ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΕΦΑΡΜΟΣΜΕΝΩΝ ΜΑΘΗΜΑΤΙΚΩΝ ΚΑΙ ΦΥΣΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΟΜΕΑΣ ΦΥΣΙΚΗΣ

Προσομοιώσεις Monte Carlo σε GPU

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Δημητρίου Ν. Καρκούλη

Επιβλέπων: Κωνσταντίνος Αναγνωστόπουλος
Επ.Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2010

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΕΦΑΡΜΟΣΜΕΝΩΝ ΜΑΘΗΜΑΤΙΚΩΝ ΚΑΙ ΦΥΣΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΟΜΕΑΣ ΦΥΣΙΚΗΣ

Προσομοιώσεις Monte Carlo σε GPU

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Δημητρίου Ν. Καρκούλη

Επιβλέπων: Κωνσταντίνος Αναγνωστόπουλος
Επ.Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 15η Ιουλίου 2010.

.....	.....	.....
Κ. Αναγνωστόπουλος Επ.Καθηγητής Ε.Μ.Π.	Ν. Τράκας Αν.Καθηγητής Ε.Μ.Π.	Κ. Φαράκος Αν.Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2010

.....

Δημήτριος Ν. Καρκούλης
Διπλωματούχος Σχολής Εφαρμοσμένων Μαθηματικών και Φυσικών Επιστημών Ε.Μ.Π.

Περιεχόμενα

Πρόλογος

Οι μέθοδοι Monte Carlo είναι ένας πολύ γενικός όρος που αναφέρεται σε στοχαστικές τεχνικές, δηλαδή σε αλγορίθμους που βασίζονται σε τυχαίες δειγματοληψίες για να συγκλίνουν σε μία λύση. Έχουν ιδιαίτερα μεγάλο εύρος εφαρμογών και αποτελούν σημαντικό εργαλείο για την Υπολογιστική Φυσική αλλά και για πολλούς άλλους τομείς. Η ιδέα πίσω από αυτές τις τεχνικές γεννήθηκε στο πρώτο μισό του 20ου αιώνα από τους *Stanislaw Ulam* και *Nicolas Metropolis* αν και η χρήση αυτών των μεθόδων σε αυτή την περίοδο ήταν πολύ περιορισμένη λόγω της έλλειψης υπολογιστικής ισχύος.

Τα τελευταία τρία χρόνια έχει σημειωθεί μία έκρηξη στην χρήση καρτών γραφικών (GPU[1]) για την επιτάχυνση αλγορίθμων γενικής χρήσης. Αυτή η στροφή προς τις κάρτες γραφικών είναι ιδιαίτερα έντονη στον κλάδο του επιστημονικού λογισμικού, αλλά πλέον επεκτείνεται σε πολλούς τομείς. Η ιδέα της γενικής χρήσης των καρτών γραφικών ξεκίνησε από τον *Mark Harris* το 2002 με τον όρο GPGPU[2] αλλά η έκρηξη σημειώθηκε το 2007 με την αρχιτεκτονική καρτών γραφικών CUDA[3]. Ήδη από το 2003 και μετά οι κάρτες γραφικών, όντας εξειδικευμένες στην μαζικά παράλληλη εκτέλεση compute-intensive υπολογισμών, είχαν προσπεράσει κατά πολύ τους κεντρικούς επεξεργαστές σε ωμή ταχύτητα[4], ενώ παράλληλα ωθούμενες από την μεγάλη ζήτηση για πιο ρεαλιστικά γραφικά συνέχισαν αυτή την ραγδαία ανάπτυξη παραμένοντας σε προσιτές τιμές. Αυτό το γεγονός έλκυσε το έντονο ενδιαφέρον της επιστημονικής κοινότητας η οποία πάντα είχε μεγάλες απαιτήσεις σε υπολογιστική ισχύ. Οι μέθοδοι Monte Carlo είναι γενικά παραληλλοποιήσημοι, γεγονός που τους καθιστά ιδανικούς για εκτέλεση σε κάρτες γραφικών.

[1] *Graphics Processing Unit*
[2] *General-Purpose computation on Graphics Processing Units*
[3] *Compute Unified Device Architecture*
[4] FLOPs - *Floating point operations per second*

Το πρότυπο Ising αποτελεί ένα μαθηματικό μοντέλο του σιδηρομαγνητισμού από τον *Ernst Ising*, ο οποίος έλυσε και την μονοδιάστατη περίπτωση του το 1925. Στην μονοδιάσταση περίπτωση όμως δεν παρατηρείται μετάβαση φάσης και λανθασμένα ο *Ernst Ising* συμπέρανε ότι κάτι τέτοιο θα συμβαίνει και σε ανώτερες διαστάσεις. Το 1944 όμως διαψεύστηκε από τον *Lars Onsager* ο οποίος ήταν και ο πρώτος ο οποίος έλυσε το δισδιάστατο πρότυπο με απουσία εξωτερικού μαγνητικού πεδίου και έδειξε ότι παρατηρείται μετάβαση φάσης. Δυο βασικοί αλγόριθμοι για το πρότυπο Ising είναι ο αλγόριθμος Metropolis και ο αλγόριθμος Wolff. Ο αλγόριθμος Metropolis, που πήρε το όνομα του από τον βασικό εφευρέτη του τον *Nicolas Metropolis*, είναι ένας αλγόριθμος με δυναμική single-spin-flip και γενική χρήση. Αντίθετα, ο αλγόριθμος Wolff, που αντίστοιχα ονομάστηκε από τον *Ulli Wolff*, είναι ένας cluster αλγόριθμος με εξειδικευμένη εφαρμογή στο πρότυπο Ising.

Σε αυτή την εργασία θα μας απασχολήσει η ανάπτυξη μίας επιταχυνμένης έκδοσης σε κάρτα γραφικών των αλγορίθμων Metropolis και Wolff για το δισδιάστατο πρότυπο Ising με κοντινές αλληλεπιδράσεις απουσία εξωτερικού μαγνητικού πεδίου. Αυτό το πρότυπο αποτελεί ένα πολύ καλά μελετημένο πρότυπο με γνωστή αναλυτική λύση. Επομένως, θα επικεντρωθούμε στην ανάπτυξη των αλγορίθμων και στην σύγκριση τους με τα αποτελέσματα ενός προγράμματος αναφοράς του οποίου η καλή λειτουργία είναι γνωστή. Στην συνέχεια, θα μελετήσουμε την συμπεριφορά της απόδοσης της κάρτας γραφικών και θα προσπαθήσουμε να την αιτιολογήσουμε βάσει των όσων αναπτύξουμε περί της αρχιτεκτονικής CUDA. Αν μη τι άλλο, από προγραμματιστική άποψη, η μεγαλύτερη διαφορά προγραμματισμού σε κεντρικό επεξεργαστή με τον προγραμματισμό σε κάρτα γραφικών, είναι ότι για την δεύτερη απαιτείται πολύ καλή γνώση της αρχιτεκτονικής και των διεργασιών που εκτελούνται στο επίπεδο του hardware.

Η υλοποίηση και η μελέτη των νέων αλγορίθμων κατέστει δυνατή χάρη στην πολύτιμη βοήθεια και θερμή υποστήριξη του καθηγητή Κωνσταντίνου Αναγνωστόπουλου τον οποίο ευχαριστώ θερμά, καθώς και στην υπομονή του κατά την μελέτη των αποτελεσμάτων η οποία οδήγησε στην εύρεση και διόρθωση πολλών λαθών. Μέσω των μαθημάτων του ανακάλυψα το πάθος μου για την υπολογιστική φυσική και τον επιστημονικό προγραμματισμό γενικότερα. Μέσω του συγγράμματος του για την Υπολογιστική Φυσική ΙΙ αλλά και την επίβλεψη της εργασίας, αποκτήθηκε πολύτιμη γνώση γύρω από το πρότυπο και τους αλγορίθμους που χρησιμοποιήθηκαν, τα οποία ακολουθήθηκαν ευλαβικά κατα την συγγραφή των πρώτων δύο κεφαλαίων. Επίσης, ευχαριστώ θερμά τους Dr. Claudio Ferrero και Dr. Alessandro Mirone, της μονάδας Ανάλυσης Δεδομένων[5] του Ευρωπαϊκού κέντρου ακτινοβο-

[5]DAU - *Data Analysis Unit*. Μονάδα στελεχωμένη από επιστήμονες από διάφορους επιστημονικούς τομείς, προσανατολισμένη στο επιστημονικό software και την μοντελλοποίηση. Είναι άρρηκτα συνδεδεμένη με τον πειραματικό και θεωρητικό τομέα του ιδρύμα-

λίας σύγχνοτρον[6], για την διάθεση του υπολογιστή GPU Tesla των εγκαταστάσεων του ιδρύματος, την υποστήριξή τους, αλλά και το εις βάθος know-how γύρω από τις κάρτες γραφικών που αποκτήθηκε κατά την διάρκεια της συνεργασίας μας. Τέλος, θα ήθελα να ευχαριστήσω το Ε.Μ.Π. και την Σχολή Εφαρμοσμένων Μαθηματικών και Φυσικών Επιστημών για το ευρύ φάσμα γνώσεων που μου παρείχε το οποίο βοήθησε στην ανάπτυξη μίας συνολικής εικόνας, κριτικής σκέψης και την ικανότητα να εμβαθύνω στον τομέα που με ενδιαφέρει περισσότερο.

τος. Μερικά εξέχοντα προγράμματα της μονάδας (μερικά σε συνεργασία με τους προηγούμενους τομείς και άλλα ιδρύματα) είναι το pyMCA για την online οπτικοποίηση και ανάλυση πειραματικών δεδομένων, το bigDFT gpu accelerated αλγόριθμος για Density Function Theory, το Shadow πρόγραμμα μοντελλοποίησης κβαντικής οπτικής (ray-tracing) για ακτίνες X αλλά και πρόσφατα και για νετρόνια και πολλά άλλα.

[6]ESRF - *European Synchrotron Radiation Facility*

Κεφάλαιο 1

Το πρότυπο Ising

1.1 Στατιστική Φυσική

Η Στατιστική Φυσική έχει σαν σκοπό την περιγραφή συστημάτων με πολύ μεγάλο αριθμό βαθμών ελευθερίας N. Για τα συστήματα αυτά οι εξισώσεις που περιγράφουν μικροσκοπικά το σύστημα είναι πρακτικά αδύνατον να λυθούν. Εν τέλει δεν είναι απαραίτητο να λυθούν αφού μερικές σωστά ορισμένες χονδροειδείς ιδιότητες (bulk properties) του συστήματος, αρκούν για να μας δώσουν τις χρήσιμες φυσικές πληροφορίες για το σύστημα[1].

1.1.1 Περιγραφή συστήματος σε επαφή με δεξαμενή θερμότητας

Υποθέτουμε ότι το σύστημα μας περιγράφεται από διακριτές καταστάσεις που μπορούν να απαριθμηθούν μέσα σε ένα σύνολο μ με αντίστοιχες ενέργειες $E_0 < E_1 < ... < E_\mu < ...$. Το σύστημα αυτό είναι σε επαφή με μεγάλη δεξαμενή θερμότητας θερμοκρασίας $\beta = 1/kT$ με το οποίο μπορεί να αλληλεπιδρά. Η επαφή με τη δεξαμενή έχει σαν αποτέλεσμα να υπάρχουν τυχαίες μεταβάσεις του συστήματος με τρόπο που προσδιορίζεται από την δυναμική του συστήματος. Οι θεμελιώδεις ποσότητες που μας ενδιαφέρουν είναι τα βάρη $w_\mu(t)$ που δίνουν την πιθανότητα να είναι το σύστημα στην κατάσταση μ την χρονική στιγμή t. Έστω ότι $R(\mu \to \nu)$ δίνουν τον ρυθμό μετάβασης $\mu \to \nu$. Τώρα μπορούμε να γράψουμε την πολύ γενική δεσπόζουσα εξίσωση:

$$\frac{dw_\mu(t)}{dt} = \sum_\nu \{w_\nu(t)R(\nu \to \mu) - w_\mu(t)R(\mu \to \nu)\} \tag{1.1}$$

$$\sum_\mu w_\mu(t) = 1 \tag{1.2}$$

Η πρώτη εξίσωση μας λέει ότι η μεταβολή του βάρος $w_\mu(t)$ είναι ίση με το ρυθμό που το σύστημα εισέρχεται στην κατάσταση μ από οποιαδήποτε άλλη ν μείον το ρυθμό με τον οποίο φεύγει από την κατάσταση μ Η δεύτερη εκφράζει ότι τα βάρη $w_\mu(t)$ ερμηνεύονται ως πιθανότητα και η πιθανότητα το σύστημα να είναι σε κάποια κατάσταση είναι ίση με 1. Αυτοί οι ρυθμοί μετάβασης προσομοιώνονται με κατάλληλες επιλογές κατά τη διάρκεια των υπολογισμών Μόντε Κάρλο. Για μεγάλα συστήματα και σε άπειρο χρόνο τα $w(t)$ συγκλίνουν σε αριθμούς p_μ, τις πιθανότητες κατάληψης ισορροπίας. Αυτές οι πιθανότητες για σύστημα σε θερμική ισορροπία με δεξαμενή θερμοκρασίας $\beta = 1/kT$, όπου k η σταθερά Boltzmann, ακολουθούν την κατανομή Boltzmann.

$$p_\mu = \frac{1}{Z} e^{-\beta E_\mu} \tag{1.3}$$

Η παράμετρος β αναφέρεται απλά ως θερμοκρασία του συστήματος και καθορίζει μία χαρακτηριστική ενέργεια για το σύστημα.

Η σταθερά Z είναι η συνάρτηση επιμερισμού του συστήματος και είναι η σταθερά κανονικοποίησης της κατανομής p_μ. Η απαίτηση $\sum_\mu p_\mu = 1$ μας δίνει

$$Z(\beta) = \sum_\mu e^{\beta E_\mu} \tag{1.4}$$

1.1.2 Μέσες τιμές

Για συστήματα με μεγάλο αριθμό βαθμών ελευθερίας κανείς ενδιαφέρεται για τη μέση τιμή μίας ποσότητας (observable) και η πιθανότητα κανείς να μετρήσει μία τιμή που να διαφέρει σημαντικά είναι αμελητέα.

Η μέση τιμή μίας φυσικής ποσότητας Q με τιμή Q_μ στην κατάσταση μ θα είναι

$$\langle Q \rangle = \sum_\mu p_\mu Q_\mu = \frac{1}{Z} \sum Q_\mu e^{-\beta E_\mu} \tag{1.5}$$

και η τυπική απόκλιση θα είναι τέτοια ώστε[1]

$$\frac{\Delta Q}{Q} \sim \frac{1}{\sqrt{N}} \tag{1.6}$$

Η εσωτερική ενέργεια U θα είναι

$$\langle U \rangle \equiv \langle E \rangle = \frac{1}{Z} \sum_\mu E_\mu e^{-\beta E_\mu} = -\frac{1}{Z} \frac{\vartheta}{\vartheta \beta} \sum_\mu e^{-\beta E_\mu} = -\frac{1}{Z} \frac{\vartheta Z}{\vartheta \beta} = -\frac{\vartheta log Z}{\vartheta \beta} \tag{1.7}$$

[1] $\frac{\Delta E}{E^\star} \sim \frac{\frac{\sqrt{N}}{\beta}}{\frac{N}{\beta}} = \frac{1}{\sqrt{N}}$[1].

Η μέση τιμή του τετραγώνου της εσωτερικής ενέργειας U θα είναι

$$\langle E^2\rangle = \frac{1}{Z}\sum_{\mu} E_{\mu}^2 e^{-\beta E_{\mu}} = \frac{1}{Z}\frac{\vartheta^2}{\vartheta\beta^2}\sum_{\mu} e^{-\beta E_{\mu}} = \frac{1}{Z}\frac{\vartheta^2 Z}{\vartheta\beta^2} \tag{1.8}$$

Από αυτές τις δύο εξισώσεις βρίσκουμε την διακύμανση

$$(\Delta E)^2 = \langle E^2\rangle - \langle E\rangle^2 = \frac{1}{Z}\frac{\vartheta^2 Z}{\vartheta\beta^2} - \left(-\frac{1}{Z}\frac{\vartheta Z}{\vartheta\beta}\right)^2 = \frac{\partial^2 logZ}{\partial^2\beta^2} \tag{1.9}$$

Με τον ίδιο τρόπο βρίσκουμε και την ειδική θερμότητα

$$C = \frac{\partial U}{\partial T} = \frac{\partial\beta}{\partial T}\frac{\partial U}{\partial\beta} = (-k\beta^2)(-\frac{\partial^2 lnZ}{\partial\beta^2}) = k\beta^2\frac{\partial^2 lnZ}{\partial\beta^2} = k\beta^2(\langle E^2\rangle - \langle E\rangle^2) \tag{1.10}$$

Βλέπουμε ότι η ειδική θερμότητα συνδέεται άμεσα με τις μικροσκοπικές στατιστικές διακυμάνσεις της ενέργειας του συστήματος. Αυτό είναι γνωστό σαν το θεώρημα γραμμικής απόκρισης. Αντίστοιχα μπορούμε να υπολογίσουμε και άλλες συζυγείς ποσότητες, όπως η μαγνήτιση M και η μαγνητική επιδεκτικότητα χ. Η Hamiltonian μαγνητικού συστήματος μέσα σε ομογενές μαγνητικό πεδίο B με μαγνήτιση M_{μ} στην κατάσταση μ είναι

$$H = E - BM \tag{1.11}$$

και η συνάρτηση επιμερισμού

$$Z = \sum_{\mu} e^{-\beta E_{\mu} + \beta B M_{\mu}} \tag{1.12}$$

Η μέση τιμή της μαγνήτισης είναι

$$\langle M\rangle = \frac{1}{Z}\sum_{\mu} M_{\mu} e^{-\beta M_{\mu} + \beta B M_{\mu}} = \frac{1}{\beta Z}\frac{\partial Z}{\partial B} = -\frac{\partial F}{\partial B} \tag{1.13}$$

Η μέση τιμή του τετραγώνου της

$$\langle M^2\rangle = \frac{1}{Z}\sum_{\mu} M_{\mu}^2 e^{-\beta M_{\mu} + \beta B M_{\mu}} = \frac{1}{\beta^2 Z}\frac{\partial^2 Z}{\partial B^2} \tag{1.14}$$

Με αντικατάσταση των προηγουμένων σχέσεων η διακύμανση θα είναι

$$(\Delta M)^2 = \langle M^2\rangle - \langle M\rangle^2 = \frac{1}{\beta^2}\left\{\frac{1}{Z}\frac{\partial^2 Z}{\partial B^2} - \frac{1}{Z^2}\left(\frac{\partial Z}{\partial B}\right)\right\} = \frac{1}{\beta^2}\frac{\partial^2 lnZ}{\partial B^2} = \frac{1}{\beta}\frac{\partial\langle M\rangle}{\partial B} \tag{1.15}$$

Τέλος βρίσκουμε την μαγνητική επιδεκτικότητα

$$\chi = \frac{1}{N}\frac{\partial\langle M\rangle}{\partial B} = \frac{\beta}{N}(\Delta M)^2 \tag{1.16}$$

1.1.3 Συναρτήσεις συσχετισμού

Μπορούμε να ορίσουμε την επιδεκτικότητα και τοπικά, θεωρώντας μαγνητικά πεδία τα οποία έχουν τιμή που εξαρτάται από την θέση στον χώρο που θεωρούμε μέσα στο σύστημα. Τότε θα έχουμε μία συνάρτηση συσχετισμού. Αν το σύστημα βρίσκεται μέσα σε έναν τέτοιο χώρο στον οποίο οι δυνατές θέσεις είναι οι διακριτές θέσεις ενός πλέγματος, τότε το μαγνητικό πεδίο θα είναι συνάρτηση της θέσης στο πλέγμα B_i και αλληλεπιδρά με το σπιν s_i

$$H = E - \sum_i B_i s_i \tag{1.17}$$

η μαγνήτιση $m_i \equiv s_i$ στην πλεγματική θέση i είναι

$$\langle s_i \rangle = \frac{1}{\beta} \frac{\partial ln Z}{\partial B_i} \tag{1.18}$$

και η συνάρτηση συσχετιμού δύο σημείων (connected two point correlation function) ορίζεται ως

$$G_c^{(2)}(i,j) = \langle (s_i - \langle s_i \rangle)(s_j - \langle s_j \rangle) \rangle = \langle s_i s_j \rangle - \langle s_i \rangle \langle s_j \rangle = \frac{1}{\beta^2} \frac{\partial^2 ln Z}{\partial B_i \partial B_j} \tag{1.19}$$

Η συνάρτηση συσχετισμού μας δίνει το βαθμό συσχετισμού των s_i s_j. Όταν αυτές είναι συσχετισμένες, δηλαδή, μεταβάλλονται μαζί, παίρνει μεγάλες θετικές τιμές. Όταν είναι ασυσχέτιστες, δηλαδή, η τιμή της μίας εξαρτάται ελάχιστα από την άλλη, η συνάρτηση θα έχει τιμή σχεδόν μηδέν.

1.2 Δισδιάστατο πρότυπο Ising

Μία κλασσική προσέγγιση μίας μαγνητικής ορμής περιγράφεται από ένα Ising spin το οποίο μπορεί να πάρει μόνο δύο τιμές

$$s_i = \begin{cases} +1, & \text{spin πάνω} \\ -1, & \text{spin κάτω} \end{cases} \tag{1.20}$$

Ας θεωρήσουμε τώρα τετραγωνικό πλέγμα με N πλεγματικές θέσεις έτσι ώστε $N = L * L = L^d$ με $d = 2$ για το δισδιάστατο πρότυπο. Κάθε πλεγματική θέση θα αντιστοιχεί σε ένα Ising spin. Η τοπολογία καθορίζεται από τις σχέσεις γειτονίας και ειδικότερα αυτές στα σύνορα του πλέγματος (συνοριακές συνθήκες). Η δυναμική του συστήματος καθορίζεται από τη μαγνητική αλληλεπίδραση μεταξύ των spin. Η αλληλεπίδραση αυτή εξασθενεί σαν r^{-3}, επομένως μπορούμε να θεωρήσουμε ότι αλληλεπιδρούν μόνο οι τέσσερις πλησιέστεροι γείτονες[2]. Το σύστημα μπορεί

[2]Υπάρχουν και εκδοχές του μοντέλου με περαιτέρω αλληλεπιδράσεις, όπως των αμέσως μετά 4 πλησιέστερων γειτόνων. Αυτοί είναι τα spin στις διαγώνιες πλεγματικές θέσεις.

να βρίσκεται υπό την επίδραση ομογενούς μαγνητικού πεδίου B του οποίου η διεύθυνση θεωρείται παράλληλη ή αντιπαράλληλη με αυτή των spin.

Η Hamiltonian ενός τέτοιου συστήματος είναι

$$H = -J \sum_{(i,j)} s_i s_j - B \sum_i s_i \tag{1.21}$$

Ο όρος J είναι η δύναμη αλληλεπίδρασης των spin για την οποία υπάρχουν οι εξής περιπτώσεις

- $J > 0$, το σύστημα είναι σιδηρομαγνητικό
- $J < 0$, το σύστημα είναι αντισιδηρομαγνητικό
- $J = 0$, τα spin δεν αλληλεπιδρούν

Στο σιδηρομαγνητικό μοντέλο, που και αυτό που μας ενδιαφέρει εδώ, η ενέργεια ενός δεσμού ελαχιστοποιείται όταν τα spin είναι ομόρροπα. Τότε θα έχει ενέργεια $-J$. Το σύστημα ενεργειακά προτιμά καταστάσεις με ομόρροπους δεσμούς και η ελάχιστη ενέργεια αντιστοιχεί σε εκείνη την κατάσταση όπου όλα τα spin έχουν την κατεύθυνση του μαγνητικού πεδίου[3] B, την θεμελιώδη κατάσταση. Η ενέργεια της είναι

$$E_0 = -JN - BN = -(2J + B)N \tag{1.22}$$

και σε περίπτωση όπου απουσιάζει το μαγνητικό πεδίο B η ενέργεια σε τετραγωνικό πλέγμα θα είναι

$$E_0 = -\sum_{i,j} (S_{i,j} S_{i,j+1} + S_{i,j} S_{i+1,j}) \tag{1.23}$$

Η συνάρτηση επιμερισμού είναι

$$Z = \sum_{s_1 \pm 1} \sum_{s_2 \pm 1} \dots \sum_{s_N \pm 1} e^{-\beta H[\{s_i\}]} \equiv \sum_{\{s_i\}} e^{\beta J \sum_{i,j} s_i s_j + \beta B \sum_i s_i} \tag{1.24}$$

όπου $\{s_i\}$ είναι μία διάταξη των spin στο πλέγμα.

1.2.1 Ιδιότητες

Στο δισδιάστατο πρότυπο Ising με τετραγωνικό πλέγμα[4] το σύστημα παρουσιάζει μετάβαση φάσης δευτέρας τάξης[5] για $T = T_c$ ή ισοδύναμα $\beta = \beta_c$. Η μετάβαση

[3] Με το μαγνητικό πεδίο να έχει θεωρηθεί ότι μπορεί να είναι μόνο παράλληλο ή αντιπαράλληλο με τα spin.

[4] Σε άλλους τύπους πλεγμάτων, που θα δούμε πιο μετά, ισχύουν τα παρακάτω αλλά η τιμή θα είναι διαφορετική.

[5] Στο μονοδιάστατο μοντέλο δεν παρατηρείται καμία μετάβαση φάσης.

είναι δευτέρας τάξης, επειδή η παράμετρος τάξης είναι συνεχής συνάρτηση της θερμοκρασίας Σύμφωνα με την λύση του Onsager

$$T_c = \frac{2}{log(1+\sqrt{2})} \approx 2,27 \tag{1.25}$$

$$\beta_c = 1/k_\beta T = \frac{1}{2} ln\left(1+\sqrt{2}\right) \approx 0.4406867935\ldots \tag{1.26}$$

Το σύστημα, δηλαδή, μεταβαίνει από κατάσταση τάξης σε κατάσταση αταξίας. Η παράμετρος τάξης στο πρότυπο Ising είναι η μέση τιμή της απόλυτης μαγνήτισης (σχέση 1.27), όπου για $(\langle|M|\rangle > 0)$ έχουμε τάξη στο σύστημα αφού θα είναι μαγνητισμένο (σιδηρομαγνητική φάση) ενώ για $(\langle|M|\rangle = 0)$ το σύστημα είναι σε πλήρη αταξία αφού δεν παρουσιάζει μαγνήτιση (παραμαγνητική φάση). Η θερμοκρασία T_c ονομάζεται κρίσιμη θερμοκρασία ή θερμοκρασία Curie[6].

$$\langle|M|\rangle = \left\langle \sum_i |s_i| \right\rangle = \frac{1}{N} \sum_{i=1}^{N} |s_i| \tag{1.27}$$

Για $\beta \neq \beta_c$ η συνάρτηση συσχετισμού έχει πεπερασμένο μήκος ενώ όσο πλησιάζουμε στην κρίσιμη περιοχή[7] τείνει στο άπειρο. Συγκεκριμένα συμπεριφέρεται ασυμπτωτικά σας

$$\xi(\beta) \equiv \xi(t) \sim |t|^{-\nu} \tag{1.28}$$

όπου $t = \dfrac{\beta_c - \beta}{\beta_c}$ η ανηγμένη θερμοκρασία. Αντίστοιχα για την συνάρτηση συσχετισμού και τα υπόλοιπα μεγέθη

$$G_c^{(2)}(i,j) \sim \frac{1}{|x_{ij}|^\eta} \tag{1.29}$$

$$M \sim |t|^\beta \tag{1.30}$$

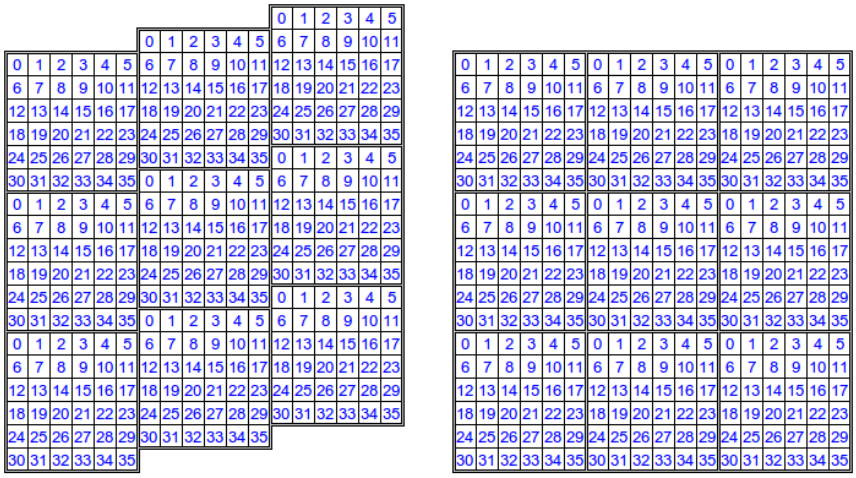
$$C \sim |t|^{-\alpha} \tag{1.31}$$

$$\chi \sim |t|^\gamma \tag{1.32}$$

Οι εκθέτες αυτοί ονομάζονται κρίσιμοι εκθέτες και οι τιμές τους παρουσιάζουν την ιδιότητα της παγκοσμιότητας, δηλαδή οι τιμές τους δεν εξαρτώνται από τις λεπτομέρειες του πλέγματος ή της αλληλεπίδρασης.

[6] Η θερμοκρασία ή το σημείο Curie ενός σιδηρομαγνητικού υλικού, είναι το αναστρέψιμο σημείο πέραν του οποίου το υλικό γίνεται παραμαγνητικό.

[7] Κρίσιμη περιοχή ονομάζεται η περιοχή γύρω από το κρίσιμο σημείο.



(α□) Ελικοειδείς συνοριακές συνθήκες

(β□) Περιοδικές συνοριακές συνθήκες

Σχήμα 1.1: Συνοριακές συνθήκες με τοροειδή τοπολογία

1.2.2 Πλέγματα και συνοριακές συνθήκες

Μερικοί τύποι πλεγμάτων για το δισδιάστατο πρότυπο Ising είναι το τετραγωνικό πλέγμα (square lattice), το τριγωνικό (triangular) και το εξαγωνικό (honeycomp). Οι κρίσιμοι εκθέτες θα είναι οι ίδιοι σε όλους τους τύπους λόγω παγκοσμιότητας. Για το δισδιάστατο τετραγωνικό η θερμοκρασία Curie είναι $T_c \approx 2,27$, για το τριγωνικό $T_c \approx 3.64$[4] και για το εξαγωνικό $T_c \approx 1.52$[4].

Για την προσομείωση απείρου πλέγματος συνήθως χρησιμοποιούνται συνοριακές συνθήκες με τοροειδή τοπολογία όπως οι περιοδικές και οι ελικοειδής. Αντίστοιχα κάποιος θα μπορούσε να χρησιμοποιήσει συνοριακές συνθήκες με κυλινδρική τοπολογία ή με ελεύθερα όλα τα σύνορα, αλλά σε αυτή την περίπτωση, τα spin των συνόρων δεν θα δίνουν σωστή στατιστική. Στο σχήμα 1.1α□ φαίνονται οι ελικοειδής συνοριακές συνθήκες ενώ στο σχήμα 1.1β□ οι περιοδικές.

Οι ελικοειδής συνοριακές συνθήκες είναι μία παραλλαγή των περιοδικών και επιφέρουν ένα μικρό επιπρόσθετο σφάλμα. Το σφάλμα όμως αυτό μπορεί να γίνει ορατό μόνο σε πολύ μικρά πλέγματα. Σε μεγάλα πλέγματα επικαλύπτεται από το στατιστικό. Χρησιμοποιούνται καθώς είναι πιο γρήγορες. Αυτό συμβαίνει πρώτον επειδή ένα spin $s_{(i,j)}$ και οι γείτονες του $s_{(i-1,j)}$ και $s_{(i+1,j)}$ θα είναι πάντα σε ένα συνεχές κομμάτι της (γραμμικής) μνήμης και δεύτερον και πιο σημαντικό, δεν απαιτείται η χρήση ”ακριβών” πράξεων ακέραιας διαίρεσης και υπολοίπου (modulo)[8].

[8]Εφόσον μία πράξη modulo γίνεται πάνω σε αριθμό που είναι δύναμη του 2, αυτή μπορεί να επιταχυνθεί σημαντικά με χρήση μεταθέσεων bits[14]. Σε μερικές περιπτώσεις

Παρόλα αυτά κάποια ειδικά χαρακτηριστικά των καρτών γραφικών, όπως θα δούμε στο κεφάλαιο 3 μπορούν να εξαλείψουν αυτό το μειονέκτημα και να επιταχύνουν σημαντικά τις περιοδικές συνοριακές συνθήκες.

Εμείς θα εργαστούμε με μία παραλλαγή του κλασσικού τετραγωνικού πλέγματος. Το τετραγωνικό πλέγμα σκακιέρα (checkerboard). Ένα τετραγωνικό πλέγμα αποτελείται από δύο υποπλέγματα τα οποία περιέχουν πλεγματικές θέσεις spin οι οποίες δεν αλληλεπιδρούν μεταξύ τους. Αυτή η ανεξαρτησία των δύο υποπλεγμάτων μας επιτρέπει να ανανεώσουμε τα spin του πρώτου υποπλέγματος παράλληλα και στην συνέχεια να χρησιμοποιήσουμε το configuration του για την ανανέωση του δεύτερου. Θα δούμε αυτό το πλέγμα πιο αναλυτικά όταν έρθει η ώρα να αναλύσουμε τον αλγόριθμο σε GPU.

ο μεταγλωττιστής κάνει αυτή την βελτιστοποίηση για μας.

Κεφάλαιο 2

Προσομοιώσεις Monte Carlo

2.1 Η μέθοδος Monte Carlo

Η μέθοδος Monte Carlo αποτελεί την πιο διαδεδομένη μέθοδο στατιστικής δειγματοληψίας, λόγω της αποτελεσματικότητας της και της γενικής της εφαρμογής. Η μέθοδος Monte Carlo είναι ιδιαίτερα χρήσιμη στην μελέτη συστημάτων με μεγάλο βαθμό ελευθερίας. Στην περίπτωση του προτύπου Ising είναι η μόνη γνωστή μέθοδος για τον υπολογισμό της συνάρτησης επιμερισμού[2]. Η βασική ιδέα πίσω από τις προσομοιώσεις Monte Carlo είναι η προσομοίωση των τυχαίων θερμικών διακυμάνσεων του συστήματος, από κατάσταση σε κατάσταση κατά την διάρκεια της προσομοίωσης.

2.1.1 Δειγματοληψία - Εκτιμητής

Ο κύριος στόχος σε μία προσομείωση Monte Carlo σε θερμικό σύστημα είναι ο προσδιορισμός της αναμενόμενης τιμής $\langle Q \rangle$ μίας φυσικής ποσότητας Q. Ο ιδανικός τρόπος να προσδιορίσουμε την αναμενόμενη τιμή είναι να πάρουμε την μέση τιμή της ποσότητας από όλες τις καταστάσεις μ του συστήματος, λαμβάνοντας υπόψη το βάρος της πιθανότητας Boltzmann για την κάθε μία

$$\langle Q \rangle = \sum_{\mu} p_{\mu} Q_{\mu} = \frac{\sum_{\mu} Q_{\mu} e^{-\beta E_{\mu}}}{\sum_{\mu} e^{-\beta E_{\mu}}} \tag{2.1}$$

που όμως δεν είναι εύκολο για μεγάλα συστήματα. Σε αυτή την περίπτωση το καλύτερο που μπορούμε να κάνουμε είναι να πάρουμε την μέση τιμή μόνο από ένα δείγμα των καταστάσεων. Επιλέγουμε δείγμα από M καταστάσεις $\{\mu_1, \mu_2, \ldots, \mu_M\}$ οι οποίες κατανέμονται σύμφωνα με την κατανομή πιθανότητας P_{μ} και ορίζουμε τον εκτιμητή της ποσότητας Q ως

$$Q_{\mu} = \frac{\sum_{i=1}^{M} Q_{\mu_i} P_{\mu_i}^{-1} e^{-\beta E_{\mu_i}}}{\sum_{j=1}^{M} P_{\mu_j}^{-1} e^{-\beta E_{\mu_j}}} \tag{2.2}$$

Ο εκτιμητής αυτός έχει την ιδιότητα να βελτιώνεται η εκτίμηση της $\langle Q \rangle$ καθώς το πλήθος των δειγμάτων M αυξάνεται και για $M \to \infty$

$$\langle Q \rangle = \lim_{M \to \infty} Q_M \tag{2.3}$$

Με κατάλληλη επιλογή της κατανομής P_μ η σύγκλιση θα γίνεται γρήγορα.

Απλή Δειγματοληψία

Στην περίπτωση της απλής δειγματοληψίας, διαλέγουμε $P_\mu = \textit{σταθ.}$ επομένως ο εκτιμητής τώρα θα είναι

$$Q_\mu = \frac{\sum_{i=1}^{M} Q_{\mu_i} e^{-\beta E_{\mu_i}}}{\sum_{j=1}^{M} P_{\mu_j}^{-1} e^{-\beta E_{\mu_j}}} \tag{2.4}$$

και είναι η πιο απλή επιλογή που μπορούμε να κάνουμε. Είναι όμως και πολύ κακή επιλογή. Είναι δυνατό να δειγματίσουμε μόνο ένα μικρό υποσύνολο των καταστάσεων του συστήματος. Ο λόγος που είναι κακή επιλογή είναι ότι ένα από τα αθροίσματα ή και τα δύο μπορούν να κυριαρχούνται από έναν μικρό αριθμό καταστάσεων, με όλες τις άλλες καταστάσεις να συνεισφέρουν ελάχιστα Αυτό το φαινόμενο είναι ιδιαίτερα αισθητό στις χαμηλές θερμοκρασίες, όπου στα αθροίσματα μπορούν να κυριαρχούν μόνο λίγες καταστάσεις, ακόμα και μόνο μία, επειδή δεν υπάρχει αρκετή θερμική ενέργεια με αποτέλεσμα το σύστημα να μένει κυρίως στάσιμο στην θεμελιώδη κατάσταση.

Δειγματοληψία με κριτήριο σημαντικότητας

Αντί να πάρουμε τις καταστάσεις M έτσι ώστε κάθε κατάσταση του συστήματος να είναι ισοπίθανο να επιλεχθεί, τις επιλέγουμε έτσι ώστε η πιθανότητα μία κατάσταση μ να επιλεγεί είναι $p_\mu = \dfrac{e^{-\beta E_\mu}}{Z}$. Ο εκτιμητής της $\langle Q \rangle$ θα γίνει

$$Q_M = \frac{\sum_{i=1}^{M} Q_{\mu_i} \left(e^{-\beta E_{\mu_i}}\right)^{-1} e^{-\beta E_{\mu_i}}}{\sum_{i=1}^{M} \left(e^{-\beta E_{\mu_i}}\right)^{-1} e^{-\beta E_{\mu_i}}} = \frac{1}{M} \sum_{i=1}^{M} Q_{\mu_i} \tag{2.5}$$

Αυτή η δειγματοληψία είναι πολύ καλύτερη από την απλή αφού η σχετική συχνότητα επιλογής μίας κατάστασης είναι αντίστοιχη των καταστάσεων που θα επέλεγε ένα πραγματικό σύστημα.

Διαδικασίες Markov

Η δυσκολία διεξαγωγής μίας προσομοίωσης Monte Carlo έγκειται στην παραγωγή ενός κατάλληλου τυχαίου συνόλου καταστάσεων σύμφωνα με την κατανομή

Boltzmann. Αν επιλέξουμε καταστάσεις τυχαία και τις αποδεχόμαστε ή τις απορρίπτουμε με πιθανότητα ανάλογη της $e^{-\beta E_\mu}$, τότε θα έχουμε παρόμοιο πρόβλημα με την απλή δειγματοληψία, θα απορρίπτουμε σχεδόν όλες τις καταστάσεις, καθώς η πιθανότητα αποδοχής τους θα είναι πάρα πολύ μικρή. Σχεδόν όλα τα σχήματα Monte Carlo κάνουν χρήση των διαδικασιών Markov για την παραγωγή του συνόλου των καταστάσεων. Μία διαδικασία Markov είναι ο μηχανισμός ο οποίος, αν θεωρήσουμε ένα σύστημα σε κατάσταση μ, παράγει μία νέα κατάσταση του συστήματος ν. Οι καταστάσεις παράγονται τυχαία, δεν θα παράγει την ίδια κατάσταση κάθε φορά που θα έχει αρχική κατάσταση μ. Η πιθανότητα παραγωγής μίας νέας κατάστασης ν για δεδομένο μ ονομάζεται πιθανότητα μετάβασης (transition probability) $P(\mu \to \nu)$ για την μετάβαση από μ σε ν. Για είναι μία διαδικασία Markov θα πρέπει να τηρούνται τέσσερις βασικές προϋποθέσεις για τις πιθανότητες μετάβασης:

1. Δεν πρέπει να εξαρτώνται από τον χρόνο
2. Θα πρέπει να εξαρτώνται μόνο από τις καταστάσεις των συγκεκριμένων μ και ν και όχι από άλλες καταστάσεις από τις οποίες πέρασε το σύστημα προηγουμένως.
3. Για την πιθανότητα μετάβασης $P(\mu \to \nu)$ θα πρέπει να ισχύει

$$\sum_{\nu} P(\mu \to \nu) = 1. \tag{2.6}$$

   Όμως η πιθανότητα μετάβασης $P(\mu \to \nu)$ μπορεί να είναι μεγαλύτερη του μηδενός.

4. Για $t \to \infty$ το δείγμα $\{\mu_i\}$ ακολουθεί την κατανομή P_μ.

Οι προϋποθέσεις (1) και (2) μας λένε ότι η πιθανότητα μίας διαδικασίας Markov να παράξει την κατάσταση ν από την κατάσταση μ είναι ίδια κάθε φορά που θα της δίνεται η κατάσταση μ ανεξαρτήτως του ιστορικού των μεταβάσεων κατάστασης.

Σε μία προσομοίωση Monte Carlo χρησιμοποιούμε διαδοχικές διαδικασίες Markov ώστε να παράγουμε μία αλληλουχία ή αλυσίδα Markov (Markov chain) καταστάσεων. Η διαδικασία Markov επιλέγεται έτσι, ώστε ξεκινώντας από οποιαδήποτε αρχική κατάσταση μ_0, μετά από ένα χρονικό διάστημα να μας δίνει μία αλληλουχία καταστάσεων οι οποίες να ακολουθούν την κατανομή Boltzmann. Η μεγαλύτερη προσπάθεια επικεντρώνεται στο προσδιορισμό των πιθανοτήτων μετάβασης $P(\mu \to \nu)$ ώστε η σύγκλιση (4) να επιτυγχάνεται γρήγορα. Είναι επίσης σημαντική η επιλογή αρχικής κατάστασης μ_0 τέτοιας ώστε να είναι μία τυπική κατάσταση του τελικού δείγματος, ώστε το σύστημα να βρεθεί γρήγορα σε κατάσταση ισορροπίας[1]. Ο χρόνος που απαιτείται για να φτάσει το σύστημα σε κατάσταση ισορροπίας ονομάζεται thermalization time.

[1] Δηλαδή να ακολουθεί την κατανομή Boltzmann

Κριτήριο εργοδοτικότητας

Το κριτήριο εργοδοτικότητας είναι η απαίτηση η διαδικασία Markov να μπορεί να περάσει από όλες τις καταστάσεις αν την αφήσουμε να τρέξει για το αναγκαίο χρονικό διάστημα. Αποτελεί απαραίτητη προϋπόθεση ώστε το δείγμα να ακολουθεί την ζητούμενη κατανομή.

Συνθήκη λεπτομερούς ισοζύγησης

Η συνθήκη λεπτομερούς ισοζύγησης (detailed balance) επιβάλλει ο ρυθμός μετάβασης κατάστασης $P(\mu \to \nu)$ να είναι κατά μέση τιμή ίδιος με τον ρυθμό μετάβασης κατάστασης $P(\nu \to \mu)$. Η αναγκαία συνθήκη για να συμβαίνει αυτό είναι:

$$\sum_{\nu} p_\mu P(\mu \to \nu) = \sum_{\nu} p_\nu P(\nu \to \mu) \tag{2.7}$$

και η ικανή (condition of detailed balance):

$$p_\mu P(\mu \to \nu) = p_\nu P(\nu \to \mu) \tag{2.8}$$

Όταν ικανοποιείται η τελευταία από τις πιθανότητες μετάβασης, τότε το σύστημα θα φτάσει σε κατάσταση θερμικής ισορροπίας[1] και θα ικανοποιεί την 2.7 η οποία μας λέει ότι η κατανομή στην κατάσταση ισορροπίας είναι η ζητούμενη κατανομή. Για την κατανομή Boltzmann η σχέση 2.8 γίνεται

$$\frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \frac{p_\nu}{p_\mu} = e^{-\beta(E_\nu - E_\mu)} \tag{2.9}$$

2.1.2 Λόγοι αποδοχής

Η σχέση (2.9) έχει πολλές λύσεις[1]. Έχουμε πει ότι μπορούμε να θέσουμε την πιθανότητα μετάβασης $P(\mu \to \nu)$ ως μη μηδενική, αρκεί να μην υπερβαίνει ποτέ την μονάδα. Μπορούμε, δηλαδή να θέσουμε τις πιθανότητες μετάβασης $P(\mu \to \nu)$ όπως θέλουμε αρκεί να τηρούμε την συνθήκη λεπτομερούς ισοζύγησης με μία αντίστοιχη αλλαγή στην πιθανότητα μετάβασης $P(\nu \to \mu)$. Τελικά, έχουμε αρκετή ελευθερία να κάνουμε τις πιθανότητες μετάβασης να παίρνουν όποιο σύνολο τιμών θέλουμε. Αυτό φαίνεται αμέσως άμα χωρίσουμε την πιθανότητα μετάβασης σε δύο όρους

$$P(\mu \to \nu) = g(\mu \to \nu) A(\mu \to \nu) \tag{2.10}$$

Ο όρος $g(\mu \to \nu)$ είναι η πιθανότητα επιλογής (selection probability), η οποία για μία δεδομένη αρχική κατάσταση μ είναι η πιθανότητα ο αλγόριθμος να παράγει μία νέα κατάσταση ν και ο όρος $A(\mu \to \nu)$ είναι ο λόγος αποδοχής (acceptance ratio). Ο Λόγος αποδοχής είναι η πιθανότητα να αποδεχτούμε αυτή την νέα κατάσταση ν.

Ένας ιδανικός αλγόριθμος θα είναι αυτός που θα έχει λόγο αποδοχής $A(\mu \to \nu) = 1$ για όλα τα ν για τα οποία $g(\mu \to \nu) > 0$. ([1, 2])

2.2 Ο αλγόριθμος Metropolis

Αυτό που κάνει τον αλγόριθμο Metropolis "διαφορετικό" είναι η επιλογή των λόγων αποδοχής.

Πρώτα επιλέγουμε ένα σύνολο πιθανοτήτων επιλογής $g(\mu \to \nu)$ και στην συνέχεια ένα σύνολο λόγων αποδοχής $A(\mu \to \nu)$ έτσι ώστε να ισχύει η συνθήκη λεπτομερούς ισοζύγησης. Ο αλγόριθμος λειτουργεί επιλέγοντας συνέχεια νέες καταστάσεις ν και απορρίπτοντας τες ή κάνοντας τες αποδεκτές τυχαία με βάση τους επιλεγμένους λόγους αποδοχής. Αν η νέα κατάσταση γίνει αποδεκτή, τότε το σύστημα αλλάζει σε αυτή την κατάσταση, αλλιώς παραμένει στην ίδια.

Ένα πραγματικό σύστημα ξοδεύει τον περισσότερο χρόνο του σε ένα υποσύνολο των καταστάσεων με μικρό εύρος ενεργειών. Επομένως, δεν θέλουμε να χάνουμε χρόνο με τον αλγόριθμο να λαμβάνει υπόψη μεταβάσεις καταστάσεων όπου η διαφορά ενέργειας είναι μεγάλη. Ένας απλός τρόπος να το αποφύγουμε αυτό στο πρότυπο Ising είναι να λαμβάνουμε υπόψη μόνο τις καταστάσεις όπου διαφέρουν κατά την εναλλαγή ενός spin. Τότε θα έχουμε μέγιστη διαφορά ενέργειας $2zJ$, όπου z το πλήθος των γειτόνων[2]. Ένας αλγόριθμος που το εφαρμόζει αυτό, θα έχει δυναμική single-spin-flip.

Στον αλγόριθμο Metropolis οι πιθανότητες επιλογής $g(\mu \to \nu)$ για κάθε μία από τις πιθανές καταστάσεις είναι επιλεγμένες να είναι ισοδύναμες και οι πιθανότητες επιλογής όλων των άλλων καταστάσεων επιλέγονται να είναι μηδέν. Έστω ότι έχουμε N πλεγματικές θέσεις, λόγω της δυναμικής single-spin-flip θα έχουμε N δυνατές καταστάσεις ν οι οποίες θα είναι προσβάσιμες από μία κατάσταση μ. Επομένως, κάθε πιθανότητα επιλογής είναι

$$g(\mu \to \nu) = \frac{1}{N} \tag{2.11}$$

και η συνθήκη λεπτομερούς ισοζύγησης παίρνει την μορφή

$$\frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \frac{g(\mu \to \nu) A(\mu \to \nu)}{g(\nu \to \mu) A(\nu \to \mu)} = \frac{A(\mu \to \nu)}{A(\nu \to \mu)} = e^{-\beta(E_\nu - E_\mu)} \tag{2.12}$$

[2] $z = 4$ για την περίπτωση των πλησιέστερων γειτόνων.

