HPC - 2D Diffusion with Stencils

Ant Colony and Pherormone Dispersion

Η Εργασία εκπονήθηκε από τους:

- Σφήκας Θεόδωρος, 1072550
- Τσούλος Βασίλειος, 1072605

❖ Εισαγωγή:

System Specifications:

Το πρόγραμμα για την παράδοση των τελικών αποτελεσμάτων εκτελέστηκε μέσω remote connection στο υπολογιστικό σύστημα που προσφέρεται στα πλαίσια του μαθήματος. Συνοπτικά ο κώδικας εκτελέστηκε σε επεξεργαστή Intel Core i7-3770 και σε σύστημα με *16GB RAM*. Ακολουθούν περαιτέρω διευκρινίσεις πάνω στα χαρακτηριστικά του επεξεργαστή και κάρτας γραφικών:

» Intel Core i7-3770 Desktop Processor :
- # of CPU Cores → 4
- # of Threads → 8
- Max. Boost Clock→ Up to 3.9GHz
- L1 Cache → 128 KB *(per core)*
- L2 Cache → 1 MB *(per core)*
- L3 Cache → 8 MB *(shared)*

» Nvidia GPGPU Tesla K40c :
- # of Cuda Cores → 2880
- # of SMXs → 15
- Cuda Compute Capability → 3.5
- L1 Cache → 128 KB *(per SMX)*
- L2 Cache → 1536 KB *(shared)*
- *DRAM* → 12GB

Οδηγίες εκτέλεσης:

Στον φάκελο που παραδώθηκε, εμπεριέχονται δύο υπό-φάκελοι για τα parts της εργαστηριακής άσκησης. Σε κάθε ένα από αυτά εμπεριέχεται ο source code, makefile for compilation τόσο του σειριακού όσο και του παράλληλου κώδικα.

Παραδοχές:

1) Με στόχο να συγκρίνουμε το Speedup του παράλληλου κώδικα και στις δύο περιπτώσεις (Part A and B) δεν έχει πραγματοποιηθεί εκτεταμένη προσπάθεια ώστε να παραλληλοποιηθεί ο σειριακός κώδικας σε επίπεδο πολυπυρηνικότητας στο επεξεργαστή (CPU). Περιοριστήκαμε σε μονοπύρηνη εκτέλεση στους υπολογιστικούς πυρήνες των ασκήσεων. Στο μέρος B της εργαστηριακής άσκησης αποκλειστικά οι αρχικοποίηση των μητρώων έχει πραγματοποιηθεί παράλληλα καθώς αυτή δεν προσμετράται και στους χρόνους εκτέλεσης των υπολογιστικών πυρήνων.

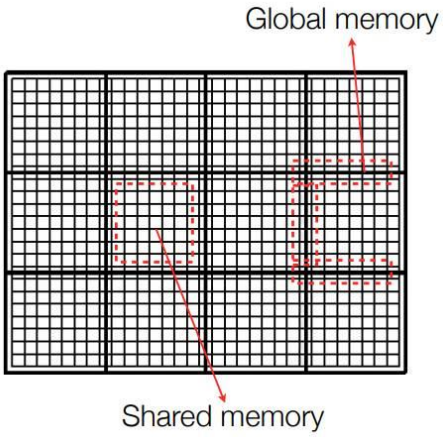
2) Με στόχο της εκπόνηση του δεύτερου μέρους της εργαστηριακής άσκησης καθώς δεν δίνονται ούτε ακριβείς οδηγίες, ούτε ένα παράδειγμα για τον σειριακό κώδικα, οδηγηθήκαμε σε μία πληθώρα σχεδιαστικών αποφάσεων κρατώντας την ουσία των ζητουμένων αναλλοίωτη. Περισσότερες λεπτομέρειες θα αναφερθούν στην συνέχεια.

❖ Part A

Αρχικά κρίνουμε σημαντικό να αναφερθούμε συνοπτικά στην φύση του προβλήματος και έπειτα στις σχεδιαστικές επιλογές που εκτελέστηκαν με στόχο την παραλληλοποίηση του κώδικα σε GPU. Το πρόβλημα του 2D Diffusion, η προσομοίωση μίας διαφορικής εξίσωσης με Dirichlet boundaries σε κεντρισμένες διαφορές στον χώρο και τον σχηματισμό της forward Euler στον χρόνο είχε πραγματοποιηθεί και στην πρώτη εργαστηριακή άσκηση. Δημιουργούμε ένα N*N μητρώο του οποίου κάθε cell μεταβάλλεται με τον χρόνο (steps της διαδικασίας) με την χρήση των τεσσάρων γειτονικών κελιών του. Το initialization του μητρώου έχει πραγματοποιηθεί στην CPU καθώς τα to do steps δεν ζητούσαν την παραλληλοποίηση του, ωστόσο αυτή δεν προσμετράται όπως προαναφέραμε στις χρονικές μας μετρήσεις.

