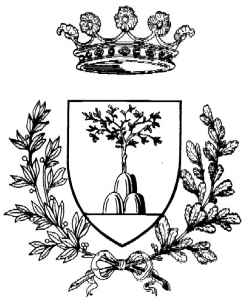
# UNIVERSITÀ DEGLI STUDI DI FERRARA

**Facoltà di Scienze Matematiche, Fisiche e Naturali**

Corso di Laurea in Informatica

# Simulazione Lattice Boltzmann su architettura Cell-BE


Primo relatore:
**Sebastiano Fabio Schifano**

Tesi di:
**Paolo Bianchi**

Correlatore:
**Francesco Belletti**



Anno Accademico 2006/2007

# Indice

# Introduzione

# Capitolo 1

# Simulazione lattice Boltzmann

## 1.1 Introduzione

Il metodo *Lattice Boltzmann* che viene qui illustrato è l'evoluzione di quelli che vengono definiti *Lattice Gas Cellular Automata* che costituiscono i modelli di simulazione per la fluido dinamica. Di seguito una breve descrizione storica degli eventi che hanno portato alla realizzazione del metodo Lattice Boltzmann. Negli anni '40 cominciarono a prendere forma nel mondo della fisica quelli che vengono chiamati *Automi Cellulari* e il cui pioniere fu *John von Neumann*, questi AC[1] sono una rappresentazione ideale dei fenomeni fisici in cui lo spazio e il tempo sono quantità discrete. Il concetto di *AC* era quello di una macchina in grado di autocontrollarsi e auto-ripararsi dal momento in cui gli fossero stati dati alcuni input iniziali. *von Neumann* pensava che l'universo fosse rappresentabile come un insieme di celle ognuna delle quali caratterizzata da uno stato interno *"intero"*, in cui le informazioni interne potessero essere rappresentate mediante un numero finito di bit (senza alcuna approssimazione o troncamento). Ogni *cella* si evolve secondo delle regole di interazione con le celle vicine (che sono le uniche a lei note) e in passi temporali (time-step) discreti. L'attivitá di evoluzione delle *celle* è simultanea su tutti gli elementi e segue un insieme di semplici regole prestabilite, uguali per tutti gli elementi. L'evoluzione ha luogo da una configurazione conosciuta del sistema e può portare anche a stati complessi e inattesi, questi comportamenti complessi ci permettono di simulare processi fisici reali.

Dopo l'avvento di *von Neumann* altri furono quelli che si cimentarono nel campo degli Automi Cellulari, nel 1970 *John Conway* produsse quello che è conosciuto come il *'game of life'*, un AC bi-dimensionale che simula l'evolversi di una società reale. Il sistema parte da una configurazione casuale e si evolve secondo uno schema di vita/morte delle celle in cui una cella può essere l'uno o l'altro stato sulla base del numero di celle morte/vive vicine nello stato precedente. Questo "semplice" gioco mise in evidenza come l'evoluzione del sistema potesse raggiungere stati molto complessi proprio come una societá reale.

Negli anni '80 alcuni studi sugli AC uni-dimensionali dimostrarono la capacità di simulare sistemi continui utilizzando strutture semplici rispetto a quelle utilizzate fino a quel momento. Fu sempre negli anni '80 che si compì un passo fondamentale per gli AC riconoscendo un *modello Lattice Gas* come *Automa Cellulare*. Questo modello consiste di una dinamica discreta di particelle che collidono e si muovono in una griglia

[1] Da questo momento in poi ci riferimemo agli Automi Cellulari con il termine AC

bidimensionale in modo da **conservare** massa e momento, in questo Automa Cellulare ogni time-step è composto di due fasi: una fase di collisione e una fase di propagazione (diffusione). Un modello del tutto simile fu ideato per i fluidi, tali metodi prendono lo stesso nome ovvero *Lattice Gas Automata*.

L'introduzione dei modelli *lattice Gas* portano ad un importante evoluzione nel mondo della simulazione dei processi fisici, si è infatti in grado di simulare i comportamenti macroscopici di un fluido o gas mediante l'utilizzo di un modello microscopico. I fenomeni macroscopici sono molto complessi e regolati da equazioni differenziali la cui risoluzione richiede molte risorse (sia in termini di potenza di calco che di tempo di elaborazione). I nuovi metodi ci evitano la risoluzione delle suddette equazioni differenziali e introducono importanti fattori di miglioramento nella simulazione dei fenomeni quali la località delle interazioni e la reversibilità delle operazioni di evoluzione del sistema.

Questo nuovo modo di approcciare allo studio del mondo macroscopico mediante il micro-mondo di interazione delle particelle ha trovato negli anni successivi ampi spazi per svilupparsi.

Nel 1986 fu ideato quello che è conosciuto come il modello FHP[2] che segue il comportamento fisico microscopico dei fluidi, ma che a livello macroscopico arriva all'equazione di *Navier-Stokes*.

Questi modelli mostrarono alcune limitazioni quando si cercava di portarli in particolari condizioni in cui i valori diventavano tali[3] per cui il fluido non ha più un comportamento di fluido laminare -cioè come se fosse composto di tanti sottili strati- ma quello di un fluido turbolento regolato da complesse equazioni.

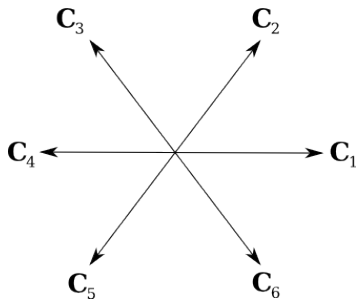



Figura 1.1: Insieme delle velocità discretizzate del *Lattice Gas Cellular Automata* nel caso bidimensionale.

Gli Automi Cellulari hanno il grande vantaggio di rendere semplice la descrizione del metodo di simulazione del processo fisico reale, così come rendono semplice l'implementazione di condizioni di bordo -anche complesse- ma il problema principale di questi metodi di simulazione è dato dalla presenza di rumori statistici nei dati (che possiamo pensare come piccoli errori) a questi si aggiunge l'impossibilità di questi metodi di rappresentare ampi processi fisici.

[2]Dal nome degli ideatori: *Frisch,Hasslacher e Pomeau*.

[3]I particolari valori sono in realtà alti valori dei numeri di *Reynolds*, che sono espressi in funzione della densità del fluido $\rho$, la velocità media del fluido $U$, la viscosità dinamica $\mu$, la viscosità cinematica $\nu$, $\nu = \mu/\rho$, la lunghezza caratteristica del corpo (per il moto in condotti equivale al diametro 2r se la sezione del condotto è circolare, altrimenti è pari al cosiddetto diametro equivalente -o diametro idraulico-).

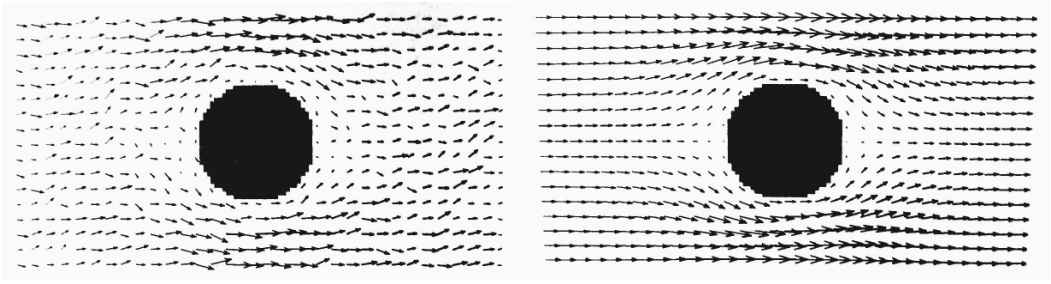



Figura 1.2: Confronto tra i due metodi nella simulazione di un fluido in cui è immerso un corpo cilindrico. Come si vede il metodo lattice Bolztmann risulta più accurato.

I problemi dei metodi LGCA sono stati risolti con l'introduzione nell'insieme dei metodi LGCA del *lattice Boltzmann Method* (LBM). Il LBM deriva dalle equazioni di Boltzmann e discende dai citati *lattice Gas Cellular Automata* di cui eredita la classificazione ma non i limiti. Da un lato gli LGCA sono in grado di simulare in modo accurato il fenomeno fisico in quanto gli stati sono descritti da valori interi, dall'altro lato abbiamo che per un numero elevato di particelle nel sistema si presentano delle '*fluttuazioni*'.

Come abbiamo detto il LBM non è soggetto alle limitazioni del LGCA pur derivando da esso.

La microdinamica di un LGCA è regolata da una relazione del tipo:

$$n_i\left(\mathbf{x}+\mathbf{c_i}, t+1\right) \; - \; n_i\left(\mathbf{x}, t\right) \; = \; \Omega\left(n_i(\mathbf{x}, t)\right), \quad (i=0, \; \ldots \, , M) \tag{1.1}$$

dove $t$ è il tempo in unità discrete, $x$ è lo spazio discretizzato,$\mathbf{n} = (n_1 \; n_2 \; \ldots \; n_6)$ sono le possibili direzioni in cui si può muovere la cella in questo caso 6 come mostrato in Figura 1.3 , con $n_i \; = \; 0, 1$ inquanto varibili booleane del modello dove 0 indica l'assenza e 1 indica la presenza di particelle in $\mathbf{x}$ con una velocità $\mathbf{c}_i$. Il secondo membro della relazione è l'operatore $\Omega_i$ di collisione ed è legato al fenomeno degli urti tra le particelle.

Nel metodo lattice Boltzmann le variabili non sono più booleane ma reali ed esprimono un valore di probabilità e la relazione che esprime l'evoluzione del modello è definita dalla seguente funzione di distribuzione:

$$F_i\left(\mathbf{x}+\mathbf{c}_i\Delta t+\Delta t\right) \; = \; -\frac{t}{\tau}\left(F_i - F_i^{(eq)}\right) \tag{1.2}$$

L'elemento a secondo membro della relazione è sempre l'operatore di collisione, abbiamo infatti che

$$\Omega_i \; = \; -\frac{t}{\tau}\left(F_i - F_i^{(eq)}\right) \tag{1.3}$$

Quello che la funzione $F_i$ esprime è la probabilità che una particella si trovi in un dato punto $x$ ad una certa velocità $\vec{\nu}$ che dipende dalle velocità discrete indicate con il termine $c_i$. Il fatto che il metodo utilizzi un modello discreto delle velocità può risultare in un certo senso innaturale, possiamo pensare che una particella sia libera di muoversi secondo una qualsiasi velocità sia in termini di direzione che di modulo, in realtà quello che accade con il modello *lattice Boltzmann* è che il set di 9 velocità per le simulazioni di un fluido

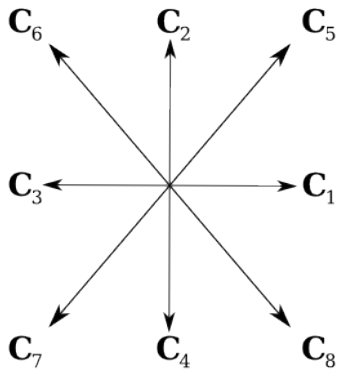



Figura 1.3: Insieme delle velocità discretizzate del *Lattice Boltzmann Method* nel caso bidimensionale D2Q9.

in un ambiente bidimensionale è più che sufficiente al fine di ricavarne il comportamento macroscopico. Il modello LBM può essere impiegato per simulazioni bi-dimensionali e tri-dimensionali, che hanno diverse configurazioni del lattice (reticolo):(vedi sez 1.3.3 lattice structures tesi di jonas latt)

- D2Q9: bi-dimensionale con 9 componenti di velocità
- D3Q15: tri-dimensionale con 15 componenti di velocità
- D3Q19: tri-dimensionale con 19 componenti di velocità
- D3Q27: tri-dimensionale con 27 componenti di velocità

il modello che a noi interessa e che verrà implementato a livello software coinvolge un reticolo bidimensionale a 9 componenti.

Quello che è importante notare del modello lattice Bolztmann è che la massa e il momento delle velocità devono conservarsi il che significa che data una massa iniziale ed una configurazione iniziale del sistema l'operatore di collisione deve seguire i seguenti vincoli:

$$\sum_i \Omega_i \;=\; 0 \qquad \sum_i \Omega_i \mathbf{c}_i \;=\; 0 \tag{1.4}$$

Parlo delle condizioni di bordo

...Proseguo parlando dall'implementazione dell'algoritmo da cui siamo partiti.

## 1.2 Descrizione FisicoMatematica

## 1.3 Caratteristiche del metodo

### 1.3.1 collide()

### 1.3.2 displace()

### 1.3.3 bc()

### 1.3.4 occupazione in memoria, calcoli

## 1.4 Attuali implementazioni e metodologie per affrontare il problema

## 1.5 Considerazioni Preliminari sul codice

# Capitolo 2

# Architettura Cell Broadband Engine

L'architettura Cell Broadband Engine è nata per un ampio utilizzo in varie fasce di utenza, dai dispositivi elettronici quali display ad alta definizione e console per i giochi a Server Blade per il calcolo.

Il Cell Broadband Engine (CBE o solo Cell) è un multiprocessore su singolo chip, composto di 9 processori che operano accedendo ad un unica memoria condivisa. La caratteristica principale del Cell è che, tutti i processori condividono lo stesso sistema di memorizzazione ( lo spazio di indirizzamento che comprende la memoria principale), ma si specializzano in due tipi diversi: il processore PowerPC (PPE, PowerPC Processor Element) e il processore Sinergico (SPE Synergistic Processor Element). L'architettura prevede un singolo PPE e 8 SPEs. Il PPE è un processore PowerPC a 64-bit in tutto e per tutto ed è in grado di eseguire sistemi operativi e applicazioni a 64 o 32 bit, mentre il secondo tipo di processore (SPE) è ottimizzato per l'esecuzione di applicazioni *compute-intensive* e non è ottimizzato per l'esecuzione del sistema operativo. Gli SPE sono processori indipendenti, che eseguono applicazioni indipendenti, ognuno dei quali ha accesso alla memoria condivisa incluso lo spazio MMIO[1].

Lo scopo dei processori sinergici è quello di fornire al PPE un supporto computazionale ad elevate prestazioni. Il PPE svolge il ruolo di coordinatore, manager delle risorse (*control-intensive*) mentre gli SPEs sono i fornitori di potenza comutazionale e quindi adatti a svolgere applicazioni *compute-intensive*.

Gli SPEs sono pensati per essere programmati mediante l'utilizzo di linguaggi di alto livello, l'*Instruction Set* mette a disposizione un insieme di istruzioni per il calcolo vettoriale (SIMD, Single-Instruction Multiple-Data) che li rendono particolarmente performanti. Come per le architetture standard le istruzioni vettoriali non sono un obbligo ma una possibilità per ottenere maggiori performance. L'impiego del PPE come coordinatore e dei vari SPE come elementi calcolatori asincroni permette di raggiungere performance di un ordine di grandezza superiore alle architetture tradizionali.

Un importante differenza tra SPE e PPE sta nel modo in cui questi effettuano l'accesso alla memoria (che è condivisa tra tutti e 9 processori). Il PPE accede alla Memoria

[1] Il Memory-mapped I/O (ingressi/uscite mappati in memoria) usa lo stesso bus per indirizzare sia la memoria che i dispositivi di I/O, e le stesse istruzioni della CPU utilizzate per leggere e scrivere la memoria sono utilizzate per accedere ai dispositivi di I/O. Per poter mappare i dispositivi di I/O in memoriza nello spazio indirizzabile dalla CPU è necessario riservare una porizione dello spazio di indirizzamento.

Principale (MainMemory) con l'utilizzo di istruzioni di **Load** e **Store** che operano tra il register-file del PPE e la memoria. Gli SPE accedono alla memoria principale mediante comandi di *accesso diretto alla memoria Direct-Memory-Access* (DMA) che operano tra *Local Store* e memoria principale. Lato SPE il prelievo delle istruzioni,il load e store dei dati operano tra Local Store e register-file dell'unità Sinergica e non sulla memoria principale.

L'organizzazione in 3 livelli (*register-file*, *LocalStore* e *Memoria Principale*) della memoria esplicita la parallelizzazione tra i calcoli e i trasferimenti di dati o istruzioni. Quello che l'architettura offre è un nuovo modo di approcciare alla parallelizzazione in cui ogni unità può fare richiesta di accedere alla memoria mentre esegue i calcoli sui dati già presenti nel *LocalStore*. Parallelamente ai calcoli i dati vengono prelevati dalla memoria principale e copiati nel *LocalStore*.

Nelle architetture tradizionali un programma che fa accesso ad un dato che non è presente in cache (dando luogo ad un Miss) viene sospeso per alcune centinaia di cicli in attesa del prelievo dei dati dalla memoria principale, comparati al numero di cicli necessari per inoltrare una richiesta DMA questi valori sono di un ordine di grandezza superiore.

L'architettura richiede uno sforzo al programmatore che deve essere in grado di pensare e realizzare in modo esplicito la parallelizzazione tra esecuzione del codice e richieste di trasferimento dati, a fronte di un guadagno considerevole sulle performance.

L'obbiettivo principale del Cell è quello di raggiungere alte performance di calcolo con la parallelizzazione del lavoro "on-chip".

Il Cell Broadband Engine cerca di superare alcuni dei problemi che affliggono le attuali architetture, come la dissipazione di calore, i tempi di accesso alla memoria e la frequenza dei processori.

Aumentare le performance dei microprocessori significa scontrarsi con la limitata capacità di dissipazione del calore piuttosto che con il numero di transistor disponibili da cui l'unico modo efficiente per poter aumentare le performance è quello di ottimizzare l'efficienza energetica dei processori tanto quanto l'incremento delle performance. Un modo per migliorare l'efficienza termica dei processori consiste nel differenziare i processori ottimizzati per l'esecuzione di applicazioni computazionalmente complesse da quelli per l'esecuzione del sistema operativo. Il Cell-BE riesce in questo fornendo un processore general-purpose (il PPE) e 8 processori Singerici per applicazioni di calcolo intensivo (gli SPEs).

Come abbiamo detto la latenza della memoria costituisce in tutti i programmi un fattore cruciale per le performance, il programmatore deve cercare di gestire esplicitamente la presenza della cache nel sistema di accesso ai dati, il che non è sempre semplice. Le performance sono per la maggior parte determinate dall'efficenza degli accessi alla memoria. Il Cell implementa due meccanismi principali per porre rimedio al dominio della memoria sulle performance, il primo consiste nell'organizzazione in 3 livelli del sistema di memorizzazione che come abbiamo già detto in precedenza può essere visto come l'insieme di *Register-File*, *LocalStore* e *MainMemory*, il secondo metodo utilizzabile per evitare di subire l'influenza dei tempi di accesso alla memoria è dato dalla possibilità di sfruttare trasferimenti DMA *asincroni* tra memoria principale e *LocalStores*, che consentirebbe al programmatore di effettuare richieste di nuovi dati mentre se ne stanno elaborando altri,

la sovrapposizione di tempi di accesso alla memoria ai tempi di calcolo costituisce un importante fattore di guadagno.

I processori convenzionali hanno sino ad ora richiesto una profondità di pipeline sempre maggiore per poter raggiungere frequenze operative sempre più elevate, ma quello che si è raggiunto oggi è un punto in cui aumentare nuovamente le profondità di pipeline puó introdurre effetti negativi. Il Cell specializza le unità in control-intensive(PPE) e compute-intensive(SPE) per raggiungere frequenze superiori. L'archiettura Cell Broadband Engine si presenta come un sistema computazionalmente efficace in grado di raggiungere elevate performance ad un costo di poco superiore ad una macchina tradizionale.

## 2.1 Elementi PPE, SPE, EIB

Possiamo pensare l'architettura come divisa in 3 parti principale: il PPE gli SPEs e il Bus di interconnessione degli elementi (Element Interconnection Bus). In figura 2.1 viene mostrato un schema concettuale dell'architettura, in questa sezione verranno descritte le singole parti.

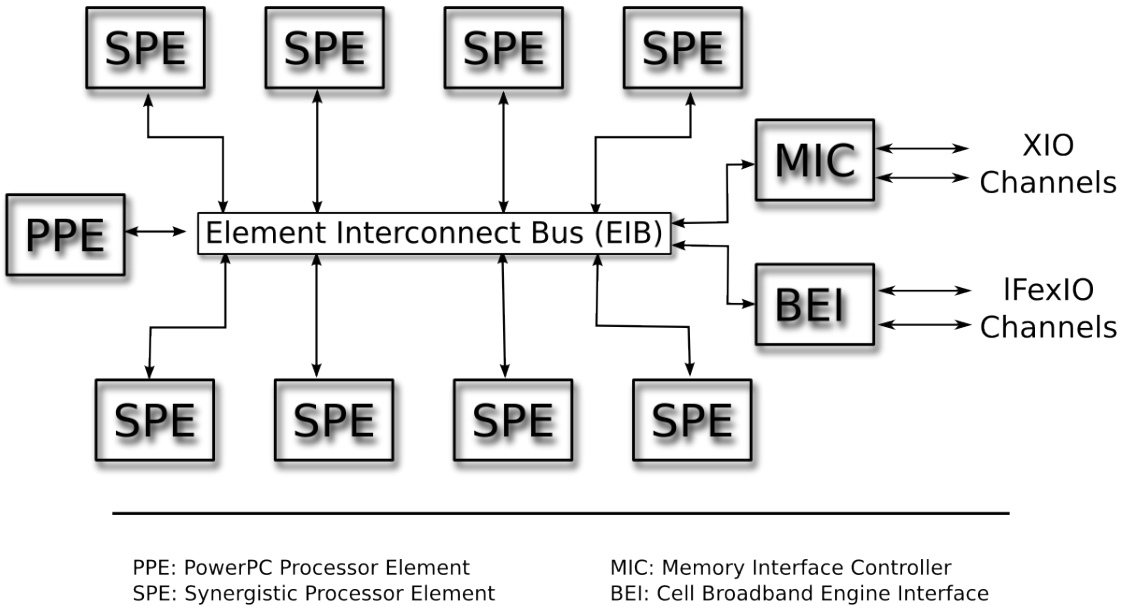



Figura 2.1: Architettura Cell Broadband Engine

Il Bus di Interconnessione ha due interfacce esterne come mostrato in figura, che sono il Memory Interface Controller (MIC) e il Cell Broadband Engine Interaface (BEI). Il MIC fornisce l'accesso alla memoria principale, supporta accessi alla memoria di grandezza 1-8,16,32,64 o 128 Bytes. Il BEI gestisce il trasferimento dati tra l'EIB e i dispositivi di I/O. Fornisce la traduzione degli indirizzi, elaborazione dei comandi, un controller interno di interrupt e l'interfacciamento dei BUS. Supporta due canali di accesso il FlexIO e I/O per i dispositivi esterni, un canale supporta solo trasferimenti non-coherent mentre l'altro

può supportare trasferimenti sia coherent che non-coherent, estendendo così il Bus di interconnessione a dispositivi esterni quali per esempio un altro Cell Broadband Engine.

### 2.1.1 PowerPC Processor Element

Il PowerPC Processor Element è il processore principale, costituito di un unità PowerPC 64bit standard con *Instruction Set* RISC[2]. Il PPE è pensato per svolgere il ruolo di coordinatore delle risorse, è in grado di far girare il sistema operativo e principalmente dovrebbe svolgere azioni di management dell'allocazione di memoria e gestire i threads degli SPEs. Il PPE è in grado di eseguire codice PowerPC standard e supporta l'instruction set PPC e le Vector/SIMD multimedia extensions.

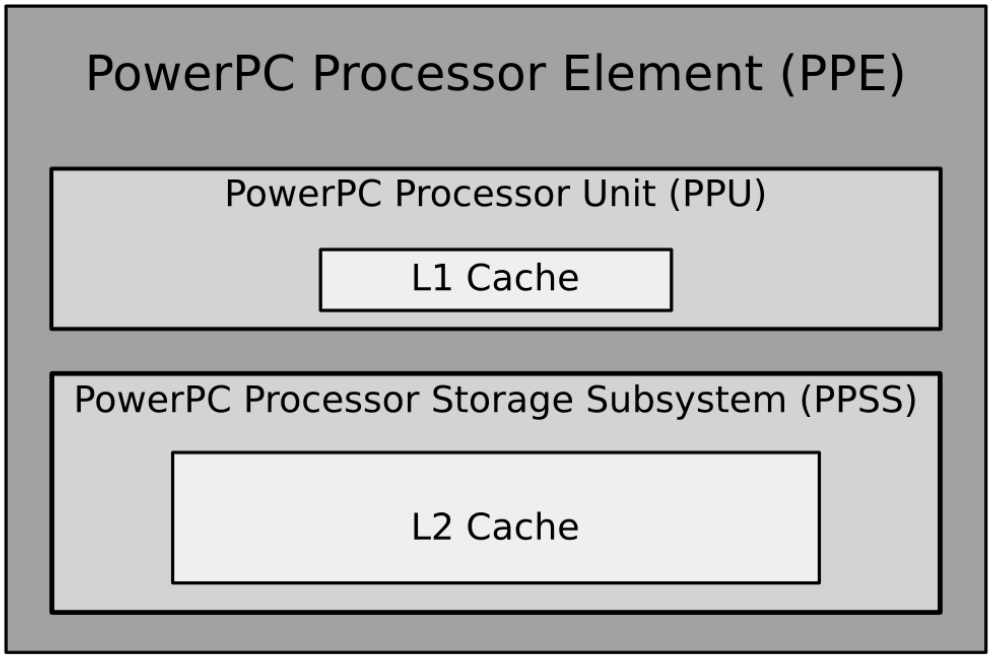



Figura 2.2: Schemattizzazione PPE

Il PPE è suddiviso in due unità principali il Power Processor Unit (PPU) e il Power Processor Storage Subsytem (PPSS). Il PPU gestisce ed esegue le istruzioni, include un set di registri a 64 Bit PowerPC standard, 32 registri vettoriali a 128 Bit, una cache di primo livello (L1 cache) per le istruzioni di 32-KB, una cache dati di primo livello di 32-KB, un unità di gestione delle istruzioni, un unità load/store, un unità fixed-point intera, una unità floating-point, un unità vettoriale, un unità di management della memoria e infine un unità di branch.