Θέλουμε οι λόγοι αποδοχής να είναι όσο το δυνατόν μεγαλύτεροι. Ο αλγόριθμος Metropolis χαρακτηρίζεται από την βέλτιστη επιλογή των λόγων αποδοχής

$$A(\mu \to \nu) = \begin{cases} e^{-\beta(E_\nu - E_\mu)} & E_\nu - E_\mu > 0 \\ 1 & \textit{αλλιώς} \end{cases} \tag{2.13}$$

Με λίγα λόγια, εάν επιλέξουμε μία νέα κατάσταση η οποία έχει ενέργεια μικρότερη ή ίση με την κατάσταση του συστήματος, θα αποδεχτούμε την μετάβαση σε αυτή την κατάσταση. Διαφορετικά, αν έχει μεγαλύτερη ενέργεια την κάνουμε αποδεχτή τυχαία με πιθανότητα $e^{-\beta(E_\nu - E_\mu)}$.

Θα θεωρήσουμε τον αλγόριθμο Metropolis με αλληλεπίδραση πλησιέστερων γειτόνων σε τετραγωνικό πλέγμα μήκους L έτσι ώστε $N = L * L = L^2$ ο συνολικός αριθμός πλεγματικών θέσεων. Ο συνολικός αριθμός των δεσμών μεταξύ των γειτόνων είναι $N_l = 2N$ για την περίπτωση των ελικοειδών συνθηκών. Θεωρούμε ότι το μαγνητικό πεδίο είναι μηδέν και ο όρος αλληλεπίδρασης τίθεται ίσως με την μονάδα. Επομένως, η Hamiltonian τώρα θα είναι

$$H = -\sum_{\langle i,j \rangle} s_i s_j \tag{2.14}$$

με το άθροισμα να είναι πάνω στα ζεύγη γειτονικών πλεγματικών θέσεων. Η συνάρτηση επιμερισμού είναι

$$Z = \sum_{s_1=\pm 1} \sum_{s_2=\pm 1} \dots \sum_{s_N=\pm 1} e^{-\beta H[\{s_i\}]} \equiv \sum_{\{s_i\}} e^{\beta \sum_{\langle ij \rangle} s_i s_j} \tag{2.15}$$

Για την διαδικασία Markov, έστω ότι το σύστημα μας είναι σε μία κατάσταση μ και εξετάζουμε τη μετάβαση στη νέα κατάσταση ν που προκύπτει από την αλλαγή της τιμής ενός spin $s_k^\nu = -s_k^\nu$. Η μεταβολή της ενέργειας είναι

$$E_\nu - E\mu = \left(-\sum_{\langle ij \rangle} s_i^\nu s_j^\nu \right) - \left(-\sum_{\langle ij \rangle} s_i^\mu s_j^\mu \right) = 2 s_k^\mu \left(\sum_{\langle ik \rangle} s_i^\mu \right) \tag{2.16}$$

Αν αυτό το άθροισμα είναι αρνητικό ή μηδέν, τότε δεχόμαστε την νέα κατάσταση ν διαφορετικά την δεχόμαστε με πιθανότητα $e^{-\beta(E_\nu - E_\mu}$.

Η ενέργεια ανά δεσμό θα δίνεται από

$$\langle e \rangle = \frac{1}{2N} \langle E \rangle = -\frac{1}{2N} \sum_{\langle ij \rangle} s_i s_j \tag{2.17}$$

με τιμές $-1 \leq \langle e \rangle \leq 1$. Η μαγνήτιση ανά πλεγματική θέση

$$\langle m \rangle = \frac{1}{N} \langle M \rangle = \frac{1}{N} \left| \sum_i s_i \right| \tag{2.18}$$

η οποία όπως είπαμε είναι η παράμετρος τάξης και επομένως παίρνει τιμές $0 \leq \langle m \rangle \leq 1$. Η ειδική θερμότητα

$$c = \beta^2 N \left\langle (e - \langle e \rangle)^2 \right\rangle = \beta^2 N \left(\langle e^2 \rangle - \langle e \rangle^2 \right) \tag{2.19}$$

και η μαγνητική επιδεκτικότητα

$$\chi = \beta N \left\langle (m - \langle m \rangle)^2 \right\rangle = \beta N \left(\langle m^2 \rangle - \langle m \rangle^2 \right) \tag{2.20}$$

2.2.1 Σφάλματα και μέδοθος jackknife

Τα σφάλματα σε μία προσομοίωση Monte Carlo κατηγοριοποιούνται σε συστηματικά και στατιστικά σφάλματα. Τα στατιστικά σφάλματα είναι αποτέλεσμα των τυχαίων αλλαγών στο προσομοιωμένο σύστημα κατά τις μετρήσεις (π.χ. οι θερμικές διακυμάνσεις) και μπορούν να εκτιμηθούν παίρνοντας πολλές μετρήσεις της ποσότητας που μας ενδιαφέρει και στην συνέχεια υπολογίζοντας την διασπορά των τιμών της. Τα συστηματικά σφάλματα οφείλονται στην διαδικασία που ακολουθήσαμε για να πάρουμε τις μετρήσεις και επηρεάζουν όλη την προσομοίωση. Τυπικό συστηματικό σφάλμα στην περίπτωση μας είναι ότι περιμένουμε πεπερασμένο χρόνο για να έρθει το σύστημα μας σε ισορροπία. Ένα άλλο συστηματικό σφάλμα θα μπορούσε να ήταν να μην συνεχίζαμε την προσομοίωση για αρκετά μεγάλο διάστημα μετά το σημείο ισορροπίας ώστε να πάρουμε ανεξάρτητες μετρήσεις. Επίσης λόγω του ότι χρησιμοποιούμε ένα διακριτό πρότυπο για να προσομοιώσουμε ένα συνεχές μοντέλο υπεισέρχονται σφάλματα στους εκτιμητές λόγω της διακριτοποίησης και του πεπερασμένου μεγέθους του πλέγματος[1]. Αυτά τα σφάλματα μειώνονται με την προσομοίωση μεγαλύτερων συστημάτων και την χρήση μεθόδων όπως την μέθοδο βάθμωσης πεπερασμένου μεγέθους (finite size scaling)

Η πιο δημοφιλής και κοινώς αποδεκτή μέθοδος εκτίμησης σφαλμάτων είναι η μέθοδος jackknife η οποία είναι μία μέθοδος resampling. Η βασική ιδέα είναι ο διαχωρισμός των αποτελεσμάτων n μετρήσεων της ποσότητας που μας ενδιαφέρει σε n_b καλάθια τα οποία περιέχουν από $b = n - (n/n_b)$ στοιχεία, και για κάθε φορά η αφαίρεση ενός bin και ο υπολογισμός της μέσης τιμής της ποσότητας από το υποσύνολο που απομένει. Στην συνέχεια θα το βάζουμε πίσω και θα αφαιρούμε ένα άλλο bin για να επαναλάβουμε τον ίδιο υπολογισμό. Από αυτή την διαδικασία θα προκύψουν τα σφάλματα $Q_0^b, Q_1^b, \ldots, Q_{n_b-1}^b$. Το σφάλμα θα είναι

$$(\delta Q)^2 = \sum_{j=0}^{n_b-1} \left(Q_j^b - \langle O^b \rangle \right)^2 = n_b \left(\langle (Q^b)^2 \rangle - \langle Q^b \rangle^2 \right) \tag{2.21}$$

Ο αριθμός των bins καθορίζεται από την μεταβολή τους έως ότου προσδιορίζουμε τις τιμές για τις οποίες η τιμή του σφάλματος είναι σταθερή.

2.2.2 Ανεξαρτησία μετρήσεων

Ο αλγόριθμος Metropolis, καθώς και γενικά οι αλγόριθμοι με δυναμική single-spin-flip, έχει το μειονέκτημα ότι στην κρίσιμη περιοχή είναι δύσκολο να πάρει κανείς ανεξάρτητες μετρήσεις καθώς το μήκος συσχετισμού θα αυξάνεται εκθετικά με το μέγεθος του πλέγματος σε αυτή την περιοχή.

Στον αλγόριθμο Metropolis, ένα βήμα θα είναι στατιστικά ισχυρά συσχετισμένο με το προηγούμενο, καθώς θα διαφέρει το πολύ κατά την τιμή ενός spin. Στην καλύτερη περίπτωση, θα έχουμε μία στατιστικά ανεξάρτητη διάταξη μετά από 1 βήμα Metropolis ανά πλεγματική θέση. Αυτή η περίπτωση θα ισχύει στο πρότυπο Ising μόνο για θερμοκρασίες μακριά από την κρίσιμη περιοχή. Στην κρίσιμη περιοχή θα χρειαζόμαστε όλο και περισσότερα βήματα ανά πλεγματική θέση (sweeps του πλέγματος) για να πάρουμε ανεξάρτητες μετρήσεις. Αυτό συμβαίνει επειδή το μήκος συσχετισμού γίνεται πολύ μεγαλύτερο από μία πλεγματική θέση.

Για την μελέτη του αυτοσυσχετισμού της διάταξης χρησιμοποιείται η συνάρτηση αυτοσυσχετισμού

$$\rho_Q(t) = \frac{\left\langle \Big(Q(t') - \langle Q \rangle\Big) \Big(Q(t'+t) - \langle Q \rangle\Big) \right\rangle_{t'}}{\left\langle (Q - \langle Q \rangle)^2 \right\rangle} \tag{2.22}$$

όπου Q η φυσική ποσότητα που μας ενδιαφέρει και $Q(t)$ η τιμή της μετά από χρόνο Monte Carlo t. Η μέση τιμή $\langle \ldots \rangle_{t'}$ είναι η μέση τιμή με αρχική τιμή όλα τα στοιχεία του δείγματος με $t^{'} < t_{max} - t$. Από σύμβαση θεωρούμε ότι θα έχουμε μία ανεξάρτητη μέτρηση της ποσότητας Q όταν η τιμή της $\rho_Q(t)$ θα έχει πέσει στο 14% της αρχικής της τιμής, το οποίο συμβαίνει μετά από χρόνο $2t$[1]. Αν έχουμε t_{max} μετρήσεις, ο αριθμός των ανεξάρτητων μετρήσεων θα είναι

$$n_Q = \frac{t_{max}}{2\tau_Q} \tag{2.23}$$

όπου τ_Q είναι ο χρόνος αυτοσυσχετισμού για την ποσότητα Q

Μακρυά από την κρίσιμη περιοχή ο χρόνος αυτοσυσχετισμού είναι μικρός και ανεξάρτητος του μήκους του πλέγματος. Αυτό όμως δεν ισχύει και κοντά στην κρίσιμη περιοχή όπου ο χρόνος αυτοσυσχετισμού για τον αλγόριθμο Metropolis είναι

$$\tau \sim \xi^z \tag{2.24}$$

όπου $z > 0$ και $\xi \sim L$, και το φαινόμενο αυτό ονομάζεται κρίσιμη επιβράδυνση και κάνει τις προσομοιώσεις για μεγάλο L απαγορευτικά ακριβές για αυτόν τον αλγόριθμο.

2.2.3 Υλοποίηση του αλγορίθμου

Ο αλγόριθμος κατά τα άλλα είναι πολύ απλός στην υλοποίηση του.

1. Επιλογή παραμέτρων προσομοίωσης (πλέγμα, θερμοκρασία, πλήθος sweeps, seed γεννήτορα ψευδοτυχαίων).

2. Επιλογή κατάλληλης αρχικής διάταξης (configuration) του πλέγματος (τάξη, αταξία, αποθηκευμένη διάταξη).

3. Υπολογισμός των λόγων αποδοχής.

4. Για κάθε sweep, επιλογή τυχαίας πλεγματικής θέσης (spin) $N = L \times L$ φορές.

5. Για κάθε τυχαία επιλεγμένο spin υπολογισμός της μεταβολής της ενέργειας από το flip των γειτόνων. Αν η μεταβολή είναι μικρότερη ή ίση του μηδενός η νέα κατάσταση γίνεται αποδεκτή και το spin αλλάζει προσανατολισμό, αλλιώς γίνεται αποδεκτή τυχαία με βάση τον λόγο αποδοχής για διαφορά ενέργειας μεγαλύτερη του μηδενός.

6. Στο τέλος κάθε sweep υπολογισμός της ενέργειας δεσμών και της μαγνήτισης.

7. Νέο sweep.

8. Όταν ολοκληρωθούν όλα τα sweeps τερματίζεται ο αλγόριθμος.

2.3 Ο αλγόριθμος Wolff

Ο αλγόριθμος του Wolff[20] ανήκει στην κατηγορία αλγορίθμων cluster και είναι μία παραλλαγή του αλγορίθμου των Swendsen και Wang[22]. Σε αντίθεση με τον αλγόριθμο Metropolis ο οποίος έχει γενική εφαρμογή, ο αλγόριθμος Wolff προκύπτει από βαθύτερη κατανόηση της δυναμικής που προκαλεί την κρίσιμη επιβράδυνση[1].

Ο αλγόριθμος Metropolis, ως αλγόριθμος δυναμικής single-spin-flip, ανανεώνει τα spin ένα - ένα. Στην κρίσιμη περιοχή όμως δημιουργούνται περιοχές με παράλληλα spin και για να έχουμε ασυσχέτιστες διατάξεις αυτές θα πρέπει να καταστραφούν και να δημιουργηθούν νέες αλλού[1]. Όμως, με τον αλγόριθμο Metropolis η πιθανότητα να αλλάξει ένα spin στην περιοχή αυτή είναι πολύ μικρή και απαιτείται μεγάλος χρόνος.

Η ιδέα πίσω από τους αλγορίθμους cluster είναι να περιοριστεί το φαινόμενο της κρίσιμης επιβράδυνσης, ανανεώνοντας τα spin σε περιοχές συγκρίσιμες με τις μεγάλες περιοχές ομοίων spin. Συγκεκριμένα για τον αλγόριθμο του Wolff, αυτό γίνεται επιλέγοντας ένα spin γεννήτορα και κατασκευή ενός cluster γύρω του. Σε κάθε βήμα θα προσθέτουμε μέλη βάση της πιθανότητας P_{add}. Η επιλογή της πιθανότητας αυτής πρέπει να είναι κατάλληλη ώστε να ικανοποιείται η συνθήκη λεπτομερούς ισοζήγησης για το πρότυπο Ising, και όλα τα spin του πλέγματος μπορούν να επιλεγούν ως γεννήτορες ισοπίθανα. Η πιθανότητα προσθήκης ενός spin στο cluster είναι $P_{add} = 1 - e^{-2\beta}$ και τότε μπορούμε να επιλέξουμε $A(\mu \to \nu) = A(\nu \to \mu) = 1$, επομένως, φτιάχνοντας το σμήνος (cluster) βάση της πιθανότητας P_{add} η νέα κατάσταση θα γίνεται πάντα αποδεκτή[3].

Μία βασική διαφορά των αλγορίθμων cluster των Wolff και
Swendsen & Wang σε σχέση με τον αλγόριθμο Metropolis, είναι ότι πλέον το πλέγμα δεν είναι κανονικό και μάλιστα σε μεγάλο βαθμό. Αυτό κάνει την υλοποίηση τέτοιων αλγορίθμων σε υπερυπολογιστές ιδιαίτερα επίπονη και σε αρκετές περιπτώσεις ασύμφορη[4]. Αυτό όμως θα μας απασχολήσει αργότερα.

2.3.1 Υλοποίηση του αλγορίθμου

Η υλοποίηση του αλγορίθμου Wolff είναι ελάχιστα πιο πολύπλοκη από την αντίστοιχη του Metropolis

1. Επιλογή παραμέτρων προσομοίωσης (πλέγμα, θερμοκρασία, πλήθος sweeps, seed γεννήτορα ψευδοτυχαίων).

2. Επιλογή κατάλληλης αρχικής διάταξης (configuration) του πλέγματος (τάξη, αταξία, αποθηκευμένη διάταξη).

3. Υπολογισμός της πιθανότητας επιλογής P_{add}.

4. Για κάθε sweep, επιλογή τυχαίας πλεγματικής θέσης (spin) ισοπίθανα ως γεννήτορα.

5. Ξεκινώντας από τον γεννήτορα, ελέγχουμε τους γείτονες, αν έχουν τον ίδιο προσανατολισμό spin με τον γεννήτορα τους και επιλεγούν βάση την πιθανότητα επιλογής, τα προσθέτουμε στο cluster.

6. Αναδρομικά κάνουμε την ίδια διαδικασία για κάθε νέο στοιχείο του cluster θεωρώντας το τώρα γεννήτορα και ελέγχοντας τους γείτονες που δεν ανήκουν στο cluster, έως ότου έχουμε ελέγξει όλα τα συνδεδεμένα spin. Επιστρέφουμε το μέγεθος του cluster.

[3] Έχουμε δηλαδή την ιδανική περίπτωση για τους λόγους αποδοχής!
[4] Σε επεξεργαστές τύπου vector

7. Αλλάζουμε την τιμή του spin για τα μέλη όλου του cluster.
8. Στην συνέχεια μετράμε την ενέργεια δεσμού και την μαγνήτιση.
9. Νέο sweep.
10. Όταν ολοκληρωθούν όλα τα sweeps τερματίζεται ο αλγόριθμος.

Κεφάλαιο 3

Η κάρτα γραφικών και το GP^2

Η κάρτα γραφικών είναι ένας συνεπεξεργαστής με βασικό ρόλο την επεξεργασία των δεδομένων που θα καταλήξουν στην οθόνη μας (γραφικών). Επομένως, δεν μπορεί να υπάρξει αυτόνομη από έναν κεντρικό επεξεργαστή.

Το GP^2 είναι τεχνική χρήσης των καρτών γραφικών για υπολογισμούς σε εφαρμογές που παραδοσιακά γίνονταν σε κανονικούς επεξεργαστές. Εμάς θα μας απασχολήσει μόνο η υλοποίηση της NVIDIA, η αρχιτεκτονική CUDA με την γλώσσα CUDA/C, που είναι η πιο διαδεδομένη και με τις περισσότερες δυνατότητες.

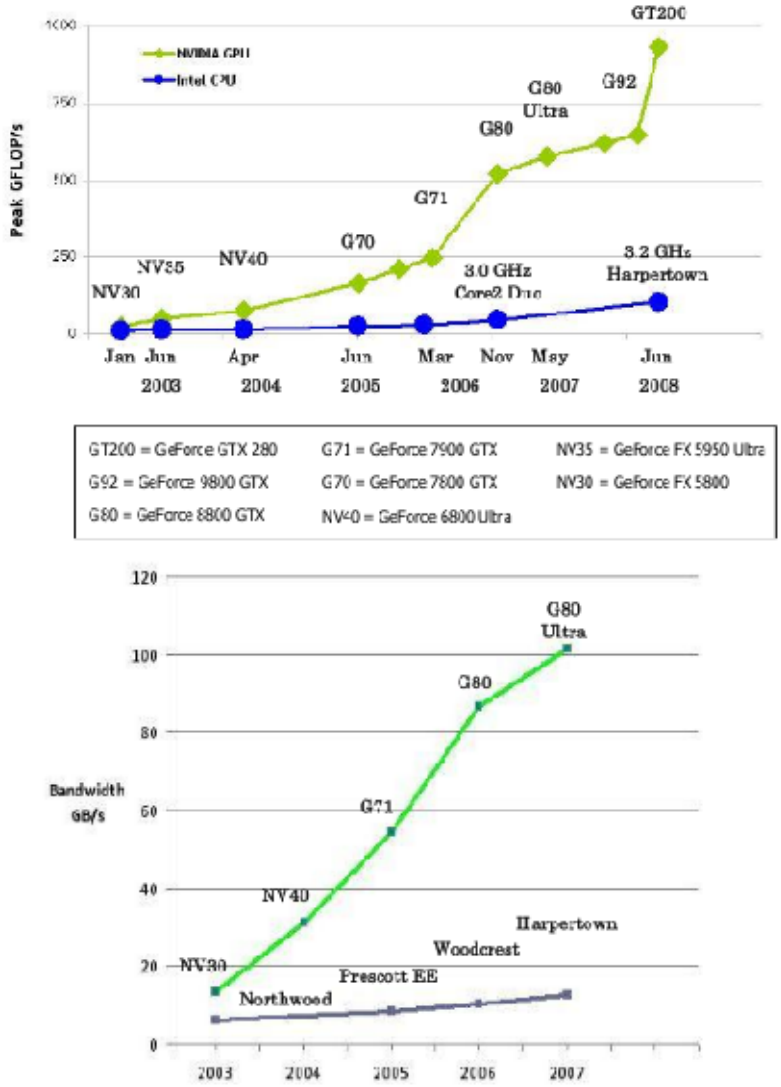
3.1 Αρχιτεκτονική CUDA

Το όνομα CUDA είναι ακρώνυμο για την ονομασία Compute Unified Device Architecture. Αυτή η αρχιτεκτονική μπορεί να βρεθεί σε όλες τις τρεις τελευταίες γενεές καρτών γραφικών της εταιρίας NVIDIA. Υπάρχει αντίστοιχη αρχιτεκτονική και για τις κάρτες γραφικών ATI. Ουσιαστικά δεν υπάρχουν μεγάλες διαφορές στην αρχιτεκτονική.

3.1.1 Η κάρτα γραφικών - Αρχιτεκτονική Single Instruction Multiple Threads

Ωθούμενη από την αυξανόμενη ανάγκη για τρισδιάστατα γραφικά υψηλής ευκρίνειας, η GPU ή αλλιώς η μονάδα επεξεργασίας γραφικών, εξελίχθηκε σε μία πανίσχυρη,προγραμματιζόμενη και σχετικά ευέλικτη μονάδα επεξεργασίας.

Ένα βασικό συστατικό της αρχιτεκτονικής των καρτών γραφικών είναι ότι δομούνται από πολλούς 'πολυεπεξεργαστές' (SM) οι οποίοι με την σειρά τους απο-



Σχήμα 3.1: Κεντρικός επεξεργαστής εναντίον επεξεργαστή γραφικών.

Σχήμα 3.2: Κεντρικός επεξεργαστής εναντίον επεξεργαστή γραφικών από την σκοπιά της φιλοσοφίας της αρχιτεκτονικής τους.

τελούνται από 8 επεξεργαστικές μονάδες (ALU - Arithmetic Logical Unit) οι οποίες ονομάζονται SP[1].

Αν κοιτάξουμε το σχήμα 3.2 μπορούμε να δούμε άμεσα την διαφορά στην φιλοσοφία μεταξύ κεντρικού επεξεργαστή και επεξεργαστή γραφικών. Ο πρώτος αφιερώνει περισσότερα κυκλώματα του στον έλεγχο ροής και την cache ενώ αποτελείται από λίγους γρήγορους πυρήνες. Ο δεύτερος αφιερώνει τα περισσότερα κυκλώματα στην επεξεργασία δεδομένων με πολλούς μικρούς πυρήνες. Κάθε ένας από αυτούς είναι πολύ αργός σε σύγκριση με μία μονάδα του κεντρικού επεξεργαστή, αλλά μπορεί να δουλέψει πολύ αποδοτικά ταυτόχρονα με τους υπολοίπους καθώς όλη η αρχιτεκτονική είναι βελτιστοποιημένη για μαζική παράλληλη επεξεργασία.

Γενικά, αν το πρόβλημα μας είναι παραλληλοποιήσιμο και ο λόγος των αριθμητικών διεργασιών ως προς τις διεργασίες μνήμης (μεταφορές - συγχρονισμός) είναι μεγάλος, τότε είναι ιδανικό για υλοποίηση σε κάρτα γραφικών.

Ένας κεντρικός επεξεργαστής είναι τύπου MIMD - Multiple Instruction Multiple Data ενώ οι κάρτες γραφικών είναι SIMT - Single Instruction Multiple Threads που είναι μία παραλλαγή των επεξεργαστών vector - 'Single Instruction Multiple Data'[2].

Με λίγα λόγια ο κεντρικός επεξεργαστής είναι βελτιστοποιημένος για να επεξεργάζεται μεγάλο όγκο δεδομένων με πολλές αποκλίσεις στην ροή (λ.χ. if..else ή αποκλίνοντα loops), ενώ μία κάρτα γραφικών προτιμάει όσο το δυνατόν περισσότερα δεδομένα, όμοια διαχωρισμένα στους πολυεπεξεργαστές, και μικρές αλλαγές στην ροή των νημάτων (αν και εφόσον τηρούνται κάποιες προϋποθέσεις θα

[1] Streaming Processors

[2] Όλοι οι σύγχρονοι κεντρικοί επεξεργαστές έχουν υλοποιημένες μερικές εντολές SIMD στα κυκλώματα τους. Τυπικές είναι τα σύνολα εντολών SSE-SSE2-SSE3 κτλ, AMD 3DNow και άλλα

δούμε αργότερα ότι μερικές προσεγμένες διακλαδώσεις δεν θα έχουν αρνητικές συνέπειες στις επιδόσεις).

Ο λόγος που οι κάρτες γραφικών δεν θέλουν μεγάλες αποκλίσεις στην ροή των νημάτων δεν είναι μόνον τα λιγότερα κυκλώματα ελέγχου ροής, αλλά και ο τρόπος που οργανώνουν τα νήματα όπως θα δούμε αργότερα αναλυτικά στην οργάνωση των δεδομένων και των νημάτων. Ο επεξεργαστής γραφικών, δεν έχει την δυνατότητα άμεσου I/O[3].

Οι αρχιτεκτονικές CUDA κατηγοριοποιούνται στις υπολογιστικές ικανότητες 1.0, 1.1, 1.2, 1.3 και 2.0. Η 2.0 είναι αρκετά πρόσφατη αρχιτεκτονική με το όνομα Fermi. Λύνει πάρα πολλά από τα προβλήματα και τους περιορισμούς που θα παρουσιάσουμε στην συνέχεια. Είναι μία αρχιτεκτονική που απευθύνεται περισσότερο στο GP[2] και τον επιστημονικό προγραμματισμό, παρά τα παιχνίδια! Από εδώ και στο εξής θα αναφερόμαστε στις αρχιτεκτονικές πριν την Fermi, συγκεκριμένα στις 1.1 και 1.3 που υπάρχει στα περισσότερα μηχανήματα αλλά και στην διάθεση του συγγραφέα. Αξίζει να αναφερθεί εδώ ότι οι αρχιτεκτονικές πριν την 1.3 δεν υποστηρίζουν αριθμούς κινητής υποδιαστολής διπλής ακρίβειας (double, ή 64bit float), ενώ παρόλο που η 1.3 τους υποστηρίζει, η εφαρμογή τους σε αυτή την αρχιτεκτονική είναι σε εμβρυακό στάδιο, με πολλές μαθηματικές εντολές να λείπουν και μεγάλη θυσία σε ταχύτητα.

3.1.2 Οργάνωση δεδομένων και νημάτων

Τα νήματα, χωρίζονται σε grids και αμέσως μετά σε blocks. Το μέγιστο μέγεθος για κάθε διάσταση (ανεξάρτητα) ενός grid είναι {65532, 65532, 1}, ενώ το μέγιστο μέγεθος για κάθε διάσταση ενός block είναι {512, 512, 64}. Ένα block δεν μπορεί να περιέχει όμως πάνω από 1024 νήματα[4], που είναι και ο μέγιστος αριθμός των ενεργών νημάτων σε κάθε στιγμή ανά πολυεπεξεργαστή. Αντίστοιχα, ο μέγιστος αριθμός των ενεργών blocks ανά πολυεπεξεργαστή είναι 8.

Το μέγεθος των grids και των blocks που θα χρησιμοποιήσουμε μπορεί να καθοριστεί κατά την εκτέλεση του προγράμματος. Τα νήματα οργανώνονται ως προς την εκτέλεση στα warps ή δύνες. Οι δύνες εξαρτώνται από την αρχιτεκτονική του ίδιου του επεξεργαστή γραφικών και το μέγεθος τους είναι δεδομένο και σταθερό, επομένως δεν εξαρτάται από το μέγεθος των blocks ή των grids. Συγκεκριμένα, το μέγεθος μίας δύνης είναι 32 νήματα.

Κάθε φορά στην κάρτα γραφικών θα εκτελείται ένα grid μόνο με τα blocks που του αναλογούν, δηλαδή το σύνολο των νημάτων όλων των blocks που ανήκουν

[3]Input / Output. Όπως να τυπώσει άμεσα στην οθόνη, ή να γράψει και να διαβάσει ένα αρχείο.

[4]Για υπολογιστική 1.3 και άνω.

στο συγκεκριμένο grid. Τα νήματα αυτά θα οργανωθούν σε δύνες από τον επεξεργαστή. Για κάθε δύνη ετοιμάζονται οι εντολές για τα νήματα της, αλλά μόνο το μισό καθεμίας μπορεί να εκτελεστεί κάθε φορά[5]. Κάθε φορά θα φορτώνεται στους πολυεπεξεργαστές το μέγιστο πλήθος δυνών προς εκτέλεση[6].

Ο τρόπος με τον οποίο θα εκτελεστούν οι δύνες δεν είναι απλά παράλληλος, η κάρτα γραφικών θα προσπαθεί πάντα να κρατάει όλους τους πολυεπεξεργαστές της πλήρως απασχολημένους. Αυτό σημαίνει ότι δεν μπορεί κανείς να μας εγγυηθεί καμία τάξη στην σειρά εκτέλεσης των νημάτων.
Αν μία δύνη πρέπει να περιμένει εξαιτίας λ.χ. μίας εντολής ανάγνωσης από ένα νήμα της προς την μνήμη, η κάρτα γραφικών θα βάλει την δύνη στην αναμονή μέχρι να ολοκληρωθεί η εντολή αυτή και θα αρχίσει να εκτελεί κάποια άλλα δύνη ή το άλλο μισό της. Επίσης, περνάει από όλες τις δύνες, εκτελώντας μόνο ένα κομμάτι του προγράμματος που τους αναλογεί κάθε φορά. Αν και υπάρχουν αρκετοί τρόποι να επιβάλουμε κάποια τάξη σε αυτό το χάος, γενικά, όπως θα δούμε και στην παρουσίαση του προγραμματισμού σε κάρτα γραφικών, ο τρόπος που επεξεργαζόμαστε τα δεδομένα μας πρέπει να είναι όσο πιο ασυσχέτιστος, ή αλλιώς, αυθεντικά παράλληλος γίνεται.

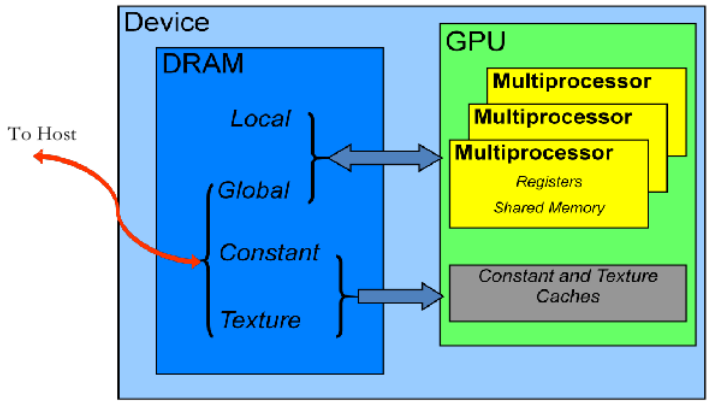
3.1.3 Τύποι μνημών

Ένα από τα στοιχεία που προκαλεί πονοκέφαλο όταν εργαζόμαστε με κάρτες γραφικών είναι οι πολλοί τύποι μνήμης αλλά και ο ρόλος της καθεμίας. Η κάρτα γραφικών δεν μπορεί να διαβάσει ή να γράψει άμεσα στην μνήμη του υπολογιστή. Αρχικά θα πρέπει να αναφέρουμε τους τέσσερις φυσικούς τύπους μνήμης και στην συνέχεια τρεις ειδικές υλοποιήσεις φυσικών μνημών:

1. Την καθολική μνήμη (Global memory).
2. Την κοινόχρηστη μνήμη (Shared memory).
3. Την τοπική μνήμη (Local Memory).
4. Τους καταχωρητές (Registers).
5. Την δεσμευμένη μνήμη συστήματος (Page-locked memory).
6. Την μνήμη σταθερών (Constant Memory).
7. Την μνήμη υφής (Texture Memory).

[5] Στην αρχιτεκτονική Fermi μπορεί να εκτελεστεί μία ολόκληρη δύνη.

[6] Η SIMT είναι παραλλαγή της SIMD από την άποψη ότι μόνο οι δύνες ακολουθούν την λογική SIMD. Αυτές τις δύνες όμως, ως οντότητα, μπορούμε να τις χειριστούμε όπως θέλουμε και να αποκλίνουν χωρίς πρόβλημα όπως θα δούμε στην συνέχεια.



Σχήμα 3.3: Μια ιδέα του τι μας περιμένει…

Η καθολική μνήμη (Global memory)

Η καθολική μνήμη ή αλλιώς 'Global memory' είναι η πιο βασική και είναι μέρος της μνήμης DRAM της συσκευής. Θα μπορούσε κανείς να την αντιστοιχήσει με την μνήμη RAM του υπολογιστή. Πριν κάνουμε οτιδήποτε, τα δεδομένα που θα επεξεργαστούμε πρέπει να μεταφερθούν στην καθολική μνήμη. Είναι η πιο μεγάλη μνήμη σε χωρητικότητα (φτάνει ως και τα 4GByte) αλλά και η πιο αργή. Σε σύγκριση με την μνήμη RAM του υπολογιστή όμως έχει σχεδόν την δεκαπλάσια ταχύτητα $(\sim 120 Gbyte/s)$[7]!

Όπως αναφέραμε, για να δει η κάρτα γραφικών τα δεδομένα μας πρέπει να τα μεταφέρουμε στην καθολική της μνήμη πρώτα και αντίστοιχα, αν θέλουμε να τα δει ο επεξεργαστής ή να τα εξάγουμε σε ένα αρχείο ή την οθόνη, πρέπει να τα μεταφέρουμε πίσω. Εδώ, ήδη μπορούμε να διαπιστώσουμε ένα πρόβλημα, αυτή η επικοινωνία είναι 'ακριβή', καθώς ο δίαυλος που ενώνει αυτά τα δύο είναι συγκριτικά πολύ αργός. Πρέπει να μην μεταφέρουμε άσκοπα δεδομένα. Αν, όπως στην περίπτωση μας, έχουμε ένα πλέγμα από το οποία θέλουμε μόνο δύο ποσότητες, δεν θα μεταφέρουμε όλο το πλέγμα, αλλά θα προσπαθήσουμε να υπολογίσουμε αυτές τις ποσότητες μέσα στην κάρτα, ακόμα και αν κάτι τέτοιο δεν είναι αποδοτικό (π.χ. μπορεί να γίνει μόνο σειριακά ή με πολύ λίγα νήματα).

Από την στιγμή που περάσουμε τα δεδομένα μας στην καθολική μνήμη, όλα τα νήματα μπορούν να διαβάσουν και να γράψουν σε αυτή. Η καθολική μνήμη είναι προσβάσιμη με συναλλαγές των 32, 64 ή 128 byte από τις δύνες. Μία διαδικασία ανάγνωσης ή εγγραφής από ή προς την καθολική μνήμη μπορεί να μεταφέρει

[7] Στις πιο πλήρης υλοποιήσεις υπολογιστικών ικανοτήτων 1.3 και άνω και γενικότερα σύγχρονων καρτών γραφικών. Για υπολογιστική ικανότητα 1.1 στην καλυτερη περίπτωση θα είναι $\sim 70 Gbyte/s$.

ως και 128 byte σε μία συναλλαγή. Πρέπει να δώσουμε σημασία σε αυτό το σκέλος, παρόλο που φαίνεται σαν λεπτομέρεια, είναι ένα από τα βασικά χαρακτηριστικά για ένα αποτελεσματικό πρόγραμμα. Η καθολική μνήμη δεν είναι απλά αργή, έχει και μεγάλη καθυστέρηση της τάξης των 500-700 κύκλων! Στην αρχιτεκτονική Fermi τα πράγματα αλλάζουν πολύ, καθώς πλέον η καθολική μνήμη μπορεί να είναι cached, επομένως μία ανάγνωση μπορεί να κοστίσει όσο λίγο όσο ένας κύκλος του ρολογιού του πολυεπεξεργαστή.

Οι προσβάσεις στην μνήμη από τα νήματα μίας δύνης πρέπει να είναι ευθυγραμμισμένες με αυτά τα μεγέθη (ανάλογα το μέγεθος της λέξης),δηλαδή η θέση της μνήμης που θα διαβάσει το πρώτο νήμα να είναι πολλαπλάσιο αυτών. Όταν τα νήματα μίας δύνης δίνουν μία εντολή ανάγνωσης από την καθολική μνήμη, η δύνη θα προσπαθήσει να οργανώσει αυτές τις εντολές για το κάθε μισό της δύνης. Σκοπός είναι με όσο το δυνατόν λιγότερες συναλλαγές να μεταφερθούν όλα τα απαιτούμενα δεδομένα για ολόκληρη την δύνη.

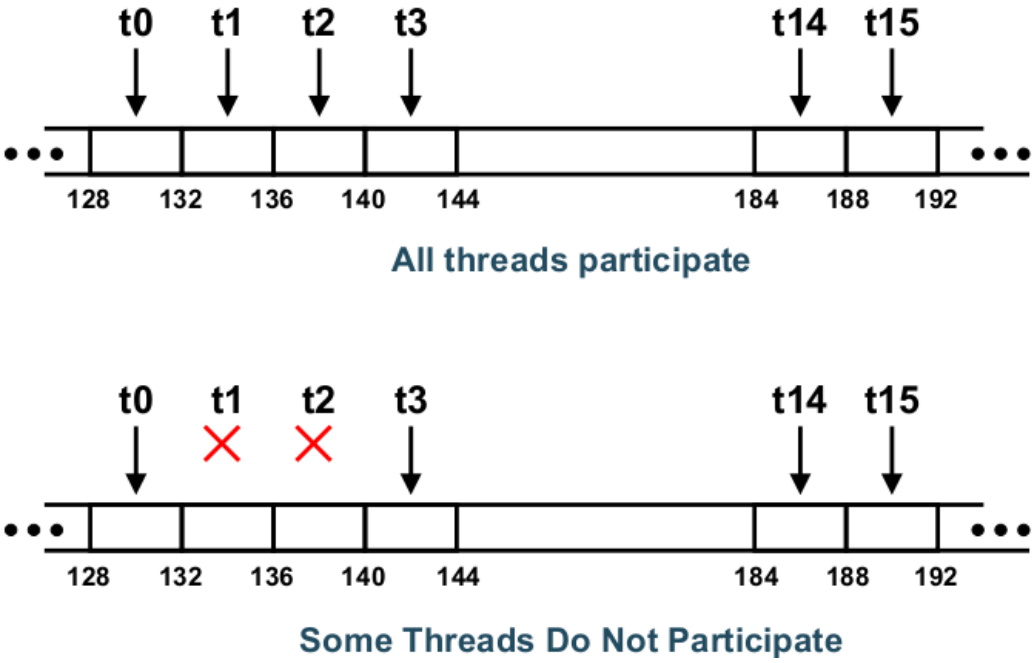
Ας πάρουμε την περίπτωση για ανάγνωση ακεραίου των 4 byte, τότε αν το πρώτο νήμα της δύνης δίνει εντολή ανάγνωσης για τις θέση 0-3, το δεύτερο για τις θέσεις 4-7 και αντίστοιχα μέχρι και το όγδοο, όπου το νήμα 8 δίνει εντολή για τις θέσεις 28-31, τότε η εντολή και για τα 8 νήματα θα πραγματοποιηθεί σε μία μόνο συναλλαγή. Όμως, όπως αναφέραμε είχαμε την δυνατότητα να μεταφέρουμε 128 byte σε μία συναλλαγή επομένως η αποδοτικότητα της συναλλαγής είναι μικρή! Το ιδανικό θα ήταν να έδιναν την εντολή 32 νήματα όπου εκεί θα είχαμε την μέγιστη δυνατή αποδοτικότητα από τον δίαυλο.

Δεν είναι πάντα εύκολο όμως να οργανώσει κανείς τόσο ωραία τις προσπελάσεις στην μνήμη. Αυτό που συμβαίνει συνήθως είναι να υπάρχει μη ευθυγραμμισμένη πρόσβαση. Αν πάρουμε πάλι το προηγούμενο παράδειγμα, μη ευθυγραμμισμένη πρόσβαση θα είχαμε αν:

- Το πρώτο νήμα έδινε εντολή για τις θέσεις 1-4 και αντίστοιχα τα υπόλοιπα με το όγδοο για τις θέσεις 29-32. Τώρα λοιπόν,θα μεταφέρουμε 64byte για 32byte χρήσιμων δεδομένων, καθώς η αρχική θέση ανάγνωσης σε σχέση με την αρχή της μνήμης δεν είναι κάποιο πολλαπλάσιο των δυνατών μεγεθών μεταφοράς.

- Το πρώτο νήμα έδινε εντολή για τις θέσεις 0-3, το δεύτερο για τις θέσεις 8-11, το τρίτο για τις θέσεις 4-7 και τα υπόλοιπα νήματα κανονικά. Εδώ, οι προσβάσεις μνήμης γι αυτό το νήμα θα γίνουν σειριακά για τις παλιές αρχιτεκτονικές. Συγκεκριμένα 8*32byte. Βλέπετε αμέσως την διαφορά! Στις νεότερες αρχιτεκτονικές αυτό το πρόβλημα έχει εν μέρει λυθεί[8].

[8] 1.3 και άνω.

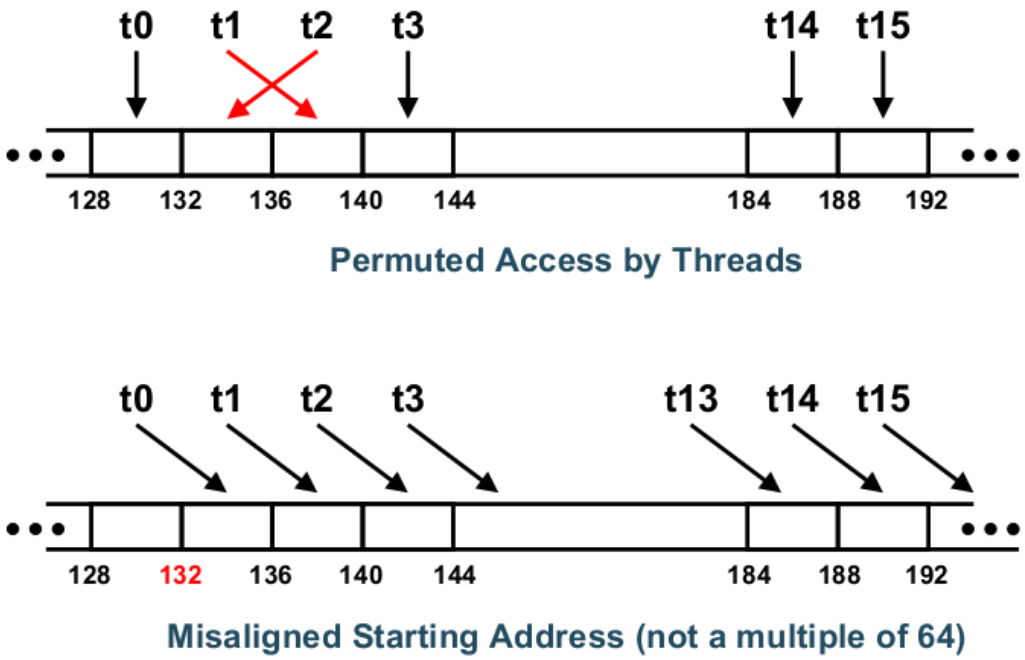


Σχήμα 3.4: ”Coalesced” συναλλαγή μνήμης.

Πρέπει να σημειωθεί ότι αν το δεύτερο νήμα δεν χρειαστεί να κάνει ανάγνωση αλλά τα υπόλοιπα διαβάσουν όπως στο αρχικό παράδειγμα, η συναλλαγή θα γίνει σε μία μόνο εντολή. Όταν ένα νήμα δεν συμμετέχει, αν το υπόλοιπα νήματα διαβάσουν ευθυγραμμισμένα οι εντολές θα εκτελεστούν σε μία συναλλαγή.

Τα σχήματα 3.5 και 3.6 αναφέρονται στα προβλήματα που αναφέραμε. Το σχήμα 3.5 είναι για αρχιτεκτονικές 1.0 ως 1.1, ενώ το δεύτερο για αρχιτεκτονική 1.3. Βλέπουμε ότι με την νεότερη αρχιτεκτονική την γλιτώνουμε πολύ πιο φτηνά, καθώς αντί να γίνουν όλες οι συναλλαγές σειριακά, απλά χωρίζονται οι συναλλαγές σε δύο αντί για μία.

Τα πράγματα γίνονται ακόμα πιο δύσκολα όταν δεν έχουμε γραμμική πρόσβαση από τα νήματα στην εγγραφή. Μόνο ένα νήμα μπορεί να γράψει σε μία διεύθυνσης της καθολικής μνήμης σε κάθε στιγμή. Αν παραπάνω από ένα νήματα επιχειρήσουν να γράψουν στην ίδια διεύθυνση, τότε η εντολή της δύνης από μία ενωμένη θα κατακερματιστεί σε ισόποσες σειριακές. Πρέπει να είμαστε προσεκτικοί, καθώς κάτι τέτοιο θα έχει δραματικές συνέπειες για την απόδοση.
Πολλές φορές είναι απλά αδύνατο να προγραμματίσουμε τις συναλλαγές με την μνήμη με τέτοιο τρόπο. Ευτυχώς, όπως είπαμε, υπάρχουν και άλλοι τύποι μνήμης!



Σχήμα 3.5: ”Uncoalesced” συναλλαγή μνήμης.

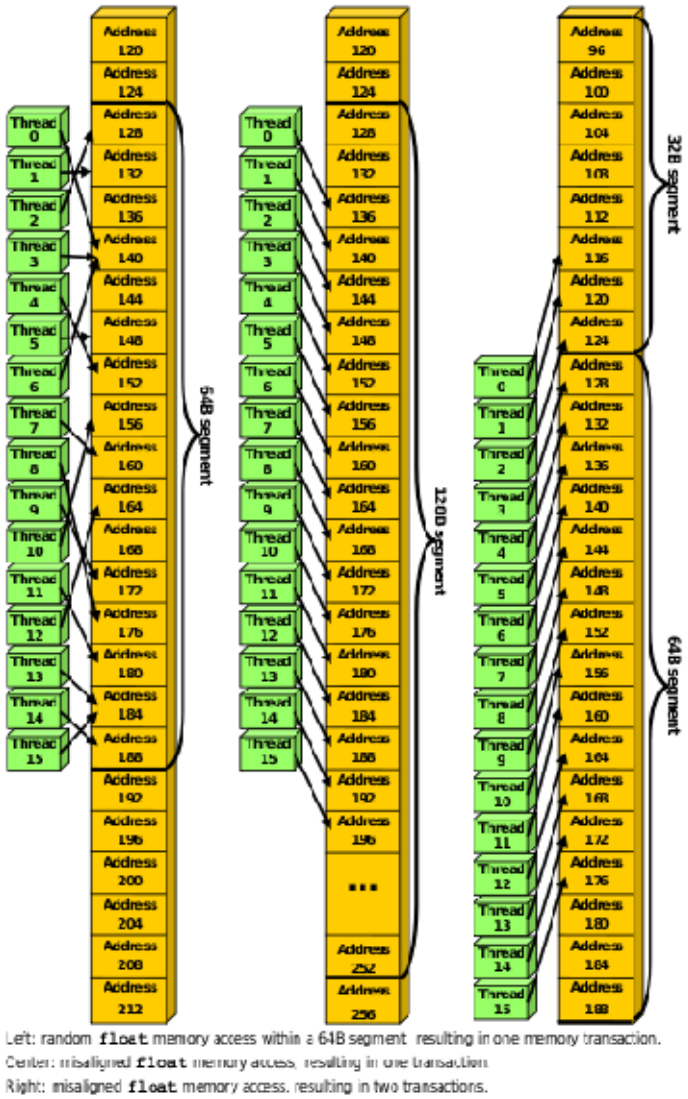
Η κοινόχρηστη μνήμη (Shared memory)

Η κοινόχρηστη μνήμη ή αλλιώς Shared memory αποτελεί μνήμη η οποία βρίσκεται στα κυκλώματα του κάθε πολυεπεξεργαστή (on chip). Είναι η πιό γρήγορη μνήμη (Tbyte/s) αλλά είναι πολύ περιορισμένη σε μέγεθος. Έχει συνολική χωρητικότητα μόλις 16 Kbyte σε κάθε πολυεπεξεργαστή[9] και έχει την ζωή του block στο οποίο ορίζεται.

Η κοινόχρηστη μνήμη είναι ορατή μόνο στα νήματα ενός block. Νήματα ενός άλλου block δεν μπορούν να διαβάσουν ή να γράψουν στην κοινόχρηστη μνήμη που είναι δεσμευμένη σε ένα άλλο block. Πώς χωρίζεται τότε αυτή η μνήμη όμως;

Είμαστε ελεύθεροι να κατανέμουμε την κοινόχρηστη μνήμη όπως θεωρούμε εμείς σωστό σε κάθε block. Η κοινόχρηστη μνήμη βέβαια, κατανέμεται ισόποσα και καθολικά, επομένως μπορούμε να ορίσουμε μόνον πόση από αυτή την μνήμη θα χρησιμοποιεί ένα block, κάθε block τότε θα έχει αυτή την διαμόρφωση. Ένας περιορισμός είναι ότι αν ορίσουμε 16 Kbyte κοινόχρηστης μνήμης σε κάθε block τότε κάθε επεξεργαστής δεν θα μπορεί να τρέξει πάνω από ένα block ταυτόχρονα! Αυτό θα δημιουργήσει προβλήματα με την πληρότητα του κάθε πολυεπεξεργαστή,

[9] Στην αρχιτεκτονική Fermi μπορεί να φτάσει τα 48Kbyte.



Σχήμα 3.6: Περιπτώσεις μη ευθυγραμμισμένων προσβάσεων για την αρχιτεκτονική 1.3. Στις παλαιότερες αρχιτεκτονικές όλες οι συναλλαγές θα γίνονταν σειριακα!

