Περιεχόμενα

Περίληψη

Στα πλαίσια της εργασίας αυτής, υλοποιήθηκε η πράξη πολλαπλασιασμού μητρώων $A^T A$ στο προγραμματιστικό μοντέλο CUDA. Για τον σκοπό αυτό δημιουργήθηκαν τρεις διαφορετικές υλοποιήσεις. Η πρώτη υλοποίηση χειρίζεται την βιβλιοθήκη μαθηματικών πράξεων cuBLAS της NVIDIA. Η δεύτερη υλοποίηση είναι ένας απλοϊκός αλγόριθμος υπολογισμού της πράξης, ενώ η τρίτη είναι η πλέον βέλτιστη λύση εξ αυτών, για την κάρτα γραφικών *Tesla C2075*. Εφόσον δημιουργήθηκαν οι παραπάνω υλοποιήσεις και ελέγχθηκε η ορθότητα των αποτελεσμάτων, συγκρίνονται σε χρονικές αποδόσεις με βασικό περιορισμό να υπάρχει μοναδικό αντίγραφο του μητρώου A στην μνήμη.

1 Υλοποίηση cuBLAS

Στο σημείο αυτό θα αναφερθούμε στην βασική δομή που ακολουθούν όλες οι υλοποιήσεις, καθώς και σε μερικά βασικά σημεία στο πρώτο ερώτημα.

1.1 Βασική δομή υλοποιήσεων

Αρχικά, εφόσον δοθούν οι διαστάσεις του μητρώου A ως command line arguments, καλούνται οι συναρτήσεις *malloc* και *cudaMalloc* ώστε να δεσμευτούν οι κατάλληλοι πόροι σε host και device αντίστοιχα. Αξίζει να σημειωθεί πως η δέσμευση μνήμης στον host γίνεται μόνο αν θέλουμε να ελέγξουμε το αποτέλεσμα, δηλαδή εκτυπώνοντας τα μητρώα A, C.

Η αρχικοποίηση του μητρώου A γίνεται εξ ολοκλήρου στο device μέσω του βοηθητικού πυρήνα *fill_matrix* με προκαθορισμένες τιμές για εύκολη επαλήθευση σωστής λειτουργίας. Στην συνέχεια καλείται ο πυρήνας υπολογισμού ή η συνάρτηση υπολογισμού στην περίπτωση της πρώτης υλοποίησης, *gpu_mul* η οποία και χρονομετράτε.

Έπειτα μέσω του βοηθητικού πυρήνα *matrix_equals_to* ελέγχεται αν το μητρώο C περιέχει τις σωστές τιμές αποτελεσμάτων, όπως αυτές προκύπτουν για το A από την συνάρτηση *fill_matrix*. Τέλος αποδευσμεύονται όλοι οι πόροι με τις κατάλληλες κλήσεις των συναρτήσεων *free* και *cudaFree* σε host και device αντίστοιχα.

Τέλος η χρονομέτρηση γίνεται μέσω της κλάσης *gpu_timer* η οποία δημιουργεί δύο events start, stop πριν και μετά την κλήση του υπολογιστικού πυρήνα αντίστοιχα στο default stream (=0). Στην συνέχεια καλείται η *Elapsed* η οποία σταματάει την εκτέλεση του προγράμματος στον host έως ότου ολοκληρωθούν όλες οι εργασίες στο stream, και να πάρουμε την χρονική διαφορά.

1.2 Κλήση cublasDgemm

Η βιβλιοθήκη cuBLAS χρησιμοποιεί column-major απεικόνιση των μητρώων στην μνήμη, διαφορετικό δηλαδή από row-major που χρησιμοποιεί η C. Η σύμβαση αυτή απαιτεί ειδικό χειρισμό στον τρόπο που περνούμε τα μητρώα από την C στην cuBLAS ώστε να γίνονται σωστά οι πράξεις. Οι πυρήνες που αναφέρθηκαν παραπάνω αναπτύχθηκαν για column-major προσπέλαση ώστε η αρχικοποίηση και η εμφάνιση του μητρώου να μην χρειάζεται αναστροφή.

