

IBM Octopiler

Compilatore per architetture scalabili

Roma, 10/04/2007

Redazione

Ing. Marco Pollini

Approvazione

Ing. Paolo Palazzari

Versione	Data	Redattore
1.0	10/04/2007	Ing. Marco Pollini

COMPILATORE PER ARCHITETTURE SCALABILI.....	1
1 INTRODUZIONE.....	1
2 PARALLELIZZAZIONE.....	1
2.1 SFRUTTARE IL PARALLELISMO.....	1
2.2 ASTRAZIONE DELLA MEMORIA.....	2
2.3 PARALLELIZZAZIONE TRAMITE OPENMP.....	2
3 SIMDIZATION.....	3
4 GENERAZIONE CODICE SPE OTTIMIZZATO.....	5
5 VALUTAZIONE PRESTAZIONI.....	6

1 Introduzione

Il compilatore IBM Octopiler si pone l'obiettivo di generare codice fortemente ottimizzato sfruttando una vasta gamma di parallelismo offerto da architetture di tipo Scale-Up e Scale-Out.

Per Scale-Up si intende la procedura di aggiungere risorse ad un singolo nodo di elaborazione di un sistema come, ad es., aggiungere memoria ad un singolo computer.

Per Scale-Out si intende invece la procedura di aggiungere più nodi ad un sistema, come, ad es., aggiungere nuovi computer ad una struttura di elaborazione distribuita.

Viene proposta una soluzione di compilatore "single source" per l'architettura Cell Broadband Engine. Questa piattaforma include un processore di tipo PowerPC e otto streaming-processors (SPE) ognuno dotato di memoria dedicata (256 KB) e unità DMA per il trasferimento dati. Inoltre ogni processore è dotato di più unità SIMD che possono processare da due valori floating-point a doppia precisione fino a valori di 16 byte per ogni istruzione.

Vengono proposte tecniche di ottimizzazione come il partizionamento dei dati e del codice, la generazione automatica di codice SIMD e altre ottimizzazioni specifiche per l'architettura Cell.

2 Parallelizzazione

2.1 Sfruttare il parallelismo

Il compilatore qui discusso genera, a partire da un codice C o Fortran, più eseguibili binari per l'unità PPE e i processori SPE, parallelizzando e vettorializzando (codice SIMD) il codice sorgente.

L'obiettivo è quello di generare codice altamente ottimizzato per i vari livelli di parallelismo fornendo un'astrazione delle complessità architetturali sottostanti e permettere all'utente di sviluppare applicazioni per un'architettura parallela con un'unica memoria condivisa.

Il punto centrale di questa strategia di compilazione consiste nel creare un'astrazione delle singole memorie degli SPE (Local Store). Ogni SPE può accedere direttamente solo alla propria memoria contenente sia dati che istruzioni e utilizza operazioni DMA per il trasferimento dati con la memoria centrale; ciò impone limiti significativi al programmatore soprattutto nel caso di elaborazione di grandi quantità di dati.

Il compilatore qui presentato adotta delle tecniche di ottimizzazione della gerarchia di memoria, di controllo via software degli accessi in cache e di partizionamento del codice che si basano sull'assunzione che tutti i dati risiedano in una memoria di sistema condivisa e che permettano il trasferimento automatico di codice e dati tra Local Store e memoria centrale preservando la coerenza di questi ultimi. Questa infrastruttura permette di sfruttare il parallelismo tra tutti gli elementi di elaborazione del processore Cell.

Attualmente l'implementazione avviene tramite direttive OpenMP (OpenMP pragmas) ma saranno supportate facilmente le esistenti tecniche di autoparallelizzazione nel framework del compilatore e potrebbero essere sviluppati anche altri paradigmi di parallelizzazione (es. UPC).

2.2 Astrazione della memoria

Software Caching. Come già detto un SPE può accedere direttamente solo al proprio Local Store, utilizzando i DMA per accessi esterni. L'utilizzo dei DMA per ogni richiesta di accesso alla memoria di sistema può degradare notevolmente le prestazioni del codice. Il software caching permette di riutilizzare i dati trasferiti da un singolo DMA tra accessi multipli che abbiano un'appropriata località spaziale e temporale. Per gestire il software caching è stata implementata una libreria runtime che memorizza temporaneamente e gestisce nel Local Store i dati prelevati dalla memoria centrale. Il compilatore identifica gli accessi alla memoria di sistema all'interno del codice SPE e li sostituisce con delle chiamate apposite alla libreria runtime .