η πληρότητα ενός πολυεπεξεργαστη ορίζεται ως ο λόγος των ενεργών δυνών ενός πολυεπεξεργαστή ως προς τον μέγιστο δυνατό αριθμό ενεργών δυνών ανά πολυεπεξεργαστή, και θα αναλύσουμε την σημασία του αργότερα.

Εκτός του καταμερισμού της κοινόχρηστης μνήμης ανά block υπάρχει και ένας άλλος καταμερισμός ο οποίος βασίζεται στην αρχιτεκτονική. Η κοινόχρηστη μνήμη χωρίζεται φυσικά σε πολλές ομάδες των δεκαέξι τραπεζών με την χωρητικότητα της κάθε τράπεζας να είναι 4byte.

Θα παρατηρήσατε ως τώρα ότι παντού αναφέρουμε πολλαπλάσια του δεκαέξι. Αυτός είναι ένας μαγικός αριθμός για τις κάρτες γραφικών στις οποίες αναφερόμαστε. Για όλα φταίνε οι δύνες και ο χωρισμός τους σε πάνω και κάτω μισό (των δεκαέξι νημάτων το καθένα φυσικά!). Ο λόγος λοιπόν, που κάθε ομάδα κοινόχρηστης μνήμης είναι μοιρασμένη σε δεκαέξι τράπεζες είναι ότι αυτές οι τράπεζες μπορούν να προσπελαστούν για ανάγνωση και εγγραφή από κάθε νήμα ενός μισού μίας δύνης, ταυτόχρονα! Αυτός είναι και ο λόγος της τεράστιας ταχύτητας τους. Αρκεί να πούμε ότι ουσιαστικά κάθε τράπεζα έχει περίπου την ίδια ταχύτητα με την καθολική μνήμη, αλλά τώρα μπορούμε να διαβάσουμε πολλές τράπεζες ανά πολυεπεξεργαστή ταυτόχρονα! Με το μέγιστο πλήθος των ταυτόχρονα ενεργών δυνών ανά πολυεπεξεργαστή να είναι τριάντα δύο , τότε θα έχουμε στην πιο αποδοτική διαμόρφωση 16*16 ταυτόχρονες συναλλαγές με την διαμοιρασμένη μνήμη ανά πολυεπεξεργαστή. Επίσης, η καθυστέρηση μίας συναλλαγής σε σχέση με της καθολικής μνήμης είναι αμελητέα.

Η κοινόχρηστη μνήμη δεν έρχεται χωρίς περιορισμούς και πονοκεφάλους για τον προγραμματιστή. Όπως είπαμε σαν μνήμη ορίζεται μόνο στο εσωτερικό ενός block και επομένως ένα νήμα από άλλο block δεν μπορεί να δει την κοινόχρηστη μνήμη ενός άλλου block. Ο όρος κοινόχρηστη λοιπόν μπορεί να προκαλεί μία σύγχυση στην αρχή. Δεν είναι κοινόχρηστη για όλα τα νήματα, γι αυτή την δουλειά υπάρχει η καθολική μνήμη.

Η κοινόχρηστη μνήμη επίσης, όταν ξεκινήσουμε να εκτελούμε το πρόγραμμα μας, θα είναι άδεια. Αφού μεταφέραμε τα δεδομένα μας στην καθολική μνήμη, εν συνεχεία πρέπει να τα μεταφέρουμε και στην κοινόχρηστη μνήμη αν σκοπεύουμε να την χρησιμοποιήσουμε και τέλος να τα μεταφέρουμε πίσω στην καθολική μνήμη αν θέλουμε να είναι διαθέσιμα στην συνέχεια. Επομένως σαν μνήμη έχει την ζωή ενός block.

Αν νομίζατε ότι με την κοινόχρηστη μνήμη γλιτώσατε από τις πολύπλοκες μορφές προσπέλασης της μνήμης, κάνετε λάθος. Η κοινόχρηστη μνήμη είναι επίσης

περίεργη στο πως προτιμάει να συναλλάσσεται με τα νήματα. Τα νήματα μίας δύνης πρέπει να συναλλάσσονται με μία ομάδα τραπεζών με ανεξάρτητο και γραμμικό τρόπο, με την ζητούμενη διεύθυνση μνήμης του πρώτου νήματος να είναι η πρώτη τράπεζα και του τελευταίου νήματος να είναι η τελευταία τράπεζα. Αν αυτή η συνθήκη ικανοποιείται, τότε η συναλλαγή θα ολοκληρωθεί ταυτόχρονα και για τα δεκαέξι νήματα. Τότε έχουμε τράπεζες χωρίς συγκρούσεις (collision-free banks)

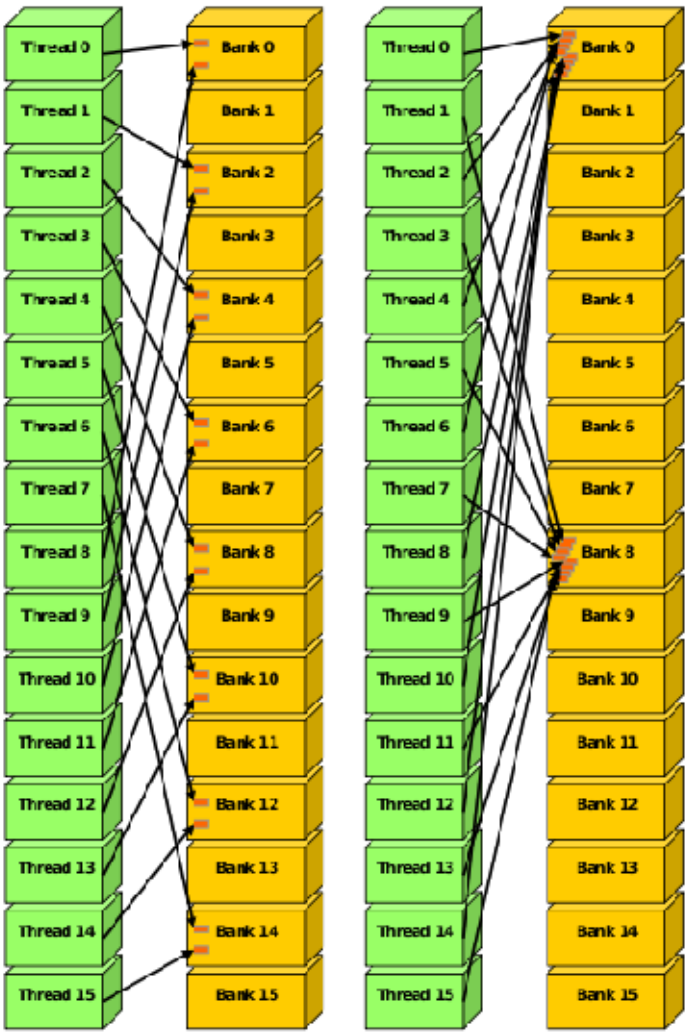
Αν κάποια νήματα ζητήσουν κάποια συναλλαγή με την κοινόχρηστη μνήμη που πέφτει στην ίδια διεύθυνση μνήμης ή σε τράπεζες διαφορετικών ομάδων (στο ίδιο block πάντα), τότε αυτή η συναλλαγή θα γίνει σειριακή για όσα από αυτά τα νήματα χρειαστεί, κάτι που θέλουμε να αποφύγουμε. Τότε έχουμε συγκρούσεις τραπεζών (bank collisions). Αυτό δεν ισχύει αν όλα τα νήματα του μισού της δύνης διαβάζουν την ίδια τράπεζα, καθώς τότε έχουμε την περίπτωση του broadcasting (εκπομπή) που υποστηρίζεται από την κοινόχρηστη μνήμη και για όλα τα εμπλεκόμενα νήματα του μισού της δύνης η συναλλαγή θα εκτελεστεί παράλληλα και ταυτόχρονα.

Στην περίπτωση που πολλά νήματα γράφουν ή διαβάζουν την ίδια τράπεζα, κανένας δεν μπορεί να μας εγγυηθεί την σειρά των προσπελάσεων και πιο νήμα θα γράψει τελευταίο. Αυτό είναι συχνό λάθος στον προγραμματισμό σε κάρτα γραφικών, αλλά ευτυχώς, όπως θα δούμε στην υποενότητα 3.2.6, μας δίνεται η δυνατότητα να συγχρονίσουμε τα νήματα ενός block σε όποιο σημείο του προγράμματος μας θέλουμε.

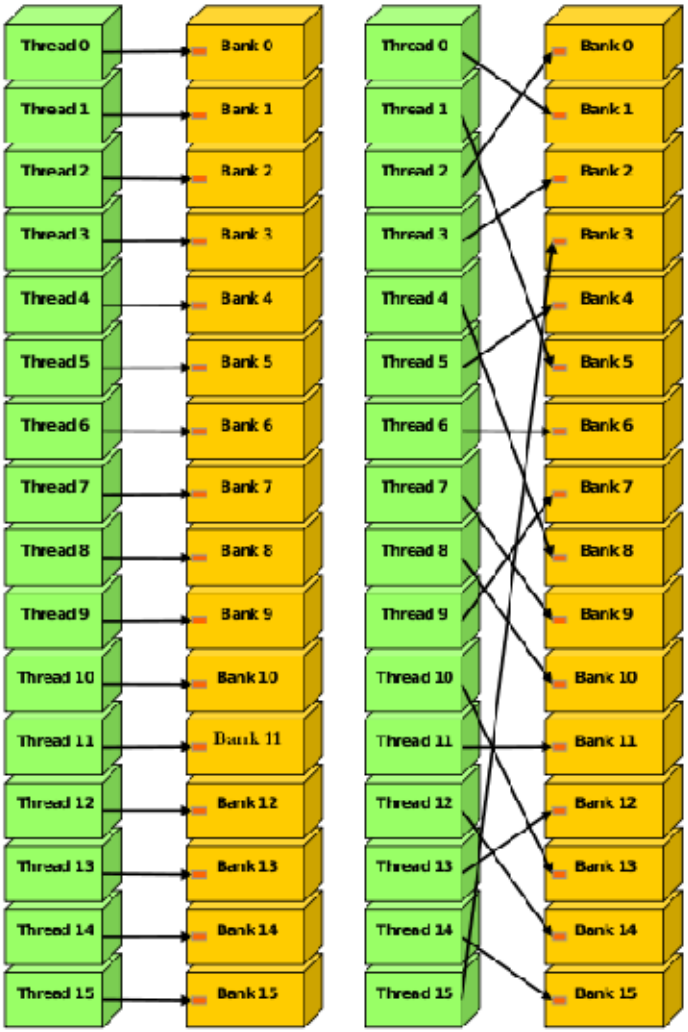
Η τοπική μνήμη (Local memory)

Η τοπική μνήμη είναι η προσωπική μνήμη κάθε νήματος. Κανένα άλλο νήμα δεν έχει πρόσβαση σε αυτή, επομένως οι συναλλαγές είναι πάντα γραμμικές και ευθυγραμμισμένες. Η τοπική μνήμη βρίσκεται στην DRAM της κάρτας γραφικών. Έχει, επομένως, την ίδια ταχύτητα και υστέρηση με την καθολική μνήμη. Εδώ δημιουργείται ένα ερώτημα, γιατί το κάθε νήμα να έχει αργή μνήμη; Η ταχύτητα μίας τοπικής μνήμης θα πρέπει να είναι της τάξης της κοινόχρηστης. Ουσιαστικά πάρα πολύ σπάνια χρησιμοποιούμε αυτή την μνήμη, γιατί δεν χρειάζεται σχεδόν ποτέ. Οι πολυεπεξεργαστές έχουν μία μορφή μνήμης, τους καταχωρητές, η οποία παίζει τον ρόλο της τοπικής μνήμης. Ουσιαστικά η τοπική μνήμη είναι η τελευταία διέξοδος όταν οι απαιτήσεις σε μνήμη κάθε νήματος είναι πολύ μεγάλες.

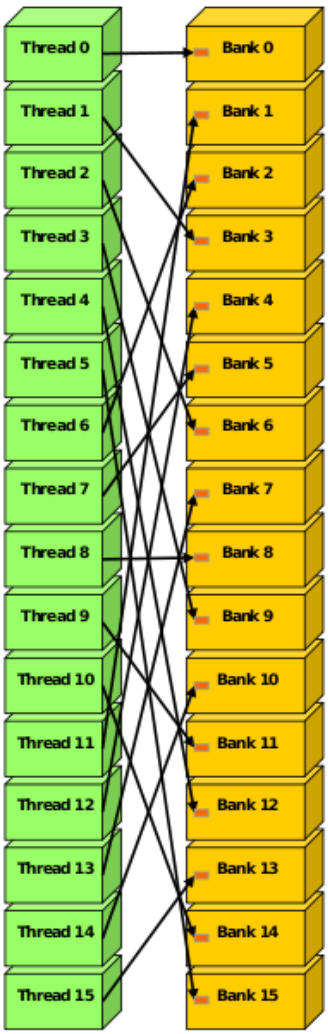
Δεν μπορούμε να ορίσουμε καν την τοπική μνήμη. Όποτε αφιερώνουμε μνήμη σε κάθε νήμα, ο μεταγλωττιστής όταν ερμηνεύει την εντολή θα δεσμεύσει έναν ή περισσότερους καταχωρητές. Τοπική μνήμη θα δεσμευτεί εφόσον περάσουμε το όριο των διαθέσιμων από την αρχιτεκτονική καταχωρητών ή αν δεσμεύσουμε πίνακα στο νήμα που είναι μεγάλος ή ο μεταγλωττιστής δεν μπορεί να καταλάβει αν θα είναι σταθερός ή/και μικρός σε μέγεθος. Η μεταφορά δέσμευσης της μνήμης



Σχήμα 3.7: ”Uncoalesced” συναλλαγή μνήμης.



Σχήμα 3.8: ”Coalesced” συναλλαγή μνήμης.



Σχήμα 3.9: ”Coalesced” συναλλαγή μνήμης.

από τους καταχωρητές στην τοπική μνήμη ονομάζεται ”spilling” και έχει δραματικές επιπτώσεις στην επίδοση ενός πολυεπεξεργαστή. Το γιατί όμως κάποιος να θέλει να πιέσει τα όρια της μνήμης καταχωρητών δημιουργώντας άμεσο κίνδυνο να κάνει το πρόγραμμα του πολύ πιο αργό λόγω spilling θα το δούμε αμέσως αφού αναλύσουμε τον ρόλο και την λειτουργία των καταχωρητών.

Οι καταχωρητές (Registers)

Όπως αναφέραμε, οι καταχωρητές είναι αυτοί που επωμίζονται τις απαιτήσεις σε μνήμη του κάθε νήματος. Οι καταχωρητές είναι μέρος των κυκλωμάτων κάθε πολυεπεξεργαστή και το μέγεθος τους εξαρτάται από την αρχιτεκτονική. Οι κάρτες γραφικών με αρχιτεκτονική 1.0 και 1.1 έχουν μόλις 8192 καταχωρητές των 32bit, ενώ στις αρχιτεκτονικές 1.2 και 1.3 υπάρχουν 16834 καταχωρητές των 32bit. Η ταχύτητα τους είναι εφάμιλλη της κοινόχρηστης μνήμης αφού ένα μισό της δύνης μπορεί να γράψει στους καταχωρητές παράλληλα σε μία συναλλαγή, αλλά χωρίς την πολυπλοκότητα που είδαμε στην καθολική και την κοινόχρηστη μνήμη.

Εφόσον τα blocks (που είναι και ο κύριος δομικός λίθος για ένα πρόγραμμα σε κάρτα γραφικών) αποτελούνται από μία ομάδα νημάτων και αυτά απαιτούν μνήμη από τους καταχωρητές, θα μετράμε το πλήθος των καταχωρητών με βάση τα blocks. Επομένως, αν ένα block έχει μέγεθος 256 νήματα και το κάθε ένα απαιτεί 10 καταχωρητές, τότε θα χρειαστούμε συνολικά 2560 καταχωρητές. Λαμβάνοντας υπόψιν ότι το μέγιστο δυνατό πλήθος των ενεργών νημάτων κυμαίνεται από 768[10] ως 1024[11], δηλαδή τρία ως τέσσερα blocks με την δεδομένη διαμόρφωση, οι καταχωρητές δεν αποτελούν εμπόδιο. Αν τώρα πάρουμε ως παράδειγμα ένα πιο απαιτητικό πρόγραμμα που χρειάζεται 20 καταχωρητές ανά νήμα, τότε στις αρχιτεκτονικές 1.0 και 1.1 θα έχουμε λιγότερα από 768 νήματα ενεργά. Όμως δεν μπορούμε να έχουν ελεύθερα νήματα σε έναν πολυεπεξεργαστή, πρέπει να αποτελούν ένα block. Επομένως, σε αυτές τις αρχιτεκτονικές θα μπορεί να εκτελείται μόλις ένα block με 256 νήματα! Αμέσως, η πληρότητα των πολυεπεξεργαστών πέφτει στο 33%, Στην περίπτωση των αρχιτεκτονικών 1.2 και 1.3 θα μπορούν να εκτελούνται μόνο 3 block με 768 νήματα από το μέγιστο στον 1024 νημάτων[12].

Είδαμε πως μεγάλη χρήση καταχωρητών μπορεί να δημιουργήσει κάποια προβλήματα στον αριθμό των ενεργών νημάτων. Υποθέσαμε ότι οι καταχωρητές αναλαμβάνουν μόνο την μνήμη που ζητάμε για κάθε νήμα. Οι καταχωρητές χρησιμοποιούνται από τον μεταγλωττιστή αυτόματα και για την αποφυγή υπολογισμού

[10] Για τις υπολογιστικές ικανότητες 1.0,1.1 και 1.2

[11] Για υπολογιστική ικανότητα 1.3. Για την 2.0 αυτός ο αριθμός είναι 1536

[12] Σε κάποιες ειδικές περιπτώσεις η μικρή πληρότητα μπορεί να μην είναι πρόβλημα. Σε κάθε περίπτωση δεν είναι το πιο σημαντικό στοιχείο για ένα αποδοτικό πρόβλημα όπως θα δούμε στην συνέχεια

παραστάσεων που χρησιμοποιούνται πάνω από μία φορά χωρίς να αλλάξει η τιμή τους στο ενδιάμεσο[13]. Αυτό, ναι μεν επιταχύνει κάπως ένα πρόγραμμα αλλά η διαφορά μπορεί να είναι πολύ μικρή. Επομένως, εδώ ερχόμαστε στο ερώτημα, αν θα έπρεπε να περιορίσουμε τον αριθμό των καταχωρητών ανά νήμα με ειδικές εντολές στον μεταγγλωτιστή, με σκοπό να επιτύχουμε μεγαλύτερη πληρότητα. Μία τέτοια παρέμβαση δεν ξέρουμε σίγουρα αν θα έχει καλά αποτελέσματα. Όπως θα δούμε στην υποενότητα 3.1.5 η πληρότητα δεν είναι απαραίτητο να είναι πάντα 100% για να έχουμε την μέγιστη αποδοτικότητα. Παράλληλα, θα μειώσουμε τους καταχωρητές που αναλαμβάνουν να μειώσουν τον αριθμητικό φόρτο, αποθηκεύοντας τοπικά σταθερές παραστάσεις και αν το παρακάνουμε θα έχουμε διάχυση της μνήμης από τους καταχωρητές στην αργή τοπική μνήμη. Δεν υπάρχει έτοιμη λύση, κάποιος θα πρέπει να πειραματιστεί για να βρει μία καλή ισορροπία!

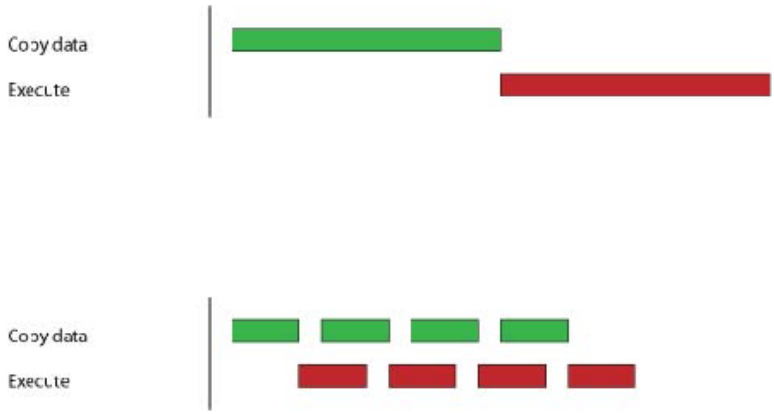
Η κλειδωμένη μνήμη συστήματος (Page-locked memory)

Η κλειδωμένη μνήμη συστήματος (Page-locked memory), είναι ουσιαστικά μνήμη RAM του υπολογιστή η οποία δεσμεύεται μέσω της κάρτα γραφικών, αλλά είναι γενικά προσπελάσιμη μόνο από τον κεντρικό επεξεργαστή (αν θέλουμε να χρησιμοποιηθεί από την κάρτα γραφικών, θα πρέπει να αντιγράψουμε τα δεδομένα στην καθολική μνήμη). Όταν η μνήμη δεσμεύεται με τέτοιο τρόπο, αφαιρείται από την τράπεζα σελιδοποίησης (paging) του συστήματος. Αυτό έχει ως αποτέλεσμα την πιο αποδοτική μεταφορά δεδομένων μέσα από τον δίαυλο κάρτας γραφικών - κεντρικού επεξεργαστή (PCI Express). Το κέρδος σε ταχύτητα μπορεί να φτάσει ως και 40% σε ορισμένες περιπτώσεις, επομένως όταν χρειάζεται να μεταφέρουμε συχνά μεγάλο όγκο δεδομένων, αποτελεί μία καλή επιλογή.

Ίσως το πιο σημαντικό πλεονέκτημα αφορά τις αρχιτεκτονικές 1.2 και 1.3 (αλλά όχι όλες τις υλοποιήσεις). Αν δεσμεύσουμε μνήμη με αυτό τον τρόπο, και η κάρτα γραφικών επιτρέπει ταυτόχρονη μεταφορά δεδομένων και εκτέλεση (σχήμα 3.10) και επίσης το πρόγραμμα μας μπορεί να χωριστεί έτσι ώστε να μην εξαρτάται από όλη την μνήμη (π.χ. δύο συναρτήσεις της κάρτας γραφικών που επεξεργάζονται δύο ανεξάρτητα κομμάτια στην μνήμη), τότε μπορούμε να κρύψουμε σε μεγάλο βαθμό τις καθυστερήσεις που προκαλούν οι αναγκαστικές μεταφορές από και προς την καθολική μνήμη.

Η κλειδωμένη μνήμη συστήματος μπορεί να παραμετροποιηθεί έτσι ώστε να αλλάζει ο τρόπος που ελέγχεται από το σύστημα και επομένως οι ιδιότητες της:

[13]Κλασσικό παράδειγμα είναι οι δείκτες (indexes) τις μνήμης.



Σχήμα 3.10: Ασύγχρονη εκτέλεση kernel με μεταφορά δεδομένων.

Φορητή μνήμη (Portable memory): Όταν δουλεύουμε με μία πολυνηματική βιβλιοθήκη όπως το OpenMP[14], οι τροποποιήσεις που επιβάλλει η δεσμευμένη μνήμη συστήματος είναι ορατές μόνο από το νήμα του κεντρικού επεξεργαστή που ελέγχει την κάρτα γραφικών. Αυτή η μνήμη παραμένει μεν προσβάσιμη και ορατή σε όλα τα νήματα του κεντρικού επεξεργαστή, αλλά αν θέλουμε να ισχύουν οι τροποποιήσεις για όλα τα νήματα τότε πρέπει να χρησιμοποιήσουμε την παράμετρο ”Φορητή μνήμη”.

Συνδυασμός-εγγραφών (Write-combining memory): Εκ του ορισμού της, η κλειδωμένη μνήμη συστήματος είναι ρυθμισμένη ώστε να μπορεί να είναι cached. Η παράμετρος συνδυασμού εγγραφών αφαιρεί αυτή την δυνατότητα απελευθερώνοντας έτσι πολλούς πόρους από την L1 και L2 cache του συστήματος και αφήνοντας την διαθέσιμη για την εφαρμογή μας. Επίσης, επειδή πλέον οι προσβάσεις σε αυτή την μνήμη δεν ελέγχονται από το σύστημα, η ταχύτητα μεταφοράς από το δίαυλο μπορεί να αυξηθεί ως και 40%. Η μνήμη αυτή όμως είναι απαγορευτικά αργή σε ανάγνωσης από τον επεξεργαστή και από την πλευρά του θα πρέπει να χρησιμοποιηθεί μόνο για εγγραφές.

Χαρτογραφημένη μνήμη (Mapped memory): Σε ορισμένες κάρτες γραφικών, με αυτή την παράμετρο, η μνήμη μπορεί να γίνει άμεσα προσβάσιμη από αυτές. Αυτή η δυνατότητα δίνεται μόνο για να γίνεται πιο εύκολη η διερεύνηση και επίλυση προβλημάτων ενός προγράμματος στην κάρτα γραφικών, και πρέπει να χρησιμοποιείται γι αυτό τον σκοπό μόνο. Στην ειδική περίπτωση των φορητών

[14] Το OpenMP είναι μία ελεύθερη βιβλιοθήκη για πολυνηματικό προγραμματισμό. Απευθύνεται μόνο σε έναν κόμβο όμως, δηλαδή σε μία μητρική με N πυρήνες. Αν κάποιος θέλει να χρησιμοποιήσει πολλούς υπολογιστές, η εναλλακτική είναι η βιβλιοθήκη MPI η οποία λειτουργεί με processes αντί για νήματα.

υπολογιστών και μόνο θα είχε κανείς όφελος σε επιδόσεις από αυτή την μνήμη, αφού ”φυσικά” η μνήμη DRAM της κάρτας γραφικών δεσμεύεται δυναμικά από την RAM. Και πάλι όμως η χρήση της σε τελικό πρόγραμμα πρέπει να αποφεύγεται.

Τέλος και η κλειδωμένη μνήμη συστήματος δεν έρχεται χωρίς προβλήματα. Αν το παρακάνουμε με την δέσμευση τέτοια μνήμης, κινδυνεύουμε να αφήσουμε το σύστημα χωρίς το απαραίτητο μέγεθος τράπεζας σελιδοποίησης με αποτέλεσμα να έχουμε δυσάρεστα αποτελέσματα στην ταχύτητα ή ακόμα και την σταθερότητα όλου του συστήματός μας!

Η μνήμη σταθερών (Constant memory)

Η μνήμη σταθερών είναι το αντίστοιχο του ορισμού const μνήμης ενός υπολογιστή. Η μνήμη, δηλαδή, περιέχει κάποια σταθερή μεταβλητή. Προσοχή όμως εδώ, είναι σταθερή μόνο στο τμήμα του προγράμματος που εκτελείται από την κάρτα γραφικών. Εκτός αυτών των τμημάτων μπορούμε να την αλλάξουμε μέσα στην ροή του προγράμματος, και η κάρτα γραφικών θα δει κανονικά την νέα τιμή στο επόμενο τμήμα που θα χρησιμοποιηθεί. Θα μπορούσαμε πχ να περάσουμε τις διαστάσεις ενός πλέγματος σε αυτήν την μνήμη.

Η μνήμη σταθερών είναι και αυτή κομμάτι της DRAM της κάρτας γραφικών. Είναι όμως πολύ γρήγορη καθώς είναι cached, είναι βελτιοστοποιημένη για broadcasting (εκπομπή) και το κόστος ανάγνωσης της είναι αμελητέο εφόσον τηρούνται κάποιοι κανόνες. Όντας βελτιστοποιημένη για broadcasting μίας πληροφορίας σε όλα τα νήματα, αυτά θα πρέπει να διαβάζουν όλα την ίδια περιοχή μίας μεταβλητή της μνήμης σταθερών κάθε φορά. Δηλαδή, αν τα νήματα θα πρέπει να διαβάσουν την χ διάσταση ενός πίνακα, τότε η πληροφορία θα μεταδοθεί σε όλα τα νήματα με συνήθως μηδενικό κόστος[15]. Αν όμως, τα μισά πρέπει να διαβάσουν την χ διάσταση και τα άλλα μισά την y διάσταση, τότε το κόστος θα είναι ισοδύναμο με ανάγνωση από την καθολική μνήμη. Το μέγεθος της είναι μόλις 8kbyte για όλη την κάρτα γραφικών.

Η μνήμη υφής (Texture memory)

Η μνήμη υφής είναι αρκετά διαφορετική από τις μνήμες που γνωρίζουμε. Οι πολυεπεξεργαστές διαθέτουν ειδικά κυκλώματα για την επεξεργασία υφών στα γραφικά. Μέσω της μνήμης υφής έχουμε την δυνατότητα να χρησιμοποιήσουμε αυτά τα κυκλώματα ως προς όφελος μας. Η μνήμη υφής λοιπόν αποτελείται από καθολική μνήμη την οποία ”δένουμε” πάνω στα κυκλώματα υφής. Η μνήμη υφής είναι ορατή σε όλα τα νήματα, αλλά σε αντίθεση με την καθολική, μπορούν μόνο να την διαβάσουν μέσω των κυκλωμάτων αυτών. Εφόσον η μνήμη πίσω από τις

[15] Σε περίπτωση cache miss θα έχει το κόστος μίας ανάγνωσης από την καθολική μνήμη.

υφές είναι καθολική, μπορούμε να γράψουμε στο κομμάτι μνήμης που είναι δεμένη η υφή, αλλά το πιο πιθανό είναι αν πάμε να ξαναδιαβάσουμε τα δεδομένα που γράψαμε να βρούμε τα παλιά. Αυτό συμβαίνει γιατί η μνήμη υφής είναι cached και βελτιστοποιημένη για αναγνώσεις. Και πώς ανανεώνονται τα δεδομένα στην μνήμη υφής; Με το πέρας της εκτέλεσης ενός τμήματος της κάρτας γραφικών η μνήμη υφής θα ανανεωθεί αυτόματα και σε νέα κλήση ενός τμήματος (συνάρτησης) της κάρτας γραφικών θα βρούμε τις νέες τιμές.

Η μνήμη υφής έχει διαστάσεις, επομένως μπορεί να οριστεί ως μονοδιάστατη, δισδιάστατη ή τρισδιάστατη και η προσπέλαση της να γίνεται από τον αντίστοιχο αριθμό συντεταγμένων. Το κάθε στοιχείο ονομάζεται texel (texture element) και η θέση του υπολογίζεται από ειδικά κυκλώματα του πολυεπεξεργαστή. Επίσης, υποστηρίζει και μία άλλη μορφή μεταβλητών που ορίζονται στις κάρτες γραφικών, τα διανύσματα. Ένα texel τελικά μπορεί να επιστέφει μία τιμή ή ένα διάνυσμα διάστασης δύο ή τέσσερα.

Μία υφή μπορεί να είναι δεμένη σε ένα κλασσικό γραμμικό κομμάτι μνήμης ή σε έναν συμπαγή τύπο πίνακα που ονομάζεται CUDA array. Στην περίπτωση που είναι δεμένη σε γραμμικό κομμάτι μνήμης, αν γράψουμε σε αυτό το κομμάτι μέσα σε ένα τμήμα του προγράμματος της κάρτας γραφικών, η υφή θα ενημερωθεί αυτόματα με το πέρας αυτού. Τα πράγματα δεν είναι τόσο απλά όμως και για τα CUDA Arrays, σε αυτά δεν έχουμε πρόσβαση εγγραφής παρά μόνο μέσω ειδικών συναρτήσεων εντός του τμήματος του προγράμματος που τρέχει στον κεντρικό επεξεργαστή. Ο λόγος είναι ότι τα CUDA Arrays δεν μπορούμε να τα δούμε ως γραμμική μνήμη. Τα δεδομένα μέσα στα συμπαγή CUDA Arrays είναι κατανεμημένα με ειδικούς αλγορίθμους[16], ώστε να επιτυγχάνεται χωρικός εντοπισμός. Αν φανταστούμε έναν μικρό κύβο και το κέντρο του, τα στοιχεία θα είναι αποθηκευμένα έτσι ώστε πάντα τα στοιχεία γύρω από αυτό το κέντρο να είναι κοντά σε αυτό από την σκοπιά της γενικής μονοδιάστατης αναπαράστασης.

Για να το εξηγήσουμε αυτό καλύτερα πρέπει να αναφέρουμε, ότι πάντα κάθε μνήμη σε υπολογιστή και σε κάρτα γραφικών, είναι μονοδιάστατη. Αν ορίσουμε έναν πίνακα τριών διαστάσεων με Nx, Ny, Nz την κάθε μία και τοποθετήσουμε για κάθε z όλα τα (y_i, z) και θα κάθε y όλα τα (x_i, y), τότε ο μετασχηματισμός από τις εικονικές τρεις διαστάσεις στην "φυσική" μία θα είναι:

$$x + N_x * (y + N_y * z)$$

Εδώ βλέπουμε ότι ακόμα και για μικρές διαστάσεις του πίνακα, από την μονοδιάστατη σκοπιά, ένα στοιχείο θα είναι πάντα μακριά από ένα σε θέση $y + 1$ ή

[16]Χρησιμοποιούνται ειδικοί αλγόριθμοι Z-reordering για την τοποθέτηση των δεδομένων.

ειδικότερα $z + 1$ [17].

Μία από τις ιδιότητες της μνήμης υφής είναι ότι δεν περιορίζεται στην προσπέλαση μέσω των "κλασσικών" συντεταγμένων όπως συμβαίνει με όλες τις μνήμες. Μπορεί να προσπελαστεί με αριθμούς κινητής υποδιαστολής (float) ως συντεταγμένες, ακόμα και κανονικοποιημένους ως προς την διάσταση[18].

Όταν χρησιμοποιούμε αριθμούς κινητής υποδιαστολής ως συντεταγμένες, τότε η θέση ενός texel είναι το κέντρο του (σχήματα 3.12 και 3.12). Δηλαδή, το στοιχείο που έχει συντεταγμένες (x, y) υπό κανονικές συνθήκες, τώρα θα έχει συντεταγμένες $(x+0,5, y+0.5)$. Μην ξεχνάμε ότι οι κάρτες γραφικών φτιάχτηκαν για γραφικά, και εξαιτίας αυτού στα κυκλώματα υφής, υπάρχουν ολοκληρωμένα που αναλαμβάνουν να εκτελούν γραμμικές παρεμβολές (linear interpolation). Έτσι λοιπόν, αν δώσουμε συντεταγμένες (x, y) αντί για $(x + 0.5, y + 0.5)$ αυτά θα αναλάβουν να μας δώσουν το αποτέλεσμα της δισδιάστατης γραμμικής παρεμβολής (bilinear interpolation) των τιμών των κοντινών texel!

Στην μνήμη υφής προβλέπεται και η συμπεριφορά σε περίπτωση υπερχείλισης της μνήμης[19]. Υπάρχουν δύο επιλογές για τι θα μας επιστρέψει η υφή σε μία ανάγνωση εκτός των ορίων της μνήμης. Η πρώτη ονομάζεται Clamp και αναφέρεται σε μνήμη υφής δεμένη σε γραμμική μνήμη ή σε CUDA Arrays. Σε αυτή την περίπτωση, η υφή θα μας επιστρέψει την τιμή του κοντινότερου συνόρου - texel (σχήμα 3.11). Η δεύτερη ονομάζεται Wrap και είναι και η πιο ενδιαφέρουσα, αφού θα μας επιστρέψει την τιμή του texel που βρίσκεται στις συντεταγμένες που δώσαμε, μείον την διάσταση της συντεταγμένης που υπερχειλίσαμε (σχήμα 3.12).

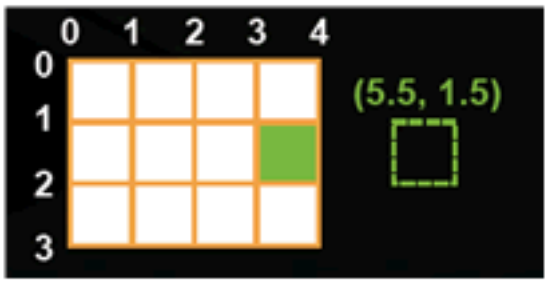
Η συμπεριφορά υπερχείλισης Wrap μας είναι ιδιαίτερα ενδιαφέρουσα αφού μας δίνει τις περιοδικές συνθήκες ενός πλέγματος χωρίς να χρειαστεί να κάνουμε εμείς ελέγχους, θα γίνουν όλα αυτόματα από εξειδικευμένα κυκλώματα! Επίσης, όπως είπαμε η μνήμη υφής είναι βελτιστοποιημένη για χωρικό εντοπισμό. Επομένως, οι γείτονες μίας πλεγματικής θέσης θα είναι πάντα "ετοιμοπαράδοτοι" (cached), ακόμα και αν είναι στην άλλη πλευρά του πλέγματος!

Εδώ κλείνουμε την συζήτηση μας περί των διαθέσιμων μνημών σε μία κάρτα γραφικών. Είμαι σίγουρος ότι μπερδευτήκατε, όπως και όλοι στην αρχή. Οπότε εδώ θα ανακεφαλαιώσουμε τα βασικά με την ελπίδα ότι θα ξεκαθαρίσουν κάποια πράγματα.

[17] Εδώ ο x ονομάζεται γρήγορος δείκτης, ενώ ο z αργός.

[18] παίρνει πραγματικές τιμές στο διάστημα $[0..1)$ αντί για $[0..N)$

[19] Υπερχείλιση μνήμης (overflow) έχουμε όταν προσπαθήσουμε να προσπελάσουμε εκτός τον ορίων της δεσμευμένης μνήμης μίας μεταβλητής.



Σχήμα 3.11: Υπερχείλιση μνήμης υφής με μέθοδο Clamp.

1. Στην καθολική μνήμη έχουν πρόσβαση ανάγνωσης και εγγραφής όλα τα νήματα.

2. Στην κοινόχρηστη μνήμη έχουν πρόσβαση ανάγνωσης και εγγραφής μόνο τα νήματα. του block στο οποίο αυτή ορίζεται. Έχει την ζωή του block.

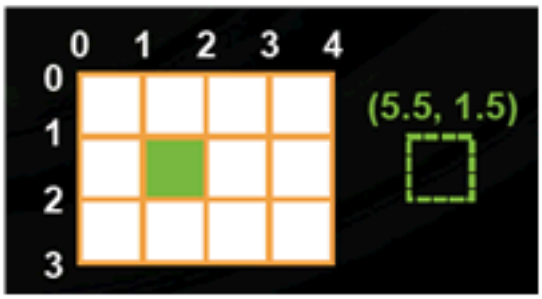
3. Στην τοπική μνήμη και στους καταχωρητές έχει πρόσβαση ανάγνωσης και εγγραφής μόνο το νήμα στο οποίο αυτή είναι ορισμένη. Έχει την ζωή του νήματος.

4. Στην κλειδωμένη μνήμη συστήματος, μαζί με τις παραλλαγές της, δεν έχει πρόσβαση κανένα νήμα (εκτός μίας περίπτωσης που είναι για debugging). Υπάρχει κατά κύριο λόγο για να ενεργοποιείται η δυνατότητα ταυτόχρονης μεταφοράς και εκτέλεσης αλλά και για να επιταχύνει τις μεταφορές δεδομένων από τον επεξεργαστή στην κάρτα γραφικών και αντίστροφα.

5. Στην μνήμη σταθερών έχουν πρόσβαση ανάγνωσης όλα τα νήματα.

6. Στην μνήμη υφής έχουν πρόσβαση ανάγνωσης όλα τα νήματα.

Η καθολική μνήμη είναι η βασική μνήμη που θα χρησιμοποιήσουμε, καθώς είναι ο μόνος τρόπος να περάσουμε τα δεδομένα μας στην κάρτα γραφικών και να τα επιστρέψουμε πίσω στο σύστημα. Πρέπει οι προσβάσεις σε αυτή να είναι ευθυγραμμισμένες στο όριο των 32, 64 ή 128 byte από την αρχή της μνήμης και γραμμικές. Αν δεν το καταφέρουμε αυτό, ένας τρόπος είναι να βάλουμε όλα τα νήματα στην αρχή να διαβάσουν την καθολική μνήμη με συνεχή τρόπο και να τα αποθηκεύσουν στην κοινόχρηστη μνήμη. Αν δεν μπορούμε να χειριστούμε τα δεδομένα σωστά και μέσω της κοινόχρηστης μνήμης τότε προσπαθούμε να κάνουμε χρήση της μνήμης υφής. Η μνήμη καταχωρητών πρέπει να χρησιμοποιείται με όριο για να μην έχουμε προβλήματα στην πληρότητα, αλλά δεν πρέπει να περιορίζεται πολύ για να αποφύγουμε το ”spilling” στην αργή τοπική μνήμη. Την μνήμη σταθερών πρέπει να την χρησιμοποιούμε όταν βλέπουμε ότι όλα τα νήματα θα διαβάσουν την ίδια θέση στην μνήμη.



Σχήμα 3.12: Υπερχείλιση μνήμης υφής με μέθοδο Wrap.

3.1.4 Ασύγχρονη και ταυτόχρονη εκτέλεση

Ταυτόχρονη εκτέλεση μεταξύ CPU και GPU

Πολλές από τις συναρτήσεις τις CUDA που τρέχουνε σε μία κάρτα γραφικών, είναι ασύγχρονες ως προς τον κεντρικό επεξεργαστή. Με το που ξεκινήσει η εκτέλεση μίας τέτοια συνάρτησης, ο έλεγχος επιστρέφεται αμέσως πίσω στον κεντρικό επεξεργαστή. Αυτό με λίγα λόγια σημαίνει ότι ο κεντρικός επεξεργαστής δεν την περιμένει να ολοκληρωθεί για να συνεχίσει να εκτελεί τις επόμενες εντολές όπως συμβαίνει στα συμβατικά προγράμματα. Στο σχήμα 3.13 φαίνεται η γενική ιδέα του ετερογενούς προγραμματισμού, αρκεί να φανταστούμε ότι το κομμάτι που εκτελείται στην κάρτα γραφικών (device) επικαλύπτει το τμήμα που εκτελείται αμέσως μετά από τον κεντρικό επεξεργαστή (host) έως ότου ο τελευταίος βρεθεί αντιμέτωπος με μία άλλη κλήση συνάρτησης της κάρτας γραφικών ή σε μία από τις ειδικές συναρτήσεις που επιβάλλουν συγχρονισμό μεταξύ των δύο συσκευών. Συναρτήσεις με τέτοια ασύγχρονη συμπεριφορά ως προς τον κεντρικό επεξεργαστή είναι:

- Οι συναρτήσεις κλήσης προγράμματος προς την κάρτα γραφικών (kernels)
- Οι μεταφορές/αντιγραφές μνήμης εσωτερικά στην κάρτα γραφικών
- Οι συναρτήσεις που δεσμεύουν μνήμη στην κάρτα γραφικών
- Μία κατηγορία συναρτήσεων μεταφοράς μνήμης μεταξύ επεξεργαστή - κάρτας γραφικών, που ονομάζονται async.

Αλληλεπικάλυψη μεταφορών μνήμης και εκτέλεσης συναρτήσεων kernel

Μερικές συσκευές με υπολογιστική ικανότητα 1.1 και άνω, έχουν την δυνατότητα να μεταφέρουν δεδομένα μεταξύ κλειδωμένης μνήμης συστήματος και την μνήμη της συσκευής DRAM. Αυτός ο τρόπος μεταφοράς υποστηρίζεται μόνο για γραμμική μνήμη (όχι CUDA Arrays).

Ταυτόχρονη εκτέλεση συναρτήσεων kernel

Μερικές συσκευές υπολογιστικής ικανότητας 2.0 (αρχιτεκτονική Fermi) μπορούν να εκτελέσουν πάνω από μία συναρτήσεις εκτέλεσης δεδομένων (kernel) ταυτόχρονα, έως τέσσερις συγκεκριμένα. Το πόσες συναρτήσεις θα "φορτωθούν" και θα εκτελεσθούν ταυτόχρονα στην κάρτα γραφικών εξαρτάται από τις απαιτήσεις τους. Εδώ τίθεται άλλος ένας περιορισμός, όπως έχουμε αναφέρει, η κάρτα γραφικών ελέγχεται από τον κεντρικό επεξεργαστή. Πιο συγκεκριμένα όταν ένα νήμα του κεντρικού επεξεργαστή εκτελεί την πρώτη συνάρτηση που σχετίζεται με την κάρτα γραφικών (πχ για μεταφορά μνήμης), τότε αυτό το νήμα δημιουργεί ένα "πλαίσιο" (context) με την κάρτα γραφικών. Οι συναρτήσεις kernel δεν πρέπει να ανήκουν στο ίδιο πλαίσιο για να μπορούν να εκτελεστούν ταυτόχρονα. Δηλαδή, αν εκτελέσουμε τέσσερα ανεξάρτητα προγράμματα που χρησιμοποιούν την κάρτα γραφικών τότε οι συναρτήσεις kernel τους πιθανό να μπορέσουν να εκτελεστούν ταυτόχρονα. Αντίθετα σε ένα πρόγραμμα δεν μπορούμε να εκτελέσουμε τέσσερις συναρτήσεις kernel ταυτόχρονα εκτός και αν χρησιμοποιούμε κάποια πολυνηματική βιβλιοθήκη όπως το OpenMP[20], όπου αν ορίσουμε 4 νήματα και καθένα δημιουργήσει το δικό της πλαίσιο με την κάρτα, οι συναρτήσεις kernel θα μπορούν να εκτελεστούν ταυτόχρονα.

Σε όλες τις παλαιότερες αρχιτεκτονικές, σε όλες τις περιπτώσεις μία συνάρτηση kernel θα περιμένει πάντα την προηγούμενη διεργασία στην κάρτα γραφικών να τελειώσει, ανεξάρτητα αν είναι από άλλο πρόγραμμα ή από το ίδιο αλλά από διαφορετικό πλαίσιο.

Ταυτόχρονες μεταφορές μνήμης

Οι συσκευές με υπολογιστική ικανότητα 2.0 μπορούν να εκτελέσουν ταυτόχρονα μία αντιγραφή μνήμης από κλειδωμένη μνήμη συστήματος στην μνήμη της κάρτας γραφικών μαζί με μία αντιγραφή μνήμης από την μνήμη σε κλειδωμένη μνήμη συστήματος.

Τα ρεύματα (streams)

Ένα ρεύμα είναι ένα σύνολο εντολών οι οποίες εκτελούνται σε σειρά. Διαφορετικά ρεύματα όμως θα μπορούν να εκτελέσουν τις εντολές τους εκτός σειράς η μία ως προς την άλλη, ή ακόμα και παράλληλα. Στην υποενότητα 3.1.3 και στο σχήμα 3.10 ουσιαστικά μιλήσαμε για ρεύματα. Τώρα που μιλήσαμε για ταυτόχρονη μεταφορά μνήμη και εκτέλεση αλλά και τα ρεύματα, μπορούμε να αναλύσουμε αυτό το σχήμα καλύτερα. Όπως βλέπουμε, στην πρώτη περίπτωση, πρώτα εκτελείται μία συνάρτηση kernel και μετά γίνεται η αντιγραφή των δεδομένων από την καθολική μνήμη πίσω στην κλειδωμένη μνήμη συστήματος. Αν χωρίσουμε την συνάρτηση

[20] http://openmp.org/wp/

kernel σε τέσσερα ανεξάρτητα τμήματα και τα ορίσουμε ως διαφορετικά ρεύματα, τότε μπορούμε να επιστρέφουμε το αποτέλεσμα της κάθε συνάρτησης kernel ενώ εκτελείται η συνάρτηση kernel του επόμενου ρεύματος. Βλέπουμε πως μπορεί να μικρύνει ο συνολικός χρόνος εκτέλεσης!

3.1.5 Πληρότητα των πολυεπεξεργαστών και τεχνικά χαρακτηριστικά

Αναφέραμε στην υποενότητα 3.1.3 την πληρότητα των πολυεπεξεργαστών. Η πληρότητα ενός πολυεπεξεργαστή ορίζεται ως:

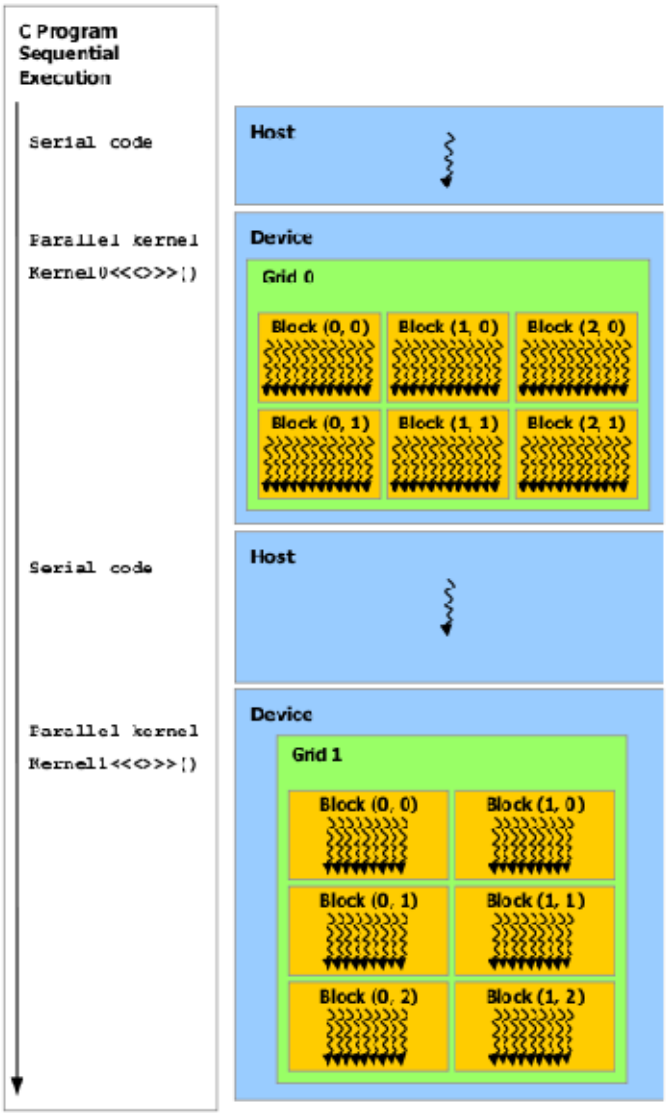
$$\frac{Warps_{effective}}{Warps_{maximum}}$$

όπου warps είναι οι δύνες των νημάτων. $Warps_{effective}$ είναι ο αριθμός τον ενεργών δυνών και $Warps_{maximum}$ ο μέγιστος αριθμός των ενεργών δυνών που ορίζεται από τα τεχνικά χαρακτηριστικά της κάρτας γραφικών. Ο αριθμός των ενεργών δυνών εξαρτάται από της απαιτήσεις και την παραμετροποίηση του προγράμματος μας:

- Το μέγεθος μνήμης καταχωρητών ανά νήμα.
- Το μέγεθος της κοινόχρηστης μνήμης ανά block.
- Το μέγεθος ενός block (σε νήματα).
- Το μέγιστο πλήθος νημάτων ανά πολυεπεξεργαστή.
- Την υπολογιστική ικανότητα της κάρτας γραφικών (που σχετίζεται και με τα δύο πρώτα).

