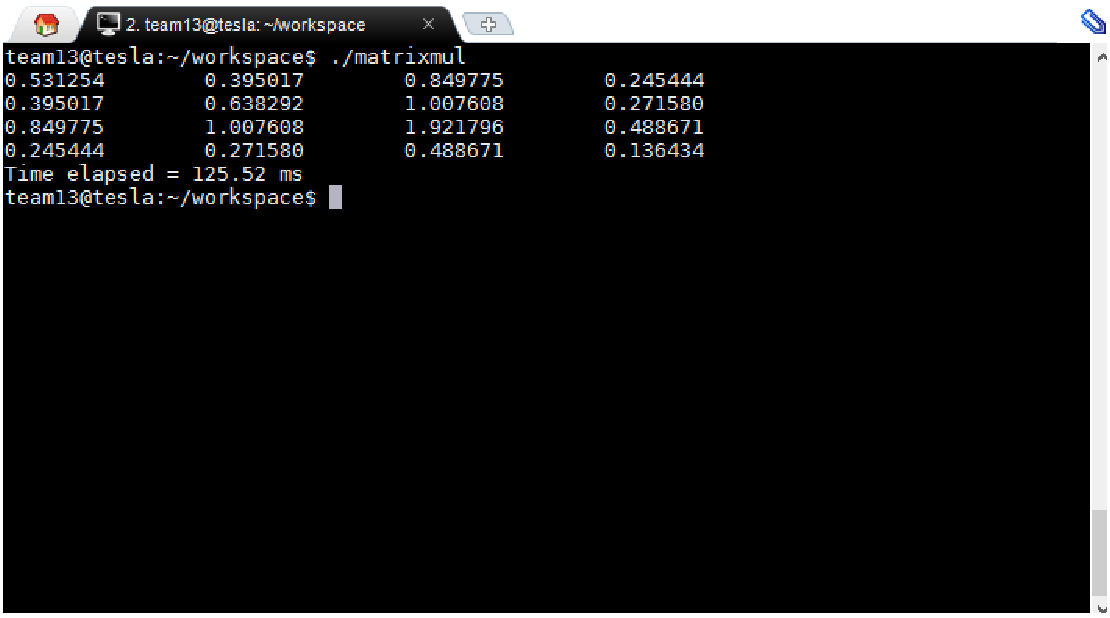
[1] Compute Unified Device Architecture
[2] General-Purpose computation on graphics processing units

# Ερώτημα 1

Σε αυτό το ερώτημα χρησιμοποιήσαμε την συνάρτηση της *CUDA cublasDgemm* για διάφορα μεγέθη μητρώων και μετρήσαμε τον χρόνο εκτέλεσης της πράξης. Παρακάτω χρησιμοποιήσαμε ένα γνωστό 4x4 μητρώο [1, 2, 3, 4, 5 κ.λ.π] και ένα τυχαίο μη τετραγωνικό μητρώο για να επιβεβαιώσουμε ότι λειτουργεί σωστά ο πολλαπλασιασμός.



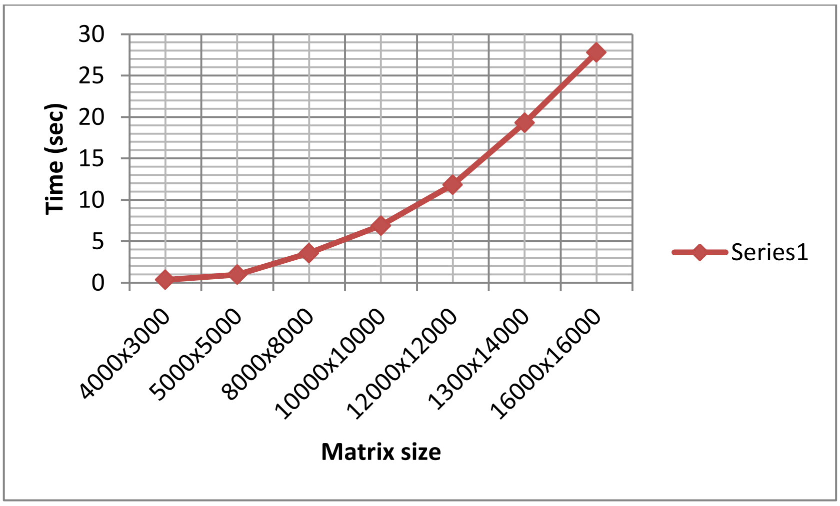

*Εικόνα 2: Μητρώο 4x4*




*Εικόνα 3: Μητρώο 3x4*

Εδώ βρίσκονται οι χρόνοι εκτέλεσης των πράξεων για τα παρακάτω μεγέθη μητρώων.

