C

APPENDICE

La grafica e il calcolo con la GPU

A cura di John Nickolls, direttore di ricerca di NVIDIA, e David Kirk, direttore di progetto di NVIDIA

L'immaginazione è più importante della conoscenza.
Albert Einstein, On Science, anni Trenta.

C.1 Introduzione

Unità di elaborazione grafica (GPU): processore ottimizzato per la grafica 2D e 3D, la riproduzione video e l'elaborazione visuale.

Elaborazione visuale (*visual computing*): combinazione di elaborazione grafica e calcolo che permette all'utente di interagire visivamente con i risultati dell'elaborazione computazionale attraverso grafici, immagini e video.

Sistema eterogeneo: un sistema che combina differenti tipi di processore. Un PC è un sistema eterogeneo CPU–GPU.

Questa appendice tratta la **GPU**, ossia l'**unità di elaborazione grafica** (*graphics processing unit*), presente in ogni PC, calcolatore portatile o desktop, e in ogni workstation. Nella sua forma più semplice, la GPU permette di visualizzare grafici 2D e 3D, immagini e video e rende possibili i sistemi operativi a finestre, le interfacce utente di tipo grafico, i videogiochi, le applicazioni visuali e la rappresentazione di sequenze video. La GPU moderna qui descritta è un multiprocessore altamente parallelo e multithread, ottimizzato per l'**elaborazione visuale**. Per fornire un'interazione visuale in tempo reale con i dati elaborati attraverso grafica, immagini e video, la GPU è basata su un'architettura unificata di grafica e calcolo che funge sia da processore grafico programmabile sia da piattaforma di calcolo parallela e scalabile. Nei PC e nelle console per videogiochi la GPU viene aggiunta alla CPU, quindi forma un **sistema eterogeneo**.

Breve storia delle GPU

Quindici anni fa le GPU non esistevano. I grafici sui PC venivano generati da un controllore vettoriale di video e grafica (VGA, *Video and Graphics Array controller*). Si trattava semplicemente di un controllore della memoria e di un generatore dei segnali per i terminali grafici connessi a una memoria DRAM. Negli anni Novanta la tecnologia dei semiconduttori si era evoluta a tal punto da permettere di aggiungere altre funzioni al controllore VGA. Già nel 1997 i controllori VGA cominciavano a incorporare alcune funzioni di accelerazione

della grafica tridimensionale (3D), come la preparazione dei triangoli, la *rasterizzazione* (la suddivisione dei triangoli nei singoli pixel in essi contenuti), l'applicazione della tessitura (cioè l'applicazione di motivi grafici ai triangoli) e lo *shading* (l'applicazione delle tonalità di colore).

Nel 2000 il chip di un processore grafico incorporava quasi tutti gli elementi di una pipeline grafica tipica delle tradizionali workstation di fascia alta e per questo si meritò un nome nuovo, che indicasse un dispositivo che si spingeva oltre le funzionalità di base di un controllore VGA. Il termine «GPU» fu coniato per sottolineare che il dispositivo grafico era diventato un processore vero e proprio.

Col passare del tempo le GPU divennero più programmabili, perché i processori programmabili sostituirono i circuiti logici (in grado di generare solamente le funzioni prefissate), pur mantenendo l'organizzazione di base della pipeline grafica 3D. In aggiunta, i calcoli diventarono man mano più precisi, passando dall'aritmetica indicizzata all'aritmetica intera e decimale in virgola fissa, in seguito all'aritmetica in virgola mobile in singola precisione e, recentemente, all'aritmetica in virgola mobile in doppia precisione. Le GPU sono diventate processori programmabili ad alto livello di parallelismo, con centinaia di unità di elaborazione (core) e migliaia di thread.

Recentemente ai processori sono state aggiunte istruzioni e strutture hardware di memoria per poter supportare linguaggi di programmazione di utilizzo generale; è stato inoltre implementato un ambiente di programmazione per consentire di programmare le GPU utilizzando linguaggi familiari, come C e C++. Questa innovazione ha reso le GPU dei processori multicore completamente programmabili e di utilizzo generale, seppure con certe limitazioni.

Tendenze delle GPU grafiche

Le GPU e i driver a loro associati implementano i modelli di elaborazione grafica OpenGL e DirectX. OpenGL è uno standard aperto di programmazione grafica 3D disponibile per la maggior parte dei calcolatori. DirectX è un insieme di interfacce di programmazione multimediale di Microsoft che include Direct3D per la grafica tridimensionale. Poiché queste **interfacce di programmazione delle applicazioni** (**API**, *Application Programming Interfaces*) hanno un comportamento ben definito, è possibile ottenere un'efficace accelerazione hardware delle funzioni di elaborazione grafica definite. Questa è una delle ragioni (oltre alla crescente densità dei dispositivi) per cui ogni 12–18 mesi vengono sviluppate nuove GPU con prestazioni doppie rispetto alla generazione precedente.

Il continuo miglioramento delle prestazioni delle GPU permette di creare applicazioni che in passato non erano possibili. L'intersezione tra elaborazione grafica e calcolo parallelo ha portato allo sviluppo di un nuovo paradigma per la grafica, conosciuto come *elaborazione visuale*. In questo paradigma, parti consistenti del modello tradizionale della pipeline grafica hardware sequenziale vengono sostituite con elementi programmabili per la geometria, i vertici e i pixel. L'elaborazione visuale in una GPU moderna combina elaborazione grafica e calcolo parallelo secondo nuove modalità, che permettono di implementare algoritmi diversi e aprono la strada a una nuova generazione di applicazioni che sfruttano l'elaborazione parallela su GPU ad alte prestazioni.

Interfaccia di programmazione delle applicazioni (API): un insieme di definizioni di funzioni e strutture dati che fornisce un'interfaccia a una libreria di funzioni.

Sistemi eterogenei

Benché la GPU all'interno di un comune PC sia il processore più potente e con il più alto grado di parallelismo, essa non può essere l'unico processore. La

CPU, attualmente multicore (e presto con un numero di core molto più elevato di quello odierno), è un processore principalmente seriale che si affianca a una GPU con moltissimi core e massicciamente parallela. Insieme, questi due tipi di processore costituiscono un sistema eterogeneo multiprocessore.

Per molte applicazioni, le prestazioni migliori si ottengono utilizzando sia la CPU sia la GPU. Questa appendice ha lo scopo di aiutarvi a comprendere quando e come ripartire al meglio il lavoro tra questi due processori caratterizzati da un livello di parallelismo sempre più elevato.

Le GPU evolvono verso i processori paralleli scalabili

Dal punto di vista funzionale, le GPU si sono evolute dai controllori VGA cablati, di limitate capacità, ai processori paralleli programmabili. Questa evoluzione è avvenuta modificando la pipeline grafica logica (basata su API) sia incorporando elementi programmabili sia rendendo gli stadi della pipeline hardware sottostante meno specializzati e più programmabili. Alla fine di questo processo, è risultato conveniente unire i diversi elementi della pipeline programmabile in un'unica schiera di svariati processori programmabili.

Nella generazione di GPU GeForce, serie 8, l'elaborazione della geometria, dei vertici e dei pixel è effettuata sullo stesso tipo di processore. Questa unificazione permette una scalabilità estrema, e la presenza di più core programmabili aumenta le prestazioni globali del sistema. Inoltre, l'unificazione dei processori fornisce anche un bilanciamento molto efficace del carico, poiché ogni funzione di calcolo può usare l'intero insieme di processori. Per contro, oggi un insieme di processori può essere costruito con pochissimi processori, poiché tutte le funzioni possono essere eseguite sulle stesse unità.

Perché CUDA e perché utilizzare le GPU per il calcolo?

Questa schiera uniforme e scalabile di processori suggerisce un nuovo modello di programmazione delle GPU. La consistente potenza di calcolo in virgola mobile offerta ha reso le GPU appetibili per risolvere problemi non grafici. Dato il notevole grado di parallelismo e di scalabilità della schiera dei processori per le applicazioni grafiche, il modello di programmazione per il calcolo più generale deve esprimere direttamente il massiccio parallelismo, consentendo comunque un'esecuzione scalabile.

Calcolo su GPU (*GPU computing*) è il termine coniato per definire l'utilizzo della GPU per l'elaborazione attraverso un linguaggio di programmazione parallelo e particolari API, senza ricorrere alle API grafiche tradizionali e al modello della pipeline grafica. Ciò è in contrasto con il precedente approccio, ossia l'**elaborazione per utilizzo generale su GPU** (**GPGPU**, *General Purpose computation on GPU*), che consiste nel programmare le GPU utilizzando le API e la pipeline grafiche per svolgere compiti non grafici.

CUDA (*Compute Unified Device Architecture*) è un modello di programmazione parallela scalabile e una piattaforma software per la GPU e per altri processori paralleli che permette al programmatore di non dover utilizzare le API grafiche e le interfacce grafiche della GPU, ma di programmare direttamente in C o C++. Il modello di programmazione CUDA ha un approccio SPMD («singolo programma, dati multipli»), in cui il programmatore scrive un programma per un thread e questo programma viene istanziato ed eseguito da molti thread in parallelo sui molteplici processori della GPU. Di fatto, CUDA fornisce anche un mezzo per programmare core multipli di CPU, per cui è un ambiente che permette di scrivere programmi paralleli per l'intero sistema eterogeneo che costituisce il calcolatore.

Calcolo su GPU (*GPU computing*): utilizzo della GPU per effettuare calcoli mediante linguaggi di programmazione parallela e API.

GPGPU: utilizzo di una GPU per calcoli generici attraverso le API grafiche e la pipeline grafica tradizionale.

CUDA: modello e linguaggio di programmazione parallela scalabile basato su C/C++. È una piattaforma di programmazione parallela per le GPU e le CPU multicore.

La GPU unifica grafica e calcolo

Con l'aggiunta di CUDA e delle capacità di calcolo delle GPU, è ora possibile utilizzare contemporaneamente la GPU sia come processore grafico sia come processore di calcolo e combinare queste due modalità di utilizzo in applicazioni di elaborazione visuale. L'architettura di elaborazione sottostante la GPU può essere descritta in due modi: attraverso l'implementazione di API grafiche programmabili e come un insieme di processori con un alto grado di parallelismo programmabili in C/C++ con CUDA.

Nonostante i processori elementari della GPU siano tutti uguali, non è necessario che i programmi dei thread SPMD siano gli stessi. La GPU può eseguire programmi di elaborazione grafica tipici di una GPU, chiamati comunemente *shader*, sui diversi elementi della grafica (geometria, vertici e pixel), ma anche eseguire thread di programmi diversi in CUDA.

La GPU è un'architettura multiprocessore realmente versatile, in grado di effettuare una grande varietà di elaborazioni. Le GPU eccellono nell'elaborazione grafica e visuale, essendo state progettate specificamente per questo tipo di applicazione, ma risultano eccellenti anche per molte altre applicazioni ad alte prestazioni di tipo generale che sono «cugine prime» delle applicazioni grafiche, in quanto devono svolgere una grande quantità di calcoli in parallelo e si basano su molte strutture regolari. In generale, le GPU sono particolarmente adatte a problemi caratterizzati da un elevato livello di parallelismo nei dati (si veda il Capitolo 7) e a problemi di dimensioni assai rilevanti, mentre sono meno indicate per problemi più piccoli e meno regolari.

Applicazioni di elaborazione visuale su GPU

L'elaborazione visuale comprende le diverse tipologie di applicazioni grafiche tradizionali e molte applicazioni nuove. L'ambito originario di una GPU era «qualsiasi cosa fosse collegato ai pixel», mentre ora include molte applicazioni non basate sui pixel ma su calcoli e/o strutture dati regolari. Le GPU sono efficaci per la grafica 2D e 3D, essendo questo lo scopo per cui sono state progettate. La grafica 2D e 3D, come abbiamo già detto, utilizza la GPU nella sua «modalità grafica», accedendo alla potenza di calcolo della GPU attraverso le API grafiche OpenGL e DirectX. I videogiochi sono basati sulla capacità di elaborazione grafica 3D.

Oltre alla grafica 2D e 3D, altre applicazioni importanti per la GPU sono l'elaborazione di immagini e video. Queste possono essere implementate utilizzando le API grafiche, oppure attraverso programmi di calcolo (impiegando CUDA per programmare la GPU nella modalità di calcolo). Con CUDA l'elaborazione delle immagini diventa semplicemente l'esecuzione di programmi vettoriali su dati paralleli. Nella misura in cui l'accesso ai dati è regolare e presenta buone caratteristiche di località, il programma sarà efficiente. L'elaborazione delle immagini di fatto è un'applicazione molto adatta alle GPU. L'elaborazione video, specialmente la codifica e decodifica (compressione e decompressione mediante algoritmi standard), è anch'essa particolarmente efficiente.

La caratteristica più importante per le applicazioni di elaborazione visuale su GPU è di poter «rompere la pipeline grafica». Le prime GPU implementavano soltanto API specifiche per la grafica, sebbene ad altissime prestazioni. Ciò andava bene se le API dovevano supportare solamente le operazioni desiderate; in caso contrario, la GPU non era in grado di accelerare l'esecuzione, perché le sue funzionalità erano prefissate e immutabili. Ora, con l'avvento di CUDA e del calcolo su GPU, le GPU possono essere programmate per implementare una pipeline virtuale differente semplicemente scrivendo un

programma CUDA che descriva il calcolo e il flusso dati desiderato. Qualsiasi applicazione è quindi ora possibile, e ciò sarà di stimolo per tentare nuovi approcci all'elaborazione visuale.

C.2 Le architetture dei sistemi GPU

In questo paragrafo prenderemo in considerazione le architetture dei sistemi GPU attualmente più diffusi: esamineremo le configurazioni di questi sistemi, le funzioni e i servizi offerti, le interfacce standard di programmazione e l'architettura interna di base delle GPU.

Architettura del sistema eterogeneo CPU-GPU

L'architettura di un sistema di calcolo eterogeneo che utilizza una GPU e una CPU può essere descritta ad alto livello prendendo in considerazione due caratteristiche principali: anzitutto il numero di sottosistemi funzionali e/o chip che sono utilizzati e il tipo di tecnologia e topologia di interconnessione; la seconda caratteristica è rappresentata dai sottosistemi di memoria disponibili per questi sottosistemi funzionali. Si veda il Capitolo 6 per una descrizione generale dei sistemi di I/O e dei chipset dei PC.

Il PC storico (1990 circa)

La Figura C.2.1 mostra un diagramma a blocchi ad alto livello di un PC degli anni Novanta. Il north bridge (si veda il capitolo 6) contiene le interfacce a banda larga che connettono la CPU, la memoria e il bus PCI. Il south bridge contiene interfacce e dispositivi ormai obsoleti: il bus ISA (audio, LAN), il controllore degli interrupt, il controllore DMA, il temporizzatore/contatore. In questo sistema, il terminale grafico veniva pilotato da un semplice sottosistema con memoria video (*framebuffer*) noto come VGA (*video graphics array*) che era collegato al bus PCI. Sottosistemi grafici con moduli di elaborazione integrati (GPU) non esistevano nei PC del 1990.