Θα αναφέρουμε εδώ ως παράδειγμα μία κάρτα γραφικών με υπολογιστική ικανότητα 1.3. Κάποιος μπορεί να κάνει τους ίδιους υπολογισμούς παίρνοντας τις παραμέτρους από τους πίνακες 3.1 και 3.2 για οποιαδήποτε άλλη κάρτα γραφικών αρχιτεκτονικής CUDA.

Αρχικά θα ξεκινήσουμε θεωρώντας ότι έχουμε ένα πρόγραμμα παραμετροποιημένο έτσι ώστε ένα block να έχει διαστάσεις 16*16, δηλαδή 256 νήματα. Έχουμε ορίσει το μέγεθος της κοινόχρηστης μνήμης να είναι 1024 στοιχεία ακεραίων (4byte το καθένα) και στην συνέχεια μετράμε πόσες μεταβλητές που δεν είναι ορισμένες ως κοινοχρηστες και δεν είναι ορίσματα της συνάρτησης υπάρχουν μέσα στην συνάρτηση. Αυτό μας δίνει μία γενική ιδέα του πλήθους των καταχωρητών. Στην υποενότητα 3.1.3 αναφέραμε ότι αυτό δεν είναι αρκετό, καθώς τι πάει στους καταχωρητές και τι όχι ελέγχεται από τον μεταγλωττιστή. Παρόλα αυτά, για τώρα θα



Σχήμα 3.13: Ετερογενής προγραμματισμός.

κάνουμε την απλούστευση ότι καταχωρητές είναι όσοι μετρήσαμε +4. Ας πούμε ότι μετρήσαμε 14 + 4. Επομένως έχουμε 18 καταχωρητές. Ας τα βάλουμε όλα κάτω τώρα για να υπολογίσουμε την πληρότητα.

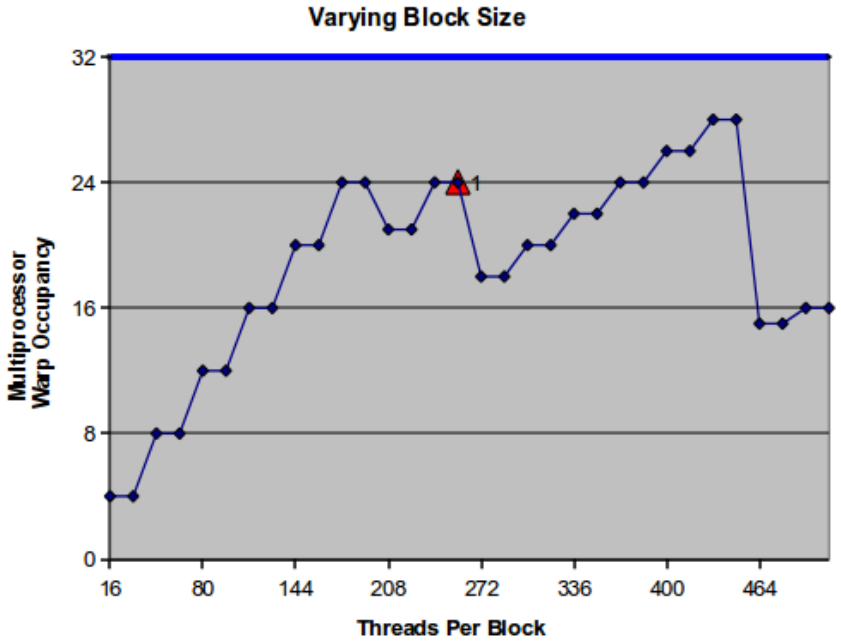
Σε κάρτα γραφικών με υπολογιστική ικανότητα 1.3, το μέγιστο πλήθος των ενεργών νημάτων ανά πολυεπεξεργαστή είναι 1024. Επομένως, λαμβάνοντας μόνο αυτή την παράμετρο υπόψη, θα μπορούν να φορτωθούν 4 block στον πολυεπεξεργαστή. Το όριο των blocks που μπορούν να επεξεργαστούν παράλληλα από έναν πολυεπεξεργαστή είναι 8. Επομένως, δεν έχουμε νήματα τα οποία δεν "χωράνε" στον πολυεπεξεργαστή. Ωραία ως εδώ, πάμε στην κοινόχρηστη μνήμη. Ορίσαμε 1024 στοιχεία ανά block των 4byte (integer, float), επομένως 4Kbyte κοινόχρηστης μνήμης χρειάζονται για κάθε block. Η διαθέσιμη μνήμη σε έναν πολυεπεξεργαστή είναι 16kbyte, επομένως λαμβάνοντας μόνο αυτή την παράμετρο υπόψη, μπορούν να φορτωθούν 4 blocks στον πολυεπεξεργαστή.

Τέλος ας πάμε στους καταχωρητές. Το πλήθος 32bit καταχωρητών ανά πολυεπεξεργαστή είναι 16.384. Εδώ πρέπει να πάρουμε τα προηγούμενα αποτελέσματα, αν τα συνδυάσουμε βρίσκουμε ότι μπορούν φορτωθούν 4 blocks. Τα 4 αυτά blocks περιέχουν 1024 νήματα και το καθένα θέλει 18 καταχωρητές. Επομένως, αμέσως βλέπουμε ότι $18 * 1024 > 16.384$. Όπως έχουμε πει, δεν ορίζονται "ελεύθερα" νήματα, επομένως αφαιρούμε ένα block και δοκιμάζουμε πάλι $18 * 768 < 16.384$. Τώρα βρήκαμε τον μέγιστο αριθμό των νημάτων (block * μέγεθος του block) που θα εκτελεστούν σε έναν πολυεπεξεργαστή. Αν διαιρέσουμε το αποτέλεσμα με το μέγεθος μίας δύνης που είναι 32 βρίσκουμε αμέσως 24 δύνες.

Για τις κάρτες γραφικών με υπολογιστική ικανότητα 1.3 το μέγιστο πλήθος ενεργών δυνών είναι 32. Επομένως, αφού θα εκτελούνται μόνο 24 δύνες κάθε φορά, η πληρότητα είναι 75%. Στο σχήμα 3.14 βλέπουμε το γράφημα των ενεργών δυνών για διάφορα μεγέθη του block με τις παραμέτρους που ορίσαμε στο παράδειγμα!

Δεν χρειάζεται πραγματικά να κάνουμε όλη αυτή την ανάλυση, υπάρχουν εργαλεία τα οποία θα δούμε στην υποενότητα 3.3 και κάνουν όλους αυτούς τους μπερδεμένους υπολογισμούς (αφού πολλά μεγέθη εξαρτώνται σχετικά μεταξύ τους) αυτόματα. Αυτό που χρειάζεται όμως είναι να έχουμε μία γενική ιδέα του πως θα επιλέξουμε τις παραμέτρους για τον αλγόριθμό μας. Τις λεπτομέρειες θα της κοιτάξουμε αφού είμαστε σε σημείο να μεταγλωττίσουμε το πρόγραμμα μας και επομένως ο μεταγλωττιστής να μας πει ακριβώς πόσοι καταχωρητές χρειάστηκαν. Τότε θα βάλουμε όλα τα δεδομένα στον υπολογιστή πληρότητας και θα μας πει την πραγματική πληρότητα και από κει θα κάνουμε ότι αλλαγές είναι απαραίτητες.

Ποιος είναι ο ρόλος της πληρότητας; Ο ρόλος της πληρότητας είναι να κρατήσει τους πολυεπεξεργαστές απασχολημένους έτσι ώστε να κρυφτούν οι καθυστερή-



Σχήμα 3.14: Πως θα μεταβληθεί η πληρότητα για διαφορετικά μεγέθη block.

σεις που προκαλούνται από αριθμητικές διεργασίες ή συναλλαγές με την μνήμη. Ένας αλγόριθμος με 100% πληρότητα δεν σημαίνει ότι θα είναι πιο γρήγορος από τον ίδιο αλγόριθμο με 75% πληρότητα. Στην ουσία, θέλουμε η πληρότητα να είναι πάνω από 50% αν ο αλγόριθμος είναι περιορισμένος από τις αριθμητικές διεργασίες, ή πάνω από 75% αν ο αλγόριθμος είναι κάνει πολλές συναλλαγές με την μνήμη. Σημασία πάνω από όλα έχει οι συναλλαγές με την μνήμη να είναι όσο το δυνατόν πιο ευθυγραμμισμένες και γραμμικές γίνεται. Η πληρότητα έχει δευτερεύοντα ρόλο στην αποδοτικότητα του αλγορίθμου, αρκεί να είναι πάντα πάνω απο 50%.

Τώρα θα προσπαθήσουμε να συμμαζέψουμε όλα τα τεχνικά χαρακτηριστικά κάθε υπολογιστικής ικανότητας στον πίνακα 3.1. Αξίζει να σημειωθεί μία επίσης πολύ σημαντική διαφορά μεταξύ των καρτών γραφικών υπολογιστικής ικανότητας 1.1 και κάτω με τις κάρτες γραφικών υπολογιστικής ικανότητας 1.2 και άνω. Οι δεύτερες υποστηρίζουν αριθμούς κινητής υποδιαστολής διπλής ακρίβειας (double ή 64bit float).

Στον πίνακα 3.2 θα αναφέρουμε μερικά από τα γενικότερα τεχνικά χαρακτηριστικά (υπολογιστική ικανότητα, αριθμός πολυεπεξεργαστών και συνολικός αριθμός πυρήνων) για μερικές από τις πιο ευρέως διαθέσιμες κάρτες γραφικών αρχιτεκτονικής CUDA.

	Υπολογιστική ικανότητα				
Τεχνικές προδιαγραφές	1.0	1.1	1.2	1.3	2.0
Μέγιστη x,y διάσταση ενός grid	65535				
Μέγιστος αριθμός νημάτων ανά block	512				1024
Μέγιστη x,y διάσταση ενός block	512				1024
Μέγιστη z διάσταση ενός block	64				
Μέγεθος δύνης (σε νήματα)	32				
Μέγιστος αριθμός blocks ανά πολυεπεξεργαστή	8				
Μέγιστος αριθμός δυνών ανά πολυεπεξεργαστή	24		32		48
Μέγιστος αριθμός νημάτων ανά πολυεπεξεργαστή	768		1024		1536
Πλήθος καταχωρητών 32bit ανά πολυεπεξεργαστή	8K		16K		32K
Μέγιστη κοινόχρηστη μνήμης ανά πολυεπεξεργαστή	16 KB				48 KB
Πλήθος τραπεζών της κοινόχρηστης μνήμης	16				32
Μέγεθος της τοπικής μνήμης ανά νήμα	16 KB				512 KB
Μέγεθος της μνήμης σταθερών	64 KB				
Cache της μνήμης σταθερών ανά πολυεπεξεργαστή	8 KB				
Cache της μνήμης υφής ανά πολυεπεξεργαστή	6 KB έως 8 KB				
Μέγιστο μήκος 1D υφής δεμένης πάνω σε CUDA Array	8192				32768
Μέγιστο μήκος 1D υφής δεμένης πάνω σε γραμμική μνήμη	2^{27} στοιχεία				
Μέγιστες διαστάσεις 2D υφής δεμένης πάνω σε γραμμική μνήμη ή CUDA Array	65536 x 32768				65536 x 65536
Μέγιστες διαστάσεις 3D υφής δεμένης πάνω σε γραμμική μνήμη ή CUDA Array	2048 x 2048 x 2048				4096 x 4096 x 4096
Μέγιστο πλήθος εντολών σε μια συνάρτηση kernel	2 εκατομμύρια				

Πίνακας 3.1: Τεχνικά χαρακτηριστικά των διαφόρων υπολογιστικών ικανοτήτων.

	Υπολογιστική ικανότητα	Πλήθος πόλυ-επεξεργαστών	Πλήθος πυρήνων
GeForce GTX 295	1.3	2x30	2x240
GeForce GTX 285, GTX 280	1.3	30	240
GeForce GTX 260	1.3	24	192
GeForce 9800 GX2	1.1	2x16	2x128
GeForce GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 8800 GTS 512, GTX 285M, GTX 280M	1.1	16	128
GeForce 8800 Ultra, 8800 GTX	1.0	16	128
GeForce 9800 GT, 8800 GT, GTX 260M, 9800M GTX	1.1	14	112
GeForce GT 240, GTS 360M, GTS 350M	1.2	12	96
GeForce GT 130, 9600 GSO, 8800 GS, 8800M GTX, GTS 250M, 9800M GT	1.1	12	96
GeForce 8800 GTS	1.0	12	96
GeForce GT 335M	1.2	9	72
GeForce 9600 GT, 8800M GTS, 9800M GTS	1.1	8	64
GeForce GT 220, GT 330M, GT 325M	1.2	6	48
GeForce 9700M GT, GT 240M, GT 230M	1.1	6	48
GeForce GT 120, 9500 GT, **8600 GTS**, 8600 GT	1.1	4	32
GeForce 210, 310M, 305M	1.2	2	16
GeForce G100, 8500 GT, 8400 GS, GT, 9500M G, 9300M G	1.1	2	16
GeForce **9300M GS**, 9200M GS, 9100M G, 8400M G, G105M	1.1	1	8
Tesla S1070	1.3	4x30	4x240
Tesla C1060	1.3	30	240

Πίνακας 3.2: Τεχνικά χαρακτηριστικά των πιο δημοφιλών καρτών αρχιτεκτονικής CUDA. Οι επισημανθείς κάρτες γραφικών χρησιμοποιήθηκαν στις προσομοιώσεις.

3.2 CUDA/C Μία επέκταση των C/C++

Αφού πλέον έχουμε μία ιδέα για το πώς θα τρέξει μία εφαρμογή σε μια κάρτα γραφικών CUDA, ας αρχίσουμε να μαθαίνουμε πώς θα προγραμματίσουμε μία τέτοια εφαρμογή.

3.2.1 Εισαγωγή στην γλώσσα προγραμματισμού CUDA/C καί το CUDA API

Η γλώσσα προγραμματισμού CUDA/C διαφέρει από τις C/C++ κυρίως στο ότι είναι γλώσσα ετερογενούς προγραμματισμού. Από την μία πλευρά ο κεντρικός επεξεργαστής (στην CUDA/C ονομάζεται host) εκτελεί σειριακό κώδικα και έχει όψη της μνήμης όπως στα κλασσικά προγράμματα, και από την άλλη η μονάδα επεξεργασίας γραφικών (device) εκτελεί παράλληλο κώδικα και έχει ειδική πρόσβαση σε περιοχές της μνήμης. Αυτό το είδος προγραμματισμού το είδαμε στο σχήμα 3.13. Η γλώσσα CUDA/C απευθύνεται μόνο σε κάρτες γραφικών της εταιρίας NVIDIA. Η έννοια του ετερογενούς προγραμματισμού δεν περιορίζεται στην CUDA/C, υπάρχουν αρκετές άλλες γλώσσες προγραμματισμού, όπως η OpenCL,η Direct Compute και η Brook++, αλλά και γλώσσες προγραμματισμού που δεν απευθύνονται καν σε κάρτες γραφικών αλλά σε εξειδικευμένα κυκλώματα FPGA σε παράλληλη χρήση με τον κεντρικό επεξεργαστή.

Το CUDA API είναι ένα σύνολο συναρτήσεων και ορισμών που αναλαμβάνουν τον συνδετικό ρόλο ανάμεσα στον κεντρικό επεξεργαστή και την κάρτα γραφικών. Υπάρχουν τρία είδη του API, το API εκτέλεσης σε χαμηλό επίπεδο, το API εκτέλεσης σε υψηλό επίπεδο και τέλος το API εκτέλεσης σε επίπεδο οδηγού της κάρτας. Τα δύο πρώτα διαφέρουν ελάχιστα στον προγραμματισμό, ουσιαστικά είναι τα ίδια, με το δεύτερο να δίνει κάποιες από της δυνατότητας της C++ στον κώδικα για την κάρτα γραφικών. Αντίθετα το API εκτέλεσης σε επίπεδο οδηγού (driver) της κάρτας γραφικών είναι πολύ διαφορετικό και δεν θα μας απασχολήσει καθόλου καθώς είναι αρκετά πιο πολύπλοκο και χρησιμοποιείται μόνο σε ειδικές περιπτώσεις.

Εν τέλει, η CUDA/C ελάχιστα διαφέρει από την C στο μεγαλύτερο κομμάτι της, γεγονός που την κάνει εξαιρετικά εύκολη στην εκμάθηση αφήνοντας έτσι το κύριο μέρος της προσπάθειας εκμάθησης προγραμματισμού σε κάρτα γραφικών να είναι το πως επεξεργάζεται η κάρτα γραφικών τις εντολές μας και πως θα την παραμετροποιήσουμε σωστά για τις ανάγκες του προγράμματος μας.

3.2.2 Νέοι τύποι μεταβλητών και μερικές βασικές συναρτήσεις

Αρχικά, θα ξεκινήσουμε με τους νέους τύπους μεταβλητών και της ειδικές συναρτήσεις που θα συναντήσουμε αρκετές φορές από εδώ και στο εξής.

Μεταβλητή τύπου enum cudaMemcpyKind kind

Ορίζει την κατεύθυνση αντιγραφής της μνήμης. Χρησιμοποιείται μόνο στις ειδικές συναρτήσεις αντιγραφής δεδομένων. Οι επιλογές είναι:

- cudaMemcpyHostToDevice: Αντιγραφή από την μνήμη RAM στην καθολική μνήμη. Είναι σύγχρονη ως προς τον κεντρικό επεξεργαστή.
- cudaMemcpyDeviceToHost: Αντιγραφή από την καθολική μνήμη στην μνήμη Ram. Επίσης είναι σύγχρονη ως προς τον κεντρικό επεξεργαστή.
- cudaMemcpyDeviceToDevice: Εσωτερική αντιγραφή μεταξύ δύο δεικτών καθολικής μνήμης. Είναι ασύγχρονη ως προς τον κεντρικό επεξεργαστή.

Μεταβλητές τύπου cudaError_t και οι συναρτήσεις χειρισμού σφαλμάτων

Σχεδόν κάθε συνάρτηση της CUDA/C επιστρέφει μία μεταβλητή τύπου cudaError_t με το πέρας της εκτέλεσης της. Οι μεταβλητές cudaError_t χρησιμεύουν για τον χειρισμό σφαλμάτων. Είναι εσωτερικός τύπος της CUDA/C και δεν μας αφορά η δομή της. Μας ενδιαφέρει μόνο πως θα δούμε αν η συνάρτηση μας εκτελέστηκε με επιτυχία ή επέστρεψε κάποιο μήνυμα σφάλματος. Για να το κάνουμε αυτό, υπάρχουν δύο συναρτήσεις χειρισμού σφαλμάτων:

const char * cudaGetErrorString (cudaError_t error) : Αποκωδικοποιεί και επιστρέφει το μήνυμα λάθους μίας cudaError_t μεταβλητής.

cudaError_t cudaGetLastError(void) : Επιστρέφει το τελευταίο σφάλμα από την εκτέλεση μίας συνάρτησης της CUDA/C.

Ας τις δούμε στην πιο λεπτομερή μορφή:

```
cudaError_t error;
char error_text[100];
error = CUDA_function();
strcpy(error_text,cudaGetErrorString(error));
printf("H cuda_function epestrepse katastash %s\n",error_text);
```

Επειδή τέτοιους ελέγχους θα πρέπει να κάνουμε συχνά, υπάρχουν πιο συμπαγείς τρόποι να κάνουμε έναν τέτοιο έλεγχο:

```
Cuda_function();
printf("error? %s n",cudaGetErrorString(cudaGetLastError()));
```

Αντίστοιχα μπορεί να ορίσει κάποιος μία μακροεντολή (macro) προεπεξεργαστή (preprocessor) όπως θα έκανε και με την απλή C. Μπορούμε να ορίσουμε μία μακροεντολή σαν την παρακάτω:

```
#define safecall(call)\
    cudaError_t err = call ; \
    if (cudaSuccess != err){\
        fprintf(stderr, "cuda error at %s:%d, %s\n",\
            __FILE__, __LINE__, cudaGetErrorString(err));\
    }
```

Μεταβλητές τύπου *dim3* και η συνάρτηση *dim3()*

Οι μεταβλητές *dim3* είναι διανύσματα ακεραίων (32bit integer) με διάσταση 3. Βάση αυτών ορίζουμε ένα από τα πιο βασικά μεγέθη στην *CUDA*, τις διαστάσεις των *blocks* και των *grids*. Οι μεταβλητές *dim3* παίρνουν τιμή μέσω της συνάρτησης:
dim3(*int* x, *int* y, *int* z)
και μπορούμε να ορίσουμε μία μεταβλητή τύπου dim3 όπως θα κάναμε με κάθε άλλο τύπο:
dim3 variable;
Μπορούμε να δώσουμε και άμεσα τιμές στα στοιχεία του διανύσματος αρκεί να λάβουμε υπόψη την αναπαράσταση της:

```
struct dim3{
    int x;
    int y;
    int z;
}
```

Ας δούμε τώρα ένα παράδειγμα. Ας θεωρήσουμε ότι έχουμε έναν πίνακα διάστασης (N_x, N_y) με δεδομένα τα οποία θέλουμε να μοιράσουμε έτσι ώστε κάθε νήμα να επεξεργάζεται ένα στοιχείο. Έχοντας δει ως τώρα ότι γενικά έναν καλός αριθμός για το μέγεθος ενός block είναι 256 νήματα θα το εφαρμόσουμε και στο παράδειγμα μας.
Ξεκινάμε ορίζοντας το μέγεθος του block:

```
dim3 blocksize = dim3(16,16,1);
```

Τώρα θα ορίσουμε και το μέγεθος του grid:

Listing 3.1: Ορισμός του μεγέθους του grid

```
/*Idaniki periptwsi,
oi diastaseis tou pinaka diairountai akrivws me tis diastaseis tou block:*/
dim3 gridsize = dim3(Nx/blocksize.x,Ny/blocksize.y,1);

/*Mi idaniki periptwsi:*/
if{Nx % blocksize.x} gridsize.x = Nx/blocksize.x + 1;
else gridsize.x = Nx/blocksize.x;\\
if{Ny % blocksize.y} gridsize.y = Ny/blocksize.y + 1;
else gridsize.y = Ny/blocksize.y;
```

Άλλες μεταβλητές διανυσματικού τύπου

Η CUDA/C ορίζει μερικούς ακόμα τύπους διανυσμάτων, θα αναφερθούμε στους πιο βασικούς:

- uint2,int2: Διάνυσμα ακεραίων με διάσταση 2.

```
struct int2{
    int x;
    int y;
}
//Διάνυσμα (0,1)
int2 vector2D;
int.x =0;
int.y =1;
```

- float2,double2: Αντίστοιχο του int2 αλλά για αριθμούς κινητής υποδιαστολής μονής και διπλής ακρίβειας.
- uchar2,char2: Το ίδιο με τα παραπάνω
- uint3,int3: Διάνυσμα ακεραίων με διάσταση 3.

```
struct int3{
    int x;
    int y;
    int z;
}
//Διάνυσμα (0,1,2)
int3 vector3D;
int.x =0;
int.y =1;
int.z =2;
```

- float3: Αντίστοιχο με int3 (προσοχή, δεν ορίζεται double2).
- uchar3,char3: Το ίδιο.

- uint4,int4: Διάνυσμα ακεραίων διάστασης 4.

```
struct int4{
    int x;
    int y;
    int z;
    int w;
}
//Διάνυσμα (0,1,2,3)
int4 vector4D,vector4D_1;
int.x =0;
int.y =1;
int.z =2;
int.w =3;
vector4D_1 = vector4D;
//Τώρα τοδίανυσμα  vector4D_1 είναι (0,1,2,3)
```

- float4: Αντίστοιχο με int4.
- uchar4,char4: Το ίδιο.

Συναρτήσεις αναγνώρισης και ενεργοποίησης συσκευών

cudaErro_t cudaGetDeviceCount(int *count) : Μας επιστρέφει των αριθμό των διαθέσιμων καρτών γραφικών που υποστηρίζουν την CUDA.

cudaError_t cudaSetDevice(int device) : Ενεργοποιεί ένα ”πλαίσιο” (context) μεταξύ ενός νήματος του κεντρικού επεξεργαστή και της κάρτας γραφικών με τον αύξοντα αριθμό ”device”.

cudaError_t cudaGetDevice(int *device) : Μας επιστρέφει τον αύξοντα αριθμό της κάρτας γραφικών με την οποία το καλών νήμα του κεντρικού επεξεργαστή έχει ενεργοποιήσει ένα ”πλαίσιο”.

Μεταβλητές τύπου cudaDeviceProp και οι σχετικές συναρτήσεις

Μία μεταβλητή τύπου cudaDeviceProp είναι τύπου struct και ορίζεται όπως κάθε μεταβλητή:

```
cudaDeviceProp variable;
```

Θα αναφέρουμε μόνο το κομμάτι του struct που μας ενδιαφέρει:

```
struct cudaDeviceProp{
int     canMapHostMemory
```

```
    //Device can map host memory with cudaHostAlloc/cudaHostGetDevicePointer.
int     deviceOverlap
    //Device can concurrently copy memory and execute a kernel.
int     kernelExecTimeoutEnabled
    //Specified whether there is a run time limit on kernels.
int     maxGridSize [3]
    //Maximum size of each dimension of a grid.
int     maxThreadsDim [3]
    //Maximum size of each dimension of a block.
int     maxThreadsPerBlock
    //Maximum number of threads per block.
int     regsPerBlock
    //32-bit registers available per block
size_t  sharedMemPerBlock
    //Shared memory available per block in bytes.
size_t  totalConstMem
    //Constant memory available on device in bytes.
size_t  totalGlobalMem
    //Global memory available on device in bytes.

}
```

Όπως βλέπουμε, είναι μία μεταβλητή που σχετίζεται με τις δυνατότητες της κάρτας γραφικών. Όντως, αυτή η μεταβλητή θα μας χρειαστεί μόνο σε δύο περιπτώσεις. Η πρώτη είναι να θέλουμε ”ρωτήσουμε” την συσκευή για τα τεχνικά χαρακτηριστικά της. Αυτή η ικανότητα μπορεί να χρησιμεύσει αν θέλουμε να κάνουμε το πρόγραμμα μας ευέλικτο, παραμετροποιώντας τις κλήσεις στην κάρτα γραφικών αυτόματα κατά την εκτέλεση. Η δεύτερη είναι, σε ένα περιβάλλον με πάνω από μία κάρτες γραφικών,να μπορούμε να διαλέξουμε εκείνη που είναι πιο κοντά στις προτιμήσεις μας. Ας δούμε τις συναρτήσεις:

cudaError_t cudaGetDeviceProperties(struct cudaDeviceProp *prop,int device) : Μας επιστρέφει μία μεταβλητή τύπου cudaDeviceProp με όλα τα τεχνικά χαρακτηριστικά της κάρτας γραφικών με αύξοντα αριθμό device.

cudaError_t cudaChooseDevice(int *device,const struct cudaDeviceProp *prop) : Μας επιστρέφει τον αύξοντα αριθμό της κάρτας γραφικών, με τεχνικά χαρακτηριστικά όσο πιο κοντά γίνεται σε αυτά που έχουμε θέσει ως επιθυμητά σε μία μεταβλητή τύπου cudaDeviceProp.

Συναρτήσεις ρητού συγχρονισμού και τερματισμού context

Όπως έχουμε αναφέρει, η λειτουργία της κάρτας γραφικών είναι γενικά ασύγχρονη ως προς την λειτουργία του κεντρικού επεξεργαστή. Σε πολλές συναρτήσεις (κυρίως όσες αναφέρονται σε μεταφορές μνήμης αλλά και εκτέλεσης προγράμματος στην συσκευή) η λειτουργία του συγχρονισμού είναι δεδομένη και δεν

χρειάζεται ρητός συγχρονισμός. Σε κάποιες περιπτώσεις ο συγχρονισμός είναι αναγκαίο να δηλωθεί ρητά. Επίσης, ο τερματισμός του πλαισίου κάρτας γραφικών - κεντρικού επεξεργαστή γενικά δεν είναι αναγκαίος αφού αυτός γίνεται αυτόματα στο τέλος του προγράμματος μας. Είναι καλή προγραμματιστική πρακτική όμως να το τερματίζουμε ρητά αν μετά από κάποιο σημείο του προγράμματος δεν χρησιμοποιείται.

cudaError_t cudaThreadSynchronize (void) : Επιβάλλει ρητό συγχρονισμό μεταξύ κάρτας γραφικών και κεντρικού επεξεργαστή. Με λίγα λόγια άρει την ασύγχρονη λειτουργία και επιβάλει στο νήμα του κεντρικού επεξεργαστή να "περιμένει" να ολοκληρωθούν οι διεργασίες στην κάρτα γραφικών.

cudaError_t cudaThreadExit (void) : Ελευθερώνει ρητά όλους τους πόρους του νήματος του κεντρικού επεξεργαστή που σχετίζονται με τον έλεγχο της κάρτας γραφικών και τερματίζει το υπάρχον πλαίσιο και συνεπώς την λειτουργία της μέσα στο πρόγραμμα. Οποιαδήποτε κλήση ως προς την κάρτα γραφικών θα δημιουργήσει ένα νέο πλαίσιο.

Μεταβλητές ελέγχου συμβάντος cudaEvent_t και σχετικές συναρτήσεις

Οι μεταβλητές ελέγχου συμβάντος (event management) είναι ένας εσωτερικός τύπος της CUDA και η δομή δεν μας απασχολεί. Χρησιμοποιούνται κυρίως για να μετρήσουμε την διαφορά χρόνου καταγραφής δύο συμβάντων. Δηλαδή, ουσιαστικά τον χρόνο που χρειάστηκε για να πάμε από το ένα συμβάν στο άλλο. Η χρήση τους είναι πολύ απλή, και σε αντίθεση με τις συναρτήσεις της C η ακρίβεια που επιστρέφουν τα συμβάντα είναι μισό μsecond! Ορίζονται ως:

```
cudaEvent_t variable;
```

cudaError_t cudaEventCreate (cudaEvent_t *event) : Δημιουργεί ένα αντικείμενο καταγραφής συμβάντος event.

cudaError_t cudaEventDestroy (cudaEvent_t event) : Αντίστοιχα, καταστρέφει ένα αντικείμενο καταγραφής συμβάντος.

cudaError_t cudaEventRecord (cudaEvent_t event, cudaStream_t stream) : Ξεκινάει την εγγραφή ενός συμβάντος event για το ρεύμα stream. Το ρεύμα στην περίπτωση μας θα είναι πάντα μηδέν και δεν μας απασχολεί.

cudaError_t cudaEventSynchronize (cudaEvent_t event) : Τα συμβάντα δεν συγχρονίζονται σε καμία περίπτωση αυτόματα. Επομένως, πριν επιχειρήσουμε να διαβάσουμε μία μεταβλητή συμβάντος της οποίας έχουμε ξεκινήσει την εγγραφή, πρέπει πρώτα ρητά να την συγχρονίσουμε. Να επιβάλλουμε, δηλαδή, στον κεντρικό επεξεργαστή να περιμένει έως ότου η εγγραφή του συμβάντος event ολοκληρωθεί.

cudaError_t cudaEventElapsedTime (float *ms, cudaEvent_t start, cudaEvent_t end) : Μας επιστρέφει τον χρόνο που έχει περάσει από την εγγραφή του συμβάντος start έως την εγγραφή του συμβάντος end. Ο χρόνος επιστρέφεται σε msecond με ακρίβεια μισό μsecond.

Η χρήση συμβάντων είναι πολύ συνηθισμένη. Μην ξεχνάμε ότι ο λόγος που παιδευόμαστε και προγραμματίζουμε σε κάρτες γραφικών είναι η επίτευξη επιδόσεών. Επομένως, το λεγόμενο profiling ή χαρακτηρισμός του προγράμματος, δηλαδή, η μελέτη της χρονικής διάρκειας που χρειάζεται κάθε συνάρτηση για να εκτελεστεί, είναι πολύ σημαντικό. Πάμε τώρα να γράψουμε ένα μικρό κομμάτι κώδικα που επιδεικνύει την χρήση των συμβάντων:

```
//Αρχικά ορίζουμετιςμεταβλητέςσυμβάντων
cudaEvent_t start,end;
float fancy_timer;

//Δημιουργούμε έναναντικείμενοκαταγραφήςσυμβάντοςγιατηνκάθεμία
cudaEventCreate(&start);
cudaEventCreate(&end);

//Ξεκινάμε τηνκαταγραφήτουσυμβάντοςέναρξης
cudaEventRecord(start);

//Καλούμε τησυνάρτησημας
cuda_run_fancy_things_on_gpu();

//Ξεκινάμε τηνκαταγραφήτουσυμβάντοςτέλους
cudaEventRecord(stop);

//Επιβάλλουμε συγχρονισμό
cudaEventSynchronize(stop);

//και τέλος...
printf("H synarthsh fancy etrekse gia %f (ms)\n",
    cudaEventElapsedTime(&fancy_timer,start,stop));

//Δεν ξεχνάμενακαθαρίσουμεπρώτα ..
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Αν δεν επιβάλλουμε συγχρονισμό η cudaEventElapsedTime θα επιστρέψει σφάλμα και εμείς στην μεταβλητή fanc_timer θα διαβάσουμε σκουπίδια.

3.2.3 Ειδική ορισμοί συναρτήσεων

Πριν αναφερθούμε στον kernel θα αναφέρουμε πως ορίζονται και διαχωρίζονται οι συναρτήσεις που αναφέρονται στην κάρτα γραφικών (device) και αυτές που αναφέρονται στον κεντρικό επεξεργαστή (host).

__device__ Μία συνάρτηση η οποία είναι ορισμένη ως __device__, εκτελείται στην κάρτα γραφικών και μπορεί να κληθεί από την κάρτα γραφικών μόνο.

__global__ Μία συνάρτηση η οποία είναι ορισμένη ως __global__, εκτελείται στην κάρτα γραφικών και μπορεί να κληθεί από τον κεντρικό επεξεργαστή μόνο.

__host__ Ως __host__ ορίζεται μία συνάρτηση η οποία θα εκτελείται στον κεντρικό επεξεργαστή και θα μπορεί να κληθεί μόνο από αυτόν (οι συναρτήσεις που εκτελούνται στην κάρτα γραφικών εξ'ορισμού δεν μπορούν να κάνουν κλήση συναρτήσεων που θα εκτελεσθούν στον κεντρικό επεξεργαστή[21]. Όταν μία συνάρτηση δεν περιέχει κανέναν από τους τρεις ορισμούς τότε θα εκτελείται όπως μία __host__. Εδώ ο ορισμός __host__ φαίνεται να είναι άχρηστος. Στην ουσία υπάρχει για ευκολία, καθώς επιτρέπεται να χρησιμοποιήσουμε πάνω από έναν ορισμό για μία συνάρτηση. Αν ορίσουμε μία συνάρτηση ως __host__ και __device__, τότε ο μεταγλωττιστής θα την μεταγλωττίσει εις διπλούν, μία έκδοση για τον κεντρικό επεξεργαστή και μία έκδοση για την κάρτα γραφικών!

3.2.4 Ο kernel,οι ειδικές μεταβλητές του και οι συναρτήσεις συσκευής

Ήρθε επιτέλους η ώρα να μιλήσουμε για τον kernel που τόσες φορές αναφέραμε ως τώρα. Ο kernel είναι η καρδιά ενός προγράμματος για GPU. Kernels ονομάζονται οι κύριες συναρτήσεις της κάρτας γραφικών. Οι συναρτήσεις αυτές, δηλαδή, που αναλαμβάνουν να επεξεργαστούν τα δεδομένα μας. Ένας kernel εφόσον εκτελείται στην κάρτα γραφικών δεν έχει την συνήθη πρόσβαση I/O που απολαμβάνουν οι κλασσικές συναρτήσεις που τρέχουν στον κεντρικό επεξεργαστή.

Οι συναρτήσεις kernel ορίζονται ως __global__, πρέπει να επιστρέφουν πάντα τιμή void και τα αρχεία που τις περιέχουν πρέπει να έχουν την κατάληξη .cu. Ας δούμε ένα παράδειγμα ορισμού:

```
__global__ void this_will_run_in_the_gpu(float *input,float *output);
```

[21] Εξού και ο περιορισμός στο I/O, μία συνάρτηση της κάρτας γραφικών δεν έχει πρόσβαση στις συναρτήσεις που αναλαμβάνουν το I/O.

Οι συναρτήσεις kernel διαφέρουν από όλες τις άλλες συναρτήσεις στον τρόπο που καλούνται. Σε μία συνάρτηση kernel εκτός από τα κλασσικά ορίσματα που περιέχει κάθε κλήση συνάρτησης, υπάρχουν και μερικά ειδικά ορίσματα με τις παραμέτρους του kernel. Οι παράμετροι του kernel είναι τα blocks, grids αλλά και η μέγιστη τιμή δυναμικά δεσμευμένης κοινόχρηστης μνήμης:

```
this_will_run_in_the_gpu<<<grids,blocks,max_dynamic_sharedmem>>>(in,out);
```

Τα <<<...>>> αναλαμβάνουν να δώσουν τις παραμέτρους στον kernel μας. Στην πραγματικότητα, συνήθως η παράμετρος της δυναμικής κοινόχρηστης μνήμης δεν αναγράφεται καν ή απλά βάζουμε μονάδα. Συνήθως προτιμάμε να ορίζουμε το μέγεθος των block στατικά και κατά κανόνα η κοινόχρηστη μνήμη ανά block ορίζεται ως συνάρτηση του μεγέθους του block.

Μία συνάρτηση kernel είναι ουσιαστικά το κομμάτι του κώδικα που θα τρέξει σε κάθε νήμα των πολυεπεξεργαστών. Είναι, δηλαδή, ένα loop από το νήμα 0 εώς το νήμα grid_size*block_size-1 στην απλουστευμένη περίπτωση που το block έχει μόνο διάσταση x. Σε αυτό το loop όμως δεν υπάρχει ο ορισμός της τάξης. Τα νήματα δεν θα εκτελεστούν με αύξοντα αριθμό, αλλά σχεδόν τυχαία. Η κάρτα γραφικών μας θα αποφασίσει πώς θα τα εκτελέσει, αρκεί να ξέρουμε εμείς ότι δεν μπορούμε να βασιζόμαστε στην ιδέα ότι ένα νήμα Α θα τρέξει πριν από ένα νήμα Β.

Ειδικές μεταβλητές

Μέσα σε κάθε kernel υπάρχουν μερικές ειδικές μεταβλητές οι οποίες ορίζονται από το σύστημα. Αυτές οι μεταβλητές μας δίνουν ανά πάσα στιγμή πιο νήμα ενός block εκτελείται, σε πιο block, ποια είναι η διάσταση του block και ποια η διάσταση του grid. Ας τις δούμε πιο αναλυτικά, αφού είναι βασικές για τον προγραμματισμό του kernel.

- **int3 threadIdx**: Είναι ο δείκτης κάθε νήματος μέσα σε ένα block. Παίρνει τιμές [0..blockDim).
- **int3 blockIdx**: Μας δίνει τον δείκτη κάθε block. Παίρνει τιμές [0..gridDim).
- **int3 blockDim**: Μας δίνει τις διαστάσεις κάθε block. Η τιμή του είναι σταθερή αφού έχει οριστεί από τις παραμέτρους που περάσαμε στον kernel.
- **int3 gridDim**: Μας δίνει τις διαστάσεις του grid, το πλήθος δηλαδή των blocks σε κάθε διάσταση. Η τιμή του επίσης είναι προκαθορισμένη από τις παραμέτρους του kernel.

Τώρα θα πάρουμε την περίπτωση των δύο διαστάσεων θα προσπαθήσουμε να βρούμε τον καθολικό γραμμικό δείκτη ενός νήματος (μην ξεχνάμε ότι γενικά οι προσπελάσεις στην μνήμη είναι γραμμικές). Ένας μετασχηματισμός από τις δύο διαστάσεις σε μία είναι ο εξής:

$$index1D = x + N_x * y \tag{3.1}$$

Το πρόβλημα θα γίνει πιο εύκολα κατανοητό με την βοήθεια των σχημάτων 3.15. Στο σχήμα 3.15α το μαύρο τμήμα είναι το block με συντεταγμένες (0,0), το μπλέ το block με συντεταγμένες (1,0), το κίτρινο (0,1) και το πράσινο (1,1). Όπως είπαμε το διάνυσμα threadIdx μας δίνει τις συντεταγμένες τοπικά στο block, δηλαδή, σε κάθε block παίρνει πάντα τις τιμές $[0..block\_size)$, με το block_size στη περίπτωση του σχήματός να είναι 8x8. Στην σχέση 3.1 αν για x θέσουμε threadIdx.x, για Nx blockDim.x και για y threadIdx.y θα πάρουμε τις γραμμικές θέσεις σε κάθε block όπως φαίνεται στο σχήμα 3.15α. Για να φτάσουμε στην καθολική γραμμική θέση κάθε νήματος πρέπει πρώτα να υπολογίζουμε τις τοπικές συντεταγμένες. Για την διάσταση x, θα είναι:

$$x_{global} = threadIdx.x + blockDim.x * blockIdx.x$$

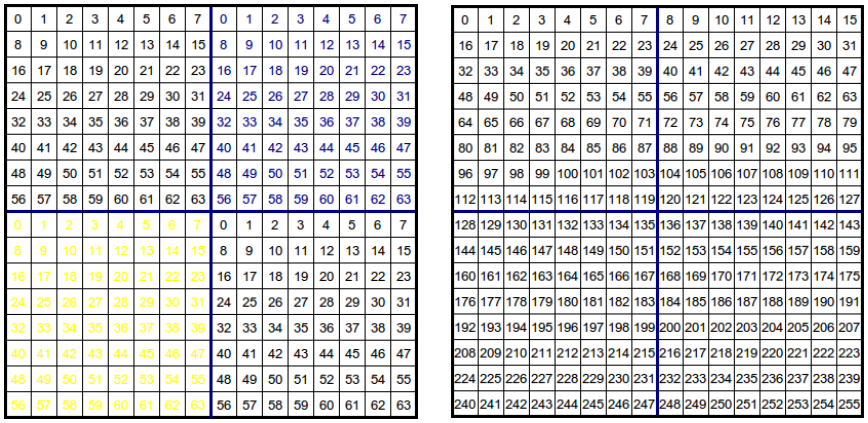
και αντίστοιχα για την y διάσταση θα είναι:

$$y_{global} = threadIdx.y + blockDim.y * blockIdx.y$$

Αν τώρα θέσουμε τις νέες συντεταγμένες x_{global} και y_{global} στην σχέση 3.1 με x την συνολική διάσταση του x, $blockDim.x * gridDim.x$ (μέγεθος του block στην διάσταση x επί το πλήθος τους), θα πάρουμε τις καθολικές γραμμικές θέσεις, όπως φαίνονται και στο σχήμα 3.15β. Αυτές είναι ιδιαίτερα χρήσιμες καθώς στην πλειονότητα των περιπτώσεων, όλες οι συναλλαγές με την καθολική μνήμη θα γίνονται με δείκτη αντίστοιχο του καθολικού και όλες οι συναλλαγές με την κοινόχρηστη μνήμη με τον τοπικό δείκτη.

```
// Οκαθολικόςδείκτηςενόςνήματοςείναι :
tid_global = threadIdx.x + blockIdx.x * blockDim.x + \
    (gridDim.x*blockDim.x)*(threadIdx.y + blockIdx.x * blockDim.y);
```

Ας υποθέσουμε τώρα ότι έχουμε έναν πίνακα 8x8 χωρισμένο σε 4 δισδιάστατα blocks με αναλογία 1-1 στοιχείων - νημάτων. Για κάποιον λόγο, πρέπει να επεξεργαστούμε με διαφορετικό τρόπο το πάνω και το κάτω μισό. Ο kernel έχουμε εξηγήσει ότι είναι σαν ένα loop πάνω σε όλα τα νήματα:



(α□) Γραμμική θέση κάθε νήματος σε ένα block

(β□) Γραμμική θέση κάθε νήματος στο σύνολο του grid

Σχήμα 3.15: Γραμμικές θέσεις νήματος.

```
__global__ void this_will_run_in_the_gpu(float *int, float *out){

    int tid_local = threadIdx.x + blockDim.x * blockIdx.x;
    int tid_global = threadIdx.x + blockIdx.x * blockDim.x + \
        (gridDim.x*blockDim.x)*(threadIdx.y + blockIdx.x * blockDim.y);

    //Στο πάνωμισόθέλουμετηντιμή      tid_global,
    // στοκάτωμισότηντιμή      tid_local
    if(blockIdx.y < blockDim.y/2) out[tid_global] = tid_global;
    else out[tid_global] = tid_local;

}
```

Αν προσπαθήσετε να τυπώσετε τα αποτελέσματα θα βρείτε το πάνω μισό ίσο με το πάνω μισό του σχήματος 3.15β□ και το κάτω μισό ίσο με το κάτω μισό του σχήματος 3.15α□!

Ένας kernel έχει την ικανότητα να περιέχει ”αδρανή” νήματα. Νήματα τα οποία δεν θα κάνουν τίποτα. Αυτό χρησιμεύει όταν τα δεδομένα μας δεν χωράνε ακριβώς στα blocks. Τότε χρειαζόμαστε ένα ή δύο blocks (ανάλογα αν είμαστε σε μία ή δύο διαστάσεις) επιπλέον. Δείξαμε πως γίνεται αυτό στο κομμάτι κώδικα 3.1 στην σελίδα 64. Τα νήματα που είναι εκτός εμβέλειας των δεδομένων μας, απλά δεν θα κάνουν τίποτα!

3.2.5 Μνήμη, οι συναρτήσεις και οι παράμετροί τους

Σε αυτή την υποενότητα θα δούμε πως ορίζονται οι διάφοροι τύποι μνήμης που αναλύσαμε στην υποενότητα 3.1.3 μέσα σε ένα πρόγραμμα. Κάθε τύπος μνήμης

έχει τους δικούς του κανόνες για το που πρέπει να οριστεί. Γενικά, για ότι δεδομένα έχουμε στον κεντρικό επεξεργαστή και θέλουμε να τα επεξεργαστούμε στην κάρτα γραφικών, θα πρέπει πρώτα να ορίσουμε ίσο αριθμό δεικτών συσκευής (device pointers) αλλά και να δεσμεύσουμε την αντίστοιχη καθολική μνήμη για το καθένα. Ένας kernel μπορεί να δεχθεί και ορίσματα τα οποία δεν είναι στον χώρο μνήμης της κάρτας γραφικών, αρκεί αυτά να είναι κάποιος απλός τύπος (integer,float κτλ) και όχι pointer. Σε περίπτωση τέτοιων ορισμάτων, η CUDA θα αναλάβει να τα περάσει αθόρυβα στην κοινόχρηστη μνήμη.

Ορισμοί μεταβλητών της κάρτας γραφικών

- *Μεταβλητές καθολικής μνήμης*. Ορίζονται κανονικά στο κυρίως πρόγραμμα, σε μία επικεφαλίδα και γενικά όπως κάθε συνηθισμένη μεταβλητή. Πρέπει, όμως να είναι pointer κάποιου τύπου. Κατά σύμβαση βάζουμε μία προθήκη πριν το όνομα της μεταβλητής για να βλέπουμε άμεσα αν μία μεταβλητή αναφέρεται στον κεντρικό επεξεργαστή ή στην κάρτα γραφικών.

```
//Μεταβλητή device,global
float *devOut;
//Μεταβλητή cpu
float *Out;
```

- *Μεταβλητές κοινόχρηστης μνήμης*. Ορίζονται μόνο μέσα σε έναν kernel με τύπο __shared__ πριν τον κανονικό τύπο της μεταβλητής. Στο παράδειγμα θα την ορίσουμε με στατικό μέγεθος, όπως συμβαίνει και στις περισσότερες περιπτώσεις

```
__global__ void gpu_function(){

    //Αυτή είναιμεταβλητή  shared
    __shared__ float variable[block_size];
}
```

- *Τοπικές μεταβλητές νήματος*. Ο μεταγλωττιστής θα αποφασίσει αν μία τοπική μεταβλητή θα δεσμευτεί στην τοπική μνήμη ή σε έναν καταχωρητή. Οι τοπικές μεταβλητές ορίζονται μόνο μέσα σε έναν kernel ή μία συνάρτηση __device__ και δεν διαφέρουν στην όψη από τον ορισμό μίας συμβατικής μεταβλητής.

