ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Παράλληλα και Κατανεμημένα Συστήματα Υπολογιστών

Άσκηση 3

ΧΑΤΖΗΘΩΜΑ ΑΝΤΡΕΑΣ

ΑΕΜ: 8026

antreasc@ece.auth.gr

Υπεύθυνος καθηγητής: κ. Πιτσιάνης Νίκος
Ημερομηνία Παράδοσης: 29/02/2016

ΕΙΣΑΓΩΓΗ

Το ζητούμενο της εργασίας αυτής ήταν η υλοποίηση ενός παράλληλου προγράμματος σε περιβάλλον CUDA για την επίλυση του προβλήματος $All\ Pair\ Shortest\ Path\ (APSP)$. Όπως αναφέρθηκε και στην εκφώνηση της άσκησης, το πρόβλημα αναφέρεται στην εύρεση της μικρότερης απόστασης ανάμεσα σε δύο κόμβους v_i και v_j ενός κατευθυνόμενου γράφου $G(V, E)$.

Για την επίλυση του προβλήματος, αναπτύχθηκε πρώτα ένας σειριακός αλγόριθμος βασισμένος στον αλγόριθμο Warshall - Floyd.

Για την δημιουργία του γράφου, αναπτύχθηκε μια συνάρτηση βασισμένη στην δοθείσα ρουτίνα MATLAB $makeAdjacency(n, p, w)$.

Ακολούθως, αναπτύχθηκαν οι 3 διαφορετικές μέθοδοι υλοποίησης του αλγόριθμου Warshall - Floyd, με τη χρήση της CUDA. Πρακτικά, κάθε μέθοδος είναι και μια διαφορετική συνάρτηση πυρήνα (kernel).

Για τον έλεγχο ορθότητας των αποτελεσμάτων των υλοποιήσεων CUDA, έγινε σύγκριση των αποτελεσμάτων αυτών με τα αποτελέσματα του σειριακού αλγόριθμου Warshall – Floyd.

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

Το compile για την σειριακή υλοποίηση έγινε με την εντολή:

$$nvcc\ apsp.cu\ -O3 -o\ apsp$$

και run με την εντολή:

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

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

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

$w = 2\ (τυχαιο)$.

Αλγόριθμος Warshall – Floyd

Η υλοποίηση του αλγόριθμου Warshall – Floyd βασίστηκε στον ίδιο τον αλγόριθμο αυτόν καθ' αυτόν. Συμβουλευτικά sites υπήρξαν κατά κύριο λόγο τα 2 παρακάτω:

- https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
- http://www.programming-algorithms.net/article/45708/Floyd-Warshall-algorithm

Επίσης, η λύση του προβλήματος *APSP* βρίσκοντας μόνο την απόσταση και όχι το μονοπάτι, καθιστά την υλοποίηση του αλγόριθμου λιγότερο πολύπλοκη.

Για την υλοποίηση του αλγόριθμου, πρώτα δημιουργείται ο γράφος καλώντας την συνάρτηση $\boldsymbol{makeAdjacency}()$**.** Ο πίνακας που αντιπροσωπεύει τον γράφο αποθηκεύεται στον πίνακα $\boldsymbol{A}$**.**

Ακολούθως, αντιγράφεται ο πίνακας $\boldsymbol{A}$ στον πίνακα $\boldsymbol{D}$. Αυτό γίνεται διότι ο $\boldsymbol{D}$ θα χρησιμοποιηθεί για την επίλυση του προβλήματος με τον αλγόριθμο Warshall - Floyd ούτως ώστε τα στοιχεία του πίνακα $\boldsymbol{A}$ να μην υποστούν αλλοίωση με αποτέλεσμα να μπορούν να ξανά-χρησιμοποιηθούν στις υπόλοιπες υλοποιήσεις με τη χρήση CUDA.

Αφού γίνει και η αντιγραφή του πίνακα, τότε μέσω της συνάρτησης $\boldsymbol{floydWarshallMethod}()$ υλοποιείται ο αλγόριθμος Warshall – Floyd και τα αποτελέσματα αποθηκεύονται στον πίνακα ***D.*** Στην κονσόλα εμφανίζεται ο χρόνος που χρειάστηκε ο αλγόριθμος για την επίλυση του προβλήματος.

Έλεγχος Ορθότητας

Για τον έλεγχο ορθότητας των αποτελεσμάτων, υλοποιήθηκε η συνάρτηση $\boldsymbol{test}(\boldsymbol{float} * \boldsymbol{A})$ η οποία 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 $\boldsymbol{printArray}(\boldsymbol{float} * \boldsymbol{array})$ η οποία εκτυπώνει τον εκάστοτε πίνακα (μόνο για πίνακες $n\,x\,n \leq 3$) που δέχεται σαν όρισμα. Με αυτό τον τρόπο επιτρέπει τον χειροκίνητο έλεγχο ορθότητας για μικρό αριθμό στοιχείων.