[2] *Reduced Instruction Set Computer* (RISC), rappresenta una filosofia di progettazione in cui l'instruction set è un insieme di operazioni semplici con latenze di esecuzione simili. Si contrappone a quella che viene definita come *Complex Instrunction Set Computer* (CISC) che predilige un instruction set molto diversificato in cui le istruzioni possono avere latenze molto diverse.

Il PPU supporta l'esecuzione simultanea di due thread, il che può essere visto come un multiprocessore a due vie con flusso di dati condiviso. A livello software questi due threads sono visti come la disponibilità di due unità di elaborazione. Lo stato di ogni thread è duplicato, inclusi i registri dell'architettura e delle estensioni, fanno eccezione solo quelli che hanno a che fare con le risorse a livello di sistema, come le partizioni logiche, la memoria e il controllo dei thread. La maggior parte delle risorse come la cache o le code sono condivise tra i due thread, non lo sono invece quelle risorse o troppo piccole o che hanno un ruolo fondamentale per le performance di applicazioni multithreaded.

Il PPSS gestisce le richieste di accesso alla memoria dal PPE e richieste proveniente dall'esterno verso il PPE come per esempio richieste dai dispositivi esterni o dagli altri processori. Il PPSS è composto di: una memoria cache di secondo livello unificata per dati e istruzioni di 512-KB, alcune code e un unità di interfacciamento sull'EIB che gestisce gli accessi al bus. La memoria viene vista come un array lineare di Bytes indicizzati a partire da 0 fino a $2^{64} - 1$, dove ogni byte è identificato da un indice chiamato anche indirizzo del Byte e dove ogni Byte contiene un valore. Ad ogni istante è possibile un solo accesso alla memoria e gli accessi seguono l'ordine stabilito dal programma.

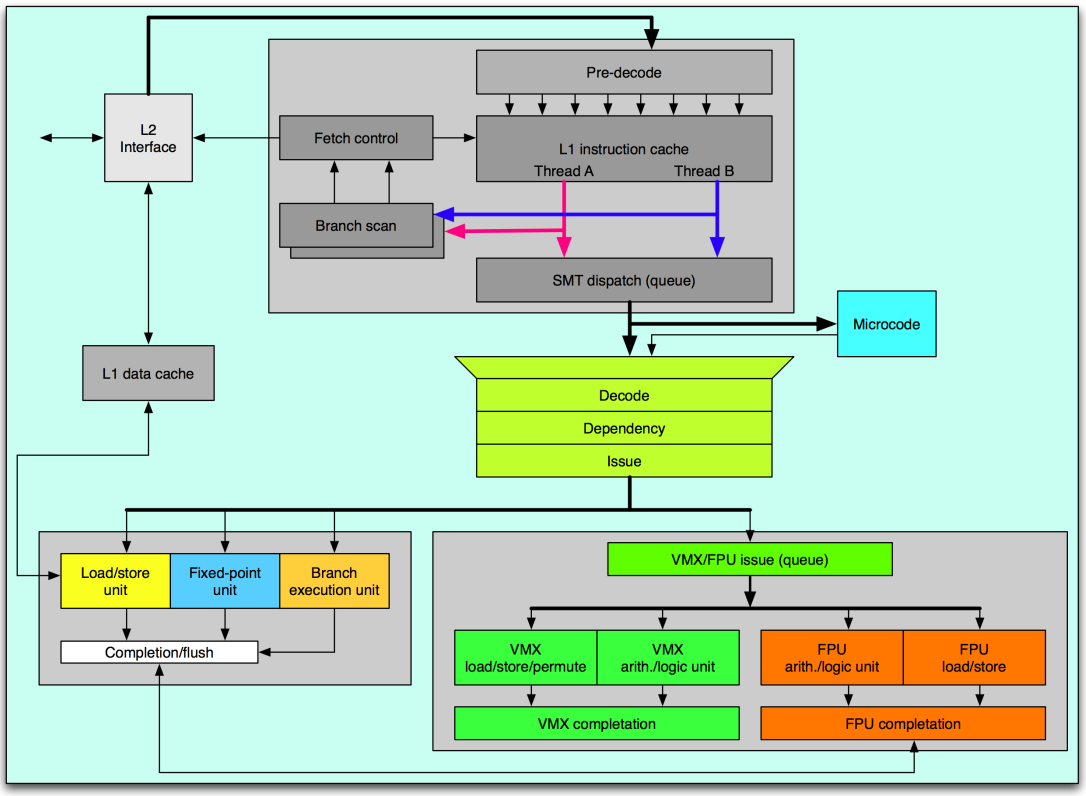



Figura 2.3: PPE nelle sue varie unità

La L2 Cache e le caches di traduzione degli indirizzi impiegano tabelle di replace-management che permettono a livello software di controllare l'impiego della cache. Questo tipo di controllo software sull'utilizzo della cache è di particolare importanza per la programmazione real-time.

Il PPE supporta due tipi di *Instruction-Set*: il PowerPC instruction set e il Vector/SI-

MD Multimedia Extension instruction set. Data la presenza delle funzioni instrinsics in C è bene comprendere meglio l'architettura sottostante, per produrre codice efficiente e ottimizzato. Le funzioni *intrinsics* si mappano direttamente su una o più istruzioni assembler vettoriali (SIMD). http://www-128.ibm.com/developerworks/power/library/pa-cellperf/ L'instruction set del PowerPC utilizza istruzioni di 4 Bytes word-aligned. Supporta l'accesso ad operandi di tipo Byte (8-Bit), halfword(16-Bit),word (32-Bit) e doubleword (64-Bit) sia in memoria che nel proprio register-file. L'instruction set fornisce anche l'accesso ad operandi word e doubleword (32,64 Bit) tra memoria e i 32 registri in floating-point presenti nell'architettura. I valori interi con segno (numeri interi positivi e negativi) sono rappresentati nella forma del complemento a 2 del numero.

L'instruction set Vector/SIMD Multimedia Extension utilizza istruzioni di 4 Bytes word-aligned, ma con operandi a 128 Bit. La maggior parte degli operandi delle istruzioni Vector/SIMD sono vettori di più elementi, come ad esempio i numeri in singola precisione, gli interi, i valori scalari , gli elementi di un vettore possono essere grandi 8.16.32 bit.

### 2.1.2 Synergistic Processor Element, SPE

Ognuno degli 8 SPE è un processore 128-bit RISC pensato per applicazioni *compute-intensive* SIMD[3]. Il processore singerico è composto di due unità principali, l'unità processore singerico (SPU) e l'unità di accesso alla memoria il *memory flow controller* (MFC), come mostrato in figura 2.4.

L'unità del processore singerico (SPU) si occupa di controllo e esecuzione delle istruzioni, include al suo interno un register-file di 128 registri a 128-bit, una memoria unificata per dati e istruzioni (il *Local Store*, LS) di 256-KB, un unità di gestione delle istruzioni, un unità di load/store (che operano sul *Local Store*), due unità fixed-point, un unità floating-point e un interfaccia verso l'MFC (DMA e channels). La SPU implementa un nuovo *instruction-set* SIMD (chiamato *SPU Instruction Set Architecture*) che è specifico dell'architettura Cell Broadband Engine.

La presenza di 128 registri a 128 bit costituisce un ampio register-file in cui vengono memorizzati tutti i valori, siano essi numeri interi, floating-point singola o doppia precisione, scalari o altro, sempre nel register-file vengono memorizzati gli indirizzi di ritorno, i risultati delle comparazioni,etc. Come conseguenza di questo grande register-file abbiamo che non è necessario implementare costose tecniche hardware per migliorare le performance. I registri del PPU sono di due tipi:

- *General-Purpose-Register (GPRs)* in cui tutti i tipi di dato possono essere memorizzati, di questi GPRs ne sono presenti 128
- *Floating-Point Status and Control Register(FPSCR)*, il processore aggiorna il FPSCR ad ogni operazioni floating-point che viene eseguita, al fine di memorizzare le informazioni relative al risultato prodotto e alle eccezioni generate.

Ogni SPU è un processore indipendente dotato di proprio program counter e ottimizzato per eseguire SPE threads creati e gestiti dal PPE. L'SPU preleva le istruzioni dal

[3]In seguito nel capitolo verrà spiegato a fondo il termine SIMD, single-instruction multiple-data, per ora basti sapere che questo tipo di istruzioni opera eseguendo le stesse operazioni su più dati contemporaneamente. Il numero di dati su cui agisce dipende dal tipo di dato che stiamo utilizzando.

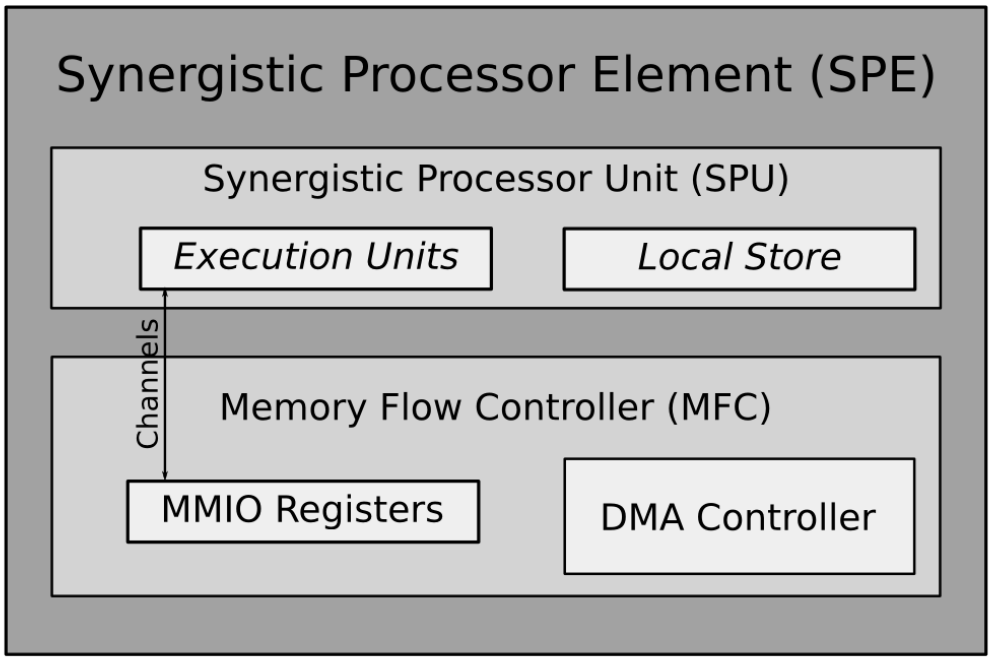



Figura 2.4: Schemattizzazione SPE

proprio *Local Store* in cui sono memorizzati anche i dati cui accede con l'utilizzo delle istruzioni di Load/Store.

La SPU ha due pipeline sulle quali è possibile schedulare due istruzioni per ottenere la simultanea esecuzione delle due, a seconda del tipo di istruzione questa viene schedulata su una pipe o sull'altra :

- pipeline 0 (even): vengono schedulate le istruzioni di Load/Store, Branch Hints, Branch resolution, interfacciamento ai canali di comunicazione e ai registri special-purpose. Sempre su questa pipeline vengono mandate in esecuzione le istruzioni di rotazione, shift e manipolazione dei bytes di un registro.

- pipeline 1 (odd): vengono schedulate le istruzioni di calcolo singola precisione, doppia precisione, intere.

L'unità MFC è composta di un controller DMA che si occupa dei trasferimenti DMA. Un qualsiasi programma che gira sul SPE, su PPE o su di un altro SPE utilizza i trasferimenti DMA per spostare dati o instruzioni tra i Local Store delle varie SPU o tra LS e Memoria Principale[4]. L'MFC interfaccia gli SPU sul Bus di interconnessione (EIB), si occupa di riservare bandae di sincronizzare le varie operazioni DMA tra i vari processori del sistema.

Per gestire i trasferimenti DMA l'unità MFC è dotata di una coda di comandi DMA. Il processo di richiesta di un trasferimento dati si svolge come segue: lo SPU invia all'MFC

[4] Con il termine Memoria Principale intendiamo qui indicare quello che in lingua inglese sui manuali di IBM viene definito come Main Storage.

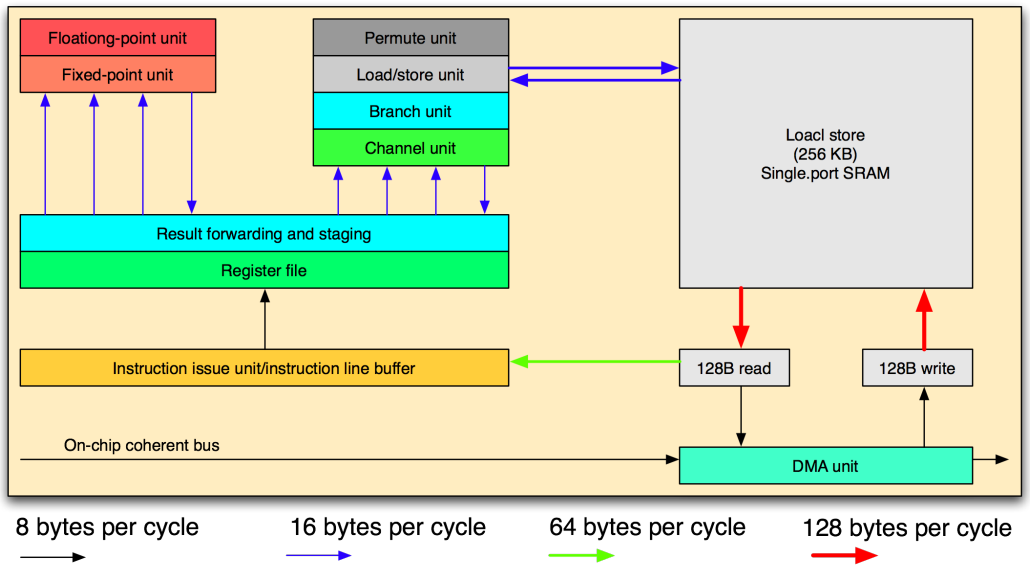



Figura 2.5: Schema SPE

una richiesta di trasferimento DMA e può proseguire nell'esecuzione del codice mentre l'unità MFC processa le richieste DMA presenti in coda in modo autonomo e asincrono. L'MFC è in grado di eseguire un insieme (una lista) di richieste DMA in modo del tutto autonomo (le liste di richieste DMA possono essere inviate all'unità MFC mediante l'uso di appositi comandi dma-list). L'indipendenza delle due unità, percui la SPU può eseguire codice mentre l'MFC gestisce i trasferimenti dati, consente la programmatore di creare software in grado di mascherare la latenza di accesso alla memoria.

La dimensione massima dei trasferimenti DMA è di 16-KB (16384-Bytes), le DMA-list possono essere inviate solo da un SPU al suo MFC. Possono essere accodate fino a 2048 richieste di 16384 Byte ciascuna. La traduzione degli indirizzi virtuali per ogni MFC è gestita dal sistema operativo (PPE-Side), gli attributi del sistema di memorizzazione sono organizzati come nell'Architettura PowerPC pagine di memoria e tabelle dei segmenti. Per applicazioni PPE privilegiate è possibile mappare gli indirizzi dei vari *Local Store* e alcune risorse MFC delle unità SPEs nello spazio di indirizzamento globale. Gli indirizzi mappati in memoria non sono coerenti.

L'unità MFC fornisce canali ed registri MMIO al fine di accodare e monitorare i comandi DMA, monitorare gli eventi degli SPUs, dare luogo a comunicazione interprocesso tramite il sistema di mailing e notifica dei segnali,accedere a risorse ausiliarie quali il registro decrementatore (timer) e altre funzioni.

Gli SPEs forniscono un ambiente di esecuzione deterministico in cui è possibile calcolare con precisione le performance che il codice può raggiungere. Le unità sinergiche non sono dotate di cache quindi i cache Miss non sono un fattore che influenza le performance. Le regole di scheduling delle istruzioni sulle pipeline sono semplici e consentono di determinare staticamente le performance del codice. Per come è stat progettata l'architettura le DMA possono accedere al *LocalStore* una volta ogni 8 cicli, mentre vengono fornite 17 istruzioni per volta dal branch target, così che l'impatto delle DMA sulle operazioni di Load e Store sia limitato.

### 2.1.3 EIB, Element Interconnect Bus

L'EIB è il principale BUS di comunicazione dell'architettura e collega insieme i vari elementi presenti "on-chip": Processore PPE, il controller della memoria (MIC), le otto unità SPEs, le due interfaccie esterne di I/O, in totale abbiamo 12 elementi connessi. L'EIB ha percorsi distinti per i traferimenti dati e per i comandi, ogni elemento connesso al BUS è collegato da un link punto a punto al concentratore degli indirizzi, quest'ultimo invia a tutti gli elementi i comandi nell'ordine in cui li ha ricevuti, riaggrega ed inoltra a tutti le risposte ai comandi DMA che costituiscono il segnali di inizio del trasferimento DMA per le unità interessate dai comandi. Il Bus di interconnessione è costituito di 4 anelli circolari unidirezionali, questi canali hanno una capacità di 16 Bytes ogni due cicli di clock del Processore Principale. I 4 anelli di interconnessione sono due in senso orario e due in senso antiorario. L'accesso al BUS viene gestito dall'*arbiter* che processa le richieste scegliendo quale dei due ring (orario o antiorario) deve soddisfare la richiesta di trasferimento dati, l'*arbiter* sceglie sempre il percorso più breve per il trasferimento dei dati assicurando che i dati non debbano viaggiare mai per più della metà della lunghezza di tutto il bus. Sempre l'*arbiter* si occupa di schedulare le richieste di trasferimento DMA per evitare che interferiscano con quelle in esecuzione, avendo come principale obiettivo la minimizzazione dei tempi di accesso alla memoria l'*arbiter* da priorità alla richieste di trasferimento dati provenienti dal memory controller (MIC).

Ogni anello permette un massimo di tre trasferimenti dati contemporanei, se il pattern di trasferimento dati lo consente potremmo avere che su ogni anello vengono eseguiti 3 trasferimenti in contemporanea, con 4 anelli a disposizione, ogni trasferimento è di 16 Bytes ogni due cicli di clock, abbiamo che la banda di picco teoria è $\frac{16\times12}{2} = 96\ Bytes$ per ciclo di clock, questo valore è una sovrastima della reale capacità di traferimento dati dell'EIB in quanto esiste un untià di management dell'EIB che impone alcuni vincoli (Arbitration Unit). (ved riferimento 27 della wiki)

Ogni unità collegata all'EIB puì simultaneamente inviare a ricevere 16B ogni ciclo del BUS, la massima capacità di trasferimento dati dell' intero EIB è vincolata dalla velocità con cui gli indirizzi vengono forniti alle varie unità che è di un indirizzo per ciclo di BUS. Da ogni indirizzo è possibile prelevare 128 Bytes con una richiesta DMA il che ci porta ad una banda teorica dell'EIB di 128B×1.6GHz = 204.8 GB/s. Oltre a questo va considerato anche il fatto che la memoria è organizzata in pagine e segmenti (ved architettura PowerPC). Nella pratica l'architettura consente ad ogni SPE una banda passante combinata di 25.6 GB/s (nel caso di una macchina 3.2 GHz) dato che i dati vengono prelevati dalla memoria è necessario tenere conto anche dela banda passante del controller di interfaccia alla memoria (MIC) che è dotato di due canali XDR che permettono -combinati- un flusso dati di 25.6 GB/s.

Nell'architettura è presente un BUS di sistema FlexIO per l'interfacciamento ad altri dispositivi (ad esempio ad un secondo Cell). Il FlexIO è organizzato in 12 linee ognuna delle quali è un collegamento punto a punto ad 8 bit unidirezionale. Cinque delle otto linee a disposizione sono linee di ingresso, mentre le rimanenti sette sono di output. In linea teoria abbiamo una banda di output pari a 36.4 GB/s mentre l'input è di 26 GB/s questo per una frequenza di 2.6 GHz,il FlexIO può assumere un valore indipendente dal clock del processore.

Possiamo affermare che la banda passante dell'EIB dipende da alcuni fattori: la di-

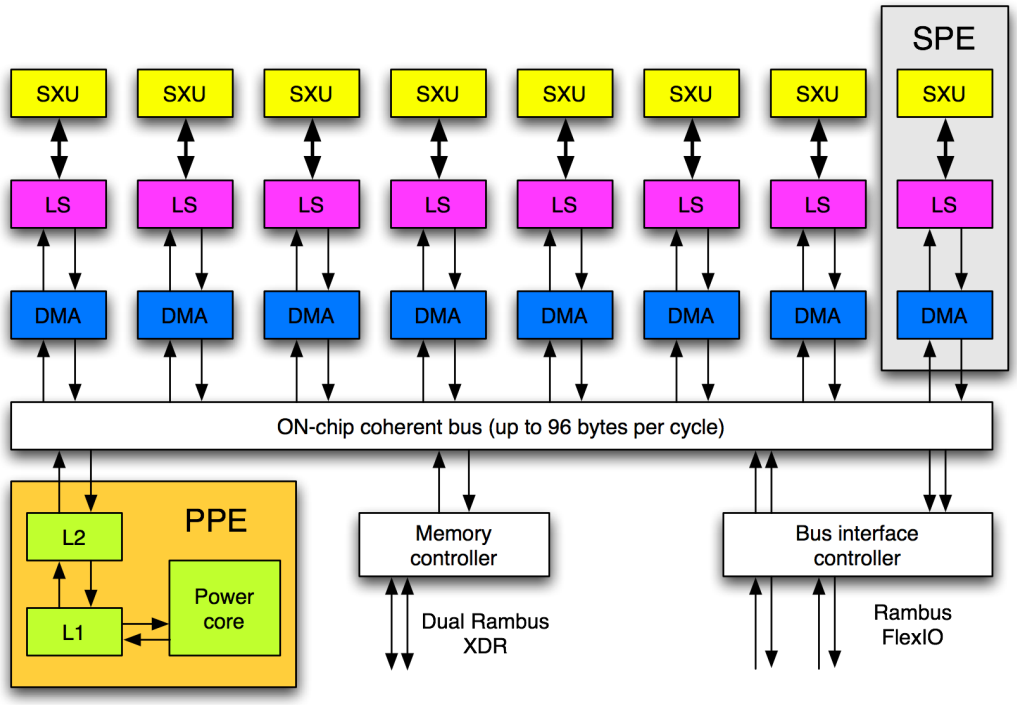



Figura 2.6: EIB, Element Interconnect Bus

stanza tra sorgente e destinazione, l'eventualità di nuove richieste di trasferimento che possono interferire con quelle attuali, il numero di chips presenti nel sistema, dal fatto che i trasferimenti avvengano *LocalStore* a *LocalStore* o *MainMemory* a *LocalStore*. Una banda passante non proprio al top delle performance può essere data dai seguenti fattoro:

- Le richieste devono accedere tutte alla stessa destinazione allo stesso tempo.
- Le richieste vengono schedulate tutte sugli stessi canali (in una sola direzione).
- Un elevato numero di richieste di dimensione ridotta (parti di una linea di cache).
- Tutte le richieste devono percorrere metà anello per raggiungere la destinazione, impedendo alle altre unità presneit sullo stesso ring di utilizzarlo.

(ved IEEE High performance interconnect)

Ogni SPU è collegato all'EIB tramite il *Memory Flow Controller (MFC)* ed è proprio quest'utilmo a fornire le funzionalità di trasferimento dati tra LocalStore e memoria principale. L'MFC ha un clock pari a quello dell'EIB ( la metà della frequenza del processore). Il Memory Flow Controller di ogni SPE gestisce e processa le code di comandi DMA provenienti dal SPU, dal PPE o da altri dispositivi. Per realizzare i trasferimenti DMA dal Main Storage al Local Store, il software in modalità privilegiata mappa i Local Stores delle varie unità SPE e gli MFCs come registri di Input/Output mappati in memoria (MMIO), il cui indirizzo viene mascherato come se fosse un indirizzo effettivo dello spazio di indirizzamento globale. Questa mappatura nello spazio di indirizzamento globale dei vari SPEs e dei loro MFCs permette al software su PPE di accedere loro e

quindi di controllare gli SPEs. Il software privilegiato che gira su PPE fornisce all'MFC le informazioni di traduzione degli indirizzi necessari per i trasferimenti DMA.

