ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Παράλληλα και Κατανεμημένα Συστήματα Υπολογιστών

Άσκηση 4

(Επιλογή 2)

ΧΑΤΖΗΘΩΜΑ ΑΝΤΡΕΑΣ

ΑΕΜ: 8026

antreasc@ece.auth.gr

Υπεύθυνος καθηγητής: κ. Πιτσιάνης Νίκος
Ημερομηνία Παράδοσης: 09/10/2016

ΕΙΣΑΓΩΓΗ

Το ζητούμενο της εργασίας αυτής είναι η βελτιστοποίηση της προηγούμενης εργασίας (Εργασία 3) διευκολύνοντας την πρόσβαση στην μνήμη χρησιμοποιώντας δύο αντίγραφα του πίνακα γειτνίασης, το ένα αντίγραφο του οποίου τα στοιχεία προσπελάζονται συνεχόμενα κατά γραμμές και ένα του οποίου τα στοιχεία προσπελάζονται συνεχόμενα κατά στήλες. Έτσι, χρησιμοποιώντας τους πίνακες $\boldsymbol{A}$ και $\boldsymbol{A^T}$, μπορεί να επιτευχθεί συνεχόμενη πρόσβαση στη μνήμη, σε κάθε βήμα του αλγορίθμου (*coalesced memory access*).

Για την προαναφερθείσα βελτιστοποίηση, δημιουργούμε τον πίνακα γειτνίασης (**A**) καλώντας την $\boldsymbol{makeAdjency}()$ και ακολούθως τον ανάστροφο (**AT**) καλώντας την $\boldsymbol{transposeAdjacency}()$.

Για την εκτέλεση του προγράμματος χρησιμοποιήθηκε το υπολογιστικό μηχάνημα $Diades$.

Το compile έγινε με την εντολή:

$$nvcc\ apsp4hadjithoma.cu.cu\ -o\ apsp$$

και run με την εντολή:

$$./apsp\ n\ p\ w$$

Όπου τα n, p, w ισοδυναμούν με: $n = 2^{[7:12]}$,

$p = [0.33\ 0.45\ 0.66]$,

$w = 2\ (τυχαιο)$.

Όλοι οι χρόνοι εκτέλεσης των υλοποιήσεων που θα παρουσιαστούν, βρίσκονται στο αρχείο $times.xls$.

Έλεγχος Ορθότητας

Για τον έλεγχο ορθότητας των αποτελεσμάτων, χρησιμοποιήθηκε η συνάρτηση $\boldsymbol{test(float * A)}$ η οποία υλοποιήθηκε στην Εργασία 3. Η συνάρτηση αυτή, δεν κάνει τίποτα άλλο από το να συγκρίνει τον εκάστοτε πίνακα που δέχεται σαν όρισμα, με τον πίνακα που προκύπτει από την επίλυση του προβλήματος με την μέθοδο Warshall – Floyd. Αν όλα τα στοιχεία των 2 πινάκων είναι ίσα, τότε εμφανίζεται το μήνυμα **Passed,** αλλιώς το μήνυμα **Failed**.

Επίσης, υλοποιήθηκε η συνάρτηση $\boldsymbol{printArray(float * array)}$ η οποία εκτυπώνει τον εκάστοτε πίνακα (μόνο για πίνακες $n\ x\ n \leq 3$) που δέχεται σαν όρισμα. Με αυτό τον τρόπο επιτρέπεται ο χειροκίνητος έλεγχος ορθότητας για μικρό αριθμό στοιχείων.

Επίλυση APSP - Γενικά

Για την υλοποίηση των 3ων μεθόδων (συναρτήσεις πυρήνα) που δίνονται στην εκφώνηση της 3ης άσκησης χρησιμοποιώντας όμως τον πίνακα γειτνίασης A^T όπως αναφέρεται στην εκφώνηση της 4ης άσκησης, υλοποιήθηκε η συνάρτηση $\boldsymbol{methods(int\ f, int\ t)}$**.** Το $\boldsymbol{f}$ παίρνει τις τιμές 1, 2 και 3 οι οποίες τιμές αντιπροσωπεύουν τις μεθόδους 1, 2 και 3 αντίστοιχα και το $\boldsymbol{t}$ παίρνει και αυτό τις τιμές 1, 2, 3 όπου το:

1: αντιπροσωπεύει τις μεθόδους που υλοποιήθηκαν στην 3η άσκηση.

2: αντιπροσωπεύει τις μεθόδους της 4ης άσκησης χρησιμοποιώντας τις υλοποιήσεις από την 3η άσκηση σε συνδυασμό με τη χρήση του A^T.

3: αντιπροσωπεύει μια καινούρια προσέγγιση στην υλοποίηση των μεθόδων αυτών, οι οποίες θα επεξηγηθούν αργότερα.

Σημείωση: Η επεξήγηση του αλγόριθμου Warshall-Floyd και των υλοποιήσεων CUDA για την επίλυση του προβλήματος asps, παραλείπεται, καθώς επεξηγήθηκε στην αναφορά της Εργασίας 3.

Μέσα στη συνάρτηση $\boldsymbol{methods(int\ f, int\ t)}$, γίνονται οι απαραίτητες προετοιμασίες για την κλήση της κάθε συνάρτησης kernel ανάλογα με την τιμή του $\boldsymbol{f}$ αλλά και του $\boldsymbol{t}$, και έπειτα καλείται η αντίστοιχη συνάρτηση πυρήνα.

Όσον αφορά τις προετοιμασίες που γίνονται κλπ για $t = 1$, είναι ακριβώς ίδιες με αυτές που αναφέρθηκαν στην 3η Εργασία, οπότε η περαιτέρω επεξήγηση θεωρείται περιττή.