Επίλυση APSP με χρήση CUDA

Για την υλοποίηση των 3ων μεθόδων (συναρτήσεις πυρήνα) που δίνονται στην εκφώνηση της άσκησης, υλοποιήθηκε η συνάρτηση $\boldsymbol{methods(int\ f)}$**.** Το $\boldsymbol{f}$ παίρνει τις τιμές 1, 2 και 3 οι οποίες τιμές αντιπροσωπεύουν τις μεθόδους 1, 2 και 3 αντίστοιχα.

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

Οι προετοιμασίες αυτές, έχουν να κάνουν με:

Την δέσμευση μνήμης στην GPU χρησιμοποιώντας την έτοιμη συνάρτηση $\boldsymbol{cudaMalloc}$**.** Συγκεκριμένα γίνεται δέσμευση ενός $n * n$ μονοδιάστατου πίνακα με το όνομα $\boldsymbol{A\_dev}$ (γίνεται μια φορά για όλες τις κλήσεις kernel). Μετά, δεσμεύονται στην CPU οι πίνακες $\boldsymbol{A\_result1}$, $\boldsymbol{A\_result2}$ ή $\boldsymbol{A\_result3}$, ανάλογα με ποια από τις 3 μεθόδους υλοποιείται. Στους πίνακες αυτούς θα αποθηκευτούν τα αποτελέσματα των αλγόριθμων ούτως ώστε να γίνει αργότερα ο έλεγχος ορθότητας.

Ακολούθως, γίνεται η έναρξη του μετρητή του χρόνου εκτέλεσης με την συνάρτηση $\boldsymbol{cudaEventRecord(start)}$. Ο μετρητής σταματάει δίνοντας το όρισμα $stop$.

Έπειτα, γίνεται αντιγραφή του πίνακα $\boldsymbol{A}$ (ο οποίος δημιουργήθηκε μέσω της $\boldsymbol{makeAdjacency}()$) από την CPU στην GPU (πίνακας $\boldsymbol{A\_dev}$) χρησιμοποιώντας την συνάρτηση $\boldsymbol{cudaMemcpy}$**.** Η αντιγραφή γίνεται από όλες τις μεθόδους.

Αμέσως πριν από την κλήση των συναρτήσεων kernel, γίνεται έλεγχος κατά πόσο ο μέγιστος αριθμός νημάτων ανά block που έχει οριστεί, επαρκεί για την επιθυμητή κλήση kernel, αλλιώς χρησιμοποιούνται περισσότερα blocks και έπειτα δεσμεύονται οι κατάλληλες μεταβλητές τύπου $dim3$.

Και τέλος, αντιγράφονται τα δεδομένα από τον πίνακα $\boldsymbol{A\_dev}$ της GPU στον πίνακα $\boldsymbol{A\_results1, 2, 3}$ της CPU αντίστοιχα με την κάθε μέθοδο και γίνεται διακοπή του μετρητή του χρόνου εκτέλεσης.

Σημειώνεται ότι:

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

- Η επιλογή για επίλυση με χρήση μονοδιάστατου πίνακα έγινε διότι στα περισσότερα παραδείγματα στο document της NVIDIA αλλά και στο internet γενικότερα επικρατεί η χρήση μονοδιάστατου πίνακα αντί για δισδιάστατου για λόγους ευκολίας δεικτοδότησης.

Βάσει εκφώνησης, οι συναρτήσεις πυρήνα που υλοποίησαν είναι οι ακόλουθες:

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

ανάθεση ενός κελιού ανά νήμα, χωρίς τη χρήση shared memory.

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

  ανάθεση ενός κελιού ανά νήμα, με τη χρήση shared memory.

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

  ανάθεση πολλαπλών κελιών ανά νήμα, με τη χρήση shared memory.

1η μέθοδος - kernel1

Στην πρώτη συνάρτηση πυρήνα που υλοποιήθηκε, δεν χρησιμοποιήθηκε καθόλου η shared memory της GPU. Επίσης για να γίνει ανάθεση ενός κελιού ανά νήμα, χρειάστηκαν τόσα νήματα όσα και τα στοιχεία του πίνακα, δηλαδή $n * n$ νήματα.

Κάθε block όμως, έχει ένα μέγιστο αριθμό νημάτων βάσει hardware. Στη συγκεκριμένη GPU ο αριθμός αυτός ανέρχεται στα 1024 νήματα ανά block. Για τον λόγο αυτό, κάθε φορά γίνεται έλεγχος αν ο αριθμός των στοιχείων του πίνακα ξεπερνά τον μέγιστο αριθμό νημάτων ανά block (Στο πρόγραμμα ορίζεται ως $\boldsymbol{maxThreadsPerBlock}$). Αν συμβαίνει αυτό, τότε ορίζονται περισσότερα blocks ώστε όλα τα κελιά να αντιστοιχούν σε ένα νήμα το κάθε ένα. Επίσης αξίζει να αναφερθεί ότι το $\boldsymbol{maxThreadsPerBlock}$, δεν ορίστηκε ίσο με 1024 όπως θα ήταν αναμενόμενο. Μετά από δοκιμές αλλά και συζητήσεις με συνάδελφους, αποφάνθηκε ότι ο αλγόριθμος τρέχει καλύτερα (=γρηγορότερα) για μικρότερο αριθμό νημάτων ανά block. Συγκριμένα ο μέγιστος αριθμός νημάτων ανά block ορίστηκε ως $\boldsymbol{maxThreadsPerBlock} = 64$ (ή $8 * 8$).