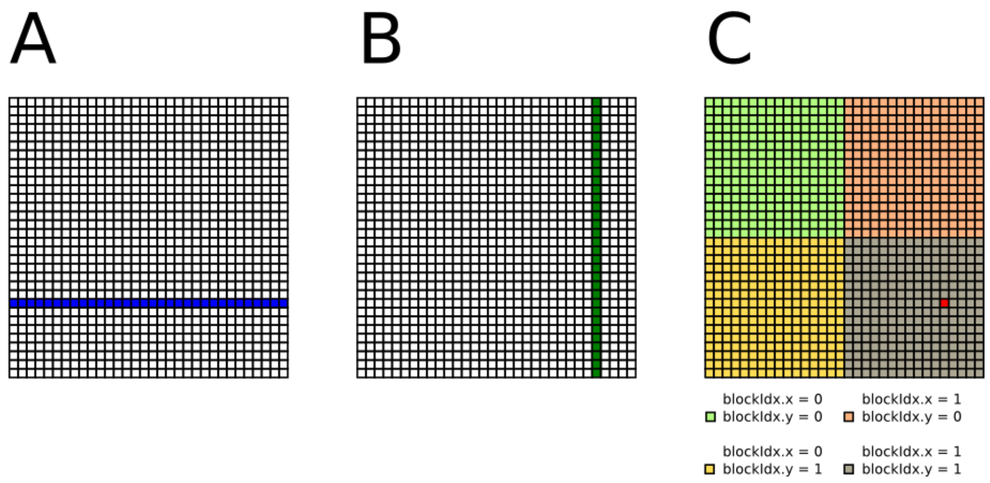
| Size | 4000x3000 | 5000x5000 | 8000x8000 | 10000x10000 | 12000x12000 | 13000x14000 | 16000x16000 |
|---|---|---|---|---|---|---|---|
| Time(sec) | 0,3 | 0,9 | 3,5 | 6,9 | 11,8 | 18,2 | 27,7 |




*Εικόνα 4: Γραφική παράσταση του χρόνου συναρτήσει της εισόδου για την cublasDgemm*

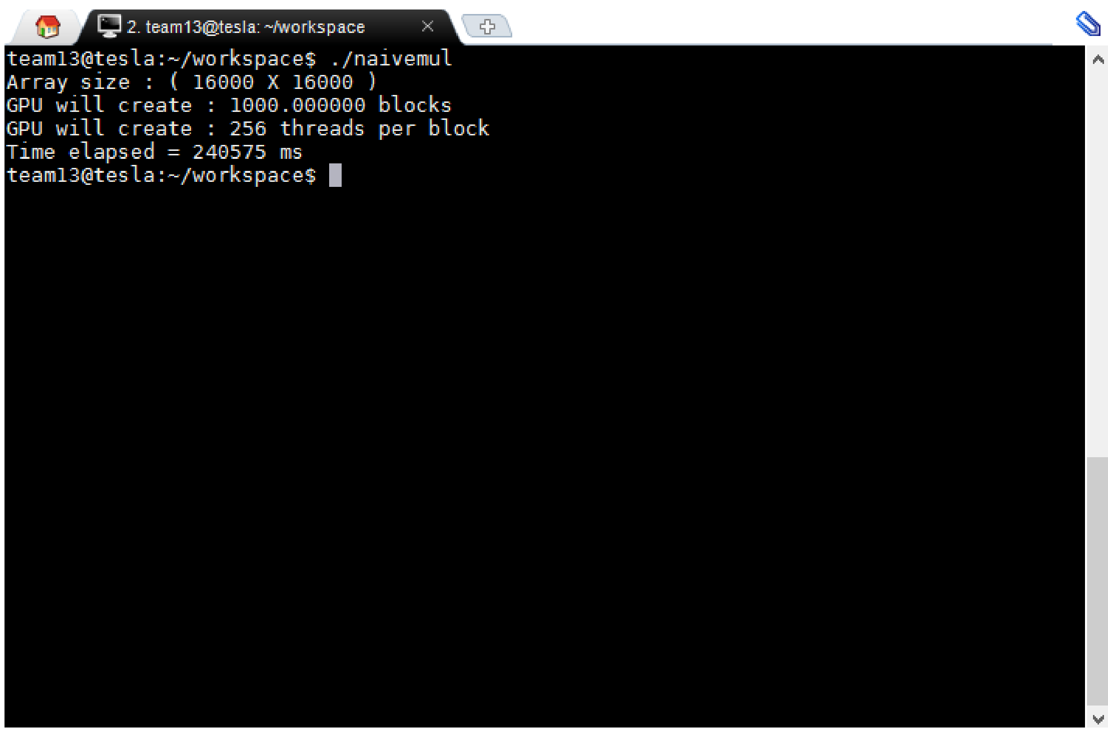
## Ερώτημα 2

Σε αυτό το ερώτημα χρειάστηκε να υλοποιήσουμε την δική μας συνάρτηση υπολογισμού του γινομένου C = $A^T \cdot A$. Χρησιμοποιήσαμε τον πιο απλό τρόπο, δημιουργήσαμε μια συνάρτηση kernel (ονόματι *matrix_mul*) και αναθέσαμε σε κάθε νήμα να διαβάζει από μια γραμμή και μία στήλη του πίνακα A και να υπολογίζει ένα στοιχείο του πίνακα C. Έχουμε τρέξει τον κώδικα για τρία διαφορετικά μεγέθη block.

