Εργασία 3
Παράλληλα και Διανεμημένα Συστήματα

Καρελής Παναγιώτης (9099)
2/1/2020

Εισαγωγή

Στα πλαίσια αυτής της εργασίας υλοποιούμε την χρονική εξέλιξη μιας παραλλαγής του *Ising Model* για δοσμένο αριθμό βημάτων k. Η αλλαγή σε σχέση με το standard Ising Model είναι ότι στην εργασία μας, η γειτονιά γύρω από το κάθε σημείο του πλέγματος (lattice point) έχει μέγεθος ενός 5x5 παραθύρου (window) και η επίδραση των γειτονικών σημείων δίνεται από έναν πίνακα βαρών w (weight matrix), όπως περιγράφεται στην εκφώνηση της εργασίας. Για τα σημεία που βρίσκονται στις άκρες του πλέγματος, ισχύουν περιοδικές οριακές συνθήκες (toroidal / periodic boundary conditions). Η υλοποίηση γίνεται σε C++ και CUDA.

Τέσσερεις εκδόσεις, που αναλύονται παρακάτω, υλοποιήθηκαν για τους σκοπούς της εργασίας. Ο κώδικας, το πρόγραμμα επαλήθευσης ορθότητας, τα γραφήματα και όλες οι μετρήσεις βρίσκονται στο GitHub: https://github.com/karelispanagiotis/PDS_Exercise_3.

V0: Sequential

Η sequential υλοποίηση είναι πολύ απλή. Χρησιμοποιούμε δύο πίνακες n x n, ο ένας για να διαβάζουμε τη τρέχουσα κατάσταση του μοντέλου κι ο άλλος για να γράφουμε την επόμενη. Για κάθε σημείο του πίνακα υπολογίζουμε το spin που θα έχει στην επόμενη κατάσταση. Μετά από κάθε βήμα, εναλλάσσουμε τους `pointers` και επαναλαμβάνουμε για όσα βήματα k μας ζητείται.

Για τον υπολογισμό του spin για κάποιο lattice point χρησιμοποιούμε την συνάρτηση
```
int calculateSpin(int *G, float *w, int n, int pos_i, int pos_j)
```
η οποία πολλαπλασιάζει τον πίνακα βαρών w με τα spins των γειτονικών σημείων, τα προσθέτει και ανάλογα με το αποτέλεσμα του αθροίσματος επιστρέφει την τιμή του spin. Αν το άθροισμα είναι 0, το spin παραμένει όπως πριν αλλιώς το spin παίρνει το πρόσημο του αθροίσματος[1].

Επιπλέον, για την εφαρμογή των οριακών συνθηκών, χρησιμοποιούμε την συνάρτηση
```
inline int calcLatticePos(int i, int j, int n)
```
η οποία χρησιμοποιεί αριθμητική υπολοίπων (modular arithmetic).

Τέλος, επειδή για περιττό αριθμό βημάτων τα δεδομένα δεν βρίσκονται στη σωστή θέση μνήμης γι' αυτόν που καλεί την συνάρτηση, απλά αντιγράφουμε τα δεδομένα.

---

[1] Παρατηρήσεις:

- Η σύγκριση με το 0 γίνεται με χρήση μια σταθερής `epsilon = 1e-6f`.
- Θεωρούμε πως το μεσαίο στοιχείο του πίνακα `w` είναι ίσο με 0,
  `w[2,2] = 0.0000`
  ώστε όταν πολλαπλασιαστεί με το spin του στοιχείου που θέλουμε να δώσει 0 και να μην συμπεριληφθεί στο άθροισμα.
- Ο πίνακας `w` είναι τύπου `float`, ώστε να μπορεί να αξιοποιηθεί καλύτερα στις εκδόσεις GPU.