La SPU si interfaccia all'MFC tramite dei canali unidirezionali che si comportano come delle code FIFO[5] a capacità fissa. Dal lato SPU i canali figurano alcuni a sola lettura (read-only) altri sola scrittura (write-only). Alcuni di questi canali sono definiti bloccanti ciò comporta che una loro interrogazione da parte dell'unità SPU blocca l'esecuzione lato SPU fino a terminazione dell' operazione di interrogazione del canale. Ogni canale ha un contatore associato che indica il numero di elementi presenti sul canale.

Il memory flow controller riceve e processa richieste DMA inviategli dallo SPU (utilizzando i canali di interfacciamento) o dal PPE (Memory Mapped I/O registers). I comandi vengono accodati nell'unità MFC e l'SPU o il PPE possono continuare l'esecuzione dei codice (parallelamente al trasferimento dei dati) utilizzando le istruzioni di polling i istruzioni bloccanti per determinare quando il trasferimento DMA è terminato. L'autonoma esecuzione dei trasferimenti dati da parte dell'MFC consente una schedulazione ottimizzata delle richieste DMA per poter nascondere le latenza dovute all'accesso alla memoria principale.

L'MFC supporta trasferimenti dati allineati 1,2,4 e Bytes o multipli di 16, fino ad una dimensione massima di 16384 Bytes per trasferimento. Il massimo delle performance è raggiungibile quando entrambi gli indirizzi di trasferimento dati (sorgente e destinazione) sono allineati a 128 Bytes e i dati trasferiti sono un multiplo di 128. L'SPU può utilizzare i comandi DMA-list per schedulare una lista di richieste di trasferimento dati fomp ad un massimo di 2048 richieste in un singolo comando DMA. Una lista DMA in memoria è rappresentato da un array di sorgente/destinazione/lunghezza del trasferimento.

### Allineamento dati

I dati in memoria sono diposti in un array lineare indirizzato al Byte, il che significa che l'indirizzo di una locazione di memoria punta direttamente ad un singolo Byte. L'architettura gestisce le letture dalla memoria in un modo "lazy" per cui non è possibile caricare un solo Bytes ma vengono comunque caricati più elementi. Facciamo un esempio: Supponiamo di avere in memoria 4 word a 32 bit, indirizzate come mostrato in tabella 2.1.3, quello che vogliamo è caricare la word 2 con indirizzo 11111001, abbiamo un metodo di caricamento allineato a 16 B per cui vengono caricati sempre 16 Bytes ciò comporta che i dati vengono letti dall'indirizzo da noi specificato senza considerare gli ultimi 4 bit dell'indirizzo (per rappresentare 16 in binario sono necessari 4 bit).

| Word Number | Binary-Address |
|---|---|
| 0 | 11110001 |
| 1 | 11110101 |
| 2 | 11111001 |
| 3 | 11111101 |

Tabella 2.1: Allineamento degli indirizzi

[5]FIFO è l'acronimo di First-In First-Out, nel contesto delle code, indica che il primo elemento ad entrare è anche il primo elemento ad uscire possiamo immaginare che una coda FIFO sia un tubo in cui vengono fatte scorrere delle palline, la prima pallina ad entrare nel tubo è anche la prima ad uscire.

Quello che succede quando effettuiamo una richiesta di caricamento dati dalla memoria è che i vengono prelevate 4 Word nonostante noi ne avessimo richiesta solo una e le altre 3 vengono scartate con operazioni di rotazione e shift. Possiamo concludere che l'utilizzo di indirizzi non allineati comporta due cose:

I l'esecuzione di operazioni aggiuntive per scartare i dati cui non siamo interessati

II una perdita di performance (se avessimo i dati allineati non dovremmo eseguire operazioni inutili)

## 2.2 Comunicazione e sincronizzazione

Tra le cose più importanti nel meccanismo di comunicazione e di trasferimento dati vi è la sincronizzazione tra i vari SPEs, l'architettura mette a disposizione due modi possibili di sincronizzare le varie unità: i segnali e le mail. Ogni SPU puó leggere i canali Sig_Notify_1 e Sig_Notify_2 utilizzando le funzioni bloccanti SPU_RdSigNotify1 e SPU_RdSigNotify2. Il PPE o gli altri SPE possono scrivere i canali dei segnali utilizzando gli indirizzi mappati in memoria. Una particolarità dei segnali consiste nel fatto che permettono la scrittura come operazione logica di somma (OR), permettendo una semplice ed efficace comunicazione inter-processori.

Un secondo meccanismo di comunicazione consiste nelle mailboxes, che sono un canale di comunicazione a 32bit per PPE ed SPE. Ogni SPU una mailbox in ingresso con lettura bloccante (read-blocking) che può ospitare quattro entry (ogni entry come abbiamo detto è di 32 bit) ed ha due mailbox di uscita con scrittura bloccante ognuna delle quali può ospitare un solo messaggio, una delle due mailbox quando viene scritta dal PPE genera un interrupt per il PPE (interruptable mailboxes).(vedi manuale architettura mail boxes). Il PPE utilizza gli indirizzi dei canali delle mail mappati in memoria per scrivere e leggere queste, allo stesso modo un SPE può scrivere una mail ad un altro SPE scrivendo l'indirizzo della mailbox corrispondente mappato in memoria nello spazio di indirizzamento globale. L'invio di mail SPE to SPE consiste in un trasferimento DMA. A differenza del meccanismo di "signaling" le mailboxes sono indicate per una sincronizzazione "punto a punto" piuttosto che per il coordinamento globale.

Esiste anche supporto avanzato alla sincronizzazione globale noto come *operazioni atomiche* (che differiscono dalle operazioni DMA atomiche) che si occupa di garantire la lettura e la scrittura esclusiva di una *lock-line* in memoria. Le operazioni atomiche sono due:

a) la getllar, (*get lock line and reserve*) che legge il valore di una variabile di sincronizzazione dalla memoria e imposta una *reserve* su questa locazione. Se il PPE o un altro SPE modifica successivamente il valore della variabile di sincronizzazione l'SPE che aveva impostato la *reserve* la perde.

b) La putllc,( *put lock line conditional*) aggiorna il valore della variabile di sincronizzazione solamente se l'SPE che ha lanciato l'istruzione detiene la *reservation* sulla variabile. Se la putllc fallisce l'SPE deve rieseguire la getllar per riacquisire la *reservation* e riprovare ad aggiornare la variabile con una nuova putllc.

Le operazioni DMA atomiche appena descritte vengono gestite dall'unità operazioni atomiche dell'MFC. Questo tipo di operazioni può essere utilizzato per meccanismi di sincronizzazione particolarmente complessi.

Le risorse di I/O mappate in memoria (MMIO) hanno un ruolo fondamentale in molti i meccanismi di sincronizzazione, possiamo dividere le risorse mappate in memoria in quattro gruppi principali:

- *Local Storage*, tutte le SPUs hanno il LS mappato nello spazio di indirizzamento globale. L'accesso al LS non è sincronizzato con l'SPU ciò comporta che sia il programmatore a garantire che il programma SPU sia realizzato per acconsentire accessi asincroni ai dati qualora si vuole sfruttare questa possibilità.

- *Problem State memory map*, le risorse di questo tipo sono pensate per essere utilizzate direttamente dalle applicazioni, come ad esempio i trasferimenti DMA, il meccanismo di mailing e di signaling.

- *Privilege 1 memory map*, l'insieme di risorse disponibili a software privilegiato (come il sistema operativo) che consentono il controllo e la monitorizzazione dell'esecuzione delle SPU.

- *Privilege 2 memory map*, il sistema operativo impiega queste risorse per accedere alle risorse degli SPEs.

## 2.3 Dma e trasferimenti atomici

Le unità DMA del Memory Flow Controller presenti in ogni SPE gestiscono la maggior parte delle comunicazioni tra SPU e gli altri elementi del Cell ed eseguono i comandi DMA impartiti dall'SPU loro associata o da altri SPE o dal PPE. Il "punto di vista" dei trasferimenti DMA è sempre quello dell'SPE, così che una richiesta dati dalla memoria al Local Store sia una GET (prelievo) mentre un salvataggio in memoria principali è una richiesta di PUT (invio).

I trasferimenti DMA sono coerenti[6] nei confronti dell'ambiente di memorizzazione principale così che tutti i processori vedano gli stessi valori in memoria.

L'unità MFC gestisce in modo autonomo e "out-of-order" le richieste di traferimenti dati, qualora vi fosse la necessità di assicurare l'ordinamento delle richieste DMA il programmatore deve farlo esplicitamente utilizzando comandi speciali quali i *barrier* e *fence* che assicurano un ordinamento tra la richiesta DMA corrente e le altre presenti in coda. I comandi mfc *fence* stabiliscono che il comando non può essere eseguito fino a che tutti i comandi precedenti con lo stesso tag-group non sono terminati. I comandi mfc *barrier* stabiliscono che tutti i prossimi comandi con lo stesso tag-group non verranno eseguiti fino a a che tutti i comandi con il tag-group specificato non saranno terminati.

[6]La coerenza della memoria principale è un problema che coinvolge i sistemi multiprocessori. Nei sistemi singolo processore abbiamo un solo elemento che ha accesso alla memoria principale e i dati scritti e letti sono visti sempre da un solo punto di vista. In architetture multiprocessore la coerenza della memoria deve essere garantita a livello hardware da meccanismi di sincronizzazione, se un processore carica dei dati per elaborarli e risalvarli mentre un altro processore accede agli stessi dati per leggere i loro valori quali valori vedrà il secondo processore quelli aggiornati o quelli vecchi? L'implementazione della memory coherence stabilisce come deve comportarsi il meccanismo di sincronizzazione.

La traduzione degli indirizzi e il controllo degli accessi DMA viene gestita dall'unità MFC MMU (Memory management Unit) che utilizza le informazioni delle tabelle di pagina e segmentazione della memoria definite nell'architettura PowerPC. L'unità di gestinoe della memoria (MMU) è dotata di un buffer di memorizzazione degli ultimi indirizzi tradotti. Il controller Dma dell'MFC (DMAC) processa i comandi accodati nel controller MFC. Il Controller MFC ha due code di comandi:

- *MFC SPU command queue*, per i comandi inviati dall'unità SPU associata utilizzando i canali di interfaccia
- *MFC Proxy command queue*, per i comandi inviati dal PPE o da altri dispositivi mediante l'utilizzo dei registri MMIO.

Il manuale dell'architettura del Cell non dice qual'è la dimensione di queste code ma raccomanda i programmatori di non assumere una grandezza arbitraria sebbene un uso inefficiente delle richieste DMA può portare all'accodamento di nuove richieste anche una volta che le code sono piene comportando una diminuzione delle performance. Nel processore Cell la coda dei comandi DMA provenienti da SPU può ospitare fino a 16 entries, mentre la coda proxy ne può ospitare fino a 8.

Per meglio comprendere il funzionamento dei trasferimenti DMA viene ora illustrato un breve esempio in cui viene fatta partire una richiesta di trasferimento dati. Per accodare una richiesta di trasferimento dati l'unitá SPE deve eseguire questi passi:

1. scrivere l'indirizzo relativo al *LocalStore* sul canale MFC_LSA
2. scrivere l'indirizzo effettivo-parte alta (EAH) sul canale MFC_EAH
3. scrivere l'indirizzo effettivo-parte bassa (EAL) sul canale MFC_EAL
4. scrivere la grandezza del traferimento dati sul canale MFC_Size
5. scrivere il tag-ID della richiesta DMA sul canale MFC_TagID
6. scrivere il class ID e l'opcode del comando sul canale MFC_Cmd

In Figura 2.7 viene mostrato il sistema DMA.

Analizzando il processo di inoltro di una richiesta DMA dal punto di vista delle unità coinvolte possiamo individuare un insieme di step eseguiti in seuquenza descritti come segue:

1. l'SPU utilizza il canale di interfaccia per inviare la richiesta DMA sulla coda SPU MFC Command queue
2. Il DMAC seleziona una comando dalla coda per eseguirlo, in generale un comando viene prelevato seguendo i seguenti criteri: a) i comandi nella coda SPU hanno priotià rispetto ai comandi nella coda Proxy (provenienti dall'esterno), b) il DMAC alterna a comandi di put comandi di get e c) il comando da eseguire deve essere *ready* (non può essere in attesa della traduzione dell'indirizzo, o dipendere da un altro comando).

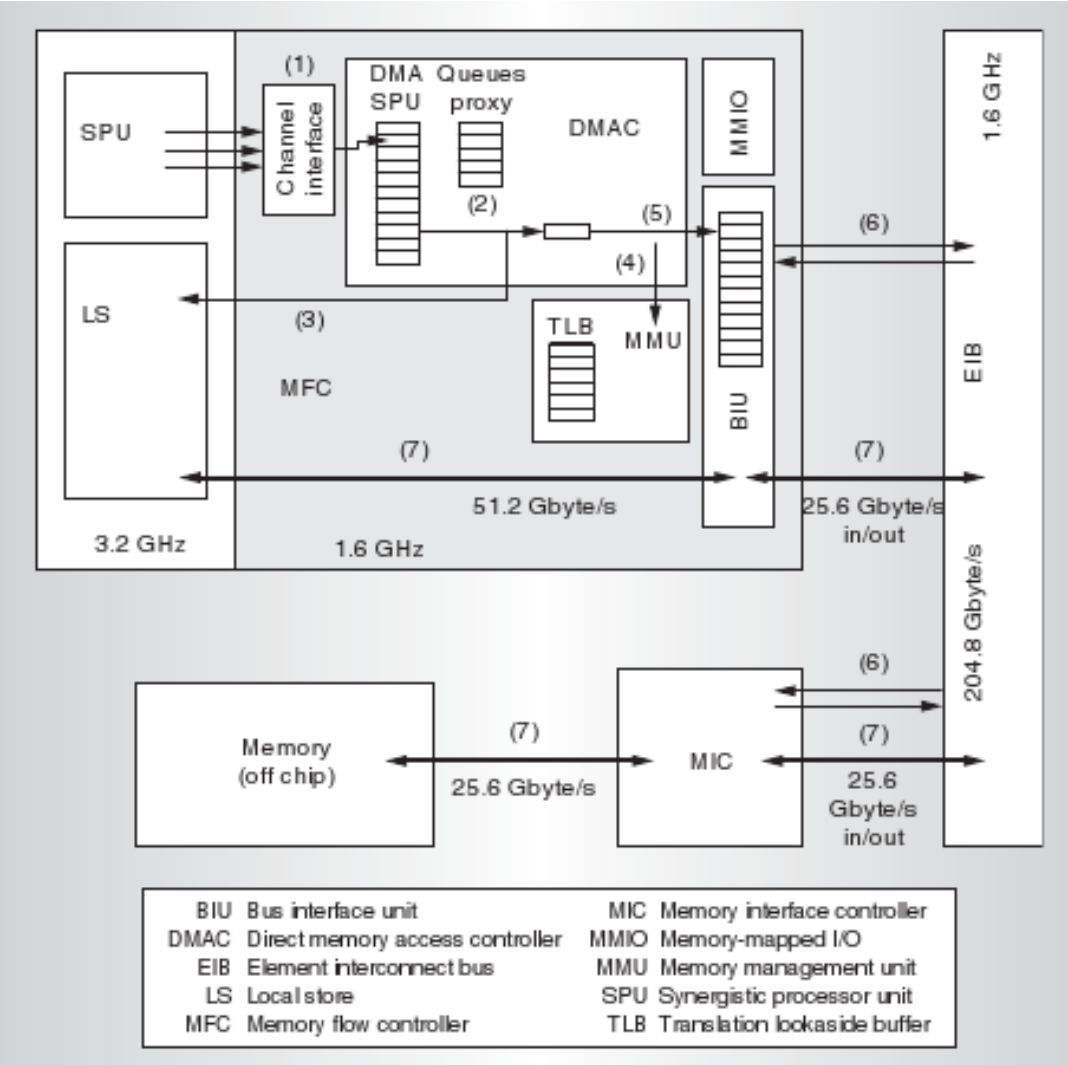



Figura 2.7: DMA Cell

3. Se il comando che viene richiesto è un DMA-list è necessario prelevare dalla memoria l'array con i dati delle richiesta quindi il DMAC inoltra la richiesta di lettura al *Local Store*. Una volta che gli elementi della lista sono disponibili DMAC aggiorna l'entry del comando DMA e lo riseleziona per essere riprocessato.

4. Se il comando DMA richiede la traduzinoe di indirizzi, il DMAC accoda la richiesta di traduzione all'MMU, una volta che la traduzione dell'indirizzo è disponibile nel TLB, l'elaborazione del comando DMA prosegue. Nel caso l'indirizzo non sia disponibile nel TLB (si ha un TLB Miss) la MMU esegue la traduzione dell'indirizzo utilizzando le "page tables" (tabelle di pagina di memoria) memorizzate in memoria principale e aggiorna il TLB. La DMA entry corrispondente al comando viene aggiornata e può proseguire nella fase di elaborazione.

5. Successivamente il comando DMA viene suddiviso in varie richieste di trasferimento dati di dimensione massima fino a 128 Bytes, successivamente il DMAC inoltra queste richieste di trasferimento dati alla BIU (Bus Interface Unit).

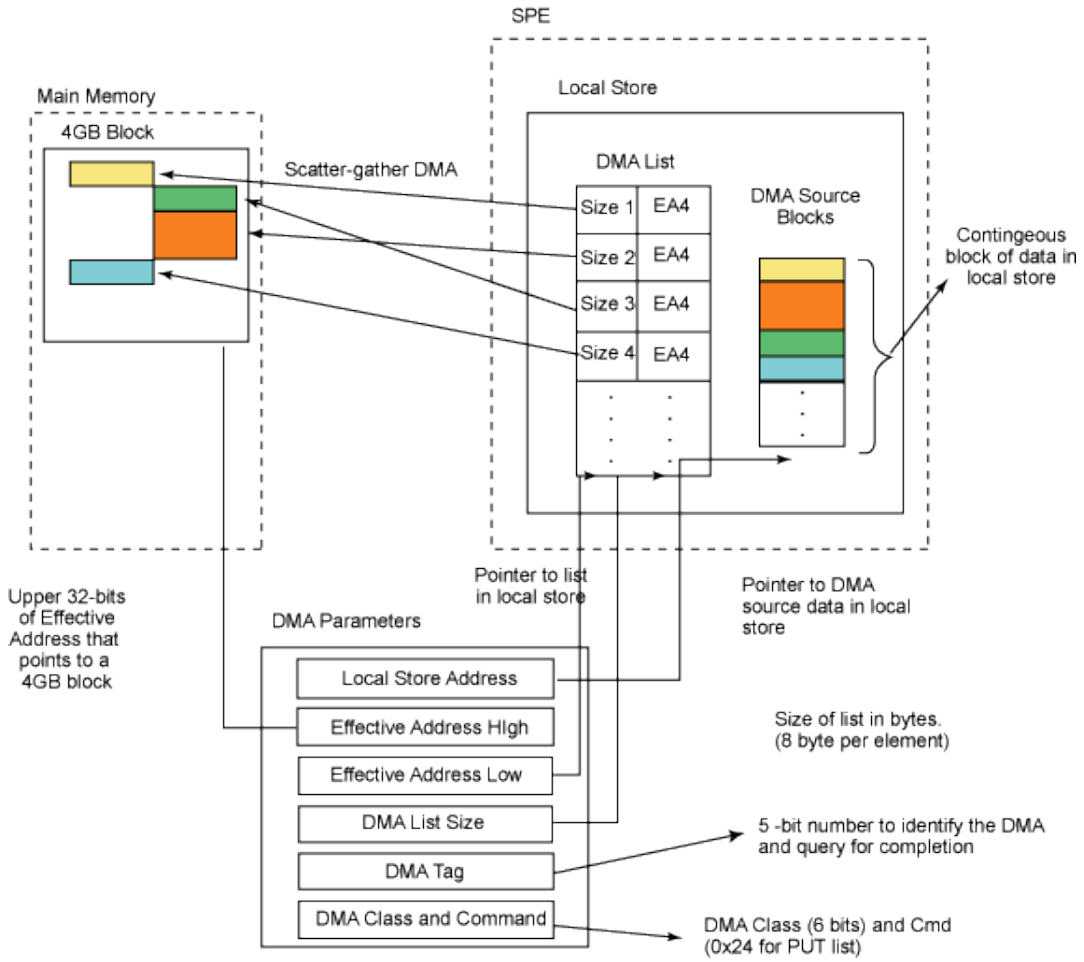



Figura 2.8: Schema DMA List

6. La BIU (Bus Interface Unit) seleziona i comandi DMA da eseguire dalla propria coda e li inoltra all'EIB. L'EIB ordina il comando rispetto alle richieste presenti e inoltra le richieste a tutti gli elementi connessi al BUS. In caso di trasferimenti che coinvolgono la Main Memory il MIC (Memory Interface Controller) conferma l'avvenuta ricezione della richiesta all'EIB che informa la BIU coinvolta della possibilità di iniziare il trasferimento.

7. L'unità di interfaccia al BUS (BIU) esegue le letture da *LocalStore* per i trasferimenti dati. L'EIB trasferisce i dati relativi alle richieste tra BIU e MIC. Il MIC esegue il trasferimento dati da e verso la memoria ("off-chip") esterna.

8. La suddivisione di un comando di trasferimento DMA in più richieste di traferimento bus che vengono accodate nelle rete di comunicazione. Il comando DMA rimane nella coda "MFC SPU command queue" fino a che tutte le richieste bus relative al comando non vengono terminate, anche se il DMAC prosegue nell'elaborazione di altri comandi DMA. Una volta che tutte le richieste bus relative ad un comando sono terminate il DMAC segnala il completamento dell'operazione DMA alla SPU e rimuove il comando dalla coda.

L'accodamento di una richiesta DMA richiede 10 Cicli, in caso di code libere, che

è il tempo necessario alla scrittura dei cinque canali coinvolti nella richiesta DMA che descrivono: indirizzo sorgente, indirizzo destinazione, grandezza del trasferimento DMA, tag-group DMA e il comando DMA. In Figura 2.8 viene mostrato uno schema di una richiesta DMA-list di cui vengono mostrati i vari elementi. Una volta che la richiesta è stata inoltrata al DMAC (DMA Controller) l'unità SPU può proseguire nell'esecuzione del programma.

## 2.4 Capacità di calcolo vettoriale

## 2.5 Istruzioni vettoriali calcolo vettroiale, etc.

## 2.6 Limitazioni del Cell nel calcolo in virgola mobile

# Capitolo 3

# LBE Cell Version

## 3.1 Introduzione

In questo capitolo verranno discusse i possibili miglioramenti applicabili al core di calcolo, verranno illustrati aspetti positivi e negativi di ogni modifica così come le implicazioni logiche di alcune modifiche dello schema dati.

Il core di calcolo è costituito da due funzioni -o *routine*- che vengono eseguite su tutta la matrice dati:

- *collide()*: implementazione dell'operatore di COLLISIONE
- *displace()*: implementazione dell'operatore di DIFFUSIONE

Di seguito analizzeremo gli aspetti teorici delle ottimizzazioni in relazione alle limitazioni e ai vincoli imposti dall'architettura e dal problema fisico. Da questo momento in poi, salvo dove espressamente indicato, con i termini LOAD e STORE di dati, faremo riferimento ad accessi alla memoria locale dell'unità di calcolo -*Local Store*- SPE.

## 3.2 Considerazioni sull'ottimizzazione della *collide*

La routine di Collide() opera sulle componenti di un punto, non accede a dati di altri punti, lo stesso dato viene impiegato più volte (*Località Temporale*[1]) e i dati sono tra loro vicini (*Località Spaziale* [2]). Necessitando di tutte le componenti di un punto per poter eseguire i calcoli, la Collide() deve averle disponibili nel Local Store. Caricarle ogni volta tramite DMA dalla Memoria Principale sarebbe troppo dispendioso.

Supponiamo quindi di avere i dati disponibili nel Local Store e di utilizzare lo schema dati originale, in cui ogni punto è costituito delle sue 9 componenti rappresentate da un array di 9 float. In memoria tale `stuttura` viene mappata su locazioni consecutive. Essendo la matrice dati definita come un array bidimensionale di $NX \times NY$ punti siamo certi che questi vengono allocati in locazioni di memoria contigue. Questa assunzione

[1]La *Località Temporale* ha luogo quando si accede ad un indirizzo $A$ ed è molto probabile che negli accessi successivi si richieda di nuovo il dato indirizzato da $A$.

[2]La *Località Spaziale* si ha quando si accede ad un indirizzo $A$, è molto probabile che gli accessi successivi siano da locazioni consecutive della memoria -*vicine*-. Gli accessi ad `Array` o a `Strutture` dati sono accessi vicini.

è la base del seguito della discussione, avere i punti contigui in memoria ci consente di caricare dalla MM al LS più dati con una singola richiesta DMA se invece avessimo i dati sparsi sarebbe necessario eseguire più richieste DMA da locazioni di memoria diverse e ciò risulterebbe poco efficace. Inoltre questo modo di organizzare i dati ci consente di determinare a *compile-time* a che distanza di trovano le componenti una dall'altra.

Per quanto detto sopra il POP è memorizzato come una sequenza di numeri float contigui tra loro, le componenti sono distanti tra loro un numero di bytes pari alla grandezza della rappresentazione in memoria di un float `sizeof( float )` ed ogni POP è a distanza $\#Componenti \times$ `sizeof( float )` [Bytes] dai propri vicini lungo l'asse $X$ (nel nostro caso un POP è $9 \times 4 = 36$ *Bytes*). In memoria avremo un POP ogni 36 Bytes come mostrato in figura 3.1.