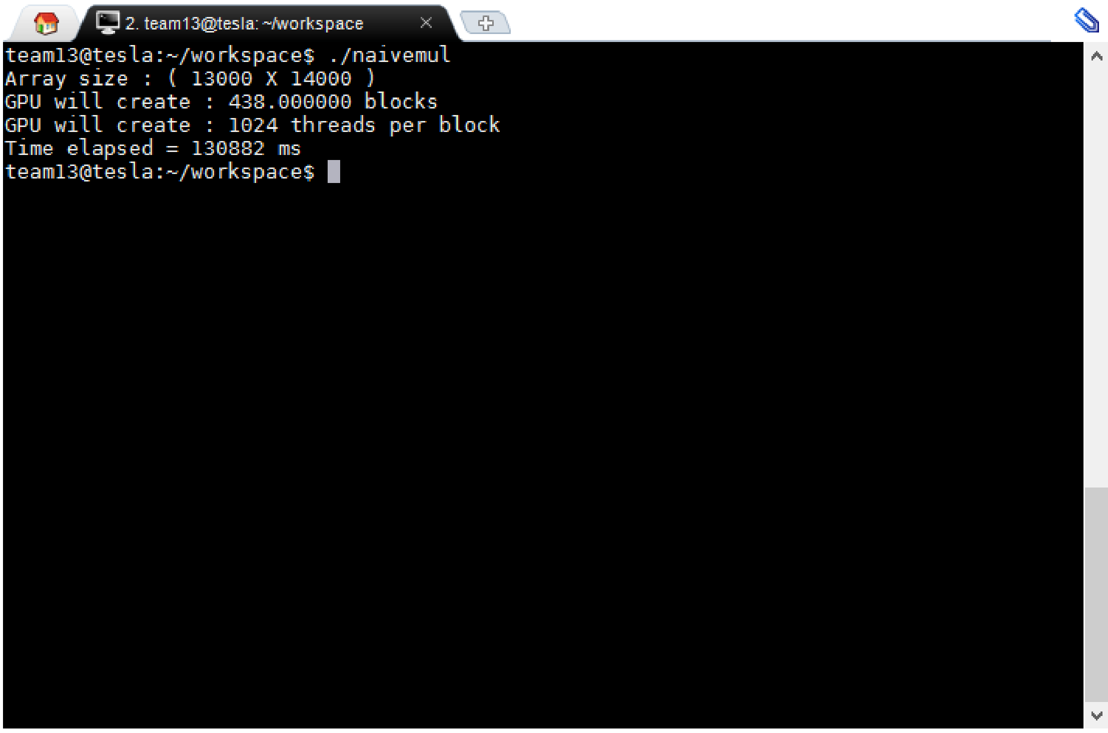



*Εικόνα 5: Παρουσιάζει τον τρόπο που υπολογίζεται κάθε στοιχείο του πίνακα C*

Εδώ παρουσιάζουμε δύο στιγμιότυπα εκτέλεσης



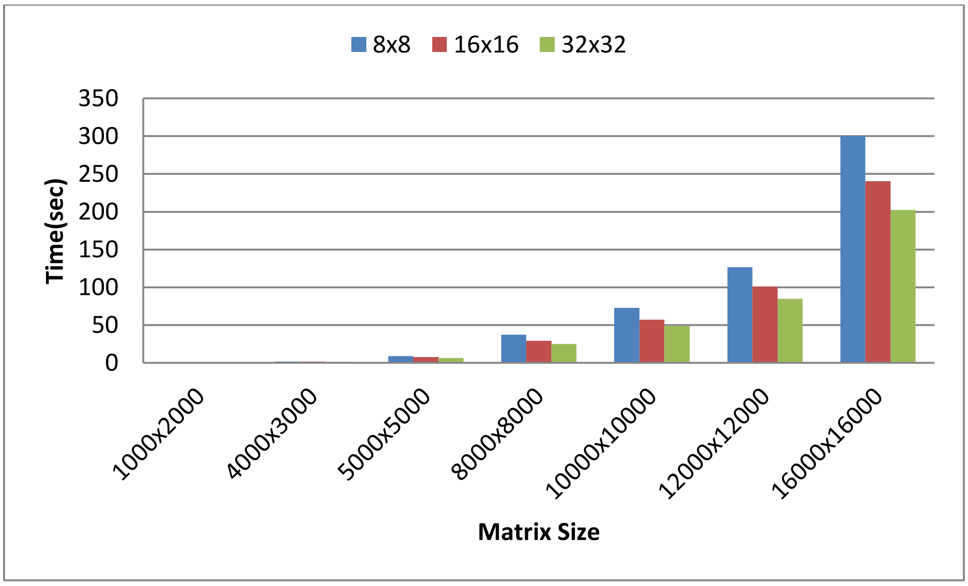

*Εικόνα 6: Block size 16x16, Matrix size 16K x 16K*




*Εικόνα 7: Block size 32x32, Matrix Size 13K x 14K*

Παρακάτω βρίσκονται οι μετρήσεις που λάβαμε για τα επόμενα μεγέθη μητρώων. Παρατηρούμε ότι όσο μεγαλύτερο είναι το *block size* τόσο καλύτερη είναι και η απόδοση του αλγορίθμου μας.

