UNIVERSITA' DI PISA
FACOLTA' DI SCIENZE MATEMATICHE FISICHE E NATURALI

Corso di Laurea Specialistica in Tecnologie Informatiche

Tesi di Laurea

Esecuzione di codice intermedio su GPU: le VM incontrano i processori grafici

Candidato

Giacomo Righetti

Relatori

Prof. Antonio Cisternino
Dott. Cristian Dittamo

Controrelatore

Prof. Marco Vanneschi

Anno accademico 2008/2009

Ringraziamenti

Indice

Introduzione 6

Parte I - Lo stato dell'arte 8

Capitolo 1 9

1.1 Applicazioni parallele 9

1.2 GPU Computing 12

1.3 Modelli architetturali 14

1.3.1 La macchina di Von Neumann 14

1.3.2 La macchina SIMD 15

1.4 Modelli di parallelismo data-parallel 16

1.5 Parametri per la valutazione delle prestazioni di applicazioni parallele 17

Capitolo 2 19

2.1 Architetture GPU: modelli hardware e di memoria 19

2.1.1 AMD 19

2.1.2 Nvidia 26

2.2 AMD vs. Nvidia 32

Capitolo 3 34

3.1 Modelli di programmazione GPU 34

3.1.2 AMD 34

3.1.3 Nvidia 45

3.2 Modelli di memoria GPU 51

3.2.1 AMD 51

3.2.2 Nvidia 51

3.3 AMD vs Nvidia 52

Capitolo 4 54

4.1 Altre API per sviluppo di applicazioni data parallel 54

4.1.1 RapidMind 54

4.1.2 OpenCL 60

4.1.3 Accelerator 70

Parte II - Strumenti 73

Capitolo 5 74

5.1 Metaprogrammazione 74

5.2 Metadata e Reflection 76
5.2.1 Metadata 76
5.2.2 Reflection 78
5.3 Strongly typed execution environments 78
5.3.1 CLI, Common Language Infrastructure 81
5.4 Assembly 86
5.4.1 Introduzione 86
5.4.2 Struttura 87
5.4.3 Formato binario 88
5.5 Layout dei tipi in memoria e P/Invoke 89
5.6 Task Parallel Library 92
Parte III - PBricks 96
Capitolo 6 97
6.1 PBricks API 97
6.1.1 Esempio di utilizzo di PBricks 97
Capitolo 7 112
7.1 PBricks internals 112
7.1.1 Architettura 112
7.2 La fase di esecuzione: sequenziale e multicore 113
Capitolo 8 116
8.1 Analisi 116
8.1.1 L'analisi dei dati 118
8.2 L'analisi del codice MSIL 119
8.3 Gestione della memoria: Global buffer, LDS, GDS 131
8.3.1 Global Buffer 131
8.3.2 LDS & GDS 135
Capitolo 9 138
9.1 Esecuzione 138
9.2 CALInterop 146
Capitolo 10 148
Risultati sperimentali 148
Capitolo 11 156

Conclusioni e sviluppi futuri 156
Appendice 158
A.1 Esempio di codice Brook+/CAL 158
A.2 Esempio di codice CUDA 161
A.3 Funzioni per gestire scritture/letture dal global buffer 162
A.4 Il sorgente del Mandelbrot prodotto da PBricks 164
A.5 Costruzione tabella dei salti “while” 165
A.6 Goto e Switch 169
A.7 Esempio di struttura prima della fase di Scan 173
Bibliografia 175

Introduzione

I sistemi di calcolo si sono significativamente evoluti dal primo computer della storia umana, e sono diventati pervasivi in ogni aspetto della vita umana. Come ogni altra scienza e disciplina, l'informatica ha attraversato diverse età nelle quali molte idee sono state esplorate e nel tempo è andata affermandosi una nozione ampiamente accettata di computer system in parte perchè era ragionevole, in parte per puro caso (le architetture x86 e i pc sono diventati lo standard per ragioni che vanno al di là di mere considerazioni tecniche). Alla fine degli anni '90 sembrava che i computer avessero raggiunto una stabilità e maturità capace di separare le competenze dei programmatori da quelle degli ingegneri di sistemi. A quell'epoca sembrava che i computer avrebbero nascosto il parallelismo dal software adottando tecniche come le architetture super-scalari fornendo un ben definito modello di calcolo basato sull'architettura di Von Neumann. Le CPU multi-core hanno leggermente cambiato questa assunzione, benchè siano diventate presto un mezzo per eseguire processi su un sistema distribuito costruito all'interno di una singola macchina e il modello è rimasto essenzialmente quello di Von Neumann.

Il bisogno di processare in modo efficiente e specializzato mesh 3D ha silenziosamente introdotto un modello di computazione diverso da quello di Von Neumann nei tradizionali PC destinati al mercato videoludico. Siccome il 3D richiede la trasformazione di un elevato numero di vertici e superfici, le schede video hanno iniziato ad esporre un'API di complessità crescente per permettere ai programmatori di controllare le computazioni eseguite dalle schede grafiche per ottenere effetti grafici in real time. Questo processo è continuato fino al punto in cui i processori grafici sono stati chiamati Graphical Programming Units (GPU) e la scheda video è divenuta un array di piccoli core Turing-equivalenti capaci di computazioni general purpose. Si è rivelato naturale cercare problemi con una struttura simile a quella del rendering 3D e per i quali fosse possibile sfruttare il notevole potere computazionale fornito da questi processori specializzati, conducendo i produttori di schede video a progettare framework per supportare questo nuovo trend. Il modello di Von Neumann è diventato infine solo una delle architetture disponibili quando l'architettura Cell BE è stata rilasciata, dove anche la CPU ha esposto il proprio parallelismo interno ai programmi in un modo non trasparente (le CPU multi-core sono a tutti gli effetti una versione più efficiente di un sistema multi-processore).

I linguaggi di programmazione sono stati progettati ispirandosi a teorie quali la calcolabilità e la complessità e i modelli Turing-based (il naturale complemento delle architetture di Von Neumann), rendendo difficile sfruttare la potenza di calcolo resa disponibile da questi nuovi processori. Un obiettivo ambizioso è lo sviluppo di un'infrastruttura di programmazione capace di distribuire in maniera automatica le computazioni in base alla particolare architettura usata per l'esecuzione del software.

Oggi giorno le macchine virtuali come il .NET CLR sono usate con successo in molte aree. Tuttavia forniscono un modello ben noto di computazione sequenziale che non può essere direttamente mappato su architettture GPU, poichè queste ultime implementano un differente modello computazionale basato sul paradigma data parallel. D'altra parte sarebbe comodo che questo mapping fosse realizzato perché renderebbe possibile usufruire in un ambiente managed (che offre già di per sè diversi vantaggi per gli sviluppatori) della grande potenza di calcolo messa a disposizione dalle GPU.

Finora l'ambito di ricerca è stato ristretto ad ambienti nativi (C, C++) e manca una sorta di collante, un ideale punto d'incontro dei due mondi. L'unico progetto di ricerca è Accelerator di Microsoft Reasearch, ma che come vedremo presenta alcune lacune (paragrafo 4.1.3). D'altra parte la stessa Microsoft sta investendo in questa direzione: basti pensare alle Parallel Extensions (si veda al riguardo il paragrafo 5.6) che saranno introdotte nel .NET framework 4.0 e faciliteranno lo sviluppo di applicazioni scalabili su architetture multi-core; la naturale prosecuzione di questo trend sembra essere appunto quella di esporre alla ormai diffusa comunità di sviluppatori in ambiente managed un sistema di programmazione efficace per GPU, che possa permettere lo sviluppo e il debugging dei soli algoritmi sequenziali e demandare in seguito, dopo averne verificato la correttezza e l'aderenza ai requisiti funzionali, l'esecuzione parallela su GPU.

In questa tesi è stato realizzato un metaprogramma in grado di generare una versione per GPU di una computazione .NET. Questo metaprogramma effettua analisi di byte code e genera un programma semanticamente equivalente. L'approccio seguito si basa sulla trasformazione di programmi compilati in formato binario (Assembly). Il programmatore sviluppa la sola versione sequenziale, ritardando la decisione di cosa, quando e come parallelizzare a runtime. La tesi è culminata nello sviluppo di *PBricks*, una libreria che realizza un mapping tra il CLR e le GPU individuando un preciso modello di memoria e un traduttore da bytecode MSIL a ATI IL (ma la soluzione è estendibile a tutti quegli ambienti d'esecuzione virtuale aventi capacità di introspezione, come ad esempio la JVM di Java, ed è al tempo stesso sufficientemente generica per permettere la generazione di codice eseguibile dalle GPU di Nvidia). Al tempo stesso l'idea alla base dei PBrick può essere ulteriormente generalizzata per permettere ad esempio l'esecuzione su cluster e sfruttata per costruirvi al di sopra un layer di schedulazione di computazioni basate su macchine virtuali su architetture eterogenee, e di questo si parlerà negli sviluppi futuri.

La presentazione del suddetto lavoro è articolata in tre sezioni: nella prima (riguardante lo stato dell'arte) si inquadrerà maggiormente il focus applicativo, con un primo capitolo riguardante le applicazioni parallele, il gpu computing, i principali modelli di parallelismo e i parametri per la valutazione delle loro prestazioni, un secondo capitolo dedicato alla descrizione delle GPU attualmente in commercio da diversi livelli di dettaglio (modello hardware, modello d'esecuzione e di memoria), un terzo capitolo dedicato alle diverse astrazioni di programmazione costruitevi al di sopra, e un quarto che coprirà le principali API alternative per la programmazione di applicazioni data parallel; nella seconda, inerente gli strumenti utilizzati per lo sviluppo del meta-programma obiettivo di questa tesi, verranno introdotti i concetti di base della metaprogrammazione, reflection e generazione dinamica del codice (capitolo quinto). Nella terza sezione infine si descriverà PBricks presentando prima un'esempio d'applicazione e il modello di programmazione (capitolo sesto), analizzando poi maggiormente nel dettaglio le sue diverse componenti e le principali problematiche affrontate (capitoli 7, 8 e 9). Al termine saranno presentati alcuni risultati sperimentali, discussi i possibili sviluppi futuri e si tireranno le somme sul lavoro svolto (capitoli 10 e 11).

Parte I - Lo stato dell’arte

Capitolo 1

In questo capitolo s'introdurrà il concetto di applicazione parallela e le differenze rispetto a una tradizionale applicazione sequenziale; si fornirà una panoramica dei diversi settori e ambiti di utilizzo, specialmente quelli di stampo scientifico, fornendo esempi concreti di applicazioni parallele. In seguito sarà poi introdotto il concetto di GPU computing e descritto il modello architetturale alla base di una GPU, ossia la macchina SIMD, con l'obiettivo di mostrarne le differenze rispetto alla macchina di Von Neumann, il modello secondo il quale è organizzata la maggior parte dei moderni elaboratori sequenziali, anch'essa descritta nel presente capitolo.

Nonostante quello che il termine GPGPU[1] voglia far credere non tutte le forme di parallelismo sono particolarmente adatte per questo tipo di macchina: si parlerà perciò del paradigma di programmazione data parallel, che invece è il modello di parallelismo che maggiormente si adatta alle caratteristiche di una GPU e che può garantire un miglior beneficio in termini di prestazioni. Al termine del capitolo saranno poi introdotti e discussi alcuni parametri per la valutazione di applicazioni parallele, grazie ai quali sarà possibile al termine della tesi caratterizzare in modo qualitativo e quantitativo i risultati ottenuti.

1.1 Applicazioni parallele

A differenza di un'applicazione sequenziale che risolve un problema tramite un algoritmo le cui istruzioni sono eseguite in sequenza e un modello computazionale caratterizzato da un singolo processore, un'applicazione parallela risolve un problema tramite un algoritmo le cui istruzioni sono eseguite in parallelo secondo un modello computazionale che prevede processori multipli e relativi meccanismi di cooperazione. L'algoritmo deve sfruttare in maniera efficiente il parallelismo intrinseco nella definizione del problema, per renderne più veloce l'esecuzione. Questo tipo di applicazioni possiede delle ben note modalità di sviluppo e problematiche collegate. Per maggiori informazioni riguardo alle metodologie di sviluppo di applicazioni parallele si consulti [49].

Le esigenze applicative sono duplici:

- Ottenere computazioni che terminano in un *tempo ragionevole*
- Eseguire nello stesso tempo e con lo stesso algoritmo *problemi più complessi* (es. simulazione su dati più grandi, o con time-step più piccolo)

Negli ultimi anni la domanda di prestazioni maggiori, e della quantità di dati da memorizzare e gestire è cresciuta esponenzialmente: sia da parte delle aziende sia dagli utenti. Le prime per la necessità di aumentare il numero di applicazioni, sempre più di tipo multi-thread, da eseguire contemporaneamente in diversi ambiti quali data-mining, sicurezza e computer graphics. In quest'ultimo settore l'aumento della necessità di potenza di calcolo è stato particolarmente evidente: basti pensare ai computer games, alla realtà virtuale utilizzata nella computer-aided surgery, ma anche alla crescente attenzione posta

[1] General-Purpose computation on Graphics Processing Units

sulle interfacce grafiche nelle moderne applicazioni. Per quanto riguarda gli utenti fino a pochi anni fa essi utilizzavano il personal computer principalmente per scrivere documenti, navigare in Internet, inviare email. Oggi, invece, eseguono applicazioni per manipolare video digitali, fotografie, giocano con computer games che si spingono fino alla simulazione della vita reale.

Applicazioni scientifiche e d'ingegneria richiedono l'accesso a piattaforme più complesse, con centinaia di processori nel campo denominato HPC[2] che offre un nuovo strumento di calcolo per sperimentare, teorizzare/modellare, computare. Tra i vantaggi offerti dall'aumento di prestazioni che questa tecnologia porta vi è la possibilità di giocare con i parametri della simulazione per studiare nuove soluzioni, o eventi su cui non si abbiano raccolto dati sperimentali (es. simulare eventi catastrofici); la possibilità di ripetere particolari eventi simulati; studiare sistemi quando non esistono teorie testate.

Tra i grandi utilizzatori della tecnologia HPC si pone certamente l'industria automobilistica. Il reparto corse della scuderia Ferrari a Maranello adotta la nuova piattaforma HPC Server 2008 di Microsoft abbinandola alla precedente infrastruttura composta da cluster Linux.

Principali usi della simulazione:

- Aerodynamics (in modo simile all'industria aerospaziale)
- Crash simulation
- Metal sheet formation
- Noise/vibration optimization

Principali benefici:

- Riduzione del time-to-market di nuove automobili
- Aumento della qualità dei prodotti
- Riduzione della necessità di costruire prototipi
- Più integrazione ed efficienza nell'intero processo manifatturiero

Esistono applicazioni che storicamente richiedono maggiore capacità di quanto i computer sequenziali possano attualmente fornire. Si tratta dell'*e-Science*: esempi di problemi scientifici che non possono essere oggi risolti in tempi ragionevoli sono la modellazione di grandi strutture di DNA, la modellazione e previsione di fenomeni meteorologici globali, e di quelli correlati (es. inquinamento, maree, ecc.); la ricerca di riserve energetiche; la modellazione del movimento di corpi celesti (N-bodies); i problemi di cosmologia, con grandissime scale spaziali e temporali.

Le applicazioni parallele hanno avuto diffusione non solo per il calcolo scientifico, ma anche in ambito business: si considerino, ad esempio, le applicazioni per piattaforme parallele data intensive come web server, DBMS[3], motori di ricerca, data mining.

---

[2] High Performance Computing

[3] Data-Base Management System

Nel caso dei Web Search Engines, il parallelismo è sfruttato nella fase di crawling, in quella d'indicizzazione e analisi dei documenti, nel query engine e nel processo di ranking delle pagine [62].

Nel caso del Data Mining, disciplina che trae origine da vari campi quali machine learning/AI, pattern recognition, statistica, database, e visualizzazione di dati scientifici, si tratta tipicamente di applicazioni commerciali/business. Esse prevedono l'esplorazione e l'analisi, in modo automatico o semi-automatico, di grandissime quantità di dati per scoprire ed estrarre pattern significativi e regole che devono essere innovative, valide, potenzialmente utili, comprensibili. A causa della mole dei dati che non sempre possono risiedere contemporaneamente tutti in memoria principale, le applicazioni devono gestire trasferimenti con la memoria secondaria durante la fase di calcolo. Alcuni dei più importanti task del data mining sono:

- La classificazione
- Il clustering
- La scoperta di regole associative

Le tecniche tradizionali (sequenziali) possono non essere adatte a causa della mole dei dati, spesso aventi alta dimensionalità e della loro natura eterogenea e distribuita.

Applicazioni di visualizzazione e grafica computazionale sono usate massicciamente da parte di recenti produzioni cinematografiche per il rendering delle scene o l'applicazione di effetti in post-produzione [63].

La tecnologia del Grid Computing[4] [52], consentendo la condivisione sicura e l'utilizzo coordinato di risorse distribuite geograficamente, mette a disposizione degli utenti un'enorme potenza di calcolo, attraverso l'acquisizione di un elevato numero di risorse computazionali difficilmente disponibile localmente. In particolare l'interconnessione dei supercalcolatori e dei cluster disponibili presso i diversi centri realizza una struttura globale che consente l'esecuzione delle applicazioni candidate alla risoluzione di complessi problemi *computationally intensive* e *data intensive*. Il calcolo su griglia, d'altra parte, impone la realizzazione di nuove applicazioni distribuite (o modifiche sostanziali alle applicazioni parallele tradizionali), di strumenti di sviluppo adeguati e la disponibilità di nuovi sistemi per la gestione dell'esecuzione delle applicazioni stesse. Fondamentalmente il modello computazionale utilizzato per la programmazione su griglia è il tradizionale modello a scambio di messaggi. In particolare le applicazioni parallele multi-sito consistono di più componenti (subjob) eseguiti in parallelo su uno o più processori di uno o più cluster o supercalcolatori presso differenti siti. Tali applicazioni possono così accedere efficacemente ad un numero di risorse computazionali molto più ampio di quello a disposizione utilizzando un qualsiasi supercalcolatore o cluster singolo. Un'opportuna distribuzione dei subjob tra le risorse disponibili consente a queste applicazioni di beneficiare in termini di prestazioni di questa potenza computazionale aggregata nonostante l'overhead addizionale introdotto dalle comunicazioni (tra i subjob) che utilizzano le reti più lente (per esempio WAN).

[4] Calcolo su griglia

Uno dei primi progetti di calcolo distribuito è stato il SETI@home[5] che utilizza la potenza dei desktop computer per analizzare i segnali elettromagnetici provenienti dallo spazio alla ricerca di eventuali prove di trasmissioni radio provenienti da intelligenza extraterrestre. Per una lista di progetti di calcolo distribuito si faccia riferimento a [50, 51].

Recentemente si sta assistendo allo sviluppo delle piattaforme per il Cloud computing e dell'HPC Pervasive computing [35]

1.2 GPU Computing

GPGPU sta per *General-Purpose computation on Graphics Processing Units*, anche noto come *GPU Computing*. Il termine GPGPU è stato coniato nel 2002 da Mark Harris, un ricercatore Nvidia nel campo della real-time computer graphics. In quel periodo le GPU di Nvidia introdussero l'architettura a shader unificati (ossia a livello hardware non c'era più distinzione tra le unità che eseguivano il passo di processing dei vertici da quello di processing dei pixel) per supportare shader programmabili per rendering in tempo reale di alta qualità, ma divenne presto evidente che i core programmabili sarebbero stati usati per esprimere computazioni diverse da quelle strettamente inerenti la grafica 3D. Infatti, usando linguaggi di shading ad alto-livello, svariati algoritmi data-intensive furono portati sulle GPU. Problemi come il protein folding, stock options pricing, SQL queries, e MRI reconstruction ottennero rimarcabili speedup grazie all'uso delle GPU. Vi erano però alcuni svantaggi significativi: in primo luogo, era richiesto ai programmatori una grande conoscenza delle API grafiche e delle architetture GPU; i problemi dovevano essere espressi in termini di coordinate di vertici, texture e programmi di shading, aumentando notevolmente la complessità di sviluppo delle applicazioni. Alcune caratteristiche base della programmazione, come le letture e scritture in posizioni arbitrarie della memoria non erano supportate, restringendo di fatto il modello di programmazione. Non era previsto un supporto per calcoli in doppia precisione, il che significava che alcune applicazioni scientifiche non potevano essere fatte girare sulle GPU. Da allora il GPGPU si è sviluppato enormemente ed è passato dall'essere considerato un'oscura pratica seguita da pochi a valida alternativa al supercomputing tradizionale (facente uso di architetture più sofisticate e componentistica migliore degli usuali computer al fine di poter svolgere con maggior efficienza le elaborazioni assegnate) da sviluppatori e ricercatori. La stessa Nvida è stata la prima a rilasciare una scheda priva di uscita video ed espressamente votata al supercalcolo (Tesla). I produttori di GPU stanno attualmente provando a fare leva su questa osservazione per creare una tool chain di programmazione capace di schedulare computazioni simili a quelle usate per lo shading (computazioni datastream) sui core delle GPU. L'attuale trend tende a distribuire calcoli di diverso tipo su CPU e GPU, vedendo i due sistemi non in antitesi ma complementari l'uno con l'altro, e anche se le GPU diverranno sempre più sofisticate col passare del tempo saranno sempre più adatte per certi scopi piuttosto che per altri: le attuali CPU sono ottimizzate per eseguire un numero minore di thread che eseguono codice in cui la probabilità di branch può anche essere elevata e caratterizzati da un mix di istruzioni diverso (in questo senso sono dette *control flow intensive*), laddove

[5] Search for Extra Terrestrial Intelligence

le GPU eccellono in task massicciamente multi-threaded, caratterizzati dalla presenza di pochi branch e da lunghe sequenze di istruzioni computazionali (in questo senso sono dette *computation intensive*). Il perché di questa distinzione risulterà più chiaro dopo l'analisi del modello hardware delle GPU (si veda per questo il capitolo 2).

Il modello generale di programmazione di una GPU è chiamato *Stream Computing* nel quale i dati arrivano sotto forma di stream (flusso di dati tipato e omogeneo) elaborato da una matrice di computing core per applicazioni di tipo CPU intensive in modalità SIMD[6]. Il non-determinismo intrinseco nella scelta di come i dati presenti negli stream sono processari dai core è dove questo modello computazionale rompe lo schema architetturale sequenziale di Von Neumann (descritto nel paragrafo 1.3.1). La computazione di ogni core è guidata da un programma riferito comunemente come uno *shader* (nella tradizionale terminologia 3D) o *kernel*. L'infrastruttura della GPU è responsabile per l'assegnazione dei core ai kernel, e ogni istanza in esecuzione di un kernel è chiamata *thread* (termine che non deve essere confuso con i thread definiti a livello del sistema operativo). Ogni thread ha associato un insieme di locazioni di output nella memoria della GPU riferite come il *domain of execution* (dominio d'esecuzione). I thread sono un'astrazione fornita dall'infrastruttura di programmazione, a un livello più basso, l'hardware grafico schedula[7] l'array di thread nel pool di processori fisici fino a che i dati di input non sono alla fine tutti processati. E' possible schedulare differenti kernel su una GPU alla volta, permettendo di processare diversi stream di input da una singola applicazione. Ogni processore fisico nella GPU può eseguire un gruppo di thread chiamati globalmente *wavefront* o *warp*. Il numero di wavefront in esecuzione contemporanea dipende dal numero di registri attivi usati da un kernel. Tuttavia, l'ottimizzazione dell'uso dei registri conduce solamente a miglioramenti nelle performance riducendo la latenza degli accessi in memoria. Se il wavefront non è supportato da un numero sufficiente di registri attivi, l'hardware riverserà comunque i dati dei thread nella memoria producendo un significativo impatto sulle performance. Un altro problema inerente le performance sono i calcoli in doppia precisione (FP64). Infatti le performance dipendono dalla complessità dell'operazione (come si vedrà in seguito a causa del minor numero di unità hardware capaci di elaborare in doppia precisione); il miglior scenario prevede approssimativamente la metà delle performance in singola precisione (FP32), benché nel caso pessimo queste possano ridursi fino ad un quarto.

Si tratta quindi di adattare a questo modello gli algoritmi comunemente utilizzati. Quest'approccio è tutt'altro che semplice. Prima di tutto un algoritmo deve essere parallelizzabile. In secondo luogo si devono utilizzare delle specifiche API[8] legate ad una particolare architettura rendendo più complesso lo sviluppo cross-platform. Nella migliore delle ipotesi il codice di un programma sequenziale va semplicemente riscritto.

[6] Single Instruction Multiple Data

[7] Il processo di mappare i thread dal dominio nei SIMD engines è chiamato *rasterizzazione*.

[8] Application Programming Interface

1.3 Modelli architetturali

Allo scopo di rendere più chiare le differenze tra CPU e GPU riportiamo i due modelli architetturali che vi sottostanno: la macchina di Von Neumann e la macchina SIMD.

1.3.1 La macchina di Von Neumann

Come descritto in [53] la macchina di Von Neumann è il modello alla base di tutti i computer sequenziali a programma memorizzato. Essa è formata da cinque componenti:

- **Input device**: riceve i dati dal mondo esterno e li invia alla memoria;
- **Memory**: tiene traccia delle istruzioni, dei dati, dei risultati intermedi e di quelli finali;
- **Program-control unit:** interpreta ed esegue le istruzioni;
- **Arithmetic and logic unit (ALU):** esegue le operazioni aritmetiche e logiche;
- **Output device**: prende i risultati e li invia al mondo esterno;

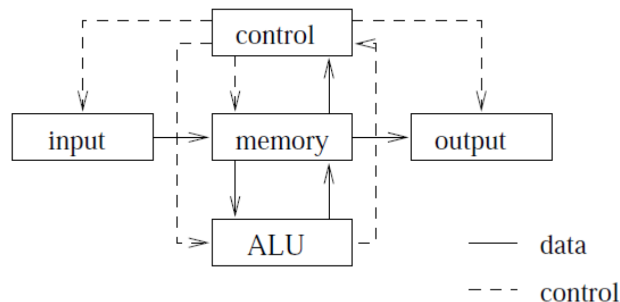


Figura 1.1, gli elementi costitutivi della macchina di Von Neumann e loro relazioni

Tale macchina esegue ininterrottamente le seguenti operazioni:

- L'unità di controllo richiede e preleva la prossima istruzione dalla memoria;
- L'unità di controllo decodifica l'istruzione;
- In base al risultato della decodifica, viene intrapresa una delle seguenti azioni:
  a. Carica gli operandi dalla memoria, salvandoli all'interno della ALU nei registri, quindi esegue un'operazione su di essi;
  b. Salva il contenuto di un registro della ALU nella memoria;
  c. Accetta un dato dal dispositivo di input salvandolo nella memoria;
  d. Prende il contenuto di una locazione di memoria e lo invia al dispositivo di output.

La descrizione di un problema è data in maniera procedurale, mentre il task computazionale è dato come una sequenza d'istruzioni. Effettuare la computazione significa semplicemente che la data sequenza di istruzioni deve essere eseguita. L'esecuzione delle istruzioni segue una precisa semantica a transizione di stato, per cui la macchina di Von Neumann si comporta come un automa a stati finiti. Siccome il task computazionale è specificato come una sequenza ordinata di istruzioni e la sequenza d'esecuzione è guidata dal programma di controllo questo modello è essenzialmente sequenziale.

Questa natura sequenziale della descrizione dei problemi forza una serializzazione anche in quei casi in cui il problema è inerentemente parallelo.

1.3.2 La macchina SIMD

In questo caso troviamo *n* ALU (indicate in figura 1.2 con il termine *processor core*) che gestiscono un flusso unico di istruzioni mentre reperiscono dalla memoria *n* flussi di dati differenti. In questo caso gli *n* processori sono necessariamente identici tra loro. Tutti i processori operano sotto il controllo di una singola e identica istruzione utilizzando operandi differenti; i processori operano ancora in modo sincrono.

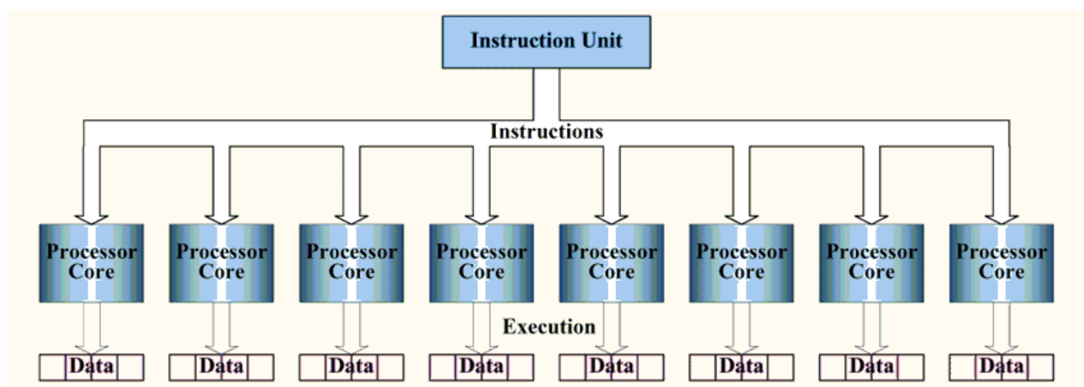


Figura 1.2, la macchina SIMD

La singola e unica istruzione può essere di tipo semplice (una somma tra due operandi) o complessa (es. merging di una lista di valori); in modo analogo il dato, diverso per ogni processore, può essere singolo o numeroso. A volte l'istruzione può contenere l'informazione che abilita o disabilita uno o più processori dall'eseguire l'istruzione (impedendo di fatto la scrittura dei risultati); in questo caso tali processori non effettuano calcolo utile fintanto che istruzioni successive non ne riabilitino l'esecuzione. Questa architettura si presta per definizione alla risoluzione di problemi che hanno a che fare con una notevole mole di dati; il parallelismo del calcolo garantisce un notevole throughput. I problemi tipici da sottoporre a una macchina SIMD si limitano però a tutti quelli che possono essere scomposti in sottoproblemi identici, ciascuno dei quali sarà risolto simultaneamente dallo stesso insieme di istruzioni. La memoria può essere realizzata come memoria unica condivisa o mediante una rete di interconnessione che collega le memorie private di ogni unità di elaborazione. Nel modello SPMD[9], più unità di elaborazione autonome eseguono simultaneamente lo stesso programma potenzialmente in punti indipendenti, piuttosto che elaborando ognuno la medesima istruzione, su dati distinti.

[9] Single Program Multiple Data

1.4 Modelli di parallelismo data-parallel

La forma di parallelismo data parallel si basa sul *partizionamento* dei dati con *replicazione* delle funzioni da eseguire. Ciascuna richiesta è riferita a tutta la struttura dati dello stato[10]. Questo modello prevede il partizionamento dello stato nei confronti di un certo numero di worker. Ogni worker esegue la stessa funzione F sui dati appartenenti alla partizione ad esso assegnata.

```
int F(int elem)
{
    // calcolo basato su elem
}

void main()
{
    int[] a = new int[m];
    int[] b = new int[m];

    for (int i = 0; i < m; i++)
        b[i] = F(a[i]);
}
```

In funzione della computazione un worker può anche comunicare con gli altri per ottenere il valore dei dati nelle altre partizioni. In generale oltre al partizionamento si può anche avere replicazione dei dati, se usati in sola lettura, per ridurre il numero di comunicazioni da effettuare. In base allo schema di comunicazione tra i worker, una computazione data-parallel può essere di tipo:

- **map** *(computazioni locali)*, ogni worker lavora esclusivamente sulla propria partizione, per cui non ci sono comunicazioni tra worker; un esempio di map è il seguente

  ```
  // stato
  int[,] A = new int [m,n];

  // scattering: assegna una riga della matrice A[i] ad ogni worker.
  forall i,j = 0 to m
      A[i, j] = F(A[i, j]);

  // gathering: recupera i dati dai worker.
  // ...
  ```

  Dove si suppone di avere m worker, ognuno dei quali esegue il metodo F su un elemento di A; il "forall" è uno pseudocodice con cui si intende esprimere l'esecuzione in parallelo di tutti i worker; ogni worker riceve la parte di stato attraverso la Scatter e ritorna le modifiche apportate mediante la Gather;
- **stencil** *(computazioni non locali)*, ogni worker necessita dei dati presenti in partizioni assegnate ad altri worker; si deve, quindi, definire la configurazione delle comunicazioni (cioè lo stencil)

[10] Esiste anche il pattern in cui ogni richiesta è riferita ad una singola partizione; utilizzata per esempio nelle memorie modulari, sequenziali o interallacciate, in cui ogni richiesta di lettura/scrittura si applica a singoli moduli.

necessarie per il calcolo; uno stencil può essere fisso o variabile a seconda che tale schema rimanga inalterato o si possa modificare durante l'esecuzione del programma parallelo.

Questa classe di pattern ben si adatta alle macchine SIMD, rendendola di fatto l'unica adottata con successo su di esse. Una trattazione completa su questo modello va oltre gli scopi di questa tesi, si rimanda per questo a [54]. Il modello di esecuzione del data parallelism è il SPMD.

1.5 Parametri per la valutazione delle prestazioni di applicazioni parallele

La teoria del calcolo parallelo studia formalmente le proprietà di architetture e programmi paralleli ricavandone *modelli di costo* (valutazioni delle prestazioni) in funzione:

- dei parametri caratteristici del problema (grado di parallelismo, grana del parallelismo e delle comunicazioni);
- ampiezza delle strutture dati sulle quali si opera in parallelo;
- la struttura e proprietà della rete di comunicazione.

Nel seguito della tesi per la valutazione delle performance del codice generato utilizzeremo i seguenti parametri:

- *tempo di completamento* T_c, tempo medio necessario a completare l'esecuzione di una computazione costituita da più operazioni;
- *efficienza relativa* ϵ_c, stabilisce il grado di utilizzazione dei moduli che compongono il sistema parallelo in formula:

$$\epsilon_c = \frac{T_{c-id}^n}{T_c^n}$$

  Dove "n" è il grado di parallellismo del sistema, T_{c-id}^n è il tempo di completamento ideale;
- *scalabilità*, misura quanto è stata "velocizzata" la computazione rispetto a quella sequenziale, in formula:

$$s_c = \frac{T_c^1}{T_c^n} = \epsilon_c * n$$

  Dove T_c^1 è il tempo di completamento con grado di parallelismo pari a 1.

In generale il tempo di completamento di un modulo è influenzato dalle comunicazioni. A livello dei processi l'overhead di comunicazione è dovuto a:

- Ritardi fisici per la trasmissione dei dati tra unità di elaborazione (messaggi tra nodi e/o accessi a memorie condivise) che dipendono dai ritardi delle strutture di interconnessione;
- Tempi di esecuzione delle funzionalità del supporto a tempo di esecuzione per la concorrenza ivi comprese quelle di scheduling a basso livello.

Per ridurre, e possibilmente eliminare questa degradazione delle prestazioni occorre cercare di mascherare la latenza delle comunicazioni con calcolo:

- Usando comunicazioni asincrone;
- Sfruttando il non determinismo per non rendere sequenziali comunicazioni e calcolo.

Capitolo 2

In questo capitolo è riportato uno studio accurato dei modelli hardware dei due maggiori produttori di schede video presi in considerazione durante lo sviluppo della tesi (ATI e Nvidia), introducendo allo stesso tempo le caratteristiche delle nuove architetture *Cypress* di ATI e *Fermi* di Nvidia. Al termine si fornirà un confronto qualitativo delle due soluzioni proposte.

2.1 Architetture GPU: modelli hardware e di memoria

Consideriamo ora le due architetture proposte da AMD e Nvidia per apprezzare come lo stesso modello concettuale (vedi paragrafo 1.2 GPU computing) sia stato implementato in modi diversi.

2.1.1 AMD

La GPU di AMD implementa una micro-architettura parallela che permette lo sviluppo di applicazioni di calcolo parallelo di tipo general-purpose. Come illustrato in figura 2.1Figura 2.1, la GPU di AMD include:

- Lo *stream processor*: è organizzato come un insieme di pipeline SIMD (dette anche *Data Parallel Processor Array* in figura 2.1), ognuna indipendente dalle altre, che opera in parallelo sugli stream di dati. Le pipeline SIMD possono processare o trasferire dati con la memoria.
- Il *memory controller*: ha accesso diretto alla memoria locale e alle aree della memoria di sistema specificate dall'host. Inoltre, per soddisfare le richieste di letture e scritture, il memory controller svolge le funzioni di un direct-memory access (DMA) controller.
- Il *command processor*: legge e avvia i commandi inviati dalla CPU host alla GPU per l'esecuzione. Il command processor notifica l'host del completamento dei commandi.
- Le *cache* sono usate per ottimizzare l'accesso ai dati e al codice nella gerarchia di memoria.

Il programmatore è maggiormente interessato alle prime due componenti, che è possibile controllare da programma: lo stream processor e il memory controller.

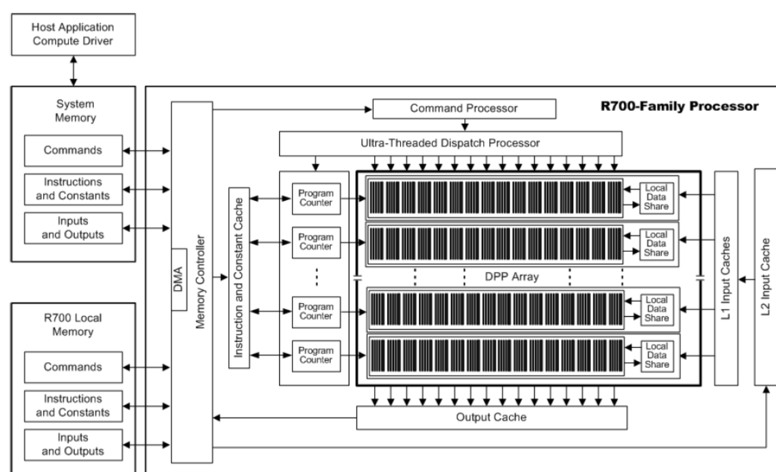


Figura 2.1, schema a blocchi della GPU di AMD

Stream Processor

Il processore RV770 [23] possiede 10 pipeline SIMD, ognuna delle quali contiene a sua volta 16 SPU[11]. Tutte le SPU appartenenti a una stessa pipeline eseguono le stesse istruzioni (poiché condividono lo stesso program counter), e ogni SPU processa 4 *thread*[12] alla volta (definiti in [48] pag. 16 come *"un'istanza di kernel in esecuzione su un SIMD engine thread processor"*), rendendolo un processore SIMD ampio 160. Ogni SPU è internamente organizzata come un processore scalare a 5 vie, permettendo fino a cinque operazioni scalari in un'istruzione VLIW[13] [24]. Quindi, nella situazione ideale ci sono un totale di 800 processori scalari sulla GPU RV770 che eseguono istruzioni concorrentemente. Uno dei cinque processori scalari della SPU può anche eseguire operazioni trascendenti (seno, coseno, logaritmo, etc.) e per ogni SPU vi è un'unità specializzata adibita alla gestione dei branch. Il consumo energetico massimo della scheda video, che risulta essere minore di 150 W (ad esempio per la Firestream 9250, o di 160 W per la Radeon HD 4870), permette di installare schede multiple su una singola scheda madre.
Lo schema a blocchi dell'RV770 è mostrato in Figura 2figura 2.2.

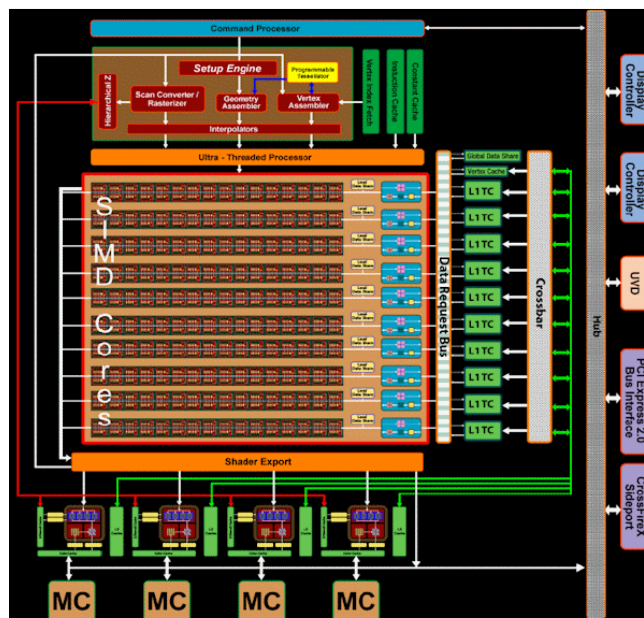


Figura 2.2, schema a blocchi dell'architettura RV770

Con il processore RV870 ATI ha introdotto l'architettura Terascale 2: il numero di transistor passa da 956 milioni a 2,15 miliardi, merito dell'adozione di un processo produttivo a 40 nm (contro i 55 nm di quello precedente). Basandoci sullo schema riportato in figura 2.3 si può notare che il numero di array SIMD è stato portato a 20, ognuno dei quali include 16 SPU, ognuna avente come nel caso dell'RV770 5 stream

---

[11] Shader Processing Units, le unità fisiche hardware che eseguono i thread.

[12] Da non confondere con i thread definiti a livello di S.O.

[13] Very Long Instruction Word

core per un totale di 1600 processori scalari. Il clock dei core è 850 MHz, mentre quello dei core della Radeon HD 4870 (dotata del processore RV770) è di 750 MHz.

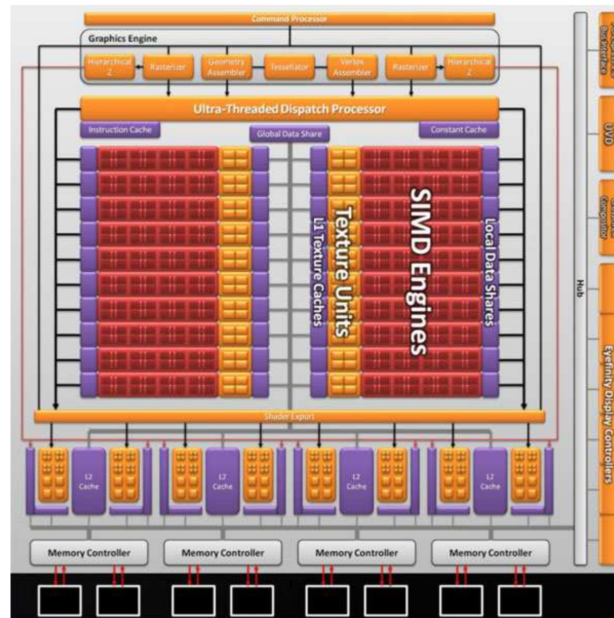


Figura 2.3, schema a blocchi dell'RV870

AMD non ha solamente raddoppiato il numero di queste unità ma ne ha migliorato l'IPC[14] dando la possibilità di eseguire nello stesso ciclo di clock una moltiplicazione e un'addizione dipendente dall'esito della precedente operazione. Si consideri il seguente esempio:

```
a = b * c
d = a + x
```

Il processore RV770 non può eseguire in un ciclo di clock queste due operazioni, mentre per l'RV870 è lecito farlo.

Sono state introdotte le operazioni DirectX 11 bit-level (bit count, insert, extract, etc.) e la FMAD[15], ossia una moltiplicazione seguita da un'addizione in cui l'arrotondamento è effettuato solo al termine del calcolo, diversamente da una standard MAD[16], che effettua due arrotondamenti. Per migliorare le prestazioni di applicazioni di video encoding e di computer vision è stata introdotta l'istruzione SAD[17], ed esposta tramite estensioni OpenCL. L'unità specializzata può eseguire operazioni MAD a 32-bit FP in un ciclo di clock.

---

[14] Instructions per cycle

[15] Fused Multiply-Add

[16] Multiply-Add

[17] Sum of Absolute Differences

La gestione dei prodotti scalari è più flessibile data la presenza delle istruzioni DP2 e DP3 per effettuare il prodotto scalare a 2 e 3 componenti, mentre l'RV770 aveva solamente l'istruzione DP4 con DP2 e DP3 implementate usando la DP4, sprecando risorse hardware inutilmente.
E' cambiata anche la gestione delle operazioni su interi: in precedenza, ognuno dei quattro stream core poteva eseguire un'addizione o operazioni di bit-shift su interi a 32-bit per ciclo di clock, e l'unità specializzata poteva eseguire una moltiplicazione o bit-shift (anch'esse su interi a 32-bit). Ora i quattro core sono in grado di compiere una moltiplicazione o addizione per ciclo, ma solo su interi a 24-bit. Questa scelta è il risultato di un compromesso tra la ricerca delle performance e la necessità di non sacrificare troppe risorse per farlo: limitandosi a operazioni su 24-bit si possono riutilizzare risorse per gestire piuttosto numeri in virgola mobile a singola precisione, massimizzando l'utilizzazione delle risorse del core.

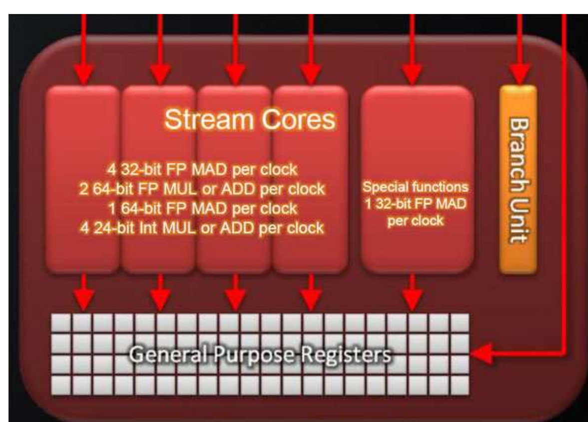


Figura 2.4, lo stream core dell'RV870

L'unità dedicata alla gestione dei branch è rimasta invariata. Tutto questo contribuisce al raggiungimento di 2.7 TFLOPS per operazioni FP32 (contro gli 1.2 TFLOPS per l'RV770 della Radeon HD 4870) e 544 GFLOPS per operazioni a doppia precisione.
Il consumo di picco è di 188 watt per la HD 5870, in idle la nuova GPU consuma invece 27 watt, un valore significativamente inferiore ai 90 watt della HD 4870.

Nell'RV770 ogni SIMD ha una logica di controllo dedicata, un'unità Texture (contenente 16 FP32 texture sampler, 4 texture address processor e 4 unità texture filter) e una cache L1 Texture (indicata con TC, in verde chiaro nella figura 2.5 a sinistra). Nel caso dell'RV870 il totale di unità adibite al texturing sale ad 80 (contro i 40 della precedente versione). L'unico modo per i core della GPU di comunicare è attraverso il *Global Data Share*, una cache condivisa tra i SIMD; questa è una differenza significativa rispetto alla soluzione proposta da Nvidia come si discuterà in seguito. Ogni unità Texture è responsabile per l'esecuzione dei fetch sulle texture, e per richiedere dati dal memory controller caricando i registri con i dati restituiti dalla cache.

Per il RV770, la costruzione del *wavefront* e l'ordine d'esecuzione dei thread dipende dal modo in cui avviene la rasterizzazione del dominio d'esecuzione. La rasterizzazione segue un pattern predefinito a zig-zag sul dominio. Attualmente, questo pattern non è disponibile pubblicamente; è noto solamente

che è basato su un multiplo di 8x8 blocchi all'interno del dominio, che uguaglia la dimensione del wavefront. Il processo di rasterizzazione è trasparente all'utente, ma potrebbe influenzare le prestazioni con le quali si accede alla memoria.

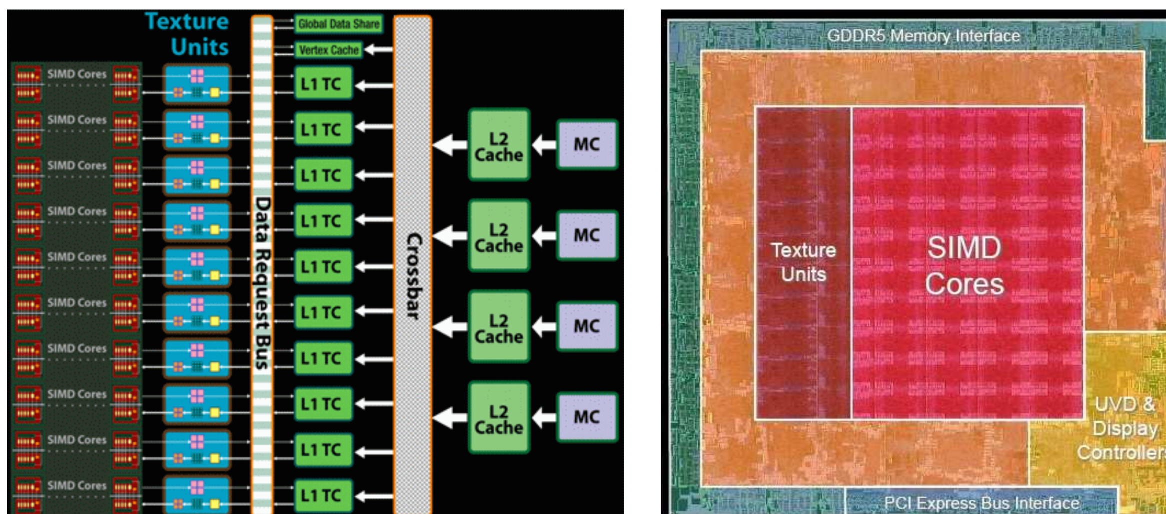


Figura 2.5. schema a blocchi dell'architettura RV770, particolare (sinistra), disposizione sul die dei principali blocchi dell'RV770 (destra).

Il processore RV770 è in grado di gestire numeri in virgola mobile a doppia precisione collegando 4 ALU tra loro all'interno di una singola SPU (esclusa quella che esegue le funzioni trascendenti) in modo da formare due coppie, ognuna delle quali in grado di eseguire una singola operazione in doppia precisione. Le prestazioni per i calcoli FP64 sono un quinto di quelle FP32: 240 GFLOPS per la Radeon HD 4870 o la FireStream 9250.

Memoria

RV770 introduce l'uso di un nuovo *distributed memory controller* (il modello precedente usava un ring bus controller) mostrato in figura 2.6. Questa interfaccia prevede che i controller (aventi ampiezza 256-bit) per la memoria siano distribuiti ai bordi del core, dove possono fornire accessi alla memoria alle unità che richiedono più banda di memoria, ossia i render back-end e le texture cache L2. Questi controller sono tutti accoppiati a un hub che gestisce tutte quelle funzionalità che richiedono minor banda di memoria come il processing video e il collegamento CrossFire inter-GPU (nel caso di più GPU installate sulla stessa macchina). Questo cambiamento riduce drasticamente la quantità di silicio richiesta per il controller, riducendo la latenza, limitando il consumo energetico e migliorando l'efficienza della banda di memoria.

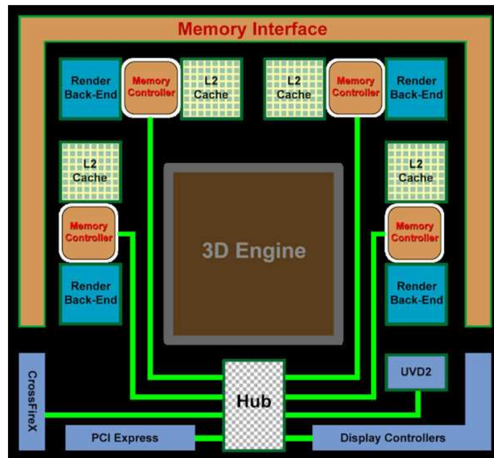


Figura 2.6, il distributed memory controller

Memoria off-chip. AMD fornisce le proprie schede Firestream con una memoria on-board da 256-bit GDDR3 (chiamata MC nella figura 2.5Figura 2. a sinistra). La Firestream 9250, con il processore RV770, ne possiede 1 GB, ma con velocità di clock di 993 MHz. La Radeon HD 4870 usa invece una memoria di tipo GDDR5 e ne possiede 2 GB. Nel caso dell'RV870 viene sempra adottato una memoria di tipo GDDR5 ma è stata aggiunta la funzionalità di EDC[18] che effettua controlli CRC sui trasferimenti di dati per avere maggiore affidabilità. Il clock della memoria è aumentato fino a raggiungere i 1200 MHz portando la banda complessiva di memoria fino a 153.6 GB/s.

Memoria on-chip. Il RV770 ha due livelli di cache:

- 4 cache L2 da 64K (per un totale di 256 KB) ognuna associata a un memory controller da 64-bit.
- 8 KB L1 cache per ogni SIMD (L1 TC, in verde in figura 2.5) per un totale di 80 KB, una cache per i dati constanti e una cache istruzioni; possiede una banda di 480 GB/s per L1 texture fetching, e di 384 GB/s per i trasferimenti tra cache di primo e secondo livello.

Sono inoltre presenti Vertex cache, Color cache, Z/Stencil cache, e un write buffer (indicato con Output Cache in figura 2.1) per gestire le scritture in uscita dagli stream core.