La figura C.2.2 illustra due configurazioni attualmente assai diffuse. Queste sono caratterizzate da una GPU (GPU discreta) e una CPU separate,

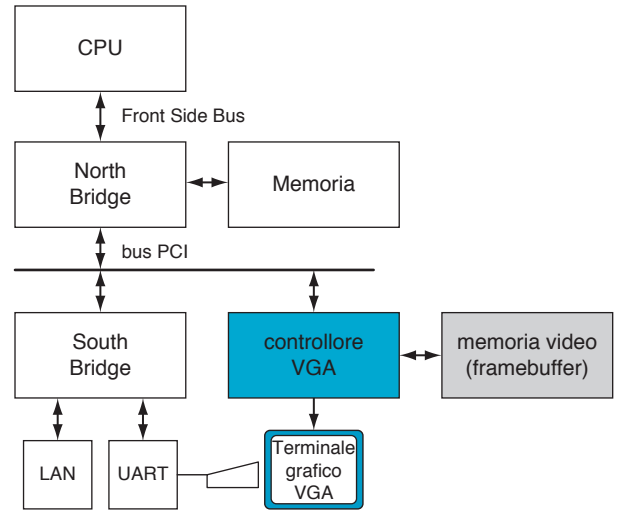


Figura C.2.1. PC storico. Il controllore VGA gestisce la visualizzazione grafica mediante la memoria video (frame buffer).

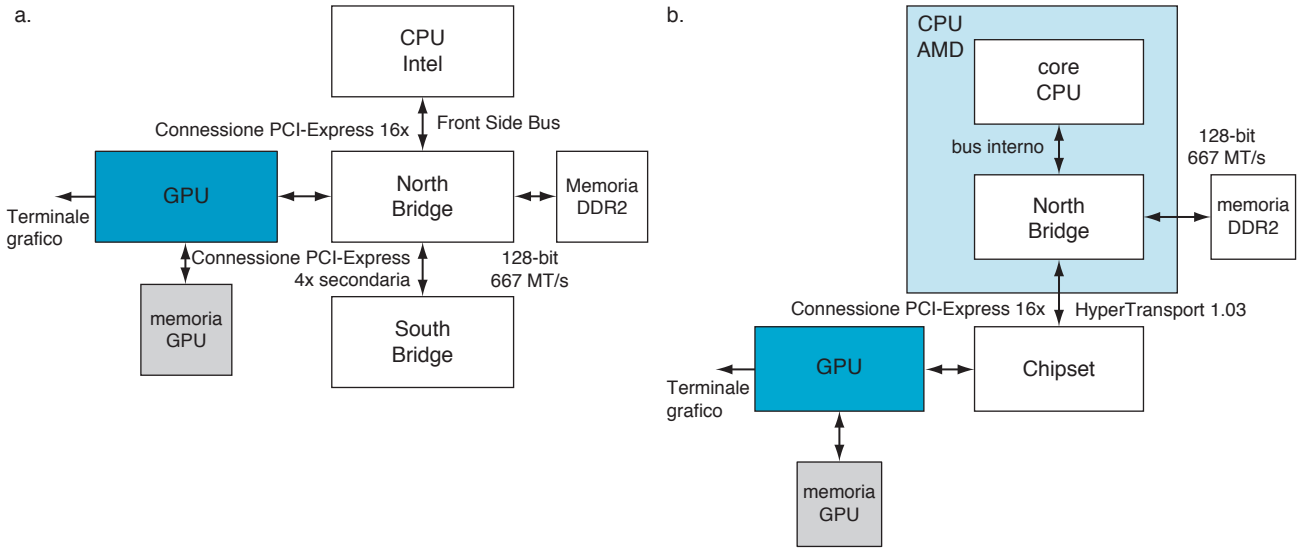


Figura C.2.2. PC contemporanei con CPU Intel e AMD. Si veda il Capitolo 6 per una descrizione dei componenti e delle interconnessioni riportate in figura.

con i rispettivi sottosistemi di memoria. Nella figura C.2.2a è riportata lo schema di una CPU Intel, alla quale è connessa la GPU mediante un collegamento **PCI-Express** 2.0 a 16 vie che fornisce una velocità di trasferimento di picco di 16 GB/s (8 GB/s per ogni direzione). Analogamente, nella figura C.2.2b è mostrato lo schema di una CPU AMD nella quale la GPU è connessa al chipset, anche in questo caso mediante PCI-Express, con la stessa larghezza di banda. In entrambi i casi, GPU e CPU possono accedere alla memoria l'una dell'altra, seppure con una larghezza di banda di trasferimento minore che nel caso di accesso diretto. Nel sistema AMD, il north bridge, o controllore di memoria, è integrato sullo stesso chip della CPU.

Una variante a basso costo di questi sistemi è rappresentata dai sistemi ad **architettura di memoria unificata** (**UMA**, *Unified Memory Architecture*), in cui non è presente la memoria della GPU e si utilizza soltanto la memoria di sistema della CPU. Questi sistemi adottano GPU con prestazioni relativamente basse, poiché le prestazioni sono comunque limitate dalla banda della memoria di sistema e dalla maggiore latenza di accesso alla memoria (la memoria associata alla GPU garantisce invece banda larga e bassa latenza).

Una variante ad alte prestazioni utilizza GPU multiple interconnesse, tipicamente da due a quattro operanti in parallelo, con i loro terminali grafici collegati in sequenza (in *daisy-chain*); ne è un esempio il sistema multi-GPU SLI (*Scalable Link Interconnect*) di NVIDIA, progettato per applicazioni di videogioco ad alte prestazioni e per le workstation.

La successiva categoria di sistemi ha una struttura con GPU integrata nel north bridge (Intel) o nel chipset (AMD), con o senza memoria grafica dedicata.

Nel Capitolo 5 è stato spiegato come le memorie cache mantengano la coerenza quando lo spazio degli indirizzi è condiviso; essendoci ora una CPU e una GPU, gli spazi di indirizzamento diventano multipli. Le GPU possono accedere alla propria memoria locale e alla memoria fisica di sistema della CPU utilizzando indirizzi virtuali che vengono tradotti da un'unità dedicata di gestione della memoria (MMU, *Memory Management Unit*) a bordo della GPU. Sarà il kernel del sistema operativo a gestire le tabelle delle pagine della GPU: si può accedere a una pagina fisica della memoria di sistema mediante transazioni sulla connessione PCI-Express coerenti o non coerenti, a seconda

PCI-Express (PCIe): un sistema di interconnessione standard di I/O che utilizza collegamenti da punto a punto. Le connessioni hanno un numero di linee e una larghezza di banda configurabili.

Architettura con memoria unificata (UMA): un'architettura di sistema in cui CPU e GPU condividono la stessa memoria di sistema. [*N.d.R.*: qui il termine UMA viene utilizzato in modo diverso rispetto al Capitolo 7, dove indicava la memoria ad accesso uniforme nei multiprocessori].

del valore di un attributo specificato nella tabella delle pagine della GPU. La CPU può accedere alla memoria locale della GPU attraverso un intervallo di indirizzi (chiamato anche «apertura») nello spazio di indirizzamento del bus PCI-Express.

Console di videogiochi

I sistemi basati su console come la Playstation 3 di Sony e l'Xbox 360 di Microsoft hanno un'architettura di sistema simile a quella di un PC sopra descritta. Le console sono progettate per essere prodotte con le stesse prestazioni e funzionalità per un periodo di tempo che può arrivare a cinque anni. Durante questo periodo un sistema può essere riprogettato più volte per sfruttare processi produttivi del silicio più avanzati, e di conseguenza fornire prestazioni costanti a costi sempre inferiori. Le console non hanno la necessità di dover essere espanse e aggiornate come i PC, per cui i principali bus interni del sistema tendono a essere progettati ad hoc piuttosto che standardizzati.

Interfacce e driver delle GPU

AGP: una versione estesa del bus PCI di I/O originale, in grado di fornire a una singola scheda fino a otto volte la banda del bus PCI. La sua funzione principale era di connettere i sottosistemi grafici alle architetture dei PC.

In un PC attuale le GPU sono connesse alla CPU attraverso il bus PCI-Express, mentre le generazioni precedenti utilizzavano il bus **AGP**. Le applicazioni grafiche chiamano funzioni delle API OpenGL (Segal e Akekey, 2006) o DirectX (Microsoft DirectX Specification), le quali utilizzano la GPU come un coprocessore. Le API inviano comandi, programmi e dati alla GPU mediante un driver del dispositivo grafico, ottimizzato per la particolare GPU utilizzata.

La pipeline grafica logica

La pipeline grafica logica è descritta nel Paragrafo C.3. La figura C.2.3 illustra i principali stadi di elaborazione e mette in evidenza gli stadi programmabili importanti: stadio di shading dei vertici, della geometria e dei pixel.

Mappatura della pipeline grafica su GPU a processori unificati

La Figura C.2.4 mostra come la pipeline logica comprendente degli stadi programmabili separati e indipendenti venga mappata su una schiera di processori distribuiti reali.

Architettura GPU unificata di base

Le architetture di GPU unificate sono basate su una schiera parallela di molti processori programmabili; questi unificano l'elaborazione effettuata dagli shader dei vertici, della geometria e dei pixel e il calcolo parallelo generico sugli stessi processori, a differenza delle GPU precedenti che avevano processori separati dedicati a ciascun tipo di elaborazione. L'insieme dei processori programmabili è strettamente integrato con alcuni processori che hanno una funzione prefissata, come il filtraggio della tessitura, la rasterizzazione, le operazioni sulla memoria video, l'anti-aliasing, la compressione, la decom-

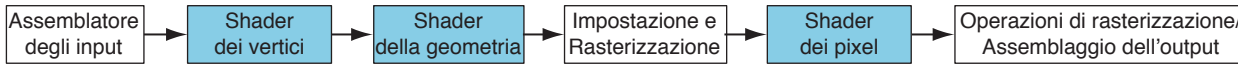


Figura C.2.3. La pipeline logica grafica. Gli stadi degli shader programmabili sono rappresentati in blu, i blocchi che implementano funzioni prefissate in bianco.

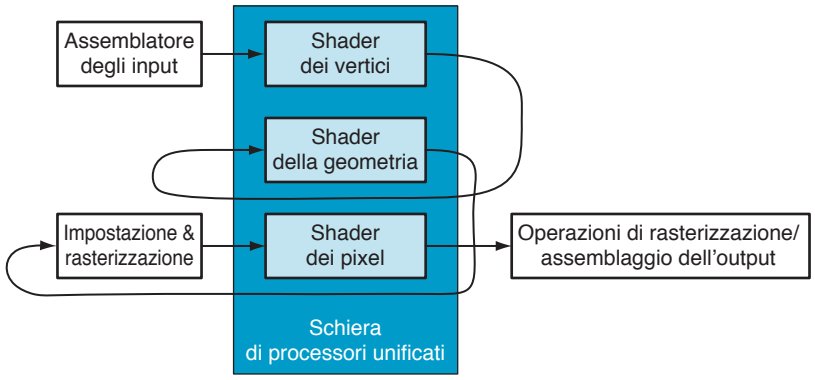


Figura C.2.4. Pipeline logica mappata su processori fisici. Gli stadi degli shader programmabili vengono eseguiti dalla schiera di processori unificati e il flusso dei dati previsto dalla pipeline logica della grafica passa attraverso i processori.

pressione, la visualizzazione, la decodifica video e l'elaborazione video ad alta definizione. Benché i processori che hanno una specifica funzione siano molto superiori a quelli programmabili di utilizzo più generale in termini di prestazioni assolute a parità di area, di costo e di potenza dissipata, concentreremo qui l'attenzione sui processori programmabili.

Rispetto alle CPU multicore, le GPU a più core presentano una differente filosofia di progetto dell'architettura, che è focalizzata sull'esecuzione di molti thread paralleli su più processori in modo efficiente. Utilizzando molti core più semplici e ottimizzando il parallelismo sui dati tra gruppi di thread, una frazione maggiore di transistor per ogni chip è dedicata all'elaborazione e non alla memoria cache interna o a funzioni accessorie.

Schiera di processori

Una schiera di processori unificati di una GPU contiene molti processori core, tipicamente organizzati in multiprocessori multithread. In Figura C.2.5 è mostrata una GPU contenente un'insieme di 112 processori core a flusso continuo (SP, *Streaming Processor*), organizzati in 14 multiprocessori multithread a flusso continuo (SM, *Streaming Multiprocessor*). Ogni core SP è altamente multithread, essendo in grado di gestire in hardware 96 thread concorrenti e il loro stato. I processori sono connessi a quattro partizioni di memoria DRAM a 64 bit attraverso una rete di interconnessione. Ogni SM contiene otto core SP, due unità per funzioni speciali (SFU), memorie cache per le istruzioni e le costanti, un'unità per le istruzioni multithread e una memoria condivisa. Questa è l'architettura Tesla di base implementata nella scheda GeForce 8800 di NVIDIA. Essa presenta un'architettura unificata, nella quale i programmi grafici tradizionali per lo shading dei vertici, della geometria e dei pixel vengono eseguiti sugli SM unificati e sui loro core SP, e i programmi di calcolo generico possono essere eseguiti anch'essi sugli stessi processori.

L'architettura a schiera di processori consente di creare configurazioni di GPU più o meno grandi, modificando il numero dei multiprocessori e il numero di partizioni di memoria. La Figura C.2.5 mostra sette gruppi di due SM che condividono la stessa unità di tessitura e la stessa memoria cache di primo livello per la tessitura stessa. L'unità di tessitura fornisce i risultati del filtraggio allo SM, dato l'insieme di coordinate della mappa. Poiché le regioni di supporto dei filtri sono spesso sovrapposte tra loro per ottenere un'unica tessitura continua, una piccola cache di primo livello del flusso di dati riduce efficacemente il numero di richieste al sistema di memoria. La schiera dei processori è connessa ai processori che effettuano operazioni di rasterizzazione (ROP), alle cache di secondo livello della tessitura, alle memorie DRAM esterne e alla memoria di sistema attraverso una rete di interconnessione estesa a tutta la GPU. Il numero dei processori e il numero delle memorie può variare,

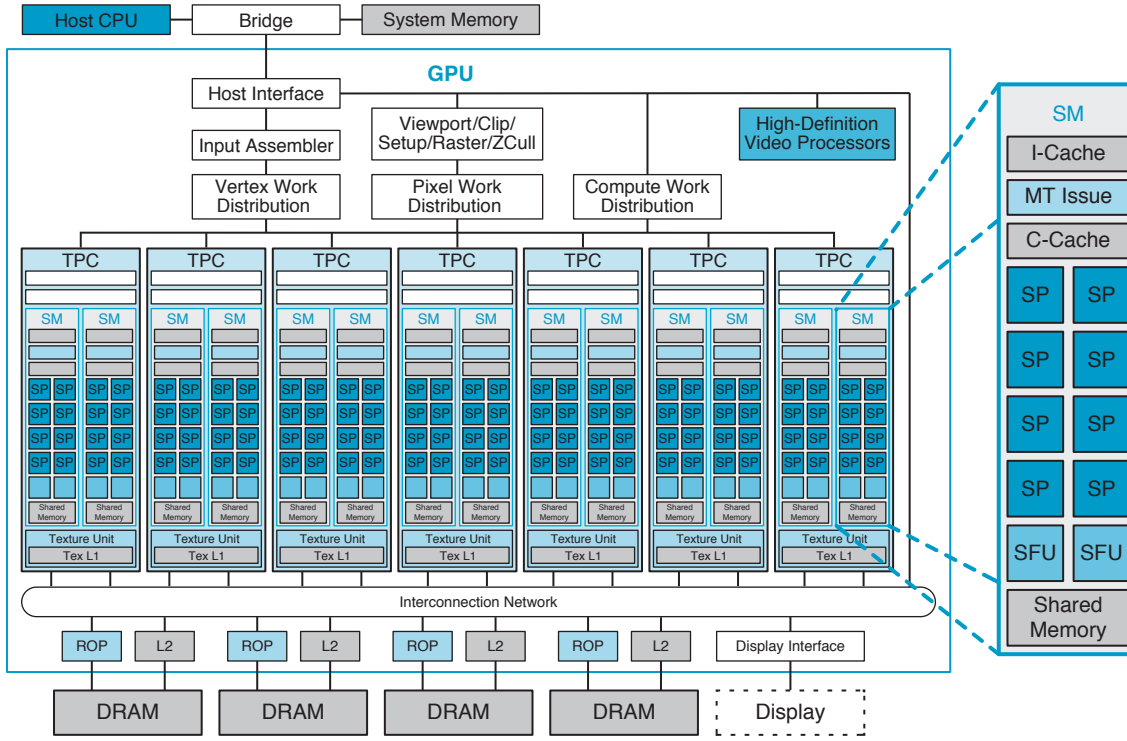


Figura C.2.5. Architettura di base di una GPU unificata. Esempio di GPU contenente 112 processori core a flusso continuo (SP) organizzati in 14 multiprocessori a flusso continuo (SM); i core sono fortemente multithread. Lo schema riproduce l'architettura Tesla di base di una GeForce 8800 NVIDIA. I processori sono connessi a quattro partizioni di memoria DRAM a 64 bit attraverso una rete di interconnessione. Ogni SM contiene otto core SP, due unità per funzioni speciali (SFU), memorie cache per le istruzioni e per le costanti, un'unità per istruzioni multithread e una memoria condivisa.

al fine di progettare sistemi basati su GPU che siano bilanciati per differenti prestazioni e destinati a differenti segmenti di mercato.