```
__global__ void gpu_function(){

    //Αυτή είναιμεταβλητή  shared
    __shared__ float variable[block_size];

    //Αυτές είναιτοπικέςμεταβλητές
    int tid_global;
    int temp[3];
    float var2;
}
```

- *Μεταβλητή κλειδωμένης μνήμης συστήματος*. Δηλώνεται κανονικά αφού είναι μεταβλητή του κεντρικού επεξεργαστή. Ουσιαστικά μία μεταβλητή γίνεται κλειδωμένης μνήμης αφού δεσμεύσουμε μνήμη γι αυτή της μεταβλητή, μέσω ειδικής συνάρτησης της CUDA.

- *Μεταβλητή σταθερών*. Οι μεταβλητές σταθερών δηλώνονται μόνο στην επικεφαλίδα του αρχείου που περιέχει τον kernel που θα τις χρησιμοποιήσει. Θα δούμε αργότερα πως θα τους δώσουμε τιμή.

```
//Αυτή είναιμεταβλητήσταθερών
__constant__ int const_Nx;

__global__ void gpu_function(){

    //Αυτή είναιμεταβλητή  shared
    __shared__ float variable[block_size];

    //Αυτές είναιτοπικέςμεταβλητές
    int tid_global;
    int temp[3];
    float var2;
    var2 = temp[0]/3 * const_Nx;
}
```

- *Μεταβλητή υφής*. Οι μεταβλητές υφών δηλώνονται μόνο στην επικεφαλίδα του αρχείου που περιέχει τον kernel που θα τις χρησιμοποιήσει. Θα δούμε αργότερα πως θα τις παραμετροποιήσουμε και θα τις δέσουμε σε μία περιοχή καθολικής μνήμης.

```
//H texRef είναιμεταβλητήυφής
texture<int,1, cudaReadModeElementType> texRef;

__global__ void gpu_function(){
...
}
```

Η ειδική περίπτωση των μεταβλητών υφής και σταθεράς

Θα αναλύσουμε αυτές τις μεταβλητές περισσότερο, αφού είδαμε ότι διαφέρουν αρκετά ως προς τους άλλους τύπους μεταβλητών και παρουσιάζουν δύο ομοιότητες: 1) Μπορούν να δηλωθούν μόνο εκτός συνάρτησης και 2) πρέπει να είναι δηλωμένες στο ίδιο αρχείο που περιέχει τις συναρτήσεις που θα τις χρησιμοποιήσουν. Θα θεωρήσουμε δύο περιπτώσεις εδώ, μία όπου ο kernel είναι στο ίδιο αρχείο με την συνάρτηση που έχει όλες τις μεταβλητές που χρειαζόμαστε για να δώσουμε τιμές σε μία υφή και σε μία σταθερά, και μία όπου ο kernel είναι σε εντελώς διαφορετικό αρχείο μαζί μόνο με τους ορισμούς αυτών των δύο μεταβλητών:

Περίπτωση 1:

```
texture<int,1, cudaReadModeElementType> texRef;
__constant__ int const_Nx;
...
void a_cpu_function(){

float *devPtr,*cpuPtr;
int Nx;
....
}

__global__ void a_gpu_function(){
...
}
```

Περίπτωση 2, αρχείο 1:

```
void a_cpu_function(){

float *devPtr,*cpuPtr;
int Nx;
....
}
```

Αρχείο 2:

```
texture<int,1, cudaReadModeElementType> texRef;
__constant__ int const_Nx;

__global__ void a_gpu_function(){
...
}
```

Για μία μεταβλητή σταθεράς, τα πράγματα είναι σχετικά απλά, στην περίπτωση 1, απλά χρησιμοποιούμε την ειδική συνάρτηση:

```
cudaError_t cudaMemcpyToSymbol (const char* symbol, const void *src,size_t count,
    size_t offset, enum cudaMemcpyKind kind)
```

Θα δώσουμε ένα μικρό παράδειγμα για να την κατανοήσουμε γρήγορα (μην σας απασχολεί το offset, είναι πολύ σπάνια η χρήση του):

```
__constant__ int const_Nx;
...
void a_cpu_function(){

float *devPtr,*cpuPtr;
int Nx;
....
cudaMemcpyToSymbol(&const_Nx,&Nx,sizeof(int),0,cudaMemcpyHostToDevice);
}
```

Για την περίπτωση 2, τότε θα πρέπει να γράψουμε μία συνάρτηση στο αρχείο 2 η οποία να παίρνει ως όρισμα το Nx και να θέτει την τιμή του στην μεταβλητή σταθεράς:

Αρχείο 2:

```
__constant__ int const_Nx;

void setconstant(int Nx){
    cudaMemcpyToSymbol(&const_Nx,&Nx,sizeof(int),0,cudaMemcpyHostToDevice);
}

__global__ void a_gpu_function(){
...
}
```

Αρχείο 1:

```
void a_cpu_function(){
    int Nx;
    setconstant(Nx);
}
```

Η περίπτωση των μεταβλητών υφής είναι αρκετά πιο πολύπλοκη και θα ασχοληθούμε μόνο με την περίπτωση γραμμικής μνήμης. Αρχικά χρειαζόμαστε μερικές νέες συναρτήσεις και τύπους μεταβλητών.

- texture. Ο ορισμός ενός texture πριν τον kernel. Έχει την μορφή:

  ```
  texture<Type, Dim, ReadMode> onoma;
  ```

  Στον τύπο μπορούμε να θέσουμε int, float, char, int2, float2, char2, int4, float4, char4. Το dim είναι η διάσταση του texture, μπορεί να είναι 1 ως 3. Τους περιορισμούς της κάθε διάστασης μπορεί κανείς να τους βρει στον πίνακα 3.1 στις τελευταίες σειρές. Το Readmode ορίζει πως θα δίνονται οι συντεταγμένες αυτής της μεταβλητής υφής. Οι επιλογές είναι cudaReadModeNormalizedFloat ή cudaReadModeElementType. Η πρώτη ορίζει ότι οι συντεταγμένες θα είναι πραγματικός αριθμός κανονικοποιημένος ως προς τις

διαστάσεις, θα παίρνει επομένως τιμές [0..1). Η δεύτερη ορίζει ότι οι συντεταγμένες μπορεί να είναι πραγματικός αριθμός ή ακέραιος, αλλά δεν θα γίνει κάποια κανονικοποίηση. Στην περίπτωση γραμμικής μνήμης, επιτρέπεται μόνο η επιλογή
cudaReadModeElementType.

- cudaChannelFormatDesc. Ορίζει τα κανάλια και τους τύπους της υφής. Ουσιαστικά το πως θα μεταφράζεται η κάθε κλήση υφής! Κάθε κανάλι είναι ο αριθμός των bytes του τύπου δεδομένων για κάθε διάσταση του διανύσματος. Έρχεται πακέτο με την συνάρτηση cudaCreateChannelDesc η οποία μας κάνει την ζωή πιο εύκολη. Προσοχή το κανάλι δεν έχει καμία σχέση με την διάσταση της υφής. Έχει σχέση με τον τύπο των δεδομένων που θα επιστρέφει μόνο. Επομένως, αν μία υφή είναι τριών διαστάσεων, μπορεί να έχει κανάλι τεσσάρων διαστάσεων. Θα είναι τότε μία τρισδιάστατη υφή με κάθε στοιχείο ένα τετραδιάστατο διάνυσμα!

```
cudaChannelFormatDesc texture_descriptor;
//1D 32bit integer vector.
texture_descriptor =
    cudaCreateChannelDesc(32,0,0,0,cudaChannelFormatKindSigned);
//2D 32bit float vector
texture_descriptor =
    cudaCreateChannelDesc(32,32,0,0,cudaChannelFormatKindFloat);
//4D 32bit float vector
texture_descriptor =
    cudaCreateChannelDesc(32,32,32,32,cudaChannelFormatKindFloat);
//Διάνυσμα τριώνδιαστάσεωνδενυποστηρίζεται !
```

- cudaBindTexture. Δένει μία υφή στο κομμάτι της (γραμμικής) καθολικής μνήμης που δείχνει ο pointer devPtr.

```
cudaError_t cudaBindTexture(size_t *offset,
    const struct textureReference *texref,
     const void *devPtr,
      const struct cudaChannelFormatDesc *desc,
       size_t size);
```

- UnbindTexture. Ελευθερώνει μία υφή.

```
cudaError_t cudaUnbindTexture(const struct textureReference *texref);
```

- tex1Dfetch. Μας επιστρέφει μία τιμή ή ένα διάνυσμα τιμών βάση των συντεταγμένων που της δίνουμε. Καλείται μόνο μέσα από έναν kernel.

```
//Κάθε στοιχείοτηςυφήςείναι 4D διάνυσμαακεραίων .
int4 tex1Dfetch(const struct textureReference texref, int x);

//Κάθε στοιχείοτηςυφήςείναι 2D διάνυσμα float.
float2 tex1Dfetch(const struct textureReference texref, int x);

//Κάθε στοιχείοτηςυφήςείναι 1D διάνυσμα char.
char tex1Dfetch(const struct textureReference texref, int x);
```

Επιτέλους πλέον μπορούμε να χρησιμοποιήσουμε τις υφές. Πρώτα λοιπόν πρέπει να δηλώσουμε την υφή, στην συνέχεια να ορίσουμε τα κανάλια της και τέλος να την δέσουμε σε γραμμική καθολική μνήμη.

Περίπτωση 1:

```
texture<float,1, cudaReadModeElementType> texRef;
...
void a_cpu_function(){

float *devPtr,*cpuPtr;
int Nx;
size_t offset;
....
cudaChannelFormatDesc texture_descriptor;
//1D 32bit integer vector.
texture_descriptor =
    cudaCreateChannelDesc(32,0,0,0,cudaChannelFormatKindFloat);
//έχουμε δεσμεύσειμνήμηγιατην devPtr
cudaBindTexture(&offset,texRef,devPtr,texture_descriptor,Nx*sizeof(float));
}

__global__ void a_gpu_function(){
...
x = tex1Dfetch(texRef, tid_global);
...
}
```

Περίπτωση 2, Αρχείο 2:

```
texture<float,1, cudaReadModeElementType> texRef;

void settexture(float *devPtr, int Nx, int &offset){
cudaChannelFormatDesc texture_descriptor;
//1D 32bit integer vector.
texture_descriptor =
    cudaCreateChannelDesc(32,0,0,0,cudaChannelFormatKindFloat);
//έχουμε δεσμεύσειμνήμηγιατην devPtr
cudaBindTexture(&offset,texRef,devPtr,texture_descriptor,Nx*sizeof(float));
}

__global__ void a_gpu_function(){
    ...
```

```
    x = tex1Dfetch(texRef, tid_global);
    ...
}
```

Αρχείο 1:

```
void a_cpu_function(){

    float *devPtr,*cpuPtr;
    int Nx;
    size_t offset;
    ....
    settexture(devPtr,Nx,offset);
}
```

Συναρτήσεις αποδέσμευσης μνήμης

Συνάρτηση cudaFree Αναλαμβάνει να ελευθερώσει την καθολική μνήμη στην οποία δείχνει ο δείκτης συσκευής devPtr.

```
cudaError_t cudaFree (void *devPtr);
```

Συνάρτηση cudaFreeHost Αναλαμβάνει να ελευθερώσει την κλειδωμένη μνήμη συστήματος που έχει οριστεί και δεσμευτεί από την κάρτα γραφικών

```
cudaError_t cudaFreeHost (void *hostPtr);
```

Συνάρτηση cudaFreeArray Ελευθερώνει ένα CUDA Array.

```
cudaError_t cudaFreeArray (struct cudaArray *array);
```

Οι υπόλοιπες συναρτήσεις δέσμευσης μνήμης

Συνάρτηση cudaMalloc Αποτελεί την πιο βασική συνάρτηση δέσμευσης μνήμης. Δεσμεύει καθολική μνήμη.

```
cudaError_t CudaMalloc(void **devPtr, size_t count)
```

Συνάρτηση cudaHostAlloc Δεσμεύει κλειδωμένη μνήμη συστήματος. Δέχεται τα ορίσματα flags τα οποία τα αναλύσαμε στην ενότητα 3.1.3 στην σελίδα 47.

```
cudaError_t cudaHostAlloc(void **hostPtr,size_t size, unsigned int flags)
```

Οι δυνατές τιμές των ”flags” είναι:

1. cudaHostAllocDefault. Σε αυτή την περίπτωση, δεσμεύει την μνήμη απλά ως κλειδωμένη μνήμη συστήματος.
2. cudaHostAllocPortable. Δεσμεύει την μνήμη ως φορητή.
3. cudaHostAllocMapped. Δεσμεύει την μνήμη ως χαρτογραφημένη.
4. cudaHostAllocWriteCombined. Δεσμεύει την μνήμη ως συνδυασμού εγγραφών.

Μπορούμε να συνδυάσουμε αυτά τα ”flags” ελεύθερα. Για την περίπτωση της cudaHostAllocMapped όμως πρέπει να έχουμε εκτελέσει την συνάρτηση cudaSetDeviceFlags με όρισμα cudaDeviceMapHost πρώτα.

Οι συναρτήσεις αντιγραφής μνήμης

Δεσμεύσαμε την μνήμη, τώρα πρέπει να την γεμίσουμε με δεδομένα όμως.. Αυτές οι συναρτήσεις αναφέρονται μόνο στην καθολική μνήμη, αφού οι σταθερές ορίζονται απευθείας, οι υφές δεν αντιγράφουν δεδομένα, αλλά δένονται σε ήδη υπάρχοντα και τέλος η κοινόχρηστη και η τοπική μνήμη ορίζονται και παίρνουν τιμές μόνο μέσα σε έναν kernel. Εδώ θα ξαναθυμίσουμε την μεταβλητή ροής των αντιγραφών, την cudaMemcpyKind. Μπορεί να έχει τις τιμές cudaMemcpyDeviceToHost, cudaMemcpyHostToDevice και cudaMemcpyDeviceToDevice.

Συνάρτηση cudaMemcpy Η πιο σημαντική συνάρτηση. Αντιγράφει δεδομένα από RAM σε καθολική, από καθολική σε RAM ή από καθολική σε καθολική.

```
cudaError_t cudaMemcpy(void *dst,const void *src,size_t count,
     enum cudaMemcpuKind kind)

//πχ GPU->CPU
cudaMemcpy(devPtr,hostPtr,Nx * sizeof(float),cudaMemcpyHostToDevice);
```

Και όμως, μόνο αυτή μπορούμε να αναφέρουμε με αυτά που έχουμε πει ως τώρα. Υπάρχουν αρκετές συναρτήσεις αντιγραφής μνήμης ακόμα, αλλά είναι αρκετά εξειδικευμένες και όλες έρχονται πακέτο με μία αντίστοιχη συνάρτηση δέσμευσης. Κάποιος μπορείς να τις βρει όλες αναλυτικά στην βιβλιογραφία[10][22].

[22]Όλοι οι οδηγοί για την CUDA διανέμονται ελεύθερα.

Ένα ολοκληρωμένο παράδειγμα

Αν συμμαζέψουμε ότι αναφέραμε ως τώρα και ας τα βάλουμε σε ένα μικρό αλλά ολοκληρωμένο παράδειγμα!

```
//naive demo!
texture<float,1, cudaReadModeElementType> texRef;
__constant__ int Nx;

#define block_size 256
...
void a_cpu_function(){

    float *devPtr,*cpuPtr;
    int Nx;
    size_t offset;
    cudaChannelFormatDesc texture_descriptor;
    ...
    cpuPtr = new float [Nx * sizeof(float)];
    //put random data in cpuPtr
    ...
    cudaMalloc(devPtr,Nx*sizeof(float));
    cudaMemcpy(&devPtr,cpuPtr,Nx*sizeof(float));
    cudaMemcpyToSymbol(&const_Nx,&Nx,sizeof(int),0,cudaMemcpyHostToDevice);

    //1D 32bit integer vector.
    texture_descriptor =
    cudaCreateChannelDesc(32,0,0,0,cudaChannelFormatKindFloat);
    cudaBindTexture(&offset,texRef,devPtr,texture_descriptor,Nx*sizeof(float));
    ...
    a_gpu_function<<<grids,blocks,1>>>(devPtr);
    cudaThreadSynchronize();
    cudaError_t err = cudaGetLastError();

    cudaUnbindTexture(texRef);
    cudaFree(devPtr);
    delete cpuPtr;

return;
}

__global__ void a_gpu_function(float *devPtr){

    __shared__ float sh_mem[block_size];
    int tid_global = threadIdx.x + blockDim.x * blockIdx.x;
    ...
    devPtr[tid_global] = tex1Dfetch(texRef, tid_global);
    ...
}
```

Προσοχή εδώ στον kernel. Διαβάζει μία υφή η οποία είναι δεμένη πάνω στην μνήμη στην οποία γράφει. Δεν έχει καμία απολύτως σημασία, αλλά κανείς δεν μπορεί να μας εγγυηθεί ότι θα μας δώσει την αλλαγμένη τιμή! Η μνήμη υφής είναι cached, πρέπει να χρησιμοποιείται μόνο για ”στατικές” αναγνώσεις και μπορεί να μας εγγυηθεί ότι θα έχει ανανεωμένες τιμές μόνο εάν τερματιστει ο kernel και

ξεκινήσει ένας νέος!

3.2.6 Συγχρονισμός της μνήμης και των νημάτων

Ο συγχρονισμός είναι ένα πολύ σημαντικό κεφάλαιο σε πολυνηματικά συστήματα. Σε πολλές περιπτώσεις είναι απαραίτητο να θέσουμε ένα φράγμα το οποίο πρέπει να φτάσουν όλα τα νήματα πριν συνεχίσουν στο επόμενο τμήμα.

Στην CUDA αλλά και γενικά στις κάρτες γραφικών δεν μπορούμε να συγχρονίσουμε όλα τα νήματα, καθώς αυτό είναι πολύ ακριβό να υλοποιηθεί στο σχεδιασμό των κυκλωμάτων αλλά και είναι πολύ επιβλαβές για τις επιδόσεις. Παρόλα αυτά μας δίνονται αρκετές επιλογές για μερικό συγχρονισμός, καθώς και κάποιες συναρτήσεις της κατηγορίας *atomic* με την χρήση των οποίων μπορούμε να είμαστε σίγουροι ότι δεν θα προσπαθήσουν να γράψουν τα νήματα ταυτόχρονα σε μία θέση μνήμης.

Η συνάρτηση synchthreads

Η συνάρτηση syncthreads είναι ο πιο συνηθισμένος τρόπος επιβολής συγχρονισμού στην CUDA. Είναι συνήθως και υπεραρκετός. Η syncthreads καλείται μέσα σε ένα kernel και συγχρονίζει τα νήματα ενός block μεταξύ τους. Είναι πολύ χρήσιμη μετά από αντιγραφές προς την κοινόχρηστη και πριν τις αντιγραφές από την κοινόχρηστη μνήμη. Όπως έχουμε εξηγήσει, η χρήση της κοινόχρηστης μνήμης είναι ένας ωραίος τρόπος να αποφύγουμε τις uncoalesced εγγραφές και αναγνώσεις από την καθολική μνήμη. Η synchtreads εκτός του ότι θα μας εγγυηθεί ότι μετά από αυτό όλα τα νήματα του block θα βρούνε τις σωστές πληροφορίες στην κοινόχρηστη μνήμη, επιβάλλει και σε όλα τα νήματα του block να διαβάσουν την καθολική μνήμη ή να γράψουν σε αυτή ταυτόχρονα, λόγω του συγχρονισμού!

```
void __syncthreads();
```

Ας δούμε ένα παράδειγμα με την χρήση της.

```
//block 1D
__global__ void a_gpu_function(float *out,float *in){

    __shared__ float sh_mem[block_size];
    int tid_global = threadIdx.x + blockDim.x * blockIdx.x;

    sh_mem[threadIdx.x] = in[tid_global];
    __syncthreads(); // perimene na teleiwsoun oles oi eggrafes

    sh_mem[threadIdx.x] = 1/tid_global;

    __synthreads(); // perimene na teleiwsoun oles oi eggrafes
    out[tid_local] = sh_mem[threadIdx.x];

}
```

Συγχρονισμός νημάτων διαφορετικών block

Είπαμε ότι ο συγχρονισμός μεταξύ διαφορετικών blocks, εντός ενός kernel δεν είναι δυνατός. Ο μόνος τρόπος να συγχρονίσουμε όλα τα νήματα είναι να τερματίσουμε τον kernel και να συνεχίσουμε με έναν νέο. Ο τερματισμός ενός kernel επιβάλλει συγχρονισμό σε όλα τα νήματα ανεξαρτήτως του block! Ας θεωρήσουμε ένα παράδειγμα, όπου ένας πίνακας 32χ32 έχει αλληλεξάρτηση ως προς το πάνω και το κάτω μισό του και ότι είναι χωρισμένος σε block των 8χ8 νημάτων. Μέσα σε έναν kernel δεν είναι δυνατό να συγχρονίσουμε τα blocks με blockIdx.y = 0 με αυτά που ανήκουν στο κάτω μισό και έχουν blockIdx.y = 1. Το πρόβλημα λύνεται απλά χωρίζοντας τον υπολογισμό σε δύο ξεχωριστούς kernel

```
kernelA_depends_onB<<<grids,blocks,1>>>();
kernelB_depends_onA<<<grids,blocks,1>>>();
```

Οι συναρτήσεις threadfence

Αυτές οι συναρτήσεις αποτελούν μία ειδική κατηγορία συγχρονισμού. Η syncthreads στην ορολογία είναι ένα barrier. Οι συναρτήσεις threadfence είναι απλά μικροί ταπεινοί φράχτες καθώς δεν επιβάλλουν ρητό συγχρονισμό στα νήματα. Παρόλα αυτά είναι πολύ χρήσιμοι σε ορισμένες περιπτώσεις αν και μπορεί να έχουν αρκετά αρνητικές συνέπειες ως προς την αποδοτική λειτουργία των πολυεπεξεργαστών. Οι συναρτήσεις threadfence λοιπόν επιβάλλουν αναμονή στα νήματα ενός block ή στο σύνολο τους, έως ότου ολοκληρωθούν οι συναλλαγές μνήμης του καλούντος νήματος.

Αν λοιπόν στο προηγούμενο παράδειγμα είχαμε αλληλεξάρτηση μόνο του πρώτου στοιχείου του πίνακα με το τελευταίο, αντί να τερματίσουμε όλο τον kernel μπορούμε να βάλουμε μία κλήση μίας συνάρτησης threadfence μαζί με μία συνθήκη

ώστε να εκτελεστεί μόνο από το πρώτο νήμα. Το αν συμφέρει να την αποφύγουμε και να χωρίσουμε τον kernel ή να την χρησιμοποιήσουμε μπορούμε να το βρούμε μόνο με δοκιμή. Σε γενικές γραμμές όμως αν επιβάλλει συχνά αναμονές στα υπόλοιπα νήματα[14, 27], είναι καλύτερο να χωρίσουμε τον kernel. Η εκτέλεση ενός kernel γενικά έχει αμελητέα καθυστέρηση.

Οι συναρτήσεις threadfence είναι οι εξής:

- threadfence_block : Περιμένει έως ότου όλες οι προσβάσεις στην καθολική ή την κοινόχρηστη μνήμη που εκτελούνται από το καλών νήμα, να γίνουν ορατές από όλα τα άλλα νήματα του ίδιου block.

```
void __threadfence_block();
```

- threadfence : Περιμένει έως ότου όλες οι προσβάσεις στην καθολική ή την κοινόχρηστη μνήμη που εκτελούνται από το καλών νήμα γίνουν ορατές από:
  1. όλα τα νήματα του ίδιου block για τις προσβάσεις στην κοινόχρηστη μνήμη.
  2. όλα τα νήματα στο σύνολο τους για τις προσβάσεις στην καθολική μνήμη.

```
void __threadfence();
```

- threadfence_system : Περιμένει έως ότου όλες οι προσβάσεις στην καθολική ή την κοινόχρηστη μνήμη που εκτελούνται από το καλών νήμα γίνουν ορατές από:
  1. όλα τα νήματα του ίδιου block για τις προσβάσεις στην κοινόχρηστη μνήμη.
  2. όλα τα νήματα στο σύνολο τους για τις προσβάσεις στην καθολική μνήμη.
  3. όλα τα νήματα του κεντρικού επεξεργαστή για τις προσβάσεις σε κλειδωμένη μνήμη συστήματος.

```
void __threadfence_system();
```

3.2.7 Καλές πρακτικές

Σε αυτή την ενότητα θα συμμαζέψουμε τα όσα είπαμε ως τώρα σχετικά με την επίτευξη επιδόσεων αλλά και θα τα δούμε λίγο πιο αναλυτικά αν χρειαστεί.

Ποια προβλήματα μπορούν να επιταχυνθούν σημαντικά από μία κάρτα γραφικών;

Λόγω των μεγάλων διαφορών μεταξύ ενός επεξεργαστή γραφικών και ενός κεντρικού επεξεργαστή, ένα πρόβλημα πρέπει να χωρίζεται έτσι ώστε ο κάθε επεξεργαστής να εκτελεί το κομμάτι του προβλήματος στο οποίο είναι καλύτερος.

- Η κάρτα γραφικών είναι ιδανική για προβλήματα ή τμήματα τους, τα οποία περιέχουν υπολογισμούς οι οποίοι μπορούν να εκτελεστούν σε πολλά στοιχεία δεδομένων ταυτόχρονα (παράλληλα). Συνήθως τέτοια προβλήματα περιέχουν αριθμητικούς υπολογισμούς σε μεγάλες ενότητες δεδομένων (όπως οι πίνακες), όπου η ίδια διεργασία μπορεί να γίνει ταυτόχρονα σε χιλιάδες ή εκατομμύρια στοιχεία. Γενικά, για να είναι ένας αλγόριθμος αποδοτικός στην κάρτα γραφικών θα πρέπει να κάνει χρήση μεγάλου αριθμού νημάτων[11].

- Οι προσβάσεις στην μνήμη από τα νήματα θα πρέπει να έχουν κάποια συνοχή. Πρέπει να τηρούνται ορισμένα πρότυπα κατά τις προσβάσεις στην μνήμη ώστε αυτές να γίνονται με όσο το δυνατόν λιγότερες συναλλαγές.

- Για να χρησιμοποιήσουμε την κάρτα γραφικών, θα πρέπει πρώτα να μεταφέρουμε τα δεδομένα μας σε αυτήν μέσω του διαύλου PCI Express (PCIe). Αυτές οι μεταφορές είναι γενικά αργές και η πολυπλοκότητα των υπολογισμών που θα εκτελεστούν στην κάρτα γραφικών θα πρέπει να δικαιολογεί το πλήθος και μέγεθος των μεταφορών. Για παράδειγμα, αν μεταφέρουμε δύο πίνακες $N \times N$ μόνο για να προσθέσουμε τα στοιχεία τους και να επιστρέψουμε το αποτέλεσμα τους, η επιτάχυνση δεν θα είναι σημαντική.

- Τα δεδομένα θα πρέπει να είναι δυνατό να μείνουν στην μνήμη της συσκευής για όσο το δυνατόν μεγαλύτερο διάστημα. Επειδή οι μεταφορές μνήμης μεταξύ των δύο συστημάτων είναι ακριβές, θα πρέπει τα δεδομένα να μένουν στην κάρτα γραφικών κατά διαδοχικές εκτελέσεις kernel.

Μέγιστη απόδοση

Για να επιτύχουμε την μέγιστη απόδοση για τον αλγόριθμό μας, θα πρέπει να ακολουθήσουμε κάποιους κανόνες για τους οποίους μιλήσαμε. Εδώ θα κατηγοριοποιήσουμε αυτούς τους κανόνες ως προς την σημαντικότητα τους[14, 11, 16].

1. Πρώτα κάποιος θα πρέπει να επικεντρωθεί στην παραλληλοποίηση του σειριακού κώδικα. Το κέρδος σε ταχύτητα που θα έχει ένας αλγόριθμος εξαρτάται σχεδόν ολοκληρωτικά από το πόσο μπορεί να παραλληλοποιηθεί. Κώδικας ο οποίος δεν μπορεί να γίνει παράλληλος σε μεγάλο βαθμό θα πρέπει να εκτελείται από τον κεντρικό επεξεργαστή. Το μεγαλύτερο (σχετικό) κέρδος σε ταχύτητα έναντι σειριακού κώδικα, μας το δίνει η μεγιστοποίηση του παράλληλου τμήματος ενός αλγορίθμου, και όχι η μεγιστοποίηση των

διαθέσιμων επεξεργαστών. Ο νόμος του Amdahl[18] μας δίνει την μέγιστη επιτάχυνση που μπορεί να αναμένεται καθώς παραλληλοποιούμε τμήματα σειριακού κώδικα.

$$S \leq \frac{1}{(1-P) + \frac{P}{N}}$$

Όπου S η μέγιστη επιτάχυνση, P είναι το κλάσμα του συνολικού χρόνου εκτέλεσης του σειριακού κώδικα που παίρνεται από το παράλληλο τμήμα. Ν είναι ο αριθμός των επεξεργαστών που επεξεργάζονται το παράλληλο τμήμα του κώδικα.

2. Εφαρμογή των κανόνων συναλλαγών της καθολικής μνήμης όπου αυτό είναι δυνατό. Οι συναλλαγές θα πρέπει να είναι όσο το δυνατόν συνεχείς ως προς το τμήμα της μνήμης και ευθυγραμμισμένες σε πολλαπλάσια των 32,64 ή 128 bytes από την αρχή του τμήματος της μνήμης.

3. Ελαχιστοποίηση των μεταφορών μνήμης μεταξύ κεντρικού επεξεργαστή και κάρτας γραφικών. Ακόμα και αν αυτό συνεπάγεται εκτέλεση κάποιων μικρών kernel οι οποίοι δεν είναι αποτελεσματικοί στην κάρτα γραφικών, έτσι ώστε να αποφύγουμε ενδιάμεσες μεταφορές. Επίσης, η αποδοτικότητα μίας μεγάλης μεταφόρας, είναι μεγαλύτερη από την αποδοτικότητα πολλών μικρών μεταφορών.

4. Αποφυγή των διακλαδώσεων της ροής ενός προγράμματος μέσα σε μία δύνη.

5. Πρέπει να έχουμε το λιγότερο 192 ενεργά νήματα συνολικά σε κάθε πολυεπεξεργαστή για να καλύψουμε τις καθυστερήσεις των αριθμητικών εντολών. Αντίστοιχα, για τις προσβάσεις στην μνήμη, 256 νήματα είναι συνήθως αρκετά.

6. Οι προσβάσεις στην κοινόχρηστη μνήμη θα πρέπει να γίνονται με τρόπο ώστε να αποφεύγονται τα συγκρούσεις τραπεζών.

7. Το πλήθος των νημάτων ανά block θα πρέπει να είναι κάποιο πολλαπλάσιο του 32, επειδή σε αυτή την περίπτωση έχουμε την καλύτερη υπολογιστική αποδοτικότητα και συνήθως έτσι επιτυγχάνεται και συνεχής και ευθυγραμμισμένη πρόσβαση στην μνήμη αρκετά πιο εύκολα.

8. Χρήση της κοινόχρηστης μνήμης, αντί της καθολικής, όπου αυτό καθίσταται δυνατό.

9. Αποφυγή συγχρονισμών όσο αυτό είναι δυνατό.

10. Χρήση των γρήγορων μαθηματικών συναρτήσεων χαμηλής ακρίβειας, όπου η ακρίβεια δεν είναι σημαντική.

11. Όλα τα νήματα ενός μισού της δύνης θα πρέπει να κάνουν πρόσβαση στην ίδια θέση της μνήμης σταθερών.

12. Αποφυγή πράξεων modulo και διαιρέσεων ακέραιων, χρήση τελεστών bit στην θέση τους όπου αυτό είναι δυνατό.

3.3 Η εργαλειοθήκη CUDA Toolkit

Μιλήσαμε για την κάρτα γραφικών, τα χαρακτηριστικά της και πως να την προγραμματίσουμε. Για να μεταγλωττίσουμε ένα πρόγραμμα κάρτας γραφικών όμως χρειαζόμαστε ειδικό μεταγλωττιστή παράλληλα με τον συνηθισμένο μεταγλωττιστή. Για να το τρέξουμε χρειαζόμαστε κάποιες ειδικές βιβλιοθήκες και τέλος για να αναλύσουμε το πρόγραμμα αλλά και να βρούμε προβλήματα χρειαζόμαστε ειδικά εργαλεία. Εδώ θα αναφερθούμε μόνο στην έκδοση της εργαλειοθήκης για λειτουργικό σύστημα Linux, αλλά ο τρόπος και τα εργαλεία δεν διαφέρουν σχεδόν καθόλου με την έκδοση της εργαλειοθήκης για άλλα λειτουργικά συστήματα όπως τα Windows και το Mac OSX.

Αρχικά κάποιος θα πρέπει να κατεβάσει την νεότερη έκδοση cuda-enabled οδηγού για την κάρτα γραφικών του από την ιστοσελίδα της NVIDIA[23] και να την εγκαταστήσει. Στην συνέχεια θα πρέπει να κατεβάσει την εργαλειοθήκη που απευθύνεται στο σύστημα του (32bit, 64bit και διανομή του Linux) από την ίδια ιστοσελίδα και να τρέξει το script εγκατάστασης.

Παράλληλα με την εργαλειοθήκη, για την μεταγλώττιση θα χρειαστούμε και τον gcc[24] με μεγάλη έκδοση 4.3.x[25]. Ο gcc συνήθως είναι εξαρχής εγκατεστημένος στις διανομές Linux, αλλά είναι πολύ πιθανό να έχουμε την νεότερη έκδοση 4.4.x η οποία δεν υποστηρίζεται. Σε κάθε περίπτωση θα πρέπει να κατεβάσουμε την έκδοση 4.3.x. Το λειτουργικό Linux υποστηρίζει πολλαπλές εκδόσεις του gcc, οπότε δεν θα υπάρξει πρόβλημα.

Πριν συνεχίσουμε στον μεταγλωττιστή, θα πρέπει να ενημερώσουμε κάποια paths του συστήματος, έτσι ώστε να βλέπουν τα εκτελέσιμα και τις βιβλιοθήκες της εργαλειοθήκης. Ανοίγουμε ένα τερματικό και δίνουμε τις παρακάτω εντολές:

```
export PATH = /usr/local/cuda/bin:\$PATH
export LD_LIBRARY_PATH = /usr/local/cuda/lib:\$LD_LIBRARY_PATH
```

[23] http://developer.nvidia.com/object/cuda_3_0_downloads.html#Linux
[24] The GNU Compiler Collection. http://gcc.gnu.org/
[25] Δεν έχει σημασία αν θα είναι ο 4.3.1 ή 4.3.2 αρκεί το τα δυο πρώτα να είναι 4.3.

3.3.1 Ο μεταγλωττιστής nvcc

Εφόσον το πρόγραμμα μας είναι έτοιμο, μπορούμε να το μεταγλωττίσουμε με τον nvcc. Προσοχή εδώ, τα αρχεία που περιέχουν κλήσεις συναρτήσεων kernel ή του CUDA θα πρέπει να έχουν κατάληξη .cu! Έστω ότι το πρόγραμμα μας αποτελείται από τα αρχεία main.cu, kernel.cu. Θα χρησιμοποιήσουμε τον nvcc όπως κάθε άλλο μεταγλωττιστή:

```
nvcc main.cu kernel.cu -o progname
```

Το οποίο θα μας δώσει το εκτελέσιμο με όνομα progname.

Είδαμε την πολύ βασική λειτουργία του nvcc. Τώρα θα δούμε λίγο πιο εξειδικευμένες λειτουργίες του.

- –ptxas-option. PTX είναι η γλώσσα μηχανής της κάρτας γραφικών, αντίστοιχη της Assembly για τον κεντρικό επεξεργαστή. Αυτή η επιλογή μας επιτρέπει να δώσουμε κάποια ειδικά ορίσματα. Εμάς μας απασχολεί μόνο η επιλογή -v. Αυτή λέει στον μεταγλωττιστή να μας δώσει αναλυτικές πληροφορίες για το μέγεθος της κοινόχρηστης μνήμης ανά block, το πλήθος των καταχωρητών, την μνήμη σταθερών, αλλά και την τοπική μνήμης που θα χρησιμοποιήσει το πρόγραμμα μας. Με αυτή την εντολή ελέγχουμε το αληθινό τελικό πλήθος των καταχωρητών που δεσμεύονται από κάθε νήμα.

  ```
  nvcc --ptxas-options=-v main.cu kernel.cu -o progname
  ```

  Το παραπάνω θα μας δώσει για έναν kernel με το όνομα gpufunc:

  ```
  ptxas info    : Compiling entry 0function '_Z24gpufuncPlPiPf'
  ptxas info    : Used 17 registers, 64+16 bytes smem, 8 bytes cmem[0],
                  28 bytes cmem[1]
  ```

- –compiler-bindir. Δίνει εντολή στον μεταγλωττιστή να μην χρησιμοποιήσει τον προεπιλεγμένο μεταγλωττιστή gcc του συστήματος, αλλά έναν άλλο που ορίζουμε στην συνέχεια. Αυτή η εντολή χρησιμεύει πολύ, καθώς συνήθως έχουμε προεπιλεγμένη μία έκδοση 4.4.x.

  ```
  nvcc --compiler-bindir /usr/bin/gcc-4.3 main.cu kernel.cu
   -o progname
  ```

- -Ox. Δίνει εντολή στον μεταγλωττιστή να εφαρμόσει βελτιστοποίηση επιπέδου x στο κομμάτι του κώδικα που αφορά την κάρτα γραφικών

  ```
  nvcc -O3 main.cu kernel.cu -o progname
  ```

- –compiler-options. Μεταφέρει τα ορίσματα της επιλογής στον μεταγλωττιστή gcc. Στο παράδειγμα λέμε στον nvcc να μεταφέρει το όρισμα -O3 στον gcc, ώστε ο gcc να εφαρμόσει βελτιστοποίηση επιπέδου 3 στον κώδικα που αφορά τον κεντρικό επεξεργαστή.

```
nvcc --compiler-options '-O3' main.cu kernel.cu -o progname
```

3.3.2 Υπολογιστής πληρότητας, CUDA Visual Profiler και ο cuda-debugger

Εδώ θα αναφέρουμε μερικά βοηθητικά εργαλεία της εργαλειοθήκης τα οποία μας βοηθούν να αναλύσουμε το πρόγραμμα μας ή να βρούμε bugs.

CUDA Occupancy Calculator, ο υπολογιστής πληρότητας

Ο υπολογιστής πληρότητας είναι ένα μικρό αρχείο υπολογιστικού φύλλου. Δεν αποτελεί ουσιαστικά μέρος της εργαλειοθήκης αλλά είναι ελεύθερος για κατέβασμα[26]. Σε αυτό επιλέγουμε αρχικά την υπολογιστική ικανότητα της κάρτας γραφικών μας και στην συνέχεια δίνουμε το μέγεθος ενός block σε νήματα, το πλήθος των καταχωρητών και το μέγεθος της κοινόχρηστης μνήμης ανά block και μας επιστρέφει την πληρότητα μαζί με μερικά διαγράμματα για το πως θα αλλάξει αυτή αν μεταβάλουμε την τιμή αυτών των παραμέτρων.

Cuda Visual Profiler

Αποτελεί μέρος της εργαλειοθήκης. Είναι ένα εξαιρετικά χρήσιμο εργαλείο καθώς μας δίνει για κάθε κλήση ενός kernel αλλά και για τις συναλλαγές μνήμης[13]. Καλείται με την εντολή

```
./usr/0local/cuda/cudaprof/bin/cudaprof
```

καθώς συνήθως δεν είναι στο PATH του λειτουργικού.

Συγκεκριμένα με τον Cuda Visual Profiler, μπορούμε να δούμε:

1. πόσες φορές κλήθηκε ένας kernel
2. πόση ώρα έκανε να εκτελεστεί κάθε φορά
3. πόσες αποκλίνουσες ροές είχαν οι δύνες (if...else)
4. πόσες coalesced και uncoalesced προσβάσεις έγιναν

[26] http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

5. cache hit και miss για την μνήμη σταθερών
6. cache hit και miss για την μνήμη υφής
7. διαγράμματα ροής χρόνου ως προς κλήσεις συναρτήσεων.

Ειδικά το τελευταίο είναι αρκετά χρήσιμο καθώς μας δίνει μία γενική εικόνα που πάσχει το πρόγραμμα μας. Αν για παράδειγμα οι μεταφορές μνήμης θέλουν περισσότερη ώρα από την εκτέλεση των ίδιων των δεδομένων.

Ο Cuda Debugger

Η εργαλειοθήκη περιέχει άλλο ένα πολύ χρήσιμο εργαλείο. Τον Cuda Debugger[12]. Ο Cuda Debugger έχει την δυνατότητα να τρέξει γραμμή προς γραμμή ένα εκτελέσιμο που περιέχει κλήσεις προς μία κάρτα γραφικών. Παρόλα αυτά, οι ικανότητες περιορίζονται μόνο στα κομμάτια κώδικα που εκτελούνται στην κάρτα γραφικών. Για το υπόλοιπο εκτελέσιμο θα πρέπει να χρησιμοποιήσουμε έναν απλό debugger. Ο Cuda Debugger μας δίνει επίσης την δυνατότητα να βλέπουμε ανά πάσα στιγμή τι τρέχει σε κάθε ενεργό νήμα κάθε πολυεπεξεργαστή. Ο Cuda Debugger καλείται με την εντολή cuda-gdb και η χρήση του δεν διαφέρει πολύ από τον κλασσικό gnu debugger.

Κεφάλαιο 4

Οι Αλγόριθμοι Metropolis και Wolff σε GPU

Σε αυτό το κεφάλαιο θα παρουσιάσουμε έναν αλγόριμο Metropolis σε GPU μαζί με δύο παραλλαγές του, καθώς και μία προσπάθεια για έναν αλγόριθμο Wolff σε GPU. Όλοι οι αλγοριθμοί χρησιμοποιούν δισδιάστατο τετραγωνικό πλέγμα. Στην συνέχεια, θα παρουσιάσουμε τα αποτελέσματα των προσομοιώσεων τα οποία και θα αναλύσουμε. Τέλος, θα συγκρίνουμε τα αποτελέσματα αλλά και τις επιδόσεις με τον αλγόριθμο των Newman and Barkema ο οποίος χρησιμοποιείται και στο μάθημα της Υπολογιστικής Φυσικής ΙΙ: Προτυποποίηση.

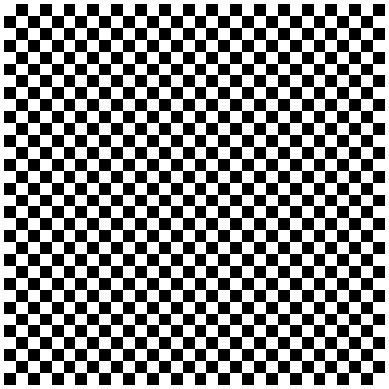
4.1 Ο αλγόριθμος GPU Metropolis και οι παραλλαγές του

Πριν αναλύσουμε τον αλγόριθμο GPU Metropolis και τις παραλλαγές του, θα πρέπει πρώτα να αναφερθούμε σε μία ειδική περίπτωση τετραγωνικού πλέγματος.

4.1.1 Το τετραγωνικό πλέγμα checkerboard

Θα υπενθυμίσουμε εδώ ότι το τετραγωνικό πλέγμα checkerboard (σχήμα 4.1)είναι μία ειδική περίπτωση τετραγωνικού πλέγματος. Διαφέρει όμως από την γενικευμένη μορφή του τετραγωνικού πλέγματος, ως προς το ότι χωρίζεται σε δύο υποπλέγματα όπως τα λευκά και τα μαύρα κελιά μίας σκακιέρας. Αυτά τα υποπλέγματα, για την περίπτωση του προτύπου Ising με αλληλεπιδράσεις κοντινών γειτόνων, είναι ανεξάρτητα μεταξύ τους. Δηλαδή, οι γείτονες ενός spin στο λευκό υποπλέγμα θα ανήκουν στο μαύρο υποπλέγμα. Έτσι τώρα μπορούμε να ελέγξουμε και να ανανεώσουμε όλα τα spin ενός υποπλέγματος, θεωρώντας το δεύτερο σταθερό. Στην συνέχεια μπορούμε να ελέγξουμε και να ανανεώσουμε το δεύ-

τερο πλέγμα, το οποίο θα έχει γείτονες στο πρώτο υπόπλεγμα, χρησιμοποιώντας ως κατάσταση spin των γειτόνων αυτές που βρήκαμε πριν.

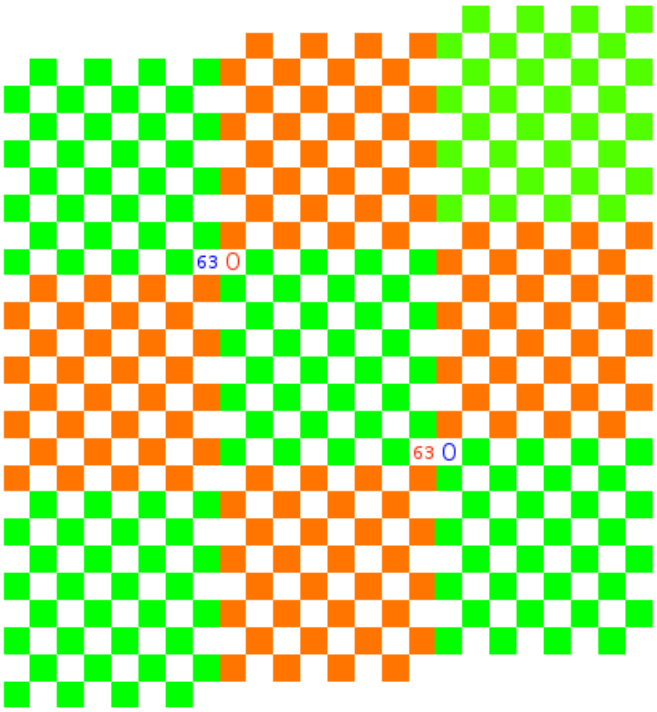


Σχήμα 4.1: Το τετραγωνικό πλέγμα checkerboard

Checkerboard και συνοριακές συνθήκες

Αναφερθήκαμε στις συνοριακές συνθήκες και στον ρόλο τους στην προσημείωση ενός άπειρου πλέγματος στην υποενότητα 1.2.2. Εμείς αποφασίσαμε να χρησιμοποιήσουμε τις ελικοειδής. Όπως έχουμε πει, οι ελικοειδής προτιμούνται καθώς είναι πιο γρήγορες. Αυτό όμως δεν είναι απαραίτητα αλήθεια στην περίπτωση των gpu αλγορίθμων. Όπως θα δούμε, θα κάνουμε εκτεταμένη χρήση υφών, και οι υφές έχουν την ιδιότητα να μας δίνουν τις περιοδικές συνθήκες αν ελέγξουμε τον γείτονα ενός συνοριακού spin. Αυτό γίνεται εφόσον έχουμε θέσει το spin ως δισδιάστατο και με την επιλογή Wrap. Με αυτές τις παραμέτρους, αν υπερχειλίσουμε την μνήμη ως προς την διάσταση x κατά ένα στοιχείο, η υφή θα μας επιστρέψει το στοιχείο με συντεταγμένη $x = 0$! Στην περίπτωση του αλγορίθμου gpu που δεν κάνει χρήση υφών, οι ελικοειδής θα είναι πιο γρήγορες. Εκτός του ότι κατά την ανάπτυξη του αλγορίθμου, δεν ήταν απολύτως γνωστές όλες οι ιδιότητες των υφών, το πρόγραμμα αναφοράς χρησιμοποιεί ελικοειδής. Μας ενδιαφέρει πρωτίστως να επιβεβαιώσουμε ότι ο αλγόριθμος μας δουλεύει σωστά, επομένως προέχει η εναρμόνιση με τον κώδικα αναφοράς. Εφόσον ο αλγόριθμος αποδειχθεί ότι δουλεύει σωστά, τότε θα προχωρήσουμε και στην περαιτέρω ανάπτυξη και βελτιστοποίηση του.

Οι περιοδικές συνθήκες είναι καλύτερες γενικά για το πλέγμα checkerboard, καθώς τα spin κάθε υποπλέγματος δεν αλληλεπιδρούν ποτέ. Αυτό όμως δεν ισχύει για τις ελικοειδής, αρκεί να δούμε τι συμβαίνει στο άρτιο τετραγωνικό πλέγμα checkerboard με ελικοειδής συνθήκες του σχήματος 4.2. Βλέπουμε ότι η πλεγματική θέση 0 του κεντρικού πλέγματος, η οποία βρίσκεται στο πάνω - αριστερό



Σχήμα 4.2: Το τετραγωνικό πλέγμα checkerboard με ελικοειδής συνοριακές. Η πλεγματική θέση 0 του κεντρικού πλέγματος, η οποία βρίσκεται στο πάνω - αριστερό σύνορο, έχει αριστερό γείτονα την πλεγματική θέση 63. Οι πλεγματικές θέσεις 0 και 63 ανήκουν στο ίδιο υπόπλεγμα, αλλά παρόλα αυτά δεν είναι ανεξάρτητες!

σύνορο, έχει αριστερό γείτονα την πλεγματική θέση 63. Οι πλεγματικές θέσεις 0 και 63 ανήκουν στο ίδιο υπόπλεγμα, αλλά παρόλα αυτά δεν είναι ανεξάρτητες. Σε άρτιο τετραγωνικό πέλμα checkerboard, τα spin ενός υποπλέγματος που βρίσκονται στο αριστερό ή δεξιό σύνορο θα έχουν έναν γείτονα, αριστερά ή δεξιά τους αντίστοιχα, ο οποίος δεν θα είναι ανεξάρτητος του υποπλέγματος αφού θα είναι μέλος του!

Αυτό το γεγονός θα κάνει το πρόβλημα μας απλά λίγο πιο πολύπλοκο, καθώς θα πρέπει να είμαστε λίγο πιο προσεκτικοί στο πως θα χειριστεί ο αλγόριθμος αυτές τις περιπτώσεις. Σε ένα περιττό πλέγμα με ελικοειδές συνοριακές συνθήκες, θα έχουμε το ίδιο πρόβλημα, αλλά στο πάνω και στο κάτω σύνορο! Και στις δύο περιπτώσεις, όλα τα άλλα spin δεν αλληλεπιδρούν.

4.1.2 Ο γενικός αλγόριθμος καθολικής μνήμης

Καταρχήν, ας ξεκινήσουμε με την φιλοσοφία του αλγορίθμου. Εφόσον τα spin ενός υποπλέγματος δεν αλληλεπιδρούν, θα ενημερώσουμε πρώτα όλα τα spin ενός υποπλέγματός και στην συνέχεια σε έναν δεύτερο kernel τα spin του δεύτερου. Αυτό γίνεται επειδή όπως έχουμε πει δεν μπορούμε να επιβάλλουμε συγχρονισμό σε όλη την συσκευή. Επομένως, αν ενημερώσουμε το πρώτο υπόπλεγμα και στον ίδιο kernel επιχειρήσουμε να ενημερώσουμε και το δεύτερο, θα πάρουμε λανθασμένα αποτελέσματα, αφού δεν μας εγγυάται κανείς ότι τα spin των πλεγματικών θέσεων ενός υποπλέγματος θα έχουν ενημερωθεί πριν διαβαστούν ως γείτονες από το άλλο υπόπλεγμα.

Για αρχή λοιπόν χωρίζουμε τον αλγόριθμο σε δύο kernel. Ο πρώτος αναλαμβάνει το πρώτο υπόπλεγμα και θα ονομάζεται μαύρος, ενώ ο δεύτερος αντίστοιχα λευκός.

Τώρα πρέπει να βρούμε πως θα χωρίσουμε τα νήματα. Μία καλή αντιστοιχία νήματος - πλέγματος είναι το κάθε νήμα να αναλαμβάνει μία περιοχή 2x2 όλου του πλέγματος[19]. Αυτή η περιοχή περιλαμβάνει δύο spin του πρώτου υποπλέγματος και δύο spin του δεύτερου υποπλέγματος. Σε κάθε kernel το κάθε νήμα θα ανανεώνει μόνο τα δύο spin που του αναλογούν και ανήκουν στο αντίστοιχο υποπλέγμα. Επομένως, όλο το πλέγμα χωρίζεται σε περιοχές 2x2. Αυτό είναι το domain decomposition ή αλλιώς data slicing και ορίζει το πώς τα δεδομένα μας μοιράζονται σε κάθε νήμα. Η λογική αυτού του διαχωρισμού είναι ότι όλα τα νήματα θα επεξεργάζονται δεδομένα. Όντως, αν είχαμε μία πλεγματική θέση για κάθε νήμα, τότε ο kernel που επεξεργάζεται το πρώτο υποπλέγμα, θα έχει τα μισά νήματα ανενεργά.

Αποφασίσαμε μεν να πάμε δια της πεπατημένης οδού, αλλά υπάρχουν και άλλοι τρόποι να έχουμε όλα τα νήματα ενεργά. Ας δούμε πάλι το σχήμα 4.1 και ας φανταστούμε ότι σπρώχνουμε όλες τις μαύρες πλεγματικές θέσεις προς τα πάνω. Θα καταλήξουμε με ένα πλέγμα του οποίου η y διάσταση πλέον θα είναι μισή. Το μόνο που αλλάζει με το προηγούμενο data slicing είναι ότι πλέον ένα νήμα, από την τοπική σκοπιά του, αντί να ανανεώνει τις πλεγματικές θέσεις (0,0) και (1,1) θα ανανεώνει μόνο μία. Συγκεκριμένα το πρώτο νήμα θα ανανεώσει την θέση (0,0) και το δεύτερο την (1,1). Αλλά τώρα ο μετασχηματισμός που θα μας δώσει την πλεγματική θέση για κάθε νήμα θα γίνει αρκετά πιο πολύπλοκος, αφού κάθε νήμα θα υπολογίζει μία πλεγματική θέση η οποία θα εξαρτάται από το αν το νήμα είναι άρτιο ή περιττό. Μπορεί να μην φαίνεται κακή ιδέα, αλλά στην πραγματικότητα είναι. Και αναφέρθηκε ως παράδειγμα της πολυπλοκότητας των καρτών γραφικών. Γιατί; Αν κάθε νήμα αντιστοιχεί σε μία και μόνο πλεγματική θέση βάση της αρτιότητας του, τότε αυτό σημαίνει ότι θα υπάρξει αποκλίνουσα ροή. Ας το δούμε από την σκοπιά μίας δύνης. Η δύνη περιέχει έστω τα 32 πρώτα νήματα. Το πρώτο

νήμα, ως περιττό θα διαβάσει την ”πάνω” θέση, ενώ το δεύτερο την κάτω και θα συνεχίσουν όλα εναλλάξ. Αυτό σημαίνει ότι *πάντα* σε μία δύνη τα μισά νήματα θα έχουν διαφορετική ροή! Αυτό δεν συμβαίνει στην 2x2 κατανομή, επειδή *κάθε* νήμα θα ανανεώσει πρώτα την πάνω πλεγματική θέση και στην συνέχεια την κάτω.

Αφού βρήκαμε πως θα μοιράσουμε τα δεδομένα μας σε κάθε νήμα, πάμε να δούμε τι θα χρειαστούμε για το κάθε ένα. Εφόσον όλα τα νήματα εργάζονται παράλληλα, θα πρέπει να παράγουν τους δικούς τους τυχαίους αριθμούς. Υπάρχουν δύο τρόποι να γίνει αυτό, ο πρώτος είναι να προϋπολογίζουμε τους τυχαίους αριθμούς μέσα σε έναν kernel και ο δεύτερος να περάσουμε τους ίδους τους seeds της γεννήτριας τυχαίων αριθμών μέσα στον kernel. Στην υλοποίηση μας επιλέξαμε να περάσουμε τα seeds και όποτε χρειαστεί να υπολογίζουμε επί τόπου έναν τυχαίο αριθμό. Χρειαζόμαστε επομένως όσα seeds όσα και τα νήματα. Στην αρχή της κάθε προσομοιώσεις, τρέχουμε την γεννήτρια στην cpu και παράγουμε πλήθος seeds ίσο με το πλήθος των νημάτων. Λόγω του συσχετισμού των seeds, πολλαπλασιάζουμε κάθε φορά ένα seed με έναν τυχαίο αριθμό. Τα seeds θα αποθηκεύονται στην κοινόχρηστη μνήμη.

Χρειαζόμαστε επίσης κοινόχρηστη μνήμη για τις πιθανότητες αλλαγής του spin. Αυτές θα τις υπολογίσουμε στην cpu και θα τις περάσουμε αρχικά στην καθολική μνήμη στην αρχή της προσομοιώσεις. Εν συνεχεία, κάθε block θα αναλαμβάνει να τις περάσει στην κοινόχρηστη μνήμη του.

Η καρδιά του αλγορίθμου δεν διαφέρει ουσιαστικά από τον αλγόριθμο αναφοράς. Αρκεί να δείξουμε τον αλγόριθμο πλέον αλλά και το API του. Κάθε αλγόριθμος καλείται και παραμετροποιείται από ειδικές συναρτήσεις οι οποίες μπορούν να κληθούν από οποιοδήποτε πρόγραμμα. Οι εσωτερικές συναρτήσεις δεν είναι ορατές από το κυρίως πρόγραμμα. Αυτές οι ειδικές συναρτήσεις λειτουργούν δηλαδή σαν μία βιβλιοθήκη.

Οι εσωτερικές συναρτήσεις είναι οι εξής:

```
__global__ void cuda_met_black_kernel_g(long*,int*,float*);
__global__ void cuda_met_white_kernel_g(long*,int*,float*);
__global__ void cuda_E_Partial_g(int *d_s,int *Epartial);
__global__ void cuda_E_Recursive_g(int *Epartial);
__global__ void cuda_M_Partial_g(int *d_s,int *Mpartial);
__global__ void cuda_M_Recursive_g(int *Mpartial);

void msetConstants_g(cudaMetParams *mp);
```

Οι δύο πρώτες είναι οι kernel, οι υπόλοιπες τέσσερις αναλαμβάνουν να υπολογίσουν την ενέργεια και την μαγνήτιση, μέσω του ανηγμένου πλέγματος. Η συνάρτηση msetConstant_g αναλαμβάνει να ορίσει την μνήμη σταθερών.

Οι συναρτήσεις που βλέπει το κυρίως πρόγραμμα είναι:

```
void setMetParams_g(int *s,int start,int L,int N,
    int nsweeps,float *prob,long seeder,int visualize_sequence,
    int saveimage,cudaMetParams *mp);

void cudaAllocMet_g(int *s,cudaMetParams *mp);

void callMetKernel_g(int *s,int *E_sequence,int *M_sequence,
    cudaMetParams *mp);

void cleanMet_g(cudaMetParams *mp);
```

- Η συνάρτηση setMetParams_g αναλαμβάνει να παραμετροποιήσει την κάρτα γραφικών. Θα ορίσει το μέγεθος του grid βάσει του μεγέθους του πλέγματος, αλλά και θα κάνει και ελέγχουν λαθών για τα δεδομένα εισαγωγής. Τέλος, θα περάσει όλες τις παραμέτρους σε μία ειδική μεταβλητή cudaMetParams.
- Η συνάρτηση cudaAllocMet_g θα αναλάβει να δεσμεύσει μνήμη για όλα τα δεδομένα στην κάρτα γραφικών.
- Η συνάρτηση callMetKernel_g είναι η συνάρτηση που εκτελεί το simulation στην κάρτα γραφικών. Με το πέρας των Monte Carlo sweeps, θα μας επιστρέψει την συνολική ενέργεια και την συνολική μαγνήτιση σε κάθε sweep καθώς και το ίδιο το πλέγμα.
- Τέλος, η συνάρτηση cleanMet_g ελευθερώνει την μνήμη στην κάρτα γραφικών.

Ας δούμε τώρα και την γεννήτρια τυχαίων αριθμών. Ουσιαστικά δεν διαφέρει από την γεννήτρια του προγράμματος αναφοράς, απλά είναι ελαφρά τροποποιημένη για να καλείται από την κάρτα γραφικών. Δηλώνεται ως device ώστε να τρέχει στην κάρτα γραφικών και ως inline ώστε ο μεταγλωττιστής αντί να περνάει κλησεις στην συνάρτηση, να αντικαταστήσει απευθείας τις κλήσεις με τον κώδικα της συνάρτησης.

```
#define d_a 16807
#define d_m 2147483647
#define d_q 127773
#define d_r 2836
#define d_conv (1.0/(d_m-1))

inline  __device__ float mdev_frandom(long &d_seed){
  long d_l;

  d_l=d_seed/d_q;
  d_seed = d_a*(d_seed-d_q*d_l) - d_r*d_l;
  if(d_seed < 0) d_seed +=d_m;
  return d_conv*(d_seed-1);
}
```

4.1. Ο ΑΛΓΟΡΙΘΜΟΣ GPU METROPOLIS ΚΑΙ ΟΙ ΠΑΡΑΛΛΑΓΕΣ ΤΟΥ99

Ο kernel

Ας δούμε τώρα κομμάτι κομμάτι τον μαύρο kernel και πώς δουλευεί ο αλγόριθμος.

- Οι δείκτες...

```
TID_black_upleft  =
   (((threadIdx.x) + blockDim.x*blockIdx.x)*2
   + const_L*(((threadIdx.y) + blockDim.y*blockIdx.y)*2));
TID_black_bottomright =
   (((threadIdx.x) + blockDim.x*blockIdx.x)*2 +
   1 + const_L*(((threadIdx.y) + blockDim.y*blockIdx.y)*2 + 1));

tid_local = threadIdx.x + mBLOCK_SIZEx*threadIdx.y;

tid_global=(threadIdx.x + blockIdx.x * blockDim.x) +
  mBLOCK_SIZEx*gridDim.x*(threadIdx.y + blockIdx.y * blockDim.y);
```

Οι δείκτες TID αναφέρονται στην γραμμική θέση στο πλέγμα σε σχέση με τις συντεταγμένες ενός νήματος και του block. Οι δείκτες tid αναφέρονται στις τοπικές και στις γενικευμμένες γραμμικές συντεταγμένες ενός νήματος. Οι πολυπλοκότητα των δεικτών TID οφείλεται στο γεγονός ότι ένα νήμα αντιστοιχεί σε ένα τμήμα 2x2 του πλέγματος.

- Η μνήμη...

```
__shared__ long  ds_seeds[mBLOCK_SIZEx*mBLOCK_SIZEy];
__shared__ float ds_prob[5];

int nn;
int sum;
int delta;
int tid_local,tid_global;
int TID_black_upleft,TID_black_bottomright;

ds_seeds[tid_local] = d_seeds[tid_global];

if(tid_local<5)ds_prob[tid_local]=d_prob[tid_local];
__syncthreads();
...
... //main algorithm
...
__syncthreads();
d_seeds[tid_global] = ds_seeds[tid_local];
```

Εδώ προσέχουμε πως πρέπει να συγχρονίζουμε τα νήματα μετά την αντιγραφή της καθολικής μνήμης στην κοινόχρηστη, ώστε να είμαστε σίγουροι ότι οι τιμές στην κοινόχρηστη μνήμη θα είναι ορατές αμέσως μετά. Αντίστοιχα, πριν την αντιγραφή από την κοινόχρηστη στην καθολική, συγχρονίσουμε όλα τα νήματα του block, ώστε η αντιγραφή να είναι coalesced. Η αντιγραφή των πιθανοτήτων είναι μία τυπική περίπτωση αποκλίνουσας

ροής. Όμως εδώ, αυτό συμβαίνει μόνο για την πρώτη δύνη κάθε block και για μία πολύ ελαφριά εργασία.

Πριν συνεχίσουμε με τον κύριο αλγόριθμο, θα μιλήσουμε λίγο για το πως τα νήματα θα επεξεργαστούν το μαύρο υπόπλεγμα. Εφόσον κάθε νήμα αναλαμβάνει δύο πλεγματικές θέσεις κάθε υποπλέγματος, αυτές θα πρέπει να ενημερωθούν σειριακά. Δηλαδή, ένα νήμα πρώτα θα ενημερώσει την πρώτη θέση και στην συνέχεια την δεύτερη.

Πρέπει να προσέξουμε τις συνοριακές μας συνθήκες. Ο αλγόριθμος θα πρέπει να ελέγξει για συνοριακές συνθήκες για τις πλεγματικές θέσεις που ανήκουν σε block που είναι στο σύνορο.Αυτό όμως δεν θα δημιουργήσει αποκλίσεις στις δύνες, επειδή η συνθήκη εξαρτάται από το block, επομένως τα νήματα μίας δύνης δεν αποκλίνουν. Απλά μία δύνη ενός block στο σύνορο θα τρέξει λίγο διαφορετικά από μία άλλη οποιαδήποτε δύνη, αυτό όμως δεν έχει καμία επίπτωση στην ταχύτητα.

Αντίστοιχα, μετά την συνθήκη για το block θα μπορούσαμε κάλιστα να θέσουμε μία συνθήκη ώστε μόνο τα νήματα πάνω στο σύνορο να ελέγχουν τις συνοριακές συνθήκες. Έτσι θα αποφεύγαμε αυτόν τον έλεγχο για τα περισσότερα νήματα ενός block. Δεν αλλάζει απολύτως τίποτα, και στις δύο περιπτώσεις, οι δύνες που περιέχουν τις πλεγματικές θέσεις στο σύνορο, πάντα θα αποκλίνουν λίγο, αυτή όμως η απόκλιση είναι μικρή και η κάρτα γραφικών μπορεί να προβλέψει τέτοιες μικρές αποκλίσεις. Στην προηγούμενη παράγραφο είπαμε ότι οι δύνες δεν θα έχουν απόκλιση, υπάρχει μία βασική διαφορά, στην πρώτη παράγραφο αναφερόμαστε στην συνθήκη αν ένα block θα πρέπει να ελέγξει για συνοριακές ή όχι, εκεί δεν αποκλίνουν. Εδώ μιλάμε για τις συνοριακές τις ίδιες, σε αυτη την περίπτωση θα υπάρξει μικρή απόκλιση.

Είπαμε όμως ότι έχουμε πρόβλημα στο δεξιό και στο αριστερό σύνορο, καθώς εκεί πάντα ο γείτονας δεξιά ή αριστερά αντίστοιχα δεν είναι ανεξάρτητος από το υπόπλεγμα. Η λύση είναι απλή, αφού ελέγξουμε τις ”πάνω” πλεγματικές θέσεις και τις ανανεώσουμε, θα επιβάλουμε έναν συγχρονισμό treadfence, ώστε αν ένα άλλο block προσπαθήσει να ελέγξει τον γείτονα μίας πλεγματικής θέσης κάτω δεξιά, θα το κάνει αφού αυτός ο γείτονας έχει ενημερωθεί και η τιμή του είναι ορατή παντού!

- Η συνθήκη για διαχωρισμό των blocks σε συνοριακά και μή:

```
if(blockIdx.x==0 || blockIdx.y==gridDim.y-1 ||
    blockIdx.y==0 || blockIdx.x==gridDim.x-1){
    //Blocks που περιέχουν πλεγματικές θέσεις στο σύνορο
    ...
    ...
```

```
}
else{
//Blocks που περιέχουν μόνο εσωτερικές πλεγματικές θέσεις
...
}
```

- Ο έλεγχος συνοριακών συνθηκών στα συνοριακά blocks για τις ”πάνω” μαύρες πλεγατικές θέσεις κάθε νήματος.

```
if((nn = TID_black_upleft + 1)      >= const_N) nn -= const_N;
sum = d_S[nn];
if((nn = TID_black_upleft - 1)      <        0) nn += const_N;
sum += d_S[nn];
if((nn = TID_black_upleft + const_L) >=const_N) nn -= const_N;
sum += d_S[nn];
if((nn = TID_black_upleft - const_L) <       0) nn += const_N;
sum += d_S[nn];
...
//Μέτρηση αλλαγών στην ενέργεια και ανανέωση
...
__threadfence();
```

Εδώ βλέπουμε πως θα υπάρξει μία μικρή απόκλιση στις δύνες στο σύνορο. Σε μία δύνη με 32 νήματα, το πρώτο της νήμα και το νήμα 16 θα έχουν συνοριακή συνθήκη. Επομένως και τα δύο μισά της δύνης θα έχουν από μία μικρή απόκλιση. Προσέξτε την κλήση της threadfence, χρειάζεται καθώς αμέσως μετά θα ανανεώσουμε τις ”κάτω” αριστερά πλεγματικές θέσεις των block που βρίσκονται στο σύνορο και κάποιοι γείτονες θα εξαρτώνται από την πρώτη ανανέωση.

- Διαφορά της ενέργειας και flip

```
delta = sum*d_S[TID_black_upleft];
if(delta <= 0) d_S[TID_black_upleft] = -d_S[TID_black_upleft];
else if(mdev_frandom(ds_seeds[tid_local]) < ds_prob[delta])
    d_S[TID_black_upleft] = -d_S[TID_black_upleft];
```

- Ο έλεγχος συνοριακών συνθηκών στα συνοριακά blocks για τις ”κάτω” μαύρες πλεγματικές θέσεις κάθε νήματος.

```
if((nn = (TID_black_bottomright) + 1)      >= const_N) nn -= const_N;
sum = d_S[nn];
if((nn = (TID_black_bottomright) - 1)      <        0) nn += const_N;
sum += d_S[nn];
if((nn = (TID_black_bottomright) + const_L) >=const_N) nn -= const_N;
sum += d_S[nn];
if((nn = (TID_black_bottomright) - const_L) <       0) nn += const_N;
sum += d_S[nn];
...
...
```

Αντίστοιχα γίνεται και ο έλεγχος για τις πλεγματικές θέσεις των εσωτερικών blocks. Η διαφορά θα είναι ότι δεν χρειάζονται πλέον έλεγχοι για συνοριακές συνθήκες και ότι δεν χρειάζεται συγχρονισμός ανάμεσα στην πάνω μαύρη και την κάτω μαύρη πλεγματική θέση ενός νήματος καθώς είναι ανεξάρτητες.

Εν συνεχεία θα εκτελεστεί ο ”λευκός” kernel με την μόνη διαφορά ότι αντί για τις πλεγματικές θέσεις TID_black θα ανανεώσει τις πλεγματικές θέσεις TID_white.

```
TID_white_upright =
   (((threadIdx.x) + blockDim.x*blockIdx.x)*2 +
    1 + const_L*(((threadIdx.y) + blockDim.y*blockIdx.y)*2));
TID_white_bottomleft =
   (((threadIdx.x) + blockDim.x*blockIdx.x)*2 +
    const_L*(((threadIdx.y) + blockDim.y*blockIdx.y)*2 + 1));
```

Αυτός είναι ο πιο γενικός αλγόριθμός, από ότι είδαμε κάνει χρήση μόνο καθολικής μνήμης για τις πλεγματικές θέσεις. Μπορούμε να κάνουμε καλύτερα από αυτό, και όντως θα δούμε ότι με την χρήση υφών, η ταχύτητα του αλγορίθμου θα φτάνει να είναι ως και διπλάσια του γενικού αλγορίθμου!

4.1.3 Ο αλγόριθμος υφής

Αυτή η παραλλαγή του αλγορίθμου κάνει χρήση των υφών αντί της καθολικής μνήμης. Ας δούμε πως θα αλλάξει το κομμάτι ελέγχου των γειτόνων και των συνοριακών συνθηκών.

- Ο έλεγχος συνοριακών συνθηκών στα συνοριακά blocks για τις ”πάνω” μαύρες πλεγατικές θέσεις κάθε νήματος.

```
if((nn = TID_black_upleft + 1)      >= const_N) nn -= const_N;
sum = tex1Dfetch(tex1DLattice,nn);
if((nn = TID_black_upleft - 1)      <        0) nn += const_N;
sum += tex1Dfetch(tex1DLattice,nn);
if((nn = TID_black_upleft + const_L) >=const_N) nn -= const_N;
sum += tex1Dfetch(tex1DLattice,nn);
if((nn = TID_black_upleft - const_L) <       0) nn += const_N;
sum += tex1Dfetch(tex1DLattice,nn);
...
...
__threadfence();
```

  Βλέπουμε την χρήση της συνάρτησης tex1Dfetch() η οποία μας επιστρέφει την τιμή της υφής tex1DLattice με συντεταγμένες nn.

- Διαφορά της ενέργειας και flip

```
delta = sum*tex1Dfetch(tex1DLattice,TID_black_upleft);
if(delta <= 0) d_S[TID_black_upleft] =
    -tex1Dfetch(tex1DLattice,TID_black_upleft);
else if(mdev_frandom(ds_seeds[tid_local]) < ds_prob[delta])
     d_S[TID_black_upleft] = -tex1Dfetch(tex1DLattice,TID_black_upleft);
```

- Ο έλεγχος συνοριακών συνθηκών στα συνοριακά blocks για τις ”κάτω” μαύρες πλεγματικές θέσεις κάθε νήματος.

```
if((nn = (TID_black_bottomright) + 1)      >= const_N) nn -= const_N;
sum = d_S[nn];
if((nn = (TID_black_bottomright) - 1)      <        0) nn += const_N;
sum += d_S[nn];
if((nn = (TID_black_bottomright) + const_L) >=const_N) nn -= const_N;
sum += tex1Dfetch(tex1DLattice,nn);
if((nn = (TID_black_bottomright) - const_L) <       0) nn += const_N;
sum += tex1Dfetch(tex1DLattice,nn);
```

Αμέσως παρατηρούμε μία διαφορά. Οι γείτονες δεξιά και αριστερά διαβάζονται από την καθολική μνήμη. Πρέπει να γίνει με αυτόν τον τρόπο επειδή αυτοί οι γείτονες ανανεώθηκαν στην προηγούμενη περίπτωση. Οι υφές είναι cached και επομένως κανένας δεν μπορεί να μας εγγυηθεί ότι θα έχουν την νέα τιμή.

Αντίστοιχα γίνεται ο έλεγχος και για τις πλεγματικές θέσεις των εσωτερικών blocks. Όμως εκεί χρησιμοποιούμε μόνο μνήμη υφής, αφού δεν υπάρχει κάποια αλληλεξάρτηση. Δεν υπάρχει, δηλαδή, η περίπτωση να αλλάξει η τιμή ενός γείτονα όπως συμβαίνει στο δεξιό και το αριστερό σύνορο.

Εν συνεχεία θα εκτελεστεί και ο ”λευκός” kernel. Αυτή η παραλλαγή του αλγορίθμου εκτελείται ως και δύο φορές ταχύτερα από τον γενικό αλγόριθμο λόγω της χρήσης των υφών! Είναι επίσης γενικά πιο γρήγορος και από την επόμενη παραλλαγή, παρόλο που αυτή κάνει χρήση κοινόχρηστης μνήμης. Έχει αρκετό ενδιαφέρον να μελετήσουμε γιατί μπορεί να συμβαίνει κάτι τέτοιο.

4.1.4 Ο αλγόριθμος διαμοιρασμένης μνήμης και υφής

Σε αυτή την παραλλαγή του αλγορίθμου, εκτός των υφών θα κάνουμε και χρήση της κοινόχρηστης μνήμης για την αποθήκευση των πλεγματικών θέσεων. Εδώ έχουμε περισσότερες αλλαγές. Στην αρχή θα πρέπει να δηλώσουμε κοινόχρηστη μνήμη για το πλέγμα και στην συνέχεια θα πρέπει να αντιγράψουμε σε αυτή την καθολική μνήμη που αντιστοιχεί στις πλεγματικές θέσεις ενός block. Τέλος, θα πρέπει να αντιγράψουμε την κοινόχρηστη μνήμη πίσω στην καθολική μνήμη.

- Η δήλωση της κοινόχρηστης μνήμης για το πλέγμα

```
__shared__ int   ds_s[mBLOCK_SIZEx*mBLOCK_SIZEy];
```

- Πως τροποποιείται ο κυρίως αλγόριθμος. Θα αναφερθούμε μόνο στην περίπτωση των ”πάνω” και ”κάτω” μαύρων πλεγματικών θέσεων στο σύνορο, καθώς φαίνονται όλες οι τροποποιήσεις που θα εφαρμοστούν και στις υπόλοιπες περιπτώσεις.

```
if(blockIdx.x==0 || blockIdx.y==gridDim.y-1 || blockIdx.y==0 ||
     blockIdx.x==gridDim.x-1){

    ds_s[tid_local] = d_S[TID_black_upleft];
    __syncthreads();
    if((nn = TID_black_upleft + 1)      >= const_N) nn -= const_N;
    sum = tex1Dfetch(tex1DLattice,nn);
    if((nn = TID_black_upleft - 1)      <        0) nn += const_N;
    sum += tex1Dfetch(tex1DLattice,nn);
    if((nn = TID_black_upleft + const_L) >=const_N) nn -= const_N;
    sum += tex1Dfetch(tex1DLattice,nn);
    if((nn = TID_black_upleft - const_L) <       0) nn += const_N;
    sum += tex1Dfetch(tex1DLattice,nn);

    delta = sum*tex1Dfetch(tex1DLattice,TID_black_upleft);

    if(delta <= 0) ds_s[tid_local] =
        -tex1Dfetch(tex1DLattice,TID_black_upleft);
    else if(mdev_frandom(ds_seeds[tid_local]) < ds_prob[delta])
         ds_s[tid_local] = -tex1Dfetch(tex1DLattice,TID_black_upleft);

    __syncthreads();
    d_S[TID_black_upleft]=ds_s[tid_local];

    __threadfence();

    ds_s[tid_local] = d_S[TID_black_bottomright];
    __syncthreads();

    //Now black down
    if((nn = (TID_black_bottomright) + 1)      >= const_N) nn -= const_N;
    sum =d_S[nn];
    if((nn = (TID_black_bottomright) - 1)      <        0) nn += const_N;
    sum += d_S[nn];
    if((nn = (TID_black_bottomright) + const_L) >=const_N) nn -= const_N;
    sum += tex1Dfetch(tex1DLattice,nn);
    if((nn = (TID_black_bottomright) - const_L) <       0) nn += const_N;
    sum += tex1Dfetch(tex1DLattice,nn);

    delta = sum*tex1Dfetch(tex1DLattice,TID_black_bottomright);


    if(delta <= 0) ds_s[tid_local] =
         -tex1Dfetch(tex1DLattice,TID_black_bottomright);
    else if(mdev_frandom(ds_seeds[tid_local]) < ds_prob[delta])
         ds_s[tid_local] = -tex1Dfetch(tex1DLattice,TID_black_bottomright);

    __syncthreads();
    d_S[TID_black_bottomright]=ds_s[tid_local];
}
```

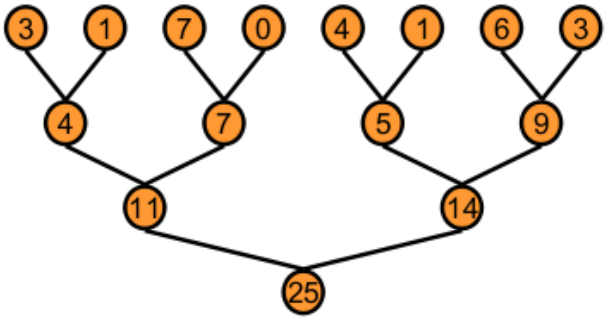
Βλέπουμε ότι κάθε φορά πρέπει να αντιγράφουμε την μνήμη και να συγχρονίζουμε. Και πάλι όμως τα οφέλη δε θα έπρεπε να είναι μεγαλύτερα λόγω της κοινόχρηστης μνήμης; Όχι απαραίτητα, καταρχήν η κοινόχρηστη μνήμη είναι βελτιστοποιημένη έτσι ώστε ένα νήμα να διαβάζει ή να γράφει μία τράπεζα της. Θα πρέπει επομένως να κάνουμε μία αρκετά πολύπλοκη και πιθανόν αργή προσπέλαση στην καθολική μνήμη αλλά και στην κοινόχρηστη μνήμη αν θέλουμε να συμπεριλάβουμε τους γείτονες. Αντί να κάνουμε κάτι τέτοιο, είναι προτιμότερο να χρησιμοποιήσουμε τις υφές. Επιπλέον, θα μπορούσαμε να ορίσουμε την κοινόχρηστη μνήμη ενός πλέγματος ως:

```
__shared__ int ds_s[2*mBLOCK_SIZEx*mBLOCK_SIZEy];
```

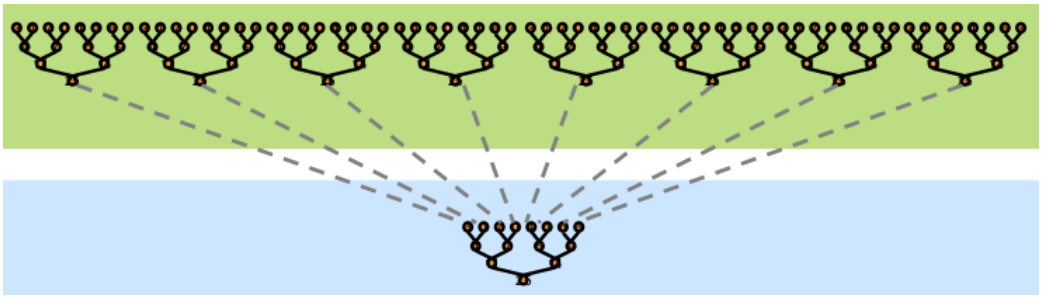
Έτσι θα αποφεύγαμε την ανάγκη να αντιγράφουμε συνέχεια την κοινόχρηστη μνήμη. Όμως, σε αυτή την περίπτωση οι απαιτήσεις ενός block σε κοινόχρηστη μνήμη εκτοξεύονται, αφού κάθε block θα απαιτεί 2Kbyte κοινόχρηστης μνήμης επιπλέον, επομένως η πληρότητα θα μειωθεί σημαντικά. Όπως έχουμε πει όμως, η πληρότητα δεν είναι το παν, και πιθανώς να μην έχει μεγάλη επίδραση στην αποδοτικότητα του αλγορίθμου. Ο λόγος που αυτή η παραλλαγή είναι ελαφρά πιο αργή από την προηγούμενη είναι ότι πρέπει να αντιγράψουμε ένα κομμάτι της μνήμης τέσσερις φορές (δύο από καθολική προς κοινόχρηστη και δύο στην αντίθετη κατεύθυνση) και ανάλογα την θερμοκρασία θα την χρησιμοποιήσουμε από πολύ σπάνια ως σχετικά συχνά. Επομένως, πολλές αντιγραφές, με μικρή χρησιμότητα, σε έναν αλγόριθμο ο οποίος έχει μικρό πλήθος αριθμητικών διεργασιών! Σε έναν kernel θα είμαστε περιορισμένοι από τις διεργασίες μνήμης ή τις αριθμητικές διεργασίες. Στον αλγόριθμο Metropolis είμαστε περιορισμένοι από τις διεργασίες μνήμης. Επομένως, προσθέτοντας επιπλέον συναλλαγές οι οποίες σε μεγάλο κομμάτι τους δεν θα χρειαστούν, δεν βοηθάει καθόλου τον αλγόριθμό μας. Αξίζει να σημειωθεί εδώ, ότι σε μεγάλα πλέγματα όπου το πλήθος των blocks είναι αρκετά μεγάλο, ένας πολυεπεξεργαστής μπορεί να κρύψει τις καθυστερήσεις που προκαλούνται από αυτές τις αντιγραφές της κοινόχρηστης μνήμης, επομένως οι διαφορές στον χρόνο εκτέλεσης του αλγορίθμου υφής και του αλγορίθμου κοινόχρηστης μνήμης εκμηδενίζονται.

4.2 Μέτρηση της ενέργειας και της μαγνήτισης

Φτάνουμε τώρα σε άλλη μία σημαντική ενότητα. Πώς θα μετρήσουμε την ενέργεια και την μαγνήτιση του πλέγματος; Αν επιστρέφουμε σε κάθε sweep το πλέγμα στην cpu τότε πολλά από τα οφέλη της χρήσης κάρτας γραφικών θα εξανεμιστούν. Έχουμε πει ότι οι μεταφορές cpu - gpu πρέπει να αποφεύγονται όσο είναι δυνατό. Αν ο αλγόριθμος εκτελεστεί για 100.000 sweeps και κάθε φορά επιστρέφουμε το πλέγμα για να μετρήσουμε τις ποσότητες που μας ενδιαφέρουν καταλαβαίνουμε



Σχήμα 4.3: Αλγόριθμος tree reduction. Κάθε νήμα θα αναλαμβάνει να προσθέσει δύο στοιχεία.



Σχήμα 4.4: Αναδρομικό tree reduction. Κάθε εκτέλεση επιστρέφει έναν πίνακα με την μισή διάσταση του αρχικού, και στοιχεία τα αθροίσματα των 2x1 υποπινάκων του αρχικού.

ότι ο αλγόριθμος δεν θα είναι αποδοτικός.

Πρέπει να βρούμε έναν τρόπο να πάρουμε αυτές τις μετρήσεις μέσα στην κάρτα γραφικών και με όσο μεγαλύτερη αποδοτικότητα γίνεται. Ευτυχώς, υπάρχουν μερικοί αλγόριθμοι οι οποίοι είναι κομμένοι και ραμμένοι στα μέτρα των καρτών γραφικών, και αναλαμβάνουν να υπολογίσουν το άθροισμα των στοιχείων ενός πίνακα παράλληλα!

Ένα τέτοιο είδος αλγορίθμων ονομάζεται parallel tree reduction[15] και φαίνεται στο σχήμα 4.3. Σε έναν αλγόριθμο tree reduction, κάθε νήμα αναλαμβάνει να προσθέσει δύο στοιχεία του πίνακα. Επομένως με το πέρας της πρώτης εκτέλεσης θα πάρουμε έναν πίνακα με την μισή αρχική διάσταση που θα έχει στοιχεία τα αθροίσματα των 2x1 υποπινάκων του αρχικού. Είναι επομένως ένας αναδρομικός αλγόριθμος όπως φαίνεται στο σχήμα 4.4.

Ας δούμε τώρα πώς θα εφαρμόσουμε αυτόν τον αλγόριθμο στην κάρτα γραφικών. Το πλέγμα αρχικά θα χωριστεί σε blocks με block_size πλήθος νημάτων το καθένα. Στην πρώτη εκτέλεση, αν πάρουμε το παράδειγμα του σχήματος 4.4, κάθε block θα αντιστοιχεί σε μία ομάδα δέντρων. Κάθε περιττό νήμα ενός block, θα προσθέτει δύο στοιχεία όπως φαίνεται στην πρώτη γραμμή του σχήματος 4.5 και θα τα σώνει στην μνήμη. Το αποτέλεσμα θα είναι ένας πίνακας με την μισή διάσταση και στοιχεία τα προηγούμενα αθροίσματα. Στην συνέχεια το κάθε νήμα θα προσθέτει τα προηγούμενα αποτελέσματα με τον ίδιο τρόπο, έως ότου αθροιστούν όλα τα στοιχεία ενός block σε ένα στοιχείο. Στο σχήμα 4.5 σε κάθε γραμμή βλέπουμε αυτή την διαδικασία. Εδώ ο αλγόριθμος θα σταματήσει και το αποτέλεσμα θα είναι ένας πίνακας με διάσταση το πλήθος των blocks. Θα χωρίσουμε με τον ίδιο τρόπο αυτόν τον πίνακα και θα εκτελούμε την ίδια αναδρομική διαδικασία, έως ότου ο πίνακας είναι μικρότερος του block_size. Τότε, θα εκτελεστεί πάλι με ένα block μόνο, όπως στο δεύτερο τμήμα του σχήματος 4.4 και θα επιστρέψει το συνολικό άθροισμα.

Αυτός ο αλγόριθμος έχει δύο βασικά προβλήματα.

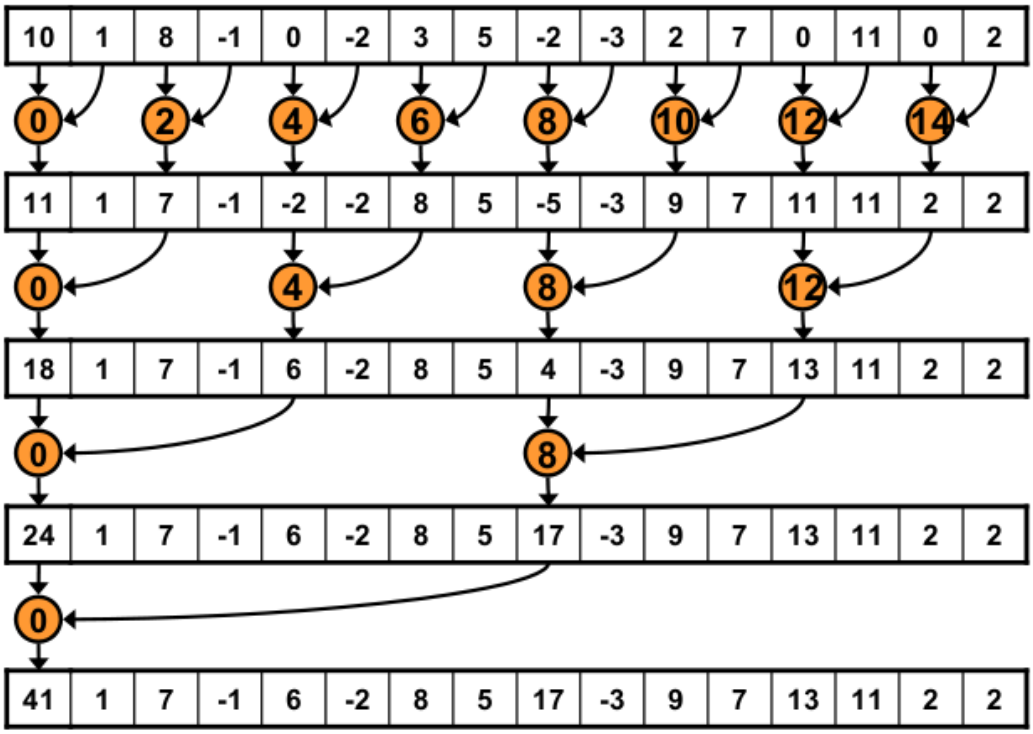
1. Ο αρχικός πίνακας θα πρέπει να διαιρείται πάντα με το 2. Θα πρέπει επομένως η διάσταση του να είναι δύναμη του 2.

2. Αν κοιτάξουμε πιο προσεχτικά το σχήμα 4.5 θα παρατηρήσουμε ότι οι προσβάσεις στην κοινόχρηστη μνήμη θα είναι πάντα uncoa- lesced.

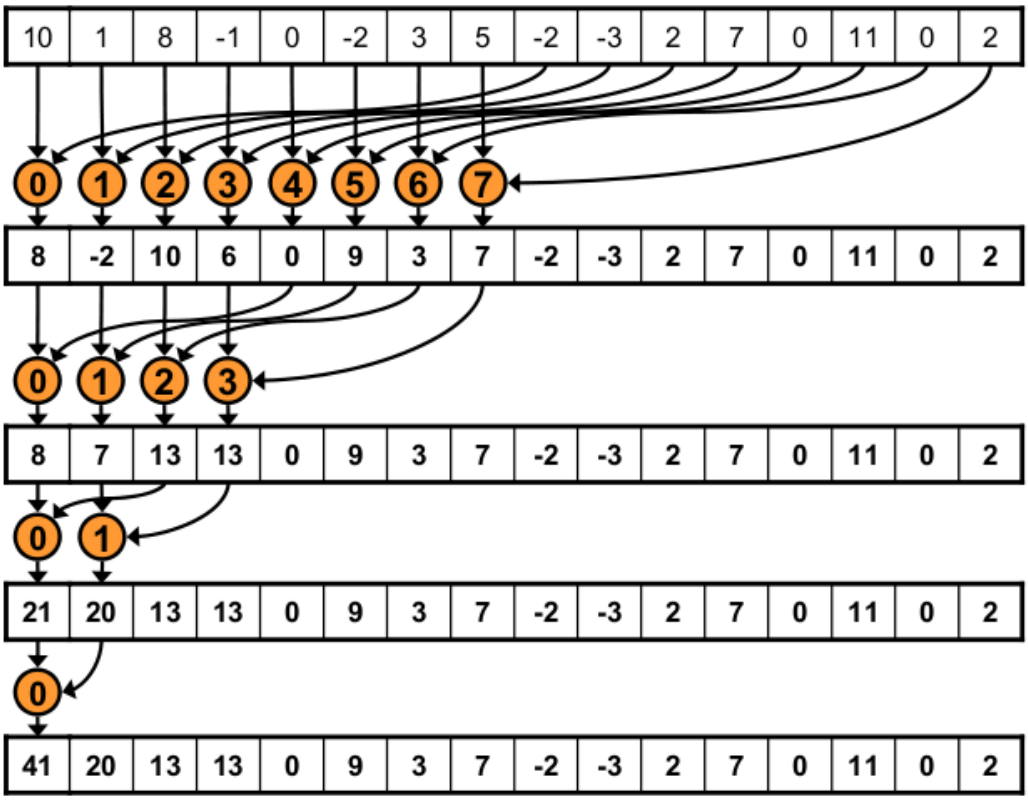
Για το πρώτο πρόβλημα δεν μπορούμε να κάνουμε τίποτα, θα πρέπει να χρησιμοποιήσουμε κάποιον άλλο αλγόριθμο στην θέση του. Στην περίπτωση μας, κάνουμε αυτόν τον έλεγχο στην αρχή της προσομοίωσης και αν η διάσταση δεν είναι δύναμη του 2 θα επιστρέφουμε το πλέγμα κάθε φορά για υπολογισμό στην cpu. Αυτό θα γίνεται μαζί με μία προειδοποίηση ότι η προσομοίωση θα γίνει με ”αργό” υπολογισμό της ενέργειας και της μαγνήτισης.

Για το δεύτερο πρόβλημα η λύση είναι απλή, αντί κάνε νήμα να προσθέτει την επόμενη θέση στον εαυτό του, θα χωρίσουμε το block στα δύο και κάθε νήμα θα προσθέτει τον εαυτό του στην αντίστοιχη θέση του δεύτερου μισού. Αυτός ο τρόπος φαίνεται στο σχήμα 4.6 και είναι περίπου 5 φορές πιο γρήγορος από τον προηγούμενο[15].

Υπάρχουν ακόμα πιο γρήγορες υλοποιήσεις που φτάνουν ως και 30 φορές επιτάχυνση από τον πρώτο που αναφέραμε. Ο αλγόριθμος υποφέρει από το γεγονός ότι τα μισά νήματα είναι ανενεργά. Παρόλα αυτά, αποφασίσαμε να μην κάνουμε το πρόγραμμα πιο πολύπλοκο. Στην συνέχεια, εφόσον το πρόγραμμα δουλεύει σωστά και δίνει την σωστή στατιστική, θα προσπαθήσουμε να τον βελτιστοποιήσουμε περισσότερο.



Σχήμα 4.5: Αναδρομικό tree reduction στην κάρτα γραφικών. Κάθε περιττό νήμα θα προσθέτει το γειτονικό του στοιχείο στον εαυτό του.



Σχήμα 4.6: Αναδρομικό tree reduction στην κάρτα γραφικών χωρίς συγκρούσεις της κοινόχρηστης μνήμης.

Η μέτρηση της ενέργειας και της μαγνήτισης στο πρόγραμμα μας

Ας δούμε τώρα πως εφαρμόζονται όλα τα παραπάνω στην περίπτωση του γενικού αλγορίθμου

- Οι συναρτήσεις μέτρησης της μαγνήτισης:

```
__global__ void cuda_M_Partial_g(int *d_s,int*Mpartial){

  __shared__ int msum[magBLOCK_SIZE];
  uint tid = threadIdx.x;
  uint seqpos=threadIdx.x + blockIdx.x*blockDim.x;
  msum[tid] = d_s[seqpos];
  __syncthreads();
    // do reduction in shared mem
    for(uint s=blockDim.x/2; s>0; s>>=1) {
        if (threadIdx.x < s) {
            msum[tid] += msum[tid + s];
        }
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0){
      Mpartial[blockIdx.x] = msum[0];
    }

}

__global__ void cuda_M_Recursive_g(int *Mpartial){

  __shared__ int msum[magBLOCK_SIZE];
  uint tid = threadIdx.x;
  uint seqpos=threadIdx.x + blockIdx.x*blockDim.x;
  msum[tid] = Mpartial[seqpos];
  __syncthreads();
    // do reduction in shared mem
    for(uint s=blockDim.x/2; s>0; s>>=1) {
        if (threadIdx.x < s) {
            msum[tid] += msum[tid + s];
        }
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0){
      Mpartial[blockIdx.x] = msum[0];
    }
}
```

Βλέπουμε δύο συναρτήσεις. Η πρώτη αναλαμβάνει την πρώτη αναδρομική εκτέλεση, φορτώνοντας τα δεδομένα από το πλέγμα. Η δεύτερη εκτελείται αναδρομικά, χρησιμοποιώντας τα δεδομένα κάθε προηγούμενου αποτελέσματος. Για την μαγνήτιση χρειαζόμαστε απλά το άθροισμα των spin.

- Οι συναρτήσεις μέτρησης της ενέργειας:

```
__global__ void cuda_E_Partial_g(int *d_s,int *Epartial){

  __shared__ int esum[enBLOCK_SIZE];
  int2 nn;
  uint tid=threadIdx.x;
  uint seqpos=threadIdx.x + blockIdx.x*blockDim.x;
  if((nn.x=seqpos + 1) >= const_N)nn.x -= const_N;
  if((nn.y=seqpos + const_L) >= const_N) nn.y -= const_N;
  esum[tid] = (d_s[nn.x]+d_s[nn.y])*d_s[seqpos];
  __syncthreads();
  //do reduction in shared mem
    for(uint s=blockDim.x/2; s>0; s>>=1) {
        if (tid < s) {
            esum[threadIdx.x] += esum[threadIdx.x + s];
        }
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0) {Epartial[blockIdx.x] = esum[0];}
}


__global__ void cuda_E_Recursive_g(int *Epartial){

  __shared__ int esum[enBLOCK_SIZE];
  uint tid=threadIdx.x;
  uint seqpos=threadIdx.x + blockIdx.x * blockDim.x;
  esum[tid] = Epartial[seqpos];
  __syncthreads();
  for(uint s=blockDim.x/2; s>0;s>>=1){
     if(tid < s){
      esum[tid] += esum[tid +s];
     }
     __syncthreads();
  }
  if(tid==0) {
    if(gridDim.x>1)Epartial[blockIdx.x] = esum[0];
    else Epartial[blockIdx.x] = -esum[0];
  }
}
```

Οι δύο συναρτήσεις εργάζονται αντίστοιχα όπως και της μαγνήτισης. Η μόνη διαφορά εδώ είναι ότι θέλουμε την ενέργεια των δεσμών, επομένως η πρώτη συνάρτηση αθροίζει την ενέργεια των δεσμών και η δεύτερη στην τελευταία κλήση της (όπου όλος ο πίνακας είναι ένα block και επομένως gridDim.x == 1) επιστρέφει την αρνητική τιμή του αθροίσματος!

- Οι δηλώσεις των συναρτήσεων:

```
__global__ void cuda_E_Partial_g(int *d_s,int *Epartial);
__global__ void cuda_E_Recursive_g(int *Epartial);
__global__ void cuda_M_Partial_g(int *d_s,int *Mpartial);
__global__ void cuda_M_Recursive_g(int *Mpartial);
```

- Πώς αυτές καλούνται σε κάθε sweep

