UNIVERSITÀ DEGLI STUDI DI MILANO

Facoltà di Scienze e Tecnologie
Laurea Triennale in Fisica

Simulazione del Modello di Ising 2D su Processore Grafico

Relatore: Prof. Nicola Manini
Correlatore: Prof. Davide Galli

Stefano Mandelli
Matricola n° 723065
A.A. 2013/2014

Codice PACS: 05.10.-a

Simulazione del Modello di Ising 2D su Processore Grafico

Stefano Mandelli
Dipartimento di Fisica, Università degli Studi di Milano,
Via Celoria 16, 20133 Milano, Italia

Sommario

In questa tesi si riportano i risultati delle simulazioni numeriche di un modello di spin il modello di Ising a campo magnetico esterno nullo, simulato con metodo Monte Carlo su processore grafico (GPU). L'utilizzo di un algoritmo che sfrutta in maniera ottimale le caratteristiche delle GPU nella simulazione di questo modello, ha permesso di raggiungere un incremento delle prestazioni notevole. Il massimo speed-up raggiunto è di un fattore 10^2 su di un reticolo di dimensione 4096×4096.

Relatore: *Prof. Nicola Manini*
Correlatore: *Prof. Davide Galli*

Indice

Introduzione

Questo lavoro di tesi ha come obiettivo quello di presentare un nuovo algoritmo che permetta di simulare l'evoluzione di un reticolo di Ising su processore grafico (GPU) in tempi inferiori che su CPU. La scelta di usare le schede grafiche come supporto per il calcolo nasce dal fatto che il modello di Ising 2D si adatta particolarmente bene alla struttura finemente parallela delle stesse. Differentemente dalle più moderne CPU, in cui il numero di threads paralleli si aggira in media sui 16, per quanto riguarda le schede grafiche il numero di threads paralleli è molto più grande. L'unità logica strettamente parallela delle schede video è il così detto *Warp* che è composto da una griglia di 32×32 threads. Per ogni ciclo di clock della scheda vengono quindi svolti in parallelo 1024 threads.

La struttura della scheda video è composta da molti warp che si attivano con una latenza molto bassa facendo quindi risultare *quasi paralleli* anche due threads appartenenti a diversi warp. Dato il gran numero di threads a disposizione, questo tipo di dispositivo si presta bene a simulare reticoli di Ising molto grandi (per essere certi di essere il più vicino possibile al limite termodinamico del problema). La struttura hardware del processore grafico di una scheda video è molto più semplice di una tradizionale CPU. I livelli di memoria sono più semplici (la memoria a disposizione si divide in due grandi aree, *memoria veloce* e *memoria lenta*) e non sono presenti tutti i livelli di buffer e di cache che invece caratterizzano le CPU quindi l'organizzazione della memoria incide molto sui tempi di esecuzione. Nel caso studiato in questa tesi è stato notato che una divisione del reticolo in una *doppia scacchiera* garantisce al meglio il rispetto delle gerarchie di memoria della scheda grafica. Non essendoci conflitti di banchi di memoria (*Bank Conflict*), la fine parallelizzazione viene sfruttata al meglio.

Affrontare l'implentazione del modello di Ising su scheda grafica ha permesso di scontrarsi con una problematica generale: la generazione parallela di numeri pseudo-random indipendenti su GPU. Gli algoritmi che generano numeri pseudo random sono strettamente seriali (per generare il numero pseudo-random r_{n+1} è necessario che prima sia stato generato il numero r_n) quindi è stato necessario scrivere l'algoritmo XorShift di Marsaglia (1992), in modo tale che venissero generate, per ogni nodo di calcolo, catene indipendenti di numeri pseudo-random. Successivamente ne è stata dimostrata l'indipendenza sottoponendo l'insieme di numeri a due test di randomicità che hanno dato esito positivo.

E' stato verificato il corretto funzionamento del programma realizzato per il caso bidimensionale, confrontando i risultati numerici con la soluzione esatta proposta da Onsager per size infinito. Per quanto riguarda il modello bidimensionale è stato valutato il guadagno, in termini di tempo di esecuzione, che una stuttura

finemente parallela come quella delle GPU può portare nella simulazione del problema su reticolo. Il lavoro ha notevoli possibilità di sviluppi futuri, in primis si possono sfruttare le strutture tridimensionali, che l'ambiente di programmazione CUDA mette a disposizione, per organizzare i threads in cubetti e quindi simulare un reticolo di Ising tridimensionale che dal punto di vista fisico è di notevole interesse in quanto non si conosce ancora una soluzione analitica del problema. Come seconda possibilità di sviluppo e utilizzo, il programma potrà essere usato e riadattato per simulare e prevedere numericamente in modo veloce tutti quei sistemi o fenomeni che scalano con gli stessi esponenti critici del modello di Ising.

1 Meccanica Statistica

In questo lavoro di tesi il sistema studiato è di tipo **statistico**. I sistemi statistici sono molto diffusi nello studio della materia condensata perchè i sistemi considerati, ad esempio i magneti, sono composti da un numero molto grande (nell'ordine di 10^{23}) di parti microscopiche interagenti che tipicamente possono essere atomi o molecole. Queste parti sono solitamente tutte uguali (atomi dello stesso tipo) e obbediscono alle più semplici leggi del moto. La difficoltà matematica nello studio di questi sistemi è insita nel gran numero di equazioni che è funzione del numero di particelle del sistema stesso. Per questo motivo è opportuno affrontare il problema in modo differente e quindi usare una strategia che permetta di descrivere questi fenomeni con poche equazioni facili da trattare per ottenere dei risultati almeno approssimati. In questo caso entrano in gioco i metodi di analisi della Meccanica Statistica che propone un approccio al problema di tipo probabilistico. È possibile quindi definire il concetto di *insiemi statistici* come tutti quegli insiemi le cui variabili sono *variabili aleatorie*. Queste sono caratterizzate da un certo range di valori (configurazioni) $\Omega = \{\mu_1, \mu_2, \cdots\}$ a cui è associata una misura di probabilità p_μ.

Si consideri un sistema governato da una certa Hamiltoniana H che descrive l'energia totale del sistema per ogni particolare configurazione μ. Se il sistema è isolato, l'unica condizione è il fatto che la sua energia rimanga costante. In questo modo il sistema rimane bloccato sempre sullo stesso livello energetico. Se non c'è degenerazione[1] il sistema rimane bloccato anche nella stessa configurazione. Se è degenere, allora il sistema può spaziare tra tutte quelle configurazioni a stessa energia ed è il tipico esempio di un **ensamble microcanonico**. Un sistema più comune, di interesse in questo lavoro di tesi, sono quelli in cui vi è la presenza di un bagno termico (**thermal resevoir**). In questo modo le condizioni che vengono imposte sul sistema sono quelle di avere N, V e T fissati, dove N è il numero di particelle del sistema, V è il suo volume e T la sua temperatura.

A tal proposito si considerino due configurazioni diverse μ_1 e μ_2 del sistema, è possibile definire un indice della facilità con cui il sistema passa dalla prima alla seconda configurazione $R(\mu_1 \to \mu_2)$. Ad un dato instante t, la dinamica del sistema è

$$\frac{d\omega_{\mu_1}(t)}{dt} = \sum_{\mu_2} \left[\omega_{\mu_2} R(\mu_2 \to \mu_1) - \omega_{\mu_1} R(\mu_1 \to \mu_2)\right] \tag{1}$$

dove $\omega_{\mu_n}(t)$ rappresenta la probabilità che il sistema si trovi nella configurazione μ_n all'istante t. Dato l'approccio numerico di questa tesi, la variabile temporale t

[1] Avere degenerazione implica che esistono più configurazioni tali che l'energia totale del sistema è sempre la stessa

non è da considerarsi come una grandezza continua ma come *passo Monte Carlo*. Partendo dall'equazione (1) è possibile definire quando un sistema è all'equilibrio, cioè quando $d\omega_{\mu_1}/dt \to 0$. Per l'equazione (1) quando i due rate totali di transizione $\omega_{\mu_2} R(\mu_2 \to \mu_1)$ e $\omega_{\mu_1} R(\mu_1 \to \mu_2)$ sono uguali. In questa condizione di equilibrio, Gibbs nel 1902, dimostrò che la probabilità di occupazione delle varie configurazioni del sistema è data dalla *statistica di Boltzmann*

$$\omega_\mu = \frac{1}{Z} e^{-\beta E_\mu} \tag{2}$$

dove E_μ è l'energia associata allo stato μ, $\beta = 1/(k_B T)$ dove k_B è la costante di Boltzmann e

$$Z = \sum_\mu e^{-\beta E_\mu} \tag{3}$$

denota la funzione di partizione del sistema. E' possibile fissare alcuni parametri del sistema. Ogni parametro fissato ha una sua variabile coniugata che è data in funzione delle derivate dell'energia libera F. Per esempio, il valore di aspettazione dell'energia interna è dato da

$$U = \frac{1}{Z} \sum_\mu E_\mu e^{-\beta E_\mu} \,. \tag{4}$$

Conoscendo la (3) è possibile scrivere l'equazione (4) come derivata di Z

$$U = -\frac{1}{Z} \frac{\partial Z}{\partial \beta} = -\frac{\partial \log Z}{\partial \beta} \,. \tag{5}$$

Anche il calore specifico è possibile scriverlo in funzione alle derivate di Z

$$C = \frac{\partial U}{\partial T} = -k_B \beta^2 \frac{\partial U}{\partial \beta} = k_B \beta^2 \frac{\partial^2 \log Z}{\partial \beta^2} \tag{6}$$

che è legato all'entropia dalla relazione

$$C = T \frac{\partial S}{\partial T} = -\beta \frac{\partial S}{\partial \beta} \,. \tag{7}$$

Uguagliando le due scritture di calore specifico ed integrando il tutto rispetto a β si trova l'entropia

$$S = -k_B \beta \frac{\partial \log Z}{\partial \beta} + k_B \log Z \,. \tag{8}$$

È possibile introdurre l'espressione per l'energia libera F. Usando le equazioni (4) e (8) si ottiene

$$F = U - TS = -kT \log Z \,. \tag{9}$$

In questo modo si è mostrato come U, F, C ed S possono essere ricavate tutte direttamente dalla funzione di partizione Z. In termodinamica ogni parametro, vincolo e campo interagente ha delle variabili coniugate che rappresentano in che modo il sistema reagisce alla perturbazione del suo parametro corrispondente. Se del sistema si fissano dei parametri, come: volume o campo magnetico esterno, le variabili coniugate a queste due grandezze sono pressione o magnetizzazione, che sono sempre fornite in funzione delle derivate di F. In questo modo si producono termini che dipendono da coppie di parametri XY dove Y è il campo che noi fissiamo e X rappresenta la variabile coniugata. In questo modo, all'equilibrio, posso definire delle *medie statistiche* per le grandezze del sistema

$$\langle X \rangle = \sum_{\mu} X_{\mu} p_{\mu} = \frac{1}{Z} \sum_{\mu} X_{\mu} e^{-\beta E_{\mu}} \,. \tag{10}$$

E possibile riscrivere la media in funzione alle derivate di Z

$$\langle X \rangle = \frac{1}{\beta} \frac{\partial \log Z}{\partial Y} = -\frac{\partial F}{\partial Y} \,. \tag{11}$$

Questo è un approccio efficace per calcolare le medie delle quantità che sono di interesse.[2] Anche se nell'Hamiltoniana non sono presenti termini che esprimono in modo diretto i campi a cui accoppiare le quantità interessanti da stimare è comunque possibile in modo semplice, costruire un campo accoppiato alla quantità di interesse e considerarlo nullo subito dopo il calcolo dell'appropriata derivata. In questo modo è possibile scrivere che la media è

$$\langle X \rangle = \frac{1}{\beta} \left(\frac{\partial \log Z}{\partial Y} \right)_{Y=0} = - \left(\frac{\partial F}{\partial Y} \right)_{Y=0} \,. \tag{12}$$

Dal punto di vista fisico sono di particolare interesse anche le fluttuazioni dell'osservabile considerata precedentemente, cioè la quantità $\langle X^2 \rangle - \langle X \rangle^2$ dove $\langle X^2 \rangle$ può essere ricavata dall'equazione (12) ottenendo

$$\begin{aligned} \langle X^2 \rangle &= \frac{1}{\beta^2 Z} \frac{\partial^2}{\partial Y^2} \sum_{\mu} e^{-\beta E_{\mu}} \bigg|_{Y=0} = \frac{1}{\beta^2 Z} \frac{\partial^2 Z}{\partial Y^2} \bigg|_{Y=0} = \\ &= \frac{1}{\beta^2} \frac{\partial^2 \ln Z}{\partial Y^2} \bigg|_{Y=0} + \left(\frac{1}{\beta} \frac{\partial \ln Z}{\partial Y} \bigg|_{Y=0} \right)^2 = \frac{1}{\beta} \frac{\partial \langle X \rangle}{\partial Y} \bigg|_{Y=0} + \langle X \rangle^2 . \end{aligned} \tag{13}$$

Concludendo è possibile scrivere la fluttuzione della variabile di interesse come

$$\langle X^2 \rangle - \langle X \rangle^2 = \frac{1}{\beta} \frac{\partial \langle X \rangle}{\partial Y} \bigg|_{Y=0} = \frac{\chi}{\beta} \tag{14}$$

dove χ indica la suscettività di X.

