# Παράλληλα και Διανεμημένα Συστήματα – 3η Εργασία (CUDA)

*Πούλιος Ηλίας ( 9155 ) – Ραφαήλ Μπουλογεώργος ( 9186 )*

Github repository: https://github.com/rafampou/PDS_ex3

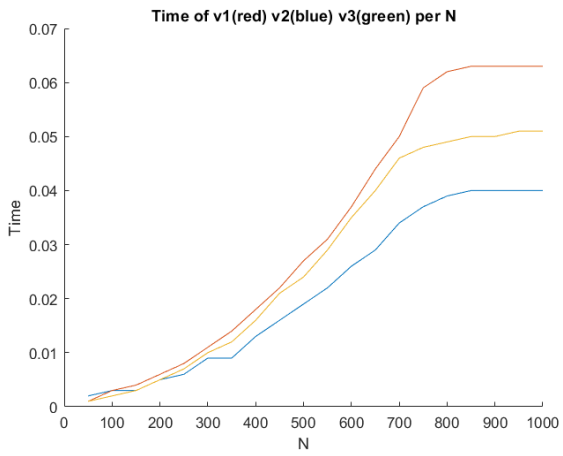
Github download zip: https://github.com/rafampou/PDS_ex3/archive/master.zip

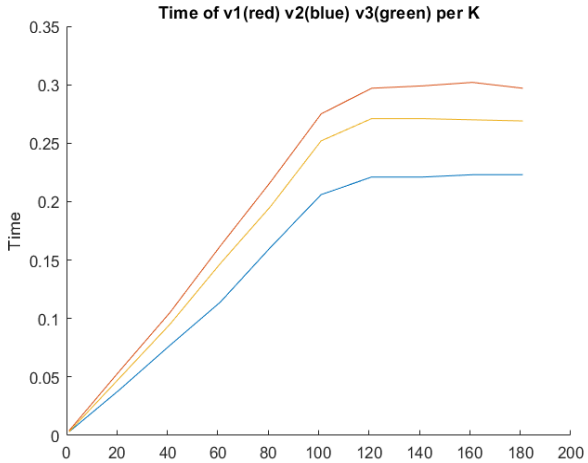
## Εισαγωγή

Η Εργασία μας ζητά να υλοποιήσουμε το μοντέλο ising με nxn διαστάσεις και k επαναλήψεις. Ο αλγόριθμος θα πρέπει αρχικά να υλοποιηθεί σειριακά και στην συνέχεια να παραλληλοποιηθεί ώστε να γίνει διαχειρίσιμος από τον επεξεργαστή της κάρτας γραφικών. Για τον λόγο αυτό χρησιμοποιούμε την CUDA που μας επιτρέπει να εκτελέσουμε εντολές σε πυρήνες των καρτών γραφικών την NVidia.

Σε όλες τις υλοποιήσεις έχουμε εσωτερικά την main ώστε να διευκολύνουμε τον έλεγχο των συναρτήσεων. Ως ορίσματα η main μπορεί να πάρει 2 αριθμού K και N. Αλλιώς χρησιμοποιεί τις τιμές που έχουν γίνει define στο πάνω μέρος των αρχείων. Στους αλγορίθμους V1 V2 V3 τα αποτελέσματα ελέγχονται από την σειριακή υλοποίηση και επαληθεύονται εκτυπώνοντας τους χρόνους. Επιπλέον ως αρχικά σημεία ορίζουμε τυχαία σημεία σε έναν πίνακα G με ίση πιθανότητα να γίνουν -1 ή 1.[1] .