Όταν όμως έχουμε t$= 2$, πρέπει επιπρόσθετα από αυτά που γίνονται στο $t = 1$, να γίνει και η αντιγραφή του πίνακα $\boldsymbol{A^T}$ (ο οποίος δημιουργήθηκε μέσω της $\boldsymbol{transposeAdjacency}()$) από την CPU στην GPU (πίνακας $\boldsymbol{AT\_dev}$) χρησιμοποιώντας την συνάρτηση $\boldsymbol{cudaMemcpy.}$ Η αντιγραφή γίνεται από όλες τις μεθόδους.

Για $t = 3$, θα γίνει αναφορά αργότερα.

Σημειώνεται ότι: Στο χρόνο σύγκρισης των μεθόδων, συμπεριλαμβάνεται και η μεταφορά των δεδομένων προς και από την device memory.

Οι συναρτήσεις πυρήνα που υλοποίησαν στην 3η εργασία είναι οι ακόλουθες:

- $\boldsymbol{kernel1(int\ k, int\ n, float\ *A)}$
- $\boldsymbol{kernel2(int\ k, int\ n, float\ *A)}$
- $\boldsymbol{kernel3(int\ k, int\ n, float\ *A,\ int\ cells)}$

Αντίστοιχα, οι συναρτήσεις πυρήνα με τη χρήση του A^T είναι:

- $\boldsymbol{kernel1\_t(int\ k, int\ n, float\ *A, float*AT)}$
- $\boldsymbol{kernel2\_t(int\ k, int\ n, float\ *A, float*AT)}$
- $\boldsymbol{kernel3\_t(int\ k, int\ n, float\ *A, float*AT, int\ cells)}$

Και τέλος οι συναρτήσεις με τη νέα προσέγγιση είναι

- $\boldsymbol{kernel1\_t\_v2(int\ k, int\ n, float\ *A, float*AT)}$
- $\boldsymbol{kernel2\_t\_v2(int\ k, int\ n, float\ *A, float*AT)}$
- $\boldsymbol{kernel3\_t\_v2(int\ k, int\ n, float\ *A, float*AT, int\ cells)}$

Υλοποιήσεις Πυρήνων με την χρήση του A^T

Γενικά, για να επιτευχθεί *coalesced memory access* χρησιμοποιώντας τον A^T πρέπει η γίνεται διαφορετική προσπέλαση στον πίνακα Α σε σχέση με τον πίνακα A^T. Συγκεκριμένα για τον Α η προσπέλαση γίνεται ανά $i * n + j$ στοιχείο ενώ για τον A^T η προσπέλαση γίνεται ανά $j * n + i$ στοιχείο. Έτσι, με αυτόν τον τρόπο επιτυγχάνεται *coalesced memory access με τα γειτονικά threads να διαβάζουν γειτονικά στοιχεία.*

Α. Υλοποιήσεις $Kernel1, 2, 3$

Ό,τι αναφέρθηκε στην Εργασία 3 με κάποιες μικροδιορθώσεις στον 2ο πυρήνα λαμβάνοντας υπόψη την παρατήρηση στο eThmmy.

Β.1. Υλοποιήσεις $kernel1, 2, 3\_t$

Η διαφορά στις υλοποιήσεις αυτές σε σύγκριση με τις υλοποιήσεις της 3ης εργασίας ($kernel1,2,3$) είναι ότι κατά τη σύγκριση των στοιχείων του πίνακα, η μια προσπέλαση γίνεται από τον ανάστροφο πίνακα γειτνίασης. Δηλαδή, πριν είχαμε:

$$A[i * n + j] > A[i * n + k] + A[k * n + j]$$

και τώρα γίνεται:

$$A[i * n + j] > AT[k * n + i] + A[k * n + j],$$

όπου φαίνεται ξεκάθαρα ότι η μια προσπέλαση είναι στον πίνακα Α και η άλλη στον A^T. Ακολούθως, σε περίπτωση που ισχύει η ανισότητα, γίνεται ανανέωση περιεχομένων του πίνακα Α καθώς και του πίνακα A^T όπως φαίνεται παρακάτω:

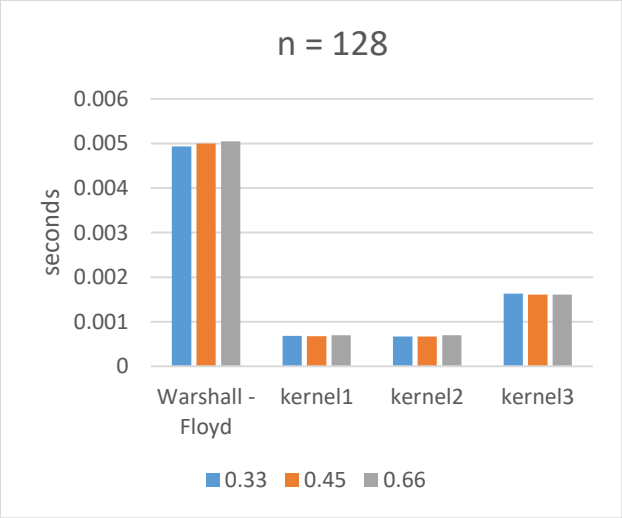
$$A[i * n + j] = AT[n * j + i] = AT[k * n + i] + A[k * n + j]$$

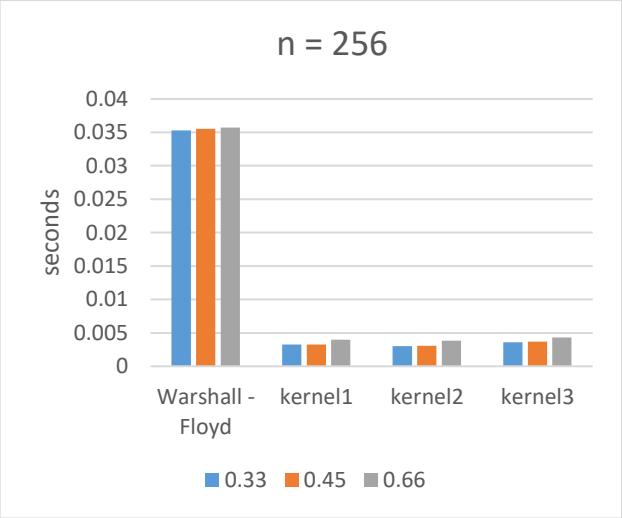
Αυτή η λογική ακολουθείται και για τους τρεις πυρήνες, με τη διαφορά ότι στον 1ο πυρήνα δεν γίνεται χρήση της shared memory, ενώ στον 2ο και στον 3ο πυρήνα γίνεται και χρήση της shared memory. Ο τρόπος με τον οποίο χρησιμοποιήθηκε η shared memory αφορά την 3η εργασία, καθώς ακολουθείται η ίδια φιλοσοφία.