[2] Le grandezze fisiche di interesse per il modello di Ising sono: magnetizzazione, energia interna e calore specifico

1.1 Il modello di Ising

Oltre ai modelli costituiti da N copie di sistemi non interagenti, il più semplice modello interagnete a cui applicare i metodi della meccanica statistica è il *modello di Ising*, che trova la sua applicazione fisica nella spiegazione dei fenomeni di magnetizzazione di materiali (anti-)ferromagnetici. Viene considerato un reticolo (ad esempio ipercubico) in D dimensioni. Ad ogni cella del reticolo viene associato uno spin s_i che può essere solo del tipo $s_i = \{+1, -1\}$ a seconda che la direzione del dipolo magnetico (o spira), associato alla cella $i - esima$ del reticolo risulti verso o verso il basso. Il sistema è descritto dall'Hamiltoniana di Ising

$$\mathcal{H} = -J \sum_{\langle ij \rangle} s_i s_j - h \sum_i s_i \tag{15}$$

dove h identifica un eventuale campo magnetico uniforme esterno. La prima sommatoria è fatta su tutte le coppie di siti primi vicini e J indica la costante di accoppiamento tra spin. Se $J > 0$ stiamo descrivendo un sistema ferromagnetico, se $J < 0$ uno anti-ferromagnetico. In questo lavoro di tesi si considera il caso il caso $D = 2$ dimensionale, $J > 0$ e campo magnetico esterno nullo, quindi $h = 0$.

Al limite termodinamico, il modello di Ising (escluso il caso 1D) presenta una transizione di fase in prossimità di una *temperatura critica* T_c. Per temperature maggiori di T_c il sistema si comporta in modo paramagnetico. Per temperature inferiori invece si ha un fenomeno di magnetizzazione spontanea. Le principali grandezze fisiche che possono essere calcolate coi metodi espressi nel paragrafo precedente sono il valor medio dell'energia del sistema $\langle E \rangle$, la magnetizzazione media del sistema $\langle M \rangle$ [4] e la capacità termica a volume costante C_V che viene calcolata col teorema di fluttuazione-diffusione [(14]. Tutte queste quantità estensive sono divergenti nel limite termodinamico. Nelle simulazioni statistiche vengono considerati dei modelli finiti di $N = D \times D$ spins che si desidera confrontare per diversi size. È utile pertanto confrontare la densità di energia $\langle e \rangle = \langle E \rangle / N$, la densità di calore specifico $c_V = C_V / N$ e di magnetizzazione che è la variabile coniugata al campo esterno h

$$m = \frac{\langle M \rangle}{N} = -\frac{1}{N}\frac{\partial F}{\partial h} = \frac{1}{N\beta}\frac{\partial \ln Z}{\partial h} = \frac{1}{NZ}\sum_\mu \left(\sum_i s_i \right)_\mu e^{-\beta E_\mu} = \frac{1}{N}\langle \sum_i s_i \rangle . \tag{16}$$

Per il caso $h = 0$ e $T > T_c$, $\langle M \rangle$ si annulla. Questo comportamento può essere spiegato nel seguente modo: per $T > T_c$ le fluttuazioni termiche prevalgono sulla tendenza del termine di interazione J. La lunghezza di correlazione è molto piccola e ogni spin ha la stessa probabilità di avere come valore $+1$ o -1, in questo modo $\langle M \rangle$ risulta nulla. Per temperature $T < T_c$ gli spin risentono fortemente

dell'interazione coi loro primi vicini. Si nota che in questo range di temperature il modello di Ising 2D, presenta una transizione di fase netta. Il sistema passa, in modo spontaneo, ad una situazione di ordine in cui gli spin sono orientati prevalentemente nella stessa direzione quindi hanno in maggioranza valore $+1$ oppure -1. I due casi, per campo magnetico esterno nullo ($h = 0$), sono equiprobabili, in quanto per $h = 0$ l'Hamiltoniana di Ising è pari per inversione di tutti gli spins, per tempi molto lunghi entrambi gli stati vengono popolati per la stessa quantità di tempo facendo risultare, anche in questo caso, $m = 0$. Per modelli finiti sufficientemente grandi è ugualmente possibile effettuare delle valutazioni e misure di magnetizzazione media spontanea del sistema per $T < T_c$. Questo si ottiene settando lo stato di partenza in modo tale che uno dei due stati sia più popolato dell'altro. Se per convenzione si sceglie uno stato di partenza con una maggioranza di spins a valore $+1$, lo stato di equilibrio dell'Hamiltoniana di Ising, con molti spin a valore $+1$ sarà favorito. In questo modo, per temperature inferiori a quella critica, si hanno dei valori di magnetizzazione non nulli e dello stesso segno ed è quindi possibile valutare l'entità della magnetizzazione spontanea.

2 Metodo Monte Carlo

Per calcolare le grandezze fisiche del sistema statistico in esame, è necessario conoscere la funzione di partizione del sistema o alcune sue derivate. Nel caso del modello di Ising, esiste una soluzione analitica solo per il caso 1D e 2D [9]. Per $D \geq 3$ la funzione di partizione al limite termodinamico è sconosciuta. Uno dei metodi che in meccanica statistica viene usato per valutare la funzione di partizione del sistema, anche senza la sua forma analitica, è quello del calcolo numerico andando a sommare ogni singolo termine. Quando però tale somma contiene troppi termini la si può approssimare con un numero finito di valutazioni. Nel caso del modello di Ising, la soluzione a questo problema non è banale. Per un piccolo reticolo di dimensione 5×5 i vari spin possono essere riarrangiati in modo da creare un numero di stati possibili di 2^N, in questo caso $2^{25} = 33\,554\,432$. Il numero è tutto sommato piccolo ed è quindi possibile effettuare una valutazione della funzione di partizione eseguendo tutte le somme e calcolare l'andamento della magnetizzazione in funzione a β tramite l'equazione (12). Le grandezze fisiche calcolate in questo modo (per esempio magnetizzazione media per spin e calore specifico per spin) deviano molto dalla previsione delle stesse grandezze per spin calcolate considerando un reticolo infinito (soluzione di Onsager). Per avvicinarci il più possibile alla condizione di reticolo infinito la tecnica che viene usata è quella del *finite size scaling* in cui si simulano reticoli molto grandi che sono una buona approssimazione del modello al limite termodinamico. Per questo motivo

è opportuno fare esperimenti con reticoli sempre più grandi e con condizioni al bordo periodiche in modo da essere il più vicino possibile al limite termodinamico e ai reticoli macroscopici formati da un numero di Avogadro di siti. Come detto precedentemente, la complessità di calcolo della funzione di partizione scala come 2^N è quindi ovvio che calcolare la funzione di partizione come somma termine a termine in modo numerico, per un reticolo costituito da più di poche decine di siti è impossibile. Per quanto grande sia la potenza di calcolo a disposizione è impossibile stimare Z in tempi accettabili. Risulta molto più conveniente avere un approccio al problema del tipo *Monte Carlo*. In questo modo non è più richiesta la conoscenza della funzione di partizione. L'idea alla base del metodo è quella di generare, un sottoinsieme di configurazioni *tipiche* che permettono di stimare nel modo migliore le quantità medie come e, c_v ed m. A questi stati viene attribuita una certa probabilità di essere occupata (*Importance sampling*).

2.1 Importance Sampling e processi di Markov

La strategia è quella di scegliere un set di configurazioni con un peso dato dalla statistica di Boltzmann. Dato che si sta effettuando una simulazione numerica, il tempo *Monte Carlo* di evoluzione del sistema è finito, quindi il set di configurazioni accessibili sarà di numero finito: $\{\mu_1 \cdots \mu_M\}$. Ogni configurazione $\{\mu_n\}$ viene generata con probabilità p_μ quindi una stima dell'osservabile X del sistema è

$$X_{MC} = \frac{\sum_{n=1}^{M} X_{\mu_n} p_{\mu_n}^{-1} e^{-\beta E_{\mu_n}}}{\sum_{n=1}^{M} p_{\mu_n}^{-1} e^{-\beta E_{\mu_n}}} = \frac{\sum_{n=1}^{M} X_{\mu_n}}{\sum_{n=1}^{M} 1} = \frac{1}{M} \sum_{n=1}^{M} X_{\mu_n} \,. \tag{17}$$

Le M configurazioni, con il loro peso dato dalla statistica di Boltzmann, vengono generate in modo random secondo un processo noto come *Catena di Markov*. L'obiettivo del processo è quello di generare lo stato $n+1$ facendo riferimento solo allo stato precedente n e con probabilità ω_μ che rispetti l'equazione (2). Per fare questo si fa evolvere il sistema in modo random dalla configurazione precedente μ alla configurazione ν. Introduciamo quindi il concetto di *probabilità di transizione* $P(\mu \to \nu)$. Per essere un vero processo di Markov è necessario che la probabilità di transizione soddisfi le seguenti caratteristiche:

- Non deve dipendere dal tempo, cioè nel nostro caso dal *passo Monte Carlo*;
- La probabilità che lo stato ha di transire allo stato ν dipende solo dallo stato μ;
- E' soddisfatto il vincolo: $\sum_\nu P(\mu \to \nu) = 1$

se sono verificate queste proprietà allora l'insieme finito di configurazioni è stato generato seguendo un processo di Markov a stati discreti (noto appunto come *Catena di Markov*). È possibile dimostrare che è possibile raggiungere qualsiasi configurazione del sistema che abbia probabilità non nulla attraverso un numero finito di passi Markov. In questo modo è garantito il fatto che il processo in uso rispetta l'ergodicità del sistema.

2.2 Detailed Balance

Una condizione aggiuntiva sui processi di Markov è quella del *Detailed Balance* o micro-reversibilità

$$p_\mu P(\mu \to \nu) = p_\nu P(\nu \to \mu)\,. \tag{18}$$

Si possono fare ulteriori considerazioni: il sistema è in equilibrio quindi

$$\sum_\nu p_\mu P(\mu \to \nu) = \sum_\nu p_\nu P(\nu \to \mu)\,, \tag{19}$$

usando il vincolo: $\sum_\nu P(\mu \to \nu) = 1$ otteniamo che

$$p_\mu = \sum_\nu p_\nu P(\nu \to \mu)\,. \tag{20}$$

La semplice soddisfazione di questa equazione garantisce l'equilibrio ma non dice nulla sul fatto che la distribuzione di probabilità per ogni stato tenda a p_μ. Si può verificare questo fatto usando le definizioni di equilibrio e di catene di Markov. Le varie $P(\mu \to \nu)$ sono gli elementi della matrice $\mathbf{P}$ che chiamiamo *Matrice di Markov*. Sia $w_\mu(t)$ la probabilità che il sistema sia nello stato μ al tempo t. Considerando il tempo in modo discreto, si può dire che la probabilità legata al fatto che il sistema al tempo $t+1$ possa trovarsi nello stato ν è data da

$$w_\nu(t+1) = \sum_\mu P(\mu \to \nu) w_\mu(t)\,. \tag{21}$$

Dal punto di vista matriciale è possibile riscrivere l'equazione (21) come

$$\mathbf{w}(t+1) = \mathbf{P} \cdot \mathbf{w}(t)\,, \tag{22}$$

in cui $\mathbf{w}(t)$ è il vettore i cui elementi sono i vari pesi $w_\mu(t)$. Se questo processo di Markov raggiunge l'equilibrio per $t \to \infty$ allora rimane ancora soddisfatta la scrittura

$$\mathbf{w}(\infty) = \mathbf{P} \cdot \mathbf{w}(\infty)\,. \tag{23}$$

Tuttavia è possibile che il processo raggiunga un equilibrio dinamico in cui le distribuzioni di probabilità ruotano intorno a certi valori. Questa rotazione viene definita come *ciclo limite*. Nel caso di $\mathbf{w}(\infty)$ questa condizione soddisfa l'equazione

$$\mathbf{w}(\infty) = \mathbf{P}^n \cdot \mathbf{w}(\infty) \tag{24}$$

dove n è la lunghezza del ciclo limite. In questo modo si è verificato che nonostante le distribuzioni di probabilità soddisfino la condizione di equilibrio, questo non garantisce il fatto che, per tempi lunghi, convergano alla distribuzione di probabilità voluta p_μ.