C.3 Programmazione delle GPU

La programmazione di una GPU multiprocessore differisce in maniera sostanziale dalla programmazione di altri multiprocessori (come le CPU multicore). Le GPU forniscono un numero di thread e un livello di parallelismo dei dati da due a tre ordini di grandezza superiori alle CPU, e sono arrivate nel 2008 a contenere centinaia di processori core e decine di migliaia di thread concorrenti. Le GPU continuano ad aumentare il loro grado di parallelismo raddoppiandolo ogni 12–18 mesi circa, in perfetto accordo con la legge di Moore. Per coprire la vasta gamma di prezzi e di prestazioni dei diversi segmenti di mercato, si producono GPU contenenti un numero di processori e thread assai variabile. Ciononostante, l'utente si aspetta che i videogiochi e le applicazioni grafiche, di visualizzazione e di calcolo funzionino con qualsiasi GPU, indipendentemente da quanti processori abbia o da quanti thread possa eseguire in parallelo; inoltre, si aspetta che le GPU più costose (con più core e più thread) eseguano le applicazioni più velocemente. Per questo motivo, i modelli di programmazione delle GPU e i programmi applicativi vengono progettati per coprire una vasta gamma di gradi di parallelismo.

Il motivo per cui è necessario disporre di molti core ed eseguire svariati thread in parallelo è rappresentato dalla grafica in tempo reale; per esempio, solo con un sistema di questo tipo è possibile effettuare il rendering di scene 3D complesse ad alta risoluzione a una frequenza compatibile con l'interattivi-

tà, ossia di almeno 60 immagini al secondo. In conseguenza di ciò, i modelli di programmazione dei linguaggi grafici di shading, come Cg (C per la grafica) e HLSL (linguaggio di shading ad alto livello), sono stati progettati per sfruttare un livello elevato di parallelismo mediante molti thread paralleli indipendenti e per funzionare su un numero qualsiasi di processori core. Analogamente, il modello di programmazione parallela e scalabile CUDA permette a generiche applicazioni di calcolo parallelo di sfruttare grandi quantità di thread paralleli e funzionare su un numero qualsiasi di processori core in modo trasparente all'applicazione.

In questi modelli di programmazione scalabili, il programmatore scrive il codice per un singolo thread, ma la GPU lancia un numero molto elevato di istanze del thread in parallelo. I programmi possono perciò funzionare in modo trasparente su un'ampia gamma di hardware paralleli. Questo semplice paradigma è derivato dalle API grafiche e dai linguaggi di *shading*, in cui si descrive come colorare un vertice o un pixel. È rimasto un paradigma efficace anche quando le GPU hanno iniziato a crescere rapidamente in termini di parallelismo e prestazioni, a partire dalla fine degli anni Novanta.

Questo paragrafo descrive brevemente la programmazione di una GPU per applicazioni di grafica in tempo reale mediante API grafiche e linguaggi di programmazione. Descrive quindi la programmazione di una GPU per l'elaborazione visuale e per applicazioni generali di calcolo parallelo utilizzando il linguaggio C e il modello di programmazione CUDA.

Programmazione della grafica in tempo reale

Le API hanno giocato un ruolo importante nello sviluppo e nel successo delle GPU e dei processori. Le API grafiche standard principali sono due: le **OpenGL** e le **Direct3D** (una delle interfacce di programmazione multimediali DirectX di Microsoft). OpenGL è uno standard aperto che fu originariamente proposto e definito dalla società Silicon Graphics. Lo sviluppo, tuttora in corso, e l'estensione dello standard OpenGL (Segal e Akekey, 2006; Kessenich, 2006) viene gestito da Khronos, un consorzio industriale. Direct3D (Blythe, 2006) è uno standard di fatto ed è stato definito e continuamente aggiornato da Microsoft e dai suoi partner. OpenGL e Direct3D sono API strutturate in maniera simile e continuano a evolvere rapidamente seguendo i progressi dell'hardware delle GPU. Esse definiscono una pipeline logica di elaborazione grafica che viene mappata sui processori e sull'hardware delle GPU, nonché i modelli di programmazione e i linguaggi per gli stadi programmabili della pipeline grafica.

OpenGL: una API grafica a standard aperto.

Direct3D: una API grafica definita da Microsoft e dai suoi partner.

Pipeline grafica logica

La figura C.3.1 illustra la pipeline grafica logica delle Direct3D 10 (la struttura di pipeline delle OpenGL è simile). L'API e la pipeline logica forniscono un'infrastruttura per il flusso continuo dei dati e l'impianto per gli stadi degli shader programmabili, evidenziati in blu. L'applicazione 3D invia alla GPU una sequenza di vertici raggruppati in primitive geometriche: punti, linee, triangoli e poligoni. L'assemblatore degli input raccoglie i vertici e le primitive e il programma di shading dei vertici esegue l'elaborazione vertice per vertice, compresa la trasformazione della posizione 3D del vertice nella sua posizione sullo schermo e la determinazione del suo colore. Il programma di shading della geometria esegue elaborazioni sulle primitive geometriche e può aggiungere o eliminare primitive. L'unità di impostazione e di rasterizzazione genera dei *frammenti di pixel* (i frammenti sono contributi potenziali ai pixel) che vengono ricoperti da una primitiva geometrica. Il programma di

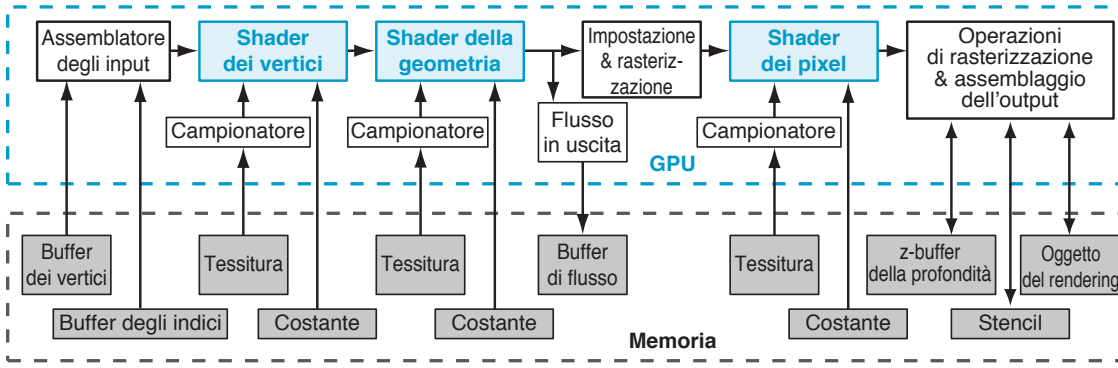


Figura C.3.1. Pipeline grafica delle Direct3D 10. Ogni stadio della pipeline logica è mappato sull'hardware della GPU o su un processore GPU. Gli stadi di shading sono colorati in blu, i blocchi contenenti una funzione prefissata in bianco e i dispositivi di memoria in grigio. Ogni stadio elabora un vertice, una primitiva geometrica o un pixel, in modalità a flusso continuo.

shading dei pixel effettua elaborazioni sui singoli frammenti, tra cui l'interpolazione dei parametri di ogni frammento e l'applicazione della tessitura e del colore. Gli shader dei pixel fanno largo uso delle funzioni di mappatura che campionano e filtrano matrici 1D, 2D o 3D, chiamate **tessiture**, utilizzando coordinate interpolate espresse in virgola mobile. Gli shader ricorrono ad accessi alla tessitura per mappe, funzioni, decalcomanie, immagini e dati. Lo stadio di elaborazione delle operazioni di rasterizzazione (o assemblaggio dell'output) effettua il controllo della profondità mediante Z-buffer e il controllo degli stencil; ossia, può scartare un frammento di pixel non visibile perché nascosto da altri oggetti o sostituire la profondità del pixel con quella del frammento. Inoltre, può effettuare un'operazione di mescolamento dei colori mescolando il colore del pixel con quello del frammento e assegnare il colore risultante al pixel.

Le API e la pipeline grafica forniscono gli ingressi, le uscite, le aree di memoria e l'infrastruttura per i programmi di shading che elaborano ogni vertice, primitiva o frammento di pixel.

I programmi degli shader grafici

Tessitura (*texture*): una matrice 1D (vettore), 2D o 3D che può essere messa in corrispondenza con coordinate interpolate mediante campionamento e filtraggio.

Shader: un programma che opera su dati grafici, per esempio vertici o frammenti di pixel.

Linguaggio di shading: un linguaggio di rendering grafico, solitamente caratterizzato da un modello di programmazione a flusso di dati o a flusso continuo (*streaming*).

Le applicazioni di grafica in tempo reale utilizzano molti **shader** per calcolare come la luce interagisce con i diversi materiali e per effettuare il rendering di illuminazioni e ombreggiature complesse. I **linguaggi di shading** sono basati su un modello di programmazione a flusso di dati o a flusso continuo (*streaming*) che trova una corrispondenza nella pipeline grafica logica.

Gli shader dei vertici mappano la posizione dei vertici dei triangoli sullo schermo, modificando la loro posizione, il loro colore e il loro orientamento. Tipicamente, un thread di uno shader dei vertici riceve in ingresso la posizione di un vertice espressa in virgola mobile (x, y, z, w) e calcola la sua posizione sullo schermo (x, y, z). Gli shader della geometria operano sulle primitive geometriche (come rette e triangoli) definite da un insieme di vertici, modificando queste primitive o generando primitive aggiuntive. Gli shader dei frammenti di pixel «dipingono» ciascun pixel, calcolando il contributo per ogni pixel (x, y) dell'immagine da sintetizzare nei canali di colore rosso, verde, blu e alpha (RGBA). Gli shader (e le GPU) utilizzano un'aritmetica in virgola mobile per tutte le operazioni sul colore dei pixel, in modo da eliminare possibili artefatti visibili. I pixel possono assumere una gamma molto estesa di valori, specialmente nella sintesi di scene caratterizzate da illuminazioni e ombre complesse o da una dinamica elevata. Per tutti e tre i tipi di shader grafico molteplici istanze di uno stesso programma possono essere lanciate in parallelo, come

thread paralleli e indipendenti, in quanto ognuno di essi opera su dati indipendenti e produce risultati indipendenti, senza «effetti collaterali». Vertici, primitive geometriche e pixel indipendenti, inoltre, permettono allo stesso programma grafico di essere eseguito su GPU di dimensioni differenti, che sono in grado di elaborare in parallelo un diverso numero di vertici, primitive e pixel. I programmi grafici, quindi, scalano in modo trasparente su GPU con prestazioni e gradi di parallelismo differenti.

Per programmare questi tre tipi di thread grafici si usa un linguaggio ad alto livello appositamente pensato per la grafica. Generalmente si impiegano **HLSL** (*High Level Shading Language*) o **Cg** (C per la grafica). Questi linguaggi sono caratterizzati da una sintassi simile a quella del C e dispongono di molte funzioni di libreria per effettuare le operazioni su matrici e funzioni trigonometriche, l'interpolazione, l'accesso e il filtraggio di tessiture. Si tratta di linguaggi di programmazione abbastanza diversi dai linguaggi generici: non sono dotati di accesso globale alla memoria, puntatori, input/output su file e ricorsione. In HLSL e Cg si assume che i programmi «vivano» all'interno di una pipeline grafica logica, per cui le operazioni di input/output sono implicite. Per esempio, uno shader dei frammenti di pixel può aspettarsi che le normali geometriche di una figura e le coordinate multiple di una tessitura vengano interpolate a partire dai valori assunti nei vertici da parte degli stadi precedenti della pipeline con funzione prefissata, e può assegnare semplicemente un valore al parametro di uscita COLOR, il quale verrà passato agli stadi successivi per essere miscelato con il valore assunto da un pixel che si trova in quella posizione (x, y).

L'hardware di una GPU crea un nuovo thread indipendente per eseguire un'operazione di shading sui vertici, sulla geometria o sui pixel, per ogni vertice, primitiva e frammento di pixel. Nei videogiochi, la maggior parte dei thread esegue shader di pixel, poiché tipicamente i pixel sono da 10 a 20 volte più numerosi dei vertici, e le sfumature di luce e le ombreggiature complesse richiedono un numero ancora maggiore di thread dei pixel rispetto a quelli dei vertici. Il modello di programmazione grafica a shader ha portato le architetture di GPU a eseguire in modo efficiente migliaia di thread indipendenti, a grana fine, su molti processori core in parallelo.