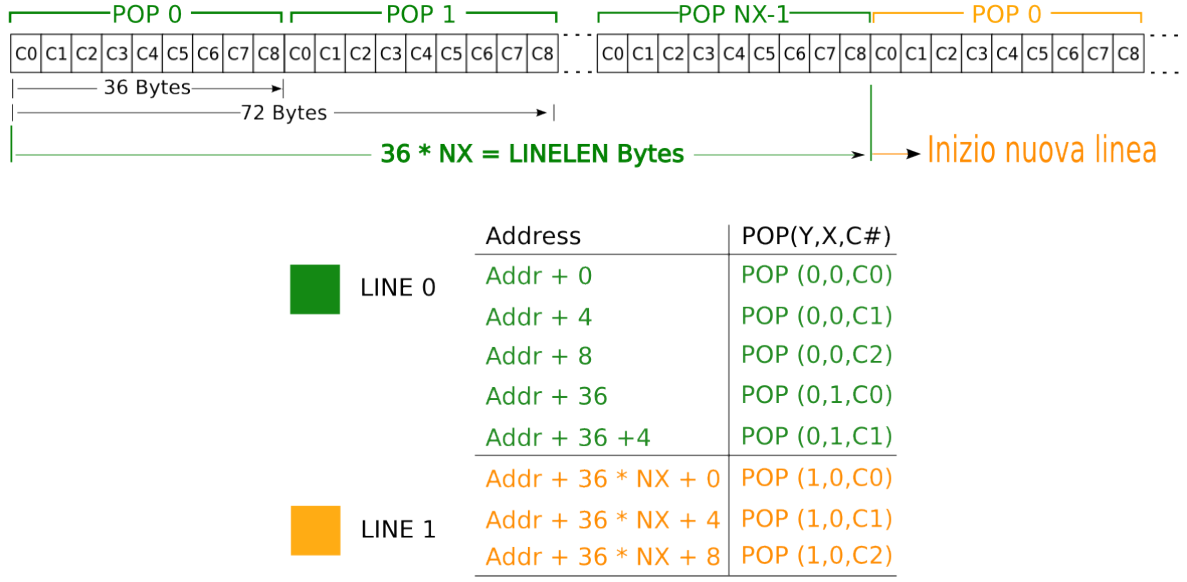



Figura 3.1: Rappresentazione in memoria della matrice punti

Il pattern di accesso ai dati della routine segue uno schema preciso, si devono caricare tutte le componenti di un punto, aggiornarle e devono essere salvate in memoria.

Esistono due metodi con cui possiamo accedere ai dati nel *Local Store*:

1. mediante l'utilizzo di un registro in cui memorizziamo l'indirizzo cui vogliamo accedere e che per ogni accesso dobbiamo aggiornare, con operazioni aritmetiche

```
lqx  rt, ra, rb
stqd rc, ra, rb
```

dove:

- `rt`: registro in cui viene caricato il dato dall'indirizzo di memoria calcolato come somma dei registri $ra + rb$

- `rc`: registro che contiene il dato da salvare in memoria all'indirizzo calcolato come somma dei registri $ra + rb$
- `ra,rb`: registri utilizzati per la creazione dell'indirizzo effettivo in memoria cui si deve accedere per caricare o salvare i dati

2. con l'utilizzo delle istruzioni *immediate*(es: `lqd`,`stqd`) queste istruzioni sono definite come segue:

```
lqd  rt, s14(ra)
stqd rc, s14(ra)
```

dove:

- `rt`: registro in cui viene caricato il dato dalla LOAD prelevandolo all'indirizzo in memoria calcolato come il valore del registro `ra` cui viene sommato la costante numerica `s14`
- `rc`: registro che contiene il dato da salvare all'indirizzo ottenuto sommando al valore del registro `ra` il valore della costante numerica `s14`
- `s14`: costante numerica definita come numero intero con segno compreso tra $[-8192; 8191]$

Data la semplicità del pattern di accesso ai dati, è inopportuno pensare di calcolare con istruzioni aritmetiche l'indirizzo sorgente appoggiandosi a registri, possiamo invece sfruttare la presenza delle istruzioni *immediate* per l'accesso ai dati nel *Local Store*.

Queste istruzioni non aggiungono istruzioni di calcolo per la creazione dell'indirizzo, che andrebbero a finire sulla pipeline 0, creando degli stalli ( prima di poter accedere alla memoria dobbiamo attendere di avere l'indirizzo disponibile, bisogna quindi attendere la terminazione dell'istruzione di calcolo) nell'utilizzo della pipeline e facendo operazioni inutili.

Unica limitazione data dalle istruzioni *immediate* del Cell è la necessità che i dati si trovino ad un offset non superiore a $\pm 2^{13}$ Bytes.

La Collide() è un'operazione che deve essere svolta su tutti i dati eseguendo sempre le stesse istruzioni per cui siamo di fronte a quello che viene chiamato parallelismo sui dati *Data Parallelism*.

Il *Data Parallelism* si verifica quando abbiamo un insieme di dati su cui dobbiamo eseguire le stesse istruzioni. Avendone la possibilità possiamo eseguire queste istruzioni invece che su di un solo dato su più dati contemporaneamente. Possiamo generalizzare dicendo che:*dato un insieme di N dati , su questi dati dobbiamo operare una stessa sequenza di M istruzioni, per un totale di $N \times M$ istruzioni supponendo le operazioni sui dati indipendenti le istruzioni sono eseguibili parallelamente.* Esistono sistemi in grado di elaborare più dati in parallelo organizzandoli in vettori lineari, riducendo così il numero di istruzioni da eseguira da $N \times M$ ad $\frac{N}{VectSize} \times M$, dove VectSize indica la grandezza del vettore in numero di elementi.

L'instruction set mette a disposizione istruzioni in grado di operare non su un singolo dato ma su più dati contemporaneamente, queste istruzioni (a.k.a. SIMD ovvero Single

Instruction Multiple Data) agiscono su dati definiti Vector che sono vettori grandi 128 bit che possono rappresentare i tipi di dati più comuni (es: int, unsigned int, float, etc.). Ogni vector può ospitare da 16 a 2 elementi a seconda del tipo, se vogliamo un vettore di byte essendo il vettore di 128 Bit possiamo ospitare 16 Bytes se immaginiamo invece di operare su numeri floating point in doppia precisione allora dobbiamo accontentarci di due elementi nel vettore.

Si osserva inoltre che questa funzione puo' essere eseguita parallelamente su più SPE senza che questo comporti particolari accorgimenti.

```
00 void collide()
01 {
02   int x, y, pp;
03   float u, v;
04   float usq, vsq, u2, v2;
05   float sumsq, sumsq2, rho;
06   float ui, vi, uv;
07   pop p_eq;
08
09
10   for (y=1; y<NY+1; y++) {
11    for (x=1; x<NX+1; x++) {
12
13      /* RHO ********************/
14      rho = m(p[y][x]);
15      u = vx(p[y][x]) / rho;
16      v = vy(p[y][x]) / rho;
17      ui = u / cs2;
18      vi = v / cs2;
19      usq = u * u;
20      vsq = v * v;
21      /* End RHO ****************/
22
23      /*UPDATE *****************/
24      sumsq  = (usq + vsq) / cs22;
25      sumsq2 = sumsq * (((float) 1.0) - cs2) / cs2;
26      u2 = usq / cssq;
27      v2 = vsq / cssq;
28      uv = ui * vi;
29
30      p_eq.p[0] = rho * rt0 * (((float) 1.0) - sumsq);
31      p_eq.p[1] = rho * rt1 * (((float) 1.0) - sumsq
                                  + u2 + ui);
32      p_eq.p[2] = rho * rt1 * (((float) 1.0) - sumsq
                                  + v2 + vi);
33      p_eq.p[3] = rho * rt1 * (((float) 1.0) - sumsq
                                  + u2 - ui);
34      p_eq.p[4] = rho * rt1 * (((float) 1.0) - sumsq
```

```
                                        + v2 - vi);
35        p_eq.p[5] = rho * rt2 * (((float) 1.0) + sumsq2
                                        + ui + vi + uv);
36        p_eq.p[6] = rho * rt2 * (((float) 1.0) + sumsq2
                                        - ui + vi - uv);
37        p_eq.p[7] = rho * rt2 * (((float) 1.0) + sumsq2
                                        - ui - vi + uv);
38        p_eq.p[8] = rho * rt2 * (((float) 1.0) + sumsq2
                                        + ui - vi - uv);
39
40        for (pp=0; pp<9; pp++)
41          p[y][x].p[pp] = p[y][x].p[pp] - INVTAU *
                            (p[y][x].p[pp] - p_eq.p[pp]);

42        /*End UPDATE ************/
43      }
44    }
45  }
```

Le istruzioni della collide sono caratterizzate da pochi accessi in memoria e molti calcoli in virgola mobile.

Nel codice possiamo identificare due gruppi principali di operazioni che chiameremo RHO e UPDATE:

- RHO : l'insieme delle istruzioni dalla riga 14 alla 20 esegue il calcolo delle variabili fisiche, di rho,u,v,usq,vsq,ui,vi
- UPDATE : l'insieme delle istruzioni dalla riga 24 alla 41 operazioni di aggiornamento delle componenti, calcolo dei termini u2,v2,uv, sumsq, sumsq2

I calcoli del gruppo RHO di istruzioni sono dispendiosi in termini di numero di operazioni e senza i valori prodotti da questi calcoli non possiano in alcun modo proseguire nell'esecuzione delle rimanenti istruzioni della routine perchè dipendono da quei valori.

E' bene chiarire che le dipendenze sui dati possono essere due tipi:

- *dipendenza diretta* data una istruzione $i$ questa produce un dato che viene utilizzato dall'istruzione $j$.
- *dipendenza indiretta* data una istruzione $j$ che dipende da $k$ e $k$ che a sua volta dipende da $i$, $j$ deve attendere la terminazione dell'istruzione $k$ e di $i$ per poter essere eseguita.

Gli stalli impediscono di sfruttare quello che è definito come l'ILP - Instruction Level Parallelism - . Il parallelismo a livello di istruzioni consiste nella presenza di istruzioni tra loro indipendenti che potremmo teoricamente eseguire in parallelo. Nel nostro caso possiamo sfruttare la pipeline e le istruzioni inserite in coda nella pipe vengono elaborate in parallelo.

Per mantenere pienamente occupata la pipeline è necessario sfruttare il più possibile il parallelismo delle istruzioni. Questo compito svolto dal compilatore implica la ricerca

di istruzioni tra loro indipendenti e lo scheduling delle stesse una dopo l'altra a distanza di un ciclo.

Def: "*il problema di riordinare, nel rispetto delle dipendenze, le istruzioni di un programma in modo da sfruttare le risorse (limitate) del sistema nel modo più efficiente e ridurre il tempo di esecuzione del programma è detto* **instruction scheduling**".

In generale quando si hanno due istruzioni dipendenti tra loro quello che si vuole fare è separarle di un numero di cicli pari alla occupazione -definita anche latenza dell'istruzione- di pipeline della istruzione sorgente (quella che produce il dato) facendo sì che in questo intervallo vengano eseguite altre istruzioni.

In questo modo quando l'istruzione che sfrutta la sorgente (il consumatore del dato) quando viene eseguita avrà il dato disponibile e non dovrà attendere. Potremmo agire direttamente sullo scheduling delle istruzioni scrivendo noi stessi il codice assembler, ma quello che si vuole è sfruttare le potenzialità e la versatilità di un linguaggio di più alto livello come il C cercando di sfruttare alcuni accorgimenti nella scrittura del codice in modo tale che il compilatore possa interpretare correttamente l'ordine di esecuzione delle istruzioni e possa applicare tutte le ottimizzazioni per un miglior scheduling e la piena occupazione della pipeline.

Esistono alcune tecniche che possiamo impiegare come l'unrolling dei loop interni e l'indirizzamento manuale, che verrà trattato in seguito. Al momento ci è sufficiente sapere che il nostro modo di scrivere codice puo' influire parecchio sulle performance, è necessario impiegare tutte le tecniche che si conoscono per migliorare il rendimento del compilatore.

### 3.2.1 Occupazione del Register File

L'architettura ci mette a disposizione un set di 128 registri a 128 bit, vogliamo ora quantificare il numero di punti che possiamo aggiornare mantenendo occupati il maggior numero di registri.

Per determinare l'occupazione del register file per POP è necessario capire dal codice quante variabili (registri virtuali) sono necessarie per il calcolo di un punto, i registri virtuali vengono poi mappati dal compilatore sui registri fisici.

In generale abbiamo un numero di registri fisici inferiore al numero di registri virtuali e il compilatore cerca di mappare nel modo migliore possibile le nostre variabili sui registri fisici, al fine di ottenere le migliori performance.

quello che vogliamo ottenere è la migliore occupazione dei registri fisici possibile al fine di risparmiare accessi alla memoria (LOAD / STORE) cioè di poter eseguire tutte le istruzioni direttamente sui valori presenti nei registri senza doverci appoggiare alla memoria[3].

Una prima stima dei registri virtuali necessari la possiamo ottenere contando le variabili impiegate:

- Global Vars: cs2, cs22, cssq,rt0,rt1,rt2 #06
- Temp Vars: 2*[C0...C8], u,v,usq,vsq,sumsq,sumsq2, u2,v2,ui,vi,uv; foreach POP #30

[3]È vero che al local store si accede in tempo costante ma avere i valori nei registri è sicuramente più performante.

Diciamo che per ogni POP abbiamo all'incirica 30 variabili occupate.
Sorgono ora alcune considerazioni sul modo di procedere:

- Possiamo pensare di eseguire le operazioni in sequenza, in questo modo occupiamo pochi registri ma questo comporta una scarsa parallelizzazione, infatti con pochi registri le dipendenze dati sono inevitabili, e questo ci riporta al problema della scarsa occupazione della pipeline

- Parallelizzazione delle operazioni: vogliamo che vengano impiegati tutti i registri possibili al fine di parallelizzare il più possibile le istruzioni, cercando di utilizzare più variabili per i valori temporanei, questo porta ad un miglioramento della velocità di esecuzione ( in quanto non dobbiamo fare più load e store), ma allo stesso tempo facendo richiesta di un numero maggiore di registri rispetto all'esecuzione sequenziale. Una volta che sono stati occupati tutti i registri disponibili bisogna riccorrere nuovamente a Load e Store, quello che si vuole è che i valori impiegati più spesso vengano mantenuti sui registri e una volta terminati i registri i valori ad andare in memoria devono esssere quelli utilizzati meno frequentemente.

- Utilizzando le intrinsics messe a disposione dall'SDK sono sicuro che le operazioni vengono mappate su una o più istruzioni ASM senza che venga eseguita una chiamata a funzioni[4], garantendoci un miglior controllo ma anche un maggior uso di registri, per spiegarne il motivo prendiamo ad esempio la riga 28

```
p_eq.p[8] = rho * rt2 * (((float) 1.0)
          + sumsq2 + ui - vi - uv);
```

mappandola sulle intrinsics questa diventa:

```
rho_rt2       = spu_mul ( rho , rt2 );

onepsumsq2    = spu_add ( onev , sumsq2 );
onepsumsq2pui = spu_add ( onepsumsq2 , ui );

tempB     = spu_add ( vi , uv );
tempB     = spu_sub ( onepsumsq2pui , tempB );
p_eq.p[8] = spu_mul ( rho_rt2 , tempB );
```

Come si vede per il calcolo del valore aggiornato ci si è dovuti appoggiare ad alcune variabili temporanee che richiedono di essere mappate su registri fisici una volta che il codice viene compilato. Vi sono alcune considerazioni che possiamo fare:

1. le variabili non sono vive (alive) tutte allo stesso momento e quindi non è necessario che siano mappate tutte su registri fisici allo stesso istante di esecuzione del programma. Quindi stiamo stimando un numero di registri superiore a quello effettivamente necessario.
2. Molti valori calcolati possono essere riutilizzati in più punti del codice, pensiamo ad esempio al valore ((float 1.0) + sumsq) o ((float 1.0) + sumsq2) il primo

[4]Si sente spesso parlare di funzioni INLINE o MACRO funzioni, in real

viene impiegato per aggiornare il valore di 5 componenti, mentre il secondo per l'aggiornamento di 4 componenti, questi valori non devono essere ricalcolati per ogni componente, ma possono essere mantenuti su registri e riutilizzati

Fatte queste considerazioni segue ora una stima dei registri virtuali necessari nel caso di utilizzo delle instrincs:

- $2 * 9$ componenti $= 18$
- 4 variabili temp
- $9 + 5$ cost fisiche (rho,rho*rt0,vsq,usq,vi,ui,u,v,uv,u2.v2)
- 8 somme intermedie (sumsq,one_m_sumsq,one_p_sumsq2)

Totale 44 variabili per ogni punto. Con 128 registri a 128 Bit ( 4 Float in un registro ) possiamo pensare di mappare su questi 122 / 44 $\times$ 4 = 2.9 $\times$ 4 punti (il 122 si ottiene sottraendo dal numero dei registri disponibili il numero di variabili globali).

Ora la stima non è realstica perché come detto in precedenza alcuni valori non necessitano di restare alive in entrambe le fasi RHO e UPDATE e si puo' migliorare sicuramente la mappatura sulle intrinsics delle operazioni di calcolo.

Questo valore numerico deve solo servirci da riferimento per sapere che nel caso peggiore possiamo permetterci di mappare circa $3 \times 4 = 12$ punti sui registri e perciò se fosse necessario potremmo pensare di eseguire il core di calcolo della collide() non su un singolo punto ma su 12 punti, in modo che le istruzioni che hanno data dependences possano essere schedulate per una piena occupazione delle pipeline.

## 3.3 Considerazionie sull'ottimizzazione della *displace*

La displace() è una funzione molta dipendenza tra le sue istruzioni. Principalmente si occupa di spostare valori da una locazione di memoria ad un'altra eseguendo una serie di accessi in lettura e scrittura. Per poter eseguire questa funzione in modo ottimale occorre che tutti i dati siano presenti nel Local Store, come per la collide(), con la differenza che in questo caso non si agisce localmente -sul punto- ma si deve accedere alle componenti dei punti vicini. L'aggiornamento di un punto consiste nel caricare il valore di una componente da una locazione di memoria e salvarla in un'altra (temporanea) il che ci indica che le istruzioni di aggiornamento delle componenti sono dipendenti tra loro a due a due, ma sono -nell'insieme- parallelizzabili, il problema è che le istruzioni di load e store occupano tutte una stessa pipeline e non possiamo pensare di schedularne più di una per ciclo.

Nel caso della simulazione D2Q9 (bidimensionale) ogni punto ha 9 componenti e 8 vicini, il pattern di accesso ai dati dipende dall'organizzazione in memoria che scegliamo per la matrice dati.

Se utilizziamo la matrice dati organizzata su più righe dove ogni punto è composto di 9 componenti e i punti sono tra loro contigui in memoria così come lo sono il punto

di vista logico, per poter aggiornare una riga dati necessitiamo di avere nel Local Store la riga superiore a quella attuale e la riga inferiore. Ignorando i casi particolari - come i bordi - per ogni riga che aggiorniamo dobbiamo avere a disposizione anche altre 2 righe che rappresentano i punti vicini a quelli che stiamo modificando.

La riga che aggiorniamo non possiamo salvarla subito in memoria (LS) perché è a sua volta il vicino superiore o inferiore di un'altra riga, è quindi necessario adottare qualche strategia di buffering temporaneo dei valori aggiornati prima di poterli salvare.

Possiamo immaginare due tipi di partizionamento dati:

- Parizionamento BIDIMENSIONALE: Ogni SPEs elabora una piccola porzione della matrice dati di dimensioni fisse memorizzabile per intero nel Local Store,possiamo calcolare il lato della matrice come

$$L_1 = \left\lfloor \sqrt{\frac{LS\_SIZE - CODE\_SIZE}{POP\_SIZE}} \right\rfloor \tag{3.1}$$

  In questo modo assegnamo ad ogni unità una quantità di dati che può essere interamente ospitata nel Local Store. Si deve però gestire correttamente la sincronizzazione degli elementi di bordo tra le partizioni vicine, per evitare problemi di inconsistenza dati.

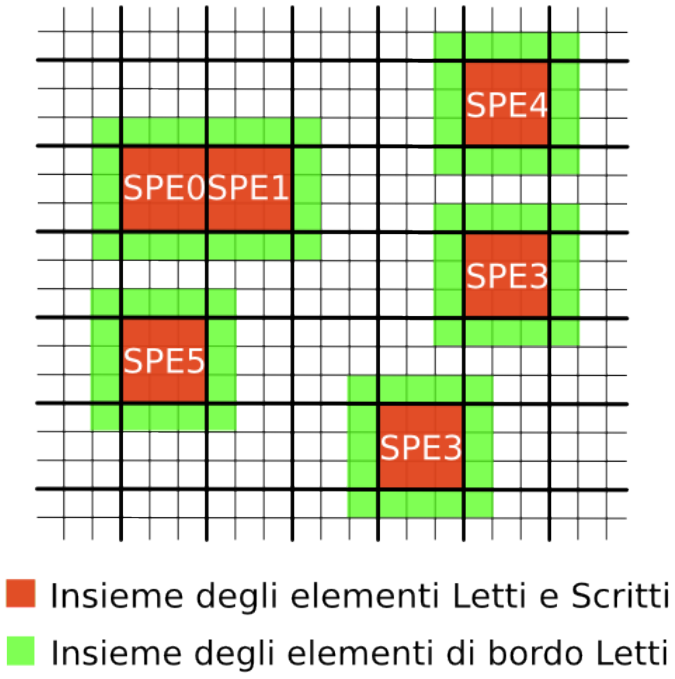



Figura 3.2: Schema di Partizionamento bidimensionale

Per semplificare il problema possiamo pensare che ogni SPEs carichi una matrice con un bordo come mostrato in figura 3.3 in questo modo i punti di bordo non devono essere salvati in memoria principale e possiamo accedere subito ai dati in lettura senza dover attendere conferme dai vicini. Volendo stimare il lato della

matrice che possiamo ospitare considerando i bordi, $L_2$:

$$L_2 = \left\lfloor \sqrt{\frac{LS\_SIZE - CODE\_SIZE}{POP\_SIZE}} - 2 \right\rfloor \tag{3.2}$$

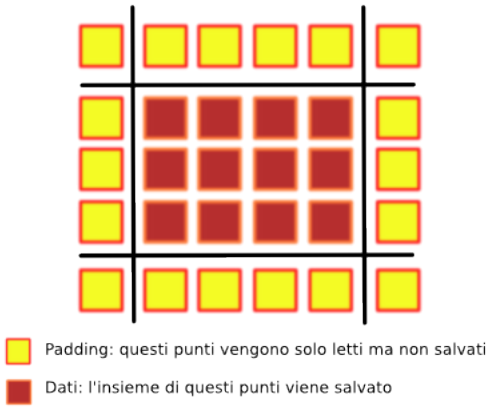



Figura 3.3: Rappresentazione matrice dati in Local Store con punti di padding

In ogni caso la sincronizzazione non è eliminabile, prima di scrivere i dati dobbiamo comunque assicurarci che i vicini abbiano già letto i valori. Possiamo cercare di ridurre ma non possiamo evitare la sincronizzazione tra uno SPE e i suoi vicini. Sapendo che questo introduce istruzioni e tempi di attesa dobbiamo cercare di mantenere al minimo l'influenza sulle performance di queste operazioni di sincronizzazione.

- Partizionamento MONODIMENSIONALE: Date $NY$ righe di lunghezza $NX$ elementi, ogni SPE deve elaborare un insieme $S$ di righe pari a

$$S = \frac{NY}{NUM\_SPE} \tag{3.3}$$

Così facendo ogni SPE ha due vicini, e necessitiamo di una sincronizzazione limitatamente alla prima e ultima riga. Questo approccio porta una considerazione importante: mentre nel caso del partizionamento quadrato indipendentemente dalla matrice dati ogni SPE deve comunicare con tutti i suoi vicini (anche nel caso migliore almeno per la scrittura dei valori aggiornati), quì la comunicazione è due a due e diminuisce la sua incidenza all'aumentare del numero di righe che ogni SPE deve elaborare ciò che il tempo speso per la sincronizzazione diventa sempre meno incidente rispetto al tempo di elaborazione dati.

Se consideriamo un POP come l'insieme di 9 numeri float, sappiamo che questi in memoria vengono mappati come $9 * 4 = 36$ Bytes Il Local Store è una memoria di 256KB condivisa tra codice e dati, supponendo di poterlo impiegare per intero per la memorizzazione dei dati avremo che in un singolo SPE possiamo memorizzare $(256 \times 1024)/36 = 7280$ $\lfloor\sqrt{7280}\rfloor = 85$ POPs. Una matrice di circa $85 \times 85$ punti non è molto grande.