Questo problema si aggira, come detto precedentemente, aggiungendo la condizione di *detailed balance* che è quella definita all'inizio del paragrafo

$$p_\mu P(\mu \to \nu) = p_\nu P(\nu \to \mu)\,. \tag{25}$$

Questa condizione indica che il sistema tende a passare dalla configurazione μ a quella ν con tanta facilità quanta quella a passare dallo stato ν a quello μ. In un ciclo limite, dove la probabilità di occupazione degli stati cambia in modo ciclico, la condizione di *detailed balance* viene violata. In questo modo vengono annullati i cicli limite e la distribuzione di probabilità degli stati converge effettivamente a quella voluta. In questo modo è stato sottolineato il fatto che le probabilità di transizione, per soddisfare il *detailed balance* devono essere del tipo

$$\frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \frac{p_\nu}{p_\mu} = e^{-\beta(E_\nu - E_\mu)}\,. \tag{26}$$

Se il vincolo $p_\mu = \sum_\nu p_\nu P(\nu \to \mu) = 1$ è soddisfatto, vengono garantite l'ipotesi ergodica, la condizione di equilibrio e ho la certezza che le distribuzioni di probabilità (le varie p_μ) sono date dal peso della statistica di Boltzmann.

2.3 Acceptance Ratios

È possibile decomporre la probabilità $P(\mu \to \nu)$ come

$$P(\mu \to \nu) = g(\mu \to \nu) A(\mu \to \nu) \tag{27}$$

in cui la parte relativa a $g(\mu \to \nu)$ viene chiamata *probabilità di selezione* ed indica la probabilità che l'algoritmo ha, partendo dalle condizioni dello stato μ, di generare uno stato ν. La parte relativa ad $A(\mu \to \nu)$ viene chiamata rapporto di accettazione o *acceptance ratio*. L'algorimo, partendo dallo stato μ genera uno stato ν, $A(\mu \to \nu)$ indica la frazione di volte in cui viene accettata e mantenuta la mossa $\mu \to \nu$. Questa frazione è una probabilità, ed è possibile sceglierla, per esempio, estraendo un numero casuale in modo uniforme tra 0 e 1.

2.4 Algoritmo di Metropolis applicato al Modello di Ising

Dato che le fluttuazioni di energia, rispetto all'energia totale del sistema, sono piccole, un modo sufficientemente efficiente di generare delle mosse Monte Carlo, è quello definito dalla *dinamica a singolo spin-flip*. I vari stati vengono generati in modo che il successivo sia differente dal precedente per il flip di un singolo spin del reticolo preso inizialmente in modo casuale. Lo stato μ e quello ν differiscono tra loro solo per il flip di un singolo spin. In questo modo è possibile definire la probabilità di selezione come

$$g(\mu \to \nu) = \frac{1}{N}\,. \tag{28}$$

Con questa probabilità di selezione, la condizione del detailed balance prende la forma

$$\frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \frac{g(\mu \to \nu)A(\mu \to \nu)}{g(\nu \to \mu)A(\nu \to \mu)} = \frac{p_\nu}{p_\mu} = e^{-\beta(E_\nu - E_\mu)}\,. \tag{29}$$

Dato che $g(\mu \to \nu) = g(\nu \to \mu)$ si sceglie l'acceptance ratio in modo tale che soddisfi l'equazione (29)

$$\frac{A(\mu \to \nu)}{A(\nu \to \mu)} = e^{-\beta(E_\nu - E_\mu)}, \tag{30}$$

da cui è possibile dedurre che

$$A(\mu \to \nu) = A_0 e^{-\frac{1}{2}\beta(E_\nu - E_\mu)}\,. \tag{31}$$

Per avere un algoritmo che sia il più efficiente possibile l'acceptance ratio deve essere significativamente diversa da zero. Il parametro A_0 è scelto in funzione ad alcune considerazioni su come è fatta l'Hamiltoniana di Ising. E' facile osservare che la differenza di energia tra due stati, in modulo, è al massimo pari a $|\Delta E| = 2zJ$ dove z è il numero di primi vicini, che nel caso di reticolo 2D vale $z = 4$, quindi per il reticolo 2D abbiamo che al massimo $|\Delta E| = 8J$. La differenza di energia tra lo stato μ e ν è

$$|E_\nu - E_\mu| \leq 2zJ. \tag{32}$$

In questo modo, il massimo possibile valore dell'esponenziale vale

$$e^{-\frac{1}{2}\beta(E_\nu - E_\mu)} \leq e^{\beta z J} \tag{33}$$

che permette di stabilire la scelta migliore possibile del coefficiente

$$A_0 = e^{-\beta z J}\,. \tag{34}$$

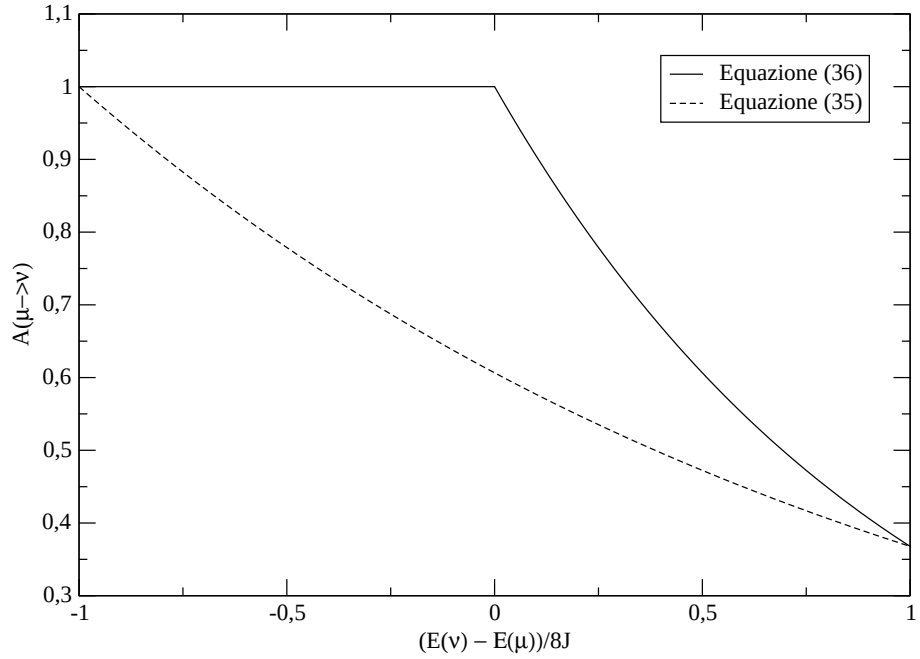


Figura 1: Acceptance ratio in funzione alla variazione di energia, per $\beta = 1$ e $J = 1$

La scrittura finale per l'acceptance ratio risulta quindi:

$$A(\mu \to \nu) = e^{-\frac{1}{2}\beta(E_\nu - E_\mu + 2zJ)}, \tag{35}$$

in modo da avere $A(\mu \to \nu) \leq 1$. Si può verificare che l' $A(\mu \to \nu)$ scritta ora è molto inefficiente. Il sistema rimane per troppo tempo nello stesso stato. Una scelta migliore che rispetta tutte le condizioni di quella precedente è data proprio dall'acceptance ratio proposta da Metropolis

$$A(\mu \to \nu) = \begin{cases} e^{-\beta(E_\nu - E_\mu)} & E_\nu - E_\mu > 0 \\ 1 & \text{altrimenti.} \end{cases} \tag{36}$$

In Fig. 1 l'equazione (36) è confrontata con la (35).

Per il modello di Ising le acceptance ratio appena descritte hanno la particolarità che possono essere calcolate mediante la sola conoscenza degli spins primi vicini allo spin di cui si propone il suo flip, questo perchè l'interazione nel modello di Ising è a corto raggio. Nel caso dell'Hamiltoniana di Ising, questa differenza è possibile scriverla in modo molto semplice, in funzione solo dagli spin primi vicini

dello spin di cui si propone il flip:

$$\begin{aligned} E_\nu - E_\mu &= -J \sum_{\langle i,j \rangle} s_i^\nu s_j^\nu + J \sum_{\langle i,j \rangle} s_i^\mu s_j^\mu = \\ &= -J \sum_{i \neq k} s_i^\nu (s_k^\nu - s_k^\mu) = -2J s_k^\mu \sum_{i \neq k} s_i^\mu \,. \end{aligned} \tag{37}$$

3 Implementazione del modello in CUDA-C

La scrittura dell'algoritmo Metropolis per il modello di Ising è basata sulla soluzione di una sequenza di problemi:

1. Conoscenza della nuova architettura di parallelizzazione CUDA;
2. Distribuzione del reticolo nella struttura logica della scheda video in modo da garantire sempre la miglior performance di calcolo;
3. Scelta di un Generatore di Numeri Pseudo-Random che si adatti bene alla struttura finemente parallela[3] della scheda video.
4. Verificare l'incremento di performance rispetto alla CPU.

3.1 Architettura delle GPU

L'organizzazione hardware delle schede video, nel momento della loro programmazione, è nascosta da una struttura logica. Come si può notare in figura 2 le architetture di CPU e GPU sono molto differenti tra loro. Nel secondo caso, dal punto di vista strutturale, i transistor dedicati all'elaborazione dati (quadrati verdi) sono molti di più e finemente distribuiti. Questa organizzazione permette al programmatore di controllare l'esecuzione del proprio programma anche nei più intimi dettagli. È infatti compito del programmatore pensare alla distribuzione della parallelizzazione del proprio problema, gestendo il modo in cui i vari threads interagiscono tra loro minimizzando i conflitti (che abbassano le performance) che si possono presentare in un paradigma di alta parallelizzazione come quello in uso in questo progetto. I conflitti coinvolgono soprattutto l'utilizzo errato dei vari livelli di memoria di cui dispone la scheda video. Le strutture di calcolo (ALU) hanno vari livelli di memoria a cui possono accedere. Ogni livello di memoria

[3]La parallelizzazione su scheda video viene spesso definita: *parallelizzazione fine*. A differenza delle CPU, una singola scheda video permette di avere centinaia (migliaia nelle ultime architetture di tipo TITAN) unità di calcolo che cooperano in modo parallelo. Per le CPU il numero di unità di calcolo, per singola macchina, si ferma sulle poche decine.

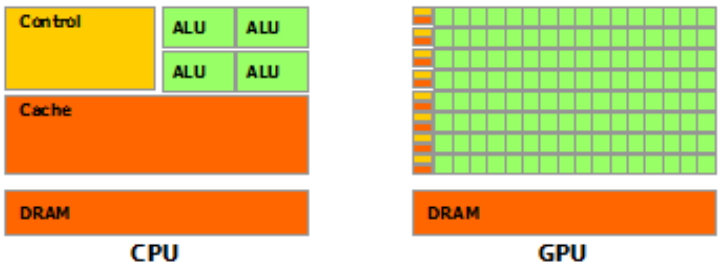


Figura 2: Confronto tra le architetture logiche di CPU (a sinistra) e GPU (a destra)

ha delle caratteristiche che in fase di organizzazione dell'algoritmo non possono essere messe in secondo piano.

3.2 Il Modello di Ising in CUDA

La struttura finemente parallela si presta in modo ottimo ad ospitare un reticolo di Ising in modo che ogni sito reticolare sia legato ad una unità di calcolo. Il problema a cui bisogna far fronte è che tutte le unità di calcolo non hanno in comune gli stessi banchi di memoria. Queste, che da ora identificheremo col loro rispettivo logico cioè il *Thread*, per motivi di costruzione della scheda sono organizzate in *Blocchi*. Ogni Blocco può accedere ad una sua propria area di memoria[4] veloce e ad una area di memoria globale, ma lenta. Il programmatore può quindi pensare che un modo per parallelizzare il problema sia quello di dividere il reticolo iniziale in piccoli pezzetti di reticolo, metterli sulla memoria condivisa ai threads di un singolo blocco ed affidare la loro evoluzione ad ogni singolo blocco. Senza ulteriori dettagli una parallelizzazione gestita in questo modo porta ad un errore. Come detto precedentemente le aree di memoria sono proprie dei singoli blocchi, quindi ogni blocchetto evolverebbe in modo a se stante non riuscendo a comunicare con il blocco adiacente. Per gli elementi di confine tra i blocchi, si può pensare di effettuare l'update sfruttando la memoria globale che è comune a tutto il reticolo, quindi a tutti i blocchi. Questo approccio porta ad una parallelizzazione del modello corretta ma poco efficace infatti la memoria globale, rispetto a quella condivisa dal singolo blocco, è molto più lenta. Un programma è tanto più efficiente quanto il programmatore riesce a rispettare la *gerarchia* delle aree di memoria. In figura 3 viene riassunta l'esecuzione di un kernel CUDA indicando, nei vari momenti, come avvengono gli accessi di memoria. Per la simulazione

[4]Propria nel senso che il Thread di un blocco, non ha modo di accedere all'area di memoria di un blocco diverso.

