Alla mia famiglia.

Indice

Introduzione **v**

1 Il problema fisico: Spin Glasses **1**
1.1 Modello di Ising 1
1.2 Metodo Monte Carlo per il Modello di Ising 3
1.2.1 L'Algoritmo di Metropolis 5
1.2.2 Il problema dei numeri casuali 6
1.3 Calcolo di Spin-Glass su architetture dedicate 7
1.3.1 JANUS 7
1.4 Calcolo di Spin-Glass su architetture standard 8

2 Multi-spin coding **11**
2.1 Algoritmo di Metropolis per il multi-spin coding 12
2.2 Implementazione per architetture x86 16
2.3 Analisi delle prestazioni 23

3 IBM Cell Broadband Engine **25**
3.1 Architettura del processore 25
3.1.1 Introduzione e storia 25
3.1.2 Visione generale dell'architettura del sistema 26
3.1.3 Il PPE 28
3.1.4 Il SPE 31
3.1.5 L'EIB 33
3.1.6 L'MFC 33
3.1.7 L'I/O 35
3.2 Modelli di programmazione 35
3.2.1 Il supporto SIMD 38
3.3 Confronto con architetture standard 40

4 Implementazione su architettura Cell-BE **43**
4.1 Requisiti dell'implementazione 43

4.2 Algoritmo per architettura Cell-BE 44
4.3 Implementazione per architettura Cell-BE 46
4.3.1 Lato PPE 47
4.3.2 Lato SPE 51
4.4 SPU Timing: analisi dell'implementazione dell'algoritmo su lato SPE 60
4.5 Ottimizzazione dell'implementazione 62
4.5.1 Vettorializzazione del codice 63
4.5.2 Loop unrolling 65
4.5.3 Cubi il cui lato è una potenza di due 66
4.5.4 Variabili di tipo register e allineamento degli scalari . . . 69
4.6 Limiti di implementazione 69

5 Analisi dei risultati 73
5.1 Validità dei risultati ottenuti 73
5.2 Simulazioni effettuate 73

Conclusioni e Sviluppi Futuri 81
Implementazione per cubi di lato 64 81
Conclusioni 83

A Implementazione dell'Algoritmo di Metropolis con multi-spin coding per architetture x86 85

B Implementazione per Cell, versione sgv11_15_3unroll 87
B.1 Lato PPE 87
B.2 Lato SPE 92
B.3 File di libreria 97

C Implementazione per Cell, versione sgv11_8_4unroll 99
C.1 Lato PPE 99
C.2 Lato SPE 104
C.3 File di Libreria 110

D Implementazione per Cell, versione sgv12_lparam 113
D.1 Lato PPE 113
D.2 Lato SPE 118
D.3 File di Libreria 122

Introduzione

L'obiettivo di questa tesi è lo studio dell'innovativa architettura IBM Cell Broadband Engine, nata dalla collaborazione tra IBM, SCEI (Sony Computer Entertainment Incorporated) e Toshiba. Lo studio è rivolto a verificare che l'architettura Cell possa essere utilizzata per la risoluzione di rilevanti problemi di fisica termodinamica e di fisica statistica. L'analisi delle potenzialità offerte da questa architettura avverrà tramite la realizzazione di un software per la simulazione di modelli matematici in grado di descrivere qualitativamente il fenomeno del ferromagnetismo (proprietà per la quale alcuni metalli presentano magnetizzazione persistente in determinate condizioni ambientali). Il modello che utilizzeremo in questo elaborato è conosciuto come *Modello di Ising*, uno dei modelli più studiati della fisica statistica [24].

Il Modello di Ising in tre dimensioni spaziali non è ancora stato risolto analiticamente. Lo studio delle proprietà deve dunque essere condotto numericamente. Ci avvarremo di un particolare metodo numerico, denominato *Metodo Monte-Carlo,* che permette il calcolo approssimato di medie pesate.

L'introduzione di questo metodo nel campo del calcolo computazionale introduce però alcuni problemi legati principalmente ai fattori di tempo. Il calcolo di medie risulta tanto più accurato quanto più a lungo viene "campionato" il sistema simulato. Grazie all'utilizzo di una variante del Metodo Monte-Carlo, ovvero l'Algoritmo di Metropolis, potremo avere ottime stime in minor tempo rispetto al metodo classico.

L'Algoritmo di Metropolis verrà utilizzato come base del progetto, ovvero lo sviluppo di un applicativo che simuli l'evoluzione nel tempo di un determinato sistema, che rappresenterà il metallo considerato, al variare della temperatura a cui è sottoposto il sistema stesso. Obiettivo della tesi è la realizzazione di una implementazione efficiente dell'Algoritmo di Metropolis per il modello di Ising su architettura Cell-BE.

Per migliorare l'efficenza tratteremo una tecnica di implementazione detta *multi-spin coding*, con la quale dimostreremo com'è possibile migliorare l'efficenza dei calcoli eseguiti nell'Algoritmo di Metropolis e come questa tecnica è adatta all'implementazione di svariati problemi di calcolo su variabili di spin.

Questa tecnica verrà illustrata tramite il supporto di una implementazione in linguaggio C dell'Algoritmo di Metropolis per generiche architetture x86.

Proseguiremo introducendo le caratteristiche tecniche dell'architettura Cell, descrivendo tutti gli elementi che la compongono. Questa architettura introduce molte nuove funzionalità rispetto ad altre architetture commerciali. L'utilizzo di questa architettura richiede l'analisi di appositi modelli di programmazioni e descriveremo dettagliatamente come implementare il programma di simulazione sul Cell.

Nella nostra simulazione per architettura Cell, sarà di fondamentale importanza l'analisi di due fattori: il fattore delle *performance* e il fattore dell'*efficenza*. Il primo permetterà il confronto della tempistica di calcolo tra l'implementazione x86 e l'implementazione Cell. Il secondo permetterà il confronto tra la nostra implementazione Cell e le prestazione teoriche che questa architettura può ottenere.

Infine, vedremo i risultati finali ottenuti dalle nostre simulazioni: descriveremo le performance e l'efficenza della nostra implementazione e vedremo come e quanto essa sia più conveniente rispetto alla versione per architettura x86.

L'elaborato è suddiviso nei seguenti capitoli:

- Capitolo 1 Il problema fisico: Spin Glasses
  Introdurremo il concetto di ferromagnetismo e uno dei modelli fisici usati per studiarlo: il Modello di Ising. Analizzeremo un metodo di risoluzione di questo modello, ovvero il Metodo Monte-Carlo e, nello specifico, una sua variante detta Algoritmo di Metropolis e vedremo come questo algoritmo può essere utilizzato nel mondo informatico.

- Capitolo 2 Multi-spin coding
  In questo capitolo descriveremo un'efficace tecnica di implementazione di calcoli su variabili spin, avvalendoci del supporto di una implementazione per architettura x86.

- Capitolo 3 IBM Cell Broadband Engine
  Il capitolo descriverà l'architettura IBM Cell-BE, analizzandone i singoli componenti e le funzionalità. Analizzeremo i modelli di programmazione da usare su questa architettura.

- Capitolo 4 Implementazione su architettura Cell-BE
  La fase di sviluppo del progetto verrà trattata in questo capitolo, dove spiegheremo come applicare la tecnica introdotta nel Capitolo 2 al-

l'architettura Cell. Descriveremo quindi l'implementazione del progetto, analizzandone le ottimizzazione ed elencandone i limiti.

- Capitolo 5 Analisi dei risultati
  In questo capitolo, tratteremo l'intera analisi dei risultati, analizzando i due fattori che hanno spinto all'uso dell'architettura Cell per le simulazioni spin glass: l'efficenza dell'implementazione e le performance. Infine descriveremo brevemente il significato fisico dei grafici prodotti dalle nostre simulazioni.

Capitolo 1

Il problema fisico: Spin Glasses

1.1 Modello di Ising

Uno dei fenomeni più interessanti che riguardano lo studio dei metalli è il *ferromagnetismo*, proprietà manifestata da alcuni materiali, quali ad esempio il ferro, la magnetite, il cobalto e il nichel, i quali presentano magnetizzazione anche in assenza di campi magnetici esterni.

In tali materiali i momenti magnetici degli atomi si orientano tutti quanti nello stesso verso manifestando, a livello macroscopico, una *magnetizzazione* totale non nulla. Tale fenomeno accade se la temperatura del materiale è al di sotto di una temperatura caratteristica, conosciuta come temperatura critica o di Curie [24]. Al di sopra della temperatura critica gli atomi sono orientati in modo completamente casuale e non producono alcuna magnetizzazione macroscopicamente rilevabile.

Uno dei modelli fisici più rilevanti per lo studio del fenomeno del ferromagnetismo è il Modello di Ising. In tale modello un materiale è rappresentato da un reticolo *cubico* d-dimensionale, di lato L, con $N = L^d$ siti. Ad ognuno degli N siti è associato il valore di uno spin S_i, con $i = 1, ..., N$. Ogni spin può avere i valori $+1$ o -1 rappresentando due possibili orientazioni opposte dei momenti magnetici.

L'insieme dei valori degli spin in un dato momento determina la configurazione che nel seguito indicheremo con $\{S\}$. Data una configurazione $\{S\}$ di un sistema, si definisce energia di interazione di uno spin S_i la quantità

$$E_i = -\sum_{<j>} S_i S_j J_{ij} \ , \tag{1.1}$$

ove con la notazione $< j >$ si rappresenta l'insieme dei vicini spaziali dello

spin S_i, e J_{ij} rappresenta la forza di interazione tra lo spin S_i e lo spin S_j, fissata ad un valore costante k. L'energia di $\{S\}$ è definita mediante la formula

$$E\{S\} = \sum_{i=1}^{N} E_i - B\sum_{i=1}^{N} S_i \ ,$$

ove B è, nel caso generale, il valore di un campo magnetico esterno che, nel seguito, considereremo per semplicità nullo.

Una variante del Modello di Ising è il *Modello di Edwards-Anderson* [18], in cui i valori delle variabili J_{ij} non sono fissate ad un valore costante k, bensì sono estratte secondo una distribuzione $P(k)$ con media 0 e varianza 1: ad esempio, $k = +1$ o $k = -1$ con la stessa probabilità.

Nonostante la semplicità del Modello di Edwards-Anderson, esso presenta una fenomenologia ricca e complessa, ed è il modello largamente più studiato di sistema di spin glass. Tali sistemi hanno una fase di bassa temperatura con spin congelati (ovvero la cui dinamica è lenta rispetto ai tempi di osservazione) ma per effetto delle costanti di accoppiamento J_{ij} concorrenti, non presentano magnetizzazione macroscopica. Per maggiori dettagli si veda [29].

Lo studio del comportamento di un materiale ferromagnetico al variare della temperatura, ovvero la spiegazione del perchè tali materiali hanno una transizione di fase ad una determinata temperatura critica sopra la quale non c'è magnetizzazione, può essere condotto esplorando le proprietà del Modello di Ising. Le proprietà del Modello di Ising, come ad esempio la relazione tra magnetizzazione residua e temperatura, non sono ancora state ricavate analiticamente per reticoli d-dimensionali, con $d > 2$.

È quindi necessario ricorrere a metodi di risoluzione numerici, simulando l'andamento del sistema a differenti temperature.

Consideriamo

$$M = \sum_{\{S\}} M(\{S\})\, P(\{S\}) \ , \tag{1.2}$$

cioè il valore medio di magnetizzazione, dato dalla sommatoria delle magnetizzazioni di tutte le possibili configurazioni $\{S\}$, pesate con la probabilità che ci si possa trovare nella configurazione $\{S\}$ stessa.

Per un sistema che si trova in equilibrio con una sorgente termica a temperatura T, la distribuzione di probabilità $P(\{S\})$ è la distribuzione di Boltzmann [26] definita come

$$P(\{S\}) = \frac{e^{-\beta E(\{S\})}}{Z} \ , \tag{1.3}$$

dove β è definito come

$$\beta = \frac{1}{kT} ,$$

in cui k è la costante di Boltzmann, T la temperatura a cui si trova il nostro sistema. Definiamo invece la costante di normalizzazione Z, come

$$Z = \sum_{\{S\}} e^{-\beta E(\{S\})} .$$

E' quindi necessario enumerare tutte le possibili configurazioni $\{S\}$ del sistema e calcolare la 1.2 utilizzando la 1.3.

Consideriamo, ad esempio, un reticolo di 32^3 spin. Le possibili configurazioni del sistema sono 2^{32^3}, e, in principio, fissato un valore della temperatura occorre determinarle tutte per individuare quella di energia minima.

In generale, viene utilizzato un metodo alternativo per il calcolo del valore della magnetizzazione, cioè il *Metodo Monte Carlo.* In seguito, per semplicità, tratteremo principalmente il Modello di Ising piuttosto che il Modello di Edwards-Anderson, notando che, il passaggio da un modello all'altro, è facilmente realizzabile.

1.2 Metodo Monte Carlo per il Modello di Ising

L'idea del Metodo Monte Carlo nasce a metà degli anni '40 all'interno del Progetto Manhattan [14], un progetto che gli Stati Uniti d'America portarono avanti durante la seconda guerra mondiale e che portò alla creazione della bomba atomica. I fautori di questo metodo furono John von Neumann e Stanisław Marcin Ulam mentre il nome "Monte Carlo" fu assegnato successivamente da Nicholas Constantine Metropolis, che si ispirò al celebre casinò monegasco.

Il Metodo Monte Carlo è un metodo generale per il calcolo di medie statistiche del tipo della 1.2. Per una trattazione approfondita del Metodo Monte Carlo si veda [18].

Per il calcolo della magnetizzazione di un sistema tramite l'utilizzo del Metodo Monte Carlo, indichiamo con

$$M = \frac{\sum_t M\{S_t\}}{N_t} , \tag{1.4}$$

la media aritmetica di un numero grande N_t, ma limitato, di $\{S\}$ generate secondo la distribuzione di Boltzmann 1.3.

Il metodo Monte Carlo permette di esplorare lo spazio delle configurazioni privilegiando quelle più interessanti, ovvero quelle il cui peso statistico è più elevato.

Consideriamo la distribuzione di Boltzmann definita nella 1.3. Per semplicità, nel seguito, indicheremo con lettere greche le possibili configurazioni $\{S\}$ del sistema (stati) e con p_α la probabilità di Boltzmann dello stato α.

Consideriamo ora un sistema generico che si trova in uno stato λ e generiamo un nuovo stato random μ, con μ diverso da λ. La probabiltà che un sistema passi dallo stato λ allo stato μ è data da $P(\lambda \rightarrow \mu)$, e deve rispettare le seguenti condizioni:

1. La probabilità non deve variare nel tempo, cioè la probabilità di passare da λ a μ deve essere costante.

2. La probabilità deve dipendere esclusivamente dallo stato λ e dallo stato μ.

3. Inoltre si richiede che

$$\sum_{\mu} P(\lambda \rightarrow \mu) = 1 \ , \tag{1.5}$$

   cioè a partire dallo stato λ è possibile raggiungere qualsiasi stato μ (condizione di ergodicità [18]) e la somma delle probabilità di raggiungere qualsiasi stato μ dev'essere uguale a 1.

Un altra condizione che deve rispettare la distribuzione di probabilità è il cosidetto *bilancio dettagliato* [18], che impone il raggiungimento dell'equilibrio dinamico del sistema in modo tale che gli stati siano distribuiti secondo p_λ. Matematicamente possiamo definire

$$p_\lambda P(\lambda \rightarrow \mu) = p_\mu P(\mu \rightarrow \lambda) \rightarrow \frac{p_\lambda}{p_\mu} = \frac{P(\mu \rightarrow \lambda)}{P(\lambda \rightarrow \mu)} \ ,$$

ovvero la probabilità di passare da μ a λ è uguale alla probabilità di passare da λ a μ. Dato che p_α, per ogni α, è pari alla distribuzione di Boltzmann ne consegue che

$$\frac{P(\mu \rightarrow \lambda)}{P(\lambda \rightarrow \mu)} = \frac{p_\lambda}{p_\mu} = \frac{e^{-\beta E(\lambda)}}{e^{-\beta E(\mu)}} = e^{-\beta[E(\mu)-E(\lambda)]} \ . \tag{1.6}$$

A partire da quest'ultima possiamo definire qualsiasi probabilità di transizione che rispetti questa condizione. Ad esempio,

$$P\left(\mu \rightarrow \lambda\right) \propto e^{-\frac{1}{2}\beta(E(\lambda)-E(\mu))} \ .$$

Una variante del Metodo Monte Carlo per la risoluzione di simulazioni spin glass è l'*Algoritmo di Metropolis* [26] che tratteremo nella prossima sezione.

1.2.1 L'Algoritmo di Metropolis

L'algoritmo di Metropolis viene utilizzato per realizzare simulazioni Monte Carlo. La particolarità dell'algoritmo di Metropolis è nell'aumento dell'efficenza rispetto ad altre scelte della probabilità di transizione nel Metodo Monte Carlo.

Partiamo con lo scomporre la probabilità $P\left(\lambda \rightarrow \mu\right)$ nel seguente modo:

$$P\left(\lambda \rightarrow \mu\right) = g\left(\lambda \rightarrow \mu\right) A\left(\lambda \rightarrow \mu\right) \ , \tag{1.7}$$

dove $g\left(\lambda \rightarrow \mu\right)$, detto probabilità di selezione, indica la probabilità con cui il nostro algoritmo genera un nuovo stato μ a partire dallo stato λ.

Il secondo termine a secondo membro $A\left(\lambda \rightarrow \mu\right)$ è detto grado di accettazione e indica la frequenza con cui avviene la transizione $\lambda \rightarrow \mu$.

Analogamente, definiamo la probabilità $P\left(\mu \rightarrow \lambda\right)$ come

$$P\left(\mu \rightarrow \lambda\right) = g\left(\mu \rightarrow \lambda\right) A\left(\mu \rightarrow \lambda\right) \ . \tag{1.8}$$

Ricordando le condizioni imposte nella sezione 1.2, abbiamo che

$$\frac{P\left(\lambda \rightarrow \mu\right)}{P\left(\mu \rightarrow \lambda\right)} = \frac{g\left(\lambda \rightarrow \mu\right) A\left(\lambda \rightarrow \mu\right)}{g\left(\mu \rightarrow \lambda\right) A\left(\mu \rightarrow \lambda\right)} \ .$$

Supponiamo, per semplicità, che $g\left(\lambda \rightarrow \mu\right) = g\left(\mu \rightarrow \lambda\right)$.

Detto questo, se abbiamo un grado di accettazione molto basso, il nostro algoritmo farà in modo che il sistema rimanga il più possibile nello stato attuale ed eviterà il più possibile di accettare nuovi stati. Una simulazione deve poter "spaziare" il più possibile attraverso tutti gli stati in cui si può trovare il sistema considerato. Per ottimizzare l'algoritmo si moltiplicano le quantità $A\left(\lambda \rightarrow \mu\right)$ e $A\left(\mu \rightarrow \lambda\right)$ per uno stesso fattore costante, in modo da alzare il più possibile il grado di accetazione.

Supponiamo poi che $P\left(\mu \rightarrow \lambda\right) > P\left(\lambda \rightarrow \mu\right)$. Ricordando che $g\left(\lambda \rightarrow \mu\right) = g\left(\mu \rightarrow \lambda\right)$, poniamo

$$A\left(\mu \rightarrow \lambda\right) = 1 \ , \tag{1.9}$$

in modo da accettare sempre tale transizione. Come conseguenza di quanto detto nella sezione 1.2, abbiamo che

$$\frac{P\left(\lambda \rightarrow \mu\right)}{P\left(\mu \rightarrow \lambda\right)} = e^{-\beta(E(\lambda)-E(\mu))} \ . \tag{1.10}$$

Usando la 1.10 unitamente alla 1.9 otteniamo

$$P\left(\lambda \rightarrow \mu\right) = e^{-\beta(E(\lambda)-E(\mu))} = e^{-\beta\Delta E} \ . \tag{1.11}$$

Relativamente alla simulazione del Modello di Ising o del Modello di Edwards-Anderson, diciamo che gli stati λ e μ corrispondono rispettivamente a due configurazioni di spin $\{S_1\}$ e $\{S_2\}$. Il passaggio da $\{S_1\}$ a $\{S_2\}$ è dato dall'inversione di un singolo spin. Se il passaggio da $\{S_1\}$ a $\{S_2\}$ corrisponde a $\Delta E < 0$, allora $P(S_1 \rightarrow S_2) = 1$ e quindi $P(S_2 \rightarrow S_1) = e^{-\beta \Delta E}$. Come si vede, il passaggio da $\{S_2\}$ a $\{S_1\}$ sarà regolato dalla 1.11: l'accettazione della nuova configurazione sarà dipendente dalla temperatura del sistema insita nel fattore β e dalla differenza dell'energia delle due configurazioni, cioè ΔE.

1.2.2 Il problema dei numeri casuali

Componente fondamentale del Metodo Monte Carlo è la generazione di numeri casuali per realizzare il test di accettazione. I numeri casuali generati da un computer vengono definiti pseudo-casuali in quanto vengono estratti tramite specifici algoritmi che si occupano della generazione di numeri apparentemente indipendenti l'uno dall'altro. Nasce quindi la necessità di avere un algoritmo che generi numeri casuali di buona qualità, ovvero con una mutua correlazione trascurabile e un periodo molto più lungo del numero di valori casuali utilizzati nella nostra simulazione.

Per la risoluzione di questo problema, nel progetto associato a questa tesi, si è deciso di utilizzare l'algoritmo di generazione dei numeri pseudo-casuali a shift-registers di Parisi-Rapuano [25]. Questo algoritmo prevede come base di partenza una sequenza $R(i)$ di numeri pseudo-casuali generati precedentemente. A partire da questo insieme, generiamo un nuovo numero casuale tramite le seguenti formule

$$\begin{cases} R(i) = R(i - \Delta_1) + R(i - \Delta_2) & , \\ R_e(i) = R(i)\, xor\, R(i - \Delta_3) & , \end{cases}$$

dove $\Delta_1 = 24$, $\Delta_2 = 55$ e $\Delta_3 = 61$, e $R_e(i)$ è il numero casuale estratto.

Come detto, un fattore importante degli algoritmi di generazione di numeri pseudo-casuali è la buona qualità dei numeri random generati. La qualità dei numeri random può essere valutata tramite normali test statistici [25].

La scelta di questo generatore di numeri random è stata fatta perché supera brillantemente i test statistici ed è molto facile da implementare ed utilizzare nei programmi.

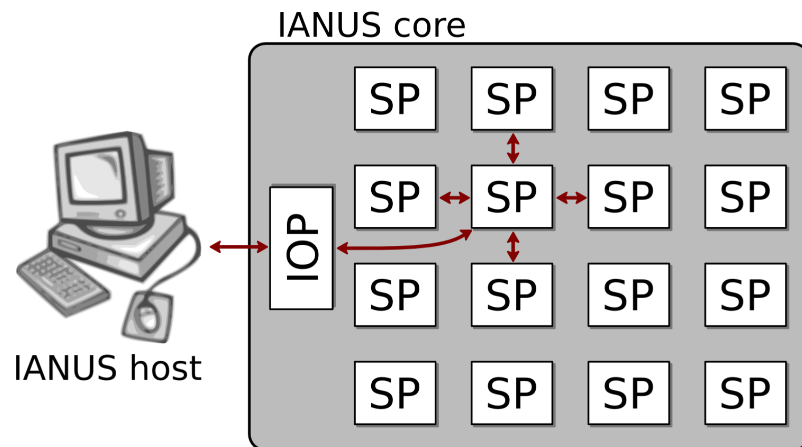

Figura 1.1: Architettura logica di JANUS.

1.3 Calcolo di Spin-Glass su architetture dedicate

1.3.1 JANUS

JANUS è il nome di un progetto nato tra la collaborazione dell'Università di Ferrara, l'Università di Roma, l'Università di Madrid e l'Istituto BiFi di Zaragoza in Spagna. Lo scopo del progetto è lo sviluppo di un sistema ad alte prestazioni implementato tramite uso di FPGA e ottimizzato al calcolo di algoritmi computazionali che richiedono un'alto grado di parallelismo.

L'architettura hardware a *FPGA* (Field Programmable Gate Array), un particolare tipo di dispositivo che essenzialmente contiene una matrice di gates con connessioni programmabili. Il dispositivo si basa principalmente su un componente di base detto cella logica; generalmente una FPGA contiene moltissime celle logiche: ad esempio, per il prototipo JANUS, è stato usato un Xilinx LX160 Virtex 4 contenente più di 152000 celle logiche e più di 4.5 *Mbit* di memoria interna. Ogni cella logica può eseguire svariate funzioni; utilizzando più celle logiche interconnesse è possibile implementare uno specifico programma. L'architettura FPGA permette la progettazione di sistemi dedicati ad alte prestazioni, mantenendo comunque un basso costo.

Lo scopo del progetto è la costruzione di un sistema che porti a massimizzare le prestazione dell'architettura FPGA e di una piattaforma software che ne semplifichi l'utilizzo. Le possibilità offerte dal progetto JANUS sono pressoché illimitate dato che l'alto grado di configurabilità del progetto permetterà di implementare una grande quantità di problemi computazionali, tra cui problemi fisici, che spesso richiedono una grande quantità di risorse computazionali.

Gli obiettivi del progetto JANUS sono:

- sviluppare un motore di simulazione di spin glass ad altissime prestazioni in grado di aggiornare un singolo spin in un tempo inferiore ad $1\,ps$.
- sviluppare una infrastruttura software che sia in grado di gestire uno specifico problema computazionale tramite un numero variabile di FPGA interconnesse.

JANUS è composto principalmente da una board contenente al massimo 16 SP (Simulation Processor). Ogni SP contiene una singola FPGA, la quale viene utilizzata esclusivamente per il calcolo computazionale. Uno schema logico dell'architettura di JANUS è mostrato in figura 1.1.

All'interno della board di JANUS è presente anche una ulteriore FPGA, denominata IOP (Input Output Processor), che si occupa della gestione dei dati in ingresso e in uscita dell'intero sistema. JANUS è collegato, tramite il modulo IOP, ad un computer host, un normale computer con kernel Linux, tramite interfaccia ethernet GigaBit sulla quale vengono spediti i dati ed i comandi per eseguire le simulazioni.

Ogni board può essere interfacciata con altre board, in modo da estendere la potenza di calcolo.

Le prestazioni di una board di JANUS per calcoli di spin glasses si aggirano intorno a $1\,ps$ per l'aggiornamento di uno spin.

Infine, per quanto riguarda i costi effettivi del progetto, basta eseguire un confronto tra il rapporto *prezzo/prestazioni* di sistemi simili: JANUS, in una configurazione a 16 SP, equivale ad un sistema di quasi 50.000 processori Pentium IV, a un sistema di quasi 300.000 processori Blue-Gene/L o a un insieme di circa 8.000 processori Cell. Se consideriamo un prezzo di 1.000$ per unità di un sistema commerciale e per ogni SP di JANUS abbiamo un costo totale di 50.000.000$, per il primo, e solo (relativamente) 500.000$, per il secondo; tale prezzo include inoltre i costi di sviluppo del progetto che rappresenta quasi il 50% della spesa totale.

1.4 Calcolo di Spin-Glass su architetture standard

L'Algoritmo di Metropolis è riassunto nell'algoritmo 1.

Come si può vedere l'algoritmo è molto semplice e facile da implementare su qualsiasi architettura standard e sui comuni computer commerciali. Inoltre si presta all'utilizzo di tecniche implementative che permettono il raggiungimento di performance eccelenti.

Algorithm 1 Algoritmo di Metropolis per una generica architettura di computer.

$\beta \leftarrow \frac{1}{T}$
for $i = 0$ to N **do**
  $\rho = random(0, 1)$
  $S_j \leftarrow$ `vicini di` S_i
  $\Delta E_i \leftarrow 0$
  for all S_j **do**
    $\Delta E_i \leftarrow \Delta E_i + 2 \cdot S_i \cdot S_j \cdot J_{ij}$
  end for
  if $\Delta E_i < 0$ **then**
    $S_i \leftarrow -S_i$
  else if $(\rho \leq e^{-\beta \Delta E})$ **then**
    $S_i \leftarrow -S_i$
  end if
end for

Consideriamo ad esempio la variabile S_i: questa variabile ha valori $+1$ e -1 . Si può quindi rappresentare questa variabile tramite un singolo bit.

Consideriamo ora la costante J_{ij}, nello specifico i suoi possibili valori: anche in questo caso, come per S_i, abbiamo che J_{ij} può valere $+1$ o -1. Anche J_{ij} è una variabile rappresentabile con un singolo bit.

Utilizzare, ad esempio, un tipo di dato predefinito del linguaggio C, sia ad esempio un `int` o `char`, per rappresentare un singolo spin S_i o una singola costante J_{ij} comporta un notevole spreco di memoria.

La soluzione ideale a questo problema è l'implementazione di tipo *multi-spin coding [18]* che tratteremo approfonditamente nel capitolo 2. Gli obiettivi di questa tecnica sono:

- Ridurre l'utilizzo di memoria.
- Sfruttare il parallelismo intrinseco offerto dalle operazioni binarie presenti su tutte le architetture standard.

Prima di passare alla descrizione del multi-spin coding come tecnica per l'aumento delle performance, è necessario fare una considerazione: l'utilizzo di architetture dedicate per i calcoli su spin permette di definire il livello massimo di prestazioni che si possono ottenere; questo livello può essere preso in considerazione quando si studiano le possibilità offerte dalle architetture commerciali, con le quali non si potranno ottenere le stesse prestazioni delle

architetture dedicate, ma possono comunque essere facilmente utilizzate nel campo scientifico.

Un altro vantaggio delle architetture commerciale è che sono facilmente reperibili sul mercato e sono subito predisposte per lo sviluppo. Le architetture dedicate richiedono in genere tempi di sviluppo più lunghi rispetto alle architetture commerciali in quanto le ultime, generalmente, offrono strumenti di sviluppo molto diffusi al contrario dell'architetture dedicate, le quali, molto spesso, richiedono strumenti di sviluppo dedicati.

La scelta dell'architettura Cell è giustificata dal fatto che è un'architettura commerciale ma soprattutto innovativa rispetto alle altre architetture quali, ad esempio, x86 o PowerPC.

Nel seguito descriveremo due implementazioni multi-spin coded dell'Algoritmo di Metropolis per modelli di tipo Ising su architetture x86, prima, e Cell, poi.

Capitolo 2

Multi-spin coding

Il *multi-spin coding* è una tecnica di implementazione che permette di massimizzare la banda tra memoria e CPU e di sfruttare il parallelismo offerto dalla logica bit-wise. I punti chiave dell'implementazione multi-spin coding sono:

1. Ogni variabile viene rappresentata con il numero minimo di bit necessari.
2. Una parola a M bit contiene molte variabili.
3. Operazioni binarie di base su una parola avvengono automaticamente in parallelo sulle variabili che essa contiene.

Nel caso di un generico calcolo su variabili spin, tipo il Modello di Ising, possiamo definire questi ultimi tramite l'utilizzo di un singolo bit: ricordiamo infatti che uno spin ha un valore nell'insieme $S_i \in \{-1, +1\}$ che possiamo trasformare nel insieme dei valori $\{0, 1\}$, cioè il valore di un singolo bit, tramite la funzione di trasformazione

$$g(S_i) = \frac{1}{2}(S_i + 1) \ .$$

In una generica parola a M bit possiamo quindi inserire M spin. Ad esempio, nel caso di $M = 32$, come nella dell'architettura x86, abbiamo che

$$\sigma_i = \left\{ g\left(S_i^0\right), ..., g\left(S_i^{31}\right) \right\} \ ,$$

Ogni bit della parola si riferisce a un sistema differente (copie): per ogni i, abbiamo quindi una parola a *M bit* contenente M sistemi.

Inoltre, dati i siti i, j, il prodotto

$$S_i S_j \ ,$$

Figura 2.1: Disposizione degli spin in una variabili 32 bit.

nell'implementazione multi-spin coding si traduce nell'operazione

$$\sigma_i \otimes \sigma_j \ ,$$

dove, sfruttando il parallelismo intrinseco offerto dall'operatore binario xor, stiamo calcolando, con una operazione, ben M prodotti.

Passiamo ora alla descrizione dell'implementazione multi-spin coding dell'Algoritmo di Metropolis per una generica architettura x86.

2.1 Algoritmo di Metropolis per il multi-spin coding

Per implementare il multi-spin coding, l'Algoritmo di Metropolis deve essere leggermente modificato ed è necessario introdurre alcune semplificazioni che facilitino la sua implementazione: ciò è richiesto principalmente per un fatto di prestazioni e di riduzione della memoria utilizzata. Le semplificazioni che introduciamo sono le seguenti:

- Codificare il valore della differenza di energia di interazione di un singolo spin con i suoi vicini. Ricordando la 1.1 e ricordando che stiamo considerando un reticolo cubico e quindi ogni spin ha 6 vicini, abbiamo che l'energia di interazione di uno spin con i suoi vicini è $\Delta E_i \in \{-6, -4, -2, 0, +2, +4, +6\}$. Ciò è facilmente dimostrabile eseguendo direttamente la sommatoria per tutte le possibili configurazioni dello spin considerato e dei suoi vicini. Come conseguenza

  $$\Delta E_i = E_i^{'} - E_i = -S_i \sum_j J_{ij} S_j - S_i \sum_j J_{ij} S_j = -2E_i$$

  Dove il pedice i di ΔE_i indica lo spin a cui è riferita l'energia ed $E_i^{'}$ è l'energia della configurazione in cui S_i è stato invertito. Risulta quindi che $\Delta E_i \in \{-12, -8, -4, 0, +4, +8, +12\}$.
  In una implementazione multi-spin coding l'energia

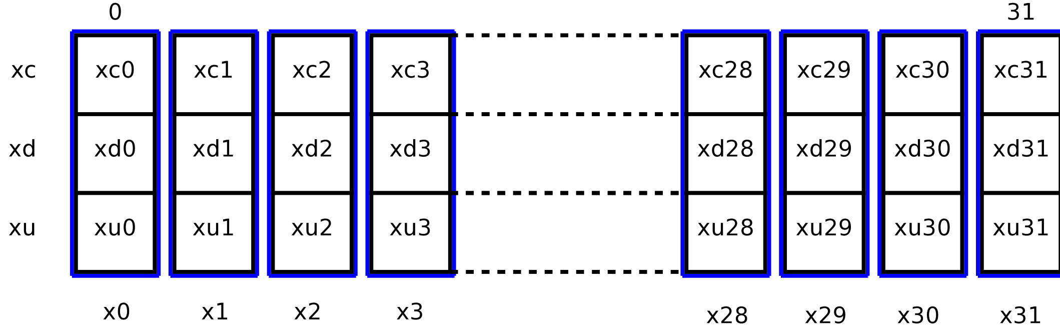

Figura 2.2: Suddivisione di x nelle sue cifre binarie xc, xd e xu.

$$E_i = -\sum_j S_i J_{ij} S_j$$

è codificata come

$$x = \sum_j \sigma_i \otimes J_{ij} \otimes \sigma_j \ , \qquad (2.1)$$

dove $x \in \{0, ..., 6\}$. Possiamo quindi definire la funzione di trasformazione della variazione di energia come

$$\Delta E_i = f(x) = 4x - 12 \ . \qquad (2.2)$$

Considerando l'insieme di valori di x dal punto di vista computazionale sono necessari solo 3 bit. Suddividiamo x nelle sue cifre binarie cioè xc, xd e xu rispettivamente per indicare il terzo, il secondo e il primo bit; inserendo le variabili xc, xd e xu in parole di M bit possiamo rappresentare il valore di x di M sistemi contemporaneamente. La suddivisione di x è visualizzata in figura 2.2.

- Per un fissato i, ad ogni spin sono associate 6 costanti di interazione J_{ij}, dove j è l'indice dei 6 vicini dello spin S_i. Per ciascuna di essi devo realizzare una struttura dati equivalente a quella utilizzata per definire gli spin.

- Esprimere il valore ρ con una dimensione di $32\,bit$ o $64\,bit$, a seconda dell'architettura, in modo da poter contenere numeri random di "buona qualità", come detto nella sezione 1.2.2.

Algorithm 2 Algoritmo di Metropolis per una generica architettura di computer con il supporto per il multi-spin coding. Dove $N = L^3$ e L è il lato del nostro cubo.

```
β ← 1/T
for i = 0 to N do
  ρ = random(0, 1)
  test ← parte intera di −log(ρ)/(4·β)
  S_j ← vicini di S_i
  x ← 0
  x ← calculateX (S_i, S_j, J_ij)
  x ←∼ x
  if terzo bit di x = 1 then
    S_i ← −S_i
  else
    if (test + ΔE_i) ≥ 4 then
      S_i ← −S_i
    else
      S_i ← S_i
    end if
  end if
end for
```

Fatte queste premesse, passiamo a descrivere l'algoritmo adattato secondo queste modifiche, riportato nel listato 2.

Una singola iterazione dell'Algoritmo di Metropolis consiste nell'aggiornamento di tutti gli spin del reticolo considerato. L'indice i si riferisce allo spin su cui si effettua il test. Nel linguaggio C, ad ogni sito i corrisponde un elemento di un array di $N = L^3$ parole a M bit.

Si noti che ogni elemento di questo array rappresenta lo spin i di ciascuno degli M sistemi presenti nell'elemento dell'array stesso; quindi, l'aggiornamento dello spin i è in realtà l'aggiornamento dello spin i di tutti gli M sistemi. Questo concetto è alla base di qualsiasi implementazione multi-spin coding.

Si noti, che questo array contiene tutti gli spin del reticolo: una parte fondamentale dell'algoritmo è la gestione dell'indicizzazione di questo array, che spiegheremo nella sezione 2.2.

Analizziamo ora le inizializzazioni delle variabili $test$, ρ, β, S_j e E_i presenti nell'algoritmo 2. La prima è il punto cruciale dell'implementazione dell'Algoritmo di Metropolis; ricordando la 1.9 e la 1.11, indichiamo con λ una determinata configurazione in cui l'energia di interazione di uno specifico spin S_i con i suoi vicini S_j, tenendo conto delle relative costanti J_{ij}, sia E_i; indi-

chiamo poi con λ', la configurazione in cui si ha $S_i' = -S_i$ e la sua relativa energia di interazione E_i'. La differenza di energia di interazione ΔE_i tra le due configurazioni λ e μ sarà evidentemente $\Delta E_i = E_i' - E_i$.

Ricordando ancora la 1.9 e la 1.11, abbiamo che se $P(\lambda \to \lambda')$ corrisponde ad un calo di energia del reticolo, $P(\lambda' \to \lambda)$ corrisponde ad un aumento. Nel primo caso, la nuova configurazione viene accettata. Nel secondo, la nuova configurazione viene accettata con una probabilità pari a $e^{-\beta \Delta E_i}$. Inoltre, ricordando la 2.2 abbiamo che

$$P(\lambda \to \lambda') = e^{-\beta(\Delta E_i)} = e^{-\beta(f(x))} ,$$

dove x rappresenta la differenza di energia nell'insieme dei valori $\{0, 1, 2, 3, 4, 5, 6\}$. Il test di accettazione della transizione $\lambda \to \lambda'$ consiste nell'estrarre un numero casuale tra 0 e 1 e verificare che

$$\rho \leq e^{-\beta(f(x))} , \tag{2.3}$$

$$log(\rho) \leq -\beta(4x - 12) , \tag{2.4}$$

$$-\frac{log(\rho)}{\beta} \geq 4x - 12 , \tag{2.5}$$

$$-\frac{log(\rho)}{4\beta} \geq x - 3 , \tag{2.6}$$

$$-\frac{log(\rho)}{4\beta} - x \geq - 3 . \tag{2.7}$$

Si noti che $\frac{log(\rho)}{4\beta}$ è sicuramente negativo, dato che $0 \leq \rho \leq 1$.