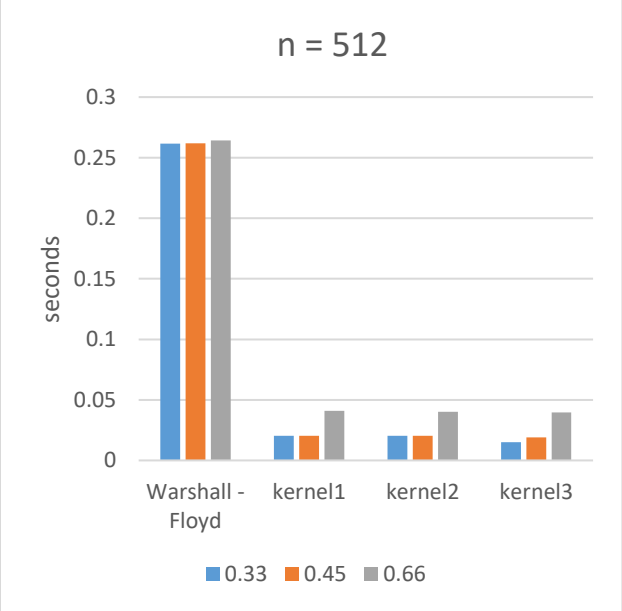
Β.2. Αποτελέσματα – Παρατηρήσεις

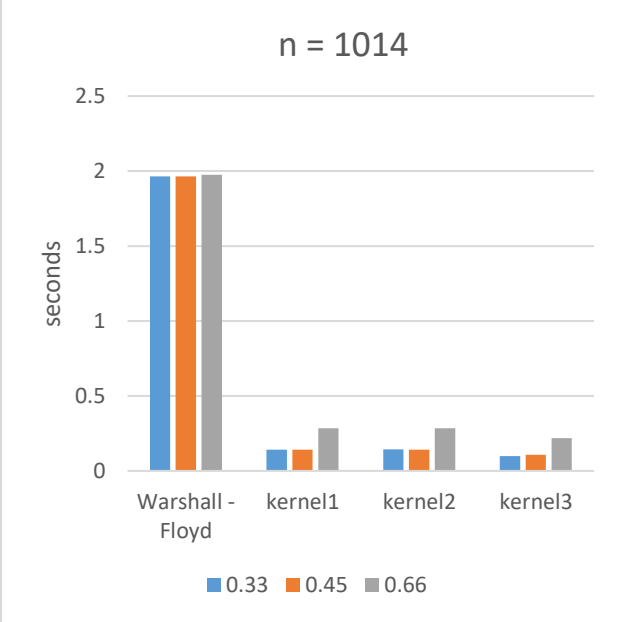
Από τις μετρήσεις που πάρθηκαν στο υπολογιστικό σύστημα $Diades$, προκύπτουν τα ακόλουθα διαγράμματα τα οποία αφορούν μόνο τις υλοποιήσεις $kernel1,2,3\_t$ για $n = 2^{[7:12]}$ και $p = [0.33\ 0.45\ 0.66]$.

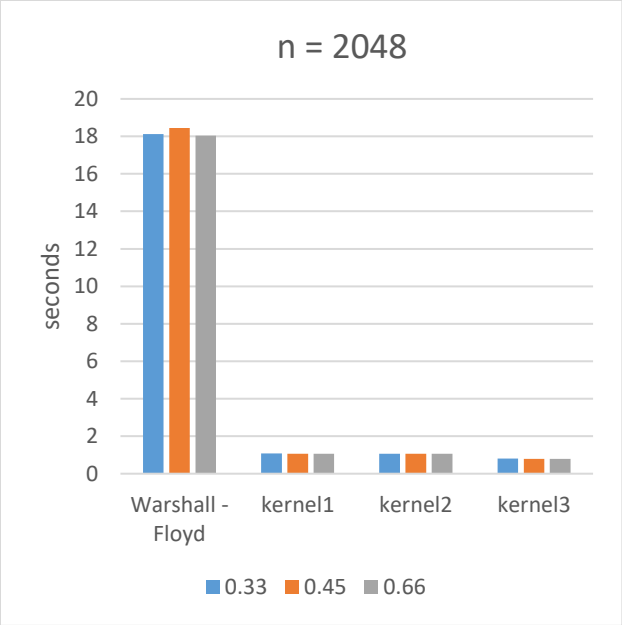
**Τα διαγράμματα δεν είναι άμεσα συγκρίσιμα μεταξύ τους, εφόσον ο άξονας του y (seconds) δεν είναι βαθμολογημένος κατά τον ίδιο τρόπο.