Static Buffering. Quando un loop in esecuzione in un SPE deve accedere ad un array di dimensioni elevate, si cerca di spezzare il loop in blocchi più corti (loop tiling). Ciò permette di precaricare una porzione significativa dell'array in un buffer statico tramite un singolo DMA invece che effettuare chiamate successive alla libreria runtime del software caching per ogni elemento dell'array.

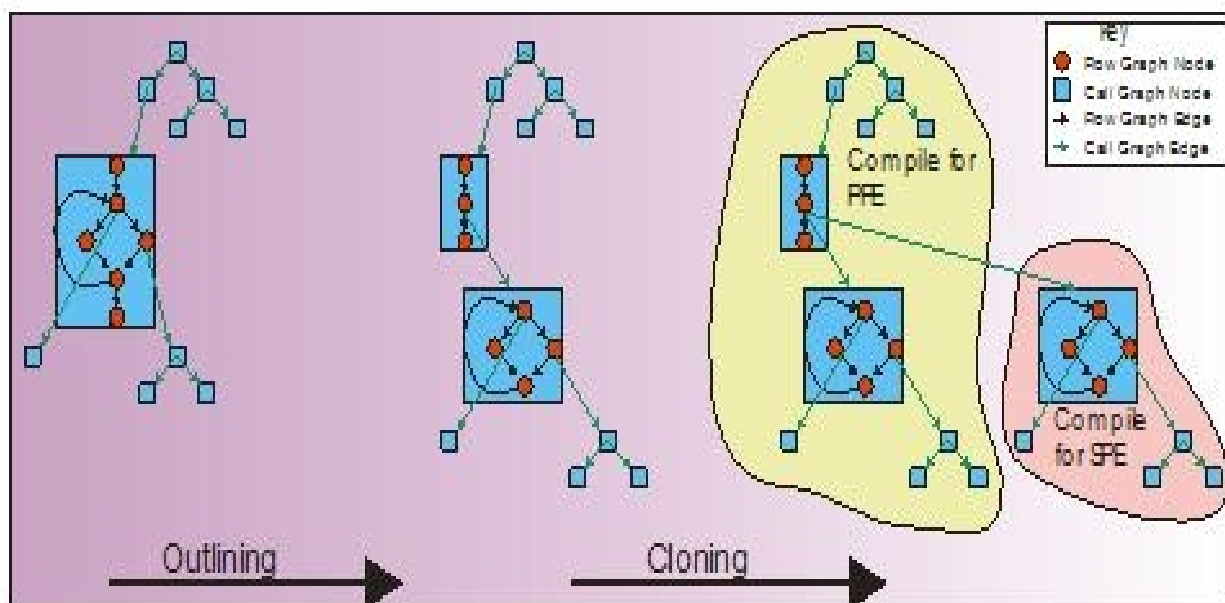
Questa è una delle ottimizzazioni che possono ridurre l'impatto sulle prestazioni dei trasferimenti DMA tra memoria di sistema e Local Store.

Code Partitioning. La tecnica del code partitioning permette di superare il limite imposto dal Local Store alla dimensione del codice eseguibile. Tramite overlay si può partizionare il codice e implementare un trasferimento automatico nel Local Store delle parti di codice richieste per l'esecuzione in un determinato istante di tempo. Il code partitioning è già integrato nel supporto per OpenMP.

2.3 Parallelizzazione tramite OpenMP

L'attuale implementazione del compilatore supporta la parallelizzazione attraverso direttive OpenMP inserite dal programmatore per denotare le sezioni di codice parallelo. Il compilatore genera un eseguibile binario PPE che dopo aver creato i thread SPE , inizia i trasferimenti DMA del codice SPE appropriato verso ogni Local Store e coordina l'esecuzione di tutti i thread.

La figura seguente illustra i passi di compilazione per OpenMP nell'architettura Cell.



Outlining. Ogni parte di codice parallelo viene delimitata e sostituita con una chiamata ad una funzione creata appositamente e la parte di codice selezionato viene copiato all'interno del corpo della nuova funzione.

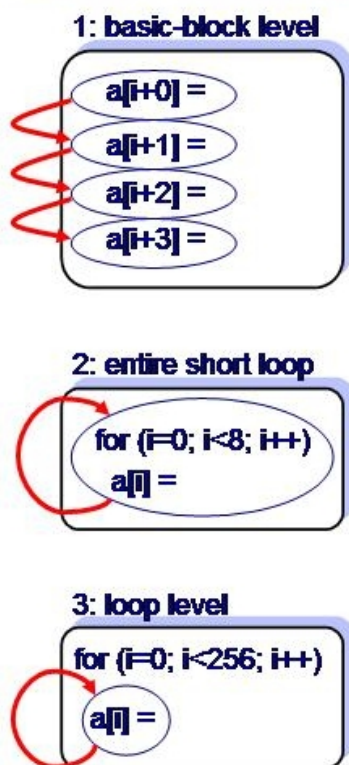
Cloning. Tutte le funzioni corrispondenti a regioni di codice parallelo vengono clonate in modo tale da avere due copie, una compilabile ed eseguibile sul PPE e l'altra compilabile ed eseguibile sugli SPE. La clonazione di tali funzioni prevede anche la clonazione di tutte le altre funzioni da queste invocate.