Il calcolo di x avviene tramite l'utilizzo della formula 2.1, ovvero codificando la

$$E_i = -\sum_j S_i \cdot J_{ij} \cdot S_j$$

come

$$x = \sum_j \sigma_i \otimes J_{ij} \otimes \sigma_j . \tag{2.8}$$

Si noti che nell'algoritmo 2 viene eseguita la negazione bit-a-bit della variabile x; si noti inoltre che la negazione bit-a-bit equivale a $\bar{x} = 7 - x$. Dal punto di vista computazionale, ciò permette una più facile esecuzione del successivo blocco di istruzioni, infatti il test si trasforma in

$$
\begin{aligned}
-\frac{log\,(\rho)}{4\beta} - 7 + \bar{x} &\geq -3 \,, \\
-\frac{log\,(\rho)}{4\beta} + \bar{x} &\geq 4 \,.
\end{aligned}
$$

Si noti che, se il bit più significativo di $\sim x$ è 1, ciò corrisponde ad una $\Delta E_i < 0$ e quindi il test ha successo. Se invece il bit più significativo di $\sim x$ è zero basta verificare che

$$
\left[-\frac{log\,(\rho)}{4\beta}\right] + \bar{x} \geq 4 \,, \tag{2.9}
$$

ovvero che ci sia un carry sul terzo bit nel calcolo del primo membro.

Il primo blocco condizionale, presente nell'algoritmo 2, esegue un controllo sul terzo bit del valore di $\bar{x}$: il terzo bit a uno indica un valore minore o uguale a 4. Il blocco condizionale più interno invece esegue l'inversione di S_i solo se la 2.9 è vera.

2.2 Implementazione per architetture x86

Descriviamo ora un'implementazione di Metropolis scritta in linguaggio C [21]; ci concentreremo principalmente sulle strutture dati e sulle istruzioni che riguardano l'implementazione vera e propria di Metropolis. L'implementazione completa si può trovare in appendice A e la funzione che descriveremo è `Metropolis_Update()`.

Per analizzare il codice che esegue una iterazione di Metropolis, suddividiamo l'algoritmo in quattro parti:

1. Calcolo del numero random e del prime termine a primo membro della 2.7.

2. Calcolo degli indici dei vicini.

3. Calcolo del valore della differenza di energia tra la configurazione corrente e la nuova configurazione, cioè il valore x della 2.9.

4. Accettazione o meno della nuova configurazione

La prima parte si svolge tramite il codice seguente:

```
1  #ifdef _RAND_64_
2  typedef unsigned long long RANDOMWORD;
3  #endif
4  #ifndef _RAND_64_
5  typedef unsigned RANDOMWORD;
6  #endif
7  ...
8  #define RANDOM ((irr+=(!ip)),((ira[irr][ip++]=ira[irr
      ][ip1++]+ira[irr][ip2++])^ira[irr][ip3++]))
9  #define N_RANDOM (-((double)(T_4))*log(((double)(
      RANDOM))*(C_inv_max_rand)))
10 ...
11 RANDOMWORD ira[256][256];
12 unsigned char ip,ip1,ip2,ip3,irr;
13 const double C_inv_max_rand=1.0/((double)(~((
      RANDOMWORD)(0))));
14 ...
15 if((IDX=(unsigned)N_RANDOM)>3) IDX=3;
```

Nella macro `RANDOM` riconosciamo il principio del generatore di numeri pseudo-random a shift-registers di Parisi-Rapuano descritto nella sezione 1.2.2.

L'utilizzo dell'operatore `,` (virgola) permette di svolgere tutte le operazioni che sono alla sua sinistra, restituendo come valore ciò che c'è alla sua destra. Nello specifico del nostro caso, a sinistra abbiamo che la variabile `irr` incrementa il proprio valore quando `ip` è pari a zero: ciò permette di cambiare riga nella matrice `ira` ed allungare ulteriormente il periodo del generatore.

Si noti che la variabili `irr`, `ip`, `ip1`, `ip2`, e `ip3` sono definite come `char` in modo che il loro valore sia compreso tra 0 e 255 in modo da evitare che vengano "mascherate" a 256, cioè la lunghezza della tabella `ira`.

Questa tabella viene inizializzata con valori interi come specifica la macro `RANDOMWORD`. Ricordando quanto detto nella sezione 2.1 a proposito del valore di ρ, i valori della tabella di `ira` devono essere normalizzati ad un valore in virgola mobile compreso tra 0 e 1: per fare ciò moltiplichiamo il valore estratto dalla macro `RANDOM` per `C_inv_max_rand`; il risultato dell'ultima moltiplicazione è proprio il valore di ρ da utilizzare nella 2.7, cioè nella macro `N_RANDOM`; in quest'ultima, il valore `T_4` deriva dalla 2.7 e quindi

$$\frac{1}{4 \cdot \beta} = \frac{T}{4} \ .$$

Infine il blocco condizionale sulla variabile `IDX` forza il valore di `N_RANDOM`

al valore 3: considerando il primo termine a primo membro della 2.9 abbiamo che per valori maggiori di 3 il test risulta sempre vero. In logica, binaria il valore decimale 3 vale 11; per valori decimali maggiori di 3 non è detto che i primi due bit valgano ancora 11. Si limita, quindi, il primo termine del primo membro della 2.9 a valori compresi tra 0 e 3 in modo da facilitare il confronto che verrà svolto nell'ultimo blocco di codice, cioè la scelta o meno della nuova configurazione.

Passiamo ora alla seconda parte dell'implementazione di Metropolis, cioè la parte che calcola le coordinate dei vicini dello spin corrente. Il codice è il seguente:

```
1  SWORD Spin[VOLUME];
2  ...
3  long opx[L] = {1,       1, ...,   1,   -(L-1)};
4  long opy[L] = {L,       L, ...,   L, -(L^2-L)};
5  long opz[L] = {L^2,   L^2, ..., L^2, -(V-L^2)};
6  long omx[L] = {L-1,    -1, ...,  -1,       -1};
7  long omy[L] = {L^2,    -L, ...,  -L,       -L};
8  long omz[L] = {V-L^2, -L^2, ..., -L^2,   -L^2,};
9  ...
10 for(iz=0,in=0; iz<L; iz++) {
11   Plus_z=opz[iz];     Minus_z=omz[iz];
12   for(iy=0; iy<L; iy++) {
13     Plus_y=opy[iy];       Minus_y=omy[iy];
14     for(ix=0; ix<L; ix++,in++) {
15       ...
16       S0=Spin[in];
17       V0[0]=Spin[in+opx[ix]]; V0[1]=Spin[in+omx[ix]];
18       V0[2]=Spin[in+Plus_y];  V0[3]=Spin[in+Minus_y];
19       V0[4]=Spin[in+Plus_z];  V0[5]=Spin[in+Minus_z];
20       ...
21     }
22   }
23 }
```

In questa implementazione abbiamo un array `Spin` contenente tutti gli spin del reticolo: la lunghezza dell'array è pari al volume del reticolo. La variabile `in` indicizza l'array `Spin` e indica lo spin corrente, cioè lo spin su cui Metropolis sta proponendo la nuova configurazione. La disposizione in memoria di ogni singolo elemento di `Spin` può essere visualizzata in figura 2.3.

Per permettere l'esecuzione di Metropolis su tutti gli spin del reticolo ven-

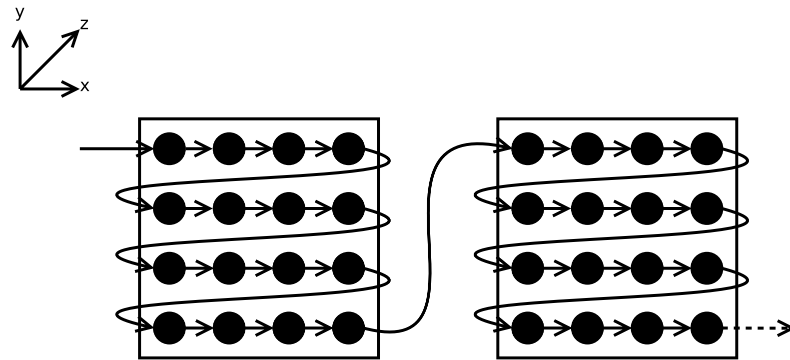

Figura 2.3: Disposizione logica in memoria dell'array `Spin`. Lo spin S_1 è il primo in alto a sinistra nel quadrato di sinistra. Ogni quadrato rappresenta una delle L facce, dove L è il lato del cubo.

gono utilizzati i tre cicli contenuti nel codice: si noti che le variabili `ix`, `iy` e `iz` rappresentano le coordinate (x, y, z) dello spin corrente all'interno del reticolo.

Le variabili `opx`, `omx`, `opy`, `omy`, `opz`, `omz` indicano a quale offset di coordinate si trovano i vicini dello spin corrente. Si noti che l'uso di queste 6 variabili impone, al reticolo, condizioni periodiche di bordo: considerando l'asse x, anche se il ragionamento può essere esteso agli altri assi, uno spin del bordo destro ha come vicino in direzione positiva il corrispettivo spin sul bordo sinistro, ovvero lo spin con coordinate $(L-1, 0, 0)$ ha come vicino in direzione positiva lo spin con coordinate $(0, 0, 0)$.

Segue poi l'assegnamento della variabile `S0`, che contiene il valore dello spin corrente, e il settaggio dell'array `V0` che contiene rispettivamente:

1. il vicino sull'asse x in direzione positiva
2. il vicino sull'asse x in direzione negativa
3. il vicino sull'asse y in direzione positiva
4. il vicino sull'asse y in direzione negativa
5. il vicino sull'asse z in direzione positiva
6. il vicino sull'asse z in direzione negativa.

Passiamo ora al successivo blocco cioè il calcolo di x. Il codice sorgente che si occupa di ciò è il seguente:

```
1  unsigned int Jx[VOLUME];
2  unsigned int Jy[VOLUME];
3  unsigned int Jz[VOLUME];
4  ...
5  Carry1=Carry2=0;
6  EU=SWORD_ZERO;   ED=SWORD_ZERO;   EC=SWORD_ZERO;
7  // 1o e 2o vicino
      //////////////////////////////////////
8    EU=S0^Jx[in          ]^V0[0];
9  prod=S0^Jx[in+omx[ix]]^V0[1]; Carry1=prod & EU; EU=
      prod^EU; ED=Carry1;
10 // 3o e 4o vicino
      //////////////////////////////////////
11 prod=S0^Jy[in          ]^V0[2]; Carry1=prod & EU; EU=
      prod^EU; ED=Carry1^ED;
12 prod=S0^Jy[in+Minus_y]^V0[3]; Carry1=prod & EU; EU=
      prod^EU; Carry2=Carry1 & ED; ED=Carry1^ED; EC=
      Carry2^EC;
13 // 5o e 6o vicino
      //////////////////////////////////////
14 prod=S0^Jz[in          ]^V0[4]; Carry1=prod & EU; EU=
      prod^EU; Carry2=Carry1 & ED; ED=Carry1^ED; EC=
      Carry2^EC;
15 prod=S0^Jz[in+Minus_z]^V0[5]; Carry1=prod & EU; EU=
      prod^EU; Carry2=Carry1 & ED; ED=Carry1^ED; EC=
      Carry2^EC;
```

Dove le prime due righe sono due reset in modo che il calcolo non venga influenzato dai calcoli sugli spin precedenti.

Le successive righe eseguono il calcolo di x presente nell'algoritmo 2, nelle quali è stato evitato l'uso di un ciclo e sono state usate solo operazioni binarie per permettere una esecuzione molto più veloce del calcolo.

Ricordando ancora una volta quanto detto nella sezione 2.1 e quanto mostrato nella figura 2.1, indichiamo con `EC`, `ED` e `EU` rispettivamente il terzo, il secondo e il primo bit di x di ogni singolo sistema.

Prima di passare all'analisi del codice, si noti l'indicizzazione delle costanti `Jx`, `Jy` e `Jz`: per i vicini positivi le tre costanti vengono indicizzate tramite la variabile `in` mentre per i vicini negativi vengono indicizzate tramite gli indici dei vicini negativi: ciò permette un'ulteriore risparmio di memoria, dato che ad ogni spin sono associate solo le tre costanti `Jx`, `Jy` e `Jz` in direzione positiva.

Per descrivere ogni singola istruzione di questo blocco di codice si inizi col notare che c'è uno schema logico ripetitivo nella chiamate delle istruzioni. Si noti inoltre che in ogni riga del codice sono presenti gli elementi di `V0` associati alle rispettive variabili `Jx`, `Jy` e `Jz`: questo denota che il ciclo usato nell'algoritmo 1, utilizzato per calcolare il valore di ΔE_i, è stato diviso nelle sue 6 iterazioni, una per ogni vicino, e calcolato tramite l'utilizzo di sole operazioni binarie. Analizziamo quindi ogni singola riga del codice, considerando un singolo sistema di spin, cioè un singolo bit delle variabili `Spin`, `EC`, `ED` ed `EU`:

1. Le variabili `EC`, `ED` e `EU` sono appena state azzerate. Nella prima iterazione dobbiamo avere che $x = \sigma_i \otimes J_{i1} \otimes \sigma_1$; ricordando quanto indicato dalla 2.8, codifichiamo, questo prodotto, tramite l'utilizzo dell'operatore `^` (xor).

2. Anche per il secondo vicino si può riconoscere lo stesso schema di istruzioni utilizzato per il primo. Qui, però, viene utilizzata la variabile `prod`, la quale ha una funzione di supporto: infatti, la somma tra `EU` e `prod` può generare riporto e, in questa evenienza, è necessario assegnare il relativo bit di `ED` a 1. Consideriamo quindi l'eventualità in cui `EU=1` ed `prod=1`; la somma binaria `EU+prod` deve restituire il valore binario 10, ovvero `ED=1` ed `EU=0`.
   La presenza o meno del riporto viene calcolata tramite l'istruzione `Carry1 =prod & EU`; nel caso sia presente il riporto l'istruzione `EU=prod^EU` setta la variabile `EU` a 0; infine, `ED` viene settato a 1 tramite l'istruzione `ED=Carry1`.

3. Nel calcolo del terzo vicino vale esattamente quanto detto per il secondo.

4. 5. 6. Nel calcolo del quarto, quinto e sesto vicino vale quanto detto per il calcolo del secondo e del terzo. In questo caso però abbiamo un secondo riporto tra `ED` ed `EC`. Per tutti e tre gli ultimi vicini possiamo trovarci nella situazione in cui `ED` ed `EU` possono essere rispettivamente a 0 e 0, 0 e 1, 1 e 0, 1 e 1. Nel caso peggiore, cioè l'ultimo, la somma binaria con 1, il contributo dato da il quarto, quinto o sesto vicino, genera il riporto `Carry1` il quale genera a sua volta il riporto `Carry2`. Il secondo riporto viene generato dall'istruzione `Carry2=Carry1 & ED`,

Consideriamo ora l'ultimo blocco di codice che permette di accettare o meno la nuova configurazione proposta da Metropolis. Il codice sorgente è il seguente:

```
1 OD={0,  0, ~0, ~0};
2 OU={0, ~0,  0, ~0};
3 ...
4 EC=~EC; ED=~ED; EU=~EU;
```

x	EC	ED	EU	$\Delta E_i = 4 \cdot x - 12$	Accetta Configurazione?
0	0	0	0	-12	Si
1	0	0	1	-8	Si
2	0	1	0	-4	Si
3	0	1	1	0	Si
4	1	0	0	+4	Test
5	1	0	1	+8	Test
6	1	1	0	+12	Test

Tabella 2.1: Possibili valori delle variabili `EC`, `ED` e `EU`

$\Delta E_i'$	~EC	~ED	~EU	Accetta Configurazione?
0	1	1	1	Si
1	1	1	0	Si
2	1	0	1	Si
3	1	0	1	Si
4	0	1	0	Test
5	0	1	1	Test
6	0	0	0	Test

Tabella 2.2: Possibili valori di `EC`, `ED` ed `EU` negati.

```
5  ...
6  IU=OU[IDX]; ID=OD[IDX];
7  ...
8  Carry1=(IU&EU);
9  Carry2=(ID&ED)|((ID|ED)&Carry1);
10 ...
11 Spin[in]^=(Carry2|EC);
```

Per analizzare le precedenti righe di codice consideriamo la tabella 2.1. Riprendiamo ora la 2.9 e riscriviamola come

$$-\left[\frac{log\,(\rho)}{4 \cdot \beta}\right] + (\sim EC, \sim ED, \sim EU) \geq 4\;. \qquad (2.10)$$

Introducendo poi la variabile `IDX` della nostra implementazione nella 2.10 otteniamo

$$IDX + (\sim EC, \sim ED, \sim EU) \geq 4\;. \qquad (2.11)$$

Possiamo quindi riscrivere la tabella 2.1 nella tabella 2.2.

IDX	ID=OD[IDX]	IU=OU[IDX]
0	000...000	000...000
1	000...000	111...111
2	111...111	000...000
3	111...111	111...111

Tabella 2.3: Rappresentazione posizionale di IDX tramite le variabili ID e IU.

Si noti che nelle tabelle è stato considerato un solo sistema per le variabili EC, ED ed EU. Per spiegare cosa avviene in questo codice partiamo dall'ultima riga: il valore di Spin[in] è calcolato a partire dall'operazione *or* binaria tra Carry2 ed EC. Ricordando che EC è il terzo bit di x di ogni sistema contenuto in un elemento di Spin[in] e in seguito alla negazione svolta nella prima riga del codice, abbiamo che se EC=1 la configurazione deve essere accettata altrimenti la configurazione deve subire il test. Il risultato del test è contenuto nella variabile Carry2. L'operatore ^= inverte il valore di Spin[in] se una delle due variabili EC o Carry2 ha valore 1.

L'esecuzione del test avviene tramite il supporto delle variabili Carry1, Carry2, ID e IU. Se si considera un solo bit delle variabili ID e IU si può vedere che esse rappresentano rispettivamente il secondo e il primo bit del valore binario di IDX, ovvero ID e IU sono la rappresentazione posizionale di IDX, come si può vedere dalla tabella 2.3. Si può quindi utilizzare ID e IU per il confronto con le variabili ED e EU: eseguendo il confronto tramite le cifre binarie, ovvero confrontando prima IU ed EU e poi ID ed ED. Il primo confronto viene eseguito dalla riga Carry1=(IU&EU) mentre il secondo confronto viene eseguito dalla riga successiva, cioè Carry2=(ID&ED)|((ID|ED)&Carry1).

L'analisi dell'implementazione per l'architettura x86 proseguirebbe, poi, con lo studio del restante codice sorgente. Data la semplicità di comprensione del restante codice, si è scelto di rimandare direttamente all'appendice A, ove è stato inserito il codice sorgente completo.

2.3 Analisi delle prestazioni

Analizziamo ora le prestazioni di questa implementazione. Sono stati effettuati molti test su questa implementazione ma noi riporteremo solo i risultati più recenti. Come prestazioni di riferimento abbiamo i valori riportati in tabella 2.4.

Si noti che il risultato finale è riferito ad un singolo spin quindi, nel caso del multi-spin coding, ad un singolo sistema della variabile Spin, ovvero un singolo bit. Questo dato, per un'architettura commerciale, è un buon risulta-

Architettura x86	
Intel Core 2 Duo 2.0GHz 64 bit	860$ps/spin$
Intel Core 2 Duo 2.4GHz 64 bit	770$ps/spin$

Tabella 2.4: Tabella dei tempi di riferimento per una generica architettura x86.

to. La necessità di un'implementazione sul Cell è quella di migliorare questi risultati ottenendo quindi prestazioni migliori. Prima di passare all'implementazione sul Cell, analizziamo l'architettura di quest'ultimo: ciò ci permetterà di capirne le potenzialità e progettare la migliore implementazioni, quella che permetterà di migliorare le prestazione ottenute su architettura x86. La descrizione dell'architettura si trova nel capitolo 3 mentre l'implementazione su questa architettura si trova al capitolo 4.

Capitolo 3

IBM Cell Broadband Engine

3.1 Architettura del processore

3.1.1 Introduzione e storia

Il progetto Cell-BE nasce nell'estate del 2000 da una collaborazione tra IBM, SCEI (Sony Computer Entertainment Incorporated) e Toshiba. L'idea alla base di questo progetto era lo sviluppo di un'architettura di sistema che avesse delle prestazioni 100 volte superiori a quelle della Playstation 2, la famosa console di videogiochi di casa Sony. I compiti affidati alle società IBM, SCEI e Toshiba furono rispettivamente lo sviluppo e la progettazione dell'architettura di sistema, la divulgazione delle specifiche e della documentazione sull'architettura e la collaborazione allo sviluppo e alla produzione in larga scala.

Alla fine dell'anno 2000 si giunse al progetto definitivo dell'architettura di sistema che combinava gran parte dell'architettura PowerPC a 64 bit, famosa per essere stata per anni l'architettura di sistema dei computer Apple, unita ad 8 unità di calcolo indipendenti inserite all'interno del processore, le quali sono in grado di operare in maniera totalmente indipendente dall'unità centrale. Il progetto iniziale prevedeva, inoltre, un bus ad altissime prestazioni in grado di far comunicare tra di loro tutte le unità interne al processore.

Gli obiettivi del nuovo processore furono i seguenti:

- Il miglioramento delle prestazioni soprattutto nei videogiochi e nelle applicazioni multimediali in generale: riduzione dei limiti imposti dalla latenza della memoria centrale, dalla larghezza di banda nel trasferimento dei dati all'interno del processore e verso la memoria centrale di sistema, aumento della potenza del processore, riduzione del numero di cicli

di clock per le operazioni più comuni e aumento della dimensione della coda delle istruzioni.

- Gestione in real-time dell'utente e della rete: sia l'utente, sia la rete, devono godere della possibilità di avere tempi di risposta brevi. Con risposta si intende la reazione ad una precedente azione compiuta dall'utente o dalla rete. Tra le principali applicazioni del Cell c'è il suo utilizzo sulla console Playstation 3, la quale deve poter dare all'utente le emozioni di una multimedialità in tempo reale, senza ritardi e senza interruzioni. Allo stesso modo, la rete deve poter essere gestita altrettanto velocemente: considerando ancora l'esempio della console Playstation 3, questa prevede il supporto per la connessione a Internet la quale contiene innumerevoli protocolli e infinite funzionalità che devono essere implementate; è necessario quindi che il processore Cell sia ottimizzato per poter gestire al meglio il carico di lavoro.

- Applicabilità ad una vasta gamma di piattaforme di sistema: il Cell è stato sviluppato per poter essere inserito in svariati sistemi di intrattenimento. Ciò richiede il supporto hardware, ma anche la corrispettiva componente software; la decisione di redigere tutta la documentazione e di distribuirla gratuitamente alle comunità di Internet ha permesso lo sviluppo di centinaia di nuovi progetti su base Cell. Come esempio, ci sono l'adattamento del kernel Linux alla piattaforma Cell sviluppato da IBM in collaborazione con la comunità di sviluppatori del famoso kernel di Linus Torvalds; altro esempio, sono le televisioni Full-HD di ultima generazione prodotte dalla stessa Toshiba in cui è presente il processore Cell.

- Supporto per l'introduzione sul mercato entro il 2005.

Con questi obiettivi si è arrivati nel 2005 con la prima versione di Cell. Passiamo ora a descrivere l'architettura di questo processore nei minimi particolari.

3.1.2 Visione generale dell'architettura del sistema

L'architettura della prima generazione di processori Cell contiene gli elementi di un processore di tipo PowerPC a 64 bit, inseriti in un modulo denominato PPE (Power Processor Element), al quale vengono aggiunti 8 unità di calcolo, chiamate SPE (Synergistic Processor Element), indipendenti dal processore principale. Ogni modulo dell'architettura Cell ha la funzionalità di accesso diretto alla memoria (DMA, Direct Memory Access) tramite un componente denominato MFC (Memory Flow Control): ogni modulo, con questo

componente può comunicare con la memoria centrale o con un'altro elemento dell'architettura.

Oltre a queste componenti sono presenti un controllore della memoria centrale, più un ulteriore controllore delle interfacce di input e output. Tutte queste unità sono interconnesse da un bus multicanale detto EIB (Element Interconnect Bus) in grado di fornire più linee di comunicazione ad ogni unità ad esso collegate. Infine è presente anche un modulo utile per le funzioni di test e debug.

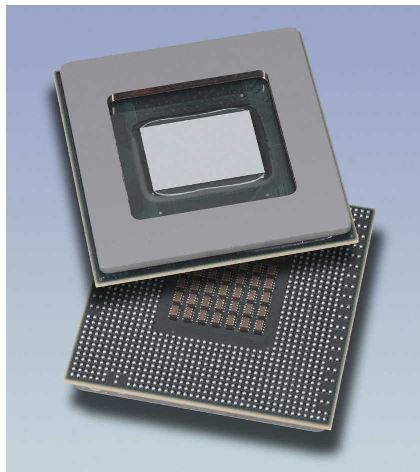

Figura 3.1: Il processore Cell inserito all'interno del proprio package.

Gli attributi chiave di questa architettura sono:

- La capacità di aumentare la frequenza di utilizzo del processore e quindi di aumentare le prestazioni pur mantenendo un basso voltaggio e un basso consumo di energia.

- Un architettura di sistema già conosciuta (PowerPC): ogni programmatore può quindi godere della propria esperienza riutilizzando nozioni già conosciute integrandole con le nuove funzionalità che sono presenti nel processore Cell.

- Il supporto per istruzioni SIMD (Single-Instruction, Multiple-Data): introduce la possibilità di eseguire una stessa istruzione su più dati dello stesso tipo in uno stesso momento, ottimizzando così le prestazione e riducendo i tempi di calcolo.

- SPE in modalità offload: ogni SPE ha a propria disposizione una memoria locale detta Local Store, in cui i dati vengono caricati tramite funzioni

DMA; ogni SPE può quindi eseguire calcoli in maniera indipendente dal processore centrale PPE e dagli altri SPE.

- Un bus di interconnessione ad alta velocità chiamato EIB: un linea di comunicazione molto veloce tra tutti gli elementi del processore che permette lo scambio di dati; inoltre fornisce uno spazio di indirizzi coerente tra tutti gli elementi del processore in modo da permettere una comunicazione efficace. Ad una frequenza di 3.2GHz, l'EIB di un Cell è in grado di superare i 300GB/s di banda aggregata quando si effettuano diversi trasferimenti. A livello pratico però ogni elemento è connesso all'EIB con un canale che può raggiungere una banda massima di 25.6GB/s.
- Un modulo di input e output ad alta velocità altamente configurabile

Tra le caratteristiche tecniche degne di nota ci sono:

- Potenza fino a 4GHz.
- Tensione minima di funzionamento 1.1V.
- Ad una frequenza di 4GHz il peak throughput rate è di 256GFlops (Giga Floating-points Operations Per Second, miliardi di istruzioni su dati in virgola mobile per secondo).
- Supporto per operazioni in virgola mobile singola precisione (IEEE754) e doppia precisione (IEEE854).

Passiamo ora a descrivere nello specifico gli elementi principali di questa architettura.

3.1.3 Il PPE

Il PPE (Power Processor Element) contiene la maggior parte degli elementi di un processore PowerPC 64 bit, per la precisione la quinta generazione denominata G5. Lo scopo principale di questa unità è fornire le funzioni comuni di controllo che ogni processore fornisce, gestire e supportare il sistema operativo e, infine, permettere l'esecuzione di applicazioni. Inoltre, può controllare il funzionamento di ogni singolo SPE tramite l'unità EIB.

Le caratteristiche tecniche principali del PPE sono:

- Principali elementi dell'architettura PowerPC 64 bit, con cui condivide gran parte del set di istruzioni.

Figura 3.2: Schema logico del processore Cell e principali componenti dell'architettura

- 32 KB di cache di livello 1, 512 KB di cache di livello 2.
- Pipeline dei comandi contenente fino a 23 stadi.
- Tecnologia Two-way Multi-processor.

L'architettura del PPE, di cui si può vedere una rappresentazione logica nell'immagine 3.3, è suddivisa nei seguenti sotto-moduli:

- IU (Instruction Unit): si occupa del caricamento delle istruzioni, ovvero della loro interpretazione ed esecuzione, e della gestione dei blocchi condizionali.
- XU (fiXed-point Unit): si occupa delle esecuzioni di calcoli su tipi di dato interi e del caricamento/salvataggio dei dati in memoria.

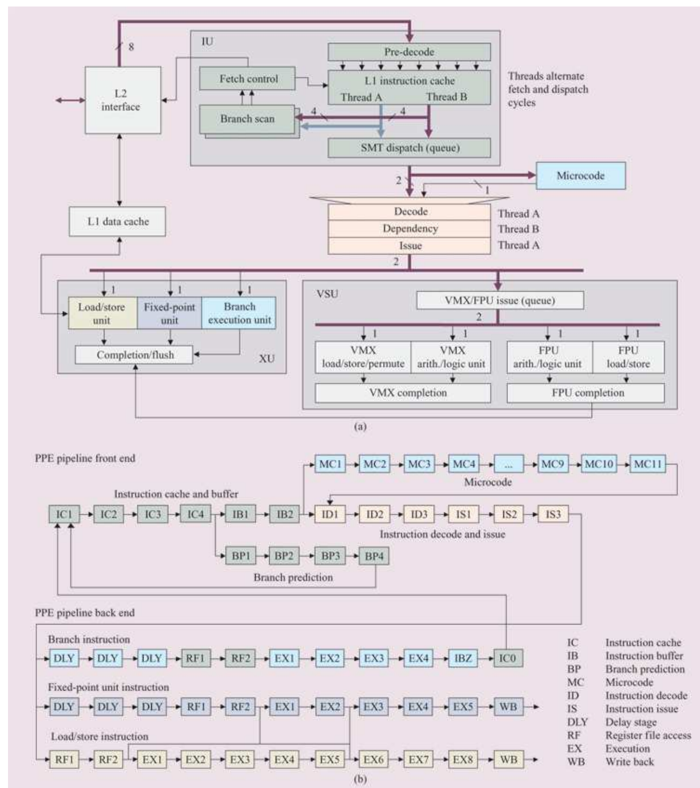

Figura 3.3: Nell'immagine: (a) l'architettura del PPE dal punto di vista logico. (b) composizione logica della pipeline dei comandi

- VSU (Vector Scalar Unit): si occupa del calcolo vettoriale e dei calcoli sui tipi di dato in virgola mobile.

La tecnologia Two-way Multi-processor offre la possibilità di avere a disposizione due thread separati che lavorano contemporaneamente in modo da aumentare le prestazioni, eseguendo più istruzioni contemporaneamente: ciò permette al sistema operativo di vedere il Cell come due processori separati. Inoltre, per come è stato progettato, il PPE non è in grado di "riordinare" la sequenza di istruzioni a run-time (*out-of-order execution* [28]). Al contrario, ad esempio, i processori AMD o Intel eseguono controlli di dipendenza tra le varie istruzioni, in modo da eseguire le istruzioni indipendenti in modo contemporaneo. Nell'architettura Cell non è presente questa funzionalità e quindi l'ordinamento delle istruzioni è stato affidato al compilatore, il quale dovrà ordinare le istruzioni in maniera adeguata, massimizzando le performance.

3.1.4 Il SPE

L'unità SPE è la chiave delle performance del processore Cell. La sua architettura è ottimizzata per favorire il calcolo intensivo, caratteristica richiesta soprattutto nel campo del multimedia e del calcolo scientifico.

Il SPE è un processore indipendente inserito all'interno del processore Cell, che è in grado di comunicare con gli altri SPE o con il PPE tramite il modulo EIB.

Le caratteristiche tecniche principale dello SPE sono:

- Supporto software/hardware per istruzioni SIMD (Single-Instruction, Multiple-Data).
- Potenza massima 5W ad una frequenza di 4GHz.

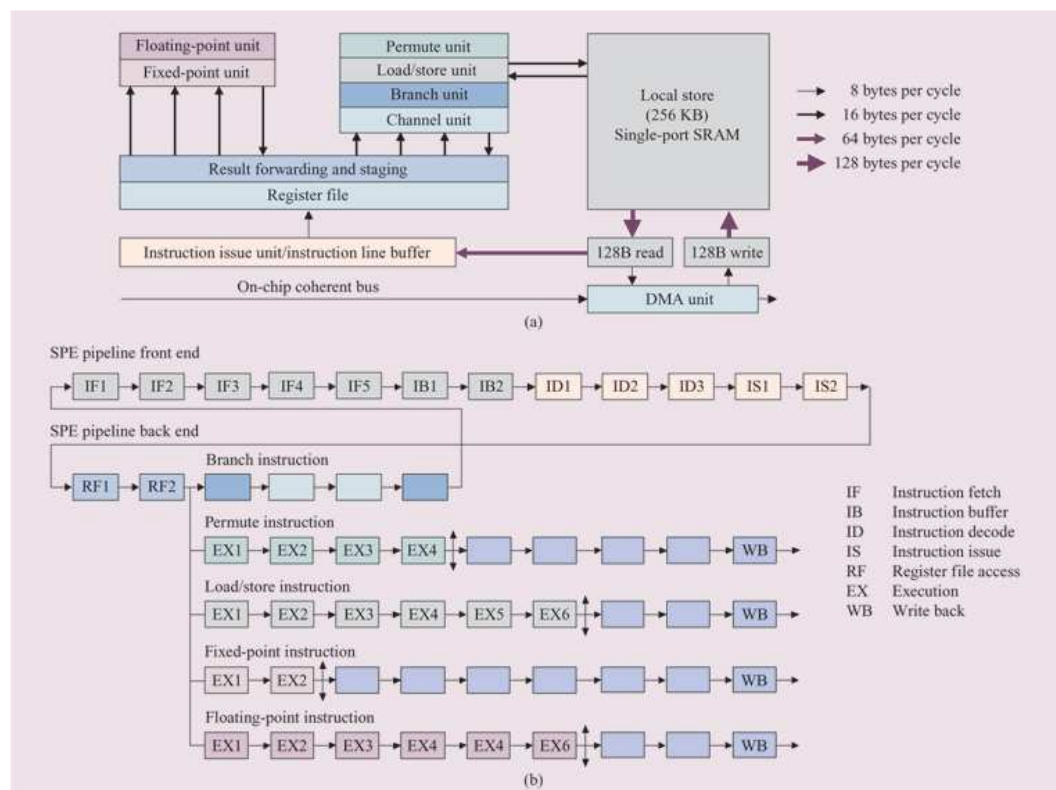

Figura 3.4: In figura: (a) gli elementi presenti nell'architettura di un SPE. (b) il diagramma della pipeline dei comandi di un SPE

L'architettura di un SPE, come si può vedere in figura 3.4, è suddivisa nei seguenti sotto-moduli:

- SPU (Synergistic Processor Unit): un'unità di calcolo "vettoriale" basata su un'architettura di tipo RISC [15] (Reduced Instruction Set Computer), una filosofia secondo la quale un'architettura supporta un ristretto numero di istruzioni molto semplici. Nello SPU si implementa principalmente le funzionalità del calcolo vettoriale su registri a 128 bit.
- LS (Local Store): una memoria di 256 kB inserita in ogni singolo SPE. Contiene tutti i dati utilizzati dal programma più il codice del programma stesso.
- Register file: l'interfaccia tra il programma eseguito e il LS, la quale mette a disposizione 128 registri a 128 bit. Ogni registro del Register File può essere visto in diversi modi a seconda della dimensione del tipo di dato, come si può vedere nella tabella 3.1.
- MFC (Memory Flow Control): è l'unità che gestisce gli accessi di ogni SPE all'EIB. Verrà trattato approfonditamente in 3.1.6.

Tipi di dato	Dimensione	Numero di elementi
int, long int, float, void*	32 bit	4
long long int, double	64 bit	2
short int	16 bit	8
char	8 bit	16

Tabella 3.1: Vari formati di un registro del Register File a seconda del tipo di dato in esso contenuto (per il linguaggio C).

La pipeline dei comandi di un SPE, come si può vedere in figura 3.4, è in grado di eseguire due tipi di istruzioni alla volta: le operazioni su valori in virgola mobile e intere vengono eseguite su una pipeline, mentre le istruzioni di caricamento/salvataggio di dati, la gestione dei blocchi condizionali e le operazioni bit-a-bit, su un'altra pipeline. Le istruzioni intere impiegano solo due cicli di clock per essere eseguite; le istruzioni in virgola mobile, le rotazioni e le operazioni di caricamento dei dati impiegano invece 6 cicli di clock.

Il cuore dello SPE, cioè l'unità di calcolo SPU, implementa una architettura SIMD associata ai registri a 128 bit del Register File, detti anche vettori. L'architettura dello SPU fornisce veloci e semplici funzioni primitive come base di partenza, con le quali il compilatore, cioè la componente software in grado di creare programmi che girano sul SPE, implementa funzioni di più alto livello, usate dagli sviluppatori per implementare gli applicativi software. L'architettura SIMD permette inoltre di introdurre la capacità di parallelizzare

i calcoli, eseguendo gli stessi contemporaneamente su un insieme di dati tra loro indipendenti.

3.1.5 L'EIB

L'EIB (Element Interconnect Bus) è il cuore del sistema di comunicazione tra tutti gli elementi del processore Cell. L'EIB collega il PPE, tutti gli SPE, ll'interfaccia della memoria principale del sistema e l'interfaccia di input/output. L'EIB, essendo un bus, è composto da più linee di comunicazione, in particolare è suddiviso in linee per i comandi, che gestiscono richieste di trasferimento di dati fra gli elementi, e linee dati, che trasportano i dati veri e propri. L'EIB consiste di una rete di 4 connessioni ad anello da 16-byte: 2 anelli trasferiscono dati in senso orario, gli altri 2 in senso antiorario.

L'EIB processa in maniera arbitraria le richieste di dati e decide quale anello deve gestire la richiesta. La scelta dell'anello avviene in modo tale che la richiesta compia il più breve percorso possibile verso la sua destinazione. Inoltre, l'EIB può applicare ad ogni richiesta una priorità in modo da limitare gli "stalli" in fase di lettura da parte del controllore della memoria centrale.

L'EIB opera ad una frequenza pari alla metà di quella dell'intero processore. L'EIB può trasferire 16-byte di dati ad ogni ciclo di clock del bus. Se consideriamo un processore Cell che lavora ad una frequenza di 3.2GHz, il picco di banda teorico di un trasferimento dati può arrivare 300GB/s. La larghezza di banda a cui lavora l'EIB dipende dai seguenti fattori:

- Tutti i richiedenti accedono alla stessa destinazione contemporaneamente, come ad esempio la stessa area di memoria sul LS di un SPE.
- Tutti i trasferimenti sono nella stessa direzione, sovraccaricando 2 dei 4 anelli disponibili.
- Sono stati effettuati un grande numero di trasferimenti parziali da parte degli elementi collegati all'EIB.
- Tutti i trasferimenti devono compiere metà percorso per raggiungere la propria destinazione.

3.1.6 L'MFC

L'MFC (Memory Flow Control) è il modulo che collega ogni LS degli SPE con l'EIB. Il suo compito è gestire i trasferimenti di dati tra il LS e tutti i restanti elementi del processore Cell. La caratteristica principale di questo modulo è il supporto delle istruzioni DMA (Direct Memory Access) che permettono il

caricamento/salvataggio dei dati presenti in memoria centrale nel LS di ogni SPE.

Ogni MFC contiene una coda dei comandi DMA in quanto ogni elemento interconnesso all'EIB può inviare comandi DMA ad uno specifico MFC contemporaneamente. Questo permette inoltre ad ogni SPE di continuare l'esecuzione del proprio programma in maniera indipendente dall'MFC. L'introduzione della coda dei comandi DMA permette all'MFC di operare in totale autonomia in maniera efficiente: l'MFC, infatti, cerca di determinare la migliore sequenza di esecuzione delle istruzioni DMA in modo da ridurre i tempi di accesso alla memoria centrale e al LS di ogni SPE.

Il modulo MFC e in generale l'architettura del processore Cell, nella sua prima generazione, impone alcuni limiti da tenere ben in considerazione quando si sviluppa una applicazione: i dati da trasferire devono essere "allineati", cioè l'area di memoria da loro occupata deve iniziare ad un indirizzo multiplo di un fattore di alineamento, e devono essere della dimensione di 1, 2, 4, 8 Byte o un multiplo di 16 Byte, fino ad una massimo di 16 kByte. Il fattore di allineamento dell'architettura Cell è tipicamente 16 ma può essere anche 128 quando sono richieste maggiori performance: ciò deriva dal fatto che i trasferimenti DMA via EIB trasferiscono blocchi di dati da 128 bit; quando la dimensione dei dati da trasferire è piccola è consigliato 128.

