Implementazione in CUDA dell'algoritmo di Metropolis per il modello di Ising in 2 dimensioni

1 Introduzione

In questo esercizio studiamo gli stati di equilibrio del modello di Ising 2-dimensionale, e verifichiamo la presenza di una transizione di fase a temperature finita.

Lo studio è effettuato tramite un algoritmo numerico che esplora l'*ensemble* canonico del sistema.

L'algoritmo è implementato prima in modo seriale in C++ e eseguito su CPU, poi in modo parallelo in CUDA, e eseguito sfruttando anche una GPU. Lo scopo dell'esercizio è di verificare e di quantificare il miglioramento della performance che deriva dalla parallelizzazione dell'algoritmo.

2 Il modello di Ising

Il modello è costituito da un insieme di variabili binarie σ_i, dette "spin", distribuite su un reticolo quadrato bidimensionale di lato a, che possono assumere i valori $+1$ o -1. Le loro interazioni sono descritte da un'Hamiltoniana della forma

$$H = \sum_{\langle i,j \rangle} J\sigma_i\sigma_j + \sum_i h\sigma_i \tag{1}$$

dove $\langle i, j\rangle$ denota una somma su i, j primi vicini. In questo esercizio, considereremo il caso in cui il campo esterno h è nullo, quindi potremo fissare il valore di J a 1 senza perdita di generalità. Per le simulazioni, considereremo reticoli finiti di lato L, aggiungendo poi un'ulteriore "cornice" di spin che fungono da condizioni al contorno, contribuendo al calcolo dell'energia ma senza che il loro valore venga mai cambiato.

3 L'algoritmo di Metropolis

Gli stati di equilibrio a cui siamo interessati sono dati dalla distribuzione di probabilità canonica, in cui ad ogni stato k è assegnata una probabilità

$$P(k) = \frac{e^{-\beta E(k)}}{Z} \tag{2}$$

dove $E(k)$ è l'energia dello stato, $\beta = \frac{1}{k_b T}$ e Z è la funzione di partizione.

Per campionare questa distribuzione di probabilità, usiamo l'algoritmo di Metropolis. Questo algoritmo specifica una regola di transizione tra vari stati del sistema, che, applicata ripetutamente, viene usata per generare un *random walk* nello spazio degli stati, in cui la

distribuzione degli stati visitati converge alla distribuzione di probabilità desiderata nel limite di un *random walk* di lunghezza infinita.

La regola di transizione è la seguente: il passaggio da uno stato A a uno stato B viene sempre accettato se $P(B) < P(A)$, mentre, se $P(B) > P(A)$, viene accettato con una probabilità $P(A)/P(B)$. Nel caso di una distribuzione di probabilità canonica, il rapporto $P(A)/P(B)$ si semplifica e dipende solo dalla differenza di energia ΔE degli stati. Nell'applicazione al modello di Ising, questo fatto rende conveniente considerare transizioni in cui viene cambiato il valore di un solo spin σ_0: infatti in questo caso la differenza di energia dipende solo dal valore dei 4 spin adiacenti a σ_0 e non dal resto del sistema.

Naturalmente l'uso di transizioni di un singolo spin implica che gli stati che costituiscono il *random walk* sono correlati tra di loro. Questo è un problema molto frequente nelle applicazioni dell'algoritmo di Metropolis, e va risolto assicurandosi che sia stato effettuato un numero sufficiente di transizioni prima di "misurare" le proprietà del sistema.

Inoltre, osserviamo se due spin non sono adiacenti, osserviamo che le transizioni che li cambiano sono completamente indipendenti, nel senso che portano allo stesso stato finale indipendentemente dall'ordine in cui vengono eseguite. Questo significa che queste transizioni possono essere eseguite in parallelo.