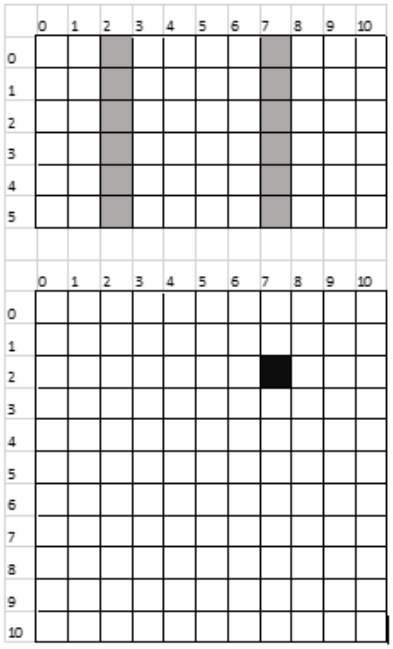
2 Απλοϊκή Υλοποίηση

2.1 Βασική δομή υλοποίησης

Όπως και στην πρώτη υλοποίηση χρησιμοποιούνται οι παρόμοιες βοηθητικές συναρτήσεις σε row-major απεικόνισή. Χρησιμοποιώντας την σχέση 2.1.1 δημιουργήθηκε ο πυρήνας υπολογισμού *gpu_mul*, ο οποίος καλείται σε block των 32x32 νημάτων. Η προφανής απλή λύση του προβλήματος είναι κάθε thread να υπολογίσει ένα στοιχείο του C, παίρνοντας την αντίστοιχη γραμμή και στήλη από το A^T και A από την global memory.

$$C[n][m] = \sum_{i=0}^{K} A[i][n] * A[i][m] \tag{2.1.1}$$

Εφόσον όμως στην μνήμη έχουμε μόνο το μητρώο A για να υπολογιστεί το στοιχείο C[m][n] πραγματοποιείται εσωτερικό γινόμενο της στήλης m και n του . Το τελικό Μητρώο C είναι τετραγωνικό με διάσταση όση και οι στήλες του . Στο σχήμα 2.1 φαίνεται το παράδειγμα της τεχνικής που εξηγήθηκε για $K = rows(A)$:



Σχήμα 2.1 : Τα διανύσματα στήλες του A Μητρώου για τον υπολογισμό του στοιχείου C[2][7]

3 Βέλτιστη Υλοποίηση

3.1 Βασική Δομή Υλοποίησης

Είναι γνωστό πως ο πολλαπλασιασμός μητρώων επιδεικνύει την μεγαλύτερη τοπικότητα αναφορών στα BLAS καθώς επαναχρησιμοποιεί πολλά από τα δεδομένα που φορτώνονται στην μνήμη για πολλαπλούς υπολογισμούς. Αυτό του το πλεονέκτημα δεν εκμεταλλεύτηκε καθόλου στην απλοϊκή υλοποίηση με συνέπεια να φορτώνονται από την global memory πολλές φορές τα ίδια στοιχεία. Οι χρόνοι μεταφοράς δεδομένων από τις μνήμες στις μονάδες επεξεργασίας ακολουθούν την παρακάτω φθίνουσα σειρά:

$$GlobalMemory >> SharedMemory > Registers$$

Αυτό σημαίνει πως μεταφορά των στοιχείων από την global memory κοστίζει και μάλιστα πολύ. Η λύση στο πρόβλημα αυτό είναι η μεταφορά τμήματος δεδομένων (Tile) προς υπολογισμό στην τοπική μνήμη, μειώνοντας την κυκλοφορία προς την global μνήμη κατά μεγάλο βαθμό, επιταχύνοντας την συνολική επίδοση.

Ένα άλλο σημείο που βελτιώθηκε είναι ο χαμένος χρόνος μεταξύ μεταφοράς δεδομένων από την global στην shared μνήμη και συγκεκριμένα ο χρόνος που καταναλώνεται στα φράγματα συγχρονισμού. Για τον σκοπό αυτό χρησιμοποιήθηκε τεχνική εκ των προτέρων προσκόμισής, η οποία προσφέρει ανεξάρτητες εντολές υπολογισμού του μητρώου C μέχρις ότου να οδηγηθούμε στο επόμενο φράγμα. Με αυτόν το τρόπο ο χρόνος αναμονής βελτιώνεται καθώς όταν ένα νήμα περιμένει από την global memory ένα στοιχείο, ένα άλλο νήμα μπορεί να εκτελεί τον πολλαπλασιασμό του. Τέλος για καλύτερη αξιοποίηση πόρων, το κάθε νήμα υπολογίζει περισσότερα από ένα στοιχεία του τελικού μητρώου όπως και θα αναλυθεί παρακάτω.