Il modulo MFC permette inoltre la comunicazione tra i vari elementi del processore Cell. I principali metodi di comunicazione sono:

- Notifiche tramite segnali: due canali dedicati tra PPE ed ogni SPE che sfruttano la funzionalità Memory Mapped I/O per la comunicazione. Questo metodo offre svariate possibilità di comunicazione: ad esempio, è possibile notificare un segnale ad un singolo modulo dell'architettura o a N moduli contemporaneamente; inoltre, è possibile eseguire operazioni logiche tra segnali.

- Mail: un canale in grado di trasferire messaggi di 32 bit tra tutti gli elementi del processore Cell. La comunicazione tramite mail è molto veloce (meno di 300ns) ed è l'ideale per la comunicazione uno-a-uno tra i moduli dell'architettura.

- Operazioni atomiche: permette la creazione di sistemi di sincronizzazione tra un SPE e il PPE o tra più SPE. Questo processo avviene tramite il bloccaggio di apposite aree di memoria a cui un singolo modulo può accedere; in caso di accesso non autorizzato, un apposita notifica tramite un segnale viene spedita al SPE che ha bloccato l'area di memoria, in modo da eseguire istruzioni per la gestione dell'errore.

3.1.7 L'I/O

La gestione dell'input/output sul processore Cell è affidata a due elementi, collegati all'EIB, che fanno da interfaccia verso l'esterno del processore: questi due elementi sono il MIC (Memory Interface Controller) e il BIC (Bus Interface Controller).

Il MIC si occupa di controllare tutti gli accessi alla memoria principale di sistema (RAM). Il modulo MIC fa da interfaccia tra l'EIB e il vero gestore della memoria principale, che è un modulo XDRAM Controller sviluppato da Rambus, una società che si occupa della creazione e dello sviluppo di tecnologie atte al miglioramento delle prestazioni delle memorie principali di sistema.

Il modulo BIC invece, consente il collegamento del processore Cell con altri processori Cell in modo da formare una rete. La connessione avviene tramite un modulo Rambus RRAC FlexIO che permette il funzionamento in due modalità differenti: la prima permette la gestione di tutto l'hardware di un computer tramite appositi supporti hardware esterni (i cosiddetti North Bridge e South Bridge presenti in qualsiasi scheda madre, adeguatamente sviluppati per interfacciarsi con il processore Cell); la seconda modalità, invece, permette di collegare più processori Cell come si può vedere in figura 3.5.

3.2 Modelli di programmazione

Data l'architettura atipica del processore Cell è opportuno descrivere i principali modelli di programmazione, dato che la presenza di più unità di calcolo, il supporto SIMD, la particolare architettura dello SPE e le restanti componenti del processore Cell non sono facili da trovare in altre architetture standard, come per esempio la nota architettura x86 o l'architettura PowerPC, presenti su ogni normale computer. Il problema è che una tale quantità di innovazioni può permettere altissime prestazioni solo se queste vengono utilizzate nella maniera corretta.

Il primo aspetto da tenere in considerazione quando si studia un modello di programmazione per il Cell è la presenza del LS in ogni SPE: essendo una memoria indipendente bisogna prevedere che l'applicativo gestisca tale memoria al meglio, caricando o salvando i dati in memoria centrale nella maniera corretta e più performante possibile.

Altro aspetto da tener bene in considerazione è la natura SIMD dello SPE, dato che il calcolo vettoriale può permettere un alto grado di parallelismo. Inoltre lo sviluppatore deve tenere in considerazione il fatto che un programma standard, cioè un programma che gira su architetture non-Cell, può essere "vettorializzato", ovvero che i suo calcoli possono essere trasformarti da

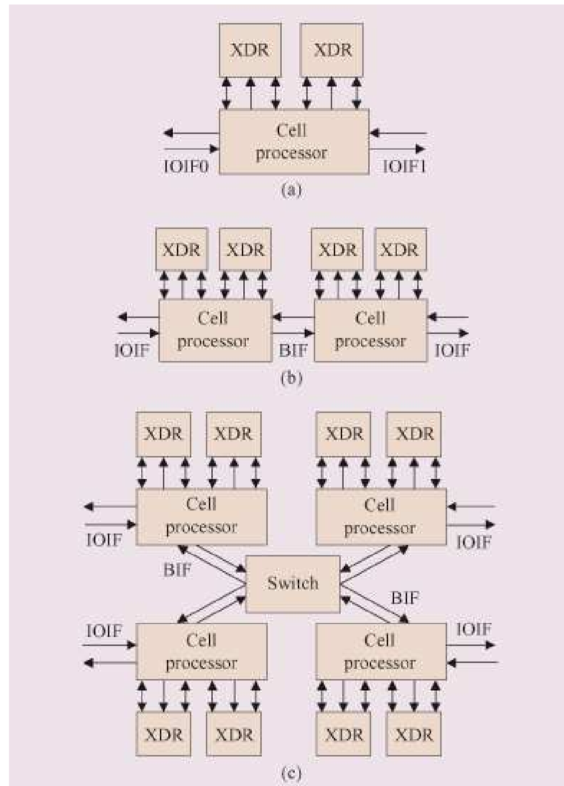

Figura 3.5: Possibili configurazioni offerte dal modulo Rambus FlexIO per il collegamento multiprocessore: (a) configurazione base per piccoli sistemi. (b) configurazione "Glueless" per avere due processori simmetrici. (c) configurazione a multi-processore simmetrico a 4 vie

scalari a vettoriali, in modo da creare parallelismo ed aumentare le prestazioni. Ovviamente, questa "conversione" deve essere adeguatamente studiata e progettata.

Infine, lo sviluppatore deve tenere in considerazione che ogni SPE può gestire un solo contesto alla volta, ovvero un SPE non può passare da un applicativo all'altro in modo rapido. Questa funzionalità, detta *context switching*, non è stata prevista nell'architettura dello SPE.

Un SPE è adatto al calcolo di problemi ripetitivi, ad esempio i problemi di tipo multimediale (compressione audio/video, calcolo vettoriale, ecc...) oppure a problemi di supervisione, nel caso l'applicativo fosse un sistema operativo.

Detto questo passiamo a descrivere i principali modelli di programmazione che possono essere utilizzati sull'architettura Cell, di cui si può trovare una approfondita descrizione in [22]:

- Modello Function Offload: in questo modello le unità SPE vengono utilizzate solo per velocizzate certi tipi di operazioni intensive, le quali penalizzerebbero fortemente le prestazioni se svolte sul PPE. Questo modello è il più semplice tra tutti ed è molto utile quando si esegue il cosiddetto "porting" da una piattaforma standard alla piattaforma Cell. L'applicativo può continuare a girare sul PPE e su tutti gli altri SPE; le sezioni critiche e quelle che richiedono alte prestazioni, invece, vengono riscritte in modo da essere eseguite su uno o più SPE.

- Modello Device Extension: questo modello è un particolare tipo di modello Function Offload; un determinato SPE si occupa di controllare una periferica di sistema e viene trasformato, quindi, in un interfaccia tra il PPE e la periferica stessa, lasciando così al PPE la possibilità di continuare l'esecuzione dei comandi successivi.

- Modello Computational Acceleration: questo modello mette al centro dell'applicativo il SPE; la funzione del PPE diviene quella di controllore di tutti gli SPE e di assegnare i vari task ad ogni SPE.

- Modello Streaming: questo modello viene utilizzato principalmente nel multimedia, soprattutto nella gestione dei dati audio e video. Il lavoro viene suddiviso equamente tra tutti gli SPE, i quali possono essere utilizzati in varie configurazioni a seconda del tipo di streaming da gestire. Al contrario del modello Function Offload, i dati possono essere scambiati tra i vari SPE senza la supervisione del PPE.

- Modello a memoria condivisa: tutte le unità, PPE ed SPE, eseguono operazioni sulla medesima area di memoria; come logico, in questo modello, deve essere implementata una politica di gestione della memoria condivisa.

- Modello Asymmetric Thread Runtime: ogni SPE viene visto come un semplice processore in grado di gestire un diverso applicativo; questo è il modello più comune di implementazione quando si hanno più applicativi indipendenti che possono essere eseguiti contemporaneamente.

Indipendentemente dal modello di programmazione scelto, all'atto pratico ci sono alcuni modi diversi per sviluppare applicazioni in grado di girare su architetture Cell. I linguaggi di programmazione utilizzabili sono:

- Codice assembly: il linguaggio di livello più basso; utilizzabile sia sul PPE sia sul SPE, permette la gestione completa di tutta l'implementazione dell'applicativo. Viene usato per la scrittura di applicativi semplici

o per implementazioni di algoritmi a bassissimo livello dato che tutta l'implementazione è lasciata allo sviluppatore. Generalmente, l'uso di questo linguaggio, avviene solo sul lato SPE.

- Codice C/C++: un altro linguaggio a basso livello, ma comunque ad un livello superiore al codice assembly, utilizzabile sia sul PPE, sia sul SPE. Offre moltissima libertà di implementazione e gode della possibilità di essere ottimizzato dal compilatore. Nel caso specifico del Cell, in particolare del SDK (Software Development Kit, insieme di software predisposto alla creazione e allo sviluppo di applicativi), sono disponibili due compilatori diversi: il primo è GCC (Gnu C Compiler), un compilatore multi-piattaforma molto conosciuto dato che ne esiste una versione per quasi tutte le architetture esistenti al mondo; l'altro compilatore invece è XLC, il compilatore proprietario di IBM, meno conosciuto ma anch'esso molto utilizzato. Entrambi i compilatori offrono buone prestazioni anche se, in base ai test svolti, il compilatore XLC crea codice leggermente più veloce di GCC. Inoltre, per questo linguaggio, sono disponibili molte librerie per facilitare l'implementazione di programmi sia sul lato PPE sia sul lato SPE.
  Il linguaggio C è stato scelto per l'implementazione del progetto che viene trattato in questa tesi.

3.2.1 Il supporto SIMD

Il supporto SIMD (Single-Instruction, Multiple-Data) è la principale caratteristica dello SPE. La filosofia che sta dietro ad una architettura SIMD è molto semplice: l'esecuzione contemporanea di uno stesso calcolo o istruzione (Single-Instruction) su un certo numero di dati indipendenti dello stesso tipo (Multiple-Data). Seguendo questa filosofia, il SPE implementa il calcolo vettoriale sui registri del proprio Register File. Facciamo un esempio per rendere semplice la comprensione di questa caratteristica: ammettiamo di avere due registri all'interno del Register File, formati da 4 interi a 32 bit, che possiamo rappresentare come in 3.6; ogni elemento del registro, cioè ogni intero al suo interno, è totalmente indipendente da tutti gli altri elementi del vettore; la filosofia SIMD impone che la somma dei due registri in 3.6 sia eseguita sommando ogni singolo elemento del primo registro con il corrispettivo del secondo registro.

Sul processore Cell, come già detto, la tecnologia SIMD è supportata sia a livello hardware sia a livello software. A livello hardware sono implementate tutte le istruzioni più semplici quali ad esempio le operazioni aritmetiche (somma, prodotto, differenza, ecc...), le operazioni bit-wise, che operano a livello

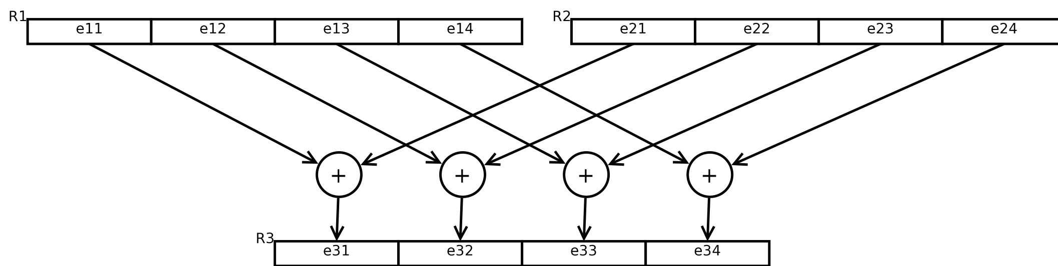

Figura 3.6: Esempio di somma di due registri del Register File in cui sono inseriti 4 interi da 32 bit.

di bit, e in generale tutte le più comuni funzioni disponibili anche per i valori scalari. Al livello software, tramite le funzioni di libreria fornite con il kit di sviluppo del Cell, sono state implementate tutte le operazioni più complesse quali, ad esempio, le operazione trigonometriche (coseno, seno, ecc...), le quali possono essere utilizzate dallo sviluppatore per scrivere funzioni più complesse. Per una lista completa di tutte le funzioni SIMD disponibili si veda [16, 17].

La natura RISC e SIMD dello SPE impongono però una problematica data dalla gestione dei valori scalari. Come gestire questi valori in una architettura mirata a principalmente al calcolo vettoriale? Per la risoluzione di questo problema sono stati introdotti nel Cell i concetti di *Scalar Layering* e di *Preferred Slot*.

Lo Scalar Layering è la suddivisione di un vettore nelle sue unità scalari, cioè la suddivisione di un vettore in 16 unità da da 8 bit, 8 unità da 16 bit, 4 unità da 32 bit, 2 unità da 64 bit o 1 unità da 128 bit.

Il concetto di Preferred Slot, invece, deriva da un limite imposto dal Register File e dal LS di un SPE. Il Register File è in grado di caricare dati dal LS, solo se questi ultimi si trovano ad indirizzi che sono multipli di 16 Byte, ovvero 128 bit, cioè la dimensione di un vettore. Se per esempio un valore scalare si trova all'indirizzo 12 non è possibile per il Register File caricare solo l'area di memoria dello scalare ma deve caricare l'unità da 16-Byte che lo contiene: in questo caso il Register File caricherà l'unità con indirizzo 0. Se consideriamo poi la posizione dello scalare all'interno del registro, esso dovrà essere allineato al 4° byte, se la sua dimensione è inferiore o uguale a 4 byte, all'8° byte se la sua dimensione è pari a 8 byte oppure non dovrà essere allineato se la sua dimensione è 16 byte. Il concetto di Preferred Slot è mostrato graficamente in figura 3.7.

Come conseguenza dei concetti dello Scalar Layering e del Preferred Slot, abbiamo che ogni valore scalare che viene memorizzato all'interno di un registro vettoriale provoca uno spreco di memoria che può diventare non indifferente se il SPE deve gestire un grande numero di valori scalari. Nonostante ciò, la

decisione da parte degli sviluppatori del Cell è stata quella di implementare il supporto a valori scalari, dato che ci sono molte operazioni, quali ad esempio l'indirizzamento di un array, che richiedono l'uso di valori scalari.

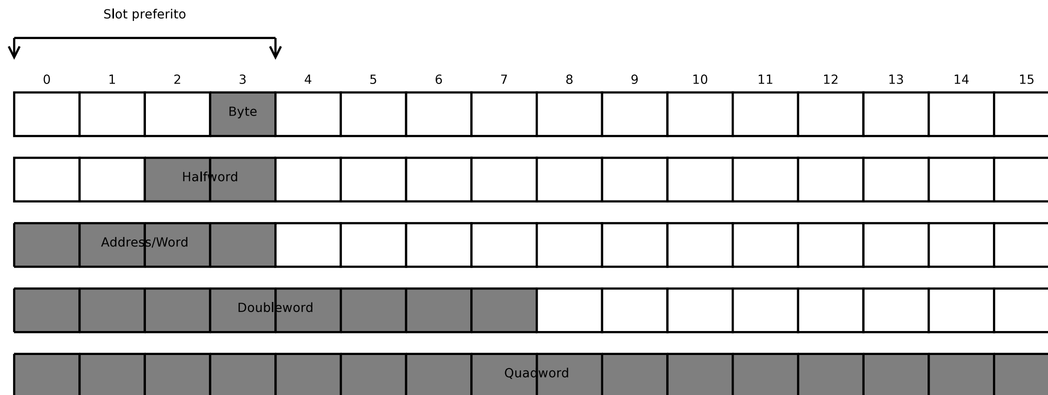

Figura 3.7: Disposizione del tipo di dato scalare all'interno di un registro del Register File di un SPE

3.3 Confronto con architetture standard

Per confrontare l'architettura Cell con le altre architetture commerciali iniziamo con il considerare la caratteristica *multicore*. Si definisce multicore un'architettura la quale prevede l'inserimento di più unità di calcolo all'interno di uno stesso processore.

L'introduzione della tecnologia multicore sul mercato è dovuta al processore Cell, il quale, con la sua uscita nel 2005, ha dato il via allo sviluppo di nuovi processori multicore anche da parte di Intel, prima, e AMD, poi.

Attualmente, sul mercato, sono presenti processori multicore tra cui ricordiamo la serie di processori *Intel Core duo*, *Intel Core 2 duo* a doppio core di calcolo, *Intel Core 2 quad* con quattro core di calcolo, *Intel XEON* a doppio core, *AMD Athlon 64 X2* a due core di calcolo e *AMD Opteron* a 1, 2 o 8 core di calcolo. Ricordiamo inoltre l'architettura multicore *PowerPC* di XBOX 360, la diretta concorrente di Playstation 3.

La differenza tra questi processori e l'architettura Cell sta nell'approccio adottato per l'implementazione multicore. I processori multicore di Intel e AMD "replicano" l'unità centrale di calcolo dell'architettura inserendo più core dello stesso tipo nel processore.

Per quanto riguarda il processore Cell, invece, abbiamo più core di calcolo ma di diverso tipo. Infatti, ricordando quanto detto nelle sezioni precedenti, il processore Cell introduce un unità PPE e 8 unità SPE.

Questa differenziazione rispetto alle architetture multicore standard di AMD e Intel sono il vero punto di forza del processore Cell.

Il vantaggio di avere due tipi diversi di core in una stessa architettura sta nel rapporto $performance/watt$. Ricordiamo, infatti, che gli SPE sono di tipo RISC, un'architettura la cui semplicità favorisce il basso consumo energetico: se associamo quindi un'architettura standard come quella del PPE ad un certo numero di unità SPE possiamo aumentare le potenzialità di calcolo mantenendo però un basso consumo energetico. Se consideriamo invece gli approcci multicore di AMD e Intel possiamo facilmente notare che il rapporto $performance/watt$ è inferiore rispetto al rapporto fornito dal Cell.

Consideriamo ora il "fattore pratico" cioè cosa cambia dal punto di vista di una applicazione che deve girare su architetture AMD o Intel rispetto al processore Cell.

I processori AMD e Intel implementano la funzione di *out-of-order execution [28]*, la possibilità di ordinare le istruzioni in diverse pipeline comandi in modo che esse vengano eseguite nella migliore sequenza possibile, massimizzando le performance. Questa funzionalità, associata al mantenimento dello stesso set di istruzioni delle precedenti architetture, permette ad un programma di beneficiare fin da subito della disponibilità di più unità di calcolo.

Nel caso del Cell, questa funzionalità non è stata implementata principalmente perché richiede una notevole quantità di energia; al suo posto si è sviluppata una soluzione di scheduling a *compile-time*, la quale richiede che sia il compilatore a generare la migliore sequenza di esecuzione delle istruzioni, quella sequenza che permette le migliori performance di esecuzione del programma. Nel caso del Cell, quindi, il compilatore utilizzato è di fondamentale importanza, dato che "definisce" pesantemente le prestazioni che il programma finale avrà.

Altra differenza è il supporto SIMD offerto dalle architetture AMD o Intel rispetto al Cell. Consideriamo, ad esempio, il set di istruzioni SIMD SSE [23](Streaming SIMD Extension) introdotto nei processori AMD e Intel. Questa estensione introduce alcune nuove istruzioni nel set e inserisce nuovi registri all'interno dell'unità centrale del processore per il calcolo in virgola mobile. Questo approccio è nettamente diverso dal Cell nel quale, come detto, tutte le funzionalità SIMD sono state implementate in ciascun SPE. In entrambi i casi un programma che dovrà fare uso di istruzioni SIMD dovrà essere modificato in maniera adeguata. Nel caso del Cell, però, il programma in questione dovrà prevedere anche e soprattutto il supporto dato dalle unità SPE per implementare il calcolo SIMD.

Nel capitolo 4 descriveremo l'implementazione Cell dell'Algoritmo di Metropolis multi-spin coded e vedremo quali saranno le modifiche che l'imple-

mentazione x86 dovrà subire per sfruttare tutte le potenzialità del processore Cell.

Capitolo 4

Implementazione su architettura Cell-BE

4.1 Requisiti dell'implementazione

Prima di trattare l'implementazione su architettura Cell è necessario definire i requisiti richiesti per l'implementazione. L'implementazione sull'architettura Cell deve corrispondere ad un aumento di prestazione del calcolo di spin glass rispetto all'architettura x86; dobbiamo progettare, quindi, un'implementazione che sia più performante e che sia in grado di calcolare un singolo spin in un tempo inferiore a quello ottenuto nell'implementazione x86. Per ottenere ciò è richiesto di esplorare le potenzialità date dalle unità SPE e, in particolare, dal supporto SIMD.

L'uso di uno o più SPE implica che si scelga uno tra i modelli di programmazione descritti nella sezione 3.2 oppure che si crei un modello ad-hoc da utilizzare per l'implementazione. Inoltre, per quanto riguarda il SPE, il supporto SIMD gode di maggiore velocità rispetto al semplice calcolo scalare per cui l'implementazione dovrà essere vettorializzata il più possibile.

L'altro fattore fondamentale è la gestione della memoria, un problema che si rivela principalmente sul lato SPE. Ad esempio, per ridurre i tempi ed aumentare le performance, si può ridurre al minimo lo scambio di dati via DMA tra PPE ed SPE e adibire gran parte della memoria del LS ai dati che descrivono il reticolo di spin. Ciò implica che il codice che viene eseguito sul SPE, contenuto anch'esso nel LS, deve essere di dimensioni ridotte, senza andare a scapito delle performance.

Infine, l'implementazione sul Cell richiede che si faccia utilizzo della tecnica del multi-spin coding introdotta nel capitolo 2. In questo caso però, sarà necessario agire su parole a 128 bit, ovvero la dimensione di un vettore.

Fatte queste considerazioni, passiamo a descrivere l'algoritmo utilizzato nella nostra implementazione.

4.2 Algoritmo per architettura Cell-BE

L'algoritmo 2 può essere portato senza particolari problemi sull'architettura Cell-BE, più precisamente, nella nostra implementazione, sarà eseguito da un singolo SPE, come impone il modello Function Offload descritto in 3.2. L'utilizzo del PPE associato ad un SPE, impone però delle aggiunte all'algoritmo originale, come si può vedere dagli algoritmi 3 e 4.

Algorithm 3 Algoritmo sul lato PPE.

for $\{\beta = 0,20; \beta <= 0,25; \beta += 0.005\}$ **do**
```
  loadConfiguration()
  startSPE()
  sendMail(Spins, Jx, Jy, Jz)
```
  for $i \rightarrow 0 \cdots 1000$ **do**
```
    waitMail()
    calculateMagnetizations()
    sendMail(go = 1)
    printBetaAndMagnetizations()
```
  end for
```
  waitMail()
  sendMail(go = 0)
```
end for

Il ciclo più esterno specifica i parametri della simulazione; come si può vedere viene eseguita una simulazione per cento valori di β compresi tra 0.20 e 0.25. Si noti che la configurazione viene ricaricata per ogni valore di β in modo da poter confrontare le varie simulazioni tramite valori coerenti, dato che l'esecuzione del programma sul SPE cambia i valori di *Spins* in memoria centrale.

L'istruzione `loadConfiguration()` permette di "generare" tutti i dati relativi al reticolo di spin, cioè le variabili `Spins`, `Jx`, `Jy` e `Jz`.

L'istruzione `startSPE()` esegue tutte le procedure tipiche per l'inizializzazione e l'uso di un SPE.

L'istruzione `sendMail(Spins, Jx, Jy, Jz)` invia una mail contenente i parametri da usare al SPE che si occuperà del calcolo. Nell'implementazione si vedrà che, in realtà, a essere spedito sarà l'indirizzo di memoria dove trovare i parametri, dato che ricordiamo che una mail può essere di soli 32 bit; inoltre,

questa istruzione segnala l'inizio della simulazione al SPE il quale, dopo aver ricevuto la mail e caricato i dati, inizia l'esecuzione della sua parte di codice.

Il ciclo più interno si occupa di far eseguire un numero adeguato di iterazioni, nel nostro caso 1000 ma possono essere superiori. Questo fattore è molto importante nel fattore della "statistica": come detto nella sezione 1.2, più il numero di iterazioni è alto più aumenta l'efficace del Metodo Monte Carlo; il numero di iterazioni qui presente, insieme al numero di iterazioni eseguito sul lato SPE rappresentano i principali fattori della simulazione.

Segue poi l'istruzione `waitMail()` che attende la fine del calcolo e le restituzione dei dati computati dal SPE; a questo punto i dati passano nelle mani del PPE, mentre il SPE si mette in attesa della mail che gli consentirà di proseguire il calcolo.

L'istruzione `calculateMagnetizations()`, come dice il nome stesso, si occupa di calcolare i valori delle magnetizzazioni ed insieme a `printBetaAndMagnetizations()` vanno a formare il risultato della simulazione. In questo modo, per ogni valore di β avremo un numero di magnetizzazioni pari al numero di iterazioni delle simulazione che vengono eseguite dal PPE.

L'istruzione `sendMail(go = 1)` termina l'attesa sul lato SPE, il quale riprende la simulazione da dove è stata interrotta.

Infine, appena fuori dal ciclo, abbiamo le istruzioni di uscita delle simulazioni per un valore di β. La prima istruzione, `waitMail()`, attende che finisca l'ultima simulazione sul lato SPE, la quale verrà scartata; infine abbiamo l'istruzione `sendMail(go = 0)` che comunica al SPE di terminare la simulazione e di uscire definitivamente dal suo programma.

Algorithm 4 Algoritmo sul lato SPE.

```
loadConfiguration()
go → 1
for go == 1 do
  for i → 1 ··· 1000 do
    metropolisUpdate()
  end for
  saveConfiguration()
  sendMailToPPE()
  go → waitMailFromPPE()
end for
```

Vediamo ora cosa succede sul lato SPE, dove è presente il vero e proprio Algoritmo di Metropolis.

L'istruzione `loadConfiguration()` carica la configurazione dalla memoria

centrale e invia la mail al PPE dell'avvenuta lettura dei parametri per la simulazione.

La riga successiva, $go \rightarrow 1$, permette la prima esecuzione di una simulazione sul lato SPE.

Il ciclo più esterno, presente nell'algoritmo 4, controlla il numero di iterazioni eseguite sul lato SPE. Al suo interno avviene la chiamata all'istruzione `metropolisUpdate()`, l'istruzione che si occupa della computazione di Metropolis. L'algoritmo di questa istruzione è pressoché identico all'algoritmo 2.

L'istruzione `saveConfiguration()` trasferisce i dati computati in memoria centrale e li rende disponibili al PPE, il quale viene avvertito tramite la mail inviata dall'istruzione `sendMailToPPE()`.

Infine, l'istruzione $go \rightarrow$ `waitMailFromPPE()` attende che il PPE abbia calcolato i valori di magnetizzazione; inoltre la variabile go indicherà al SPE quando continuare o quando uscire dalla simulazione.

4.3 Implementazione per architettura Cell-BE

Per l'implementazione di Metropolis su architettura Cell, la scelta del linguaggio di programmazione è ricaduta sul linguaggio C. Le motivazione di questa scelta sono le seguenti:

- E' un linguaggio molto conosciuto ed è quindi possibile trovare molta documentazione, molto esempi ed è facile avere supporto.

- E' un linguaggio molto tecnico che permette una grande flessibilità di programmazione, soprattutto a livello di memoria.

- E' un linguaggio molto ben supportato dal kit di sviluppo fornito da IBM per il processore Cell: oltre alle librerie standard presenti nel linguaggio C, sono presenti tutte le librerie per il supporto SIMD e per la gestione e l'uso degli SPE.

In questa sezione faremo largo uso delle istruzioni delle istruzioni del linguaggio C [21] e delle relative istruzioni SIMD: si consiglia, quindi, la lettura di [16,17]. Passiamo ora a descrivere l'implementazione sui lati PPE ed SPE. Il codice sorgente completo di alcune delle versioni finali sviluppate sono disponibili nelle appendici B, C e D.

4.3.1 Lato PPE

Come imposto dal modello Function Offload descritto in 3.2, il PPE ha come scopo principale quello di fare da controllore all'esecuzione sul SPE. Tra le altre funzioni del PPE ci sono:

- Inizializzazione di tutte le variabili del reticolo e dei dati di simulazioni.
- Passaggio dei parametri di simulazioni al SPE.
- Calcolo dei risultati finali.

Il cuore dell'Algoritmo di Metropolis sul lato PPE è rappresentato dalla funzione `user_main()`, visualizzabile nelle appendici B, C e D. Il codice sorgente di questa funzione è:

```
1  void user_main (spe_program_handle_t *
      program_pointer) {
2    unsigned i, j;
3    for(beta = BETA_START; beta < BETA_STOP; beta +=
        BETA_INC) {
4      loadConfigurationFromFile();
5      resetMagnetizationsVars();
6      startSPE(program_pointer);
7      sendDataMailToSPE();
8      for(i = 0; i < PPE_ITERATIONS; i++) {
9        waitMailFromSPE();
10       calculateMagnetization();
11       sendGoMail();
12     }
13     endMagnetizationCalculus();
14     printBetaAvgMagnAndSigma()
15     sendDontGoMail();
16     stopSPE();
17   }
18 }
```

Passiamo ora ad analizzare come vengono implementate le principali funzioni di questo codice. La prima, l'inizializzazione dei dati di simulazione, avviene tramite la funzione `loadConfigurationFromFile()` il cui codice sorgente è:

```
1  void loadConfigurationFromFile() {
2    set_defaults();
3    init_random();
```

```
4   init_spins();
5   init_couplings();
6
7   RUN_STATUS_LIST.status=0;
8
9   SPE_PARAMS_LIST.Spins_addr     = (ADDR)Spins;
10  SPE_PARAMS_LIST.Jx_addr        = (ADDR)Jx;
11  SPE_PARAMS_LIST.Jy_addr        = (ADDR)Jy;
12  SPE_PARAMS_LIST.Jz_addr        = (ADDR)Jz;
13  SPE_PARAMS_LIST.ira_addr       = (ADDR)ira;
14  SPE_PARAMS_LIST.run_ticks_addr = (ADDR)&(
       RUN_TICKS_LIST);
15  SPE_PARAMS_LIST.sg_param_addr  = (ADDR)&
       SG_PARAMS_LIST;
16
17  SG_PARAMS_LIST.T_4             = (1 / (beta * 4));
18  SG_PARAMS_LIST.C_inv_max_rand  = C_inv_max_rand;
19  SG_PARAMS_LIST.ip              = ip;
20  SG_PARAMS_LIST.ip1             = ip1;
21  SG_PARAMS_LIST.ip2             = ip2;
22  SG_PARAMS_LIST.ip3             = ip3;
23  SG_PARAMS_LIST.irr             = irr;
24  SG_PARAMS_LIST.iter            = SPE_ITERATIONS;
25 }
```

La funzione `set_defaults()` imposta i valori di default utilizzati nella simulazione. La funzione `init_random()` inizializza la tabella iniziale dei numeri random, come viene imposto nella sezione 1.2.2, la quale verrà poi trasferita allo SPE; la funzione `init_random()` inizializza inoltre i vari indici della tabella.

Le funzioni `init_spins()` e `init_couplings()` inizializzano le variabili e le costanti del reticolo, ovvero le variabili `Spins`, `Jx`, `Jy` e `Jz`; si noti la definizione di quest'ultime nel e la definizione di `ira` seguente codice sorgente:

```
1 unsigned int Spins[VOLUME][4] __attribute__ ((aligned
     (128)));
2 unsigned int Jx   [VOLUME][4] __attribute__ ((aligned
     (128))),
3              Jy   [VOLUME][4] __attribute__ ((aligned
                 (128))),
4              Jz   [VOLUME][4] __attribute__ ((aligned
                 (128)));
```

```
5 unsigned int ira[256][4]      __attribute__ ((aligned
    (128)));
```

Le variabili `Spins`, `Jx`, `Jy` e `Jz` sono dichiarate come una matrice. La definizione di queste variabili differisce dalla definizione che verrà usata sul lato SPE in quanto dobbiamo definire ciascuna riga della variabile `Spins` e delle costanti `Jx`, `Jy` e `Jz` come un'area di memoria pari a 128 bit. In questo modo rappresentiamo 128 sistemi diversi di spin al contrario dell'implementazione x86 in cui se ne rappresentavano solo 32. Questa particolare definizione è necessaria dato che `Spins`, `Jx`, `Jy` e `Jz`, definite sul lato PPE e quindi in memoria centrale, verranno utilizzate come vettori sul lato SPE. Anche nel caso della variabile `ira` abbiamo una matrice di 4 colonne dato che, anche questa variabile, rappresenterà un array di vettori sul lato SPE.

Analizziamo ora il blocco di codice che si occupa di passare i parametri di simulazione allo SPE. Il codice sorgente che ha questa funzione è:

```
1  void sendDataMailToSPE() {
2    unsigned int out_mail = 0;
3    ...
4    // High address (mfc_ea2h(...))
5    out_mail = (unsigned int)((unsigned long long int)
       (&(SPE_PARAMS_LIST  ))>>32)&0xffffffff;
6    PPE_SM(THREADS, &out_mail);
7
8    // Low address (mfc_ea2l(...))
9    out_mail = (unsigned int)((unsigned long long int)
       (&(SPE_PARAMS_LIST  )))&0xffffffff;
10   PPE_SM(THREADS, &out_mail);
11 }
```

Dove si può notare che viene inviata una doppia mail attraverso la macro `PPE_SM()`: ricordando che una mail può trasferire solo valori di 32 bit, come detto nella sezione 3.1.6, dobbiamo utilizzare due mail per trasferire i valori di un indirizzo della memoria centrale, il quale ha dimensione pari a 64 bit. Questo indirizzo verrà utilizzato dallo SPE per eseguire un trasferimento DMA dalla memoria centrale al LS dello SPE stesso. Per la definizione della struttura `SPE_PARAMS_LIST` si rimanda alle appendici B, C e D.

Infine, il blocco di codice in cui avviene il calcolo dei risultati finali, cioè:

```
1  void user_main (spe_program_handle_t *
     program_pointer) {
2    unsigned i, j;
```

```
3     for(beta = BETA_START; beta < BETA_STOP; beta +=
         BETA_INC) {
4       ...
5       resetMagnetizationsVars();
6       ...
7       for(i = 0; i < PPE_ITERATIONS; i++) {
8         ...
9         calculateMagnetization();
10        ...
11      }
12      endMagnetizationCalculus();
13      printBetaAvgMagnAndSigma()
14      ...
15    }
16  }
```

Il calcolo della magnetizzazione avviene ad ogni iterazioni sul lato PPE. La funzione `resetMagnetizationsVars()` resetta i valori parziali del calcolo della magnetizzazione. La funzione `calculateMagnetization()` esegue il calcolo parziale della magnetizzazione.

La funzione `endMagnetizationCalculus()` termina il calcolo della magnetizzazione mentre la macro `printBetaAvgMagnAndSigma()` stampa il valore sul file di output.

Per descrivere questo calcolo, si considerino k risultati indipendenti presi a differenze di tempo Δt molto grande. Nel seguito, indicheremo con $< A >$ la media aritmetica di N rilevamenti di una osservabile A, come

$$< A >= \frac{1}{N} \sum_{k=1}^{N} A_k \; .$$

Il risultato finale della nostra simulazione sarà composto dal valore medio della magnetizzazione $< M >$ di ogni singolo sistema dei 128 rappresentati, ovvero 128 magnetizzazioni medie per ogni valore di β. Ogni media sarà calcolata a partire dalle magnetizzazioni intermedie rilevate ad ogni iterazione sul lato PPE. Per ogni media sarà definita una stima dell'errore δM [18], definita come

$$\delta M \simeq \sqrt{\frac{1}{(k-1)} \left(< M^2 > - < M >^2\right)} \; .$$

Passiamo ora a definire ciò che avviene sul lato SPE.

4.3.2 Lato SPE

Quanto detto per l'implementazione per architetture x86 nella sezione 2.2, può essere valido per l'implementazione sul Cell. Il problema è che in questo modo non si sfruttano le vere potenzialità di questa architettura. L'implementazione per architetture x86 deve essere quindi adeguatamente trasformata per poter essere efficace sull'architettura Cell e in particolare sull'unità SPE.

Analizziamo ora i tratti più significativi dell'implementazione della nostra simulazione sul lato SPE. Definiamo le variabili `Spins`, che sostituisce la variabile `Spin` dell'implementazione x86, `Jx`, `Jy` e `Jz` nel seguente modo:

```
1 vector unsigned int Spins[VOLUME] __attribute__ ((
     aligned (128)));
2 vector unsigned int Jx[VOLUME]    __attribute__ ((
     aligned (128)));
3 vector unsigned int Jy[VOLUME]    __attribute__ ((
     aligned (128)));
4 vector unsigned int Jz[VOLUME]    __attribute__ ((
     aligned (128)));
```

La parola chiave `vector` specifica il tipo di dato SIMD supportato dal SPE: ciò permette di portare gran parte dell'implementazione x86 sull'architettura Cell. Tutte le quattro variabili vengono definite come array di lunghezza pari al numero di spin del reticolo, valore specificato tramite l'etichetta `VOLUME`. Il modificatore `__attribute__ ((aligned (128)))` specifica che l'area di memoria adibita ad una specifica variabile deve iniziare ad un indirizzo che sia un multiplo di 128: quest'ultimo punto è caratteristico delle variabili che vengono utilizzate nei trasferimenti DMA da memoria centrale al LS e viceversa; per maggiori dettagli si veda la sezione 3.1.6.

Si noti che, al contrario dell'architettura x86, un registro Cell e quindi una variabile di tipo `vector` ha dimensione 128 bit. Questo permette di trattare 128 sistemi indipendenti adottando la tecnica del multi-spin coding, pur mantenendo la stessa logica d'uso delle operazioni binarie che si è fatto nell'implementazione x86.

La trasformazione in `vector` delle variabili principali che descrivono il reticolo di spin richiede, però, l'uso di funzioni specifiche per il Cell, in particolare l'uso delle funzioni binarie SIMD. Come fatto nella sezione 2.2, analizziamo prima di tutto l'implementazione dell'Algoritmo di Metropolis suddividendolo nelle sue parti fondamentali e analizzando, quindi, queste parti una ad una. Infine analizzeremo parti di invio e ricezione delle mail e la parte adibita ai trasferimenti DMA che il SPE deve eseguire per avere a disposizione tutti i dati da utilizzare nella simulazione.

Iniziamo con il primo blocco cioè il calcolo del numero random e il primo termine del primo membro della 2.7. Il codice che ha questo compito è il seguente:

```
1  vector unsigned int ira[256] __attribute__ ((aligned
      (128)));
2  vector unsigned int p_ips;
3  ...
4  IDX = spu_add(ira[spu_extract(p_ips, 1)], ira[
      spu_extract(p_ips, 2)]);
5  ira[spu_extract(p_ips, 0)] = IDX;
6  IDX = spu_xor(IDX, ira[spu_extract(p_ips, 3)]);
7
8  // Incremento e maschero a 255
9  p_ips = spu_add(p_ips, ones);   p_ips = spu_and(p_ips
      , mask);
10
11 IDXf = spu_mul(spu_convtf(IDX, 0), C_inv_max_rand);
12 IDXf = spu_mul(_log2f4(IDXf), LN_2);
13 IDXf = spu_mul(IDXf, negative_T_4);
14
15 // Forzo a 3 il valore di IDX
16 select = spu_cmpabsgt(IDXf, THREES);
17 IDXf   = spu_sel(IDXf, THREES, select);
18 IDX    = spu_convtu(IDXf, 0);
```

Si noti che qui le variabili `IDX`, `IDXf` e `ira` sono tutte di tipo `vector` cioè variabili vettoriali. Si noti anche che il vettore `ira` è dichiarato come array di vettori e non come matrice quadrata di dimensione 256: questa scelta riduce il periodo del generatore dei numeri casuale ma è imposta dal ridotto spazio disponibile nel LS di un SPE.