Αφού οριστούν οι μεταβλητές τύπου $dim3$, καλείται n φορές (για κάθε ενδιάμεσο κελί/στοιχείο) η συνάρτηση $\boldsymbol{kernel1}$.

Η δεικτοδότηση εντός της συνάρτησης kernel έγινε βάσει τις διαστάσεις του block, αλλά και με τη βοήθεια του παρακάτω pdf :
http://cs.calvin.edu/courses/cs/374/CUDA/CUDA-Thread-Indexing-Cheatsheet.pdf

Αφού γίνουν οι υπολογισμοί, αντιγράφονται τα δεδομένα από την GPU στον πίνακα $\boldsymbol{A\_results1}$ της CPU, όπως ακριβώς αναφέρθηκε και παραπάνω (προηγούμενη σελίδα).

Τέλος, γίνεται επαλήθευση των αποτελεσμάτων συγκρίνοντας τα με τα αποτελέσματα από τον σειριακό αλγόριθμο Warshall - Floyd (πίνακας $\boldsymbol{D}$).

2η μέθοδος – kernel2

Για την δεύτερη μέθοδο, η υλοποίηση είναι παρόμοια με την πρώτη μέθοδο με τη μόνη διαφορά ότι στη συνάρτηση kernel χρησιμοποιείται και η shared memory.

Η φιλοσοφία πίσω από την shared memory είναι ότι η πρόσβαση σε αυτή γίνεται πιο γρήγορα από ότι στην global memory διότι κάθε block φέρει την δικιά του shared memory. Για τον λόγο αυτό, η shared memory ενός block είναι προσβάσιμη από τα νήματα του ιδίου του block και όχι από τα νήματα άλλων blocks.

Για την υλοποίηση της $\boldsymbol{kernel2}$ χρησιμοποιήθηκε static shared memory η οποία δηλώθηκε εντός της συνάρτησης kernel.

Στην αρχή, η χρήση της shared memory έγινε για ένα μόνο κελί σε κάθε κλήση της $kernel2$ και η βελτίωση στο χρόνο εκτέλεσης ήταν αρκετά ικανοποιητική. Ακολούθως έγινε χρήση της shared memory για 2 συνολικά κελιά και τέλος για 3. Παρατηρήθηκε λοιπόν, ότι καλύτεροι χρόνοι επιτεύχθηκαν με την χρήση της shared memory για ένα μόνο κελί και όχι για 2 ή 3 όπως κανείς θα περίμενε.

Έτσι η προσπέλαση στη shared memory για ένα κελί κατά τη διαδικασία ελέγχου if αλλά και κατά τη διαδικασία ανάθεσης τιμής, δίνει ικανοποιητικά αποτελέσματα ως προς το χρόνο εκτέλεσης του αλγορίθμου. Τα αποτελέσματα αυτά θα συγκριθούν αργότερα με τις υπόλοιπες μεθόδους.

Αφού γίνουν οι υπολογισμοί, τα επόμενα βήματα είναι ακριβώς ίδια με την 1η μέθοδο.

3η μέθοδος – kernel3

Όπως και στη δεύτερη μέθοδο, έτσι και στη τρίτη χρησιμοποιήθηκε η shared memory. Η διαφορά είναι ότι σε κάθε νήμα ανατίθενται περισσότερα από ένα κελιά του πίνακα. Ο αριθμός των κελιών ορίζεται με την μεταβλητή $\boldsymbol{cellsPerTherads}$. Μετά από δοκιμές αλλά και συζητήσεις με συνάδελφους, ο αλγόριθμος είχε καλύτερες αποδόσεις για αριθμό κελιών ανά νήμα ίσο με 4.

Η υλοποίηση της συνάρτησης πυρήνα $\boldsymbol{kernel3}$ αλλά και οι προετοιμασίες που γίνονται πριν από την κλήση της, είναι παρόμοιες με την υλοποίηση και τις προετοιμασίες της 2ης μεθόδου, απλά τώρα πρέπει να ληφθεί υπόψη και ο αριθμός των κελιών ανά νήμα. Έτσι σε κάθε κλήση της *kernel3* δεσμεύεται shared memory ίση με *cellsPerThreads* και άρα γίνεται ανάθεση *cellsPerThreads* κελιών σε κάθε νήμα.