Date queste considerazioni, scegliamo di seguire per le transizioni uno schema a scacchiera, in cui ad ogni step della simulazione vengono cambiati prima tutti gli spin sulla scacchiera "nera", poi quelli sulla scacchiera "bianca". Inoltre, usiamo un parametro N_{int} che specifica il numero di step che lasciamo passare prima di misurare le quantità fisiche del sistema. Questo è equivalente a comporre la distribuzione finale prendendo uno stato ogni N_{int} invece che con tutti gli stati visitati durante il *random walk*. Fisseremo questo parametro in modo che le misure successive risultino scorrelate tra di loro.

Una volta deciso come applicare l'algoritmo di Metropolis, possiamo dunque usarlo per calcolare i valori attesi di qualsiasi grandezza fisica nell'ensemble canonico. In questo esercizio, ci poniamo l'obiettivo di misurare l'andamento della magnetizzazione M al variare della temperatura T, e delle correlazioni spin-spin $C_{ij} = \langle \sigma_i \sigma_j \rangle$ al variare di T e della distanza r tra σ_i e σ_j. Una "misura" della magnetizzazione corrisponderà alla media sugli stati selezionati da un singolo *random walk* di $M = \sum_i \sigma_i / N$, mentre per misurare $C(r)$, per ogni stato eseguiremo una media tra tutte le coppie di spin σ_i e σ_j che distano r l'uno dall'altro. Per ottenere dei risultati statisticamente rilevanti, poi, ripeteremo la misura N_{rep} volte, e considereremo la media degli N_{rep} risultati come il valore finale di M e di C_{ij}.

4 Implementazione

Abbiamo implementato il modello così descritto prima in C++ e poi in CUDA. La versione in C++ usa lo stesso algoritmo, ma esegue le varie transizioni una dopo l'altra senza nessuna

parallelizzazione.

Riportiamo qua avanti alcuni dettagli non ovvi dell'implementazione.

4.1 Variabili binarie

I dati immagazzinati nella RAM di un computer sono organizzati in modo tale che la lettura e scrittura di dati abbia un limite inferiore di 8 bit di capacità. Corrispondentemente, in tutte le implementazioni di C++ e di CUDA non esistono tipi di variabili con dimensione inferiore a 8 bit. Per il modello di Ising ovviamente ogni spin ha bisogno di un solo bit, dunque usare una variabile diversa per ogni spin non è efficiente.

Per ovviare a questo problema, abbiamo usato uno schema adattato da quello descritto in [2], in cui i vari bit che costituiscono una variabile vengono tutti utilizzati per registrare spin distinti. Attraverso l'uso delle operazioni binarie di C++ `&` (AND), `|` (OR), `^` (XOR), etc., è possibile effettuare alcune operazioni in modo parallelo sui vari spin, diminuendo il tempo di esecuzione, oltre che l'uso di memoria, rispetto a un'implementazione più ingenua.

Gli spin sono organizzati come rappresentato in figura [1]: per ognuno degli N^2 punti del reticolo viene allocata una variabile da N_{bit} bit, dove N_{bit} può essere 8, 16, o 32, che però poi ospita N_{bit} spin che vengono considerati indipendenti. Questo dà luogo a N_{bit} reticoli paralleli, che possono essere studiati simultaneamente e producono N_{bit} risultati indipendenti. Questo diminuisce il numero di volte che è necessario ripetere la simulazione per ottenere N_{rep} misure indipendenti: per esempio, se usiamo $N_{bit} = 32$ e vogliamo ottenere N_{rep} dell'ordine di 100, abbiamo bisogno soltanto di 3 iterazioni.