Si può riconoscere il generatore di numeri pseudo-casuali a shift-register di Parisi-Rapuano, rappresentato nel codice dalle righe 3, 4 e 5.

L'uso del vettore `p_ips` è stato introdotto per sostituire le variabili `ip`, `ip1`, `ip2`, `ip3` le quali sono state inserite rispettivamente nei quattro elementi del vettore stesso: quest'ultimo è stato introdotto per sfruttare la funzionalità SIMD dato che tutte le quattro variabili `ip`, `ip1`, `ip2`, `ip3` presenti nell'implementazione x86 subivano tutte la stessa operazione, cioè l'incremento di uno.

Si noti che il vettore `p_ips` è di tipo intero, ovvero che la dimensione dei suoi elementi è di 32 bit, al contrario di quanto fatto nell'implementazione x86, in cui le quattro variabili erano di tipo `char`, ovvero di dimensione 8 bit. Ciò

richiede che ogni elemento del vettore `p_ips` venga mascherato a 255, cioè il valore di ogni elemento dev'essere compreso tra 0 e 255, come si vede dalla riga 9.

L'introduzione della variabile `IDXf`, una copia in virgola mobile del valore intero di `IDX`, e delle funzioni di conversione `spu_convtf()` e `spu_convtu()` è richiesto dal fatto che non è possibile eseguire *cast esplicito* che permetterebbe l'uso di `IDX` con `_log2f4()`.

Infine forziamo il valore di `IDX` a 3 tramite il supporto del vettore intero `THREES`, contenente 4 elementi di valore 3; è stato evitato l'uso di un blocco condizionale dato che avremmo dovuto replicarlo 4 volte e si è utilizzata una tecnica detta *branch-free implementation*, sulla quale rimandiamo alla sezione 4.5.1.

Passiamo ora al successivo blocco cioè il calcolo degli indici dei vicini. Il codice che esegue questo calcolo è il seguente:

```
1  vector signed int vSIDE_1 = {SIDE-1, SIDE-1, SIDE-1,
      SIDE-1};
2  vector signed int vSIDE   = { SIDE,  SIDE,  SIDE,
        SIDE};
3  vector signed int vZERO   = {    0,     0,     0,
           0};
4  unsigned int inSpin __attribute__ ((aligned (4)));
5  vector signed int multiplier = {1, SIDE, FACE, 0};
6  ...
7  for(z=0; z<SIDE; z++) {
8    for(y=0; y<SIDE; y++) {
9      for(x=0; x<SIDE; x++) {
10       vector signed int xyz   = { x, y, z, 0 };
11       vector signed int adder = { (y * SIDE) + (z *
            FACE), x + (z*FACE), x + (y * SIDE), 0 };
12
13       inSpin = (x + (y * SIDE) + (z * FACE));
14       ...
15       // Coordinate positive
16       xyzp    = spu_add(xyz, 1);
17       pos_sel = spu_cmpeq(xyzp, SIDE);
18       xyzp    = spu_sel(xyzp, vZERO, pos_sel);
19
20       // Coordinate negative
21       xyzm    = spu_add(xyz, -1);
22       neg_sel = spu_cmpgt(vZERO, xyzm);
```

```
23          xyzm     = spu_sel(xyzm, vSIDE_1, neg_sel);
24
25          xyzp = spu_madd((vector signed short)xyzp, (
                vector signed short)multiplier, adder);
26          xyzm = spu_madd((vector signed short)xyzm, (
                vector signed short)multiplier, adder);
27          ...
28       }
29    }
30 }
```

Questa implementazione è valida per cubi di qualsiasi lato; quando il lato del cubo è una potenza di 2 esiste un'implementazione più prestazionale che viene descritta nella sezione 4.5.3.

Per descrivere questa implementazione consideriamo le coordinate (x, y, z) all'interno di un reticolo cubico di lato L. Per indicizzare l'array `Spins` non è possibile utilizzare le coordinate dello spin ma a partire da quest'ultime è possibile calcolare il corrispondente indice. Ricordiamo che il reticolo che stiamo considerando è cubico e che quindi tutte e tre le coordinate possono avere valori compresi tra 0 e L; ricordiamo inoltre la disposizione in memoria degli spin nell'array di vettori `Spins` mostrato in figura 2.3; possiamo calcolare l'indice da utilizzare nell'array `Spins` come

$$i = x + L \cdot y + L^2 \cdot z \ , \tag{4.1}$$

dove i è l'indice dello spin corrente. Nello stesso modo possiamo calcolare gli indici dei vicini come

$$\begin{aligned}
i_{x_+} &= (x+1)\, mod\, L + Ly + L^2 z \ , & (4.2)\\
i_{x_-} &= (x-1)\, mod\, L + Ly + L^2 z \ , & (4.3)\\
i_{y_+} &= x + L((y+1)\, mod\, L) + L^2 z \ , & (4.4)\\
i_{y_-} &= x + L((y-1)\, mod\, L) + L^2 z \ , & (4.5)\\
i_{z_+} &= x + Ly + L^2((z+1)\, mod\, L) \ , & (4.6)\\
i_{z_-} &= x + Ly + L^2((z-1)\, mod\, L) \ , & (4.7)
\end{aligned}$$

dove l'operatore *mod* indica il resto della divisione tra la parte alla sua sinistra e la parte alla sua destra e, nel nostro caso, viene utilizzato per forzare il valore della parte sinistra a valori compresi tra 0 e L. Ricordiamo inoltre che $SIDE = L$ è il lato del cubo e $FACE = L^2$ è l'area di una faccia nella nostra implementazione. Si noti che la 4.1 corrisponde al calcolo di `inSpin`

nel precedente codice sorgente. Per quanto riguarda le equazioni 4.2, 4.3, 4.4, 4.5, 4.6 e 4.7, le variabili i_{x+}, i_{y+} e i_{z+} saranno contenute nella variabile `xyzp` mentre le variabili i_{x-}, i_{y-} e i_{z-} saranno contenute nella variabile `xyzm`. Inoltre ciascuno dei membri destri delle equazioni 4.2, 4.3, 4.4, 4.5, 4.6 e 4.7 può essere riscritto come

$$
\begin{aligned}
i_{x+} &= m_x \cdot ((x+1)\,mod\,L) + a_x \;, && (4.8)\\
i_{x-} &= m_x \cdot ((x-1)\,mod\,L) + a_x \;, && (4.9)\\
i_{y+} &= m \cdot ((y+1)\,mod\,L) + a_y \;, && (4.10)\\
i_{y-} &= m_y \cdot ((y-1)\,mod\,L) + a_y \;, && (4.11)\\
i_{z+} &= m_y \cdot ((z+1)\,mod\,L) + a_z \;, && (4.12)\\
i_{z-} &= m_y \cdot ((z-1)\,mod\,L) + a_z \;, && (4.13)
\end{aligned}
$$

dove

$$
\begin{aligned}
m_x &= 1 \;,\\
m_y &= L = SIDE \;,\\
m_z &= L^2 = FACE \;,
\end{aligned}
$$

e

$$
\begin{aligned}
a_x &= Ly + L^2 z = (SIDE) \cdot y + (FACE) \cdot z \;,\\
a_y &= x + L^2 z = x + (FACE) \cdot z \;,\\
a_z &= x + Ly = x + (SIDE) \cdot y \;.
\end{aligned}
$$

Si noti che nel codice sorgente le variabili m_x, m_y e m_z sono i primi tre elementi della variabile `multiplier` mentre a_x, a_y e a_z sono i primi tre elementi di `adder`. L'uso di `multiplier` e di `adder` avviene nelle ultime due righe usando la funzione `spu_madd()` che esegue la moltiplicazione tra i suoi primi due parametri ed inseguito somma il terzo parametro.

Infine, per quanto riguarda $((x+1)\,mod\,L)$, $((x-1)\,mod\,L)$, $((y+1)\,mod\,L)$, $((y-1)\,mod\,L)$, $((z+1)\,mod\,L)$, $((z-1)\,mod\,L)$ nelle equazioni 4.8, 4.9, 4.10, 4.11, 4.12 e 4.13 non è possibile tradurle "letteralmente" in linguaggio C ma è necessario che si esegua un codice differente. Tramite rappresentazione matematica e prendendo in considerazione per esempio l'asse x, anche se il ragionamento può essere esteso anche agli altri assi, abbiamo che:

$$
x+1 = \begin{cases} 0, & se\,(x+1) = L \\ x+1, & altrimenti \end{cases} \;, \qquad (4.14)
$$

$$x - 1 = \begin{cases} L, & se\,(x-1) < 0 \\ x-1, & altrimenti \end{cases}. \qquad (4.15)$$

Si noti che le equazioni 4.14 e 4.15 vengono eseguite tramite l'uso delle istruzioni `spu_add()`, `spu_cmpeq()`, `spu_cmpgt()` e `spu_sel()`. Anche in questo caso, è stato evitato l'uso di un blocco condizionale utilizzando la tecnica *branch-free implementation* sulla quale rimandiamo alla sezione 4.5.1.

Passiamo ora al successivo blocco di codice, cioè il calcolo del differenza di energia generato dalla nuova configurazione. Il codice sorgente C che esegue questo calcolo è il seguente:

```
1  EU     = spu_xor(Spins[inSpin], spu_xor(Jx[inSpin],
      Spins[spu_extract(xyzp, 0)]) );
2  prod   = spu_xor(Spins[inSpin], spu_xor(Jx[
      spu_extract(xyzm, 0)], Spins[spu_extract(xyzm, 0)])
       );
3  Carry1 = spu_and(prod,      EU);
4  EU     = spu_xor(prod,      EU);
5  ED     = Carry1;
6  prod   = spu_xor(Spins[inSpin], spu_xor(Jy[inSpin],
      Spins[spu_extract(xyzp, 1)]) );
7  Carry1 = spu_and(prod,      EU);
8  EU     = spu_xor(prod,      EU);
9  ED     = spu_xor(Carry1,    ED);
10 prod   = spu_xor(Spins[inSpin], spu_xor(Jy[
      spu_extract(xyzm, 1)],Spins[spu_extract(xyzm, 1)])
      );
11 Carry1 = spu_and(prod,      EU);
12 EU     = spu_xor(prod,      EU);
13 Carry2 = spu_and(Carry1,    ED);
14 ED     = spu_xor(Carry1,    ED);
15 EC     = spu_xor(Carry2,    EC);
16 prod   = spu_xor(Spins[inSpin], spu_xor(Jz[inSpin],
      Spins[spu_extract(xyzp, 2)]) );
17 Carry1 = spu_and(prod,      EU);
18 EU     = spu_xor(prod,      EU);
19 Carry2 = spu_and(Carry1,    ED);
20 ED     = spu_xor(Carry1,    ED);
21 EC     = spu_xor(Carry2,    EC);
22 prod   = spu_xor(Spins[inSpin], spu_xor(Jz[
```

```
   spu_extract(xyzm, 2)], Spins[spu_extract(xyzm, 2)])
    );
23 Carry1 = spu_and(prod,     EU);
24 EU     = spu_xor(prod,     EU);
25 Carry2 = spu_and(Carry1,   ED);
26 ED     = spu_xor(Carry1,   ED);
27 EC     = spu_xor(Carry2,   EC);
```

Dove si può facilmente notare che tutti gli operatori binari presenti nell'implementazione x86 sono stati sostituiti dalle corrispettive istruzioni SIMD. Inoltre si è evitato l'uso dell'array `V0` e della variabile `S0`: al suo posto si è indicizzato direttamente l'array `Spins` in modo da rendere il codice più pulito. Dal punto di vista logico il calcolo non cambia e quindi non si ripeterà quanto già detto nella sezione 2.2.

Il successivo blocco di codice è quello che decide l'accettazione o meno della nuova configurazione di spin. Il suo codice sorgente è:

```
1 EC = spu_xor(EC, 0xffffffff);
2 ED = spu_xor(ED, 0xffffffff);
3 EU = spu_xor(EU, 0xffffffff);
4
5 Carry1 = spu_and(SG_PARAMETERS.OU[spu_extract(IDX, 0)
   ].vec, EU);
6 Carry2 = spu_or(spu_and(SG_PARAMETERS.OD[spu_extract(
   IDX, 0)].vec, ED), spu_and(spu_or(SG_PARAMETERS.OD[
   spu_extract(IDX, 0)].vec, ED), Carry1));
7
8 Spins[inSpin] = spu_xor(Spins[inSpin], spu_or(Carry2,
    EC));
```

Anche qui gli operatori binari sono stati sostituiti con le corrispettive istruzioni SIMD fornite dal kit di sviluppo del Cell. Un breve nota va fatta per le prime tre istruzioni, in cui il complemento a uno o alternativamente il *not* bit-a-bit, presente nell'implementazione x86, viene eseguito tramite l'istruzione `spu_xor()` il cui secondo parametro è il valore `0xffffffff`: se consideriamo una variabile `a` di dimensione pari a 32 bit abbiamo che `~a = a^0xffffffff`. Anche qui si rimanda alla sezione 2.2 per ulteriori chiarimenti in riferimento alla metodologia di calcolo utilizzata.

Un ulteriore blocco di codice presente nell'implementazione Cell ma non nell'implementazione x86 è lo scambio di email tra PPE ed SPE. Le due funzioni che si occupano del invio e della ricezione delle email sono le seguenti:

```
1 void sendMailToPPE() {
2   do { } while(spu_stat_out_mbox () == 0);
3   spu_write_out_mbox(1);
4 }
5
6 int waitMailFromPPE() {
7   while(spu_stat_in_mbox() == 0) {};
8   return spu_read_in_mbox ();
9 }
```

La prima istruzione esegue l'invio di una mail. Il costrutto `do {} while();` il cui test è `spu_stat_out_mbox() == 0` attende fino a quando la capacità della coda è diversa da zero, in pratica fino a quando almeno una posizione della coda non è disponibile per l'inserimento di una nuova mail in uscita. L'invio della mail vero e proprio avviene tramite la funzione `spu_write_out_mbox(1)`, la quale accetta come parametro una variabile di 32 bit.

Nella seconda funzione il SPE attende l'arrivo di una mail da parte del PPE tramite il costrutto `while() {}` il quale viene eseguito finché il test, ovvero l'istruzione `spu_stat_in_mbox() == 0`, è vero. All'arrivo di una mail `spu_stat_in_mbox()` restituirà un valore diverso da zero; questo farà terminare il ciclo e la mail verrà letta tramite l'istruzione `spu_read_in_mbox()`.

Trattiamo ora l'ultimo ma fondamentale blocco di codice presente nel lato SPE e che non è presente nell'implementazione x86. Quest'ultimo blocco si occupa dei trasferimenti DMA dalla memoria centrale al LS dello SPE. Ricordando quanto detto nell'algoritmo 4, sul lato SPE sono presenti due trasferimenti di dati: il primo, dalla memoria centrale al LS, si occupa di caricare i dati della simulazione mentre il secondo, dal LS alla memoria centrale, si occupa di salvare i dati computati che verranno utilizzati dal PPE per il calcolo dei risultati finali.

L'istruzione che si occupa dei trasferimenti DMA dalla memoria centrale al LS è l'istruzione `mfc_get()` mentre l'istruzione che trasferisce dati dal LS alla memoria centrale è `mfc_put()`; entrambe le istruzioni accettano i seguenti parametri:

1. Indirizzo di un area di memoria sul LS dello SPE.

2. Valore di un indirizzo all'interno della memoria centrale.

3. Dimensione in byte dell'area di memoria da trasferire.

4. 5. 6. Gli ultimi tre parametri sono riservati ad un uso avanzato dei comandi DMA [17].

La funzione che si occupa dei trasferimenti dei parametri della simulazione e quindi dalla memoria centrale al LS dello SPE è

```
1  void loadConfiguration() {
2    register int offset;
3    unsigned int i;
4
5    mfc_get( (void *) &SPE_PARAMETERS, addr,
6             sizeof (spe_params), 1, 0, 0 );
7    mfc_write_tag_mask ( 1 << 1 );
8    mfc_read_tag_status_all ();
9
10   offset = 0;
11   for(i=0; i<VOLUME; i++) {
12     mfc_get( (void *) &Spins + offset,
13              SPE_PARAMETERS.Spins_addr + offset,
14              16, 1, 0, 0 );
15     mfc_get( (void *) &Jx    + offset,
16              SPE_PARAMETERS.Jx_addr    + offset,
17              16, 1, 0, 0 );
18     mfc_get( (void *) &Jy    + offset,
19              SPE_PARAMETERS.Jy_addr    + offset,
20              16, 1, 0, 0 );
21     mfc_get( (void *) &Jz    + offset,
22              SPE_PARAMETERS.Jz_addr    + offset,
23              16, 1, 0, 0 );
24     offset += 16;
25   }
26   mfc_get( (void *) &ira, SPE_PARAMETERS.ira_addr,
27             sizeof (ira), 1, 0, 0 );
28   mfc_get( (void *) &SG_PARAMETERS, SPE_PARAMETERS.
       sg_param_addr,
29             sizeof (SG_PARAMETERS), 1, 0, 0 );
30   mfc_write_tag_mask ( 1 << 1 );
31   mfc_read_tag_status_all ();
32
33   ...
34 }
```

Per la definizione della struttura `SPE_PARAMETERS` si rimanda alle appendici B, C e D.

La funzione che si occupa di trasferire i dati computati dal LS dello SPE alla memoria centrale è

```
1  void saveConfiguration() {
2    register int offset;
3    unsigned int i;
4
5    offset = 0;
6
7    for(i=0; i<VOLUME; i++) {
8      mfc_put( (void *) &Spins + offset, SPE_PARAMETERS
          .Spins_addr + offset,
9      16, 1, 0, 0 );
10     offset += 16;
11   }
12   mfc_write_tag_mask ( 1 << 1 );
13   mfc_read_tag_status_all ();
14 }
```

In entrambi i blocchi di codice precedenti, sono state usate le funzioni `mfc_write_tag_mask()` e `mfc_read_tag_status_all()` per avere la certezza che i dati siano stati completamente trasferiti prima di essere utilizzati.

Il codice del lato SPE a cui si è arrivati è frutto di molteplici test effettuati. Di fondamentale importanza è stato l'analisi e lo studio del codice assembly generato a partire dal codice sorgente C. Nella prossima sezione mostreremo l'utilizzo dello strumento software che ha aiutato a raggiungere questo codice.

4.4 SPU Timing: analisi dell'implementazione dell'algoritmo su lato SPE

L'applicazione SPU Timing è lo strumento fondamentale per l'analisi dell'implementazione di un qualsiasi algoritmo sul lato SPE. Il suo scopo è quello di analizzare le singole istruzioni assembly del codice prodotto dal compilatore e che verrà eseguito sul SPE, analizzandone le seguenti caratteristiche:

- Ciclo di clock: viene indicato a quale ciclo di clock viene eseguita ogni singola istruzione assembly.
- Tipo di istruzione: ricordando quanto detto nella sezione 3.1.4 a proposito della doppia pipeline di comandi, l'applicazione SPU Timing analizza il tipo di istruzione indicando su quale pipeline andranno eseguite.

- Parallelismo: indica se due istruzioni vengono eseguite in parallelo o meno.
- Cicli richiesti: indica il numero di cicli di clock richiesti da ogni singola istruzione.

L'applicazione SPU Timing prende in input un file di programma solo compilato, cioè un programma che non ha subito linking, il cui contenuto è linguaggio assembly del set di istruzioni dello SPE.

Un esempio di file di output è il seguente:

```
                                                                      ...
                                                                     .L64:
000703 0D    34                                                       il $5,0
000703 1D    3456                                                     fsmbi $41,0
000704 0D     45                                                      il $54,128
000704 1D     4                                                       lnop
                                                                     .L65:
000705 0D      56                                                     ori $27,$4,0
000705 1D      5678                                                   rotqbyi $11,$4,4
000706 0D       67                                                    ai $63,$4,1
000706 1D       6789                                                  rotqbyi $64,$4,8
000707 0D        7890                                                 shli $28,$27,4
000707 1D        7890                                                 rotqbyi $12,$4,12
000708 0D         89                                                  andi $70,$63,255
000708 1D         8901                                                shufb $17,$5,$5,$51
000709 0           9012                                               shli $24,$11,4
000710 0            0123                                              shli $77,$64,4
000711 0D            1234                                             shli $65,$12,4
000711 1D            1234                                             rotqbyi $78,$70,4
000712 0D             23                                              ai $29,$70,1
000712 1D             2345                                            rotqbyi $18,$70,8
000713 0D              34                                             ori $16,$70,0
000713 1D              345678                                         lqx $72,$24,$31
000714 0D               45                                            andi $75,$29,255
000714 1D               456789                                        lqx $3,$77,$31
000715 0D                5678                                         shli $7,$78,4
000715 1D                5678                                         rotqbyi $79,$70,12
000716 0D                 6789                                        shli $25,$18,4
000716 1D                 6789                                        rotqbyi $13,$75,4
000717 0D                  78                                         a $22,$17,$36
000717 1D                  7890                                       rotqbyi $9,$75,8
000718 0D                   8901                                      shli $73,$16,4
000718 1D                   8901                                      rotqbyi $2,$75,12
000719 0D                    90                                       selb $6,$17,$22,$37
000719 1D                    9012                                     shlqbyi $19,$75,0
000720 0                      01                                      a $66,$72,$3
000721 0                       1234                                   shli $69,$13,4
000722 0D                       2345                                  shli $11,$9,4
000722 1D                       234567                                stqx $66,$28,$31
000723 0D                        34                                   ori $76,$6,0
000723 1D                        345678                               lqx $15,$65,$31
000724 0D                         45                                  sf $74,$36,$17
                                                                       ...
```

Analizzando il file di output si possono definire i significati di ogni singola colonna: la prima indica il ciclo di clock in cui viene eseguita l'istruzione;

la seconda indica la pipeline su cui viene eseguita l'istruzione; segue, poi, un carattere tra cui una D (d maiuscola), d (d minuscola) o uno spazio vuoto rispettivamente per indicare se avviene parallelismo, se non avviene parallelismo a causa di una dipendenza tra i registri, se non avviene parallelismo. Seguono poi le colonne che indicano il numero di cicli di clock che una determinata istruzione impiega per essere eseguita. Infine vengono riportate le istruzioni associate ai dati che si trovano nelle colonne a sinistra.

Nel caso di progetti scritti in linguaggio assembly, SPU Timing permette di indicare il giusto ordine con cui eseguire le istruzioni in modo da avere la massime prestazioni possibili.

Nel caso di progetti scritti in linguaggi di più alto livello come il linguaggio C, l'applicazione SPU Timing serve per verificare la qualità del codice generato dal compilatore utilizzato: durante la fase di sviluppo del progetto è stato eseguito anche il confronto di prestazioni tra compilatori diversi, nello specifico, GCC e XLC. L'analisi del primo compilatore ha rivelato un codice assembly molto pulito, ben ordinato e con una discreta facilità di comprensione. Questo ha permesso di verificare come le istruzioni venissero schedulate, ovvero quale fosse il loro ordine di esecuzione.

D'altro canto, l'utilizzo del compilatore XLC ha rivelato un codice con prestazioni migliori nonostante producesse codice di dimensioni maggiori. L'analisi del file di output di SPU Timing non era di facile comprensione mentre la schedulazione delle istruzione era portata al massimo del parallelismo. Una particolarità del codice prodotto da XLC era che l'implementazione dell'Algoritmo di Metropolis veniva codificata tramite una tecnica chiamata *loop unrolling*, che spiegheremo nella sezione 4.5.

L'applicazione SPU Timing è in definitiva lo strumento semplice ma al tempo stesso ideale per ottimizzare l'implementazione di programmi sul lato SPE. Inoltre, nonostante sia più adatto ad implementazioni in codice assembly, può rivelarsi utile, come si è visto, anche per l'uso con linguaggi a livello più alto come il linguaggio C.

4.5 Ottimizzazione dell'implementazione

Durante la fase di sviluppo del progetto sono state scritte 19 versioni diverse del progetto; da queste versioni sono state create 5 versioni finali. Il motivo dello sviluppo di così tante versioni sta nella ricerca delle prestazioni, uno dei fattori fondamentali delle simulazioni su computer. In ogni versione è stata sperimentata una modifica o più modifiche alla versione iniziale fino ad arrivare alle versioni finali del progetto che rappresentano le migliori prestazioni ottenute.

La lista di tutte le ottimizzazioni fatte è abbastanza lunga e a tratti molto tecnica. Descriveremo quindi le principali modifiche fatte dalla versione 0, la prima versione del progetto, fino ad arrivare alle 5 versioni finali, di cui mostreremo i risultati nel capitolo 5.

In questa sezione descriveremo l'uso di alcune istruzioni del linguaggio C [21] fornite nel kit di sviluppo di IBM per il processore Cell. Per una lista completa di queste istruzioni e la loro approfondita descrizione si vedano [16, 17].

4.5.1 Vettorializzazione del codice

Questa è la principale ottimizzazione che viene richiesta quando si sviluppa un programma qualsiasi per SPE. Richiede di trasformare, dove possibile, le variabili scalari in variabili vettoriali in modo da poter eseguire i calcoli tramite il supporto SIMD dello SPE. Per spiegare questa ottimizzazione facciamo uso di un semplice esempio; si analizzi il seguente codice in linguaggio C:

```
1 int i;
2 unsigned int a[16], b[16], c[16];
3 ...
4 for(i=0; i<16; i++) {
5   a[i] = b[i] + c[i];
6 }
```

Il codice, molto semplice, somma ciascuno dei 16 elementi delle variabili `b` e `c` e ne salva il risultato in `a`. Questo è il tipico esempio di codice totalmente vettorializzabile. Convertiamo le variabili `a`, `b` e `c` in array di vettori nel seguente modo:

```
1 vector unsigned int a[4], b[4], c[4];
```

Ricordando quanto detto nella tabella 3.1, si noti che la dimensione di `a`, `b` e `c` è passata da 16 a 4 in quanto, ogni vettore di tipo `unsigned int`, contiene 4 elementi. Il restante codice dell'esempio può essere modificato in questo modo:

```
1 int i;
2 ...
3 for(i=0; i<4; i++) {
4   a[i] = spu_add(b[i], c[i]);
5 }
```

Dove si è fatto uso dell'istruzione `spu_add()` disponibile nel kit di sviluppo IBM nella libreria SIMD. Con questa istruzione eseguiamo 4 somme di va-

lori interi in due soli cicli di clock. Rispetto al semplice calcolo scalare, ciò rappresenta un forte aumento di prestazioni.

Una problematica introdotta dalla conversione dal calcolo scalare al calcolo vettoriale è la presenza di eventuali blocchi condizionali in cui il test debba essere effettuato su variabili vettoriali. Facciamo un esempio di questa problematica tramite il seguente codice sorgente C:

```
1 vector unsigned int a;
2 ...
3 if(spu_extract(a, 0) == 0) spu_insert(1, a, 0);
4 if(spu_extract(a, 1) == 0) spu_insert(1, a, 1);
5 if(spu_extract(a, 2) == 0) spu_insert(1, a, 2);
6 if(spu_extract(a, 3) == 0) spu_insert(1, a, 3);
```

Dove si sono utilizzate le istruzioni `spu_extract()` e `spu_insert()` rispettivamente per estrarre e inserire elementi all'interno di un vettore. Per convertire il seguente codice ed utilizzare il supporto SIMD utilizziamo una pratica detta *branch-free implementation*, con la quale si implementa il codice in questo modo:

```
1 vector unsigned int a;
2 vector unsigned int zeroes = {0, 0, 0, 0}
3 vector unsigned int ones   = {1, 1, 1, 1};
4 ...
5 select = spu_cmpeq(a, zeros);
6 a = spu_select(a, ones, select);
```

Dove sono stati introdotti due vettori di supporto `zeroes`, `ones` e sono state usate le istruzioni `spu_cmpeq()` e `spu_select()`; la prima di queste due istruzioni fa parte di una famiglia utilizzata per eseguire confronti tra vettori, nello specifico tra gli elementi che compongono il vettore. Appartengono a questa stessa famiglia le istruzioni: `spu_cmpabseq()`, `spu_cmpabsgt()`, `spu_cmpeq()`, `spu_cmpgt()`, delle quali si possono avere le specifiche vedendo [17]. Tutte le istruzioni di questa famiglia restituiscono come valore di ritorno un vettore contenente elementi i cui bit sono o tutti zero o tutti uno: nel primo caso la condizione non è stata rispettata mentre nel secondo caso si; a questo punto viene usata l'istruzione `spu_select()` la quale accetta come parametri due vettori contenenti i dati più un vettore che è il valore di ritorno di una delle istruzioni di confronto elencate prima. L'istruzione `spu_select()` selezionerà il bit i di uno dei due vettori di dati a seconda che il relativo bit i del vettore di confronto sia rispettivamente 0 o 1. Gli elementi del valore di ritorno di `spu_select()`, ovvero gli elementi della variabile `a`, varranno 1 se il loro pre-

cedente valore era 0. In questo modo si sono evitati i 4 blocchi condizionali passando ad un codice di sole due istruzioni.

4.5.2 Loop unrolling

Il loop unrolling, detto anche loop unwinding, è una particolare tecnica che può migliorare le prestazioni dei cicli loop. I principali vantaggi di questa tecnica sono:

- Riduzione dell'*overhead* introdotto dai cicli loop.
- Possibilità di migliore scheduling delle istruzioni e quindi di aumento del parallelismo.
- Riduzione del numero di *branch*, ovvero di salti da una istruzione ad un altra.

Il principale svantaggio di questa tecnica è l'aumento della dimensione del codice, un fattore che è da tenere ben in considerazione quando ci si trova in architetture con una memoria limitata.

La tecnica del loop unrolling consiste nel replicare più volte tutte le istruzioni che formano il loop, riducendo il numero di cicli eseguiti. Vediamo un semplice esempio:

```
1 for(int i=0; i<100; i++) {
2   dosomething(a[i]);
3 }
```

Su questo ciclo può essere tranquillamente applicato il loop unrolling, trasformando il precedente codice nel seguente:

```
1 for(int i=0; i<100; i=i+5) {
2   dosomething(a[i]);
3   dosomething(a[i+1]);
4   dosomething(a[i+2]);
5   dosomething(a[i+3]);
6   dosomething(a[i+4]);
7 }
```

Nulla vieta di ripetere la funzione `dosomething()` un numero maggiore di volte: ad esempio si potrebbe ripetere dieci volte incrementando la variabile `i` di dieci. Data la semplicità dell'esempio, si noti che potremmo tranquillamente

eliminare il ciclo e ripetere la funzione `dosomething()` per cento volte eliminando completamente l'overhead del ciclo. Si noti inoltre come ciò comporterebbe un forte aumento del numero di righe di codice utilizzato.

Consideriamo ora un esempio ripreso da [20], più complesso e più legato all'architettura Cell. Si consideri il seguente codice sorgente:

```
1 for(int i=0; i<8; i++) {
2   if((i % 2) = 0) do_evenstuff(i);
3   else            do_oddstuff(i);
4 }
```

Ricordando quanto detto a proposito delle pipeline comandi nel SPE nella sezione 3.1.4, abbiamo due istruzioni, `do_evenstuff()` e `do_oddstuff()`, che agiscono rispettivamente sulla pipeline 0 e sulla pipeline 1. Queste due istruzioni possono quindi essere parallelizzate senza particolari problemi in questo modo:

```
1 for(int i=0; i<8; i=i+2) {
2   do_evenstuff(i);   do_oddstuff(i+1);
3 }
```

Oppure, dato il ridotto numero di iterazioni del ciclo, il precedente codice può essere riscritto come:

```
1 do_evenstuff(0);   do_oddstuff(1);
2 do_evenstuff(2);   do_oddstuff(3);
3 do_evenstuff(4);   do_oddstuff(5);
4 do_evenstuff(6);   do_oddstuff(7);
```

Dal punto di vista delle prestazioni abbiamo un grosso incremento di velocità e di riduzione dei tempi, non solo dovuto al parallelismo: il loop unrolling applicato all'architettura Cell permette di semplificare il lavoro di schedulazione delle istruzioni che compie il compilatore; in questo modo si possono avere più parallelismo e meno stalli dovuti ad istruzioni che devono essere eseguite su una stessa pipeline comandi.

4.5.3 Cubi il cui lato è una potenza di due

Andiamo ora a definire un particolare tipo di implementazione per il calcolo degli indici dei vicini. L'ottimizzazione in questione è applicabile quando il lato del cubo è una potenza di 2. L'idea è nata vedendo come l'indice di uno spin cambiava dal punto di vista binario muovendosi in varie direzioni all'interno del reticolo. Per spiegare questa ottimizzazione definiamo un reticolo di spin cubico di larghezza $L = 4$ contenente quindi $N = L^3$ spin, notando ovviamente

che anche N è ovviamente una potenza di due. Memorizziamo, quindi, ogni spin in un array di N elementi. Usando la logica imposta dal linguaggio C, avremo che il primo spin avrà indice 0, il secondo spin avrà indice 1, ecc... l'ultimo spin avrà indice $N-1$. Notiamo poi che un qualsiasi valore intero che sia anche una potenza di due ha tutti i bit a zero meno uno, come impone la matematica binaria.

Suddividiamo ora il nostro reticolo in L facce in modo che ciascuna contenga L^2 spin. A loro volta, suddividiamo ogni faccia in L righe in modo che ogni riga contenga L spin. Consideriamo ora la tabella 4.1 in cui sono stati riportati il valori delle coordinate (x, y, z) di ogni spin e il relativo valore `inSpin`, in rappresentazione decimale e binaria.

`inSpin`	z	y	x	Binario ($MSB...LSB$)
0	0	0	0	000000
1	0	0	1	000001
2	0	0	2	000010
3	0	0	3	000011
4	0	1	0	000100
5	0	1	1	000101
6	0	1	2	000110
7	0	1	3	000111
8	0	2	0	001000
...				...
15	0	3	3	001111
16	1	0	0	010000
...				...
63	3	3	3	111111

Tabella 4.1: Rappresentazione decimale, binaria e in valori (x, y, z) della variabile `inSpin`.

Notiamo che nella notazione binaria, a partire da LSB, i primi due bit indicano la coordinata x, il terzo e il quarto bit indicano la coordinata y, il quinto e sesto bit indicano la coordinata z. Possiamo notare che per andare dalle coordinate (x, y, z) a $(x+1, y, z)$ basta incrementare di 1 l'indice dello spin corrente; per andare dalle coordinate (x, y, z) a $(x, y+1, z)$ basta incrementare di 4 l'indice dello spin corrente; per andare dalle coordinate (x, y, z) a $(x, y, z+1)$ basta incrementare di 16 l'indice dello spin corrente. Lo stesso

ragionamento può essere applicato per il calcolo dei vicini negativi, avendo cura di decrementare invece di incrementare.

Per calcolare gli indici degli spin vicini possiamo quindi sfruttare questa proprietà: usando il supporto SIMD inseriamo l'indice dello spin corrente in due vettori, uno per i vicini positivi, l'altro per i vicini negativi; ricordiamo che gli spin vicini sono 6 e quindi l'ultimo elemento di questi due vettori rimane inutilizzato e può essere ignorato. Inseriamo nel primo elemento dei due vettori i vicini sull'asse x, nel secondo elemento i vicini sull'asse y e nel terzo elemento i vicini sull'asse z. Definiamo poi due altri vettori come i seguenti:

```
1 const vector unsigned int coords_mask = { SIDE-1, (
    SIDE-1)<<EXP_SIDE, (SIDE-1)<<EXP_SIDE_2, 0 };
2 const vector unsigned int coords_idnc = { 1,      1<<
    EXP_SIDE,        1<<EXP_SIDE_2,        0 };
```

Dove `SIDE` rappresenta la dimensione $L = 2^n$ del lato del cubo, `EXP_SIDE` è l'esponente della n della potenza di 2 del lato del reticolo, `EXP_SIDE_2` è $2 \cdot n$. Nel nostro caso, per un reticolo cubico di lato $L = 4$ abbiamo che `SIDE=4`, `EXP_SIDE=2` e `EXP_SIDE_2=4`. In questo modo possiamo scrivere il codice per il calcolo dei vicini nel seguente modo:

```
1 xyzp = spu_add(xyz, coords_idnc);        xyzm =
    spu_sub(xyz, coords_idnc);
2 xyzp = spu_sel(xyz, xyzp, coords_mask);   xyzm =
    spu_sel(xyz, xyzm, coords_mask);
```

Dove `xyzp` rappresenta il vettore contente i vicini in direzione positiva, `xyzm` i vicini in direzione negativa. Le due `spu_sel()` si occupano di "mascherare", cioè di selezionare solo i bit relativi ai valori delle coordinate (per la coordinata x i primi due bit, per la coordinata y il terzo e quarto, ecc...).

Si noti che con piccole variazioni è possibile trattare parallelepipedi piuttosto che cubi: basta definirne le dimensioni, le quali saranno tutte potenze di 2, e inserirle al giusto posto nei vettori `coords_mask` e `cords_idnc`.

Il limite di una implementazione di questo tipo sta nel fatto che le tre coordinate e quindi l'indice degli spin deve essere inserito in una variabile a 32 bit in modo da avere due soli vettori contenenti ciascuno tre coordinate. Il valore massimo teorico di spin indicizzabili tramite questo tipo di variabile è ovviamente $2^{32} = 4294967296$ e in termini di lato del cubo abbiamo che $L_{max} =^3 \sqrt{2^{32}}$.

Questa diversa implementazione del calcolo dei vicini permette di ridurre in modo sensibile il numero di istruzioni necessarie per il calcolo dei vicini introducendo un numero esiguo di variabili. Dal punto di vista delle performance si è notato una riduzione dei tempi di calcolo. La forte limitazione imposta

dal fatto che le dimensioni del cubo o comunque le dimensioni di un generico parallelepipedo, siano una potenza impone che questa implementazione si applichi ad un ristretto numero di casi.

4.5.4 Variabili di tipo register e allineamento degli scalari

L'uso di variabili di tipo `register`, caratteristiche del linguaggio C, è una pratica molto utilizzata nel codice dei kernel e in generale in programmi dove si richiedono alte prestazioni. Lo scopo del modificatore `register` è quello di indicare che una specifica variabile non deve subire il salvataggio in memoria centrale e che il suo valore deve rimanere il più possibile all'interno di un registro del processore. Il principale vantaggio sta nella velocità d'accesso al valore della variabile: trovandosi in un registro del processore può essere immediatamente utilizzata nei calcoli e nelle operazione che svolge il processore. Il modificatore `register` influisce su tutto il processo di compilazione in quanto sarà la sequenza delle istruzioni assembly a determinare il registro associato a quella variabile e quanto tempo rimarrà in tale registro. In effetti, il modificatore non impone che la variabile `register` rimanga in un determinato registro per tutto il tempo di esecuzione del programma ma, piuttosto, che essa ci resterà il più possibile.

Un ulteriore ottimizzazione che possono subire gli scalari sul lato SPE sono gli allineamenti. Quando ci sono istruzioni che utilizzano vettori e scalari come parametri le variabili scalari subiscono una rotazione per trovarsi nel cosiddetto Preferred Slot, trattato nella sezione 3.2.1. Le rotazioni impiegano alcuni cicli di clock per essere eseguite e quindi è opportuno evitarle. Tramite l'allineamento, lo stesso che si applica alle variabili utilizzate nei trasferimenti DMA, si può forzare uno scalare a trovarsi nel Preferred Slot.

Questa pratica viene utilizzata principalmente con il compilatore GCC.

4.6 Limiti di implementazione

La nostra implementazione del calcolo di spin glass ha portato alla scrittura di cinque versioni finali. Ciascuna versione ha specifiche caratteristiche e limiti. Iniziamo quindi con il definire le caratteristiche di ogni versione tramite la tabella 4.2.

Nelle prime tre versioni, cioè quelle il cui nome inizia per "sgv11", la dimensione del cubo è prefissata e la sua modifica richiede la ricompilazione dei sorgenti. Nelle ultime due versioni, cioè quelle il cui nome inizia per "sgv12", la dimensione del cubo è parametrica ed è possibile impostarla a piacere.