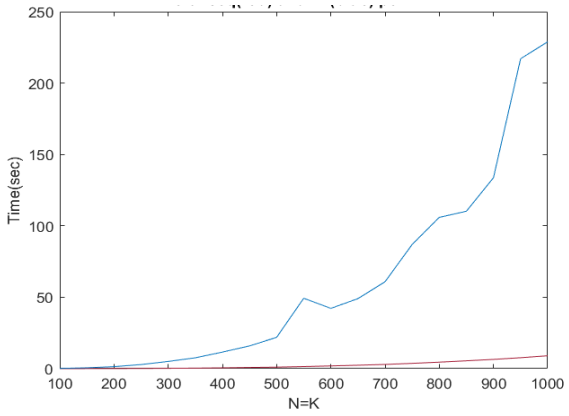
## Στατιστικά και ταχύτητα




Από το διάγραμμα δεξιά (N = 500) βλέπουμε ότι καθώς το K αυξάνει πάνω από μια τιμή ( 107 για την δική μας περίπτωση ) ο χρόνος δεν αυξάνεται καθώς ο πίνακας δεν έχει καμία αλλαγή, ενώ για τις υπόλοιπες τιμές η αύξηση είναι γραμμική.
Κατά την αύξηση του N (διάγραμμα αριστερά, K = 20) βλέπουμε ότι για μικρές τιμές ο αλγόριθμος v3 δεν είναι αποτελεσματικός ενώ για πολύ μεγάλες τιμές έχουμε βελτίωση σχεδόν 50% σε σχέση με τον v1.

[1] Η ίση πιθανότητα εξαρτάται από την συνάρτηση rand.




Στο αριστερό διάγραμμα βλέπουμε πως αυξάνεται ο χρόνος της σειριακής υλοποίησης σε σχέση με τον χρόνο της v1 για αυξανόμενο N=K. Οι τιμές αυτές είναι αρκετά ακραίες για πολύ μεγάλες τιμές του K, αλλά φανερώνουν έντονα την επιτάχυνση του αλγορίθμου. Στην πραγματικότητα ο χρόνος θα σταματούσε να ανεβαίνει μετά από έναν αριθμό K καθώς τα στοιχεία του πίνακα θα παρέμεναν ίδια σε κάθε επανάληψη.

**Εικόνα 1 Χρόνος v1(κόκκινος) sequential(μπλε) ως προς N=K**

## Σειριακή υλοποίηση

Στην σειριακή υλοποίηση εκτελούμε το μοντέλο ising όπως ορίζεται. Η δυσκολία στο σημείο αυτό έγκειται στην αποφυγή των if statements για τα στοιχεία που βρίσκονται στα περιθώρια του πίνακα, καθώς η εκτέλεσή τους αποτελεί δαπανηρή διαδικασία για την GPU. Για τον σκοπό αυτό υπολογίζουμε τα γειτονικά σημεία χρησιμοποιώντας το υπόλοιπο της διαίρεσης της θέσης του σημείου με το πλήθος των σημείων.

## $V_1$ GPU with one thread per moment

Ορίζουμε μια συνάρτηση kernel, και δεσμεύουμε μνήμη n*n σε 2 πίνακες, έναν για ανάγνωση και έναν για εγγραφή. Χρησιμοποιούμε το μέγιστο μέγεθος του block 32 x 32 και δημιουργούμε ένα πλέγμα N/32 x N/32 . Όμως επειδή το N είναι τυχαίος παίρνουμε τον αμέσως μεγαλύτερο ακέραιο. Εκτελούμε το kernel k φορές, δηλαδή για κάθε σάρωση.