E' presente una piccola area di memoria ampia 16 KB chiamata *Local Data Share* (introdotta col processore RV670), che è interamente sotto il controllo del programmatore. Essa permette ai kernel di condividere dati tra diversi thread. Inoltre, l'RV770 aggiunge un'ulteriore area di memoria (sempre ampia 16 KB) chiamata *Global Data Share* (il riquadro verde posto più in alto in Figura 2.), per permettere la comunicazione tra array di SIMD. Nel caso dell'RV870 (figura 2.7) la memoria LDS passa a 32 KB, e quella GDS aumenta sino a 64 KB. Per ogni SIMD engine troviamo una cache L1 di 8K, per un totale di 160 KB di cache di primo livello. La dimensione della cache L2 è stata raddoppiata (ora è di 128 KB) per un totale di 512 KB ma le velocità sono state portate a 1TB/s per operazioni di fetch dalla memoria texture L1 e fino a 435 GB/s di banda tra la cache L1 e quella L2.

[18] Error Detection Code

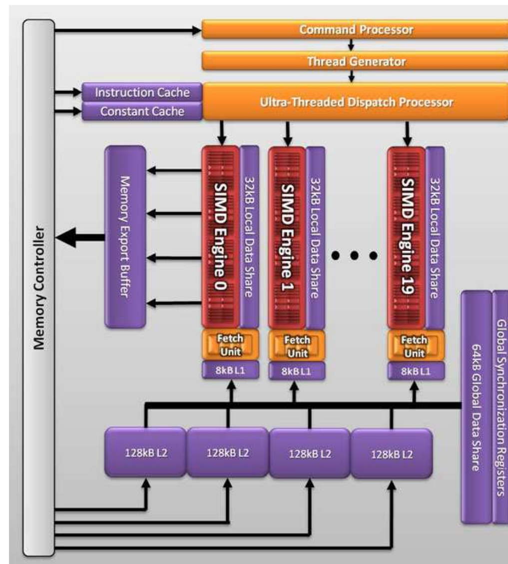


Figura 2.7, LDS, GDS e cache L1/L2 dell'RV870

Le SPU non accedono direttamente alla memoria locale della GPU; invece esse inviano richieste di lettura o scrittura in memoria tramite unità hardware dedicate (unità Render Back-End). Ci sono due modi di accedere la memoria – *cached* e *uncached*. Al di là del caching, la principale differenza tra le due è che la *memoria uncached* permette scritture in locazioni arbitrarie (scatter), mentre la *memoria cached* solo verso la locazione del dominio associata al thread.

Memoria cached

Gli stream allocati nella memoria cache hanno una limitazione hardware di 8192 elementi per stream mono dimensionali e di 8192 x 8192 elementi per stream bidimensionali. Il tipo degli elementi di questi stream può essere uno qualsiasi dei tipi di dato supportati (float, int), inclusi i tipi vettoriali (float2, float4, int4).

Quando il risultato di un kernel ha bisogno di essere scritto sullo stream d'uscita, il dato è inviato alle unità *render back-end* che scrivono in una cache d'uscita e trasferiscono i dati dalla cache verso la memoria. Nel caso dell'RV770 queste sono 16, mentre raddoppiano nel caso dell'RV870.

Memoria un-cached

Il global buffer è uno spazio di memoria, allocato linearmente, che può essere scritto e letto in modo un-cached. Su questo buffer è possibile eseguire scritture (e letture) su locazioni d'indirizzo arbitrario che sono dette s*catters* (g*athers* le letture). Le scritture di tipo scatter non sono sincronizzate, ed è responsabilità dello sviluppatore garantire che le letture avvengano dopo le scritture. Siccome questi accessi in scrittura avvengono in modo un-cached, le performance sono minori rispetto alle scritture standard.

Comunicazioni CPU – GPU

Tutte le comunicazioni e i trasferimenti di dati tra il sistema (host) e la GPU avvengono sul canale PCI-Express (PCI-E). I trasferimenti dal sistema alla GPU occorrono grazie al DMA engine. Quest'unità DMA

può eseguire in modo asincrono rispetto al resto della GPU permettendo trasferimenti di dati in parallelo anche quando la GPU è occupata nell'esecuzione di un precedente kernel. Le applicazioni possono richiedere un trasferimento in DMA da CAL (l'API a basso livello di ATI, vedi paragrafo 3.1.2) usando speciali procedure che permettono la copia di buffer dati tra risorse remote e locali al dispositivo; per maggiori dettagli si guardi [25]. Con l'architettura Cypress, che supporta le DirectX 11, i processori grafici possono accettare flussi di dati da un numero qualsiasi di core della CPU, diversamente dai modelli precedenti che potevano solo accettare dati da un core di CPU alla volta.

2.1.2 Nvidia

Nvidia realizza le sue GPU come un insieme di Thread Processor Clusters (TPC), ognuno equipaggiato con un'unità Texture (TEX) e un array scalabile di multithreaded Streaming Multiprocessors (SM), con memoria condivisa on-chip. Per gestire centinaia di thread che eseguono parecchi programmi differenti, il multiprocessor impiega una nuova architettura chiamata SIMT:[19] la principale differenza rispetto al modello SIMD è che la dimensione dei vettori in fase di elaborazione non ha un'ampiezza predefinita. L'unità multiprocessor SIMT crea, gestisce, schedula, ed esegue thread in gruppi di thread simultanei, i *warp* (corrispondenti ai wavefront di ATI). I singoli thread che compongono un warp SIMT partono insieme allo stesso indirizzo di programma, ma sono comunque liberi di effettuare branching ed eseguire indipendentemente l'uno dell'altro avendo i propri indirizzi d'istruzione e stato dei registri. In caso di branch si assiste però alla stessa degradazione delle performance che caratterizza le GPU ATI.

Stream Processor

Il processore che equipaggia la GeForce GTX 280 [26,27,28] è un array MIMD di 10 processori SIMT, chiamati TPC in figura 2.8 (sinistra), dove ogni TPC è un insieme di 3 shader processors (SM) ed un'unità TEX. Come illustrato in figura 2.8 (destra), ogni SM comprende 8 ALU scalari, chiamate Stream Processor (SP), ognuna in grado di effettuare calcoli FP32 e su interi a 32-bit, una singola ALU a 64-bit per supportare FP64, e un pool di Shared Memory ampio 16 KB. Perciò ci sono un totale di 240 processori scalari con un consumo massimo di potenza della scheda grafica di 236 W.

[19] Single-Instruction Multiple-Thread

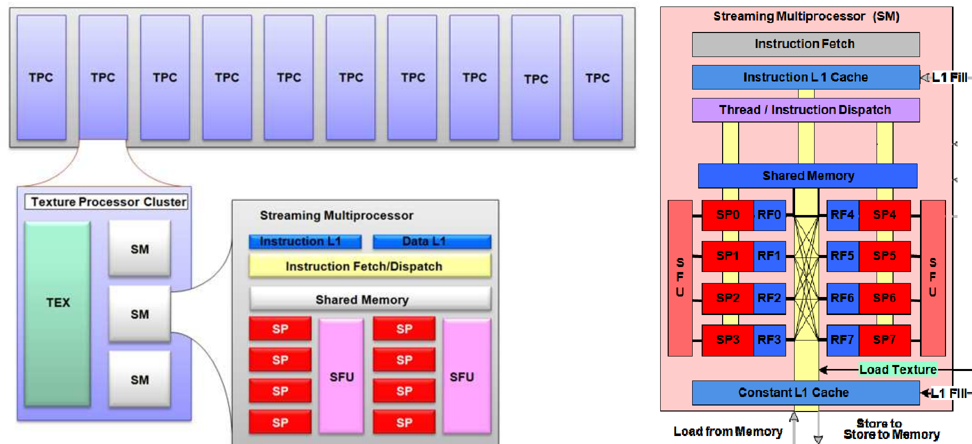


Figura 2.8. GeForce GTX 280 Graphics Processing Architecture (sinistra) e una visione dettaglia di un SM (destra)

Ogni SP, in ogni SM, esegue la medesima istruzione delle altre ad ogni ciclo di clock, ma ogni SM in un thread processor cluster può eseguire la propria istruzione. Pertanto in ogni istante, gli SM in un cluster stanno potenzialmente eseguendo una diversa istruzione di un programma shader in modo SIMD. Questo vale anche per la ALU FP64 per ogni SM, la quale può eseguire allo stesso tempo delle unità FP32, ma condivide i pool di memoria shared, e l'hardware per la schedulazione con esse quindi le due non possono andare pienamente allo stesso tempo.

La prima GPU basata sull´architettura Fermi dispone di ben 512 CUDA core, organizzati in 16 SM, ed è realizzata con oltre 3 miliardi di transistor utilizzando un processo produttivo a 40nm. Nel caso dell'architettura Fermi, il nuovo SM progettato da Nvidia include un certo numero di novità che vanno dal nuovo CUDA Core, alle unità di Load/Store e riguardano anche quelle speciali. Nella figura riportata qui sotto possiamo apprezzare quali siano i blocchi principali che ne costituiscono l´ossatura:

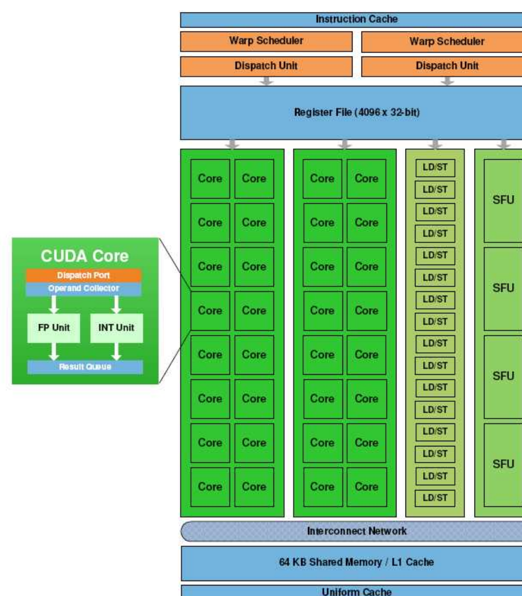


Figura 2.9, il nuovo stream multiprocessor della Fermi

Il cuore dello SM è costituito dal *CUDA Core*: esso dispone di un´unità ALU per i calcoli su numeri interi ed un´unità FPU per i calcoli su numeri in virgola mobile. L´aderenza alle norme IEEE 754-2008 che definiscono uno standard per l´aritmetica in virgola mobile permette all´architettura Fermi di gestire istruzioni FMA (già introdotte nella descrizione dell'architettura Cypress di ATI) sia a singola che a doppia precisione. La ALU della nuova architettura Fermi supporta nativamente una precisione a 32-bit per tutte le istruzioni rappresentando un´evoluzione di GT200 anche in questo senso (GT200 supporta una precisione a 24-bit). In aggiunta la ALU è ottimizzata anche per eseguire operazioni a doppia precisione (64-bit).

Oltre ai 32 CUDA Core (quattro volte il numero di quelli del GT200), lo SM dispone di 16 *unità di Load/Store* (indicate nello schema come LD/ST): ciò significa che gli indirizzi sorgente e destinazione del dato possono essere calcolati per un massimo di 16 thread per SM per ogni ciclo di clock. Tali unità sono in grado di caricare o memorizzare il dato sia sulla cache sia sulla memoria RAM.

Le unità *SFU*[20] presenti in numero di 4 per ogni SM, si occupano di eseguire in maniera efficiente le operazioni trascendenti. Una SFU è in grado di eseguire una di queste operazioni per thread per ciclo di clock. La cosa interessante è che le unità speciali sono disaccoppiate dalle unità di dispatch permettendo a queste di attivare altre unità di esecuzione mentre sono impegnate.

Con l´obiettivo principale di creare un prodotto capace di offrire una potenza eccezionale per gli ambienti HPC, Nvidia non poteva tralasciare il supporto all´*aritmetica a doppia precisione* alla quale fanno capo applicazioni di algebra lineare, simulazione numerica e chimica quantistica. In passato questo aspetto era stato un po´ trascurato ma con Fermi Nvidia ci punta decisamente offrendo la possibilità di gestire fino a 16 FMA con operandi a doppia precisione per ciclo di clock e per SM. Le prestazioni di picco nelle operazioni floating point a doppia precisione sono 8 volte superiori a quelle del GT200.

In figura 2.10Figura 2. (sinistra) [27] è possibile osservare uno schema a blocchi del die del chip. Confrontandolo con il die dell'AMD RV770, figura 2.5 (destra), è possibile notare come differiscano le due soluzioni proposte: Nvidia ha posto i core agli angoli del die e la memoria al centro, laddove AMD ha fatto l'opposto mettendo i core al centro. Tuttavia con la Fermi Nvidia adotta la stessa organizzazione di AMD.

[20] Special Function Unit

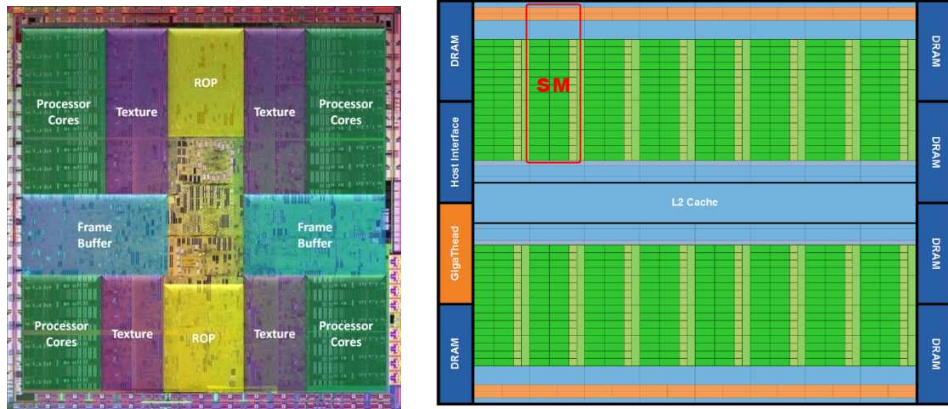


Figura 2.10, locazioni dei principali blocchi sul die della GeForce GTX 280 (sinistra), e l'organizzazione del die della Fermi (destra)

Nello parte alta dello schema a blocchi dell´architettura Fermi (figure 2.9/2.11) troviamo due unità indicate come *Warp Scheduler* seguite da altrettante *dispatch unit*: questo quartetto è in grado di gestire 32 thread paralleli (chiamati appunto warp) e può far sì che due warp siano attivati ed eseguiti contemporaneamente. In pratica i due Warp Scheduler dell´architettura Fermi selezionano due warp e rilasciano un´istruzione per ogni warp ad uno dei gruppi di 16 core, alle 16 unità di LD/ST o alle quattro SFU. Visto che i warp sono unità indipendenti, lo scheduler dell´architettura Fermi non deve controllare eventuali dipendenze nel flusso di istruzioni, con tanto di vantaggi in termini di performance.

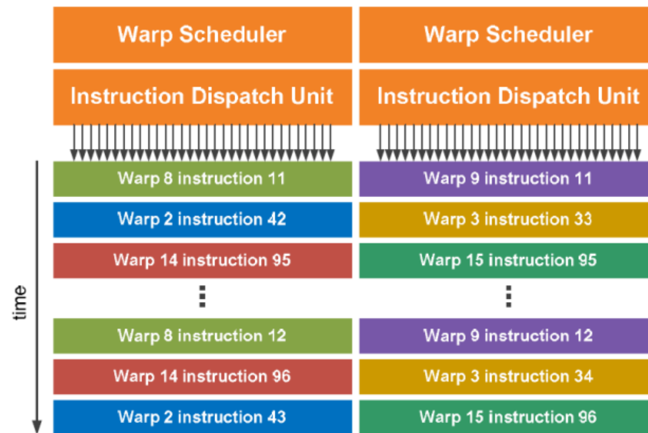


Figura 2.11, selezione e dispatching delle istruzioni ai warp da parte degli Warp Scheduler e unità Instruction Dispatch

A livello più esterno Nvidia ha integrato un altro blocco che prende il nome di *GigaThread Scheduler* il quale si occupa della pianificazione e distribuzione globale dei thread assegnandoli - a blocchi - agli SM (dove gli warp scheduler gestiscono localmente la schedulazione, come spiegato in precedenza). Questo scheduler gestisce efficientemente anche il *context switching* e, secondo i dati forniti dal produttore lo fa in appena 25 microsecondi, ben 10 volte più rapidamente delle GPU di precedente generazione. Infine, l´architettura Fermi permette l´*esecuzione concorrente* (come illustrato in figura 2.12) di più

kernel della stessa applicazione, feature molto utile per sfruttare al massimo tutte le risorse messe a disposizione dalla GPU.

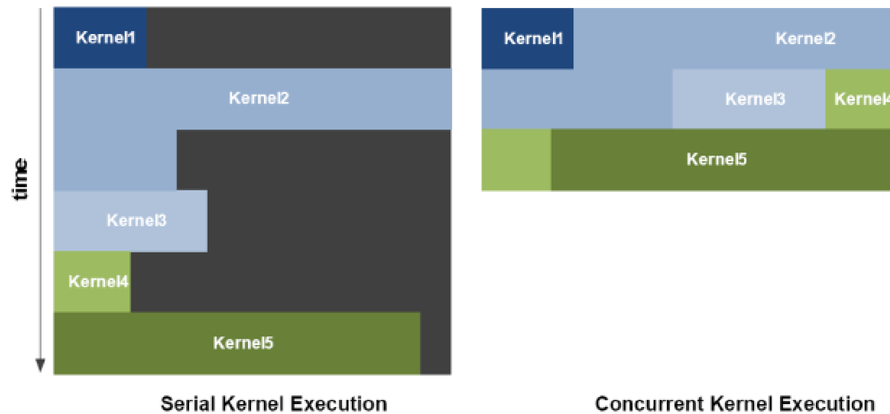


Figura 2.12, esecuzione seriale e concorrente dei kernel su GT200 e Fermi

Memoria

Memoria Off-chip. La GeForce GTX 280 ha 1 GB di memoria on-board a 512-bit GDDR3 con velocità di clock di 1107 MHz, mentre l'Nvidia Tesla[21] C1060 Computing Processor ha 4 GB 512-bit GDDR3 aventi velocità di clock di 800 MHz. Gli spazi di *memoria locale* e *globale* sono realizzati come regioni read-write un-cached della memoria della GPU. Quindi gli accessi in *memoria locale* sono costosi quanto quelli in memoria globale. Per questa ragione avvengono solamente per alcune variabili automatiche e sono sempre *coalesced* (si consulti a riguardo [29]), quindi, avvengono per-thread per definizione.

Per quel che riguarda la Fermi, attorno al core principale, al cui centro si trova la grossa linea delle memorie cache L2, sono disposti diversi blocchi: sei partizionamenti per le memorie video, ognuno da 64-bit per un bus complessivo ampio 384-bit (minore rispetto alla GTX 280) e in grado di pilotare 6 GB di GDDR5 SDRAM, un Host Interface che connette la GPU alla CPU attraverso un bus PCI Express. Dal punto di vista delle memorie, inoltre, Nvidia introduce per la prima volta se stessa nel mondo delle GDDR5 ma, cosa più importante, porta nel mondo delle schede grafiche una tecnologia di correzione dell´errore per le memorie. Fermi, infatti, supporta i codici *SECDED ECC*[22] che permettono di correggere errori su singolo bit e rilevare errori su più bit forzando la riesecuzione del codice. Il problema di bit di memoria errati a causa d'interferenze fisiche si fa sempre più importante al crescere del numero di GPU installate, come accade ad esempio con grossi sistemi cluster. Per questo il produttore californiano ha integrato le tecniche ECC in tutta l´architettura Fermi, dai registri alle memorie cache e condivisa oltre che nelle memorie GDDR5.

[21] L'equivalente del modello FireStream di ATI/AMD

[22] Single-Error Correct Double-Error Detect ECC

Memoria On-chip. Ogni SM ha la memoria on-chip dei seguenti 4 tipi:

- Un insieme di *registri* locali a 32-bit per processore.
- Una cache dati parallela ampia 16 KB, detta *shared memory*, che è condivisa da tutti i processori e implementa lo spazio di memoria condivisa.
- Una *cache constanti* a sola lettura che è condivisa da tutti i processori e velocizza le letture dallo spazio di memoria costanti, che è implementato come una regione a sola lettura della memoria del dispositivo.
- Una *cache texture* in sola lettura condivisa da tutti i processori e che è realizzata come una regione a sola lettura della memoria del dispositivo.

Siccome è on-chip, lo spazio di *memoria shared* è molto più veloce degli spazi di memoria locale e globale. Infatti, per tutti i thread di un warp, accedere alla memoria shared è veloce quanto accedere ad un registro fintanto che non ci sono conflitti tra le richieste dei thread. Un processore impiega 4 cicli di clock per comandare una istruzione di memoria per un warp, tuttavia quando si accede alla memoria locale o globale, ci sono, in aggiunta, dai 400 ai 600 cicli di clock di latenza della memoria.

Per ottenere un'alta banda di memoria, la memoria shared è divisa in moduli di memoria aventi medesima ampiezza, chiamati banchi, che possono essere acceduti simultaneamente (organizzazione interallaciata). Tuttavia, se gli indirizzi di due richieste di accesso in memoria ricadono nello stesso banco, c'è un conflitto e gli accessi devono essere serializzati. L'hardware, se necessario, suddivide una richiesta per la memoria che ha conflitti di banco in diverse richieste separate prive di conflitti, diminuendo la banda di memoria di un fattore pari al numero di richieste separate per la memoria.

La Fermi introduce due livelli di cache unitamente al livello di memoria più tradizionale, migliorando non solo le prestazioni del chip con applicazioni di tipo GP ma semplificando anche la programmazione dello stesso offrendo una cache L1 *configurabile*:

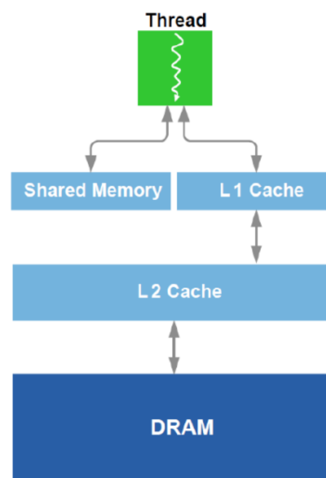


Figura 2.13, la cache L1 configurabile

ogni SM dell´architettura Fermi dispone infatti di un buffer di memoria on-chip da 64 KB che può essere partizionato in 16 KB di cache L1 e 48 KB di memoria condivisa oppure in 48 KB di cache L1 e 16 KB di

memoria condivisa. Ciò significa che le applicazioni che fanno un uso intensivo di memoria condivisa, ovvero nei casi in cui i thread devono scambiarsi molte informazioni, avere una maggiore dotazione di questa permette di ottenere migliori prestazioni. Tutte quelle applicazioni che invece attualmente usano la memoria condivisa come cache L1 ma gestendola in maniera software, possono ottenere enormi vantaggi evitando codice inutile e facendo leva su una più efficiente gestione hardware. La gerarchia creata da Nvidia con Fermi, nella quale è prevista una cache L1 dedicata e una cache L2 condivisa permette anche di risolvere alcuni problemi importanti legati al GPU Computing. Una GPU tradizionale prevede due path separati per la lettura e la scrittura dei dati in memoria, cosa assolutamente sconveniente nel momento in cui si devono eseguire programmi General Purpose per i quali si potrebbero creare dei conflitti di tipo *read-after-write*. Nell´architettura Fermi, invece, è stato implementato un path unico per le operazioni di load e store.

Comunicazioni CPU – GPU

Tutte le comunicazioni e i trasferimenti di dati tra il sistema e la GPU avvengono sul canale PCI-Express (PCI-E). I transferimenti dal sistema alla GPU avvengono attraverso il DMA engine. Quest'unità DMA può girare in modo asincrono dal resto della GPU, permettendo trasferimenti di dati in parallelo quando la GPU è occupata ad eseguire un precedente kernel. Le applicazioni CUDA (l'API di Nvidia, si veda a riguardo il prossimo capitolo) possono richiedere un trasferimento in DMA usando speciali routine che consentono la copia di buffer di dati tra risorse remote e locali al dispositivo (per maggiori dettagli si consulti [28]). Con l'architettura Fermi Nvidia ha rinnovato l'interfaccia verso l'host per permettere trasferimenti bidirezionali concorrenti tra la memoria di sistema e quella della GPU, che si sovrappongono completamente al tempo di calcolo di CPU e GPU. Questo significa che c'è la possibilità di avere un flusso ininterrotto tra la CPU e la GPU quando entrambe stanno elaborando.

2.2 AMD vs. Nvidia

Dopo l'introduzione tecnica, presentiamo un'analisi qualitativa dell'architettura di AMD e Nvidia. Entrambi i produttori implementano il medesimo paradigma SIMD, ma le loro visioni divergenti dei processori grafici hanno condotto a differenti architetture. Nvidia adotta ancora un processo produttivo a 65 nm per la propria serie GTX200 (contro i 55 nm della serie HD 4000 di ATI), costruendo die che sono il 55% più grandi di quelli AMD, ma con performance similari sia in termini di banda che di shading power. Nonostante entrambi i produttori siano passati recentemente ai 40 nm per i loro nuovi modelli, la tendenza sembra riconfermarsi visto che il die della HD 4870 è di 334 mm^2 contro i 500 mm^2 circa attesi per la Fermi. Con 1400 milioni di transistors, il consumo medio dell'Nvidia GTX 280 è quasi il doppio dell'AMD Firestream. I progettisti Nvidia hanno tentato di ridurre il consumo aggiungendo un chip che monitora costantemente il livello di utilizzo della GPU. Usando tale informazione il driver aggiusta automaticamente la frequenza, il voltaggio e l'attività di ogni parte della circuiteria. Tuttavia, test dimostrano che la GTX 280 non va mai veramente indietro al livello minimo, e come conseguenza ha significativi problemi di surriscaldamento. Questo diventa critico, per esempio in un cluster di GPUs, dove ci possono essere centinaia di schede GPU che lavorano in parallelo.

Per quel che riguarda la banda della memoria, allo scopo di fornire alla GPU quanti più dati possibili, Nvidia, fedele al proprio concetto di GPU, introduce un bus a 512-bit, raggiungendo più di 140 GB/s di banda (contro i 115.2 GB/s dell'HD 4870), e quindi con i più alti fill rate. Viceversa, con il RV770, AMD ha concentrato i propri sforzi su una GPU dalle ridotte dimensioni del die, con un più basso numero di transistor, ma con un maggior numero di ALU, 800 nell'RV770 contro 240 dell'Nvidia GTX 280. Inoltre, AMD equipaggia le sue ultime schede video con la nuova generazione GDDR (GDDR5 ancora con il bus a 256 bit), per esempio la Radeon HD 4870, che dispone di una banda di memoria solamente il 3% inferiore a quella dell'Nvidia GTX 280 con un bus dalla dimensione doppia.

Una delle caratteristiche più interessanti introdotte da Nvidia e solo recentemente da AMD, con l'RV770, è la memoria shared on-chip. E' estremamente veloce e sotto il controllo degli sviluppatori, che possono usarla per le comunicazioni fra thread dello stesso blocco. Questo permette di incrementare le prestazioni di alcuni ordini di grandezza, poichè riduce considerabilmente il numero di trasferimenti di dati CPU-GPU. In ogni caso la memoria shared è di piccole dimensioni, al più 16 KB per blocco di thread, quindi il programmatore deve usarla accuratamente dato che ogniqualvolta il wavefront non è supportato da un adeguato numero di memoria on-chip, l'hardware riverserà i dati dei thread nella memoria off-chip, avendo un impatto significativo sulle performance. Questo comportamento a runtime è abbastanza seccante, perché non è ciò che il programmatore si aspetterebbe dal proprio programma, e può condurre ad un debugging ingarbugliato, costringendo il programmatore a cercare ogni allocazione di memoria, valutando per ognuna se può eccedere il limite della memoria shared. Se un programmatore configura l'esecuzione di un'applicazione CUDA in modo tale che tutti i thread risiedano in un singolo blocco, allo scopo di superare il limite della memoria shared, dovrebbe distribuire i thread su blocchi distinti. In questo modo ogni blocco (e ogni thread in esso) può accedere ai propri 16 KB di memoria shared. Tuttavia, questa soluzione introduce un ulteriore problema. Attualmente, nè Nvidia nè AMD forniscono un'interfaccia di programmazione che permetta la condivisione di dati tra thread appartenenti a blocchi distinti tramite memoria on-chip. Con Nvidia, questo non è possibile, dato che nessuna memoria cache globale è disponibile sul loro chip (se non nella recente Fermi non ancora commercializzata), mentre AMD dota il proprio RV770 con una cache globale. Questa feature è molto importante perchè da ai programmatori l'opportunità di implementare efficientemente quegli algoritmi che non hanno un workload embarrassingly parallel, per esempio con sincronizzazione globale intrinseca, come l'algoritmo First Reaction Method di Gillespie.

Capitolo 3

In questo capitolo come prima cosa prima si darà uno sguardo allo stato dell'arte degli SDK per la programmazione di GPU dei due maggiori produttori nel campo, AMD e Nvidia. Questo per organizzare il design della libreria sviluppata successivamente in modo che fosse sufficientemente generale per coprire entrambe le API. Molte scelte di design sono state influenzate dal tentativo di unificare i due modelli di programmazione. Nonostante le due piattaforme possano sembrare simili a prima vista, a un'analisi più attenta si sono trovate differenze significative nei due approcci, modelli di memoria e capacità di computazione. Alcune di queste differenze sono legate all'hardware sottostante, ma molte sono dovute a differenti scelte di progetto delle astrazioni di programmazione. Abbiamo anche considerato le astrazioni di programmazione proposte e i sottostanti modelli computazionali, tracciando alcune considerazioni preliminari. E' presentato con sufficiente livello di dettaglio il CAL IL, il linguaggio intermedio che verrà utilizzato come linguaggio target della compilazione della parte GPU di PBricks.

3.1 Modelli di programmazione GPU

I modelli di programmazione rappresentano un fattore critico per sfruttare tutta la potenza computazionale potenzialmente disponibile sulla GPU, ed essi devono nascondere abbastanza dettagli riguardo alla struttura sottostante del sistema in modo che lo sviluppatore possa programmare la GPU senza avere una conoscenza estensiva dei livelli inferiori. Come abbiamo discusso nel capitolo 1 il modello naturale per la programmazione di GPU è lo stream processing, benchè abbiamo appena visto nel precedente capitolo che lo stesso modello astratto dovrebbe essere mappato su architetture differenti. In questa sezione rivolgiamo la nostra attenzione ai modelli di programmazione proposti dai due produttori.

Ogni modello di programmazione di successo deve prendere in considerazione le molte caratteristiche e problematiche delle GPU introdotte nelle precedenti sezioni (ad esempio huge register set, tiny stack, control flow, threading issues) assieme a tutte quelle feature che rappresentano casi speciali di feature legate ad aspetti prettamente grafici. Inoltre, detto modello dovrebbe considerare che l'ISA[23] e l'architettura cambiano ogni 6/12 mesi.

3.1.2 AMD

L'approccio di programmazione di AMD fornisce un'interfaccia cross-platform alle GPU AMD, compatibile con i cambiamenti futuri, che permette ai programmatori di effettuare ottimizzazioni specifiche ad un dispositivo. Lo stack software include le seguenti componenti, presentate seguendo una gerarchia di livelli top-down:

- Librerie, quali l'AMD Core Math Library (ACML) che include implementazioni di tutte le routine BLAS e LAPACK assieme alle procedure FFT, Math transcendental e Random Number Generator;
- API bindings, per l'interoperabilità con DirectX;

[23] Instruction Set Architecture

- Compilatore open-source per Brook+, Performance Profiling Tools come il GPU Shader Analyzer, e l'AMD CodeAnalyst.
- AMD Runtime, che contiene il Compute Abstraction Layer (CAL), una libreria Device Driver che permette alle applicazioni di interagire con i core che processano lo stream al livello più basso possibile, se necessario, per prestazioni ottimizzate, mantenendo allo stesso tempo la compatibilità in avanti.

Compute Abstraction Layer

CAL è una libreria device-driver situata al di sopra dell'AMD Close-To-Metal Hardware Abstraction Layer. CAL fornisce le seguenti funzionalità:

- Gestione del Dispositivo
- Gestione delle Risorse
- Generazione del Codice
- Caricamento dei Kernel ed Esecuzione
- Supporto Multi-device

Il modello computazionale è indipendente dal processore (figura 3.1) e permette all'utente di passare facilmente tra svolgere una computazione dalla GPU alla CPU e vice versa. L'API dovrebbe permettere di scrivere un load balancer dinamico al di sopra di CAL. CAL dovrebbe essere un'implementazione leggera, e dovrebbe facilitare lo sviluppo di librerie simili a Brook+ al di sopra di esso.

Un sistema CAL comprende un processo master in esecuzione sulla CPU che guida uno o più dispositivi. Un device è un componente hardware capace di far girarare programmi CAL (kernel). Un dispositivo ha uno o più processori come mostrato nel capitolo precedente. Il kernel è eseguito su questi processori ed è implementato usando il linguaggio intermedio AMD (IL). Un dispositivo è connesso a due sotto-sistemi di memoria - locale e remota. Il processo master può leggere e scrivere sia verso la memoria locale sia verso quella remota di ogni dispositivo, benchè tipicamente, il master ha velocità di scrittura e lettura maggiori verso la memoria remota del dispositivo. Il processo master invia comandi per l'esecuzione usando un device context. Il processo master è anche in grado di interrogare il contesto per conoscere lo stato di completamento di questi task. Gli input e gli output del programma possono essere settati sia nella memoria locale che in quella remota. Una computazione è invocata settando uno o più output e specificando una regione (dominio d'esecuzione) all'interno di questi output che deve essere computata. Nel caso di un dispositivo che abbia più processori (come un dispositivo GPU), uno schedulatore distribuisce il carico ai vari processori SIMD sul device.

L'astrazione CAL divide i comandi in due tipi: *device commands* e *context commands*. I *device commands* principalmente coinvolgono l'allocazione di risorse (in memoria locale e remota). Un contesto è una coda di comandi che sono inviati al dispositivo. E' possibile avere code parallele per differenti parti del dispositivo. Le risorse sono create sui dispositivi e sono mappate nei contesti; questo dev'essere fatto per fornire scoping e access control dall'interno della coda dei comandi. Ogni contesto rappresenta un'unica coda e ogni coda opera indipendentemente dalle altre.

I *context commands* (ad esempio *calCtxRunProgram*) accodano le loro azioni nel contesto fornito. Il dispositivo non esegue I comandi fintanto che la coda non è flushata; questo avviene implicitamente quando la coda è piena o esplicitamente tramite chiamate all'API CAL.

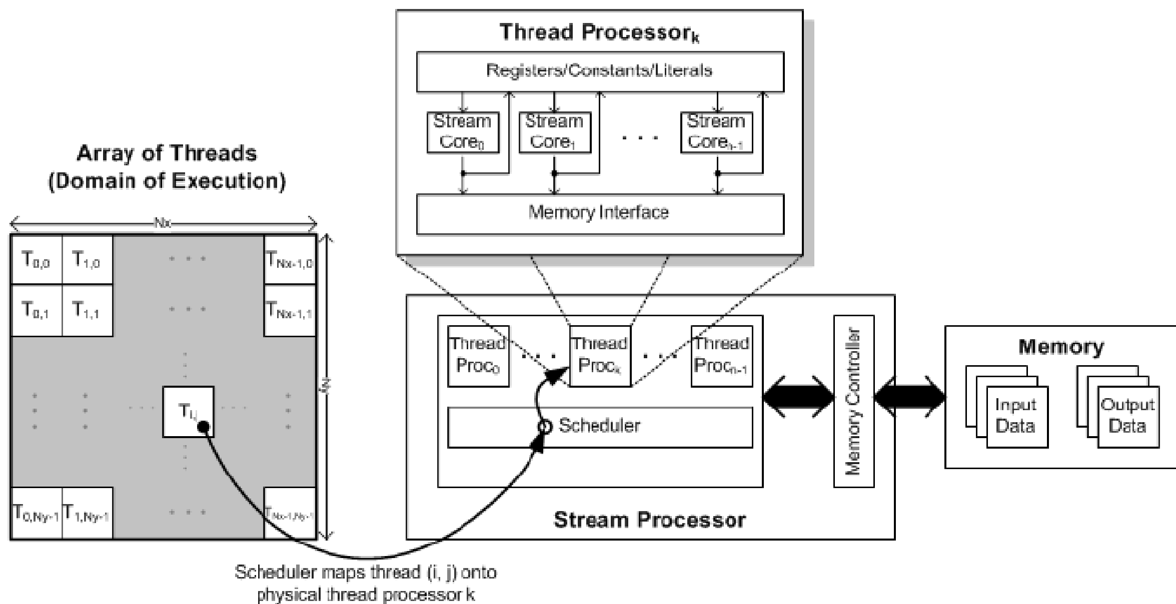


Figura 3.1, CAL trasforma effettivamene la GPU in un grande array di calcolo SIMD virtualizzato.

Le risorse sono accessibili in contesti multipli sullo stesso device e rappresentano la stessa memoria sottostante. La condivisione dati tra contesti è possibile mappando la stessa risorsa in contesti multipli. La sincronizzazione di contesti multipli è responsabilità del client. Come illustrato a sinistra in figura 2.5, ci sono due memorie condivise sul RV770; tuttavia, la versione corrente dell'SDK non fornisce alcun metodo per accederle, benchè dovrebbe essere disponibile nella prossima release.

Per scrivere un'applicazione CAL, i programmatori devono seguire questi passi:

- Scrivere il kernel nell'IL AMD.
- Inizializzare l'ambiente CAL, aprire una connessione al dispositivo CAL.
- Compilare e caricare il kernel.
- Allocare la memoria.
- Preparare i dati di input nella memoria.
- Associare la memoria ai buffer di input e output.
- Specificare il dominio d'esecuzione.
- Eseguire il kernel.

Compiler

Il compilatore CAL fornisce un'interfaccia runtime ad alto livello per compilare i device kernel scritti nello pseudo-assembly AMD IL e generare l'ISA specifico ad una GPU. Le routine del compilatore possono essere invocate sia a runtime che offline. Il primo meccanismo è tipicamente usato durante lo sviluppo

dei kernel quando lo sviluppatore ha costantemente bisogno di modificare il kernel e testare i risultati di output dati dal kernel. Il secondo è adatto per applicazioni che includono kernel che siano già stati sviluppati in precedenza e che hanno bisogno solamente di essere caricati e invocati a runtime. Questo meccanismo evita l'overhead associato al dover compilare il kernel ogni volta che l'applicazione è eseguita.

Intermediate Language

L'IL è un linguaggio pseudo-assembly che può essere utilizzato per sviluppare sia programmi grafici (vertex, geometry e pixel shaders) sia programmi general purpose data-parallel (kernel). L'IL è progettato in modo tale che i programmi possano essere sviluppati per girare su molteplici piattaforme senza il bisogno di essere riscritti per ognuna d'esse. L'IL è un linguaggio senza tipi, perciò le variabili, i parametri, etc., non hanno uno specifico tipo e possono rappresentare interi a 32-bit con segno, senza segno, numeri in virgola mobile a singola precisione, e numeri in virgola mobile a doppia precisione a 64-bit (supportati solamente quando la piattaforma sottostante supporti i numeri a doppia precisione) senza dover prima definire il tipo di una variabile.

L'utilizzo dell'IL rende più agevole l'ottimizzazione dei kernel, poichè i programmatori hanno maggiore controllo sull'allocazione di memoria, sull'utilizzo dei registri e set d'istruzioni usato senza richiedere una profonda conoscenza dell'architettura GPU. Allo stesso tempo, in modo simile a tutti i linguaggi assembly, la curva d'apprendimento è più alta rispetto a linguaggi ad alto livello quali il C. Mettendo a disposizione il linguaggio Brook+ per definire i kernel, AMD permette anche a programmatori non-HPC di avvantaggiarsi delle prestazioni dei processori stream.

Poiché l'IL verrà utilizzato come linguaggio target della compilazione della parte GPU di PBricks e per anticipare alcune delle problematiche incontrate (trattate nel capitolo 8) di seguito riportiamo una trattazione più esaustiva di questo linguaggio (per una trattazione completa si consulti [24]).

Instruction Packet	Description
1	IL_Lang token.
2	IL_Version token.
3	IL_OpCode token describing the operation of the first instruction in the stream and the beginning of the first IL instruction packet.
...	*... More IL instruction packets.*
n (number of 32-bit tokens in the stream)	*IL instruction packet for an* END *instruction.*

Figura 3.2, formato dello stream rappresentante un kernel

Il client (per la precisione la componente host del runtime di CAL) passa i kernel come uno stream di token a 32-bit organizzati come pacchetti di istruzioni a lunghezza variabile. L'intestazione comprende il tipo di linguaggio e la versione, seguiti dai pacchetti che rappresentano le istruzioni. Il kernel deve terminare necessariamente con l'istruzione `end`.

I pacchetti d'istruzione contengono tutte le informazioni necessarie per eseguire la singola istruzione specificata nel token IL_OpCode. Queste informazioni possono includere dati riguardanti gli operandi, etichette ed eventuali modificatori. C'è una classe particolare d'istruzioni usate dichiarare risorse, sampler e registri. E'importante che ogni dichiarazione di oggetto avvenga prima del primo utilizzo nel codice. Non è invece necessario raggruppare tutte le dichiarazioni in un singolo blocco.

Il linguaggio usato per definire il kernel può essere `il_vs`, `il_ps`, `il_cs`, che indicano rispettivamente un vertex shader, un pixel shader o un computer shader; alcune delle funzionalità più avanzate sono disponibili sono nei compute shader.

Le tipologie principali di registri sono:

- Registri di input (`v`), sono registri di sola lettura, contengono le coordinate all'interno della risorsa di input del work-item su cui il thread deve lavorare. E' possibile dichiarare una risorsa di input grazie all'istruzione `dcl_input`.
- Registri di output (`o`), sono registri di sola scrittura. Permettono di scrivere su una risorsa di output alla posizione del dominio d'esecuzione corrispondente al thread calcolato. E' possibile dichiarare una risorsa di output grazie all'istuzione `dcl_output`.
- Registri costanti (`cb`), sono registri di sola lettura. E' possible dichiarare un buffer delle costanti come: `dcl_cb cbm[n]` dove `m` può assume valori nel range [0, 14], e `n` può essere massimo 4K.
- Registri temporanei (`r`), usabili sia in lettura che in scrittura.
- Registro relativo al global buffer (`g`), una risorsa accedibile in lettura e scrittura a qualunque posizione. Questo la contraddistingue dalle risorse di input e output, che una volta fissato il thread del dominio d'esecuzione sono accedibili in una sola ben determinata posizione.
  C'è un solo registro per indicare il global buffer, mentre a livello di API si potrebbero teoricamente definire anche più risorse globali. Ogni indirizzo corrisponde a una locazione a 128-bit organizzata come 4 doppie parole. Esempio di utilizzo:

  ```
  mov g[2].x, r0
  ```

- Registro `vObjIndex`, contiene l'indice assoluto dell'elemento corrente calcolato (il primo elemento del dominio d'esecuzione ha indice 0, il secondo 1, e cosi via).
- Registro `vWinCoord`, il suo utilizzo è possibile solo in un pixel shader. Le prime due componenti, x e y contengono le coordinate del pixel all'interno del dominio d'esecuzione. La terza componente w contiene la coordinata z del pixel nello spazio di finestra. Assume gli stessi valori dei registri di input.

Vi sono tre tipi d'indirizzamento dei registri:

- *Assoluto*, `mov r0, r[9]`
- Base relative, `mov r0, r[r1.x + 5]`, solo un livello di annidamento è permesso.

- Loop relative, `mov r0, r[aL + 5],` gli offset sono relativi ad un registro particolare `aL`, usato per contare le iterazioni in un loop.

Si può creare un input buffer tramite l'istruzione:

```
dcl_resource_id(n)_type(fmt,unnorm)_fmtx(fmt)_fmty(fmt)_fmtx(fmt)_fmtw(fmt)
```

dove `n` è l'id associato alla risorsa e `fmt` ne indica il formato (int, float, double, etc.). Grazie all'istruzione di sample (ossia lettura da una risorsa) un kernel può leggere i propri valori di input:

```
sample_resource(n)_sampler(m)  dst, src
```

dove generalmente `src` è un registro di input (che contiene le coordinate dell'elemento correntemente computato) e `dst` è il registro in cui scrivere il valore letto. E' inoltre possibile dichiarare costanti letterali (i cui valori non sono chiaramente modificabili) all'interno del corpo del kernel:

```
dcl_literal l0, 0,1,2,3
```

i valori delle costanti letterali possono essere utilizzati come indice nel caso di indirizzamento relativo. I valori possono essere di tipo decimale o esadecimale.

Write mask e operazione di swizzle

Poiché la dimensione dei registri è di 128 bit, è possibile definire maschere per accedere alle singole parole di un registro. Generalmente vengono usate nelle operazioni di scrittura per indicare quali componenti andare a scrivere: possono essere specificate una o più componenti, con un "_" si indica che la componente non sarà scritta, ed è possibile forzare valori delle componenti ad assumere i valori 0 o 1. Ad esempio:

```
r0.x
ro.y
r0.x_zw
r0.w_y
r0.__10
```

Tramite l'operazione di *swizzle* è possibile riarrangiare e/o replicare le componenti di un registro:

```
mov r0.xy, r1.wz
```

In questo caso si sta copiando il solo contenuto delle componenti w e z del registro `r1` rispettivamente nelle componenti x e y del registro `r0`.

Istruzioni

Uno stream IL è formato principalmente da pacchetti di istruzioni. Ogni istruzione inizia con un token IL_OpCode; il tipo e numero di token che seguono varia a seconda dell'operazione e di eventuali modificatori. Se presenti le informazioni sulle destinazioni precedono quelle sulle sorgenti.

Per quasi tutte le operazioni sono definite le versioni tipate su interi, float e double. Generalmente questo avviene premettendo all'opcode il tipo desiderato. Ad esempio: `add` è la somma tra float, `iadd` è la somma tra int, `d_add` è la somma tra double.

Benchè IL sia un linguaggio non tipato, sono richieste istruzioni di conversione per cambiare il formato dei dati, in modo tale che i valori degli operandi siano sempre nel formato consistente con quello dell'operazione desiderata. Tali istruzioni di conversione sono `itof src` e `ftoi src`, che rispettivamente convertono un valore intero in valore a virgola mobile a singola precisione e viceversa. L'utilizzo di operandi che non rispettano il tipo dell'operazione non provoca errori in esecuzione, ma produce valori inaspettati.

Nel caso di istruzioni per valori in virgola mobile a doppia precisione è necessario sottolineare che questi occupano due parole all'interno di un registro quindi è necessario utilizzare write mask del tipo `.xy` o `.zw` per indicare il registro destinazione. Ad esempio per effettuare la somma di un numero in doppia precisione con se stesso si può fare:

```
d_add r1.xy, r2.xy, r2.xy
```

o

```
d_add r1.zw, r2.xy, r2.xy
```

Istruzioni per il controllo di flusso

Determinano l'ordine con il quale le istruzioni sono eseguite dall'hardware. Il CAL IL presenta una scelta insolita a livello di linguaggio che ha creato non pochi problemi in fase di analisi e traduzione (capitolo 9): non fornisce istruzioni di salto. Al suo posto troviamo istruzioni di più alto livello come *if*, *whileloop*, *continue*, etc., che se da un lato facilitano lo sviluppo direttamente in codice IL, dall'altro complicano notevolmente la scrittura di generatori di codice.

Il costrutto della *selezione* può essere espresso tramite l'istruzione:

```
ifc_relop(X)src1, src2
```

dove `X` è uno dei valori della seguente enumerazione:

- `eq`, equal
- `ge`, greater than or equal
- `gt`, greater than
- `le`, less than or equal

- lt, less than
- ne, not equal

oppure con if_logicalz src (if_logicalnz src). In questo caso il registro src deve contenere un valore intero che viene di cui viene testata l'uguaglianza col valore zero.

L'istruzione else, indica l'inizio di un blocco di statement per il ramo "else" di uno statement di selezione ed endif termina lo statement di selezione.

I cicli sono esprimibili con whileloop / endloop. Gli statement che rappresentano il corpo del ciclo devono essere racchiusi tra queste due istruzioni. Per esprimere le condizioni di terminazione si usano le seguenti istruzioni:

- break, termina incondizionatamente un loop, il controllo passa all'istruzione successiva a quella di endloop.
- breakc_relop(X) src1, src2; esprime una condizione di terminazione usando valori in virgola mobile a singola precisione, considera la sola componente x degli operandi.
- break_logicalz src (break_logicalnz src), esprime la condizione di terminazione di un loop usando interi; deve essere espressa una componente del registro src. Il break viene effettuato se tutti i bit nella componente selezionata sono 0.
- continue, passa incondizionatamente il controllo all'istruzione immediatamente successiva al primo loop (cioè l'istruzione che segue whileloop). Gli stessi modificatori presentati poc'anzi per l'istruzione break si applicano anche all'istruzione continue.

Definizione di funzioni

E' possibile definire funzioni all'interno di un kernel; queste devono seguire la definizione della funzione principale (il main):

```
func <integer-label>
… corpo della funzione …
ret
endfunc
```

In caso di presenza di funzioni è necessario indicare la fine della funzione principale con l'istruzione endmain. Le funzioni possono essere chiamate con l'istruzione call seguita dalla label che indica il nome della funzione. E' permessa la ricorsione diretta o indirezza. Il nesting delle chiamate (funzioni richiamate da altre funzioni) è permesso fino a 32 livelli di profondità. Ulteriori chiamate sono ignorate. Usandio l'istruzione call logicalz src <integer-label> (o logicalnz) si passa il controllo alla funzione avente come identificatore integer-label se e solo se la componente x del registro src è diversa (o uguale a) zero. E' possibile ritornare da qualsiasi funzione usando l'istruzione ret_dyn.

Modello di accesso per Local Shared Memory

Ci sono due modelli per l'accesso a questo tipo di memoria:

- *owner compute rules*, ogni thread di un gruppo possiede una porzione dell'area di memoria LDS. La dimensione dell'area è dichiarata nel kernel usando l'istruzione `dcl_lds_size_per_thread n` dove `n` è la dimensione in dword, che non può essere maggiore di 64 e deve essere un multiplo di 4 (essendo le celle a dimensione fissa di 128-bit). Ogni thread del gruppo può solamente scrivere nella propria porzione di memoria. Tuttavia è possibile leggere la propria area e quella dei thread, e quella degli altri thread. Gli indirizzamenti avvengono tramite un ID (del thread) e un offset (per accedere alla singola cella di memoria).
- *general read write*, ogni thread può leggere e scrivere a qualsiasi indirizzo della memoria LDS. Le sincronizzazioni sono lasciate al programmatore.

E' possibile specificare anche il modello d'accesso a livello di wavefront che può essere relativo o assoluto: nel primo caso per ogni wavefront sarà assegnata una porzione adiacente di spazio di memoria, mentre nel secondo caso tutti i wavefront faranno riferimento alla stessa porzione di memoria, rendendo di fatto possibile per un thread sovrascrivere i dati di un altro thread.

```
dcl_lds_sharing_mode _wavefrontRel/_wavefrontAbs
```

E' possibile conoscere l'ID di un thread all'interno del proprio gruppo (id relativo) e l'ID assoluto tramite i registri `vTid` e `vaTid`. Tramite il registro `vTGroupid` si può conoscere l'indice del gruppo. Le letture avvengono con l'istruzione `lds_read_vec dst, src.xy`. La componente x del registro `src` deve essere settata al valore del thread_id, e la componente y rappresenta l'offset. Le scritture avvengono tramite `lds_write_vec _lOffset(n) dst, src. Il` modificatore `_Offset(n)` è opzionale, serve per indicare la posizione all'interno dell'area di memoria posseduta dal thread in cui scrivere; se omesso la scrittura avverrà in posizione 0, se presente deve essere un multiplo di 4 e minore di 60. `dst` contrariamente a quanto si potrebbe pensare non indica un registro, bensì è usato solamente per fornire una write mask.

Le sincronizzazioni avvengono tramite l'istruzione `fence [_threads][_lds][_memory][_sr]` che è l'equivalente di un'istruzione di tipo *barrier*. I modificatori indicano quale tipo di barrier effettuare. Almeno una delle opzioni deve essere inclusa; i compute shader possono usare tutte le opzioni, mentre i pixel shader solamente `_memory`.

Con `dcl_num_thread_per_group n1, n2, n3` si dichiara il numero di thread all'interno di un gruppo. Il prodotto delle 3 costanti non può essere maggiore di 1024. Sulla famiglia di dispositivi HD 4000 `n2` e `n3` devono essere posti a 1.