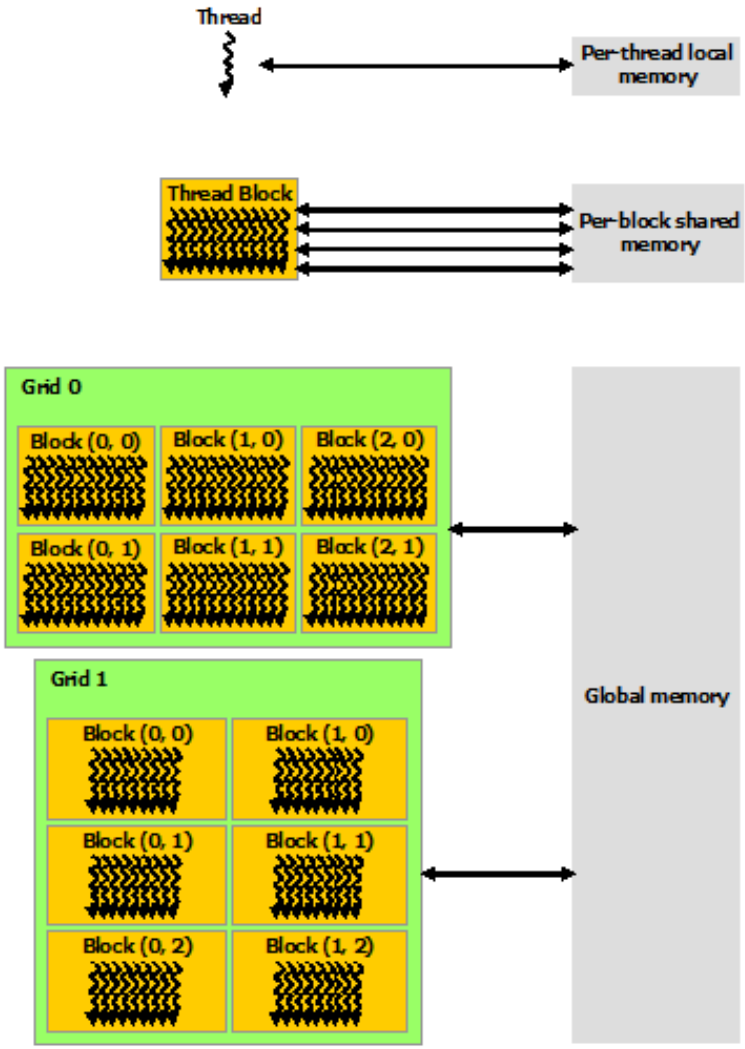


Figura 3: Schema di lancio di un kernel CUDA con relativi accessi di memoria

del Modello di Ising tramite l'algoritmo di Metropolis sono stati usati i seguenti livelli di memoria:

- *Global Memory*: Memoria Globale della scheda su cui vengono inizialmente caricate le configurazioni iniziali del reticolo. E' in comune a tutti i Threads di ogni Blocco. E' una memoria lenta sia in scrittura che lettura;
- *Texture Memory*: Memoria Globale comune a tutti i threads di ogni blocco. Ha la proprietà di essere molto veloce in lettura ma molto lenta in scrittura (molto di più addirittura rispetto anche alla global). Per questo viene considerata come una memoria in read-only.

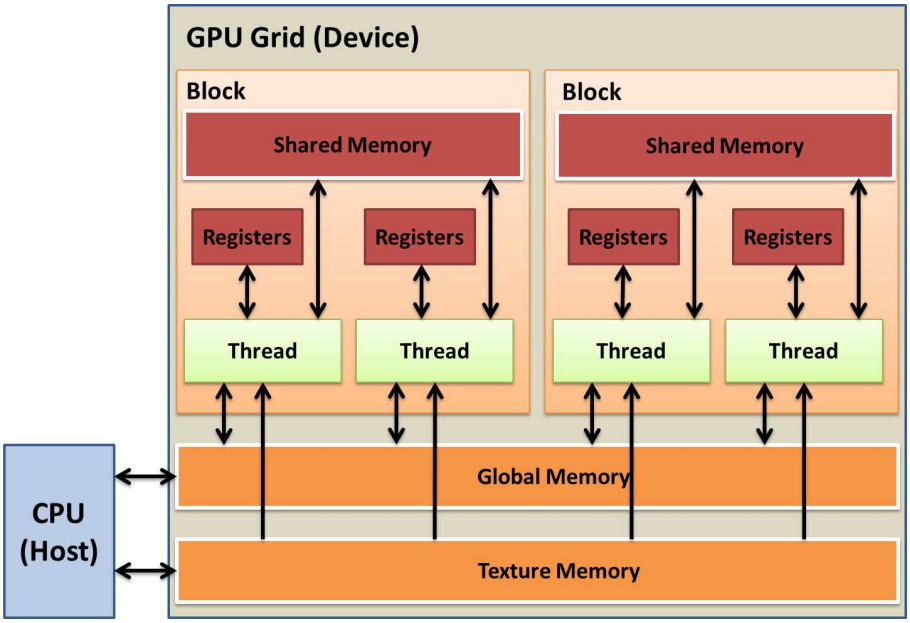


Figura 4: Architettura di una GPU. Sono messi in evidenza i possibili accessi di memoria tra le varie strutture logiche

- *Shared Memory*: Memoria Condivisa dal singolo blocco. Threads di blocchi differenti accedono a banchi di memoria shared differenti. E' una memoria molto veloce sia in lettura che scrittura;
- *Registro*: Piccolissima area di memoria propria del singolo thread. Diversi thread, anche dello stesso blocco, non posso accedere allo stesso registro.

In figura 4 viene presentato uno schema esauriente di quali sono gli accessi di memoria permessi tra le varie parti logiche della scheda.

3.3 Strategia di Organizzazione del Reticolo

Come accennato nel paragrafo precedente le performance del programma sono in relazione all'utilizzo della memoria della scheda video. L'obiettivo è quello di costruire il programma in modo tale da usare sempre banchi di memoria ad alta velocità di lettura-scrittura e garantire sempre il rispetto della gerarchia delle allocazioni di memoria, in modo da evitare conflitti di banco (*bank conflict*). Nel nostro caso, la memoria più veloce che abbiamo a disposizione è quella di tipo shared. L'obiettivo che ci si pone è quello di utilizzare sempre la memoria shared senza dover ricorrere a quella global per l'update dei siti di reticolo di separazione tra un blocco e l'altro. Per raggiungere questo scopo il reticolo è stato decomposto in una *doppia scacchiera*. Il primo livello di scacchiera avviene

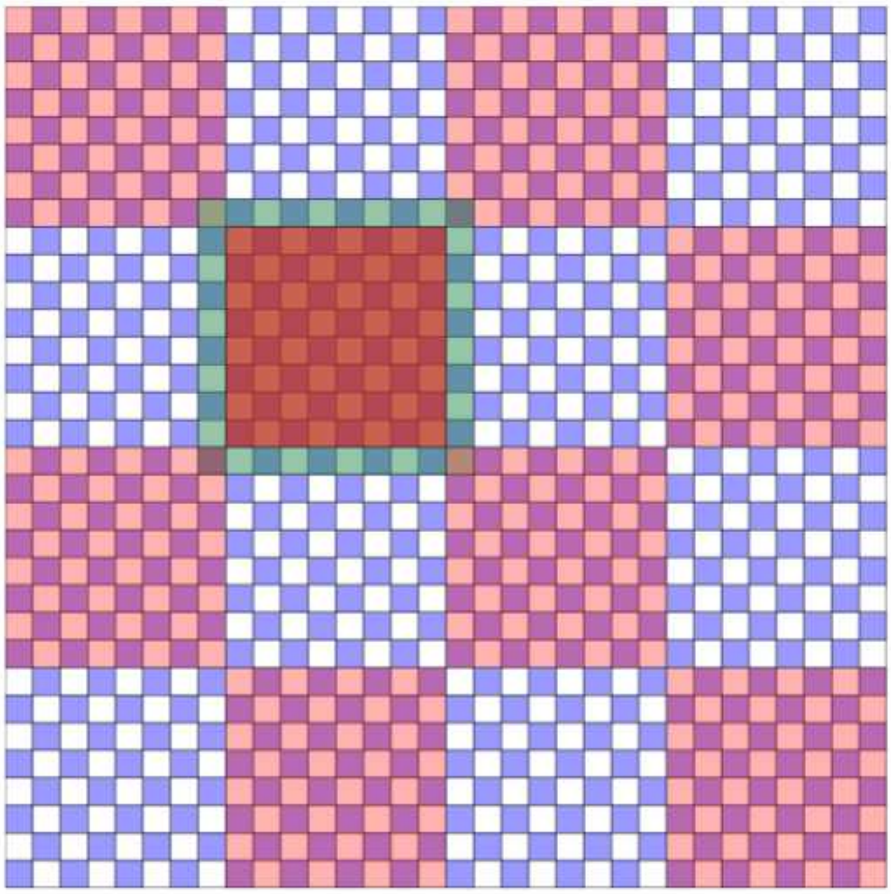


Figura 5: Decomposizione del reticolo in una doppia scacchiera

a livello di blocchi, il secondo livello di scacchiera avviene a livello dei threads di ogni blocco. Il punto fondamentale consiste nell'allocazione, per ogni singolo blocco, di una matrice con 2 righe e 2 colonne in più rispetto al numero di threads usati per il blocco nelle due dimensioni x, y. In questo modo posso lavorare sul blocco avendo memorizzato nella memoria veloce anche gli elementi di confine tra blocco e blocco. Dato che il modello in studio prevede interazioni tra spins primi vicini, all'interno del singolo blocco i threads è possibile aggiornali a scacchiera. In questo modo non ci sono conflitti con gli aggiornamenti dei vari siti del reticolo. Gli spin contenuti nelle 2 righe e 2 colonne in più, appartengono nel momento del loro update, al blocco primo vicino, quindi anche i blocchi vanno aggiornati seguendo uno schema a scacchiera. Se tutti i blocchi venissero aggiornati in modo parallelo senza questa divisione, ci sarebbero dei conflitti nell'update degli spin di confine tra blocchi primi vicini. Usando una struttura a doppia scacchiera, tutti i threads dei vari blocchi lavorano basandosi sui primi vicini in modo corretto. In figura 5 è rappresentato in modo grafico come avviene

la decomposizione del reticolo. Concentrandosi sul blocco rosso, vediamo che al suo interno gli spin vengono aggiornati a scacchiera. Se tutti i blocchi venissero aggiornati contemporaneamente, gli spin presenti nell'alone verdino sarebbero soggetti ad un conflitto. Separando l'esecuzione del kernel in due parti, la prima che agisce sui blocchi rossi, la seconda su quelli non colorati, vengono trattati in modo corretto anche gli spin di confine tra blocchi primi vicini. Ricapitolando, l'algoritmo elaborato lavora nel seguente modo:

- il reticolo, dalla memoria global viene diviso in sottoreticoli affidati ai vari blocchi della scheda video;
- Viene lanciato un primo Kernel che lavora solo sui blocchi pari. Questo kernel, aggiorna ogni blocco in 2 step:
  - Prima vengono aggiornati i threads pari;
  - Successivamente vengono aggiornati i threads dispari;
- Viene lanciato un secondo kernel che lavora sui blocchi dispari. Tutti i threads dei blocchi dispari vengono aggiornati in due step come avviene per quelli pari.

I due kernel che vengono lanciati, in realtà sono lo stesso kernel lanciato due volte, la prima con un parametro di offset 0 la seconda volta con una parametro di offset di 1. Un ulteriore accorgimento che è stato preso riguarda il modo in cui sono allocati i valori di $e^{-\beta\Delta E}$. Come visto nella sezione precedente, la differenza di energia tra due stati μ e ν è possibile scriverla come nell'equazione (37) Le possibili variazioni di energia sono un numero finito di valori compresi tra $-8J$ a $+8J$. Questi otto valori sono fissi quindi è possibile allocare all'inizio gli otto valori di $e^{-\beta\Delta E}$ che verranno ogni volta usati per fare il confronto con la variabile random r generata. Questo passaggio è stato ottimizzato nel senso che gli otto valori non sono stati allocati in una zona di memoria qualsiasi come la global memory, ma sono stati allocati nella texture. Come già spiegato, ogni livello di memoria ha la caratteristica di essere più o meno veloce in scrittura/lettura. La texture è lentissima in scrittura ma è la memoria più veloce, presente sulle GPU, in fase di lettura. Dato che sono solo 8 i valori da allocare, il tempo speso per farlo (anche se la texture è lenta in scrittura) è molto piccolo e i vari confronti avvengono in modo molto veloce perché il dato viene richiamato molto più velocemente rispetto ad una sua allocazione in una differente area di memoria.