## V2 GPU with one thread computing a block of moments

Ορίζουμε το ***MACRO SPINS_PER_THREAD_DIM*** το οποίο δείχνει τη διάσταση του block of moments που θα υπολογίζει το κάθε thread. Δηλαδή αν ορίσουμε το ***SPINS_PER_THREAD_DIM*** ίσο με 3 τότε το κάθε thread θα υπολογίσει 3*3 spins. Τα spins μοιράζονται στα threads ως εξής (έστω ***SPINS_PER_THREAD_DIM 3):*** Αν φανταστούμε τον πίνακα n*n των spins τότε το 'πρώτο' thread του grid (threadIdx.x + blockIdx.x * blockDim.x = 0 && threadIdx.y + blockIdx.y * blockDim.y = 0) αναλαμβάνει να υπολογίσει το πρώτο πάνω αριστερά 3*3 block of spins. Συγκεκριμένα αναλαμβάνει τα spins 0, 1, 2, n, n+1, n+2, 2n, 2n+1, 2n+2. Το 'δεύτερο' thread στη πρώτη σειρά του grid (threadIdx.x + blockIdx.x * blockDim.x = 0 && threadIdx.y + blockIdx.y * blockDim.y = 1) αναλαμβάνει τα spins 3,4,5,n+3,n+4,n+5,2n+3,2n+4,2n+5. Το 'δεύτερο' thread στη πρώτη στήλη του grid (threadIdx.x + blockIdx.x * blockDim.x = 1 && threadIdx.y + blockIdx.y * blockDim.y = 0) αναλαμβάνει τα spins 3n,3n+3,3n+2,4n,4n+1,4n+2,5n,5n+1,5n+2 κτλ. Έτσι όλο το μπλοκ αναλαμβάνει τελικά $32^2 \cdot SPINS\_PER\_THREAD\_DIM^2$ .

*Από αυτά που ακούσαμε στο μάθημα, εκ των υστέρων, καταλάβαμε ότι θα ήταν καλύτερο το κάθε thread να μην υπολογίζει διαδοχικά spins. Ωστόσο λόγω του ότι πλησιάζει η εξεταστική δε καταφέραμε να διορθώσουμε την υλοποίησή μας.*

## V3 GPU with multiple thread sharing common input moments

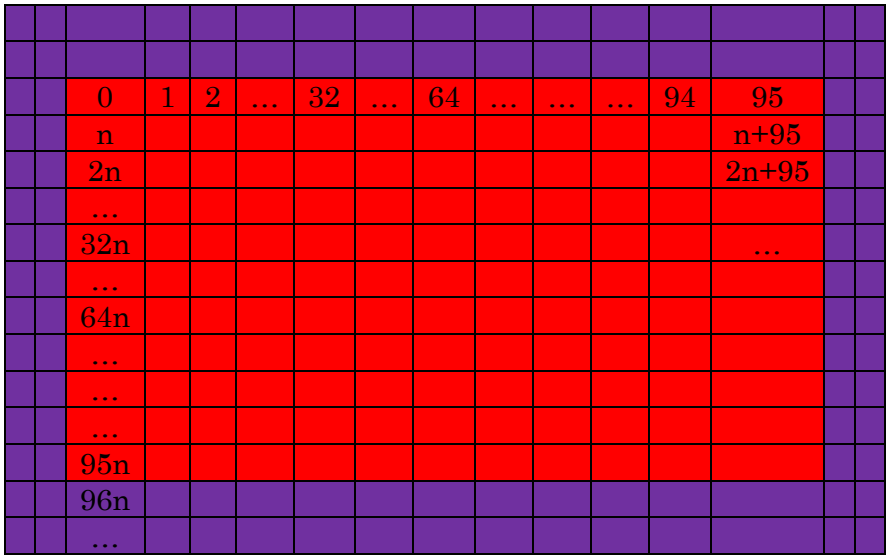
Η ίδια διαδικασία με το V2 ακολουθείται και στο V3. Δηλαδή κάθε thread αναλαμβάνει ένα block of spins. Η διαφορά είναι ότι στο V3 πριν αρχίσουμε τους υπολογισμούς περνάμε όλα τα απαραίτητα spins στη shared memory.