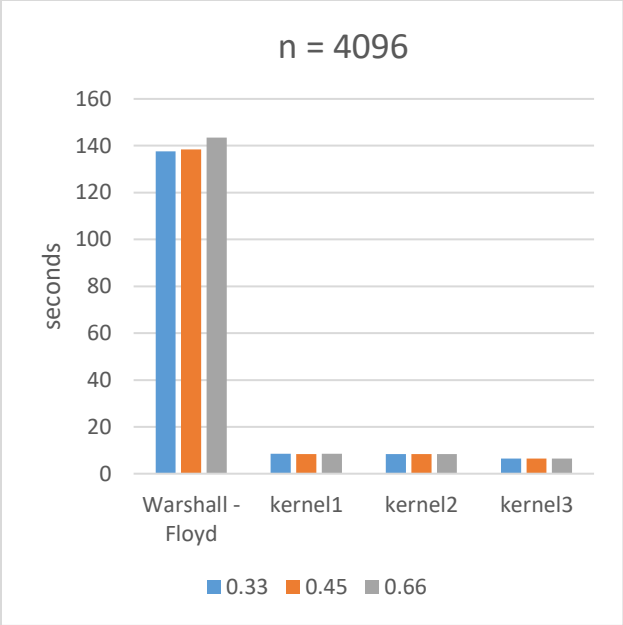










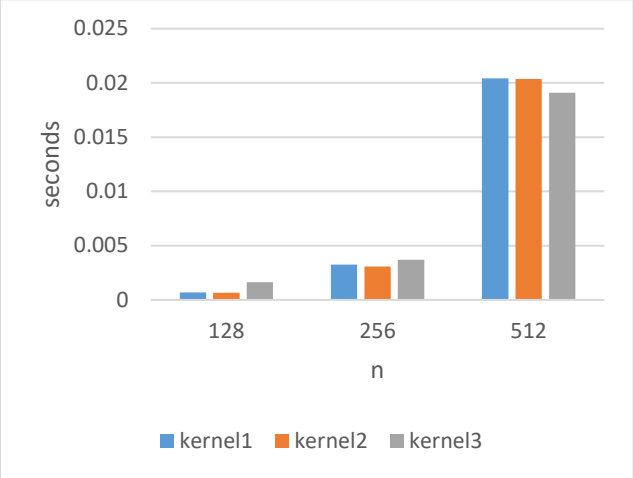


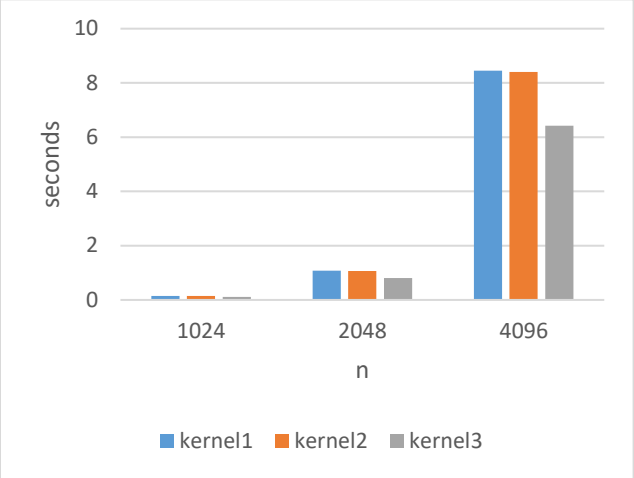
Από τα διαγράμματα παρατηρούμε μεγάλη βελτίωση στο χρόνο εκτέλεσης με τις υλοποιήσεις CUDA έναντι της σειριακής υλοποίησης.

Αρχικά, βλέπουμε ότι για τις τιμές του $p = [0.33\ 0.45\ 0.66]$ δεν παρατηρείται ιδιαίτερη διαφορά στους χρόνους εκτέλεσης. Αλλάζοντας την παράμετρο p, οι χρόνοι εκτέλεσης των αλγορίθμων μεταβάλλονταν όλοι κατά τον ίδιο βαθμό, με αποτέλεσμα να μην επηρεάζεται η σύγκριση πριν και μετά την χρήση του A^T. Αυτό, φαίνεται ξεκάθαρα και από τα Data Bars σε κάθε κελί του αρχείου $times.xls$. Για τον λόγο αυτό, αλλά και για εξοικονόμηση χώρου, από εδώ και στο εξής τα διαγράμματα που θα παρουσιαστούν θα αφορούν την τιμή $p = 0{,}33$ μόνο, παρόλα αυτά, όλοι οι χρόνοι εκτέλεσης βρίσκονται στο σχετικό αρχείο excel που αναφέρθηκε.