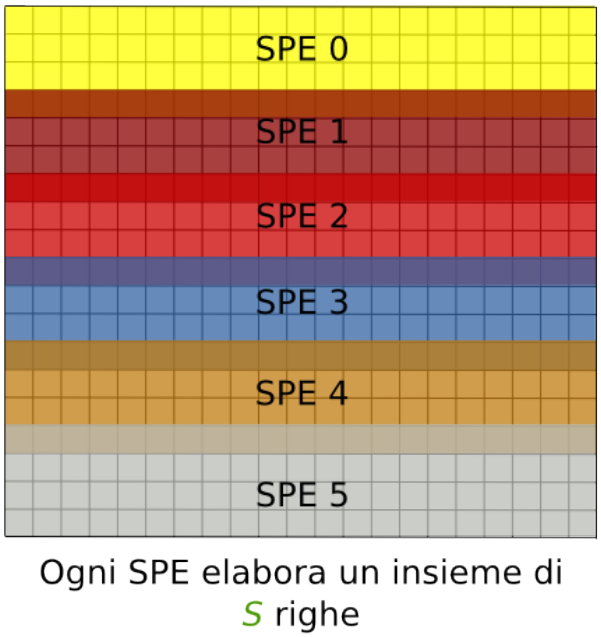



Figura 3.4: Schema di Partizionamento monodimensionale

Supponendo di poter dividere la matrice dati in N porzioni quadrate di dimensione $85 * 85$ punti, non considerando le eventuali sincronizzazioni con gli elementi vicini per i punti di bordo, avremo che ogni SPE deve caricare dalla memoria $85 \times 85 \times 36$ Bytes = 260100 Bytes che si traduce in 15 richieste DMA da 16384 Bytes (ved Cell Architettura) e una da 13430.

La matrice deve essere anche salvata il che implica che queste richieste vengono eseguite 2 volte per un totale di trasferimento dati pari a $260100 \times 2 = 520200\ Bytes = 508\ KB$ ad ogni STEP. Accedere alla memoria, elaborare l'insieme completo di dati e salvarlo in memoria ad ogni STEP dell'algoritmo puo' non essere il modo più efficiente di agire. Quello che si vuole ottenere è la sovrapposizione dei tempi di accesso alla memoria (Load + Store) con i tempi di calcolo.

Possiamo utilizzare il partizionamento rettangolare accennato prima a cui affianchiamo un meccanismo di buffering che ci consenta di sovrapporre il caricamento di un intera riga al calcolo di un'altra, il caso migliore ci consentirebbe di mascherare completamente il tempo di caricamento dalla memoria, mantenendo sempre occupato lo SPEs nel calcolo di valori aggiornati.
Possiamo cercare di stimare il numero di operazioni necessarie per mascherare il trasferimento di una riga per fare ciò dobbiamo prendere in considerazione la Bandwidth disponibile e il tempo di aggiornamento di un intera riga.

```
01 void displace()
02 {
03   int x, y;
04   pop buffer[2][NX+2];
05   int xm, xp, ym, yp;
06   int plane = 0;
```

```
07
08    for (y=1; y<NY+1; y++) {
09      ym = y-1;
10      yp = y+1;
11      for (x=1; x<NX+1; x++) {
12        xm = x-1;
13        xp = x+1;
14
15        buffer[plane][x].p[0] = p[y ][x ].p[0];
16        buffer[plane][x].p[1] = p[y ][xm].p[1];
17        buffer[plane][x].p[3] = p[y ][xp].p[3];
18        buffer[plane][x].p[2] = p[ym][x ].p[2];
19        buffer[plane][x].p[5] = p[ym][xm].p[5];
20        buffer[plane][x].p[6] = p[ym][xp].p[6];
21        buffer[plane][x].p[7] = p[yp][xp].p[7];
22        buffer[plane][x].p[4] = p[yp][x ].p[4];
23        buffer[plane][x].p[8] = p[yp][xm].p[8];
24      }
25
26      plane = (plane == 0 ? 1 : 0);
27
28      for (x=1; x<NX+1; x++)
29        p[y-1][x] = buffer[plane][x];
30    }
31
32    plane = (plane == 0 ? 1 : 0);
33
34    y = NY+1;
35    for (x=1; x<NX+1; x++)
36      p[y-1][x] = buffer[plane][x];
37  }
```

La funzione è composta da istruzioni di assegnamento, cioé una serie di Load e Store in memoria. Se osserviamo come vengono eseguite le letture si nota che il pattern di accesso ai dati è più articolato di quello della collide(), in particolare qui accediamo sempre a POP *diversi* e componenti *diverse*.

Supponendo i punti contigui in memoria e organizzati per righe come propone il modello fisico, avremo che gli indirizzi cui accediamo si collocano a distanze calcolabili semplicemente; per accedere ad un punto della riga precedente (Ym) corrispondente al punto attuale dobbiamo sottrarre dall'indirizzo corrente la lunghezza in bytes della riga mentre per accedere ad una componente precisa di quel punto è necessario sommare all'indirizzo appena calcolato $\#C * N - bytes$ dove *N-Bytes* è il numero di Bytes che occupa una componente -ovvero un numero float- in memoria.

Possiamo ricavare dal codice una tabella dei valori di offset utilizzati :

- LINELEN = Lunghezza in Bytes della linea
- POPSIZE = Lunghezza in Bytes di un POP

- CS = Lunghezza in Bytes di una Componente ( sizeof(float))
- Supponiamo di avere POPSIZE = 36 Bytes, CS = 4 Bytes e con linee di 128 punti, con
- LINELEN = 128 * 36 = 4608 Bytes

| Punto | OFFSET | POP36BL128 |
|---|---|---|
| P(ym,xm)[5] | -LINELEN - POPSIZE + 5*CS | -4608 - 36 + 20 = -4624 |
| P(ym,x )[2] | -LINELEN + 0 + 2*CS | -4608 + 0 + 8 = -4600 |
| P(ym,xp)[6] | -LINELEN + POPSIZE + 6*CS | -4608 + 36 + 24 = -4548 |
| P(y ,xm)[1] | -POPSIZE + 0 + 1*CS | -36 + 0 + 4 = -32 |
| P(y ,x )[0] | 0 + 0 + 0 | + 0 + 0 + 0 = 0 |
| P(y ,xp)[3] | +POPSIZE + 0 + 3*CS | +36 + 0 + 12 = +48 |
| P(yp,xm)[8] | +LINELEN - POPSIZE + 8*CS | +4608 - 36 + 32 = +4604 |
| P(yp,x )[4] | +LINELEN + 0 + 4*CS | +4608 + 0 + 16 = +4624 |
| P(yp,xp)[7] | +LINELEN + POPSIZE + 7*CS | +4608 + 36 + 28 = +4672 |

Tabella 3.1: Tabella Offset accesso alla memoria per *displace()*

Per ogni punto che aggiorniamo vogliamo utilizzre gli offset per l'accesso in memoria, come per il codice della collide è necessario sfruttare il più possibile le istruzioni messe a disposizione dall'instruction set per ottimizzare gli accesi, come per la collide() vogliamo sfruttare le istruzioni di load immediate con gli offset della tabella 3.1. Sfruttando le immediate siamo sicuri di non aggiungere calcoli aritmetici che andrebbero a finire sulle pipeline0.

Introdurre le istruzioni vettoriali nel codice della displace() non è "*semplice*" come per la collide() quì alcuni fattori come gli accessi sparsi in memoria giocano un ruolo chiave. La collide() agiva sul punto e non dipendeva in alcun modo dai dati dei vicini il che la rende una funzione perfetta per l'utilizzo delle istruzioni vettoriali, qui invece abbiamo bisogno di dati lontanti, in particolare i dati vengono caricati da una locazione e salvati in un'altra, è necessario fare alcune considerazioni sulle possibili rappresentazioni dei dati in memoria prima di proseguire oltre.
Il POP come abbiamo definito precedentemente è un array di nove numeri in virgola mobile a singola precisione -float- . Le istruzioni vettoriali operano su variabili - *vector* - a 128 Bit ovvero 4 dati float. Possiamo scegliere di ridefinire il POP come POPVECTOR in 2 modi diversi :

- POPVECTOR come insieme di 3 vector float, il 3 è dato dal fatto che abbiamo 9 componenti e ogni Vector contiene 4 elementi per rappresentare 9 elementi con i vector dobbiamo quindi utilizzarne 3, con uno spreco di 3 elementi. $3*4 = 12-9 = 3$ Elementi in esubero Questa rappresentazione è pero' inefficiente per tre motivi:

  1. Il POPVECTOR così definito non è efficiente per la funzione collide() inquanto richiederebbe nuove istruzioni per l'estrazione delle componenti dal vector e per il repacking.

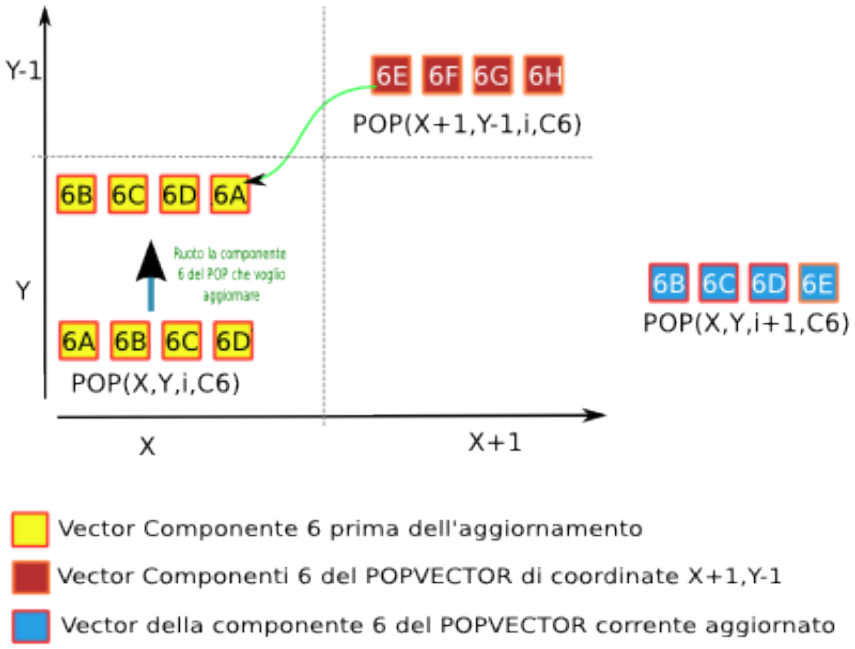



Figura 3.5: Aggiornamento della componente 6 di un POPVECTOR

2. La displace() prende le componenti aggiornate da vari punti diversi e operare su vector per estrarre un singolo elemento richiederebbe un numero maggiore di istruzioni.
3. Stiamo sprecando 3 elementi per ogni POP, su matrici piccole questo rappresenta un piccolo valore ma $3 : 12 = x : 100$ in percentuale stiamo sprecando il 25% di spazio in memoria.

In conclusione definendo il POPVECTOR in questo modo risulterebbe troppo inefficiente, finirebbe per richiedere un numero di istruzioni tale da essere comparabile al codice non vettorizzato.

- POPVECTOR come insieme di 9 vector float, raggruppiamo insieme 4 punti tra loro consecutivi. Così facendo non abbiamo elementi di padding o inutilizzati e possiamo sfruttare le istruzioni vettoriali sia nella collide() che nella displace(). Sia la collide() che la displace() ottengono così un fattore di guadagno importante, ma nella displace esistono alcuni casi particolari: prendiamo la riga 20, per aggiornare la componente 6 del POP corrente la carichiamo dal pop della riga precedente ma in posizione x+1 (xp). Questo significa che la componente 6 si muove verso sinistra, pensando alla rappresentazione in memoria dei POPVECTOR appena definiti significa che il primo elemento del sesto vector del POPVECTOR di coordinate x+1,y-1 deve essere estratto, ed inserito nell'ultimo elemento del POPVECTOR del punto corrente che a sua volta deve aver già ruotato i suo elementi di una posizione verso SX. In Figura 3.5 viene mostrata una schematizzazione di quello che sarebbe questa operazione.

  Il codice corrispondente a questa operazione sarebbe qualcosa di moltro simile al seguente, supponendo di avere nella variabile comp6ymxp il vector float corrispon-

dente alla componente 6 del punto da cui preleviamo il nuovo valore e di avere nella var comp6b il vector float della componente 6 del POPVECTOR corrente.

```
buffer[plane][x ].p[6] =
          spu_insert
          (
            // estrazione di comp6ymxp[0]
            spu_extract ( comp6ymxp, 0 ),
            // rotazione del vettore di 4 Bytes
            spu_rlqwbyte ( comp6b, 4 )   ,
            3
          );

buffer[plane][x].p[6] = p[ym][xp].p[6];
```

Questa operazione ovviamente deve essere ripetuta per ogni POPVECTOR ovvero NX/4 volte.

È importante sottolineare come una diversa disposizione dei dati in memoria implica un diverso modo di risolvere il problema. Cambiando lo schema dati possiamo rendere più semplici alcune operazioni ma contemporaneamente potremmo incappare nell'impossibilità di eseguirne altre. Pensiamo per esempio ad un organizzazione dei punti in memoria in cui mettiamo prima tutte le componenti 0 di tutti i punti, poi tutte le componenti 1, e cosi' via per le restanti, la displace sarebbe avvantaggiata da un simile modo di organizzare i dati, potrebbe fare prima l'aggiornamento delle componenti 0 di tutti i punti, poi quello di tutte le componenti 1 e così per le altre. Questo però è del tutto inefficiente se pensiamo alla funzione collide(), deve accedere a tutte le componenti di un punto ciò implica che dovremmo caricare 9 valori da indirizzi sparsi in memoria principale (questo influisce sulla velocità dei trasferimenti DMA) con evidenti ritardi.

## 3.4 Sovrapposizione delle due

Dopo aver analizzato gli aspetti principali delle funzioni di calcolo cerchiamo ora di capire quanto si può ottenere dalla sovrapposizione delle due funzioni.

Da notare che entrambe le routine fanno uso di entrambe le pipe, ma con proporzioni diverse. La fusione delle due funzioni ci dovrebbe portare nel migliore dei casi al mascheramento completo del tempo di esecuzione della più breve delle due nel tempo della più lunga.

La *collide()* esegue poche load (*pipeline 1*) e un certo numero di operazioni floating point (*pipeline 0*), la *displace()* invece esegue una serie di accessi in memoria (*pipeline 1*) e qualche operazione per il calcolo degli indirizzi (*pipeline 0*). Sovrapponendole avremmo effettivamente la piena occupazione delle pipeline o ci sono altri fattori che porterebbero ad un miglioramento delle performance?

Vi sono alcune considerazioni da fare:

- unire semplicemente il codice in un unica funzione può essere un'idea ma possiamo sicuramente fare di meglio: pensiamo alle componenti che carichiamo per la

displace, se non modifichiamo in qualche modo il codice, le istruzioni della collide ricaricheranno gli stessi dati dalla memoria, così per ogni componente non ho una load e una store ma 2 load e 2 store. Sappiamo che la funzione di displace deve caricare le componenti di un punto dai vicini per aggiornare quelle del punto locale, sappiamo inoltre che la displace agisce sulle componenti locali del punto, possiamo immaginare che una volta caricati i nuovi valori delle componenti dai vicini, su questi stessi valori possiamo eseguire la routine di collide e solo una volta terminate queste istruzioni procedere al salvataggio in memoria.

- possiamo pensare di caricare i dati seguendo lo schema della displace, una volta che tutte le componenti sono caricate nelle variabili temporanee su queste eseguo le operazioni delle collide, terminato l'insieme delle istruzioni della collide() possiamo procedere al saltavaggio in memoria. Da sottolineare il fatto che prima di poter eseguire le operazioni delle collide le istruzioni di caricamento devono essere terminate e solo una volta terminate le istruzioni di calcolo si può procedere al salvataggio dei dati in memoria il che significa che abbiamo nuovamente una dipendenza dati.

- unendo istruzioni su **pipeline 0** e istruzioni sulla **pipeline 1** possiamo beneficiare del Dual-Issue [5]

Il modo migliore di procedere è un'ottimizzazione per gradi in cui prima ci concentriamo sul codice del core di calcolo al fine di renderlo il più veloce e performance possibile, poi si mettono in atto le tecniche che conosciamo per limitare la dipendenza sui dati.

In questo senso un efficace schema di Loop Unrolling potrebbe garantirci un incastro perfetto tra la istruzioni. Come abbiamo detto nella sezione 3.2 la presenza di stalli implica l'impossibilità di sfruttare appieno la pipeline schedulando le istruzioni una dietro l'altra.

Per ottenere un buon risultato è necessario rifarsi ad un meccanismo di unrolling che ci consenta di eseguire un insieme di istruzioni su un POP mentre l'altro insieme di istruzioni viene eseguito su di un altro. Pensiamo ad esempio ad un meccanismo di *unrolling* di due punti, come mostrato in Figura 3.6. Distaccando la fase di calcolo da quella di caricamento ci assicuriamo del fatto che quando avremo bisogno dei dati questi saranno già disponibili perchè le istruzioni di Load saranno terminate.

Possiamo pensare ad un *unrolling* più spinto in cui elaboriamo non più due punti ma quattro punti, consentendoci di separare ulteriormente le istruzioni con dipendenza dati. Volendo fare attenzione al numero di variabili di cui ci serviamo, rischiamo di incorrere in quelle che vengono definite *false dipendenze* . In Figura 3.7 ne è mostrato un esempio.

Di dipendenze ne esistono di tre tipi:

- *Data dependences* questo tipo di dipendenza si verifica quando esiste un'effettiva relazione tra due istruzioni, una produce il dato mentre l'altra lo consuma. In questa tipologia rientrano anche le dipendenze dirette, indirette spiegate nella sezione

[5] Il Dual-issue si verifica quando siamo in presenza di due istruzioni di cui la prima può essere eseguita sulla pipeline 0 e la secondo sulla pipeline 1, consiste nella contemporanea esecuzione di due istruzioni sulle due diverse pipeline, visto come sono composte le funzioni del core di calcolo, supponendo di poter mantere indipendenza dati tra le istruzioni, dovremmo poter sfruttare appieno questo meccanismo.

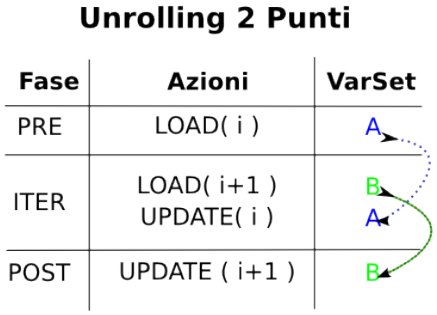



Figura 3.6: Loop Unrolling di due punti

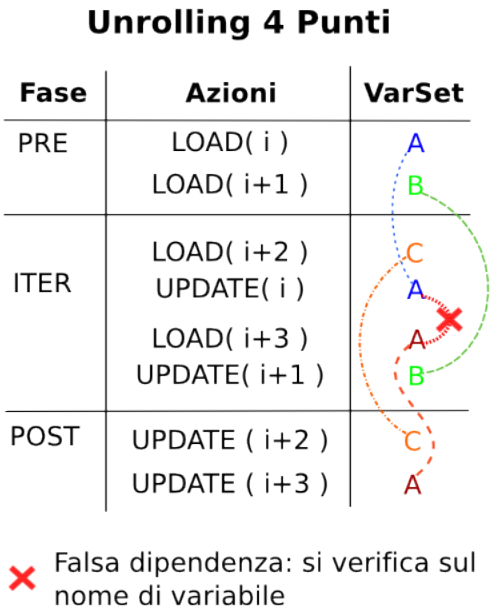



Figura 3.7: Loop Unrolling quattro punti

3.2. Un ulteriore classificazione di queste dipendenze viene dalla loro facilità di individuazione.

- *esplicite* quando i dati passano da un'istruzione ad un altra attraverso i registri
- *ambigue* quando lo scambio dati avviene mediante memoria oppure quando si verificano dei *branch* nel codice

L'ordine di esecuzione delle istruzioni con relazioni di dipendenza deve essere rispettato, bisogna quindi fare attenzione che questo non comporti un peggioramento dell'ILP.

- *Name Dependences* si verifica quando due istruzioni utilizzina una risorsa (sia essa un registro o una locazione di memoria), detta *name*, ma non si verifica trasferimento di dati o informazioni. Le *Name Dependences* differiscono dalle *Data dependences* in quanto non vi è passaggio di informazioni tra le istruzioni, possiamo quindi di-

re che le *Name Dependences* non sono vere e proprie dipendenze e possono essere eliminate rinominando le risorse, per questo motivo vengono anche chiamate *false dipendenze*.

- Le *Control dependences* definiscono la dipendenza di un'istruzione $i$ nei confronti di una istruzione di *branch*. Si verfificano quando abbiamo del codice con delle istruzioni sottoposte ad esecuzione condizionata. Pensiamo ad esempio ai costrutti dei *loop* o ai costrutti di selezione.

## 3.5 Performance attesa

### 3.5.1 Core di calcolo

Nella sezione precedente ci siamo soffermati sull'analisi delle ottimizzazioni possibili delle funzioni di calcolo. Per ottenere il meglio dall'architettura è necessario sfruttare appieno l'insieme delle sue potenzialitè quindi il calcolo vettoriale con le istruzioni SIMD, di cui abbiamo già dibattuto l'applicabilità alle routine, l'impiego del *Dual-Issue* e infine la parallelizzazione del lavoro sulle varie unità SPE.Quelle che seguono sono considerazioni sulle performance che possiamo aspettarci avendo noti alcuni aspetti del codice e dell'architettura.

Le funzioni di *collide()* e *displace()* occupano **pipeline** diverse, sono quindi unificabili in una funzione che opera su tutti i dati. Come detto nella sezione precedente possiamo cercare di sfruttare la meglio le loro caratteristiche in fase di *merging* passando alle istruzioni delle collide() gli elementi caricati su variabili dalle istruzioni della displace(). Così facendo elimineremmo una serie di *load* e *store* ma introdurremmo una dipendenza dati, per eliminare questa dipendenza possiamo impiegare l'*unrolling*.

La fusione delle funzioni dovrebbe consentire un pieno utilizzo del *Dual-Issue*, una volta che sono state introdotte le istruzioni vettoriali dovremmo poter vedere una mascheramento completo delle istruzioni dell'operatore di DIFFUSIONE nelle istruzioni dell'operatore di COLLISIONE.

### 3.5.2 Memoria

Ogni POP occupa $36 Bytes$ nella sua versione originale e $36 \times 4 = 144 Bytes$ nella versione vettoriale -POPVECTOR-, il *Local Store* è grande 256 KB di cui una parte viene utilizzata per la memorizzazione del codice che deve essere eseguito. Dall'utility `spu-size` [6] possiamo ottenere un primo valore di occupazione del codice *spe-side* preso dalla prima implementazione su SPE, ancora privo di ottimizzazioni, i valori sono mostrati in tabella 3.2.

Dai valori mostrati in tabella 3.2 si evince che nel *Local Store* sono ancora inutilizzati 84 KBytes ma per il momento non ci preoccupiamo di questo volendo concentrare gli sforzi sull'ottimizzazione del core di calcolo non è necessario occupare tutto lo spazio disponibile. Va ricordato che i dati devono essere *allineati* affinchè possano essere trasferiti da e verso la *Main Memory* con le DMA.

[6] L'utility `spu-size` fornita dal *Software Development Kit* fornisce la dimensione di ogni sezione del codice eseguibile.

| section | size | addr |
|---|---|---|
| .init | 36 | 0 |
| .text | 8392 | 40 |
| .fini | 28 | 8432 |
| .rodata | 176 | 8464 |
| .ctors | 8 | 8640 |
| .dtors | 8 | 8648 |
| .jcr | 4 | 8656 |
| .data | 1072 | 8672 |
| .bss | 148272 | 9856 |
| Total | 175447 | 0 |

Tabella 3.2: Dimensioni dei vari segmenti del codice eseguibile caricato su SPE

Una volta realizzato il codice ottimizzato si possono sperimentare dimensioni della matrice nel *Local Store* che consentano una migliore occupazione dello spazio disponibile.

Per le funzioni è importante poter accedere ai dati nel *Local Store* e non in memoria principale perché i dati possono essere caricati dal LS al Register File in un tempo costante[7]. Accedendo ai dati in tempo costante possiamo permetterci di caricare da una posizione qualsiasi del *Local Store*, senza che la posizione dei dati abbia un impatto sulle performance inoltre questo evita gli stalli non predicibili a *compile-time*. Ciò ,come abbiamo visto per la *collide()* SIMD, può essere un punto a favore, il problema si complica un pochino con la displace mappata sulle istruzioni vettoriali che richiede un trattamento particolare per quelle componenti che vengono spostate diagonalmente.

### 3.5.3 Conclusioni

Avendo note alcune caratteristiche dell'architettura possiamo cercare di ricavare alcuni valori teorici di performance attese, data la quantità di dati da trasferire, la banda disponibile possiamo ricavare il numero di GFlops richiesto da N SPE.

Dalle caratteristiche delle macchine utilizzate per la sperimentazione sappiamo che la potenza di picco di uno SPE in singola precisione è:

$$P_{SPE} = freq\,[GHz] \times 8\,[fops] = freq \times 8\ GFlops \tag{3.4}$$

Nel nostro caso abbiamo a disposizione due macchine:

- La PlayStation 3 con un clock di 3.2 GHz:

$$P_{PS3} = 3.2\,[GHz] \times 8\,[fops] = 25.6\ GFlops \tag{3.5}$$

- Il CAB con un clock di 2.8 GHz:

$$P_{CAB} = 2.8\,[GHz] \times 8\,[fops] = 22.4\ GFlops \tag{3.6}$$

[7]Nella descrizione dell'architettura è stata mostrata l'assenza di elementi di caching verso il *Local Store*, i tempi di accesso locale a tale memoria sono quindi costanti.