Un esempio di shader dei pixel

Consideriamo il seguente programma Cg di shading dei pixel che implementa la tecnica di rendering della «mappatura d'ambiente». Per ogni thread di pixel, questo shader riceve cinque parametri in ingresso, tra cui le coordinate 2D in virgola mobile dell'immagine della tessitura, necessarie a campionare il colore sulla superficie, e un vettore 3D in virgola mobile che fornisce la direzione della riflessione sulla superficie della direzione di osservazione. Gli altri tre parametri «uniformi» non variano dall'istanza di un pixel (thread) a quella successiva. Lo shader ricava il colore da due immagini di tessitura: un accesso a una tessitura 2D per ricavare il colore della superficie e un accesso a una tessitura 3D contenuta in un cubo (sei immagini corrispondenti alle facce di un cubo) per ottenere il colore del mondo esterno corrispondente alla direzione di riflessione. Quindi, le quattro componenti in virgola mobile (rosso, verde, blu, alfa) del colore finale vengono calcolate applicando una media pesata, chiamata *lerp* o funzione di interpolazione lineare dei valori ricavati.

```
void riflessione(
   float2              CoordTessitura     : TEXCOORD0,
   float3              dir_riflessione    : TEXCOORD1,
   out float4          colore             : COLOR,
```

```
    uniform float          lucentezza,
    uniform sampler2D      MappaSuperficie,
    uniform samplerCUBE    MappaAmbiente)
{
// Ricava il colore della superficie da una tessitura
    float4 ColoreSuperficie = tex2D(MappaSuperficie,
    CoordTessitura);

// Ricava il colore riflesso campionando una mappa cubica
    float4 ColoreRiflesso = texCUBE(MappaAmbiente,
    dir_riflessione);

// L'output è la media pesata dei due colori
    colore = lerp(ColoreSuperficie, ColoreRiflesso, lucentezza);
}
```

Nonostante questo programma di shading sia lungo solo tre linee di codice, attiva una gran quantità di hardware della GPU. Per ogni prelievo di tessitura, il sottosistema di accesso alla tessitura deve effettuare accessi multipli alla memoria per campionare i colori nell'intorno delle coordinate di campionamento, e poi calcolare per interpolazione il risultato finale utilizzando l'aritmetica in virgola mobile. Le GPU multithread eseguono migliaia di questi leggeri shader di pixel Cg in parallelo, intrecciandoli pesantemente per minimizzare la latenza dovuta alla lettura della tessitura e alla memoria.

Cg focalizza l'attenzione del programmatore su un singolo vertice, primitiva o pixel, che viene implementato nella GPU come un singolo thread; il programma di shading, in modo trasparente, scala l'elaborazione per sfruttare il parallelismo dei thread sui processori disponibili. Essendo un linguaggio dedicato alle applicazioni, Cg mette a disposizione un ricco insieme di tipi di dati, funzioni di libreria e costrutti di linguaggio particolarmente utili per implementare le diverse tecniche di rendering.

La figura C.3.2 mostra un esempio di pelle sintetizzata da uno shader. La pelle reale appare molto differente perché la luce viene rifratta all'interno della pelle diverse volte prima di essere riflessa. In questo shader complesso sono stati simulati tre diversi strati di pelle, ciascuno con il suo effetto di diffusione della luce e i suoi parametri di scattering, per dare alla pelle una profondità e un aspetto traslucido. Lo scattering può essere simulato mediante la convoluzione di un filtro di sfocatura (*blurring*) in uno «spazio di tessitura» appiattito, in cui il rosso viene sfuocato più del verde e il blu viene sfuocato meno degli altri colori. Questo shader Cg compilato esegue 1400 istruzioni per calcolare il colore di un solo pixel.

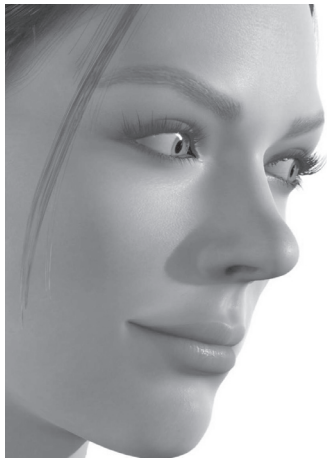


Figura C.3.2. Immagine sintetizzata con una GPU. Per conferire alla pelle profondità e un aspetto traslucido, lo shader dei pixel modellizza tre diversi strati di pelle, ognuno con il proprio comportamento di diffusione della luce e i propri parametri di scattering. Lo shader esegue 1400 istruzioni per generare le componenti cromatiche rossa, verde, blu e alfa di ogni frammento di pixel della pelle.

Quando le GPU hanno raggiunto prestazioni elevate nel calcolo in virgola mobile e una larghezza di banda assai ampia nel trasferimento a flusso continuo con la memoria, sono diventate interessanti per l'esecuzione di applicazioni fortemente parallele diverse da quelle grafiche. All'inizio, la possibilità di sfruttare una tale potenza di calcolo era vincolata dal dover esprimere le applicazioni come algoritmi di rendering grafico, e questo approccio risultava spesso scomodo e limitante. Più recentemente, il modello di programmazione CUDA ha reso disponibile una modalità molto più semplice per sfruttare la scalabilità del calcolo in virgola mobile ad alte prestazioni e la larghezza di banda della memoria delle GPU, basata sul linguaggio C.

Programmazione di applicazioni di calcolo parallelo

CUDA, Brook e CAL sono interfacce di programmazione per GPU focalizzate sull'elaborazione parallela dei dati più che sulla grafica. CAL (*Compute Abstraction Layer*) è un'interfaccia in linguaggio assembler a basso livello per

le GPU AMD. Brook è un linguaggio per flusso continuo di elaborazione adattato alle GPU da Buck et al. (2004). CUDA è stato sviluppato da NVIDIA (2007) ed è un'estensione dei linguaggi C e C++ per la programmazione parallela e scalabile di GPU e CPU multicore. In seguito descriveremo il modello di programmazione CUDA come è stato presentato in un articolo da Nickolls, Buck, Garland e Skadron (2008).

Con questo nuovo modello, la GPU eccelle nelle elaborazioni parallele a livello di dati e nel throughput, riuscendo a eseguire applicazioni di calcolo ad alte prestazioni con le stesse prestazioni disponibili per le applicazioni di grafica.

Decomposizione di un problema con parallelismo a livello di dati

Per mappare in modo efficace problemi di calcolo di grandi dimensioni su un'architettura altamente parallela, il programmatore o il compilatore devono scomporre il problema in tanti problemi più piccoli che possono essere risolti in parallelo. Per esempio, il programmatore può partizionare un grande vettore di dati contenente il risultato in blocchi e poi partizionare ulteriormente ogni blocco in elementi più piccoli, così che i blocchi del risultato possano essere determinati indipendentemente e in parallelo e gli elementi di ogni blocco possano essere elaborati in parallelo. La figura C.3.3 mostra la scomposizione di un vettore contenente il risultato in una griglia 3 × 2 di blocchi, dove ogni blocco è ulteriormente scomposto in una griglia 5 × 3 di elementi. La scomposizione in griglie su due livelli si mappa in modo naturale sull'architettura delle GPU: i multiprocessori calcolano i blocchi-risultato in parallelo, mentre thread paralleli elaborano i singoli elementi del risultato.

Il programmatore scrive un programma che calcola una sequenza di matrici contenenti il risultato, partizionando ogni griglia in blocchi-risultato a grana grossa che possono essere calcolati indipendentemente in parallelo. Il programma calcola ogni blocco di risultati con una schiera di thread paralleli a grana fine, suddividendo il lavoro fra i thread in modo che ciascuno calcoli uno o più elementi di risultato.

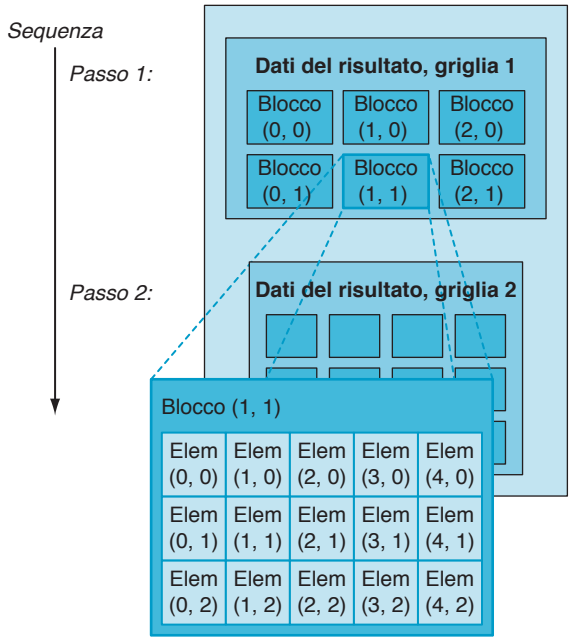


Figura C.3.3. **Scomposizione dei dati in una griglia di blocchi di elementi che vengono calcolati in parallelo.**

Programmazione scalabile parallela con CUDA

Il modello di programmazione scalabile parallela CUDA estende i linguaggi C e C++ per sfruttare un elevato grado di parallelismo per applicazioni generali su multiprocessori paralleli a elevato grado di parallelismo, in particolare sulle GPU. I primi esperimenti con CUDA hanno mostrato che molti programmi complessi possono essere espressi semplicemente con poche astrazioni di facile comprensione. Da quando NVIDIA ha rilasciato CUDA nel 2007, gli sviluppatori hanno prodotto rapidamente programmi paralleli scalabili per un'ampia gamma di applicazioni, tra cui l'elaborazione di dati sismici, la chimica computazionale, l'algebra lineare, la soluzione di sistemi con matrici sparse, l'ordinamento, i modelli di fisica e l'elaborazione visuale. Queste applicazioni funzionano bene su centinaia di processori core e migliaia di thread concorrenti. Le GPU NVIDIA con architettura unificata per grafica ed elaborazione Tesla (descritte nei Paragrafi C.4 e C.7) eseguono programmi CUDA e sono ampiamente disponibili sui portatili, sui PC, sulle workstation e sui server. Il modello CUDA è applicabile anche ad altre architetture di elaborazione parallela a memoria condivisa, come le CPU multicore (Stratton, 2008).

CUDA fornisce tre astrazioni fondamentali: una **gerarchia di gruppi di thread**, **memorie condivise** e **sincronizzazione a barriera**, che permettono una strutturazione chiara del parallelismo all'interno di codice C convenzionale per ciascun thread della gerarchia. Livelli multipli di thread, di memoria e di sincronizzazione permettono un parallelismo sui dati e un parallelismo di thread a grana fine, innestati su un parallelismo sui dati e un parallelismo di procedura a grana grossa. Le astrazioni guidano il programmatore a suddividere il problema dapprima in spezzoni di problema, che possono essere risolti indipendentemente in parallelo, e quindi in spezzoni ancora più piccoli, che possono anch'essi essere risolti in parallelo. Il modello di programmazione scala in modo trasparente su un numero molto elevato di processori: un programma CUDA compilato può essere eseguito su un qualsiasi numero di processori; soltanto il sistema runtime ha bisogno di conoscere il numero reale dei processori.

Il paradigma CUDA

Kernel: un programma o funzione per un thread, progettato per essere eseguito da molti thread.

Blocco di thread: un insieme di thread concorrenti che eseguono lo stesso programma di thread e possono collaborare per il calcolo del risultato.

Griglia: un insieme di blocchi di thread che eseguono lo stesso programma kernel.

CUDA è un'estensione minimale dei linguaggi di programmazione C e C++. Il programmatore scrive un programma sequenziale che richiama **kernel** paralleli, i quali possono contenere semplici funzioni o programmi completi. Un kernel viene eseguito in parallelo su un insieme di thread paralleli. Il programmatore organizza questi thread in una gerarchia di blocchi di thread e di griglie di blocchi di thread. Un **blocco di thread** è un insieme di thread concorrenti che possono collaborare tra loro mediante sincronizzazione a barriera e accesso condiviso allo spazio di memoria privato del blocco. Una **griglia** (*grid*) è un insieme di blocchi di thread che possono essere eseguiti ciascuno in modo indipendente, quindi in parallelo.

Quando deve essere lanciato in esecuzione un kernel, il programmatore specifica il numero di thread per blocco e il numero di blocchi costituenti la griglia. A ciascun thread viene assegnato un numero identificativo (univoco) di thread, `threadIdx`, all'interno del blocco di thread che lo contiene, e a ciascun blocco di thread un numero di blocco univoco, `blockIdx`, all'interno della sua griglia. CUDA supporta blocchi di thread contenenti fino a 512 thread. Per comodità, i blocchi di thread e le griglie possono essere organizzati in matrici a 1, 2 o 3 dimensioni a cui si accede mediante i campi indice `.x`, `.y` e `.z`.

Come esempio molto semplice di programmazione parallela consideriamo due vettori, x e y, ciascuno contenente n elementi in virgola mobile e supponiamo di voler calcolare il risultato della funzione $y = ax + y$ per un dato valore dello scalare a. Questa funzione corrisponde al kernel chiamato `SAXPY` contenuto nella libreria di algebra lineare BLAS. La figura C.3.4 riporta il codice C che effettua tale calcolo sia su un processore seriale sia in parallelo utilizzando CUDA.

Lo specificatore `__global__` indica che la procedura è il punto di ingresso di una procedura kernel. I programmi CUDA lanciano kernel paralleli con una chiamata a funzione estesa seguendo la seguente sintassi:

```
kernel<<<dimGriglia, dimBlocco>>>(... elenco parametri ...);
```

dove `dimGriglia` e `dimBlocco` sono vettori tridimensionali di tipo `dim3` che specificano, rispettivamente, le dimensioni della griglia in numero di blocchi e la dimensione del blocco in numero di thread. Se le dimensioni non vengono specificate, il valore di default è 1.

Nel codice di figura C.3.4 viene lanciata una griglia di n thread che assegna un thread a ogni elemento dei vettori e inserisce 256 thread in ogni blocco. Ogni singolo thread calcola l'indice di un elemento a partire dal proprio identificativo, ID, del blocco e del thread e quindi effettua il calcolo desiderato sull'elemento del vettore corrispondente. Confrontando la versione sequenziale e parallela di questo codice si nota che queste sono sorprendentemente simili. Il codice sequenziale consiste di un ciclo in cui ogni iterazione è indipendente da tutte le altre. Cicli come questi possono essere trasformati in modo meccanico in kernel paralleli: ogni iterazione del ciclo diventa un thread indipendente. Assegnando un singolo thread a ogni elemento di output, non abbiamo bisogno di sincronizzazione i thread per scrivere i risultati in memoria.

Il testo di un kernel CUDA è semplicemente una funzione C scritta per un thread sequenziale. È quindi in genere molto semplice da scrivere, soprattutto rispetto al codice parallelo per operazioni vettoriali. Il parallelismo è deter-

Calcolo di y = ax + y mediante un ciclo sequenziale:

```
void saxpy_sequenziale(int n, float alpha, float *x, float *y)
{
   for(int i = 0; i<n; ++i)
       y[i] = alpha*x[i] + y[i];
}

// Chiama il kernel SAXPY sequenziale
saxpy_sequenziale(n, 2.0, x, y);
```

Calcolo di y = ax + y in parallelo utilizzando CUDA:

```
__global__
void saxpy_parallelo(int n, float alpha, float *x, float *y)
{
   int i = blockIdx.x*blockDim.x + threadId.x;
   if (i<n)    y[i] = alpha*x[i] + y[i];
}

// Chiama il kernel SAXPY parallelo (256 thread per blocco)
int numblocchi = (n+255) / 256;
saxpy_parallelo<<<numblocchi, 256>>>(n, 2.0, x, y);
```

Figura C.3.4. Confronto fra codice sequenziale (sopra) in C e codice parallelo (sotto) in CUDA per la funzione SAXPY (si veda il Capitolo 7). I thread paralleli CUDA sostituiscono il ciclo sequenziale del C: ogni thread calcola lo stesso risultato di un'iterazione di ciclo. Il codice parallelo calcola n risultati con n thread, organizzati in blocchi di 256 thread.

minato in modo chiaro ed esplicito specificando le dimensioni della griglia di elaborazione e il numero dei suoi blocchi di thread quando il programma kernel viene lanciato in esecuzione.