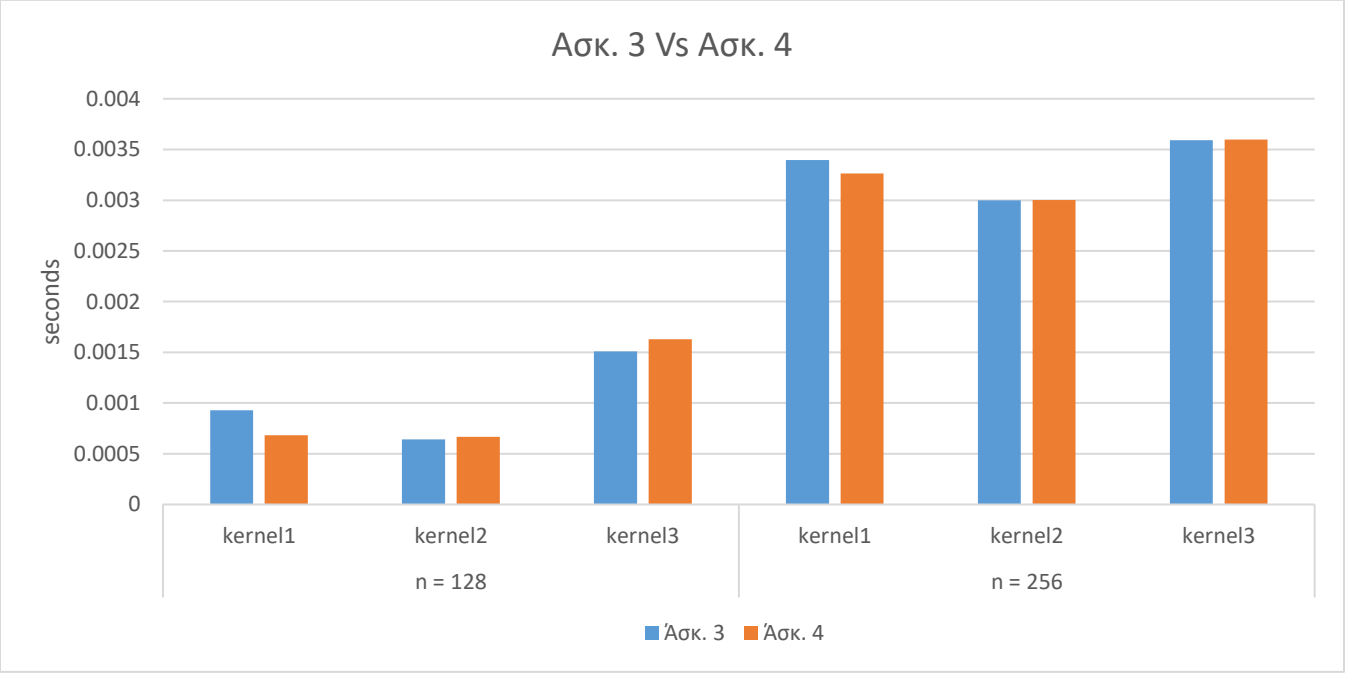
Σχετικά με τους χρόνους εκτέλεσης, παρατηρούμε ότι ο kernel2 είναι ελάχιστα πιο γρήγορος από τον kernel1 και σε κάποιες περιπτώσεις είναι σχεδόν ίσοι. Όσον αφορά τον kernel3, αρχικά έχουμε βελτίωση σε σχέση με τη σειριακή υλοποίηση αλλά είναι αρκετά πιο αργός σε σχέση με τους kernel1 και 2. Όσο το n αυξάνεται, ο kernel3 μειώνεται σε σχέση με τους υπόλοιπους kernels, και για $n \geq 512$ γίνεται γρηγορότερος. Αυτές οι παρατηρήσεις, φαίνονται καλύτερα και από τα παρακάτω 2 διαγράμματα.

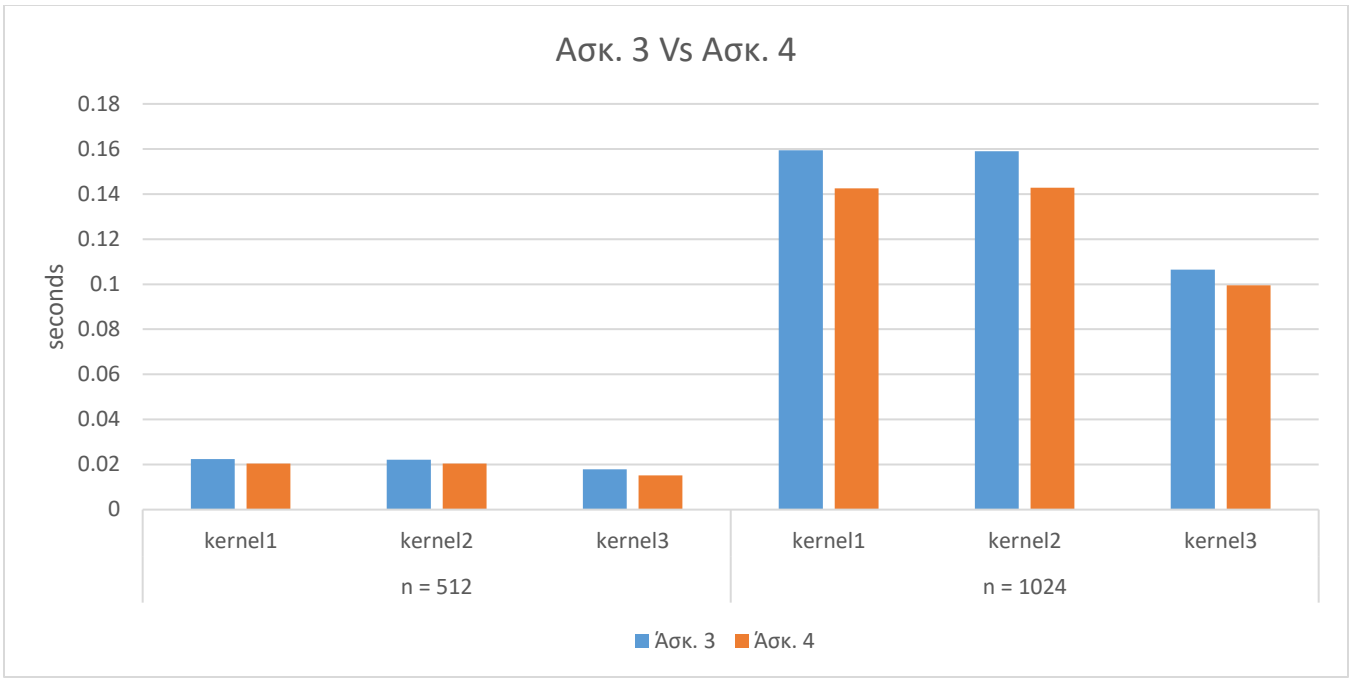
**Υπενθυμίζεται ότι ο άξονας του y (seconds) στα 2 διαγράμματα δεν είναι βαθμολογημένος κατά τον ίδιο τρόπο.

