

Barcelona Supercomputing Center

Cell Superscalar

Compilatore Cell source to source

Roma, 05/04/2007

Redazione

Ing. Marco Pollini

Approvazione

Ing. Paolo Palazzari

Versione	Data	Redattore
1.0	05/04/2007	Ing. Marco Pollini

1 INTRODUZIONE.....	1
2 COMPILATORE SOURCE TO SOURCE CELLSS.....	1
2.1 OBIETTIVI.....	1
2.1.1 <i>Struttura</i>	1
2.1.2 <i>Sintassi del compilatore e comportamento</i>	3
2.1.3 <i>Ambiente runtime</i>	4

1 Introduzione

Il progetto Cell Superscalar prevede di estrarre il parallelismo funzionale presente in un programma sequenziale sfruttando l'architettura parallela Cell Broadband Engine di STI.

Tramite semplici direttive inserite nel codice sorgente (dette “annotazioni”), il compilatore genera un codice C modificato, e una libreria dedicata sfrutta il parallelismo esistente costruendo a runtime un grafo delle dipendenze dei task in esecuzione. La schedulazione dei task e la gestione dei dati tra i vari processori viene eseguita a runtime. E' stata inoltre implementata una politica di gestione dei task che tenga conto della località dei dati al fine di ridurre l'overhead dovuto ai trasferimenti di questi ultimi.

2 Compilatore source to source CellSs

2.1 Obiettivi

Il programmatore si limita ad inserire, nel codice sequenziale, le direttive (simili a quelle utilizzate in OpenMP) prima della dichiarazione delle funzioni che devono essere eseguite su un dato SPE e la parte di codice così indicata viene separata dal codice principale e assegnata al SPE scelto.

Tuttavia la direttiva inserita non assicura che la parte di codice scelta venga parallelizzata ma solo che venga eseguita su un dato SPE; per realizzare il parallelismo Cell Superscalar prevede la creazione a runtime di un grafo delle dipendenze dei dati nel quale ogni nodo rappresenta un'istanza della funzione scelta precedentemente tramite annotazione e i rami tra i nodi rappresentano le dipendenze tra i dati. A partire da questo grafo si è in grado di schedulare l'esecuzione di più task indipendenti (nodi) su i diversi SPE.

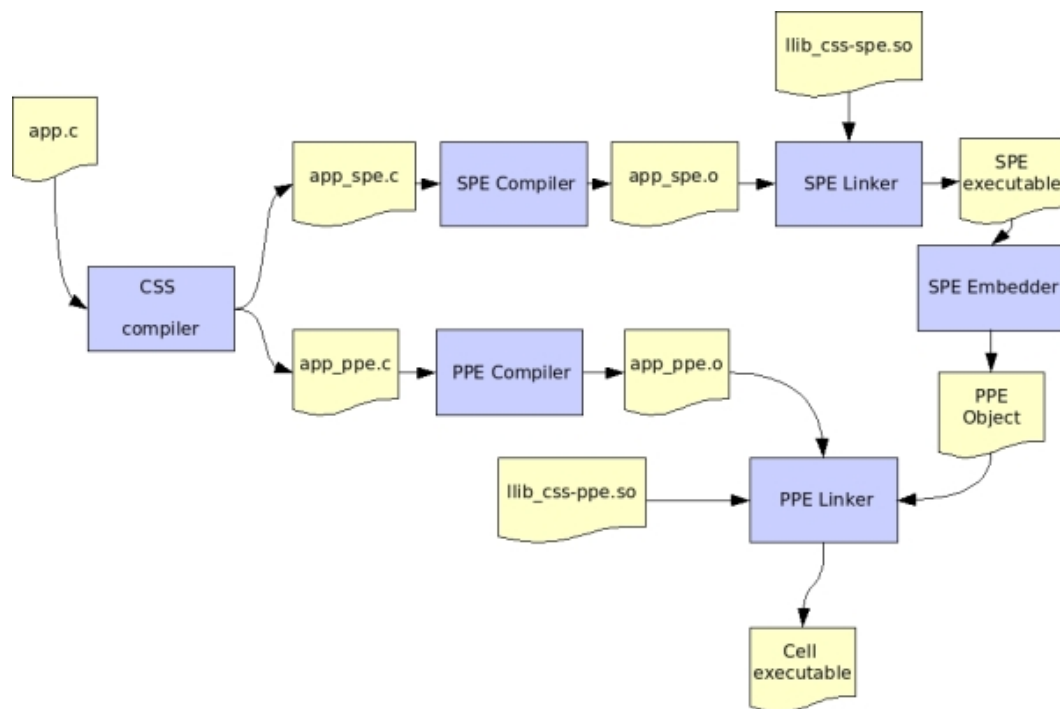
Per incrementare le prestazioni dell'applicazione sono utilizzate tecniche di ottimizzazione del tipo data dependency analysis, data renaming e data locality exploitation.

Cell Superscalar è composto da due componenti: un compilatore source to source e una libreria a runtime (runtime library).

Questo tipo di compilatore fornisce dunque un modello di programmazione flessibile di alto livello, mentre per ottimizzazioni vettoriali SIMD e sul codice di basso livello si fa riferimento al compilatore Octopiler di IBM.

2.1.1 Struttura

La figura seguente mostra il flusso che un codice utente segue per l'ottimizzazione e la generazione di un eseguibile CELL BE. Dato un codice sequenziale in linguaggio C tramite le direttive CellSs vengono generati due sorgenti C differenti.



Il primo file (app_ppe.c) corrisponde al main del programma e deve essere compilato con il compilatore PPE per generare un file oggetto PPE (app_ppe.o).