Brook+

Brook+ è un'estensione del linguaggio C per la programmazione stream cross-platform; esso dispone di una sintassi per rappresentare e manipolare Streams, e per specificare Kernels. E' uno standard open source progettato per essere semplice da usare, capace di interoperare con backend multipli (ossia DirectX and OpenGL)

La struttura di un programma Brook+ può essere modellata attorno ad un grafo, dove i nodi manipolano dati e gli archi indicano il flusso dei dati all'interno del sistema. Un *nodo* può o ristrutturare i dati o effettuare computazioni, ma non entrambe. I nodi che ristrutturano dati sono riferiti come operatori stream, mentre i nodi che eseguono calcolo come kernels. Un *arc* (stream) connette due nodi. Non fornisce alcuno storage, invece mappa l'uscita di un nodo nell'input(s) di uno o più altri nodi.

Gli stream risiedono nella memoria GPU e la loro dimensione è limitata da restrizioni dell'hardware della GPU. Non possono essere acceduti direttamente dal programma dell'applicazione host, ma devono essere usati degli operatori stream.

Tutte le operazioni Brook+ sono non-bloccanti a meno che non sia necessaria la sincronizzazione. La sincronizzazione è gestita dal runtime di Brook+. Pertanto, è importante valutare attentamente l'ordine di chiamata delle funzioni.

L'interfaccia di programmazione Brook+ consiste di:

- Un insieme minimale di estensioni al linguaggio C che permette al programmatore di specificare porzioni del codice sorgente per l'esecuzione sul dispositivo.
- Una libreria runtime divisa in:
  - *Host component*, che è eseguita sulla CPU e fornisce le funzioni per controllare ed accedere ad uno o più dispositivi di calcolo dall'host.
  - *GPU kernel*, che legge i dati di input, esegue la computazione stream parallel e scrive i dati di output.
  - *Common component*, che fornisce tipi vettoriali predefiniti e un sottoinsieme della libreria standard C che sia supportato sia sull'host sia sul codice device.

Per scrivere un'applicazione Brook+, i programmatori devono seguire questi passi:

- Scrivere il kernel.
- Inizializzare l'ambiente Brook+.
- Allocare la memoria.
- Associare la memoria agli stream di input e output.
- Preparare i dati di input in memoria.
- Eseguire il kernel.

Durante lo sviluppo del kernel, uno sviluppatore deve considerare alcune restrizioni, cosi come dichiarato della specifica di Brook+ [30], a causa della natura dei kernel. Questi sono eseguiti sulla GPU e comunicano con la CPU solamente tramite streams, quindi tutti gli accessi alla memoria di sistema sono limitati alle letture dagli stream di input/gather. Per quel che riguarda la gestione della memoria locale,

l'allocazione dinamica della memoria non è permessa, e in generale i puntatori non sono supportati. Inoltre all'interno di un kernel non è permesso scrivere variabili statiche o globali. Per quel che riguarda le chiamate di funzione, è vietato chiamare funzioni che non siano kernel dall'interno di un kernel e la ricorsione, mentre è possible chiamare non-void kernel. Per quel che riguarda la firma dei kernel, solo chiamate a kernel che restituiscono `void` come tipo di ritorno sono permesse dall'applicazione.

Compiler

Il compilatore Brook+, chiamato *brcc*, è un compilatore source-to-source che genera codice C++ che viene usato dal runtime. *brcc* si affida a tool di terze parti (il compilatore CAL driver) per la generazione del codice e le ottimizzazioni. E' basato sul parser open-source cTool, sul generatore di codice IL e sul componente CAL runtime. In figura 3.3Figura 3.3, sono mostrati i blocchi costituitivi e i relativi output. Il generatore di codice AMD IL traduce la computazione kernel in istruzioni IL, senza effettuare alcuna ottimizzazione. Inoltre mappa i parametri del kernel negli stream CAL e nei parametri non-stream, e genera i metadati per descrivere l'informazione di mapping.

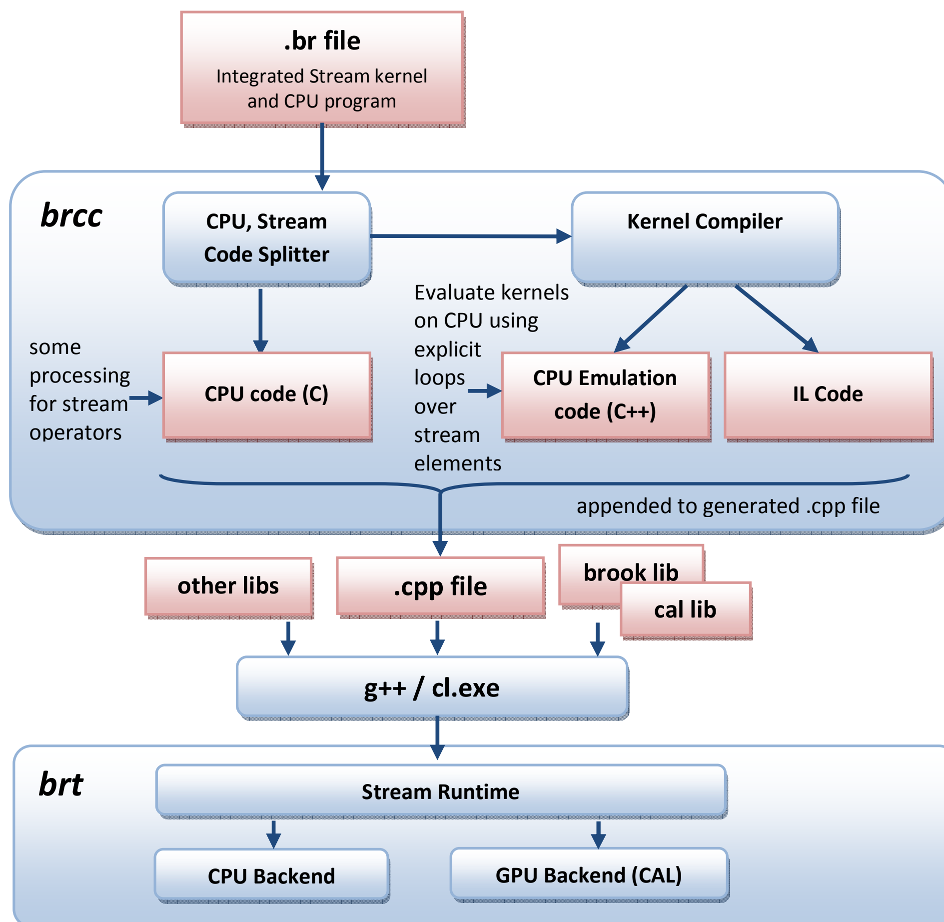


Figura 3.3. Architettura di Brook+: compiler + runtime.

Si faccia riferimento all'appendice per un esempio di codice che mostri i differenti obiettivi di progetto di CAL e Brook+.

CAL garantisce un controllo più fine sulle implementazioni del kernel, dovuto al linguaggio di livello assembler fornito. Come conseguenza, CAL richiede anche che i programmatori prendano in considerazione aspetti non funzionali della computazione su stream, come la gestione del dispositivo, la gestione delle risorse, la generazione del codice, caricamento ed esecuzione del kernel.

Brook+ fornisce una visione astratta della computazione GPU, cosi che gli sviluppatori possano focalizzarsi solamente sullo sviluppo dei kernel, senza dover considerare nessun dettaglio inerente la GPU. Questo linguaggio a più alto livello rende più facile iniziare a scrivere computazioni su stream, tenendo bassa la curva d'apprendimento. Tuttavia, tutte le ottimizzazioni sono affidate al framework sottostante.

3.1.3 Nvidia

Nvidia offre un modello di programmazione parallela e un ambiente software, chiamato CUDA, progettati per superare la sfida dello sviluppo di applicazioni che richiedono il parallelismo al fine di sfruttare il crescente numero di core GPU. Allo stesso tempo, il design mantiene una curva d'appredimento bassa per i programmatori che sono familiari con i linguaggi di programmazione standard come il C. Un CUDA possiede tre astrazioni chiave: una gerarchia di gruppi di thread, memorie condivise, e sincronizzazione tramite barrier che sono esposte allo sviluppatore semplicemente come un insieme minimale di estensioni al C.

Compute Unified Device Architecture

CUDA è un'architettura software per l'emissione e gestione di computazioni sulla GPU come un dispositivo di elaborazione di dati in parallelo, senza il bisogno di mapparle nella specifica API grafica [27, 29, 31]. E' disponibile per la serie 8 della GeForce 8, la serie GeForce GTX 2xx, le soluzioni Tesla, ed alcune delle soluzioni Quadro.

CUDA estende il C dando la possibilità al programmatore di definire speciali funzioni C (kernels) che, all'invocazione, sono eseguite N volte in parallelo da N differenti thread CUDA, al contrario di una sola come le normali funzioni C. Ogni thread è contenuto in un blocco (thread block). I thread all'interno di un blocco possono cooperare tra di loro grazie alla condivisione di dati usando *shared memory*, e sincronizzare la propria esecuzione per coordinare gli accessi in memoria. Il numero di thread per blocco è soggetto a restrizioni date le limitate risorse di memoria di un processore core. Sull'architettura Nvidia Tesla, un blocco di thread può contenere fino a 512 thread. Tuttavia, un kernel può essere eseguito da molteplici blocchi di thread della stessa dimensione (detti thread-group), cosicchè il numero totale di thread è uguale al numero di thread per blocco moltiplicato per il numero di blocchi. Gli sviluppatori possono specificare il numero di blocchi e il numero di thread per ogni blocco tramite il primo e il secondo parametro della firma della funzione kernel.

Lo stack sofware CUDA è composto da diversi livelli: un device driver, un'interfaccia di programmazione per le applicazioni (API) e il suo runtime, e due librerie matematiche di alto livello di uso comune, CUFFT e CUBLAS. L'interfaccia di programmazione CUDA consiste di:

- Un insieme minimale di estensioni al linguaggio C che permettono al programmatore di specificare porzioni del codice sorgente per l'esecuzione sul dispositivo.
- Una libreria runtime divisa in:
    - *Host component*, che è eseguita sulla CPU e fornisce le funzioni per controllare ed accedere ad uno o più dispositivi di calcolo dall'host.
    - *Device component (kernel)*, che è eseguita sul dispositivo e fornisce funzioni specifiche del dispositivo. Questa può essere di due tipi: __global__ e __device__.
    - *Common component*, che fornisce dei tipi vettoriali predefiniti e un sottoinsieme della libreria standard del C che è supportato sia sul codice host che in quello device.

CUDA introduce dei qualificatori per i tipi delle funzioni allo scopo di definire dove una funzione è eseguita, quali altre funzioni può chiamare e quali la possono invocare:

- Eseguibile sull'host, chiamabile dalla componente host, le funzioni di tipo __host__ possono invocare tutte quelle funzioni che non sono di tipo __device__.
- Eseguibile sul dispositivo, chiamabile dalla componente host, le funzioni di tipo __global__ possono invocare esclusivamente le funzioni di tipo __device__.
- Eseguibile sul dispositivo, chiamabile dalla componente device, sono le funzioni di tipo __device__.

Le funzioni di tipo __global__ e __device__ hanno le stesse restrizioni dei kernel Brook+, questo significa che non supportano la ricorsione, non vi possono essere dichiarate variabiali statiche, e non possono avere un numero variabile di argomenti. Come per Brook+, le funzioni __global__ sono asincrone quindi il controllo ritorna al codice chiamante prima che il dispositivo abbia completato la sua esecuzione. Una caratteristica importante è sulla signature, siccome le funzioni __global__ devono specificare la propria configurazione d'esecuzione. Questa definisce la dimensione della griglia e i blocchi che saranno usati per eseguire la funzione sul dispositivo, nonchè lo stream associato. E' specificata inserendo un'espressione del tipo *<<< Dg, Db, Ns, S >>>* tra il nome della funzione e la lista degli argomenti tra parentesi, dove:

- ***Dg*** specifica le dimensioni della griglia, in modo tale che Dg.x * Dg.y è uguale al numero dei blocchi in fase di avvio, mentre Dg.z è inutilizzato;
- ***Db*** specifica le dimensioni di ogni blocco, tale che Db.x * Db.y * Db.z è uguale al numero dei thread per blocco;
- ***Ns*** specifica il numero di byte nella memoria condivisa che sono allocati dinamicamente per ogni blocco in aggiunta alla memoria allocata staticamente; questa memoria allocata dinamicamente è usata da ognuna delle variabili dichiarate come un array esterno; è un argomento opzionale che per default vale 0;
- ***S*** specifica lo stream associato; è un argomento opzionale che per default vale 0.

La componente Device Runtime mette a disposizione una *funzione di sincronizzazione*, utilizzabile per coordinare la comunicazione tra thread dello stesso blocco. Essa permette di implementare la reduce

direttamente sul dispositivo, evitando inutili comunicazioni GPU-CPU, per esempio con algoritmi che implementano il paradigma stencil.

Un'altra importante feature fornita sono le *istruzioni atomiche*. Effettuano operazioni atomiche di lettura-modifica-scrittura su parole di 32-bit o 64-bit che risiedono nella memoria global o shared. L'operazione è atomica nel senso che è garantita l'indivisibilità (eseguita senza interferenza da altri thread). Queste istruzioni sono molto importanti dato che il loro utilizzo rende possibile implementare la sincronizzazione tra thread sulla memoria globale, quindi tra thread multipli appartenenti a blocchi distinti.

Per scrivere un'applicazione CUDA, i programmatori devono seguire i seguenti passi:

- Scrivere il kernel.
- Inizializzare l'ambiente CUDA.
- Allocare la memoria.
- Preparare i dati di input in memoria.
- Eseguire il kernel.

Il compilatore

La compilazione CUDA separa le funzioni device dal codice host, compila le prime utilizzando un compilatore/assemblatore proprietario Nvidia, chiamato *nvcc*, compila il codice host usando un compilatore C/C++ general purpose che è disponibile sulla piattaforma, e in seguito inserisce le funzioni GPU compilate come immagini caricate nel file oggetto sull'host. Nella fase di linking, sono aggiunte specifiche librerie runtime CUDA per permettere la chiamata di procedure SIMD remote e per permettere manipolazioni esplicite della GPU come l'allocazione di buffer di memoria GPU e trasferimenti di dati da host a GPU. Nel seguente schema è mostrato l'intero processo di compilazione. Nella toolchain CUDA ci sono:

- *cudafe*, il front end CUDA. E' invocato due volte: la prima volta per dividere effettivamente il file .cu passato come input in codice host e codice device; la seconda volta è per effettuare la dead code analysis sul file .gpu generato al passo precedente.
- *nvopencc*, l'implementazione Nvidia di Open64[24] che genera codice PTX.
- *ptxas* è il compilatore Parallel Thread Execution (PTX) che traduce il PTX pseudo-assembly nell'ISA specifico di una GPU.

Lo scopo di *nvcc* è nascondere gli intricati dettagli della compilazione CUDA agli sviluppatori.

[24] Open64 è un compilatore C/C++ open-source originariamente creato per l'architettura Intel Itanium.

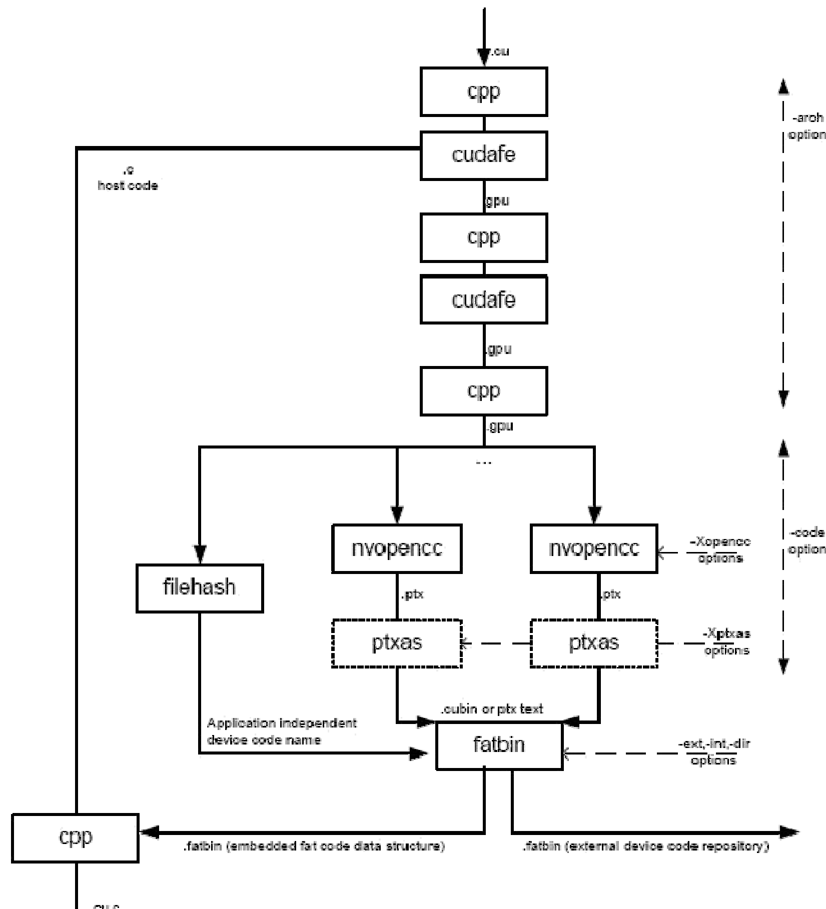


Figura 3.4, il processo di compilazione di un'applicazione CUDA

Parallel Thread Execution

PTX definisce una macchina virtuale e un ISA (linguaggio pseudo-assembly) per l'esecuzione di general-purpose parallel thread. E' progettato per essere efficiente sulle GPU Nvidia supportando le caratteristiche definite dalle architetture Nvidia.

I programmi PTX sono tradotti e tempo di installazione nel set di instruzioni dello specifico hardware. Il traduttore PTX-to-GPU e il driver consentono alle GPU Nvidia di essere usate come computer paralleli programmabili.

Il modello di programmazione PTX è esplicitamente parallelo: un programma PTX specifica l'esecuzione di un dato thread facente parte di un array di thread paralleli. Un cooperative thread array, o CTA, è un vettore di thread che eseguono concorrentemente o in parallelo un kernel. I thread all'interno di un CTA possono comunicare l'uno com l'altro. Per coordinare la comunicazione tra thread dello stesso CTA, si possono specificare punti di sincronizzazione dove i thread attendono fino a quando tutti i thread del CTA non sono giunti. I thread all'interno di un CTA eseguono in modo SIMT in gruppi (detti warp). Per quel che riguarda le performance, è importante ricordare che se thread di un warp divergono tramite

branch condizionali dipendenti dai dati, il warp esegue sequenzialmente ogni percorso di branch intrapreso, disabilitando i thread che non sono su quel percorso, e quando tutti i percorsi sono completati, i thread convergono nuovamente nello stesso flusso d'esecuzione. Per un esempio di programma CUDA si faccia riferimento all'appendice.

I programmatori possono implementare i kernel usando il C quindi la curva d'apprendimento è abbastanza bassa. Tuttavia, anche se è possibile ottimizzare il codice a questo livello (ad esempio con la gestione della memoria shared), CUDA non può dare lo stesso controllo a grana fine fornito da CAL e il compilatore attuale non sembra abbastanza maturo per produrre codice altamente performante. L'uscita della Fermi introduce anche il nuovo set d'istruzioni PTX 2.0. Il PTX 2.0 aggiunge nuove funzionalità che permettono di migliorare la programmabilità della GPU, le prestazioni e l´accuratezza dei risultati forniti. Fra queste troviamo in primis il pieno supporto alla programmazione diretta con il *linguaggio C++* e poi il supporto alla precisione a 32-bit, uno spazio indirizzi unificato per variabili e puntatori, indirizzamento a 64-bit e nuove istruzioni per OpenCL e DirectCompute. Lo spazio di indirizzi unificato si rende necessario per poter programmare in C++: questo linguaggio fa infatti uso di puntatori per indicare gli oggetti che contengono variabili e funzioni. Sinora con il PTX versione 1.0 si avevano a disposizione tre differenti spazi di indirizzi per le operazioni di load e store e tali spazi venivano resi noti solo a runtime. La versione 2.0 di PTX crea uno spazio unificato utilizzando puntatori a 40-bit che permettono di indirizzare fino ad un terabyte di memoria mentre le operazioni di load e store supportano indirizzamenti a 64-bit per poter gestire anche sviluppi futuri.

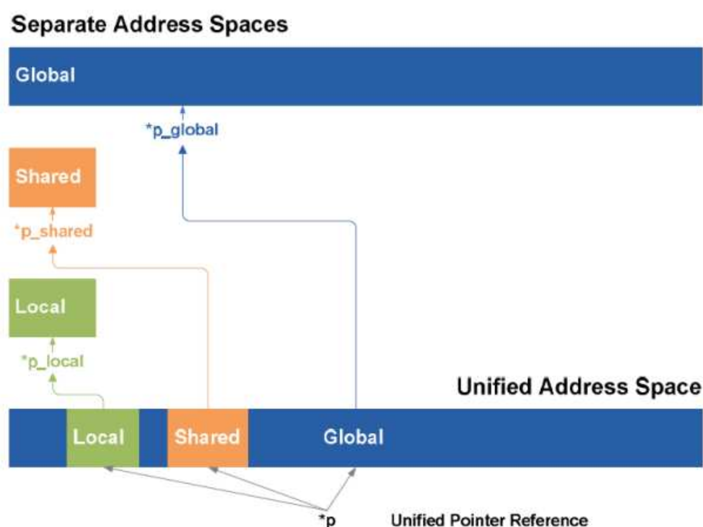


Figura 3.5, differenza tra spazio degli indirizzi distinto e unificato

Essendo un'architettura pensata per CUDA, Fermi è automaticamente ottimizzata anche per l´esecuzione di codice OpenCL e DirectCompute i cui paradigmi non differiscono affatto da quelli utilizzati nell´ambiente proprietario Nvidia. Il supporto per lo standard IEEE 754-2008 implica anche la disponibilità di 4 algoritmi di approssimazione (nearest, zero, infinito positivo, infinito negativo) e della gestione in hardware dei numeri subnormal. Quest´ultima feature permette di avvicinarsi allo zero (con

subnormal sono indicati numeri molto piccoli compresi fra lo zero ed il numero più piccolo di un dato sistema floating point) spendendo risorse molto limitate.

Nvidia Nexus

Nexus è il primo ambiente di sviluppo progettato specificatamente per supportare applicazioni massicciamente parallele CUDA C, OpenCL, e DirectCompute. Colma il gap in termini di produttività tra il codice CPU e quello GPU dando la possibilità di effettuare il debugging del codice su hardware parallello e analisi delle performance direttamente all'interno di Microsoft Visual Studio, il più diffuso ambiente di sviluppo integrato sotto Windows. Nexus permette agli sviluppatori Visual Studio di scrivere e debuggare codice sorgente per GPU allo stesso modo e con gli stessi tool e interfacce che sono abituati ad usare per gestire il codice CPU, inclusi i breakpoint sul sorgente e sui dati, e l'ispezione della memoria. Inoltre, Nexus estende le funzionalità di Visual Studio offrendo tool per gestire il parallelismo massiccio, come la capacità di concentrarsi e debuggare su un singolo thread tra le migliaia di thread in esecuzione parallela, e la capacità di visualizzare con facilità ed efficienza i risultati calcolati da tutti i thread paralleli. Nexus è l'ambiente di sviluppo ideale per sviluppare applicazioni co-processing che sfruttano sia la CPU sia la GPU: cattura eventi inerenti le prestazioni su entrambi i processori, e presenta le informazioni per lo sviluppatore su un'unica timeline correlata. Questo permette agli sviluppatori di vedere come le proprie applicazioni si comportano e operano sull'intero sistema, piuttosto che tramite una visione ristretta, che si concentra su un particolare sottosistema o processore.

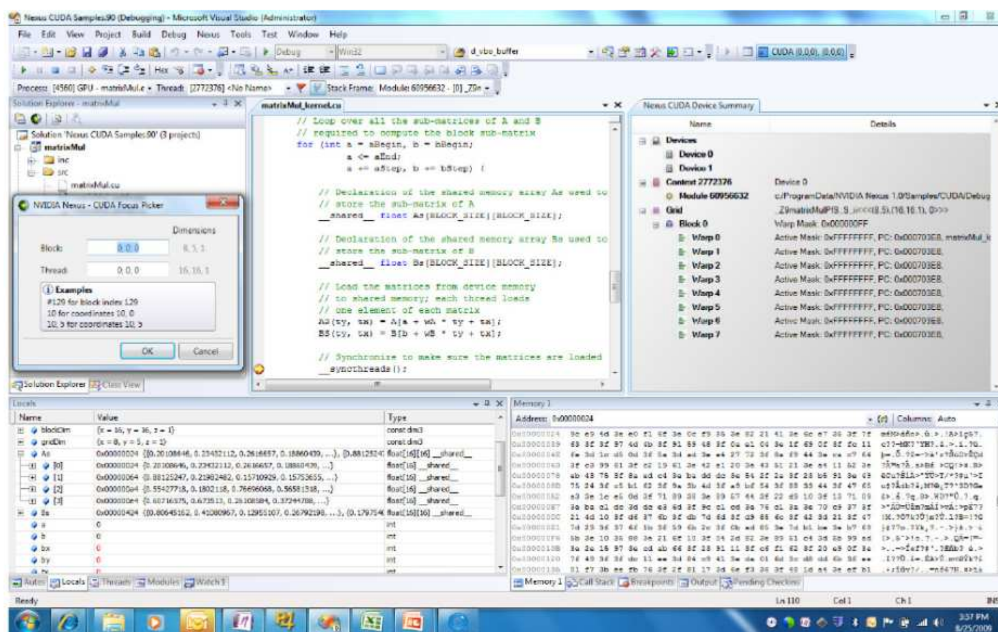


Figura 3.6, l'ambiente Nexus integrato all'interno di Microsoft Visual Studio

3.2 Modelli di memoria GPU

L'interfaccia con la memoria dipende in modo stretto dalle scelte di progetto e limitazioni dell'architettura sottostante.

3.2.1 AMD

Tutti i kernel CAL hanno accesso ai sottosistemi di memoria locale e remota dei device CAL. Nel caso delle GPU, la memoria locale corrisponde alla memoria video ad alta velocità situata sulla graphics board. In modo simile, la memoria remota corrisponde a memoria che non è locale al dato device ma è ancora visibile ad un insieme di dispositivi. Tutti i blocchi di memoria fisica allocati da un'applicazione CAL per essere usati nei device kernel sono riferiti come risorse.

Le risorse possono essere allocate sia localmente ad uno specifico dispositivo sia in modo remoto. In caso di allocazione remota, CAL permette all'applicazione di controllare la lista di dispositivi che possono accedere direttamente alla risorsa. La memoria remota può servire come un meccanismo per condividere risorse di memoria tra dispositivi multipli.

Le risorse CAL sono usate come input, output e costanti nei kernel CAL. Per gli input e le costanti, è desiderabile per prima cosa inizializzare i contenuti dei buffer di memoria dall'applicazione ospite. Per far ciò, la risorsa deve essere mappata nello spazio di indirizzamento dell'applicazione. Siccome una risorsa mappata non può essere usata in un kernel CAL, è necessario effettuare l'unmapping della risorsa prima del suo utilizzo. Quando una risorsa è stata allocata, essa deve essere legata ad un dato contesto prima di essere usata in un kernel CAL. Le risorse CAL non sono specifiche di un contesto. Quindi, devono essere prima mappate nello spazio di indirizzamento di un dato contesto prima di essere riferite da quel contesto. Quest'operazione restituisce un handle che può essere usato nelle seguenti operazioni, come lettura e scrittura nella risorsa.

3.2.2 Nvidia

I thread CUDA durante la loro esecuzione possono accedere a dati appartenenti a diversi spazi di memoria:

- Memoria locale, è una memoria per-thread, usata per le variabili automatiche e i registri. Infatti, una variabile automatica dichiarata all'interno di codice device senza alcuno specifico qualificatore generalmente risiede in un registro. Tuttavia in alcuni casi il compilatore potrebbe scegliere di piazzarla nella memoria locale. Questo è ad esempio il caso di ampie strutture dati o array che altrimenti consumerebbero troppo spazio dei registri.
- Memoria condivisa, è visibile a tutti i thread di un blocco ed ha lo stesso tempo di vita del blocco. E' responsabilità dei programmatori permettere comunicazioni inter-thread, e minimizzare spostamenti di dati con la memoria globale. Gli indirizzi di memoria shared sono mappati nei banchi di memoria come segue: i banchi sono organizzati in modo tale che parole consecutive da 32-bit sono assegnate a banchi consecutivi (organizzazione interallacciata) ed ognuno ha una banda di 32 bit per 2 cicli di clock. La memoria shared è inoltre caratterizzata da un meccanismo di broadcast in base al quale una parola da 32-bit può essere letta e trasmessa a diversi thread simultaneamente quando è servita una richiesta di lettura dalla memoria. Questo riduce il numero di conflitti a livello di banco quando diversi thread di un half-warp leggono da un indirizzo all'interno della stessa parola a 32-bit.

- Memoria globale, dove tutti i thread, in ogni blocco, possono accedere, permettendo comunicazioni inter-block.
- Ci sono inoltre due spazi aggiuntivi di memoria a sola lettura accessibili da tutti i thread: gli spazi di memoria constanti e texture. La memoria texture offre anche differenti modi d'indirizzamento, cosi come per il filtraggio dei dati, per alcuni specifici formati di dati. Gli spazi di memoria globale, costante, e texture sono persistenti rispetto a successive esecuzioni di kernel della stessa applicazione.

3.3 AMD vs Nvidia

Anche se offrono architetture similari SIMD, Nvidia e AMD hanno scelto modelli di programmazione abbastanza differenti. Come mostrato negli esempi di codice (vedi appendice), AMD fornisce due principali livelli di astrazione, con diversi livelli di controllo sui dispositivi, risorse e ottimizzazioni dei kernel. Con Brook+, i programmatori dovrebbero valutare solamente gli aspetti data-parallel della propria applicazione, senza prendere in considerazione i dettagli dell'architettura sottostante. Mentre usando CAL, è richiesta la considerazione degli aspetti non-funzionali della versione parallela per GPU (cioè compilazione del kernel, gestione del dispositivo e delle risorse), e conoscere l'IL per l'implementazione del kernel. Essendo un linguaggio di basso livello (pseudo-assembly), l'IL permette una migliore ottimizzazione dei kernel rispetto a Brook+, ma al costo di una più elevata curva d'apprendimento. Tuttavia, AMD fornise alcuni utili tool, per semplificare lo sviluppo in IL, come l'AMD ShaderAnalyzer [32], che compila al volo sia codice Brook+ in codice IL, sia IL in codice ISA di una specifica GPU. Sia Brook+ che CAL sono estensioni del linguaggio C (CAL come libreria), con Brook+ basato su una macchina virtuale offerta da CAL. Pertanto, l'espressività di Brook+ non è cosi diversa da quella di CAL se si eccettua il livello di dettaglio con cui il dispositivo è esposto agli sviluppatori. Inoltre è importante notare che *brcc* è solamente un traduttore, perciò nessuna ottimizzazione viene effettuata sul codice Brook+. Quindi la questione è se Brook+ sia o no la corretta risposta alla richiesta del mercato di un linguaggio di programmazione per GPU ad alto livello. La stessa domanda è valida per CUDA di Nvidia, poiché si tratta anch'esso di un'estensione al linguaggio C.

Un altro aspetto rilevante dell'interfaccia di programmazione è l'opportunità per gli sviluppatori di specificare la configurazione di esecuzione per ogni invocazione di kernel. Usando CUDA, i programmatori possono informare il compilatore, e il runtime sul numero di thread e blocchi richiesti per ogni computazione su GPU. Il runtime prenderà ancora decisioni sulla schedulazione dei thread e sulla distribuzione del carico nel rispetto dei vincoli dell'utente. Laddove utilizzando l'AMD CAL è solamente possibile definire il dominio d'esecuzione per ogni kernel; tuttavia, cambiando la dimensione del dominio, cambia anche il wavefront. Solamente a livello di IL si può specificare il numero di thread facenti parte di un gruppo.

Le toolchain di AMD e Nvidia sono molto simili, con un passo iniziale di traduzione che produce codice C standard, seguito dal reale passo di compilazione in codice ISA per la GPU. Attualmente, i traduttori disponibili non sembrano ancora maturi, siccome dai test sono riportate alcune segnalazioni di errori sbagliate da entrambi i compilatori brcc e nvcc. Per esempio, alcune parole chiave C non sono

riconosciute. Più nel dettaglio, nvcc protesta con le costanti dichiarate come *const type*s invece di usare #define. In aggiunta ha problemi a tradurre in eseguibile codice C non banale, specialmente usando il costrutto *switch-case*. Difatti, il codice intermedio (ptx) generato è cosi complesso ed enorme che il compilatore ptxas non riesce a gestirlo producendo un eseguibile privo delle prestazioni attese.

Capitolo 4

In questo capito si darà una panoramica delle altre API di programmazione per lo sviluppo di applicazioni data parallel su GPU. Si darà una descrizione della piattaforma RapidMind (recentemente acquisita da Intel), OpenCL, che mira ad unificare lo sviluppo su GPU e CPU multicore, ed infine Accelerator, che a tutt'oggi rimane l'unica soluzione basata su virtual machine (ma che come vedremo presenta alcuni problemi). Citiamo qui per completezza anche DirectCompute, che verrà introdotta nelle nuove DirectX 11 di Microsoft con il rilascio di Windows 7 (per maggiori informazioni si consulti [41])

4.1 Altre API per sviluppo di applicazioni data parallel

4.1.1 RapidMind

Intel combinerà la tecnologia della piattaforma RapidMind [38, 39] con il proprio progetto di ricerca Ct[25] [37], che è focalizzato sul facilitare lo sfruttamento dei propri chip multicore. RapidMind offre agli sviluppatori dei tool in grado di astrarre la programmazione data-parallel dall'hardware, cosi come i più conosciuti linguaggi di programmazione hanno fatto in passato per i processori a singolo core, fornendo allo stesso tempo performance forward-scaling tra i processori manycore e multicore. La tecnologia Ct non è legata a nessuna specifica architettura di processori, ma il modello sottostante richiede un'architettura di calcolo parallelo generalizzato come quella che si trova nei processori multicore e manycore.

RapidMind è una piattaforma di sviluppo ed un runtime che permette ad applicazioni single threaded di usufruire pienamente dei processori multi-core. Con RapidMind, i programmatori continuano a scrivere codice in standard C++ e a usare tool e processi pre-esistenti. Per ottenere risultati velocemente, gli sviluppatori, devono convertire solamente le parti più critiche delle proprie applicazioni. La restante parte del loro codice non è modificata. La piattaforma RapidMind quindi parallelizza l'applicazione tra i molteplici core e gestisce la sua esecuzione.

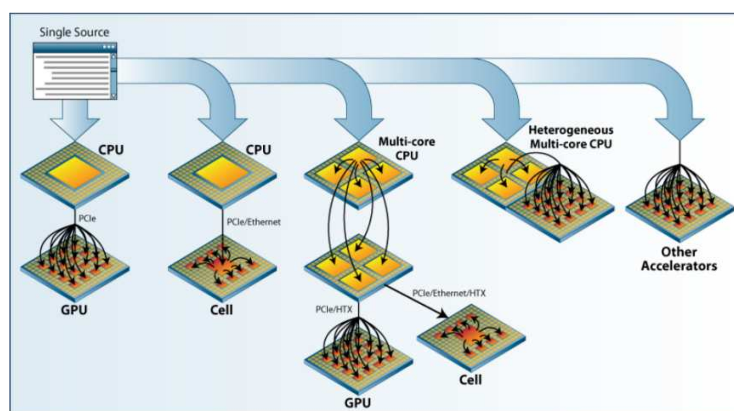


Figure 4.1, un programma scritto una volta può essere eseguito su più piattaforme

[25] C for Throughput Computing

La piattaforma RapidMind permette agli sviluppatori di parallelizzare facilmente e velocemente le applicazioni:

- Un'applicazione è espressa dal programmatore come una sequenza di funzioni applicate ad array
- La piattaforma RapidMind ripartisce automaticamente i dati dell'applicazione e il calcolo tra i core
- Il modello di calcolo su stream SPMD usato dalla piattaforma RapidMind, può scalare facilmente fino ad un largo numero di core mantenendo al tempo stesso la semplicità data dall'utilizzo di un singolo thread di controllo
- La componente runtime di RapidMind è inserita nell'applicazione risultante e gestisce ed ottimizza il carico di lavoro sul processore target

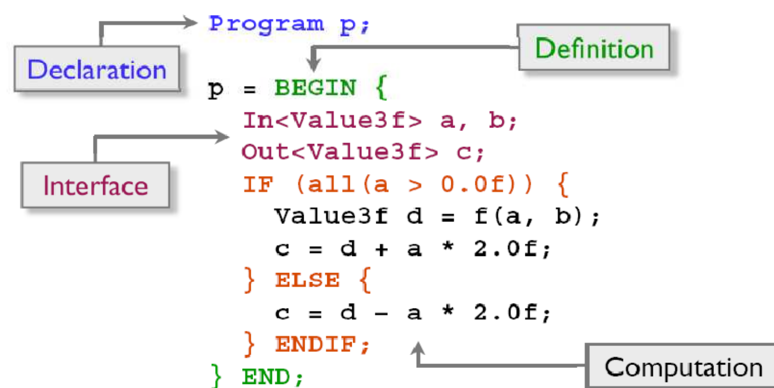


Figura 4.2, un tipico programma RapidMind

La piattaforma RapidMind svolge tutto il lavoro di basso livello richiesto per ottenere prestazioni elevate sui processori multi-core. In aggiunta ad adattare la computazione alle capacità e caratteristiche dello specifico processore target del deployment, s'incarica della gestione delle comunicazioni e del flusso dei dati tra il processore host e il/i dispositivo/i target.

Le componenti sono:

- Il *Code Optimizer* analizza e ottimizza la computazione per eliminare gli overhead, ad esempio rimuovendo le istruzioni che non è necessario eseguire.
- Il *Load Balancer* pianifica e sincronizza il lavoro per tenere pienamente occupati tutti i core.
  Uno schedulatore controlla quali risorse hardware sono disponibili sul sistema e decide il numero appropriato di kernel da mandare in esecuzione. Questo processo include l'utilizzo di un compilatore just-in-time (JIT) che traduce i sorgenti binari in un formato eseguibile dall'hardware di sistema. Per questo motivo i programmi scritti per questa piattaforma, funzioneranno su eventuale nuovo hardware, senza il bisogno di ricompilare l'applicazione.

- Il *Data Manager* riduce i colli di bottiglia riguardanti i dati.
- *Logging/Diagnostics* rilevano e riportano i colli di bottiglia sulle prestazioni.

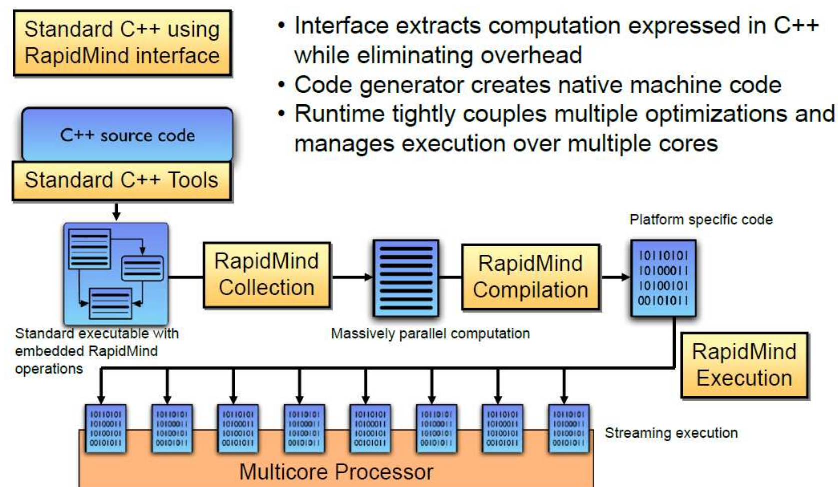


Figura 4.3, il processo di compilazione di RapidMind

La piattaforma RapidMind offre un insieme di back end (detti *Processor Support Modules*). Ognuno fornisce servizi che supportano l'esecuzione dei programmi RapidMind su un particolare processore. Lo sviluppatore non deve avere a che fare con i dettagli di ogni processore, ed è libero di scrivere applicazioni portabili lavorano su una varietà di processori target.

- Il backend x86 esegue i programmi RapidMind sulle CPU x86 di Intel e AMD
- Il backend GPU esegue i programmi RapidMind su una varietà di Graphics Processing Units (GPUs) sia di ATI che Nvidia
- Il backend Cell BE esegue i programmi RapidMind sui SPEs del Cell BE Broadband Engine
- Il backend Debug esegue i programmi RapidMind sul processore host, compilando i programmi con il compilatore C

Gli acceleratori quali le GPU o il Cell danno un'opportunità per un ulteriore miglioramento delle prestazioni rispetto a quelle di un tradizionale processore x86 (the host). Senza cambiare la logica applicativa, se è disponibile l'hardware, si può ottenere uno speed up addizionale usando gli acceleratori. La piattaforma RapidMind si fa carico autonomamente di gestire i movimenti di dati e la ripartizione del calcolo tra l'acceleratore e l'host.

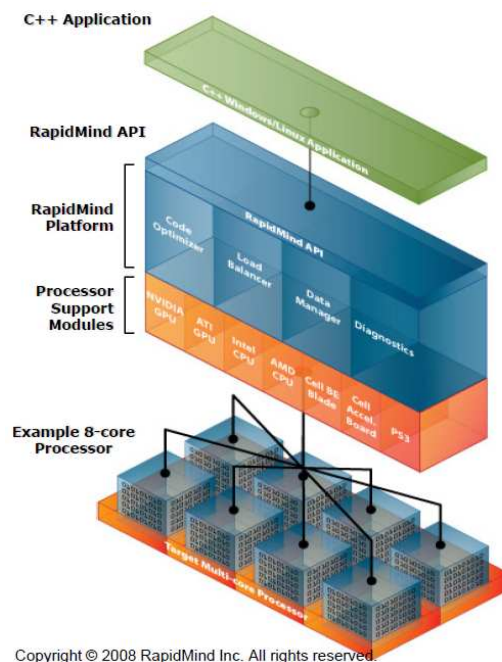


Figura 4.4, L'architettura della piattaforma RapidMind.

Dopo aver identificato le parti dell'applicazione che necessitano dell'accelerazione, il processo complessivo di integrazione si svolge in questo modo:

1. Rimpiazzare i tipi: lo sviluppatore sostituisce i tipi numerici che rappresentano i numeri interi e a virgola mobile con i tipi equivalenti della piattaforma.

2. Catturare le computazioni: quando l'applicazione utente è in esecuzione, le sequenze di operazioni numeriche invocate dall'applicazione utente possono essere catturate, registrate e dinamicamente compilate in un programma oggetto dalla piattaforma RapidMind dal componente chiamato *Streaming Execution Manager* (un approccio che come vedremo sarà ripreso anche da Accelerator).

3. Esecuzione su stream: il runtime della piattaforma RapidMind è usato per gestire l'esecuzione parallela dei programmi oggetti sulla piattaforma hardware target, che può essere la GPU, il processore Cell, o una CPU multicore.

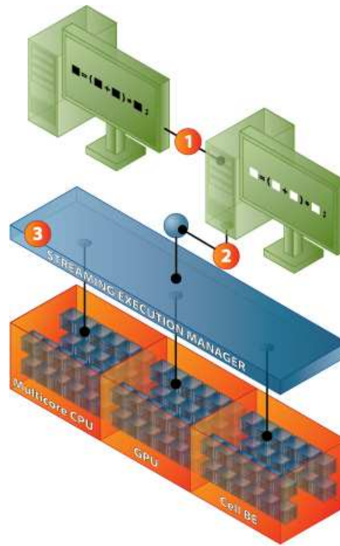


Figura 4.5, quali componenti della piattaforma sono responsabili per i 3 passi

Esempio

Step1 – Replace types

```
#include <cmath>

float f;
float a[512][512][3];
float b[512][512][3];

float func(
 float r, float s
) {
 return (r + s) * f;
}

void func_arrays() {
 for (int x = 0; x<512; x++) {
  for (int y = 0; y<512; y++) {
   for (int k = 0; k<3; k++) {
    a[y][x][k] =
      func(a[y][x][k],b[y][x][k]);
   }
  }
 }
}
```

```
#include <rapidmind/platform.hpp>

Value1f f;
Array<2,Value3f> a(512,512);
Array<2,Value3f> b(512,512);

Value3f func(
 Value3f r, Value3f s
) {
 return (r + s) * f;
}
```

Step 2 – Capture computation

```
#include <cmath>

float f;
float a[512][512][3];
float b[512][512][3];

float func(
 float r, float s
) {
 return (r + s) * f;
}

void func_arrays() {
 for (int x = 0; x<512; x++) {
  for (int y = 0; y<512; y++) {
   for (int k = 0; k<3; k++) {
    a[y][x][k] =
      func(a[y][x][k],b[y][x][k]);
   }
  }
 }
}
```

```
#include <rapidmind/platform.hpp>

Value1f f;
Array<2,Value3f> a(512,512);
Array<2,Value3f> b(512,512);

Value3f func(
 Value3f r, Value3f s
) {
 return (r + s) * f;
}

void func_arrays() {
 Program func_prog = BEGIN {
  In<Value3f> r, s;
  Out<Value3f> q;
  q = func(r,s);
 } END;
 . . .
}
```

Step3 - Parallel execution

```
#include <cmath>

float f;
float a[512][512][3];
float b[512][512][3];

float func(
 float r, float s
) {
 return (r + s) * f;
}

void func_arrays() {
 for (int x = 0; x<512; x++)
  for (int y = 0; y<512; y++) {
   for (int k = 0; k<3; k++) {
    a[y][x][k] =
      func(a[y][x][k],b[y][x][k]);
   }
  }
 }
}
```

```
#include <rapidmind/platform.hpp>

Value1f f;
Array<2,Value3f> a(512,512);
Array<2,Value3f> b(512,512);

Value3f func(
 Value3f r, Value3f s
) {
 return (r + s) * f;
}

void func_arrays() {
 Program func_prog = BEGIN {
  In<Value3f> r, s;
  Out<Value3f> q;
  q = func(r,s);
 } END;
 a = func_prog(a,b);
}
```

4.1.2 OpenCL

OpenCL[26] [42] è uno standard aperto di settore per la programmazione parallela general purpose di CPU, GPU e altri processori che fornisce agli sviluppatori un mezzo per realizzare software portabile e pieno accesso alla potenza di calcolo di queste piattaforme eterogenee. OpenCL supporta un ampio spettro di applicazioni, che vanno dal software embedded e consumer fino a soluzioni HPC, mediante un'astrazione portabile, di basso livello e ad alte prestazioni. Creando un'efficiente interfaccia di programmazione close-to-the-metal, OpenCL costituirà il livello fondazionale alla base di tutta una serie di tool platform-indipendent, middleware e applicazioni di calcolo parallelo. OpenCL è destinato soprattutto a giocare un ruolo sempre più significativo nelle emergenti applicazioni grafiche interattive che combinano algoritmi generali di calcolo parallelo con le pipeline di rendering grafico. Il framework OpenCL consiste di un'API per coordinare la computazione parallela tra un insieme eterogeneo di processori ed un linguaggio cross-platform con un ben specificato ambiente di calcolo.

Lo standard OpenCL:

- Supporta modelli di programmazione data-parallel and task-based parallel
- Utilizza un sottoinsieme dell'ISO C99 con estensioni per il parallelismo
- Adotta lo standard IEEE 754 per quel che riguarda il formato per la rappresentazione dei numeri in virgola mobile, il set di operazioni effettuabili su questi, e il metodo di arrotondamento usato; E' cosi garantita la consistenza matematica dei risultati su piattaforme differenti
- Definisce un profilo di configurazione per i dispositivi mobili ed embedded
- Interopera efficientemente con OpenGL, OpenGL ES e altre APIs grafiche

Il framework contiene le seguenti componenti:

OpenCL Platform layer: il platform layer dà la possibilità al programma host di conoscere quali dispositivi OpenCL (assieme alle loro capability) siano disponibili e di creare contesti.

OpenCL Runtime: il runtime permette ad un programma host di manipolare i contesti una volta che siano stati creati.

OpenCL Compiler: il compilatore OpenCL crea gli eseguibili dei programmi che contengono i kernel OpenCL. Il linguaggio di programmazione OpenCL C implementato dal compilatore supporta un sottoinsieme del linguaggio ISO C99 con estensioni per il parallelismo.

[26] Open Computing Language

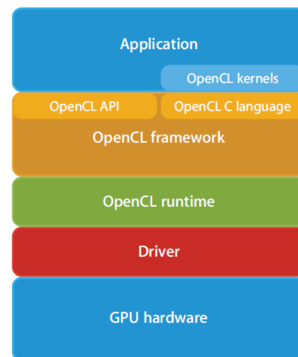


Figura 4.6, l'architettura OpenCL e la sua posizione rispetto al livello applicativo e a quelli sottostanti (driver/hardware)

Usando OpenCL i programmatori possono scrivere programmi general purpose che sono eseguiti sulle GPU senza il bisogno di mappare i loro algoritmi su una particolare API grafica 3D come OpenGL o DirectX. Il target cui si rivolge OpenCL sono programmatori esperti che vogliono scrivere codice portabile ma al tempo stesso efficiente. Questi includono gli sviluppatori di librerie, i fornitori di servizi middleware, e i programmatori di applicazioni orientate alle performance. Dunque OpenCL fornisce un'astrazione hardware di basso livello (molti dettagli dell'hardware sottostante sono esposti) assieme ad un framework per supportare la programmazione.

Per descrivere le idee chiave che stanno dietro ad OpenCL, useremo una gerarchia di modelli:

- Platform Model
- Execution Model
- Memory Model
- Programming Model

Platform Model

Il platform model per OpenCL è definitio in figura 4.7. Il modello consiste di un *host* connesso ad uno o più *dispositivi OpenCL* (detti compute device). Un dispositivo OpenCL è diviso in una più *compute unit* (CU) che sono ulteriormente suddivise in uno o più *processing element* (PE). Le computazioni su un dispositivo avvengono all'interno dei processing element. Un'applicazione OpenCL è eseguita su un host in accordo con i modelli nativi della piattaforma. L'applicazione OpenCL invia i *commandi* dall'host per eseguire computazioni sui processing element all'interno del device. I processing element dentro una compute unit eseguono in modo lockstep un singolo stream d'istruzioni come unità SIMD o come unità SPMD (ogni PE mantiene il proprio program counter).

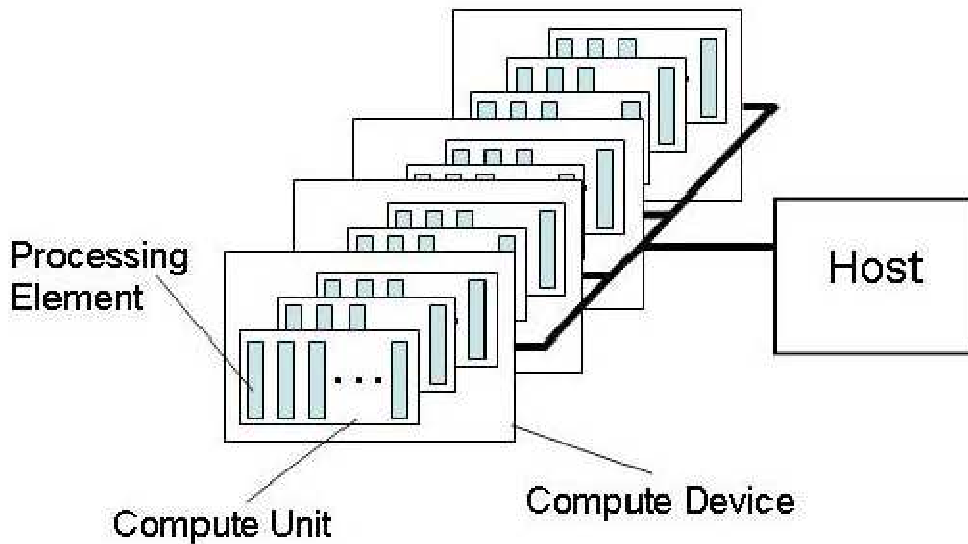


Figura 4.7: Platform model. un host più uno o più compute devices ognuno con uno o più compute units ognuno con uno o più processing elements.

Execution Model

L'esecuzione di un programma OpenCL consta di due parti: i *kernel* che sono eseguiti su uno o più dispositivi OpenCL e un *programma host* che è in esecuzione sull'host. Il programma host determina il contesto dei kernel e gestisce la loro esecuzione.

Il cuore del modello d'esecuzione di OpenCL è definito da come i kernel sono eseguiti. Quando un kernel è sottoposto all'esecuzione dall'host, è definito un *index space* (spazio degli indici). Un'istanza del kernel esegue per ogni punto contenuto in questo index space. Tale istanza di kernel è chiamata *work-item* ed è identificata dal proprio punto nell'index space, che rappresenta un ID globale per il work-item. Ogni work-item esegue lo stesso codice ma lo specifico flusso d'esecuzione all'interno del codice e i dati sui quali opera possono variare a seconda del work-item.