3.4 Generatore di Numeri Pseudo-Random

3.4.1 Il problema della parallelizzazione

La generazione di numeri pseudo-random è il problema più complesso da risolvere per simulazioni in parallelo. I numeri random sono generati in serie, cioè la generazione del numero x_n dipende solo dal suo precedente x_{n-1}. Trasportare un processo da seriale a parallelo è solitamente complesso. Un approccio che viene usato in questi casi, indipendentemente dall'algoritmo di generazione della successione di numeri pseudo-random, è quello di avere una funzione (il vero generatore) che può essere richiamata più volte da ogni nodo di calcolo. Ogni nodo di calcolo passa alla funzione diversi seed scelti in modo opportuno in funzione al tipo di generatore. Il risultato sono tante sequenze di numeri pseudo-random indipendenti.

3.4.2 Generatori Lineari-Congruenziali (LCG)

Gli algoritmi di generazione di numeri random più semplici sono quelli di tipo *lineare congruenziale* (LCG). La definizione di un LCG è data da

$$x_{n+1} = ax_n + c \pmod{m}. \tag{38}$$

Per appropriati valori di a, c ed m il periodo di questa classe di generatori è m. Per un'efficiente implementazione è inutile prendere il valore m più grande della dimensione del più grande intero rappresentabile sulla propria architettura. In questo modo il periodo di questa classe di generatori è sempre ristretto a $m \leq 2^{64}$. Il vantaggio di questo tipo di generatori è la grande velocità. Per il caso in studio però non sono consigliati in quanto il periodo troppo piccolo non permette di avere numeri pseudo-random indipendenti per tutta la lunghezza della simulazione.

3.4.3 Generazione di numeri random attraverso due parametri

La scarsa qualità dei numeri pseudo-random generati dagli algoritmi di tipo LCG (Generatori Lineari Congruenziali) rende necessaria la ricerca di un nuovo RNG (Random Numbers Generator). Numeri pseudo-random di buona qualità e con periodo più lungo degli LCG si possono creare usando algoritmi di tipo MWC (*Multiply With Carry*) proposti da Marsaglia e Zaman (1991). L'algoritmo prevede che il numero x_{n+1} venga generato dal suo precedente inserendo un ulteriore shift dato dal parametro c_{n+1}

$$\begin{aligned} x_{n+1} &= ax_n + c_n \pmod{m} \\ c_{n+1} &= [(ax_n + c_n)/m]. \end{aligned} \tag{39}$$

Rispetto agli algoritmi di tipo LCG è possibile generare catene di numeri pseudo-random significativamente più lunghe. Per valori di a scelti ad-hoc, è stato trovato che il periodo di una serie di numeri pseudo-random generata in questo modo è di $p = am - 2$. Per valori di a leggermente più piccoli di $m = 2^{32}$ il periodo si avvicina molto a quello di un LCG a 64bit. Il valore di a scelto è quello suggerito nell'articolo di Marsaglia cioè $a = 4294967118$. In questo modo è possibile creare un flusso di numeri pseudo-random, la cui implementazione si adatta molto bene all'architettura finemente parallela delle schede video e garantisce una periodicità più lunga rispetto ai generatori lineari congruenziali. Dal punto di vista dell'implementazione su GPU, il tutto avviene in modo molto naturale. Al posto di allocare un solo vettore di numeri iniziali primi tra loro (cosa che si fa per gli algoritmi di tipo LGC), se ne alloca un secondo con i valori di c_0 generati seguendo l'equazione (39). Per ogni `threadIdx` abbiamo quindi una coppia di valori x_n e c_n che generano serie indipendenti di numeri pseudo-random.

3.4.4 Generazione numeri pseudo-random tramite xorshift

Gli algoritmi di tipo MWC, non sono però il massimo dell'ottimizzazione che si può raggiungere. Una classe di generatori più efficiente proposta da Marsaglia è quella dei generatori **xorshift**. Consistono nel scegliere in modo opportuno, un vettore di semi iniziali in cui le componenti non sono numeri interi ma bits. Se si sceglie un vettore x di lunghezza w, è possibile dimostrare che si ottiene un ottimo e molto veloce generatore di numeri pseudo-random effettuando su questo vettore operazioni di shift. Definiamo la matrice $\mathbf{L}$

$$\mathbf{L} = \begin{pmatrix} 0 & 0 & \cdots & 0 \\ 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 \end{pmatrix} \tag{40}$$

come matrice di *spostamento a sinistra* che trasforma il vettore $x = (x_1, \cdots, x_w)$ in $x = (x_2, \cdots, x_w, 0)$. L'operazione totale di XORShift proposta da Marsaglia è però di or esclusivo quindi per effettuare uno xorshift di a posizioni si ha che

$$x_{n+1} = x_n(\mathrm{Id} \oplus \mathbf{L}^a). \tag{41}$$

Con il simbolo $\oplus$ si intende la somma con mod 2 su ogni singolo elemento. Un vettore del tipo $x_n = (1, 1, 0, 1)$ viene trasformato con uno xorshift sinistro di

$a = 1$ nel vettore x_{n+1} diventando

$$x_{n+1} = x_n(\mathrm{Id} \oplus \mathbf{L}^1) = (1,1,0,1) \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} = (0,1,1,1). \tag{42}$$

Questa operazione, per quanto riguarda il linguaggio di programmazione C, può essere semplicemente svolta tramite due operazioni di bitwise. Dal punto di vista del codice, l'equazione (41) risulta molto semplice `x = x ^ (x<<a)`. In letteratura viene consigliato di usare quattro step di shift. L'algoritmo implementato è quindi

$$x_{n+1} = x_n(\mathrm{Id} \oplus \mathbf{L}^a)(\mathrm{Id} \oplus \mathbf{R}^b)(\mathrm{Id} \oplus \mathbf{L}^c)(\mathrm{Id} \oplus \mathbf{R}^d) \tag{43}$$

dove $\mathbf{R}$ è la matrice di *spostamento a destra* ed è definita come $\mathbf{R} = \mathbf{L}^T$. I parametri a, b, c, d sono le posizioni di shift rispettivamente a sinistra, destra, sinistra e ancora a destra. E' dimostrato che per valori opportuni dei parametri a, b, c, d[5] si hanno sequenze di numeri pseudo-random con periodo di $2^w - 1$. In questo caso $w = 128$ quindi la catena di numeri è molto lunga ed è adeguata per essere utilizzata in simulazioni lunghe. La generazione in parallelo di numeri pseudo-random, avviene tramite l'allocazione in memoria di quattro vettori contenenti i seed iniziali. L'i-esimo nodo di calcolo, per creare i 128 bit iniziali, prende gli i-esimi elementi dei quattro vettori, costruisce la prima trasformazione del tipo in equazione (43) e genera il numero x_n. I vettori usati per creare gli iniziali 128 bit sono generati con un LCG. Per garantire la randomicità dello stream di numeri sono stati usati due test. Test del χ^2 (Tabella 1) e lo **squeeze test**[6]. Per il test dei χ^2 sono stati usati $k - 1 = 9$ gradi di libertà. Si calcola

$$D = \sum_{i=1}^{k} \frac{(o_i - e_i)^2}{e_i} \tag{44}$$

dove o_i sono le frequenze osservate ed e_i quelle di aspettazione per l'i-esima cella dell'istogramma e lo si confronta con $\chi^2_{[1-\alpha,1-k]}$ dove α è il livello di confidenza. Se $D < \chi^2_{[1-\alpha,1-k]}$ allora il test è passato. Nel caso in studio si ha $\chi^2_{[0,9\,,\,9]} = 14,68$ mentre quello calcolato risulta essere $D = 8,2156$. Il test risulta quindi passato. Il secondo test di randomicità usato fa parte della suite diehard [6]. Si parte dal valore iniziale $k = 2^{31}$ e si esegue una successione di j moltiplicazioni con numeri random $r_j \in [0,1)$ fino a quando il numero iniziale viene compresso a $k = 1$. Effettuate le prime j moltiplicazioni necessarie per la compressione, usando i primi j numeri random, ci si sposta nella successione di numeri da analizzare riapplicando lo stesso procedimento con i numeri random da $j + 1$. In questo modo ho

[5] Marsaglia [6] nel suo articolo consiglia $a = 20$, $b = 11$, $c = 27$, $d = 6$

Range	Aspettazione	Osservato	χ^2
$0,0-0,1$	10000	10044	$0,193600$
$0,1-0,2$	10000	9927	$0,532900$
$0,2-0,3$	10000	9895	$1,102500$
$0,3-0,4$	10000	10141	$1,988100$
$0,4-0,5$	10000	10117	$1,368900$
$0,5-0,6$	10000	10057	$0,324900$
$0,6-0,7$	10000	9943	$0,324900$
$0,7-0,8$	10000	9869	$1,716100$
$0,8-0,9$	10000	9946	$0,291600$
$0,9-1,0$	10000	10061	$0,372100$
Totale:			$8,215600$

Tabella 1: Valori di χ^2 per un set di 10^5 numeri random.

tanti valori di j che identificano il numero necessario di moltiplicazioni da usare (nelle varie parti della successione di dati) per contrarre il numero 2^{31} all'unità. Se la distribuzione dei numeri pseudo-random è la stessa per ogni sottosuccessione di numeri pseudo-random le j moltiplicazioni necessarie per la compressione saranno le stesse (entro una certa sigma). Questo indica che la velocità, con cui viene compresso il valore iniziale, è la stessa in ogni parte della successione di numeri pseudo-random. Il fatto che all'interno di una cella,[6] i numeri siano distribuiti in modo uniforme è garantito dal passaggio del test χ^2 precedentemente descritto. Nell'istogramma in figura 6 è rappresentata la distribuzione dei valori di j. Il diagramma è fatto con 17 bins quindi i gradi di libertà sono 14 in quanto abbiamo tre vincoli dati da $\sum_k N_k = N_{tot}$, il calcolo della media e della deviazione standard. Il χ^2 corrispondente a questo numero di gradi di libertà è di 21,06. Quello che si ricava dal fit gaussiano dell'istogramma è 14,08. Anche in questo caso il calcolato è minore di quello teorico. Il test risulta passato.

3.5 Descrizione del codice

L'obiettivo principale di questo lavoro è quello di scrivere un codice per simulare un reticolo di Ising in CUDA-C[7] e ottimizzarlo in modo tale che i tempi di esecuzione siano significativamente inferiori di quelli su CPU. Per raggiungere questo

[6] In questo caso identifico col termine *cella* quella parte dello stram di dati composta da un numero sufficiente di numeri pseudo-random necessari per comprimere il valore 2^{31} all'unità.

[7] Linguaggio di programmazione molto simile al C, proposto da NVIDIA, per programare sulle loro schede grafiche

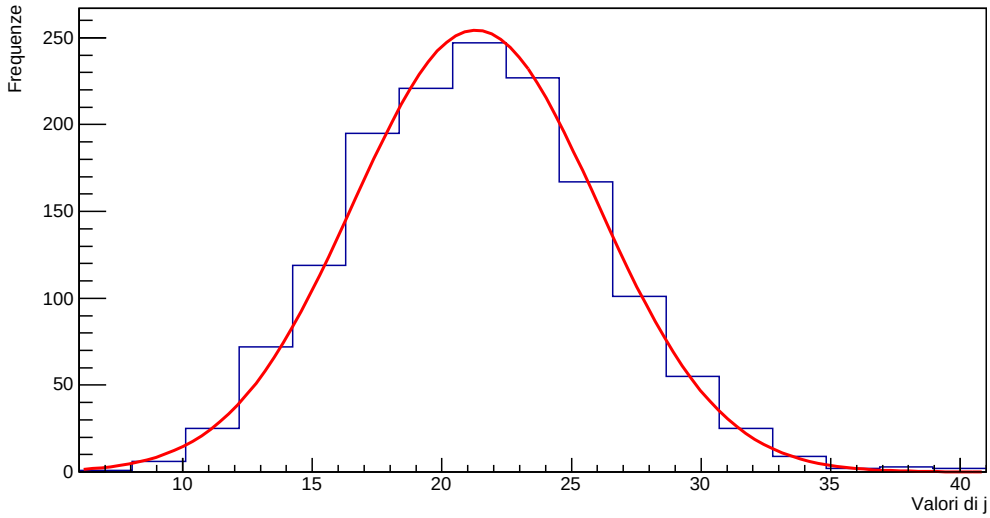


Figura 6: Istogramma rappresentate la distribuzione dei valori j, cioè il numero di moltiplicazioni per numeri random successivi in grado di contrarre 2^{31} a 1.

obiettivo è necessario, come anticipato nelle precedenti sezioni, prestare attenzione all'architettura del codice e alla distribuzione dei dati in memoria. Un uso errato dei vari livelli di memoria compromette in modo sensibile le tempistiche di esecuzione della simulazione. A tale scopo vengono presentate e descritte, in modo esteso, le parti di codice principali del programma.