Il secondo file (app_spe.c) corrisponde al codice che viene eseguito nei vari SPE su chiamata del main, viene compilato con il compilatore SPE per ottenere il codice oggetto SPE e linkato con le librerie SPE per ottenere infine l'eseguibile SPE caricato nel Local Store. Comunque, al fine della corretta esecuzione del programma, è necessario inserire l'eseguibile SPE nel codice eseguibile PPE.

Per questo motivo l' SPE embedder genera il PPE object che viene poi linkato con il codice oggetto PPE generato precedentemente e le librerie PPE per generare l'eseguibile Cell finale.

Il codice main è eseguito nel PPE e ogni volta che viene trovata una parte di codice da eseguire in un SPE viene effettuata una chiamata alla runtime library che crea un nodo rappresentativo del task in esecuzione e controlla la presenza di dipendenze, dovute a nodi precedenti, aggiungendo dei rami tra essi. Se non ci sono dipendenze e se c'è un SPE libero, il task viene schedulato per l'esecuzione; il trasferimento dei dati necessari viene effettuato tramite DMA. La chiamata alla runtime library è non bloccante e dunque se il task non è pronto per l'esecuzione o non ci sono SPE liberi, l'esecuzione continua nel programma principale.

E' importante sottolineare che tutte queste operazioni (schedulazione dei task, verifica delle dipendenze e trasferimento dati) sono trasparenti all'utente, il quale si limita a indicare nel codice sequenziale quali parti devono essere eseguite su SPE.

2.1.2 Sintassi del compilatore e comportamento

La versione attuale del compilatore source to source supporta le seguenti caratteristiche:

1. Scelta della funzione da eseguire come task su SPE
2. Possibilità di specificare la direzione dei parametri della funzione
3. Possibilità di specificare array come parametri e le loro dimensioni

Nel codice main il compilatore CellSs inserisce le seguenti parti di codice:

1. Chiamate alla CellSs runtime library per l'inizializzazione e la finalizzazione delle funzioni
2. Registrazione in una apposita tabella delle funzioni annotate alla quale può accedere in seguito il task SPE.
3. Sostituzione delle chiamate originali alle funzioni annotate con chiamate alla funzione "Execute" della libreria runtime.

L'attuale implementazione del compilatore è basata sul compilatore Mercurium, originariamente progettato per OpenMP ma ragionevolmente portabile su CellSuperscalar.

Esempio di codice con annotazione (direttiva pragma):

```
#pragma css task input (a[],b[], index_i, index_j) output (c[])

void array_op(float a[N], float b[N], float c[N], int index_i, int index_j)

main(){
    ...
    array_op(A, B, C, i, j);
    ...
}
```

Assumiamo che la funzione array_op richieda elevate risorse di computazione e per questo venga annotata per essere eseguita su un SPE. La funzione ha 4 parametri di ingresso (a,b,index_i,index_j) e un parametro di uscita (c). Le parentesi { } dopo il nome del parametro stanno ad indicare che questi sono array.

2.1.3 Ambiente runtime

L'ambiente CellSs runtime è suddiviso in due parti: una parte è eseguita sul PPU e l'altra su ogni SPU. Nel PPU differenziamo il thread master dal thread helper. La differenza fondamentale con il codice utente originale sta nel fatto che il compilatore CellSs aggiunge le chiamate alla funzione `Execute` ogni volta che compare una funzione annotata nel codice originale. A runtime le chiamate a questa funzione sono responsabili del comportamento richiesto dall'applicazione nel processore Cell.

Ad ogni chiamata `Execute` il thread master compie le seguenti operazioni:

1. Aggiunta di un nodo nel grafo dei task
2. Analisi della dipendenza dei dati con task precedenti
3. Ridenominazione dei parametri ingresso/uscita: simile alla tecnica di register renaming. Per ogni chiamata a funzione che richieda la scrittura di un suo parametro, viene allocata una nuova area di memoria invece che sovrascrivere la locazione originale. Ciò permette di eseguire tale funzione indipendentemente da ogni altra precedente che legga o scriva tale parametro. Questa tecnica permette di ridurre la dipendenza dei dati utilizzando memoria aggiuntiva e aumenta la possibilità di estrarre il parallelismo.

Il thread helper è quello che gestisce la schedulazione dei task e monitorizza la loro esecuzione nei vari SPU. Dato un certo grafo dei task, il thread helper schedula i thread per l'esecuzione sui vari SPU

Lo schedulatore ha le seguenti linee guida:

1. Un task può essere schedulato solo se i task predecessori hanno completato l'esecuzione.
2. Per ridurre l'overhead dei DMA, più thread in parallelo (strand - "fila") sono inviati allo stesso SPU.
3. La località dei dati è sfruttata mantenendo i dati di uscita degli SPU nei rispettivi local store e schedulando nello stesso SPU quei task che riutilizzano questi stessi dati.

Il thread helper sincronizza gli SPU e comunica con loro utilizzando una specifica area di memoria del PPU per ogni SPU. Inoltre indica la lunghezza di ogni "fila" di task che deve essere eseguita e le informazioni relative ai dati di ingresso e di uscita dei task.

Gli SPU eseguono dei loop in attesa dei task da eseguire. Ogni volta che una fila di task è schedulata per l'esecuzione, l'SPU esegue un DMA per caricare i dati di input, esegue i vari task e scrive i dati di output nella memoria del PPU. Ogni SPU si sincronizza con il PPU per indicare la fine della fila di task utilizzando un'area di memoria riservata nel PPU, come detto sopra.