I work-item sono organizzati in *work-group*. I work-group rappresentano una decomposizione a grana più grossa dell'index space. Ai work-group sono assegnati degli ID univoci con la stessa dimensione di quella dell'index space usato per i work-item. Ai work-item è assegnato un unico ID locale relativo al proprio work-group cosicché un singolo work-item può essere identificato univocamente dal proprio ID globale o da una combinazione del proprio ID locale e ID del work-group. I work-item in un dato work-group sono eseguiti in modo concorrente all'interno dei processing element di una singola compute unit.

L'index space supportato in OpenCL 1.0 è chiamato NDRange. L'NDRange è uno spazio degli indici N-dimensionale, dove N può assumere i valori uno, due o tre. Un NDRange è definito da un array d'interi di lunghezza N specificando l'estensione dell'index space in ogni dimensione. Ogni ID globale o locale dei

work-item sono tuple N-dimensionali. Le componenti degli ID globali sono valori compresi nell'intervallo che va da zero sino a numero degli elementi in quella dimensione meno uno.

Ai work-group sono assegnati ID usando un approccio similare a quello poc'anzi descritto per gli ID globali dei work-item.

Un array di lunghezza N definise il numero dei work-group in ogni dimensione. I work-item sono assegnati ad un work-group e viene dato loro un ID locale avente le componenti comprese nell'intervallo che va da zero alla grandezza del work-group in quella dimensione meno uno. Quindi, la combinazione dell'ID di un work-group ID e dell'ID locale a quel work-group definiscono univocamente un work-item. Ogni worm-item è identificabile in due modi: grazie all'indice globale, e tramite l'indice di work-group assieme all'indice locale all'interno del work group.

Per esempio, si consideri lo spazio degli indici a 2-dimensioni in figura 4.8. Introduciamo l'index space per i work-item (Gx, Gy) e la dimensione di ogni work-group (Sx, Sy). Gli indici globali definiscono uno spazio degli indici (G_x, G_y) aventi numero totale degli elementi pari al prodotto di G_x e G_y. Gli indici locali definiscono uno spazio degli indici (S_x, S_y) in cui il numero dei work-item in un singolo work-group è il prodotto di S_x per S_y. Dalla dimensione di ogni work-group e il numero totale di work-item possiamo calcolare il numero dei work-group. Uno spazio degli indici 2-dimensionale è usato per identificare univocamente un work-group. Ogni work-item è identificato dal proprio ID glocale (g_x, g_y) o dalla combinazione dell'ID del work-group (w_x, w_y), la grandezza di ogni work-group (S_x, S_y) e l'ID locale (s_x, s_y) all'interno del particolare work-group in modo tale che:

$$(g_x, g_y) = (w_x \cdot s_x + s_x, w_y \cdot s_y + s_y)$$

Il numero dei work-group può essere calcolato come:

$$(w_x, w_y) = \left(\frac{g_x}{s_x}, \frac{g_y}{s_y}\right)$$

Dato un global ID e la grandezza del work-group size, l'ID del work-group per un work-item è calcolato come:

$$(w_x, w_y) = \left(\frac{g_x - s_x}{s_x}, \frac{g_y - s_y}{s_y}\right)$$

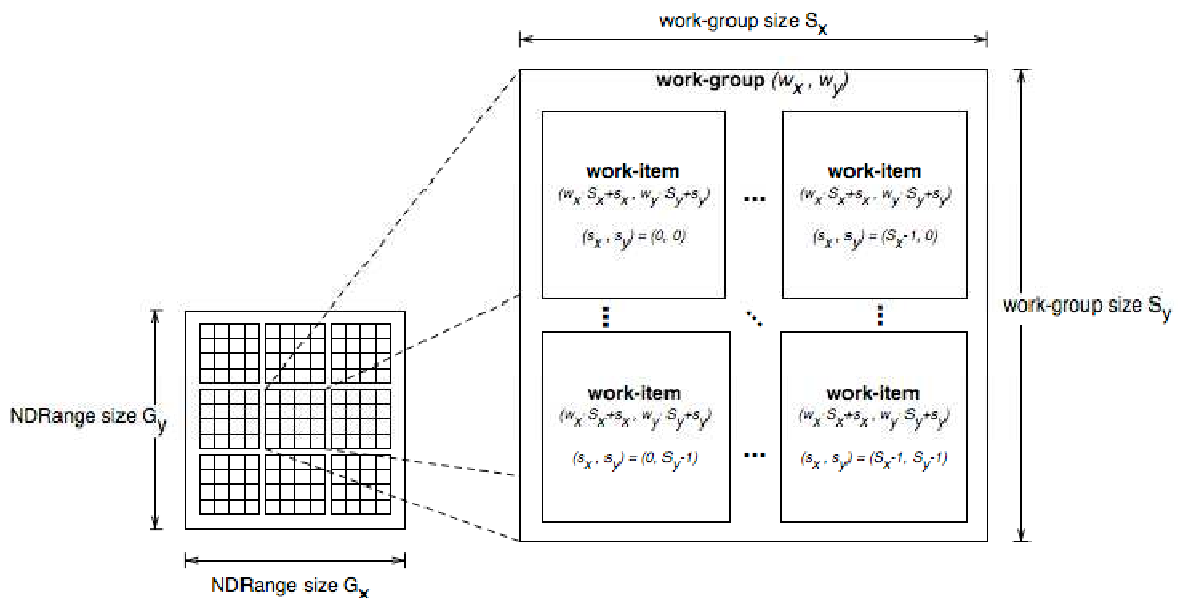


Figura 4.8, Un esempio di spazio degli indici NDRange in cui sono mostrati i work-items, i loro ID globali e il loro equivalente espresso come coppie di work-group e ID locale.

Una vasta gamma di modelli di programmazione può essere mappata su questo modello d'esecuzione. OpenCL supporta esplicitamente due di questi modelli: il modello di programmazione *data parallel* e il modello di programmazione *task parallel* (si veda a riguardo il paragrafo *Programming Model*).

Execution Model: Context e Command Queues

L'host definisce un contesto per l'esecuzione dei kernel. Il contesto include le seguenti risorse:

1. **Devices**: la collezione di dispositivi OpenCL usati dall'host.
2. **Kernels:** le funzioni OpenCL che sono eseguite sui dispositivi OpenCL.
3. **Program Objects:** il codice sorgente del programma e l'eseguibile che implementano i kernel.
4. **Memory Objects:** un insieme di oggetti rappresentanti aree di memoria visibili sia all'host sia ai dispositivi OpenCL.

I memory object contengono i valori sui quali le istanze di un kernel possono operare. Il contesto è creato e manipolato dall'host usando funzioni dell'API OpenCL. L'host crea una struttura dati chiamata *command-queue* per cooordinare l'esecuzione dei kernel sui dispositivi. L'host inserisce i comandi nella command-queue che sono poi eseguiti sui dispositivi all'interno di un contesto.

Questi includono:

- **Kernel execution commands**: eseguono un kernel sui processing element di un dispositivo.
- **Memory commands**: trasferiscono dati a, da, o tra memory object, o mappano e unmappano i memory object dallo spazio d'indirizzamento dell'host.
- **Synchronization commands**: vincolano l'ordine d'esecuzione dei comandi.

La command-queue schedula i comandi per l'esecuzione su un dispositivo. Questi sono eseguiti in modo asincrono tra l'host e il dispositivo. L'ordine relativo con il quale due o più comandi sono eseguiti può essere uno dei seguenti due modi:

- **In-order Execution**: i comandi sono avviati seguendo l'ordine con il quale appaiono nella coda comandi e sono completati in ordine. In altre parole, un comando precedente nella coda deve essere completato prima che quello seguente abbia inizio. Questo serializza l'ordine d'esecuzione dei comandi in una coda.
- **Out-of-order Execution**: i comandi sono emessi in ordine, ma non si aspetta la terminazione di un comando prima di iniziare l'esecuzione del successivo. Ogni vincolo d'ordinamento è imposto dal programmatore tramite espliciti comandi di sincronizzazione.

I comandi riguardanti l'esecuzione dei kernel e quelli inerenti la memoria immessi in una coda generano degli oggetti *evento*. Questi sono usati per controllare l'esecuzione tra i comandi e per coordinare l'esecuzione tra l'host e i dispositivi.

E' possibile associare a un singolo contesto molteplici code. I comandi di queste code sono eseguiti concorrentemente e in modo indipendente senza alcun meccanismo esplicito all'interno di OpenCL per sincronizzarli tra di loro.

Execution Model: le categorie dei kernel

Il modello d'esecuzione di OpenCL supporta due categorie di kernel:

- **I kernel OpenCL** sono scritti con il linguaggio di programmazione OpenCL C e compilati con il compilatore OpenCL. Tutte le implementazioni di OpenCL supportano questo tipo di kernel. Le implementazioni possono fornire altri meccanismi per creare kernel OpenCL.
- **I kernel nativi** sono acceduti tramite puntatori a funzioni host. I kernel nativi sono accodati per l'esecuzione assieme ai kernel OpenCL su un dispositivo e condividono i memory object con i kernel OpenCL. Per esempio, questi kernel nativi possono essere funzioni definite nel codice applicativo o esportate da una libreria. Va rilevato che la capacità di eseguire kernel nativi è una funzionalità opzionale all'interno di OpenCL e la semantica dei kernel nativi è dipendente dalla specifica implementazione. L'API OpenCL include funzioni per richiedere le caratteristiche di un dispositivo e determinare se questa capacità è supportata.

Memory Model

I work-item che eseguono un kernel hanno accesso a quattro distinte regioni di memoria:

- **Global Memory**. Questa regione di memoria permette accessi in lettura/scrittura a tutti i work-item in tutti i work-group. I work-item possono leggere da o scrivere in ogni elemento di un memory object. Le letture e le scritture nella memoria globale possono essere cachate secondo le capacità del dispositivo.
- **Constant Memory**: una regione della memoria globale che rimane costante durante l'esecuzione di un kernel. L'host alloca e inizializza i memory object immessi nelle memorie costanti.
- **Local Memory**: una regione di memoria locale a un work-group. Questa regione può essere usata per allocare variabili che sono condivise da tutti i work-item in quel work-group. Può essere realizzata usando regioni dedicate della memoria di un dispositivo OpenCL. In alternativa, la regione di memoria locale può essere mappata su sezioni della memoria globale.
- **Private Memory**: una regione di memoria privata di un work-item. Variabili definite nella memoria privata di un work-item non sono visibili a un altro work-item.

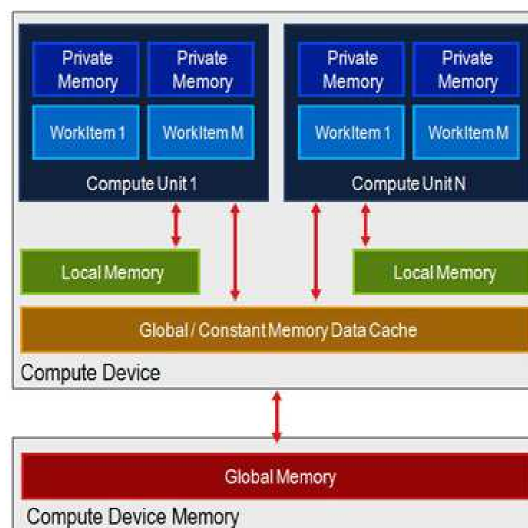


Figure 4.9, il modello di memoria di OpenCL

La tabella 3.1 descrive se il kernel o l'host possono allocare da una regione di memoria, il tipo di allocazione (statica, cioè a tempo di compilazione oppure dinamica, cioè a tempo d'esecuzione) e quali tipi di accessi sono permessi, cioè se il kernel o l'host possono leggere e/o scrivere nella regione di memoria.

	Global	Constant	Local	Private
Host	Allocazione dinamica Accessi lettura/scrittura	Allocazione dinamica Accessi lettura/scrittura	Allocazione dinamica Nessun accesso	Nessuna allocazione Nessun accesso
Kernel	Nessuna allocazione Accessi lettura/scrittura	Allocazione statica Accesso in sola lettura	Allocazione statica Accesso in sola lettura	Allocazione statica Accesso in sola lettura

Tabella 4.1 Regioni di memoria - Allocation and Memory Access Capabilities

Le regioni di memoria e come queste si relazionano con il platform model è descritto in figura 4.10.

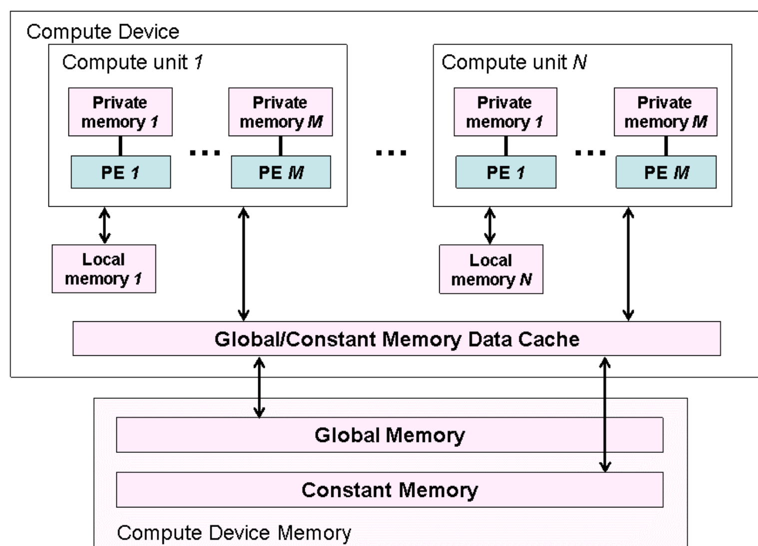


Figure 4.10: architettura concettuale di un dispositivo OpenCL con i processing element (PE), compute unit e compute device. L'host non è mostrato.

L'applicazione che è in esecuzione sull'host usa l'API OpenCL per creare i memory object nella memoria globale, e per accodare i memory command che operano su questi memory object.

L'host e i modelli di memoria di un dispositivo OpenCL sono, per lo più, indipendenti l'uno dell'altro. Questo è una necessità, poiché l'host è definito al di fuori di OpenCL. Essi, tuttavia, a volte necessitano di interagire. Questa interazione avviene in uno dei due modi seguenti: copiando esplicitamente dati o

mappando regioni di un memory object. Per copiare i dati esplicitamente, l'host accoda comandi per trasferire dati tra i memory object la memoria dell'host. Questi comandi per il trasferimento di memoria possono essere bloccanti o non-bloccanti.

La chiamata di una funzione OpenCL per il trasferimento di memoria bloccante ritorna una volta che le risorse di memoria associate sull'host possono essere riusate in modo sicuro. Per un trasferimento di memoria non-bloccante, la chiamata di funzione OpenCL ritorna non appena il comando è accodato indipendentemente dal fatto che la memoria host sia usabile in tutta sicurezza.

Il metodo d'interazione tramite mapping/unmapping tra l'host e i memory object OpenCL permette all'host di mappare una regione facente parte del memory object nel proprio spazio d'indirizzamento. Il comando di memory map può essere bloccante o non-bloccante. Una volta che la regione di un memory object è stata mappata, l'host può leggere o scrivere in questa regione. L'host compie l'unmapping della regione quando gli accessi (in lettura e/o scrittura) verso questa regione mappata dall'host sono completati.

Memory Consistency

OpenCL usa un modello rilassato di consistenza della memoria; vale a dire che lo stato della memoria visibile in un dato istante a un work-item non è garantito che sia consistente con quello di tutti gli altri work-item. All'interno di un work-item la memoria ha consistenza a livello di load/store. La memoria locale è consistente tra tutti i work-item in un singolo work-group dopo una work-group barrier. La memoria globale è consistente tra i work-item in un singolo work-group dopo una work-group barrier, ma non ci sono garanzie sulla consistenza di memoria tra differenti work-group che stanno eseguendo un kernel. La consistenza della memoria per i memory object condivisi tra i comandi accodati è forzata ad un punto di sincronizzazione.

Programming Model

Il modello d'esecuzione di OpenCL supporta i modelli di programmazione *data parallel* e *task parallel*, cosi come supporta soluzioni ibride di questi due modelli. Il modello primario che guida il design di OpenCL è quello data parallel.

Data Parallel Programming Model

Il modello di programmazione data parallel definisce una computazione in termini di una sequenza di istruzioni applicate ad elementi multipli di un memory object. L'index space associato con il modello d'esecuzione OpenCL definisce i work-item e come i dati mappano sui work-item. In un modello rigorosamente data parallel, c'è un mapping uno-a-uno tra i work-item e l'elemento di un memory object sul quale il kernel può essere eseguito in parallelo. OpenCL implementa una versione rilassata del modello di programmazione data parallel nella quale questo mapping non è un requisito necessario. OpenCL espone un modello di programmazione data parallel gerarchico. Ci sono due modi di specificare

la suddivisione gerarchica. Nel modello esplicito il programmatore definisce il numero totale di work-item da eseguire in parallelo e anche come i work-item sono divisi tra i work-group. Nel modello implicito, il programmatore specifica solamente il numero complessivo di work-item da eseguire in parallelo, e la divisione in work-group è gestita dall'implementazione di OpenCL.

Task Parallel Programming Model

Il modello di programmazione task parallel definisce un modello nel quale una singola istanza di un kernel è eseguita indipendentemente da ogni index space. Questo è logicamente equivalente a eseguire un kernel su una compute unit con un work-group contenente un singolo work-item. In base a questo modello, gli utenti esprimono il parallelismo:

- Usando tipi di dato vettoriali implementati dal dispositivo,
- Accodando task multipli, e/o
- Accodando kernel nativi sviluppati usando un modello di programmazione ortogonale a OpenCL.

Sincronizzazione

Ci sono due domini di sincronizzazione in OpenCL:

- I work-item in un singolo work-group
- I comandi accodati nella/e command-queue in un singolo contesto

La sincronizzazione tra work-item in un singolo work-group è realizzata usando una work-group barrier. Tutti i work-item di un work-group devono eseguire la barrier prima che sia permesso a qualcuno di loro di continuare l'esecuzione al di là della barrier. Si noti che la work-group barrier deve essere incontrata da tutti i work-item di un work-group che eseguono il kernel o da nessuno in assoluto. Non c'è alcun meccanismo di sincronizzazione tra work-group.

I punti di sincronizzazione tra i comandi nelle command-queue sono:

- *Command-queue barrier*. La command-queue barrier assicura che tutti i comandi accodati in precedenza abbiano finito l'esecuzione e ogni aggiornamento risultante ai memory object sia visibile ai successivi comandi accodati prima che cominci la loro esecuzione. Questa barrier può essere solamente usata per sincronizzare i comandi in una singola command-queue.
- Attendere su un *evento*. Tutte le funzioni dell'API OpenCL che accodano comandi restituiscono un evento che identifica il comando e i memory object aggiornati da questo. E' garantito a un comando successivo che sta attendendo tal evento che tutti gli aggiornamenti a questi memory object siano visibili prima che l'esecuzione del comando possa avere inizio.

4.1.3 Accelerator

A differenza delle API precedentemente presentate, questo sistema, sviluppato da Microsoft Research, cerca di colmare il gap con le virtual machine. I programmatori utilizzano un linguaggio convenzionale ad alto livello (come C#) e una libreria che fornisce solamente operazioni data-parallel di alto livello. Questo rappresenta un vantaggio poiché essendo costruita al di sopra di .NET è automaticamente accessibile a tutti i linguaggi compatibili con il framework. La libreria compila al volo queste operazioni generando codice pixel shader. Il limite di questa soluzione è dato appunto dal forte accoppiamento con una particolare API grafica, che lo rende un lavoro "datato": come si è discusso in precedenza (capitolo 3) sono state sviluppate dai fornitori di schede video librerie espressamente orientate all'HPC; generando invece direttamente un pixel shader è probabile che non tutte le capacità offerte da una moderna scheda video possano essere sfruttate. Una seconda limitazione è il legame con l'API grafica di Microsoft che ne limita l'utilizzo in ambienti Linux (a meno di non usare wrapper per le DirectX come WineX). PBricks pur essendo sviluppato come strato al di sopra del CLR, può essere utilizzato su ambiente Linux sul quale sia stata installata una macchina virtuale Mono [45].

Per la versione sequenziale è necessario riconvertire gli algoritmi preesistenti usando i tipi di dato (più stringenti rispetto a quelli che vedremo saranno usati in PBricks) e le operazioni definite su questi. Un altro parametro con cui valutare l'efficacia di una soluzione è dato dalla facilità di debugging. Nel caso di Accelerator, anche laddove il programma si debba sviluppare da zero non è possibile effettuare introspezione sullo stato della GPU. PBRicks invece offre un costruttore di computazione sequenziale (`SeqPBrick`) che permette il debugging con tool e strumenti standard .NET (si veda a riguardo il capitolo 6).

Il modello di programmazione prevede la scrittura di un metodo e l'utilizzo al suo interno di tipi concreti derivati da un tipo di dato astratto (`ParallelArray`). Si trovano cosi `IntParallelArray`, `FloatParallelArray`, `BoolParallelArray`, `Float4ParallelArray`. Non c'è supporto per il tipo di dato double poiché all'epoca non era supportato dalle schede video. (In PBricks questo è previsto)

Gli array data parallel forniscono solo funzioni di aggregazione definite su tutto l'insieme dei dati. Tali operazioni sono un sottoinsieme di quelle che si trovano in linguaggi come APL [44] ed includono: operazioni aritmetiche e booleane element-wise (che restituiscono un intero array, MAP), riduzioni (max, min, somma, prodotto, …), trasformazioni (expand, pad, shift, rotate, transpose, …), Basic linear algebra. Non sono invece supportati gli accessi ai singoli elementi (cosa possibile in PBricks, si veda a riguardo la descrizione degli `Scatter`/`GatherStream` nel capitolo 6), l'aliasing, l'aritmetica dei puntatori.

E' stato scelto un approccio di tipo funzionale alla programmazione: ogni operazione crea un nuovo data parallel array. Per questioni di efficienza la libreria non effettua immediatamente le operazioni data-parallel, costruendo invece un grafo della computazione e compilando on-demand nel particolare codice pixel shader e chiamate all'API (DirectX) per il setup e avvio della computazione. E' stato cioè adottato un approccio di tipo *lazy*: la conversione esplicita da data parallel array a normali array .NET avvia il processo di compilazione ed esecuzione. L'uso lazy permette alcune ottimizzazioni: dato il grafo esprimente la computazione si possono semplificare alcune sue parti, ad esempio rimuovendo le

sottoespressioni comuni e calcolandole una sola volta, oppure evitando copie e allocazioni di strutture dati temporanee.

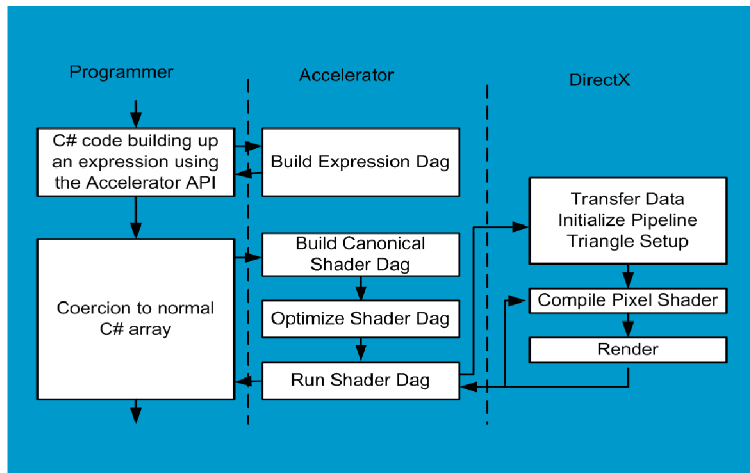


Figura 4.11, schema che riassume l’esecuzione di un programma Accelerator

Esempio: convoluzione 2D

```
using Microsoft.Research.DataParallelArrays;

static float[,] Blur(float[,] array, float[] kernel)
{
    float[,] result;

    // Convert C#-array to data-parallel array.
    DFPA parallelArray = new DFPA(array);

    // Compute blur by shifting the entire original image by
    // "i" pixels and multiplying with the appropriate weight.
    FPA resultX = new FPA(0f, parallelArray.Shape);
    for (int i = 0; i < kernel.Length; i++)
    {
        int[] shiftDir = new int[] {0, i};

        // Operator overloading.
        resultX += PA.Shift(parallelArray, shiftDir) * kernel[i];
    }

    FPA resultY = new FPA(0f, parallelArray.Shape);
    for (int i = 0; i < kernel.Length; i++)
    {
        int[] shiftDir = new int[] { i, 0 };
        resultY += PA.Shift(resultX, shiftDir) * kernel[i];
    }

    // Convert result back to C#-array.
    PA.ToArray(resultY, out result);

    parallelArray.Dispose();
    return result;
}
```

Parte II - Strumenti

Capitolo 5

Nella prima parte di questo capitolo si introduce il concetto di meta-programmazione e come questa sia resa possibile tramite i metadati e il sistema di introspezione (reflection) del framework .NET. Si parla poi di ambienti d'esecuzione virtuale soffermandosi sul CLR, la macchina virtuale di Microsoft. Se ne descrive la struttura, il modello d'esecuzione e il linguaggio intermedio MSIL. Questa spiegazione è necessaria poiché PBricks analizza codice intermedio (MSIL) .NET.
Segue una breve digressione sulla differenza tra codice a registri e codice a stack, poiché CAL IL e PTX sono linguaggi basati su macchina virtuale a registri, spiegandone le differenze in termini di prestazioni.
E' necessario inoltre comprenderne il modello di memoria, dato che per l'implementazione di PBricks bisogna gestire lo spostamento di dati con aree non gestite della memoria (aree unmanaged); bisogna aver quindi ben chiare le problematiche coinvolte e come il CLR le risolva mediante il servizio di P/Invoke e il meccanismo di pinning della memoria.
Bisogna inoltre descrivere dove e come il codice intermedio è salvato (assembly).

5.1 Metaprogrammazione

Nei moderni sistemi informatici vi è una crescente domanda di applicazioni in grado di adattarsi dinamicamente alle caratteristiche e ai cambiamenti dello specifico ambiente di utilizzo. Si rende necessario, quindi, avere la possibilità di modificare i diversi aspetti dei componenti, di configurare e assemblare componenti mediante processi automatici. Per ottenere questo occorrono tecnologie per pubblicare le funzionalità e per manipolare i parametri (e le implementazioni) dei componenti. Questo è quanto consente la metaprogrammazione.

Un meta-programma è un programma che analizza e manipola altri programmi, compreso se stesso. I programmi generati (elaborati) sono detti *programmi oggetto*. I compilatori e pre-processori sono esempi di meta-programmi che operano su altri programmi. Quando, invece, un meta-programma opera su se stesso si parla di *reflection* (vedere paragrafo 5.2.2).

Esistono diversi modi di classificare i sistemi di meta-programmazione. Un primo modo si basa sul *quando* il meta-programma viene eseguito: *statico,* prima dell'esecuzione, *runtime* altrimenti. Un secondo modo si basa, invece, sul *chi* stabilisce la distinzione tra programma e meta-programma: *manualmente*, l'utente annota il codice per distinguere tra i due, *automaticamente* il sistema individua implicitamente la parte di codice da eseguire. Questi sistemi si possono anche distinguere in funzione del fatto che il meta-linguaggio coincida con il programma oggetto *(sistema omogeneo)* o meno *(sistema eterogeneo)*. Esistono molti esempi di linguaggi che supportano sistemi di meta-programmazione con potenzialità molto diverse, ad esempio il C supporta la metaprogrammazione statica mediante le *macro testuali*:

```
#define MAX(a,b) ((a>=b) ? a : b)

void main()
{
	int intValue1 = ..., intValue2 = ...;
```

```
        float floatValue1 = ..., floatValue2 = ...;

        int maxI = MAX(intValue1, intValue2);
        float maxF = MAX(floatValue1, floatValue2);
}
```

Una macro testuale viene espansa al fine di modificare il testo di un programma: a fronte della semplicità del sistema, vi sono diversi svantaggi. Prima di tutto le sostituzioni effettuate nel codice sono puramente testuali senza controllo sulla correttezza di quanto inserito. Inoltre non si ha la possibilità di fornire informazioni recuperabili a tempo d'esecuzione: l'espansione della macro si esaurisce in compilazione.

Il C++ supporta la metaprogrammazione attraverso il meccanismo dei *template* e quello del *constant folding.* Di seguito è riportato un esempio d'uso del template metaprogramming per calcolare il fattoriale:

```
template <int N>
struct Factorial
{
    enum { value = N * Factorial<N - 1>::value };
};

template <>
struct Factorial<0>
{
    enum { value = 1 };
};

void main()
{
    int x = Factorial<0>::value; // == 1
    int y = Factorial<4>::value; // == 24
}
```

Quando viene istanziato il template, il compilatore genera la versione della classe `Factorial` appropriata, nell'esempio con N = 0 e poi N = 4. Essendo `value` il valore di un'enumerazione viene applicato ad esso il costant folding, ossia il valore simbolico è sostituito con il valore attuale: nel primo caso il valore sostituito è trovato direttamente nella struct che lo definisce, `Factorial<0>`, nel secondo caso, la definizione di `value` è data ricorsivamente in termini di un'altra struct; il processo ha termine quando si raggiunge il caso base, ossia si utilizza `value` del tipo `Factorial<0>`. I valori intermedi via via calcolati sono utilizzati nelle espressioni valutate in precedenza, fino a fornire il valore finale dell'espressione costante `Factorial<4>::value`. In questo modo è possibile calcolare il fattoriale a tempo di compilazione. Il C++ viene considerato un linguaggio a 2 livelli [1] in quanto un programma può contenere sia codice statico, valutato a tempo di compilazione, sia dinamico, eseguito a runtime. I template sono la parte del codice C++ eseguita in compilazione: per questo il sistema di meta-programmazione è statico. Inoltre è automatico in quanto per identificare il meta-programma il programmatore non inserisce nel codice alcuna annotazione specifica come ad esempio degli attributi (custom attribute) del framework .NET. Infine è eterogenea poiché il meta-programma esprime in maniera diversa costrutti presenti nel programma oggetto. In C++ un meta-programma è in grado di

generare codice sfruttando la capacità del compilatore di inserire metodi *inline*: essendo questo processo ricorsivo il codice può essere accumulato. Il meta-programma è in grado di ispezionare il codice mediante le meta-funzioni che hanno acceso alle informazioni usate dal compilatore.

5.2 Metadata e Reflection

Introdotto il concetto di meta-programmazione, è importante per la compresione della presente tesi, esaminare con maggiore dettaglio due concetti alla base della meta-programmazione in .NET: metadata e reflection. Per *reflective system* si intende un sistema in grado di modificare il proprio stato di esecuzione e la propria rappresentazione. Per far questo deve avere un modello di se stesso e del proprio ambiente d'esecuzione.

5.2.1 Metadata

I metadata forniscono il modello del sistema ovvero le informazioni necessarie a descrivere un programma; sono memorizzati in formato binario, neutrale rispetto al linguaggio, sia in memoria sia in un file eseguibile. I metadata vengono supportati direttamente in maniera standard sin dalla prima versione del CLR. Indipendentemente dal linguaggio utilizzato con la compilazione viene prodotto un *managed module* (vedi paragrafo 5.3.1): un file eseguibile standard, PE[27] che richiede il CLR per poter essere eseguito. Il managed module è formato da 4 parti [2]:

- PE header simile al COFF[28], indica il tipo del file;
- CLR header, contiene le informazioni che caratterizzano un modulo come managed module;
- Metadata, insieme di tabelle tra cui quelle che descrivono i tipi (e relativi membri) definiti e riferiti nel codice utente (in linguaggio intermedio IL),
- Codice IL prodotto a tempo di compilazione che il runtime compilerà in istruzioni specifiche (native) della CPU

Ogni tipo di dato, con i relativi membri, definito e/o riferito in un modulo o assembly viene descritto in un metadata. In fase di esecuzionie il runtime carica i metadata in memoria per ottenere informazioni quali:

- descrizione di un assembly:
    - identità
    - tipi esportati
    - gli assembly da cui dipende quello considerato
    - informazioni di sicurezza, permessi di esecuzione
- descrizione dei tipi di dato

[27] Portable Executable

[28] Common Object File Format

- nome, scope, classe base e interfacce implementate
- membri (metodi, campi, proprietà, eventi e tipi annidati)

- attributi (elementi descrittivi aggiuntivi)

Per esempio i metada relativi ai metodi di una classe sono memorizzati in una tabella MethodDef [5] che riporta le suguenti informazioni:

- *RVA, Relative Virtual Address*, utilizzato dal runtime per calcolare l'indirizzo di memoria iniziale del codice MSIL del metodo;
- *ImplFlags* e *Flags* contengono maschere di bit che descrivono il metodo (ad esempio indicano se si tratta di un costruttore);
- *Name* è il nome del metodo;
- *Signature* indice nel blob heap della definizione della signature del metodo;

Row	RVA	ImpFlags	Flags	Name	Signature
1	0x00002050	IL Managed	Public Reuse Slot SpecialName RTSSpecialName .ctor	.ctor (constructor)	
2	0x0000208c	IL Managed	Public Reuse Slot	generate	object

Tabella 5.1, esempio della tabella MethodDef che fornisce informazioni sul *costruttore* e sul metodo *generate* di una classe

Per ulteriori dettagli sulle tabelle dei metadati e sul formato del PE si rimanda a [6,7,5,2]. I linguaggi supportati da .NET utilizzano metadati per descrivere se stessi in maniera automatica e trasparente allo sviluppatore.

I vantaggi introdotti dai metadati sono:

- i moduli e gli assembly sono *self-describing* ovvero memorizzano tutte le informazioni necessarie per interagire con altri moduli; ogni compilatore CLR produce allo stesso tempo codice IL e metadata che inserisce nello stesso file, rendendo impossibile una mancanza di sincronizzazione tra i due. Con i metadata si ha, quindi, un modello di programmazione più semplice perché consentono di eliminare l'utilizzo di file IDL[29] e file header per riferire componenti esterni;
- *interoperabilità* tra i linguaggi e *semplificazione* della progettazione basata sui componenti in quanto mediante le informazioni nei metadata in un programma si può estendere una classe presente in un altro modulo (riferito) e scritta in un altro linguaggio;

[29] Interface Definition Language

- maggior controllo sul comportamento a runtime dei programmi, mediante speciali metadata, gli attributi;

I metadata vengono, ad esempio, impiegati dal Garbage Collector per determinare quali oggetti sono raggiungibili e quali possono essere riferiti da quale oggetto; dal processo di verifica del CLR per garantire che il codice utente esegua solo operazioni sicure.

5.2.2 Reflection

Per *reflection* si intende la capacità di un programma di accedere al proprio stato interno d'esecuzione e possibilmente di modificarlo: è quindi un concetto base dei sistemi auto-adattivi. Due sono gli aspetti fondamentali legati alla reflection:

- *introspezione*, il sistema osserva e decide in base al proprio stato;
- *intercessione*, il sistema modifica la propria esecuzione o altera la propria interpretazione.

Il CLR supporta entrambi gli aspetti dando al programmatore la possibilità di individuare i membri, invocare metodi, accedere agli attributi dei tipi a tempo d'esecuzione, di creare istanze di classi il cui nome non è noto fino a runtime. In un reflective system quando il codice sorgente di un programma viene compilato le informazioni relative alla sua struttura vengono preservate sotto forma di metadata nel codice prodotto di più basso livello. Come visto nel paragrafo 5.2.1 mediante i metadata, a runtime, si può modificare la sequenza delle operazioni da eseguire. Il CLR espone il suo reflective system attraverso le classi contenute nel namespace `System.Reflection`. Tutti i tipi sono self-describing e le loro definizioni possono essere accedute tramite il metodo `GetType` ereditato da `System.Object`. Il CLR consente anche la scrittura di definizioni di tipo (per informazioni a riguardo si consulti `System.Reflection.Emit` su [5]). Il loader del CLR gestisce gli application domain (`AppDomain`), in particolare il caricamento di ogni assembly nell'application domain corretto, e provvede al controllo del layout di memoria della gerarchia dei tipi in ogni assembly. A titolo d'esempio se si vogliono ottenere a tempo d'esecuzione i metodi definiti in una classe (`MyType`) basta invocare il metodo `GetMethods` come segue:

```
MethodInfo[] mi = typeof(MyType).GetMethods(
                                     BindingFlags.Public |
                                     BindingFlags.DeclaredOnly |
                                     BindingFlags.Instance);
```

come parametro viene richiesta una bitmask che permette di definire come filtrare i metodi trovati.

5.3 Strongly typed execution environments

Negli ultimi anni il numero di linguaggi basati su macchine virtuali è incrementato significativamente. Ci sono diverse motivazioni a supporto di quest'andamento: l'hardware sta diventando sempre più veloce e possiamo pagare alcuni overhead per ottenere più riusabilità, sicurezza e robustezza dai nostri

programmi; la programmazione è diventata un compito difficile che richiede un numero sempre maggiore di servizi.
La garbage collection e librerie che forniscono funzionalità built-in sono spesso considerate un prerequisito per un linguaggio di programmazione. I linguaggi di programmazione basati su macchina virtuale permettono ai programmi di essere eseguiti su diverse piattaforme al solo costo di effettuare il porting dell'ambiente d'esecuzione piuttosto che dover ricompilare l'intero programma. Le macchine virtuali offrono inoltre la possibilità di ottenere maggiore sicurezza: l'execution engine media tutti gli accessi alle risorse effettuati dai programmi verificando che il sistema non possa essere compromesso.
Java è un linguaggio di programmazione di successo basato su macchina virtuale che è stato considerato più vicino a linguaggi compilati come il C++ piuttosto che a linguaggi interpretati come il Perl. Nel passato anche altri linguaggi con la stessa architettura, essenzialmente p-code, sono stati proposti [8] ma Java è stato il primo ad aver avuto un enorme impatto sulla programmazione mainstream. Anche Microsoft dal 2002 ha intrapreso la strada dei linguaggi basati su macchina virtuale basandoli sul Common Language Infrastructure (CLI) standardizzato dall'ECMA [6] e ISO [10]. Il core del CLI è un sistema d'esecuzione virtuale (*virtual execution system*) meglio noto come Common Language Runtime (CLR). Sia la JVM [11] sia il CLR [6] implementano una macchina virtuale multi-threaded stack-based, che offre molti servizi come ad esempio il caricamento dinamico, la garbage collection, la compilazione Just In Time (JIT).
Quando la macchina virtuale è basata su stack le operazioni leggono i valori dallo stack degli operandi e pushano i risultati sullo stack. Moltri altri linguaggi adottano una virtual machine a stack: OCaml [12], Python [13], TEA [14], XSLTVM [15], sono solo alcuni esempi.
Un'alternativa alle macchine virtuali basate su stack sono le macchine virtuali a registri. Queste macchine offrono l'astrazione dei registri invece che quella dello stack degli operandi per passare i valori alle istruzioni; un esempio di macchina virtuale a registri è Parrot [16], progettata e sviluppata per fare da host a numerosi linguaggi dinamici come Tcl, Javascript, Ruby, Lua, Scheme, PHP, Python, Perl 6, APL. Benchè entrambi i modelli siano Turing equivalenti c'è sempre stato un fervente dibattito tra gli sviluppatori su quale soluzione offra le migliori prestazioni. In [17], per esempio, è proposto un interprete alternativo per Python che è basato su registri.

Basandoci su quanto detto in [18] possiamo valutare il costo d'esecuzione di un'istruzione VM in un interprete come la somma di tre componenti:

- Dispatching dell'istruzione
- Caricamento degli operandi
- Esecuzione dell'operazione

Il dispatch dell'istruzione richiede il fetch (ossia il caricamento) della successiva istruzione VM dalla memoria, e il salto al corrispondente segmento dell'interprete che implementa l'istruzione VM. Una certa computazione può spesso essere espressa usando meno istruzioni su una macchina a registri rispetto alla sua controparte sulla macchina a stack.

Per esempio, l'assegnamento a variabile locale `a = b + c` può essere tradotto in uno pseudocodice IL a stack:

```
load c,
load b
add
store a
```

In una macchina virtuale a registri, lo stesso codice si sarebbe potuto esprimere solamente con la singola istruzione:

```
add a, b, c
```

Quindi, le macchine virtuali a registri, hanno il potenziale per ridurre significativamente il numero di *instruction dispatch*, che sono realizzati nella maggior parte dei compilatori (anche se esistono tecniche meno costose) con con dei grandi statement switch, tradotti con salti incondizionati difficilmente predicibili.

La locazione degli operandi deve essere esplicitata nel codice a registri, mentre in quello a stack gli operandi sono posizionati in relazione allo stack pointer. Questo ha come conseguenza che l'istruzione a registri è generalmente più lunga di quella a stack e richiede più fetch in memoria per essere eseguita. La ridotta dimensione del codice e il ridotto numero di fetch sono il motivo principale del perché le architetture a stack sono cosi popolari nelle VM.

Il costo d'esecuzione è indipendente rispetto al tipo di linguaggio intermedio, ma le ottimizzazioni (come la rimozione d'invarianti ed espressioni comuni) sono più facilmente realizzabili con un linguaggio a registri. Si veda il capitolo 11 (Sviluppi Futuri) per una serie di ottimizzazioni legate alla compilazione, dove il codice target è a registri.
Il CLR effettua compilazione Just in Time (paragrafo 5.3.1) (non è quindi un interprete), per cui questa distinzione non è poi cosi significativa, ma il codice intermedio target delle GPU di ATI e Nvidia è a registri, da cui deriva la possibilità e (necessità nel caso si vogliano migliorare le performance) di ottimizzare l'uso dei registri ad esempio con tecniche di analisi dataflow [19].

In questa tesi concentriamo la nostra attenzione su macchine virtuali basate su stack. In particolare siamo interessati alle macchine virtuali type oriented. La JVM e il CLR sono esempi di queste macchine laddove CVM, la macchina virtuale di OCaml, non lo è. CVM non offre la possibilità di definire tipi: espone solamente semplici operazioni e gli unici tipi al di là delle stringhe e dei tipi numerici sono le chiusure e i word blocks. Il nostro interesse ricade in quegli ambienti d'esecuzione che contengono informazioni riguardanti i tipi e la loro struttura. In particolare è necessario che l'ambiente sia in grado di riflettere i tipi e i loro metodi a tempo d'esecuzione. La JVM e il CLR sono buoni esempi di questi ambienti.
Uno *Strongly Typed Execution Environment* (STEE, ambiente d'esecuzione fortemente tipato) è un ambiente d'esecuzione che implementa una macchina virtuale avente un type-system estensibile, capacità di reflection e un modello d'esecuzione che garantisca che i tipi dei valori possano sempre essere stabiliti e i valori sempre acceduti solamente usando gli operatori definiti su di essi.

5.3.1 CLI, Common Language Infrastructure

Il CLI fornisce una specifica [6] per il codice eseguibile e per il suo ambiente di esecuzione VES[30]. Il VES da supporto diretto all'insieme dei tipi di dati *built-in*, definisce una macchina astratta con associato un modello e uno stato un insieme di costrutti per il controllo del flusso di esecuzione e un modello di gestione delle eccezioni.

CLR

Un'implementazione del VES è il Common Language Runtime. Il CLR è un supporto multi-piattaforma e multi-linguaggio. Per quanto riguarda i linguaggi di programmazione supportati troviamo: C#, F#, Managed C++, VB.NET, Jscript, J#, Fortran, Perl, Python, Scheme e altri. Per ognuno di essi si ha un apposito compilatore in grado di sfruttare le funzionalità fornire dal CLR. In ogni caso un'applicazione .NET compilata viene rappresentata con un formato intermedio indipendente dal linguaggio di partenza, dalla particolare architettura della CPU (x86, x64, PowerPC) e dal sistema operativo (Windows, Linux, etc.). Questo formato consente ad applicazioni scritte con linguaggi diversi di interoperare sia come chiamate di metodi, sia a livello di ereditarietà delle classi. L'insieme di regole che garantiscono l'interoperabilità tra le applicazioni in .NET è definito nel CLS[31] che fa parte dello standard ECMA-335 [6]. In breve esso limita le convenzioni sui nomi, i tipi di dato, i tipi di funzioni ed altro per stabilire un comune denominatore tra i diversi linguaggi. In realtà il CLS è solo una specifica rispetto alla quale il CLR fornisce un sovra-insieme di funzionalità: è possibile scrivere applicazioni non conformi al CLS che possono comunque essere eseguire dal CLR.

La macchina a stati

Un programma espresso in linguaggio intermedio è eseguito da un thread: l'esecuzione inizia da un metodo in cui possono essere invocati altri metodi. Ad ogni chiamata di metodo corrisponde la creazione di un nuovo *stack frame*. Uno dei principali obiettivi del CLR è nascondere i dettagli sul frame di una chiamata di metodo a livello di generatore di codice intermedio. Per consentire quest'astrazione il *frame* di una chiamata è integrato nel modello a stati del CLR. Questo ha determinato la definizione di due stati: *globale* e *del metodo.*

Lo **stato globale** è costituito dai *managed heap* e da uno *spazio d'indirizzamento condiviso*; il CLR ne gestisce l'accesso da parte dei thread di controllo concorrenti. In generale le istanze, i campi statici e gli elementi degli array possono essere acceduti da più thread, di conseguenza le eventuali *race condition* devono essere gestite.

Lo **stato del metodo** descrive l'ambiente in cui un metodo è eseguito: detto anche *invocation frame stack.* In questo stato sono presenti i seguenti elementi:

- Un *puntatore* alla successiva istruzione IL che deve essere eseguita del metodo considerato;

[30] Virtual Execution System

[31] Common Language Specification

- Un *evalutation stack*, locale al metodo e preservato tra una chiamata e la successiva, non può essere indirizzato;
- Un array di *variabili locali* al metodo, con indice iniziale 0, per il quale i valori delle variabili sono preservati come nell'evalutation stack;
- Un array degli *argomenti* del metodo, con indice iniziale 0;
- *methodInfo*, contenente informazioni in sola lettura sul metodo (signature, i tipi delle variabili locali e altro);
- Un *pool* di memoria locale, allocabile dinamicamente mediante apposite istruzioni, le porzioni di memoria allocate sono indirizzabili;
- *Riferimento* allo stato di ritorno, utilizzato per ristabilire lo stato del metodo che aveva invocato il metodo terminato;
- *Descrittore* per la sicurezza, utilizzato solamente dal sistema di sicurezza del CLI.

Evaluation stack

L'*evalutation stack* è costituito da slot che possono memorizzare qualunque tipo di dato, compresa l'istanza di un value type di cui è stato fatto l'unbox. Lo stato dello stack (l'altezza e i tipi degli elementi presenti) in un qualsiasi punto di un programma deve essere identico per ogni possibile percorso del flusso di controllo. Per esempio non è possibile eseguire un programma che cicla per un numero non stabilito di volte, inserendo ad ogni ciclo un nuovo elemento sullo stack.
Il numero massimo di elementi contenuti è definito dalla variabile `MaxStack` presente nell'header di un metodo e specificata nel codice IL da un'apposita direttiva *.maxstack*. Il suo valore è stabilito dal compilatore o dall'`ILGenerator` (in generazione a runtime) prima dell'esecuzione del metodo.

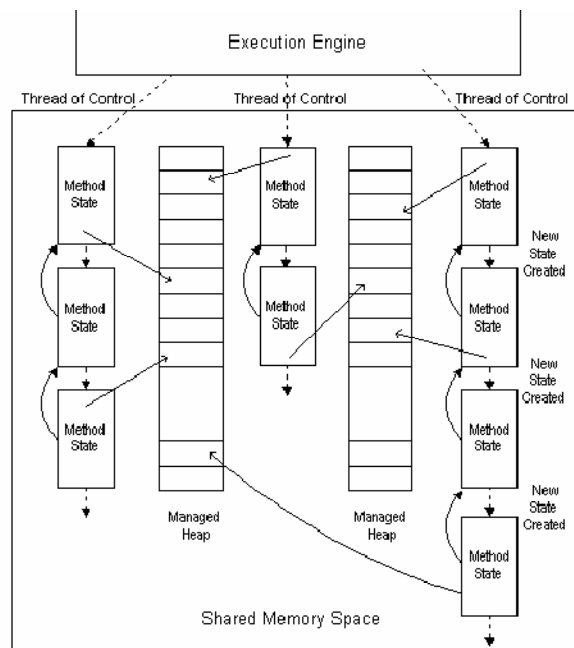


Figura 6.1, CLI Machine State Model

IL - Intermediate Language

I compilatori supportati dal CLI generano per ogni file sorgente:

- managed module (introdotto nel paragrafo 5.2.1) rappresenta le funzionalità previste nei metodi di una applicazione codificate in un formato astratto binario noto come MSIL o CIL;
- metadata (vedi paragrafo 5.2.1) la cui definizione e semantica sono parte dello standard ECMA-335.

Il codice IL è *managed*, ovvero gestito dal runtime attraverso:

1. Il controllo sui tipi (verifica e conversione) durante l'esecuzione;
2. La gestione delle eccezioni, non più a carico del sistema operativo;
3. Garbage collection per identificare ed eliminare automaticamente gli oggetti non più in uso.

Un'istruzione IL è composta da un codice operativo (opcode) e in alcuni casi anche da un parametro [2]. Una sequenza d'istruzioni IL del codice di un metodo può essere *valida* o *verificabile* o nessuna delle due. Il JIT si occupa di controllare la *validità* del codice, per cui se questo non è valido sicuramente non viene eseguito. La *verificabilità* è un problema di sicurezza e non di compilazione. Un codice è verificabile se non contiene hack: per esempio *stack overflow.* Un apposito algoritmo consente di avere sempre un numero corretto di slot dell'evalutation stack liberi per eseguire le istruzioni IL. Per ottenere questo l'algoritmo simula tutti i possibili percorsi del flusso di controllo, valutando di volta in volta se si ha uno stato dello stack valido per ogni istruzione.
Il linguaggio IL fornisce un insieme d'istruzioni per caricare valori sullo stack degli operandi e per salvare i valori che si trovano sullo stack nuovamente nella memoria; queste includono istruzioni per trattare valori costanti, variabili locali, parametri, campi di una classe, etc. L'invocazione di un metodo assume che gli argomenti siano caricati sullo stack, e il valore di ritorno è piazzato sullo stack al posto degli argomenti. A livello di IL si ha visibilità degli oggetti, con apposite istruzioni per crearli e gestirli, e si ha la possibilità di manipolare gli array direttamente. Vi sono numerose istruzioni IL disponibili per costruire programmi, esse vanno dalle comuni operazioni aritmetiche a quelle per controllare il flusso di controllo, a quelle per effettuare la chiamata di metodo.
Lo stack astratto del CLR esegue operazioni non solo sugli interi. Esso ha un ricco type system che include stringhe, interi, booleani, float, double, e cosi via. Tuttavia le istruzioni IL non sono tipate: ad esempio, l'IL fornisce l'istruzione `add` che somma gli ultimi due operandi in cima allo stack; non ci sono versioni distinte della `add` che operano su interi a 32-bit o 64-bit. Quando l'istruzione è eseguita, determina il tipo degli operandi sullo stack ed effettua l'operazione appropriata.

Consideriamo la seguente classe espressa in C#:

```
class Program
{
    public int Add3(int x, int y, int z)
    {
        int tmp = add(x, y);
        return add(tmp, z);
    }
}
```

Il metodo `Add3` è compilato nel seguente frammento di codice IL:

```
.method public hidebysig instance int32 Add3(int32 x, int32 y, int32 z) cil
managed
{
    .maxstack 2
    .locals init (
        [0] int32 tmp)
    L_0000: ldarg.1
    L_0001: ldarg.2
    L_0002: add
    L_0003: stloc.0
    L_0004: ldloc.0
    L_0005: ldarg.3
    L_0006: add
    L_0007: ret
}
```

Compilatore Just in Time

La simulazione della macchina a stack avviene traducendo l'IL e la semantica dello stack nel linguaggio macchina del livello sottostante. Questo può avvenire o a tempo d'esecuzione tramite il JIT compiler o a priori da servizi quali Ngen[32]. Per illustrare come lavora il JIT vediamo un esempio (figura 5.2) di compilazione di un metodo `Main`. All'avvio dell'esecuzione del metodo, il CLR determina tutti I tipi che riferisce. Per gestire gli accessi il CLR alloca una apposita struttura dati interna: in figura 5.2 il metodo `Main` riferisce il solo tipo `Console`, per cui una sola struttura dati viene allocata. Questa contiene tanti elementi quanti sono i metodi definiti nel tipo di dato. Ogni elemento memorizza l'indirizzo di memoria dove si trova l'implementazione del metodo.