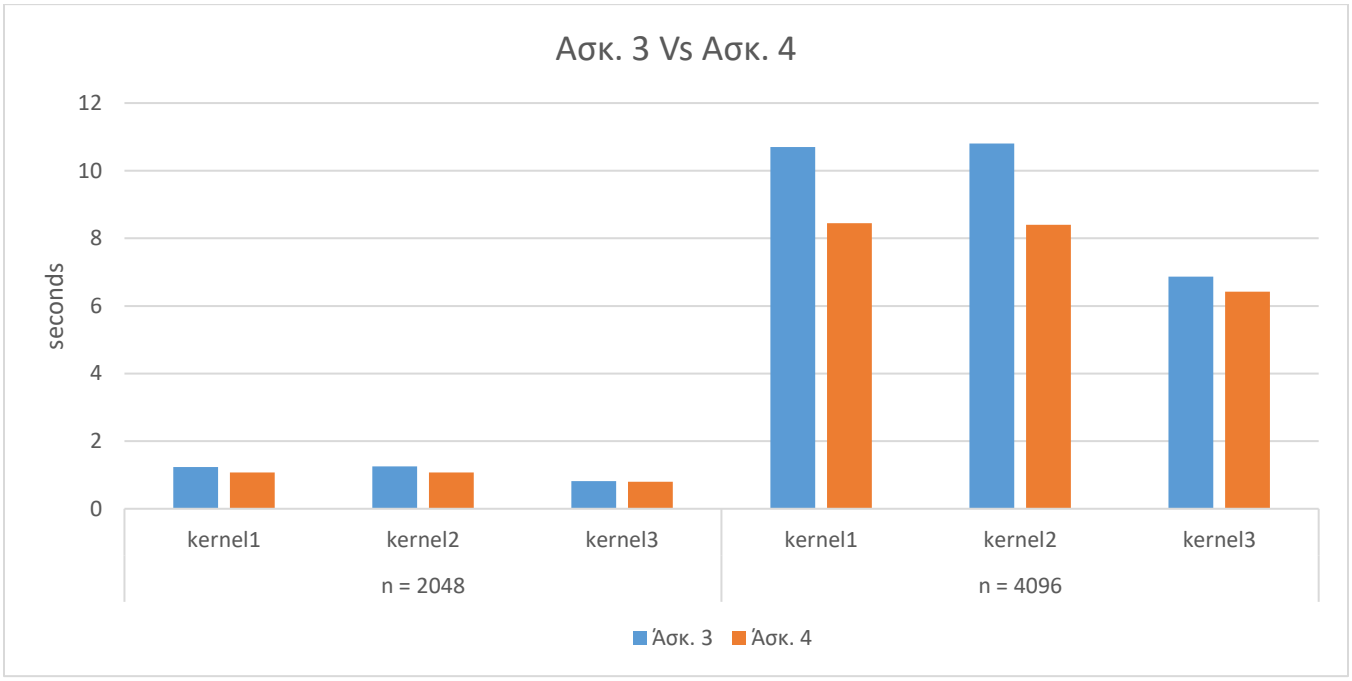




Παρατηρούμε λοιπόν, πως προέκυψαν παρόμοια συμπεράσματα με την 3η εργασία. Αυτό είναι λογικό εφόσον η γενική ιδέα υλοποίησης των kernels, πέραν της χρήσης του A^T, είναι ίδια. Γι' αυτό, περισσότερο νόημα θα είχε να συγκρίνουμε τους χρόνους αυτούς με τους αντίστοιχους της 3ης εργασίας. Παρακάτω παρουσιάζονται τα διαγράμματα σύγκρισης των υλοποιήσεων της 3ης και 4ης εργασίας.







Όπως εύκολα παρατηρούμε από τα διαγράμματα, η χρήση του πίνακα A^T σε συνδυασμό με coalesced memory access βελτιώνει τον χρόνο εκτέλεσης του αλγόριθμου, εκτός από τις περιπτώσεις που το $n = 128$ και λογικά για κάθε $n \leq 128$. Συγκεκριμένα, η μεγαλύτερη βελτίωση παρατηρείται για $n = 4096$, με τους kernel1 και kernel2 να έχουν την μεγαλύτερη βελτίωση (~2 seconds) σε σχέση με την βελτίωση που έχει ο kernel3 (~0.5 seconds), στο συγκεκριμένο n.

Γενικά, η βελτίωση στο χρόνο εκτέλεσης των αλγόριθμων χρησιμοποιώντας τον A^T για *coalesced memory access*, σε σχέση με την υλοποίηση της Εργασίας 3, ήταν σχετικά μικρή. Βέβαια, ίσως να είχε περισσότερο νόημα σε αρκετά μεγαλύτερα προβλήματα.

C.2. Υλοποιήσεις $kernel1, 2, 3, \_t\_v2$

Επειδή η βελτίωση στις υλοποιήσεις $kernel1,2,3\_t$ δεν ήταν ιδιαίτερα μεγάλη, έγινε προσπάθεια για μια καινούρια προσέγγιση και ακολουθήθηκε μια εντελώς διαφορετική στρατηγική στον διαμοιρασμό των στοιχείων του πίνακα A στα διάφορα blocks και threads. Πλέον, το κάθε block περιλαμβάνει σταθερό αριθμό από threads, συγκεκραμένα n στοιχεια, τα οποια εκφράζουν τον δείκτη i των στοιχειων του πινακα A. Επομένως, κατά αυτόν τον τρόπο, στο κάθε block τα στοιχεία μεταβάλλονται μόνο κατά τον δείκτη i και όχι κατά τον j. Αυτό, στην περίπτωση χρήσης της shared memory, μας επιτρέπει να θέσουμε μια φορά το στοιχείο $(j * n + k)$ στην shared memory και να το χρησιμοποιησουμε σε ολα τα στοιχεια του block για την επίτευξη του αλγόριθμου. Ο δείκτης πλέον που δείχνει τη μεταβολή του j ειναι η y συνιστωσα του δεικτη των blocks, δηλαδη το $blockIdx.y$, άρα ο δεικτης αυτος θα εχει παντοτε n στοιχεια. Για την εύρυθμη λειτουργία του αλγόριθμου, με σκοπό τη σύγκριση όλων των στοιχείων μεταξύ τους η x συνιστωσα του αριθμου των $blocks(blockIdx.x)$ ειναι ιση με $n + threadsPerBlock)/threadsPerBlock$, οπου n ο αριθμος των στοιχειων του πινακα A και $threadsPerBlock$ ο αριθμος των threads σε καθε block.