V1: GPU with one thread per moment

Σε αυτήν την έκδοση και στις επόμενες, χρησιμοποιούμε την υπολογιστική ισχύ της κάρτας γραφικών για τον υπολογισμό του μοντέλου μετά από k επαναλήψεις. Σε όλες αυτές τις εκδόσεις, γίνεται μεταφορά των δεδομένων του προβλήματος στην μνήμη της GPU (device memory) με τις συναρτήσεις `cudaMalloc(), cudaMemcpy()`. Αφού υπολογιστούν τα τελικά αποτελέσματα, τότε μεταφέρονται ξανά, από τη Device Memory στη κύρια μνήμη του επεξεργαστή.

Στην CUDA V1, ζητείται η κλήση μιας kernel function με grid που να ταιριάζει στο Ising Model. Δηλαδή ένα grid διαστάσεων n x n block, με ένα thread ανά block. Κάθε block υπολογίζει ένα spin της επόμενης κατάστασης του μοντέλου.

```
dim3 grid2D(n, n);
calculateSpin<<<grid2D, 1>>>()
```

Για να προσδιορίσει κάθε block ποιο Lattice Point να υπολογίσει, χρησιμοποιούνται οι μεταβλητές `blockIdx.x, blockIdx.y`. Η μόνη διαφορά στη συνάρτηση `calculateSpin()` είναι ότι πλέον είναι τύπου `void` και παίρνει ως επιπλέον όρισμα έναν ακόμη πίνακα, στον οποίο γράφουμε την επόμενη κατάσταση του μοντέλου.

Φυσικά, το μειονέκτημα αυτής της έκδοσης είναι ο μεγάλος αριθμός block που μετά από κάποιο όριο δεν μπορούν να εξυπηρετηθούν παράλληλα, ανάλογα με τις δυνατότητες της GPU.

V2: GPU with one thread computing a block of moments

Για να αντιμετωπιστεί το παραπάνω πρόβλημα, στη CUDA V2 αναθέτουμε περισσότερη δουλειά ανά block.

Προς το παρόν, ο μέγιστος αριθμός threads ανά block είναι 1024 [CUDA Specifications]. Επιπλέον για να πετύχουμε μεγάλο Occupancy, καλή πρακτική είναι να επιλέγουμε το μέγεθος block να είναι πολλαπλάσιο του warp size, το οποίο είναι 32 για όλες τις GPU της nvidia. Αναλογιζόμενοι και τις ανάγκες της CUDA V3, δηλαδή όσο μεγαλύτερο το block τόσο καλύτερα[2], επιλέγουμε το παραπάνω όριο των 1024 threads/block. Δηλαδή επιλέγουμε block μεγέθους 32x32.

V3. GPU with multiple thread sharing common input moments

Σ' αυτή την έκδοση, εκμεταλλευόμαστε την shared memory, η οποία ανήκει σε κάθε block, έχουν πρόσβαση τα threads του κάθε block, κι είναι πολύ ταχύτερη από την Device Memory. Επιπλέον, η φύση του προβλήματος είναι τέτοια, που κάνει πολλές φορές ανάγνωση από τη μνήμη τα ίδια δεδομένα, όπως είναι η κάθε γειτονιά του lattice και ο πίνακας w.

Αρχικά φέρνουμε στην shared memory τον πίνακα βαρών w (σειρές 82-83). Έπειτα, για τον υπολογισμό ενός lattice block 32x32 χρειαζόμαστε έναν πίνακα 36x36 τον οποίον γεμίζουμε στις σειρές 49-79. Φροντίζουμε τα δεδομένα που περιέχει ο πίνακας 36x36 να ικανοποιούν τις οριακές συνθήκες.

Επαλήθευση Ορθότητας

Αφού επαληθεύτηκε η ορθότητα της sequential έκδοσης, με τον online tester και τα binary αρχεία, χρησιμοποιήθηκε στο πρόγραμμα sampleGen το οποίο δημιουργεί ising models και την εξέλιξη τους για k = {1, 3, 5, 8} για δοσμένο n. Ο έλεγχος των CUDA versions έγινε για όλα τα παραπάνω k και n = {100, 226, 462, 735, 2232, 3508}.