Per capire come opera il kernel di aggiornamento siti è opportuno analizzare le variabili usate in fase di lancio. Il valore assunto dalle variabili `grid` e `block` è fondamentale. Queste due variabili sono di tipo `dim3` e rappresentano il modo in cui viene organizzato il funzionamento della scheda video. Il parametro `grid` indica in quanti blocchi è divisa la scheda, mentre il parametro `block` indica quanti thread appartengono ad un blocco. Il tipo di variabile `dim3`, permette di definire le variabili pensandole tridimensionali. Se viene definita solo una dimensione, le altre due vengono di default settate a 1. Nel caso in studio, per aiutare la costruizione della doppia scacchiera è stata scelta la seguente definizione delle variabili:

```
1 dim3 grid(GRIDL, GRIDL/2);
2 dim3 block(BLOCKL, BLOCKL/2);
```

il parametro `GRIDL` è definito come la dimensione del lato del reticolo diviso il numero di threads per ogni lato. Definendo `grid.y` come la metà dei blocchi, questo permette di posizionarsi nella cella del reticolo di interesse usando lo stesso kernel

con dei parametri di offset. Per effettuare l'aggiornamento completo di tutti i siti reticolari, il kernel viene lanciato due volte. Questo è dovuto alla strategia di parallelizzazione che si è voluta adottare. Come descritto nella sezione precedente, il reticolo di Ising viene diviso in due tipi di tassellature. Una tassellatura a size grande e una a size fine, in modo da creare una *doppia scacchiera*. I due kernel si occupano, reciprocamente, di aggiornare prima i blocchi pari della scacchiera grossa e successivamente, quelli dispari. Questo avviene lanciando due volte lo stesso tipo di codice con un parametro di offset. Nel primo lancio, l'ultimo parametro che viene passato al kernel è 0, questo fa si che vengano aggiornati i blocchi grandi pari.

```
kernel<<<grid,block>>>(sD,a_d,b_d,c_d,d_d,0);
kernel<<<grid,block>>>(sD,a_d,b_d,c_d,d_d,1);
```

Rilanciandolo una seconda volta, con un parametro di offset pari a 1, il kernel si occupa di aggiornare i blocchi grandi dispari. Ogni blocco viene aggiornato sfruttando questa volta una scacchiera a tassellatura fine (a livello di singolo spin). Ad ogni singolo thread viene affidato l'aggiornamento di un singolo sito reticolare. I siti reticolari vengono aggiornati, all'interno del kernel, in parallelo a scacchiera. La figura 7 rappresenta in modo grafico quello che avviene durante il lancio dei due kernel. I migliori risultati, in termini di tempi di esecuzione, si ottengono allocando il reticolo non sulla memoria globale ma sulla memoria di tipo *shared* che è molto più veloce sia in lettura che scrittura. Il problema della memoria shared consiste nel fatto che i banchi di memoria sono propri del blocco a cui appartengono. Banchi di memoria shared, appartenenti a diversi blocchi, non possono comunicare tra loro. Risulta quindi necessario elaborare delle condizioni al contorno per i blocchetti di tipo shared per rendere possibile la comunciazione tra due blocchi reticolari primi vicini. Ogni blocco shared, a meno del suo bordo, viene riempito con una scacchiera fine nel seguente modo:

```
//Indica lo shift di blocchi nella dimensione X
unsigned int Xoffset = blockIdx.x*BLOCKL;
//Indica lo shift di blocchi nella dimensione Y
unsigned int Yoffset = (2*blockIdx.y+(blockIdx.x+offset)
   %2)*BLOCKL;

/*Definizione del sottoreticolo con bordo nella memoria
  shared */
__shared__ int sS[(BLOCKL+2)*(BLOCKL+2)];

/*Se non sono sul bordo*/
```

```
11 sS[(2*threadIdx.y+1)*(BLOCKL+2)+threadIdx.x+1] =
12  s[(Yoffset+2*threadIdx.y)*L+(Xoffset+threadIdx.x)];
13
14 sS[(2*threadIdx.y+2)*(BLOCKL+2)+threadIdx.x+1] =
15  s[(Yoffset+2*threadIdx.y+1)*L+(Xoffset+threadIdx.x)];
```

La matrice sS è un'area di memoria ti tipo shared dove vengono memorizzati i vari sottoreticoli. La matrice s invece è un'area di memoria di tipo global dove è presente il reticolo totale. La variabile BLOCKL indica il numero di threads per blocco. A riga 8 viene allocata nella memoria shared, una matrice con due colonne e due righe in più. Con delle opportune condizione al bordo è possibile memorizzare nella memoria shared dello stesso blocco, gli spin da aggiornare più un bordo composto dagli spin primi vicini appartenenti ai blocchi grandi primi vicini. Da riga 11 a riga 15 si riempiono a scacchiera le parti interne della matrice nella memoria shared. L'`Yoffset` definito come `Yoffset=(2blockIdx.y+(blockIdx.x+offset)%2)*BLOCKL` è in funzione al valore di `offset` con cui è stato lanciato il kernel. Questo fa in modo che vengano presi solo i blocchi grandi pari nel caso di `offset=0` e dispari nel caso di `offset=1`. Nel momento in cui viene riempito il bordo della matrice nella memoria shared, quest'operazione viene svolta con accorgimenti opportuni. Le parte di codice che si occupa di questo è la seguente:

```
1  /*riempio il bordo superiore della sS*/
2  if(threadIdx.y == 0)
3   if(Yoffset == 0){
4    sS[threadIdx.x+1] = s[(L-1)*L+Xoffset+threadIdx.x];
5   else
6    sS[threadIdx.x+1] = s[(Yoffset-1)*L+Xoffset+threadIdx.x
      ];
7   }
8  /*riempio il bordo inferiore della sS*/
9  if(threadIdx.y == (BLOCKL/2)-1)
10  if(Yoffset == L-BLOCKL)
11   sS[(BLOCKL+1)*(BLOCKL+2)+(threadIdx.x+1)] = s[Xoffset+
      threadIdx.x];
12  else
13   sS[(BLOCKL+1)*(BLOCKL+2)+(threadIdx.x+1)] = s[(Yoffset+
      BLOCKL)*L+Xoffset+threadIdx.x];
14
15 /*riempio il bordo sinistro dell sS*/
```

```
16 if(threadIdx.x == 0){
17  if(blockIdx.x == 0){
18   sS[(2*threadIdx.y+1)*(BLOCKL+2)] = s[(Yoffset+2*
      threadIdx.y)*L+(L-1)];
19   sS[(2*threadIdx.y+2)*(BLOCKL+2)] = s[(Yoffset+2*
      threadIdx.y+1)*L+(L-1)];
20  }
21  else{
22   sS[(2*threadIdx.y+1)*(BLOCKL+2)] = s[(Yoffset+2*
      threadIdx.y)*L+(Xoffset-1)];
23   sS[(2*threadIdx.y+2)*(BLOCKL+2)] = s[(Yoffset+2*
      threadIdx.y+1)*L+(Xoffset-1)];
24  }
25 }
26 /*riempio il bordo destro della sS*/
27 if(threadIdx.x == BLOCKL-1){
28  if(blockIdx.x == GRIDL-1){
29   sS[(2*threadIdx.y+1)*(BLOCKL+2)+BLOCKL+1] = s[(Yoffset
      +2*threadIdx.y)*L];
30   sS[(2*threadIdx.y+2)*(BLOCKL+2)+BLOCKL+1] = s[(Yoffset
      +2*threadIdx.y+1)*L];
31  }
32  else{
33   sS[(2*threadIdx.y+1)*(BLOCKL+2)+BLOCKL+1] = s[(Yoffset
      +2*threadIdx.y)*L+Xoffset+BLOCKL];
34   sS[(2*threadIdx.y+2)*(BLOCKL+2)+BLOCKL+1] = s[(Yoffset
      +2*threadIdx.y+1)*L+Xoffset+BLOCKL];
35  }
36 }
37 __syncthreads();
```

La prima condizione, che compare a riga 2, riguarda lo riempimento del bordo superiore. Se il valore di `threadIdx.y` è pari a zero, allora vuol dire che si sta considerando la prima riga della matrice nella memoria shared. In funzione al valore di `Yoffset` il riempimento del bordo superiore varia. Nel caso in cui `Yoffset=0` questo indica che si sta considerando il primo blocco del reticolo totale. La condizione da usare è quella di periodicità del reticolo di Ising. Se L è la dimensione del reticolo totale, i siti da percorrere per raggiungere l'ultima riga

del reticolo saranno $L(L-1)$.[8] Se `Yoffset` è diverso da zero si sta considerando un blocco generico all'interno del reticolo. Il bordo superiore sarà quindi riempito con i valori che si trovano sulla riga `L(Yoffset-1)`.[9] A riga 9 è presente la condizione di riempimento del bordo inferiore. Il ragionamento è identico a quello precedente. Nel momento in cui `threadIdx.y==BLOCK/2-1`,[10] questo indica che si sta considerando l'ultima riga della matice nella memoria shared. Anche in questo caso il bordo inferiore, viene riempito in funzione al tipo di blocco considerato. Se il blocco ha `Yoffset=L-BLOCKL` vuol dire che devono essere posizionati gli elementi appartenenti all'ultimo blocco grande del reticolo e la condizione di riempimento della matrice nella memoria shared sarà la condizione periodica sul reticolo di Ising. Se `Yoffset` è diverso da zero il bordo inferiore della matrice nella memoria shared viene riempito con i valori della prima riga del blocco inferiore primo vicino. Lo stesso procedimento viene usato nelle righe da 16 fino a 36 per definire le condizioni al bordo laterali della matrice. Il tutto si conclude con un __syncthreads(). Questa istruzione indica che prima di eseguire l'aggiornamento degli spin, deve prima completarsi l'operazione di posizionamento del reticolo sulla memoria shared. A questo punto è possibile passare all'aggiornamento dei siti. Anche in questo caso, l'aggiornamento avverrà in due step, prima tutti gli spin relativi ai threads pari, successivamente quelli relativi ai threads dispari. Il codice che esegue l'update dei siti è il seguente:

```
1 #define sS(x,y) sS[(y+1)*(BLOCKL+2)+x+1]
2 unsigned int *aa = &a[(blockIdx.y*GRIDL + blockIdx.x)*
    THREADS+n];
3 unsigned int *bb = &b[(blockIdx.y*GRIDL + blockIdx.x)*
    THREADS+n];
4 unsigned int *cc = &c[(blockIdx.y*GRIDL + blockIdx.x)*
    THREADS+n];
5 unsigned int *dd = &d[(blockIdx.y*GRIDL + blockIdx.x)*
    THREADS+n];
```

[8] Questo è dovuto al fatto che le matrici in realtà sono identificate da indici monidimensiosali. Per richiamare l'elemento della riga i e colonna j di una matrice $m \times m$, l'indice unico da usare sarà $im + j$

[9] In questo caso è bene ricordare che il numero di blocchi in y è la metà del numero di blocchi in x. Il posizionamento degli elementi del reticolo totale appartenenti al j-esimo blocco grande, vengono elaborati dai threads appartenenti all'indice di blocco su scheda grafica pari a `blockIdx.y=j/2`