3.2 Επεξήγηση αλγορίθμου

Προσαρμόζοντας τον αλγόριθμο για την κάρτα γραφικών *Tesla C2075*, ορίζουμε μέγεθος tile στην shared memory διάστασης 48x48 ενώ blocks των 16x16 νημάτων. Δηλαδή κάθε νήμα στο block υπολογίζει συνολικά 9 στοιχεία του τελικού μητρώου C.

Κάθε νήμα έχει τρεις σειρές από καταχωρητές:

- Τους καταχωρητές $rC[3][3]$ όπου αποθηκεύουν προσωρινά το μερικό εσωτερικό γινόμενο, προτού ενημερωθεί το μητρώο C στο τέλος του πυρήνα.

- Τους καταχωρητές $rA[3]$ και $rA\_T[3]$ στους οποίους αποθηκεύονται τμήματα από την shared memory για τον υπολογισμό του εσωτερικού γινομένου. Ουσιαστικά η σειρά καταχωρητών rA έχει μια στήλη από ένα μπλοκ του μητρώου A ενώ οι rA_T έχουν μια γραμμή από το μπλοκ του μητρώου A^T που είναι αποθηκευμένα στην shared memory. Το αποτέλεσμα του γινόμενου τους συσσωρεύεται στο rC.
- Επιπρόσθετα χρησιμοποιούμε δυο ακόμα σειρές από καταχωρητές τους $ra[3]$ και $ra\_T[3]$ για την εκ των προτέρων προσκόμισή των στοιχείων του επόμενου προς υπολογισμό tile από την global memory στην shared.

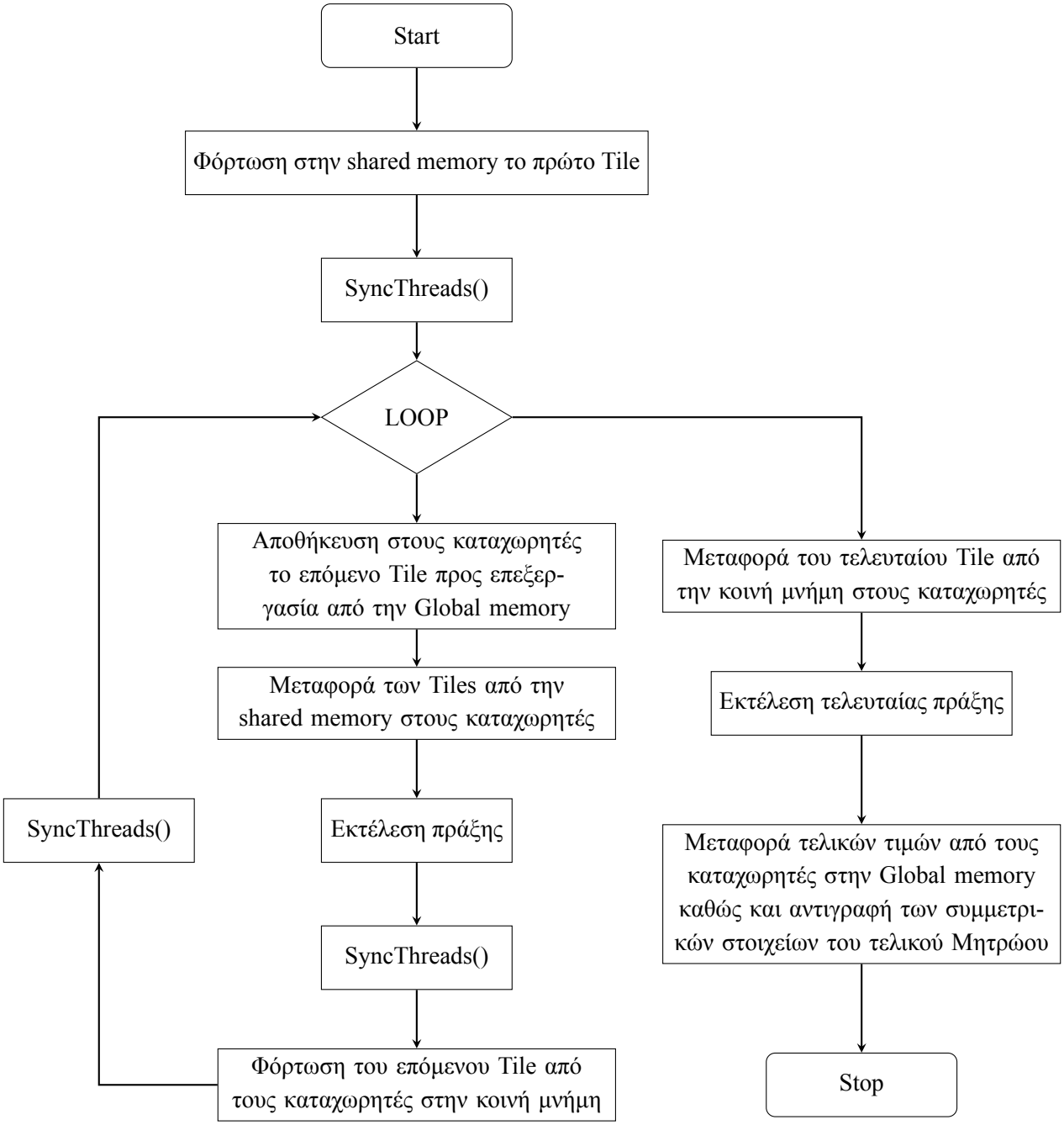
Αρχικά στην εκτέλεση του αλγορίθμου, η σειρά καταχωρητών rC αρχικοποιείται $(= 0)$ ενώ υπολογίζονται τα κατάλληλα offsets, τιμές που καθορίζουν το στοιχείο στο tile στο οποίο αναφέρεται το νήμα, σε κάθε επανάληψη, σε σχέση με το αρχικό μητρώο A. Στην συνέχεια φορτώνεται από την global memory το πρώτο tile στην shared memory (δεν έχουν ακόμα οριστεί οι καταχωρητές rA και rA_T και γι'αυτό τον λόγο κάθε νήμα αρχικά φορτώνει το tile απευθείας από την global μνήμη).