Αποτελέσματα – Παρατηρήσεις – Συμπεράσματα

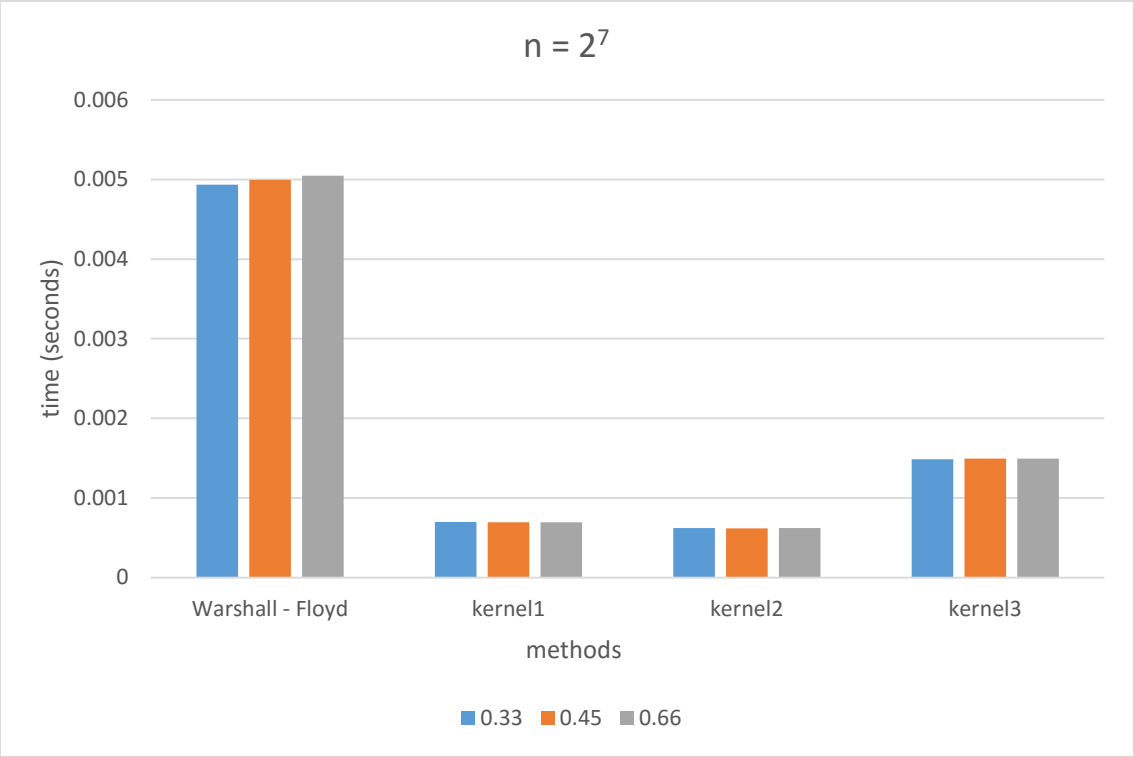
Από τις μετρήσεις που έγιναν στο υπολογιστικό σύστημα $Diades$, η βελτίωση στο χρόνο εκτέλεσης των αλγόριθμων με τη χρήση του παράλληλου προγραμματισμού

σε περιβάλλον CUDA, έναντι της σειριακής υλοποίησης του αλγόριθμου Warshall - Floyd, είχε γίνει ιδιαίτερα αισθητή.

Οι τιμές των παραμέτρων δεν ξέφυγαν από τις τιμές που δόθηκαν στις οδηγίες της άσκησης.

Όλοι οι χρόνοι εκτέλεσης των αλγορίθμων, βρίσκονται στο αρχείο $times.xlsx$

Παρακάτω παρατίθενται τα διαγράμματα που δημιουργήθηκαν με το πέρας των αλγορίθμων, για όλους τους πιθανούς συνδυασμούς τιμών.