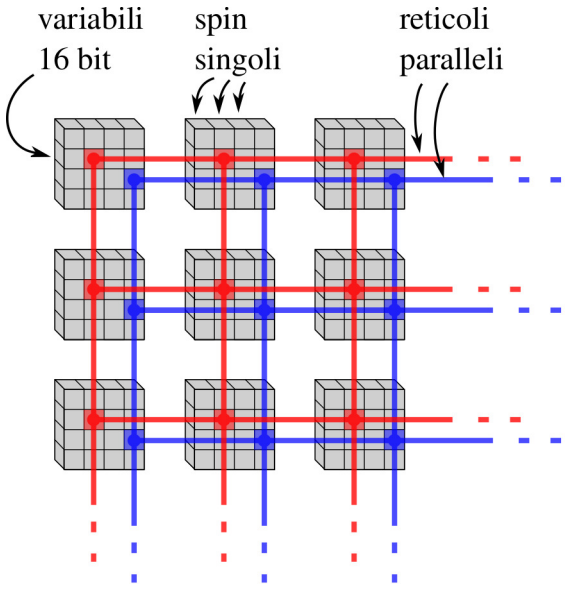

Figura 1: Organizzazione dei bit in N_{bit} reticoli paralleli.

Come abbiamo accennato, per decidere se accettare la transizione che cambia il valore di uno spin, usiamo un algoritmo binario che agisce in parallelo su tutti i bit di una variabile. Consideriamo uno spin σ_0 e i suoi primi vicini $\sigma_i, i = 1...4$. Queste variabili σ hanno valori 0 o 1 invece che $+1$ e -1, in accordo con la loro rappresentazione binaria.

A seconda di quanti tra i $\sigma_i, i = 1...4$ sono uguali o opposti a σ_0, il valore di ΔE che corrisponde al cambiamento di σ_0 può assumere 5 valori discreti: $\Delta E = -8J, -4J, 0, +4J, +8J$. Se $\Delta E < 0$ la transizione è accettata, mentre per $\Delta E = +4, +8J$, viene accettata con una probabilità $P = e^{-8\beta J}, e^{-4\beta J}$.

Definiamo delle variabili binarie $x_i, i = 1...4$ definite da $x_i = \sigma_0 \text{ ^ } \sigma_i$, e $j_i, i = 1...4$ definite da

$$\begin{aligned} j1 &= x1 \text{ \& } x2, \\ j2 &= x1 \text{ ^ } x2, \\ j3 &= x3 \text{ \& } x4, \\ j4 &= x3 \text{ ^ } x4, \end{aligned} \tag{3}$$

che rappresentano le somme con riporto delle x_i a coppie di due, e infine le variabili casuali `exp4`, `exp8` definite da

$$\begin{aligned} \texttt{exp4} = 1 \text{ con } P = e^{-4\beta J}, \ \ \texttt{exp4} = 0 \text{ altrimenti }, \\ \texttt{exp8} = 1 \text{ con } P = e^{-8\beta J}, \ \ \texttt{exp8} = 0 \text{ altrimenti }. \end{aligned} \tag{4}$$

Queste variabili possono essere combinate in una variabile f definita da:

$$f = (j1 \text{ | } j3) \text{ | } (\text{ ~}(j1 \text{ ^ } j3) \text{ \& } (j2 \text{ \& } j4)) \text{ | } ((j2 \text{ | } j4) \text{ \& exp4}) \text{ | } (\texttt{exp8}). \tag{5}$$

f vale 1 se la transizione deve essere accettata e 0 altrimenti. La tabella [1] illustra come l'espressione di f viene determinata a partire dai valori dei σ_i.

4.2 Implementazione parallela in CUDA

Nell'implementazione in CUDA, l'obiettivo è diminuire i tempi di esecuzione svolgendo il numero maggiore possibile di transizioni in parallelo, sfruttando l'elevato numero di *core* della scheda grafica.

Sulla scheda grafica i *thread* sono divisi in blocchi di massimo 1024. Questa divisione corrisponde a una divisione nella memoria disponibile sulla scheda: mentre esiste anche una memoria globale accessibile a tutti i *core*, per ciascun blocco esiste anche una zona di memoria condivisa condivisa con tutti i *thread* del blocco, a cui si può accedere in modo più veloce.