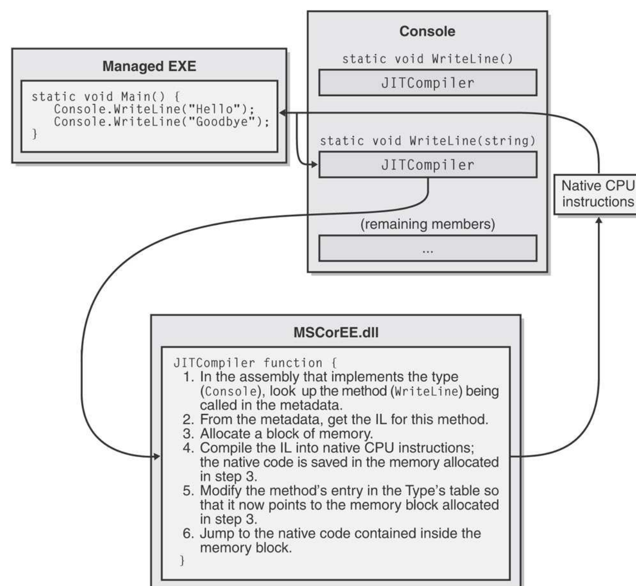


Figura 5.2, invocazione iniziale di un metodo

[32] Native Image Generator

La struttura è inizializzata inserendo in ogni elemento l'indirizzo di una funzione (JITCompile in figura 5.2) predefinita, interna e realizzata dal JIT. Nel `Main` quando il metodo `WriteLine` è chiamato per la prima volta, al suo posto viene eseguito il JIT che compila il metodo in istruzioni native della CPU: il JIT esamina l'assembly in cui è definito il metodo cercandone i relativi metadata, dopo di che verifica e compila il codice IL. Le istruzioni native prodotte sono salvate in un blocco di memoria allocato dinamicamente, il cui indirizzo è inserito nella struttura dati interna, rimpiazzando quello della funzione JIT per il metodo `WriteLine`. Alla fine il JIT salta a questo indirizzo. Ora `WriteLine` può essere eseguito. Alla terminazione, il controllo ritorna al Main. Nelle successive invocazioni di `WriteLine`, essendo il suo codice già verificato e compilato, il controllo può passare direttamente alla prima delle sue istruzioni native (figura 5.3). Ne consegue che si ha overhead solo alla prima chiamata di un metodo. Nelle successive invocazioni si avrà un'esecuzione al massimo delle prestazioni. Inoltre si deve considerare che la maggior parte delle applicazioni tendono ad invocare sempre gli stessi metodi. L'utilizzo di memoria dinamica per le istruzioni native permette di allocare spazio solo per il tempo in cui un metodo viene eseguito durante l'arco di vita del processo per poi rilasciarlo subito dopo. Questo comporta che in una successiva esecuzione dell'applicazione o se due istanze appartenenti ad AppDomain distinti vengono eseguite contemporaneamente il JIT compila nuovamente il codice IL. Il secondo passo di compilazione a runtime comporta un certo overhead aggiuntivo, ma grazie alla maggiore conoscenza da parte del JIT dell'ambiente di esecuzione questo può essere ridotto o eliminato, ottenendo nei seguenti casi anche prestazioni superiori a quelle del codice *unmanaged* (come per il C++):

- il JIT rileva il tipo di CPU al fine di emettere istruzioni specifiche per ottenere le migliori performance;
- il JIT determina che alcune condizioni o test hanno sempre un certo esito sulla macchina utilizzata, per cui emette un codice specializzato, più piccolo e quindi più veloce da eseguire;
- il CLR può esaminare l'esecuzione e nel mentre stabilire di ricompilare il codice IL riducendo ad esempio i *branch prediction* errati.

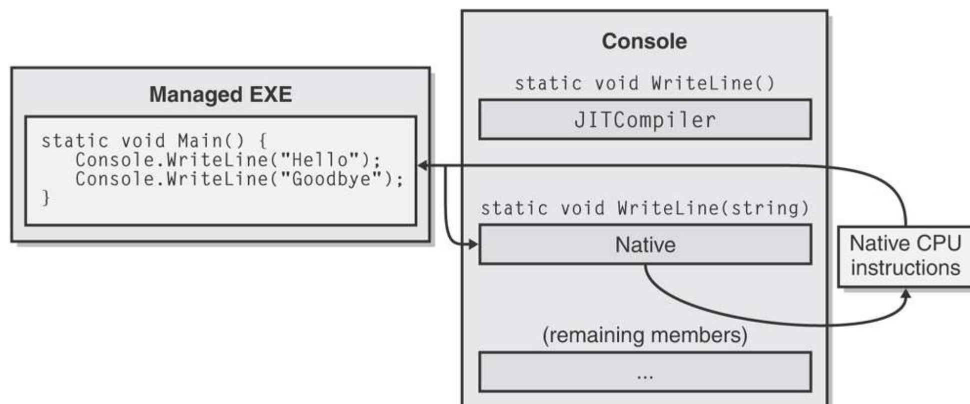


Figura 5.3, invocazione successiva di un metodo

5.4 Assembly

5.4.1 Introduzione

IL CLR non lavora con i *moduli managed* ma con gli *assembly*. Un assembly raggruppa uno o più managed module e/o file resource.

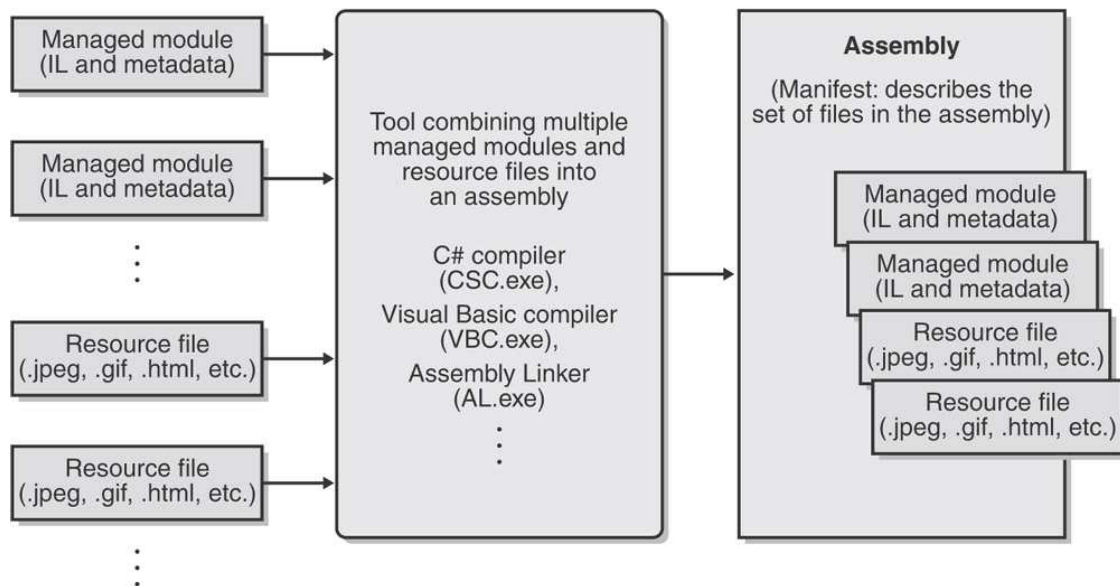


Figura 5.4, raggruppamento di moduli managed in assembly

Un assembly rappresenta:

- unità base per la gestione della sicurezza, a cui viene assegnato un insieme di permessi valutati a tempo di caricamento;
- per la gestione delle versioni, la più piccola unità a cui è assegnato un numero di versione; tutti i tipi e le risorse nello stesso assembly hanno la stessa versione; nel *manifest*, come vedremo, sono elencate le versioni degli assembly da cui dipende quello considerate;
- unità base di deployment, all'avvio di una applicazione solo l'assembly inizialmente richiesto deve essere presente, gli eventuali altri vengono recuperate su richiesta.

Un assembly può essere *statico* o *dinamico.* Statico quando comprende oltre ai tipi previsti nel .NET CTS[33] anche delle risorse come per esempio bitmap, file JPEG; gli assembly statici vengono memorizzati su disco nei file PE. *Dinamico* quando vengono eseguiti direttamente in memoria senza essere precedentemente salvati su disco. Un assembly, inoltre, può essere *privato* o *condiviso.* A livello di struttura e funzionalità non si hanno differenze. L'assembly privato è memorizzato nella sola directory dell'applicazione o in una sua sottodirectory. In questo modo risulta non accedibile da altre applicazioni. Generalmente viene creato dallo stesso autore degli altri componenti dell'applicazione per cui i requisiti

[33] Common Type System

sui nomi e versioni non sono rigidi. L'unico vincolo importante è che il nome deve essere unico all'interno dell'applicazione.
Un assembly è condiviso quando può essere utilizzato da più applicazioni. Questo comporta che la responsabilità su di esso è affidata a gruppi o organizzazioni: ne sono esempio gli assembly della .NET framework class library. Di conseguenza il tipo di partizionamento, i requisiti sui nomi e le versioni sono molto più vincolanti rispetto a quelli per gli assembly privati. Il nome che identifica un assembly deve essere globalmente unico. A garanzia di questo viene utilizzata una coppia di chiavi pubblica/privata generate da un algoritmo per la crittografia. Il CLR applicando questo algoritmo assicura l'utente sulla provenienza dell'assembly che vuole utilizzare. Un assembly condiviso viene memorizzato nella *Global Assembly Cache (GAC)*: per i dettagli si rimanda a [7,5].

5.4.2 Struttura

Indipendentemente dal tipo di assembly, vi è uno speciale metadato che descrive come gli elementi in un assembly sono correlati: il manifest. Nel manifest sono presenti:

- l'identità, il numero di versione ed opzionalmente la chiave pubblica dell'autore (tipicamente con assembly condivisi);
- il contenuto, i tipi e le risorse (esposte per essere usati esternamente) e dove questi sono memorizzati;
- dipendenze, l'elenco dei file che fanno parte dell'assembly e di quelli (esterni) riferiti;
- permessi di sicurezza
- custom attribute definiti per i componenti del manifest.

I diversi elementi che costituiscono un assembly (manifest, metadata relativi ai tipi, codice IL e un insieme di risorse) possono essere raggruppati in un unico file o in file multipli (figura 5.5). In questo secondo caso il collegamento tra i file che ne fanno parte non è gestito dal file system, ma dal CLR sulla base di quanto specificato nel manifest.

Figura 5.5, assembly organizzato in un singolo file (sinistra), assembly su più file (destra)

5.4.3 Formato binario

I metadati nel file sono organizzati in una base di dati relazionale che comprende numerose tabelle. Ogni tabella contiene dei valori interi con i quali vengono normalmente indicizzate altre tabelle. I dati di un programma sono memorizzati in quattro heap: stringhe, identificatori, GUID[34] a 128 bit e altre informazioni come la signature delle variabili locali. Le informazioni sui metodi definiti in un assembly sono inserite in un'apposita tabella *Method*. I nomi delle classi e dei metodi sono memorizzati nell'apposito heap *Strings*, mentre le stringhe usate nei metodi sono contenute nell'heap *User Strings*. Come illustrato in figura 5.6 l'indirizzo della prima istruzione nel codice IL di un metodo, detto *indirizzo virtuale relativo* (RVA), è mantenuto in un campo di ogni elemento di una *lista di metodi*: insieme di righe consecutive nella tabella *Method.*

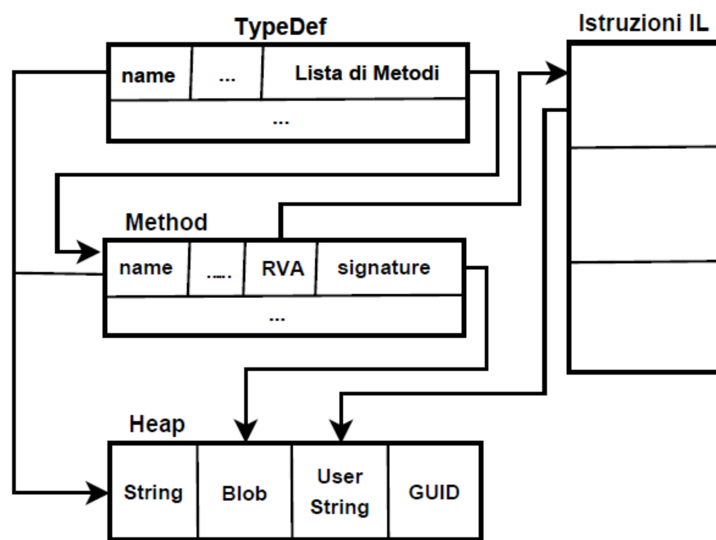


Figura 5.6: organizzazione di un assembly, metadati e codice IL.

In ognuno di questi elementi vi è anche un campo *Signature* che mantiene un indice del *Blob heap* dove sono memorizzate le signature delle variabili locali ad un metodo. Questa lista a sua volta viene memorizzata per ogni classe nella tabella *TypeDef* contenente le informazioni sui tipi definiti in un assembly.

Il meta-programma sviluppato nella presente tesi sfrutta la libreria `CLIFileRW` [3] per poter esaminare il codice IL. Il modello di reflection del CLR, infatti, non dà supporto per accedere alle istruzioni IL di un metodo nascondendo i dettagli sulla struttura degli assembly. La libreria è stata progettata per dare un accesso a basso livello fornendo però quelle astrazioni necessarie a colmare il gap tra la struttura binaria del file e gli oggetti ottenibili con la reflection. Quest'astrazione si concretizza in un cursore, `ILCursor`, dello stream di istruzioni IL del body di un metodo dato il suo `MethodInfo`. Nel `CLIFileRW` per avere accesso ad un file nel modo più performante possibile è stato utilizzato il *memory mapping*; con un

[34] Globally Unique IDentifier

apposito insieme di tipi si effettua correttamente la lettura della memoria senza utilizzare puntatori che avrebbero introdotto codice *unsafe*: la memoria viene esposta come un array di byte.

5.5 Layout dei tipi in memoria e P/Invoke

Per default, l'esatto layout in memoria di un tipo è *opaco*. Il CLR usa uno schema di *layout virtuale* e riordina i campi per ottimizzare l'uso e gli accessi alla memoria. Se il layout dei campi coincidesse con quello di definizione all'interno dei tipi potrebbe essere necessario effettuare padding per evitare accessi non allineati ai singoli campi. Per evitare questo il CLR riordina i campi in modo tale che non sia necessario effettuare packing e non vi sia al tempo stesso spreco di memoria per avere tutti i dati allineati. Questo può avvenire perché tutti gli oggetti sono riferiti tramite *handle*, simili ai puntatori, ma che a differenza di questi non sono dereferenziabili e non vi è definita un'aritmetica (non è possibile sommare un intero a un handle, e poi accedere alla locazione di memoria corrispondente all'indirizzo risultante). D'altra parte il programmatore C/C++ vuole avere un controllo totale sul proprio programma (proprio per usare operazioni aritmetiche sui puntatori), per cui gli viene garantito che il layout in memoria coincida con l'ordine di dichiarazione. Per questo motivo, per scrivere programmi che manipolano esplicitamente la memoria basandosi sul formato di un tipo in memoria, è richiesto un meccanismo per evitare questo layout automatico e controllare esplicitamente il layout di un tipo. Il CLR permette di esplicitare il meccanismo e le modalità di layout in memoria nel seguente modo:

```
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Auto)]
public struct Foo
{
    public short a;
    public double b;
    public short c;
}

[StructLayout(LayoutKind.Sequential)]
public struct Bar
{
    public short a;
    public double b;
    public short c;
}

[StructLayout(LayoutKind.Explicit)]
public struct Baz
{
    [FieldOffset(8)]
    public short a;
    [FieldOffset(0)]
    public double b;
    [FieldOffset(2)]
    public short c;
}
```

Questi tre tipi sono logicamente equivalenti, ma ognuno ha una diversa rappresentazione in memoria. Su una macchina x86 i campi di `Foo` sono ordinati {a, c, b}, quelli di `Bar` sono ordinati {a, b, c} e quelli di `Baz` sono ordinati {b, c, a}. Poiché C# assume per default un layout di tipo sequenziale per le struct, l'attributo sul tipo `Bar` è superfluo.

Il CLR permette l'interoperazione di codice managed con codice unmanaged: ossia è possibile invocare dall'interno di un programma gestito routine native C/C++, contenute in moduli unmanaged (quindi non soggetti a garbage collection e a tutti gli altri servizi offerti dal CLR) e viceversa. Questo tipo d'invocazioni è detto *cross-mode*: in entrambi i casi, il codice emesso per la chiamata è considerabilmente differente rispetto a quello per la chiamata tra metodi che avvengono nello stesso modo d'esecuzione (managed-managed / unmanaged-unmanaged). Consideriamo un esempio di utilizzo del P/Invoke:

```
public static class CALRuntime
{
    [DllImport("aticalrt")]
    public static extern CALresult calDeviceGetCount(out uint count);
    …
}
```

Usare P/Invoke è abbastanza semplice. P/Invoke permette di marcare i metodi come importati da una classica DLL pre-CLR utilizzando il custom-attribute `DllImport` (contenuto nel namespace `System.Runtime.InteropServices`). A questi metodi deve essere applicato il modificatore *extern* e la loro signature deve coincidere con quella della funzione target nella DLL esterna (a meno di non usare un'opzione dell'attributo `DllImport` per specificare un identificatore differente). Alla fine, ogni metodo sul quale può essere effettuato P/Invoke possiede due signature: una esplicita visibile dal codice gestito che effettua la chiamata, e una implicita che è quella attesa dalla funzione della DLL esterna. E' compito dell'engine di P/Invoke inferire la signature implicita basandosi su regole di mapping di default oppure su annotazione esplicite tramite custom attributes. Queste informazioni sono usate dall'engine di P/Invoke per chiamare le funzioni `LoadLibrary` e `GetProcAddress`, rispettivamente, proprio prima di invocare il metodo annotato.
A seconda del tipo del parametro, il P/Invoke engine potrebbe aver bisogno di effettuare una conversione in memoria (si pensi ad esempio alla diversa rappresentazione delle stringhe in C e C#). I tipi che possono essere copiati senza conversione sono detti tipi *blittable*. Al contratio i tipi che richiedono la conversione sono detti *non-blittable*. Chiaramente le performance di una chiamata P/Invoke che fa uso di soli tipi blittable sono superiori rispetto ad una che usa tipi non-blittable.
Oltre a questo overhead, per comprendere i costi aggiuntivi implicati dall'uso del meccanismo di P/Invoke si deve tenere in considerazione un altro aspetto fondamentale che differenzia gli ambienti gestiti da quelli non gestiti: la garbage collection.

Senza addentrarci nei dettagli dell'algoritmo di garbage collection [20], è sufficiente dire che dopo una fase iniziale di raccolta informazioni (fase di *mark*) per determinare gli oggetti non più utilizzati, per evitare la frammentazione dell'heap avviene una seconda fase detta di *compact:* gli oggetti ancora in uso sono spostati in modo tale che tutta la memoria disponibile sia concentrata in'unica zona. Questo

permette allocazioni più veloci, perché non è necessario ricercare un’area di memoria sufficientemente grande (come invece fa il runtime del C/C++). Il gestore di marshalling blocca automaticamente la memoria allocata nell'heap di runtime se il relativo indirizzo viene passato a una funzione non gestita, per impedire al garbage collector lo spostamento del blocco di memoria durante la fase di compact. Infatti, se il codice unmanaged stesse utilizzando proprio quell’area di memoria nel momento in cui questa venisse spostata, si creerebbero seri problemi di consistenza. Questa operazione è detta *pinnning* della memoria e può impattare negativamente sulle prestazioni della collection in quanto tende a creare frammentazione. Il sovraccarico di P/Invoke è compreso tra le 10 e le 30 istruzioni x86 per chiamata. Oltre a questo costo fisso, il marshalling crea un ulteriore sovraccarico. Tra i tipi copiabili caratterizzati dalla stessa rappresentazione nel codice gestito e non gestito non sono presenti costi di marshalling. La conversione tra `int` e `Int32`, ad esempio, non prevede alcun costo.

5.6 Task Parallel Library

PBricks permette l'esecuzione di computazioni data parallel sfruttando architetture multicore. Questo grazie all'introduzione del `MulticorePBrick`. Tale componente è stato realizzato come metro di giudizio delle performance, quindi non è del tutto ottimizzato. Manca anche la sincronizzazione sui dati condivisi, quindi può essere utilizzato solo per realizzare computazioni di tipo map. Per il suo sviluppo ci siamo serviti della Task Parallel Library di Microsoft. Nel seguito sarà darà una descrizione generale della libreria e del suo funzionamento interno. Per affrondire le parti non coperte si veda [21].

Task Parallel Library (TPL) è un componente progettato per semplificare la scrittura di codice gestito in grado di utilizzare automaticamente più processori. Utilizzando questa libreria, è possibile esprimere il potenziale parallelismo nel codice sequenziale esistente, in base al quale le attività parallele esposte verranno eseguite simultaneamente su tutti i processori disponibili. In questo modo si ottengono generalmente notevoli aumenti di velocità. TPL nasce dalla collaborazione tra Microsoft Research, il team CLR di Microsoft e il team Parallel Computing Platform e costituisce un importante componente della libreria Parallel Extensions che verrà rilasciata come parte del .NET 4.0 Framework.

Parallel Extensions offre diversi modi per esprimere il parallelismo nel codice:

- *Declarative data parallelism* - Parallel Language Integrated Query (o Parallel LINQ, PLinq) è un'implementazione di LINQ-to-Objects [22] che esegue query in parallelo, scalando il più possibile per utilizzare tutti i core e processori disponibili su una macchina. Grazie al fatto che le query sono dichiarative, il programmatore ha la possibilità di esprimere ciò che vuole ottenere, piuttosto che il modo con il quale farlo. In questo senso si parla di approccio dichiarativo o funzionale.
- *Imperative data parallelism* - Parallel Extensions contiene anche meccanismi per esprimere operazioni imperative data-oriented come i cicli for e foreach, facendosi carico di dividere equamente il lavoro nel ciclo (ossia le strutture dati sulle quali si itera) in modo tale che possa essere eseguito su hardware parallelo.
- *Imperative task parallelism* - Piuttosto che usare i dati per guidare il parallelismo, Parallel Extensions permette di esprimere il potenziale parallelismo tramite espressioni e statements che prendono la forma di lightweight tasks. Parallel Extensions schedula questi task per l'esecuzione su hardware parallelo e fornisce la possibilità di cancellazione o attesa su uno o più tasks.

5.6.2 Task e Task Manager

Il principale concetto nella libreria Parallel Extensions è il *Task* (sopra al quale sono costruiti anche PLinq e le versioni parallele di for e foreach), che è una piccola unità di codice, solitamente rappresentata come una *lambda function* (un *delegate* nel gergo .NET), che può essere eseguita indipendentemente dal resto del programma. In quest'accezione è semanticamente equivalente a un thread, ma al contrario di questo un task è un oggetto più leggero e non incorre nell'overhead dovuto alla creazione di un thread del sistema operativo. I task sono accodati e gestiti dal *Task Manager* nella propria coda task globale, e infine sono schedulati per l'esecuzione usando un pool di thread (OS) gestito anch'esso dal

task manager. Per default, sono creati tanti thread quanti processori (o core) sono presenti nel sistema, benché questo numero possa essere modificato manualmente.

5.6.3 Bilanciamento del carico - Work stealing

Nel passaggio dalla versione sequenziale a quella parallela di un programma gli sviluppatori spesso finiscono con il dividere il lavoro in modo statico. Ad esempio, in un ray tracer, l'immagine viene spesso suddivisa in parti uniformi, ciascuna delle quali viene elaborata da un thread separato. In generale, non si tratta di una buona idea poiché il carico di lavoro effettivo potrebbe essere suddiviso in modo non uniforme. Se la parte inferiore dell'immagine richiede, ad esempio, il doppio del tempo per il calcolo a causa dei riflessi, i thread relativi alla parte superiore dell'immagine restano per gran parte del tempo in attesa del completamento dei thread relativi alla parte inferiore. Anche se il lavoro viene suddiviso in modo uniforme, questa situazione può comunque verificarsi a causa di errori di pagina o di altri processi del sistema eseguiti in concomitanza. Per ovviare a questo problema e quindi ottenere una maggiore scalabilità su sistemi con più processori, TPL utilizza tecniche di appropriazione del lavoro per adattare e distribuire dinamicamente gli elementi di lavoro sui thread di lavoro. Ad ogni thread è associata una coda locale di task; quando inattivo, cerca di raccogliere task dalla propria coda locale; se la trova vuota, raccoglie un gruppo di task dalla coda globale, e li inserisce nella propria coda locale, per poi eseguirli uno a uno. La disciplina con la quale accede alla propria coda locale è di tipo LIFO, poiché è più probabile che i dati di un task accodato dopo stiano in cache mentre quelli di task creati meno recentemente genereranno dei cache miss con maggiore probabilità. Se anche la coda globale è vuota, il thread cercherà task nelle code dei propri vicini e prenderà i task che sono stati in coda per più tempo (*work stealing*) applicando in questo caso una disciplina FIFO, perché è meno probabile che i dati di un task vecchio stiano in cache.

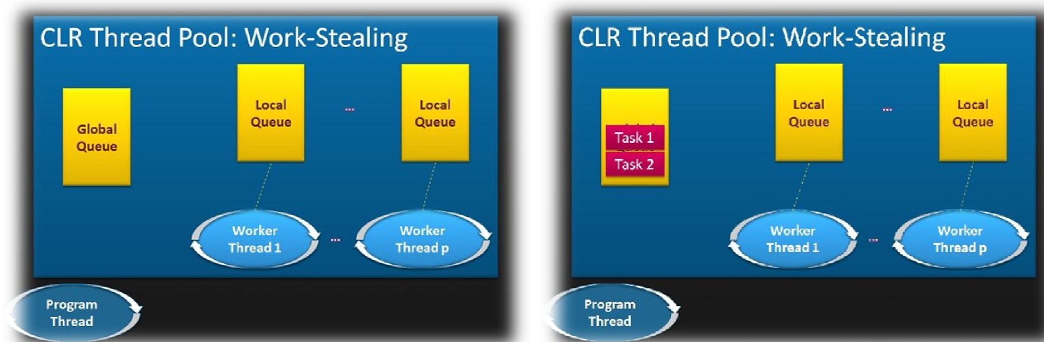


Figura 5.7, le code sono inizialmente vuote (sinistra); sono depositati due task nella coda globale (destra)

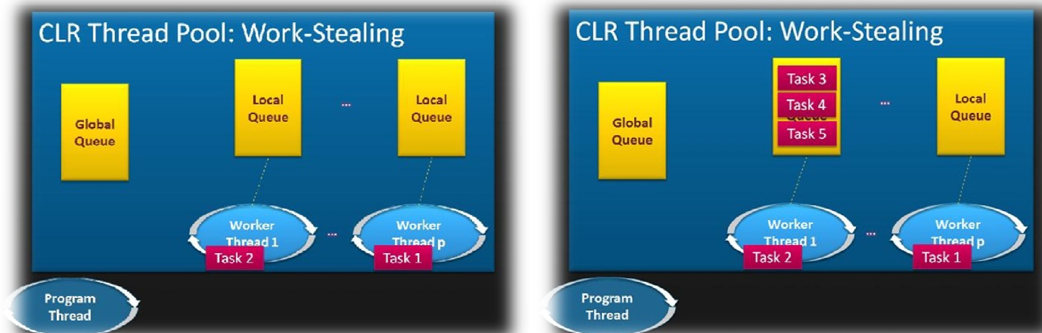


Figura 5.8, i worker thread prelevano i task dalla coda globale (sinistra); l'esecuzione del task 2 crea due task, questi sono accodati localmente al worker thread 1 (destra)

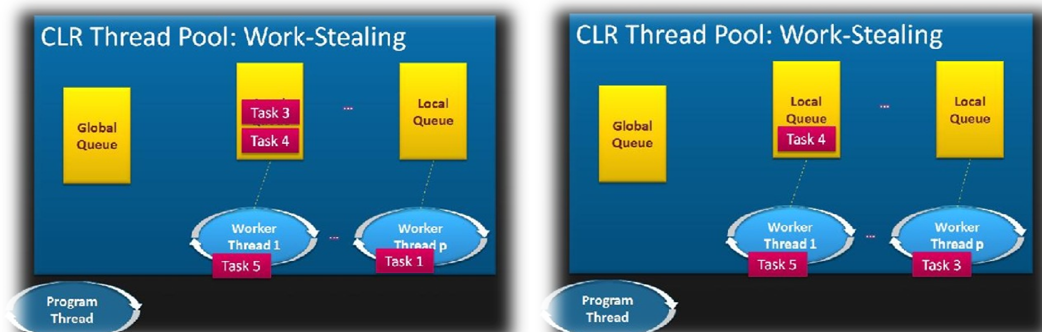


Figura 5.9, il worker thread 1 termina il proprio lavoro e preleva il primo task accodato (sinistra); il worker thread p termina il proprio lavoro e preleva l'ultimo task accodato nella coda del worker thread 1 (destra)

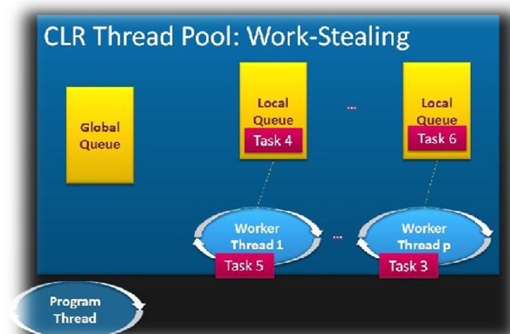


Figura 5.10, l'esecuzione del task 3 genera un altro task che è accodato localmente al worker thread p

Il vantaggio è che non esiste quasi alcuna sincronizzazione tra i thread di lavoro poiché le code di lavoro sono distribuite e la maggior parte delle operazioni viene svolta in locale per il thread, condizione essenziale per la scalabilità. Inoltre se il carico di lavoro non è uniforme, è possibile che un thread di lavoro impieghi molto tempo per un'attività specifica, ma in questo caso ci saranno altri thread di lavoro

che si approprieranno del lavoro della sua coda, tenendo impegnati tutti i processori. La distribuzione dinamica del lavoro è fondamentale nelle applicazioni comuni, poiché è difficile prevedere il tempo necessario per l'esecuzione di un'attività. Questo è vero soprattutto per i sistemi desktop in cui i processori sono condivisi tra molti processi differenti e in cui è impossibile prevedere il tempo di esecuzione necessario ai thread di lavoro.

Oltre a eseguire la distribuzione dinamica del lavoro, la libreria regola dinamicamente il numero di thread di lavoro in caso di blocco dei thread. Leggere file, attendere che un tasto venga premuto e recuperare un nome utente (per accedere alla rete in un dominio) sono alcuni esempi di operazioni di blocco. Se un'attività viene inconsapevolmente bloccata, le prestazioni possono risentirne poiché diminuisce il livello di simultaneità (ma il programma resta ancora in esecuzione). Per migliorare le prestazioni, la libreria rileva automaticamente la presenza di thread di lavoro bloccati e, se necessario, ne inserisce altri aggiuntivi per mantenere il livello di concorrenza. Una volta sbloccate le operazioni, alcuni thread di lavoro potranno essere ritirati per ridurre il costo del passaggio da un thread all'altro.

Quando si trovano in esecuzione, i task sono indipendenti l'uno dall'altro e il cambio di stato di un task non influenza gli altri. Come conseguenza, se essi utilizzano una risorsa condivisa, dovranno essere soggetti a sincronizzazione utilizzando lock o altri costrutti.

E' importante far notare che le primitive della libreria esprimono soltanto il potenziale parallelismo, ma non lo garantiscono. Ad esempio, su un computer a processore singolo, i cicli for paralleli vengono eseguiti in sequenza, adattandosi alle prestazioni del codice strettamente sequenziale. Tuttavia, su un computer dual core, la libreria utilizza due thread di lavoro per eseguire il ciclo in parallelo, a seconda del carico di lavoro e della configurazione. Ciò significa che è possibile oggi introdurre il parallelismo nel codice e che le applicazioni utilizzeranno automaticamente più processori quando questi saranno disponibili. Nello stesso tempo, il codice funzionerà correttamente sui computer più vecchi a processore singolo.

Parte III - PBricks

Capitolo 6

6.1 PBricks API

In questo capitolo si darà uno sguardo d'insieme alla libreria PBricks dal punto di vista dell'utilizzatore finale. Si vedrà come è possibile definire una computazione e come lanciarla in sequenziale, piuttosto che su GPU o su multicore. Si daranno esempi su come realizzare calcoli di tipo map, reduce usando gli `Stream` e come collegare tra di loro le esecuzioni di più kernel (tramite il costruttore di computazione `PipePBrick`). Sarà infine spiegato il modello di memoria, ossia come i diversi livelli della gerarchia di memoria di una GPU sono resi disponibili al programmatore.
Bisogna precisare che l'obiettivo della presente tesi non è tanto quello di fornire una strutturazione delle computazioni esplicitando pattern di programmazione parallela (per questo sono disponibili soluzioni di più alto livello quali Muskel [58], OcamlP3l [59], SkeTo [57], etc.), o di poter esprimere una forma di parallelismo generica come è possibile fare in Assist [60], quanto di permettere l'esecuzione su GPU, pertanto, si assume che venga adottata una forma di parallelismo data parallel.

6.1.1 Esempio di utilizzo di PBricks

Supponiamo di voler realizzare un semplice programma che mappi una funzione su un determinato insieme di dati in ingresso $y = f(x)$, ad esempio per raddoppiare l'insieme dei dati di ingresso.
L'utilizzo di PBricks prevede la scrittura di una classe con all'interno un metodo che rappresenta la funzione da modellare:

```
class MyComputation
{
    public void SimpleKernel(InputStream<int> source, OutputStream<int> dest)
    {
        dest.Write(source.Current * 2);
    }
}
```

Il metodo rappresenterà il kernel eseguito in parallelo. Si noti l'uso di particolari tipi per modellare i parametri (`InputStream` e `OutputStream`) e il fatto che lo stream dei risultati sia passato come argomento del metodo: questo permette una facile generalizzazione nel caso in cui si voglia che il kernel restituisca più di uno stream di output.

```
    class Program
    {
        static void Main(string[] args)
        {
            var array = Enumerable.Range(1, 1024).ToArray();

            var source = Stream.CreateInput(array);
            var results = Stream.CreateOutput(source);

            var brick = new SeqPBrick(new MyComputation(), "SimpleKernel");
            brick.GetResults(new[] { source }, new[] { results });
```

```
            foreach (var item in results)
            {
                // Code that use "item"
                // ...
            }
        }
    }
```

Lo stream di input viene creato a partire da un normale array:

```
var source = Stream.CreateInput(array);
```

e lo stream di output è creato a partire da quello di input:

```
var results = Stream.CreateOutput(source);
```

Successivamente si istanzia un `SeqPBrick`, ossia si indica che il metodo `SimpleKernel` dovrà essere eseguito in sequenziale:

```
var brick = new SeqPBrick(new MyComputation(), "SimpleKernel");
```

E' presente anche un overload nel caso in cui la computazione non faccia uso di variabili d'istanza:

```
var brick = new SeqPBrick(typeof(MyComputation), "SimpleKernel");
```

L'idea è che passando l'istanza di un oggetto è possibile riusarla in seguito, sia su PBricks che rappresentano metodi diversi, sia sullo stesso PBrick nel caso si voglia salvare lo stato della computazione e riprenderlo successivamente.

Con il metodo `GetResults` viene lanciata la computazione, passando come parametri gli array contenenti gli stream in ingresso e uscita:

```
brick.GetResults(new[] { source }, new[] { results });
```

Sui risultati è poi possibile iterarare in questo modo:

```
foreach (var item in results)
{
    // Code that use "item"
    // ...
}
```

che è equivalente a:

```
var enumerator = results.GetEnumerator();

while (enumerator.MoveNext())
{
    var item = enumerator.Current;

    // Code that use "item"
```

```
    // ...
}
```

Gli unici tipi permessi come parametri formali di un metodo kernel, oltre agli stream, sono i tipi primitivi e gli array di tipi primitivi:

```
public void KernelWithConstantBuffers(InputStream<int> source,
                                      OutputStream<int> dest,
                                      int c1, int[] c2)
{
    dest.Write(source.Current * c1 * c2[0]);
}
```

All'interno dei metodi è possibile riferire variabili statiche o d'istanza della classe. Ad esempio:

```
    class ComputationWithState
    {
        [ComputeField]
        int localSharedData;

        [ComputeField]
        static int globalSharedData;

        /// <summary>
        /// Used to test local shared memory.
        /// </summary>
        public void LocalSharedMemoryKernel(InputStream<int> source,
                                            OutputStream<int> dest)
        {
            localSharedData += source.Current * 2;
            dest.Write(localSharedData);
        }

        /// <summary>
        /// Used to test global shared memory.
        /// </summary>
        public void GlobalSharedMemoryKernel(InputStream<int> source,
                                             OutputStream<int> dest)
        {
            globalSharedData += source.Current * 2;
            dest.Write(globalSharedData);
        }
    }
```

I membri (statici, d'istanza) della classe usati per rappresentare lo stato della computazione devono essere annotati con l'attributo `ComputeField`, questo perchè è possibile usare all'interno della classe anche membri che non realizzano stato, ad esempio quando si voglia partire da codice preesistente per evitare di dover estrarre da una classe i campi e i metodi inerenti la computazione al fine di minimizzare le modifiche.

Da quanto visto fin'ora si nota che il modello di programmazione è semplificato rispetto allo sviluppo con CAL, CUDA o Brook+ poichè è solamente necessario (dopo aver definito i kernel in un linguaggio ad alto livello) istanziare gli stream ed eseguire il calcolo delegandolo ad un particolare PBrick.

Naturalmente è possibile usare all'interno del metodo qualsiasi tipo di costrutto: selezione, iterazione mediante cicli for, while, do/while. Fanno eccezione i costrutti *switch* e *goto* che come mostrato nella sezione A.6 dell'appendice presentano alcuni sostanziali problemi.

Vi sono alcune limitazioni per quel che riguarda le operazioni effettuabili all'interno di un kernel:

- non si possono istanziare oggetti
- non si possono chiamare metodi della FCL[35]
- i kernel non possono chiamare se stessi e/o chiamare altri metodi

L'ultimo punto costituisce una limitazione dell'attuale versione di PBricks (Sviluppi Futuri) laddove i primi due costituiscono un vincolo forte dovuto al modello sottostante: non è possibile esprimere chiamate a codice arbitrario all'interno di un GPU (ad esempio avrebbe poco senso aprire una connessione ad un DBMS!).

Alla costruzione del PBrick viene controllato che il metodo passato come kernel soddisfi determinati requisiti:

- Il metodo deve esistere.
- Il metodo non può restituire valori.
- Deve esserci almeno uno stream di input e uno di output.
- Per via di limitazioni ATI un solo `ScatterStream`/`GatherStream` è permesso.
- Solo tipi primitivi sono permessi per i parametri formali diversi dagli stream.
- La genericità degli stream deve essere 1, e questo deve essere un tipo primitivo.

Costruendo differenti tipi di PBrick è possibile eseguire la computazione su multicore (`MulticorePBrick`) oppure su una scheda video avente capacità di GPU computing (`GPUPBrick`). Un `GPUPBrick` rappresenta l'esecuzione su un dispositivo di calcolo (sia esso costituito da una o più schede video collegate assieme) *omogeneo*. L'esecuzione su dispositivi *eterogenei* (ossia appartenenti a produttori distinti, o aventi diverse capacità e/o prestazioni diverse) è delegata a un livello superiore. Si è scelta questa soluzione per semplificare lo sviluppo che in caso contrario avrebbe dovuto prevedere uno schedulatore in grado di gestire il bilanciamento del carico di lavoro delle unità di calcolo in base alle capacità di quest'ultime e capace di interoperare con dispositivi diversi. Questa generalizzazione è però prevista negli sviluppi futuri.

Nel caso si voglia lanciare la computazione su GPU è necessario comunicare alla libreria il tipo di scheda video presente (potrebbero essere presenti nel sistema due tipi di schede video):

```
PBricksSettings.Provider = new ATIProvider();
```

[35] Framework Class Library

`PBricksSettings` è un singleton [4], realizzato come classe statica che permette di settare (e cambiare da qualsiasi punto del codice) il `Provider` per poter ad esempio comandare l'esecuzione su una scheda Nvidia; `ATIProvider` è la factory concreta [4].

Come vedremo in seguito (nel capitolo 9) una parte del sistema, lo *Scheduler*, s'incarica di gestire la presenza di GPU multiple[36] (per sfruttare le tecnologie CrossFire di ATI o SLI di Nvidia), partizionando i dati e lanciando l'esecuzione in contemporanea.

Ogni PBrick eredita da `PBrick`, la classe base che rappresenta un "brick" computazionale. Di seguito sono riportate l'interfaccia pubblica e gerarchia delle classi:

```
public abstract class PBrick
{
    public PBrick(object state, string method)
    public PBrick(Type type, string method)

    public void GetResults(BaseInputStream[] inStreams,
                           BaseOutputStream[] outStreams,
                           params object[] constants)
}
```

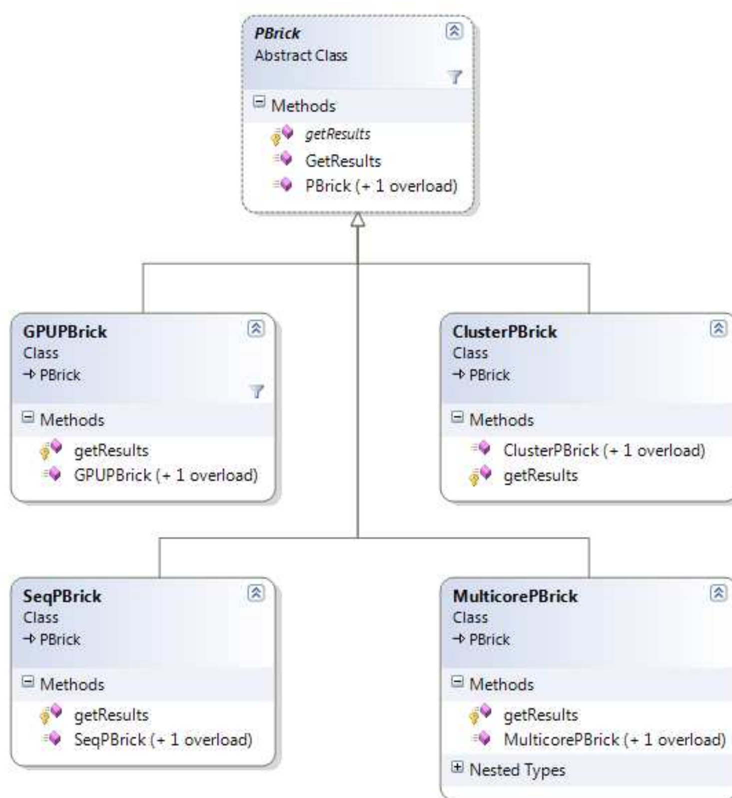


Figura 6.1, la tassonomia dei PBrick

[36] Con il vincolo che siano dello stesso produttore

E' possibile concatenare più step di calcolo utilizzando il PBrick `PipePBrick`, passando tramite costruttore i PBrick che ne costituiscono gli step:

```
var array = Enumerable.Range(0, 1024).ToArray();

var source = Stream.CreateInput(array);
var results = Stream.CreateOutput(source);

var state = new MyComputation();

var pipePBrick =
    new PipePBrick(new MulticorePBrick(state, "SimpleKernel"),
                   new GPUPBrick(state, "SimpleKernel"));

pipePBrick.GetResults(new [] { source }, new[] { results });
```

In questo caso la computazione prevede due step: un primo step eseguito su multicore e il secondo su GPU. Attualmente `PipePBrick` esegue il calcolo in modalità *batch*, ossia inizia uno step solamente quando sono disponibili tutti i dati del passo precedente. Chiaramente questo comporta un maggior tempo di completamento della computazione (dovuto al minor sfruttamento del parallelismo utilizzabile) in tutti i quei casi in cui si possa effettivamente tenere occupate più unità di elaborazione. Si è optato per questa soluzione perchè altrimenti sarebbe stato necessario sviluppare uno schedulatore in grado di decidere (con un preciso modello dei costi delle comunicazioni e una stima della grana del calcolo) la giusta grana dei messaggi scambiati tra le unità costituenti gli stadi del pipe e avviare molteplici volte le computazioni su dispositivi e dati diversi; questo esulava dallo scopo principale del nostro lavoro. Ciò non toglie, come ricordato negli sviluppi futuri, una possibile estensione del sistema.

La realizzazione di una libreria utilizzando PBricks dovrebbe seguire una strutturazione a livelli del tipo:

```
// Livello 2: utente programmatore di kernel.
class Kernels
{
    public void Transpose(InputStream<int> a, OutputStream<int> b)
    {
        b.Write(a.ElementAt(b.Y, b.X));
    }

    public void Sum(InputStream<int> a,
                    InputStream<int> b,
                    OutputStream<int> c)
    {
        c.Write(a.Current + b.Current);
    }

    public void Multiply(GatherStream<int> a,
                         GatherStream<int> b,
                         OutputStream<int> c, int k)
    {
        int total = 0;

        for (int j = 0; j < k; j++)
            total += a[c.Y, j] * b[j, c.X];
```

```
            c.Write(total);
        }
    }


    // Livello 1: utente implementatore della libreria.
    public static class MatrixLib
    {
        static MatrixLib()
        {
            PBricksSettings.Provider = new ATIProvider();
        }

        public static int[,] GPU_Sum(int[,] a, int[,] b)
        {
            var streamA = new InputStream<int>(a);
            var streamB = new InputStream<int>(b);
            var streamC = new OutputStream<int>(a.Length);

            var brick = new GPUPBrick(typeof(Kernels), "MatrixSum");
            brick.GetResults(new[] { streamA, streamB }, new[] { streamC });

            return streamC.ToMatrix2D();
        }

        …
    }


    // Livello 0: utente finale della libreria.
    class Program
    {
        public static void Main()
        {
            // Setup matrices code.
            // a = …; b = …;

            var c = MatrixLib.GPU_Sum(a, b);
        }
    }
```

Modello di memoria

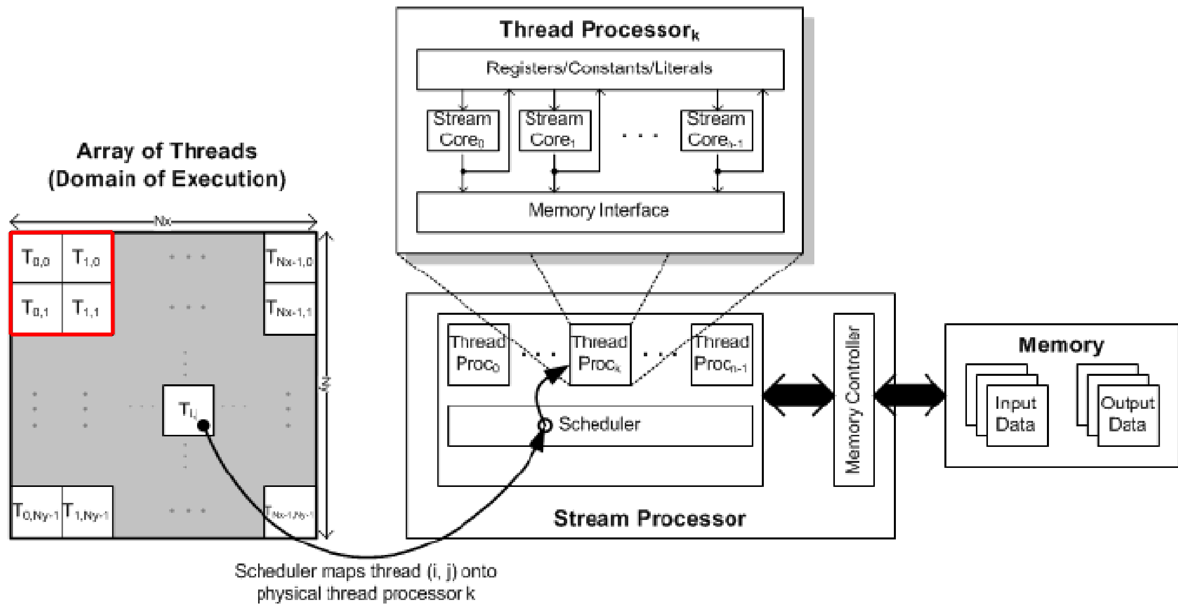
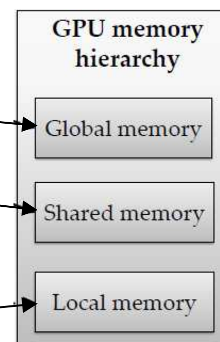


Figura 6.2, il modello d'esecuzione su GPU

Riprendiamo la figura 6.1 che mostra il modello d'esecuzione di una computazione su GPU; in rosso sono evidenziati i thread appartenenti ad uno stesso blocco. La gerarchia di memoria di una GPU costuita da (riprendendo la terminologia Nvidia) memoria globale, condivisa e locale è resa accessibile al programmatore di alto livello con il seguente mapping:

```
class ComputationWithState
{
    [ComputeField]
    int _lField;

    [ComputeField]
    static int _gField;

    //...

    public void Kernel(InputStream<int> source,
                       OutputStream<int> dest)
    {
        int local;

        //...
    }
}
```



La cooperazione tra thread di uno stesso blocco avviene mediante i campi d'istanza, la cooperazione tra gruppi avviene mediante campi statici della classe.

Gli `Stream` sono il mezzo con il quale passare i dati di input e leggere i risultati del calcolo:

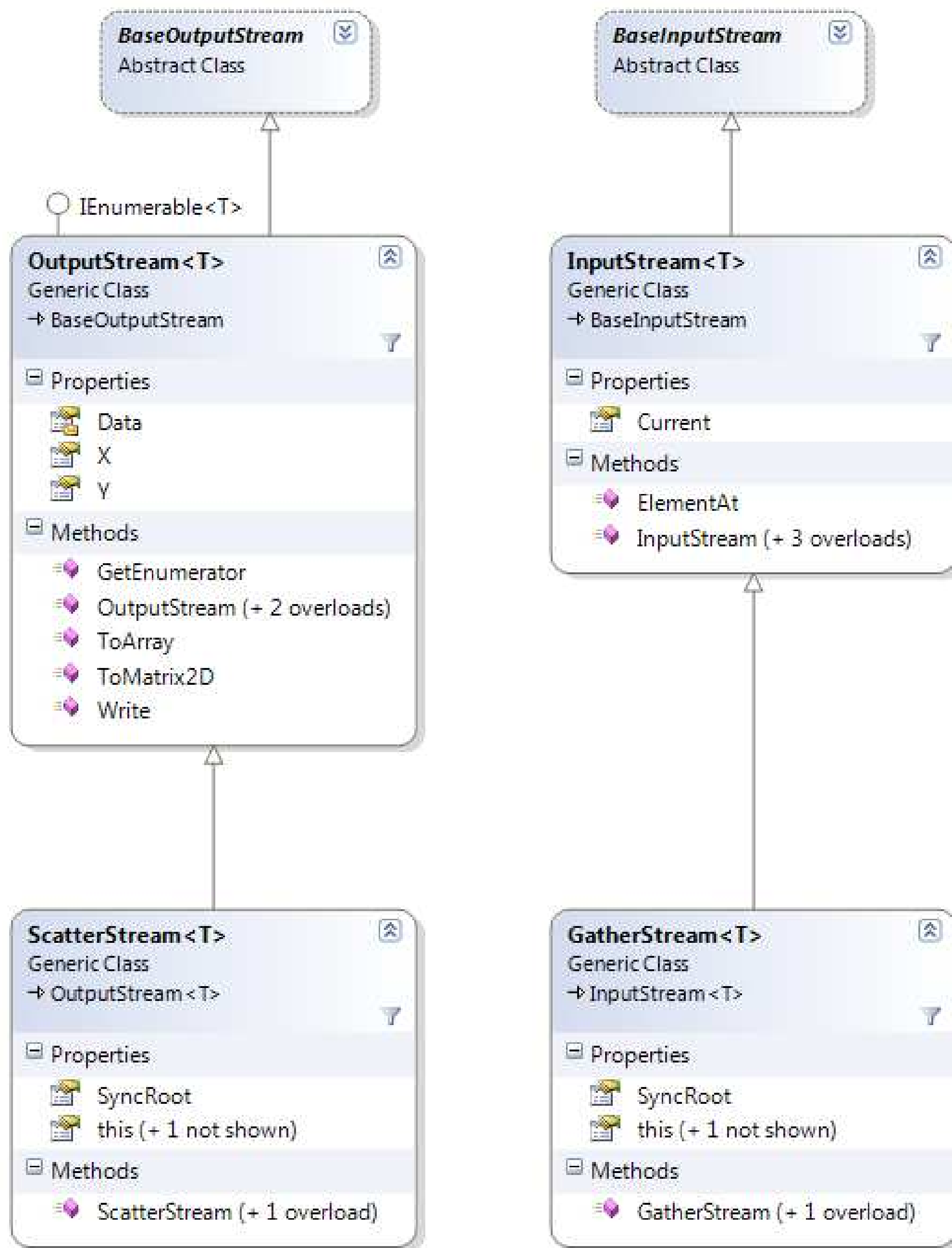


Figura 6.3, la tassonomia degli stream

Sono state definite due superclassi comuni `BaseInputStream` e `BaseOutputStream` per poter definire il metodo `GetResult` come generico rispetto al tipo degli stream ma al tempo stesso per avere

il controllo statico sulla distinzione tra quelli di input e quelli di output. In queste classi base sono fattorizzate le funzionalità di split ed è presente l'area dati, un `System.Array`.

```
public class InputStream<T> : BaseInputStream
{
    public InputStream(ICollection<T> collection)
    public InputStream(T[,] matrix)

    public T Current

    public T ElementAt(int x, int y)
}
```

La proprietà `Current` permette di accedere all'elemento correntemente calcolato dal kernel. Il modello d'esecuzione riprende quello della GPU. Grazie al metodo `ElementAt` è possibile sfruttare una feature di basso livello offerta dalle GPU, cioè la possibilità di comandare il *sample* (la lettura) di una risorsa ad una posizione diversa da quella dell'elemento del dominio d'esecuzione correntemente calcolato, laddove se ne conoscano con esattezza le coordinate (che non sono banalmente gli indici all'interno del dominio d'esecuzione).
Un esempio di utilizzo del metodo `ElementAt` è il calcolo efficiente della trasposta di una matrice:

```
public void Transpose(InputStream<int> a, OutputStream<int> b)
{
    b.Write(a.ElementAt(b.Y, b.X));
}
```

```
public class OutputStream<T> : BaseOutputStream, IEnumerable<T>
{
    public OutputStream(int size)

    public void Write(T item)

    public T[,] ToMatrix2D()
    public T[] ToArray()

    public int X
    public int Y

    public IEnumerator<T> GetEnumerator()
}
```

Tramite le proprietà `X` e `Y`, si accede alle coordinate all'interno del dominio d'esecuzione dell'elemento correntemente calcolato. Con il metodo `Write` si scrive il risultato nel dominio d'esecuzione.
E' esposto un enumeratore per scorrere i risultati calcolati dal kernel e due metodi per la conversione dello stream in una matrice bidimensionale (`ToMatrix2D`) o in un array (`ToArray`).