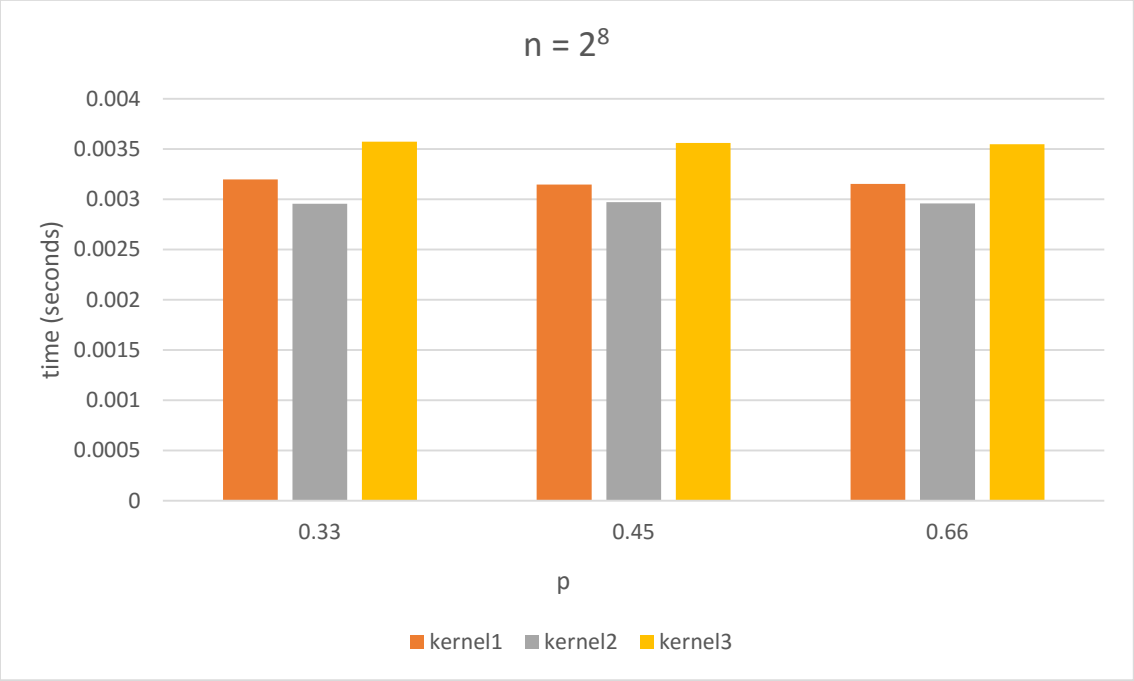


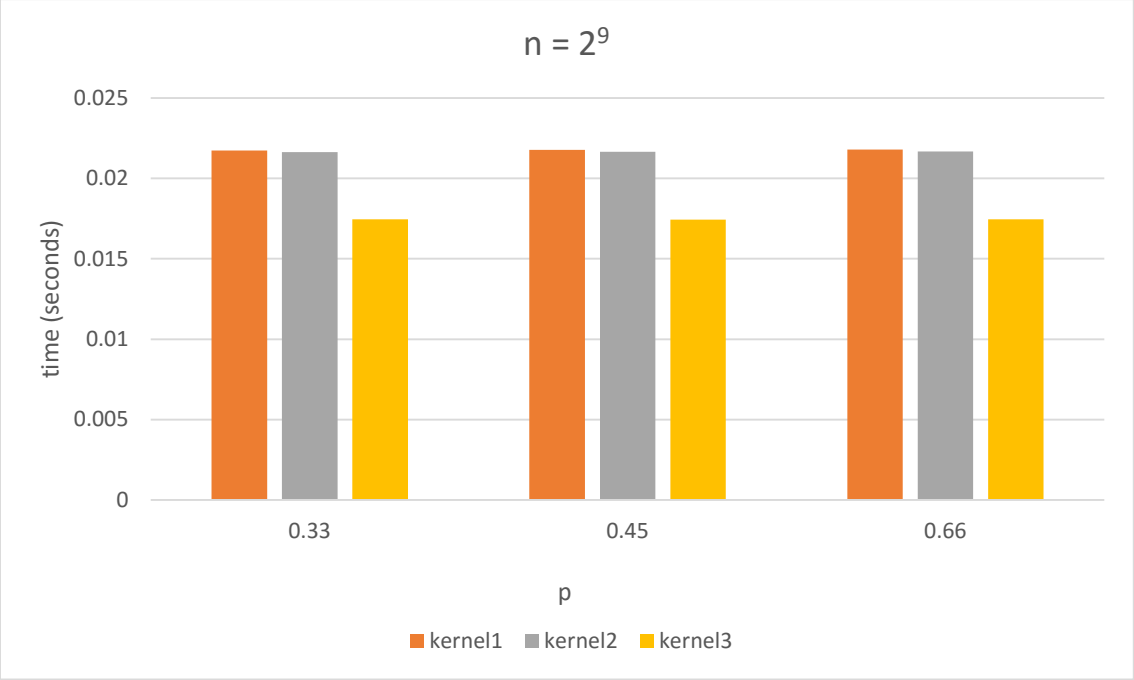
Η βελτίωση της απόδοσης του αλγορίθμου είναι εμφανή. Την μεγαλύτερη απόδοση έχει η συνάρτηση $kernel2$ με την $kernel1$ να ακολουθεί με πολύ μικρή διαφορά και τέλος η $kernel3$ να έχει σχεδόν τον διπλάσιο τους χρόνο εκτέλεσης από τους $kernel1$ και $kernel2$. Επίσης, για τις τιμές του $p = [0.33\ 0.45\ 0.66]$ δεν παρατηρείται ιδιαίτερη διαφορά στους χρόνους εκτέλεσης.

Η βελτίωση που υπέστη με τη χρήση των $kernel1$ και $kernel2$ ήταν περίπου 7πλάσια σε σχέση με την αρχική υλοποίηση του αλγόριθμου Warshall – Floyd.

Παρακάτω παρατίθενται τα διαγράμματα για $n = 2^{[8:9]}$.