Dato che, per calcolare le energie dei suoi 4 primi vicini più di se stesso, ogni spin deve essere letto 5 volte, risulta conveniente dividere gli spin in blocchi e caricare i valori degli spin sulla memoria condivisa prima di utilizzarli.

Questo vuol dire che in ogni blocco, oltre agli spin che quel blocco deve aggiornare, bisogna caricare anche una "cornice" esterna che faccia da condizione al contorno per il

σ_i	x_i	$\Delta E/J$	$j_1\ j_2$	$j_3\ j_4$	termine in $f(\sigma_i)$
1	1				
101	1 1	-8	1 0	1 0	
1	1				
0	0				
101	1 1	-4	0 1	1 0	(j1 | j3)
1	1				
0	0		0 0	1 0	
001	0 1	0	oppure		~(j1^j3) & (j2&j4)
1	1		0 1	0 1	
0	0				
000	0 0	+4	0 0	0 1	(j2|j4) & exp4
1	1				
0	0				
000	0 0	+8	0 0	0 0	exp8
0	0				

Tabella 1: Illustrazione della forma di f. Nel terzo caso, i j_i possono assumere due configurazioni diverse a seconda di come i gli x_i vengono accoppiati, ma solo la seconda di queste necessita di un termine specifico in f.

blocco. Tuttavia, se questo procedimento viene ripetuto in parallelo per blocchi adiacenti, si possono introdurre delle correlazioni tra le transizioni, dato che il risultato di un *flip* su un blocco dipende dai valori degli spin di quelli dell'altro blocco. Questo diventa ulteriormente problematico perché in CUDA l'ordine dell'esecuzione dei vari blocchi non è garantito.

Per risolvere questo problema, dividiamo anche i blocchi in una "macro-scacchiera" bianca e una nera, e aggiorniamo prima l'una e poi l'altra. Dato che il numero di *thread* che possono essere eseguiti in modo veramente simultaneo è comunque limitato, dell'ordine di 100, e i vari blocchi vengono organizzati internamente in modo seriale dal *block scheduler* interno alla GPU, questo normalmente non comporta una perdita di performance.

5 Risultati

La misura della magnetizzazione riproduce la curva attesa, in cui esiste una temperatura critica finita al di sotto della quale la magnetizzazione si avvicina a -1, mentre al di sopra si

avvicina a 0. Il valore della temperatura critica è qualitativamente compatibile con il valore

$$T_c = \frac{2J}{k_b \ln(1+\sqrt{2})} \approx 2.2692 \frac{J}{k_b} \tag{6}$$

ottenuto dalla soluzione analitica del modello [1]. La curva ottenuta è riportata in figura [2].

Anche l'andamento della correlazione $C_{ij} = \langle \sigma_i \sigma_j \rangle$ è quello atteso. La figura [3] rappresenta i valori di C_{ij} al variare di r per tre temperature diverse. Osserviamo come al di sotto di T_c la correlazione resta sempre vicina a 1 indipendentemente da r, mentre al di sopra decade velocemente a zero con una lunghezza di correlazione finita. Al punto critico $T = T_c$ la correlazione per $r \to \infty$ tende a un valore intermedio lontano sia da 1 che da 0.