Se potessimo pensare di sfruttare tutta la potenza degli SPE, avremmo che con $N$ SPE la potenza di calcolo è data da:

$$P = freq\,[GHz] \times 8\,[fops] \times N \;=\; GFlops \tag{3.7}$$

questa formula possiamo applicarla ai dati della PS3 e del CAB per ottenere il picco teorico di performance di elaborazione:

$$P_{PS3} \;=\; 3.2 \times 8 \;\times 6 \;=\; 153.6 \;\; GFlops \tag{3.8}$$

$$P_{CAB} \;=\; 2.8 \times 8 \;\times 8 \;=\; 179.2 \;\; GFlops \tag{3.9}$$

Date questi valori di performance utilizzando il numero di operazioni per punto ricavato dalla 3.21 possiamo calcolare il massimo teorico del nostro problema:

$$PS3\ 11.28 \;:\; 48 \;=\; x \;:\; 153.6 \rightarrow \; 36GFlops \;\rightarrow\; 6\ GFlops \times \; SPE \tag{3.10}$$

$$CAB\ 11.28 \;:\; 64 \;=\; x \;:\; 179.2 \rightarrow \; 31.584GFlops \;\rightarrow\; 3.948\ GFlops \times \; SPE \tag{3.11}$$

- La BandWidth disponibile è facilmente calcolabile, detto il BUS di trasferimento dati è in grado di trasferire 16 Bytes ogni due cicli di clock:

$$BW\ [GB/s] \;=\; \frac{Clock\ [GHz]}{2} \times \; 16\ Bytes \;=\; Clock \times 8 \tag{3.12}$$

  Note le frequenze di clock delle due macchine a disposizione possiamo ricavare la BW delle due come segue:

$$BW_{PS3}\ [GB/s] \;=\; \frac{3.2\ [GHz]}{2} \times \; 16\ Bytes \;=\; 25.6\ [GB/s] \tag{3.13}$$

$$BW_{CAB}\ [GB/s] \;=\; \frac{2.8\ [GHz]}{2} \times \; 16\ Bytes \;=\; 22.4\ [GB/s] \tag{3.14}$$

  Supponendo di utilizzare la *PlayStation3* la BW sarebbe di 25.6 GB/s,ogni punto fisico (POP) è grande 36 Bytes, possiamo ricavare il numero di punti caricati in un secondo come mostrato nella formula 3.15.

$$POP/s = \frac{(25.6 \times 10^9)}{36} = 711 \times 10^6\,[POP/sec] \tag{3.15}$$

- I punti vanno però caricati dalla *Main Memory* e una volta elaborati devono essere anche salvati nella *MM* L'equazione 3.16 mostra il valore dei POP caricati e salvati in un secondo.

$$\frac{711 \times 10^6}{2} = 355 \times 10^6\,[\ POP/sec\ ] \tag{3.16}$$

- per ogni punto la fase di collisione richiede circa 100 operazioni in virgola mobile,

$$10^2 ops \times 355 \times 10^6 = 355 \times 10^8\,[fops] \tag{3.17}$$

  che possiamo semplficare in

$$\frac{355 \times 10^8}{10^9} = 35.5\ GFlops \tag{3.18}$$

La 3.18 esprime la massima performance raggiungibile dall'algoritmo con una banda passante di 25.6 GB/s. Ogni SPE dovrebbe quindi arrivare almeno ad un valore di $\frac{35.5}{N\_SPE}$ GFlops.

Sappiamo che ogni unità SPE può lavorare su un numero di elementi a 32bit -*word32*- per secondo pari al rapporto tra la banda passante per ogni SPE:

$$\frac{BW}{(8 \times N)} = numero\ di\ word32\ per\ secondo\ elaborate\ da\ ogni\ SPE \tag{3.19}$$

Da questo possiamo dire che affinchè la memoria non risulti un collo di bottiglia, per ogni *word32* devono essere eseguiti un numero $K$ di calcoli in singola precisione:

$$K = \frac{P}{\left(\frac{BW}{8 \times N}\right)} = \frac{(P \times 8 \times N)}{BW} = \frac{(64 \times freq \times N)}{\left(\left(\frac{freq}{2}\right) \times 16\right)} = 8 \times N \tag{3.20}$$

Nel nostro caso specifico abbiamo 100 fops per ogni punto rappresentato da 9 componenti *word32*: per ogni *word32* eseguiamo un numero di operazioni pari a

$$N_{op} = \frac{100}{9} = 11.28 \left[\frac{fops}{w32}\right] \tag{3.21}$$

Considerando il numero di SPE disponibili sulle due macchine abbiamo che ogni macchina può eseguire un numero di operazioni al secondo su una w32 pari a:

$$N_{opPS3} = 6 \times 8 = 48 \left[\frac{ops}{w32}\right] \tag{3.22}$$

$$N_{opCAB} = 8 \times 8 = 64 \left[\frac{ops}{w32}\right] \tag{3.23}$$

Con questi valori possiamo stabilire dei limiti che nel corso dell'analisi delle ottimizzazioni verranno approfonditi.

## 3.6 Sviluppo della versione singolo SPE

Per poter raccogliere dati realistici su quanto le modifiche che stiamo apportando stiano migliorando il codice sono stati realizzati alcuni script in **Perl** che sfruttano l'output prodotto dall'utility `spu-timing` [8].

Lo script principale al quale ci siamo appoggiati per l'analisi del codice ASM prodotto dal compilatore è `act`.

`Act` prende come parametri l'etichetta di inizio della porzione di ASM che vogliamo analizzare e il nome del file di output realizzato con `spu-timing`
(es: `./act L16 spe_program.s.timing` ), in uscita ci vengono fornite diverse informazioni, di seguito vengono citate le principali:

- Il numero di istruzioni, divise per tipo (floating point, integer, load/store,etc.)

[8] L'utility `spu-timing` fornita con il *Software Development Kit* permette di eseguire un'analisi statica del codice di output prodotto dal compilatore. I risultati di questa analisi consistono in una serie di informazioni come ad esempio lo scheduling delle istruzioni, dove viene sfruttato il *Dual-Issue*, dove si verificano gli stalli, etc.

- il numero di *Dual-Issue* sfruttati e quelli non sfruttati[9]
- il numero di stalli riscontrati
- il numero di istruzioni non *Dual-Issue*
- il numero di istruzioni divise per pipeline e l'occupazione delle stesse
- una stima delle performance basata sul numero di cicli richiesti per l'esecuzione dello spezzone di codice preso in esame

Ci si è serviti anche di un altro script che per l'analisi dell'occupazione delle pipeline, `tvt` che ci consente di identificare quali registri vengono utilizzati dalle varie istruzioni e su che PIPE sta operando tale istruzione, in questo modo è anche possibile vedere le istruzioni e i relativi parametri (registri) che sono state eseguite parallelamente sulle due pipeline (*Dual-Issue*). Un'attenta analisi delle informazioni prodotte con l'ausilio degli script consente di dare un'interpretazione più dettagliata del codice assembler prodotto.

### 3.6.1 Condizioni iniziali

Inizialmente per semplicità abbiamo deciso di trattare matrici di dimensioni tali da essere memorizzate per intero nel *Local Store* in questo modo evitiamo di dover predere in considerazione i tempi necessari al traferimento dati dal *Local Store* alla *Memoria Principale*, inoltre assumiamo di utilizzare un solo SPE, per evitare di prendere in considerazione, per il momento, le problematiche relative alla suddivisione del lavoro su più SPE e l'inevitabile sincronizzazione tra di essi.

Il nostro obiettivo è ottenere la massima velocità dal core di calcolo nel rispetto delle considerazioni fatte nella sezione 3.5.3 .

Il *Local Store* è grande 256 KB e il codice portato su singolo SPE è all'incirca 28KB. Potremmo utilizzare tutto lo spazio rimanente per memorizzare dati,che corrisponde a una matrice all'incirca di $128 \times 48 \times 36 = 221184$ Bytes.

Per comodità scegliamo una matrice dati di dimensioni $128 \times 32 \times 36 = 147456$ Bytes, perchè così nel trasferirla da PPE a SPE e viceversa ci limitiamo ad utilizzare 9 richieste DMA con dimensione di 16384 KB che è il massimo che può essere trasferito in una singola DMA.

In tutta la sezione che segue assumiamo sempre di avere le routine che lavorano su intere righe dove una riga si estende lungo l'asse delle ascisse (X).

### 3.6.2 Il codice non ottimizzato

Il codice *spe-side* non ottimizzato è stato ottenuto spostando le funzioni di calcolo dal sorgente C PPE al sorgente C per SPE e aggiungendo il codice necessario per il caricamento e salvataggio della matrice dati dalla *Memoria Principale* al *Local Store*. Il core dell'algoritmo non è stato modificato.

[9] il compilatore è in grado di capire quali operazioni possono potenzialmente andare in parallelo, a volte per *name depences* non vengono schedulate contemporaneamente

Quello che ci aspettiamo è che il programma così fatto sia globalmente più lento del codice x86, dal momento che nella versione **Cell** dobbiamo caricare la matrice dati dalla *MM* al *LS*, eseguire i calcoli e salvarla in *MM*.

Le performance che ci possiamo aspettare non sono 25 Gflops, ma nel migliore dei casi $\frac{25}{4} - 6.25$ poichè il compilatore non dovrebbe essere in grado di auto-vettorizzare.

In questa versione i fattori di ottimizzazione che abbiamo citato nella sezione precedente non hanno ancora avuto applicazione (vettorizzazione, unrolling, etc.), ma abbiamo lasciato la piena libertà al compilatore di eseguire le ottimizzazioni. Alcune considerazioni preliminari sulle performance di questa versione:

| **PIPE0** | **Rt** | **Ra** | **Rb** | **Rc** | **PIPE1** | **Rt** | **Ra** | **Rb** | **Rc** |
|---|---|---|---|---|---|---|---|---|---|
| ori | $4 | $2 | 0 | - | rotqby | $3 | $2 | $16 | - |
| - | - | - | - | - | lqd | $2 | 48 | $sp | - |
| - | - | - | - | - | - | - | - | - | - |
| - | - | - | - | - | rotqbyi | $2 | $2 | 12 | - |
| fs | $2 | $3 | $2 | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - | - | - |
| fnms | $3 | $2 | $24 | $3 | - | - | - | - | - |
| - | - | - | - | - | - | - | - | - | - |
| - | - | - | - | - | shufb | $4 | $3 | $4 | $43 |
| - | - | - | - | - | - | - | - | - | - |
| - | - | - | - | - | stqd | $4 | 0 | $16 | - |
| ai | $16 | $16 | 36 | - | lqd | $2 | 0 | $15 | - |
| - | - | - | - | - | - | - | - | - | - |
| ori | $3 | $2 | 0 | - | rotqby | $2 | $2 | $15 | - |
| - | - | - | - | - | - | - | - | - | - |
| fs | $11 | $2 | $11 | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - | - | - |
| fnms | $2 | $11 | $24 | $2 | - | - | - | - | - |
| - | - | - | - | - | - | - | - | - | - |
| - | - | - | - | - | shufb | $3 | $2 | $3 | $44 |
| - | - | - | - | - | - | - | - | - | - |
| - | - | - | - | - | stqd | $3 | 0 | $15 | - |
| ai | $15 | $15 | 36 | - | - | - | - | - | - |

Tabella 3.3: tvt output, occupazione pipeline e register file

- non sfruttando le istruzioni vettoriali, si ha almeno un fattore 4 di perdita di performance perchè il codice non viene eseguito su una unità **scalar** ma viene mappato su istruzioni vettoriali. Inoltre sono necessarie una serie di operazioni di rotazione,crezione di maschere per lo shuffle, estrazioni e inserimenti di valori diversamente non richieste. In tabella 3.4 vengono confrontate in numero le operazioni necessarie per aggiornare 4 punti trattandoli come scalari o come vettoriali. Dalla tabella possiamo ricavare che il codice scalare è almeno 6.6 volte più lento.

| | Scalar | Vector |
|---|---|---|
| LOAD | 4 | 1 |
| ROTATE | 4 | × |
| CALCULATE | 4 | 1 |
| ROTATE | 4 | × |
| STORE | 4 | 1 |
| **TOTALE** | 20 | 3 |

Tabella 3.4: Confronto istruzioni scalari vs vettoriali

- le due routine principali sono separate, questo non consente di sfruttare le capacità di *Dual-Issueing* del CBE

- osservando il codice assembler si nota che il compilatore non riesce a schedulare bene le istruzioni. Probabilmente aumentando il numero di istruzioni di una sezione, il compilatore potrebbe meglio individuare qualche parallelismo tra le istruzioni e quindi mettere in atto uno scheduling più efficiente. Inoltre l'output evidenzia una scarsa occupazione del register file poichè le istruzioni vengono eseguite utilizzando pochi registri

- osservando la parte finale della routine collide si nota la presenza di molti stalli

```
0D 01                                                  ----- ori     $4,$2,0
1D 0123                                                      rotqby  $3,$2,$16
1   123456                                                   lqd     $2,48($sp)
1    -----7890                                               rotqbyi $2,$2,12
0         ---123456                                          fs      $2,$3,$2
0             -----789012                                    fnms    $3,$2,$24,$3
1                  -----3456                                 shufb   $4,$3,$4,$43
1                       ---789012                            stqd    $4,0($16)
0D                          89                               ai      $16,$16,36
1D                          890123                           lqd     $2,0($15)
0D                           -----45                         ori     $3,$2,0
1D                                4567                       rotqby  $2,$2,$15
0                                  ---890123                 fs      $11,$2,$11
0                                      -----456789           fnms    $2,$11,$24,$2
1  0123                                      -----           shufb   $3,$2,$3,$44
1   ---456789                                                stqd    $3,0($15)
0D      56                                                   ai      $15,$15,36
```

Il codice di output del compilatore è evidentemente poco performante, non capisce dove può vettorizzare i cicli, impiega pochi registri non riconoscendo il parallelismo delle istruzioni e creando molti stalli. In tabella 3.3 viene mostrato uno spezzone dell'output di `tvt` per l'analisi delle pipe lanciato sulla routine collide() relativo alle righe mostrate nella tabella precedente.

In ogni caso il programma gira leggermente più veloce di quanto non faccia l'x86, probabilmente a causa del fatto che la frequenza della *PlayStation 3* è superiore a quella della macchina x86, inoltre la presenza della cache che nella versione x86 non viene adeguatamente gestita, mentre per il CBE, a parte le operazioni DMA all'inizio e alla fine, i dati risiedono sempre nella memoria più veloce, il *Local Store*.

### Statistiche

Utilizzando lo script perl possiamo quantificare in numeri le performance delle routine *collide()* e *displace()*, il tool analizza il loop interno delle due funzioni (la parte reiterata su tutti i dati).

Nella tabella 3.5 vengono mostrati i dati della *collide()* in primo luogo possiamo notare che il numero delle operazioni di accesso in memoria non ha riscontro con quello atteso.

Sappiamo infatti che la routine agisce su un punto caricando tutte le sue componenti e utilizzando tre variabili globali ne calcola i valori aggiornati; un punto è composto di 9 componenti ma il numero di load è 28 e quello di store (che ci dovremmo aspettare essere uguale) è 12.

Quello che possiamo supporre è che il compilatore abbia in qualche modo cercato di ottimizzare il codice, è però evidente che non è riuscito a ricostruire bene lo schema logico della funzione. Il codice così scritto è ricco di stalli e la performance è di 8 Gflops.

| | |
|---|---:|
| loads | 28 |
| stores | 12 |
| Total Instructions | 259 |
| Dual Issues | 63 |
| Failed Dual Issues | 18 |
| Stalls | 412 |
| Not Dual Issued | 196 |
| Expected Cycles | 228 |
| float instructions | 119 |
| float operations | 608 |
| Expected Time | 71.25 ns |
| Expected Performance | 8.53 Gflops |

Tabella 3.5: Statistiche codice *collide()* scalare

Le prestazioni della routine di *displace()* sono mostrate dall'output in tabella 3.6, da notare che in questo il numero di load è esattamente il doppio di quello che ci aspetteremo mentre il numero delle è il valore previsto.

Non vengono espressi valori in Gflops di performance perchè la routine non esegue

| | |
|---|---:|
| loads | 18 |
| stores | 9 |
| Total Instructions | 65 |
| Dual Issues | 13 |
| Failed Dual Issues | 2 |
| Stalls | 78 |
| Not Dual Issued | 52 |
| Expected Cycles | 59 |
| Expected Time | 18.43 ns |

Tabella 3.6: Statistiche codice *displace()* scalare

calcoli in floating point. Sono presenti alcune operazioni intere per il calcolo degli indirizzi ma non sono rilevanti (anche se cercheremo di ridurle il più possibile). In figura 3.8 possiamo vedere come le performance della versione Cell scalare su singolo SPE rispetto alla versione x86 sia di poco superiore. Possiamo concludere che la versione scalare del codice non viene ottimizzata dal compilatore in modo accettabile.

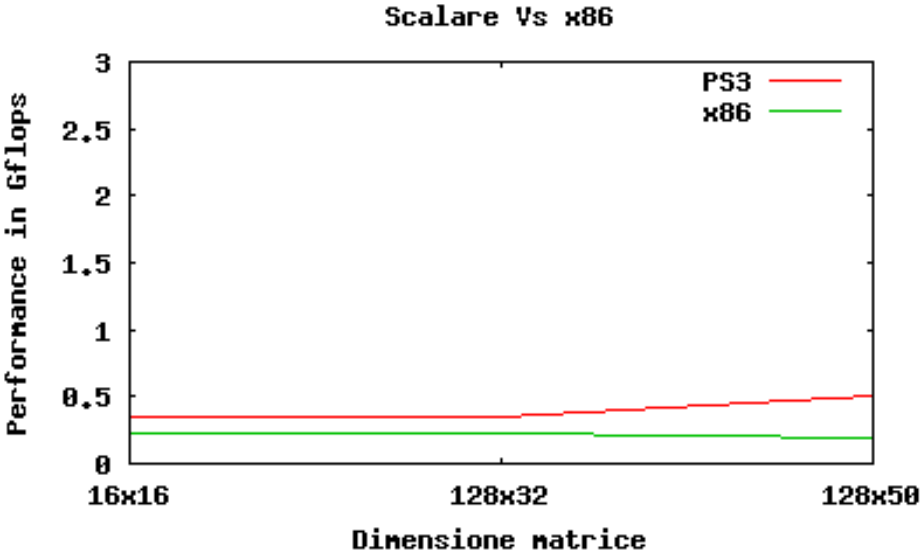



Figura 3.8: Performance reali nell'elaborare una matrice di 128x32 punti

Nel seguito del capitolo verranno illustrate le modifiche che hanno portato a miglioramenti significativi.

### 3.6.3 Utilizzo delle estensioni vettoriali

Come è stato illustrato nel capitolo dedicato all'architettura, il Cell mette a disposizione una serie di *features* che lo rendono particolarmente adatto all'impiego in task *compute intensive*. L'impiego di registri a 128Bit e la possibilità parallelizzare il lavoro su più unità di calcolo sono sicuramente la sua forza.

Nell'algoritmo le funzioni del core di calcolo devono essere riscritte per sfruttare le istruzioni SIMD, nella sezione precedente abbiamo analizzato le implicazioni teoriche che questo comporta.

Lo schema dati scelto è a raggruppamento di quattro punti contigui dove gli elementi della matrice dati -i POPVECTOR- sono definiti come segue:

Listing 3.1: POPVECTOR data type definition

```
typedef struct {
 vector float p[9];
} POPVECTOR;
```

La figura 3.9 mostra come sono organizzate le componenti del POPVECTOR.

Il nostro obiettivo è ottenere un codice privo di stalli che sfrutti appieno le pipeline dove ogni routine vogliamo occupa una sola pipeline.

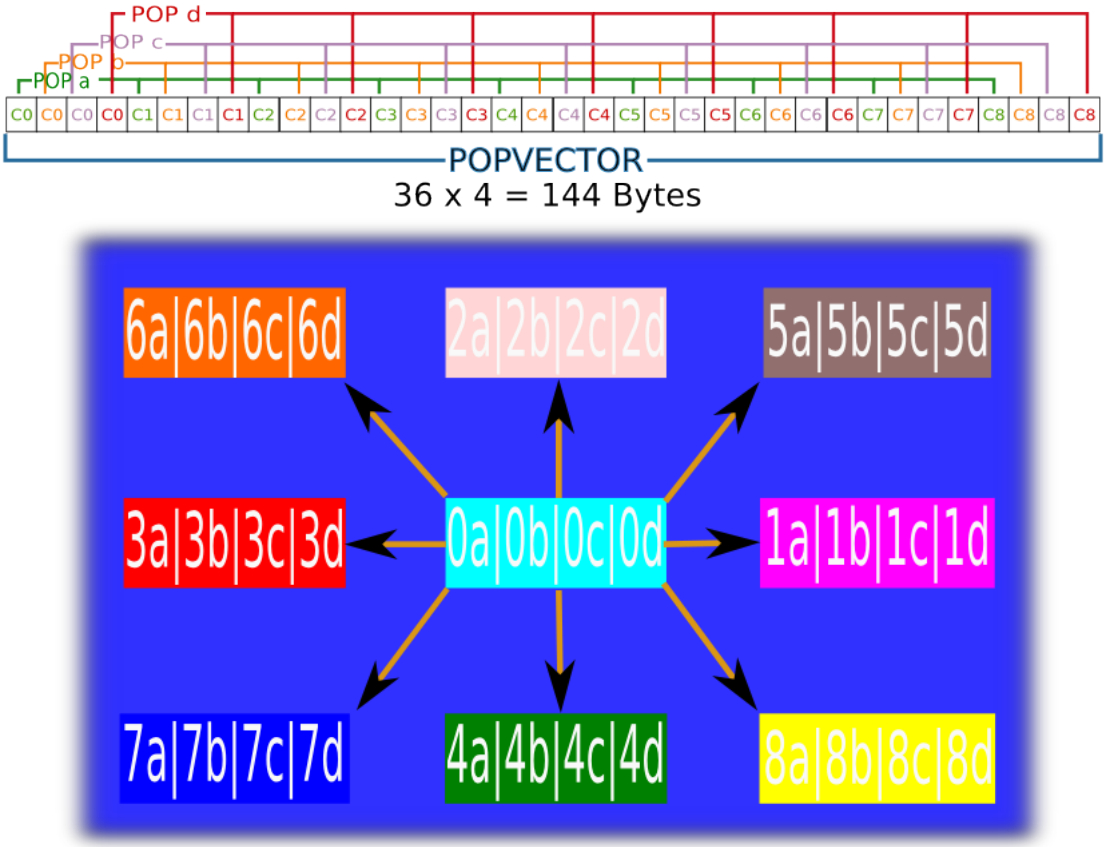



Figura 3.9: POPVECTOR

### Vettorizzazione della collide

Il codice che abbiamo visto nella sezione 3.2 deve essere ripensato per poter sfruttare le istruzioni vettoriali. Nel riscrivere il codice le operazioni vanno rimappate nel modo più efficente possibile sfruttando a pieno le possiblità offerte dall'Instruction Set. Esistono singole istruzioni in grado di eseguire due operazioni tra tre operandi operandi, un esempio ne è la seguente:

```
d = spu_madd( a, b, c)
```

che esegue la moltiplicazione di ogni elemento di $a$ per il corrispondente elemento di $b$,e somma i corrispondenti elementi del vettore $c$, salvando il risultato dell'operazione nel vettore $d$, come mostrato dalla formula 3.24:

$$d = a \times b + c \tag{3.24}$$

La `spu_madd()` consente di eseguire con una sola istruzione quello che un processore SISD eseguirebbe in 8 operazioni come segue: supponiamo a,b,c,d vettori di 4 elementi, il codice corrispondente alla `spu_madd()` per un processore senza unità vettoriale sarebbe il seguente:

```
d[0] = a[0] * b[0] ;
d[1] = a[1] * b[1] ;
d[2] = a[2] * b[2] ;
d[3] = a[3] * b[3] ;

d[0] = d[0] + c[0] ;
d[1] = d[1] + c[1] ;
d[2] = d[2] + c[2] ;
d[3] = d[3] + c[3] ;
```

Il passo più complesso di tutto il procedimento di riscrittura del codice con le istruzioni vettoriali consiste nella riscrittura di sequenze di operazioni in istruzioni singole.

La matematica ci può venire in aiuto inquanto definisce dei criteri, dei vincoli che dobbiamo rispettare nel ripensare le operazioni.