L'esecuzione parallela e la gestione dei thread sono automatiche. Creazione, pianificazione dell'esecuzione e terminazione dei thread è gestita dal sistema sottostante. Le GPU con architettura Tesla di fatto effettuano tutta la gestione dei thread direttamente in hardware. I thread di un blocco vengono eseguiti in modo concorrente e possono venire sincronizzati in corrispondenza di una **barriera di sincronizzazione**, chiamando la primitiva `__syncthreads()`. Questa garantisce che nessun thread del blocco possa proseguire l'esecuzione fino a quando tutti i thread dello stesso blocco non avranno raggiunto la barriera. Dopo averla superata, è garantito che questi thread vedranno i dati che i thread hanno scritto in memoria prima di aver raggiunto la barriera. I thread all'interno di uno stesso blocco, quindi, possono comunicare tra loro scrivendo e leggendo nella memoria condivisa del blocco in corrispondenza della barriera di sincronizzazione.

Barriera di sincronizzazione: i thread attendono in corrispondenza di una barriera di sincronizzazione fino a quando tutti i thread appartenenti al blocco non hanno raggiunto la barriera.

Operazione atomica di memoria: una sequenza di operazioni di lettura, modifica o scrittura della memoria che viene completata senza essere interrotta da altri accessi alla memoria.

Poiché i thread di un blocco possono condividere la memoria e sincronizzarsi mediante le barriere, essi risiederanno fisicamente sullo stesso processore o multiprocessore. Tuttavia, il numero di blocchi di thread può essere significativamente maggiore del numero dei processori. Il modello CUDA di programmazione dei thread virtualizza i processori e fornisce al programmatore totale flessibilità sul grado di granularità con cui rendere parallelo il codice, consentendo quindi di scegliere il livello di granularità che il programmatore ritiene più conveniente. La virtualizzazione in thread e blocchi di thread permette di scomporre il problema in maniera intuitiva, poiché il numero dei blocchi può essere determinato più dalla dimensione dei dati da elaborare che dal numero di processori del sistema. La virtualizzazione permette inoltre allo stesso programma CUDA di essere eseguito su un numero ampiamente variabile di processori core.

Per gestire questa virtualizzazione degli elementi di elaborazione e garantire la scalabilità su più processori, CUDA richiede che i blocchi di thread possano essere eseguiti in modo indipendente: si deve poter eseguire i blocchi in un qualsiasi ordine, in parallelo o in sequenza. Blocchi differenti non hanno possibilità di comunicazione diretta, benché essi possano coordinare le loro attività utilizzando **operazioni atomiche di memoria** sulla memoria globale visibile da tutti i thread. Questo requisito di indipendenza consente di avviare a esecuzione blocchi di thread in un qualsiasi ordine su un qualsiasi numero di processori, rendendo il modello CUDA scalabile su un numero arbitrario di core, come pure su una grande varietà di architetture parallele. Ciò aiuta anche a evitare la possibilità che si verifichi un blocco critico (*deadlock*). Un'applicazione può eseguire griglie multiple di funzioni sia in modo indipendente sia dipendente. Griglie indipendenti possono essere eseguite in maniera concorrente in caso di risorse hardware sufficienti. Le griglie dipendenti vengono eseguite in sequenza, con un'implicita barriera inter-kernel tra loro, garantendo così che l'esecuzione di tutti i blocchi della prima griglia termini prima che inizi l'esecuzione di uno dei blocchi della seconda griglia, dipendente dalla prima.

Durante la loro esecuzione, i thread possono accedere a dati appartenenti a diversi spazi di memoria. Ogni thread dispone di una **memoria locale** (*local memory*) privata. CUDA utilizza la memoria locale per le variabili private dei thread che non riescono a essere allocate nei registri di thread, ma anche le aree di stack delle procedure (*stack frame*) e per il riversamento dai registri (*register spilling*). Ogni blocco di thread dispone di una **memoria condivisa** (*shared memory*), visibile da tutti i thread del blocco, che ha lo stesso tempo di vita del blocco stesso. Infine, tutti i thread hanno accesso alla **memoria globale**

Memoria locale: memoria locale e privata di ciascun thread.

Memoria condivisa: memoria condivisa da tutti i thread di un blocco.

Memoria globale: memoria dedicata alle applicazioni che viene condivisa da tutti i thread.

(*global memory*). I programmi dichiarano le varabili in memoria condivisa o globale con gli specificatori `__shared__` e `__device__`, rispettivamente. Su una GPU con architettura Tesla, questi spazi di memoria corrispondono a memorie fisicamente separate: la memoria condivisa, dedicata a ogni blocco, è una RAM *on chip* a bassa latenza, mentre la memoria globale risiede nella DRAM veloce della scheda grafica.

La memoria condivisa è tipicamente una memoria a bassa latenza vicina a ogni processore (come una cache L1). È quindi in grado di offrire alte prestazioni nella comunicazione e nella condivisione dei dati tra i thread del blocco. Poiché la memoria condivisa ha lo stesso tempo di vita del corrispondente blocco di thread, il codice del kernel tipicamente inizializzerà i dati nelle variabili condivise, effettuerà l'elaborazione richiesta utilizzando queste variabili e infine copierà i risultati dalla memoria condivisa alla memoria globale. Blocchi di thread appartenenti a griglie sequenzialmente dipendenti comunicano mediante la memoria globale, utilizzandola per leggere i dati in ingresso e scrivere i risultati.

La Figura C.3.5 mostra il diagramma dei livelli innestati di thread, blocchi di thread e griglie di blocchi di thread; essa mostra anche i livelli corrispondenti di memoria: locale, condivisa e globale per la condivisione dei dati a livello di thread, di blocco di thread e di applicazione.

Un programma gestisce lo spazio di memoria globale visibile dai kernel mediante chiamate runtime a CUDA, quali `cudaMalloc()` e `cudaFree()`. I kernel possono essere eseguiti su un dispositivo fisicamente diverso, come nel caso dei kernel eseguiti su GPU. Di conseguenza, l'applicazione deve utilizzare `cudaMemcpy()` per copiare i dati tra lo spazio allocato e la memoria di sistema della macchina host.

Il modello di programmazione CUDA ha uno stile simile al diffuso modello **programma singolo per dati multipli** (**SPMD**, *Single-Program Multiple Data*): il parallelismo è espresso in modo esplicito e ogni kernel viene eseguito con un numero prefissato di thread. Tuttavia, CUDA è più flessibile della

Programma singolo per dati multipli (SPMD): uno stile di programmazione parallela nel quale tutti i thread eseguono lo stesso programma. I thread SPMD sono tipicamente sincronizzati mediante sincronizzazione a barriera.

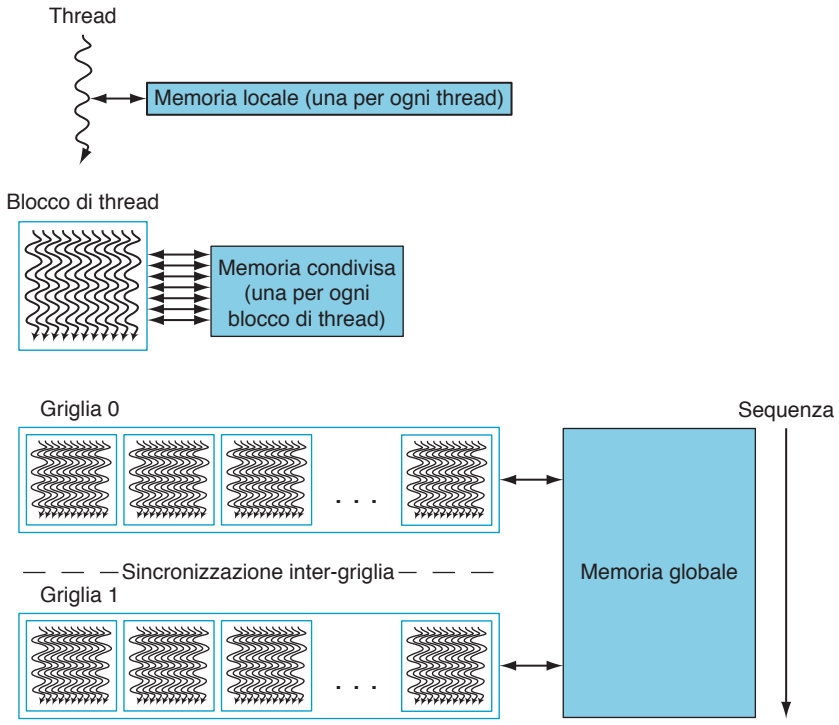


Figura C.3.5. I livelli di granularità innestati: thread, blocco di thread e griglia dispongono di corrispondenti livelli di condivisione della memoria (locale, condivisa e globale). La memoria locale, una per ogni thread, appartiene al thread. La memoria condivisa, una per ogni blocco, è condivisa da tutti i thread di un blocco. La memoria globale, una per ogni applicazione, è condivisa da tutti i thread.

maggior parte delle implementazioni SPMD perché ogni chiamata a kernel crea dinamicamente una nuova griglia con il numero di blocchi e di thread richiesto da quella parte dell'applicazione. Il programmatore può scegliere il grado di parallelismo più adatto per ogni kernel, anziché dover progettare tutte le fasi di elaborazione in modo da utilizzare lo stesso numero di thread. La figura C.3.6 mostra un esempio di frammento di codice CUDA di tipo SPMD. Il codice per prima cosa istanzia il kernel `kernelF` su una griglia 2D di 2 × 3 blocchi di thread, dove ogni blocco 2D consiste di 5 × 3 thread. Successivamente viene istanziato `kernelG` su una griglia monodimensionale di quattro blocchi di thread, contenenti sei thread ciascuno. Dato che `kernelG` dipende dai risultati di `kernelF`, i due kernel vengono separati da una barriera di sincronizzazione inter-kernel.

I thread concorrenti di un blocco di thread esprimono un parallelismo dei dati e dei thread a grana fine. I blocchi di thread indipendenti delle griglie esprimono un parallelismo di dati a grana grossa. Griglie indipendenti esprimono un parallelismo di procedura a grana grossa. Il kernel è costituito semplicemente dal codice C per un thread della gerarchia.

Restrizioni

Per questioni di efficienza e di semplicità di implementazione, il modello di programmazione CUDA presenta alcune restrizioni. I thread e i blocchi di thread possono essere creati soltanto invocando un kernel parallelo, e non

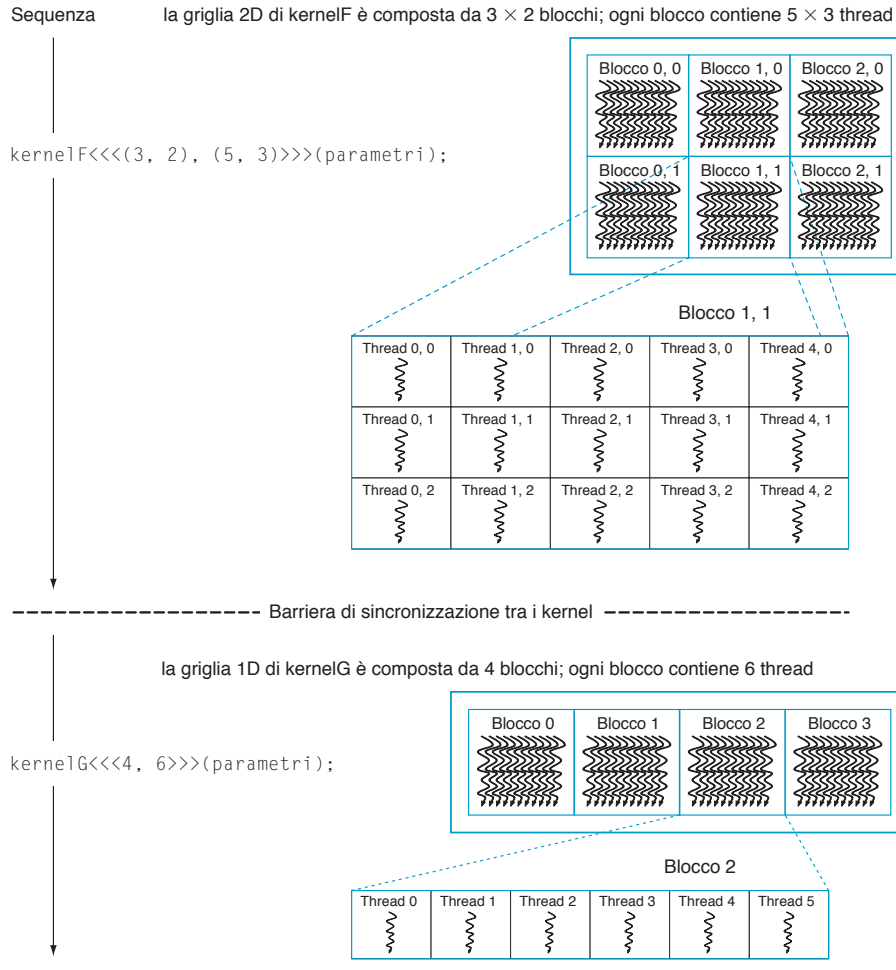


Figura C.3.6. **Sequenza di kernelF istanziato su una griglia 2D di blocchi 2D di thread, una barriera di sincronizzazione interkernel, seguita da kernelG istanziato su una griglia 1D di blocchi 1D di thread.**

dall'interno dei kernel paralleli; assieme al requisito di indipendenza dei blocchi di thread, ciò rende possibile l'esecuzione di programmi CUDA con uno *scheduler* («pianificatore») semplice, che introduce un sovraccarico minimo in fase di esecuzione. Di fatto, nell'architettura GPU Tesla la gestione e la pianificazione dell'esecuzione dei thread e dei blocchi di thread è implementata in hardware.

Il parallelismo di procedura può essere espresso a livello dei blocchi di thread, ma è difficile esprimerlo all'interno di un blocco, perché le barriere di sincronizzazione agiscono su tutti i thread del blocco. Per consentire ai programmi CUDA di essere eseguiti su un numero qualsiasi di processori, non sono permesse dipendenze tra diversi blocchi di thread della stessa griglia di kernel, cioè i blocchi devono necessariamente essere eseguiti in modo indipendente tra loro. Dato che CUDA richiede che i blocchi di thread siano indipendenti e possano quindi essere eseguiti in un qualsiasi ordine, la combinazione dei risultati prodotti dai diversi blocchi viene in generale ottenuta lanciando un secondo kernel su una nuova griglia di blocchi di thread (anche se i blocchi di thread possono coordinare la loro attività, per esempio, incrementando mediante operazioni atomiche puntatori a code).

Le chiamate a funzioni ricorsive non sono al momento permesse nei kernel CUDA. La ricorsione, infatti, non è adatta ai kernel pesantemente paralleli, poiché lo spazio di stack da mettere a disposizione per le decine di migliaia di thread che potrebbero essere attivi richiederebbe quantità notevoli di memoria. Gli algoritmi sequenziali normalmente espressi in forma ricorsiva, come Quick Sort, solitamente sono implementati meglio attraverso un parallelismo sui dati innestati piuttosto che con la ricorsione esplicita.

Per gestire l'architettura di sistema eterogenea costituita da una CPU e una GPU, ognuna con il proprio sistema di memoria, i programmi CUDA devono copiare i dati e i risultati tra le due memorie. Il sovraccarico dovuto all'interazione CPU–GPU e al trasferimento dati viene minimizzato utilizzando meccanismi di trasferimento DMA a blocchi e interconnessioni veloci. Problemi di calcolo intensivo sufficientemente grandi da avere bisogno di un incremento di prestazioni fornito da una GPU ammortizzano tale sovraccarico meglio dei problemi di piccole dimensioni.