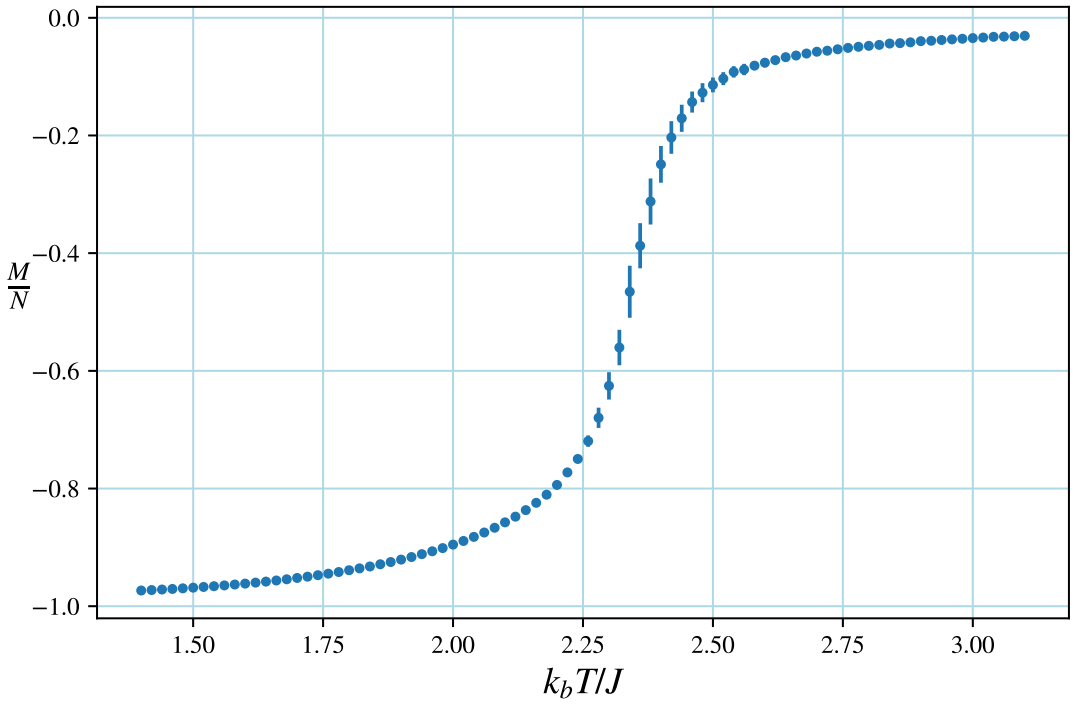

Figura 2: Curva di magnetizzazione. I risultati sono stati ottenuti da simulazioni con parametri: $L = 196, N_{rep} = 96, t_{max} = 500$.

6 Valutazione della performance

La tabella [6] riporta i tempi di esecuzione e il numero medio di *spin flip* effettuati per secondo per l'implementazione su CPU senza variabili binarie, quella su CPU con variabili