Nome versione	Dimensione massima cubo	Unrolling	Calcolo vicini
sgv11_15_3unroll	15	3	standard
sgv11_16_4unroll	16 (19)	4	2^n
sgv11_8_4unroll	8	4	2^n
sgv12_lparam	<=15		standard
sgv12.1_lparam	<=16 (19)		standard

Tabella 4.2: Caratteristiche delle versioni finali del progetto. Con il simbolo 2^n si indica il calcolo ottimizzato per cubi il cui lato è una potenza di 2, trattato nella sezione 4.5.3.

In generale, dai test effettuati su questa implementazione, si è notato che la massima dimensione del cubo è 15: infatti valori superiori non permettevano la compilazione dato che le variabili che venivano definite sul lato SPE occupavano un'area di memoria superiore a quella disponibile sul LS.

Per spiegare questo comportamento si pensi alla dimensione delle variabili `Spins`, `Jx`, `Jy` e `Jz` le quali sono tutti array con un numero di elementi pari al volume del reticolo. Possiamo facilmente calcolare l'area di memoria in byte occupata da queste variabili tramite la formula

$$s = (sizeof(vector\,unsigned\,int) \cdot 4 \cdot L^3)/1024 \tag{4.16}$$

Dove 4 indica le quattro variabili `Spins`, `Jx`, `Jy` e `Jz`.

Dato che la dimensione di un vettore è di 16-byte, per $L = 15$, abbiamo che $s = 211kB$. Ricordando che la dimensione del LS di un SPE è $256kB$, abbiamo che il codice sorgente può essere al massimo $45kB$.

Per effettuare la simulazione su cubi di dimensioni maggiori è stato deciso di scrivere due versioni differenti di calcolo di spin glass, le versioni "sgv11_16_4unroll" e "sgv12.1_lparam". Queste due fanno utilizzo di una solo variabile `J` al posto delle tre `Jx`, `Jy` e `Jz`. Nello specifico, questa variabile `J` è inizializzata a valori costanti pari a $k = 1$. Se consideriamo la 4.16 in questo caso abbiamo che

$$s = (sizeof(vector\,unsigned\,int) \cdot 2 \cdot L^3)/1024 \tag{4.17}$$

abbiamo che $s = 105kB$ per un cubo di lato 15. Ciò permette di estendere le dimensioni del cubo fino a 19.

Altro limite imposto dalla nostra implementazione è dato dalla modalità di calcolo dei vicini dello spin, nel lato SPE all'interno dell'Algoritmo di Metropolis. Come già detto nella sezione 4.5.3, le versioni "sgv11_16_4unroll" e "sgv11_8_4unroll" che implementano questa particolare tecnica di calcolo dei vicini sono limitati dalla dimensione del cubo il quale deve essere una potenza

di 2. Per le altre versioni il calcolo dei vicini è stato liberato da questa limitazione utilizzando una modalità di calcolo che si possa eseguire su qualsiasi dimensione.

Definiti i limiti delle implementazioni possiamo quindi passare ad analizzare i risultati ottenuti da ogni singola versione sviluppata. Questo argomento sarà trattato nel capitolo 5.

Chapter 5

Analisi dei risultati

5.1 Validità dei risultati ottenuti

Per verificare la validità della nostra simulazione per l'architettura Cell abbiamo agito in due modi:

1. Abbiamo fatto in modo che l'implementazione per x86 e l'implementazione per Cell "evolvessero" nello stesso modo così da verificare, per vari valori di dimensione del lato L del reticolo, che i dati risultanti di magnetizzazione fossero uguali.

2. Forzando i valori di $J_{ij} = 1$.

Nel primo caso si semplicemente pensato di partire da una stessa tabella di valori random e da uno stesso reticolo: nello specifico, abbiamo fatto in modo che entrambe le implementazioni partissero da una stessa configurazione $\{S_0\}$ e i relativi valori di J_{ij} fossero uguali. Questa prima verifica è stata implementata principalmente nei primi mesi di sviluppo.

Il secondo tipo di validazione è stata utilizzata principalmente nelle versioni finali del progetto e fa parte del risultato finale di ogni simulazione. Imporre le costanti $J_{ij} = 1$ significa che si avvantaggiano le configurazioni in cui gli spin vicini sono allineati.

5.2 Simulazioni effettuate

Prima di analizzare i risultati ottenuti, si consideri la tabella 5.1 contenente tutti i dati tecnici degli host utilizzati per effettuare le simulazioni.

Alcune note vanno fatte a proposito di ogni singolo host su cui sono state fatte le simulazioni.

Nome host	ps3	cab	cc03.juice
Architettura	Cell PS3	IBM Cell	IBM Cell
Frequenza (MHz)	3192	2800	3200
Quantità processori	1	1	2
Quantità SPE	6	8	8
Memoria (MB)	256	4000	1000

Table 5.1: Dati tecnici degli host utilizzati per le simulazioni.

Hostname	ps3	cab	cc03.juice
Beta (intervallo)	0.010-0.500		
Beta (split)	100		
Iterazioni (PPE)	1000	1000	100
Iterazioni (SPE)	1000	1000	1000

Table 5.2: Dati tecnici delle simulazioni.

L'architettura di *ps3*, una Playstation 3 con sistema operativo operativo Fedora Core 6, differisce dalle normali architetture Cell, in quanto il numero di SPE disponibili allo sviluppatore è limitato: gli SPE disponibili solo 6 dato che un SPE è riservato al firmware interno della Playstation 3 mentre l'ultimo SPE è bloccato.

L'host *cab*, acronimo di Cell Accelerator Board, è una scheda con interfaccia PCI Express prodotta da Mercury, da inserire all'interno di un normale computer. Questa scheda mette a disposizione un "emulatore" hardware, basato su sistema operativo Linux, in grado di aumentare le prestazioni del processore Cell fino a 180 GFlops. Mercury dichiara inoltre che le prestazioni del cab possono essere dalle 15 alle 30 volte maggiori per i calcoli in virgola mobile e fino a 100 volte superiori nei calcoli scientifici rispetto, ad esempio, ad architetture basate su tecnologia AMD, Intel o PowerPC.

Per quanto riguarda l'host *cc03.juice*, esso è un server IBM Blade QS20, basato su sistema operativo Fedora Core 6, in cui sono contenuti due processori Cell collegati tramite interfaccia FlexIO. La memoria disponibile è pari a 1 GB divisa tra i due processori. La particolarità di questi server è che si ha la possibilità di scegliere come i processori eseguiranno il programma: è possibile, per esempio, avere a disposizione 16 SPE oppure è possibile utilizzare i due processori separatamente.

Analizziamo ora le caratteristiche di tutte le simulazioni effettuate tramite la tabella 5.2.

Le simulazioni sono state tutte eseguite su 100 valori di β nell'intervallo 0.01...0.50.

	Hostname			
	ps3	cab	cc03.Juice	
Versioni				
sgv11_15_3unroll	252,59	287,95	251,97	ps/spin
sgv11_16_4unroll	224,59	256,02	224,04	ps/spin
sgv11_8_4unroll	225,99	257,62	225,43	ps/spin
sgv12_lparam	408,44	465,61	407,43	ps/spin
sgv12.1_lparam	407,98	465,08	406,98	ps/spin

Table 5.3: Tabella dei tempi ottenuti per tutte le versioni finali del progetto e per tutti gli host utilizzati.

Una piccola nota va fatta a proposito del numero di iterazioni sul lato PPE nella simulazioni sullo host *cc03.juice*, sul quale si è dovuto eseguire un numero minore di iterazioni a causa di un fattore tecnico relativo ai tempi di utilizzo dell'host.

Passiamo ora ad analizzare i tempi delle simulazioni riportati nella tabella 5.3.

Il miglior tempo è stato ottenuto nella versione sgv11_16_4unroll sullo host *cc03.juice*.

Iniziamo con l'analizzare le versioni *unrolled*, la tecnica spiegata nella sezione 4.5.2. Per spiegare i benefici del loop unrolling si utilizzi la tabella 5.4, i cui risultati sono stati presi da una test eseguito su un cubo di lato $L = 8$ durante lo sviluppo del progetto.

Analizzando inizialmente le prestazioni del compilatore GCC si può notare una riduzione dei tempi fino a 4 unrolling, dopo di che inizia il degrado delle prestazioni. Questo degrado non viene rilevato quando si compila utilizzando XLC. Ciò dipende da una diversa strategia di compilazione adottata dai due compilatori, cioè una diversa schedulazione delle istruzioni all'interno dell'implementazione dell'Algoritmo di Metropolis.

Si noti inoltre che, il raggiugimento di 8 unrolling è dovuto alle ridotte dimensioni delle variabili che descrivono il reticolo, le quali danno il maggiore contributo in fatto di occupazione del LS dello SPE.

Ritornando alla tabella 5.3, si noti poi che gli host su cui si sono ottenuti i migliori risultati sono *ps3* e *cc03.juice*. Ciò dipende principalmente dalla differenza di clock che hanno i diversi host.

Si analizzi ora l'efficenza delle implementazioni x86 e Cell. Riprendendo l'algoritmo 1 sono necessarie $n = 15$ operazioni per eseguire l'aggiornamento di uno spin. Supponendo che ogni operazione venga fatta in un ciclo di clock, consideriamo ora la frequenza $f = 3200\,MHz$ del host *cc03.juice* a cui cor-

	Compilatore	
Unrolling	GCC	XLC
0	346,79	287,97
2	239,02	246,90
4	227,43	227,74
8	238,14	223,15

Table 5.4: Tabella dei tempi rilevati per vari tipi di unrolling rilevati tramite test eseguito durante lo sviluppo.

risponde un periodo di clock τ pari a

$$\tau = \frac{1}{f} = 0,3125 \cdot 10^{-9}\, s\ .$$

Il tempo T_a richiesto per l'aggiornamento di uno spin è

$$T_a = n\tau = 4,6875 \cdot 10^{-9}\, s\ . \tag{5.1}$$

Consideriamo ora l'algoritmo di una implementazione *multi-spin coded*, la quale necessita di circa $n = 80$ operazioni. Allo stesso modo della 5.1, definiamo T_b come

$$T_b = n\tau = 25 \cdot 10^{-9}\, s\ ;$$

ricordando che un algoritmo *multi-spin coded* aggiorna M sistemi contemporaneamente, avremo che l'aggiornamento di un solo spin degli M sistemi impiegherà un tempo T_{msc} pari a

$$T_{msc} = \frac{T_b}{M} = 781 \cdot 10^{-12}\, s\ .$$

Considerando che un un programma multi-spin coded per l'architettura Cell, $M = 128$, abbiamo che il tempo di aggiornamento teorico di uno spin su un SPE è pari a

$$T_{id} = \frac{T_b}{M} = 195 \cdot 10^{-12}\, s\ .$$

Sia $T_{max} = 465,61\, ps$ il tempo di aggiornamento di uno spin della peggiore implementazione, e $T_{min} = 224,04\, ps$ il tempo di aggiornamento della migliore; definiamo l'efficenza massima e minima delle varie implementazioni per architettura Cell come

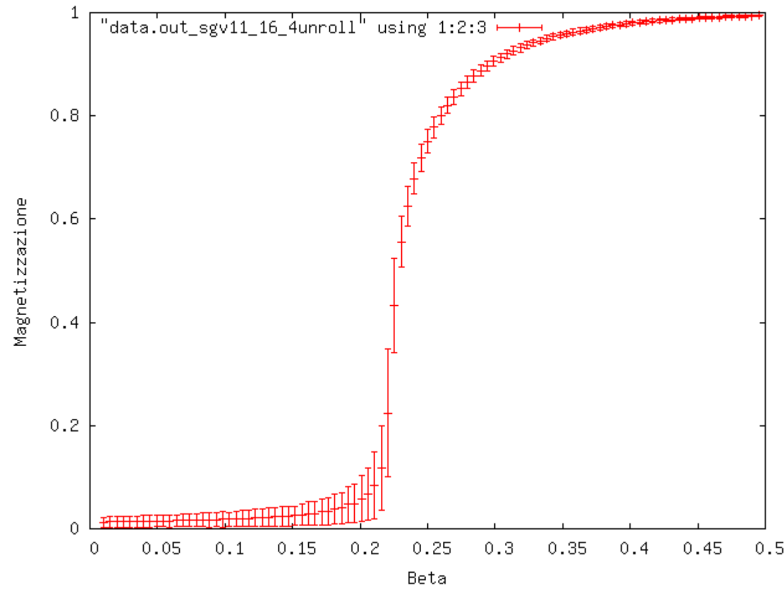

Figure 5.1: Grafico dei risultati finali per la simulazione effettuata da sgv11_16_4unroll.

$$E_{max} = \frac{T_{id}}{T_{max}} = 87\% ,$$
$$E_{min} = \frac{T_{id}}{T_{min}} = 42\% ,$$

da cui si evince che l'efficenza di un algoritmo spin glass va dal 42%, per la versione sgv12_lparam, al 87%, per la versione sgv11_8_4unroll.

Passiamo ora ad discutere i grafici dei valori di magnetizzazione in funzione della temperatura, con relative barre di incertezza.

I 5.1, 5.2 e 5.3 corrispondondono rispettivamente ai dati estratti tramite le simulazioni effettuate dalle versioni sgv11_16_4unroll, sgv11_15_3unroll e sgv11_8_4unroll del progetto.

Nei grafici è possibile notare che per piccoli valori di β, ovvero per alte temperature T, l'agitazione termica non permette al sistema di orientarsi. Ogni spin fluttua rapidamente tra i valori -1 e $+1$, per cui si ha $M \simeq 0$.

Per valori di β grandi, cioè per temperature T basse, si può notare che M si approssima all'unità, ovvero una larga frazione degli spin è orientata nello stesso verso.

Il passaggio da un regime all'altro al variare della T è caratterizzato da una brusca variazione di M, caratteristica delle transizioni di fase magnetiche.

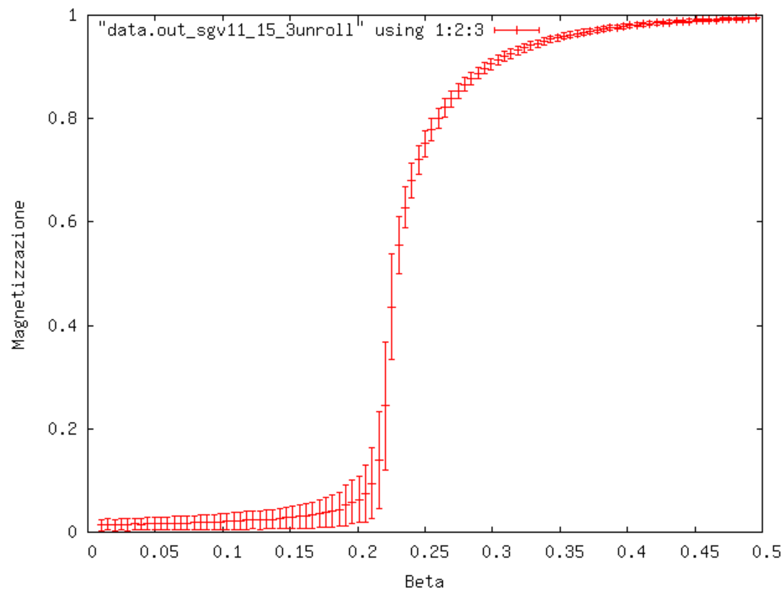

Figure 5.2: Grafico dei risultati finali per la simulazione effettuata da sgv11_15_3unroll.

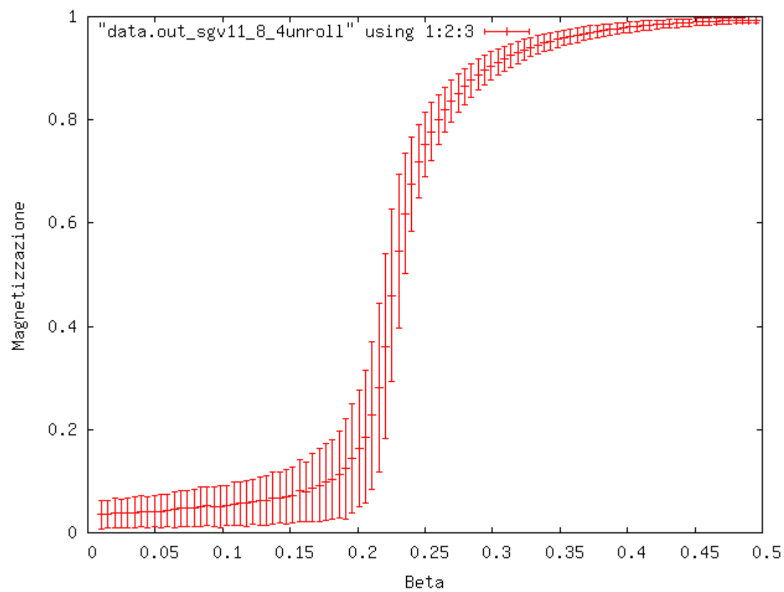

Figure 5.3: Grafico dei risultati finali per la simulazione effettuata da sgv11_8_4unroll.

Il modello permette quindi di rappresentazione qualitativa del fenomeno del ferromagnetismo.

Si noti che la pendenza di M in funzione di β intorno alla transizione è maggiore nei sistemi di taglia più grande: all'aumentare della taglia del sistema il passaggio da un regime all'altro si fa sempre più brusco; nei sistemi reali, in cui il numero di particelle è nell'ordine del Numero di Avogadro ($> 6 \cdot 10^{23}$) la derivata di M rispetto a B diventa discontinua. Per maggiori dettagli sullo studio di effetti di taglia finita, si rimanda, ad esempio, a [18].

Conclusioni e Sviluppi Futuri

Implementazione per cubi di lato 64

I limiti dell'implementazione della nostra simulazione impongono cubi di dimensione inferiore o uguale 15. Questo limite impone che la nostra simulazione non possa essere utilizzata in "produzione", non tanto per l'implementazione stessa ma piuttosto per la limitata dimensione del cubo che è in grado di gestire; in genere, infatti, il valore del lato del cubo, con cui si ottengono risultati utili, può arrivare fino a 64 per il Modello di Edwards-Anderson.

Inoltre, la nostra implementazione fa uso di un solo SPE: questo limita le reali potenzialità offerte dall'architettura Cell.

E' necessario quindi progettare una nuova implementazione che sfrutti appieno il l'architettura, utilizzando tutte le sue potenzialità, e che posssa gestire cubi di dimensioni maggiori di 15, approfittando dell parallelismo tra SPE.

Descriviamo brevemente una possibile implementazione per un cubo di lato 64 che potrebbe essere svolta senza particolari modifiche all'implementazione attuale e sfruttando ogni singolo SPE per l'esecuzione dell'Algoritmo di Metropolis.

Inizamo con una semplice considerazione: data la scarsa capienza del LS di un SPE, un cubo di lato 64 e le relative variabili che lo descrivono, non possono essere contenute nella memoria di un SPE. Si puó comunque risolvere il problema suddividendo il cubo in parallelepipedi piú piccoli, visualizzabili in figura 5.4, i cui dati che li descrivono possono quindi essere inseriti nel LS di un SPE.

Se dividiamo il cubo in parallelepipedi piú piccoli delle dimensioni di 64x8x4 possiamo calcolare, a partire dal volume del parallelepipedo in questione, lo spazio occupato dalle variabili `Spins`, `Jx`, `Jy` e `Jz` in questo modo:

$$[\text{sizeof(vector unsigned int)} \cdot 4\,(64 \cdot 8 \cdot 4)]\,/1024$$

Sostituendo i relativi valori otteniamo:

$$(16\,Byte \cdot 4 \cdot (64 \cdot 8 \cdot 4))/1024 = 128kBytes$$

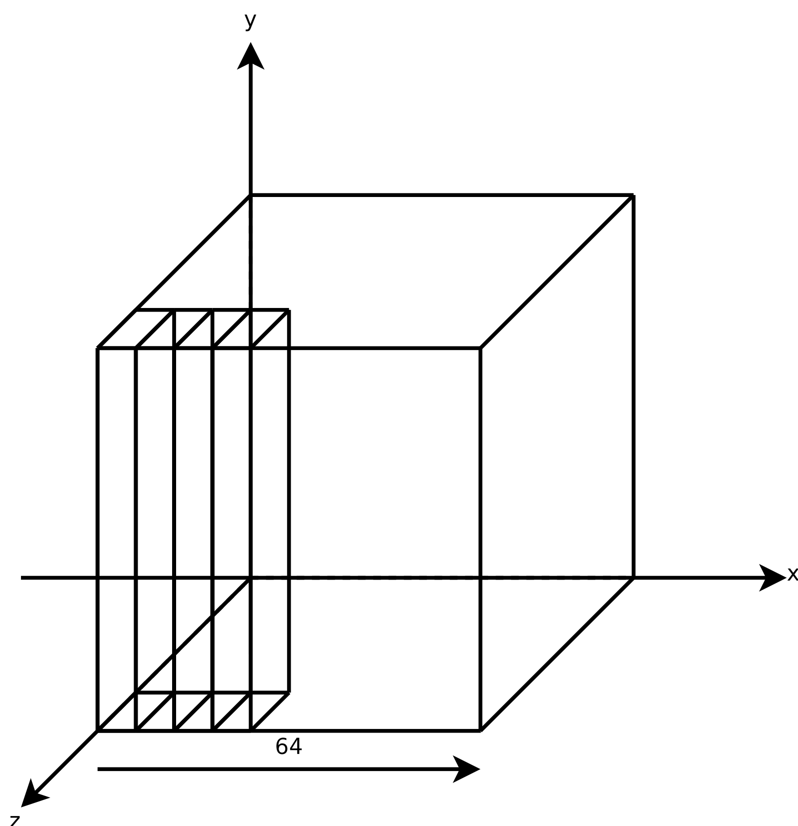

Figure 5.4: Suddivisione del cubo di lato 64 in parallelepipedi più piccoli i quali possono essere inseriti nel LS di un SPE.

Questo valore equivale esattamente alla meta' della dimensione totale del LS di un SPE; la parte di memoria restante é piú che sufficente a contenere il codice per eseguire il calcolo di Metropolis.

Un singolo SPE eseguirá Metropolis su ogni singolo parallelepipedo lungo l'asse z fino a completare il calcolo di tutti i parallelepipedi. In particolare, dato un cubo di larghezza 64 e dato che nell'architettura Cell ci sono tipicamente 8 SPE, il calcolo di spin glass puo' essere svolto facilmente senza dover riutilizzare piu' volte uno stesso SPE.

L'incoveniente di una implementazione di questo tipo sono i calcoli dei bordi dei parallelepipedi, in particolare dei bordi che confinano con un parallelepipedo calcolato da un'altro SPE. In questo caso si deve prevedere che ogni SPE si carichi in memoria le intere facce dei vicini (in figura 5.4, le facce dei vicini a destra e a sinistra del parallelepipedo in questione).

Conclusioni

Scopo della tesi è lo sviluppo e l'ottimizzazione di un progetto riguardante la simulazione dei modelli di Ising e di Edwards-Anderson su l'architettura IBM Cell-BE.

I risultati ottenuti hanno dimostrato che la nostra implementazione ha ottenuto un efficenza massima del 87% rispetto al picco teorico di prestazioni dell'architettura; le migliori prestazioni ottenute dalle varie versioni del progetto finale hanno dimostrato che la nostra implementazione può raggiungere i $225\,ps$ per il calcolo di un singolo spin, un tempo nettamente inferiore all'implementazione multi-spin coded per architettura x86 che abbiamo analizzato in questo elaborato.

Il lavoro svolto ha quindi dimostrato che l'architettura Cell è adatta al calcolo scientifico nel campo della fisica statistica, in particolare al calcolo intero e l'uso delle istruzioni binarie. Ciò permette di considerare l'architettura Cell per futuri progetti di rilevanza scientifica, nello specifico al calcolo di problemi di fisica termodinamica di sistemi di spin su reticolo.

Lo sviluppo del progetto ha dimostrato inoltre i limiti imposti da questa architettura, tra i quali il principale è la ridotta dimensione del LS di un SPE, imponendo scelte che limitano l'utilità del progetto sviluppato.

La descrizione dell'implementazione sull'architettura Cell ha dimostrato quali ottimizzazioni possono essere utilizzate quando si sviluppa un progetto di calcolo scientifico su spin.

Le difficoltà di sviluppo maggiori si sono rilevate principalmente sull'uso di una architettura considerata atipica rispetto alle altre architetture presenti sul mercato. La gestione delle comunicazioni tra le varie componenti e l'utilizzo di un SPE come supporto al PPE hanno richiesto un approfondito studio del progetto e degli algoritmi che si sono utilizzati per giungere all'implementazione finale.

La nostra implementazione fa uso di un solo SPE. Come sviluppi futuri sarebbe possibile utilizzare più SPE, in modo da aumentare l'efficenza.

Appendix A

Implementazione dell'Algoritmo di Metropolis con multi-spin coding per architetture x86

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>
#include <unistd.h>  /* UNIX standard function definitions */
#include <fcntl.h>   /* UNIX file descriptors controls */
#include <errno.h>   /* UNIX system calls error codes definitions*/
#include <sys/times.h>
#include <sys/stat.h>
#include <sys/types.h>
#define RANDOM ((irr+=(!ip)),((ira[irr][ip++]=ira[irr][ip1++]+ira[irr][ip2++])^ira[irr][ip3++]))
#define N_RANDOM (-((double)(T_4))*log(((double)(RANDOM))*(C_inv_max_rand)))
#define _SAMP_64_
#ifdef _SAMP_64_
typedef unsigned long long SWORD;
#define _NSAMPLES_ 64
#endif
#ifdef _RAND_64_
typedef unsigned long long RANDOMWORD;
#endif
#ifndef _RAND_64_
typedef unsigned RANDOMWORD;
#endif
const SWORD SWORD_ZERO=(SWORD)(0);
const double C_inv_max_rand=1.0/((double)(~((RANDOMWORD)(0))));
long L;
SWORD * Jx;
SWORD * Jy;
SWORD * Jz;
SWORD OD[16],OU[16];
long * opx,* opy,* opz,* omx,* omy,* omz;
RANDOMWORD ira[256][256];
unsigned char ip,ip1,ip2,ip3,irr;
int main() {
}
void Metropolis_Update(SWORD * Spin, double work_T){
  register long in, ix, iy, iz;
  unsigned IDX;
  register SWORD S0;
  register SWORD prod;
  register SWORD EU,ED,EC,IU,ID;
  register SWORD Carry1, Carry2;
  long Plus_y,Minus_y,Plus_z,Minus_z;
  double T_4;
  SWORD V0[6];
  T_4=0.25*work_T;
  for(iz=0,in=0; iz<L; iz++){
    Plus_z=opz[iz];
    Minus_z=omz[iz];
    for(iy=0; iy<L; iy++){
```

```
      Plus_y=opy[iy];
      Minus_y=omy[iy];
      for(ix=0; ix<L; ix++,in++){
        float n_random = N_RANDOM;
        if((IDX=(unsigned)n_random)>3) IDX=3;
        IU=OU[IDX];
        ID=OD[IDX];
        S0=Spin[in];
        V0[0]=Spin[in+opx[ix]];     V0[1]=Spin[in+omx[ix]];
        V0[2]=Spin[in+Plus_y];      V0[3]=Spin[in+Minus_y];         /*incremented below*/
        V0[4]=Spin[in+Plus_z];      V0[5]=Spin[in+Minus_z];         /*incremented below*/
        Carry1=Carry2=0;
        EU=SWORD_ZERO;      ED=SWORD_ZERO;          EC=SWORD_ZERO;         //most significant bit of the energy sum
        EU = S0^Jx[in]^V0[0];
        prod=S0^Jx[in+omx[ix]]^V0[1];
        Carry1=prod & EU  ; EU=prod^EU  ; ED=Carry1;
        prod=S0^Jy[in]^V0[2];
        Carry1=prod & EU  ; EU=prod^EU  ; Carry2=Carry1 & ED;   ED=Carry1^ED;   EC=Carry2;
        prod=S0^Jy[in+Minus_y]^V0[3];
        Carry1=prod & EU  ; EU=prod^EU  ; Carry2=Carry1 & ED;   ED=Carry1^ED;   EC=Carry2^EC;
        prod=S0^Jz[in]^V0[4];
        Carry1=prod & EU  ; EU=prod^EU  ; Carry2=Carry1 & ED;   ED=Carry1^ED;   EC=Carry2^EC;
        prod=S0^Jz[in+Minus_z]^V0[5];
        Carry1=prod & EU  ; EU=prod^EU  ; Carry2=Carry1 & ED;   ED=Carry1^ED;   EC=Carry2^EC;
/* X = sum(S^J^S)  */
/*           X        EC   ED   EU     deltaE=(4X-12)   flip?   ~EC  ~ED  ~EU */
/*           0        0    0    0         -12          always    1    1    1  */
/*           1        0    0    1         -8           always    1    1    0  */
/*           2        0    1    0         -4           always    1    0    1  */
/*           3        0    1    1          0           always    1    0    0  */
/*           4        1    0    0         +4            test     0    1    1  */
/*           5        1    0    1         +8            test     0    1    0  */
/*           6        1    1    0         +8            test     0    0    1  */
/*  All spins with ~EC=1 will flip (correspond to deltaE<=0). Among those with ~EC=0 (deltaE>0) */
/*      only the spins for which R <= exp(-beta*deltaE) i.e. -(T/4)logR-(EC,ED,EU) >= -3 i.e. -(T/4)logR+(~ED,~EU) >= 4; */
/*      Note that (T/4)logR >=3 would always flip spins,so only 4 IDX values (0,1,2,3) have to be sampled*/
/*      Actually an usual LUT is not needed. */
        EC=~EC; ED=~ED; EU=~EU;         /* negation is convenient: see above*/
        /*OD[IDX],OU[IDX] represent IDX positionally*/
        Carry1=(IU&EU);
        Carry2=(ID&ED)|((ID|ED)&Carry1);
        /*Carry2=1 if and only if (T/4)logR+(~ED,~EU) >= 4*/
        Spin[in]^=(Carry2|EC);
      }
    }
  }
}
```

Appendix B

Implementazione per Cell, versione sgv11_15_3unroll

B.1 Lato PPE

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <unistd.h>
#include "ppe_lib.h"
#include "common.h"
#include "user_common.h"
// Number of SPE to use
#define SPE_THREADS     1
//-----------------------------------------------------------------------------------------
// List of SPE-clients
extern spe_program_handle_t spe_program;
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
// PPE-SPE Sharing
unsigned int Spins[VOLUME][4] __attribute__ ((aligned (128)));
unsigned int Jx[VOLUME][4]    __attribute__ ((aligned (128))),
             Jy[VOLUME][4]    __attribute__ ((aligned (128))),
             Jz[VOLUME][4]    __attribute__ ((aligned (128)));
unsigned int ira[256][4]      __attribute__ ((aligned (128)));
// Magnetizations
double magnetization  [N_REPLICAS];
double avg_magnet     [N_REPLICAS];
double data           [N_REPLICAS];
double data_2         [N_REPLICAS];
double sigma          [N_REPLICAS];
unsigned char ip,
              ip1,
              ip2,
              ip3,
              irr;
const float C_inv_max_rand = 1.0/((double)(~((unsigned/* long int*/)(0))));
double T_4;
FILE * fp_data_out;
unsigned int zero = 0;
unsigned int one  = 1;
unsigned int result;
unsigned int PPE_ITERATIONS = 1000;
unsigned int SPE_ITERATIONS = 1000;
double BETA_START     = 0.01;
double BETA_STOP      = 0.5;
double BETA_SPLIT     = 100.0;
double BETA_INC;
//-----------------------------------------------------------------------------------------
myspe_t THREADS;
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
// PPE -> SPE ( MM -> LS )
spe_params SPE_PARAMS_LIST        __attribute__ ((aligned (128)));
spinglasses_params SG_PARAMS_LIST __attribute__ ((aligned (128)));
```

```
// SPE -> PP ( LS -> MM )
volatile run_status RUN_STATUS_LIST __attribute__ ((aligned (128)));
volatile run_ticks RUN_TICKS_LIST   __attribute__ ((aligned (128)));
//------------------------------------------------------------------------------------------
//------------------------------------------------------------------------------------------
// Functions declaration
void loadConfigurationFromFile();
void set_defaults();
void init_random();
void init_spins();
void init_couplings();
//------------------------------------------------------------------------------------------
void startSPE(spe_program_handle_t * program_pointer) {
  START_SPE(THREADS, program_pointer);
}
//------------------------------------------------------------------------------------------
float beta;
void sendDataMailToSPE() {
  unsigned int out_mail = 0;
  PR_DEBUG ("PPE: sending SPE parameters MM address\n");
  // High address (mfc_ea2h(...))
  out_mail = (unsigned int) ((unsigned long long int)(&(SPE_PARAMS_LIST  ))>>32)&0xffffffff;
  PPE_SM(THREADS, &out_mail);   // Send SPE parameters address (high)
  // Low address (mfc_ea2l(...))
  out_mail = (unsigned int) ((unsigned long long int)(&(SPE_PARAMS_LIST  )))&0xffffffff;
  PPE_SM(THREADS, &out_mail);   // Send SPE parameters address
}
#define waitMailFromSPE()          PPE_RM(THREADS, &result);
#define sendGoMail()               PPE_SM(THREADS, &one)
#define sendDontGoMail()           PPE_SM(THREADS, &zero)
#define printBetaAvgMagnAndSigma() fprintf(fp_data_out, "%f", beta); \
  for(j=0; j<N_REPLICAS; j++) fprintf(fp_data_out, " %lf %lf",       \
                                  avg_magnet[j], sigma[j]);      \
  fprintf(fp_data_out, "\n");
#define stopSPE()                  STOP_SPE(THREADS)
void resetMagnetizationsVars() {
  int k;
  // Reset of magnetizations vars
  for(k=0; k<N_REPLICAS; k++) {
    magnetization[k] = avg_magnet[k] = data[k] = data_2[k] = sigma[k] = 0;
  }
}
void calculateMagnetization() {
  int i, j, k;
  for(i=0; i<VOLUME; i++) {
    for(k=0; k<4; k++) {
      for(j=0; j<32; j++) {
        magnetization[(k*sizeof(unsigned int)*4)+j] +=
               ( ((Spins[i][k] >> j) & 0x1) ) ? 1 : -1;
      }
    }
  }
  for(i=0; i<N_REPLICAS; i++) {
    // Normalization of the magnetizations
    magnetization[i] /= (double)VOLUME;
    // Save the values for the future
    data[i]   +=  pow(magnetization[i], 2.0);
    data_2[i] += fabs(magnetization[i]);
    magnetization[i] = 0;
  }
}
void endMagnetizationCalculus() {
  int k;
  for(k=0; k<N_REPLICAS; k++) {
    avg_magnet[k]  = (data_2[k] / (double)PPE_ITERATIONS);
    // Media dei quadrati dei valori assoluti delle magnetizzazioni
    data[k]        = (data[k]   / (double)PPE_ITERATIONS);
    // Media dei valori assoluti delle magnetizzazioni elevata al quadrato
    data_2[k]      = pow( (data_2[k] / (double)PPE_ITERATIONS), 2.0);
    sigma[k]       = sqrt( data[k] - data_2[k] );
  }
}
void user_main (spe_program_handle_t * program_pointer) {
  unsigned i, j;
  for(beta = BETA_START; beta < BETA_STOP; beta += BETA_INC) {
    loadConfigurationFromFile();
    resetMagnetizationsVars();
    startSPE(program_pointer);
    sendDataMailToSPE();
    for(i = 0; i < PPE_ITERATIONS; i++) {
      waitMailFromSPE();
```

```
      calculateMagnetization();
      sendGoMail();
    }
    endMagnetizationCalculus();
    printBetaAvgMagnAndSigma()
    sendDontGoMail();
    stopSPE();
  }
}
//------------------------------------------------------------------------------------------
//------------------------------------------------------------------------------------------
int user_end () {
  // Close files
  fclose(fp_data_out);
  // Display tick times
  double metropolis_time;
  metropolis_time = (((RUN_TICKS_LIST.tick_cnt[0]*(double)1.0) * TICK_PERIOD) * exp10(12));
  // Print Statistics
  printf ("Metropolis ticks average: %f ticks\n", ((RUN_TICKS_LIST.tick_cnt[0] * 1.0) /
                                                  (PPE_ITERATIONS * SPE_ITERATIONS)));
  printf ("Metropolis time: %f ps\n", metropolis_time);
  printf ("Metropolis spin update time: %f ps/spin\n", (metropolis_time) /
                                  (SPE_ITERATIONS * VOLUME * N_REPLICAS));
  return 0;
}
//------------------------------------------------------------------------------------------
//------------------------------------------------------------------------------------------
void run(spe_program_handle_t * program_pointer) {
  //----------------------------------------------------------------------------------------
  user_main (program_pointer);
  //----------------------------------------------------------------------------------------
  //----------------------------------------------------------------------------------------
  user_end ();
  //----------------------------------------------------------------------------------------
}
//------------------------------------------------------------------------------------------
//------------------------------------------------------------------------------------------
static char * params = "ha:b:c:d:e:";
void usage() {
  printf("Usage of spinglasse_ppe:\n" \
         "\t-a <float>\tbeta start value  (default: %f)\n"   \
         "\t-b <float>\tbeta stop value   (default: %f)\n"   \
         "\t-c <int>  \tbeta split value  (default: %.0f)\n" \
         "\t-d <int>  \titerations on PPE (default: %d)\n"   \
         "\t-e <int>  \titerations on SPE (default: %d)\n"   \
         "", BETA_START, BETA_STOP, BETA_SPLIT, PPE_ITERATIONS, SPE_ITERATIONS);
}
int parseArguments(int argc, char * * argv) {
  int i, tmp;
  opterr = 0;  /* don't want writing to stderr */
  while ( (i = getopt(argc, argv, params)) != -1) {
    switch (i) {
      /*
       * Handling options
       */
    case 'h':       /* help option */
      usage();
      return -1;
      break;
    case 'a':        /* beta start value */
      BETA_START = atof(optarg);
      if(BETA_START <= 0.0) {
        fprintf(stderr, "beta start value must be greater than 0\n");
        return -1;
      }
      break;
    case 'b':        /* beta start value */
      BETA_STOP = atof(optarg);
      if(BETA_STOP <= 0.0) {
        fprintf(stderr, "beta stop value must be greater than 0\n");
        return -1;
      }
      if(BETA_STOP <= BETA_START) {
        fprintf(stderr, "beta stop value must be greater than beta start value\n");
        return -1;
      }
      break;
    case 'c':        /* beta split value */
      tmp = atoi(optarg);
      if(tmp <= 0) {
        fprintf(stderr, "beta split value must be greater than 0\n");
        return -1;
      }
```

```
      BETA_SPLIT = (double)tmp;
      break;
    case 'd':        /* PPE iterations */
      tmp = atoi(optarg);
      if(tmp <= 0) {
        fprintf(stderr, "PPE iterations must be greater than 0\n");
        return -1;
      }
      PPE_ITERATIONS = tmp;
      break;
    case 'e':        /* SPE iterations */
      tmp = atoi(optarg);
      if(tmp <= 0) {
        fprintf(stderr, "SPE iterations must be greater than 0\n");
        return -1;
      }
      SPE_ITERATIONS = tmp;
      break;
    case '?':
      printf("Unrecognized options -%c\n", optopt);
      usage();
    default:         /* should not reached */
      usage();
    }
  }
  return 0;
}
int main (int argc, char * * argv) {
  int spe;
  if(parseArguments(argc, argv)) {
    exit(1);
  }
  BETA_INC = ((BETA_STOP - BETA_START) / BETA_SPLIT);
  // Alignment and size check of ALL buffers involved in DMA transfers
  if((CHECK_DMA_128B_ALIGNMENT((unsigned int)Spins)) != 0) {
    printf("ERROR: Spins is not 128bytes aligned: %p\n", (void *)Spins);
    exit(1);
  }
  if((CHECK_DMA_128B_ALIGNMENT((unsigned int)Jx)) != 0) {
    printf("ERROR: Jx is not 128bytes aligned: %p\n", (void *)Jx);
    exit(1);
  }
  if((CHECK_DMA_128B_ALIGNMENT((unsigned int)Jy)) != 0) {
    printf("ERROR: Jy is not 128bytes aligned: %p\n", (void *)Jy);
    exit(1);
  }
  if((CHECK_DMA_128B_ALIGNMENT((unsigned int)Jz)) != 0) {
    printf("ERROR: Jz is not 128bytes aligned: %p\n", (void *)Jz);
    exit(1);
  }
  if((CHECK_DMA_128B_ALIGNMENT((unsigned int)ira)) != 0) {
    printf("ERROR: ira is not 128bytes aligned: %p\n", (void *)ira);
    exit(1);
  }
  if((CHECK_DMA_SIZE(sizeof(spe_params))) != 0) {
    printf ("ERROR: spe_params type size is not valid: %ld bytes\n\n", sizeof(spe_params));
    exit (1);
  }
  if((CHECK_DMA_SIZE(sizeof(spinglasses_params))) != 0) {
    printf ("ERROR: spe_params type size is not valid: %ld bytes\n\n", sizeof(spe_params));
    exit (1);
  }
  if((CHECK_DMA_SIZE(sizeof(run_ticks))) != 0) {
    printf ("\nERROR: run_ticks type size is not valid: %lu bytes\n\n", sizeof(run_ticks));
    exit (1);
  }
  for ( spe = 0; spe < SPE_THREADS; spe++ ) {
  }
  //------------------------------------------------------------------------------------------
  // Global variables initialization
  if(!(stdout = fopen("stdout_at_" HOSTNAME, "w"))) {
    printf ("\nERROR: can't create/write file stdout_at_" HOSTNAME "\n\n");
    exit (1);
  }
  if(!(fp_data_out = fopen("data.out_at_" HOSTNAME, "w"))) {
    printf ("\nERROR: can't create/write file data.out_at_" HOSTNAME "\n\n");
    exit (1);
  }
  //------------------------------------------------------------------------------------------
  //------------------------------------------------------------------------------------------
  run ( &spe_program );
  //------------------------------------------------------------------------------------------
  exit (0);
```

```
}
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
// Spin Glasses
void loadConfigurationFromFile() {
  set_defaults();
  init_random();
  init_spins();
  init_couplings();
  // Init the status struct
  RUN_STATUS_LIST.status=0;
  //---------------------------------------------------------------------------------------
  //---------------------------------------------------------------------------------------
  // Init the param struct
  SPE_PARAMS_LIST.Spins_addr     = (ADDR)Spins;
  SPE_PARAMS_LIST.Jx_addr        = (ADDR)Jx;
  SPE_PARAMS_LIST.Jy_addr        = (ADDR)Jy;
  SPE_PARAMS_LIST.Jz_addr        = (ADDR)Jz;
  SPE_PARAMS_LIST.ira_addr       = (ADDR)ira;
  SPE_PARAMS_LIST.run_ticks_addr = (ADDR)&(RUN_TICKS_LIST);
  SPE_PARAMS_LIST.sg_param_addr  = (ADDR)&SG_PARAMS_LIST;
  SG_PARAMS_LIST.T_4             = (1 / (beta * 4));
  SG_PARAMS_LIST.C_inv_max_rand  = C_inv_max_rand;
  SG_PARAMS_LIST.ip              = ip;
  SG_PARAMS_LIST.ip1             = ip1;
  SG_PARAMS_LIST.ip2             = ip2;
  SG_PARAMS_LIST.ip3             = ip3;
  SG_PARAMS_LIST.irr             = irr;
  SG_PARAMS_LIST.iter            = SPE_ITERATIONS;
  //---------------------------------------------------------------------------------------
}
void set_defaults() {
  // OU ///////////////////////////////////////////////////
  SG_PARAMS_LIST.OU[0].array[0]=0;
  SG_PARAMS_LIST.OU[0].array[1]=0;
  SG_PARAMS_LIST.OU[0].array[2]=0;
  SG_PARAMS_LIST.OU[0].array[3]=0;
  SG_PARAMS_LIST.OU[1].array[0]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[1].array[1]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[1].array[2]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[1].array[3]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[2].array[0]=0;
  SG_PARAMS_LIST.OU[2].array[1]=0;
  SG_PARAMS_LIST.OU[2].array[2]=0;
  SG_PARAMS_LIST.OU[2].array[3]=0;
  SG_PARAMS_LIST.OU[3].array[0]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[3].array[1]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[3].array[2]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[3].array[3]=~((unsigned int)(0));
  // OD ///////////////////////////////////////////////////
  SG_PARAMS_LIST.OD[0].array[0]=0;
  SG_PARAMS_LIST.OD[0].array[1]=0;
  SG_PARAMS_LIST.OD[0].array[2]=0;
  SG_PARAMS_LIST.OD[0].array[3]=0;
  SG_PARAMS_LIST.OD[1].array[0]=0;
  SG_PARAMS_LIST.OD[1].array[1]=0;
  SG_PARAMS_LIST.OD[1].array[2]=0;
  SG_PARAMS_LIST.OD[1].array[3]=0;
  SG_PARAMS_LIST.OD[2].array[0]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[2].array[1]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[2].array[2]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[2].array[3]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[3].array[0]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[3].array[1]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[3].array[2]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[3].array[3]=~((unsigned int)(0));
}
void init_spins() {
  int i;
  for(i=0; i<VOLUME; i++) {
    Spins[i][0] = rand();
    Spins[i][1] = rand();
    Spins[i][2] = rand();
    Spins[i][3] = rand();
  }
}
void init_couplings(){
  int i;
  for(i=0; i<VOLUME; i++) {
    Jx[i][0] = 1; rand();
    Jx[i][1] = 1; rand();
```

```
    Jx[i][2] = 1; rand();
    Jx[i][3] = 1; rand();
    Jy[i][0] = 1; rand();
    Jy[i][1] = 1; rand();
    Jy[i][2] = 1; rand();
    Jy[i][3] = 1; rand();
    Jz[i][0] = 1; rand();
    Jz[i][1] = 1; rand();
    Jz[i][2] = 1; rand();
    Jz[i][3] = 1; rand();
  }
}
void init_random() {
  int i;
  srand(SEED);
  ip = rand();
  irr = rand();
  for (i=0; i<256; i++) {
    // All systems have the same random number
    ira[i][0] = ira[i][1] = ira[i][2] = ira[i][3] = rand();
    // 4 random numbers for 4 systems with 32 replicas
    /*ira[i][1] = */ rand();
    /*ira[i][2] = */ rand();
    /*ira[i][3] = */ rand();
  }
  ip1=(ip+232)%256; ip2=(ip+201)%256; ip3=(ip+195)%256;
}
//-----------------------------------------------------------------------------------------
```