Implicazioni per un'architettura

I modelli di programmazione parallela per grafica e calcolo hanno prodotto una sostanziale differenza tra l'architettura delle GPU e quella delle CPU. Gli aspetti principali dei programmi per GPU che influenzano l'architettura della GPU stessa sono:

- *Uso estensivo del parallelismo dei dati a grana fine:* i programmi di shading descrivono come elaborare un singolo pixel o vertice; i programmi CUDA descrivono come calcolare un singolo risultato.
- *Modello di programmazione fortemente organizzato in thread:* un programma di shading a thread elabora un singolo pixel o vertice; il thread di un programma CUDA è in grado di generare un singolo risultato. Una GPU è in grado di creare ed eseguire milioni di questi thread per ogni immagine, a 60 immagini al secondo.
- *Scalabilità:* un programma deve aumentare automaticamente le sue prestazioni quando vengono messi a sua disposizione altri processori, senza che si debba ricompilare il codice.
- *Elaborazione intensiva in virgola mobile (o intera).*
- *Supporto all'elaborazione di grande quantità di dati.*

C.4 Architettura multiprocessore multithread

Per coprire diversi segmenti di mercato, le GPU presentano un numero scalabile di multiprocessori; di fatto, le GPU sono multiprocessori composti da multiprocessori. Inoltre, ogni multiprocessore è fortemente multithread, così da poter eseguire in maniera efficiente molti thread di shader di pixel e di vertici a grana fine. Una GPU di base possiede da due a quattro multiprocessori, mentre le GPU utilizzate nelle console di videogiochi o nelle piattaforme di calcolo ne contengono dozzine. Qui analizziamo in dettaglio l'architettura di un multiprocessore multithread di questo tipo, una versione semplificata del multiprocessore a flusso continuo (SM) Tesla di NVIDIA descritto nel Paragrafo C.7.

Perché è meglio utilizzare un multiprocessore invece di molti processori indipendenti? Il parallelismo all'interno dei multiprocessori fornisce elevate prestazioni per il codice locale e supporta estensivamente l'esecuzione di thread multipli (per i modelli di programmazione parallela a grana fine descritti nel Paragrafo C.3). I singoli thread di un blocco vengono eseguiti insieme all'interno di un multiprocessore e possono condividere i dati. L'architettura multiprocessore multithread qui descritta possiede otto processori core scalari organizzati in un'architettura fortemente interconnessa e sono in grado di eseguire fino a 512 thread (il multiprocessore SM descritto nel Paragrafo C.7 esegue fino a 768 thread). Per garantire la maggiore efficienza possibile in termini di occupazione dello spazio e di potenza assorbita, il multiprocessore condivide tra gli otto processori core le unità funzionali grandi e complesse, tra cui la cache delle istruzioni, l'unità delle istruzioni multithread e la memoria RAM condivisa.

Il multithreading massivo

I processori GPU hanno un alto livello di multithreading per raggiungere i seguenti obiettivi:

- Nascondere la latenza del caricamento dei dati dalla memoria e della tessitura dalla DRAM.
- Supportare i modelli di programmazione degli shader grafici paralleli a grana fine.
- Supportare i modelli di programmazione per il calcolo parallelo a grana fine.
- Rendere virtuali i processori fisici come thread e blocchi di thread per fornire una scalabilità trasparente.
- Semplificare il modello di programmazione parallela riducendolo alla scrittura di un programma sequenziale per un singolo thread.

La latenza del caricamento dei dati e della tessitura dalla memoria può richiedere centinaia di cicli di clock del processore, poiché le GPU sono tipicamente dotate di cache a flusso continuo di piccole dimensioni (a differenza delle CPU, che invece impiegano cache di grandi dimensioni). Una richiesta di lettura in memoria, in genere, richiede l'intero tempo della latenza di accesso alla DRAM più la latenza di interconnessione e il tempo di caricamento dei dati nei buffer.

L'organizzazione multithreading aiuta a riempire i tempi di latenza con altre elaborazioni: mentre un thread attende il completamento del caricamento dei dati dalla memoria o del prelievo di una tessitura, il processore può eseguire un altro thread. I modelli di programmazione paralleli a grana fine generano migliaia di thread indipendenti, i quali possono mantenere impegnati molti processori nonostante le lunghe latenze di memoria viste dal singolo thread.

Un programma grafico di shading dei vertici o dei pixel è un programma per un singolo thread che elabora un vertice o un pixel. Similmente, un programma CUDA è un programma in C per un singolo thread che calcola un risultato. I programmi di grafica e di calcolo istanziano molti thread paralleli rispettivamente per generare immagini complesse e per calcolare matrici di grandi dimensioni che contengono il risultato. Per bilanciare dinamicamente i carichi di lavoro variabili nel tempo dei thread degli shader dei vertici e dei pixel, ogni multiprocessore esegue in modo concorrente molteplici programmi di thread e differenti tipi di programmi di shading.

Per supportare il modello di programmazione indipendente dei vertici, delle primitive geometriche e dei pixel dei linguaggi di shading grafico e il modello di programmazione a singolo thread del C/C++ di CUDA, ogni thread della GPU dispone dei propri registri privati, di memoria privata dedicata al thread, di un registro *program counter* e di un registro dello stato di esecuzione del thread, ed è quindi in grado di eseguire un frammento di codice in maniera indipendente. Per eseguire in modo efficiente centinaia di thread leggeri e concorrenti, il multiprocessore della GPU implementa il multithreading in hardware, ossia gestisce ed esegue centinaia di thread concorrenti in hardware, senza sovraccarichi per la pianificazione dell'esecuzione. I thread concorrenti dello stesso blocco possono sincronizzarsi a una barriera mediante una singola istruzione. La semplicità della creazione dei thread, la pianificazione dell'esecuzione dei thread senza sovraccarichi e la sincronizzazione veloce mediante barriera supportano in modo efficiente il parallelismo a grana molto fine.

L'architettura multiprocessore

Un multiprocessore unificato per grafica e calcolo deve essere in grado di eseguire programmi di shading di vertici, di geometria e di frammenti di pixel, oltre a programmi di calcolo parallelo. Come mostra la Figura C.4.1, il multiprocessore riportato come esempio contiene otto processori scalari (SP), ognuno dei quali equipaggiato di un gran numero di registri multithread (RF,

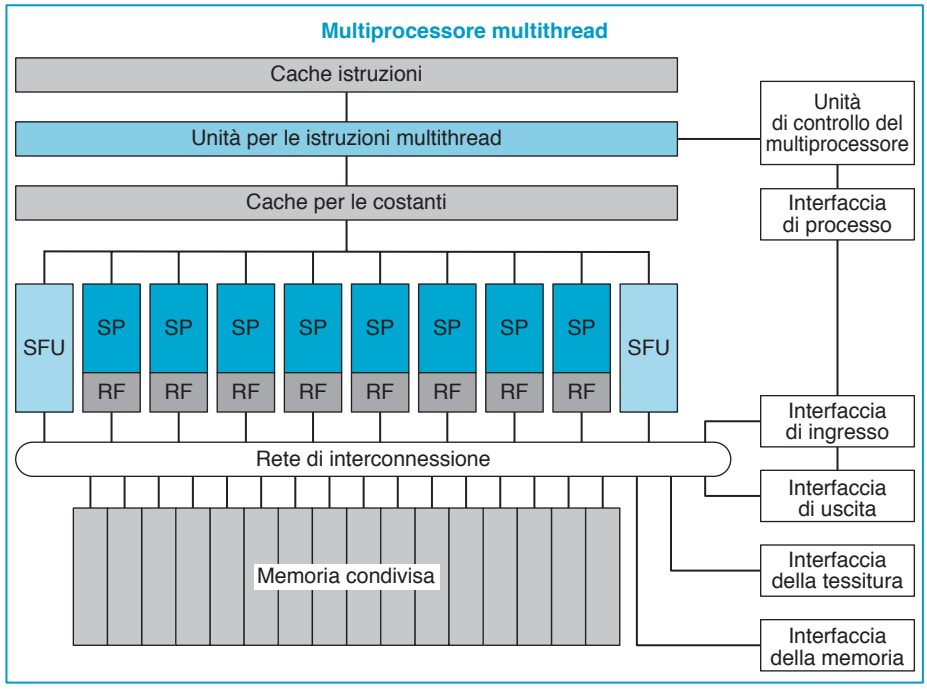


Figura C.4.1. Multiprocessore multithread contenente otto processori core scalari (SP). Gli otto core SP dispongono ciascuno di un grosso insieme di registri multithread (RF) e condividono una cache istruzioni, un'unità di lancio di istruzioni multithread, una cache per le costanti, due unità per funzioni speciali (SFU), una rete di interconnessione e una memoria condivisa a banchi multipli.

Register File), due unità per funzioni speciali (SFU), un'unità per le istruzioni multithread, una cache istruzioni, una cache a sola lettura per le costanti e una memoria condivisa.

La memoria condivisa di 16 kB contiene i buffer per i dati della grafica e i dati condivisi per il calcolo. Le variabili CUDA dichiarate come `__shared__` risiedono nella memoria condivisa. Per mappare ripetutamente il flusso di elaborazione della pipeline logica grafica sul multiprocessore, come mostrato nel Paragrafo C.2, i thread dei vertici, della geometria e dei pixel dispongono di buffer di ingresso e uscita indipendenti, e i carichi di lavoro arrivano e se ne vanno indipendentemente dall'esecuzione dei thread.

Ogni core SP contiene unità aritmetiche scalari intere e in virgola mobile che eseguono la maggior parte delle istruzioni. Il processore SP implementa il multithreading in hardware ed è in grado di gestire fino a 64 thread. La pipeline di ogni SP esegue un'istruzione scalare per ogni thread per ogni ciclo di clock, la cui frequenza può variare tra 1,2 GHz e 1,6 GHz, a seconda del modello. Ogni processore SP possiede un insieme di registri (RF) di grosse dimensioni, costituito da 1024 registri a 32 bit di utilizzo generale, partizionati tra i thread assegnati al processore. Nei programmi vengono dichiarate le richieste di registri, tipicamente da 16 a 64 registri scalari a 32 bit per ogni thread. Il processore SP può eseguire in modo concorrente molti thread che impiegano pochi registri, oppure un numero minore di thread che necessitano di più registri. Il compilatore ottimizza l'allocazione dei registri, con lo scopo di bilanciare il costo del riversamento dei registri in memoria con il costo derivante da un numero minore di thread. I programmi di shading dei pixel utilizzano spesso 16 registri o meno, permettendo a ogni SP di eseguire fino a 64 thread di shading dei pixel, coprendo così le lunghe latenze dei prelievi della tessitura. I programmi CUDA compilati necessitano spesso di 32 registri per thread, limitando così ogni SP a 32 thread e imponendo che un programma kernel abbia al massimo 256 thread per ogni blocco, anziché il massimo consentito di 512 thread (in questo multiprocessore di esempio).

La pipeline della SFU esegue istruzioni di thread che calcolano funzioni speciali e interpolano gli attributi dei pixel a partire dagli attributi primitivi dei vertici. Queste istruzioni possono essere eseguite in modo concorrente con altre istruzioni in esecuzione sugli SP. L'unità SFU verrà descritta più avanti.

Il multiprocessore esegue le istruzioni di prelievo della tessitura nell'unità di tessitura attraverso l'interfaccia della tessitura, mentre utilizza l'interfaccia della memoria per istruzioni di lettura, scrittura e accesso atomico alla memoria esterna. Queste istruzioni possono essere eseguite in maniera concorrente con le istruzioni in esecuzione sugli SP. L'accesso alla memoria condivisa utilizza una rete di interconnessione a bassa latenza tra i processori SP e i banchi di memoria condivisa.

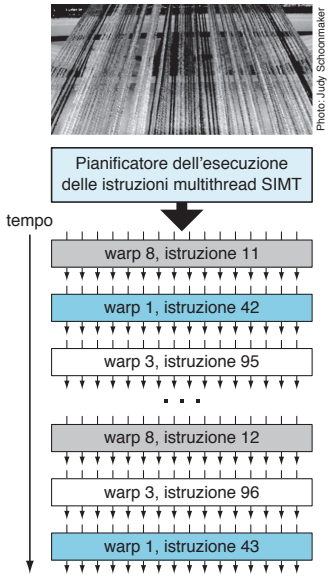


Figura C.4.2. Pianificazione dell'esecuzione di warp multithread SIMT. Il pianificatore (*scheduler*) seleziona un warp pronto per l'esecuzione e lancia in esecuzione un'istruzione in modo sincrono sui thread paralleli che costituiscono il warp. Poiché i warp sono indipendenti, il pianificatore può scegliere ogni volta un warp differente.

Singola istruzione e thread multipli (SIMT): un'architettura di processore che applica un'istruzione a molteplici thread indipendenti eseguiti in parallelo.

Warp: l'insieme dei thread paralleli che eseguono contemporaneamente la medesima istruzione in un'architettura SIMT.

Singola istruzione e thread multipli (SIMT)

Per gestire ed eseguire in modo efficiente centinaia di thread che svolgono una grande quantità di programmi diversi, il multiprocessore adotta un'architettura a **singola istruzione e thread multipli** (**SIMT**, *Single Instruction Multiple-Thread*). Essa crea, gestisce, pianifica l'esecuzione ed esegue thread concorrenti in gruppi di thread paralleli denominati **warp**. Il termine warp («ordito di un tessuto») deriva dalla tessitura a telaio, la prima tecnologia con fili (thread) in parallelo. La fotografia in Figura C.4.2 mostra un ordito (warp) di fili (thread) paralleli uscenti da un telaio. Questo esempio di multiprocessore utilizza una dimensione di warp di 32 thread ed esegue quattro thread in ciascuno degli otto core SP, in quattro cicli di clock. Anche il multiprocessore Tesla SM, che verrà descritto nel Paragrafo C.7, utilizza una dimensione di

warp di 32 thread paralleli ed esegue quattro thread per ogni core SP per ottenere un'elevata efficienza in condizioni di abbondanza di thread di pixel e di calcolo. I blocchi di thread sono costituiti da uno o più warp.

Questo esempio di multiprocessore SIMT gestisce un gruppo di 16 warp, per un totale di 512 thread. I singoli thread paralleli che compongono un warp sono dello stesso tipo e partono contemporaneamente dallo stesso indirizzo di codice, ma sono poi liberi di procedere e seguire le biforcazioni del codice in modo indipendente. Ogni volta che viene lanciata in esecuzione un'istruzione, l'unità istruzioni multithread SIMT seleziona un warp che è pronto per eseguire l'istruzione successiva e invia quell'istruzione ai thread attivi di quel warp. Un'istruzione SIMT viene diffusa in modo sincrono sui thread paralleli attivi del warp. Alcuni thread possono risultare inattivi, a causa delle biforcazioni indipendenti del codice o dell'attesa di variabili. In questo multiprocessore ogni processore core scalare SP esegue un'istruzione sui quattro singoli thread di un warp impiegando quattro cicli di clock, rispecchiando così lo stesso rapporto di 4:1 che esiste tra il numero di thread di un warp e il numero di core.