La possibilità di leggere/scrivere in posizioni arbitrarie delle risorse di input/output è esposta tramite le seguenti due classi:

```
public class ScatterStream<T> : OutputStream<T>
{
      public ScatterStream(InputStream<T> source)
      public ScatterStream(int size)

      public object SyncRoot

      public T this[int index]
      public T this[int x, int y]
}

public class GatherStream<T> : InputStream<T>
{
    public GatherStream(ICollection<T> source)
    public GatherStream(T[] data)

    public T this[int index]
    public T this[int x, int y]
}
```

I due indexer permettono di vedere lo stream come una matrice piuttosto che come un array ed effettuare scritture/letture usando uno o due indici. Si sarebbe potuto accorpare queste funzionalità direttamente all'interno dei tipi `InputStream` e `OuputStream`; la motivazione per cui questo non è stato fatto è rendere chiaro al programmatore che usando Scatter/Gather stream si andrà ad usare il buffer globale (che come spiegato nel capitolo 2, non ha scritture cached quindi offre minori prestazioni). Per questo motivo nel caso egli voglia realizzare calcoli di tipo *map*, nei quali il valore calcolato dipende solo ed esclusivamente dagli elementi correnti degli stream allora potrà avvantaggiarsi usando `InputStream` e `OutputStream` ottenendo prestazioni migliori. Nel caso in cui sia espressamente richiesto l'accesso a posizioni arbitrarie si può comunque farlo usando `GatherStream` e `ScatterStream`.
Ad esempio un caso in cui risulta utile poter accedere ad elementi in posizione arbitraria è la moltiplicazione di matrici:

```
public void Multiply(GatherStream<int> a, GatherStream<int> b,
                     OutputStream<int> c, int k)
{
    int total = 0;

    for (int j = 0; j < k; j++)
        total += a[c.Y, j] * b[j, c.X];

    c.Write(total);
}
```

Un’altra motivazione che ha condotto a questo design è il maggior numero di assunzioni che si possono fare per la gestione della sincronizzazione: utilizzando `GatherStream` non c’è alcun bisogno di sincronizzare perché sono permessi accessi in sola lettura; se al contrario si fossero accorpate le due funzionalità (dentro un’ipotetica classe `GlobalStream`) questa assunzione sarebbe caduta, complicando l’implementazione perché si sarebbe dovuta ricercare la presenza di scritture e, in caso affermativo, emettere istruzioni di sincronizzazione anche per ogni accesso in lettura (penalizzando le performance) al fine di evitare corse critiche.

Reduce

E’ possibile realizzare un calcolo di tipo *reduce* all’interno di un kernel utilizzando un oggetto che implementa la seguente interfaccia:

```
public interface IReduce<T>
{
    T Value { get; }

    void Sum(T a);

    void Mul(T a);

    void Max(T a);

    void Min(T a);
}
```

Ecco un paio di esempi di kernel che effettuano reduce:

```
public void ReduceKernel(InputStream<int> source, IReduce<int> r)
{
    r.Sum(source.Current);
}

public void ReduceKernel2(InputStream<int> source, IReduce<int> r)
{
    if (source.Current > 10)
    {
        r.Sum(5);
    }
}
```

La classe `Reduce` offre questo metodo per la creazione di oggetti `Reduce`:

```
public static IReduce<T> Create()
```

Un esempio di utilizzo è:

```
var r = Reduce<int>.Create();
```

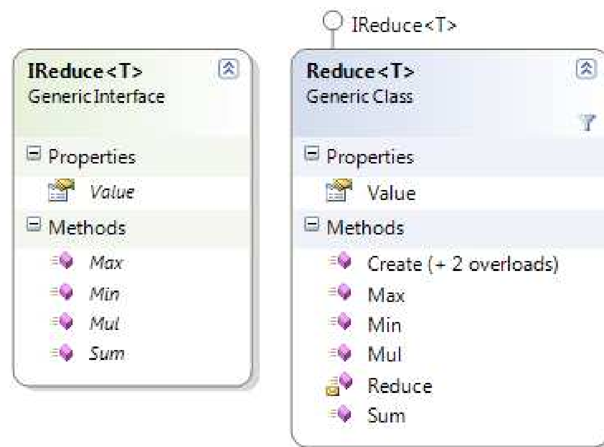


Figura 6.4, l'interfaccia IReduce<T> e la classe Reduce<T>

E' possibile effettuare reduce su tipi non primitivi, usando il seguente overload del metodo `Create`, che accetta un oggetto capace di effettuare calcoli sul tipo "custom" desiderato:

```
public static IReduce<T> Create(T value, ICalculator<T> calc)
```

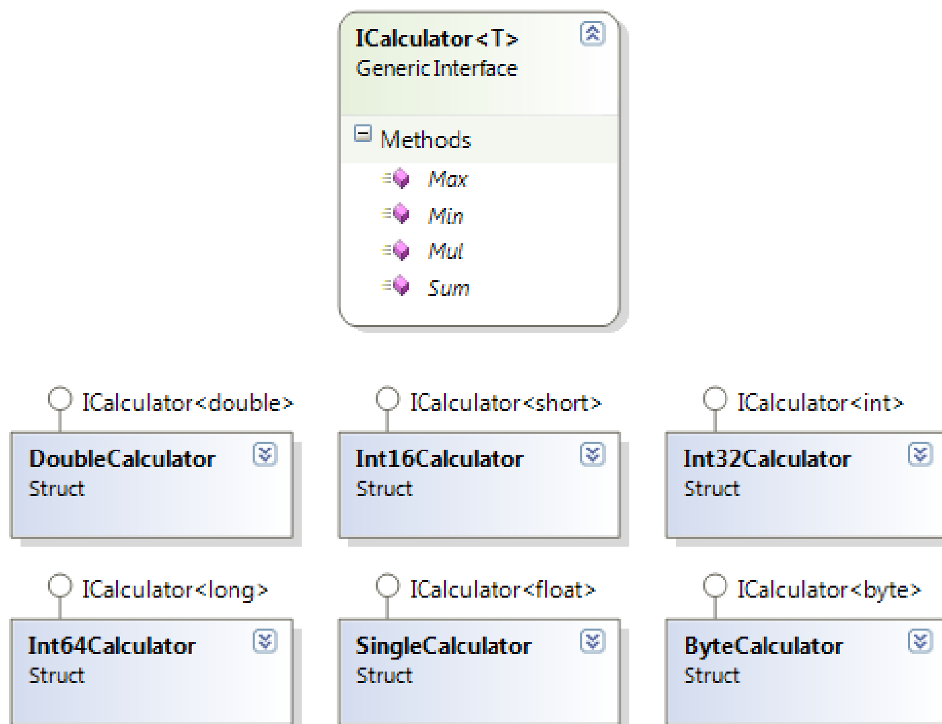


Figura 6.5, l'intefaccia ICalculator<T> e i diversi *calculator* forniti per default.

Il motivo dell'utilizzo di `ICalculator` è dovuto al fatto che non c'è una classe base per i tipi numerici o un'interfaccia che definisce operazioni aritmetiche su di essi all'interno del framework .NET. In caso contrario sarebbe stato possibile definire questa classe come (dove `Number` è la superclasse citata poc'anzi):

```
public class Reduce<T> where T : Number
{
    public T Value
    {
        get; set;
    }

    internal Reduce(T value)
    {
        Value = value;
    }

    public void Sum(T a)
    {
        Value += a
    }

    //...
}
```

Data la mancanza di questa classe e siccome gli operatori di somma, moltiplicazione, etc. non sono definiti su una generica classe T, è necessario delegare ad un'entità esterna (un'implementazione dell'interfaccia `ICalculator<T>`) il calcolo effettivo:

```
internal class Reduce<T> : IReduce<T>
{
    private ICalculator<T> _calc;

    public T Value
    {
        get; set;
    }

    internal Reduce(T value, ICalculator<T> calc)
    {
        Value = value;
        _calc = calc;
    }

    public void Sum(T a)
    {
        Value = _calc.Sum(Value, a);
    }

    //...
}
```

Ad esempio l'`ICalculator` per gli interi a 32-bit è cosi definito:

```
/// <summary>
/// Int32 calculator.
/// Provides the IReduce methods over int32 domain.
/// </summary>
struct Int32Calculator : ICalculator<int>
{
    #region ICalculator<int> Members

    public int Sum(int a, int b)
    {
        return a + b;
    }

    //...
}
```

Capitolo 7

Dopo aver presentato l'API verso il programmatore, passiamo ora a descrivere come è strutturato internamente PBricks. In questo capitolo si descriverà quello che avviene quando è richiesta l'esecuzione di una computazione in modalità sequenziale e multicore, dandone una descrizione accennata per quel che riguarda la fase di esecuzione per GPU. Lasciando ai prossimi due capitoli le descrizioni accurate.

7.1 PBricks internals

7.1.1 Architettura

Pricks è diviso concettualmente in tre parti:

1. Il modulo *Core*, che contiene la definizione di tutte le interfacce e classi astratte derivanti dall'analisi di Cuda e CAL e che dovrebbero generalizzare sufficientemente il modello di programmazione
2. Il modulo *ATI*, contenente la definizione delle classi concrete per tali interfacce e classi astratte. Questi tipi sono esposti al livello Core tramite un'architettura a Provider (essenzialmente una factory [4]).
3. Il modulo *CALInterop*, il layer d'interoperazione per poter far dialogare il modulo ATI con i driver di CAL.

Gli algoritmi per la fase di analisi sono realizzati nel modulo Core, in modo tale da essere indipendenti da una particolare scheda video, ed è concentrata buona parte del codice che descrive gli step necessari al lancio della computazione su una GPU, mentre nel modulo ATI è racchiusa la maggior parte del codice che effettua il setup del calcolo (come gli algoritmi per il partizionamento dei dati) e la realizzazione concreta degli step per l'avvio del calcolo. Nel modulo CALInterop oltre al wrapper per i driver CAL, sono definite routine di comodo che il modulo ATI utilizza frequentemente, come il codice per il trasferimento dei dati da CPU a GPU e viceversa.

Nella figura seguente sono riportate le dipendenze tra i moduli (assembly) di PBricks.

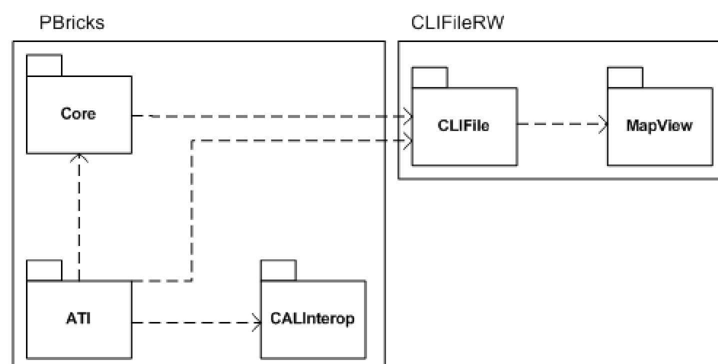


Figura 7.1, diagramma delle dipendenze tra gli assembly

7.2 La fase di esecuzione: sequenziale e multicore

Da quanto visto nel capitolo precedente l'uso di PBricks prevede due fasi:

1. Definizione dei kernel e strutturazione della computazione mediante la costruzione di uno o più PBricks (Application)
2. Lancio della computazione (tramite l'invocazione del metodo `GetResults`)

Della prima fase si è già discusso nel capitolo precedente. Ora concentriamoci sulla seconda fase.

All'invocazione del metodo `GetResults` vengono effettuati controlli di coerenza sul numero e tipo dei parametri passati: devono coincidere con quelli del kernel invocato. Questo controllo non è realizzabile staticamente per via del livello di indirezione. Siccome tutti i controlli sono comuni ai PBrick questi vengono effettuati nella superclasse e si delega l'esecuzione effettiva del calcolo alle classi derivate utilizzando il design pattern *Template Method* [4] di cui il metodo protetto `getResults` che si differenzia per ogni PBrick concreto, ne è l'implementazione:

```
protected abstract void getResults(BaseInputStream[] inStreams,
                                   BaseOutputStream[] outStreams,
                                   params object[] constants);
```

Descriviamo brevemente qua cosa avviene in sequenziale e multicore lasciando ai successivi due capitoli la descrizione del caso per GPU. Nel caso di `SeqPBrick` tramite iteratori sono scorsi gli stream di input elemento per elemento e tramite `Invoke` (Reflection) è chiamato il metodo che rappresenta il kernel passando i parametri (stream, costanti):

```
    while (inStreams.All(stream => stream.MoveNext()))
    {
        _method.Invoke(_state, args);
    }
```

Nel caso del `MulticorePBrick` è utilizzata la libreria TPL: sono creati un numero di `Task` pari al numero di core ed è effettuato il partizionamento dei dati. Per quanto visto in precedenza (paragrafo 5.6) si delega a TPL il compito di decidere il numero ottimale di thread da lanciare.

```
    var tasks = new TPLTask[cores];
    for (int i = 0; i < cores; i++)
    {
        tasks[i] =
            TPLTask.Create(o =>
            {
                var p = (TPLKernelParameters)o;
                var args = p.ToArray();

                while (p.Inputs.All(stream => stream.MoveNext()))
                {
                    _method.Invoke(_state, args);
                }
            }, parameters[i]);
    }
```

```
// Wait for all tasks to complete.
TPLTask.WaitAll(tasks);
```

E’ applicata un’ottimizzazione (*sharing*) legata alla presenza di memoria condivisa: ogni frammento di `OutputStream` condivide l'area dati dello stream dal quale è creato ma gli è permesso di scrivere solo nella propria porzione. Questo evita inutili copie risparmiando tempo e allocando un minor quantitativo di memoria. Il medesimo procedimento è applicato anche per partizionare gli stream di input sui quali ogni `Task` deve lavorare. Questo evita anche la fase di *merge* poiché ogni scrittura nel frammento dello stream di output di fatto corrisponderà ad una scrittura nello stream originale. Non è necessario gestire il caso di scritture sovrapposte (con relativa sincronizzazione) poiché le scritture nei frammenti non sono esposte al programmatore ma sono gestite dalla libreria che per costruzione (avvengono usando indici che per costruzione sono distinti per ogni frammento di stream) ne garantisce la correttezza.

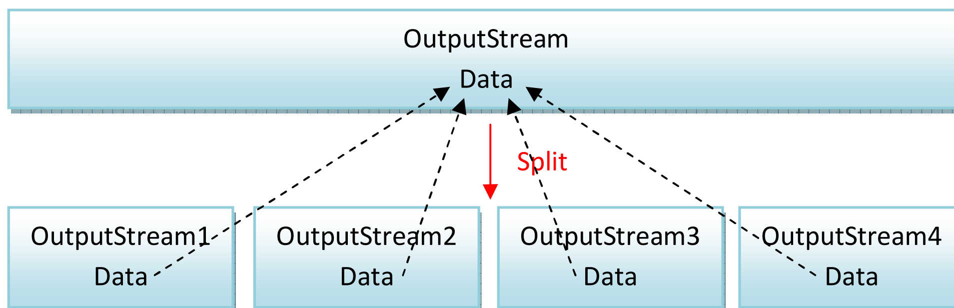


Figura 7.2, la fase di split

Nel caso di `GPUPBrick` gli step che PBricks compie quando viene invocato questo metodo sono:

- Analisi “meta” il cui risultato è una descrizione del kernel (`Kernel`)
- Esecuzione (`Scheduler`)

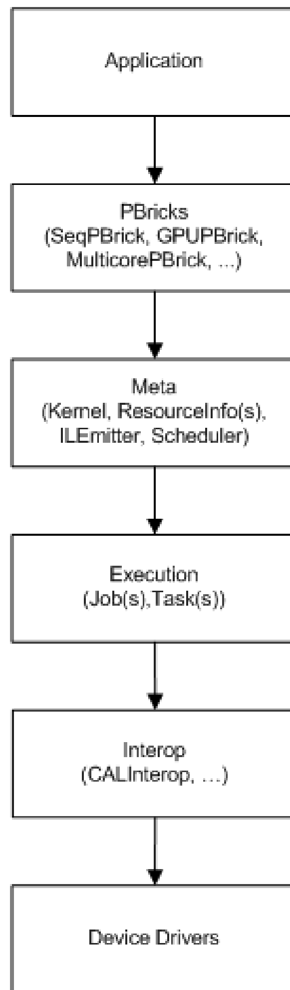


Figura 7.3, stratificazione a livelli e le entità coinvolte nell’esecuzione su GPU

Capitolo 8

In questo capito si descrive nei dettagli la fase di analisi, che si compone principalmente di due passi: l'analisi del metodo e l'analisi dei dati utilizzati. La gestione della memoria condivisa e del buffer globale verrà discussa al termine del capitolo.

8.1 Analisi

Per eseguire una computazione su GPU è necessario conoscere il sorgente espresso in linguaggio intermedio della GPU che ne descrive i passi, il quale verrà caricato al momento dell'esecuzione, e fornire i dati al dispositivo sui quali avverrà il calcolo.

Per la generazione del codice sorgente a basso livello è richiesta sia un'analisi delle istruzioni MSIL che costituiscono il programma ad alto livello sia del tipo di dati usati, perché ogni utilizzo di risorsa deve essere dichiarato esplicitamente nel sorgente. Per questo motivo l'analisi si divide in due parti: *analisi del metodo* e *analisi dei dati*. L'analisi del metodo ha come obiettivo quello di costruire una struttura dati che verrà attraversata in fase di generazione per emettere istruzioni del linguaggio target.

I dati utilizzati da un kernel prevedono: parametri di tipo stream, membri statici o d'istanza della classe che definisce il kernel, parametri di tipo primitivo, array di tipi primitivi e costanti utilizzate all'interno del metodo. Le informazioni raccolte sono conservate all'interno della classe `Kernel`. Essa terrà traccia sia del codice generato, sia delle risorse usate tramite degli opportuni *descrittori di risorse*. Sarà poi necessario collegare questi descrittori con i dati effettivi, per guidare la successiva fase di esecuzione (descritta nel prossimo capitolo) nella quale bisogna indicare con opportune chiamate a livello di driver che un determinato buffer di risorsa indentifica una certa risorsa dentro il sorgente del kernel.

Dovendo generare codice a registri, la classe `Kernel` s'incarica di tener traccia di quali registri sono impiegati. L'informazione sui registri è costituita da una tabella che associa un nome simbolico identificante una certa risorsa (sia essa un parametro stream, una variabile locale, una costante) col nome di registro che effettivamente sarà poi impiegato nel sorgente generato.

La fase di analisi che si svolge interamente dentro al modulo "Core" deve essere indipendente da una particolare architettura grafica (ATI/Nvidia). Per questo motivo sono state identificate delle interfacce e/o classi astratte e il codice dentro "Core" è accoppiato unicamente a queste. L'istanziazione dei componenti concreti specifici ad una architettura è delegata ai moduli delle architetture concrete (nel caso che si è implementato, nel modulo "ATI"). All'interno del singleton `PBricksSettings` è presente un riferimento a `Provider` (un'interfaccia). Tramite ques'interfaccia è possibile richiedere un componente specifico dopo aver istanziato il provider concreto (`ATIProvider`, `NvidiaProvider`, …)

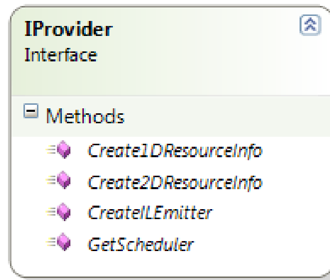


Figura 8.1, l'interfaccia IProvider

Le informazioni riguardanti i dati si esplicano nella costruzione di una o più istanze della classe `Resource`. Per poter collegare l'area dati degli stream con queste risorse, ogni risorsa possiede un riferimento ad un'istanza della classe `ResourceInfo` (descritta più approfonditamente nel successivo capitolo) tramite il campo `Data`. L'idea è che l'insieme d'informazioni raccolto nel meta-programma è fattorizzato all'interno di `Resource`, mentre `ResourceInfo` incapsula le informazioni specifiche di un determinato produttore, e legate all'API esposta da questo. In tal modo per realizzare il porting verso un'altra API (ad esempio CUDA di Nvidia) è necessario definire una classe derivata dalla sola `ResourceInfo`.

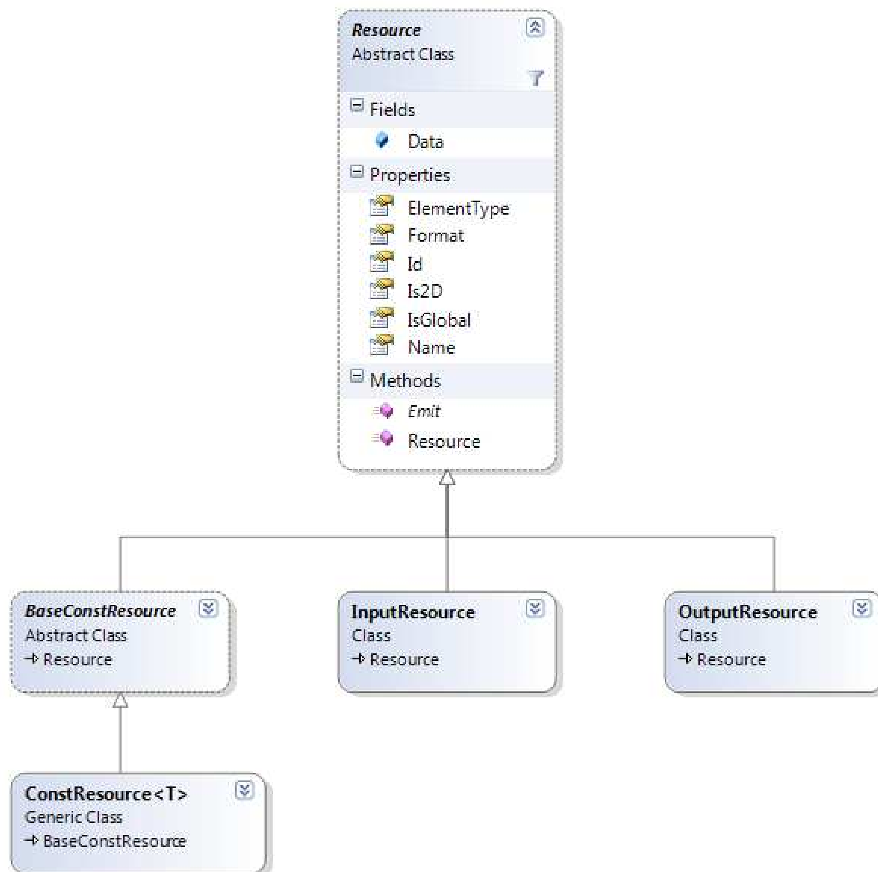


Figura 8.2, la gerarchia di classi Resource

8.1.1 L’analisi dei dati

L’analisi dei dati viene effettuata alla creazione della classe `Kernel` all’interno dei metodi:

- `trackConstantParameters`
- `evalMemoryUsage`
- `evalResources`
- `addStreams`

Per ogni parametro che non sia uno stream è allocata una risorsa, e ad essa viene assegnato un identificatore univoco. A questa viene riservato all’interno della tabella registri un nome simbolico `arg` seguito dall’identificatore della risorsa. E’ necessario conoscerne il tipo per dichiarlo all’interno del sorgente generato. La lunghezza non è ancora nota in questa fase, e questo è il motivo per cui successivamente bisogna collegare i dati con le risorse.

Nel metodo `evalMemoryUsage` si collezionano informazioni per la gestione della memoria condivisa. Di questo si parlerà più approfonditamente nel paragrafo 8.3.2.

Le costanti utilizzate all’interno del metodo sono divise per tipo, e per ognuno di questi è creato un buffer a sé: `evalResources` serve per tracciare tutti i valori costanti caricati sullo stack. Effettua una prima scansione sul sorgente MSIL e per ogni istruzione di tipo `ldc.X` (dove `X` può essere `i4`, `r4`, `r8`) invoca il metodo `addConstValue` che viene usato per aggiungere un singolo valore costante: utilizzando una tabella indicizzata per tipo (`int`, `float`, etc.) crea una nuova `ConstResource` se il tipo della costante non è trovato nelle risorse, altrimenti aggiunge il valore ad una risorsa già esistente. Al termine aggiunge all'array delle risorse (`_resources`) le `ConstResource` create.

Per ogni parametro `InputStream`, `OutputStream`, `Gather`/`ScatterStream` viene creata una risorsa, e viene riservato un nome simbolico all’interno della tabella registri per poter riferire questa risorsa all’interno del sorgente generato. Le risorse create vengono aggiunte alla lista di risorse del `Kernel`.

8.2 L'analisi del codice MSIL

La scopo di questa fase è effettuare la traduzione del programma di alto livello in uno semanticamente equivalente nel linguaggio target della GPU (in questo caso il CAL IL). Tutto questo avviene dentro al metodo `EvalBody` della classe `Kernel`. Questo compito è stato notevolmente complicato dalla mancanza d'istruzioni di salto di cui CAL IL è privo (come accennato nel paragrafo 3.1.2). E' stato necessario in un certo senso invertire il processo di compilazione per ricostruire i costrutti di alto livello. Partendo da un frammento di programma di alto livello, nel quale si esprime ad esempio la selezione:

```
if (current > 0)
{
    current--;
}
else
{
    current++;
}
```

L'MSIL generato è:

```
L_0007: ldloc.0
L_0008: ldc.i4.0
L_0009: ble.s L_0010

L_000b: ldloc.0
L_000c: ldc.i4.1
L_000d: sub
L_000e: stloc.0
L_000f: ret

L_0010: ldloc.0
L_0011: ldc.i4.1
L_0012: add
L_0013: stloc.0
L_0014: ret
```

Questo è il sorgente a "basso livello" che si deve generare:

```
ifc_relop(gt) r3, r2
  sub r4, r1, cb0[0].y
  mov r1, r4
  ret_dyn
else
  iadd r5, r1, cb0[0].y
  mov r1, r5
endif
```

Nel caso in cui le istruzioni di salto fossero state presenti sarebbe stato possibile effettuare una traduzione 1-a-1 tra le istruzioni del bytecode MSIL e quelle del CAL IL. In uno pseudolinguaggio dotato dei salti l'equivalente del precedente programma sarebbe stato:

```
jmp_cond(le) r3, r2, else

then:
   sub r4, r1, cb0[0].y
   mov r1, r4
   jmp end

else:
   iadd r5, r1, cb0[0].y
   mov r1, r5

end:
   …
```

In particolar modo un'istruzione di branch può essere gestita emettendo il corrispondente opcode e inserendo al momento opportuno la label necessaria ad indicare il target del salto. Questo procedimento non richiede altra informazione se non quella dell'opcode dell'istruzione correntemente analizzata. Tale gestione, pertanto, può essere definita "locale", in contrapposizione ad una gestione "non locale", che richiede cioè ulteriori informazioni inerenti la struttura del codice per determinare quale opcode emettere. Si consideri ad esempio un'istruzione di salto condizionato all'indietro: questa potrebbe essere usata per tradurre un costrutto *while* piuttosto che un *dowhile* e non c'è modo guardando solamente tale istruzione di discriminare tra le due possibilità.

Consideriamo questo frammento:

```
int x = 0;
int y = x + 10;
```

tradotto in MSIL come:

```
L_0001: ldc.i4.0
L_0002: stloc.0
L_0003: ldloc.0
L_0004: ldc.i4.s 10
L_0006: add
L_0007: stloc.1
```

L'idea alla base dell'algoritmo (che con alcune modifiche permette di gestire anche i casi cui si è accennato poc'anzi) è quella di costruire per ogni istruzione del linguaggio intermedio di partenza un *nodo*, e tenere traccia in questo nodo delle dipendenze in termini di dati. Queste sono usate per emettere il codice a registri in una fase successiva.
Viene cosi a crearsi una struttura gerarchica che modella il programma da tradurre; i nodi a livello root sono conservati all'interno di una lista (`kernelStms`) nel `Kernel`.

Considerando il precedente frammento si nota che l'istruzione di store (`stloc`) per scrivere nella variabile locale di indice 1 richiede un solo valore[37] dallo stack, il risultato della `add`, e a sua volta la `add` ha bisogno di prelevare dallo stack i propri due operandi, ossia la costante pushata dall'istruzione `ldc.i4.s` e il valore della variabile locale di indice 0, caricato sullo stack dalla `ldloc.0`.

Utilizzando una pila di nodi detta `currentStmStack` per tenere traccia dei nodi che aggiungono valori sulla pila degli operandi e che saranno poi inglobati all'interno di altri nodi, si può esprimere in pseudocodice l'algoritmo base per la traduzione da codice MSIL a CAL IL. Si noti come questa possa essere fatta con una sola scansione lineare del codice di partenza:

```
foreach (instr)
{
   newNode = createNode(instr, currentStmStack)

   currentStmStack.push(newNode)

   if (stackHeight = 0)
      newNode.Scan()
}
```

L'operazione `createNode` costruisce un nuovo nodo in base al tipo dell'istruzione esaminata prelevando un numero di nodi da `currentStmStack` pari al numero di operandi richiesti dall'istruzione. Quando il nodo corrispondente all'istruzione esaminata, prelevando i propri operandi dallo stack, ne azzera l'altezza, significa che ha a disposizione tutti i dati richiesti. E' quindi possibile prelevare il nodo da `currentStmStack` e farne la scansione, generando il frammento di codice per lo specifico statement (mediante il metodo `Scan`). Questo è quello che l'algoritmo fa nel caso in cui le istruzioni esaminate non includano istruzioni di salto.

Per ogni istruzione si costruisce un nodo istanza di una classe che deriva da `Node`, distinguendo tra nodi *foglia* (ossia che non hanno figli) come `NodeLdloc` (usato per tener traccia di una variabile locale acceduta in lettura), nodi con *singolo figlio* come `NodeStloc`, usato per tener traccia della scrittura su una variabile locale (in questo caso il nodo figlio è il sottoalbero dell'espressione che rappresenta il valore da assegnare alla variabile) e nodi con *figli multipli*, come nel caso di `NodeBinaryOp`, che rappresenta un'espressione binaria e che pertanto ha due nodi figli e di `NodeCall`, che modella una chiamata a metodo e per questo deve avere un numero variabile di figli pari al numero di argomenti del metodo chiamato.

[37] Per ogni istruzione MSIL è possibile conoscere grazie alla libreria CLIFileRW il numero di operazioni di pop e push sullo stack degli operandi che la sua esecuzione richiede.

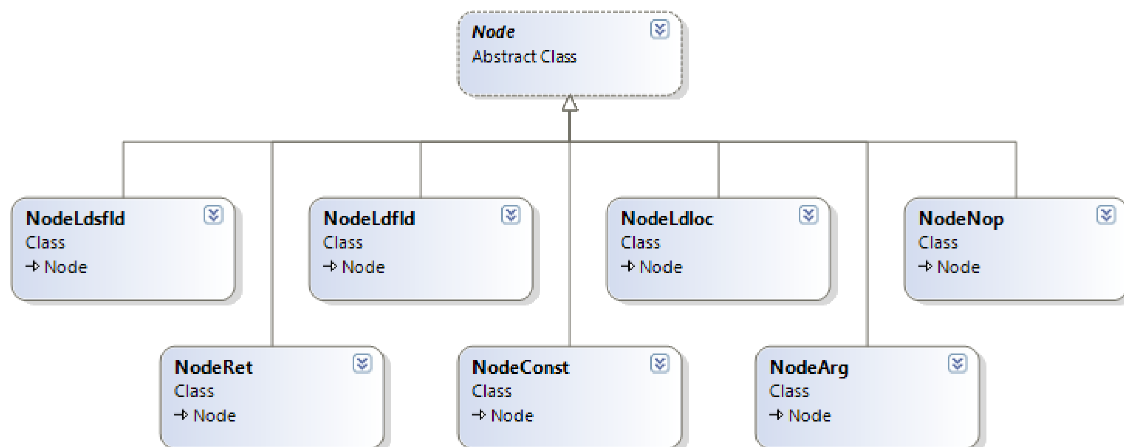


Figura 8.3, i nodi foglia

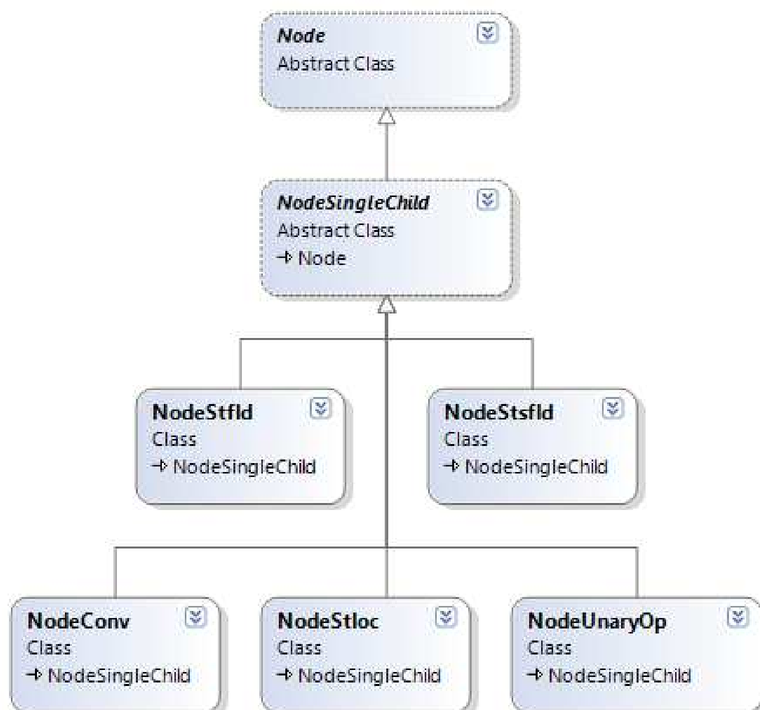


Figura 8.4, i nodi con un singolo figlio

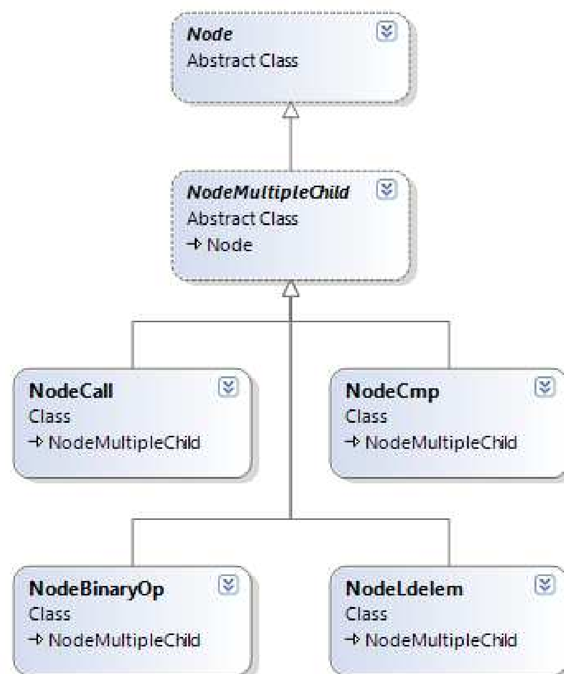


Figura 8.5, i nodi con più di un figlio

Per gestire anche le istruzioni di salto, considerando i vincoli dettati dal linguaggio target, è necessario capire quale costrutto ad alto livello esse individuino: una selezione, piuttosto che un'iterazione con controllo sull'uscita iniziale (*for*, *while*) o in coda (*do-while*) per i salti condizionati o l'inizio di un ramo *else* o il salto che porta alla valutazione della guardia di un ciclo while per ogni salto incondizionato. Da qui il bisogno di individuare un insieme di regole e proprietà per associare ad ogni istruzione di salto il corrispondente costrutto di alto livello. Queste proprietà si basano sulla struttura del sorgente MSIL, cioè su come si relazionano tra di loro gli opcode a livello di posizioni reciproche: ad esempio, le istruzioni successive ad un salto condizionato in avanti e precedenti il target di tale salto codificano il ramo "*then*" di un costrutto di selezione. Queste relazioni sono però influenzate dalla *modalità di compilazione* (debug o release), che fa si che il compilatore di alto livello generi codice intermedio con proprietà differenti per quel che riguarda i salti (si veda in appendice, la sezione A.5 per alcuni esempi di queste differenze). Questo ha complicato notevolmente lo sviluppo dell'algoritmo per la fase di analisi perché è stato necessario determinare condizioni che generalizzassero entrambe le modalità di compilazione.

Un'altra sorgente di problemi è stata la necessità di conservare informazioni circa gli annidamenti tra statement, poiché costrutti annidati nel sorgente ad alto livello si devono tradurre con corrispondenti costrutti annidati anche nel sorgente a basso livello. Per chiarire questo concetto si veda il seguente esempio:

```
public void KernelWithNestedWhileLoopAndIfThenElse(InputStream<int> source,
                                                    OutputStream<int> dest)
{
    int current = source.Current;
    int result = 0;

    if (result == 0)
    {
        int x = 8;
        dest.Write(x);
    }
    else
    {
        if (result > 0)
        {
            while (current > 0)
            {
                current /= 2;
                result++;
            }
        }
        else
        {
            int y = 7;
            dest.Write(y);
        }
    }

    dest.Write(result);
}
```

Nel corrispondente programma CAL IL è possibile notare come il livello di astrazione sia essenzialmente quello del programma di partenza, se si eccettua l'utilizzo dei registri anziché delle variabili:

```
if_logicalz r3.x
   mov r4, cb0[0].y
   mov o0, r4
else
   itof r5, cb0[0].x
   itof r6, r2
   ifc_relop(gt) r6, r5
      whileloop
         itof r7, cb0[0].x
         itof r8, r1
         breakc_relop(le) r8, r7.x
         udiv r9, r1, cb0[0].z
         mov r1, r9
         iadd r10, r2, cb0[0].w
         mov r2, r10
      endloop
   else
      mov r11, cb0[1].x
      mov o0, r11
```

```
    endif
endif
mov o0, r2
ret
end
```

La ricostruzione della gerarchia degli annidamenti tra costrutti (che è stato necessario desumere a partire da un linguaggio intermedio, di più basso livello) è stata la maggior sorgente di difficoltà nella fase di analisi. Questo requisito ha portato alla definizione della seguente gerarchia di classi per modellare le istruzioni MSIL per il controllo di flusso (salti condizionati / incondizionati):

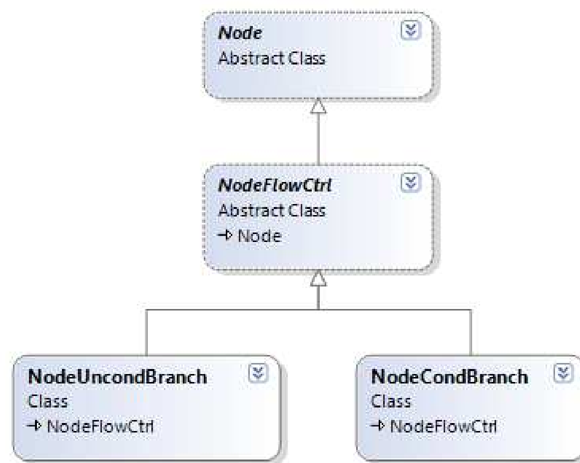


Figura 8.6, la gerarchia di classi per le istruzioni MSIL relative al controllo di flusso

Più nel dettaglio, a differenza degli altri nodi usati per modellare valori o operazioni, che erano essenzialmente indipendenti l'uno dall'altro (se si eccettua la dipendenza per tener traccia degli operandi sullo stack), per questo tipo d'istruzioni bisogna modellare anche la relazione gerarchica data dall'annidamento. Di fatto ogni nodo che eredita da `NodeCtrlFlow` può essere visto come la radice di un sotto-albero che modella la relazione di annidamento tra i blocchi di codice (struttura che è ricostruita applicando le proprietà trovate e di cui si è discusso prima).

```
public abstract class NodeFlowCtrl : Node
{
    private long _branchLabelId;

    protected List<BodyEntry> _parent;

    protected BranchType _branchType;

    protected List<BodyEntry> _target;

    protected List<BodyEntry> _seq;

    // ...
}
```

`BranchType` è un membro di tipo enumerazione che può assumere i valori `{ If, Else, While, DoWhile }`, `BodyEntry` è una coppia costituita da un'istanza della classe `Node` e dalla posizione che l'istruzione ha all'interno del sorgente. I nodi per i salti condizionati (`NodeCondBranch`) necessitano di tener traccia anche dell'opcode MSIL della condizione, e dei nodi (uno o due) per emettere il codice atto a valutare la condizione della guardia.

Per effettuare la generazione del codice è richiesta una ben precisa struttura che non può essere garantita con una sola passata del codice intermedio MSIL. E' necessaria una fase di *Fix* (metodi `Fix` e `FixDoWhile`) per sistemare questo problema: utilizzando il riferimento alla lista degli statement del nodo padre, `_parent`, presente nei `NodeCtrlFlow`, vengono spostati nodi.

Successivamente alla fase di Fix la struttura dei nodi è la seguente:

- Ogni nodo `NodeCondBranch` che rappresenta una selezione contiene all'interno di `Target` gli statement del ramo "else" e all'interno di `Seq` gli statement di un eventuale ramo "then".
- Ogni nodo `NodeCondBranch` che rappresenta un'iterazione di tipo while/dowhile contiene in `Target` la lista degli statement che rappresentano il corpo del ciclo.
- Nodi relativi a statement dello stesso blocco sono presenti o nella lista di statement a livello root, oppure in una di quelle dei `NodeCtrlFlow`.

Consideriamo il seguente esempio per chiarire questi concetti:

```
if (source.Current > 0)
    dest.Write(1);
else
    dest.Write(-1);
```

il cui corrispondente sorgente MSIL è:

```
L_0000: ldarg.1
     // preparazione guardia
L_0001: callvirt instance !0
            [Core]PBricks.Core.InputStream`1<int32>::get_Current()
L_0006: ldc.i4.0
L_0007: ble.s L_0011
     // ramo "then"
L_0009: ldarg.2
L_000a: ldc.i4.1
L_000b: callvirt instance void
            [Core]PBricks.Core.OutputStream`1<int32>::Write(!0)
L_0010: ret
     // ramo "else"
L_0011: ldarg.2
L_0012: ldc.i4.m1
L_0013: callvirt instance void
            [Core]PBricks.Core.OutputStream`1<int32>::Write(!0)
L_0018: ret
```

Dopo la creazione dei nodi per le istruzioni di valutazione della guardia (indicate in azzurro), che sono inglobati all'interno del `NodeCondBranch` relativo all'istruzione di salto di etichetta `L_0007`, si effettua una chiamata ricorsiva (in modo tale da poter riprendere la scansione da dove si era interrotto prima del salto), andando ad analizzare le istruzioni a partire da quella target del salto (in rosso).

E' necessario aggiungere una pila di altezze di stack per tener traccia del punto in cui si sospende l'analisi poiché dopo ogni salto l'altezza dello stack viene assunta essere 0 (cioè il codice MSIL salta in posizioni in cui lo stack è alto 0), ma al momento del salto sullo stack degli operandi potrebbero esservi uno o più valori. Se cosi non si facesse si correrebbe il rischio di non generare alcuni nodi terminato il passo ricorsivo. Al momento del ritorno dalla chiamata ricorsiva (dopo la valutazione della seconda istruzione `ret` in marrone) viene fatto *pop* su questa pila per ripristinare lo stato dell'analisi a cui si era giunti. Gli statement del ramo "else" sono ora presenti nella lista `Target` del `NodeCondBranch`.

Dopo aver effettuato il salto (e al ritorno) si prosegue ad analizzare la successiva istruzione MSIL. I nodi delle successive istruzioni sono inseriti nella lista `Seq` del `NodeCondBranch`. Per evitare di analizzare istruzioni già considerate si è previsto l'utilizzo di una tabella in cui salvare le posizioni già visitate.

Viene effettuato backtracking:

1. quando si sono visitate tutte le istruzioni (il cursore segnala EOF);
2. l'istruzione corrente è già stata visitata.

Nell'esempio considerato il secondo bracktracking avviene sull'istruzione `ldarg.2`, perché già visitata alla prima chiamata ricorsiva. Non essendovi più chiamate pendenti l'algoritmo di analisi termina.

Per poter decidere il `branchType` di ogni nodo `NodeCtrlFlow` si verifica se il salto è in avanti; in caso affermativo si tratta sicuramente di un nodo di tipo "if", altrimenti si utilizza una tabella costruita con una fase di preprocessing (descritta più in dettaglio nella sezione A.5 dell'appendice) che tiene traccia di quali istruzioni di salto traducono cicli while (e di conseguenza individua anche quali istruzioni - perché non presenti - traducono i salti per i cicli dowhile). Nel caso di salti incondizionati si controlla anche in questo caso una struttura dati precedentemente costruita per sapere se l'istruzione è relativa al salto che indica la fine di un ramo "else" piuttosto che quella che porta alla valutazione della guardia di un ciclo "while". Si consulti la sezione A.7 dell'appendice per un esempio più complesso di struttura finale ottenuta dall'algoritmo di analisi.

Riassumendo l'algoritmo di analisi di un metodo prevede quattro fasi:

1. la fase di preprocessing, nella quale sono raccolte informazioni sui salti incondizionati / condizionati all'indietro;
2. sono costruiti i nodi dell'albero di sintassi astratta che descrive il programma;
3. si applicano due passi di `Fix` per riorganizzare la struttura dei nodi in un modo che sia quello atteso per la fase di scan;
4. viene effettuata la scansione della foresta di nodi (detta fase di *Scan*) ottenuta al termine dei due precedenti passi. E' in questa fase che è generato il sorgente finale.

L'ultima fase è effettuata invocando il metodo `Scan` sui nodi presenti nella lista di statement a livello root. Questo metodo è definito come virtuale nella superclasse `Node`:

```
internal virtual string Scan()
{
    return string.Empty;
}
```

Ogni nodo ha la possibilità di ridefinirlo per gestire in modo opportuno la generazione del codice. La stringa restituita rappresenta il nome del registro usato dal particolare nodo. I nodi foglia allocano nuovi registri o riutilizzano registri esistenti nel caso in cui referenzino dati già allocati. Ad esempio `NodeConst` si limita a restituire al chiamante il registro che gli viene passato all'atto della creazione:

```
public class NodeConst : Node
{
    // ...

    internal override string Scan()
    {
        return _regName;
    }
}
```

`NodeLdloc` restituisce il registro verso il quale è effettuata la lettura, associando un nuovo registro al nome simbolico passato alla costruzione se questo non era ancora stato inserito nella tabella registri:

```
public class NodeLdloc : Node
{
    // ...

    internal override string Scan()
    {
        if (!_regs.ContainsKey(_regName))
        {
            _regs.Add(_regName, SymTable.GetNextRegister(_opdType));
        }

        return _regs[_regName];
    }
```

I nodi che posseggono figli, come `NodeBinaryOp`, invocano ricorsivamente su questi la scansione determinando cosi i registri usati in lettura (gli operandi), e restituendo il registro nel quale l'operazione va a scrivere:

```
internal override string Scan()
{
    string arg1 = _nodes[0].Scan();
    string arg2 = _nodes[1].Scan();
    string dest = SymTable.GetNextRegister(_opdType);

    _opdType =
        _nodes[0].OperandType > _nodes[1].OperandType ?
```

```
                    _nodes[0].OperandType :
                    _nodes[1].OperandType;

            _emitter.EmitBinaryOp(_opCode, _opdType, dest, arg2, arg1);

            return dest;
        }
```

In previsione dell'introduzione di registri tipati nel CAL IL si fa uso nella fase di `Scan` di un `SymbolTable` che fornisce il metodo `GetNextRegister` il quale prende in ingresso un `TypeCode` (`int32`, `single`, `double`, etc.) e restituisce la stringa col nome del prossimo registro disponibile per quel particolare tipo di dato.

Nel caso dei nodi che modellano istruzioni per il controlli di flusso (`NodeCtrlFlow`) avviene una generazione diversa a seconda che del `branchType` del nodo:

- `if`, si generano le istruzioni per il controllo della guardia, poi quelle relative ai nodi contenuti in `Seq` e poi in `Target`, se questo contiene almeno un nodo;
- `while`, si generano le istruzioni per il controllo della guardia del ciclo, poi quelle relative ai nodi contenuti in `Target;`
- `dowhile`, si generano le istruzioni relative ai nodi contenuti in `Target` e poi quelle per il controllo della guardia del ciclo.

La fase di scansione invoca metodi dell'interfaccia `ILEmitter`, di cui il `Kernel` tramite il `Provider` ne possiede un'istanza concreta per emettere gli opcode del linguaggio target desiderato. Ad esempio sono presenti `EmitMove` per generare un'istruzione move, `EmitBinaryOp` per generare un'operazione binaria, `EmitSynch` per emettere l'istruzione di sincronizzazione del linguaggio target e cosi via.

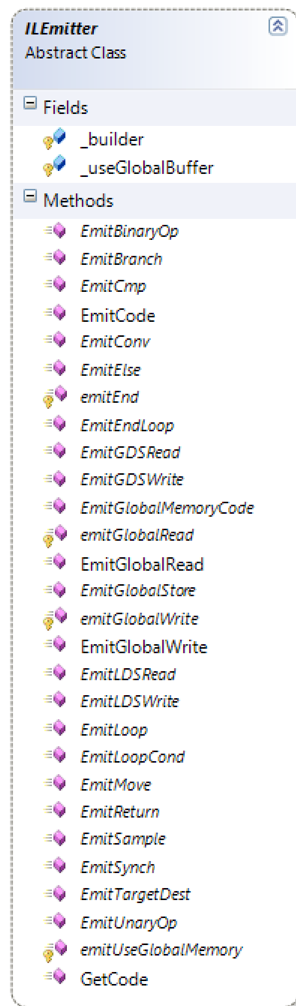


Figura 8.7, l'interfaccia di ILEmitter

8.3 Gestione della memoria: Global buffer, LDS, GDS

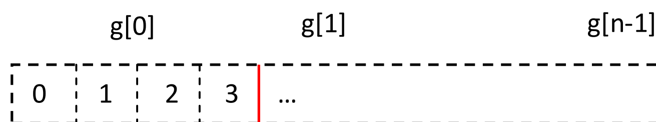
Una trattazione a parte meritano la gestione del global buffer (esposto tramite le classi `Scatter/GatherStream`) e delle gerarchie di memoria condivisa sulla GPU. Ripetiamo brevemente che l'area di memoria condivisa tra thread appartenenti ad uno stesso gruppo è esposta mediante letture e scritture sui campi d'istanza della classe in cui sono definiti i kernel e che quella tra thread di gruppi diversi è manipolabile mediante letture e scritture sui campi statici.