Εφόσον όλα τα νήματα ολοκληρώσουν τις προηγούμενες διαδικασίες αρχίζει μια επανάληψη μέχρι να επεξεργαστούν όλα τα στοιχεία του αρχικού μητρώου A κατά tiles. Αρχικά μπαίνοντας στο βρόγχο αυξάνουμε τα offsets για να δείχνουν στο επόμενο προς φόρτωση και επεξεργασία tile. Έπειτα αποθηκεύουμε το νέο tile στην σειρά καταχωρητών ra και ra_T όπως αναφέρθηκε προηγούμενος. Από το υπάρχον tile στην shared memory, φορτώνουμε στους καταχωρητές rA και rA_T τις αντίστοιχες τιμές και εκτελούμε τον πολλαπλασιασμό προσθέτοντας το αποτέλεσμα στις τιμές των καταχωρητών rC

$$rC[n][m] += rA\_T[m] * rA[n]$$

Όταν όλα τα νήματα του block τελειώσουν με την παραπάνω εκτέλεση φορτώνουν τα νέα tiles από τους καταχωρητές ra και ra_T στην κοινή μνήμη και τελειώνει η επανάληψη. Όταν φτάσει στο τελευταίο μπλοκ προς φόρτωση και επεξεργασία του μητρώου A βγαίνουμε από την επανάληψη , κάνουμε την πράξη του πολλαπλασιασμού με το τελευταίο tile που είχε φορτωθεί στην τελευταία επανάληψη και αποθηκεύουμε τις τελικές τιμές των rC στο μητρώο C.

3.3 Flowchart



3.4 Ζητήματα απόδοσης

3.4.1 Bank conflicts

Βελτιώνοντας την απόδοση της εφαρμογής αντιμετωπίσαμε bank conflicts. Τα bank conflicts εμφανίζονται στην shared memory, όταν νήματα του ίδιου warp ζητάνε διαφορετικά δεδομένα από ίδια banks. Υπάρχουν 32 banks, που αποτελούνται από 32 συνεχόμενα bit ή μία λέξη. Οι διευθύνσεις των bank δίνονται εναλλάξ π.χ. η λέξη[0] ανήκει στο πρώτο bank ή bank-0, η λέξη[31] στο bank-31, η λέξη[32] ανήκει στο bank-0 (32%32) κ.ο.κ..