3 Simdization

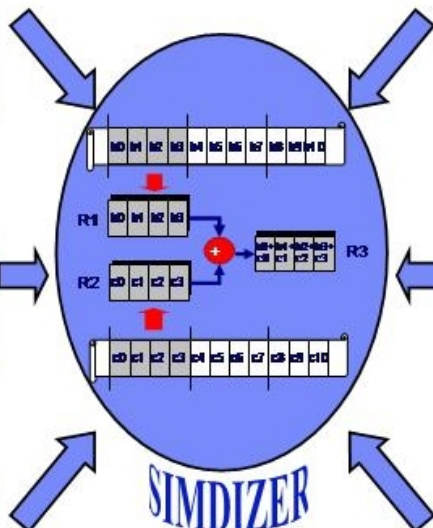
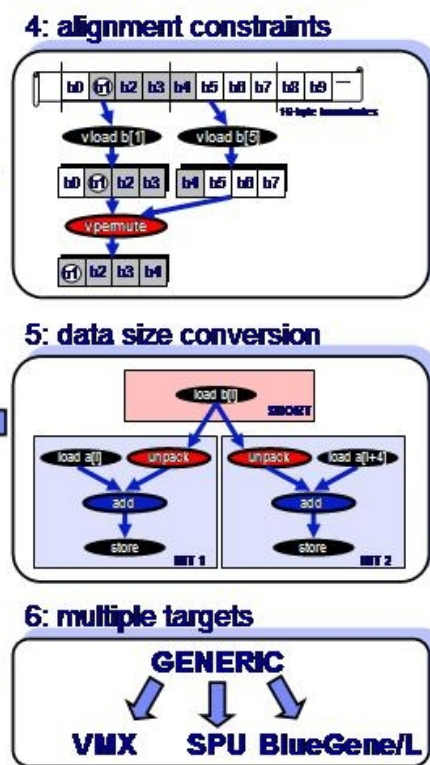
La vettorializzazione SIMD (anche SIMDization) è già implementata nei compilatori IBM XL e supporta più linguaggi di programmazione (C, C++, Fortran) e più architetture (SPE, VMX, BG/L).

Come mostrato nella figura successiva, la vettorializzazione coinvolge più operazioni diverse che interagiscono fra loro. Per prima cosa il parallelismo SIMD deve essere estratto dall'applicazione in esame, cosa che può richiedere l'analisi di un basic-block, di un loop o di entrambi. Una volta che il parallelismo è stato esplicitato, devono essere verificati i molteplici vincoli dettati dalle unità hardware SIMD, ad es. allineamento della memoria, lunghezza dei vettori e instruction set.

Extract Parallelism



Satisfy Constraints



L'approccio utilizzato è quello di una rappresentazione virtuale dei vettori in modo tale da estrarre il parallelismo SIMD da più parti nel codice. I registri virtuali non hanno vincoli di allineamento e posso avere dimensioni arbitrarie. Successivamente i vettori virtuali sono specializzati verso il basso, progressivamente, sui registri SIMD del processore utilizzato.

Segue un sommario delle principali caratteristiche della vettorializzazione SIMD

- Vettorializzazione SIMD a livello di loop
- Loop con induzione/riduzione delle variabili locali, computazione invariante, array con accessi di tipo stride-one.
- Loop con piccole costanti di iterazione
- Loop con tipi di dati diversi
- Vettorializzazione SIMD a livello di Basic-block
- Loop-unrolling
- Computazione su membri adiacenti di dati aggregati

- Packing di dati scalari
- Packing quasi-isomorfo
- Copia aggregata delle dichiarazioni
- Gestione allineamenti
- Reallineamento dei dati al volo (solo in caso di disallineamento relativo)
- Minimizzazione dell'overhead di reallineamento
- Versioning per allineamento relativo (for BG/L)
- SIMD prologue/epilogue per store disallineate
- Scalar prologue/epilogue per store disallineate per evitare falsa condivisione e per protezione della memoria
- Other optimizations
- Informazioni diagnostiche dei loop vettorializzabili e motivazioni su quelli non vettorializzabili
- Analisi di allineamento interprocedura
- Riconoscimento dell'idioma per sfruttare istruzioni hardware speciali
- Distribuzione del loop per isolare dichiarazioni vettorializzabili
- Recupero degli array per loop con puntatori aritmetici
- Analisi a runtime delle dipendenze