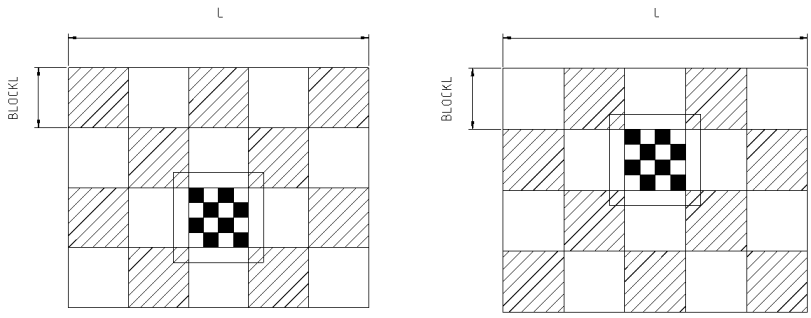
[10] Questo indica che siamo all'ultima riga della matrice nella memoria nella shared memory perchè i thread in y come i blocchi sono la metà di quelli in x. Per costruire la scacchiera si procede di passo `2*threadIdx.y+threadIdx.x`

```
unsigned int x = threadIdx.x;
unsigned int y1= 2*threadIdx.y+(threadIdx.x%2);
unsigned int y2= 2*threadIdx.y+((threadIdx.x+1)%2);
int ie=0;

int ide = sS(x,y1)*(sS(x-1,y1)+sS(x+1,y1)+sS(x,y1+1)+sS(x,
    y1-1));
if(xorshift(aa, bb, cc, dd) < tex1Dfetch(boltzT, ide+2*DIM
    )){
 sS(x,y1) = -sS(x,y1);
 ie -= 2*ide;
}
__syncthreads();

ide = sS(x,y2)*(sS(x-1,y2)+sS(x+1,y2)+sS(x,y2+1)+sS(x,y2
    -1));
if(xorshift(aa, bb, cc, dd) < tex1Dfetch(boltzT, ide+2*DIM
    )){
 sS(x,y2) = -sS(x,y2);
 ie -= 2*ide;
}
___syncthreads();
s[(Yoffset+2*threadIdx.y)*L+Xoffset+threadIdx.x] = sS[(2*
    threadIdx.y+1)*(BLOCKL+2)+threadIdx.x+1];
s[(Yoffset+2*threadIdx.y+1)*L+Xoffset+threadIdx.x] = sS
    [(2*threadIdx.y+2)*(BLOCKL+2)+threadIdx.x+1];
a[(blockIdx.y*GRIDL+blockIdx.x)*THREADS+n] = *aa;
b[(blockIdx.y*GRIDL+blockIdx.x)*THREADS+n] = *bb;
c[(blockIdx.y*GRIDL+blockIdx.x)*THREADS+n] = *cc;
d[(blockIdx.y*GRIDL+blockIdx.x)*THREADS+n] = *dd;
```

Come si può notare dalle definizioni delle variabili $y1$ e $y2$, per muoversi a scacchiera tra gli elementi della matrice sS è possibile farlo in modo molto semplice con un parametro di offset (il $+1$ nella definizione di $y2$) e una divisione di modulo due. Una volta effettuato l'update di tutti i siti reticolari nella shared memory, tutto il reticolo viene riportato in modo coalescente nella memoria globale andando a sovrascrivere il vecchio reticolo. Alla fine di questo step si ottiene il reticolo su memoria globale completamente aggiornato. A questo punto inizia un nuovo step. Il reticolo viene nuovamente diviso in sottoreticoli allocati in shared memory e viene nuovamente fatta l'operazione di update dei vari siti.



(a) offset= 0 (b) offset= 1

Figura 7: Organizzazione della doppia scacchiera. Con i tratteggi sono messi in evidenza i blocchi che vengono aggiornati.

3.6 Tempi di Esecuzione

Per confrontare le prestazioni del programma su GPU si valutano i tempi (relativamente su GPU e CPU) che intercorrono tra una proposta di flip e l'altra. I programmi su GPU e CPU usano lo stesso generatore di numeri random in questo modo, a parità di algoritmo, per ogni spin il risultato che si ottiene è visibile in Figura 8. Il grafico rappresentas il tempo medio che intercorre tra una proposta di spin flip e la successiva, in funzione al size del reticolo stesso. Da questo grafico possiamo notare alcuni particolari di interesse:

1. Su GPU il tempo per spin tra una proposta e l'altra è quasi sempre inferiore a quello su CPU, tranne che per reticoli molto piccoli;

2. Su GPU il tempo per spin decade molto velocemente con l'ingrandirsi del reticolo fino a raggiungere un valore approssimativamente stabile, quando tutti i threads della GPU sono utilizzati;

3. Su CPU il tempo per spin rimane approssimativamente costante in funzione al size del reticolo.

Normalizzare il tempo di esecuzione al numero di spin equivale ad avere un'informazione diretta sull'efficienza della singola unità di calcolo del mio processo parallelo. Dal grafico si nota che per un reticolo di dimensione 1024×1024 le proposte di flip, per singola unità di calcolo, su CPU avvengono intervallate da un tempo $t_{CPU} = 2,195$ ns, mentre su scheda video $t_{GPU} = 0,037$ ns. Questo è una dato importante che sottolinea come questi tipi di problemi si prestino bene ad essere implementati su scheda video. Il guadagno, su singola unità di calcolo è di quasi due ordini di grandezza. Considerando il fatto che le CPU moderne possono mettere a disposizione solo alcune decine di unità di calcolo intensivo

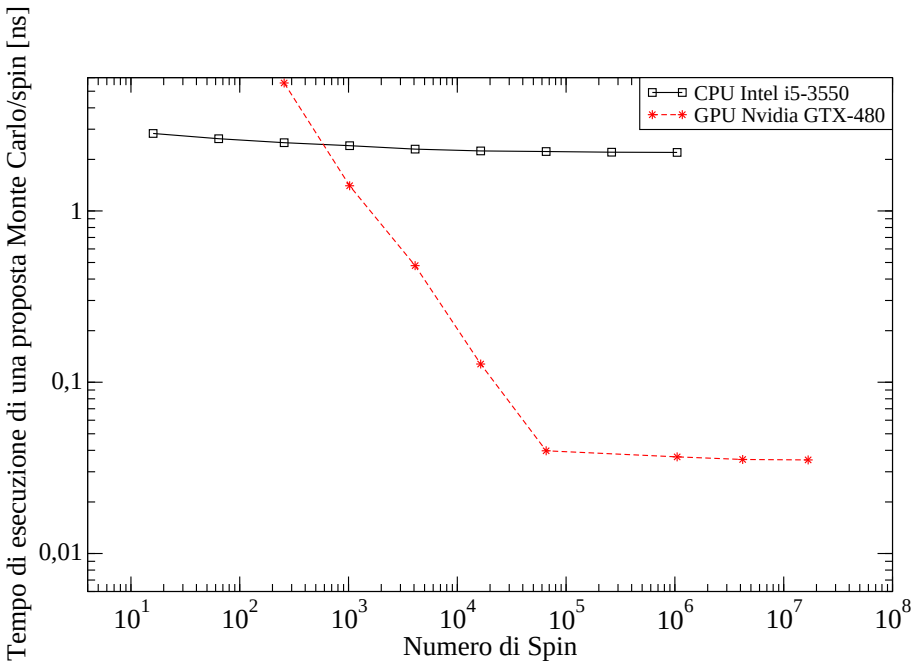


Figura 8: Tempo impiegato per proporre una mossa Metropolis per ogni spin, in funzione al size del reticolo.

ci vorrebbe un cluster di 10 CPU solo per raggiungere la potenza di calcolo di una singola unità di calcolo delle schede video. Considerando che in parallelo, la scheda video utilizzata (Nvidia-GTX480), può mettere a disposizione 480 unità di calcolo, è palese il fatto che il guadagno, in termini di tempo totale di esecuzione è immenso. Ci vorrebbe un cluster composto da 4800unità di calcolo per raggiungere dei tempi di esecuzione totale che si avvicinino a quelli delle GPU. Il motivo per cui il grafico dei tempi, per le GPU, varia molto col size è dovuto al fatto che usando un reticolo piccolo si hanno solo pochissimi threads che lavorano. Con l'aumentare del size la scheda si riempie e satura raggiungendo il massimo dell'ottimizzazione.

4 Risultati delle simulazioni

La simulazione ha inizio con tutti gli spin del reticolo inizializzati a $+1$. In Figura 9 è presentato il reticolo iniziale. Un pixel nero indicana valore dello spin $+1$, un pixel bianco indica invece valore dello spin -1.

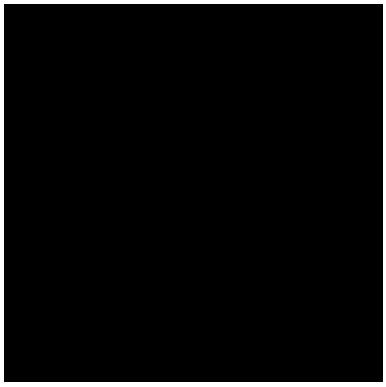
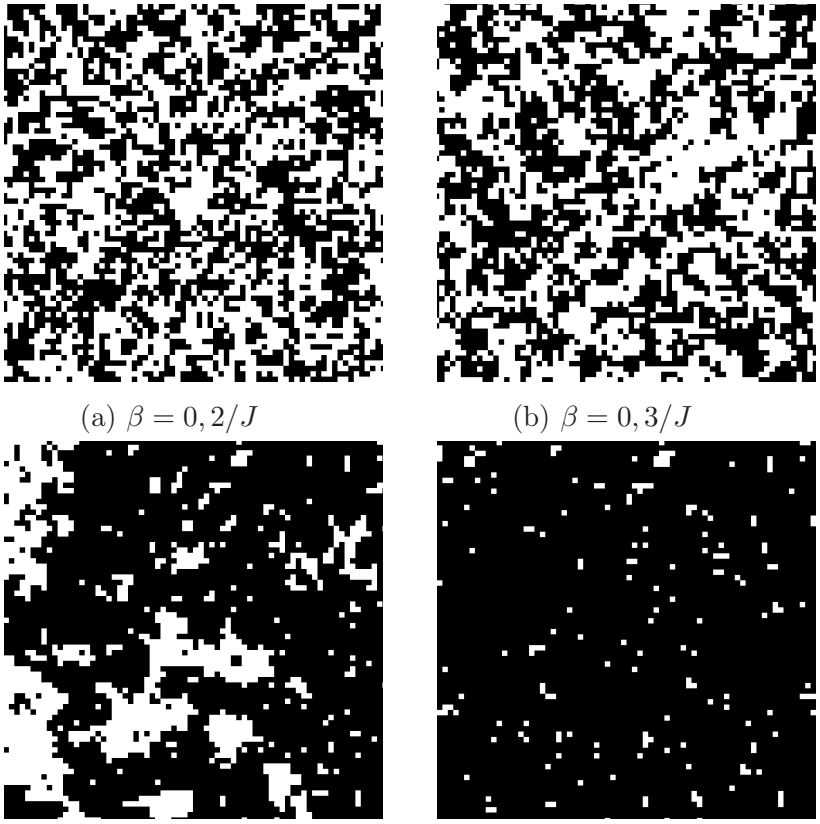


Figura 9: Reticolo iniziale settato con tutti gli spin up ($s_i = +1$)

Prima di calcolare le medie il sistema viene fatto termalizzare con 100 step Monte Carlo per spin. Confrontando Figura 10, rappresenta un esempio di stato termine della termalizzazione e Figura 11 che riporta uno stato alla fine della simulazione, è possibile notare che per diversi valori di β, le configurazioni di spin sono qualitativamente simili. Questo indica che gli steps di termalizzazione usati sono sufficienti per termalizzare il sistema.

Dopo la fase di termalizzazione vengono calcolate le medie in particolare i valori di magnetizzazione media per spin e calore specifico per spin (dato dalle fluttuazioni dell'energia media). Il confronto viene fatto per diverse taglie per verificare l'avvicinamento al limite termodinamico. In Figura 12 sono riportati i valori di magnetizzazione media per spin. Come si può notare i punti calcolati si avvicinano alla soluzione esatta del modello di Onsager per il reticolo 2D infinito.

In Figura 13 è rappresentato il calore specifico per diversi size del reticolo. In Figura 14 è rappresentato il confronto tra la curva di calore specifico per un reticolo di size 4096×4096 e la soluzione di Onsager infinito dimensionale. Come si può notare la curva di calore specifico simulata segue abbastanza fedelmente la soluzione di Onsager per reticolo infinito dimensionale. I punti, per valori più elevati di β, sono più dispersi. La motivazione di questo comportamento è presente nel grafico in Figura 15 per β molto grandi il numero di mosse accettate cala in modo esponenziale, questo suggerische che per β grande è il caso di fare più iterazioni. In questo caso, per $\beta \geq 0,5J^{-1}$ il numero di passi Monte Carlo è stato aumentato di due ordine di grandezza in modo tale da tener contenuto l'errore sui dati.