Όταν λοιπόν διαφορετικές λέξεις - δεδομένα ζητηθούν από το ίδιο bank τα αιτήματα θα σειριοποιηθούν (Bank Conflict) και η απόδοση θα μειωθεί. Το πρόβλημα στην περίπτωση μας είναι μεγαλύτερο, καθώς για το κάθε δεδομένο τύπου double χρειάζονται 2 αιτήματα από το κάθε νήμα (double = 64bit = 2 banks). Έτσι για να αποφύγουμε αυτού του είδους τα conflicts προσθέτουμε ένα padding στις μεταβλητές – πίνακες που αποθηκεύουμε στην shared memory. Το μέγεθος του padding υπολογίστηκε πειραματικά.

3.4.2 Loop unrolling

Μία ακόμη βελτιστοποίηση που έρχεται με μικρό κόστος είναι αυτή της ξεδίπλωσης βρόγχων. Το "ξετύλιγμα" αυτών έγινε αυτόματα από τον *nvcc* χρησιμοποιώντας την ντιρεκτίβα *#pragma unroll*, η οποία επιδρά πάνω σε επαναλήψεις με σταθερό μήκος επαναλήψεων.

3.4.3 Symmetric C speedup

Τέλος μία βελτιστοποίηση στον αλγόριθμο ήταν κατά τον υπολογισμό της πράξης $A^T A$, γνωρίζοντας πως το αποτέλεσμα C είναι ένα συμμετρικό μητρώο, υπολογίζονται τα μισά block. Με έναν απλό έλεγχο στην αρχή του υπολογιστικού πυρήνα σταματάμε τα κάτω τριγωνικά μπλοκ από το να κάνουν τον υπολογισμό των τιμών τους, ενώ αυτές ενημερώνονται από τα συμμετρικά τους στο άνω τριγωνικό τμήμα. Έτσι καταφέρνουμε να μειώσουμε και άλλο το χρόνο εκτέλεσης.

3.5 Αποτελέσματα Μετρήσεων

Στους συγκεντρωτικούς πίνακες 3.5 και 3.5 παρατίθενται οι πειραματικές μετρήσεις για τετραγωνικές και μη διαστάσεις του A. Οι μετρήσεις χρόνου είναι σε ms: Όπως είναι εύκολο να παρατηρηθεί η απλοϊκή υλοποίηση είναι πολύ χρονοβόρα σε όλες τις περιπτώσεις. Για μικρά μητρώα οι χρόνοι είναι σχετικά μικροί αλλά σε πιο ρεαλιστικά μεγέθη προβλήματος έχει πολύ κακή επίδοση. Είναι εύκολο να

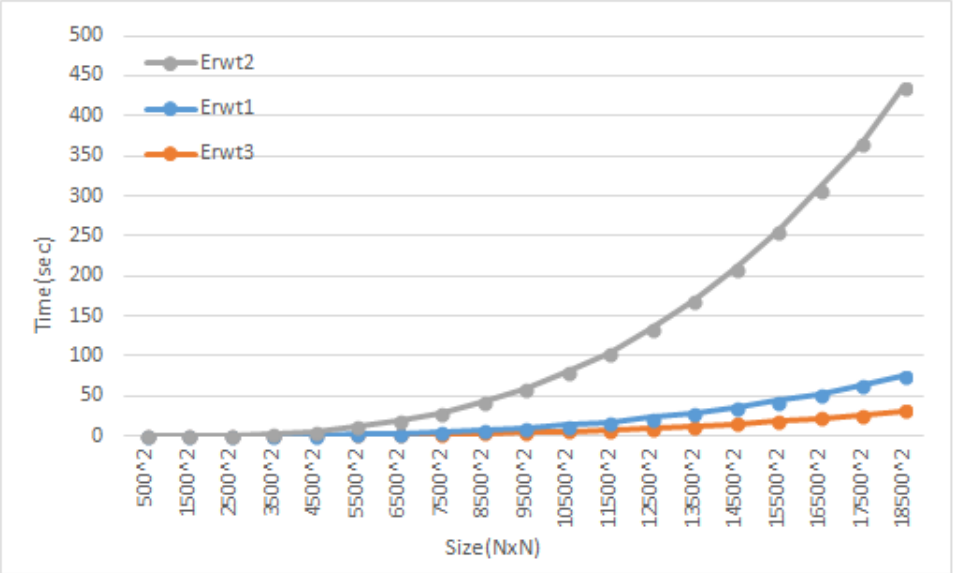
Size	cuBlas	Naive	Optimized
500x500	1,073	6,974	0,835
1500x1500	22,573	180,851	17,921
2500x2500	104,699	868,518	79,603
3500x3500	306,367	2425,482	209,281
4500x4500	639,365	5137,474	443,847
5500x5500	1179,054	9392,390	810,816
6500x6500	1924,438	15629,293	1341,195
7500x7500	2902,237	23864,572	2061,611
8500x8500	4302,248	34880,836	2977,607
9500x9500	5931,083	48458,543	4131,801
10500x10500	7957,450	65922,313	5578,560
11500x11500	10558,385	86009,234	7340,344
12500x12500	13514,066	110914,188	9412,655
13500x13500	17130,453	138974,625	11852,910
14500x14500	21147,750	173428,047	14993,040
15500x15500	25684,318	210843,969	18281,053
16500x16500	31266,859	255267,141	21683,658
17500x17500	37002,031	302650,219	26032,162
18500x18500	43581,590	360222,531	30854,096