L'architettura di un processore SIMT è simile a quella dei processori a singola istruzione e dati multipli (SIMD), in cui si applica una singola istruzione a corsie di elaborazione multiple di dati. La differenza sta nel fatto che l'architettura SIMT applica un'istruzione a thread multipli indipendenti in parallelo e non semplicemente a corsie multiple di dati. Un'istruzione di un processore SIMD controlla un vettore di corsie multiple di dati, tutte assieme, mentre un'istruzione di un processore SIMT controlla un singolo thread ed è l'unità istruzioni SIMT, guadagnando in efficienza, a lanciare quell'istruzione su un warp di thread paralleli e indipendenti. Il processore SIMT trova parallelismi tra i dati fra thread in fase di esecuzione, in maniera analoga a come un processore superscalare trova parallelismi tra le istruzioni in fase di esecuzione.

Un processore SIMT ottiene piena efficienza e massime prestazioni quando tutti i thread di un warp seguono lo stesso flusso di esecuzione. Se alcuni thread divergono a causa di un salto condizionato dipendente dai dati, l'esecuzione diventa sequenziale per ognuna delle due diramazioni possibili del codice e, quando tutti i flussi di esecuzione giungono a conclusione, i thread si ricongiungono in un unico flusso. Un codice contente una biforcazione del tipo *if else* in due flussi di esecuzione di uguale lunghezza ha un'efficienza del 50%. Per gestire thread indipendenti che divergono e convergono, il multiprocessore utilizza uno stack di sincronizzazione delle diramazioni. Warp differenti vengono eseguiti in maniera indipendente fra loro alla massima velocità possibile, anche se stanno eseguendo flussi di codice comuni o disgiunti. Ne consegue che le GPU SIMT sono decisamente più efficienti e flessibili in relazione ai frammenti di codice divergenti rispetto alle GPU delle precedenti generazioni, poiché i loro warp sono molto più sottili se paragonati all'ampiezza delle istruzioni SIMD.

Diversamente dalle architetture vettoriali SIMD, le SIMT permettono ai programmatori di scrivere sia codice parallelo a livello di thread (per thread singoli e indipendenti) sia codice parallelo a livello di dati (per thread multipli coordinati tra loro). Perché il codice sia corretto, il programmatore può sostanzialmente ignorare gli attributi dell'esecuzione SIMT dei warp; tuttavia, si può ottenere un incremento sostanziale delle prestazioni avendo cura che il codice raramente abbia bisogno di divergere all'interno di un warp. In pratica, si tratta di un caso analogo al ruolo delle linee di una cache nel codice tradizionale: la dimensione della linea di cache può essere tranquillamente ignorata se l'obiettivo è la correttezza del codice, mentre deve essere considerata nella struttura del codice se l'obiettivo è raggiungere le massime prestazioni di picco.

L'esecuzione dei warp SIMT e le biforcazioni del codice

L'approccio SIMT alla pianificazione dell'esecuzione dei warp indipendenti è più flessibile della pianificazione effettuata dalle architetture GPU precedenti. Un warp comprende thread paralleli dello stesso tipo: di vertice, di geometria, di pixel o di calcolo. L'unità di base per l'elaborazione degli shader dei frammenti di pixel è il quadrilatero di 2 pixel per 2, implementato mediante quattro thread di shading di pixel. L'unità di controllo del multiprocessore racchiude i quadrilateri di pixel in un warp. Analogamente, essa raggruppa i vertici e le primitive geometriche in warp e racchiude anche i thread di calcolo in un warp. Un blocco di thread contiene uno o più warp. L'architettura SIMT condivide l'unità di prelievo e distribuzione delle istruzioni in modo efficiente tra i thread paralleli di uno stesso warp, ma richiede che tutti i thread del warp siano attivi per ottenere la massima prestazione.

Tale multiprocessore unificato pianifica l'esecuzione ed esegue molteplici tipi di warp in modo concorrente, permettendo quindi di eseguire warp di vertice e di pixel. Lo scheduler dei warp lavora a una frequenza minore della frequenza di clock del processore, perché deve riempire le corsie di quattro thread per ogni processore core; durante ogni ciclo di pianificazione, esso seleziona un warp per l'esecuzione di un'istruzione SIMT, come mostrato in Figura C.4.2. Le istruzioni di un warp vengono lanciate in esecuzione come quattro gruppi di otto thread, generando i risultati su quattro cicli di clock del processore. La pipeline del processore impiega diversi cicli di clock per completare ogni istruzione. Se il numero di warp attivi moltiplicato per il numero di cicli di clock per warp supera la latenza della pipeline, il programmatore può ignorare tale latenza. Per questo multiprocessore, uno scheduling di tipo *round robin* di otto warp ha un periodo di 32 cicli di clock tra due istruzioni successive dello stesso warp. Se il programma è in grado di mantenere attivi 256 thread per ciascun multiprocessore, latenze di istruzione fino a 32 cicli di clock possono essere tenute nascoste al singolo thread sequenziale. Tuttavia, se il numero di warp attivi è piccolo, la profondità della pipeline del processore diviene visibile e può causare lo stallo dei processori.

Un problema progettuale interessante è l'implementazione di una pianificazione dell'esecuzione dei warp con sovraccarico nullo per una combinazione dinamica di programmi di warp differenti e di tipi diversi di programma. Il pianificatore (*scheduler*) deve selezionare un warp ogni quattro cicli di clock in modo da lanciare in esecuzione un'istruzione per ciascun ciclo di clock per ogni thread, il che equivale a un IPC = 1,0 per ogni processore core. Dato che i warp sono indipendenti, le uniche dipendenze possono nascere tra le istruzioni sequenziali di uno stesso warp. Lo scheduler utilizza una tabella sulla quale annotare le dipendenze tra i registri, per identificare i warp nei quali i thread attivi sono pronti a eseguire un'istruzione. Questo rende prioritari tutti i warp pronti e seleziona quindi per il lancio in esecuzione il warp con la priorità più alta. Il calcolo della priorità deve tener conto del tipo di warp, del tipo di istruzione, nonché del desiderio di equità nei confronti di tutti i warp attivi.

La gestione dei thread e dei blocchi di thread

L'unità di controllo del multiprocessore e quella delle istruzioni gestiscono i thread e i blocchi di thread. L'unità di controllo accetta richieste di elaborazione e dati in ingresso e arbitra l'accesso alle risorse condivise, comprese l'unità della tessitura, il cammino di accesso alla memoria e i collegamenti di I/O. Per i carichi di lavoro grafici, essa crea e gestisce in modo concorrente tre tipi di thread grafici: vertici, geometria e pixel. Ognuno di questi tipi di elaborazione

grafica possiede collegamenti indipendenti di I/O e viene accumulato e raggruppato in warp SIMT di thread paralleli che eseguono lo stesso programma di thread. L'unità di controllo alloca un warp libero e i registri per i thread del warp e inizia l'esecuzione di warp sul multiprocessore. Ogni programma dichiara il numero di registri per thread di cui ha bisogno; l'unità di controllo fa partire un warp solo quando è in grado di allocare il numero di registri richiesto. Quando tutti i thread di warp hanno terminato l'esecuzione, l'unità di controllo distribuisce i risultati e libera i registri e le risorse del warp.

L'unità di controllo crea **insiemi di thread cooperativi** (**CTA**, *Cooperative Thread Arrays*), i quali implementano blocchi di thread CUDA sottoforma di uno o più warp di thread paralleli. Essa crea un CTA quando può produrre tutti i warp del CTA e allocare tutte le risorse richieste dal CTA. Oltre ai thread e ai registri, un CTA richiede l'allocazione di memoria condivisa e le barriere. Prima di lanciare il CTA, il programma dichiara la quantità di memoria richiesta e l'unità di controllo attende fino a quando può allocare una quantità di memoria sufficiente. Si producono quindi i warp del CTA con la frequenza con cui vengono lanciati in esecuzione i warp, in modo tale che un programma CTA parta eseguendo da subito il codice con le prestazioni massime del multiprocessore. L'unità di controllo si accorge quando tutti i thread di un CTA hanno terminato, e libera quindi le risorse condivise utilizzate dal CTA e dai suoi warp.

Insieme di thread cooperativi (CTA): un insieme di thread concorrenti che esegue lo stesso programma di thread e può cooperare per calcolare un risultato. Un CTA di GPU implementa un blocco di thread CUDA.

Le istruzioni dei thread

I processori di thread SP eseguono istruzioni scalari sul singolo thread, a differenza delle precedenti architetture GPU che eseguivano istruzioni su vettori con quattro componenti per ogni programma di shading di vertici o di pixel. I programmi di shading dei vertici in genere calcolano vettori di posizione (x, y, z, w), mentre i programmi di shading dei pixel calcolano vettori di colore (rosso, verde, blu, alfa). Tuttavia, i programmi di shading stanno diventando sempre più lunghi e più scalari, ed è quindi sempre più difficile impegnare completamente anche solo due delle componenti di un'architettura vettoriale a quattro componenti delle precedenti GPU. In effetti, l'architettura SIMT parallelizza l'esecuzione su 32 thread di pixel indipendenti, anziché parallelizzare le quattro componenti vettoriali di un pixel. I programmi C/C++ di CUDA contengono in prevalenza codice scalare per il singolo thread. Le GPU precedenti impiegavano la tecnica del *vector packing* (cioè il raggruppamento dell'elaborazione in sottovettori, con lo scopo di guadagnare efficienza), ma ciò rendeva più complesso sia l'hardware dello scheduler sia il compilatore, dato che questo riesce a gestire più facilmente le istruzioni scalari. Le istruzioni per la tessitura rimangono vettoriali, accettando in ingresso un vettore sorgente di coordinate e restituendo un vettore di colore filtrato.

Per supportare GPU diverse, con differenti formati binari delle microistruzioni, i compilatori dei linguaggi ad alto livello di grafica e di calcolo generano istruzioni intermedie a livello assembler (per esempio, le istruzioni vettoriali Direct3D o le istruzioni scalari PTX), che vengono poi ottimizzate e tradotte nelle microistruzioni binarie della GPU. La definizione dell'insieme di istruzioni PTX di NVIDIA (*Parallel Thread eXecution*) (2007) ha fornito ai compilatori un'architettura dell'insieme di istruzioni (ISA) stabile, garantendo la compatibilità per parecchie generazioni di GPU con l'insieme delle microistruzioni che sono in continua evoluzione. Nella fase di ottimizzazione, vengono semplicemente espanse le istruzioni vettoriali Direct3D in microistruzioni binarie scalabili multiple. Le istruzioni scalari PTX sono tradotte quasi con corrispondenza uno a uno in microistruzioni binarie scalari, sebbene alcune PTX vengano espanse in più microistruzioni e istruzioni

multiple PTX possano essere tradotte in un'unica microistruzione. Poiché le istruzioni intermedie a livello assembler utilizzano registri virtuali, in fase di ottimizzazione si analizzano le dipendenze tra i dati e allocati i registri reali. L'ottimizzazione, inoltre, elimina le parti di codice sorgente che non verranno mai eseguite (il cosiddetto «dead code»), concentra le istruzioni in un'unica istruzione (quando possibile) e ottimizza i punti di divergenza e convergenza delle diramazioni del codice SIMT.

L'architettura dell'insieme delle istruzioni (ISA)

L'architettura dell'insieme delle istruzioni (ISA) per i thread qui descritta è una versione semplificata dell'ISA PTX dell'architettura Tesla, un insieme di istruzioni scalari a registri che comprende operazioni su interi, in virgola mobile, logiche e di conversione, funzioni speciali, istruzioni di controllo del flusso e di accesso alla memoria, e operazioni sulla tessitura. In Figura C.4.3 sono elencate le istruzioni PTX di base per i thread di una GPU; per i dettagli si rimanda alle specifiche PTX di NVIDIA (2007). Il formato di un'istruzione è:

```
opcode.type d, a, b, c;
```

dove `d` è l'operando destinazione, `a`, `b` e `c` sono gli operandi sorgente e `.type` è uno dei seguenti tipi di dati:

Tipo	Specificatore: .type
Bit non tipizzati: 8, 16, 32 e 64 bit	.b8, .b16, .b32, .b64
Intero senza segno: 8, 16, 32 e 64 bit	.u8, .u16, .u32, .u64
Intero con segno: 8, 16, 32 e 64 bit	.s8, .s16, .s32, .s64
Virgola mobile: 16, 32 e 64 bit	.f16, .f32, .f64

Gli operandi sorgente sono numeri scalari su 32 o 64 bit contenuti nei registri, numeri immediati o costanti; gli operandi predicato sono numeri booleani su 1 bit. Gli operandi destinazione sono registri, a eccezione dell'istruzione di scrittura in memoria. Le istruzioni vengono trasformate in istruzioni eseguite in modo condizionato applicando loro il prefisso `@p` oppure `@!p`, dove `p` è il registro predicato. Le istruzioni di memoria e tessitura trasferiscono scalari o vettori contenenti da due a quattro componenti, fino a 128 bit in totale. Le istruzioni PTX specificano il comportamento di un thread.

Le istruzioni aritmetiche PTX operano su numeri in virgola mobile su 32 o 64 bit, e su interi con e senza segno. Le GPU più recenti supportano i dati e le operazioni in virgola mobile su 64 bit in doppia precisione (si veda il Paragrafo C.6). Sulle attuali GPU, le istruzioni PTX su 64 bit, intere e logiche, vengono tradotte in due o più microistruzioni binarie che eseguono operazioni su 32 bit. Le istruzioni per le funzioni speciali delle GPU sono limitate ai dati in virgola mobile su 32 bit. Le istruzioni di controllo di flusso nei thread sono le istruzioni di salto condizionato (`branch`), di chiamata (`call`) e di ritorno (`return`) da procedura e di uscita dal thread (`exit`), e la sincronizzazione a barriera (`bar.sync`). L'istruzione di salto condizionato `@p bra target` utilizza un registro predicato `p` (o `!p`), precedentemente scritto da un'istruzione di confronto e assegnamento di predicato (`setp`), per determinare se il thread debba eseguire il salto. Anche altre istruzioni possono essere predicate, eseguite cioè sulla base del valore vero o falso di un registro predicato.