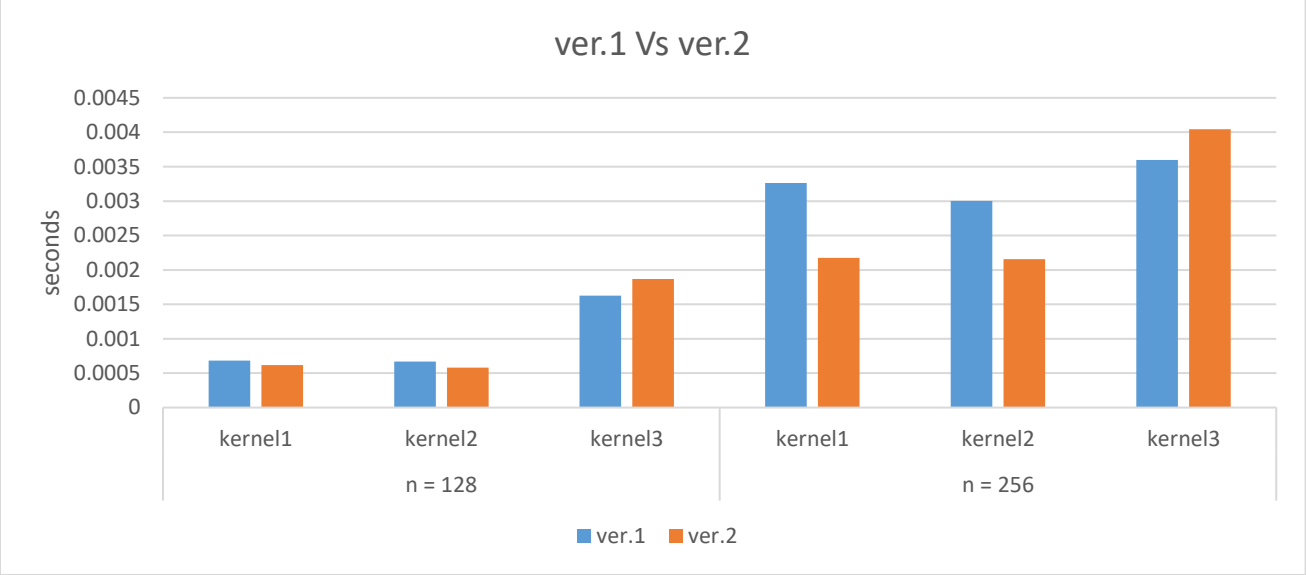
Μετά από δοκιμές βρέθηκε πως ο βέλτιστος αριθμός threads για κάθε block στο συγκεκριμένο πρόβλημα είναι 128. Αυτό ισχύει στην περίπτωση του 1ου kernel.

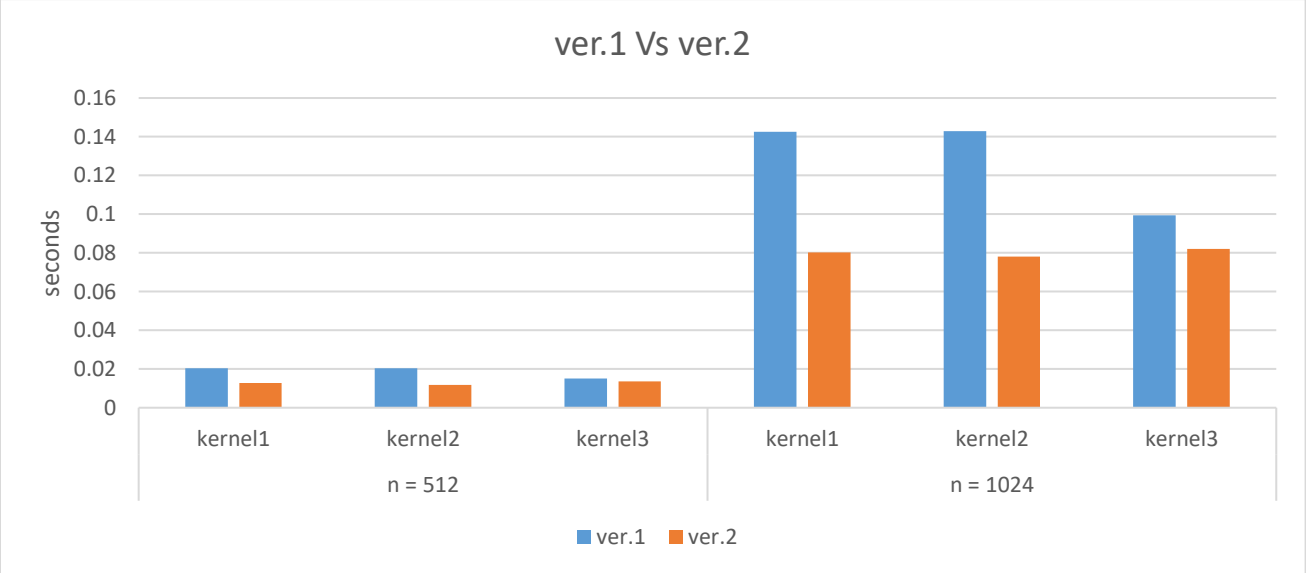
Στον 2ο kernel ο αριθμός threads για κάθε block είναι 256.

