Σπαΐας Γεώργιος
ΑΕΜ: 8910

Parallel and Distributed Systems

Assignment 3: CUDA Implementation of 2D Ising Model

Για την εργασία αυτή χρησιμοποιήθηκε υπολογιστής με **Intel Core i5-4690K @ 3.50GHz** 4-πύρηνο επεξεργαστή και GPU **NVIDIA GeForce GTX 960** με υποστήριξη CUDA (Maxwell 5.2) σε MS Windows 10. Οι παράλληλες υλοποιήσεις έγιναν με την βοήθεια του Visual Studio 2022, των απαραίτητων extensions και του **Nsight Debugger** για CUDA C/C++.

V0. Sequential Implementation and Validation

Η σειριακή υλοποίηση ήταν ιδιαίτερα κατανοητή. Ο υπολογισμός της τελικής κατάστασης του lattice μας έγινε με την κλήση της compute_State() σε κάθε iteration. Για την αποθήκευση του Lattice σε κάθε επανάληψη χρησιμοποιήσαμε 2 double pointers G και G_transition με το απαραίτητο μέγεθος τους οποίους κάναμε swap μετά από κάθε υπολογισμό νέας κατάστασης. Για την εύρεση των indexes των γειτονικών σημείων χωρίς if statements χρησιμοποιήσαμε το modulo του δείκτη με το μήκος του πίνακα ώστε να μην ξεφεύγουμε από τα όρια.

Neighbor #		Neighbor i	Neighbor j
1	(i+1,j)	*(i+1)%n*	*j*
2	(i−1,j)	*(i−1+n)%n*	*j*
3	(i,j+1)	*i*	*(j+1)%n*
4	(i,j−1)	*i*	*(j−1+n)%n*

Για το validation του κώδικα μας αρχικά ελέγξαμε τα αποτελέσματα για μια αρχική κατάσταση με ιδιαίτερη δομή και προβλεπόμενη σε κάθε επανάληψη μορφή. Στην συνέχεια εμπιστευόμενοι πλέον τον κώδικα μας παρήγαμε και αποθηκεύσαμε κάποια lattices συγκεκριμένου μεγέθους (eval.bin) και το αποτέλεσμα τους για ορισμένες επαναλήψεις για χρήση στις επόμενες υλοποιήσεις.

Οι ιδιαίτεροι πίνακες που χρησιμοποιήθηκαν στην αρχή , έχουν την εξής δομή:

1	-1	1	…	1	-1
-1	1	-1		-1	1
…	…	…	…	…	…
1	1	1	…	1	-1
-1	-1	-1	…	-1	1

Οι πίνακες με την παραπάνω δομή (εναλλάξ) με **άρτιο n** μας δίνουν για **περιττό** αριθμό επαναλήψεων την αντίθετη κατάσταση ενώ για **άρτιο** αριθμό επαναλήψεων ακριβώς την ίδια κατάσταση. Με την χρήση αυτών των πινάκων και πινάκων με ίδια spin σε μια ολόκληρη σειρά ή στήλη τεστάραμε την συμπεριφορά του κώδικα.

V1. Parallel Implementation with a Single moment per Thread

Για την 1η παράλληλη υλοποίηση βασιστήκαμε πάνω στον σειριακό κώδικα μας του V0. Με βάση αυτόν διαμορφώσαμε τα απαραίτητα utility functions σε `__device__` functions και την ***compute_State()*** σε `__global__` .
Στην συνέχεια διαμορφώσαμε το grid (**calculate_grid()**) μας έτσι ώστε να έχουμε τον απαραίτητο αριθμό από blocks και thread ανά block ώστε κάθε thread να υπολογίζει μόνο ένα σημείο σε κάθε iteration.
Για την ευκολότερη αποθήκευση και διαχείριση του lattice μας το μετατρέψαμε από 2D array σε 1D με μήκος **nxn** και προσαρμόσαμε αναλόγως τον τρόπο εύρεσης των γειτονικών indexes.