---

[2] Όσο μεγαλύτερο γίνει το lattice block το οποίο θα υπολογίσει ένα block του Kernel τόσο λιγότερες φορές θα χρειαστεί να κάνουμε πρόσβαση στη device memory (βλ. CUDA V3).

Conditional Statements και Warp Divergence

Warp Divergence προκύπτει όταν μόνο μερικά threads από ένα warp ακολουθούν το μονοπάτι μέσα σε κάποιο if. Ο conditional κώδικας αντικαταστάθηκε όπως παρακάτω, όμως ο χρόνος εκτέλεσης έγινε μεγαλύτερος. Οπότε κρατήθηκε ο conditional κώδικας στα αριστερά.

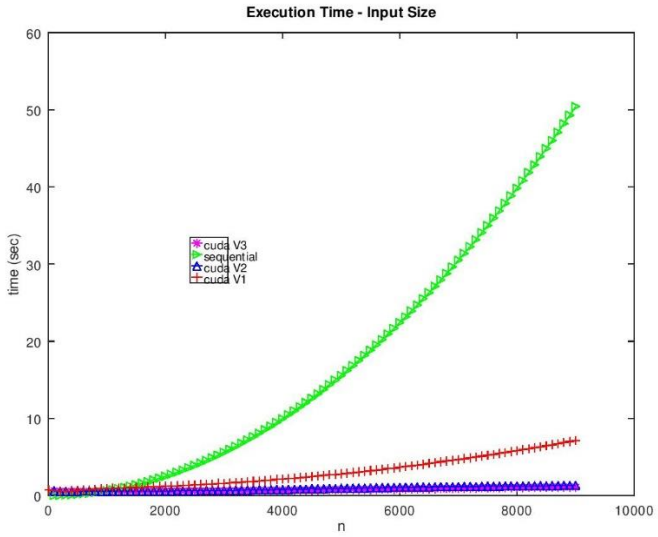
```
if(fabsf(result) < epsilon )

next[gindex] = current[gindex];
else if(result < 0)
   next[gindex] = -1;
else
   next[gindex] = 1;
```

```
bool isZero = fabsf(result)<epsilon;
int spin = 1 - 2*(result<0);
next[gindex] = isZero*current[gindex]
             + (!isZero)*spin;
```

Benchmarks

Οι παρακάτω μετρήσεις έγιναν στην συστοιχία του ΑΠΘ στο gpu partition. Παρακάτω παρουσιάζεται γράφημα του χρόνου εκτέλεσης συναρτήσει του μεγέθους εισόδου n.



Παρατηρούμε ότι η σειριακή έκδοση είναι ανάλογη του n^2, και ότι οι εκδόσεις CUDA V2 και V3 είναι γρηγορότερες της CUDA V1. Η διαφορά των V2 και V3 δεν γίνεται φανερή για τόσο μικρό δείγμα δεδομένων, αλλά φαίνεται στους πίνακες (τα δείγματα ξεκινούν από 100 έως 9000 με βήμα 100). Για τις μετρήσεις αυτές k = 15. [Το γράφημα αυτό κι ακόμη ένα μόνο με τον χρόνο εκτέλεσης των CUDA υπάρχουν στο GitHub].

n = 2000	k = 10	k = 100
Sequential V0	1.663	16.581
CUDA V1	1.105	2.942
CUDA V2	0.584	0.918
CUDA V3	0.565	0.732

n = 10000	k = 10	k = 100
Sequential V0	41.45	414.019
CUDA V1	6.035	51.829
CUDA V2	1.291	3.050
CUDA V3	1.081	1.988

n = 28000	k = 10	k = 100
Sequential V0	325.098	—
CUDA V1	41.126	400.214
CUDA V2	2.851	15.877
CUDA V3	2.020	7.956

Όλες οι μετρήσεις βρίσκονται στο GitHub σε αρχεία [version]_log.txt σε μορφή CSV, στους φακέλους bench/cluster-*.