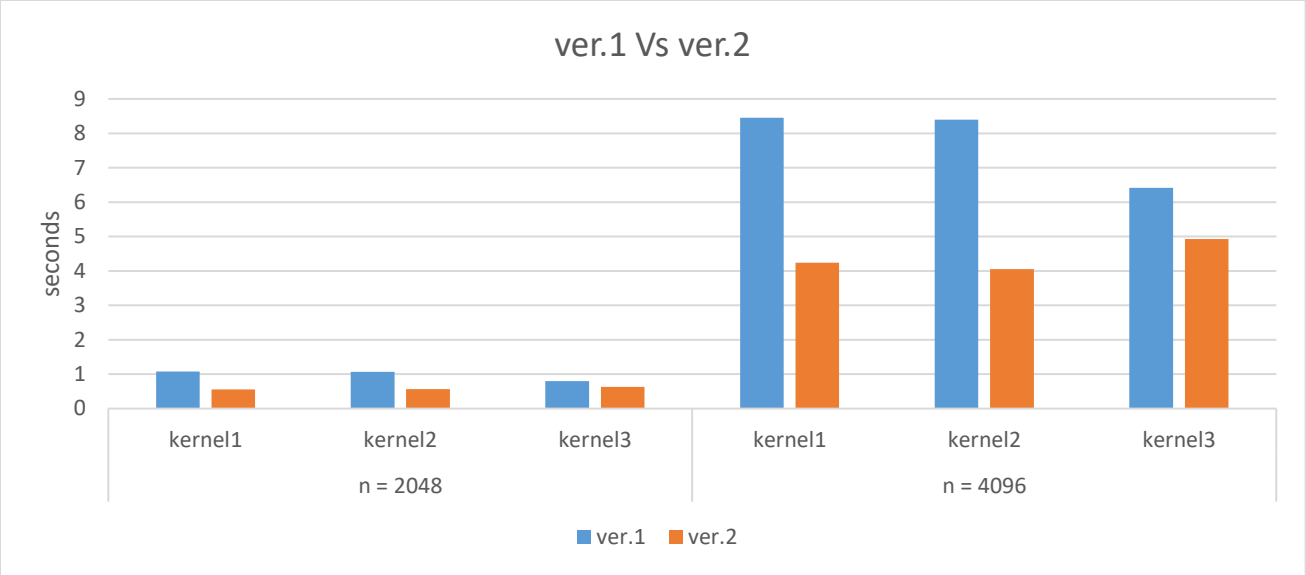
Στον 3ο kernel, η συνιστώσα $blockIdx.x$ είναι ίση με $n + threadsPerBlock)/(threadsPerBlock * cellsPerThread)$, καθώς πλέον το κάθε thread αναλαμβάνει περισσότερα του ενός στοιχεία για σύγκριση (συγκεκριμένα 4 σε κάθε thread). Επιπλέον, ο αριθμός των threads εξαρτάται από τον αριθμό των στοιχείων του κάθε thread και δεν είναι σταθερός όπως στην περίπτωση του 2ου kernel.

C.2. Αποτελέσματα – Παρατηρήσεις

Παρακάτω παρουσιάζονται κάποια διαγράμματα που αφορούν τους χρόνους εκτέλεσης των $\boldsymbol{kernel1,2,3\_t\_v2}$ σε σχέση με τους $\boldsymbol{kernel1,2,3\_t}$. Επειδή οι $kernel1,2,3\_t\_v2$ ήταν γρηγορότεροι σε όλες σχεδόν τις περιπτώσεις από τους $kernel1,2,3\_t$, η σύγκριση τους με τους kernels της 3ης εργασίας θεωρήθηκε περιττή.







Είναι φανερό ότι οι υλοποιήσεις$kernel1,2,3,_t_v2$ υπερτερούν έναντι των υλοποιήσεων $kernel1,2,3,\_t$. Στην περίπτωση όμως του $kernel3,\_t\_v2$, και μετά από πάρα πολλές δοκιμές και αλλαγές του κώδικα, παρατηρούμε ότι είναι πιο αργός από τους $kernel1,\_t\_v2$ και $kernel2,\_t\_v2$, πράγμα το οποίο δεν συνέβαινε στις προηγούμενες υλοποιήσεις. Παρόλα αυτά, ο $kernel3\_t\_v2$ εξακολουθεί να είναι γρηγορότερος από όλους τους $kernel1,2,3,\_t$ και $kernel1,2,3$.

Εν κατακλείδι…

Στην Εργασία αυτή, παρατηρούμε πως η διευκόλυνση στην πρόσβαση στην μνήμη με την τεχνική *coalesced memory access* χωρίς καμία άλλη αλλαγή της αρχικής δομής του αλγόριθμου, επιφέρει μικρές βελτιώσεις. Από την άλλη, παρατηρούμε πως ο συνδυασμός της τεχνικής *coalesced memory access* σε συνδυασμό με μια πιο προσεγμένη και μελετημένη υλοποίηση του αλγόριθμου, μπορούμε να πετύχουμε μεγαλύτερες και πιο ικανοποιητικές βελτιώσεις που αγγίζουν το 50% της αρχικής υλοποίησης CUDA και μεγαλύτερες του 100% της αρχικής σειριακής υλοποίησης.

Επίσης, σχετικά με το υπολογιστικό μηχάνημα $diades$, παρατηρήθηκε μεγάλη αστάθεια ως προς την απόδοση του, εφόσον με τον ίδιο εκτελέσιμο κώδικα εμφάνιζε διαφορετικούς χρόνους εκτέλεσης (με διαφορές μεγαλύτερες των 100 sec στον σειριακό αλγόριθμο και μερικών seconds στις υπόλοιπες υλοποιήσεις) σε διαφορετικές χρονικές στιγμές της ημέρας. Οπότε, κάποιες αποκλίσεις των χρόνων εκτέλεσης είναι πιθανόν να υπάρχουν.

Βιβλιογραφία – Χρήσιμα Links

http://www.cc.gatech.edu/~hyesoon/spr10/lec_cuda5.pdf

http://stackoverflow.com/questions/5041328/cuda-coalesced-memory/5044424#5044424

http://stackoverflow.com/questions/5041328/cuda-coalesced-memory/21789126#21789126

https://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/