Για αποφυγή περιττών κλήσεων μέσα στην global και καθώς δεν χρησιμοποιούνται αλλού τις δώσαμε το tag `__forceinline__` προκειμένου να τις τοποθετήσει ο compiler μέσα στο σώμα της global.

Όπως και παραπάνω χρησιμοποιήσαμε τις ειδικές μορφές πίνακα που περιγράψαμε παραπάνω, τους πίνακες του eval.bin και τέλος για τυχαίους πίνακες τρέχουμε και την σειριακή υλοποίηση για να συγκρίνουμε τόσο αποτελέσματα όσο και χρόνους.

V2. Parallel Implementation with Multiple moments per Thread

Για την δεύτερη παράλληλη πατήσαμε και πάλι στην V1 και προσαρμόσαμε τα απαραίτητα στοιχεία. Αρχικά, προσαρμόσαμε τον υπολογισμό του grid ώστε σε κάθε thread να αντιστοιχούν b σημεία (pointsPerThread). Έπειτα, προσθέσαμε έναν Thread Index στην Compute_state() ώστε να μοιράζουμε σε κάθε νήμα ένα μοναδικό μέρος της δουλειάς. Επιπλέον φροντίσαμε για κάθε b να μην φεύγουμε εκτός ορίων του lattice σε περιπτώσεις που:

$$NumberOf\,Blocks \cdot ThreadsPerBlock \cdot PointsPerThread > n \cdot n$$

Η επαλήθευση των αποτελεσμάτων έγινε με τις μεθόδους που χρησιμοποιήθηκαν και παραπάνω.

V3. Shared Memory Parallel Implementation

Για την τρίτη παράλληλη υλοποίηση επιπλέον χρησιμοποιήσαμε και shared memory για κάθε block προκειμένου να αποφύγουμε συνεχή και πιο χρονοβόρα reads/writes στην global μνήμη της GPU. Όπως και προηγουμένως έτσι και τώρα βασιστήκαμε στην προηγούμενη υλοποίηση για την δημιουργία αυτής με κατάλληλες προσαρμογές.

Αρχικά με βάση τα *pointsPerThread* κάνουμε dynamic allocation (extern) ενός `__shared__ shrd_tile[ ]` πίνακα ο οποίος έχει διαστάσεις τέτοιες ώστε να αποθηκεύει όλα τα points που έχουν αναλάβει τα threads του κάθε block αλλά και όλα τα σημεία γειτνίασης που δεν περιέχονται στο workload των threads.

Έτσι, αφού κάθε thread γράψει στην shared memory τα Points του και επιπλέον το *thread Id* 0 καθενός block προσθέσει τα έξτρα στοιχεία που είναι απαραίτητα συγχρονίζονται τα threads και συνεχίζουν με το κομμάτι υπολογισμού της νέας κατάστασης για τα σημεία τους και στην συνέχεια.

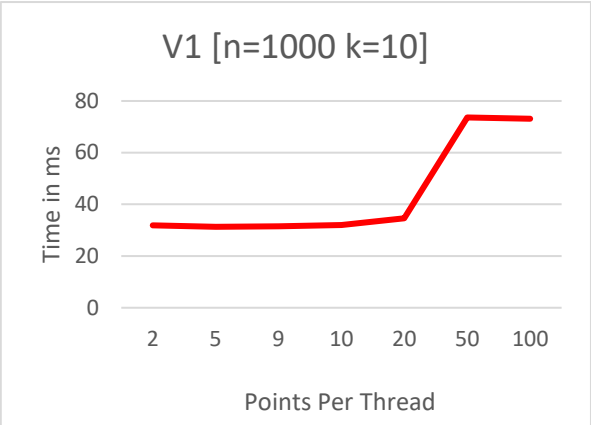
Η επαλήθευση των αποτελεσμάτων έγινε όπως και παραπάνω.