| Size | 4000x3000 | 5000x5000 | 8000x8000 | 10000x10000 | 12000x12000 | 13000x14000 | 16000x16000 |
|---|---|---|---|---|---|---|---|
| Block 8x8 | 1,92 | 8,9 | 37 | 72,9 | 126,6 | 200 | 300,6 |
| Block 16x16 | 1,62 | 7,7 | 29 | 57,3 | 100,6 | 160 | 240,4 |
| Block 32x32 | 1,33 | 6,1 | 25 | 48,8 | 84,8 | 134 | 202,2 |




*Εικόνα 8: Γραφική παράσταση απόδοσης naïve_mul*

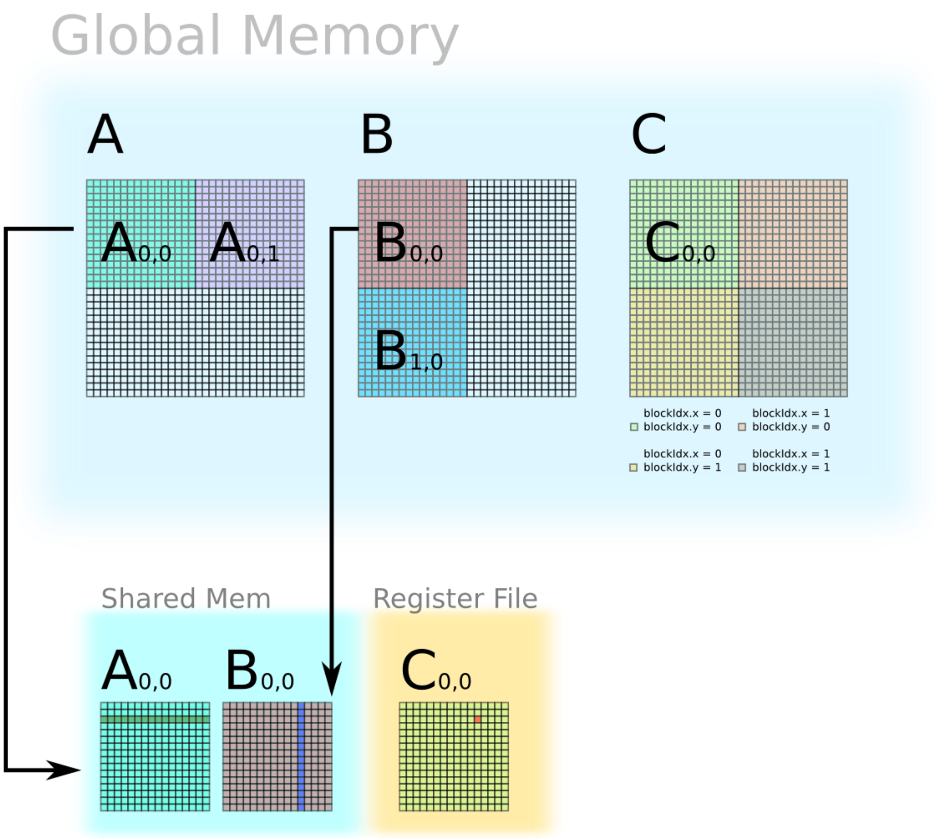
## Ερώτημα 3

Σε αυτό το ερώτημα μας ζητήθηκε να βελτιστοποιήσουμε τον κώδικα μας χρησιμοποιώντας διάφορες γνωστές τεχνικές για τον πολλαπλασιασμό $A^T \cdot A$. Μερικές απο τις πιο γνωστές τεχνικές είναι το tiling, coalesced memory access, prefetching, loop unrolling κ.λ.π. Έχουμε δημιουργήσει δύο εκδόσεις αυτού του ερωτήματος. Το κάναμε αυτό για να δείξουμε ότι σε κάθε επόμενη έκδοση χρησιμοποιούμε και έναν λίγο πιο βελτιστοποιημένο αλγόριθμο και έτσι πετυχαίνουμε καλύτερα αποτελέσματα σε ότι αφορά τους χρόνους εκτέλεσης.