B.2 Lato SPE

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <spu_mfcio.h>
#include <spu_intrinsics.h>
#include <spu_internals.h>
#include <assert.h>
#include <simdmath/log2f4.h>
#include "../common.h"
#include "../user_common.h"
#include "../spe_lib.h"
//-----------------------------------------------------------------------------------------
// Shared data
spe_params SPE_PARAMETERS                        __attribute__ ((aligned (128)));
spinglasses_params SG_PARAMETERS                 __attribute__ ((aligned (128)));
vector unsigned int Spins[VOLUME]                __attribute__ ((aligned (128)));
vector unsigned int Jx[VOLUME]                   __attribute__ ((aligned (128)));
vector unsigned int Jy[VOLUME]                   __attribute__ ((aligned (128)));
vector unsigned int Jz[VOLUME]                   __attribute__ ((aligned (128)));
vector unsigned int ira[SPE_RANDOM_VECTOR_SIZE]  __attribute__ ((aligned (128)));
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
// Local data
const vector unsigned int ones     = { (signed int)1, (signed int)1,
                                       (signed int)1, (signed int)1 };
const vector unsigned int mask     = { 0xff, 0xff, 0xff, 0xff };
const vector float THREES          = { 3.0, 3.0, 3.0, 3.0 };
const vector signed int multiplier = { 1,      SIDE,   FACE,   0 };
const vector signed int vSIDE_1    = { SIDE-1, SIDE-1, SIDE-1, SIDE-1 };
const vector signed int vSIDE      = { SIDE, SIDE, SIDE, SIDE };
const vector signed int vZERO      = {    0,     0,     0,     0 };
// The alignement causes a minor numbers of rotations when used in association with a vector
unsigned int inSpin __attribute__ ((aligned (4)));
unsigned int x, y, z, go = 1;
vector unsigned int p_ips = {0, 0, 0, 0};
// Tick times
unsigned int t_start         = 0;   // Start ticks
unsigned int t_end           = 0;   // End ticks
unsigned int t_avg           = 0;
vector float C_inv_max_rand;
vector float negative_T_4;
vector float LN_2 = { (float)0.69314718055995, (float)0.69314718055995,
                      (float)0.69314718055995, (float)0.69314718055995 };
unsigned long long int addr;
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
run_status RUN_STATUS __attribute__ ((aligned (128)));
run_ticks RUN_TICKS   __attribute__ ((aligned (128)));
```

```
//------------------------------------------------------------------------------------------
void sendMailToPPE() {
  do { } while(spu_stat_out_mbox () == 0);
  spu_write_out_mbox(1);
}
int waitMailFromPPE() {
  while(spu_stat_in_mbox() == 0) {};
  return spu_read_in_mbox ();
}
void loadConfiguration() {
  register int offset;
  unsigned int i;
  //----------------------------------------------------------------------------------------
  // Get param struct from gived address
  mfc_get( (void *) &SPE_PARAMETERS, addr, sizeof (spe_params), 1, 0, 0 );
  mfc_write_tag_mask ( 1 << 1 );
  mfc_read_tag_status_all ();
  offset = 0;
  for(i=0; i<VOLUME; i++) {
    mfc_get( (void *) &Spins + offset, SPE_PARAMETERS.Spins_addr + offset, 16, 1, 0, 0 );
    mfc_get( (void *) &Jx    + offset, SPE_PARAMETERS.Jx_addr    + offset, 16, 1, 0, 0 );
    mfc_get( (void *) &Jy    + offset, SPE_PARAMETERS.Jy_addr    + offset, 16, 1, 0, 0 );
    mfc_get( (void *) &Jz    + offset, SPE_PARAMETERS.Jz_addr    + offset, 16, 1, 0, 0 );
    offset += 16;
  }
  mfc_get( (void *) &ira, SPE_PARAMETERS.ira_addr, sizeof (ira), 1, 0, 0 );
  mfc_get( (void *) &SG_PARAMETERS, SPE_PARAMETERS.sg_param_addr, sizeof (SG_PARAMETERS),
                                                                                1, 0, 0 );
  mfc_write_tag_mask ( 1 << 1 );
  mfc_read_tag_status_all ();
  C_inv_max_rand = spu_splats((float)SG_PARAMETERS.C_inv_max_rand);
  negative_T_4   = spu_splats((float)-SG_PARAMETERS.T_4);
  p_ips = spu_insert((signed int)SG_PARAMETERS.ip,  p_ips, 0);
  p_ips = spu_insert((signed int)SG_PARAMETERS.ip1, p_ips, 1);
  p_ips = spu_insert((signed int)SG_PARAMETERS.ip2, p_ips, 2);
  p_ips = spu_insert((signed int)SG_PARAMETERS.ip3, p_ips, 3);
}
void saveConfiguration() {
  register int offset;
  unsigned int i;
  offset = 0;
  for(i=0; i<VOLUME; i++) {
    mfc_put( (void *) &Spins + offset, SPE_PARAMETERS.Spins_addr + offset, 16, 1, 0, 0 );
    offset += 16;
  }
  mfc_write_tag_mask ( 1 << 1 );
  mfc_read_tag_status_all ();
}
//------------------------------------------------------------------------------------------
int user_main ( ) {
  unsigned int i;
  vector float IDXf;
  vector unsigned int IDX,
                      select,
                      Carry1,
                      Carry2,
                      EU,
                      ED,
                      EC,
                      prod;
  vector signed int xyzp  = {0, 0, 0, 0},
                    xyzm  = {0, 0, 0, 0};
  vector unsigned int pos_sel, neg_sel;
  unsigned int haddr, laddr;
  //----------------------------------------------------------------------------------------
  // Load SPE parameters; waiting for param struct address
  while ( spu_stat_in_mbox () == 0 ) {} haddr = spu_read_in_mbox ();
  while ( spu_stat_in_mbox () == 0 ) {} laddr = spu_read_in_mbox ();
  addr = (((unsigned long long int)haddr)<<32) | laddr;
  //----------------------------------------------------------------------------------------
  // Metropolis Update
  loadConfiguration();
  do {
    // Counter reset
    spu_writech (SPU_WrDec, 0x7fffffff);
    t_start = spu_readch (SPU_RdDec);
    for(i=0; i<SG_PARAMETERS.iter; i++) {
      // Loop through vectors
      for(z=0; z<SIDE; z++) {
        for(y=0; y<SIDE; y++) {
          for(x=0; x<SIDE; x++) {
```

```
vector signed int xyz   = { x, y, z, 0 };
vector signed int adder = { (y * SIDE) + (z * FACE), x + (z*FACE),
                            x + (y * SIDE), 0 };
inSpin = (x + (y * SIDE) + (z * FACE));
////////////////////////////////////////////////////////////////////////////////
// RANDOM
//
IDX = spu_add(ira[spu_extract(p_ips, 1)], ira[spu_extract(p_ips, 2)]);
ira[spu_extract(p_ips, 0)] = IDX;
IDX = spu_xor(IDX, ira[spu_extract(p_ips, 3)]);
// Add one to p_ips and mask at 255
p_ips = spu_add(p_ips, ones);
p_ips = spu_and(p_ips, mask);
////////////////////////////////////////////////////////////////////////////////
// IDXf = N_RANDOM;
//
IDXf = spu_mul(spu_convtf(IDX, 0), C_inv_max_rand);
////////////////////////////////////////////////////////////////////////////////
// Logarithm
//
IDXf = spu_mul(_log2f4(IDXf), LN_2);
////////////////////////////////////////////////////////////////////////////////
// Parte finale di N_RANDOM
//
IDXf = spu_mul(IDXf, negative_T_4);
////////////////////////////////////////////////////////////////////////////////
// Blocco dell'if()
//
select = spu_cmpabsgt(IDXf, THREES);
IDXf   = spu_sel(IDXf, THREES, select);
IDX    = spu_convtu(IDXf, 0);
////////////////////////////////////////////////////////////////////////////////
// Calcolo le coordinate dei vicini
//
// Coordinate positive                // Coordinate negative
xyzp = spu_add(xyz, 1);               xyzm = spu_add(xyz, -1);
pos_sel = spu_cmpeq(xyzp, SIDE);      neg_sel = spu_cmpgt(vZERO, xyzm);
xyzp = spu_sel(xyzp, vZERO, pos_sel); xyzm = spu_sel(xyzm, vSIDE_1, neg_sel);
xyzp = spu_madd((vector signed short)xyzp, (vector signed short)multiplier,
                                                                  adder);
xyzm = spu_madd((vector signed short)xyzm, (vector signed short)multiplier,
                                                                  adder);
////////////////////////////////////////////////////////////////////////////////
// Calcolo il valore dello spin
//
EU     = spu_xor(Spins[inSpin], spu_xor(Jx[inSpin],
                   Spins[spu_extract(xyzp, 0)]));
prod   = spu_xor(Spins[inSpin], spu_xor(Jx[spu_extract(xyzm, 0)],
                                   Spins[spu_extract(xyzm, 0)]) );
Carry1 = spu_and(prod,     EU);
EU     = spu_xor(prod,     EU);
ED     = Carry1;
prod   = spu_xor(Spins[inSpin], spu_xor(Jy[inSpin],
                   Spins[spu_extract(xyzp, 1)]));
Carry1 = spu_and(prod,     EU);
EU     = spu_xor(prod,     EU);
// Carry2 = spu_and(Carry1,   ED);
ED     = spu_xor(Carry1,   ED);
// EC     = Carry2;
prod   = spu_xor(Spins[inSpin], spu_xor(Jy[spu_extract(xyzm, 1)],
                                   Spins[spu_extract(xyzm, 1)]) );
Carry1 = spu_and(prod,     EU);
EU     = spu_xor(prod,     EU);
Carry2 = spu_and(Carry1,   ED);
ED     = spu_xor(Carry1,   ED);
EC     = spu_xor(Carry2,   EC);
prod   = spu_xor(Spins[inSpin], spu_xor(Jz[inSpin],
                  Spins[spu_extract(xyzp, 2)]) );
Carry1 = spu_and(prod,     EU);
EU     = spu_xor(prod,     EU);
Carry2 = spu_and(Carry1,   ED);
ED     = spu_xor(Carry1,   ED);
EC     = spu_xor(Carry2,   EC);
prod   = spu_xor(Spins[inSpin], spu_xor(Jz[spu_extract(xyzm, 2)],
                                   Spins[spu_extract(xyzm, 2)]) );
Carry1 = spu_and(prod,     EU);
EU     = spu_xor(prod,     EU);
Carry2 = spu_and(Carry1,   ED);
ED     = spu_xor(Carry1,   ED);
EC     = spu_xor(Carry2,   EC);
```

```
            EC = spu_xor(EC, 0xffffffff);   /* EC=~EC; ED=~ED; EU=~EU;            */
                                            /* negation is convenient: see above */
            ED = spu_xor(ED, 0xffffffff);
            EU = spu_xor(EU, 0xffffffff);
            /*OD[IDX],OU[IDX] represent IDX positionally*/
            Carry1 = spu_and(SG_PARAMETERS.OU[spu_extract(IDX, 0)].vec, EU);
            Carry2 = spu_or(spu_and(SG_PARAMETERS.OD[spu_extract(IDX, 0)].vec, ED),
                    spu_and(spu_or(SG_PARAMETERS.OD[spu_extract(IDX, 0)].vec, ED), Carry1));
            /*Carry2=1 if and only if (T/4)logR+(~ED,~EU) >= 4*/
            Spins[inSpin] = spu_xor(Spins[inSpin], spu_or(Carry2, EC));
////////////////////////////////////////////////////////////////////////////////////////////
            x++;
            xyz = spu_insert(x, xyz, 0); // xyz   = { x, y, z, 0 };
            xyz = spu_insert(y, xyz, 1);
            xyz = spu_insert(z, xyz, 2);
            adder = spu_insert((y * SIDE) + (z * FACE), adder, 0);
            adder = spu_insert(x + (z*FACE),            adder, 1);
            adder = spu_insert(x + (y * SIDE),          adder, 2);
            inSpin = (x + (y * SIDE) + (z * FACE));
            ////////////////////////////////////////////////////////////////////////////////
            // RANDOM
            //
            IDX = spu_add(ira[spu_extract(p_ips, 1)], ira[spu_extract(p_ips, 2)]);
            ira[spu_extract(p_ips, 0)] = IDX;
            IDX = spu_xor(IDX, ira[spu_extract(p_ips, 3)]);
            // Add one to p_ips and mask at 255
            p_ips = spu_add(p_ips, ones);
            p_ips = spu_and(p_ips, mask);
            ////////////////////////////////////////////////////////////////////////////////
            // IDXf = N_RANDOM;
            //
            IDXf = spu_mul(spu_convtf(IDX, 0), C_inv_max_rand);
            ////////////////////////////////////////////////////////////////////////////////
            // Logarithm
            //
            IDXf = spu_mul(_log2f4(IDXf), LN_2);
            ////////////////////////////////////////////////////////////////////////////////
            // Parte finale di N_RANDOM
            //
            IDXf = spu_mul(IDXf, negative_T_4);
            ////////////////////////////////////////////////////////////////////////////////
            // Blocco dell'if()
            //
            select = spu_cmpabsgt(IDXf, THREES);
            IDXf   = spu_sel(IDXf, THREES, select);
            IDX    = spu_convtu(IDXf, 0);
            ////////////////////////////////////////////////////////////////////////////////
            // Calcolo le coordinate dei vicini
            //
            // Coordinate positive                // Coordinate negative
            xyzp = spu_add(xyz, 1);               xyzm = spu_add(xyz, -1);
            pos_sel = spu_cmpeq(xyzp, SIDE);      neg_sel = spu_cmpgt(vZERO, xyzm);
            xyzp = spu_sel(xyzp, vZERO, pos_sel); xyzm = spu_sel(xyzm, vSIDE_1, neg_sel);
            xyzp = spu_madd((vector signed short)xyzp, (vector signed short)multiplier,
                                                                                adder);
            xyzm = spu_madd((vector signed short)xyzm, (vector signed short)multiplier,
                                                                                adder);
            ////////////////////////////////////////////////////////////////////////////////
            // Calcolo il valore dello spin
            //
            EU     = spu_xor(Spins[inSpin], spu_xor(Jx[inSpin],
                              Spins[spu_extract(xyzp, 0)]));
            prod   = spu_xor(Spins[inSpin], spu_xor(Jx[spu_extract(xyzm, 0)],
                                           Spins[spu_extract(xyzm, 0)]) );
            Carry1 = spu_and(prod,     EU);
            EU     = spu_xor(prod,     EU);
            ED     = Carry1;
            prod   = spu_xor(Spins[inSpin], spu_xor(Jy[inSpin],
                             Spins[spu_extract(xyzp, 1)]) );
            Carry1 = spu_and(prod,     EU);
            EU     = spu_xor(prod,     EU);
            Carry2 = spu_and(Carry1,   ED);
            ED     = spu_xor(Carry1,   ED);
            EC     = Carry2;
            prod   = spu_xor(Spins[inSpin], spu_xor(Jy[spu_extract(xyzm, 1)],
                                           Spins[spu_extract(xyzm, 1)]) );
            Carry1 = spu_and(prod,     EU);
            EU     = spu_xor(prod,     EU);
            Carry2 = spu_and(Carry1,   ED);
            ED     = spu_xor(Carry1,   ED);
```

```
            EC     = spu_xor(Carry2,   EC);
            prod   = spu_xor(Spins[inSpin], spu_xor(Jz[inSpin],
                                  Spins[spu_extract(xyzp, 2)]) );
            Carry1 = spu_and(prod,     EU);
            EU     = spu_xor(prod,     EU);
            Carry2 = spu_and(Carry1,   ED);
            ED     = spu_xor(Carry1,   ED);
            EC     = spu_xor(Carry2,   EC);
            prod   = spu_xor(Spins[inSpin], spu_xor(Jz[spu_extract(xyzm, 2)],
                                              Spins[spu_extract(xyzm, 2)]) );
            Carry1 = spu_and(prod,     EU);
            EU     = spu_xor(prod,     EU);
            Carry2 = spu_and(Carry1,   ED);
            ED     = spu_xor(Carry1,   ED);
            EC     = spu_xor(Carry2,   EC);
            EC = spu_xor(EC, 0xffffffff);    /* EC=~EC; ED=~ED; EU=~EU;           */
                                             /* negation is convenient: see above */
            ED = spu_xor(ED, 0xffffffff);
            EU = spu_xor(EU, 0xffffffff);
            /*OD[IDX],OU[IDX] represent IDX positionally*/
            Carry1 = spu_and(SG_PARAMETERS.OU[spu_extract(IDX, 0)].vec, EU);
            Carry2 = spu_or(spu_and(SG_PARAMETERS.OD[spu_extract(IDX, 0)].vec, ED),
                    spu_and(spu_or(SG_PARAMETERS.OD[spu_extract(IDX, 0)].vec, ED), Carry1));
            /*Carry2=1 if and only if (T/4)logR+(~ED,~EU) >= 4*/
            Spins[inSpin] = spu_xor(Spins[inSpin], spu_or(Carry2, EC));
//////////////////////////////////////////////////////////////////////////////////////////
            x++;
            xyz = spu_insert(x, xyz, 0);
            xyz = spu_insert(y, xyz, 1);
            xyz = spu_insert(z, xyz, 2);
            adder = spu_insert((y * SIDE) + (z * FACE), adder, 0);
            adder = spu_insert(x + (z*FACE),            adder, 1);
            adder = spu_insert(x + (y * SIDE),          adder, 2);
            inSpin = (x + (y * SIDE) + (z * FACE));
            ////////////////////////////////////////////////////////////////////////////////
            // RANDOM
            //
            IDX = spu_add(ira[spu_extract(p_ips, 1)], ira[spu_extract(p_ips, 2)]);
            ira[spu_extract(p_ips, 0)] = IDX;
            IDX = spu_xor(IDX, ira[spu_extract(p_ips, 3)]);
            // Add one to p_ips and mask at 255
            p_ips = spu_add(p_ips, ones);
            p_ips = spu_and(p_ips, mask);
            ////////////////////////////////////////////////////////////////////////////////
            // IDXf = N_RANDOM;
            //
            IDXf = spu_mul(spu_convtf(IDX, 0), C_inv_max_rand);
            ////////////////////////////////////////////////////////////////////////////////
            // Logarithm
            //
            IDXf = spu_mul(_log2f4(IDXf), LN_2);
            ////////////////////////////////////////////////////////////////////////////////
            // Parte finale di N_RANDOM
            //
            IDXf = spu_mul(IDXf, negative_T_4);
            ////////////////////////////////////////////////////////////////////////////////
            // Blocco dell'if()
            //
            select = spu_cmpabsgt(IDXf, THREES);
            IDXf   = spu_sel(IDXf, THREES, select);
            IDX    = spu_convtu(IDXf, 0);
            ////////////////////////////////////////////////////////////////////////////////
            // Calcolo le coordinate dei vicini
            //
            // Coordinate positive                // Coordinate negative
            xyzp = spu_add(xyz, 1);               xyzm = spu_add(xyz, -1);
            pos_sel = spu_cmpeq(xyzp, SIDE);      neg_sel = spu_cmpgt(vZERO, xyzm);
            xyzp = spu_sel(xyzp, vZERO, pos_sel); xyzm = spu_sel(xyzm, vSIDE_1, neg_sel);
            xyzp = spu_madd((vector signed short)xyzp, (vector signed short)multiplier,
                                                                              adder);
            xyzm = spu_madd((vector signed short)xyzm, (vector signed short)multiplier,
                                                                              adder);
            ////////////////////////////////////////////////////////////////////////////////
            // Calcolo il valore dello spin
            //
            EU     = spu_xor(Spins[inSpin], spu_xor(Jx[inSpin],
                                  Spins[spu_extract(xyzp, 0)]) );
            prod   = spu_xor(Spins[inSpin], spu_xor(Jx[spu_extract(xyzm, 0)],
                                              Spins[spu_extract(xyzm, 0)]) );
            Carry1 = spu_and(prod,     EU);
```

```
            EU     = spu_xor(prod,     EU);
            ED     = Carry1;
            prod   = spu_xor(Spins[inSpin], spu_xor(Jy[inSpin],
                            Spins[spu_extract(xyzp, 1)]) );
            Carry1 = spu_and(prod,     EU);
            EU     = spu_xor(prod,     EU);
            Carry2 = spu_and(Carry1,   ED);
            ED     = spu_xor(Carry1,   ED);
            EC     = Carry2;
            prod   = spu_xor(Spins[inSpin], spu_xor(Jy[spu_extract(xyzm, 1)],
                                   Spins[spu_extract(xyzm, 1)]) );
            Carry1 = spu_and(prod,     EU);
            EU     = spu_xor(prod,     EU);
            Carry2 = spu_and(Carry1,   ED);
            ED     = spu_xor(Carry1,   ED);
            EC     = spu_xor(Carry2,   EC);
            prod   = spu_xor(Spins[inSpin], spu_xor(Jz[inSpin],
                            Spins[spu_extract(xyzp, 2)]) );
            Carry1 = spu_and(prod,     EU);
            EU     = spu_xor(prod,     EU);
            Carry2 = spu_and(Carry1,   ED);
            ED     = spu_xor(Carry1,   ED);
            EC     = spu_xor(Carry2,   EC);
            prod   = spu_xor(Spins[inSpin], spu_xor(Jz[spu_extract(xyzm, 2)],
                                   Spins[spu_extract(xyzm, 2)]) );
            Carry1 = spu_and(prod,     EU);
            EU     = spu_xor(prod,     EU);
            Carry2 = spu_and(Carry1,   ED);
            ED     = spu_xor(Carry1,   ED);
            EC     = spu_xor(Carry2,   EC);
            EC = spu_xor(EC, 0xffffffff);   /* EC=~EC; ED=~ED; EU=~EU;           */
                                            /* negation is convenient: see above */
            ED = spu_xor(ED, 0xffffffff);
            EU = spu_xor(EU, 0xffffffff);
            /*OD[IDX],OU[IDX] represent IDX positionally*/
            Carry1 = spu_and(SG_PARAMETERS.OU[spu_extract(IDX, 0)].vec, EU);
            Carry2 = spu_or(spu_and(SG_PARAMETERS.OD[spu_extract(IDX, 0)].vec, ED),
                    spu_and(spu_or(SG_PARAMETERS.OD[spu_extract(IDX, 0)].vec, ED), Carry1));
            /*Carry2=1 if and only if (T/4)logR+(~ED,~EU) >= 4*/
            Spins[inSpin] = spu_xor(Spins[inSpin], spu_or(Carry2, EC));
          }
        }
      }
    }
    t_end = spu_readch (SPU_RdDec);
    if(t_avg == 0) t_avg = (t_start - t_end);
    else t_avg = (t_avg + (t_start - t_end)) / 2;
    saveConfiguration();
    sendMailToPPE();
    go = waitMailFromPPE();
  } while(go);
  RUN_TICKS.tick_cnt[0] = t_avg;
  //---------------------------------------------------------------------------------------
  //---------------------------------------------------------------------------------------
  // Put ticks
  mfc_put ( (void *) &RUN_TICKS, SPE_PARAMETERS.run_ticks_addr, sizeof (run_ticks),
                                                                     1, 0, 0 );
  mfc_write_tag_mask ( 1 << 1 );
  mfc_read_tag_status_all ();
  //---------------------------------------------------------------------------------------
  return (0);
}
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
int main (/* unsigned long long spu_id, unsigned long long argv */) {
  user_main ( );
  return (0);
}
//-----------------------------------------------------------------------------------------
```

B.3 File di libreria

```
//-------------------------------------------------------------------------------------------//
//                                                                                           //
// Define here all data structures and constant that have to be shared between PPE and SPEs. //
//                                                                                           //
```

```
//-------------------------------------------------------------------------------------------//
#ifndef __USER_COMMON__
#define __USER_COMMON__
//-------------------------------------------------------------------------------------------
// cab params
#define CAB_TIMEBASE        (14318000)
#define CAB_HOSTNAME        "cab"
// cc03.juice params
#define CC09_JUICE_TIMEBASE (14318000)
#define CC09_JUICE_HOSTNAME "cc09.juice"
// ps3.fe.infn.it params
#define PS3_TIMEBASE        (79800000)                           // PS3 Timebase
#define PS3_HZ              (3.2e9)                              // PS3 Frequency HZ
#define PS3_TICK_PERIOD     (1.0 / PS3_TIMEBASE)                 // Tick period in seconds
#define PS3_CLOCK_PERIOD    (1.0 / PS3_HZ)                       // PS3 clock period sec
#define PS3_TICK_N_CLK      (PS3_TICK_PERIOD / PS3_CLOCK_PERIOD) // Number of clocks in a tick
#define PS3_HOSTNAME        "ps3.fe.infn.it"
#define TICK_PERIOD         (1.0 / PS3_TIMEBASE)
#define HOSTNAME            PS3_HOSTNAME
//-------------------------------------------------------------------------------------------
// Parameters of SPE
#define ADDR unsigned long long int
typedef union _array_to_vector {
unsigned int        array[4] __attribute__ ((aligned(16)));
#ifndef PPE_SIDE
vector unsigned int vec;
#endif
} array_to_vector;
typedef struct {
  unsigned char pad[4];
  array_to_vector OD[4];
  array_to_vector OU[4];
  unsigned char ip,
                ip1,
                ip2,
                ip3,
                irr;
  float C_inv_max_rand;
  double T_4;
  unsigned int iter;
} spinglasses_params;
typedef struct {
  unsigned char pad[4];           // Padding
  unsigned int idn;               // SPE id            ( 4B)
  ADDR Spins_addr;                // Spins address     ( 8B)
  ADDR Jx_addr;                   // Jx address        ( 8B)
  ADDR Jy_addr;                   // Jy address        ( 8B)
  ADDR Jz_addr;                   // Jz address        ( 8B)
  ADDR ira_addr;                  // Random address    ( 8B)
  ADDR run_ticks_addr;            // Ticks struct addr ( 8B)
  ADDR sg_param_addr;             // Sg    struct addr ( 8B)
                                  // - Total --------- (??B)
} spe_params;
//-------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------
// It VERY important that sizeof(run_status)<=16, because it must be
// transferred with an ATOMIC dma operation
typedef struct {
  unsigned int status;
} run_status;
//-------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------
typedef struct {
  unsigned char pad[10];           // Padding
  unsigned int tick_cnt[1];
} run_ticks;
//-------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------
// Metropolis
#define SIDE       (15)
#define FACE       (SIDE * SIDE)
#define VOLUME     (FACE * SIDE)
// Size of the random table
#define SPE_RANDOM_VECTOR_SIZE (256)
// Seed of the random numbers
#define SEED (unsigned int)1
// Number of replicas (size of a vector unsigned int in bits)
#define N_REPLICAS (128)
//-------------------------------------------------------------------------------------------
#endif
```

Appendix C

Implementazione per Cell, versione sgv11_8_4unroll

C.1 Lato PPE

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <unistd.h>
#include "ppe_lib.h"
#include "common.h"
#include "user_common.h"
// Number of SPE to use
#define SPE_THREADS     1
//-----------------------------------------------------------------------------------------
// List of SPE-clients
extern spe_program_handle_t spe_program;
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
// PPE-SPE Sharing
unsigned int Spins[VOLUME][4] __attribute__ ((aligned (128)));
unsigned int Jx[VOLUME][4]    __attribute__ ((aligned (128))),
             Jy[VOLUME][4]    __attribute__ ((aligned (128))),
             Jz[VOLUME][4]    __attribute__ ((aligned (128)));
unsigned int ira[256][4]      __attribute__ ((aligned (128)));
// Magnetizations
double magnetization  [N_REPLICAS];
double avg_magnet     [N_REPLICAS];
double data           [N_REPLICAS];
double data_2         [N_REPLICAS];
double sigma          [N_REPLICAS];
unsigned char ip,
              ip1,
              ip2,
              ip3,
              irr;
const float C_inv_max_rand = 1.0/((double)(~((unsigned/* long int*/)(0))));
double T_4;
FILE * fp_data_out;
unsigned int zero = 0;
unsigned int one  = 1;
unsigned int result;
unsigned int PPE_ITERATIONS = 1000;
unsigned int SPE_ITERATIONS = 1000;
double BETA_START     = 0.01;
double BETA_STOP      = 0.5;
double BETA_SPLIT     = 100.0;
double BETA_INC;
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
myspe_t THREADS;
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
```

```
// PPE -> SPE ( MM -> LS )
spe_params SPE_PARAMS_LIST        __attribute__ ((aligned (128)));
spinglasses_params SG_PARAMS_LIST __attribute__ ((aligned (128)));
// SPE -> PP ( LS -> MM )
volatile run_status RUN_STATUS_LIST __attribute__ ((aligned (128)));
volatile run_ticks RUN_TICKS_LIST   __attribute__ ((aligned (128)));
//-------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------
// Functions declaration
void loadConfigurationFromFile();
void set_defaults();
void init_random();
void init_spins();
void init_couplings();
//-------------------------------------------------------------------------------------------
void startSPE(spe_program_handle_t * program_pointer) {
  START_SPE(THREADS, program_pointer);
}
//-------------------------------------------------------------------------------------------
float beta;
void sendDataMailToSPE() {
  unsigned int out_mail = 0;
  PR_DEBUG ("PPE: sending SPE parameters MM address\n");
  // High address (mfc_ea2h(...))
  out_mail = (unsigned int) ((unsigned long long int)(&(SPE_PARAMS_LIST  ))>>32)&0xffffffff;
  PPE_SM(THREADS, &out_mail);   // Send SPE parameters address (high)
  // Low address (mfc_ea2l(...))
  out_mail = (unsigned int) ((unsigned long long int)(&(SPE_PARAMS_LIST  )))&0xffffffff;
  PPE_SM(THREADS, &out_mail);   // Send SPE parameters address
}
#define waitMailFromSPE()          PPE_RM(THREADS, &result);
#define sendGoMail()               PPE_SM(THREADS, &one)
#define sendDontGoMail()           PPE_SM(THREADS, &zero)
#define printBetaAvgMagnAndSigma() fprintf(fp_data_out, "%f", beta);                \
  for(j=0; j<N_REPLICAS; j++) fprintf(fp_data_out, " %lf %lf", avg_magnet[j], sigma[j]); \
  fprintf(fp_data_out, "\n");
#define stopSPE()                  STOP_SPE(THREADS)
void resetMagnetizationsVars() {
  int k;
  // Reset of magnetizations vars
  for(k=0; k<N_REPLICAS; k++) {
    magnetization[k] = avg_magnet[k] = data[k] = data_2[k] = sigma[k] = 0;
  }
}
void calculateMagnetization() {
  int i, j, k;
  for(i=0; i<VOLUME; i++) {
    for(k=0; k<4; k++) {
      for(j=0; j<32; j++) {
        magnetization[(k*sizeof(unsigned int)*4)+j] +=
               ( ((Spins[i][k] >> j) & 0x1) ) ? 1 : -1;
      }
    }
  }
  for(i=0; i<N_REPLICAS; i++) {
    // Normalization of the magnetizations
    magnetization[i] /= (double)VOLUME;
    // Save the values for the future
    data[i]   +=  pow(magnetization[i], 2.0);
    data_2[i] += fabs(magnetization[i]);
    magnetization[i] = 0;
  }
}
void endMagnetizationCalculus() {
  int k;
  for(k=0; k<N_REPLICAS; k++) {
    avg_magnet[k]  = (data_2[k] / (double)PPE_ITERATIONS);
    // Media dei quadrati dei valori assoluti delle magnetizzazioni
    data[k]        = (data[k]   / (double)PPE_ITERATIONS);
    // Media dei valori assoluti delle magnetizzazioni elevata al quadrato
    data_2[k]      = pow( (data_2[k] / (double)PPE_ITERATIONS), 2.0);
    sigma[k]       = sqrt( data[k] - data_2[k] );
  }
}
void user_main (spe_program_handle_t * program_pointer) {
  unsigned i, j;
  for(beta = BETA_START; beta < BETA_STOP; beta += BETA_INC) {
    loadConfigurationFromFile();
    resetMagnetizationsVars();
    startSPE(program_pointer);
    sendDataMailToSPE();
```

```
    for(i = 0; i < PPE_ITERATIONS; i++) {
      waitMailFromSPE();
      calculateMagnetization();
      sendGoMail();
    }
    endMagnetizationCalculus();
    printBetaAvgMagnAndSigma()
    sendDontGoMail();
    stopSPE();
  }
}
//-------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------
int user_end () {
  // Close files
  fclose(fp_data_out);
  // Display tick times
  double metropolis_time;
  metropolis_time = (((RUN_TICKS_LIST.tick_cnt[0]*(double)1.0) * TICK_PERIOD) * exp10(12));
  // Print Statistics
  printf ("Metropolis ticks average: %f ticks\n", ((RUN_TICKS_LIST.tick_cnt[0] * 1.0) /
                                                  (PPE_ITERATIONS * SPE_ITERATIONS)));
  printf ("Metropolis time: %f ps\n", metropolis_time);
  printf ("Metropolis spin update time: %f ps/spin\n", (metropolis_time) /
                                   (SPE_ITERATIONS * VOLUME * N_REPLICAS));
  return 0;
}
//-------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------
void run(spe_program_handle_t * program_pointer) {
  //-----------------------------------------------------------------------------------------
  user_main (program_pointer);
  //-----------------------------------------------------------------------------------------
  //-----------------------------------------------------------------------------------------
  user_end ();
  //-----------------------------------------------------------------------------------------
}
//-------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------
static char * params = "ha:b:c:d:e:";
void usage() {
  printf("Usage of spinglasse_ppe:\n" \
         "\t-a <float>\tbeta start value  (default: %f)\n"   \
         "\t-b <float>\tbeta stop value   (default: %f)\n"   \
         "\t-c <int>  \tbeta split value  (default: %.0f)\n" \
         "\t-d <int>  \titerations on PPE (default: %d)\n"   \
         "\t-e <int>  \titerations on SPE (default: %d)\n"   \
         "", BETA_START, BETA_STOP, BETA_SPLIT, PPE_ITERATIONS, SPE_ITERATIONS);
}
int parseArguments(int argc, char * * argv) {
  int i, tmp;
  opterr = 0;  /* don't want writing to stderr */
  while ( (i = getopt(argc, argv, params)) != -1) {
    switch (i) {
      /*
       * Handling options
       */
    case 'h':       /* help option */
      usage();
      return -1;
      break;
    case 'a':        /* beta start value */
      BETA_START = atof(optarg);
      if(BETA_START <= 0.0) {
        fprintf(stderr, "beta start value must be greater than 0\n");
        return -1;
      }
      break;
    case 'b':        /* beta start value */
      BETA_STOP = atof(optarg);
      if(BETA_STOP <= 0.0) {
        fprintf(stderr, "beta stop value must be greater than 0\n");
        return -1;
      }
      if(BETA_STOP <= BETA_START) {
        fprintf(stderr, "beta stop value must be greater than beta start value\n");
        return -1;
      }
      break;
    case 'c':        /* beta split value */
      tmp = atoi(optarg);
      if(tmp <= 0) {
        fprintf(stderr, "beta split value must be greater than 0\n");
```

```
        return -1;
      }
      BETA_SPLIT = (double)tmp;
      break;
    case 'd':        /* PPE iterations */
      tmp = atoi(optarg);
      if(tmp <= 0) {
        fprintf(stderr, "PPE iterations must be greater than 0\n");
        return -1;
      }
      PPE_ITERATIONS = tmp;
      break;
    case 'e':        /* SPE iterations */
      tmp = atoi(optarg);
      if(tmp <= 0) {
        fprintf(stderr, "SPE iterations must be greater than 0\n");
        return -1;
      }
      SPE_ITERATIONS = tmp;
      break;
    case '?':
      printf("Unrecognized options -%c\n", optopt);
      usage();
      return -1;
    default:         /* should not reached */
      usage();
    }
  }
  return 0;
}
int main (int argc, char * * argv) {
  int spe;
  if(parseArguments(argc, argv)) {
    exit(1);
  }
  BETA_INC = ((BETA_STOP - BETA_START) / BETA_SPLIT);
  // Alignment and size check of ALL buffers involved in DMA transfers
  if((CHECK_DMA_128B_ALIGNMENT((unsigned int)Spins)) != 0) {
    printf("ERROR: Spins is not 128bytes aligned: %p\n", (void *)Spins);
    exit(1);
  }
  if((CHECK_DMA_128B_ALIGNMENT((unsigned int)Jx)) != 0) {
    printf("ERROR: Jx is not 128bytes aligned: %p\n", (void *)Jx);
    exit(1);
  }
  if((CHECK_DMA_128B_ALIGNMENT((unsigned int)Jy)) != 0) {
    printf("ERROR: Jy is not 128bytes aligned: %p\n", (void *)Jy);
    exit(1);
  }
  if((CHECK_DMA_128B_ALIGNMENT((unsigned int)Jz)) != 0) {
    printf("ERROR: Jz is not 128bytes aligned: %p\n", (void *)Jz);
    exit(1);
  }
  if((CHECK_DMA_128B_ALIGNMENT((unsigned int)ira)) != 0) {
    printf("ERROR: ira is not 128bytes aligned: %p\n", (void *)ira);
    exit(1);
  }
  if((CHECK_DMA_SIZE(sizeof(spe_params))) != 0) {
    printf ("ERROR: spe_params type size is not valid: %ld bytes\n\n", sizeof(spe_params));
    exit (1);
  }
  if((CHECK_DMA_SIZE(sizeof(spinglasses_params))) != 0) {
    printf ("ERROR: spe_params type size is not valid: %ld bytes\n\n", sizeof(spe_params));
    exit (1);
  }
  if((CHECK_DMA_SIZE(sizeof(run_ticks))) != 0) {
    printf ("\nERROR: run_ticks type size is not valid: %lu bytes\n\n", sizeof(run_ticks));
    exit (1);
  }
  // Controllo se SIDE e' una potenza di 2
  if(!ISAPOWEROF2(SIDE)) {
    printf ("\nERROR: SIDE must be a power of 2: %d\n\n", SIDE);
    exit (1);
  }
  // Controllo EXP_SIDE
  if(log2(SIDE) != EXP_SIDE) {
    printf ("\nERROR: EXP_SIDE must be %.0f for SIDE equals to %d: %d\n\n", log2(SIDE),
                                                                        SIDE, EXP_SIDE);
    exit (1);
  }
  for ( spe = 0; spe < SPE_THREADS; spe++ ) {
  }
  //----------------------------------------------------------------------------------------
```

```
  // Global variables initialization
  if(!(stdout = fopen("stdout_at_" HOSTNAME, "w"))) {
    printf ("\nERROR: can't create/write file stdout_at_" HOSTNAME "\n\n");
    exit (1);
  }
  if(!(fp_data_out = fopen("data.out_at_" HOSTNAME, "w"))) {
    printf ("\nERROR: can't create/write file data.out_at_" HOSTNAME "\n\n");
    exit (1);
  }
  //----------------------------------------------------------------------------------------
  //----------------------------------------------------------------------------------------
  run ( &spe_program );
  //----------------------------------------------------------------------------------------
  exit (0);
}
//------------------------------------------------------------------------------------------
//------------------------------------------------------------------------------------------
// Spin Glasses
void loadConfigurationFromFile() {
  set_defaults();
  init_random();
  init_spins();
  init_couplings();
  // Init the status struct
  RUN_STATUS_LIST.status=0;
  //----------------------------------------------------------------------------------------
  //----------------------------------------------------------------------------------------
  // Init the param struct
  SPE_PARAMS_LIST.Spins_addr     = (ADDR)Spins;
  SPE_PARAMS_LIST.Jx_addr        = (ADDR)Jx;
  SPE_PARAMS_LIST.Jy_addr        = (ADDR)Jy;
  SPE_PARAMS_LIST.Jz_addr        = (ADDR)Jz;
  SPE_PARAMS_LIST.ira_addr       = (ADDR)ira;
  SPE_PARAMS_LIST.run_ticks_addr = (ADDR)&(RUN_TICKS_LIST);
  SPE_PARAMS_LIST.sg_param_addr  = (ADDR)&SG_PARAMS_LIST;
  SG_PARAMS_LIST.T_4             = (1 / (beta * 4));
  SG_PARAMS_LIST.C_inv_max_rand  = C_inv_max_rand;
  SG_PARAMS_LIST.ip              = ip;
  SG_PARAMS_LIST.ip1             = ip1;
  SG_PARAMS_LIST.ip2             = ip2;
  SG_PARAMS_LIST.ip3             = ip3;
  SG_PARAMS_LIST.irr             = irr;
  SG_PARAMS_LIST.iter            = SPE_ITERATIONS;
  //----------------------------------------------------------------------------------------
}
void set_defaults() {
  // OU //////////////////////////////////////////////////
  SG_PARAMS_LIST.OU[0].array[0]=0;
  SG_PARAMS_LIST.OU[0].array[1]=0;
  SG_PARAMS_LIST.OU[0].array[2]=0;
  SG_PARAMS_LIST.OU[0].array[3]=0;
  SG_PARAMS_LIST.OU[1].array[0]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[1].array[1]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[1].array[2]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[1].array[3]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[2].array[0]=0;
  SG_PARAMS_LIST.OU[2].array[1]=0;
  SG_PARAMS_LIST.OU[2].array[2]=0;
  SG_PARAMS_LIST.OU[2].array[3]=0;
  SG_PARAMS_LIST.OU[3].array[0]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[3].array[1]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[3].array[2]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[3].array[3]=~((unsigned int)(0));
  // OD //////////////////////////////////////////////////
  SG_PARAMS_LIST.OD[0].array[0]=0;
  SG_PARAMS_LIST.OD[0].array[1]=0;
  SG_PARAMS_LIST.OD[0].array[2]=0;
  SG_PARAMS_LIST.OD[0].array[3]=0;
  SG_PARAMS_LIST.OD[1].array[0]=0;
  SG_PARAMS_LIST.OD[1].array[1]=0;
  SG_PARAMS_LIST.OD[1].array[2]=0;
  SG_PARAMS_LIST.OD[1].array[3]=0;
  SG_PARAMS_LIST.OD[2].array[0]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[2].array[1]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[2].array[2]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[2].array[3]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[3].array[0]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[3].array[1]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[3].array[2]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[3].array[3]=~((unsigned int)(0));
}
```

```
void init_spins() {
  int i;
  for(i=0; i<VOLUME; i++) {
    Spins[i][0] = rand();
    Spins[i][1] = rand();
    Spins[i][2] = rand();
    Spins[i][3] = rand();
  }
}
void init_couplings(){
  int i;
  for(i=0; i<VOLUME; i++) {
    Jx[i][0] = 1; rand();
    Jx[i][1] = 1; rand();
    Jx[i][2] = 1; rand();
    Jx[i][3] = 1; rand();
    Jy[i][0] = 1; rand();
    Jy[i][1] = 1; rand();
    Jy[i][2] = 1; rand();
    Jy[i][3] = 1; rand();
    Jz[i][0] = 1; rand();
    Jz[i][1] = 1; rand();
    Jz[i][2] = 1; rand();
    Jz[i][3] = 1; rand();
  }
}
void init_random() {
  int i;
  srand(SEED);
  ip = rand();
  irr = rand();
  for (i=0; i<256; i++) {
    // All systems have the same random number
    ira[i][0] = ira[i][1] = ira[i][2] = ira[i][3] = rand();
    // 4 random numbers for 4 systems with 32 replicas
    /*ira[i][1] = */ rand();
    /*ira[i][2] = */ rand();
    /*ira[i][3] = */ rand();
  }
  ip1=(ip+232)%256; ip2=(ip+201)%256; ip3=(ip+195)%256;
}
//-------------------------------------------------------------------------------------------
```