Come esempio prendiamo calcolo di RHO ($\rho$) nella collide classica viene chiamata una MACRO che esegue la somma si tutte le componenti del punto passato come parametro. Vediamo un esempio di come è stato rimappato il codice del calcolo di $\rho$ nel passaggio alle istruzioni vettoriali:

```
00 // Prima della vettorizzazione (POP)
01 #define m(a) (a.p[0]+a.p[1]+a.p[2]+a.p[3]    \
02                 +a.p[4]+a.p[5]+a.p[6]+a.p[7]   \
03                 +a.p[8])
04 rho = m( d[y][x] );

05  // Con l'uso delle intrinsics (POPVECTOR)
06 tempA = spu_add( d[y][xv].p[0] , d[y][xv].p[1] );
07 tempB = spu_add( d[y][xv].p[2] , d[y][xv].p[3] );
08 tempC = spu_add( d[y][xv].p[4] , d[y][xv].p[5] );
09 tempD = spu_add( d[y][xv].p[6] , d[y][xv].p[7] );

10 tempA = spu_add( tempA, tempB );
11 tempC = spu_add( tempC, tempD );

12 rho   = spu_add( tempA, C8 );
13 rho   = spu_add( rho, tempC);
```

Possiamo fare alcune osservazioni sul codice sopra scritto:

- la versione vettoriale sta operando su quattro punti contemporaneamente, perchè raggruppati come descritto dalla definizione di POPVECTOR
- il modo in cui sono state scritte le istruzioni cerca di mettere in evidenzia le operazioni parallelizzabili, guardiamo alle righe dalla 06 alla 09, quelle istruzioni operano su dati diversi, sono quindi perfettamente parallelizzabili, le uniche righe che devono essere eseguite in sequenza sono la 12 e la 13 in quanto vi è una dipendenza

Un esempio più complesso rispetto al precedente è l'impiego dell'istruzione `spu_nmsub()` che come la `spu_madd()` esegue due operazioni in una sempre su vettori ( 8 fops ).

```
00 // d[y][x].p[0] valore presente in memoria della
01 // componente 0
02 // p_eq.p[0] valore della componente 0 aggiornato
03 d[y][x].p[0] = p[y][x].p[0] - INVTAU *
                              (p[y][x].[0] - p_eq.p[0]);

04 // suppongo di vare il TC0 il valore aggiornato
05 //  della componente 0
06 // per il POPVECTOR corrente e in C0 ho il valore
07 //  originale

08 TC0 = spu_sub( C0 - TC0 );
09 C0  = spu_nmsub( TC0, invtau, C0 );
```

Una volta completata la rimappatura del codice della *collide* ci si è trovati davanti al problema degli stalli, ora che il codice sfrutta le istruzioni vettoriali come possiamo far sì che le istruzioni vengano schedulate per una piena occupazione della pipeline?

Possiamo dividere la *collide()* in modo tale da poter evidenziare l'indipendenza delle istruzioni. Pensiamo ad esempio di dividere la routine in due parti, una prima parte di calcolo delle variabili fisiche e una seconda parte di aggiornamento delle componenti del punto. Così che ad ogni ciclo vengano elaborati due punti invece di uno solo. Un esempio

**Unrolling *Collide***

| Fase | Azioni | VarSet |
|---|---|---|
| PRE | RHO( x , y ) | A |
| ITER | RHO( x+1,y ) | B |
| | UPDATE( x , y ) | A |
| | RHO( x+2,y ) | A |
| | UPDATE ( i+1 ) | B |

Figura 3.10: Unrolling collide

di quello che si ottiene come codice di output dall'utility `spu-timing` è mostrato dal listato seguente. Considerazione generale: un codice con più istruzioni ma in *Dual-Issue* può essere più veloce di un codice con meno istruzioni. Il numero di "**D**" è aumentato il che significa che siamo riusciti a fare più operazioni in meno cicli di clock. [10]

```
0D 789012                    fnms    $7,$8,$7,$38
1D 789012                    lqa     $2,ff
0   890123                   fa      $18,$13,$19
0    901234                  fnms    $3,$3,$17,$40
0D    012345                 fa      $12,$11,$10
1D    012345                 lqd     $40,16($5)
```

[10] La `D` indica che il compilatore è riuscito a schedulare due istruzioni per essere eseguite contemporaneamente una su una pipe e una sull'altra.

```
0D     123456                    fnms    $6,$6,$17,$39
1D     123456                    lqd     $39,32($5)
0       234567                   fnms    $8,$8,$9,$37
0       345678                   fnms    $7,$7,$17,$38
0D       456789                  fa      $15,$15,$26
1D       456789                  lqd     $38,48($5)
0         567890                fa      $3,$3,$2
0          678901               fm      $14,$21,$14
0D          789012              fa      $4,$12,$18
1D          789012              stqd    $6,-256($5)
0D           890123             fnms    $8,$8,$17,$37
1D           890123             lqd     $37,64($5)
0             901234            fs      $12,$12,$18
0              012345           fa      $15,$15,$16
0D              123456          fs      $13,$13,$19
1D              123456          stqd    $3,-272($5)
0D               234567         fs      $11,$11,$10
1D               234567         lqa     $2,ff
0d                345678        fnms    $4,$14,$4,$36
1d                -456789       stqd    $8,-224($5)
0d                  567890      fnms    $12,$14,$12,$35
1d                  -6789       frest   $6,$15
0                     789012    fa      $3,$28,$27
0                      890123   fs      $7,$7,$2
0D                      901234  fnms    $4,$4,$17,$36
1D                      901234  lqd     $36,80($5)
0D                       0123456 fi     $6,$15,$6
```

Utilizzando `act` otteniamo i dati mostrati in tabella 3.7 dove è ancora incomprensibile il numero delle load mentre il numero delle store è esattamente quello necessario per memorizzare due punti (due POPVECTOR), come si vede il numero degli stalli si è ridotto di un fattore 10. Da ricordare che in questo modo stiamo aggiornando ad ogni ciclo 8 (4 punti perchè POPVECTOR che considerando l'*unrolling* diventano 8) punti mentre nella versione scalare ne stavamo aggiornando solamente uno.

| | |
|---|---:|
| loads | 47 |
| stores | 18 |
| Total Instructions | 229 |
| Dual Issues | 75 |
| Failed Dual Issues | 24 |
| Stalls | 49 |
| Not Dual Issued | 154 |
| Expected Cycles | 192 |
| float instructions | 152 |
| float operations | 800 |
| Expected Time | 60.00 ns |
| Expected Performance | 13.33 Gflops |

Tabella 3.7: Statistiche codice *collide()* vettoriale

### Vettorizzazione della displace

Nel vettorizzare la *displace()* è necessario prestare particolare attenzione al comportamento delle componenti laterali (1,3,5,6,7,8) che richiedono alcune operazioni "speciali"

rispetto alle altre componenti. Per aggiornare queste componenti è necessario avere a disposizione i valori delle componenti del punto corrente e del suo predecessore. Prendiamo in esame il caso delle componenti che si muovono verso destra (1,5,8): Per aggiornare il POPVECTOR corrente è necessario avere in memoria anche il precedente, nel dettaglio supponendo di utilizzare già un unrolling di due punti (vista la necessità di avere un POPVECTOR e il suo precedente viene quasi automatico) per aggiornare un punto eseguiamo le seguenti operazioni:

- supponendo di avere in una variabile che chiameremo CA la componente del punto di ascisse X-1 ruotata di 4 bytes a destra e in CB di avere la componente del punto di ascisse X sempre ruotata di 4 bytes a destra, aggiorniamo il valore della componente con il valore ottenuto estraendo dal componente CA il primo elemento ed inserendolo come primo elemento CB
- aggiorniamo CA caricandole la componente ruotata di 4 bytes a destra del punto di ascisse X+1
- aggiorniamo il valore della componente del punto di ascisse X+1 con il valore ottenuto estraendo dalla componente CB il primo elemento ed inserendolo a primo elemento di CA
- aggiorniamo CB caricando il valore della componente del punto di ascisse X+2 ruotato di 4 bytes a destra

Questo che abbiamo descritto è il loop interno che richiede alcune istruzioni di contorno prima e dopo per il caricamento e il salvataggio. La figura 3.11 mostra graficamente il processo appena descritto.

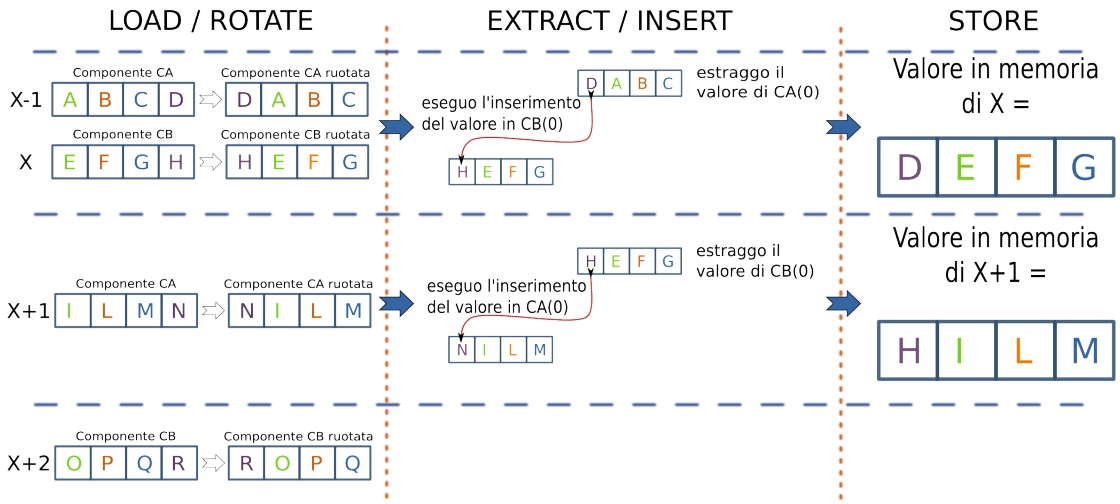



Figura 3.11: Aggiornamento componenti vettoriali

Questo significa che per ogni componente che si muove verso destra devo eseguire:

- 2 estrazioni di un elemento da un vettore
- 2 inserimenti di un elemento in un vettore

- 2 salvataggi di un vettore in memoria
- 2 caricamenti di un vettore dalla memoria
- 2 rotazioni di un vettore

per un totale di 10 istruzioni eseguite tutte sulla **pipeline 1**, se da un lato abbiamo il fatto positivo che queste istruzioni occupano una stessa pipeline, il che implica che prendendo in considerazione tutte le componenti da aggiornare il numero di operazioni dovrebbe essere tale da consentire di coprire la latenza tra le istruzioni dipendenti e quindi eliminare gli stalli, dall'altra parte abbiamo che queste 10 istruzioni probabilmente possono essere evitate organizzando diversamente i dati. Le considerazioni appena fatte valgono per le componenti che si muovono lateralmente quindi queste operazioni "*superflue*" coinvolgono 6 delle 9 componenti il che significa che se riuscissimo ad evitarle le performance migliorerebbero ulteriormente. Per le componenti 0,2 e 4 la displace è una pura e semplice copia da una locazione di memoria ad un'altra, percui sono sufficienti una Load e una Store.

Nel codice di output prodotto dall'utility `spu-timing` vediamo che gli stalli sono quasi del tutto scomparsi e che le istruzioni vengono schedulate una dopo l'altra occupando quasi unicamente la pipeline 1, l'eccezione è data dalla presenza di alcune operazioni di addizione per la creazione degli indirizzi, in questo senso dobbiamo cercare di far capire al compilatore che può sfruttare le istruzioni immediate come già accennato nella sezione precedente.

La struttura di unrolling viene mostrata in figura 3.12. Dalla tabella 3.8 vediamo che

**Unrolling *Displace***

| Fase | Azioni | VarSet |
|---|---|---|
| PRE | LOAD X-1 | A |
| | STORE X-1 ⇦ (X-1, ) | (A,A) |
| | LOAD X | B |
| ITER | STORE X ⇦ (X,X-1) | (B,A) |
| | LOAD X+1 | A |
| | STORE X+1 ⇦ (X+1,X) | (A,B) |
| | LOAD X+2 | B |
| POST | STORE X+2 ⇦ (N,N-1) | (B,B) |

Figura 3.12: Unrolling displace, gli estremi dell'ITER sono $x \in [1, N-2]$

le cose sono molto migliorate rispetto alla versione originale sempre tenendo conto che anche qui stiamo lavorando su POPVECTOR e che quindi quando aggiorniamo un punto in reltà ne stiamo aggiornando quattro contemporaneamente.

Un'altra importante considerazione che dobbiamo fare è che la *displace()* non può salvare direttamente in memoria (stiamo sempre e solo considerando il *Local Store* non stiamo parlando di *Main Memory*) ma deve servirsi di un meccanismo di *Double Buffering*

| | |
|---|---|
| loads | 18 |
| stores | 18 |
| Total Instructions | 87 |
| Dual Issues | 45 |
| Failed Dual Issues | 2 |
| Stalls | 1 |
| Not Dual Issued | 42 |
| Expected Cycles | 65 |
| Expected Time | 20.31 ns |

Tabella 3.8: Statistiche codice *displace()* vettoriale

che ci consente di creare i valori aggiornati delle varie componenti salvandoli temporaneamente in un buffer che è in grado di ospitare due righe intere, così che mentre stiamo prendendo i nuovi valori della riga Y possiamo salvare in memoria i valori della riga Y-2.

Sappiamo che una riga per poter essere aggiornata ha bisogno dei valori presenti nella riga superiore (Y+1) e inferiore(Y-1) il che implica che non possiamo salvare subito in memoria la riga Y-1 una volta calcolati i suoi valori aggiornati possiamo farlo solo una volta che i valori in memoria non devono più essere utilizzati e cioè quando siamo 2 righe avanti.

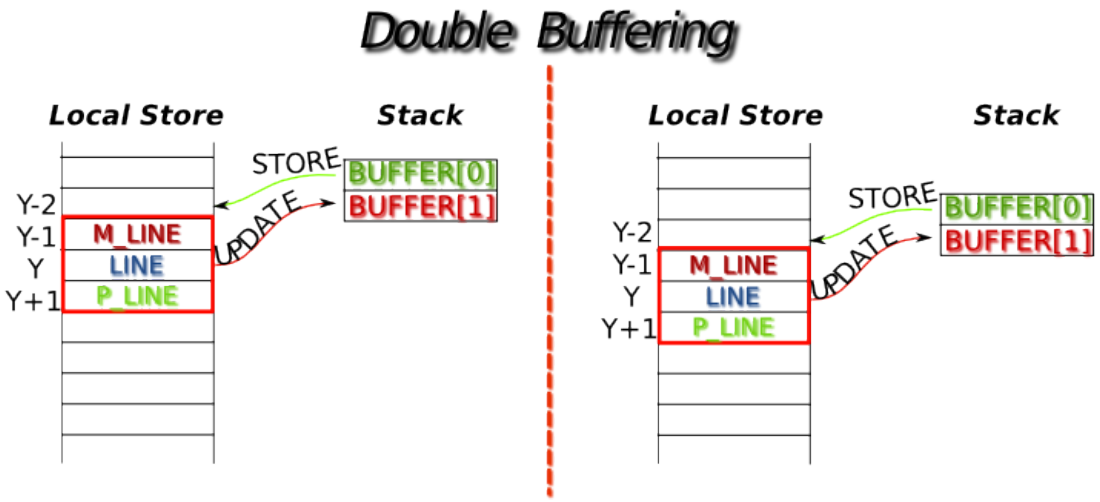



Figura 3.13: Schema *Double Buffering*

### 3.6.4 Riorganizzazione delle strutture dati

Le istruzioni vettoriali hanno portato ad un notevole miglioramento delle performance ma permangono alcuni problemi che devono essere assolutamente risolti:

- presenza di istruzioni inutili, come abbiamo visto nella *displace()* per l'aggiornamento delle componenti laterali

- nelle operazioni come si può notare dagli schemi di unrolling permane una dipendenza, anche se meno pesante rispetto al codice originale, tra le varie operazioni

per eliminare questi problemi si è studiato un nuovo modo di organizzare i dati in "pacchetti" di 16 elementi (4 POPVECTOR). Quello che vogliamo è che in un POPVECTOR non siano presenti le componenti di punti contigui ma di punti distanti un certo intervallo - **DIST**- che nel caso specifico è 4.

La figura dovrebbe rendere chiara la disposizione, ogni riga rappresenta un Vector Float di una qualsiasi delle componenti. Questo nuovo modo di organizzare i dati richiede

**Nuovo Schema Dati**

| | | | | |
|---|---|---|---|---|
| 0 | DIST | 2*DIST | 3*DIST | POPVECTOR 0 |
| 1 | 1 + DIST | 1 + 2*DIST | 1+ 3*DIST | POPVECTOR 1 |
| 2 | 2 + DIST | 2 + 2*DIST | 2 + 3*DIST | POPVECTOR 2 |
| ... | ... | ... | ... | POPVECTOR 3 |
| (DIST-1) | (DIST-1) + DIST | (DIST-1) + 2*DIST | (DIST-1) + 3*DIST | POPVECTOR (DIST-1) |

Ogni riga rappresenta un Vector Float

Figura 3.14: In questa nuova organizzazione i vector contengono le componenti di punti tra loro distanti `DIST`.

un unrolling pari alla distanza tra i punti presenti in uno stesso vector, nel nostro caso abbiamo punti distanti 4 elementi il che implica un unrolling di 4.

Questo tipo di schema dati introduce una limitazione sulle dimensioni che possiamo assegnare alla matrice abbiamo infatti che gli elementi di ogni riga devono essere un multiplo di 16. Dal punto di vista pratico questo compora che la matrice non può assumere dimesioni di NX arbitrarie.

### collide

La *collide()* opera sul punto, avere punti contigui o avere punti distanti tra loro non comporta nessuna particolare modifica.

### displace

La nuova organizzazione dati ci consente di ottimizzare sul numero di operazioni eseguite, il meccanismo di buffering resta lo stesso ma ora il numero di istruzioni necessario per le componenti laterali è drasticamente ridotto. Con il nuovo schema dati l'aggiornamento delle componenti laterali è eseguito come quello delle altre componenti, con una semplice copia (fatta eccezione per gli estremi del pacchetto dati). Questo significa che più punti elaboriamo in uno stesso iter (ovvero più è grande il pacchetto) meno operazioni sono richieste, perchè abbiamo meno "bordi". La figura 3.15 mostra come ora lo schema di aggiornamento di una componente laterale sia diventato più semplice. Quello che notiamo è che per ogni componente laterale eseguiamo 3 operazioni di copia (per i POPVECTOR non di bordo) e solamente una rotazione un'estrazione e un inserimento, il che significa che per aggiornare 16 punti dobbiamo eseguire 3 load+store, 1 load + rotate, 1 load, 1 extract e 1 insert. Per aggiornare una componente laterale per 16 punti occorrono 10 operazioni, con lo schema dati precedente occorrevano 10 operazioni per aggiornare la

componente laterale di 4 punti. Abbiamo ottimizzato il numero di istruzioni riducendolo di un fattore 4, il che non è male. Possiamo vedere con `act` come sono cambiate le istruzioni. Possiamo notare dalle statistiche mostrate in tabella 3.9 che la funzione per

| | |
|---|---|
| loads | 36 |
| stores | 36 |
| Total Instructions | 111 |
| Dual Issues | 41 |
| Failed Dual Issues | 0 |
| Stalls | 0 |
| Not Dual Issued | 70 |
| Expected Cycles | 91 |
| Expected Time | 28.43 ns |

Tabella 3.9: Statistiche codice *displace()* con il nuovo schema dati che opera su 16 punti

elaborare 16 punti richiede soli 28.43 nanosecondi, 8 secondi in più rispetto alla versione con la vecchia organizzazione con la differenza sostanziale che si lavora su 16 punti e non su 4 punti.

Dalla tabella si evince anche come la routine abbia perfettamente interpretato lo schema di accesso in memoria, eseguendo solo i caricamenti di cui aveva bisogno. Utilizzando l'aritmetica dei puntatori per accedere alla memoria i dati di output di `act` non subiscono la minima variazione, tutto resta esattamente uguale.

### 3.6.5 Unione di *displace* e *collide*

Ora che le routine hanno raggiunto un buon livello di performance possiamo pensare di unirle in un unica funzione. Questo ci dovrebbe portare ad un ulteriore miglioramento perchè come detto in precedenza, ci aspettiamo che il numero di accessi in memoria sia dimezzato.

Sia la funzione di *displace()* che di *collide()* accedono alle componenti dei punti, possiamo mettere insieme le due funzioni facendo sì che i dati vengano caricati in variabili temporanee secondo lo schema di accesso alla memoria della displace e poi su queste variabili temporanee eseguiamo le istruzioni della collide, successivamente salviamo in memoria i valori così calcolati per le componenti. Se pensiamo di elaborare 4 punti per iter (come nel caso della *displace()* con il nuovo schema dati) dovremmo ottenere esattamente 36 load e 36 store.

La fusione delle due routine dovrebbe consentire un ulteriore sfruttamento del *Dual-Issue*, avendo più punti da elaborare all'interno di uno stesso iter possiamo pensare che il compilatore scheduli le istruzioni di store in memoria del primo POPVECTOR in parallelo con le istruzioni di calcolo dei nuovi valori aggiornati per le componenti del secondo POPVECTOR.

Non bisogna trascurare la presenza del *Double-Buffering*. Come abbiamo detto una riga appena aggiornata non può essere salvata direttamente in memoria (LS), deve prima essere memorizzata temporaneamente in un buffer LOCALE, una volta che non avremo più bisogno del valore originale in memoria possiamo salvarla. Il salvataggio della

riga presente nel buffer in memoria introduce ulteriori operazioni sulla **pipeline 1** ma questo non è assolutamente un problema, dato che stiamo eseguendo le operazioni su 4 POPVECTOR il *Dual-Issue* ci aspettiamo che sia quasi totale.

### Merging delle funzioni di collide e displace in una singola routine

Nello scrivere delle due un unica funzione si sono create delle MACRO (a.k.a *inline functions*) tali da poter richiamare uno stesso insieme di istruzioni su vari POPVECTOR. Le MACRO ci consentono inoltre di selezionare quale tipo di accesso alla memoria preferiamo se lo standard C o un indirizzamento manuale realizzato mediante somma di offset.

In questo modo possiamo verificare se un diverso modo di eseguire l'indirizzamento produce qualche differenza di performance.

Lo schema di esecuzione delle operazioni è descritto in figura 3.16 dove le Azioni in *blu* fanno accesso al *Local Store*, le azioni di *COPY* sono in rosso perchè leggono dal BUFFER locale (su stack) e scrivono sul *Local Store*.

L'insieme delle operazioni dovrebbe essere tale da consentire al compilatore di schedularle sfruttando bene il *Dual-Issue* l'ipotesi migliore che possiamo fare è che il compilatore eviti tutte le istruzioni inutili, e che schedulando le istruzioni di accesso alla memoria parallelamente a quelle che eseguono i calcoli le prime vengano completamente mascherata nelle ultime (in quanto in numero superiore) in termini di tempi di esecuzione.

| | |
|---|---:|
| loads | 72 |
| stores | 72 |
| Total Instructions | 513 |
| Dual Issues | 319 |
| Failed Dual Issues | 2 |
| Stalls | 1 |
| Not Dual Issued | 194 |
| Expected Cycles | 354 |
| Expected Time | 110.62 ns |
| Expected Performance | 14.46 Gflops |

Tabella 3.10: Statistiche codice *displace()* con il nuovo schema dati che opera su 16 punti

In tabella 3.10 sono mostrati i dati in output di `act` da cui possiamo capire che il codice sta eseguendo il numero esatto di accessi che ci aspettiamo abbiamo 36 Load e 36 Store dovute al caricamento / aggiornamento dei valori del pacchetto di punti corrente e abbiamo altrettante Load e Store dovute all'operazione di copia del buffer locale in memoria (LS).

Per renderci conto di quanto le cose siano migliorate applicando le varie modifiche al core di calcolo segue una tabella riassuntiva dei vari dati normalizzati all'aggiornamento di 16 punti (4 POPVECTOR, ovvero un gruppo nella nuova organizzazione dati), così che possiamo confrontare direttamente le performance delle varie versioni su una stessa quantità di dati. Nella tabella 3.11 sono state evidenziate in rosso 3 voci, due indicano i cicli necessari per l'esecuzione delle routine di calcolo con la nuova organizzazione dati, mentre l'altra voce evidenziata mostra i cicli necessari per eseguire le routine unificate

| | Scalare | Vettoriale | Nuovo schema dati | Combo |
|---|---|---|---|---|
| Total Instructions | 5184 | 632 | 569 | 513 |
| Dual Issues | 1216 | 300 | 191 | 319 |
| Failed Dual Issues | 320 | 52 | 48 | 2 |
| Stalls | 7840 | 100 | 98 | 1 |
| Not Dual Issued | 3968 | 392 | 378 | 194 |
| Displace Cycles | 944 | 130 | 91 | - |
| Collide Cycles | 3648 | 384 | 384 | - |
| Expected Cycles | 4592 | 514 | 475 | 354 |
| Speed-up | 1 | 8,93 | 9,67 | 12,97 |
| Expected Gflops | 1,11 | 9,96 | 10,78 | 14,46 |

Tabella 3.11: Comparativa prestazioni per l'aggiornamento di 16 punti (1600 fops).

in una singola funzione evidenziando il fatto che il numero di cicli per l'esecuzione della funzione unica è minore della somma dei cicli necessari per la collide e quindi minore anche della somma dei cicli necessari all'esecuzione di entrambe le funzioni. Questo perchè come abbiamo avuto modo di capire il *Dual-Issue* permette un forte incremento delle performance. Nella routine unificata abbiamo molte operazioni che devono essere schedulate e questo evidentemente gli consente di sfruttare al meglio la presenza delle due pipeline.

In conclusione abbiamo che la versione unificata del core di calcolo è la più efficiente con uno Speed-up totale di 12,97, da questo core di elaborazione possiamo pensare di ricavare una versione parallela per l'impiedo di tutte le unità SPE disponibili.