- *1η Έκδοση:* ***clevermulSM.cu*** *| 2η Έκδοση:* ***clevermulSym.cu***

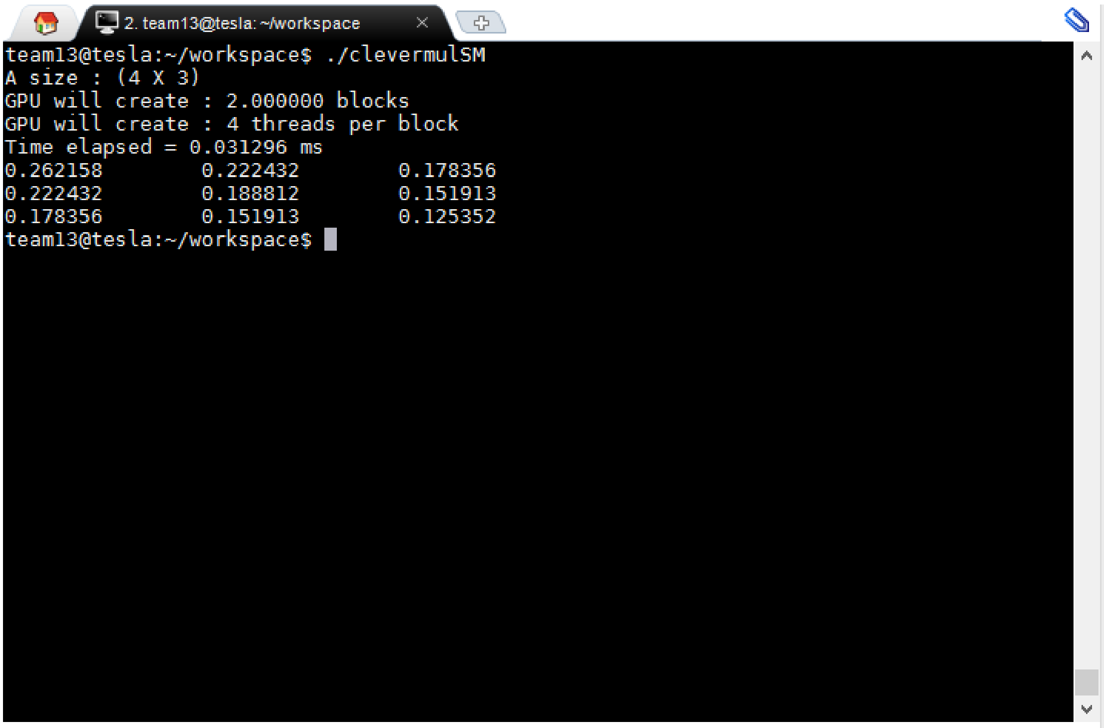
### *1η Έκδοση*

Η τεχνική που εφαρμόστηκε εδώ είναι η διαίρεση του αλγορίθμου σε πλακίδια, η οποία χρησιμοποιεί την κοινόχρηστη (shared memory) μνήμη για να μειώσει συνολικά τις προσπελάσεις προς την καθολική μνήμη. Χρησιμοποιήσαμε δύο πίνακες *Asub, Bsub,* οι οποίοι ορίστηκαν στην κοινή μνήμη και έχουν μέγεθος [TILE_WIDTH][TILE_WIDTH] ο καθένας. Έπειτα διαβάσαμε τον πίνακα A που βρίσκεται στην global memory της GPU και αποθηκεύσαμε TILE_WIDTH x TILE_WIDTH στοιχεία στους πίνακες Asub, Bsub. Κάθε νήμα ουσιαστικά είναι υπεύθυνο σε κάθε φάση του αλγορίθμου να μεταφέρει και ένα στοιχείο από τον πίνακα A που βρίσκεται στην global memory στην κοινή μνήμη. Σε αυτήν την έκδοση του προγράμματος χρησιμοποιήθηκε η τεχνική *coalesced memory access* κατα την διάρκεια που προσπελαύναμε τον πίνακα A για να αποθηκεύσουμε τα στοιχεία του στον Bsub που βρίσκονται στην κοινή μνήμη.