Πίνακας 1: Χρόνος εκτέλεσης για μητρώο A(NxN) σε ms

Size	cuBlas	Naive	Optimized
250x500	0,570	3,386	0,463
750x1500	11,451	90,520	9,294
1250x2500	52,332	423,349	40,172
1750x3500	141,682	1152,235	105,523
2250x4500	320,589	2452,576	222,644
2750x5500	590,782	4563,408	406,717
3250x6500	964,656	7761,969	671,951
3750x7500	1456,138	11834,445	1030,867
4250x8500	2155,037	17387,340	1491,243
4750x9500	2967,366	24126,561	2059,071
5250x10500	3988,701	32879,125	2784,587
5750x11500	5274,079	42965,281	3652,860
6250x12500	6744,590	55363,922	4689,003
6750x13500	8545,609	69391,531	5904,020
7250x14500	10546,562	86676,516	7398,440
7750x15500	12787,450	105108,047	9099,423
8250x16500	15574,900	127502,445	10770,195
8750x17500	18555,031	151280,891	12833,893
9250x18500	21775,717	179886,000	15313,980
9750x19500	25576,779	209357,594	17860,482
10250x20500	29781,045	244419,813	20572,283

Πίνακας 2: Χρόνος εκτέλεσης για μητρώο A(NxM) σε ms

δικαιολογηθεί λόγο των πολλαπλών προσπελάσεων στην καθολική μνήμη χωρίς επαναχρησιμοποίηση των στοιχείων.



Σχήμα 3.2 : Χρόνοι εκτέλεσης συναρτήσει μεγέθους μητρώου A

Στο γράφημα 3.2 παρατηρείται πόσο κακή είναι η επίδοση της απλοϊκής υλοποίησης σε σχέση με τις άλλες δύο. Ενώ η υλοποίηση της *Optimized* σε σχέση με την *cuBLAS* είναι πολύ κοντά ενώ η διαφορά με την τελευταία φαίνεται για μεγάλα μεγέθη μητρώων. Η διαφορά αυτή των δύο υλοποιήσεων έγκειται στην χρήση της συμμετρίας για τον υπολογισμό του μητρώου.

Βιβλιογραφία

[1]. Reference: GPU Gems : Avoiding Bank Conflicts

[2]. Reference: MAGMA : Matrix Algebra on GPU and Multicore Architectures

[3]. Reference: CUDA Toolkit Documentation - cuBLAS

[4]. Reference: NVIDIA CUDA - Programming Guide

[5]. Reference: CUDA PROGRAMMING - A Developer's Guide to Parallel Computing with GPUs , by Shane Cook

[6]. Reference: Matrix Multiplication with CUDA , by Robert Hochberg

[7]. Reference: Matrix computations on the GPU CUBLAS and MAGMA by example , by Andrzej Chrzeszczyk and Jakub Chrzeszczyk

[8]. Reference: CUDA BY EXAMPLE - An Introduction to General -Purpose GPU Programming , by Jason Sanders and Edwaed Kandrot