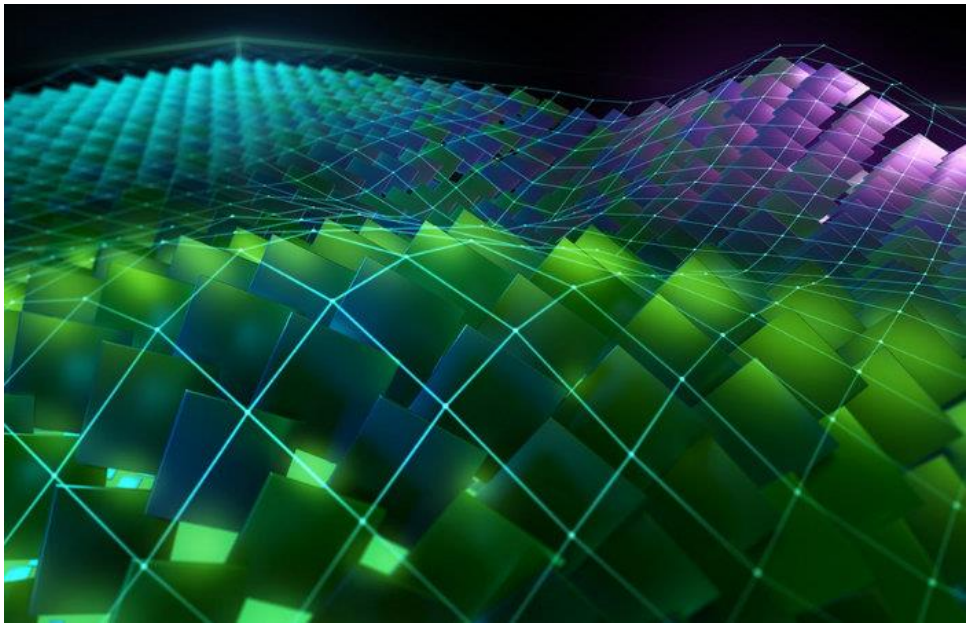


Εξέλιξη μοντέλου Ising με CUDA

Τζάτσης Νικόλαος

Παράλληλα και Διανεμημένα Συστήματα

3/2/2022

https://github.com/nikos-tz/3_cuda_code

Υλοποιήσεις

- V0 (Σειριακό)

Η υλοποίηση της V0 είναι εξαιρετικά απλή, έγινε σύμφωνα με τις οδηγίες τις εκφώνησης και το μόνο ιδιαίτερο σημείο ήταν η επίτευξη των περιοδικών συνοριακών συνθηκών χωρίς τη χρήση εντολών if το οποίο έγινε με τον παρακάτω τύπο:

`i = ( n + ( i % n ) ) % n` , όπου `i` είναι ο δείκτης του στοιχείου που θέλουμε να βρούμε και n είναι η διάσταση του (n*n) πλέγματός μας. Η ορθότητα της υλοποίησής μας φαίνεται από το ότι έχουμε τα επιθυμητά αποτελέσματα για k = 1 (είναι εύκολο να τυπώσουμε το πλέγμα πριν και μετά την κλήση της V0 και να καταλάβουμε αν λειτούργησε σωστά η υλοποίησή μας αρκεί να θέσουμε μικρό ν) και από το ότι το πλέγμα μας από ένα σημείο και μετά καταλήγει σε κύκλο 2 καταστάσεων, δηλαδή η κατάσταση του πλέγματος για k = m και για k = m+2 είναι ίδια για όλα τα m > από κάποιο k'. Το πρόγραμμα test_convergence δείχνει τα παραπάνω και το χρησιμοποιήσαμε για να βρούμε για ποια k καταλήγουμε σε επαναλαμβανόμενη κατάσταση ανάλογα με το ν που έχουμε θέσει. Τα αποτελέσματα ήταν πως για ν = 1024, 4000, 8000, 12000, 16000 είχαμε k = 10, 28, 30, 32, 35. Ενδεικτικά παραθέτουμε τους χρόνους t (σε δευτερόλεπτα) που κατέγραψε η V0 και αντιστοιχούν στα παραπάνω ν και k: t = [0,82 35 150 350 700].