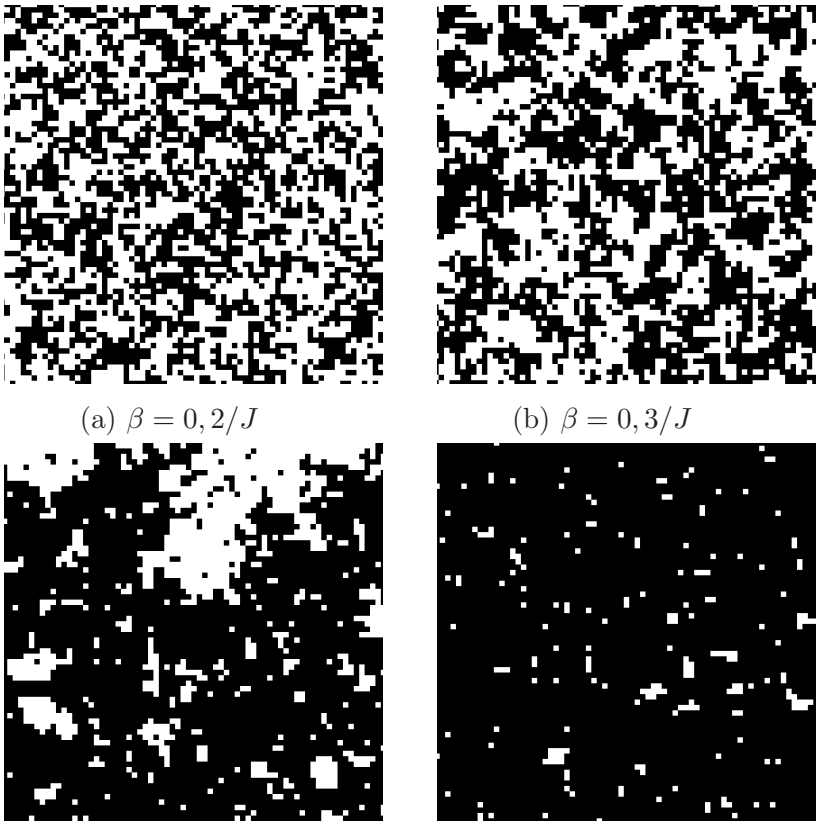
(a) $\beta = 0,2/J$ (b) $\beta = 0,3/J$

(c) $\beta = 0,44/J \sim \beta_C$ (d) $\beta = 0,5/J$

Figura 10: Porzione 70×70 di un reticolo 1024×1024 al termine delle 100 iterazioni di termalizzazione.

5 Conclusioni

Vista Figura 8, il modello di Ising si presta ottimamente ad essere parallelizzato in modo fine. Nel momento in cui vengono saturate le unità di calcolo della scheda grafica il guadagno di prestazioni è di circa due ordini di grandezza rispetto allo stesso algoritmo su CPU. Come è possibile notare in figura 12 e figura 13, i dati ottenuti dalle simulazioni numeriche sono compatibili con la soluzione del modello di Ising bidimensionale a reticolo infinito. Questo indica che le dimensioni dei reticoli che possono venire simulati con le GPU sono tali da approssimare il limite termodinamico molto da vicino.



(a) $\beta = 0,2/J$ (b) $\beta = 0,3/J$

(c) $\beta = 0,44/J \sim \beta_C$ (d) $\beta = 0,5/J$

Figura 11: Porzione 70×70 di un reticolo 1024×1024 al termine delle 10^6 iterazioni a diversi valori di β.

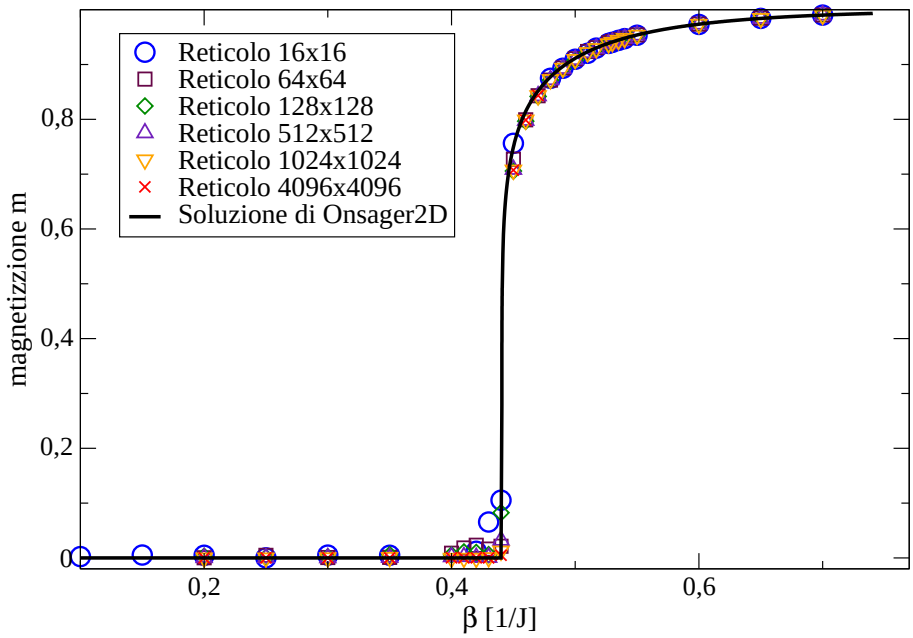


Figura 12: Magnetizzazione, per diversi valori di size, in funzione alla temperatura inversa β

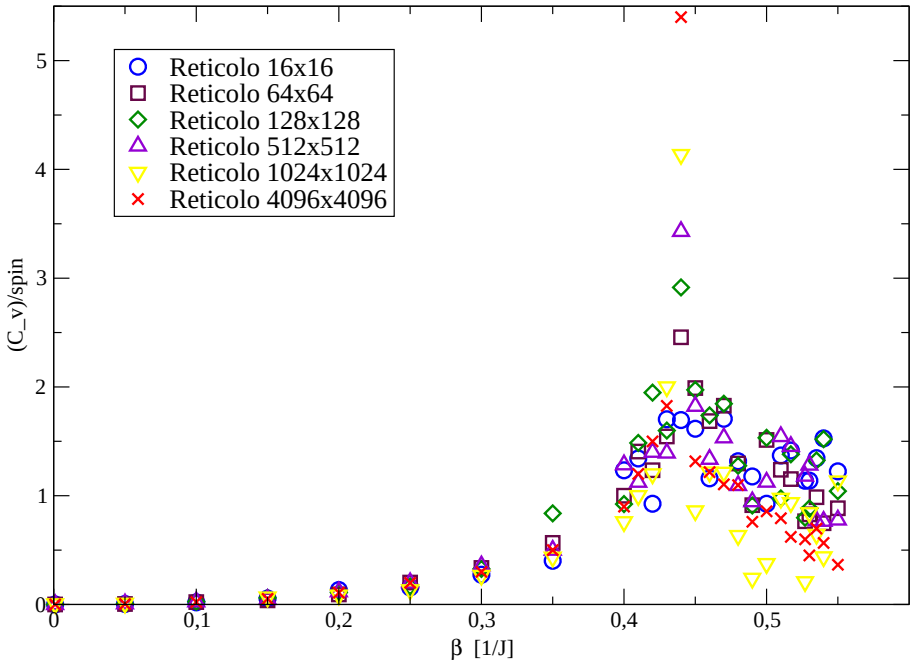


Figura 13: Calore specifico in funzione di β, per diverse taglie del reticolo di Ising. Le barre d'errore statistico sono omesse per chiarezza.

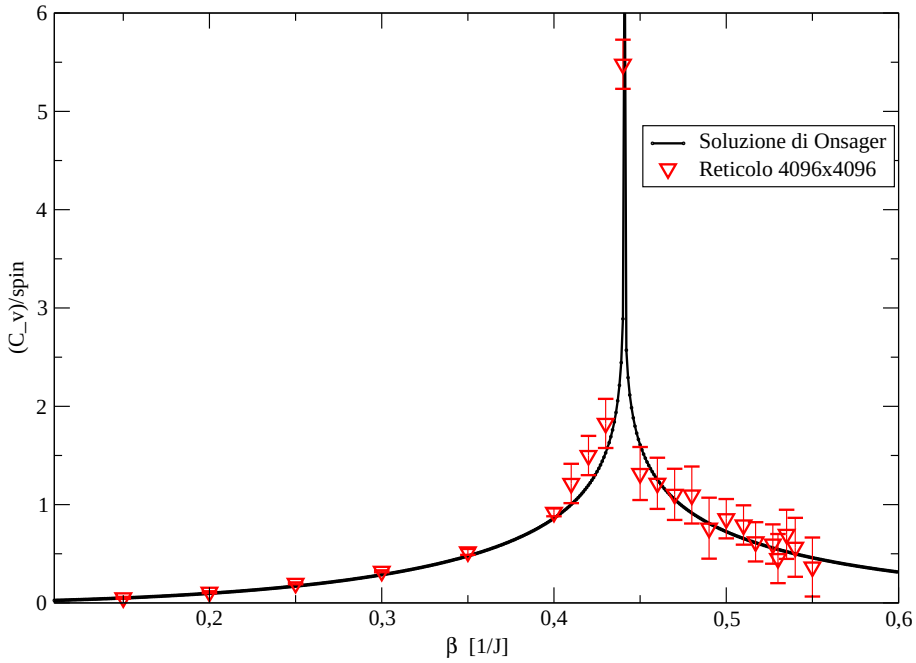


Figura 14: Confronto tra il calore specifico funzione di β calcolato dalle simulazioni per un reticolo 4096×4096 e la soluzione di Onsager per il reticolo infinito.

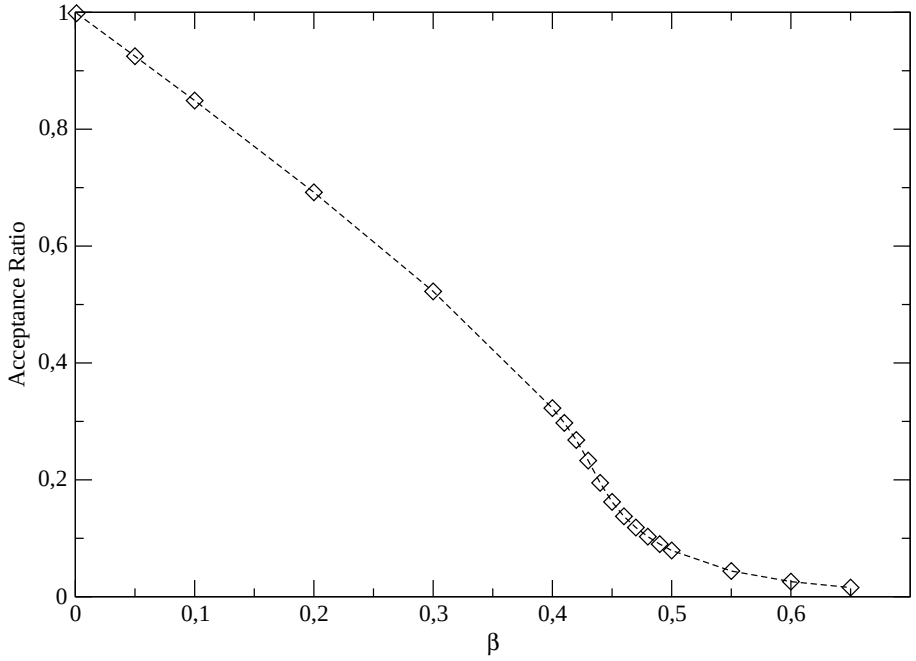


Figura 15: Percentuale di mosse accettate, in funzione di β per un reticolo 1024×1024

Riferimenti bibliografici

[1] M.E.J. Newman and G.T. Barkema *Monte Carlo Methods in Statistical Physics* (Claredon Press, Oxford, 2001).

[2] K. Huang, *Statistical Mechanics*, 2nd Ed. (Wiley, 1987).

[3] C. Itzkyson and J-M. Drouffe *Statistical Field Theory, Vol 1* (Cambridge University Press, Cambridge, 1989).

[4] C. Itzkyson and J-M. Drouffe *Statistical Field Theory, Vol 2* (Cambridge University Press, Cambridge, 1989).

[5] R.J. Baxter, *Exactly Solved Models in Statistical Mechanics* (Academic Press Limited, Londra, 1989).

[6] G. Marsaglia, J. Stat. Softw., **8**, 1 (2003).

[7] M. Manssen M. Weigel and A.K. Hartmann, Europ. Phys. J. Spec. Top., **210**, 55-71 (2012)

[8] E. Ising, Z. Phys., **31**, 253-258 (1925)

[9] L. Onsager Phys. Rev., **65** , 117-149 (1944)

[10] N. Metropolis A.W. Rosenbluth M.N. Rosenbluth A.H. Teller and E.Teller, J. of Chem. Phys. **21** 1087-1092 (1953)