Σημείωση: Επειδή όπως παρατηρήθηκε οι αλλαγές στις τιμές του $p = [0.33\ 0.45\ 0.66]$ δεν έχουν μεγάλη επιρροή στους χρόνους εκτέλεσης, τα παρακάτω διαγράμματα θα παρουσιαστούν λίγο αλλιώς ούτως ώστε να δίνεται περισσότερη έμφαση στους χρόνους εκτέλεσης αλλάζοντας από την μια μέθοδο στην άλλη και όχι στους χρόνους εκτέλεσης αλλάζοντας τις τιμές p. Επίσης οι χρόνοι εκτέλεσης του σειριακού αλγόριθμου Warshall – Floyd παραλείπονται ούτως ώστε να γίνεται πιο εύκολα αντιληπτή η διαφορά μεταξύ των 3 μεθόδων CUDA.

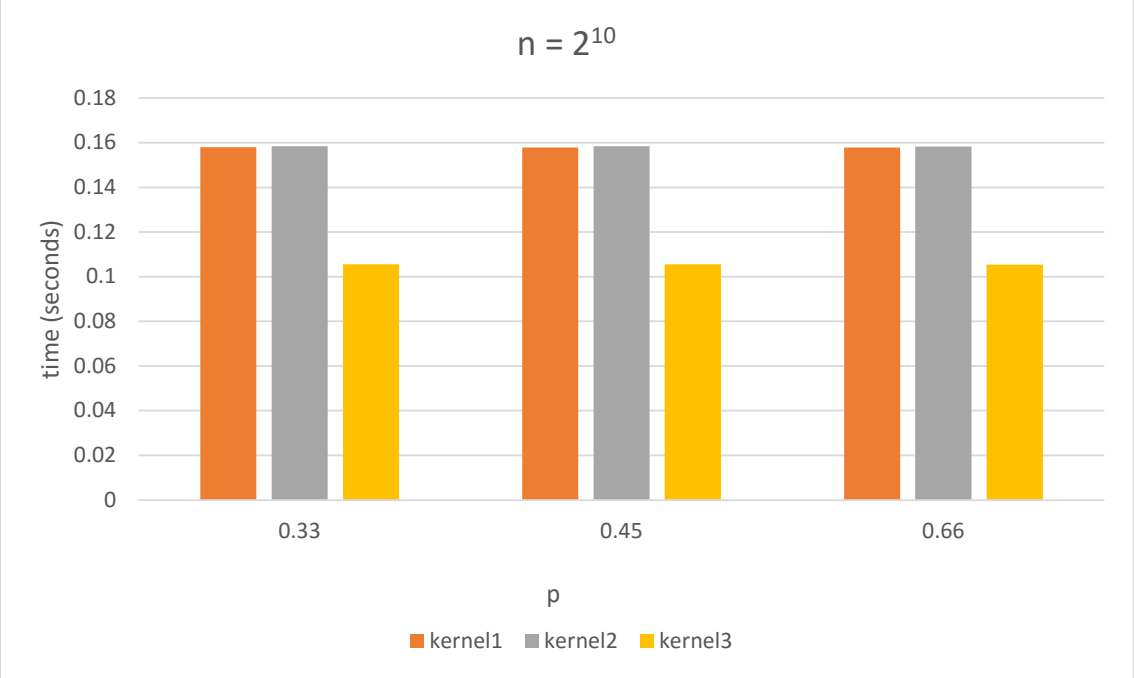


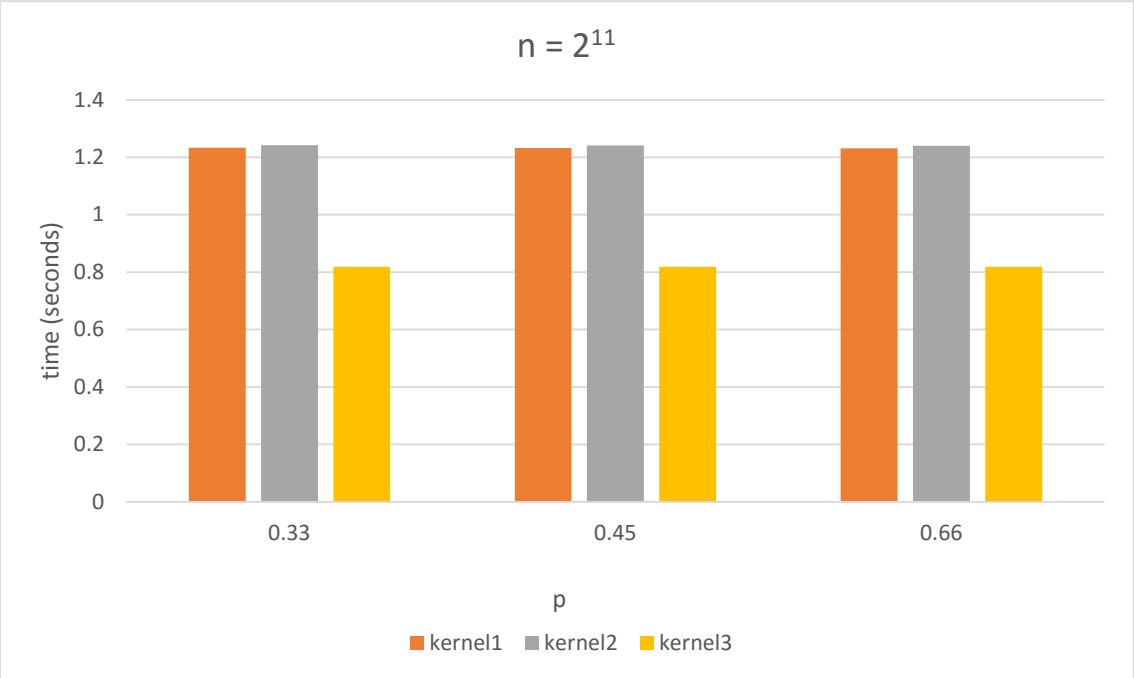


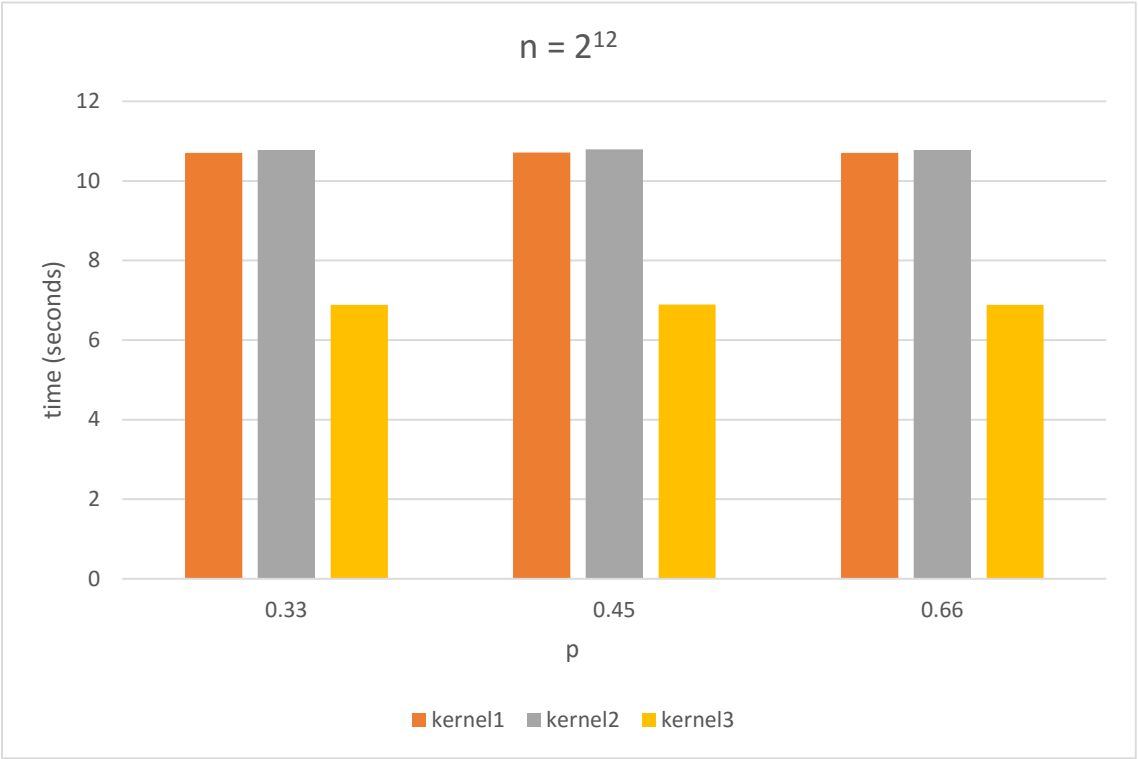
Όπως φαίνεται από τα παραπάνω διαγράμματα, η 3η μέθοδος με την $kernel3$ αρχίζει να υπερτερεί των υπολοίπων για $n = 2^9$. Επίσης η 2η μέθοδος δείχνει να

έχει καλύτερους χρόνους από την 1η μέθοδο για $n = 2^8$, παρόλα αυτά για $n = 2^9$ οι χρόνοι είναι περίπου ίδιοι.