```
for(int isweep=0;isweep<mp->nsweeps;isweep++){
  cuda_met_black_kernel_g<<<mp->mgrid,mp->mblock>>>
        (mp->md_seeds,mp->d_s,mp->d_prob);
  cuda_met_white_kernel_g<<<mp->mgrid,mp->mblock>>>
        (mp->md_seeds,mp->d_s,mp->d_prob);

  cuda_E_Partial_g<<<mp->emgrid,mp->emblock>>>
        (mp->d_s,mp->d_Epartial);
  cuda_M_Partial_g<<<mp->emgrid,mp->emblock>>>
        (mp->d_s,mp->d_Mpartial);

  recursivegrid = mp->emgrid.x;
  while((recursivegrid.x)>enBLOCK_SIZE){
    recursivegrid= dim3(recursivegrid.x/enBLOCK_SIZE,1,1);
    cuda_E_Recursive_g<<<recursivegrid,mp->emblock>>>
          (mp->d_Epartial);
    cuda_M_Recursive_g<<<recursivegrid,mp->emblock>>>
          (mp->d_Mpartial);
  }
  cuda_E_Recursive_g<<<1,recursivegrid,1>>>(mp->d_Epartial);
  cuda_M_Recursive_g<<<1,recursivegrid,1>>>(mp->d_Mpartial);

  cudaMemcpy(&(E_sequence[isweep]),mp->d_Epartial,sizeof(int),
    cudaMemcpyDeviceToHost);
  cudaMemcpy(&(M_sequence[isweep]),mp->d_Mpartial,sizeof(int),
    cudaMemcpyDeviceToHost);
}
```

Μετά την ανανέωση όλου του πλέγματος, θα τρέξουν οι πρώτες αναδρομικές αθροίσεις για την κάθε ποσότητα. Στην συνέχεια, εκτελούμε αναδρομικά το αποτέλεσμα τους με νέο μέγεθος grid το προηγούμενο ως προς το μέγεθος του block. Όταν πλέον το αποτέλεσμα χωράει σε ένα block εκτελούμε τον αλγόριθμο άλλη μία φορά και παίρνουμε το συνολικό άθροισμα το οποίο επιστρέφουμε στον επεξεργαστή. Εδώ το σωστό θα ήταν να κρατούσαμε τα αποτελέσματα του κάθε sweep στην κάρτα γραφικών και να τα επιστρέφουμε στο τέλος της προσομοίωσης. Παρότι μεταφέρουμε μόνο δύο ακεραίους, έχουμε πει ότι μία μεταφορά ακεραίων με πλήθος N είναι πιο γρήγορη από N μεταφορές ενός ακεραίου. Αυτό θα γίνει σε μεταγενέστερη έκδοση του προγράμματος.

Οι αλγόριθμοι μέτρησης της ενέργειας και της μαγνήτισης για τις άλλες δύο παραλλαγές του αλγορίθμου Metropolis διαφέρουν ελάχιστα από αυτήν του γενικού αλγορίθμου. Η μόνη διαφορά βρίσκεται στην πρώτη αναδρομική συνάρτηση. Τα δεδομένα δεν φορτώνονται από την καθολική μνήμη, αλλά απευθείας από την υφή που χρησιμοποιήσαμε και στον αλγόριθμο Metropolis. Αυτό όμως δεν τις κάνει απαραίτητα πιο γρήγορες, καθώς όλες οι αναγνώσεις από την καθολική μνήμη είναι ευθυγραμμισμένες και coalesced. Επίσης, οι υφές χρησιμοποιούνται μόνο στην πρώτη αναδρομική εκτέλεση του πρώτου αλγορίθμου!

Listing 4.1: Οι αλγόριθμοι μέτρησης της ενέργειας δεσμών στην έκδοση για τον Metropolis υφής

```
__global__ void cuda_E_Partial_ns(int *Epartial){

  __shared__ int esum[enBLOCK_SIZE];
  int2 nn;
  uint tid=threadIdx.x;
  uint seqpos=threadIdx.x + blockIdx.x*blockDim.x;
  if((nn.x=seqpos + 1) >= const_N)nn.x -= const_N;
  if((nn.y=seqpos + const_L) >= const_N) nn.y -= const_N;
  //Η μόνηδιαφοράβρίσκεταιστηνεπόμενηεντολή
  esum[tid] = (tex1Dfetch(tex1DLattice,nn.x) +
   tex1Dfetch(tex1DLattice,nn.y))*tex1Dfetch(tex1DLattice,seqpos);
  __syncthreads();
  //do reduction in shared mem
    for(uint s=blockDim.x/2; s>0; s>>=1) {
        if (tid < s) {
            esum[threadIdx.x] += esum[threadIdx.x + s];
        }
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0) {Epartial[blockIdx.x] = esum[0];}
}


__global__ void cuda_E_Recursive_ns(int *Epartial){

  __shared__ int esum[enBLOCK_SIZE];
  uint tid=threadIdx.x;
  uint seqpos=threadIdx.x + blockIdx.x * blockDim.x;
  esum[tid] = Epartial[seqpos];
  __syncthreads();
  for(uint s=blockDim.x/2; s>0;s>>=1){
     if(tid < s){
      esum[tid] += esum[tid +s];
     }
     __syncthreads();
  }
  if(tid==0) {
    if(gridDim.x>1)Epartial[blockIdx.x] = esum[0];
    else Epartial[blockIdx.x] = -esum[0];  }
}
```

4.3 Ο αλγόριθμος GPU Wolff

Ο Αλγόριθμος Wolff[20] έχει μία ιδιαιτερότητα η οποία τον κάνει δύσκολο να υλοποιηθεί παράλληλα, ειδικά στις αρχιτεκτονικές SIMD[24]. Η ιδιότητα αυτή είναι ότι το πλέγμα πλέον δεν είναι κανονικό αλλά ούτε τοπικό! Το cluster είναι δυναμικό και μπορεί να βρίσκεται σε οποιαδήποτε περιοχή του πλέγματος, η οποία και να αλλάζει με την πάροδο των υπολογισμών. Επομένως, αν κάποιος θέλει να ανανεώσει όλο το πλέγμα με παράλληλο τρόπο, αρχικά θα έχει να αντιμετωπίσει το πρόβλημα των πολλών αδρανών νημάτων. Σε αντίθεση ο αλγόριθμος Swendsen-Wang[22, 2] είναι σχετικά πιο κατάλληλος[24].

Ο αλγόριθμος Wolff αλλά και ο αλγόριθμος Swendsen-Wang θα έχουν επίσης πρόβλημα σε αρχιτεκτονικές SIMD και SIMT λόγω της αποκλίνουσας ροής. Η SIMT είναι πιο ανεκτική, καθώς ο περιορισμός της αποκλίνουσας ροής ισχύει μόνο για τις δύνες.

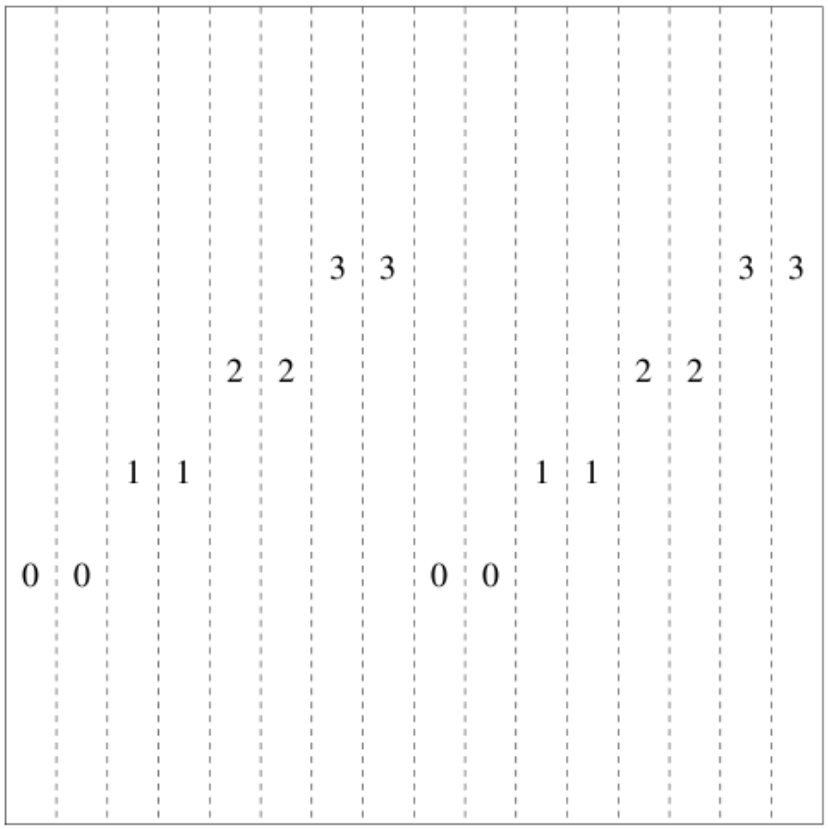
Αν προσπαθήσουμε να υλοποιήσουμε άμεσα στον αλγόριθμο βασιζόμενοι στην cpu έκδοση, θα δούμε ότι είναι και δύσκολο αλλά και ότι θα καταλήξουμε με έναν μη αποδοτικό αλγόριθμο. Ο αλγόριθμος που θα παρουσιάσουμε αποτελεί μία πρώτη προσπάθεια και θα δούμε ότι πάσχει σε πολλούς τομείς, αλλά και ότι έχει περιθώριο για αρκετή βελτίωση οι οποίες θα γίνουν σε επόμενες εκδόσεις.

Πριν φτάσουμε στον αλγόριθμο μας, θα μελετήσουμε μία MIMD υλοποίηση του αλγορίθμου Wolff των Bae et al.[24] οι οποίοι βασίστηκαν στα ευρήματα των Fang et al.[23] πάνω στην πιο αποδοτική κατανομή δεδομένων για έναν αλγόριθμο παράλληλου maze routing. Η μέθοδος της κατανομής των δεδομένων που ακολούθησαν ονομάζεται scattered strip partitioning. Σε αυτή την μέθοδο, τα δεδομένα χωρίζονται σε στήλες ο αριθμός των οποίων είναι κάποιο πολλαπλάσιο του αριθμού των επεξεργαστών. Αυτές εν συνεχεία διανέμονται στους επεξεργαστές. Οι στήλες όμως αυτές δεν διανέμονται με την σειρά, αλλά με περίπου κυκλικό τρόπο όπως στο σχήμα 4.7.

Αυτή η κατανομή προσφέρει τον καλύτερο επιμερισμού φόρτου στους επεξεργαστές. Αφού εφαρμοστεί αυτή η κατανομή, στην συνέχεια επιλέγεται ένα τυχαίο σημείο εκκίνησης. Κάθε επεξεργαστής, έχει το τοπικό του πλέγμα (στήλες). Κάθε επεξεργαστής, έχει μία λίστα στην οποία περιλαμβάνονται οι πλεγματικές θέσεις που ανήκουν στο τοπικό πλέγμα και είναι μέρος της τρέχουσας γενιάς του cluster.

Για κάθε μία πλεγματική θέση, αναπτύσσεται δεσμός εάν ικανοποιείται η πιθανότητα του αλγορίθμου wolff. Εάν ένας επεξεργαστής φτάσει σε ένα τοπικό όριο, στέλνει τον γείτονα που είναι εκτός ορίου στον επεξεργαστή στον οποίο αυτό ανήκει. Στην συνέχεια όλοι επεξεργαστές ελέγχουν τις πλεγματικές θέσεις που έχουν δεχθεί και αν αυτές υπάρχουν ήδη στην λίστα, αν δεν υπάρχουν τότε μπαίνουν στην λίστα της επόμενης γενιάς του cluster και ανανεώνεται το spin του. Τέλος ελέγχεται αν υπάρχουν πλεγματικές θέσεις στην νέα γενιά. Αν όχι, το sweep τερματίζεται. Στην MIMD έκδοση είχαν επιτάχυνση 10x ενώ στην SIMD είχαν χειρότερες επιδόσεις από την σειριακή εκτέλεση.

Ας δούμε τώρα τον αλγόριθμο που αναπτύξαμε. Σε αυτόν τον αλγόριθμο, μελετώνται μόνο τοπικά πλέγματα των 16x16. Παίρνουμε στατιστική εκτελώντας πολλά μικρά πλέγματα παράλληλα (εκατοντάδες χιλιάδες). Κάθε πλέγμα αντιστοιχεί σε ένα block. Το πρώτο νήμα κάθε block επιλέγει μία τυχαία πλεγματική θέση εκκίνησης. Η πλεγματική θέση προστίθεται στην λίστα και στην συνέχεια όταν ένα νήμα βρίσκει τον εαυτό του μέσα στην λίστα, ελέγχει τους γείτονες του και τους βάζει



Σχήμα 4.7: Κυκλική κατανομή των δεδομένων για δεδομένα χωρισμένα σε 16 στήλες, 4 επεξεργαστές και εύρος στήλης 2

στην λίστα εάν τηρούν τις προϋποθέσεις του αλγορίθμου wolff. Αυτή η διαδικασία συνεχίζεται έως ότου η λίστα είναι κενή.

Ήδη βλέπουμε τα πρώτα προβλήματα με αυτόν τον αλγόριθμο. Αρχικά δεν μπορεί να μελετήσει μεγάλα πλέγματα. Στην συνέχεια, για να ενημερωθεί η λίστα πρέπει να συγχρονιστούν τα νήματα για να μην προσπαθήσουν να γράψουν όλα στην ίδια θέση. Τέλος, λίγα νήματα μπορούν να τρέξουν παράλληλα ανά block.

Δεν θα ασχοληθούμε πολύ με αυτόν τον αλγόριθμο, αφού κρίθηκε ως μη αποδοτικός και αργός.

4.4 Προσομοιώσεις,ανάλυση και επεξεργασία των δεδομένων

Έχοντας αναλύσει πλήρως τους αλγορίθμους, τον προγραμματισμό σε κάρτα γραφικών, αλλά και αφού δείξαμε ότι ένας αλγόριθμος σε GPU δεν θα είναι απαραίτητα πιο γρήγορος από τον αντίστοιχο σειριακό, σε αυτήν την ενότητα θα ασχοληθούμε με τις προσημειώσεις. Αφού αναλύσουμε πως έγινε η διεξαγωγή των προσομοιώσεων και οι αναλύσεις των δεδομένων θα προχωρήσουμε στην άμεση σύγκριση των αποτελεσμάτων με τον αλγόριθμο αναφοράς των Newman and Barkema[2] που χρησιμοποιείται στο μάθημα της Υπολογιστικής Φυσικής ΙΙ: Προτυποποίηση [1]. Τέλος, αφού αναλύσουμε τους αλγορίθμους ως προς την σωστή λειτουργία τους και εξάγουμε σωστές στατιστικές ποσότητες, θα προχωρήσουμε και στην παρουσίαση των επιδόσεων. Εξάλλου ο στόχος της χρήσης των καρτών γραφικών, παρά τις ιδιαιτερότητες τους και τους αυστηρούς περιορισμούς που επιβάλλουν, είναι να επιτύχουμε σημαντική επιτάχυνση σε έναν αλγόριθμο.

4.4.1 Τρόπος διεξαγωγής, αυτοματοποιημένες προσομοιώσεις και αναλύσεις

Για την διεξαγωγή των προσομοιώσεων χρησιμοποιήθηκαν tcsh[1], bash[2] και awk[3] scripts από το μάθημα της Υπολογιστικής Φυσικής ΙΙ, αλλά και τροποποιήσεις τους. Οι προσομοιώσεις εκτελέστηκαν σε μεγάλο βαθμό αυτοματοποιημένα, όπως και η ανάλυση των δεδομένων. Για την δημιουργία διαγραμμάτων χρησιμοποιήθηκε η εφαρμογή ανοικτού λογισμικού gnuplot[4].

Για τις προσομοιώσεις χρησιμοποιήθηκε ένας υπολογιστής GPU Tesla με κάρτες γραφικών Tesla C1060 υπολογιστικής ικανότητας 1.3. Επίσης εκτελέστηκαν προσομοιώσεις και σε μία GeForce 9300M GS φορητού υπολογιστή με υπολογιστική ικανότητα 1.1. Στον πρώτο υπολογιστή, ο επεξεργαστής είναι Intel Xeon E5540[5] με χρονισμό 2.53GHz, ενώ στον δεύτερο Intel Core 2 Duo P8400 με χρονισμό 2.23Ghz.

Αυτοματοποιημένη εκτέλεση

Ας δούμε πρώτα το script που χρησιμοποιήθηκε για την εκτέλεση:

[1] http://www.tcsh.org/Welcome
[2] http://www.gnu.org/software/bash/
[3] http://www.vectorsite.net/tsawk.html
[4] http://www.gnuplot.info/
[5] Αρχιτεκτονική Nehalem.

```
set blocksx = (4 8 16 32)
set blocksy = (2 4)
set betas  = (0.100 0.150 0.200 0.250 0.300 0.340 \
              0.380 0.400 0.410 0.420 0.430 0.440 \
              0.450 0.460 0.470 0.480 0.490 0.500 \
              0.520 0.560 0.600 0.650 0.700 0.750 \
              0.800 0.850 0.900)
set seed   = 45000
set sweeps = 1000000
# ###################################################################

foreach blx ($blocksx)
 foreach bly ($blocksy)
  foreach L ($Ls)
   set start = "-S $seed -m2"
   set dir   = L$L
   foreach beta ($betas)
    set fname = L${L}b${beta}n${sweeps}
    set exe  = "./Ising_gpu${blx}x${bly}"
    if(-e $exe) then
     @ lim = $blx * 2
     if($L >= $lim) then
      if( ( $blx == 4 ) && ( $L > 8 ) ) continue
      if( ( $blx == 8 ) && ( $L > 16 ) ) continue
      if( ( $blx == 16 ) && ( $L > 32 ) ) continue
      if( ! -d $dir ) mkdir $dir
      echo "GPU Simulating for L= $L beta= $beta and nsweeps= $sweeps"
      ($exe -L $L -b $beta $start -n $sweeps>${fname}_out)>& ${fname}_err
      foreach e ( met met_ns met_g err out)
       gzip -f ${fname}_$e
       mv   -f ${fname}_$e.gz $dir
      end
     endif
    endif
    set start = "-s 0 -m2"
   end
  end
 end
end
```

Θέτουμε συνθήκες για το μέγεθος του block καθώς το μήκος του πλέγματος πρέπει πάντα να είναι διπλάσιο από το μέγεθος του block στην διάσταση x. Αυτό συμβαίναι διότι κάθε νήμα αναλαμβάνει ένα 2x2 υποπλέγμα, επομένως το ελάχιστο μέγεθος του πλέγματος πρέπει να είναι $2 * blocksize_x$. Το εκτελέσιμο μας ονομάζεται Ising_gpu και δέχεται αρκετά ορίσματα. Εδώ θα χρησιμοποιήσουμε μόνο τα ορίσματα -s, -S, -L, -n και -m.

- -s Ορίζει την αρχική κατάσταση του πλέγματος. Δέχεται τις τιμές 0,1,2. Για τιμή 0 το πλέγμα θα έχει ψυχρό configuration, για τιμή 1 ζεστό και για τιμή 2 θα χρησιμοποιεί το configuration της προηγούμενης εκτέλεσης. Η προεπιλεγμένη τιμή είναι 0.

- -S Ορίζει το seed της γεννήτριας ψευδοτυχαίων αριθμών. Η προεπιλεγμένη τιμή είναι 45000

- -L Ορίζει το μέγεθος του πλέγματος. Η ελάχιστη τιμή για την απλή έκδοση

είναι 32 και δέχεται τιμές μόνο πολλαπλάσια του 32. Η ελάχιστη τιμή αντικατοπτρίζει το μέγεθος του προεπιλεγμένου block (16x4 δηλαδή 32x8 πλεγματικές θέσεις). Έχουμε όμως και εκδόσεις με μικρότερο μέγεθος block ώστε να μπορέσουμε να μελετήσουμε μικρότερα πλέγματα, όμως σε αυτές τις εκδόσεις δεν θα πρέπει να περιμένουμε πολύ μεγάλη επιτάχυνση.

- -n Ορίζει το πλήθος των sweeps. Η προεπιλεγμένη τιμή είναι 1.
- -m Ορίζει στο πρόγραμμα τον αλγόριθμο που θα χρησιμοποιήσει. Παίρνει τιμές από 1 έως και 9. Θα δούμε αναλυτικά όλες τις επιλογές στην τεκμηρίωση του προγράμματος. Εδώ αρκεί να αναφέρουμε ότι η τιμή 2 λέει στο πρόγραμμα να εκτελέσει τον αλγόριθμο υφής, και η τιμή 7 όλες τις παραλλαγές του αλγορίθμου

Με την ολοκλήρωση κάθε προσομοίωσης, το πρόγραμμα θα μας επιστρέφει και τον συνολικό χρόνο εκτέλεσης στην κάρτα γραφικών.

Η ανάλυση των αποτελεσμάτων έγινε με την χρήση των παρακάτω εργαλείων:

- histogram. Εφαρμογή του μαθήματος για δημιουργία ιστογραμμάτων από τα time histories.
- jack. Εφαρμογή του Δρ. Κ. Αναγνωστόπουλου για την μέθοδο των σφαλμάτων με την χρήση της μεθόδου jackknife, αλλά και την μελέτη του αυτοσυσχετισμού.
- merge. Script του Δρ. Κ. Αναγνωστόπουλου, γραμμένο σε awk, για την ένωση δύο αρχείων ανά στήλη.
- analyze. Script του μαθήματος, ελαφρά τροποποιημένο, για την εξαγωγή των ποσοτήτων και των σφαλμάτων τους.

```
#!/bin/tcsh -f

# ######################################################################
# Set Parameters for the analysis:
set Ls      = (8 16 32 64 128 256 512 1024 2048)
# ######################################################################

set cdir = $cwd #we have this to point to the original place we started from
foreach L ($Ls)
 set dir = $cdir/L$L
 cd $dir
 set files = (L*met_ns.gz)
 set NL    = `awk -v L=$L 'BEGIN{print 2*L*L}'` # number of bonds on lattice
 set N     = `awk -v L=$L 'BEGIN{print   L*L}'` # number of sites on lattice
 foreach f ($files)
  #get beta from filename: Regular expression means:
  #match a "b" in filename, 0 or more of digits, a dot (.) and then again
  #0 or more digits
  set beta = `echo $f |   perl -ne '/b(\d*\.\d*)/;print $1;'`
```

```
  set fname = $f:r     # remove extensions to get filename
  #energy histogram
  echo "###### Begin L=$L beta=$beta ######"
  zgrep  -v "#" $f | awk -v NL=$NL '{print $1/NL}'  | \
            ../histogram -v f=$NL > $fname.hse
  #no absolute value here, we want to check symmetry
  zgrep  -v "#" $f | awk -v N=$N   '{print $2/N}' | \
            ../histogram -v f=$N  > $fname.hsm

  echo -n "L,b $L $beta"
  zgrep  -v "#" $f | awk -v NL=$NL -v ig=$1 'NR>ig{print $1/NL}'  | \
  ../jack -d 1000000| grep -v "#"| \
  awk '{printf "\ne  %s %s    \n", $1, $2}'

  #we compute <|M|>
  zgrep  -v "#" $f | \
  awk -v N=$N -v ig=$1 'NR>ig{if($2<0){$2=-$2};print $2/N}' | \
  ../jack -d 1000000| grep -v "#"| \
  awk '{printf "am  %s %s\n", $1, $2}'

  #we compute <M>
  zgrep  -v "#" $f | \
  awk -v N=$N -v ig=$1 'NR>ig{print $2/N}' | \
  ../jack -d 1000000| grep -v "#"| \
  awk '{printf "m   %s %s \n", $1, $2}'

  #we compute Cv=(b^2/N)(<E^2>-<E>^2)
  zgrep  -v "#" $f | awk -v NL=$NL -v ig=$1 'NR>ig{print $1/NL}' | \
  ../jack -d 1000000| grep -v "#" | awk -v NL=$NL -v b=$beta \
  '{printf "Cv  %s %s    \n", $3*b*b*NL,$4*b*b*NL}'