*Πούλιος Ηλίας ( 9155 ) – Ραφαήλ Μπουλογεώργος ( 9186 )*

Έστω πάλι ότι το *SPINS_PER_THREAD_DIM* είναι ίσο με 3. Άρα στη shared memory θα πρέπει να φέρουμε $(32 \cdot SPINS\_PER\_THREAD\_DIM + 4)^2 = 10000$. Το + 4 μπαίνει διότι για τους υπολογισμούς των spins που βρίσκονται στα όρια χρειαζόμαστε επιπλέον spins. Η διαδικασία της μεταφοράς των spins από την global στη shared memory πρέπει να μοιραστεί μεταξύ των threads. Ωστόσο δεν μπορεί να μοιραστεί ισάξια διότι έχουμε 32*32 = 1024 threads και πρέπει να μετακινήσουμε 10000 spins. Η διαδικασία γίνεται ως εξής: το 'πρώτο' thread του block (threadIdx.x = 0 && threadIdx.y = 0) φέρνει τα spins που είναι ζωγραφισμένα με κόκκινο, το 'δεύτερο' thread στη πρώτη σειρά του block (threadIdx.x = 0 && threadIdx.y = 1) φέρνει τα spins που είναι ζωγραφισμένα με πράσινο, , το 'δεύτερο' thread στη πρώτη στήλη του block (threadIdx.x = 1 && threadIdx.y = 0) φέρνει τα spins που είναι ζωγραφισμένα με μπλέ κτλ.

| 0 | 1 | 2 | … | 32 | 33 | 34 | … | 64 | 65 | 66 | … | 96 | 97 | 98 | 99 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 100 | 101 | 102 | | 132 | 133 | 134 | | 164 | 165 | | | | | | |
| … | | | | | | | | | | | | | | | |
| 3200 | | | | | | | | | | | | | | | |
| 3300 | | | | | | | | | | | | | | | |
| 3400 | | | | | | | | | | | | | | | |
| … | | | | | | | | | | | | | | | |
| 6400 | | | | | | | | | | | | | | | |
| 6500 | | | | | | | | | | | | | | | |
| 6600 | | | | | | | | | | | | | | | |
| … | | | | | | | | | | | | | | | |
| 9600 | | | | | | | | | | | | | | | |
| 9700 | | | | | | | | | | | | | | | |
| 9800 | | | | | | | | | | | | | | | |
| 9900 | | | | | | | | | | | | | | | |

Οι δείκτες (0, 1, 2 κτλ.) μέσα στα κελιά δεν αντιστοιχούν στους δείκτες της global memory. Δε σημαίνει, δηλαδή, ότι το 'πρώτο' thread του block (threadIdx.x = 0 && threadIdx.y = 0) που φέρνει τα κόκκινα θα φέρει από την global memory τα στοιχεία με δείκτες 0, 32, 64 κτλ. Παρακάτω βλέπουμε ποια στοιχεία της global memory θα έχει η shared memory του πρώτου block (blockIdx.x = 0 && blockIdx.y = 0):




Δηλαδή το στοιχείο [2 , 2]* της shared memory (του πρώτου block !) θα έχει το 0 της global memory

*η shared είναι μονοδιάστατη. Χάριν ευκολίας χρησιμοποιούμε δισδιάστατο δείκτη.

Ο έλεγχος για το αν υπάρχουν ή όχι αλλαγές στον πίνακα των spins μετά από κάθε επανάληψη γίνεται ως εξής:

- Στη σειριακή υλοποίηση μετά από κάθε επανάληψη συγκρίνουμε σειριακά όλα τα spins των δύο πινάκων (pointers). Αν βρούμε ότι δύο spins δεν είναι ίδια σταματάμε τη διαδικασία και ο αλγόριθμος συνεχίζει στην επόμενη επανάληψη.
- Στην παράλληλη υλοποίηση ο έλεγχος γίνεται παράλληλα. Μετά από κάθε κλήση του kernel ising καλούμε το kernel checkForNoChanges το οποίο συγκρίνει τα στοιχεία των δύο πινάκων (pointers). Ωστόσο δεν συγκρίνει όλα τα στοιχεία αλλα ένα μέρος τους (συγκεκριμένα συγκρίνει 32*32*4 spins δηλαδή ουσιαστικά 4 blocks). Αν αυτά τα στοιχεία είναι ίδια τότε καλούμε εκ νέου το kernel checkForNoChanges αλλα αυτή τη φορά το βάζουμε να ελέγξει όλα τα στοιχεία.