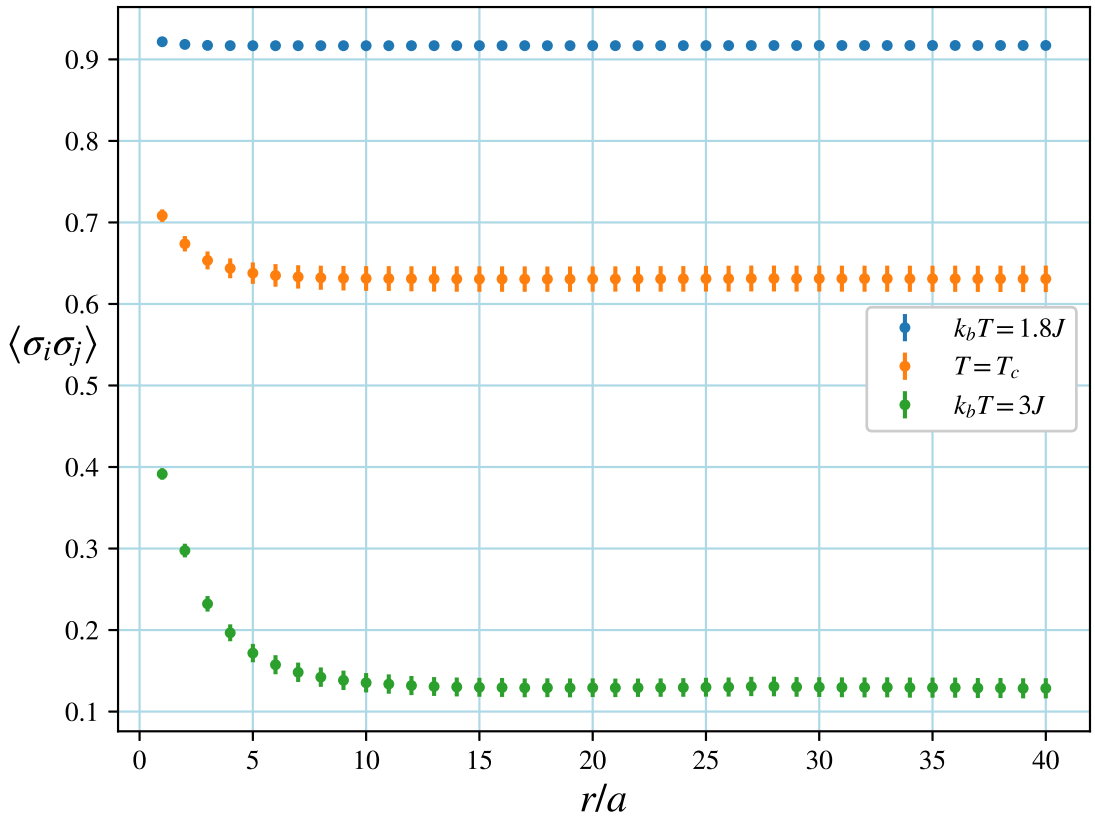

Figura 3: Correlazione spin-spin. I risultati sono stati ottenuti da simulazioni con parametri: $L = 148, N_{rep} = 64, t_{max} = 100$.

binarie, e quella finale su GPU. I test si riferiscono a un ciclo di esecuzione che misura la sola magnetizzazione a varie temperature diverse, in modo da produrre una curva analoga a quella della figura [2]. Tutti i programmi sono stati testati con gli stessi parametri e lo stesso numero di *spin flip* totali, anche se nelle versioni che usano variabili binarie, lo stesso numero *flip* si ottiene con meno ripetizioni.

È da ricordare che, dato che l'uso delle variabili binarie comporta una velocizzazione nel calcolo dell'energia e nella valutazione della condizione di accettazione, ma non nella lettura degli spin, il guadagno di performance che ne consegue varia a seconda del numero di aggiornamenti che viene lasciato passare tra una misura e un'altra.

Osserviamo che la velocizzazione dovuta alla parallelizzazione su GPU è del 46.7%. Nei dispositivi a nostra disposizione, la versione per CPU ha a disposizione un singolo *core* a 2.13 GHz di frequenza, mentre quella su GPU ha 480 *core* a 701 MHz. Questo porta a una

Programma	***spin flip*/ms**	**velocizzazione parziale**	**velocizzazione totale**
CPU semplice	6808.389	1x	1x
CPU + *bit spins*	34540.218	5.073x	5.073x
GPU + *bit spins*	1614572.817	46.745x	237.144x

Tabella 2: Performance delle varie versioni dell'algoritmo. Tutte le simulazioni avevano come parametri $L = 196, N_{rep} = 64, t_{max} = 200$, and covered a range of 15 temperatures.

velocizzazione attesa massima del 62.5%. Ricordando che nel nostro problema l'accesso alla memoria globale della GPU è necessariamente molto frequente, il che limita la velocità di esecuzione complessiva, consideriamo questo risultato soddifacente e in linea con le nostre aspettative.

Riferimenti

[1] O. Lars. "Crystal Statistics. I. A Two-Dimensional Model with an Order-Disorder Transition". In: *Physical Review* 65 (1944), pp. 117–149. DOI: :10.1103/PhysRev.65.117.

[2] Y. Kanada N. Ito. "Monte carlo simulation of the ising model and random number generation on the vector processor". In: *Proceedings of Super-computing 90* (1990), pp. 753–763. DOI: 10.1109/SUPERC.1990.130097.