4 Generazione codice SPE ottimizzato

In aggiunta alle tradizionali ottimizzazioni vengono implementate le seguenti:

Dati scalari e unità SIMD. La maggior parte delle istruzioni degli SPE, incluse le istruzioni di memoria, sono di tipo SIMD operanti su dati a 128 bit. Di conseguenza il codice scalare presente in un programma deve essere adattato per essere eseguito correttamente nelle unità SIMD.

Da notare che tutte le operazioni di store scalari devono essere modificate in istruzioni read-modify-write poichè le istruzioni di scrivono in memoria blocchi di 16 byte alla volta; inoltre eseguendo un'allocazione stringente dei registri e allocando valori scalari temporanei in locazioni di memoria di 16 byte distinte, si può evitare gran parte di questo overhead a prezzo di un utilizzo maggiore di memoria.

Ottimizzazione dei branch. L'hardware degli SPE non è dotato di unità per la predizione dinamica dei salti, ma possiede un'istruzione apposita detta "branch hint instruction" che indica i probabili taken branch. Il compilatore inserisce questo tipo di istruzione quando opportuno.

Schedulazione, fetch e bundling delle istruzioni. Ogni SPE supporta il dual-issue di istruzioni indipendenti ma presenta alcune restrizioni legate alla struttura del codice. Per sfruttare il dual-issue e utilizzare entrambe le pipeline di esecuzione (even e odd), le istruzioni adatte alla even pipeline devono essere schedulate ad indirizzi pari del PC (Program Counter) e quelle per la odd pipeline ad indirizzi dispari. La struttura può essere modificata aggiungendo delle NOP. Un'altra ottimizzazione del compilatore riguarda l'istruzione fetch dalla memoria locale. Questa può incrementare le prestazioni di un codice che sfrutta la memoria in modo intensivo nel momento in cui sia le operazioni di load/store e il fetch delle istruzioni condividano la stessa porta di accesso alla memoria locale, con priorità delle load/store più alta rispetto al fetch delle istruzioni.

5 Valutazione prestazioni

L'incremento di prestazioni ottenuto con codice SPE ottimizzato è riportato in Figura 1.