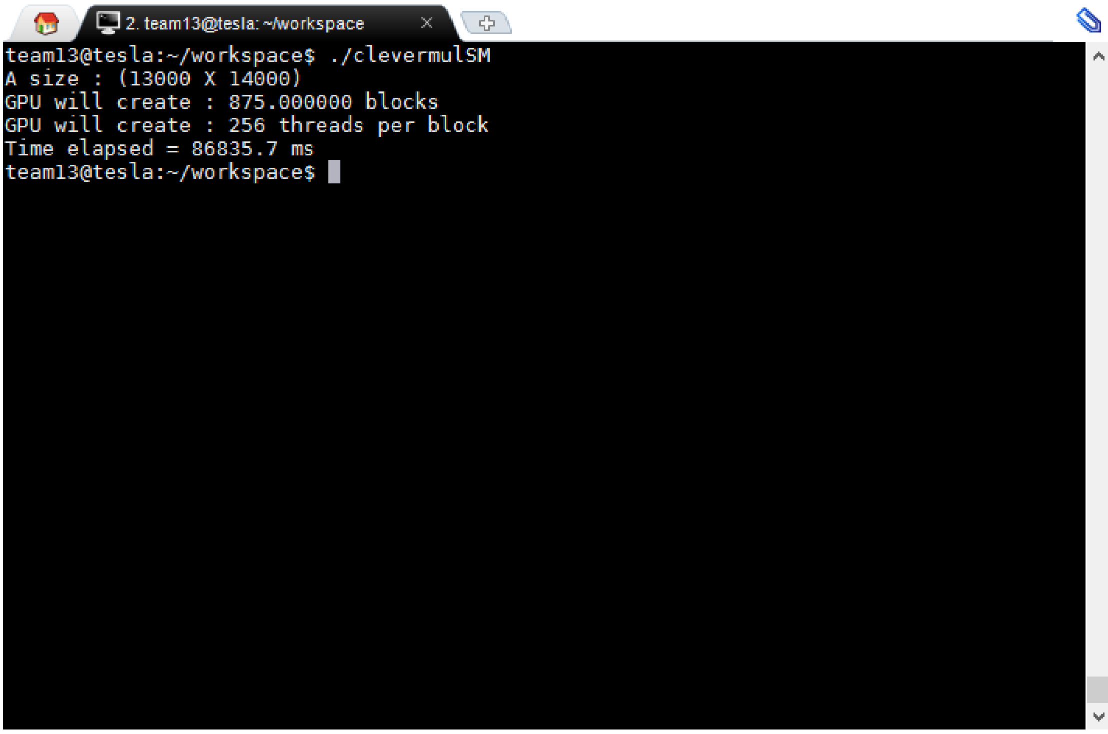



*Εικόνα 9: Τρόπος υλοποίησης tiled αλγορίθμου*

Εδώ παρουσιάζουμε δύο στιγμιότυπα εκτέλεσης για να αποδείξουμε την ορθότητα του αλγορίθμου. Στην αρχή τρέχουμε ενα μη τετραγωνικό μητρωο και έπειτα ένα πολύ μεγάλο μητρώο(16K x 16K)