C.2 Lato SPE

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <spu_mfcio.h>
#include <spu_intrinsics.h>
#include <spu_internals.h>
#include <assert.h>
#include <simdmath/log2f4.h>
#include "../common.h"
#include "../user_common.h"
#include "../spe_lib.h"
//-------------------------------------------------------------------------------------------
// Shared data
spe_params SPE_PARAMETERS                        __attribute__ ((aligned (128)));
spinglasses_params SG_PARAMETERS                 __attribute__ ((aligned (128)));
vector unsigned int Spins[VOLUME]                __attribute__ ((aligned (128)));
vector unsigned int Jx[VOLUME]                   __attribute__ ((aligned (128)));
vector unsigned int Jy[VOLUME]                   __attribute__ ((aligned (128)));
vector unsigned int Jz[VOLUME]                   __attribute__ ((aligned (128)));
vector unsigned int ira[SPE_RANDOM_VECTOR_SIZE]  __attribute__ ((aligned (128)));
//-------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------
// Local data
const vector unsigned int ones        = { (signed int)1, (signed int)1,
                                           (signed int)1, (signed int)1 };
const vector unsigned int mask        = { 0xff, 0xff, 0xff, 0xff };
const vector float THREES      = { 3.0, 3.0, 3.0, 3.0 };
const vector unsigned int coords_mask = { SIDE-1, (SIDE-1)<<EXP_SIDE,
                                           (SIDE-1)<<EXP_SIDE_2, 0 };
const vector unsigned int coords_idnc = { 1, 1<<EXP_SIDE, 1<<EXP_SIDE_2, 0 };
// The alignement causes a minor numbers of rotations when used in association with a vector
unsigned int inSpin __attribute__ ((aligned (4)));
unsigned int x, y, z, go = 1;
vector unsigned int p_ips = {0, 0, 0, 0};
```

```
// Tick times
unsigned int t_start = 0;   // Start ticks
unsigned int t_end   = 0;   // End ticks
unsigned int t_avg   = 0;
vector float C_inv_max_rand;
vector float negative_T_4;
vector float LN_2 = { (float)0.69314718055995, (float)0.69314718055995,
                      (float)0.69314718055995, (float)0.69314718055995 };
unsigned long long int addr;
//-------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------
run_status RUN_STATUS __attribute__ ((aligned (128)));
run_ticks RUN_TICKS   __attribute__ ((aligned (128)));
//-------------------------------------------------------------------------------------
void sendMailToPPE() {
  do { } while(spu_stat_out_mbox () == 0);
  spu_write_out_mbox(1);
}
int waitMailFromPPE() {
  while(spu_stat_in_mbox() == 0) {};
  return spu_read_in_mbox ();
}
void loadConfiguration() {
  register int offset;
  unsigned int i;
  //-----------------------------------------------------------------------------------
  // Get param struct from gived address
  mfc_get( (void *) &SPE_PARAMETERS, addr, sizeof (spe_params), 1, 0, 0 );
  mfc_write_tag_mask ( 1 << 1 );
  mfc_read_tag_status_all ();
  offset = 0;
  for(i=0; i<VOLUME; i++) {
    mfc_get( (void *) &Spins + offset, SPE_PARAMETERS.Spins_addr + offset, 16, 1, 0, 0 );
    mfc_get( (void *) &Jx    + offset, SPE_PARAMETERS.Jx_addr    + offset, 16, 1, 0, 0 );
    mfc_get( (void *) &Jy    + offset, SPE_PARAMETERS.Jy_addr    + offset, 16, 1, 0, 0 );
    mfc_get( (void *) &Jz    + offset, SPE_PARAMETERS.Jz_addr    + offset, 16, 1, 0, 0 );
    offset += 16;
  }
  mfc_get( (void *) &ira, SPE_PARAMETERS.ira_addr, sizeof (ira), 1, 0, 0 );
  mfc_get( (void *) &SG_PARAMETERS, SPE_PARAMETERS.sg_param_addr, sizeof (SG_PARAMETERS),
                                                                                1, 0, 0 );
  mfc_write_tag_mask ( 1 << 1 );
  mfc_read_tag_status_all ();
  C_inv_max_rand = spu_splats((float)SG_PARAMETERS.C_inv_max_rand);
  negative_T_4   = spu_splats((float)-SG_PARAMETERS.T_4);
  p_ips = spu_insert((signed int)SG_PARAMETERS.ip,  p_ips, 0);
  p_ips = spu_insert((signed int)SG_PARAMETERS.ip1, p_ips, 1);
  p_ips = spu_insert((signed int)SG_PARAMETERS.ip2, p_ips, 2);
  p_ips = spu_insert((signed int)SG_PARAMETERS.ip3, p_ips, 3);
}
void saveConfiguration() {
  register int offset;
  unsigned int i;
  offset = 0;
  for(i=0; i<VOLUME; i++) {
    mfc_put( (void *) &Spins + offset, SPE_PARAMETERS.Spins_addr + offset, 16, 1, 0, 0 );
    offset += 16;
  }
  mfc_write_tag_mask ( 1 << 1 );
  mfc_read_tag_status_all ();
}
//-------------------------------------------------------------------------------------
int user_main ( ) {
  unsigned int i;
  vector float IDXf;
  vector unsigned int IDX,
                      select,
                      Carry1,
                      Carry2,
                      EU,
                      ED,
                      EC,
                      prod;
  vector unsigned int xyz,
                      xyzp  = {0, 0, 0, 0},
                      xyzm  = {0, 0, 0, 0};
  unsigned int haddr, laddr;
  //-----------------------------------------------------------------------------------
  // Load SPE parameters; waiting for param struct address
  while ( spu_stat_in_mbox () == 0 ) {} haddr = spu_read_in_mbox ();
  while ( spu_stat_in_mbox () == 0 ) {} laddr = spu_read_in_mbox ();
  addr = (((unsigned long long int)haddr)<<32) | laddr;
```

```
//--------------------------------------------------------------------------------------
// Metropolis Update
loadConfiguration();
do {
  // Counter reset
  spu_writech (SPU_WrDec, 0x7fffffff);
  t_start = spu_readch (SPU_RdDec);
  for(i=0; i<SG_PARAMETERS.iter; i++) {
    // Loop through vectors
    for(inSpin=0; inSpin < VOLUME; inSpin++) {
      xyz = spu_splats(inSpin);
      ////////////////////////////////////////////////////////////////////////////////
      // RANDOM
      //
      IDX = spu_xor(ira[spu_extract(p_ips, 0)] = spu_add(ira[spu_extract(p_ips, 1)],
                      ira[spu_extract(p_ips, 2)]), ira[spu_extract(p_ips, 3)]);
      // Incremento e maschero a 0xff, cosi' da forzare a valori compresi tra 0 e 244.
      p_ips = spu_add(p_ips, ones);
      p_ips = spu_and(p_ips, mask);
      ////////////////////////////////////////////////////////////////////////////////
      // IDXf = N_RANDOM;
      //
      IDXf = spu_mul(spu_convtf(IDX, 0), C_inv_max_rand);
      ////////////////////////////////////////////////////////////////////////////////
      // Calcolo del logaritmo
      //
      IDXf = spu_mul(_log2f4(IDXf), LN_2);
      ////////////////////////////////////////////////////////////////////////////////
      // Parte finale di N_RANDOM
      //
      IDXf = spu_mul(IDXf, negative_T_4);
      ////////////////////////////////////////////////////////////////////////////////
      // Blocco dell'if()
      //
      select = spu_cmpabsgt(IDXf, THREES);
      IDXf   = spu_sel(IDXf, THREES, select);
      IDX    = spu_convtu(IDXf, 0);
      ////////////////////////////////////////////////////////////////////////////////
      // Calcolo le coordinate dei vicini
      //
      // Coordinate positive                    // Coordinate negative
      xyzp = spu_add(xyz, coords_idnc);         xyzm = spu_sub(xyz, coords_idnc);
      xyzp = spu_sel(xyz, xyzp, coords_mask);   xyzm = spu_sel(xyz, xyzm, coords_mask);
      ////////////////////////////////////////////////////////////////////////////////
      // Calcolo il valore dello spin
      //
      EU     = spu_xor(Spins[inSpin], spu_xor(Jx[inSpin], Spins[spu_extract(xyzp, 0)]) );
      prod   = spu_xor(Spins[inSpin], spu_xor(Jx[spu_extract(xyzm, 0)],
                                   Spins[spu_extract(xyzm, 0)]) );
      Carry1 = spu_and(prod,     EU);
      EU     = spu_xor(prod,     EU);
      ED     = Carry1;
      prod   = spu_xor(Spins[inSpin], spu_xor(Jy[inSpin], Spins[spu_extract(xyzp, 1)]) );
      Carry1 = spu_and(prod,     EU);
      EU     = spu_xor(prod,     EU);
      // Carry2 = spu_and(Carry1,   ED);
      ED     = spu_xor(Carry1,   ED);
      // EC     = Carry2;
      prod   = spu_xor(Spins[inSpin], spu_xor(Jy[spu_extract(xyzm, 1)],
                                   Spins[spu_extract(xyzm, 1)]) );
      Carry1 = spu_and(prod,     EU);
      EU     = spu_xor(prod,     EU);
      Carry2 = spu_and(Carry1,   ED);
      ED     = spu_xor(Carry1,   ED);
      EC     = spu_xor(Carry2,   EC);
      prod   = spu_xor(Spins[inSpin], spu_xor(Jz[inSpin], Spins[spu_extract(xyzp, 2)]) );
      Carry1 = spu_and(prod,     EU);
      EU     = spu_xor(prod,     EU);
      Carry2 = spu_and(Carry1,   ED);
      ED     = spu_xor(Carry1,   ED);
      EC     = spu_xor(Carry2,   EC);
      prod   = spu_xor(Spins[inSpin], spu_xor(Jz[spu_extract(xyzm, 2)],
                                   Spins[spu_extract(xyzm, 2)]) );
      Carry1 = spu_and(prod,     EU);
      EU     = spu_xor(prod,     EU);
      Carry2 = spu_and(Carry1,   ED);
      ED     = spu_xor(Carry1,   ED);
      EC     = spu_xor(Carry2,   EC);
      EC = spu_xor(EC, 0xffffffff); // EC=~EC; ED=~ED; EU=~EU;
                               /* negation is convenient: see above*/
```

```
        ED = spu_xor(ED, 0xffffffff);
        EU = spu_xor(EU, 0xffffffff);
        /*OD[IDX],OU[IDX] represent IDX positionally*/
        Carry1 = spu_and(SG_PARAMETERS.OU[spu_extract(IDX, 0)].vec, EU);
        Carry2 = spu_or(spu_and(SG_PARAMETERS.OD[spu_extract(IDX, 0)].vec, ED),
                spu_and(spu_or(SG_PARAMETERS.OD[spu_extract(IDX, 0)].vec, ED), Carry1));
        /*Carry2=1 if and only if (T/4)logR+(~ED,~EU) >= 4*/
        Spins[inSpin] = spu_xor(Spins[inSpin], spu_or(Carry2, EC));
////////////////////////////////////////////////////////////////////////////////////////////
        inSpin++;
        xyz = spu_splats(inSpin);
        ////////////////////////////////////////////////////////////////////////////////////
        // RANDOM
        //
        IDX = spu_xor(ira[spu_extract(p_ips, 0)] = spu_add(ira[spu_extract(p_ips, 1)],
                      ira[spu_extract(p_ips, 2)]), ira[spu_extract(p_ips, 3)]);
        // Incremento e maschero a 0xff, cosi' da forzare a valori compresi tra 0 e 244.
        p_ips = spu_add(p_ips, ones);
        p_ips = spu_and(p_ips, mask);
        ////////////////////////////////////////////////////////////////////////////////////
        // IDXf = N_RANDOM;
        //
        IDXf = spu_mul(spu_convtf(IDX, 0), C_inv_max_rand);
        ////////////////////////////////////////////////////////////////////////////////////
        // Calcolo del logaritmo
        //
        IDXf = spu_mul(_log2f4(IDXf), LN_2);
        ////////////////////////////////////////////////////////////////////////////////////
        // Parte finale di N_RANDOM
        //
        IDXf = spu_mul(IDXf, negative_T_4);
        ////////////////////////////////////////////////////////////////////////////////////
        // Blocco dell'if()
        //
        select = spu_cmpabsgt(IDXf, THREES);
        IDXf   = spu_sel(IDXf, THREES, select);
        IDX    = spu_convtu(IDXf, 0);
        ////////////////////////////////////////////////////////////////////////////////////
        // Calcolo le coordinate dei vicini
        //
        // Coordinate positive                    // Coordinate negative
        xyzp = spu_add(xyz, coords_idnc);         xyzm = spu_sub(xyz, coords_idnc);
        xyzp = spu_sel(xyz, xyzp, coords_mask);   xyzm = spu_sel(xyz, xyzm, coords_mask);
        ////////////////////////////////////////////////////////////////////////////////////
        // Calcolo il valore dello spin
        //
        EU     = spu_xor(Spins[inSpin], spu_xor(Jx[inSpin], Spins[spu_extract(xyzp, 0)]) );
        prod   = spu_xor(Spins[inSpin], spu_xor(Jx[spu_extract(xyzm, 0)],
                                        Spins[spu_extract(xyzm, 0)]) );
        Carry1 = spu_and(prod,     EU);
        EU     = spu_xor(prod,     EU);
        ED     = Carry1;
        prod   = spu_xor(Spins[inSpin], spu_xor(Jy[inSpin], Spins[spu_extract(xyzp, 1)]) );
        Carry1 = spu_and(prod,     EU);
        EU     = spu_xor(prod,     EU);
        Carry2 = spu_and(Carry1,   ED);
        ED     = spu_xor(Carry1,   ED);
        EC     = Carry2;
        prod   = spu_xor(Spins[inSpin], spu_xor(Jy[spu_extract(xyzm, 1)],
                                        Spins[spu_extract(xyzm, 1)]) );
        Carry1 = spu_and(prod,     EU);
        EU     = spu_xor(prod,     EU);
        Carry2 = spu_and(Carry1,   ED);
        ED     = spu_xor(Carry1,   ED);
        EC     = spu_xor(Carry2,   EC);
        prod   = spu_xor(Spins[inSpin], spu_xor(Jz[inSpin], Spins[spu_extract(xyzp, 2)]) );
        Carry1 = spu_and(prod,     EU);
        EU     = spu_xor(prod,     EU);
        Carry2 = spu_and(Carry1,   ED);
        ED     = spu_xor(Carry1,   ED);
        EC     = spu_xor(Carry2,   EC);
        prod   = spu_xor(Spins[inSpin], spu_xor(Jz[spu_extract(xyzm, 2)],
                                        Spins[spu_extract(xyzm, 2)]) );
        Carry1 = spu_and(prod,     EU);
        EU     = spu_xor(prod,     EU);
        Carry2 = spu_and(Carry1,   ED);
        ED     = spu_xor(Carry1,   ED);
        EC     = spu_xor(Carry2,   EC);
        EC = spu_xor(EC, 0xffffffff); // EC=~EC; ED=~ED; EU=~EU;
                                      /* negation is convenient: see above*/
```

```
        ED = spu_xor(ED, 0xffffffff);
        EU = spu_xor(EU, 0xffffffff);
        /*OD[IDX],OU[IDX] represent IDX positionally*/
        Carry1 = spu_and(SG_PARAMETERS.OU[spu_extract(IDX, 0)].vec, EU);
        Carry2 = spu_or(spu_and(SG_PARAMETERS.OD[spu_extract(IDX, 0)].vec, ED),
                    spu_and(spu_or(SG_PARAMETERS.OD[spu_extract(IDX, 0)].vec, ED), Carry1));
        /*Carry2=1 if and only if (T/4)logR+(~ED,~EU) >= 4*/
        Spins[inSpin] = spu_xor(Spins[inSpin], spu_or(Carry2, EC));
////////////////////////////////////////////////////////////////////////////////////////////
        inSpin++;
        xyz = spu_splats(inSpin);
        ////////////////////////////////////////////////////////////////////////////////////
        // RANDOM
        //
        IDX = spu_xor(ira[spu_extract(p_ips, 0)] = spu_add(ira[spu_extract(p_ips, 1)],
                              ira[spu_extract(p_ips, 2)]), ira[spu_extract(p_ips, 3)]);
        // Incremento e maschero a 0xff, cosi' da forzare a valori compresi tra 0 e 244.
        p_ips = spu_add(p_ips, ones);
        p_ips = spu_and(p_ips, mask);
        ////////////////////////////////////////////////////////////////////////////////////
        // IDXf = N_RANDOM;
        //
        IDXf = spu_mul(spu_convtf(IDX, 0), C_inv_max_rand);
        ////////////////////////////////////////////////////////////////////////////////////
        // Calcolo del logaritmo
        //
        IDXf = spu_mul(_log2f4(IDXf), LN_2);
        ////////////////////////////////////////////////////////////////////////////////////
        // Parte finale di N_RANDOM
        //
        IDXf = spu_mul(IDXf, negative_T_4);
        ////////////////////////////////////////////////////////////////////////////////////
        // Blocco dell'if()
        //
        select = spu_cmpabsgt(IDXf, THREES);
        IDXf   = spu_sel(IDXf, THREES, select);
        IDX    = spu_convtu(IDXf, 0);
        ////////////////////////////////////////////////////////////////////////////////////
        // Calcolo le coordinate dei vicini
        //
        // Coordinate positive                    // Coordinate negative
        xyzp = spu_add(xyz, coords_idnc);         xyzm = spu_sub(xyz, coords_idnc);
        xyzp = spu_sel(xyz, xyzp, coords_mask);   xyzm = spu_sel(xyz, xyzm, coords_mask);
        ////////////////////////////////////////////////////////////////////////////////////
        // Calcolo il valore dello spin
        //
        EU     = spu_xor(Spins[inSpin], spu_xor(Jx[inSpin], Spins[spu_extract(xyzp, 0)]) );
        prod   = spu_xor(Spins[inSpin], spu_xor(Jx[spu_extract(xyzm, 0)],
                                   Spins[spu_extract(xyzm, 0)]) );
        Carry1 = spu_and(prod,     EU);
        EU     = spu_xor(prod,     EU);
        ED     = Carry1;
        prod   = spu_xor(Spins[inSpin], spu_xor(Jy[inSpin], Spins[spu_extract(xyzp, 1)]) );
        Carry1 = spu_and(prod,     EU);
        EU     = spu_xor(prod,     EU);
        Carry2 = spu_and(Carry1,   ED);
        ED     = spu_xor(Carry1,   ED);
        EC     = Carry2;
        prod   = spu_xor(Spins[inSpin], spu_xor(Jy[spu_extract(xyzm, 1)],
                                   Spins[spu_extract(xyzm, 1)]) );
        Carry1 = spu_and(prod,     EU);
        EU     = spu_xor(prod,     EU);
        Carry2 = spu_and(Carry1,   ED);
        ED     = spu_xor(Carry1,   ED);
        EC     = spu_xor(Carry2,   EC);
        prod   = spu_xor(Spins[inSpin], spu_xor(Jz[inSpin], Spins[spu_extract(xyzp, 2)]) );
        Carry1 = spu_and(prod,     EU);
        EU     = spu_xor(prod,     EU);
        Carry2 = spu_and(Carry1,   ED);
        ED     = spu_xor(Carry1,   ED);
        EC     = spu_xor(Carry2,   EC);
        prod   = spu_xor(Spins[inSpin], spu_xor(Jz[spu_extract(xyzm, 2)],
                                   Spins[spu_extract(xyzm, 2)]) );
        Carry1 = spu_and(prod,     EU);
        EU     = spu_xor(prod,     EU);
        Carry2 = spu_and(Carry1,   ED);
        ED     = spu_xor(Carry1,   ED);
        EC     = spu_xor(Carry2,   EC);
        EC = spu_xor(EC, 0xffffffff); // EC=~EC; ED=~ED; EU=~EU;
                              /* negation is convenient: see above*/
```

```
        ED = spu_xor(ED, 0xffffffff);
        EU = spu_xor(EU, 0xffffffff);
        /*OD[IDX],OU[IDX] represent IDX positionally*/
        Carry1 = spu_and(SG_PARAMETERS.OU[spu_extract(IDX, 0)].vec, EU);
        Carry2 = spu_or(spu_and(SG_PARAMETERS.OD[spu_extract(IDX, 0)].vec, ED),
                spu_and(spu_or(SG_PARAMETERS.OD[spu_extract(IDX, 0)].vec, ED), Carry1));
        /*Carry2=1 if and only if (T/4)logR+(~ED,~EU) >= 4*/
        Spins[inSpin] = spu_xor(Spins[inSpin], spu_or(Carry2, EC));
//////////////////////////////////////////////////////////////////////////////////////////
        inSpin++;
        xyz = spu_splats(inSpin);
        ////////////////////////////////////////////////////////////////////////////////
        // RANDOM
        //
        IDX = spu_xor(ira[spu_extract(p_ips, 0)] = spu_add(ira[spu_extract(p_ips, 1)],
                        ira[spu_extract(p_ips, 2)]), ira[spu_extract(p_ips, 3)]);
        // Incremento e maschero a 0xff, cosi' da forzare a valori compresi tra 0 e 244.
        p_ips = spu_add(p_ips, ones);
        p_ips = spu_and(p_ips, mask);
        ////////////////////////////////////////////////////////////////////////////////
        // IDXf = N_RANDOM;
        //
        IDXf = spu_mul(spu_convtf(IDX, 0), C_inv_max_rand);
        ////////////////////////////////////////////////////////////////////////////////
        // Calcolo del logaritmo
        //
        IDXf = spu_mul(_log2f4(IDXf), LN_2);
        ////////////////////////////////////////////////////////////////////////////////
        // Parte finale di N_RANDOM
        //
        IDXf = spu_mul(IDXf, negative_T_4);
        ////////////////////////////////////////////////////////////////////////////////
        // Blocco dell'if()
        //
        select = spu_cmpabsgt(IDXf, THREES);
        IDXf   = spu_sel(IDXf, THREES, select);
        IDX    = spu_convtu(IDXf, 0);
        ////////////////////////////////////////////////////////////////////////////////
        // Calcolo le coordinate dei vicini
        //
        // Coordinate positive                    // Coordinate negative
        xyzp = spu_add(xyz, coords_idnc);         xyzm = spu_sub(xyz, coords_idnc);
        xyzp = spu_sel(xyz, xyzp, coords_mask);   xyzm = spu_sel(xyz, xyzm, coords_mask);
        ////////////////////////////////////////////////////////////////////////////////
        // Calcolo il valore dello spin
        //
        EU     = spu_xor(Spins[inSpin], spu_xor(Jx[inSpin], Spins[spu_extract(xyzp, 0)]) );
        prod   = spu_xor(Spins[inSpin], spu_xor(Jx[spu_extract(xyzm, 0)],
                                   Spins[spu_extract(xyzm, 0)]) );
        Carry1 = spu_and(prod,     EU);
        EU     = spu_xor(prod,     EU);
        ED     = Carry1;
        prod   = spu_xor(Spins[inSpin], spu_xor(Jy[inSpin], Spins[spu_extract(xyzp, 1)]) );
        Carry1 = spu_and(prod,     EU);
        EU     = spu_xor(prod,     EU);
        Carry2 = spu_and(Carry1,   ED);
        ED     = spu_xor(Carry1,   ED);
        EC     = Carry2;
        prod   = spu_xor(Spins[inSpin], spu_xor(Jy[spu_extract(xyzm, 1)],
                                   Spins[spu_extract(xyzm, 1)]) );
        Carry1 = spu_and(prod,     EU);
        EU     = spu_xor(prod,     EU);
        Carry2 = spu_and(Carry1,   ED);
        ED     = spu_xor(Carry1,   ED);
        EC     = spu_xor(Carry2,   EC);
        prod   = spu_xor(Spins[inSpin], spu_xor(Jz[inSpin], Spins[spu_extract(xyzp, 2)]) );
        Carry1 = spu_and(prod,     EU);
        EU     = spu_xor(prod,     EU);
        Carry2 = spu_and(Carry1,   ED);
        ED     = spu_xor(Carry1,   ED);
        EC     = spu_xor(Carry2,   EC);
        prod   = spu_xor(Spins[inSpin], spu_xor(Jz[spu_extract(xyzm, 2)],
                                   Spins[spu_extract(xyzm, 2)]) );
        Carry1 = spu_and(prod,     EU);
        EU     = spu_xor(prod,     EU);
        Carry2 = spu_and(Carry1,   ED);
        ED     = spu_xor(Carry1,   ED);
        EC     = spu_xor(Carry2,   EC);
        EC = spu_xor(EC, 0xffffffff); // EC=~EC; ED=~ED; EU=~EU;
                              /* negation is convenient: see above*/
```

```
        ED = spu_xor(ED, 0xffffffff);
        EU = spu_xor(EU, 0xffffffff);
        /*OD[IDX],OU[IDX] represent IDX positionally*/
        Carry1 = spu_and(SG_PARAMETERS.OU[spu_extract(IDX, 0)].vec, EU);
        Carry2 = spu_or(spu_and(SG_PARAMETERS.OD[spu_extract(IDX, 0)].vec, ED),
                    spu_and(spu_or(SG_PARAMETERS.OD[spu_extract(IDX, 0)].vec, ED), Carry1));
        /*Carry2=1 if and only if (T/4)logR+(~ED,~EU) >= 4*/
        Spins[inSpin] = spu_xor(Spins[inSpin], spu_or(Carry2, EC));
      }
    }
    t_end = spu_readch (SPU_RdDec);
    if(t_avg == 0) t_avg = (t_start - t_end);
    else t_avg = (t_avg + (t_start - t_end)) / 2;
    saveConfiguration();
    sendMailToPPE();
    go = waitMailFromPPE();
  } while(go);
  RUN_TICKS.tick_cnt[0] = t_avg;
  //------------------------------------------------------------------------------------
  //------------------------------------------------------------------------------------
  // Put ticks
  mfc_put ( (void *) &RUN_TICKS, SPE_PARAMETERS.run_ticks_addr, sizeof (run_ticks),
                                                                   1, 0, 0 );
  mfc_write_tag_mask ( 1 << 1 );
  mfc_read_tag_status_all ();
  //------------------------------------------------------------------------------------
  return (0);
}
//--------------------------------------------------------------------------------------
//--------------------------------------------------------------------------------------
int main (/* unsigned long long spu_id, unsigned long long argv */) {
  user_main ( );
  return (0);
}
//--------------------------------------------------------------------------------------
```

C.3 File di Libreria

```
//------------------------------------------------------------------------------------------//
//                                                                                          //
// Define here all data structures and constant that have to be shared between PPE and SPEs.    //
//                                                                                          //
//------------------------------------------------------------------------------------------//
#ifndef __USER_COMMON__
#define __USER_COMMON__
//--------------------------------------------------------------------------------------
// cab params
#define CAB_TIMEBASE        (14318000)
#define CAB_HOSTNAME        "cab"
// cc03.juice params
#define CC09_JUICE_TIMEBASE (14318000)
#define CC09_JUICE_HOSTNAME "cc09.juice"
// ps3.fe.infn.it params
#define PS3_TIMEBASE        (79800000)                         // PS3 Timebase
#define PS3_HZ              (3.2e9)                            // PS3 Frequency HZ
#define PS3_TICK_PERIOD     (1.0 / PS3_TIMEBASE)               // Tick period in seconds
#define PS3_CLOCK_PERIOD    (1.0 / PS3_HZ)                     // PS3 clock period sec
#define PS3_TICK_N_CLK      (PS3_TICK_PERIOD / PS3_CLOCK_PERIOD) // Number of clocks in a tick
#define PS3_HOSTNAME        "ps3.fe.infn.it"
#define TICK_PERIOD         (1.0 / PS3_TIMEBASE)
#define HOSTNAME            PS3_HOSTNAME
//--------------------------------------------------------------------------------------
// Parameters of SPE
#define ADDR unsigned long long int
typedef union _array_to_vector {
unsigned int        array[4] __attribute__ ((aligned(16)));
#ifndef PPE_SIDE
vector unsigned int vec;
#endif
} array_to_vector;
typedef struct {
  unsigned char pad[4];
  array_to_vector OD[4];
  array_to_vector OU[4];
  unsigned char ip,
              ip1,
              ip2,
```