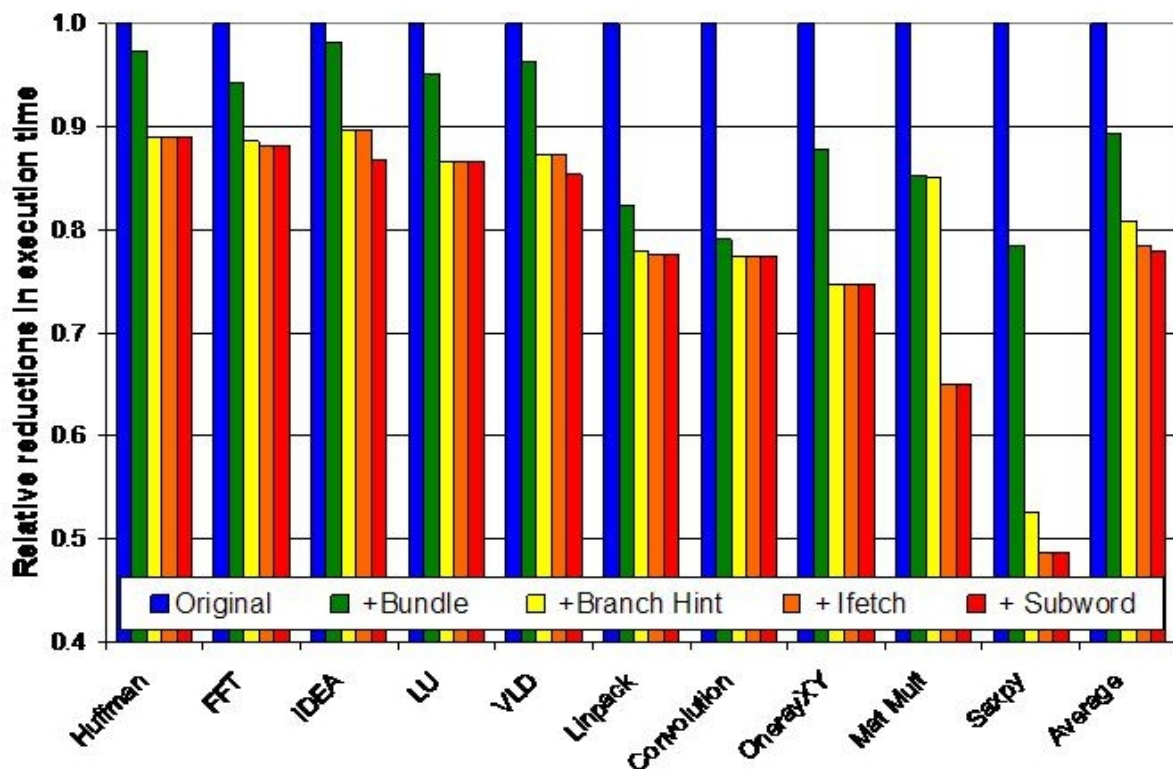


Figure 1. Incremento di prestazioni ottenuto con codice SPE ottimizzato.

La figura 1 rappresenta la riduzione nel tempo di esecuzione per ogni tipo di ottimizzazione effettuata relativamente alle prestazioni del compilatore originale (ottimizzazione livello O3, schedulazione per risorse SPE e modello di latenza). Viene riscontrata una riduzione media del 22%, con un range tra 11% e 51%.

L'incremento di prestazioni ottenuto dalla generazione automatica di codice SIMD è mostrato in Figura 2.

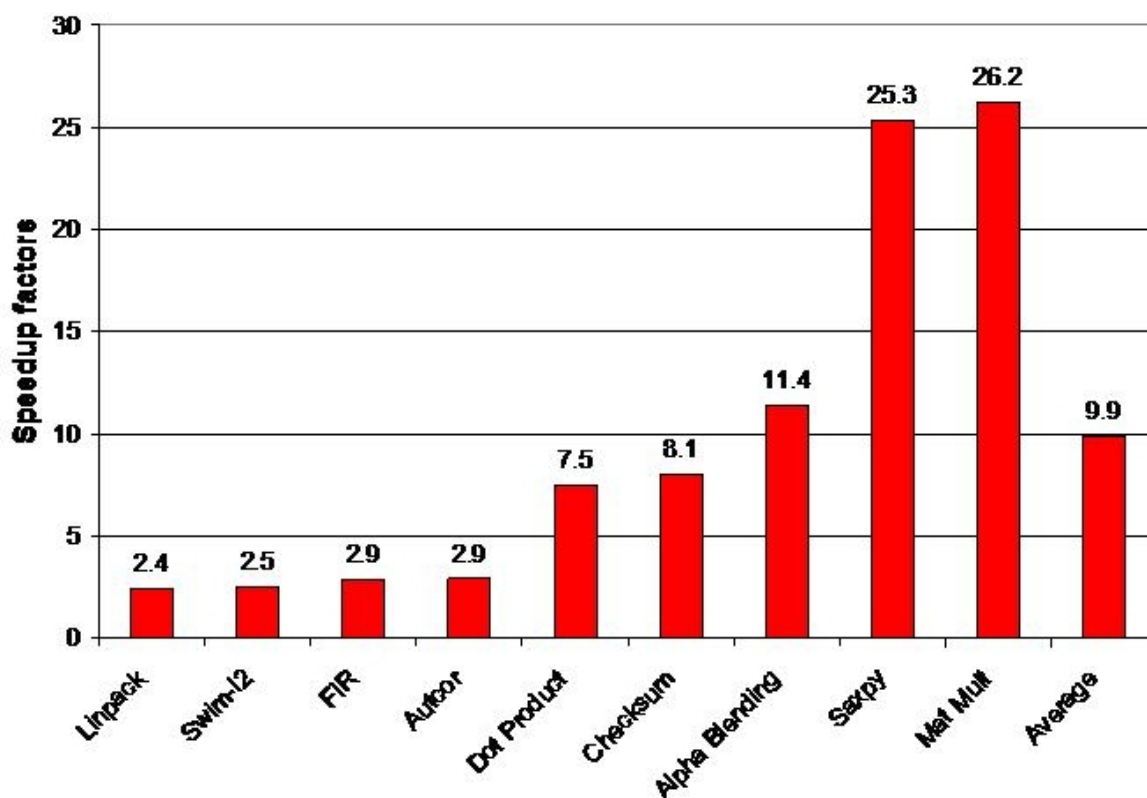


Figure 2. incremento di prestazioni ottenuto dalla generazione automatica di codice SIMD.

La Figura 2 presenta lo speedup ottenuto con codice vettorializzato SIMD automaticamente.

Lo speedup medio è pari a 9.9 e varia tra 2.4 e 26.2