*Εικόνα 10: 4x3 matrix size & 2x2 block size*

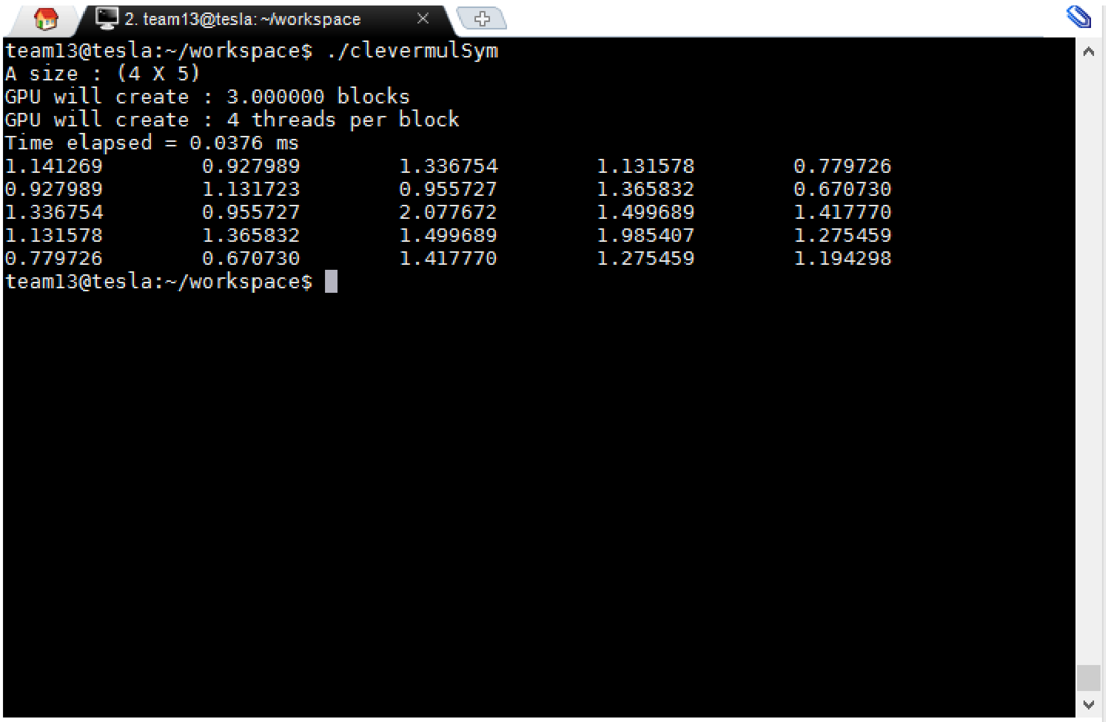



*Εικόνα 11: 13K x 14K matrix size & 16x16 block size*

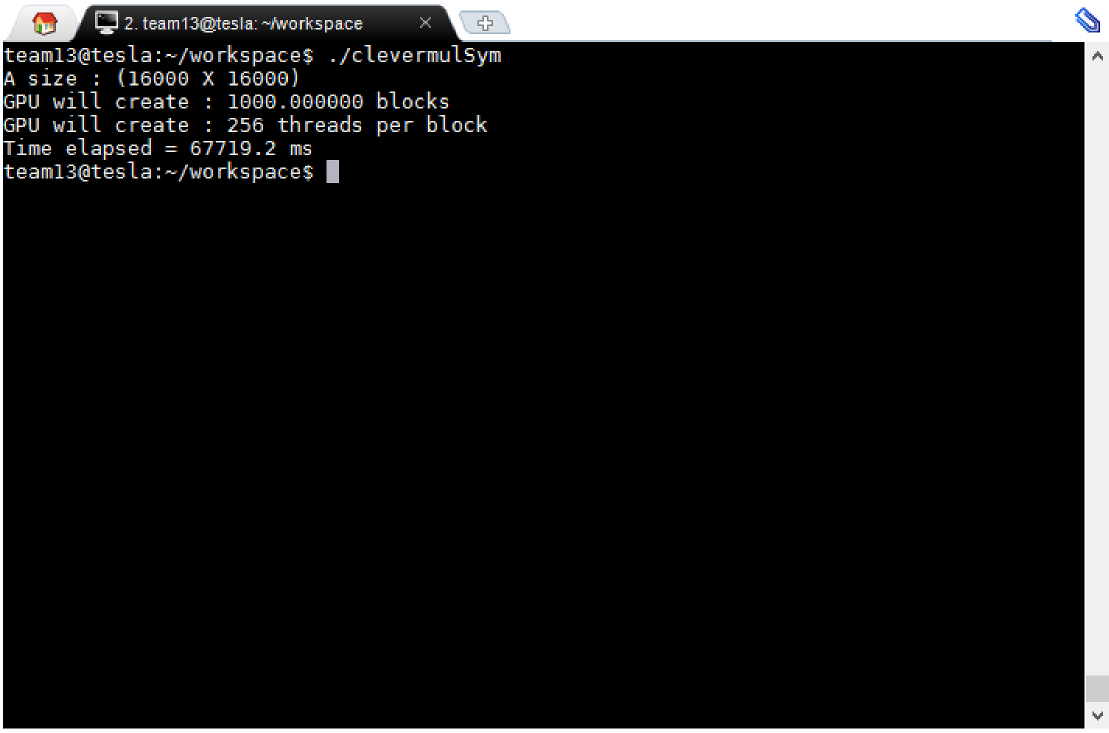
## $2^{η}$ Έκδοση

Σε αυτήν την έκδοση του προγράμματος εκμεταλλευτήκαμε το γεγόνος ότι έχουμε να πολλαπλασιάσουμε έναν πίνακα με τον ανάστροφο του *($A^T \cdot A$)*. Οπότε προκειμένου να βελτιώσουμε την απόδοση του αλγορίθμου μας υπολογίσαμε μόνο τα στοιχεία που βρίσκονται στην κύρια διαγώνιο και πάνω απο αυτήν και μετά τα αντιγράψαμε και κάτω απο την κύρια διαγώνιο. Το γεγονός ότι ο πίνακας που προκύπτει είναι συμμετρικός μας επιτρέπει να το κάνουμε. Τέλος χρησιμοποιήσαμε και την τεχνική του *loop unrolling*. Με αυτόν τον τρόπο βελτιώνουμε λίγο ακόμα την απόδοση του αλγορίθμου μας καθώς έχουμε αφαιρέσει τον δεύτερο βρόγχο for. Κερδίζουμε επειδή έχει αφαιρεθεί η εντολή διακλάδωσης και η ενημέρωση του μετρητή που υπάρχει μέσα στο for loop.

Ακολουθούν ορισμένα screenshot από διάφορες εκτελέσης του αλγορίθμου για να διαπιστωθεί η ορθή επαλήθευση των αποτελεσμάτων. Παρακάτω βλέπουμε ότι ο αλγόριθμος μας δουλεύει και όταν τα μητρώα δεν είναι πολλαπλάσια του block size.