  #we compute Xv
  zgrep -v "#" $f | \
  awk -v N=$N -v ig=$1 'NR>ig{if($2<0){$2=-$2};print $2/N}' | \
  ../jack -d 1000000|grep -v "#" |awk -v L=$L -v N=$N -v b=$beta \
  '{printf "Xv-L%d %d %f %s %s    \n",L,N,b, $3*b*N,$4*b*N}'
 end
end
```

- differences. Script με το οποίο παίρνουμε σε ένα αρχείο τον λόγο της διαφοράς των αποτελεσμάτων μεταξύ cpu και gpu μαζί με το σφάλμα.
- anfinal. Script για την μαζική περισυλλογή των αναλύσεων και αρχειοθέτηση αναλύσεων, σε φόρμα συμβατή με το gnuplot.

Οι προσομοιώσεις

Αρχικά εκτελέστηκε το script αυτόματων προσομοιώσεων για την gpu. Χρησιμοποιούμε τα εκτελέσιμα με μικρά μεγέθη block μόνο για μικρά πλέγματα. Για την περίπτωση των προσομοιώσεων στην cpu χρησιμοποιήσαμε ένα αντίστοιχο script με την διαφορά ότι αφαιρέθηκαν οι μεταβλητές των μεγεθών του block καθώς δεν χρειάζονται.

Ενώ είναι δυνατό να εκτελέσουμε τις προσομοιώσεις για τον ενσωματωμένο αλγόριθμο αναφοράς του προγράμματος, θεωρήθηκε ορθότερο να χρησιμοποι-

ηθεί απευθείας το εκτελέσιμο του μαθήματος. Ο βασικός λόγος είναι ότι το εκτελέσιμο για GPU παρόλο που έχει δυνατότητα να εκτελέσει μόνο τον αλγόριθμο CPU, είναι μεταγλωττισμένο με υποστήριξη CUDA, επομένως δεν μπορεί να εκτελεστεί σε υπολογιστή χωρίς κάρτα γραφικών με υποστήριξη CUDA. Είναι ορθότερο επομένως να χρησιμοποιηθεί το εκτελέσιμο του μαθήματος σε έναν ελεύθερο υπολογιστή, και ταυτόχρονα να γίνει η προσομοίωση του αλγορίθμου GPU στον υπολογιστή με την αντίστοιχη κάρτα γραφικών.

Εφόσον οι προσομοιώσεις ολοκληρωθούν συνεχίζουμε με την εκτέλεση των script ανάλυσης, τα οποία θα μας δώσουν σε λίγα αρχεία όλες τις πληροφορίες που χρειαζόμαστε για τις φυσικές ποσότητες που θέλουμε να μελετήσουμε. Δηλαδή, την ενέργεια ανά δεσμό, την απόλυτη μαγνήτιση ανά πλεγματική θέση, την ειδική θερμότητα και την μαγνητική επιδεκτικότητα.

4.4.2 Ανάλυση των δεδομένων, συγκρίσεις με τον αλγόριθμο αναφοράς

Πριν κάποιος εκτελέσει την ανάλυση των φυσικών ποσοτήτων, πρέπει πρώτα να σιγουρευτεί ότι χρησιμοποιεί μετρήσεις παρμένες από ένα σύστημα σε θερμική ισορροπία. Για να το κάνουμε αυτό, κάνουμε τα γραφήματα των time histories για κάθε θερμοκρασία των δύο αλγορίθμων. Κοιτάμε για πιο sweep και μετά στα δύο γραφήματα μία ποσότητα (π.χ. ενέργεια) διακυμαίνεται γύρω από την ίδια τιμή. Έχοντας βρει το sweep για το οποίο μπορούμε να θεωρήσουμε ότι έχουμε θερμική ισορροπία, αγνοούμε όλα τα προηγούμενα sweeps και συνεχίζουμε στην ανάλυση.

Μας ενδιαφέρει να συγκρίνουμε άμεσα τα αποτελέσματα με τον αλγόριθμο αναφοράς για να μελετήσουμε την σωστή λειτουργία του. Θα μπορούσαμε αντίστοιχα να επιχειρήσουμε να υπολογίσουμε μέσω *finite size scaling* την τιμή της κρίσιμης θερμοκρασίας και να δούμε αν αυτή συμφωνεί με την αναλυτική λύση του *Onsager*. Ένας άλλος τρόπος θα ήταν ο υπολογίσμος του *Binder cumulant* ο οποίος ορίζεται ως:

$$U_4(T) = 1 - \frac{\langle M(T)^4 \rangle}{3 \langle M(T)^2 \rangle^2} \tag{4.1}$$

και στην κρίσιμη θερμοκρασία γίνεται ανεξάρτητος του μήκους του πλέγματος. Επομένως, αρκεί να βρούμε για ποια θερμοκρασία συμβαίνει αυτό.

Υπολογισμός φυσικών ποσοτήτων

Με γνωστό το σημείο για το οποίο έχουμε θερμική ισορροπία, θέτουμε το σημείο ως όρισμα στο script ανάλυσης, αγνοώντας έτσι όλες τις προηγούμενες μετρήσεις.

- Για την ενέργεια ανά δεσμό, παίρνουμε την τιμή της ενέργειας για κάθε sweep του συστήματος σε θερμική ισορροπία και την διαιρούμε με το πλήθος των δεσμών $2 * N = 2 * L * L$. Τέλος, με τη χρήση του προγράμματος jack παίρνουμε την μέση τιμή και το σφάλμα.

- Για την μαγνήτιση ανά πλεγματική θέση, παίρνουμε την τιμή της μαγνήτισης για κάθε sweep του συστήματος σε θερμική ισορροπία και την διαιρούμε με το πλήθος των spin $N = L * L$. Αντίστοιχα με τη χρήση του προγράμματος jack παίρνουμε την μέση τιμή και το σφάλμα.

- Για την απόλυτη μαγνήτιση ανά πλεγματική θέση, παίρνουμε την τιμή της μαγνήτισης όπως πριν, αλλά τώρα αν είναι αρνητική της αλλάζουμε πρόσημο και αντίστοιχα την διαιρούμε με το πλήθος των spin $N = L * L$ και παίρνουμε την μέση τιμή και το σφάλμα με τον ίδιο τρόπο.

- Για την ειδική θερμότητα, βρίσκουμε την τιμή της ενέργειας ανά δεσμό όπως πριν. Για να υπολογίσουμε την διακύμανση χρησιμοποιούμε πάλι το πρόγραμμα jack και επιστρέφουμε το αποτέλεσμα με τον συντελεστή $3 * b^2 * 2 * L * L$ αφού $C = 3 * b^2/N * (\Delta E)^2$ όπου εδώ $N = 2 * L * L$ και το οποίο περιλαμβάνεται ήδη στον υπολογισμό της ενέργειας ανά δεσμό, επομένως λόγω του τετραγώνου στην διακύμανση έχει τετραγωνιστεί και πρέπει να πολλαπλασιάσουμε με αυτό.

- Για την μαγνητική επιδεκτικότητα, βρίσκουμε την απόλυτη μαγνήτιση όπως πριν. Αντίστοιχα βρίσκουμε την διακύμανση μέσω του προγράμματος jack και τώρα ο συντελεστής είναι $3 * b * L * L$, αφού εδώ $N = L * L$.

Συγκρίσεις των αποτελεσμάτων και γραφήματα

Εφόσον έχουμε υπολογίσει όλες τις φυσικές ποσότητες που μας ενδιαφέρουν, μπορούμε να τις συγκρίνουμε με τα αποτελέσματα του αλγορίθμους αναφορά. Αυτό αρχικά γίνεται εποπτικά με κοινά γραφήματα και στην συνέχεια παίρνοντας τον συντελεστή διαφοράς:

$$R = 2 * \frac{O_{cpu} - O_{gpu}}{O_{cpu} + O_{gpu}} \tag{4.2}$$

όπου O μία ποσότητα και cpu/gpu ο αλγόριθμος αντίστοιχα. Το σφάλμα αυτού του συντελεστή βρίσκεται από το θεώρημα διάδοσης των σφαλμάτων (error propagation theorem):

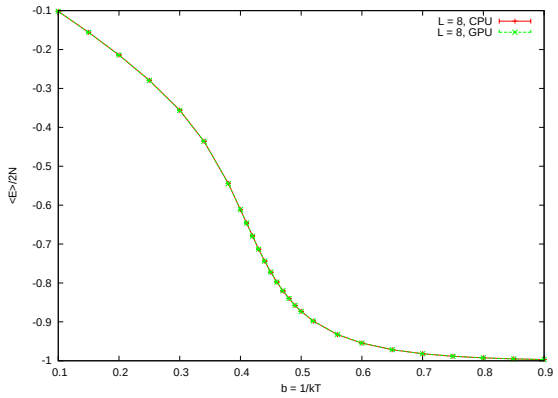
$$\begin{aligned}(\delta R)^2 &= \left(\frac{\partial R}{\partial O_{cpu}} * \delta O_{cpu}\right)^2 + \left(\frac{\partial R}{\partial O_{gpu}} * \delta O_{gpu}\right)^2 = \\ &= 16 * \left(\frac{O_{gpu}^2}{(O_{cpu} + O_{gpu})^4} * \delta O_{cpu}^2 + \frac{O_{cpu}^2}{(O_{cpu} + O_{gpu})^4} * \delta O_{gpu}^2\right)\end{aligned} \tag{4.3}$$

Σχεδιάζουμε τα διαγράμματα κάθε ποσότητας ως προς το β για κάθε πλέγμα και στην συνέχεια τα αντίστοιχα διαγράμματα του λογαρίθμου των διαφορών με το β. Διαφορές εντός του σφάλματος είναι αναμενόμενες καθώς οι αρχικές συνθήκες είναι διαφορετικές. Επίσης, για την μέση μαγνήτιση είναι πιθανό να έχουμε αντίστροφα αποτελέσματα και μεγάλες διαφορές από την κρίσιμη περιοχή και μετά, αυτό συμβαίνει επειδή μετά την κρίσιμη θερμοκρασία, το σύστημα παίζει ανάμεσα σε καταστάσεις με όλα τα spin πάνω ή όλα κάτω. Επειδή τα βήματα Metropolis είναι "αργά" είναι δύσκολο να δούμε τόσες αντιστροφές ώστε η μέση τιμή να είναι μηδέν (όπως συμβαίνει στον αλγόριθμο Wolff πχ). Επίσης το αν θα επιλέξει θέσεις με τα περισσότερα spin πάνω η κάτω, εξαρτάται από τις αρχικές συνθήκες, που εξ'ορισμού είναι διαφορετικές (ακόμα και για τον ίδιο γεννήτορα της γεννήτριας τυχαίων αριθμών, η κατανομή των τυχαίων αριθμών γίνεται διαφορετικά στους αλγορίθμους gpu).

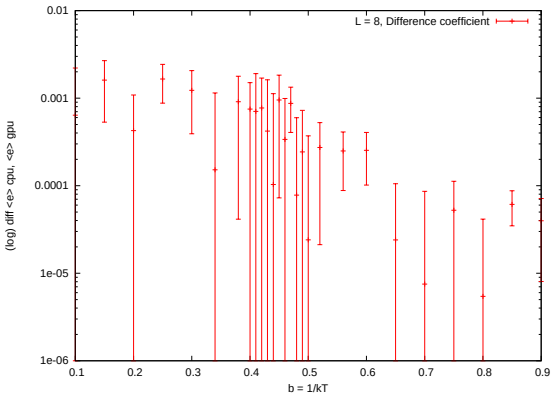
Για την σύγκριση μας ενδιαφέρουν τα μικρά πλέγματα ώστε να αποφύγουμε προβλήματα οφειλόμενα σε έντονα εξαρτημένες μετρήσεις. Στα σχήματα 4.8 έως 4.11 βλέπουμε τα γραφήματα για τις φυσικές ποσότητες που μας ενδιαφέρουν για τον αλγόριθμο αναφοράς και τον αλγόριθμο υφής. Δίπλα σε κάθε ένα βρίσκουμε το γράφημα της διαφοράς με τον κάθετο άξονα σε λογαριθμική κλίμακα. Εφόσον βλέπουμε ότι τα αποτελέσματα μας, εντός του στατιστικού σφάλματος, συμφωνούν κάνουμε τα διαγράμματα 4.12 στα οποία φαίνεται πως συμπεριφέρονται οι ποσότητες καθώς πλησιάζουμε στο θερμοδυναμικό όριο. Τυπική συμπεριφορά στο θερμοδυναμικό όριο για την κρίσιμη θερμοκρασία είναι, αλλαγή κλίσης για την ενέργεια, ασυνέχεια για την μαγνήτιση με μηδενική μαγνήτιση για β_c^- και μη μηδενική για β_c^+ και τέλος για την ειδική θερμότητα και την μαγνητική επιδεκτικότητα με β_c^- και β_c^+ να τήνουν στο άπειρο.

4.4.3 Επιδόσεις των νέων αλγορίθμων, η σχέση τους με την αρχιτεκτονική των καρτών

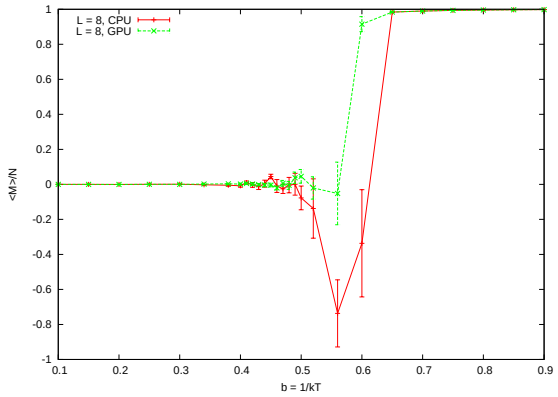
Ο κύριος λόγος για προγραμματισμό σε GPU είναι η επιτάχυνση ενός αλγορίθμους. Επομένως, μετά την μελέτη ενός αλγορίθμου ως προς την καλή λειτουργία του, σειρά έχει η σύγκριση των επιδόσεων. Για να μετρήσουμε τους χρόνους εκτέλεσης, εκτελούμε προσομοιώσεις σε ένα μεγάλο εύρος πλεγμάτων για τρεις θερμοκρασίες (0.100, 0.440 και 0.900) για όλους τους αλγορίθμους gpu και τον ενσωματωμένο αλγόριθμο cpu. Σε κάθε εκτέλεση, το πρόγραμμα επιστρέφει σε ένα αρχείο τους χρόνους εκτέλεσης κάθε αλγορίθμου. Χρησιμοποιήθηκε μία τροποποιημένη έκδοση του script εκτέλεσης με την μόνη διαφορά ότι πλέον επιτρέπει σε μικρά μεγέθη block να τρέξουν μεγάλα πλέγματα. Οι αλγόριθμοι gpu εκτελέστηκαν ξεχωριστά από τον αλγόριθμο cpu, ώστε να επιταχυνθεί η διεξαγωγή των δοκιμών καθώς ο αλγόριθμος cpu θα τρέχει ξεχωριστά σε πανομοιότυπο επεξεργαστή.



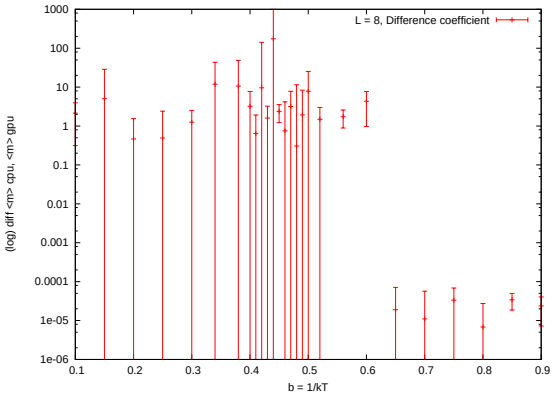
Η μέση ενέργεια ανά δεσμό $\langle E \rangle$



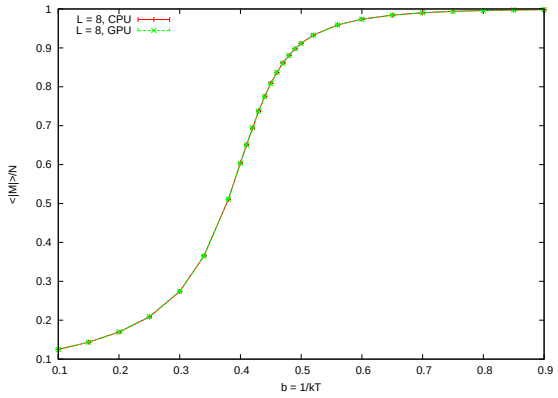
Συντελεστής διαφορών $\langle E \rangle$



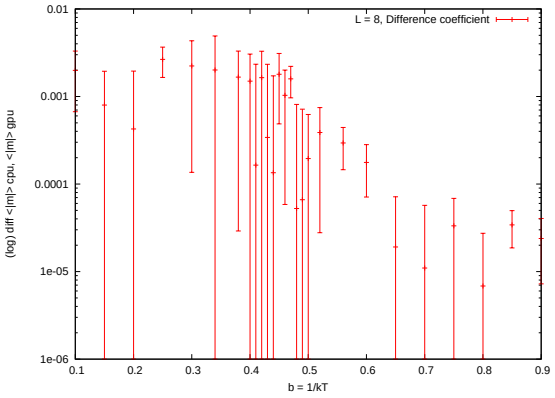
Η μέση μαγνήτιση ανά πλεγματική θέση $\langle M \rangle$



Συντελεστής διαφορών $\langle M \rangle$

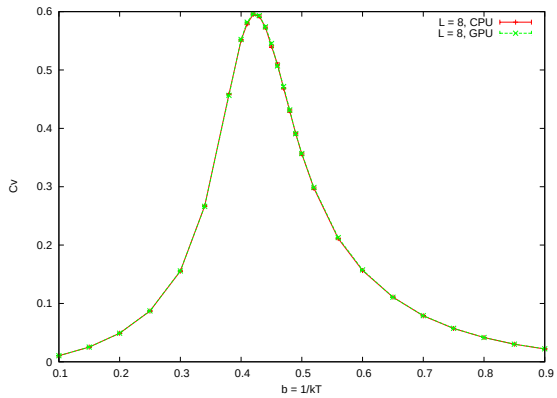


Η μέση απόλυτη μαγνήτιση ανά πλεγματική θέση $\langle |M| \rangle$

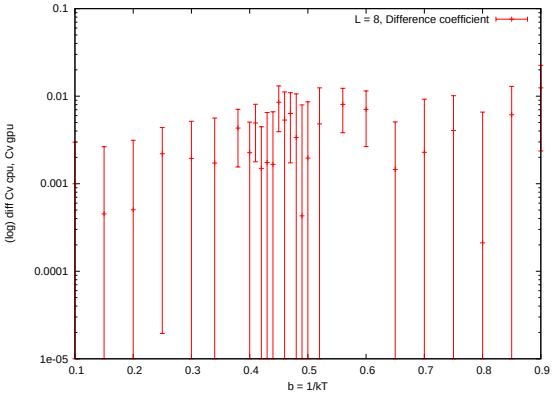


Συντελεστής διαφορών $\langle |M| \rangle$

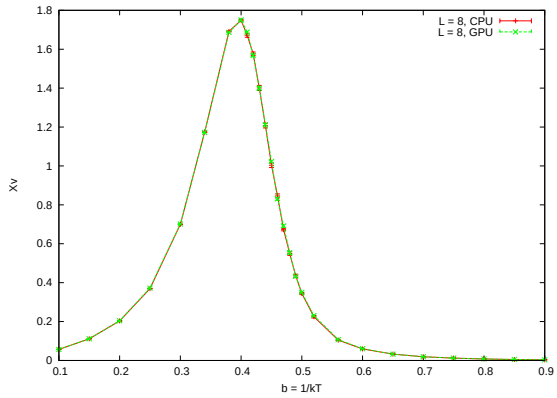
Σχήμα 4.8: Σύγκριση αποτελεσμάτων των αλγορίθμων gpu και cpu για L 8.



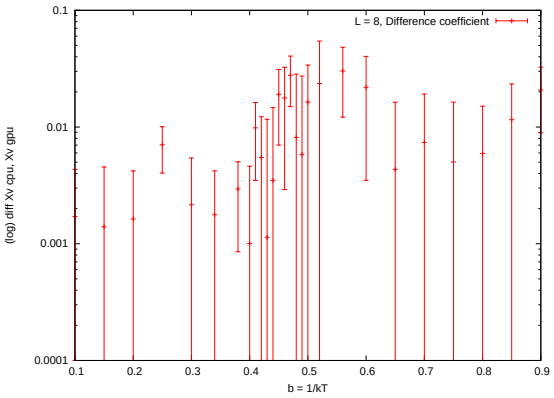
Η ειδική θερμότητα C



Συντελεστής διαφορών C

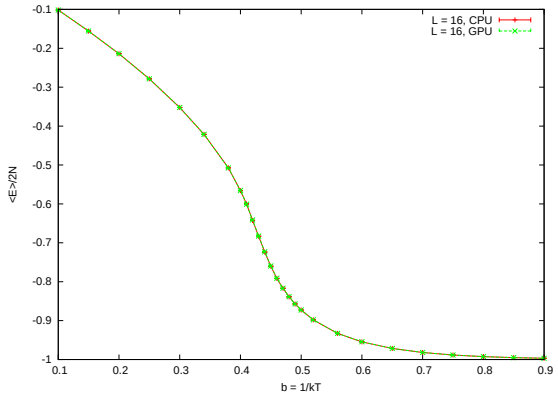


Η μαγνητική επιδεκτικότητα χ

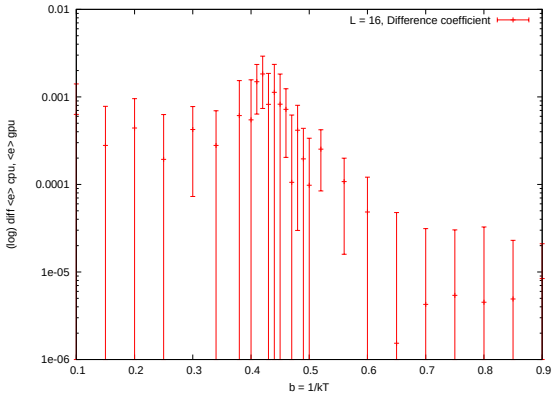


Συντελεστής διαφορών χ

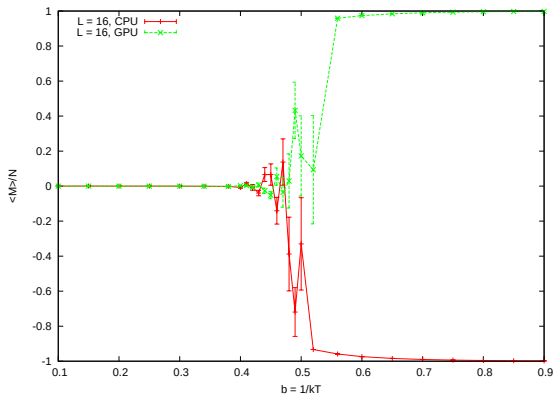
Σχήμα 4.8: Σύγκριση αποτελεσμάτων των αλγορίθμων gpu και cpu για L 8 (συνέχεια).



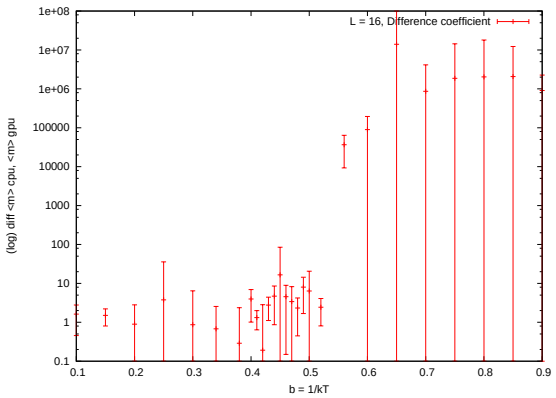
Η μέση ενέργεια ανά δεσμό $\langle E \rangle$



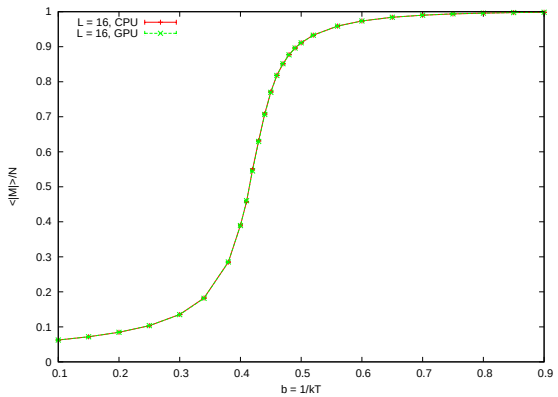
Συντελεστής διαφορών για την ενέργεια $\langle E \rangle$



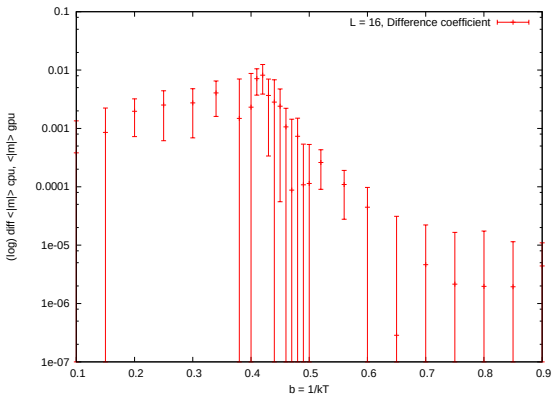
Η μέση μαγνήτιση ανά πλεγματική θέση $\langle M \rangle$



Συντελεστής διαφορών για την μαγνήτιση $\langle M \rangle$

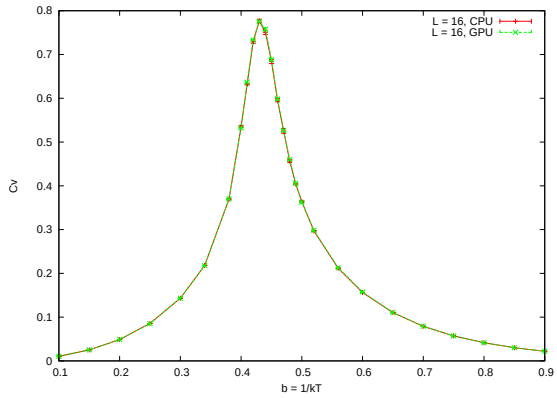


Η μέση απόλυτη μαγνήτιση ανά πλεγματική θέση $\langle |M| \rangle$

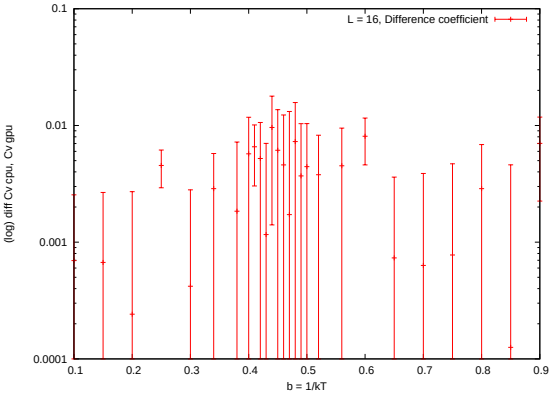


Συντελεστής διαφορών για την απόλυτη μαγνήτιση $\langle |M| \rangle$

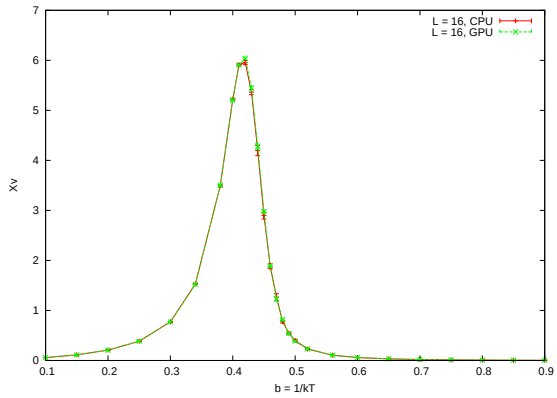
Σχήμα 4.9: Σύγκριση αποτελεσμάτων των αλγορίθμων gpu και cpu για L 16.



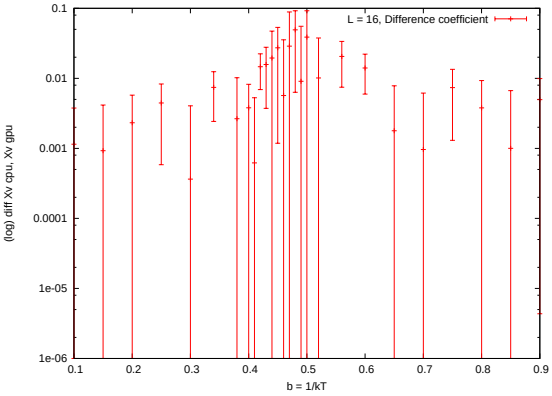
Η ειδική θερμότητα C



Συντελεστής διαφορών για την ειδική θερμότητα C

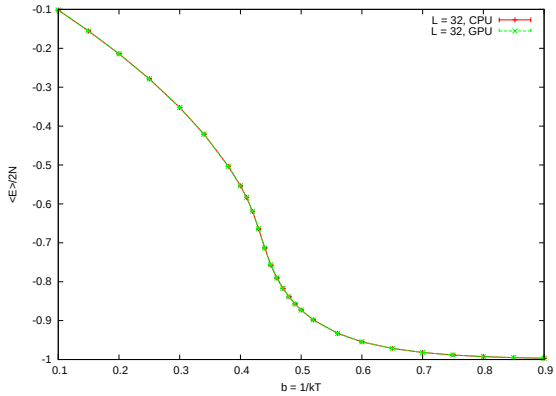


Η μαγνητική επιδεκτικότητα χ

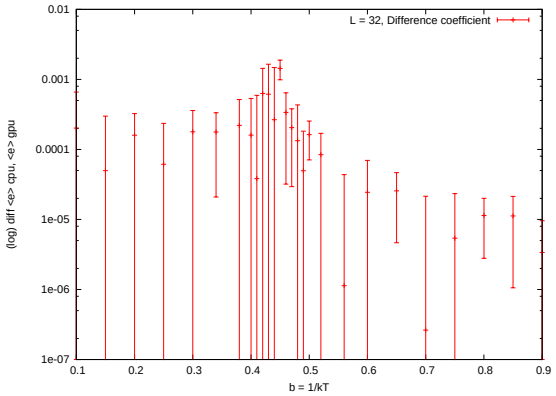


Συντελεστής διαφορών για την μαγνητική επιδεκτικότητα χ

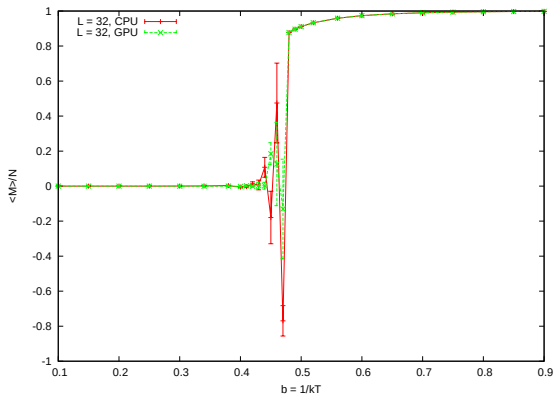
Σχήμα 4.9: Σύγκριση αποτελεσμάτων των αλγορίθμων gpu και cpu για L 16 (συνέχεια).



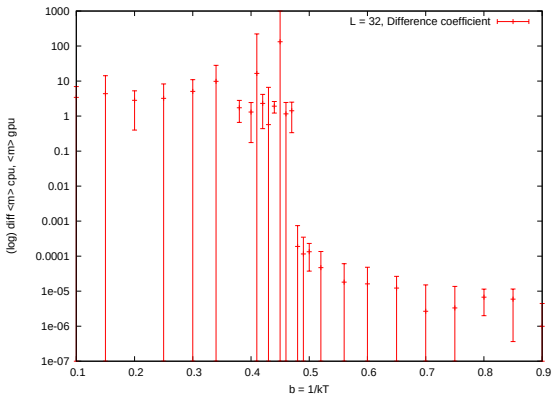
Η μέση ενέργεια ανά δεσμό $\langle E \rangle$



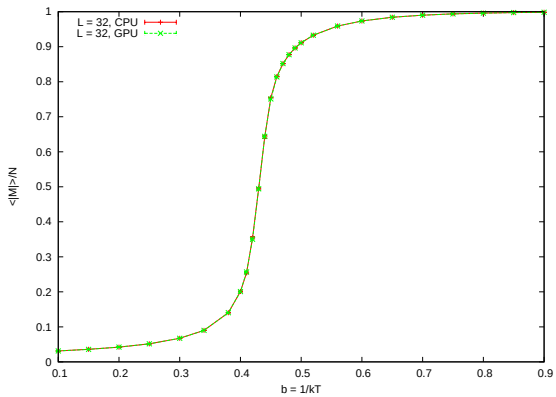
Συντελεστής διαφορών για την ενέργεια $\langle E \rangle$



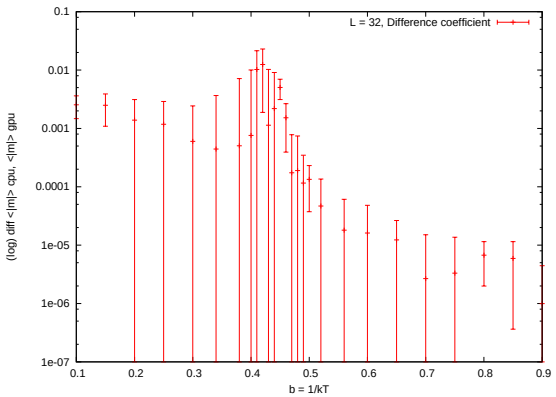
Η μέση μαγνήτιση ανά πλεγματική θέση $\langle M \rangle$



Συντελεστής διαφορών για την μαγνήτιση $\langle M \rangle$

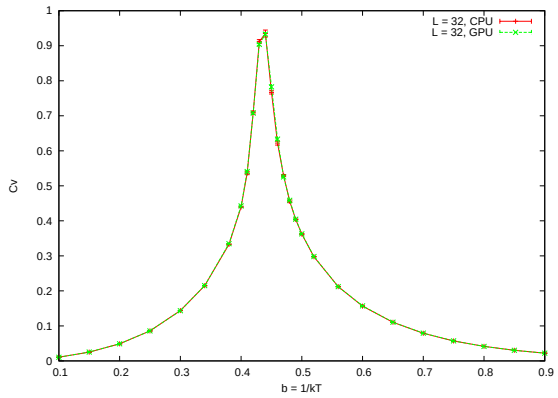


Η μέση απόλυτη μαγνήτιση ανά πλεγματική θέση $\langle |M| \rangle$

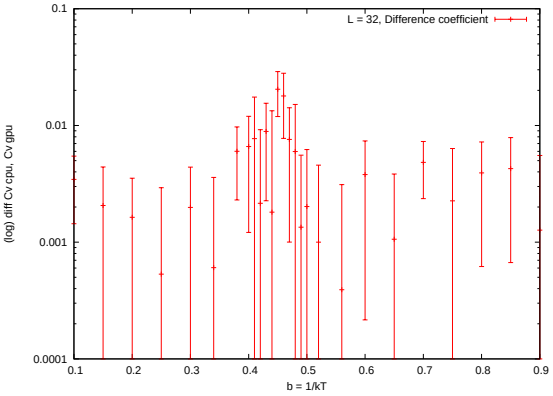


Συντελεστής διαφορών για την απόλυτη μαγνήτιση $\langle |M| \rangle$

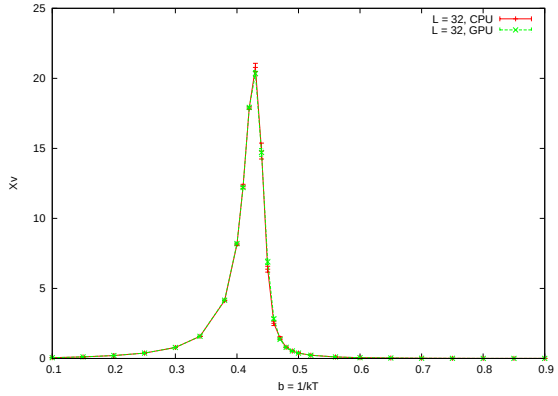
Σχήμα 4.10: Σύγκριση αποτελεσμάτων των αλγορίθμων gpu και cpu για L 32.



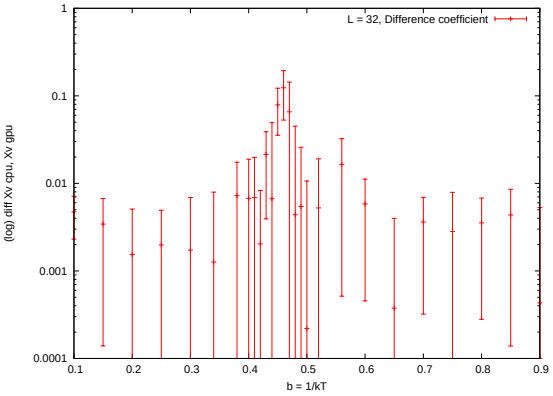
Η ειδική θερμότητα C



Συντελεστής διαφορών για την ειδική θερμότητα C

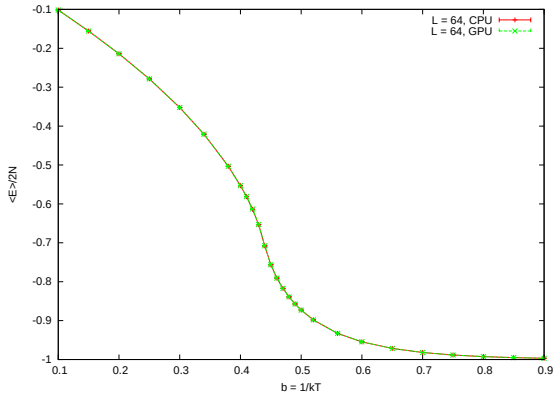


Η μαγνητική επιδεκτικότητα χ

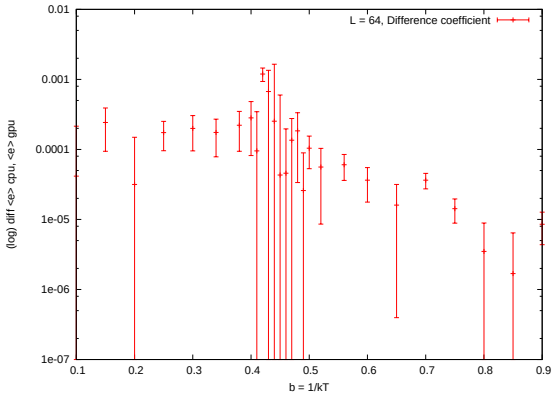


Συντελεστής διαφορών για την μαγνητική επιδεκτικότητα χ

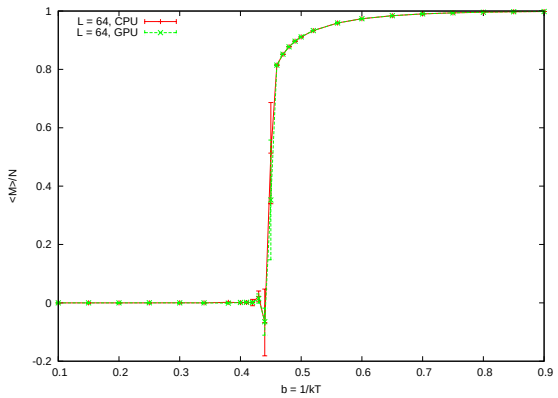
Σχήμα 4.10: Σύγκριση αποτελεσμάτων των αλγορίθμων gpu και cpu για L 32 (συνέχεια).



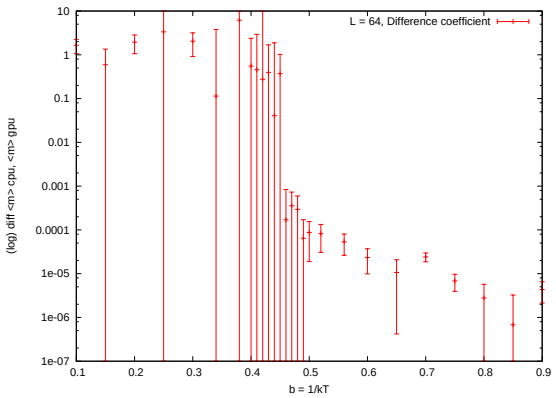
Η μέση ενέργεια ανά δεσμό $\langle E \rangle$



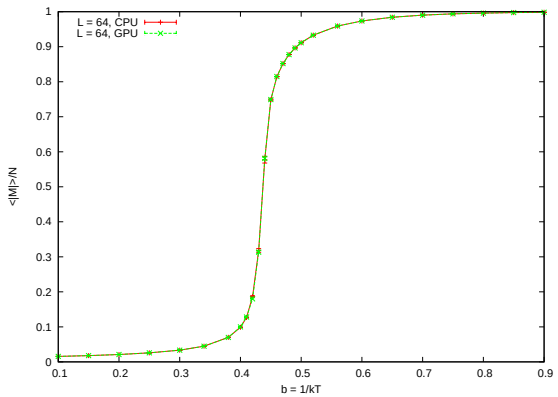
Συντελεστής διαφορών για την ενέργεια $\langle E \rangle$



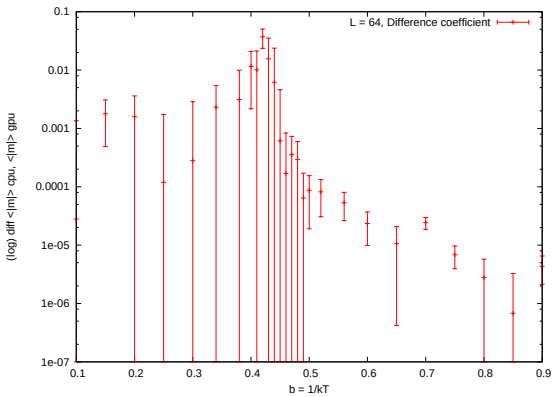
Η μέση μαγνήτιση ανά πλεγματική θέση $\langle M \rangle$



Συντελεστής διαφορών για την μαγνήτιση $\langle M \rangle$

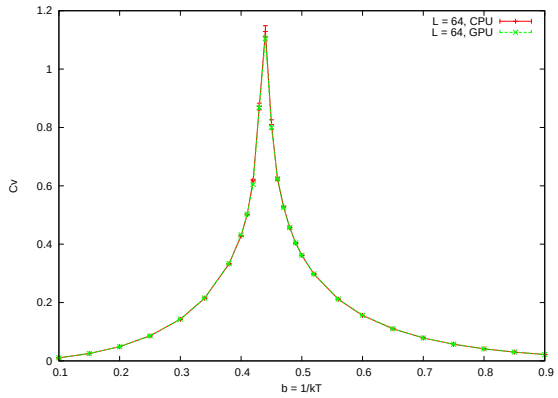


Η μέση απόλυτη μαγνήτιση ανά πλεγματική θέση $\langle |M| \rangle$

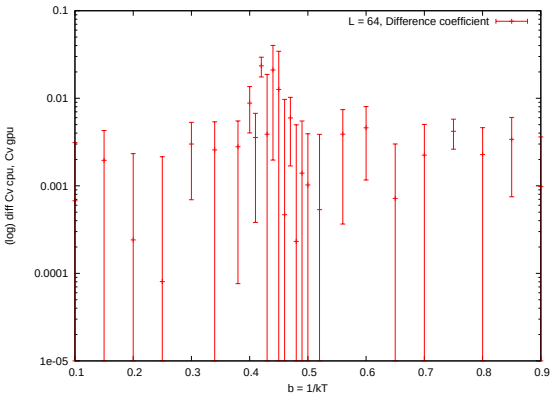


Συντελεστής διαφορών για την απόλυτη μαγνήτιση $\langle |M| \rangle$

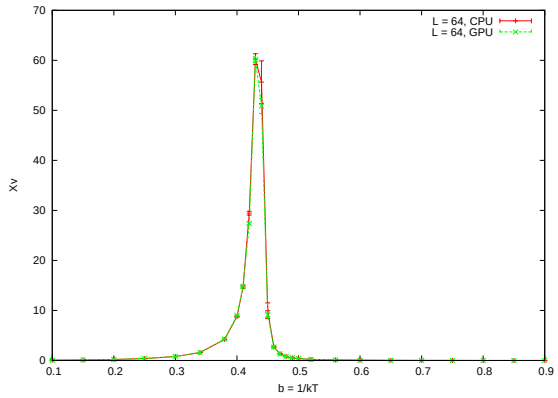
Σχήμα 4.11: Σύγκριση αποτελεσμάτων των αλγορίθμων gpu και cpu για L 64.



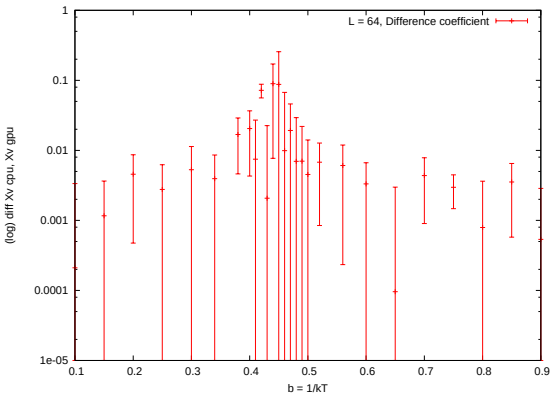
Η ειδική θερμότητα C



Συντελεστής διαφορών για την ειδική θερμότητα C

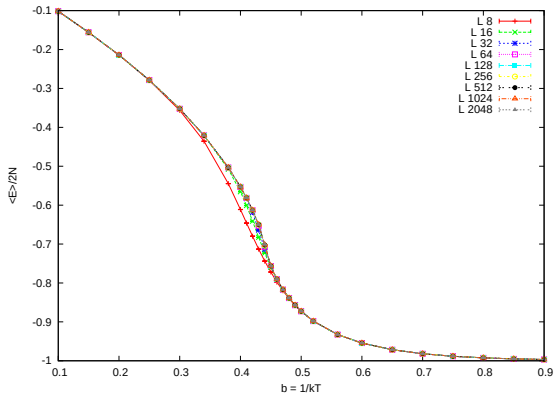


Η μαγνητική επιδεκτικότητα χ

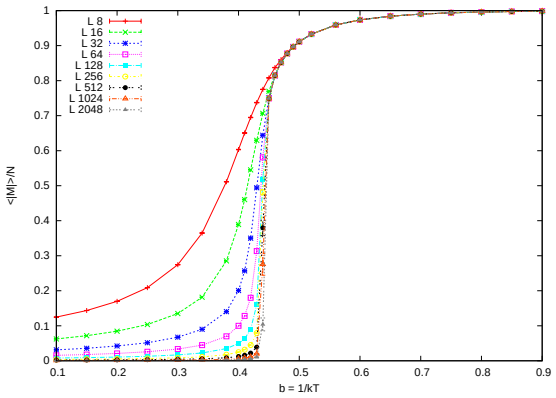


Συντελεστής διαφορών για την μαγνητική επιδεκτικότητα χ

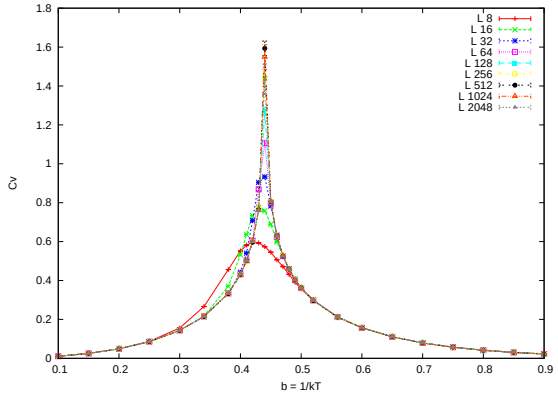
Σχήμα 4.11: Σύγκριση αποτελεσμάτων των αλγορίθμων gpu και cpu για L 64 (συνέχεια).



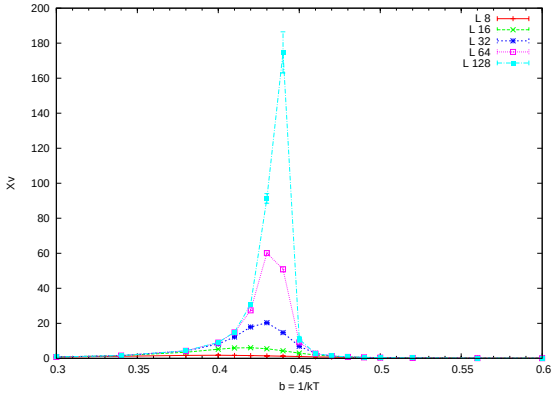
Η μέση ενέργεια ανά δεσμό $\langle E \rangle$



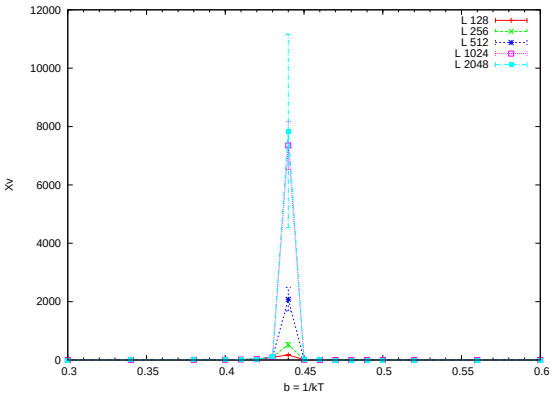
Η μέση απόλυτη μαγνήτιση ανά πλεγματική θέση $\langle |M| \rangle$



Η ειδική θερμότητα C



Η μαγνητική επιδεκτικότητα χ για μικρά L



Η μαγνητική επιδεκτικότητα χ για μεγάλα L

Σχήμα 4.12: Αποτελέσματα του αλγορίθμου υφής gpu για L 8 έως 2048.

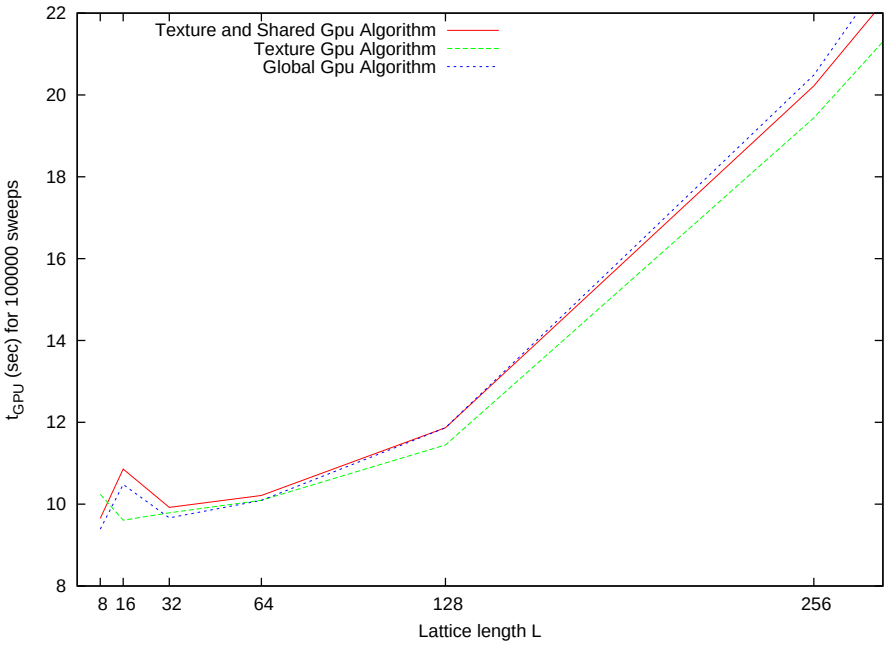
Μας ενδιαφέρει ο λόγος της επιτάχυνσης, επομένως κάνουμε τα γραφήματα του χρόνου εκτέλεσης για διαφορετικά πλέγματα και διαφορετικά μεγέθη block. Αφού βρούμε το βέλτιστο μέγεθος block και τον βέλτιστο αλγόριθμο για κάθε μήκος πλέγματος, κάνουμε το διάγραμμα του χρόνου εκτέλεσης του βέλτιστου αλγορίθμου με το βέλτιστο block configuration και του αλγορίθμου cpu ως προς τα μεγέθη των πλεγμάτων. Τέλος κάνουμε το διάγραμμα του λόγου των χρόνων εκτέλεσης μεταξύ gpu και cpu για να δούμε άμεσα το γράφημα της επιτάχυνσης και πως κάνει ”scaling” ο αλγόριθμος με την αύξηση του μήκους πλέγματος.

Αρχικά, παρατηρούμε στο σχήμα 4.15α□ ότι για μικρά πλέγματα ο αλγόριθμος σε gpu εκτελείται ακόμα και πιο αργά από τον σειριακό. Σε αυτές τις περιπτώσεις ο συνολικός αριθμός των blocks είναι πολύ μικρός και οι δυνατότητες της κάρτας γραφικών αξιοποιούνται στο ελάχιστο. Όπως έχουμε πει, είναι απαραίτητο σε μία κάρτα γραφικών όχι μόνο να έχουμε ενεργά το μέγιστο συνολικό πλήθος νημάτων, αλλά αρκετά περισσότερα. Ο λόγος είναι ότι όταν έχουμε περισσότερα νήματα από το μέγιστο πλήθος που υποστηρίζεται, θα είναι μεν ανενεργά αλλά θα χρησιμοποιηθούν ”έξυπνα” από την κάρτα γραφικών ώστε να κρύψει τις καθυστερήσεις που προκαλούνται από αριθμητικές πράξεις και προσβάσεις στην μνήμη και επομένως θα έχουμε καλύτερο scaling. Βλέπουμε στα γραφήματα 4.15 πως καθώς το L αυξάνεται έως έναν αριθμό, ο χρόνος εκτέλεσης παραμένει πρακτικά στάσιμος, ενώ ακόμα και για L 1024 έως L 2048 ο λόγος του χρόνου εκτέλεσης είναι αρκετά μικρότερος από αυτόν του κεντρικού επεξεργαστή, αλλά και τον θεωρητικό αν δεν υπήρχε καμία αλλαγή στην απόδοση[6].

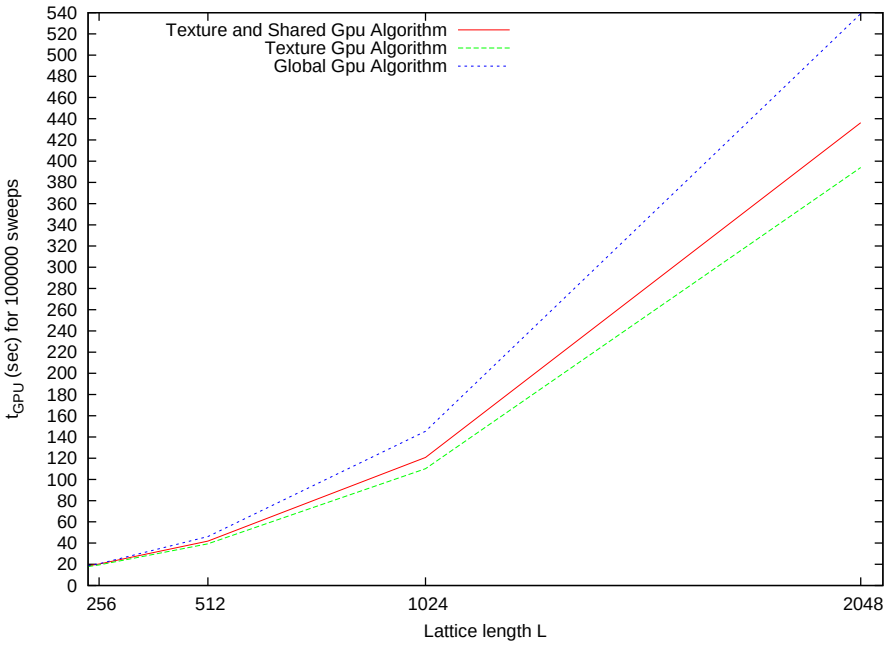
Παρατηρούμε ότι ο αλγόριθμος υφής υπερέχει ακόμα και του αλγορίθμου κοινόχρηστης μνήμης και υφής. Ο λόγος είναι ότι στην δεύτερη περίπτωση επιβαρύνουμε τον αλγόριθμο με διαδικασίες ανάγνωσης και εγγραφής στην κοινόχρηστη μνήμη, ενώ αυτές σε πολλές περιπτώσεις δεν χρειάζονται. Ο αλγόριθμος καθολικής μνήμης, κάνει μόνο χρήση της αργής καθολικής μνήμης, επομένως είναι αναμενόμενο να είναι και ο πιο αργός.

Σημαντικό ενδιαφέρον έχουν οι διαφορές στην ταχύτητα μεταξύ διαφορετικών μεγεθών των block. Είναι λίγο πιο πολύπλοκο να συμπεράνουμε τον λόγο των διαφορών σε κάποιες περιπτώσεις. Αρχικά, για πολύ μικρά πλέγματα είναι δύσκολο να βγάλουμε συμπεράσματα καθώς ο χρόνος εκτέλεσης είναι πολύ μικρός και τυχαίοι παράγοντες που θα μπορούσαν να προσθέσουν μερικά δέκατα του δευτερολέπτου θα επηρεάζουν σημαντικά τα αποτελέσματα. Εκτός του αλγορίθμου Metropolis, εδώ έχει σημασία και ο αλγόριθμος υπολογισμού της ενέργειας και της μαγνήτισης.

[6]Τετραπλασιασμός του πλήθους των πλεγματικών θέσεων με μηδενικό scaling θεωρητικά σημαίνει τετραπλασιασμός του χρόνου εκτέλεσης.

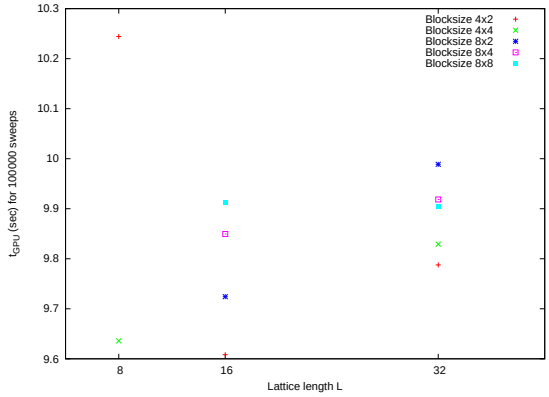


Μήκος πλέγματος L 8 έως 256

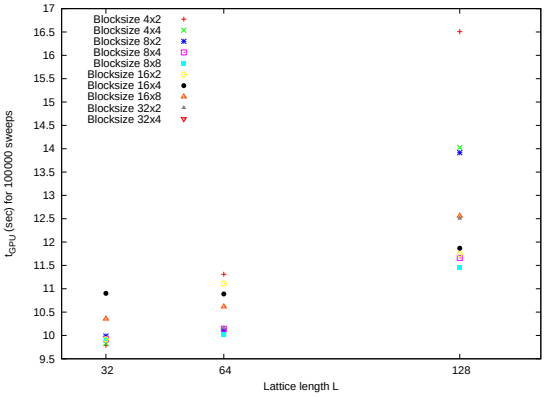


Μήκος πλέγματος L 256 έως 2048

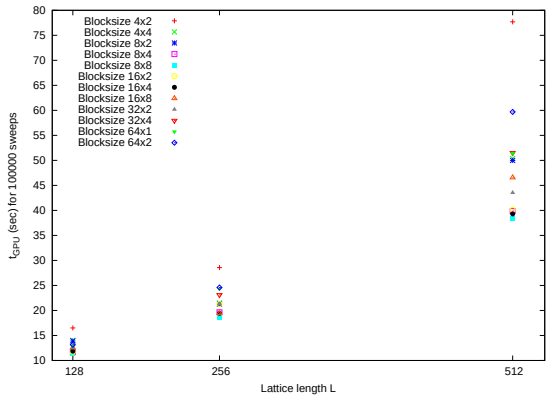
Σχήμα 4.13: Σύγκριση χρόνων εκτέλεσης των αλγορίθμων GPU για διάφορα πλέγματα.



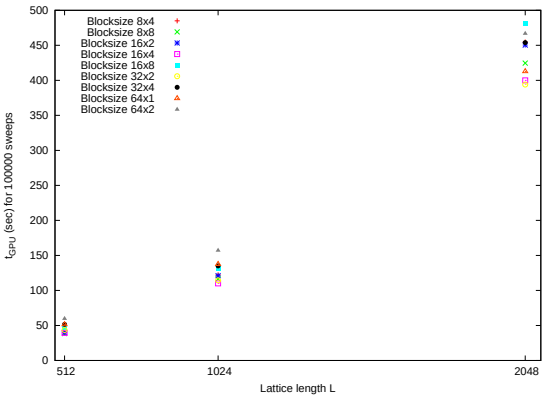
Μήκος πλέγματος L 8 έως 32



Μήκος πλέγματος L 32 έως 128

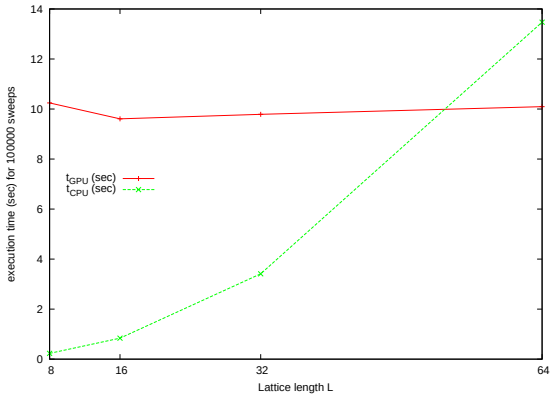


Μήκος πλέγματος L 128 έως 512

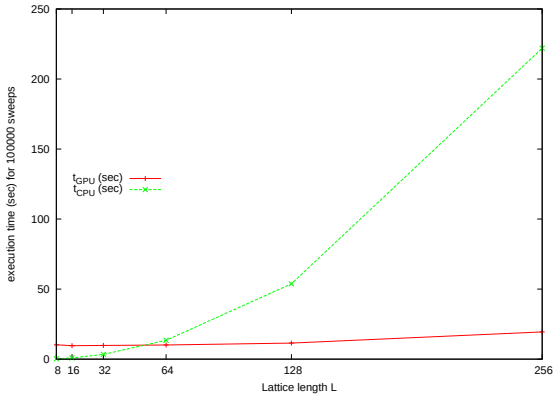


Μήκος πλέγματος L 512 έως 2048

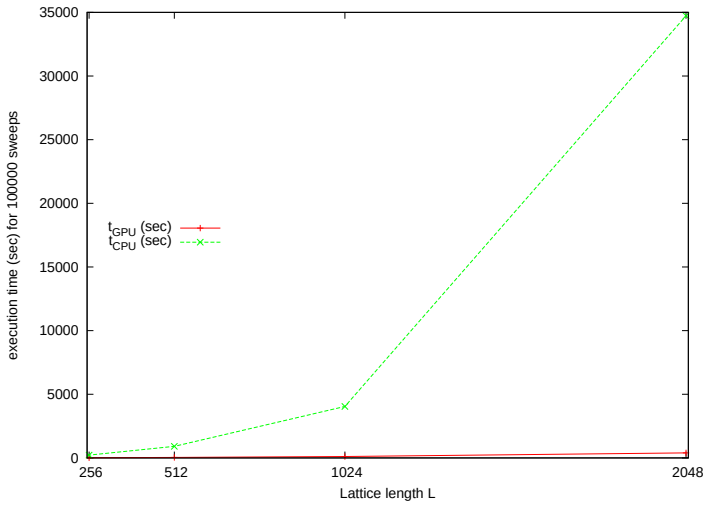
Σχήμα 4.14: Σύγκριση χρόνων εκτέλεσης του βέλτιστου αλγορίθμου κοινόχρηστης μνήμης για διάφορα μεγέθη block και διάφορα πλέγματα.



Μήκος πλέγματος L 8 έως 64

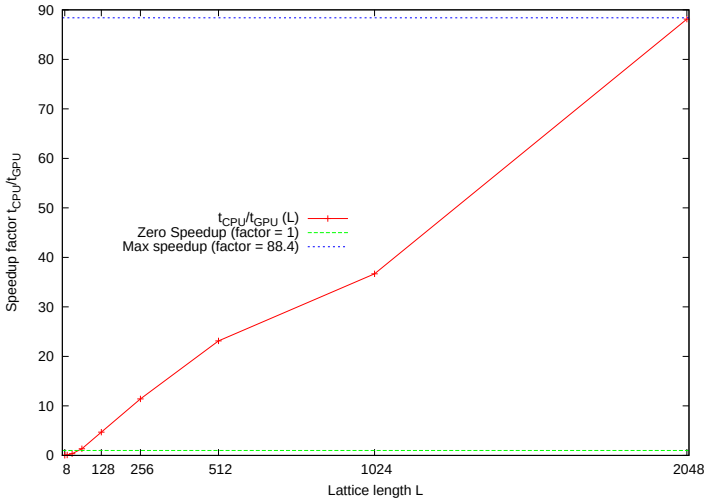


Μήκος πλέγματος L 8 έως 256



Μήκος πλέγματος L 256 έως 2048

Σχήμα 4.15: Σύγκριση χρόνων εκτέλεσης μεταξύ βέλτιστου αλγορίθμου GPU με βέλτιστο μέγεθος block με τον σειριακό αλγόριθμο για διάφορα L.



Σχήμα 4.16: Ο λόγος του χρόνου εκτέλεσης t_{CPU}/t_{GPU} του σειριακού αλγορίθμου αναφοράς ως προς τον βέλτιστο αλγόριθμο GPU με το βέλτιστο μέγεθος block για διάφορα L (συντελεστής επιτάχυνσης).

Όσο το μέγεθος $block_x * block_y$ είναι μικρό και συγκεκριμένα μικρότερο του 64, έχουμε μικρή πληρότητα, αφού οι πολυεπεξεργαστές είναι σε μεγάλο βαθμό αδρανείς. Αν αυτό το μέγεθος είναι μεγαλύτερο του 64 τότε θα έχουμε μικρή πληρότητα λόγω του αλγορίθμου υπολογισμού ενέργειας και μαγνήτισης. Στον Metropolis ένα νήμα αντιστοιχεί σε τέσσερις πλεγματικές θέσεις, ενώ στον αλγόριθμο υπολογισμού ενέργειας και μαγνήτισης, ένα νήμα αντιστοιχεί σε μία πλεγματική θέση. Επομένως, αν το μέγεθος του block για τον Metropolis είναι 128 το μέγεθος του block για τον αλγόριθμο υπολογισμού θα είναι 512 και κατά την διάρκεια εκτέλεσης του το 50% των πόρων του πολυεπεξεργαστή είναι αδρανείς[7].

Αν στα διαγράμματα σύγκρισης για διάφορα μεγέθη block συμπεριλαμβάναμε και τον αλγόριθμο κοινόχρηστης μνήμης, θα παρατηρούσαμε διαφορά για μέγεθος block στην διάσταση x μικρότερο από 16. Αυτό θα συμβαίνει καθώς πλέον οι αναγνώσεις και εγγραφές προς την κοινόχρηστη μνήμη πριν και μετά την εκτέλεση του κυρίως αλγορίθμου[8] δεν θα είναι αποδοτικές. Η κοινόχρηστη μνήμη έχει 16 τράπεζες και το βέλτιστο είναι οι προσβάσεις μας σε αυτές να είναι γραμμικές και να συμμετέχουν και τα 16 νήματα του μισού μίας δύνης.

[7]Αφού το μέγιστο πλήθος ενεργών νημάτων ανά πολυεπεξεργαστή είναι 1024.

[8]Για να αρχικοποιήσουμε την κοινόχρηστη μνήμη και στην συνέχεια για να επιστρέψουμε τα δεδομένα.

4.5 Συμπεράσματα

Ο αλγόριθμος είναι αποδοτικός για μήκος πλέγματος μεγαλύτερο του 128 και με επίτευξη της μέγιστης επιτάχυνσης για L 2048. Παρόλα αυτά, ο αλγόριθμος Metropolis έχει το πρόβλημα της κρίσιμης επιβράδυνσης για μεγάλα πλέγματα που τον καθιστά ασύμφορο στην προσομείωση τους. Επομένως, ενώ ο αλγόριθμος Metropolis ανήκει στην κατηγορία των πλήρως παραλληλοποιήσιμων αλγορίθμων, έχει σαφές όριο στο μέγεθος του πλέγματος το οποίο μπορεί να προσομοιώσει αποδοτικά. Αντίθετα, ο αλγόριθμος Wolff, χρησιμοποιώντας μη κανονικό πλέγμα και με περιορισμένους δεσμούς ανά sweep, είναι πολύ δύσκολο να υλοποιηθεί παράλληλα σε πολλούς επεξεργαστές. Αυτό το γεγονός τον καθιστά μη αποδοτικό σε GPU των οποίων η φιλοσοφία έγκειται στην μαζική εκτέλεση παράλληλων στοιχείων σε πάρα πολλές παράλληλες επεξεργαστικές μονάδες. Στην προσπάθεια που έγινε για έναν αλγόριθμο Wolff σε GPU, τα αποτελέσματα ήταν ιδιαίτερα απογοητευτικά και στις περισσότερες περιπτώσεις υπήρχε επιβράδυνση.

Ο σκοπός της χρήσης των καρτών γραφικών για επιστημονικό λογισμικό δεν αποσκοπεί στην αντικατάσταση του κεντρικού επεξεργαστή. Είδαμε πως κατά την διάρκεια της διεξαγωγής των προσομοιώσεων, οι εκτελέσεις στον κεντρικό επεξεργαστή γίνονταν ταυτόχρονα σε άλλο επεξεργαστή. Σκοπός λοιπόν είναι να επιταχύνει κομμάτια προγραμμάτων τα οποία μπορούν να επιταχυνθούν σημαντικά. Το ιδανικό είναι αυτές οι δύο πολύ διαφορετικές αρχιτεκτονικές να χρησιμοποιούνται ταυτόχρονα για την επίτευξη της μεγαλύτερης δυνατής απόδοσης. Στους σύγχρονους υπολογιστές βρίσκει κανείς δύο με τέσσερις επεξεργαστικούς πυρήνες και μία κάρτα γραφικών. Αν επιτύχει να μοιράσει τον φόρτο ενός αλγορίθμου σε όλους τους πόρους του συστήματος, τότε σαφώς θα επιτύχει πολύ καλύτερες επιδόσεις από έναν σειριακό αλγόριθμο. Γι αυτό τα τελευταία χρόνια έχουν αρχίσει να δημιουργούνται υβριδικοί υπερυπολογιστές, υπερυπολογιστές οι οποίοι αποτελούνται από πολλούς κεντρικούς επεξεργαστές, κάρτες γραφικών αλλά και επεξεργαστές PowerPC Cell με σκοπό την συνεργασία όλων, μοιράζοντας τον φόρτο στην πιο αποδοτική αρχιτεκτονική για την συγκεκριμένη διεργασία.

Τέλος, ο αλγόριθμος Metropolis και ο Wolff που παρουσιάστηκαν εδώ, αποτελούν μία πρώτη προσπάθεια. Κατά την ανάπτυξη των αλγορίθμων γεννήθηκαν πολλές νέες ιδέες, όπως για παράδειγμα η χρήση των ιδιοτήτων των υφών για εφαρμογή ”γρήγορων” περιοδικών συνθηκών για τετραγωνικό πλέγμα, αλλά και η ενεργή συνεργασία κεντρικού επεξεργαστή και κάρτας γραφικών κατά την εκτέλεση ενός sweep Wolff. Επομένως, με το πέρας αυτής της διπλωματικής εργασίας, ο σκοπός είναι η εφαρμογή αυτών των νέων ιδεών αλλά και των γνώσεων που αποκτήθηκαν στην πορεία για την ανάπτυξη ενός ακόμα πιο αποδοτικού αλγορίθμου Metropolis αλλά κυριώς με λιγότερους περιορισμούς και ενός αποδοτικού παράλληλου αλγορίθμου Wolff.

Βιβλιογραφία

[1] Κ. Αναγνωστόπουλος, *Σημειώσεις Υπολογιστικής Φυσικής ΙΙ*.

[2] M. E. J. Newman and G. T. Barkema, *Monte Carlo Methods in Statistical Physics*.

[3] K. Binder and D.W. Heermann, *Monte Carlo Methods in Statistical Physics*.

[4] P. Meyer, Thesis (2000), *Computational Studies of Pure and Dilute Spin Models*.

[5] N. Drakos, R. Moore, (1999), *Jackknife Error Estimates*.

[6] Rob Farber, (2009), *CUDA, Supercomputing for the Masses*.

[7] NVIDIA Co. et al., (2008), *GPU Gems 3*. (διατίθεται δωρεάν http://developer.nvidia.com/object/gpu-gems-3.html)

[8] NVIDIA Co., *CUDA Programming Guide 2.3*.

[9] NVIDIA Co., *CUDA Programming Guide 3.0*.

[10] NVIDIA Co.,*CUDA Reference Manual*.

[11] NVIDIA Co., *CUDA Best Practices Guide*.

[12] NVIDIA Co., *CUDA-GDB User Manual: The NVIDIA CUDA Debugger*.

[13] NVIDIA Co., *Visual Profiler User Guide*.

[14] Mark Harris, *Optimizing CUDA*.

[15] Mark Harris, *Optimizing Parallel Reduction in CUDA*.

[16] Damir A. Jamsek, IBM Research (2009), *Designing and optimizing compute kernels on NVIDIA GPU*, Proceedings of the 2009 Asia and South Pacific Design Automation Conference.

[17] Keh-Ying Lin and Yu-En Lin, *Low-Temperature Series Expansions for the Ising Model on a Checkerboard Lattice with First and Second Neighbour Interactions*, CHINESE JOURNAL OF PHYSICS VOL. 39, NO. 3 JUNE 2001

[18] Amdahl, *G.M. Validity of the single-processor approach to achieving large scale computing capabilities*. In AFIPS Conference Proceedings vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483-485.

[19] T. Preis, P. Virnau, W. Paul, and J. J. Schneider, *GPU accelerated Monte Carlo simulation of the 2D Ising model*. Journal of Computational Physics 228, 4468-4477 (2009) doi:10.1016/j.jcp.2009.03.018

[20] Wolff, Ulli (1989), *Collective Monte Carlo Updating for Spin Systems*. Physical Review Letters 62 (4): 361, doi:10.1103/PhysRevLett.62.361, PMID 10040213

[21] Wolff, Ulli (2006) *Monte Carlo errors with less errors*. Comput.Phys.Commun.156:143-153,2004; Erratum-ibid.176:383,2007 doi:10.1016/S0010-4655(03)00467-3 10.1016/j.cpc.2006.12.001 http://arxiv.org/abs/hep-lat/0306017v4

[22] Swendsen R. H., and Wang, J., *Nonuniversal critical dynamics in Monte Carlo simulations*. Phys. Rev. Lett., 58(2):86–88, 1987

[23] Y.-Y. Fang, I.-L. Yen and R. Dubash (1993), *Improving the performance of Lee's maze routing algorithm on parallel computers via semi-dynamic mapping strategies*. Technical Report CPS-93-35, Michigan State University, December, 1993.

[24] S.Bae, S.H. Ko, P.D. Coddington (1995), *Parallel Wolff cluster algorithms*. International Journal of Modern Physics C 6: 197, 1995, doi:10.1142/S0129183195000150

[25] Alexei Strelchenko, (2009), *Computational physics on graphics hardware architectures: Monte Carlo simulations of Ising model*.

[26] Florian Wende, *Implementation of the 2D Ising Model using CUDA*.

[27] Wu-chun Feng and Shucai Xiao, *To GPU Synchronize or Not GPU Synchronize?*. Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), Paris, France, May 2010. http://synergy.cs.vt.edu/pubs/papers/feng-iscas2010-gpusync.pdf

[28] Maged Elhajal, Benjamin Canals, Claudine Lacroix, (2001), *Comparison of several tetrahedra-based lattices*. Canadian Journal of Physics for the proceedings of the Highly Frustrated Magnetism 2000 Conference, Waterloo, Ontario, Canada, June 11-15, 2000. doi: 10.1139/cjp-79-11-12-1353 http://arxiv.org/abs/cond-mat/0104331v2