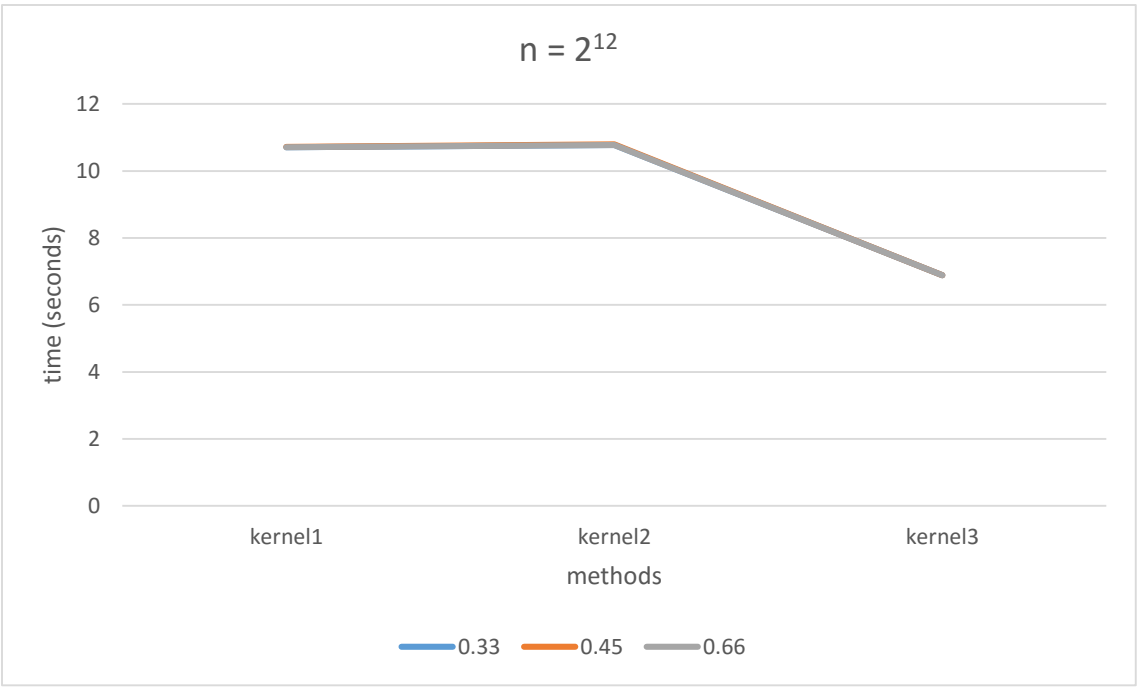
Παρακάτω παρατίθενται τα διαγράμματα για $n = 2^{[10:12]}$.







Όπως εύκολα διαπιστώνει κανείς, όσο μεγαλώνει το p, ο χρόνος εκτέλεσης της 3ης μεθόδου βελτιώνεται περισσότερο και η διαφορά μεταξύ της 3ης μεθόδου από την 1η και τη 2η αυξάνεται ελάχιστα. Η διαφορά αυτή γίνεται εύκολα αντιληπτή και από το παρακάτω διάγραμμα για $n = 2^{12}$.



Συμπεράσματα:

Με το πέρας εκτέλεσης των αλγορίθμων, διαπιστώνεται η μεγάλη βελτίωση που επιφέρει η υλοποίηση του αλγόριθμου Warshall – Floyd με την χρήση της GPU χρησιμοποιώντας το API της CUDA. Η βελτίωση αυτή, αυξάνεται όσο αυξάνεται και ο αριθμός των στοιχείων του γράφου, πράγμα το οποίο παρατηρήθηκε και στις προηγούμενες ασκήσεις. Δηλαδή, όσο μεγαλώνει το μέγεθος του προβλήματος, τόσο μεγαλύτερη είναι και η βελτίωση επίλυσης του προβλήματος με εργαλεία παράλληλου προγραμματισμού.

Η χρήση της shared memory για ένα κελί ανά νήμα ($kernel2$) δεν επιφέρει μεγάλη βελτίωση συγκριτικά με την 1η μέθοδο ($kernel1$) στην οποία δεν χρησιμοποιήθηκε η shared memory. Μάλιστα η μόνη περίπτωση που θα έλεγε κανείς ότι η $kernel2$ έχει εμφανή καλύτερη απόδοση από την $kernel1$ είναι η περίπτωση όπου $n = 2^8$. Για όλα τα υπόλοιπα n, οι 2 μέθοδοι έχουν περίπου την ίδια απόδοση με τους χρόνους εκτέλεσης τους να διαφέρουν ελάχιστα.

Η χρήση της shared memory για περισσότερα από ένα κελιά ανά νήμα επιφέρει βελτίωση για όλα τα $n = 2^{[7:12]}$, παρόλα αυτά όπως φαίνεται και από τα διαγράμματα η 3η μέθοδος έχει νόημα για $n \geq 2^9$ καθώς για μικρότερα n, η 1η και η 2η μέθοδος έχουν καλύτερους χρόνους εκτέλεσης.

Συγκεκριμένα για $n = 2^{12}$ με την 1η και 2η μέθοδο επιτυγχάνεται περίπου 13 φορές βελτίωση συγκριτικά με τον αρχικό Warshall – Floyd αλγόριθμο, ενώ με την 3η μέθοδο η βελτίωση αυτή είναι περίπου 20πλάσια. Τα νούμερα αυτά προκύπτουν από τους χρόνους εκτέλεσης των μεθόδων που βρίσκονται στο αρχείο $times.xlsx$.