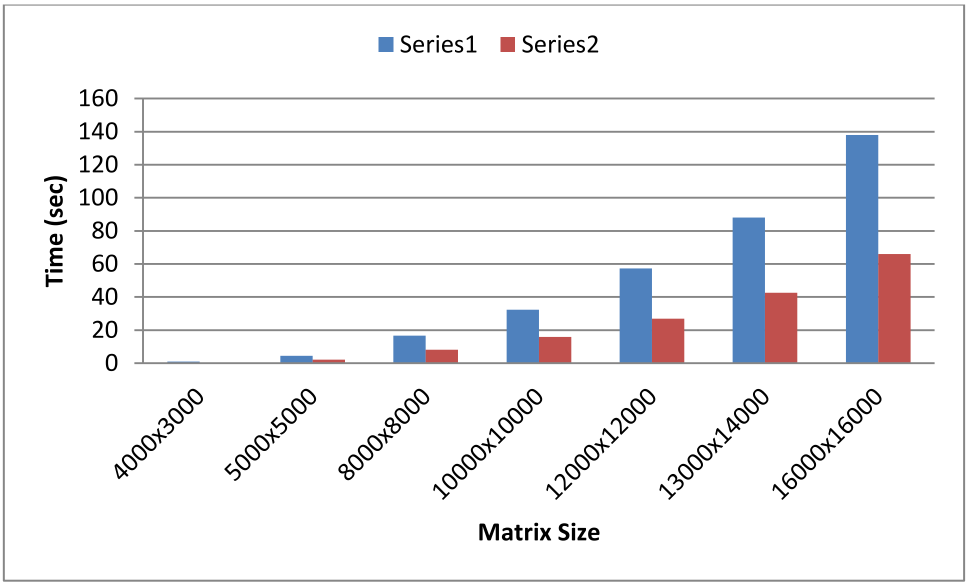



*Εικόνα 12: 4x5 Matrix size με 2x2 block size*




*Εικόνα 13: 16K x 16K*

| Size | 4000x3000 | 5000x5000 | 8000x8000 | 10000x10000 | 12000x12000 | 13000x14000 | 16000x16000 |
|---|---|---|---|---|---|---|---|
| Shared Memory | 0.9 | 4.4 | 16.7 | 32.4 | 57.2 | 88.1 | 138.2 |
| Symmetric/LU | 0.4 | 2.1 | 8.2 | 15.8 | 27.3 | 42.6 | 66.8 |

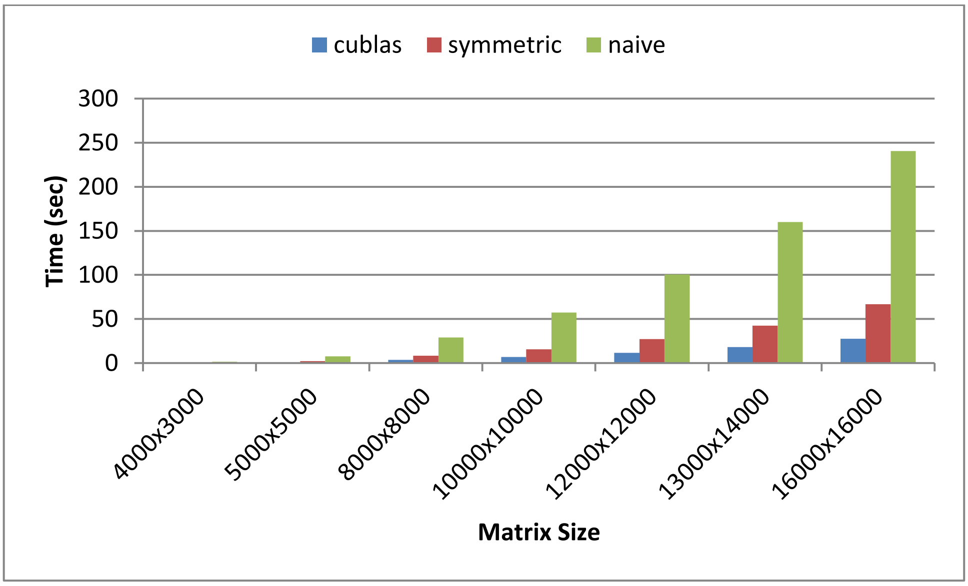



*Εικόνα 14: Series 1 →Shared Memory, Series 2 → Symmetric & Loop Unrolling*

**Παρατήρηση:**

- Σε αυτό το σημείο πρέπει να επισημάνουμε την αισθητή διαφορά που βλέπουμε στον χρόνο εκτέλεσης απο την στιγμή που εκμεταλλευόμαστε την συμμετρικότητα του αποτελέσματος. Για την ακρίβεια κάνουμε λίγες περισσότερες απο τις μισές πράξεις σε σχέση με την πρώτη εκδοση του προγράμματος που χρησιμοποιεί μονο shared memory & coalesced memory access και γι αυτό τον λόγο ο χρόνος εκτέλεσης μείωνεται στην μέση.

Τέλος ακολουθεί μια τελευταία γραφική παράσταση που παρουσιάζει την απόδοση του naivemul (ερωτήματος 2) σε σχεση με τον τελικό αλγόριθμο που υλοποιήσαμε στο ερώτημα 3 (clevermulSym) ο οποίος χρησιμοποιεί shared memory, coalesced memory access, loop unrolling και εκμεταλλεύεται και την συμμετρικότητα του τελικού πίνακα και σε σχέση με την συνάρτηση της cuda cublasDgemm.




*Εικόνα 15: Απόδοση cublas VS symmetric VS naive*