Με στόχο την παραλληλοποίηση σε κάρτες γραφικών πρέπει να σημειώσουμε πως υπάρχει ένα trade off μεταξύ της ταχύτητας των ρολογιών της κάρτας, τα οποία είναι πολύ μικρότερα από αυτά της CPU, προσφέρει ωστόσο την δυνατότητα μαζικού παραλληλισμού. Με στόχο την εκμετάλλευση του, δημιουργούνται ορισμένοι προβληματισμοί. Η οδήγηση ενός εκ φύσεως sequential κώδικα, κώδικα που απαιτεί δηλαδή μεγάλο αριθμό από dependant πληροφορίες, εξαρτήσεις και execution branches (if statements , jumps etc) οδηγεί σε Warp divergence και πιθανά σε χειρότερους χρόνους εκτέλεσης. Με στόχο την αποφυγή λοιπών των data dependencies μεταξύ του μητρώου και των advanced τιμών του, δημιουργούμε 2 μητρώα N*N τα οποία γίνονται swapped σε κάθε κάλεσμα της propagate (advance to the next time step). Στο πρώτο αποθηκεύονται οι τιμές των propagated cells ενώ στο δεύτερο διατηρούνται οι προηγούμενες τιμές αποφεύγοντας το data dependency.
Ένα δεύτερο πρόβλημα που συχνά παρουσιάζεται είναι η απαίτηση συχνής μεταφοράς δεδομένων μεταξύ της κάρτας γραφικών και της primary memory. Στην περίπτωση του κώδικα μας, με στόχο της βέλτιση δυνατή μεταφορά δεδομένων δεν χρησιμοποιείται κανένας συγχρονισμός και μεταφορά μεταξύ των διαδοχικών εκτελέσεων της Propagate Density. Με στόχο σε κάθε βήμα να λαμβάνουμε τα διαγνωστικά μηνύματα μέσω της συνάρτησης getMoment() απαιτείται το compilation να γίνεται με την χρήση arguments : ”debug=1” .

Η μεγαλύτερη χρονοβελτίωση σημειώθηκε επιτυγχάνοντας καλύτερο cache optimization. Συγκεκριμένα καθώς κάθε thread κάνει access ένα στοιχείο αποθηκευμένο στην global memory, απαιτείται σημαντικός χρόνος χρόνος κατά την μεταφορά των δεδομένων συγκριτικά με το αν τα έκαναν access από shared block. Καθώς οι pointers γίνονται συνεχώς swapped σε κάθε computational kernel δεν μπορούμε να αποθηκεύσουμε εξαρχής τα δεδομένα μας σε ένα read only - texture - Symbol περιοχές της κάρτας για γρήγορο accessing. Ως εκτούτου προτού υπολογίσουμε τις κεντρισμένες διαφορές κάθε thread κάθε thread αποθηκεύει σε αντίστοιχο index σε ένα πίνακα στην shared memory την πληροφορία του Input Matrix που "αντιπροσωπεύει". Οι τιμές αυτές που φορτώνονται - μετακινούνται στην shared memory του streaming multiprocessor. Με στόχο την εκμετάλλευση του locality των δεδομένων και των threads μας χρησιμοποιούμε ένα grid από blocks και threads και μία δισδιάστατη αναπαράσταση του abstraction πάνω στο μητρώο. Εφόσον σχεδόν κάθε τιμή γίνεται accessed 6 φορές από γειτονικά threads (Εξαίρεση αποτελούν τα boundary cells) επιθυμούμε την αποθήκευση της στην shared memory. Πρόβλημα δημιουργούνται ωστόσο στις τιμές των boundary cells σε κάθε block τα οποία απαιτούν εκ νέου accesses σε global memory, warp divergence και if statements. Οφείλουμε να σημειώσουμε πως η ιδέα επίλυσης του προαναφερθέντος φαινομένου προέκυψε από διαφάνειες σε μάθημα HPC του Boston University. Αρχικά κάνουμε padding στο κάθε block ώστε το block_size αυξάνεται κατά 2. Παραδειγματικά ένα block 8*8 = 64 threads "αντιπροσωπεύει" χώρο 10*10 = 100 cells. Με στόχο να πετύχουμε μέγιστο utilization της κάρτας, χρησιμοποιούμε τα threads δύο φορές για μεταφορά ουσιώδης πληροφορίας τόσο στις 64 εσωτερικές θέσεις όσο και στα "ghost - halo" cells της shared μνήμης. Ύστερα από συγχρονισμό των threads σε κάθε block εκτελούμε τον υπολογισμό με τις κεντρισμένες διαφορές κάνοντας access αποκλειστικά πληροφορία στην shared memory.