8.3.1 Global Buffer

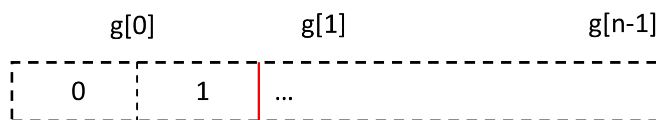
L'informazione relativa all'utilizzo o meno di `Gather/ScatterStream` è raccolta dentro al metodo `addStreams` alla creazione del `Kernel`. In questo caso sono sempre creati dei descrittori di risorse di tipo `Input/OutputResource` ma con un flag si marca il descrittore in modo tale che si possa in seguito distinguerlo da quelli per gli stream di input e di output. Questo verrà usato successivamente dal codice di allocazione delle risorse in fase di esecuzione per allocarlo in una diversa area di memoria.

Il global buffer di ATI, accedibile tramite il registro `g`, usa un allineamento a 128-bit. In fase di trasferimento dati da/alla GPU, la funzione driver tiene conto di questo fattore e all'interno di una stessa cella di memoria del buffer globale inserisce un numero diverso di elementi in base al formato dei dati trasferiti.

Nel caso di interi/float a 32 bit:



Nel caso di interi/double a 64 bit:



Questo implica che letture e scritture (effettuate all'interno del kernel generato) per questo buffer debbano prendere in considerazione sia il formato dei dati, sia il numero di componenti, generando non solo un indice all'interno di questo registro per riferire una precisa cella, ma anche uno spiazzamento relativo alla singola cella (realizzato tramite maschere). Ignorare lo spiazzamento provocherebbe un comportamento inatteso, perché si riferirebbero dati sbagliati. Questo problema non si presenta con le letture/scritture nei registri generali (`r`), dove al massimo vi è uno spreco di memoria utilizzando un registro da 128-bit per contenere numeri a 32-bit. Infatti letture e scritture nei registri coinvolgono solamente la GPU laddove con il global buffer c'è il bisogno di conciliare due diverse modalità di

riferimento ai dati: quella su macchina virtuale, che rimane invariata cambiando il tipo di dati e quella a basso livello che invece varia. Ad esempio, si consideri il seguente kernel:

```
public void KernelWithGatherStream(GatherStream<int> source,
                                   OutputStream<int> dest)
{
    int a = source[0];
    int b = source[1];
    int c = source.Current;
    int d = source[2, 2];

    dest.Write(a + b + c + d);
}
```

Modificando il tipo di `source` (da `int` a `double`) non c'è bisogno di cambiare il modo con cui si riferiscono i singoli elementi: questo avviene tramite l'astrazione ad alto livello dell'array grazie al quale ogni elemento corrisponderà sempre ad una singola cella dell'array. A livello di GPU, invece, all'interno della prima locazione del global buffer, essendo il formato dei dati int a 32-bit saranno allocati 4 numeri interi, per cui ci sarà la necessità di leggere con offset diversi all'interno di questa locazione. Sia `a` che `b` stanno dentro la stessa locazione ma devono essere riferite:

```
g[0].x, g[0].y
```

Nel caso di `double` invece, essendo questo un formato a 64-bit, i dati relativi ad `a` e `b` saranno riferiti come:

```
g[0].xy, g[0].zw
```

Da qui la necessità di generare un sorgente del kernel diverso a seconda che si vada ad operare con dati da 128, 64 o 32 bit.
Tuttavia durante la fase di scansione (nel metodo `evalIL` della classe `Kernel`) non si ha ancora conoscenza sul formato dei dati, perché come si è visto in precedenza il collegamento dei descrittori di risorse con i dati avviene solo successivamente. Perciò durante questa fase, quando s'incontrano chiamate a metodo per effettuare una lettura da `GatherStream` o scrittura su `ScatterStream` (metodo `Scan` della classe `NodeCall`), non viene generato direttamente in codice finale, ma sono posti dei *placeholder*, che verrano sostituiti solamente quando ci sarà l'informazione necessaria.

Più dettagliatamente quello che avviene durante la scansione di un `NodeCall` per quel che riguarda la gestione del global buffer è che nel caso di utilizzo del metodo `Write`, il global buffer è riferito con indice uguale all'elemento corrente (contenuto nel registro `vObjIndex0.x`):

```
_emitter.EmitGlobalWrite("vObjIndex0.x", item);
```

dove `item` è il registro che contiene il dato che si vuole scrivere.

Nel caso di `get_Current`:

```
_emitter.EmitGlobalRead("vObjIndex0.x", dest);
```

dove `dest` è il registro nel quale verrà copiato il dato letto dal buffer globale.

La chiamata all'indexer `set_Item` a due dimensioni (ad esempio `source[2,2]`) deve produrre un indice unico visto che nel CAL IL `g` è indirizzato con un numero solamente:

```
    index = SymTable.GetNextRegister(TypeCode.Int32);

    _emitter.EmitBinaryOp(OPCODES.Mul, TypeCode.Int32, index, x, y);
    _emitter.EmitGlobalWrite(index, item);
```

dove `item` è il registro che contiene il dato che si vuole scrivere.

Viene emessa una moltiplicazione tra i due indici x e y e poi il registro contenente il risultato (`index`) viene usato per indicare l'offset all'interno del global buffer. Nel caso di indexer a una dimensione questo non avviene.

Lo stesso avviene per l'indexer `get_Item`:

```
    string index = SymTable.GetNextRegister(TypeCode.Int32);

    _emitter.EmitBinaryOp(OPCODES.Mul, TypeCode.Int32, index + ".x", x, y);
    _emitter.EmitGlobalRead(index, dest);
```

I metodi `EmitGlobalRead` e `EmitGlobalWrite` inseriscono i placeholder all'interno del sorgente, indicando però l'indice col quale accedere al buffer e rispettivamente il registro destinazione della lettura e quello sorgente per la scrittura:

```
protected override void emitGlobalRead(string index, string dest)
{
    _builder.AppendLine(string.Format("GATHER({0},{1})", index, dest));
}

protected override void emitGlobalWrite(string index, string src)
{
    _builder.AppendLine(string.Format("SCATTER({0},{1})", index, src));
}
```

All'interno del PBrick, dopo aver collegato ogni Resource con il proprio ResourceInfo (e quindi note le componenti dell'area dati di ogni risorsa), si può completare la generazione del codice IL del kernel. Questa è detta fase di *finalizzazione* del codice, e avviene invocando il metodo FinalizeCode:

```
/// <summary>
/// Completes code generation emitting global buffer manipulation
/// (scatter/gather) code.
/// </summary>
internal void FinalizeCode()
{
    _resources
        .FindAll(r => r.IsGlobal)
        .ForEach(r => _emitter.EmitGlobalMemoryCode(r.Name,
                                              (uint)r.Data.ComponentSize));
}
```

Attualmente vi è una limitazione ad un solo global buffer (c'è un solo identificatore disponibile "g" per referenziarlo). Questo limite potrebbe essere presto superato, per cui si può pensare che vengano introdotti più registri per identificare risorse globali ("g1", "g2", etc.). Per questo motivo il codice che gestisce il global buffer non può che essere indipendente da un particolare identificatore, per cui questo è passato come parametro al metodo ricavandolo dalla proprietà Name di Resource. E' necessario inoltre passare come parametro anche il numero di componenti di una risorsa poiché ad ognuno di questi buffer potrebbe corrispondere un diverso numero di componenti.
Per ridurre la quantità di codice sorgente generato si emettono particolari funzioni IL da noi definite per leggere e scrivere su global buffer a seconda che si stia operando con dati a 64-bit o 32-bit. Queste sono riportate in appendice (sezione A.3). All'interno di EmitGlobalMemoryCode, usando delle espressioni regolari (tramite la classe RegEx), sono ricercati i placeholder e sostituiti con chiamate a tali funzioni. Nel caso a 128-bit (non ancora supportati perché il type system .NET non fornisce nativamente un tipo di dato a 128-bit) non c'è bisogno di fare calcoli per determinare la corretta locazione, per cui si può sostiture con una semplice mov. In tutti gli altri casi, in base al numero di componenti viene emessa un'istruzione di call verso la specifica funzione, aggiungendo istruzioni di mov per effettuare il passaggio di parametri, che ricordiamo nel CAL IL avviene tramite registri.

Nel caso di lettura (gather) deve essere emessa una mov aggiuntiva dopo la call per gestire il passaggio del risultato tramite parametro. Il registro destinazione è noto, mentre quello sorgente dipenderà dal registro usato dalla particolare funzione. Anche in questo caso si usa un placeholder inserito in fase di scansione dal NodeStloc:

```
public override void EmitGlobalStore(string dest)
{
    _builder.AppendLine(string.Format("GLOBAL_STORE({0})", dest));
}
```

Questo placeholder in fase di finalizzazione verrà trasformato in una istruzione mov avente come registro target quello riferito nel proseguio del programma, e come registro sorgente quello che la particolare funzione di gather chiamata usa per salvare il valore letto dal buffer.

8.3.2 LDS & GDS

Le problematiche inerenti all'uso della memoria condivisa (sia essa locale o globale) riguardano la possibilità di utilizzare solo tipi primitivi e array di tipi primitivi, e la necessità di stabilire un mapping tra il layout a livello di macchina virtuale e quello a livello di memoria su GPU (mediante l'utilizzo di offset e maschere), in modo simile a quanto detto in precedenza per l'utilizzo del buffer globale. Essendo questa un'area soggetta ad accessi concorrenti è necessario garantire una corretta sincronizzazione.

Durante la creazione di un'istanza della classe `Kernel`, all'interno del metodo `evalMemoryUsage` sono cercati tutti i membri d'istanza marcati con il custom attribute `ComputeField`, perché tra questi devono figurare solo tipi primitivi. Mediante l'utilizzo di `ComputeField` si da la possibilità alla classe che definisce la computazione di possedere anche campi di tipo non primitivo a patto che non siano utilizzati per esprimere lo stato di una computazione, e che quindi non siano riferiti all'interno di un metodo kernel. La classe `Kernel` tiene traccia del `TypeCode` di questi campi e dell'occupazione totale di memoria perché potrebbe verificarsi il caso in cui l'ammontare di memoria richiesta per i dati condivisi sia maggiore di quella supportata dalla particolare scheda video. E' sollevata eccezione nel caso in cui non si rispettino i vincoli imposti dall'esecuzione su GPU.

Per ognuno di questi campi è costruita un'istanza della classe `ComputeFieldInfo`:

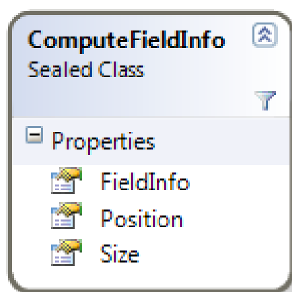


Figura 8.8, la classe ComputeFieldInfo

dove `Size` è la dimensione in byte del campo, `Position` è l'indice relativo alla posizione con la quale il campo figura nella definizione della classe (limitatamente ai soli `ComputeField`), e `FieldInfo` è l'oggetto di reflection che descrive il campo.

In fase di analisi del metodo, i nodi concreti che descrivono operazioni di lettura/scrittura su memoria LDS, (o GDS) sono `NodeStfld`, `NodeStsfld`, `NodeLdfld`, `NodeLdsfld`. Il `Kernel` possiede due flag `UseSharedMemory`, `UseGlobalMemory` che vengono settati al primo utilizzo in lettura o scrittura della memoria condivisa locale (o globale). Questo per poter successivamente andare a modificare l'header del sorgente generato. Infatti per poter usare LDS/GDS è necessario dichiarare un programma CAL IL di tipo *compute shader* anziché *pixel shader*, ed inoltre la capacità "LDS" (o "GDS") deve essere supportata dal device. Il frammento aggiunto all'header è il seguente:

```
il_cs_2_0
dcl_num_thread_per_group 64
dcl_lds_size_per_thread + k.FieldsMemUsage
dcl_lds_sharing_mode _wavefrontAbs
```

dove `k.FieldsMemUsage` indica il totale della memoria utilizzata per i field locali.

In fase di scansione (`Scan`) nei nodi `NodeStfld`, `NodeStsfld`, `NodeLdfld`, `NodeLdsfld` sono chiamati i seguenti metodi dell'`ILEmitter` per gestire scritture/letture nei due livelli di memoria:

- `EmitLDSWrite`
- `EmitLDSRead`
- `EmitGDSWrite`
- `EmitGDSRead`

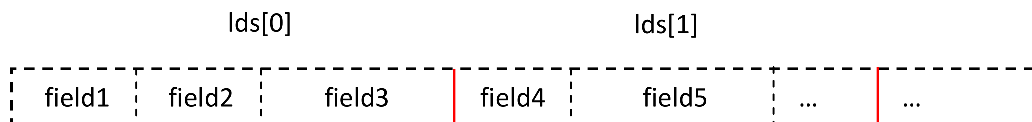
Dopo ogni emissione d'istruzione di scrittura è emessa anche un'istruzione di sincronizzazione (metodo `EmitSynch`) per garantire che tutte le scritture siano state completate e visibili agli altri thread e non possa essere effettuato il riordino delle letture/scritture al di là di questa istruzione.

Per ogni `ComputeField` è necessario conoscere l'offset rispetto all'indirizzo base del buffer condiviso e la maschera con cui sarà referenziato nelle operazioni di lettura/scrittura all'interno del programma IL generato.

Si prenda in considerazione il seguente caso:

```
class MyComputation
{
    [ComputeField]
    int field1;   // 32bit

    [ComputeField]
    int field2;   // 32bit

    [ComputeField]
    long field3; // 64bit

    [ComputeField]
    float field4; // 32bit

    [ComputeField]
    double field5; // 64bit

    //..
}
```

Il layout all'interno della memoria condivisa locale sarà:



Field name	Offset	Mask
field1	0	x
field2	0	y
field3	0	zw
field4	4	x
field5	4	yz

Tabella 8.1, il mapping tra i ComputeField e offset e mask

Queste informazioni devono essere note ai nodi perchè devono generare istruzioni IL del tipo:

```
lds_write_vec _lOffset(0/4/../60) mem.MASK src
```

```
lds_read_vec dest, src (src dipende dall'offset)
```

Nella fase di analisi, sono passati l'indice all'interno dei campi locali della classe e il riferimento all'intero array dei `ComputeFieldInfo` ai nodi `NodeLdfld`, `NodeLdsfld`, `NodeStfld` e `NodeStsfld` in modo tale che questi, in fase di `Scan`, siano in grado di passare all'`ILEmitter` le informazioni sufficienti per generare offset e maschera. Gli offset e le maschere sono contenuti in 3 array (`_offsets`, `_masks`, `_offsetsTable` - gli offset espressi in relazione a costanti letterali) che vengono calcolati alla prima operazione di lettura o scrittura. I dettagli per il calcolo dell'offset e delle maschere sono contenuti nel metodo `calcOffsetsAndMasks` che richiama al suo interno i metodi `calcOffsetTable` e `emitOffsets`. Basti dire che il calcolo dell'offset di un `ComputeField` dipende dalla dimensione della singola cella di memoria (attualmente 128-bit) e dai `ComputeField` che lo precedono (oltre che dalla loro dimensione in byte); per la maschere invece solamente dai `ComputeField` aventi stesso offset che precedono il campo preso in considerazione.

Nel caso di scritture è emessa la seguente istruzione:

```
_builder.AppendLine(
    string.Format("lds_write_vec _lOffset({0}) mem.{1} {2}",
        _offsets[index], _masks[index], src));
```

Nel caso di letture è necessario comunicare l'id del thread che effettua l'operazione (`vTid0`) e il suo offset[38]:

```
EmitMove(string.Format("{0}.x", src), "vTid0.x0");
EmitMove(string.Format("{0}._y", src), _offsetsTable[index]);

_builder.AppendLine(string.Format("lds_read_vec {0} {1}", dest, src));
```

La condivisione tra SIMD, effettuata tramite memoria globale, benchè prevista dall'architettura hardware non è esposta a livello di API, per cui la fase di analisi effettua questa gestione, ma mancando le istruzioni per realizzarla effettivamente, viene attualmente sollevata eccezione durante l'emissione del codice.

[38] Espresso sotto forma di costante letterale

Capitolo 9

In questo capitolo si descriverà la gestione della fase di esecuzione di un kernel, dei passi necessari a lanciare una computazione e di come questi siano stati astratti nello *schedulatore* mediante i concetti di *job* e *task*; si accennerà brevemente al supporto *multidevice* e all'algoritmo di partizionamento per suddividere il carico di lavoro. Infine verrà descritto il layer di interoperazione che costituisce i livello più basso di PBricks.

9.1 Esecuzione

Dopo la fase di analisi il codice del metodo kernel che si desidera eseguire è stato compilato nel linguaggio IL target e i dati degli stream e/o risorse costanti sono stati collegati nei descrittori di risorse. Ricordiamo brevemente i passi che è necessario compiere per lanciare una computazione su GPU.

Per quel che riguarda CAL:

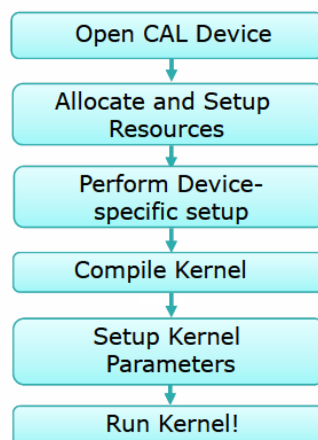


Figura 9.1, i passi necessari ad avviare una computazione GPU, usando CAL

Come riportato in 3.1.3 (e in sezione A.2 per un esempio di programma) i passi per CUDA sono:

- Inizializzare l'ambiente CUDA.
- Allocare la memoria.
- Preparare i dati di input in memoria.
- Eseguire il kernel.

Queste due sequenze di operazioni, molto simili tra loro, sono state astratte e vengono gestite dallo `Scheduler`. Questa è l'entità che s'interfaccia con il livello driver CAL (nel nostro caso), e utilizzando i risultati della fase di analisi (sorgente, descrittori di risorse con relativi buffer di dati lato CPU), si fa carico di avviare la computazione:

- gestendo le allocazioni di risorse su GPU;
- effettuando il trasferimento di dati;

- applicando il partizionamento per sfruttare schede in modalità SLI/CrossFire, oppure nel caso in cui non sia possibile allocare interamente i dati nella memoria GPU.

`Scheduler` è una classe astratta; tramite il `Provider` si chiede un'istanza dello scheduler. Nello specifico è stato realizzato `l'ATI_Scheduler`. Il `GPUPBrick` richiede l'esecuzione di un kernel invocando il metodo `Run` (che richiede un `Kernel` come parametro) che è il solo metodo publico esposto da questa classe.

Prima di poter eseguire la computazione bisogna però aggiungere un'intestazione al sorgente IL ottenuto al passo precedente: nell'header del programma se ne specifica il tipo (pixel shader piuttosto che compute shader), viene dichiarato un registro di input e nel caso si faccia uso di memoria condivisa o globale vengono aggiunte ulteriori informazioni: numero di thread per gruppo, dimensione della memoria condivisa per thread, modalità di condivisione della memoria. Viene inoltre emesso il codice per la dichiarazione delle risorse: per far ciò si utilizza un `IEmitter` (in questo caso l'emitter delle risorse), anche questo implementato mediante l'utilizzo del design pattern visitor [4].

Il metodo `createEmitter` è il factory method invocato alla creazione dello scheduler per istanziare un emitter concreto: ogni schedulatore deve implementare questo metodo. Si è preferita questa decisione (anziché delegare questo compito al `Provider`) per via dell'accoppiamento che lega uno specifico scheduler con il proprio emitter delle risorse.

Lo schedulatore possiede una tabella di kernel per evitare di ricompilare un kernel già processato per i run successivi al primo. E' invocato il metodo `emitKernelIL` che al suo interno esegue:

```
_emitter.EmitKernelHeader(k, builder);

k.Resources.ForEach(r => r.Emit(_emitter, builder));
```

E' presente un overload del metodo `Emit` per ogni tipo di risorsa (input, output, const) e per ognuna delle risorse del kernel è invocato questo metodo.

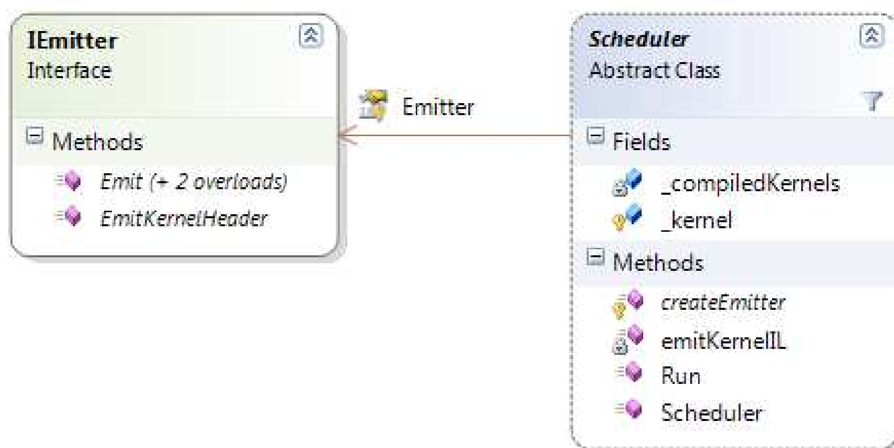


Figura 9.2, il rapporto tra lo scheduler e l'emitter delle risorse

Nel caso di `InputResource` viene dichiarata una risorsa utilizzando informazioni quali l'id, il rango (risorsa monodimensionale o a due dimensioni) e il formato, ossia il tipo di dato che essa contiene.

Nel caso di `OutputResource` viene dichiarato un registro di output, a meno che non si tratti del buffer globale (il cui registro `g` non è necessario dichiarare).

Nel caso di `ConstResource` viene dichiarato un buffer delle costanti, e a seconda del numero di componenti viene divisa la lunghezza per tenere conto che sulla GPU le locazioni sono da 128 bit.

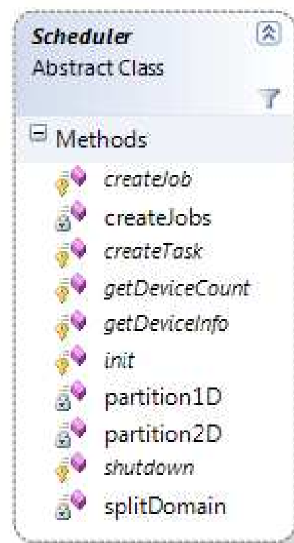


Figura 9.3, la classe astratta Scheduler

La fase di esecuzione che si svolge all'interno del metodo `Run` prevede:

- *inizializzazione (*`init`*)*, serve per inizializzare il sistema CAL
- *creazione job* (`createJobs`)
- *esecuzione job* (metodo `Run` di `Job`)
- *terminazione* (`shutdown`), chiude l'ambiente CAL

Un `Job` rappresenta l'esecuzione su un device. Un job può implicare uno o più `Task`. L'astrazione del job è usata per fattorizzare la fase di apertura di un device, di compilazione da IL in ISA GPU-specific e relativo caricamento su GPU. Un `Task` rappresenta una computazione su un determinato dominio; modella la fase di allocazione e preparazione dati di input e quella dell'esecuzione propriamente detta.

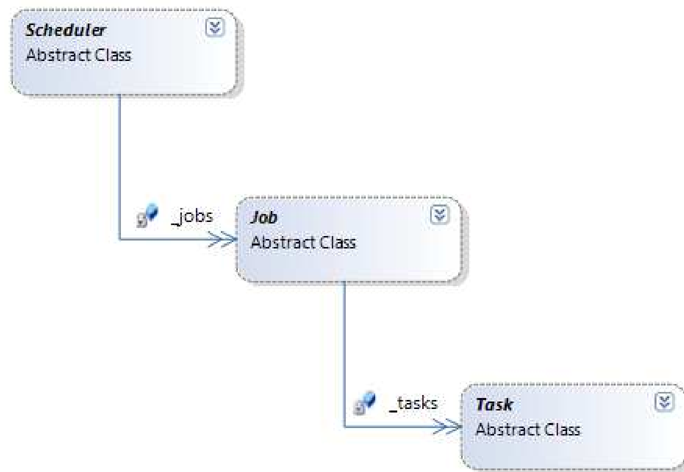


Figura 9.4, il rapporto tra Scheduler, Job e Task

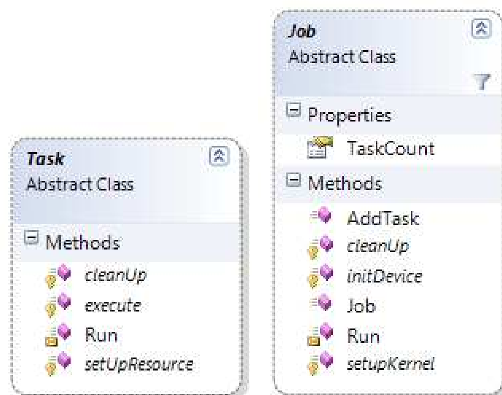


Figura 9.5, le classi Job e Task

Durante la fase di *creazione job*, lo schedulatore in base al numero di dispositivi (omogenei) presenti nel sistema crea un determinato numero di job, sfruttando in questo modo la possibilità di usare più schede video contemporaneamente per effettuare calcolo. Ad ognuna di esse viene assegnata una porzione distinta del dominio d'esecuzione (tramite un semplice algoritmo di partizionamento statico). Lo schedulatore utilizza il factory method `createJob` (nel nostro caso `l'ATIScheduler` crea un `CALJob`) per richiedere una versione del job legata da una particolare API.

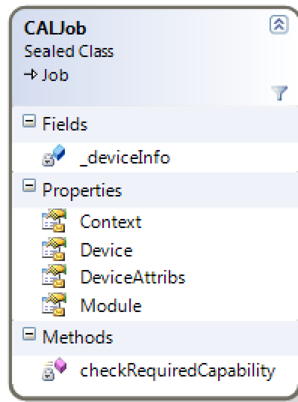


Figura 9.6, la classe CALJob

Un `Job` al momento della propria creazione invoca il metodo `initDevice`, all'interno del quale richiede al device desiderato informazioni e attributi, apre il device, crea il contesto, e verifica la propria compatibilità con il device, controllando che quest'ultimo supporti i compute shader nel caso sia richiesto l'uso di memoria condivisa locale o globale (`checkRequiredCapability`).
Indipendentemente dal numero di device disponibili, è necessario suddividere il dominio d'esecuzione (tramite partizionamento ricorsivo) creando più `Task` se non è possibile allocare interamente i dati nella memoria GPU.

`partition1D`

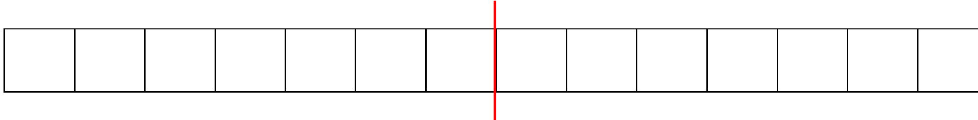


Figura 9. 7 il partizionamento di un dominio 1D

`partition2D`

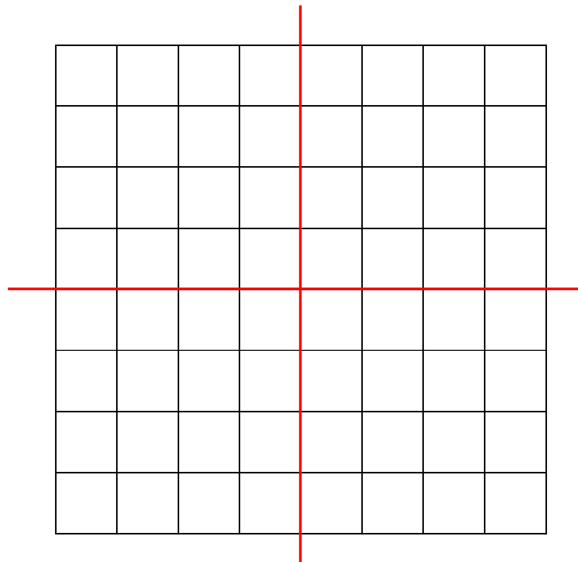


Figura 9.8, il partizionamento di un dominio 2D

Per ogni partizione è poi creato un `Task` usando il factory method `createTask`.

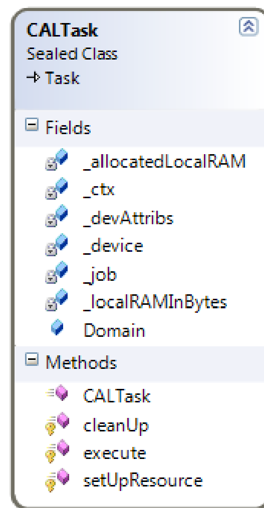


Figura 9.9, la classe CALTask

Si passa ora all'*esecuzione job*: nel caso di job singolo, il metodo `Run` del `Job` è eseguito dal thread corrente. Altrimenti, ci si affida al `ThreadPool` per avviarli concorrentemente in modo tale da non serializzare l'avvio delle computazioni, rendendo di fatto inutile il supporto multidevice offerto dall'architettura sottostante.

```
if (_jobs.Count == 1)
    _jobs.Peek().Run(kernelSrc, _kernel.Resources);
else
    _jobs.ForEach(j =>
        ThreadPool.QueueUserWorkItem(o =>
            ((Job)o).Run(kernelSrc, _kernel.Resources), j)
        );
```

Quello che fa internamente il metodo `Run` di un `Job` è:

```
internal void Run(string kernelSource, IEnumerable<Resource> resources)
{
    setupKernel(kernelSource);
    _tasks.ForEach(t => t.Run(resources));
    cleanUp();
}
```

effettuare il setup del kernel, lanciare sequenzialmente i `Task`, dopodiché fare il `cleanUp`, ossia disallocare le risorse utilizzate. Il setup del kernel prevede la compilazione del sorgente espresso in IL nell'ISA specifico di un processore, il linking (nel caso siano generati più kernel dipendenti l'uno dall'altro), e il caricamento all'interno di un modulo (ossia in un contesto) dell'immagine ottenuta dalla fase di linking. Al termine dell'esecuzione nel `cleanUp`, verrà effettuato l'unload del modulo, distrutto il contesto e chiuso il device.

Il run per ogni Task prevede:

```
internal void Run(IEnumerable<Resource> resources)
{
    resources.ForEach(r => setUpResource(r));
    execute(resources);
    cleanUp(resources);
}
```

che ogni risorsa inerente quel particolare Task debba essere allocata: vengono effettuati controlli per verificare che siano supportate risorse di tipo double (nel caso la risorsa contenga numeri in doppia precisione), e nel caso in cui si richieda l'uso del global buffer se questo è disponibile. Inoltre le risorse che rappresentano Scatter/GatherStream devono essere di tipo 1D e avere un padding a 64 elementi.
Ogni risorsa è allocata usando i metodi esposti dal layer di *Interop*: se c'è memoria a sufficienza sul device le risorse saranno allocate localmente ad esso, altrimenti verranno allocate in remoto (nella memoria di sistema), naturalmente i tempi di accesso in questo caso saranno maggiori.
Dopo l'allocazione, i dati delle risorse di input e di quelle costanti sono trasferiti da CPU a GPU: collegata a ogni Resource vi è una ResourceInfo, che contiene i dati effettivi e le informazioni che li descrivono: come dimensioni, numero di componenti, puntatore all'area dati. E' presente anche il GCHandle che viene restituito dopo l'operazione di *pinning* (descritta nel cap.5) per bloccare l'area di memoria gestita dal runtime del CLR in modo tale che dopo la fase d'esecuzione la copia dei risultati avvenga nella giusta posizione. Un'istanza di ResourceInfo è costruita a partire da un array, specificando durante la creazione se si voglia una risorsa 1D o 2D. Nel caso bidimensionale le dimensioni Width ed Height in modo tale che siano sempre potenze di 2.
Nel caso di ATI è stata definita la sottoclasse CALResourceInfo, contenente dati specifici di CAL, come CALMem, CALresource, CALformat.

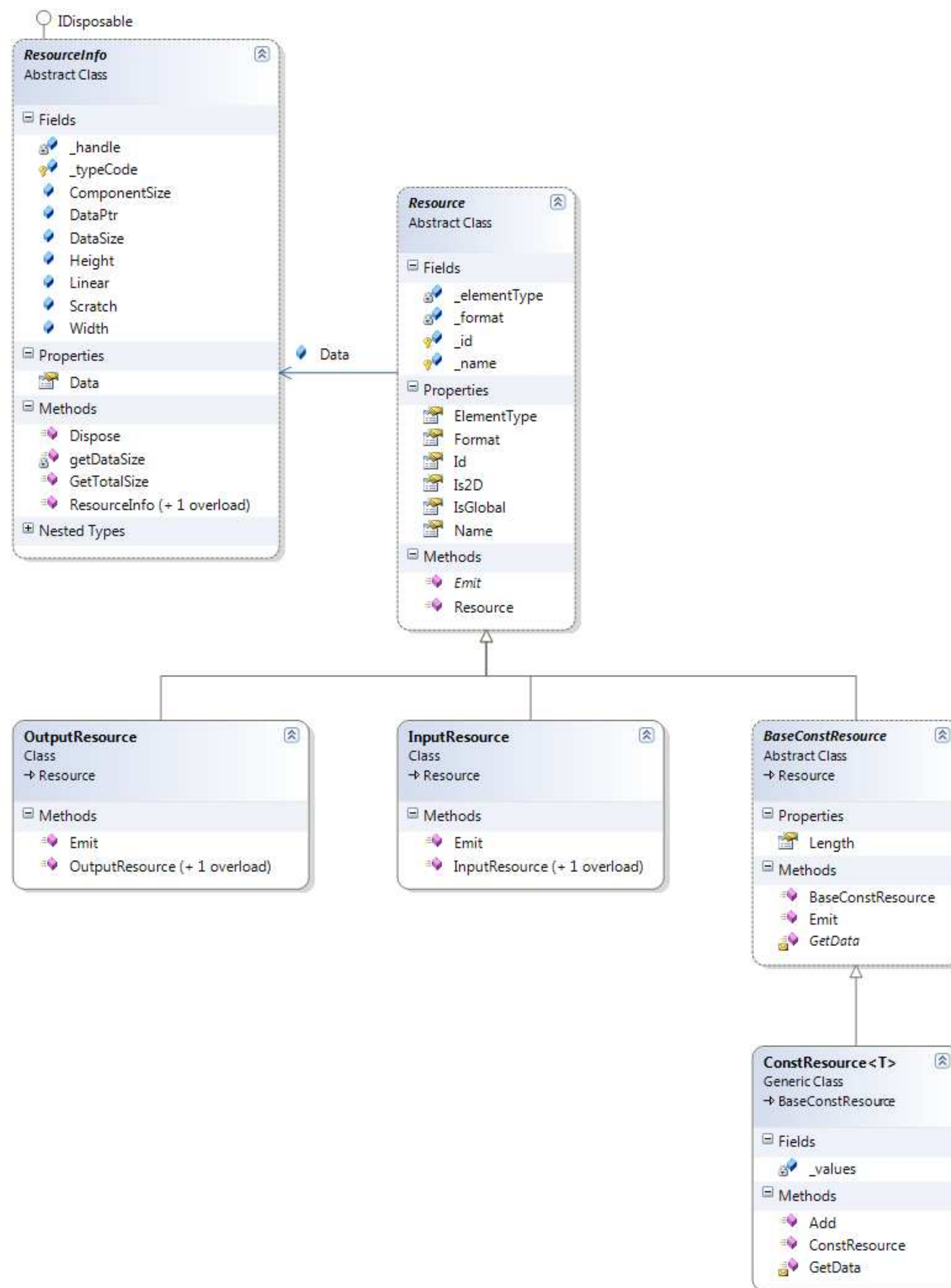


Figura 9.10, il legame tra le classi Resource e ResourceInfo

A questo punto è lanciata la computazione sul particolare (sotto)dominio d'esecuzione del `Task`. Se ne attende il completamento e al termine i risultati sono trasferiti dalle risorse di output della GPU alla CPU. Si procede al `cleanUp` delle risorse: per ognuna di esse vengono liberate le risorse allocate lato GPU e rilasciati oggetti del runtime.

A seguito del trasferimento da GPU i dati sono già direttamente copiati nell'area dati all'interno della `ResourceInfo`, per cui accessibili al chiamante che aveva invocato l'esecuzione su GPU.

9.2 CALInterop

Per poter dialogare con i driver CAL è stato necessario costruire un mapping tra il mondo managed e quello unmanaged. Il .NET framework nasconde molta della complessità dietro il meccanismo di P/Invoke (marshaling, unmarshaling dei parametri) come visto nel capitolo 5.

Sono state definite due classi statiche (contenenti solo metodi statici) che essenzialmente si comportano da wrapper. Troviamo quindi dentro `CALCompiler` il wrapper per le funzioni del compilatore CAL:

```
/// <summary>
/// Provides methods to handle CAL compiler.
/// </summary>
public static class CALCompiler
{
    ...

    /// <summary>
    /// Compiles a source language string to the specified target device and
    /// return a compiled object.
    /// </summary>
    /// <param name="obj">Created object.</param>
    /// <param name="language">Source language designation.</param>
    /// <param name="source">String containing kernel source code.</param>
    /// <param name="target">Machine target.</param>
    /// <returns>
    /// CAL_RESULT_OK on success, CAL_RESULT_ERROR if there was an error.
    /// </returns>
    [DllImport("aticalcl")]
    public static extern CALresult calclCompile(out CALobject obj,
        CALlanguage language, string source, CALtarget target);

    ...
}
```

e dentro `CALRuntime` il wrapper per il runtime di CAL:

```
#region CALRuntime

/// <summary>
/// Provides methods to handle CAL runtime.
/// </summary>
public static class CALRuntime
{
    CAL Subsystem Functions

    CAL Device Functions

    CAL Resource Functions

    CAL Context Functions

    CAL Image Functions

    CAL Module Functions

    CAL Error/Debug Helper Functions
}
```

Figura 9.3, il wrapper per il runtime di CAL e le diverse categorie di funzioni in esso contenuto

Il tutto è stato incluso in un progetto di tipo *Library*. Dopo la compilazione di questa si è ottenuto un assembly contenente la libreria (una .dll) necessaria per chiamare le funzioni driver da un qualsiasi programma .NET. Questo modulo è usato dal modulo ATI.

Sfruttando i due wrapper definiti in precedenza, è stata creata una classe "helper" CAL che espone funzionalità di uso, definite in termini di funzioni driver, come ad esempio le routine per il trasferimento dati da/a GPU CPU:

```
public static bool CopyDataToGPU(ref CALcontext ctx,
                                 ref CALresource resource,
                                 ResourceInfo data)

public static bool CopyDataFromGPU(ref CALcontext ctx,
                                   ref CALresource resource,
                                   ResourceInfo data)
```

Queste al loro interno richiamano `copyFrom`, `copyTo`, che a loro volta utilizzano funzioni definite a livello di OS come `Memcpy` e `Memset`.
`copyFrom` e `copyTo` sono ottimizzate in modo tale da copiare in un colpo solo se viene passata una risorsa lineare altrimenti scandiscono la risorsa riga per riga (saltando in base al pitching della risorsa).

Capitolo 10

Risultati sperimentali

I test sono stati condotti su una macchina con la seguente configurazione hardware / software:

- AMD Turion X2 Ultra 64, 2200 MHz core clock speed
- 2 GB RAM,
- Windows 7 RC Build 7100
- Scheda video ATI Radeon HD 3200, 500MHz core clock, 400MHz memory clock, Driver Packaging Version 8.632.1.2, 512 MB memory (condivisa)

Per determinare il tempo di completamento è stata utilizzata la classe `System.Diagnostics.Stopwatch` che espone i metodi `Start`, `Stop`, `Reset`, e la proprietà `ElapsedMilliseconds` (che riporta il tempo trascorso tra l'invocazione dei metodi `Start` e `Stop` espresso in millisecondi).

Ci interessa soprattutto capire qual è la grana del calcolo oltre la quale si riescono a bilanciare i costi di gestione della compilazione, ottenendo prestazioni che migliorano quelle del corrispettivo calcolo sequenziale e multicore. Siccome il risultato della compilazione è cachato per esecuzioni successive, si è deciso di testare anche l'esecuzione su GPU al secondo run, in modo tale da eliminare i costi di compilazione.
Per ogni configurazione di test sono stati eseguiti diversi run effettuando una media dei valori ottenuti, in modo tale da ridurre l'incidenza di valori outlayer.
Per poter variare la grana del calcolo abbiamo applicato le funzioni sequenziali `f`, a dati di dimensione via via maggiore, utilizzando multipli `n` di 1024 interi come input; il numero `n` per noi rappresenta la grana di calcolo. Bisogna ricordare inoltre che l'esecuzione su GPU prevede due comunicazioni attraverso il bus PCI-Express per passare i dati di input e ritornare i dati di output. Quindi all'aumentare della grana del calcolo aumenta l'incidenza di queste comunicazioni ma data la velocità del bus (almeno 5 GB/s), si è deciso di trascurare questa componente.

Nella prima configurazione di test la funzione `f` è un semplice metodo che duplica gli elementi passati in ingresso dallo stream:

```
class MyComputation
{
    public void TestKernel(InputStream<float> source,
                           OutputStream<float> dest)
    {
        dest.Write(source.Current * 2);
    }
}
```

Il sorgente CAL IL prodotto (in release) è:

```
il_ps_2_0
dcl_input_position_interp(linear) v0
dcl_cb cb0[1]
dcl_resource_id(0)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtx(float)_fmtw(f
loat)
dcl_output_generic o0
sample_resource(0)_sampler(0) r0, v0
mul r1, r0, cb0[0].x
mov o0, r1
ret
end
```

Vediamo come variano i tempi di completamento in funzione della grana `n` del calcolo:

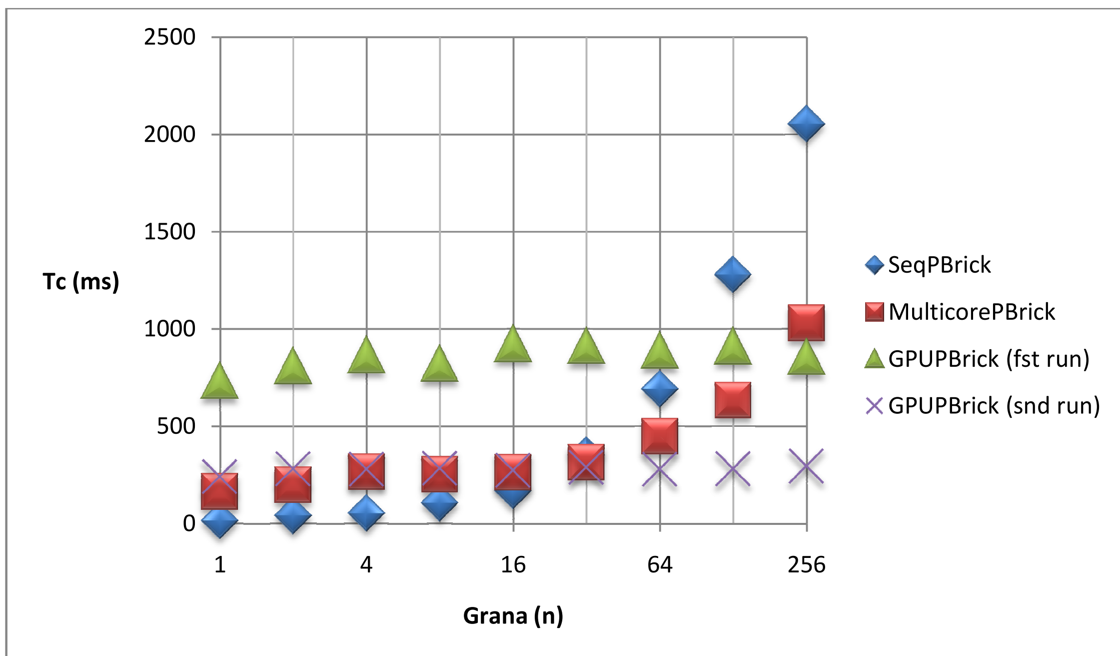


Figura 10.1, Tempo di completamento in funzione della grana di calcolo

Questo primo test è stato effettuato con grana del calcolo molto fine; per valori di `n` compresi tra 1 e 4 si può notare come le prestazioni del `GPUPBrick` (anche a compilazione effettuata) siano inferiori rispetto al caso sequenziale e a quello multicore. Anche questo presenta valori peggiori rispetto al caso sequenziale; per `n` uguale a 8 si assiste al sostanziale allineamento delle prestazioni del PBrick GPU con quello multicore, allineamento che prosegue fino a `n` uguale a 16.
Per valori intorno a 16, tutte le modalità d'esecuzione (a parte il GPUPBrick first run) presentano i medesimi risultati.

A partire da valori della grana del calcolo pari a 64, il `GPUPBrick` domina le altre configurazioni (sempre però al secondo run). Si noti come il tempo di completamento si mantenga pressoché costante rispetto alla grana del calcolo, laddove la versione sequenziale aumenta sensibilmente. Infine, superato il valore 256, anche la versione che incorre negli overhead della compilazione riesce ad ottenere prestazioni migliori del caso sequenziale e di quello parallelizzato utilizzando il brick multicore.

Qui possiamo osservare i precedenti valori espressi in un grafico avente il tempo di completamento espresso secondo una scala logaritmica in base 10:

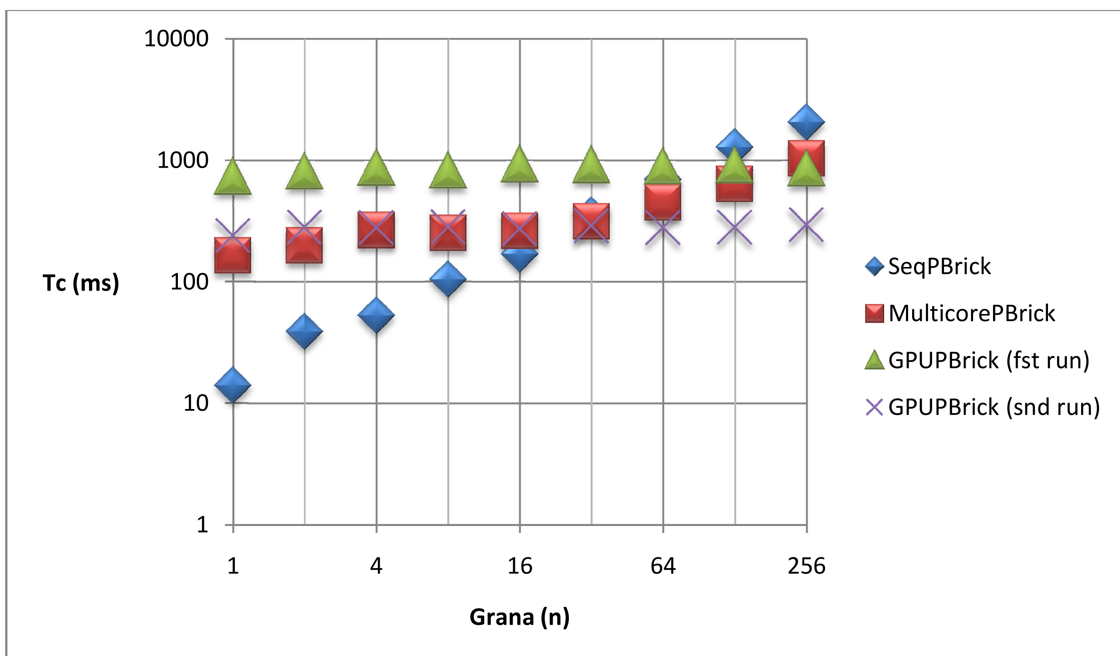


Figura 10.2, Tempo di completamento in funzione della grana di calcolo (scala logaritmica)

L'andamento del tempo di completamento è rispecchiato dall'andamento della scalabilità. Questo significa che all'aumentare della grana del calcolo i vantaggi della parallelizzazione rispetto al caso sequenziale aumentano, pur rimanendo lontani dalla scalabilità ideale (nel caso in esame 64).

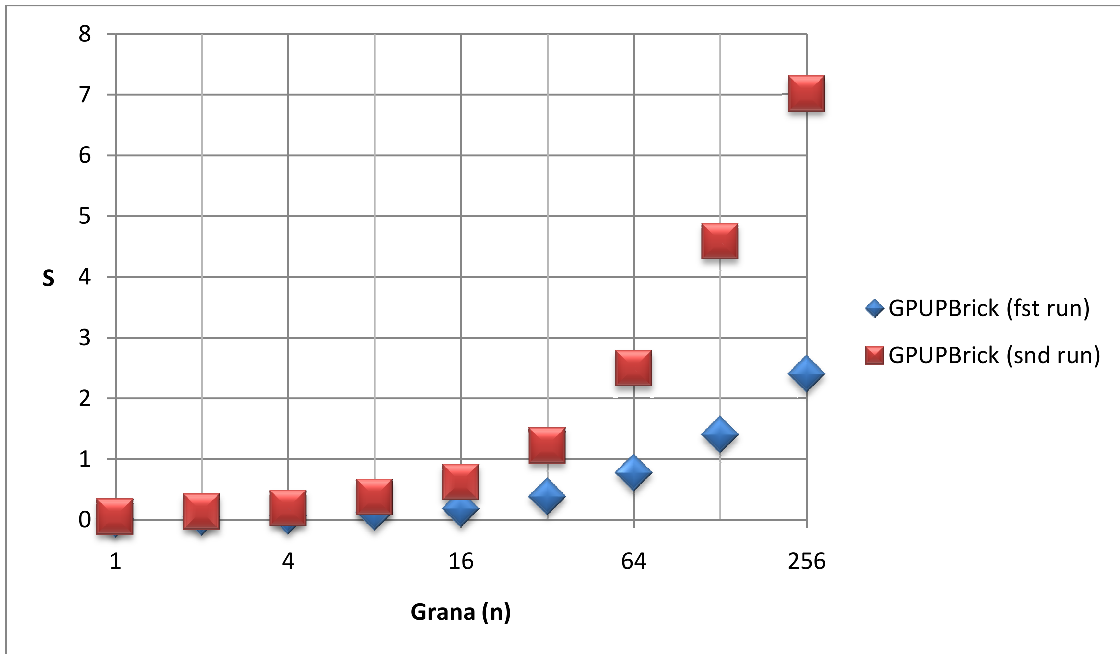


Figura 10.3, Scalabilità in funzione della grana di calcolo

La stessa funzione è stata poi applicata ad un configurazione a grana grossa (1024 * 1024 * 16 interi) ottenendo in questo caso valori molto soddisfacenti: scalabilità pari a 62,06 ed efficienza relativa di 0,96. Questo è dovuto alla forma di parallelismo adottata, poiché la forma *map* (che di fatto è quella che tacitamente è stata applicata) è un tipico paradigma adatto ai casi in cui la grana del calcolo sia molto grossa.

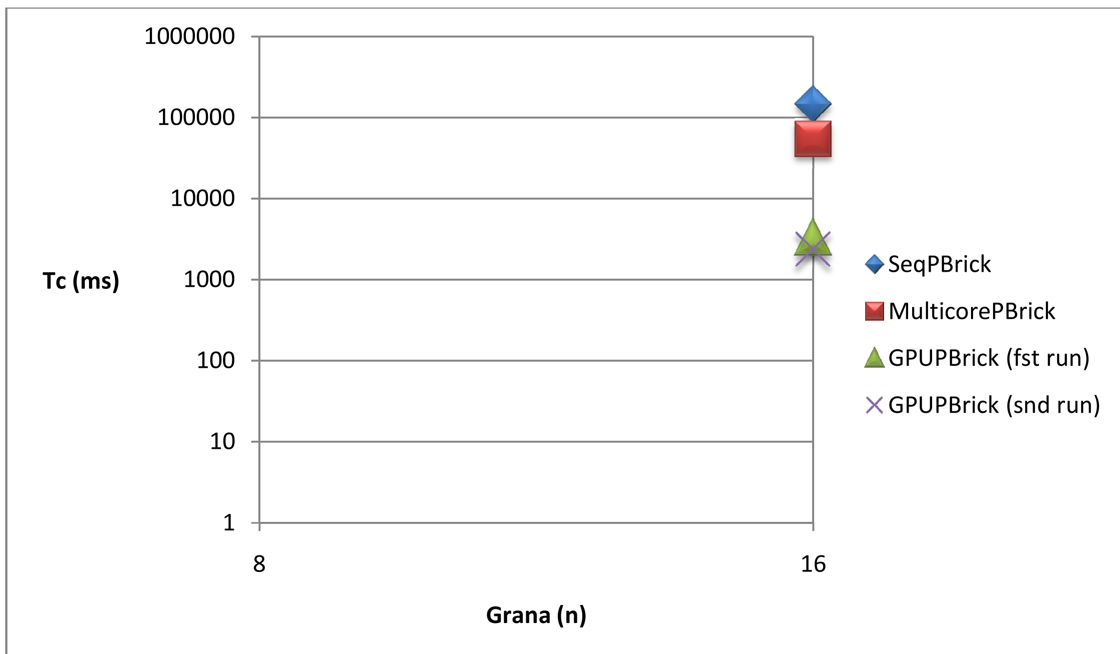


Figura 10.4, Tempo di completamento in funzione della grana di calcolo (grana grossa)

La secondo configurazione di test ha previsto lo sviluppo di un generatore di immagini dell'insieme di Mandelbrot.