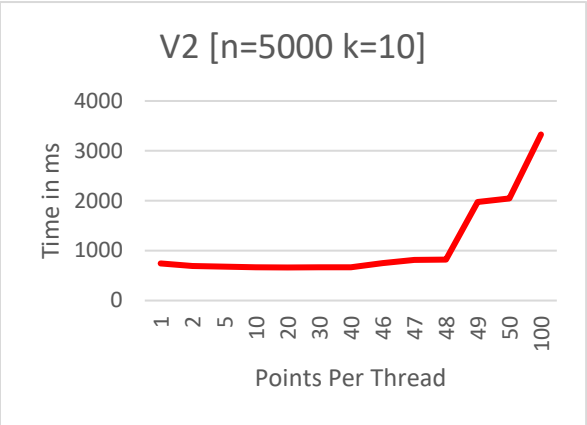
Results & Comments

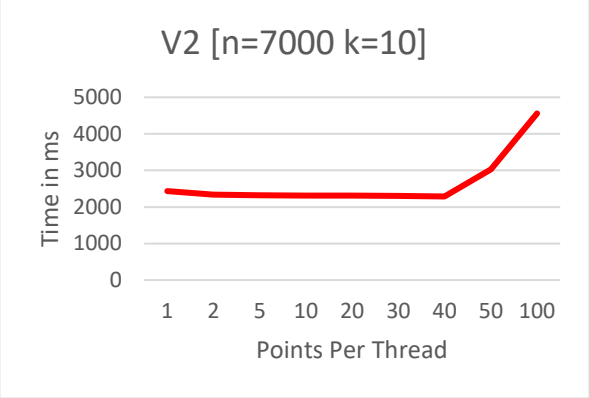
Για την μέτρηση της απόδοσης των υλοποιήσεων κάναμε δοκιμές με τρεις τιμές κυρίως του n= 1000, 5000, 7000 και για αριθμό βημάτων k=10 .
Η χρονομέτρηση έγινε με τις `cudaEventRecord()`, `cudaEventElapsedTime()` χρονομέτρησηγια τις υλοποιήσεις με CUDA και με την `getTickCount()` για την σειριακή υλοποίηση μιας και όλα αναπτύχθηκαν σε περιβάλλον Windows. Παρακάτω φαίνονται οι μέσοι χρόνοι της κάθε υλοποίησης για τα προαναφερθέντα workloads σε milliseconds :

Version	1000	5000	7000
V0	343 ms	8015 ms	15578 ms
V1	32.818 ms	704.246 ms	2389.519 ms
V2	31.363 ms	666.820 ms	2304.781 ms
V3	–	–	–

Για την βελτιστοποίηση της απόδοσης σε σχέση με τα σημεία που αναλαμβάνει κάθε thread κάναμε δοκιμές με διάφορες τιμές του pointsPerthread στα workloads που χρησιμοποιήθηκαν και παραπάνω.







Από τα παραπάνω διαγράμματα και τα αποτελέσματα που πήραμε βλέπουμε:

- Πως έχουμε ραγδαία βελτίωση της απόδοσης για τις παράλληλες υλοποιήσεις συγκριτικά με την σειριακή και ειδικά όσο αυξάνεται το n.
- Πως για μικρό n δεν προσφέρει μεγάλη επιτάχυνση ο διαμοιρασμός περισσότερης δουλειάς σε κάθε Thread σε σχέση με το ένα σημείο ανά Thread.
- Ακόμα βλέπουμε πως η απόδοση βελτιστοποιείται για PointsPerThread κάτω από 1% του n (ppt=30-50). Επιπλέον παρατηρούμε πως μετά από ένα Threshold έχουμε μεγάλη αύξηση του χρόνου εκτέλεσης που μπορούμε να υποθέσουμε πως οφείλεται στα υπερβολικά πολλά reads και writes στην κύρια μνήμη της GPU.

Notes

Η υλοποίηση μου για το V3 εμφάνισε κάποια bugs τα οποία δεν μπόρεσα να εντοπίσω και να λύσω στα πλαίσια της προθεσμίας που μας δόθηκε.
Η υλοποίηση με την shared memory δούλεψε σωστά αλλά μόνο για μικρά $n \leq 80$ και συνεπώς δεν μπόρεσα να συλλέξω μετρήσεις συγκρίσιμες με τις άλλες υλοποιήσεις και κατ' επέκταση να βελτιστοποιήσω την παράμετρο pointsPerThread.