```
                ip3,
                irr;
  float C_inv_max_rand;
  double T_4;
  unsigned int iter;
} spinglasses_params;
typedef struct {
  unsigned char pad[4];          // Padding
  unsigned int idn;              // SPE id            ( 4B)
  ADDR Spins_addr;               // Spins address     ( 8B)
  ADDR Jx_addr;                  // Jx address        ( 8B)
  ADDR Jy_addr;                  // Jy address        ( 8B)
  ADDR Jz_addr;                  // Jz address        ( 8B)
  ADDR ira_addr;                 // Random address    ( 8B)
  ADDR run_ticks_addr;           // Ticks struct addr ( 8B)
  ADDR sg_param_addr;            // Sg    struct addr ( 8B)
                                 // - Total --------- (??B)
} spe_params;
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
// It VERY important that sizeof(run_status)<=16, because it must be
// transferred with an ATOMIC dma operation
typedef struct {
  unsigned int status;
} run_status;
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
typedef struct {
  unsigned char pad[10];          // Padding
  unsigned int tick_cnt[1];
} run_ticks;
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
// Metropolis
#define SIDE       (8)
#define FACE       (SIDE * SIDE)
#define VOLUME     (FACE * SIDE)
// Size of the random table
#define SPE_RANDOM_VECTOR_SIZE (256)
// Seed of the random numbers
#define SEED (unsigned int)1
// Number of replicas (size of a vector unsigned int in bits)
#define N_REPLICAS (128)
#define ISAPOWEROF2(_x) (!((SIDE - 1) & SIDE))
#define EXP_SIDE        (3)
#define EXP_SIDE_2      (EXP_SIDE * 2)
//-----------------------------------------------------------------------------------------
#endif
```

Appendix D

Implementazione per Cell, versione sgv12_lparam

D.1 Lato PPE

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <unistd.h>
#include <malloc.h>
#include "ppe_lib.h"
#include "common.h"
#include "user_common.h"
#define SPE_THREADS     1
//-----------------------------------------------------------------------------------------
// List of SPE-clients
extern spe_program_handle_t spe_program;
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
// PPE-SPE Sharing
unsigned int * Spins       __attribute__ ((aligned (128)));
unsigned int * Jx          __attribute__ ((aligned (128))),
* Jy          __attribute__ ((aligned (128))),
* Jz          __attribute__ ((aligned (128)));
unsigned int ira[256][4]  __attribute__ ((aligned (128)));
// Magnetizations
double magnetization  [N_REPLICAS];
double avg_magnet     [N_REPLICAS];
double data           [N_REPLICAS];
double data_2         [N_REPLICAS];
double sigma          [N_REPLICAS];
unsigned char ip,
              ip1,
              ip2,
              ip3,
              irr;
const float C_inv_max_rand = 1.0/((double)(~((unsigned/* long int*/)(0))));
double T_4;
FILE * fp_data_out; /** ... */
unsigned int zero = 0;
unsigned int one  = 1;
unsigned int result;
unsigned int PPE_ITERATIONS = 1000;
unsigned int SPE_ITERATIONS = 1000;
double BETA_START     = 0.01;
double BETA_STOP      = 0.5;
double BETA_SPLIT     = 100.0;
double BETA_INC;
int SIDE = 15, FACE, VOLUME;
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
myspe_t THREADS;
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
// PPE -> SPE ( MM -> LS )
```

```
spe_params SPE_PARAMS_LIST __attribute__ ((aligned (128)));
spinglasses_params SG_PARAMS_LIST  __attribute__ ((aligned (128)));
// SPE -> PP ( LS -> MM )
volatile run_status RUN_STATUS_LIST __attribute__ ((aligned (128)));
volatile run_ticks RUN_TICKS_LIST  __attribute__ ((aligned (128)));
//-----------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------
// Functions declaration
void loadConfigurationFromFile();
void set_defaults();
void init_random();
void init_spins();
void init_couplings();
//-----------------------------------------------------------------------------------------
void startSPE(spe_program_handle_t * program_pointer) {
  START_SPE ( THREADS,program_pointer);
}
//-----------------------------------------------------------------------------------------
float beta;
void sendDataMailToSPE() {
  unsigned int out_mail = 0;
  PR_DEBUG ("PPE: sending SPE parameters MM address\n");
  // High address (mfc_ea2h(...))
  out_mail = (unsigned int) ((unsigned long long int)(&(SPE_PARAMS_LIST  ))>>32)&0xffffffff;
  PPE_SM ( THREADS,&out_mail);               // Send SPE parameters address
  // Low address (mfc_ea2l(...))
  out_mail = (unsigned int) ((unsigned long long int)(&(SPE_PARAMS_LIST  )))&0xffffffff;
  PPE_SM ( THREADS,&out_mail);               // Send SPE parameters address
}
#define waitMailFromSPE()      PPE_RM(THREADS, &result);
#define sendGoMail()           PPE_SM(THREADS, &one)
#define sendDontGoMail()       PPE_SM(THREADS, &zero)
#define printBetaAndSpinsSum()     fprintf(fp_data_out, "%f", beta);          \
  for(j=0; j<N_REPLICAS; j++) fprintf(fp_data_out, " %lf", magnetization[j]); \
  fprintf(fp_data_out, "\n");
#define printBetaAvgMagnAndSigma() fprintf(fp_data_out, "%f", beta);                      \
  for(j=0; j<N_REPLICAS; j++) fprintf(fp_data_out, " %lf %lf", avg_magnet[j], sigma[j]); \
  fprintf(fp_data_out, "\n");
#define stopSPE()              STOP_SPE(THREADS)
void resetMagnetizationsVars() {
  int k;
  // Reset of magnetizations vars
  for(k=0; k<N_REPLICAS; k++) {
    magnetization[k] = avg_magnet[k] = data[k] = data_2[k] = sigma[k] = 0;
  }
}
void calculateMagnetization() {
  int i, j, k;
  for(i=0; i<(4 * VOLUME); i+=4) {
    for(k=0; k<4; k++) {
      for(j=0; j<32; j++) {
        magnetization[(k*sizeof(unsigned int)*4)+j] +=
              ( ((Spins[i + k] >> j) & 0x1) ) ? 1 : -1;
      }
    }
  }
  for(i=0; i<N_REPLICAS; i++) {
    // Normalization of the magnetizations
    magnetization[i] /= (double)VOLUME;
    // Save the values for the future
    data[i]   +=  pow(magnetization[i], 2.0);
    data_2[i] += fabs(magnetization[i]);
    magnetization[i] = 0;
  }
}
void endMagnetizationCalculus() {
  int k;
  for(k=0; k<N_REPLICAS; k++) {
    avg_magnet[k]  = (data_2[k] / (double)PPE_ITERATIONS);
    // Media dei quadrati dei valori assoluti delle magnetizzazioni
    data[k]        = (data[k]   / (double)PPE_ITERATIONS);
    // Media dei valori assoluti delle magnetizzazioni elevata al quadrato
    data_2[k]      = pow( (data_2[k] / (double)PPE_ITERATIONS), 2.0);
    sigma[k]       = sqrt( data[k] - data_2[k] );
  }
}
void user_main (spe_program_handle_t * program_pointer) {
  unsigned i, j;
  for(beta = BETA_START; beta < BETA_STOP; beta += BETA_INC) {
    loadConfigurationFromFile();
    resetMagnetizationsVars();
```

```
    startSPE(program_pointer);
    sendDataMailToSPE();
    for(i = 0; i < PPE_ITERATIONS; i++) {
      waitMailFromSPE();
      calculateMagnetization();
      sendGoMail();
    }
    endMagnetizationCalculus();
    printBetaAvgMagnAndSigma()
    sendDontGoMail();
    stopSPE();
  }
}
//---------------------------------------------------------------------------------------
//---------------------------------------------------------------------------------------
int user_end () {
  // Close files
  fclose(fp_data_out);
  // Display tick times
  double metropolis_time;
  metropolis_time  = (((RUN_TICKS_LIST.tick_cnt[0]*(double)1.0) * TICK_PERIOD) * exp10(12));
  // Print Statistics
  printf ("Metropolis ticks average: %f ticks\n", ((RUN_TICKS_LIST.tick_cnt[0] * 1.0) /
                                                   (PPE_ITERATIONS * SPE_ITERATIONS)));
  printf ("Metropolis time: %f ps\n", metropolis_time);
  printf ("Metropolis spin update time: %f ps/spin\n", (metropolis_time) /
                                   (SPE_ITERATIONS * VOLUME * N_REPLICAS));
  return 0;
}
//---------------------------------------------------------------------------------------
//---------------------------------------------------------------------------------------
void run ( spe_program_handle_t * program_pointer ) {
  //-------------------------------------------------------------------------------------
  user_main (program_pointer);
  //-------------------------------------------------------------------------------------
  //-------------------------------------------------------------------------------------
  user_end ();
  //-------------------------------------------------------------------------------------
}
//---------------------------------------------------------------------------------------
//---------------------------------------------------------------------------------------
static char * params = "ha:b:c:d:e:f:";
void usage() {
  printf("Usage of spinglasse_ppe:\n" \
         "\t-a <float>\tbeta start value  (default: %f)\n"   \
         "\t-b <float>\tbeta stop value   (default: %f)\n"   \
         "\t-c <int>  \tbeta split value  (default: %.0f)\n" \
         "\t-d <int>  \titerations on PPE (default: %d)\n"   \
         "\t-e <int>  \titerations on SPE (default: %d)\n"   \
         "\t-f <int>  \tside of cube      (default: %d)\n"   \
         "", BETA_START, BETA_STOP, BETA_SPLIT, PPE_ITERATIONS, SPE_ITERATIONS, SIDE);
}
int parseArguments(int argc, char * * argv) {
  int i, tmp;
  opterr = 0;  /* don't want writing to stderr */
  while ( (i = getopt(argc, argv, params)) != -1) {
    switch (i) {
      /*
       * Handling options
       */
    case 'h':       /* help option */
      usage();
      return -1;
      break;
    case 'a':        /* beta start value */
      BETA_START = atof(optarg);
      if(BETA_START <= 0.0) {
        fprintf(stderr, "beta start value must be greater than 0\n");
        return -1;
      }
      break;
    case 'b':        /* beta start value */
      BETA_STOP = atof(optarg);
      if(BETA_STOP <= 0.0) {
        fprintf(stderr, "beta stop value must be greater than 0\n");
        return -1;
      }
      if(BETA_STOP <= BETA_START) {
        fprintf(stderr, "beta stop value must be greater than beta start value\n");
        return -1;
      }
      break;
    case 'c':        /* beta split value */
```

```
      tmp = atoi(optarg);
      if(tmp <= 0) {
        fprintf(stderr, "beta split value must be greater than 0\n");
        return -1;
      }
      BETA_SPLIT = (double)tmp;
      break;
    case 'd':        /* PPE iterations */
      tmp = atoi(optarg);
      if(tmp <= 0) {
        fprintf(stderr, "PPE iterations must be greater than 0\n");
        return -1;
      }
      PPE_ITERATIONS = tmp;
      break;
    case 'e':        /* SPE iterations */
      tmp = atoi(optarg);
      if(tmp <= 0) {
        fprintf(stderr, "SPE iterations must be greater than 0\n");
        return -1;
      }
      SPE_ITERATIONS = tmp;
      break;
    case 'f':        /* side of cube */
      tmp = atoi(optarg);
      if((tmp <= 2) || (tmp > 15)) {
        fprintf(stderr, "Side of cube must be greater than 2 and less than 15\n");
        return -1;
      }
      SIDE = (unsigned int)tmp;
      break;
    case '?':
      printf("Unrecognized options -%c\n", optopt);
      usage();
      return -1;
    default:         /* should not reached */
      usage();
    }
  }
  return 0;
}
int main (int argc, char * * argv) {
  if(parseArguments(argc, argv)) {
    exit(1);
  }
  BETA_INC = ((BETA_STOP - BETA_START) / BETA_SPLIT);
  FACE   = (SIDE * SIDE);
  VOLUME = (FACE * SIDE);
  Spins = memalign(128, 4 * VOLUME * sizeof(unsigned int));
  Jx    = memalign(128, 4 * VOLUME * sizeof(unsigned int));
  Jy    = memalign(128, 4 * VOLUME * sizeof(unsigned int));
  Jz    = memalign(128, 4 * VOLUME * sizeof(unsigned int));
  if((!Spins) || (!Jx) || (!Jy) || (!Jz)) {
    printf("ERROR: calloc() failed\n");
    exit(1);
  }
  // Alignment and size check of ALL buffers involved in DMA transfers
  if((CHECK_DMA_128B_ALIGNMENT((unsigned long long int)Spins)) != 0) {
    printf("ERROR: Spins is not 128bytes aligned: %p\n", (void *)Spins);
    exit(1);
  }
  if((CHECK_DMA_128B_ALIGNMENT((unsigned long long int)Jx)) != 0) {
    printf("ERROR: Jx is not 128bytes aligned: %p\n", (void *)Jx);
    exit(1);
  }
  if((CHECK_DMA_128B_ALIGNMENT((unsigned long long int)Jy)) != 0) {
    printf("ERROR: Jy is not 128bytes aligned: %p\n", (void *)Jy);
    exit(1);
  }
  if((CHECK_DMA_128B_ALIGNMENT((unsigned long long int)Jz)) != 0) {
    printf("ERROR: Jz is not 128bytes aligned: %p\n", (void *)Jz);
    exit(1);
  }
  if((CHECK_DMA_128B_ALIGNMENT((unsigned int)ira)) != 0) {
    printf("ERROR: ira is not 128bytes aligned: %p\n", (void *)ira);
    exit(1);
  }
  if((CHECK_DMA_SIZE(sizeof(spe_params))) != 0) {
    printf ("ERROR: spe_params type size is not valid: %ld bytes\n\n", sizeof(spe_params));
    exit (1);
  }
  if((CHECK_DMA_SIZE(sizeof(spinglasses_params))) != 0) {
    printf ("ERROR: spe_params type size is not valid: %ld bytes\n\n", sizeof(spe_params));
```

```
    exit (1);
  }
  if((CHECK_DMA_SIZE(sizeof(run_ticks))) != 0) {
    printf ("\nERROR: run_ticks type size is not valid: %lu bytes\n\n", sizeof(run_ticks));
    exit (1);
  }
  //------------------------------------------------------------------------------------
  // Global variables initialization
  if(!(stdout = fopen("stdout_at_" HOSTNAME, "w"))) {
    printf ("\nERROR: can't create/write file stdout_at_" HOSTNAME "\n\n");
    exit (1);
  }
  if(!(fp_data_out = fopen("data.out_at_" HOSTNAME, "w"))) {
    printf ("\nERROR: can't create/write file data.out_at_" HOSTNAME "\n\n");
    exit (1);
  }
  //------------------------------------------------------------------------------------
  //------------------------------------------------------------------------------------
  run ( &spe_program );
  //------------------------------------------------------------------------------------
  exit (0);
}
//--------------------------------------------------------------------------------------
//--------------------------------------------------------------------------------------
// Spin Glasses
void loadConfigurationFromFile() {
  set_defaults();
  init_random();
  init_spins();
  init_couplings();
  //------------------------------------------------------------------------------------
  // Init the status struct
  RUN_STATUS_LIST.status=0;
  //------------------------------------------------------------------------------------
  //------------------------------------------------------------------------------------
  // Init the param struct
  SPE_PARAMS_LIST.Spins_addr     = (ADDR)Spins;
  SPE_PARAMS_LIST.Jx_addr        = (ADDR)Jx;
  SPE_PARAMS_LIST.Jy_addr        = (ADDR)Jy;
  SPE_PARAMS_LIST.Jz_addr        = (ADDR)Jz;
  SPE_PARAMS_LIST.ira_addr       = (ADDR)ira;
  SPE_PARAMS_LIST.run_ticks_addr = (ADDR)&(RUN_TICKS_LIST  );
  SPE_PARAMS_LIST.sg_param_addr  = (ADDR)&SG_PARAMS_LIST  ;
  SG_PARAMS_LIST.T_4             = (1 / (beta * 4));
  SG_PARAMS_LIST.C_inv_max_rand  = C_inv_max_rand;
  SG_PARAMS_LIST.ip              = ip;
  SG_PARAMS_LIST.ip1             = ip1;
  SG_PARAMS_LIST.ip2             = ip2;
  SG_PARAMS_LIST.ip3             = ip3;
  SG_PARAMS_LIST.irr             = irr;
  SG_PARAMS_LIST.iter            = SPE_ITERATIONS;
  SG_PARAMS_LIST.SIDE            = SIDE;
  SG_PARAMS_LIST.FACE            = (SIDE * SIDE);
  SG_PARAMS_LIST.VOLUME          = (SIDE * SIDE * SIDE);
  SG_PARAMS_LIST.Q               = (unsigned int)(VOLUME / 1024 );
  SG_PARAMS_LIST.R               = (unsigned int)(VOLUME % 1024 );
  //------------------------------------------------------------------------------------
}
void set_defaults() {
  // OU /////////////////////////////////////////////////
  SG_PARAMS_LIST.OU[0].array[0]=0;
  SG_PARAMS_LIST.OU[0].array[1]=0;
  SG_PARAMS_LIST.OU[0].array[2]=0;
  SG_PARAMS_LIST.OU[0].array[3]=0;
  SG_PARAMS_LIST.OU[1].array[0]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[1].array[1]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[1].array[2]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[1].array[3]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[2].array[0]=0;
  SG_PARAMS_LIST.OU[2].array[1]=0;
  SG_PARAMS_LIST.OU[2].array[2]=0;
  SG_PARAMS_LIST.OU[2].array[3]=0;
  SG_PARAMS_LIST.OU[3].array[0]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[3].array[1]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[3].array[2]=~((unsigned int)(0));
  SG_PARAMS_LIST.OU[3].array[3]=~((unsigned int)(0));
  // OD /////////////////////////////////////////////////
  SG_PARAMS_LIST.OD[0].array[0]=0;
  SG_PARAMS_LIST.OD[0].array[1]=0;
  SG_PARAMS_LIST.OD[0].array[2]=0;
  SG_PARAMS_LIST.OD[0].array[3]=0;
  SG_PARAMS_LIST.OD[1].array[0]=0;
```

```
  SG_PARAMS_LIST.OD[1].array[1]=0;
  SG_PARAMS_LIST.OD[1].array[2]=0;
  SG_PARAMS_LIST.OD[1].array[3]=0;
  SG_PARAMS_LIST.OD[2].array[0]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[2].array[1]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[2].array[2]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[2].array[3]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[3].array[0]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[3].array[1]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[3].array[2]=~((unsigned int)(0));
  SG_PARAMS_LIST.OD[3].array[3]=~((unsigned int)(0));
}
void init_spins() {
  int i;
  for(i=0; i<(4 * VOLUME); i++) Spins[i] = rand();
}
void init_couplings(){
  int i;
  for(i=0; i<(4 * VOLUME); i+=4) {
    Jx[i + 0] = 1; rand();
    Jx[i + 1] = 1; rand();
    Jx[i + 2] = 1; rand();
    Jx[i + 3] = 1; rand();
    Jy[i + 0] = 1; rand();
    Jy[i + 1] = 1; rand();
    Jy[i + 2] = 1; rand();
    Jy[i + 3] = 1; rand();
    Jz[i + 0] = 1; rand();
    Jz[i + 1] = 1; rand();
    Jz[i + 2] = 1; rand();
    Jz[i + 3] = 1; rand();
  }
}
void init_random() {
  int i;
  srand(SEED);
  ip = rand();
  irr = rand();
  for (i=0; i<256; i++) {
    ira[i][0] = ira[i][1] = ira[i][2] = ira[i][3] = rand();
    /*ira[i][1] = */ rand();
    /*ira[i][2] = */ rand();
    /*ira[i][3] = */ rand();
  }
  ip1=(ip+232)%256; ip2=(ip+201)%256; ip3=(ip+195)%256;
}
//-------------------------------------------------------------------------------------------
```

D.2 Lato SPE

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <spu_mfcio.h>
#include <spu_intrinsics.h>
#include <spu_internals.h>
#include <assert.h>
#include <simdmath/log2f4.h>
#include <libmisc.h>
#include <string.h>
#include "../common.h"
#include "../user_common.h"
#include "../spe_lib.h"
#define MAX_TRANSFER_SIZE 16384
//-------------------------------------------------------------------------------------------
// Shared data
spe_params SPE_PARAMETERS                       __attribute__ ((aligned (128)));
spinglasses_params SG_PARAMETERS                __attribute__ ((aligned (128)));
// Pointers must be static!!! otherwise they will change their value for unknown reasons
static vector unsigned int * Spins;
static vector unsigned int * Jx;
static vector unsigned int * Jy;
static vector unsigned int * Jz;
vector unsigned int ira[SPE_RANDOM_VECTOR_SIZE] __attribute__ ((aligned (128)));
//-------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------
// Local data
const vector unsigned int ones = { (signed int)1, (signed int)1,
                                   (signed int)1, (signed int)1 };
```

```
const vector unsigned int mask = { 0xff, 0xff, 0xff, 0xff };
const vector float THREES      = { 3.0, 3.0, 3.0, 3.0 };
vector signed int multiplier;
vector signed int vSIDE_1;
vector signed int vSIDE;
vector signed int vZERO        = {    0,    0,    0,    0 };
unsigned int inSpin __attribute__ ((aligned (4)));
unsigned int x, y, z, go = 1;
vector unsigned int p_ips = {0, 0, 0, 0};
// Tick times
unsigned int t_start = 0;   // Tick di partenza
unsigned int t_end   = 0;   // Tick di fine
unsigned int t_avg   = 0;
vector float C_inv_max_rand;
vector float negative_T_4;
vector float LN_2 = { (float)0.69314718055995, (float)0.69314718055995,
                      (float)0.69314718055995, (float)0.69314718055995 };
unsigned long long int addr;
// Sizes of cube must be calculated on SPE side
unsigned int SIDE, FACE, VOLUME, Q, R;
//-----------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------
run_status RUN_STATUS __attribute__ ((aligned (128)));
run_ticks RUN_TICKS  __attribute__ ((aligned (128)));
//-----------------------------------------------------------------------------------
void sendMailToPPE() {
  do { } while(spu_stat_out_mbox () == 0);
  spu_write_out_mbox(1);
}
int waitMailFromPPE() {
  while(spu_stat_in_mbox() == 0) {};
  return spu_read_in_mbox ();
}
void loadConfiguration() {
  register int offset;
  unsigned int i;
  //---------------------------------------------------------------------------------
  // Get param struct from gived address
  mfc_get( (void *) &SPE_PARAMETERS, addr, sizeof (spe_params), 1, 0, 0 );
  mfc_write_tag_mask ( 1 << 1 );
  mfc_read_tag_status_all ();
  mfc_get( (void *) &ira, SPE_PARAMETERS.ira_addr, sizeof (ira), 1, 0, 0 );
  mfc_get( (void *) &SG_PARAMETERS, SPE_PARAMETERS.sg_param_addr, sizeof (SG_PARAMETERS),
                                                                                1, 0, 0 );
  mfc_write_tag_mask ( 1 << 1 );
  mfc_read_tag_status_all ();
  SIDE   = (SG_PARAMETERS.SIDE);
  FACE   = (SG_PARAMETERS.FACE);
  VOLUME = (SG_PARAMETERS.VOLUME);
  Q      = (SG_PARAMETERS.Q);
  R      = (SG_PARAMETERS.R);
  Spins = calloc_align(VOLUME, sizeof(vector unsigned int), 8);
  Jx    = calloc_align(VOLUME, sizeof(vector unsigned int), 8);
  Jy    = calloc_align(VOLUME, sizeof(vector unsigned int), 8);
  Jz    = calloc_align(VOLUME, sizeof(vector unsigned int), 8);
  if((!Spins) || (!Jx) || (!Jy) || (!Jz)) {
    printf("ERROR (SPE): calloc() failed\n");
    exit(1);
  }
  offset = 0;
  for(i=0; i<VOLUME; i++) {
    mfc_get( (void *) Spins + offset, SPE_PARAMETERS.Spins_addr + offset, 16, 1, 0, 0 );
    mfc_get( (void *) Jx    + offset, SPE_PARAMETERS.Jx_addr    + offset, 16, 1, 0, 0 );
    mfc_get( (void *) Jy    + offset, SPE_PARAMETERS.Jy_addr    + offset, 16, 1, 0, 0 );
    mfc_get( (void *) Jz    + offset, SPE_PARAMETERS.Jz_addr    + offset, 16, 1, 0, 0 );
    offset += 16;
  }
  mfc_write_tag_mask ( 1 << 1 );
  mfc_read_tag_status_all ();
  C_inv_max_rand = spu_splats((float)SG_PARAMETERS.C_inv_max_rand);
  negative_T_4   = spu_splats((float)-SG_PARAMETERS.T_4);
  p_ips = spu_insert((signed int)SG_PARAMETERS.ip,  p_ips, 0);
  p_ips = spu_insert((signed int)SG_PARAMETERS.ip1, p_ips, 1);
  p_ips = spu_insert((signed int)SG_PARAMETERS.ip2, p_ips, 2);
  p_ips = spu_insert((signed int)SG_PARAMETERS.ip3, p_ips, 3);
  multiplier = spu_insert(1,    multiplier, 0);
  multiplier = spu_insert(SIDE, multiplier, 1);
  multiplier = spu_insert(FACE, multiplier, 2);
  vSIDE_1 = spu_insert(SIDE-1, vSIDE_1, 0);
  vSIDE_1 = spu_insert(SIDE-1, vSIDE_1, 1);
```

```
  vSIDE_1 = spu_insert(SIDE-1, vSIDE_1, 2);
  vSIDE_1 = spu_insert(SIDE-1, vSIDE_1, 3);
  vSIDE = spu_insert(SIDE, vSIDE, 0);
  vSIDE = spu_insert(SIDE, vSIDE, 1);
  vSIDE = spu_insert(SIDE, vSIDE, 2);
  vSIDE = spu_insert(SIDE, vSIDE, 3);
}
void saveConfiguration() {
  register int offset = 0;
  unsigned int i;
  offset = 0;
  for(i=0; i<VOLUME; i++) {
    mfc_put( (void *) Spins + offset, SPE_PARAMETERS.Spins_addr + offset, 16, 1, 0, 0 );
    offset += 16;
  }
  mfc_write_tag_mask ( 1 << 1 );
  mfc_read_tag_status_all ();
}
//-------------------------------------------------------------------------------------------
int user_main ( ) {
  unsigned int i;
  vector float IDXf;
  vector unsigned int IDX,
                      select,
                      Carry1,
                      Carry2,
                      EU,
                      ED,
                      EC,
                      prod;
  vector signed int xyzp  = {0, 0, 0, 0},
                    xyzm  = {0, 0, 0, 0};
  vector unsigned int pos_sel, neg_sel;
  unsigned int haddr, laddr;
  //-----------------------------------------------------------------------------------------
  // Load SPE parameters; waiting for param struct address
  while ( spu_stat_in_mbox () == 0 ) {} haddr = spu_read_in_mbox ();
  while ( spu_stat_in_mbox () == 0 ) {} laddr = spu_read_in_mbox ();
  addr = (((unsigned long long int)haddr)<<32) | laddr;
  //-----------------------------------------------------------------------------------------
  // Metropolis Update
  loadConfiguration();
  do {
    // Resetto il contatore
    spu_writech (SPU_WrDec, 0x7fffffff);
    t_start = spu_readch (SPU_RdDec);
    for(i=0; i<SG_PARAMETERS.iter; i++) {
      // Loop through vectors
      for(z=0; z<SIDE; z++) {
        for(y=0; y<SIDE; y++) {
          for(x=0; x<SIDE; x++) {
            vector signed int xyz   = { x, y, z, 0 };
            vector signed int adder = { (y * SIDE) + (z * FACE), x + (z*FACE),
                                        x + (y * SIDE), 0 };
            inSpin = (x + (y * SIDE) + (z * FACE));
            /////////////////////////////////////////////////////////////////////////////
            // RANDOM
            //
            IDX = spu_add(ira[spu_extract(p_ips, 1)], ira[spu_extract(p_ips, 2)]);
            ira[spu_extract(p_ips, 0)] = IDX;
            IDX = spu_xor(IDX, ira[spu_extract(p_ips, 3)]);
            // Incremento e maschero a 0xff, cosi' da forzare a valori compresi tra 0 e 244.
            p_ips = spu_add(p_ips, ones);
            p_ips = spu_and(p_ips, mask);
            /////////////////////////////////////////////////////////////////////////////
            // IDXf = N_RANDOM;
            //
            IDXf = spu_mul(spu_convtf(IDX, 0), C_inv_max_rand);
            /////////////////////////////////////////////////////////////////////////////
            // Calcolo del logaritmo
            //
            IDXf = spu_mul(_log2f4(IDXf), LN_2);
            /////////////////////////////////////////////////////////////////////////////
            // Parte finale di N_RANDOM
            //
            IDXf = spu_mul(IDXf, negative_T_4);
            /////////////////////////////////////////////////////////////////////////////
            // Blocco dell'if()
            //
            select = spu_cmpabsgt(IDXf, THREES);
            IDXf   = spu_sel(IDXf, THREES, select);
```

```
          IDX    = spu_convtu(IDXf, 0);
          ///////////////////////////////////////////////////////////////////////////////
          // Calcolo le coordinate dei vicini
          //
          // Coordinate positive                 // Coordinate negative
          xyzp = spu_add(xyz, 1);                xyzm = spu_add(xyz, -1);
          pos_sel = spu_cmpeq(xyzp, SIDE);       neg_sel = spu_cmpgt(vZERO, xyzm);
          xyzp = spu_sel(xyzp, vZERO, pos_sel); xyzm = spu_sel(xyzm, vSIDE_1, neg_sel);
          xyzp = spu_madd((vector signed short)xyzp, (vector signed short)multiplier,
                                                                              adder);
          xyzm = spu_madd((vector signed short)xyzm, (vector signed short)multiplier,
                                                                              adder);
          ///////////////////////////////////////////////////////////////////////////////
          // Calcolo il valore dello spin
          //
          EU     = spu_xor(Spins[inSpin], spu_xor(Jx[inSpin],
                               Spins[spu_extract(xyzp, 0)]) );
          prod   = spu_xor(Spins[inSpin], spu_xor(Jx[spu_extract(xyzm, 0)],
                                             Spins[spu_extract(xyzm, 0)]) );
          Carry1 = spu_and(prod,     EU);
          EU     = spu_xor(prod,     EU);
          ED     = Carry1;
          prod   = spu_xor(Spins[inSpin], spu_xor(Jy[inSpin],
                               Spins[spu_extract(xyzp, 1)]) );
          Carry1 = spu_and(prod,     EU);
          EU     = spu_xor(prod,     EU);
          // Carry2 = spu_and(Carry1,   ED);
          ED     = spu_xor(Carry1,   ED);
          // EC     = Carry2;
          prod   = spu_xor(Spins[inSpin], spu_xor(Jy[spu_extract(xyzm, 1)],
                                             Spins[spu_extract(xyzm, 1)]) );
          Carry1 = spu_and(prod,     EU);
          EU     = spu_xor(prod,     EU);
          Carry2 = spu_and(Carry1,   ED);
          ED     = spu_xor(Carry1,   ED);
          EC     = spu_xor(Carry2,   EC);
          prod   = spu_xor(Spins[inSpin], spu_xor(Jz[inSpin],
                               Spins[spu_extract(xyzp, 2)]) );
          Carry1 = spu_and(prod,     EU);
          EU     = spu_xor(prod,     EU);
          Carry2 = spu_and(Carry1,   ED);
          ED     = spu_xor(Carry1,   ED);
          EC     = spu_xor(Carry2,   EC);
          prod   = spu_xor(Spins[inSpin], spu_xor(Jz[spu_extract(xyzm, 2)],
                                             Spins[spu_extract(xyzm, 2)]) );
          Carry1 = spu_and(prod,     EU);
          EU     = spu_xor(prod,     EU);
          Carry2 = spu_and(Carry1,   ED);
          ED     = spu_xor(Carry1,   ED);
          EC     = spu_xor(Carry2,   EC);
          EC = spu_xor(EC, 0xffffffff);   /* EC=~EC; ED=~ED; EU=~EU;           */
                                          /* negation is convenient: see above */
          ED = spu_xor(ED, 0xffffffff);
          EU = spu_xor(EU, 0xffffffff);
          /*OD[IDX],OU[IDX] represent IDX positionally*/
          Carry1 = spu_and(SG_PARAMETERS.OU[spu_extract(IDX, 0)].vec, EU);
          Carry2 = spu_or(spu_and(SG_PARAMETERS.OD[spu_extract(IDX, 0)].vec, ED),
                  spu_and(spu_or(SG_PARAMETERS.OD[spu_extract(IDX, 0)].vec, ED), Carry1));
          /*Carry2=1 if and only if (T/4)logR+(~ED,~EU) >= 4*/
          Spins[inSpin] = spu_xor(Spins[inSpin], spu_or(Carry2, EC));
        }
      }
    }
  }
  t_end = spu_readch (SPU_RdDec);
  if(t_avg == 0) t_avg = (t_start - t_end);
  else t_avg = (t_avg + (t_start - t_end)) / 2;
  saveConfiguration();
  sendMailToPPE();
  go = waitMailFromPPE();
} while(go);
RUN_TICKS.tick_cnt[0] = t_avg;
//---------------------------------------------------------------------------------------
//---------------------------------------------------------------------------------------
// Put ticks
mfc_put ( (void *) &RUN_TICKS, SPE_PARAMETERS.run_ticks_addr, sizeof (run_ticks),
                                                                       1, 0, 0 );
mfc_write_tag_mask ( 1 << 1 );
mfc_read_tag_status_all ();
```

```
  //-----------------------------------------------------------------------------------
  return (0);
}
//-------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------
int main (/* unsigned long long spu_id, unsigned long long argv */) {
  user_main ( );
  return (0);
}
//-------------------------------------------------------------------------------------
```

D.3 File di Libreria

```
//-----------------------------------------------------------------------------------------//
//                                                                                         //
// Define here all data structures and constant that have to be shared between PPE and SPEs.    //
//                                                                                         //
//-----------------------------------------------------------------------------------------//
#ifndef __USER_COMMON__
#define __USER_COMMON__
//-------------------------------------------------------------------------------------
// cab params
#define CAB_TIMEBASE        (14318000)
#define CAB_HOSTNAME        "cab"
// cc03.juice params
#define CC09_JUICE_TIMEBASE (14318000)
#define CC09_JUICE_HOSTNAME "cc09.juice"
// ps3.fe.infn.it params
#define PS3_TIMEBASE        (79800000)                          // PS3 Timebase
#define PS3_HZ              (3.2e9)                             // PS3 Frequency HZ
#define PS3_TICK_PERIOD     (1.0 / PS3_TIMEBASE)                // Tick period in seconds
#define PS3_CLOCK_PERIOD    (1.0 / PS3_HZ)                      // PS3 clock period sec
#define PS3_TICK_N_CLK      (PS3_TICK_PERIOD / PS3_CLOCK_PERIOD) // Number of clocks in a tick
#define PS3_HOSTNAME        "ps3.fe.infn.it"
#define TICK_PERIOD         (1.0 / PS3_TIMEBASE)
#define HOSTNAME            PS3_HOSTNAME
//-------------------------------------------------------------------------------------
// Parameters of SPE
#define ADDR unsigned long long int
typedef union _array_to_vector {
unsigned int        array[4] __attribute__ ((aligned(16)));
#ifndef PPE_SIDE
vector unsigned int vec;
#endif
} array_to_vector;
typedef struct {
  unsigned char pad[4];
  array_to_vector OD[4];
  array_to_vector OU[4];
  unsigned char ip,
              ip1,
              ip2,
              ip3,
              irr;
  float C_inv_max_rand;
  double T_4;
  unsigned int iter;
  unsigned int SIDE, FACE, VOLUME, Q, R;
} spinglasses_params;
typedef struct {
  unsigned char pad[4];           // Padding sempre all'inizio.
  unsigned int idn;  // SPE id            ( 4B)
  ADDR Spins_addr;                // Spins address     ( 8B)
  ADDR Jx_addr;                   // Jx address        ( 8B)
  ADDR Jy_addr;                   // Jy address        ( 8B)
  ADDR Jz_addr;                   // Jz address        ( 8B)
  ADDR ira_addr;                  // Random address    ( 8B)
  ADDR run_ticks_addr;            // Ticks struct addr ( 8B)
  ADDR sg_param_addr;             // Sg    struct addr ( 8B)
                                  // - Total --------- (??B)
} spe_params;
//-------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------
// It VERY important that sizeof(run_status)<=16, because it must be
// transferred with an ATOMIC dma operation
typedef struct {
  unsigned int status;
} run_status;
//-------------------------------------------------------------------------------------
```

```
//----------------------------------------------------------------------------------------
typedef struct {
  unsigned char pad[10];           // Padding sempre all'inizio.
  unsigned int tick_cnt[1];
} run_ticks;
//----------------------------------------------------------------------------------------
//----------------------------------------------------------------------------------------
// Metropolis
#define DEFAULT_T (1.1)
#define _NSAMPLES_ 64
/*
#define SIDE       (15)
#define FACE       (SIDE * SIDE)
#define VOLUME     (FACE * SIDE)*/
// Dimensione di una entry nell'area degli spin (dword a 64bit)
//#define PPE_SPIN_AREA_ENTRY_SIZE   (8) // 8bytes = 64bits
#define PPE_SPIN_AREA_ENTRY_SIZE   (sizeof(unsigned int)) // 8bytes = 64bits
// Dimensione area di memoria degli spin sull'PPE
#define PPE_SPIN_AREA_SIZE         (PPE_SPIN_AREA_ENTRY_SIZE) * K
// Numero di vector int per i randoms
#define SPE_RANDOM_VECTOR_SIZE (256)
// #define SEED (unsigned int)123456789
#define SEED (unsigned int)1
/*#define Q (unsigned int)(VOLUME / 1024 )
#define R (unsigned int)(VOLUME % 1024 )*/
#define N_REPLICAS (128)
////////////////////////////////////////////////////////////////////////////////
// Definisco una macro per attivare/disattivare il conteggio dei tick
//#define TICKS_COUNT
#ifdef TICKS_COUNT
# define TICKS_CODE(__x) __x
#else
# define TICKS_CODE(__x)
#endif
//----------------------------------------------------------------------------------------
#endif
```

Bibliografia

[1] Michael Kistler (IBM Austin Research Laboratory), Michael Perrone (IBM TJ Watson Research Center) e Fabrizio Petrini (Pacific Northwest National Laboratory), **Cell Multiprocessor Communication Network: built for speed**

[2] Michael Gschwind, H. Peter Hofstee, Brian Flachs and Martin Hopkins (IBM), Yukio Watanabe (Toshiba) e Takeshi Yamazaki (Sony Computer Entertainment), **Synergistic Processing In Cell's Multicore Architecture**

[3] Daniel A. Brokenshire (IBM STI Design Center), **Maximizing the power of the Cell Broadband Engine processor: 25 tips to optimal application performance**

[4] E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao e R. Koo, **Using advanced compiler technology to exploit the performance of the Cell Broadband Engine™ architecture**

[5] Jonathan Bartlett (Director of Technology, New Medio), **Programming high-performance applications on the Cell BE processor, Part 1: An introduction to Linux on the PLAYSTATION 3**

[6] Jonathan Bartlett (Director of Technology, New Medio), **Programming high-performance applications on the Cell BE processor, Part 2: Program the synergistic processing elements of the Sony PLAYSTATION 3**

[7] Jonathan Bartlett (Director of Technology, New Medio), **Programming high-performance applications on the Cell BE processor, Part 3: Meet the synergistic processing unit**

[8] Jonathan Bartlett (Director of Technology, New Medio), **Programming high-performance applications on the Cell BE processor, Part 4: Program the SPU for performance**

[9] Jonathan Bartlett (Director of Technology, New Medio), **Programming high-performance applications on the Cell BE processor, Part 5: Programming the SPU in C/C++**

[10] Power Architecture editors, developerWorks, IBM e Lewin Edwards (Design Engineer, Freelance), **An introduction to compiling for the Cell Broadband Engine architecture, Part 1: A bird's-eye view**

[11] Power Architecture editors, developerWorks e IBM, **An introduction to compiling for the Cell Broadband Engine architecture, Part 2: Optimizing for the SPE**

[12] Power Architecture editors, developerWorks e IBM, **An introduction to compiling for the Cell Broadband Engine architecture, Part 3: Make the most of SIMD**

[13] Power Architecture editors, developerWorks e IBM, **An introduction to compiling for the Cell Broadband Engine architecture, Part 4: Partitioning large tasks**

[14] **Progetto Manhattan**, `http://it.wikipedia.org/wiki/Progetto_Manhattan`

[15] **RISC**, `http://en.wikipedia.org/wiki/RISC`

[16] **SIMD Math Library Specification for Cell Broadband Engine Architecture, version 1.0**

[17] **C/C++ Language Extensions for Cell Broadband Engine Architecture, version 2.4**

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

[19] Walter T. Grandy, **Foundations of Statistical Mechanics**

[20] **Loop Unrolling**, `http://en.wikipedia.org/wiki/Loop_unrolling`

[21] Brian W. Kernighan e Dennis M. Ritchie, **Linguaggio C**, Jackson, Milano, 1980, 88-7056-211-5

[22] J. A. Kahlem, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer e D. Shippy, **Introduction to the Cell multiprocessor**

[23] **Streaming SIMD Extension**, `http://it.wikipedia.org/wiki/Streaming_SIMD_Extensions`

[24] N. W. Ashcroft e N. D. Mermin, **Solid state physics**, Harcourt, 1976, 0030839939

[25] George Marsaglia e Wai Wan Tsang, **Some Difficult-to-pass Tests of Randomness**, Vol. 7, Issue 3, Jan 2002

[26] C. Kittel, **Elementary Statistical Physics**, Dover Publications, 2004

[27] N. Metropolis, A. N. Rosenbluth, M. N. Rosenbluth, A. H. Teller e E. Teller, **Equation of state calculation by fast computing machines**, Journal of Chemical Physics, 1953

[28] **Out-of-order execution**, `http://en.wikipedia.org/wiki/Out-of-order_execution`

[29] M. Mezard, G. Parisi e M. Virasoro, **Spin Glass Theory and Beyond**