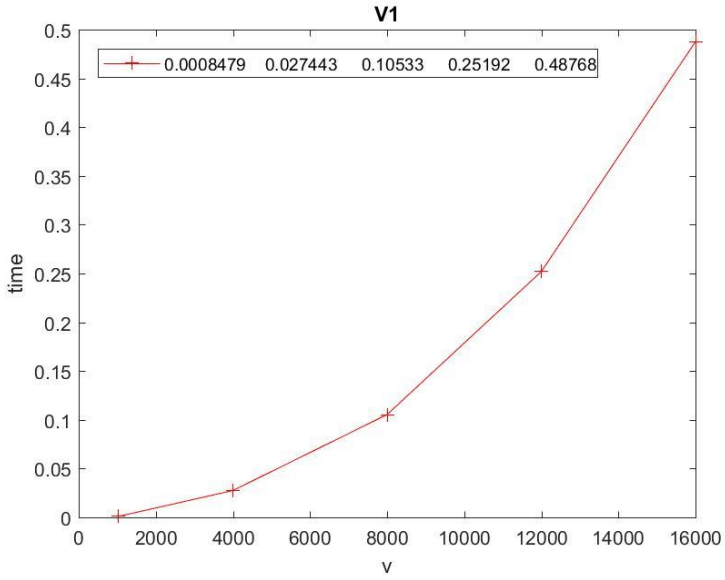
- V1

Το σημαντικότερο πρόβλημα που αντιμετωπίσαμε στο V1 και γενικά στις cuda υλοποιήσεις ήταν το αν θα υπολογίζαμε τις k καταστάσεις μέσα στο kernel με ένα for loop ή αν το kernel θα υπολόγιζε μόνο μία κατάσταση και θα το καλούσαμε k φορές από την main. Η πρώτη επιλογή θα απαιτούσε να συγχρονίζαμε όλα τα thread από όλα τα block μετά από κάθε κατάσταση, καθώς δεν θα θέλαμε ένα thread να θέλει να χρησιμοποιήσει την τιμή του γειτονικού του spin για k = m (ώστε να υπολογίσει την τιμή του δικού του spin για k = m+1) και το γειτονικό του thread να μην έχει προλάβει να γράψει αυτήν την τιμή ή να την έχει ανανεώσει για k = m+1. Επειδή όμως, δεν καταφέραμε να

βρούμε κάποιον τρόπο για να συγχρονίσουμε thread από διαφορετικά block (για thread που ανήκουν στο ίδιο block είναι εύκολος ο συγχρονισμός) οδηγηθήκαμε στην δεύτερη επιλογή. Για την V1 χρησιμοποιήσαμε grid και block μιας διάστασης και για λόγους ευκολίας θέσαμε οι ρίζες του αριθμού των block και του αριθμού των thread / block να ακολουθούν την παρακάτω εξίσωση:

```
num_blocks_sqrt * threads_per_block_sqrt = n
```

Έπειτα στο kernel γίνεται το κατάλληλο indexing ώστε να βρει κάθε thread τη θέση του στο πλέγμα και να υπολογίσει το spin του. Οι χρόνοι (σε δευτερόλεπτα) που σημείωσε η V1 είναι οι εξής:

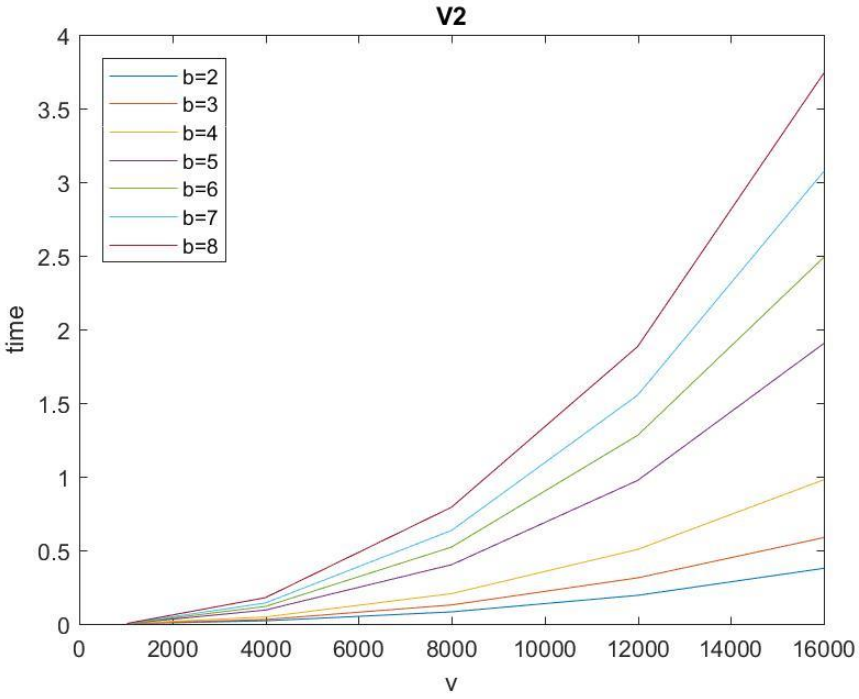


- V2

Στην V2 χρησιμοποιήσαμε grid και block 2 διαστάσεων και επιπλέον για να μοιράσουμε τέλεια το πλέγμα μας στα thread θέσαμε:

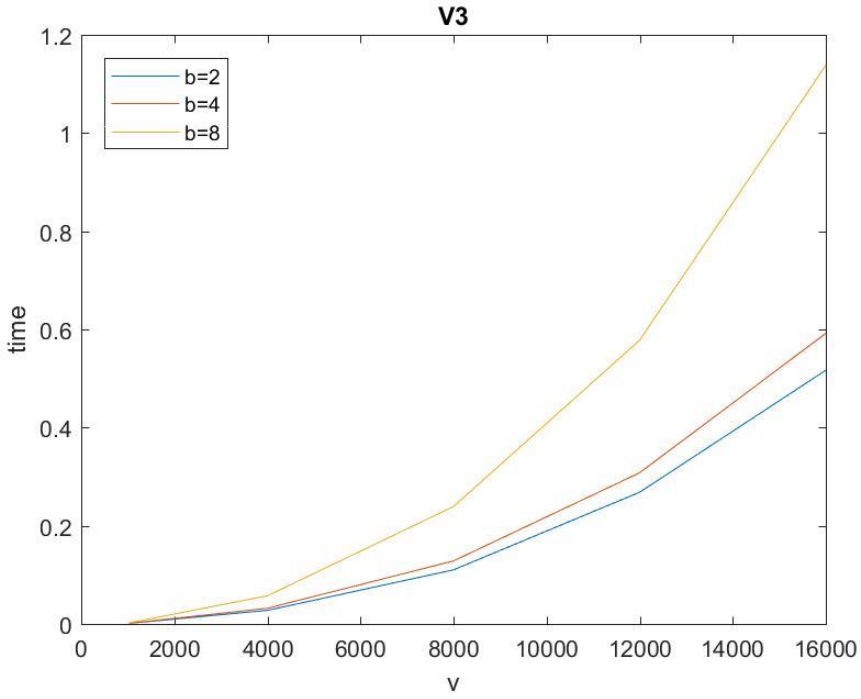
```
n = num_blocks_sqrt * threads_per_block_sqrt * moments_per_thread_sqrt
```

Έπειτα στο kernel γίνεται το κατάλληλο indexing και μετά ακολουθεί ένα for loop ώστε κάθε thread να υπολογίσει momments_per_thread_sqrt^2 spin. Οι χρόνοι της V2 για διάφορα b = moments_per_thread_sqrt είναι οι παρακάτω και είναι εμφανές πως για b = 2 έχουμε τους μικρότερους χρόνους.