## 3.7 Sviluppo della versione multi-SPE

Di seguito viene illustrata la parallelizzazione del core di calcolo. In questo processo di divisione del lavoro tra le varie unitá SPE devono essere trattate le problematiche relative alla sincronizzazione al fine di garantire le massime performance e la consistenza dei dati[11].

Il tipo di suddivisione che andiamo ad operare è tale percui gli SPEs eseguono le stesse istruzioni su porzioni di dati diverse. Per partizionare i dati si è adottato quello che abbiamo definito come "*partizionamento monodimensionale*".

Il *partizionamento monodimensionale* assume che ogni SPE elabori una porzione contigua di righe, così da ridurre la sincronizzazione alle sole righe superiori e inferiori di ogni porzione. Così come per l'analisi del core di calcolo anche qui vogliamo che le routine di calcolo elaborino intere righe quindi dobbiamo dimensionare la matrice in modo tale che il numero di elementi lungo l'asse X (ogni riga è composta di NX elementi) sia tale da consentirne la memorizzazione nel *Local Store* che ricordiamo essere di 256 KB.

Un'altra importante *limitazione* è dettata dal modo in cui elaboriamo i dati, come abbiamo visto prima il core di calcolo elabora un insieme di 4 POPVECTOR per iter,

[11] La consistenza dei dati è mantenuta se i dati vengono tutti letti, elaborati e poi scritti in modo ordinato, qualora un SPE ad un iterzione $i$ leggesse un valore aggiornato da un altro SPE ad uno step $i+1$ i dati risulterebbero corrotti.

il che significa che la dimensione NX della matrice deve essere divisibile per 16, il che riduce le grandezze di riga utilizzabili.

### 3.7.1 Meccanismo di sincronizzazione

Di seguito viene descritto il meccanismo di sincronizzazione si rende però necessario spiegare alcuni termini prima di poter descrivere l'interazione tra gli SPEs vicini:

**TOP_LINE** è la riga che vogliamo leggere dal vicino superiore

**FIRST_LINE** è la prima riga che aggiorniamo e che vogliamo scrivere

**LAST_LINE** è l'ultima riga che abbiamo calcolato e vogliamo salvarla in memoria

**BOTTOM_LINE** è la riga che vogliamo leggere dal vicino inferiore

**$SPE_N$** rappresenta il vicino superiore

**$SPE_S$** rappresenta il vicino inferiore

Il meccanismo di sincronizzazione generico per uno SPE centrale che interagisce cioé con entrambi i vinici è composto di 3 fasi principali:

**PRE**
1. attesa del permesso in lettura della TOP_LINE da parte del vicino superiore $SPE_N$, questa riga mi serve per poter calcolare la mia prima riga aggiornata
2. attesa del permesso in lettura della BOTTOM_LINE da $SPE_S$ questa riga vogliamo precaricarla per permettere al vicino inferiore di salvare subito la sua prima riga (FIRST_LINE di $SPE_S$)
3. non appena ho caricato la TOP_LINE dal vicino superiore $SPE_N$ gli comunico che può procedere nella scrittura del valore aggiornato della sua LAST_LINE
4. non appena ho caricato la BOTTOM_LINE dal vicino inferiore $SPE_S$ gli comunico che può procedere nella scrittura del valore aggiornato della sua FIRST_LINE

**ITER** come SPE vorrei poter scrivere la prima linea che ho elaborato devo quindi attendere dal mio vicino il permesso in scrittura della mia FIRST_LINE

**POST**
1. l'iterazione su tutte le righe è terminata resta da salvare l'ultima riga per fare questo devo attendere il permesso di scrittura LAST_LINE dal mio vicino inferiore $SPE_S$
2. avendo scritto la mia FIRST_LINE comunico al vicino superiore $SPE_N$ che può leggere la sua BOTTOM_LINE
3. allo stesso modo avendo terminato la scrittura della mia LAST_LINE comunico al vicino inferiore $SPE_S$ che può leggere la sua TOP_LINE

La figura 3.17 dovrebbe rendere più chiaro il meccanismo di sincronizzazione tra SPEs vicini.

Per poter essere efficienti è necessario sovrapporre i tempi di trasferimento dati dalla *Main Memory* al *Local Store* con i tempi di calcolo. Possiamo pensare di utilizzare un

buffer nel *Local Store* che ci consenta di memorizzare 5 righe. Cinque righe perchè come abbiamo detto in precedenza per aggiornare una riga servono i valori della stessa e delle due vicine (superiore e inferiore), inoltre sappiamo di non poter scrivere direttamente il valore aggiornato in memoria principale inquanto i valori originali dobbiamo mantenerli anche per elaborare la riga successiva, necessitiamo quindi di una riga in cui salvare i valori appena calcolati. Un altra riga è necessaria per poter richiedere il caricamento delle successive righe da elaborare in modo asicrono.

Il *partizionamento monodimensionale* fa sì che ogni SPE debba elaborare un insieme di righe contiguo, e ci consente di trattare problemi di dimensione arbitraria lungo l'asse delle ordinate mentre lungo le ascisse siamo limitati ad un valore di NX tale da consentire di memorizzare 5 righe di grandezza NX nel *Local Store*.

Possiamo immaginare il meccanismo delle 5 linee come il seguente insieme di operazioni: supponiamo di elaborare la riga o linea *"l"*

- una volta terminata la richiesta di salvataggio della riga $l$-2 (se vi è stata) carico la riga $l$+2 nel buffer
- elaboro la riga $l$ servendomi di $l-1$,$l$ e di $l+1$ già presenti nel *LocalStore*
- se sono alla riga $l >= 2$ salvo in memoria (LocalStore $\rightarrow$ MainMemory) la riga $l-2$

La figura dovrebbe chiarire lo schema di funzionamento dei 5 buffer.

La comunicazione a livello implemetativo viene realizzata mediante l'utilizzo delle mail, durante lo sviluppo si è cercato di valutare l'impatto dell'infrasfruttura di comunicazione sulle performance. La tabella 3.7.1 mostra le tempistiche dei vari segmenti di codice, questi valori si riferiscono ai tempi di esecuzione di un iterazione e si sono ricavati facendo girare il programma con una matrice dati di 896x896 punti per 10000 iterazioni sulla *PlayStation3*.

| **Full Time** | **Combo Time** | **Pre Time** | **SYNC Time** | **STALL Time** |
|---|---|---|---|---|
| 2919862 | 975605 | 2919862 | 38182 | 1888060 |
| **Full %** | **Combo %** | **PRE %** | **SYNC %** | **STALL %** |
| 100.0000 | 33.4127 | 1.307676 | 0.219034 | 64.6626 |

Tabella 3.12: Tempi dei vari segmenti del codice multi SPE

Nell'ordine in tabella compaiono i tempi espressi in nanosecondi per l'esecuzione di: tutto il codice di un iterazione, il codice della routine di calcolo unificata, il tempo necessario alla fase di pre come descritta dallo schema di sincronizzazione, delle istruzioni di sincronizzazione (invio e ricezione mail) e i tempi di attesa di terminazione dei trasferimenti DMA. I valori mostrati in tabella 3.7.1 ci permettono di affermare che la sincronizzazione non è un fattore dominante. Allo stesso tempo notiamo come la percentuale di tempo spesa in attesa della terminazione dei trasferimenti DMA superi il 50%, il che significa che il fattore dominante per le prestazioni è proprio la banda.

### 3.7.2 Performance

Di seguito vengono riportati i dati ottenuti facendo girare la versione ottimizzata del programma. Il parametro di riscontro della correttezza della versione ottimizzata è dato

dal confronto tra i valori di uscita della versione originale del programma con quelli della versione ottimizzata. I grafici mostrati nelle figura 3.19 e 3.20 si riferiscono alle prestazioni della *PlayStation3* e al CAB. L'algoritmo nella versione ottimizzata è stato fatto girare per 10000 iterazioni su di una matrice dati di $896 \times 896$ punti.

Come possiamo vedere l'andamento delle due curve è molto simile, la *PlayStation3* raggiunge il picco massimo di performance con 4 SPE, mentre il CAB ha il picco in corrispondenza dei 5 SPE. Entrambi passati il punto di picco si stabilizzano su di un valore vicino al picco ma leggermente inferiore. Confrontando i valori della Bandwidth vediamo che anche le bandwidth seguono un andamento simile a quello delle performance anche se con valori leggermente diversi.

Quello che vediamo è che le performance sono limitate superiormente dalla banda aggregata che si raggiunge, con una banda di 25.7 GB/s avevamo ipotizzato una performance di 35.5. GFlops, ma la bandwidth che riusciamo ad ottenere con l'implementazione dell'algoritmo attuale non supera i 21 GB/s, il che significa che le performance non possono essere superiori a 29 Gflops come mostrato dall'equazione 3.25.

$$\frac{35.5}{25.6} = \frac{x}{21.2} \rightarrow x = \frac{35.5 \times 21.2}{25.6} = 29\ Gflops \tag{3.25}$$

In Figura3.21 sono messe a confronto le curve delle performance con quelle della bandwidth e si vede chiarmente che hanno lo stesso andamento. Quello che possiamo dare per certo è che le performance sarebbero superiori in nel caso in cui i tempi di calcolo fossero più lunghi.

Possiamo concludere che la limitazione delle performance è data dalla bandwidth e che utilizzare un numero di SPE superiore a quello in cui vengono raggiunte le performance massime non ha alcun senso.

### Compensazione della massa

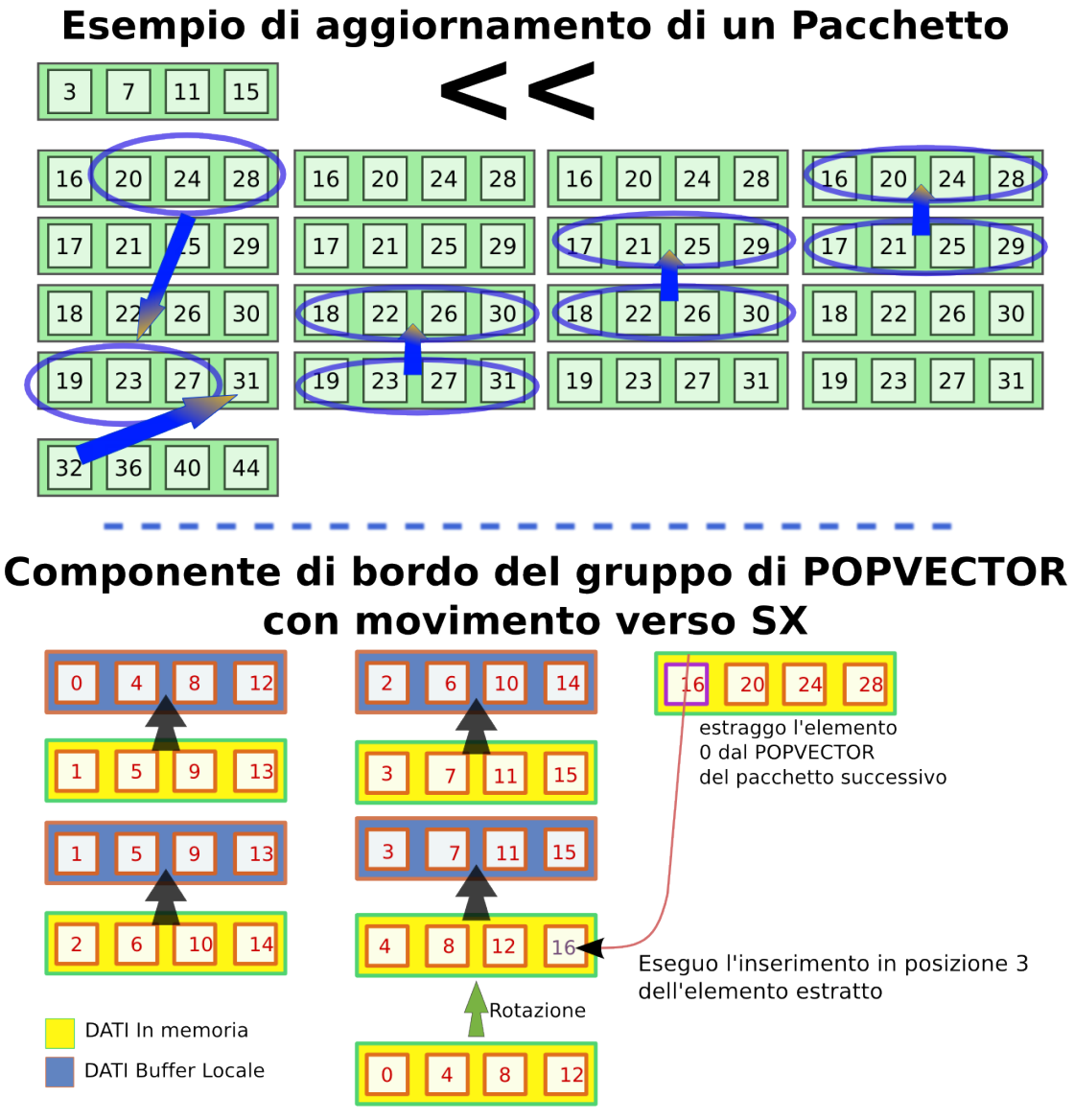



Figura 3.15: Aggiornamento componenti laterali con il nuovo schema dati

**Combo**

| Fase | Azioni | VarSet |
|---|---|---|
| | LOAD( C0,C2,C4) | A B C D |
| | LOAD_R( C1,C5,C8) | A B C D |
| | LOAD_L(C3,C6,C7) | A B C D |
| | COPY( Y-2, 0 ) | BUF 0 |
| | COPY( Y-2, 1 ) | BUF 1 |
| | COPY( Y-2, 2 ) | BUF 2 |
| | COPY( Y-2, 3 ) | BUF 3 |
| ITER | RHO( 0 ) | A |
| | RHO( 1 ) | B |
| | RHO( 2 ) | C |
| | RHO( 3 ) | D |
| | UPDATE( 0 ) | A |
| | UPDATE( 1 ) | B |
| | UPDATE( 2 ) | C |
| | UPDATE( 3 ) | D |
| | STORE(C0,C2,C4) | A B C D |
| | STORE_R(C1,C5,C8) | A B C D |
| | STORE_L(C3,C6,C7) | A B C D |

Figura 3.16: Operazioni della *Combo()*

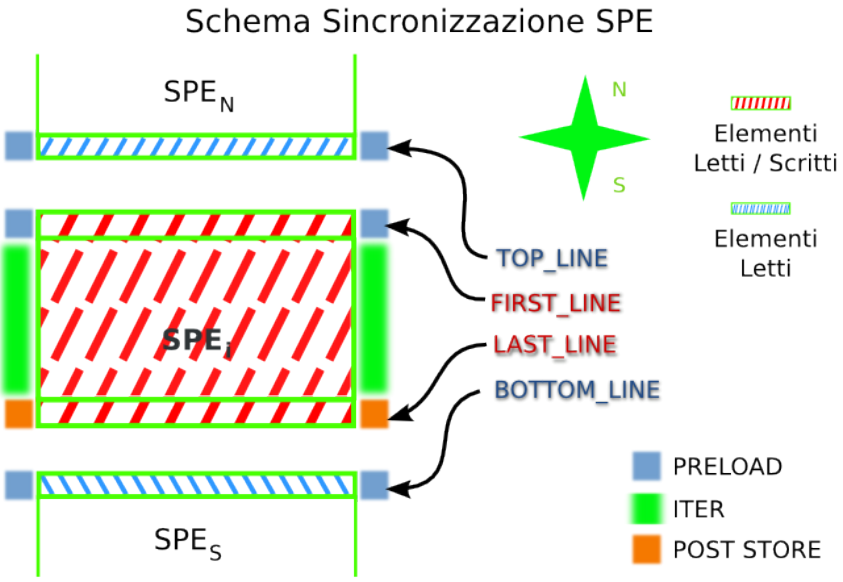



Figura 3.17: Schema di sincronizzazione tra SPEs vicini

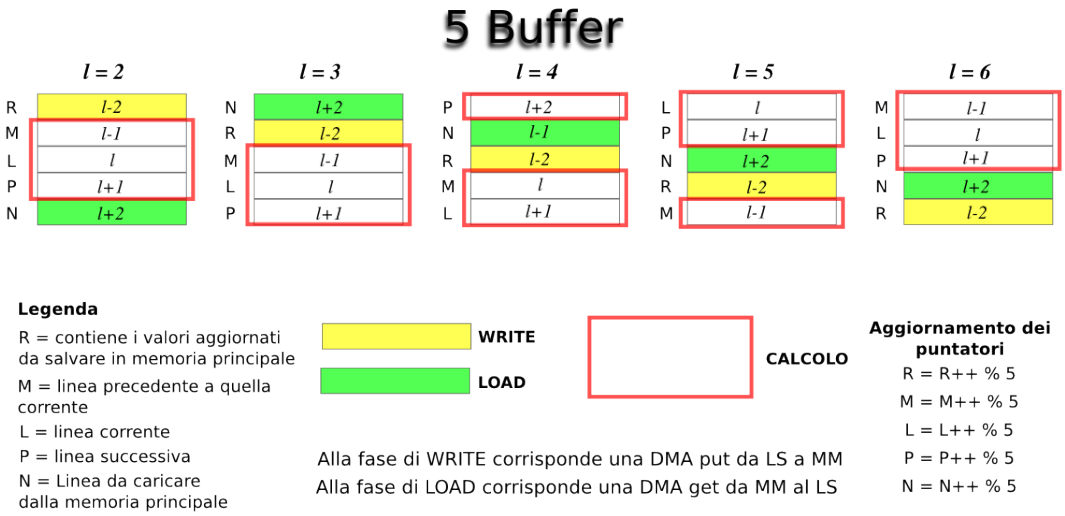



Figura 3.18: Schema MultiBuffer

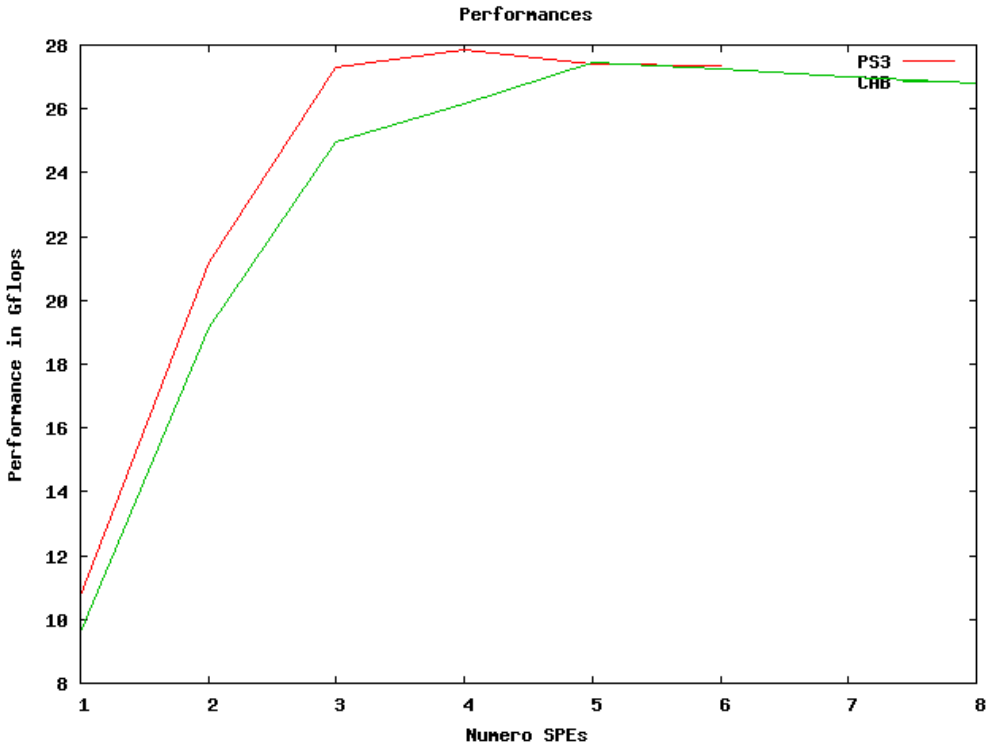



Figura 3.19: Grafico delle Performance

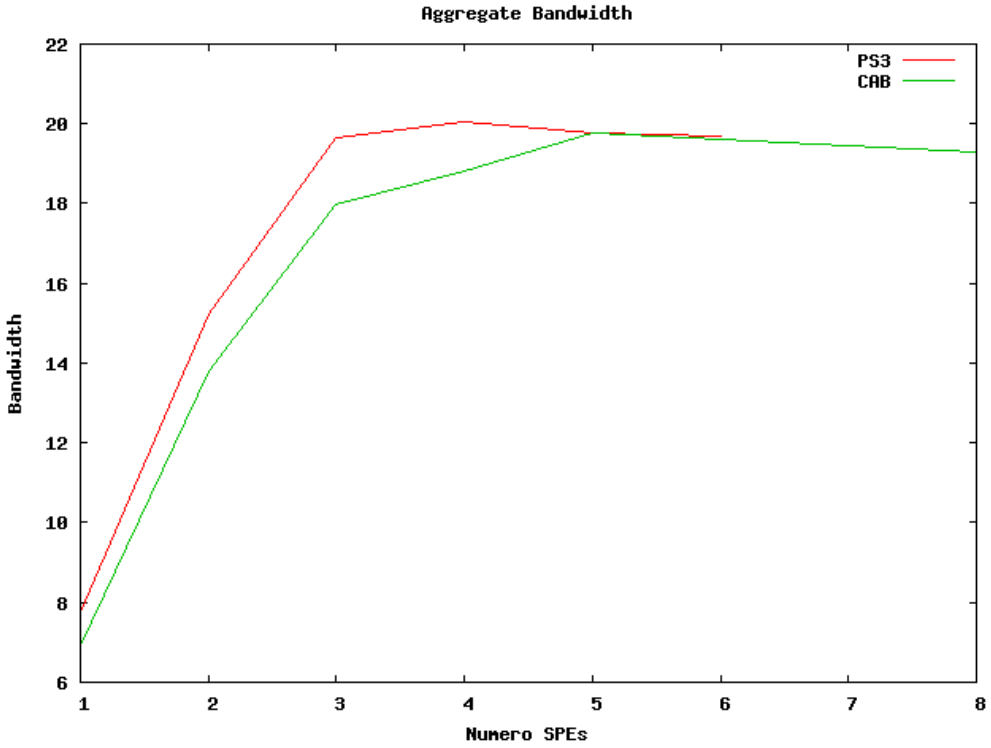



Figura 3.20: Grafico della banda passante

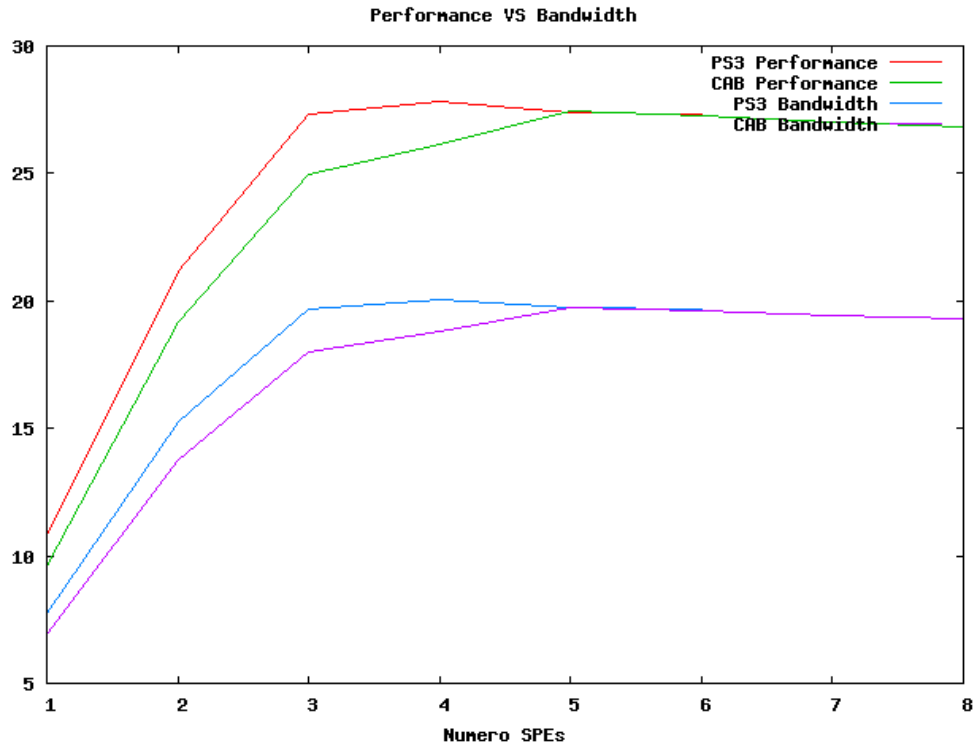



Figura 3.21: Curve performance a confronto con la bandwidth

# Capitolo 4

# Computing Accuracy

## 4.1 Cenni sulla rappresentazione dei numeri in virgola mobile

## 4.2 Come attualmente le architettura implementano il calcolo

## 4.3 double vs single

## 4.4 Come il Cell implementa il calcolo in virgola mobile

## 4.5 Conseguenze per l'algoritmo

## 4.6 Conlusioni

# Conclusioni