	Time in seconds	
Matrix Dimension :	**N = 1024**	**N = 2048**
Serial Execution :	32.0992	540.625
Cuda execution :	1.33776	21.4717
Speedup Ratio :	23.9947	25.1784

Η προαναφερθείσα μεθοδολογία όπως φαίνεται παρουσιάζει σημαντικά βελτιστοποιημένους χρόνους σε σύγκριση με κάθε άλλη προσπάθεια που είχαμε υλοποιήσει και φυσικά σε σύγκριση με τον σειριακό κώδικα. Επιπροσθέτως ο κώδικας φαίνεται να είναι scalable καθώς αυξάνοντας το μέγεθος του μητρώου φαίνεται να βελτιώνεται και η απόδοση της παραλληλοποίησης.

❖ Part B

Αρχικά για την πραγματοποίηση του προβλήματος χρησιμοποιήσαμε 1 μητρώο N*N το οποίο αρχικοποιείται τυχαία με φερορμόνη μεταξύ των τιμών 0 και της MaxPher. Καθώς δεν ορίζεται ο αριθμός των ants συγκεκριμένα αποφασίσαμε πως ανά 5 cells στο αρχικό μητρώο θα υπάρχει μία πιθανότητα ένα μυρμήγκι να οριστεί τυχαία μία από αυτές. Ως εκτούτου ο μέγιστος αριθμός μυρμηγκιών είναι N*N/5. Ωστόσο λόγο της τυχαιότητας ο αριθμός τους παρουσιάζει διακυμάνσεις. Είναι σημαντικό να αναφερθεί εκ νέου πως η κάρτα γραφικών δεν παρουσιάζει επιθυμητές χρονοβελτιστοποιήσεις εάν ο κώδικας μας απαιτεί σημαντικό αριθμό ατομικών εντολών ή σημαντικό warp divergence.
Συνοπτικά κρίναμε πως δεν μπορεί κάθε μέρος της διαδικασίας να παραλληλοποιηθεί μαζικά επαρκώς και αποφασίσαμε να εστιάσουμε τον παραλληλισμό στην GPU στο ζητούμενο " Τα κύτταρα που δεν περιέχουν μυρμήγκια χάνουν ένα ποσοστό της υπάρχουσας φερομόνης σε κάθε χρονικό βήμα ". Τα μυρμήγκια προσομοιάστηκαν σαν δύο AOS (array of structures), κάθε ένα από τα οποία συγκρατεί είτε την παλιά είτε την καινούργια θέση των μυρμηγκιών. Το structure αποτελείται από δύο integers οι οποίοι δεικτοδοτούν την θέση του αντίστοιχου μυρμηγκιού στο μητρώο της φερορμόνης. Καθώς ο αριθμός των μυρμηγκιών είναι πολύ λίγος αντί να ψάχνουμε μέσα στο μητρώο της φερορμόνης την ύπαρξη ή ανυπαρξία μυρμηγκιών γνωρίζουμε a priori τις θέσεις τους και σειριακά υπολογίζουμε την νέα θέση του κάνοντας disperse την φερορμόνη της προηγούμενης ταυτοχρόνως. Στην περίπτωση που θέλαμε να παραλληλοποιήσουμε αυτό το μέρος της διαδικασίας θα απαιτούταν μεγάλο κομμάτι της διαδικασίας να άνηκε σε critical regions καθώς δεν μπορούμε να κάνουμε disperse την φερορμόνη χωρίς την ύπαρξη race conditions και δεν γίνεται να μετακινηθεί ένα μυρμήγκι χωρίς να ελέγξει την ύπαρξη άλλων μυρμηγκιών. Αν και ο συγκεκριμένος έλεγχος θα μπορούσε να εκμεταλλευτεί τον παραλληλισμό της κάρτας γραφικών, ο χαμηλός αριθμός των μυρμηγκιών, εκ του αποτελέσματος, υποδεικνύει πως δεν αποτελεί η συγκεκριμένη πράξη το μεγαλύτερο workload της διαδικασίας.
Αντίθετα η ανανέωση κάθε κελιού του μητρώου φερορμόνης, και ο έλεγχος για το εάν υπάρχει μυρμήγκι σε αυτό, παρουσιάζει πολύ καλύτερο στόχο παραλληλισμού. Καθώς οι θέσεις των μυρμηγκιών σε αυτό το βήμα δεν μεταβάλλονται, μπορούν να αποθηκευτούν σε read only, fast access cache στην μνήμη της κάρτας και ελέγχονται από κάθε thread. Σε περίπτωση που βρεθεί κάποιο μυρμήγκι ανανεώνεται ένα flag και στην συνέχεια γίνεται κατάλληλη ποσοστιαία μείωση της φερορμόνης. Αποφεύγουμε με την χρήση τριαδικών τελεστών το warp divergence και επιπλέον πράξεις από συγκεκριμένα threads.
Οι χρόνοι φαίνεται να επιβεβαιώνουν τις σχεδιαστικές μας αποφάσεις :

N = 500	**Serial Code Execution**	**Ants Number = 8972**	**Execution Time (S) = 234.1638**
	Parallel Code Execution	**Ants Number = 8862**	**Execution Time (S) = 8.953885**