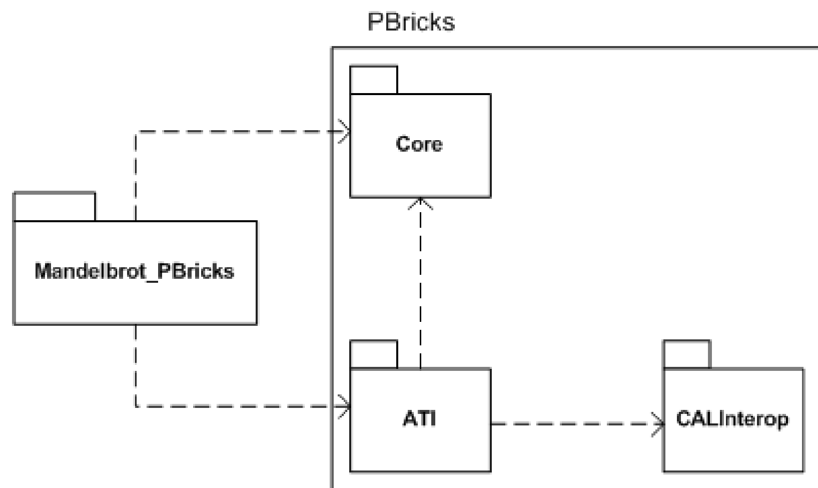


Figura 10.5, dipendenze dall'applicazione con PBricks

```
public void ComputeMandelbrotIndex(InputStream<float> rows,
                                   InputStream<float> cols,
                                   OutputStream<float> results)
{
    var row = rows.Current;
    var col = cols.Current;

    float x = (col * SampleWidth) / ImageWidth + OffsetX;
    float y = (row * SampleHeight) / ImageHeight + OffsetY;

    float y0 = y;
    float x0 = x;

    for (var i = 0f; i < MaxIterations; i++)
    {
        if (x * x + y * y >= 4f)
        {
            results.Write((i % 255f) + 1f);
            return;
        }
        float xtemp = x * x - y * y + x0;
        y = 2f * x * y + y0;
        x = xtemp;
    }

    results.Write(0f);
}
```

Le linee di codice in rosso indicano le differenze rispetto alla versione sequenziale. Per il sorgente prodotto dalla compilazione si faccia riferimento alla sezione A.4 dell'appendice.

La versione così definita non è il modo migliore di esprimere il Mandelbrot in parallelo secondo il paradigma data parallel a causa della:

- Varianza del calcolo
- Grana del calcolo troppo fine

Sarebbe meglio passare gli estremi delle regioni da calcolare (anche per trasferire meno dati possibili).

I risultati attesi non sono ottimali data la presenza all'interno del kernel di un ciclo e una selezione (le architetture GPU gestiscono male queste situazioni poiché a causa di un solo program counter per wavefront (come spiegato nel capitolo 2) nel caso di flussi di controllo divergenti è inevitabile la serializzazione del calcolo perché non possono essere intrapresi in parallelo entrambi i rami. Come si può leggere anche in ([48] par. 1.2.3, pag.30) "Il controllo di flusso, come il branching, è effettuato combinando tutti i necessari cammini all'interno del wavefront. Ciò significa che se thread all'interno dello stesso wavefront divergono, tutti i cammini sono eseguiti sequenzialmente. Per esempio, se un thread contiene un branch con due cammini, il wavefront prima esegue un cammino e poi l'altro cammino. Il tempo totale per eseguire il branch è dato dalla somma di ogni cammino. Un punto importante che è necessario sottolineare è che se anche un solo thread all'interno del wavefront diverge allora anche tutti gli altri devono seguirlo. Ad esempio: se due branch A e B, occupano lo stesso intervallo di tempo t per essere eseguiti da un wavefront, il tempo totale per l'esecuzione, se qualcuno dei thread diverge è 2t.
I cicli vengono gestiti in maniera simile, dove un wavefront occupa un SIMD engine finchè c'è almeno un thread nel wavefront che sta ancora calcolando. Quindi il tempo totale d'esecuzione per il wavefront è determinato dal thread col tempo d'esecuzione maggiore. Ad esempio se t è il tempo impiegato per eseguire una singola iterazione di un loop, e all'interno di un wavefront tutti i thread eseguono il loop una volta soltanto, ad eccezione di un singolo thread che esegue il loop 100 volte, il tempo d'esecuzione dell'intero wavefront è 100t."
Quello che probabilmente avviene è che c'è sbilanciamento del carico poiché i punti del piano che raggiungono dopo la condizione di terminazione del ciclo bloccano gli altri thread in elaborazione.

Il test è stato condotto fissando un valore per la dimensione dell'immagine frattale e misurandone poi il tempo di completamento (dove nel tempo di completamento non sono inclusi i tempi per la fase di rendering). Da un'analisi dei risultati si può osservare un miglioramento più accentuato (pur non raggiungendo mai prestazioni vicine a quelle ideali) con l'incremento del carico di lavoro. La grana del calcolo in questo caso è stata fatta variare modificando la dimensione dell'immagine di partenza.

I risultati ottenuti sono qualitativamente quelli attesi: per valori bassi (dimensioni inferiori a 128x128), domina la versione sequenziale, seguita dalla versione multicore e poi da quella per gpu. Sopra una certa soglia il caso sequenziale comincia a degradare, e già per valori relativamente bassi (256x256) la soluzione basata su gpu fornisce i risultati migliori rispetto alle altre.

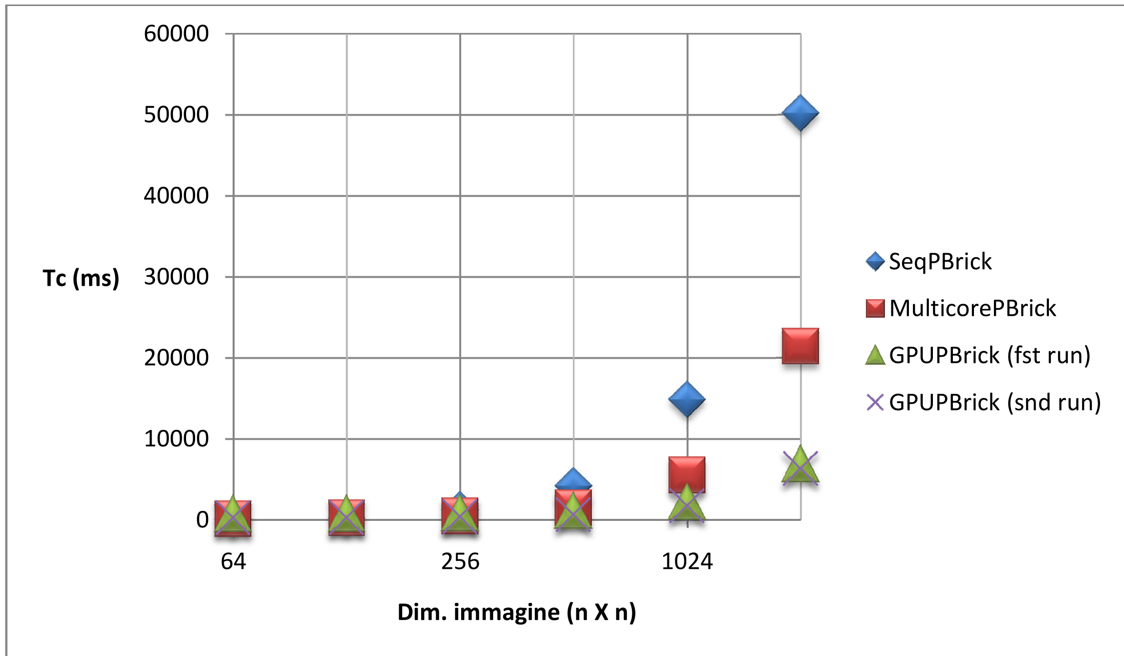


Figura 10.6, Tempo di completamento in funzione della grana di calcolo (dimensione dell'immagine)

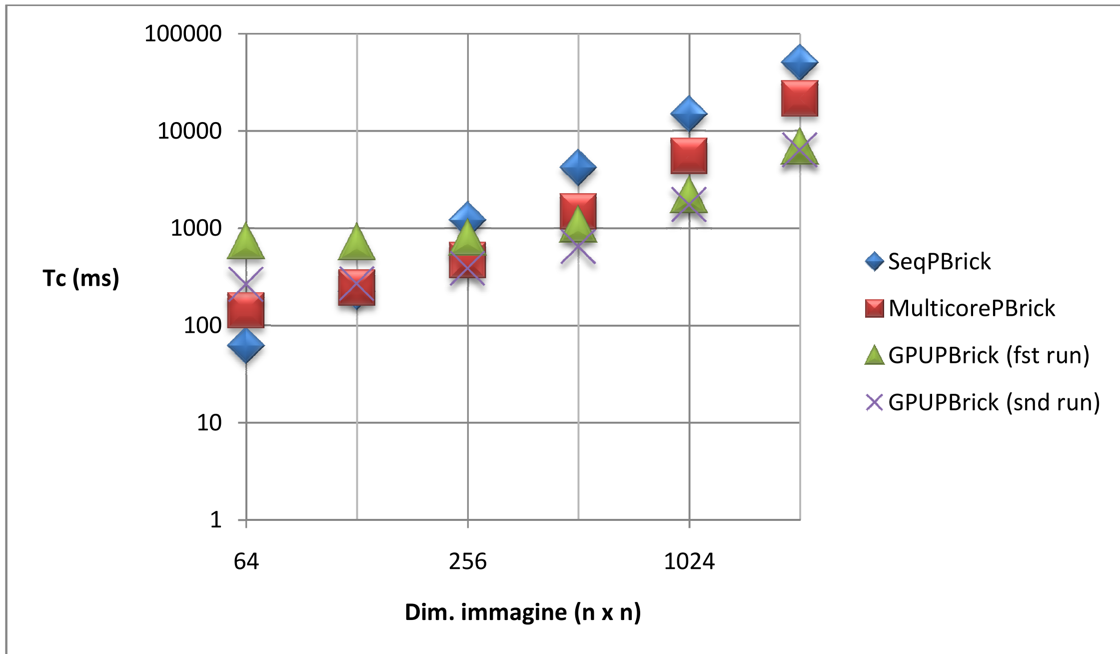


Figura 10.7, Tempo di completamento in funzione della grana di calcolo (dimensione dell'immagine), espresso secondo scala logaritmica

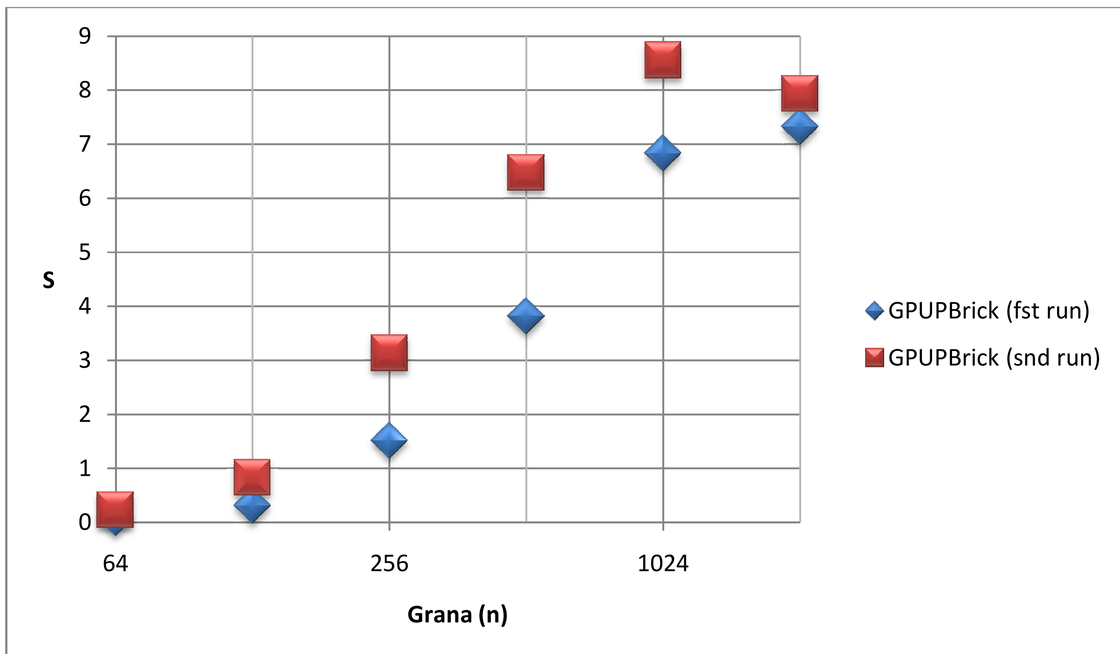


Figura 10.8, Scalabilità in funzione della grana di calcolo

La scalabilità tende a salire all'aumentare della grana del calcolo fino ad un massimo per immagini di 1024x1024, anche se questa situazione è molto soggetta alle condizioni nelle quali è eseguito il run: ripetendo il test non sempre si verifica questo andamento.

Capitolo 11

In questo capitolo tiriamo le somme sul lavoro svolto e suggeriamo alcuni possibili miglioramenti che potranno essere realizzati in futuro.

Conclusioni

In questa tesi abbiamo studiato le soluzioni fornite dai due produttori di schede video ATI e Nvidia in modo tale da conoscerne similarità e differenze: sia in termini hardware sia in termini di modello di programmazione. Questo ha permesso di definire un sotto insieme di funzionalità comuni ad entrambe e le si è esposte al programmatore in un modo semplice ed intuitivo, che non richiedesse grossi cambiamenti concettuali rispetto a quello che è abituato a fare nella programmazione quotidiana. Occorre precisare che lo scopo finale di questo lavoro non è tanto quello di fornire una soluzione definitiva allo specialista di calcolo parallelo, che continuerà a lavorare utilizzando API di basso livello al fine di ottenere performance realmente interessanti, quanto quello di provare che il mapping tra macchine virtuali e GPU fosse possibile, in modo tale da rendere disponibile questa tecnologia a sviluppatori meno esperti di problematiche inerenti il calcolo parallelo, e di sviluppo su GPU.

Le finalità identificate all'inizio del lavoro sono state realizzate: il mapping tra le macchine virtuali e le GPU è davvero fattibile: è ora possibile effettuare l'esecuzione su GPU di metodi definiti in normali tipi .NET (sicuri che verrà rispettata la semantica iniziale); al tempo stesso si può effettuare il debugging dei metodi in sequenziale senza il bisogno di dover modificare sostanzialmente il codice. Le gerarchie di memoria presenti su GPU sono state esposte con un preciso modello di memoria.

Grazie agli sviluppi futuri da me accennati questa libreria potrà essere ulteriormente sviluppata o integrata all'interno di altri progetti. Ricordiamo infatti la collaborazione all'interno del CVS Lab per sviluppare *4Centaury* [61]. Il mio lavoro dovrebbe essere integrato per realizzare la compilazione verso schede ATI.

Lo sviluppo del presente lavoro di tesi ha costituito per me un momento fondamentale nella formazione complessiva ricevuta con la laurea specialistica in tecnologie informatiche. Si è trattato di un'esperienza gratificante che mi ha permesso di approfondire argomenti che hanno sempre suscitato il mio interesse, come la programmazione parallela e la metaprogrammazione, facendomi conoscere un settore in pieno sviluppo dell'informatica, come quello del GPU computing.

Mi sono dovuto confrontare con sfide quali l'analisi di codice a livello intermedio che ha notevolmente impreziosito il mio bagagliaio di conoscenze tecniche inerenti il CLR e le macchine virtuali; mi ha inoltre permesso di lavorare a fianco di persone estremamente competenti che mi hanno guidate verso un processo di maturazione.

Sviluppi futuri

Il lavoro lungi dall'essere completo presenta diverse possibilità di miglioramento e sviluppo. Vi sarebbero tutta una serie di ottimizzazioni legate all'uso dei registri: per ridurre il numero di registri utilizzati e il numero di istruzioni `mov` impiegate dovrebbero essere applicati algoritmi di dataflow analysis; oppure fattorizzare espressioni costanti in modo da eliminarne la valutazione a run time, o ancora replicare quelle ottimizzazioni che vengono effettuate solamente dal JIT e che pertanto non sono ancora disponibili in fase di analisi di codice intermedio.

Altre possibilità di sviluppo includono:

- Estendere i tipi di dato supportati nelle computazioni: ossia permettere di avere stream di tipi custom (generici e non generici);
- Fornire un'implementazione del `ClusterPBrick` (per l'esecuzione su cluster) e del costruttore di computazione `ConcurrentPBrick`, per indicare allo schedulatore che due PBrick devono essere, ove possibile, lanciati in concorrenza;
- Definire una precisa semantica del calcolo delle computazioni che possono essere espresse combinando tra loro i PBrick;
- Realizzare un binding anche per CUDA, ossia realizzare le classi concrete definite nel modulo *Core* al fine di poter eseguire computazioni anche su schede Nvidia;
- Dare la possibilità di invocare metodi all'interno di un metodo kernel. Questo comporterebbe una compilazione in cascata di tutti i metodi coinvolti;
- Completare l'implementazione della *Reduce*, che attualmente è esposta solo a livello di API di alto livello, ma non è ancora del tutto implementata;
- Migliorare le funzionalità dello schedulatore per dare la possibilità al costruttore di computazione `PipePBrick` di decidere, grazie ad un modello dei costi delle comunicazioni, la grana ideale delle comunicazioni tra CPU e GPU nel caso in cui gli stadi prevedano computazione su CPU e GPU;
- Supporto alla gestione di dispositivi eterogenei (usare contemporaneamente schede ATI e Nvidia);
- Realizzare il mapping della gestione eccezioni tramite eventi CAL.

Appendice

A.1 Esempio di codice Brook+/CAL

Forniamo un semplice codice in modo da spiegare le principali parti di un programma CAL e quelle di un programma Brook+. Il seguente codice implementa la funzione saxpy in doppia precisione. La funzione saxpy è una combinazione di una moltiplicazione scalare (a, cb0 nel codice IL) e addizione vettoriale (x e y).

```
kernel void ILKernel(double a, double x<>, double y<>, out double result<>)
{
	result = a * x + y;
}

int main(int argc, char** argv)
{
	unsigned int Length = 100;
	double a = 10.0;
	double* InData1;
	double* InData2;
	double* Result;

	// Memory allocation.
	InData1 = (double*) malloc(sizeof(double) * Length);
	InData2 = (double*) malloc(sizeof(double) * Length);
	Result = (double*) malloc(sizeof(double) * Length);

	// Set input values.
	for (int i = 0; i < Length; ++i)
	{
		InData1[i] = (double) rand();
		InData2[i] = (double) rand();
	}

	// Set domain.
	double indata1<Length>;
	double indata2<Length>;
	double result<Length>;

	streamRead(indata1, InData1);
	streamRead(indata2, InData2);

	// Run compute kernel.
	ILKernel(a, InData1, InData2, OutData);

	// Get result.
	streamWrite(OutData, result);

	// Clean up and exit.
	free(InData1);
	free(InData2);
	free(Result);

	return 0;
```

```
}
```

Listato 1: esempio di codice Brook+

```
const CALchar* ILkernel =
      "il_ps_2_0\n"
      "dcl_input_position_interp(linear_noperspective) v0.xy__\n"
      "dcl_output_generic o0.x___\n"
      "dcl_cb cb0[1]\n"
      "dcl_resource_id(0)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtz(float)
                                          _fmtw(float)\n"
      "dcl_resource_id(1)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtz(float)
                                          _fmtw(float)\n"
      "sample_resource(0)_sampler(0) r0, v0.xyxx\n"
      "sample_resource(1)_sampler(0) r1, v0.xyxx\n"
      "mad_ieee o0.x___, cb0[0].x, r0.x, r1.x\n"
      "end\n";

int main(int argc, char** argv)
{
      // Initialization.
      calInit();
      CALuint numDevices = 0;
      calDeviceGetCount(&numDevices);
      CALdeviceinfo info;
      calDeviceGetInfo(&info, 0);
      CALdevice device = 0;
      calDeviceOpen(&device, 0);
      CALcontext ctx = 0;
      calCtxCreate(&ctx, device);

      // Compile and link kernel.
      CALdeviceattribs attribs;
      attribs.struct_size = sizeof(CALdeviceattribs);
      calDeviceGetAttribs(&attribs, 0);
      CALobject obj;
      calclCompile(&obj, CAL_LANGUAGE_IL, ILkernel, attribs.target);

      // Link object into an image.
      CALimage image = NULL;
      calclLink(&image, &obj, 1);

      // Memory allocation.
      // allocate input/output resources and map them into the context.
      unsigned int Length = 100;
      CALresource InData1 = 0; CALresource InData2 = 0;
      calResAllocLocal1D(&InData1, device, Length, CAL_FORMAT_DOUBLE_1, 0);
      calResAllocLocal1D(&InData2, device, Length, CAL_FORMAT_DOUBLE_1, 0);
      CALresource Result = 0;
      calResAllocLocal1D(&Result, device, Length, CAL_FORMAT_DOUBLE_1, 0);
      CALresource a = 0;
      calResAllocRemote1D(&a, &device, 1, 1,  CAL_FORMAT_DOUBLE_1, 0);
      CALuint pitch1 = 0, pitch2 = 0;
      CALmem InMem1 = 0, InMem2 = 0;

      // Set input values.
```

```
double* findata1 = NULL;
double* findata2 = NULL;
calCtxGetMem(&InMem1, ctx, InData1);
calCtxGetMem(&InMem2, ctx, InData2);

calResMap((CALvoid**)&findata1, &pitch1, InData1, 0);
for (int i = 0; i < Length; ++i)
      findata1[i * pitch1] = rand();
calResUnmap(InData1);

calResMap((CALvoid**)&findata2, &pitch2, InData2, 0);
for (int i = 0; i < Length; ++i)
      findata2[i * pitch2] = rand();
calResUnmap(InData2);

double* constPtr = NULL;
CALuint constPitch = 0;
CALmem constMem = 0;

// Map constant resource to CPU and initialize values.
calCtxGetMem(&constMem, ctx, a);
calResMap((CALvoid**)&constPtr, &constPitch, a, 0);
constPtr[0] = 10.0;
calResUnmap(a);

// Mapping output resource to CPU and initializing values.
void* res_data = NULL;
CALuint pitch3 = 0;
CALmem OutMem = 0;
calCtxGetMem(&OutMem, ctx, Result);
calResMap(&res_data, &pitch3, Result, 0);
memset(res_data, 0, pitch3 * Length * sizeof(double));
calResUnmap(Result);

// Load module and set domain.
CALmodule module;
calModuleLoad(&module, ctx, image);
CALfunc func;
CALname InName1, InName2, OutName, ConstName;

calModuleGetEntry(&func, ctx, module, "main");
calModuleGetName(&InName1, ctx, module, "i0");
calModuleGetName(&InName2, ctx, module, "i1");
calModuleGetName(&OutName, ctx, module, "o0");
calModuleGetName(&ConstName, ctx, module, "cb0");

calCtxSetMem(ctx, InName1, InMem1);
calCtxSetMem(ctx, InName2, InMem2);
calCtxSetMem(ctx, OutName, OutMem);
calCtxSetMem(ctx, ConstName, constMem);
CALdomain domain = {0, 0, Length, 1};

// Run compute kernel.
CALevent event;
calCtxRunProgram(&event, ctx, func, &domain);

// wait for function to finish.
```

```
	while (calCtxIsEventDone(ctx, event) == CAL_RESULT_PENDING) { };

	// Get result.
	CALuint pitch4 = 0;
	double* foutdata = 0;

	calResMap((CALvoid**)& foutdata, & pitch4, Result, 0);
	for (int i = 0; i < Length; ++i)
		foutdata [i * pitch4] = (double)(i * pitch4);
	calResUnmap(Result);

	// Clean up and exit.
	calModuleUnload(ctx, module);
	calclFreeImage(image);
	calclFreeObject(obj);

	calCtxReleaseMem(ctx, InMem1);
	calResFree(InData1);
	calCtxReleaseMem(ctx, InMem2);
	calResFree(InData2);
	calCtxReleaseMem(ctx, constMem);
	calResFree(a);
	calCtxReleaseMem(ctx, OutMem);
	calResFree(Result);

	calCtxDestroy(ctx);
	calDeviceClose(devic );
	calShutdown();
}
```

Listato 2: esempio di codice CAL

A.2 Esempio di codice CUDA

```
__global__ void ILKernel(float a, float* InData1,
                         float* InData2, float* Result)
{
	int idx = blockIdx.x * blockDim.x + threadIdx.x;
	Result[idx] = InData1[idx] * a + InData2[idx];
}

int main(int argc, char** argv)
{
	float* InitData1; float* InitData2;
	float* InData1; float* InData2;
	float* Result; float* HostResult;
	float a = 10.0;
	unsigned int Length = 100;

	// Initialization.
	CUDA_DEVICE_INIT(argc, argv);
	cudaStream_t stream;
	CUDA_SAFE_CALL(cudaStreamCreate(&stream));
```

```
	// Memory allocation.
	CUDA_SAFE_CALL(cudaMallocHost((void**)&HostResult, Length));
	memset(HostResult, 0, Length);
	CUDA_SAFE_CALL(cudaMalloc((void**)&InData1, sizeof(float) * Length));
	CUDA_SAFE_CALL(cudaMalloc((void**)&InData2, sizeof(float) * Length));
	CUDA_SAFE_CALL(cudaMalloc((void**)&Result, sizeof(float) * Length));

	// Set input values.
	InitData1 = (float*) malloc(sizeof(float) * Length);
	InitData2 = (float*) malloc(sizeof(float) * Length);
	for (int i = 0; i < Length; ++i)
	{
		InitData1[j] = (float) rand();
		InitData2[j] = (float) rand();
	}
	CUDA_SAFE_CALL(cudaMemcpy(InData1, InitData1, sizeof(float) * Length,
	                          cudaMemcpyHostToDevice));
	CUDA_SAFE_CALL(cudaMemcpy(InData2, InitData2, sizeof(float) * Length,
	                          cudaMemcpyHostToDevice));

	// Run compute kernel.
	int n = 16 * 1024 * 1024;
	dim3 threads = dim3(512, 1);
	dim3 blocks  = dim3(n / threads.x, 1);
	GLIteration<<<blocks, threads, 0, stream>>>(a, InData1, InData2,
	                                            OutData);

	// Get result.
	cudaMemcpy(HostResult, Result , sizeof(float) * Length,
	           cudaMemcpyDeviceToHost);

	// Clean up and exit.
	CUDA_SAFE_CALL(cudaFree(InData1));
	CUDA_SAFE_CALL(cudaFree(InData2));
	CUDA_SAFE_CALL(cudaFree(Result));
	CUDA_SAFE_CALL(cudaFree(HostResult);
	free(InitData1); free(InitData2);
	return 0;
}
```

Listato 3: esempio di codice CUDA

A.3 Funzioni per gestire scritture/letture dal global buffer

```
func 1
  dcl_literal l0, 0x1, 0x0, 0x0, 0x0
  and {0}.x, {1}.x, l0.x
  ishr {0}.y, {1}.x, l0.x
  if_logicalz {0}.x
    mov g[{0}.y].xy__, {2}.xx
  else
    mov g[{0}.y].__zw, {2}.xx
```

```
  endif
  ret
endfunc
```

Listato 4, la funzione globalScatter2

```
func 2
  dcl_literal l0, 0x3, 0x2, 0x1, 0x0
  and {0}.x, {1}.x, l0.x
  ishr {0}.y, {1}.x, l0.y
  switch {0}.x
    default
      mov g[{0}.y].x___, {2}.x
      break
    case 1
      mov g[{0}.y]._y__, {2}.x
      break
    case 2
      mov g[{0}.y].__z_, {2}.x
      break
    case 3
      mov g[{0}.y].___w, {2}.x
      break
  endswitch
  ret
endfunc
```

Listato 5, la funzione globalScatter1

```
func 3
  dcl_literal l0, 0x1, 0x0, 0x0, 0x0
  and {0}.y, {0}.x, l0.x
  ishr {0}.x, {0}.x, l0.x
  if_logicalz {0}.y
    mov {1}, g[{0}.x].xyxy
  else
    mov {1}, g[{0}.x].zwzw
  endif
  ret
endfunc
```

Listato 6, la funzione globalGather2

```
func 4
  dcl_literal l0, 0x3, 0x2, 0x1, 0x0
  and {0}.y, {0}.x, l0.x
  ishr {0}.x, {0}.x, l0.y
  switch {0}.y
    default
      mov {1}, g[{0}.x].x
      break
    case 1
      mov {1}, g[{0}.x].y
```

```
      break
    case 2
      mov {1}, g[{0}.x].z
      break
    case 3
      mov {1}, g[{0}.x].w
      break
  endswitch
  ret
endfunc
```

Listato 7, la funzione globalGather2

A.4 Il sorgente del Mandelbrot prodotto da PBricks

```
dcl_cb cb0[3]
dcl_resource_id(0)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtx(float)_fmtw(f
loat)
dcl_resource_id(1)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtx(float)_fmtw(f
loat)
dcl_output_generic o0
sample_resource(0)_sampler(0) r0, v0
mov r1, r0
sample_resource(1)_sampler(1) r2, v0
mov r3, r2
mul r4, r3, cb0[0].x
div r5, r4, cb0[0].y
add r6, r5, cb0[0].z
mov r7, r6
mul r8, r1, cb0[0].w
div r9, r8, cb0[1].x
add r10, r9, cb0[1].y
mov r11, r10
mov r12, r11
mov r13, r7
mov r14, cb0[1].z
whileloop
breakc_relop(ge) r14, cb0[2].w
mul r15, r11, r11
mul r16, r7, r7
add r17, r16, r15
ifc_relop(ge) r17, cb0[1].w
mod r18, r14, cb0[2].x
add r19, r18, cb0[2].y
mov o0, r19
ret_dyn
else
mul r20, r11, r11
```

```
mul r21, r7, r7
sub r22, r21, r20
add r23, r22, r13
mov r24, r23
mul r25, cb0[2].z, r7
mul r26, r25, r11
add r27, r26, r12
mov r11, r27
mov r7, r24
endif
add r28, r14, cb0[2].y
mov r14, r28
endloop
mov o0, cb0[1].z
ret
end
```

Listato 8, il sorgente del Mandelbrot generato dalla compilazione

A.5 Costruzione tabella dei salti "while"

Nella modalità di compilazione debug per ogni ciclo è emessa sempre una coppia di istruzioni salto-incondizionato, salto-condizionato all'indietro (listato A.1).

```
while (current > 2)
{
    while (current > 1)
    {
        current--;
    }
}
```

```
L_0008: br.s L_001c
L_000a: nop
L_000b: br.s L_0013
L_000d: nop
L_000e: ldloc.0
L_000f: ldc.i4.1
L_0010: sub
L_0011: stloc.0
L_0012: nop
L_0013: ldloc.0
L_0014: ldc.i4.1
L_0015: cgt
L_0017: stloc.1
L_0018: ldloc.1
L_0019: brtrue.s L_000d
L_001b: nop
L_001c: ldloc.0
L_001d: ldc.i4.2
L_001e: cgt
```

```
L_0020: stloc.1
L_0021: ldloc.1
L_0022: brtrue.s L_000a
L_0024: ret
```

Listato A.1, compilazione *debug,* (salti e relativo target hanno medesimo colore)

Ogni salto condizionato ha il proprio target posto sempre a +2 (si guardino i valori esadecimali delle etichette) rispetto alla posizione del corrispondente salto incondizionato. Sfruttando questa considerazione la prima versione dell'algoritmo effettuava una scansione nel sorgente salvando la posizione di ogni salto incondizionato e verificando per ogni salto condizionato se il target di salto soddisfaccesse questa proprietà: in caso positivo la coppia di istruzioni traduceva un costrutto *while* (salvando in una tabella[39] la posizione la posizione del salto cond.), altrimenti *dowhile*.

In modalità *release* questa proprietà viene a mancare poiché il compilatore nel caso di cicli annidati ottimizza il sorgente riutilizzando in alcune situazioni casi il salto incondizionato. Di seguito sono riportati tre casi in cui questo avviene. Con lo stesso colore sono marcate le istruzioni di salto con il relativo target di salto.

A)

```
while (current > 2)
{
    while (current > 1)
    {
        current--;
    }
}
```

```
L_0007: br.s L_0011
L_0009: ldloc.0
L_000a: ldc.i4.1
L_000b: sub
L_000c: stloc.0
L_000d: ldloc.0
L_000e: ldc.i4.1
L_000f: bgt.s L_0009
L_0011: ldloc.0
L_0012: ldc.i4.2
L_0013: bgt.s L_000d
```

B)

```
while (current > 2)
{
    do
    {
        current--;
    }
```

[39] La tabella usata dal meta-programma per decidere il `branchType` di un `NodeCtrlFlow`.

```
        while (current < 0);
    }

L_0007: br.s L_0011
L_0009: ldloc.0
L_000a: ldc.i4.1
L_000b: sub
L_000c: stloc.0
L_000d: ldloc.0
L_000e: ldc.i4.0
L_000f: blt.s L_0009
L_0011: ldloc.0
L_0012: ldc.i4.2
L_0013: bgt.s L_0009
```

C)

```
    do
    {
        while (current > 0)
        {
            current--;
        }
    }
    while (current < 0);

L_0007: br.s L_000d
L_0009: ldloc.0
L_000a: ldc.i4.1
L_000b: sub
L_000c: stloc.0
L_000d: ldloc.0
L_000e: ldc.i4.0
L_000f: bgt.s L_0009
L_0011: ldloc.0
L_0012: ldc.i4.0
L_0013: blt.s L_000d
```

La differenza sta nel target del salto incondizionale `br.s`:

A. Tra i due branch condizionali
B. Tra i due branch condizionali, ma in questo caso i target dei salti condizionali coincidono
C. In posizione precedente ad entrambi i salti condizionali e coincidente al target del salto condizionale che esprime la guardia del ciclo esterno

Per gestire anche questi tre casi l'algoritmo è stato modificato in modo tale da verificare:

A. Se il target di un salto condizionato ricade in un posizione compresa tra gli estremi (coppie di posizioni di salti incondizionati/condizionati) di un ciclo rilevato in precedenza, allora anche questo costituisce un ciclo *while*;

B. Se un successivo salto condizionato ha il target nuovamente posto a +2 rispetto alla posizione del salto incondizionato facente parte di un ciclo precedentemente rilevato, allora questo viene considerato di tipo *do-while* e viene invece aggiunto l'ultimo tra i cicli *while*;
C. Se si verifica la condizione A ma il target del salto condizionato in esame coincide con quello del salto incondizionato allora è considerato come *do-while*.

In aggiunta si è poi scoperto che non sempre i cicli hanno bisogno di un salto incondizionato per essere espressi: nel seguente caso il ciclo while più esterno manca dell'istruzione `br`, però a farne le veci c'è un'istruzione `ret` posta in posizione relativa -1 rispetto al target del salto condizionato all'indietro:

```
if (current > 4)
{
    current++;
}
else
{
    while (current > 2)
    {
        current--;

        while (current == 2)
        {
            current = 3;
        }
    }
}
```

```
L_0009: ble.s L_001c
L_000b: ldloc.0
L_000c: ldc.i4.1
L_000d: add
L_000e: stloc.0
L_000f: ret
L_0010: ldloc.0
L_0011: ldc.i4.1
L_0012: sub
L_0013: stloc.0
L_0014: br.s L_0018
L_0016: ldc.i4.3
L_0017: stloc.0
L_0018: ldloc.0
L_0019: ldc.i4.2
L_001a: beq.s L_0016
L_001c: ldloc.0
L_001d: ldc.i4.2
L_001e: bgt.s L_0010
L_0020: ret
```

Questo avviene perchè lo statement *while* è la prima istruzione del ramo *else*; aggiungendo infatti uno statement qualsiasi prima del ciclo, viene generata l'istruzione `br` che ci si attenderebbe. L'algoritmo è stato esteso in modo tale da includere anche questi casi, cercando non solo salti incondizionati posti in

posizione relativa -2 rispetto al target del salto condizionato in esame, ma anche istruzioni `ret` poste in posizione relativa -1.

Un'ulteriore differenza tra la compilazione debug e quella release (oltre alla banale eliminazione delle istruzioni `nop`) è che mentre in release vengono utilizzate istruzioni `ret` anche all'interno dei costrutti di selezione. Questo ha complicato principalmente la fase di *fix*.

A.6 Goto e Switch

La mancanza di istruzioni di salto nel CAL IL ha avuto come prima conseguenza l'impossibilità di tradurre programmi di alto livello nei quali si faccia uso di istruzioni *goto*:

```
public void KernelWithGoto(InputStream<int> source, OutputStream<int> dest)
{
    int current = source.Current;
    int result = 0;

test:
    if (current >= 10)
    {
        goto decr;
    }
    else
    {
        dest.Write(result);
        return;
    }

decr:
    result++;
    current /= 2;
    goto test;
}
```

Questo perchè a differenza di selezione e iterazione, che è stato possibile tradurre ricostruendo la struttura di alto livello, nel caso di *goto* sarebbe stata necessaria una trasformazione stessa del programma di alto livello in uno semanticamente equivalente (laddove questo fosse stato possibile), riconducibile a casi già trattati per poi darne una traduzione effettiva.
Anche il costrutto di *switch* è stato trattato nella medesima maniera (nonostante nel CAL IL sia presente un'istruzione di switch), perché a causa di ottimizzazioni del compilatore di alto livello non è stato possibile individuare delle proprietà che generalizzassero i diversi casi possibili. Si considerino a riguardo i seguenti esempi:

```
public void KernelWithSwitch(InputStream<int> source, OutputStream<int> dest)
{
      switch (source.Current)
      {
            case 1:
                dest.Write(0);
                break;
```

```
            case 2:
                dest.Write(1);
                break;
        }
}
```

Versione debug:

```
.method public hidebysig instance void KernelWithSwitch(class
[Core]PBricks.Core.InputStream`1<int32> source, class
[Core]PBricks.Core.OutputStream`1<int32> dest) cil managed
{
    .maxstack 2
    .locals init (
        [0] int32 CS$4$0000)
    L_0000: nop
    L_0001: ldarg.1
    L_0002: callvirt instance !0
[Core]PBricks.Core.InputStream`1<int32>::get_Current()
    L_0007: stloc.0
    L_0008: ldloc.0
    L_0009: ldc.i4.1
    L_000a: sub
    L_000b: switch (L_001a, L_0024)
    L_0018: br.s L_002e
    L_001a: ldarg.2
    L_001b: ldc.i4.0
    L_001c: callvirt instance void
[Core]PBricks.Core.OutputStream`1<int32>::Write(!0)
    L_0021: nop
    L_0022: br.s L_002e
    L_0024: ldarg.2
    L_0025: ldc.i4.1
    L_0026: callvirt instance void
[Core]PBricks.Core.OutputStream`1<int32>::Write(!0)
    L_002b: nop
    L_002c: br.s L_002e
    L_002e: ret
}
```

Versione release:

```
.method public hidebysig instance void KernelWithSwitch(class
[Core]PBricks.Core.InputStream`1<int32> source, class
[Core]PBricks.Core.OutputStream`1<int32> dest) cil managed
{
    .maxstack 2
    .locals init (
        [0] int32 CS$0$0000)
    L_0000: ldarg.1
    L_0001: callvirt instance !0
[Core]PBricks.Core.InputStream`1<int32>::get_Current()
    L_0006: stloc.0
    L_0007: ldloc.0
    L_0008: ldc.i4.1
```

```
    L_0009: sub
    L_000a: switch (L_0018, L_0020)
    L_0017: ret
    L_0018: ldarg.2
    L_0019: ldc.i4.0
    L_001a: callvirt instance void
[Core]PBricks.Core.OutputStream`1<int32>::Write(!0)
    L_001f: ret
    L_0020: ldarg.2
    L_0021: ldc.i4.1
    L_0022: callvirt instance void
[Core]PBricks.Core.OutputStream`1<int32>::Write(!0)
    L_0027: ret
}
```

In modalità debug utilizza l'istruzione `switch` e per ogni *case* dello switch un salto incondizionato, mentre in modalità release è usata l'istruzione `ret` per forzare l'uscita.
Inoltre, modificando i valori dei *case* si ottiene invece:

```
public void KernelWithSwitch(InputStream<int> source, OutputStream<int> dest)
{
      switch (source.Current)
      {
            case 3:
                dest.Write(0);
                break;

            case 7:
                dest.Write(1);
                break;
      }
}
```

Versione debug:

```
.method public hidebysig instance void KernelWithSwitch(class
[Core]PBricks.Core.InputStream`1<int32> source, class
[Core]PBricks.Core.OutputStream`1<int32> dest) cil managed
{
    .maxstack 2
    .locals init (
        [0] int32 CS$4$0000)
    L_0000: nop
    L_0001: ldarg.1
    L_0002: callvirt instance !0
[Core]PBricks.Core.InputStream`1<int32>::get_Current()
    L_0007: stloc.0
    L_0008: ldloc.0
    L_0009: ldc.i4.3
    L_000a: beq.s L_0012
    L_000c: ldloc.0
    L_000d: ldc.i4.7
    L_000e: beq.s L_001c
```

```
    L_0010: br.s L_0026
    L_0012: ldarg.2
    L_0013: ldc.i4.0
    L_0014: callvirt instance void
[Core]PBricks.Core.OutputStream`1<int32>::Write(!0)
    L_0019: nop
    L_001a: br.s L_0026
    L_001c: ldarg.2
    L_001d: ldc.i4.1
    L_001e: callvirt instance void
[Core]PBricks.Core.OutputStream`1<int32>::Write(!0)
    L_0023: nop
    L_0024: br.s L_0026
    L_0026: ret
}
```

Utilizza per ogni *case* dello switch una coppia di istruzioni `beq.s, br.s.`

Versione release:

```
.method public hidebysig instance void KernelWithSwitch(class
[Core]PBricks.Core.InputStream`1<int32> source, class
[Core]PBricks.Core.OutputStream`1<int32> dest) cil managed
{
    .maxstack 2
    .locals init (
        [0] int32 CS$0$0000)
    L_0000: ldarg.1
    L_0001: callvirt instance !0
[Core]PBricks.Core.InputStream`1<int32>::get_Current()
    L_0006: stloc.0
    L_0007: ldloc.0
    L_0008: ldc.i4.3
    L_0009: beq.s L_0010
    L_000b: ldloc.0
    L_000c: ldc.i4.7
    L_000d: beq.s L_0018
    L_000f: ret
    L_0010: ldarg.2
    L_0011: ldc.i4.0
    L_0012: callvirt instance void
[Core]PBricks.Core.OutputStream`1<int32>::Write(!0)
    L_0017: ret
    L_0018: ldarg.2
    L_0019: ldc.i4.1
    L_001a: callvirt instance void
[Core]PBricks.Core.OutputStream`1<int32>::Write(!0)
    L_001f: ret
}
```

Utilizza per ogni *case* dello switch una coppia di istruzioni `beq.s, ret.`

In definitiva per compilare anche il costrutto di switch si sarebbe finito col complicare enormemente la gestione dei precedenti casi, per giunta per una motivazione (la mancanza di istruzioni di salto) che ci

auguriamo venga presto corretta da AMD. Inoltre lo scopo che ci eravamo proposti all'inizio non era tanto quello di coprire l'intera gamma di programma esprimibili (eventualmente trattabili in futuro) quanto di dimostrare la fattibilità del mapping tra GPU e virtual machine.

A.7 Esempio di struttura prima della fase di Scan

```
Root
|
|-> NodeStloc "loc0"
    |-> NodeCall "get_Current"

|-> NodeStloc "loc1"
    |-> NodeConst "cb0[0].x"

|-> NodeCondBranch IF "Brtrue"
    |-> guardNodes
        |-> NodeLdLoc "loc1"

    |-> TARGET
        |-> NodeCondBranch IF "Ble"
            |-> guardNodes
                |-> NodeConst "cb0[0].x"
                |-> NodeLdLoc "loc1"

            |-> TARGET
                |-> NodeStloc "loc3"
                |-> NodeCall "Write"
                    |-> NodeLdloc "loc3"
                    |-> NodeArg "o0"

            |-> SEQ
                |-> NodeCondBranch DOWHILE "Bgt"
                    |-> guardNodes
                        |-> NodeConst "cb0[0].x"
                        |-> NodeLdloc "loc0"

                    |-> TARGET
                        |-> NodeStloc "loc0"
                        |-> NodeStloc "loc1"

                    |-> SEQ
                        |-> {}

                |-> NodeUncondBranch ELSE

    |-> SEQ
        |-> NodeStloc "loc2"
        |-> NodeCall "Write"
```

```
            |-> NodeLdloc “loc2”
            |-> NodeArg “o0”

        |-> NodeUncondBranch ELSE

|-> NodeCall “Write”
    |-> NodeLdloc “loc1”
    |-> NodeArg “o0”

|-> NodeRet
```

Bibliografia

[1] K. Czarnecki, U.V. Eisenacker, Generative Programming – Methods, Tools and Applications, Addison Wesley, Reading MA, (2000)

[2] S. Lidin, Inside Microsoft .NET IL assembler, Microsoft Press (2002)

[3] A. Cisternino, Multi-stage and Meta-programming support in strongly typed execution engines, tesi di dottorato di ricerca (2003)

[4] Gamma, Helm, Johnson & Vlissides, Design Patterns (the Gang of Four book), Addison-Wesley (1994)

[5] MSDN Microsoft Developer Network Web Site: http://msdn.microsoft.com/

[6] ECMA 335, Common Language Infrastructure (CLI) specification, http://www.ecma.ch/ecma1/STAND/ecma-335.htm

[7] J. Richter: Applied Microsoft .NET framework programming, Microsoft Press (2002)

[8] Krall, A., Efficient JavaVM Just-in-Time Compilation, International Conference on Parallel Architectures and Compilation Techniques, edited by Jean-Luc Gaudiot, North-Holland, Paris, pp. 205-212 (1998)

[9] ECMA 334, C# specification, http://www.ecma.ch/ecma1/STAND/ecma-334.htm

[10] ISO/IEC 23271, Information technology -- Common Language Infrastructure, available at http://www.iso.org/

[11] Lindholm, T., and Yellin, F., The Java™ Virtual Machine Specification, Second Edition, Addison-Wesley (1999)

[12] OCaml VM, http://pauillac.inria.fr/~lebotlan/docaml_html/english/

[13] Python VM, http://www.python.org/

[14] TEA VM, http://www.acroname.com/brainstem/brainstem.html

[15] XSLTVM, http://www.gca.org/papers/xmleurope2000/papers/s35-03.html

[16] Parrot VM, http://www.parrot.org/

[17] Pereira, and D., Aycock, J., Instruction set Architecture of Mamba, a new Virtual Machine for Python, Technical Report 2002-706-09, Department of Computer Science, The University of Calgary, http://pharos.cpsc.ucalgary.ca/Dienst/UI/2.0/Describe/ncstrl.ucalgary_cs/2002-706-09

[18] Y.Shi, D.Gregg, A.Beatty, M.A.Ertl, Virtual Machine Showdown: Stack Versus Registers, ACM (2005)

[19] S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kauffman (1997)

[20] D. Box, C. Sells, Essential .NET volume 1: The common language runtime, Addison Wesley (2002)

[21] Microsoft Parallel Extensions to the .NET Framework 3.5 June 2008 Community Technology Preview (CTP) http://www.microsoft.com/downloads/details.aspx?FamilyId=348F73FD-593D-4B3C-B055-694C50D2B0F3&displaylang=en

[22] LINQ to Objects, http://www.hookedonlinq.com/LINQtoObjects5MinuteOverview.ashx

[23] Zheng, Bixia, Derek Gladding and Micah Villmow. Building a High Level Language Compiler for GPGPU. Tucson, Arizona: ACM SIGPLAN conference (2008)

[24] AMD**,** *Compute Abstraction Layer (CAL) Technology, Intermediate Language (IL) Reference Manual.* s.l. : AMD (2007)

[25] AMD, *AMD Compute Abstraction Layer Programming Guide v.1.0 beta.* s.l. (2008)

[26] Nvidia, GeForce GTX 280. http://www.Nvidia.com/object/geforce_gtx_280.html

[27] Nvidia GT200 GPU and Architecture Analysis. *Beyond3D.* http://www.beyond3d.com/content/reviews/51/1

[28] Abi-Chahla, Fedy, Charpentier, Florian**.** Nvidia GeForce GTX 260/280 Review. *Tom's Hardware.* http://www.tomshardware.com/reviews/Nvidia-gtx-280,1953.html

[29] Nvidia, *Nvidia CUDA - Compute Unified Device Architecture, Programming guide v.2.3.1.* Santa Clara, CA 95050 : Nvidia Corporation, (2009)

[30] AMD, *Brook+ Language Specification v1.0 beta. (*2008)

[31] Buck, Ian, *High Performance Computing with CUDA.* USA : SC07, 2007. SUPERCOMPUTING 2007 CUDA Tutorial

[32] AMD, AMD ShaderAnalyzer. *AMD Developer Central.* AMD, 2008. http://developer.amd.com/gpu/shader/Pages/default.aspx

[33] Nvidia, Fermi Compute Architecture Whitepaper (2009)

[34] A. Cisternino, C. Dittamo, GPU White Paper (2008)

[35] D. Buono, Caratteristiche della programmazione di applicazioni context-aware e una proposta di modello ad alte prestazioni (2009)

[36] S.Orlando, Introduzione “Calcolo ad Alte Prestazioni”, http://www.dsi.unive.it/~calpar/1_Intro.pdf

[37] Intel Ct http://software.intel.com/en-us/data-parallel/

[38] M. McCool, The RapidMind Platform for Portable Programming of Multi-Core Processors and Many-Core Accelerators, SHARCnetSymposium (2008)

[39] J. Palsberg, RapidMind http://www.cs.ucla.edu/~palsberg/course/cs239/F07/slides/rapidmind.pdf

[40] D. Tarditi, S. Puri, J. Oglesby, Accelerator, Using Data Parallelism to Program GPUS for General-Purpose Uses, no. MSR-TR-2005-184, (2006)

[41] DirectCompute, http://openvidia.sourceforge.net/index.php/DirectCompute

[42] The OpenCL Specification Version: 1.0 Document Revision: 48, Khronos OpenCL Working Group

[43] D. Leijen, J. Hall, Optimize Managed Code For Multi-Core Machines, http://msdn.microsoft.com/en-us/magazine/cc163340.aspx

[44] A. D. Falkoff, K.E. Iverson, The Design of APL (1973)

[45] Mono, http://www.mono-project.com/Main_Page

[46] C. Dittamo, Tecniche di parallelizzazione di programmi sequenziali basate su annotazioni, tesi (2005)

[47] C. Angelini, F. Abi-Chahla, Radeon HD 5870: DirectX 11, Eyefinity e tanta potenza, Tom's Hardware, (2009)

[48] AMD, Stream Computing User Guide, (2009)

[49] F. Baiardi, A. Tomasi, M. Vanneschi, Architettura dei Sistemi di Elaborazione – Vol II, F. Angeli, Milano (1991)

[50] World Community Grid, http://www.worldcommunitygrid.org

[51] Lista dei progetti di calcolo distribuito, http://it.wikipedia.org/wiki/Lista_dei_progetti_di_calcolo_distribuito

[52] J. Joseph, C. Fellenstein, Grid Computing, IBM (2003)

[53] D.K.G. Campbell, A Survey of Models of Parallel Computation, University of York (1997)

[54] M. Vanneschi, Architetture parallele e distribuite. SEU, 2008-09.

[55] C. Dittamo, On expressing different concurrency paradigms on virtual execution systems, (2009)

[56] A. Cisternino, C. Dittamo, Filling the gap between GPGPUs and Virtual Machine Computational Models (2009)

[57] K. Matsuzaki, H. Iwasaki, K. Emoto, Z. Hu, A library of constructive skeletons for sequential style of parallel programming. In InfoScale '06: Proceedings of the 1st international conference on Scalable information systems, page 13, New York, NY, USA, 2006. ACM.

[58] M. Danelutto, M. Aldinucci, P. Dazzi, Muskel: a skeleton library supporting skeleton set expandability. Scalable Computing: Practice and Experience, 8(4):325_341, (2007)

[59] R. Di Cosmo, Z. Li, X. Leroy, S. Pelagatti. Skeletal parallel programming with ocamlp3l 2.0. In Parallel Processing Letters, volume 1, pagg. 1-13, (2006).

[60] M. Vanneschi. The programming model of assist, an environment for parallel and distributed portable applications. Parallel Computing, 28(12):1709-1732, (2002).

[61] 4-Centauri, http://www.codeplex.com/4centauri

[62] A. Gulli, Survey sugli algoritmi e sulle architetture usate dai search engine

[63] Computer-generated imagery, http://en.wikipedia.org/wiki/Computer-generated_imagery