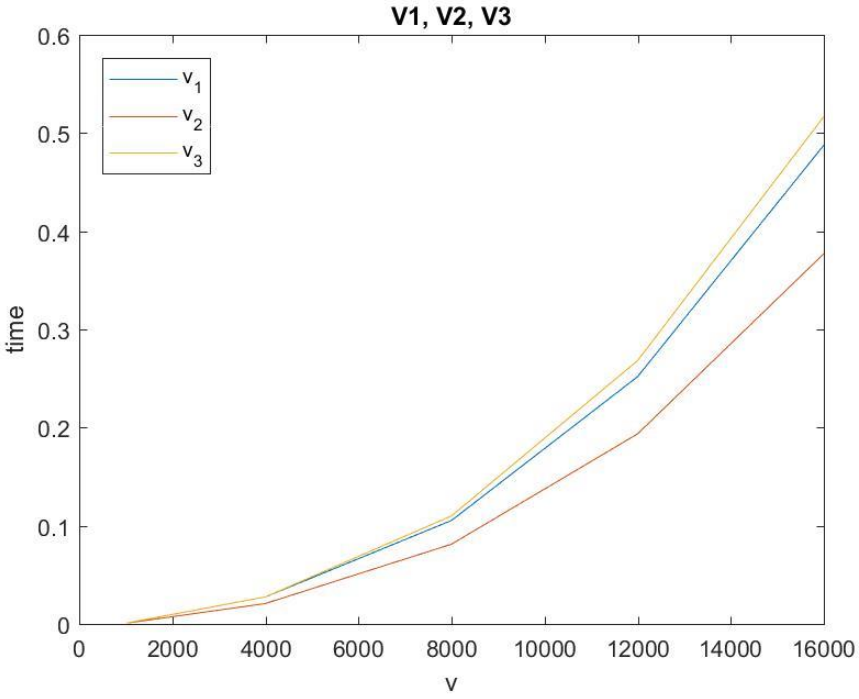
- V3

Η main συνάρτηση της V3 είναι ίδια με αυτήν της V2, οπότε περνάμε στο kernel όπου χρησιμοποιούμε έναν πίνακα read_shared, μέσα στη shared memory, 2 διαστάσεων και με την κάθε διάσταση να έχει τιμή ίση με (threads_per_block_sqrt * b) + 2 ώστε να βάλουμε σε αυτόν τον πίνακα όλες τις τιμές για τα spin που θα υπολογίσει το συγκεκριμένο block αλλά και τις τιμές από τα γειτονικά spin που θα χρειαστεί για να τα υπολογίσει (εξού και το +2 στην διάσταση). Αφού βάλουμε τις τιμές μας από την read (που είναι στην global memory) στην read_shared υπολογίζουμε τα spin για την επόμενη κατάσταση και τα βάζουμε στην write_shared (που είναι και αυτή προφανώς στη shared memory) και έπειτα την περνάμε στην write (που είναι στην global memory). Προκειμένου να μην ξεπεράσουμε το όριο της shared memory μνήμης έπρεπε το γινόμενο threads_per_block * b να μην ξεπερνάει την τιμή 72. Με βάση τα αποτελέσματα από την V2 περιμένουμε να έχουμε τους πιο μικρούς χρόνους για b = 2 κάτι το οποίο συνέβη όπως βλέπουμε παρακάτω.



Σύγκριση V1-V2-V3

Παρακάτω βλέπουμε μαζί τους χρόνους των V1, V2, V3 (προφανώς τα V2, V3 είναι με τα βέλτιστα b). Αυτό που παρατηρούμε είναι πως το V2 είναι πιο γρήγορο από το V1, όμως το V3 είναι πιο αργό και από το V2 αλλά και από το V1, κάτι που μας προβληματίζει καθώς θα έπρεπε να είχαμε τα πιο γρήγορα αποτελέσματα με τη χρήση της shared memory.



Τέλος να σημειωθεί πως οι υλοποιήσεις μας έτρεξαν στη συστοιχία Αριστοτέλης του πανεπιστημίου μας, και πιο συγκεκριμένα στην κάρτα Nvidia Tesla P100.