Le istruzioni di accesso alla memoria

L'istruzione `tex` preleva e filtra i campioni di tessitura dalle matrici di tessitura 1D, 2D e 3D contenute in memoria mediante il sottosistema di tessitura. Il

Istruzioni PTX di base per i thread di una GPU

Gruppo	Istruzione	Esempio	Significato	Commenti
Aritmetiche	aritmetiche .*type* = .s32, .u32, .f32, .s64, .u64, .f64			
	add.*type*	add.f32 d, a, b	d = a + b;	
	sub.*type*	sub.f32 d, a, b	d = a – b;	
	mul.*type*	mul.f32 d, a, b	d = a * b;	
	mad.*type*	add.f32 d, a, b	d = a*b + c;	moltiplica e somma
	div.*type*	div.f32 d, a, b	d = a / b;	microistruzioni multiple
	rem.*type*	rem.f32 d, a, b	d = a % b;	resto intero
	abs.*type*	abs.f32 d, a	d = |a|;	
	neg.*type*	neg.f32 d, a	d = 0 – a;	
	min.*type*	min.f32 d, a, b	d = (a<b) ? a:b;	v.m.: seleziona non NaN
	max.*type*	max.f32 d, a, b	d = (a>b) ? a:b;	v.m.: seleziona non NaN
	setp.cmp.*type*	setp.lt.f32 p, a, b	p = (a < b);	confronta e assegna il predicato
	comparazione numerica: .*cmp* = eq, ne, lt, gt, ge; comparazione elementi non ordinati: .*cmp* = equ, neu, ltu, gtu, geu, num, nan			
	mov.*type*	mov.b32 d, a;	d = a;	copia
	selp.*type*	selp.f32 d, a, p, b	d = p? a:b;	seleziona in base al predicato
	cvt.type.*atype*	cvt.f32.s32 d, a	d=converti(a);	conversione da atype in dtype
Funzioni speciali	funzione speciale: .*type* = .f32 (per alcune .f64)			
	rcp.*type*	rcp.f32 d, a	d = 1/a;	reciproco
	sqrt.*type*	sqrt.f32 d, a	d =sqrt(a);	radice quadrata
	rsqrt.*type*	rsqrt.f32, d, a	d = 1/sqrt(a);	reciproco della radice quadrata
	sin.*type*	sin.f32, d, a	d = sin(a)	seno
	cos.*type*	cos f32, d, a	d = cos(a)	coseno
	lg2.*type*	lg2.f32, d, a	d = log(a)/log(2)	logaritmo in base 2
	ex2.*type*	ex2.f32, d, a	d = 2 ** a;	esponenziale in base 2
Logiche	logiche .*type* = .pred, .b32, .b64			
	and.*type*	and.b32 d, a, b	d = a & b;	reciproco
	or.*type*	or.b32 d, a, b	d = a | b;	radice quadrata
	xor.*type*	xor.b32 d, a, b	d = a ^ b;	reciproco della radice quadrata
	not.*type*	not.b32 d, a, b	d = ~a;	complemento a 1
	cnot.*type*	cnot.b32 d, a, b	d = (a==0) ? 1:0;	operazione not logica del C
	shl.*type*	shl.b32 d, a, b	d = a << b;	scorrimento a sinistra
	shr.*type*	shr.b32 d, a, b	d = a >> b;	scorrimento a destra
Accesso a memoria	memoria .*space* = .global, shared, local, const; .*type* = .b8, .u8, .s8, .b16, .b32, .b64			
	ld.space.*type*	ld.global .b32, d, [a+off]	d=*(a+off);	lettura di *space* da memoria
	st.space.*type*	st.shared.b32 [d+off], a	*(d+off)=a;	Scrittura di *space* in memoria
	tex.nd.dtyp.*btype*	tex.2d.v4.f32.f32 d, a, b	d=tex2d(a,b);	lettura di un campione della tessitura
	atom.spc.op.*type*	atom.global.add.u32 d,[a],b atom.global.cas.b32 d,[a],b,c	atomic {d=*a; *a=op(*a,b);	operazione atomica di lettura-modifica-scrittura
	atom .*op* = and, or, xor, add, min, max, exch, cas; .*spc* = .global; .*type* = .b32			

Gruppo	Istruzione	Esempio	Significato	Commenti
Controllo di flusso	branch	`@p bra target`	`if (p) goto target;`	salto condizionato
	call	`call (ret), func, (param)`	`ret=func (param);`	chiamata a funzione
	ret	`ret`	`return;`	ritorno da funzione
	`bar.sync`	`bar.sync d`	`wait for threads; /*attesa dei thread*/`	sincronizzazione a barriera
	exit	`Exit`	`exit;`	termine dell'esecuzione di un thread

Figura C.4.3. Istruzioni PTX di base per i thread di una GPU.

prelievo della tessitura in genere viene effettuato utilizzando coordinate interpolate, in virgola mobile. Quando un thread grafico di shading di pixel calcola il colore di un pixel di un frammento, il processore che esegue le operazioni di rasterizzazione miscela il colore del pixel con il colore alla posizione (x, y) e scrive in memoria il colore risultante.

Per supportare le richieste del calcolo e del linguaggio C/C++, l'ISA PTX Tesla è dotato di istruzioni di lettura e scrittura della memoria. Queste si basano sull'indirizzamento intero al byte, utilizzando indirizzi formati dal contenuto di un registro a cui viene sommato uno spiazzamento, per facilitare la convenzionale ottimizzazione di codice da parte del compilatore. Le istruzioni di lettura/scrittura della memoria sono comuni nei processori, ma rappresentano una novità significativa nelle GPU con architettura Tesla, poiché le precedenti GPU fornivano soltanto gli accessi ai pixel e alla tessitura richiesti dalle API della grafica.

Per le applicazioni di calcolo, le istruzioni di lettura e scrittura accedono ai tre spazi di memoria di lettura/scrittura che implementano i corrispondenti tre spazi di memoria CUDA descritti nel Paragrafo C.3:

- Memoria locale, per i dati temporanei privati di ogni singolo thread, implementata nella DRAM esterna.
- Memoria condivisa, per l'accesso a bassa latenza ai dati condivisi dai thread che cooperano nello stesso CTA/blocco di thread, implementata in SRAM sul chip.
- Memoria globale, per grandi quantità di dati condivise da tutti i thread di un'applicazione di calcolo, implementata nella DRAM esterna.

Le istruzioni di lettura e scrittura della memoria `ld.global`, `st.global`, `ld.shared`, `st.shared`, `ld.local` e `st.local` accedono rispettivamente agli spazi di memoria globale, condivisa e locale. I programmi di calcolo utilizzano l'istruzione di sincronizzazione veloce a barriera, `bar.synch`, per sincronizzare i thread all'interno di un CTA o blocco di thread, i quali comunicano tra loro per mezzo della memoria condivisa e di quella globale.

Per incrementare la larghezza di banda della memoria e ridurre il sovraccarico di lavoro, le istruzioni load/store della memoria globale e locale fondono tutte le richieste dei singoli thread paralleli di uno stesso warp SIMT nella richiesta di un unico blocco di memoria, quando gli indirizzi cadono all'interno dello stesso blocco e soddisfano i criteri di allineamento. La fusione delle richieste dei dati dalla memoria fornisce un notevole incremento delle prestazioni rispetto a richieste separate da parte dei singoli thread. L'elevato numero di thread del multiprocessore, unito alle richieste di lettura di grandi

quantità di dati, aiuta a colmare le latenze dovute alla lettura della memoria locale e globale implementate nella DRAM esterna.

Le più recenti GPU con architettura Tesla, inoltre, forniscono efficienti operazioni atomiche sulla memoria mediante le istruzioni `atom.op.u32`, tra cui le operazioni su interi `add`, `min`, `max`, `and`, `or`, `xor`, `exchange` e `cas` (confronta e scambia di posto), facilitando le operazioni di traduzione del codice sequenziale in codice parallelo e la gestione delle strutture dati parallele.

La sincronizzazione a barriera per la comunicazione tra i thread

La sincronizzazione veloce a barriera permette ai programmi CUDA di comunicare frequentemente attraverso la memoria condivisa e la memoria globale, semplicemente chiamando la funzione `__syncthreads()` come parte di ogni passo di comunicazione tra i thread. La funzione intrinseca di sincronizzazione genera una singola istruzione `bar.sync`. Tuttavia, implementare una sincronizzazione veloce a barriera tra thread per i 512 thread di ogni blocco CUDA è molto complesso.

Raggruppando i thread in warp SIMT di 32 thread si riduce la complessità della sincronizzazione di un fattore 32. I thread attendono a una barriera all'interno dello scheduler dei thread SIMT, in modo da non utilizzare cicli di processore durante l'attesa. Quando un thread esegue un'istruzione `bar.sync`, incrementa il contatore del numero di thread arrivati alla barriera e lo scheduler registra che il thread è in attesa. Quando tutti i thread del CTA sono arrivati alla barriera, il conteggio corrisponderà al numero di thread e lo scheduler libererà tutti i thread in attesa, facendo riprendere la loro esecuzione.

I processori a flusso continuo (SP)

Un processore core multithread a flusso continuo (*streaming processor*) è l'elaboratore principale delle istruzioni nel multiprocessore. Il suo insieme di registri (RF) fornisce 1024 registri scalari a 32 bit per un massimo di 64 thread ed è in grado di eseguire tutte le operazioni fondamentali in virgola mobile, tra cui `add.f32`, `mul.f32`, `mad.f32` (moltiplica e somma in virgola mobile), `min.f32`, `max.f32`, `setp.f32` (confronta in virgola mobile e imposta il predicato). Le operazioni di somma e prodotto in virgola mobile sono compatibili con lo standard IEEE 754 per i numeri in virgola mobile in singola precisione, compresi i valori *not a number* (NaN) e infinito. Il core SP implementa anche tutte le istruzioni PTX di aritmetica intera, confronto, conversione e istruzioni logiche riportate nella tabella di Figura C.4.3.

Le operazioni `add` e `mul` in virgola mobile utilizzano la modalità di arrotondamento *round to nearest even* (al numero pari più vicino) IEEE come modalità di arrotondamento predefinita. L'istruzione *multiply add* (moltiplica e somma) in virgola mobile, `mad.f32`, effettua una moltiplicazione con troncamento seguita da una somma con arrotondamento *round to nearest even*. Il processore SP trasforma gli operandi in ingresso di tipo denormalizzato nel numero zero, preservandone il segno. I risultati che vanno al di sotto dell'intervallo degli esponenti rappresentabili dopo l'arrotondamento vengono anch'essi trasformati in zero (preservandone il segno).

Le unità per le funzioni speciali (SFU)

Alcune istruzioni dei thread possono essere eseguite sulla SFU in modo concorrente con altre istruzioni di thread che vengono eseguite sui processori SP. La SFU esegue le istruzioni per le funzioni speciali riportate in Figura C.4.3, le

quali calcolano l'approssimazione in virgola mobile su 32 bit del reciproco di un numero, del reciproco della radice quadrata e delle principali funzioni trascendenti. Essa implementa anche l'interpolazione piana in virgola mobile su 32 bit per gli shader dei pixel, fornendo interpolazioni accurate degli attributi, come il colore, la profondità e le coordinate di tessitura.

La pipeline dalla SFU produce il risultato di una funzione speciale in virgola mobile a 32 bit per ciclo di clock; le due SFU contenute in ciascun multiprocessore eseguono quindi istruzioni di funzioni speciali a un quarto della frequenza di esecuzione delle istruzioni semplici eseguite dagli otto SP. Le SFU eseguono anche le istruzioni di moltiplicazione `mul.f32` in parallelo agli otto SP, aumentando così la velocità di calcolo di picco fino al 50% per i thread che presentano la combinazione adatta di istruzioni.

Per il calcolo delle funzioni, la SFU dell'architettura Tesla utilizza un'approssimazione quadratica basata sull'approssimazione minimax migliorata per approssimare il reciproco di un numero, il reciproco della radice quadrata e le funzioni $\log_2 x$, 2^x, seno e coseno. L'accuratezza della stima delle funzioni varia da 22 a 24 bit di mantissa. Per maggiori dettagli sull'aritmetica delle SFU, si veda il Paragrafo C.6.

Confronto con altri multiprocessori

In confronto ad altre architetture vettoriali SIMD, come l'SSE dell'x86, il multiprocessore SIMT può eseguire singoli thread in maniera indipendente invece di eseguirli sempre insieme a gruppi sincroni. L'hardware SIMT ricerca il parallelismo tra i dati nei thread indipendenti, mentre l'hardware SIMD richiede che sia il software a esprimere in modo esplicito il parallelismo tra i dati in ogni istruzione vettoriale. Un'architettura SIMT esegue un warp di 32 thread in modo sincrono quando i thread seguono lo stesso flusso di esecuzione, ma può eseguire ogni thread in modo indipendente quando il flusso di esecuzione dei thread diverge. Il vantaggio è notevole, perché i programmi e le istruzioni SIMT descrivono semplicemente il comportamento di un singolo thread indipendente, anziché un vettore di dati SIMD con quattro o più corsie di elaborazione dei dati. Nonostante ciò, il multiprocessore SIMT presenta un'efficienza pari a un'architettura SIMD, distribuendo l'area e il costo di un'unità istruzioni sui 32 thread di un warp e sugli otto processori core a flusso continuo. L'approccio SIMT unisce l'efficienza dei processori SIMD e la produttività del multithreading, eliminando la necessità di codificare esplicitamente i vettori SIMD per le condizioni al contorno e per le divergenze parziali del flusso di elaborazione.

Il multiprocessore SIMT impone solamente un leggero sovraccarico di lavoro, essendo il multithread implementato in hardware, con sincronizzazione hardware a barriera. Ciò permette agli shader grafici e ai thread CUDA di esprimere un parallelismo a grana molto fine. Grafica e programmi CUDA utilizzano i thread per esprimere un parallelismo tra i dati a grana fine per ogni thread del programma, piuttosto che forzare il programmatore a esprimere il parallelismo con istruzioni vettoriali SIMD. È più semplice e più produttivo sviluppare codice scalare per un singolo thread che codice vettoriale. Inoltre, il multiprocessore SIMT esegue il codice con un'efficienza pari ai processori SIMD.

Accoppiando otto processori core a flusso continuo in un multiprocessore e utilizzando un numero scalabile di tali multiprocessori, si crea un multiprocessore a due livelli costituito a sua volta da multiprocessori. Il modello di programmazione CUDA sfrutta tale gerarchia a due livelli fornendo thread individuali per il calcolo parallelo a grana fine e griglie di blocchi di thread per le operazioni parallele a grana grossa. Lo stesso programma strutturato in

thread può effettuare operazioni sia a grana fine sia a grana grossa. Per contro, le CPU con istruzioni vettoriali SIMD devono utilizzare due differenti modelli di programmazione per effettuare operazioni a grana fine e a grana grossa: thread paralleli a grana grossa su core differenti e istruzioni vettoriali SIMD per il parallelismo dei dati a grana fine.

I multiprocessori multithread in sintesi

L'esempio di multiprocessore GPU basato su architettura Tesla che abbiamo visto è fortemente multithread, poiché è in grado di eseguire fino a 512 thread leggeri in modo concorrente che supportano shader di pixel e thread CUDA. Esso impiega una variante dell'architettura SIMD e un approccio multithreading chiamato SIMT (singola istruzione, thread multipli) per distribuire in modo efficiente un'istruzione a un warp di 32 thread paralleli, pur permettendo a ogni thread di scegliere diramazioni del codice ed essere eseguito in modo indipendente. Ogni thread esegue il proprio flusso di istruzioni su uno degli otto processori core a flusso continuo (SP) ad architettura multithreading, per un massimo di 64 thread.

L'architettura dell'insieme delle istruzioni (ISA) PTX è un'ISA scalare di tipo lettura/scrittura a registri che descrive l'esecuzione di un singolo thread. Dato che le istruzioni PTX sono ottimizzate e tradotte in microistruzioni binarie per la specifica GPU, le istruzioni hardware possono evolvere rapidamente senza dover riprogettare da capo i compilatori e gli strumenti software che generano le istruzioni PTX.

C.5 Il sistema parallelo di memoria

Al di fuori della GPU stessa, il sottosistema di memoria è il componente più importante per le prestazioni di un sistema grafico. I carichi di lavoro della grafica richiedono una velocità di trasferimento molto alta da e verso la memoria. Le operazioni di scrittura e miscelazione (lettura-modifica-scrittura) dei pixel, la lettura e la scrittura del buffer di profondità e la lettura delle mappe di tessitura, oltre alla lettura dei comandi e dei dati dei vertici o degli attributi degli oggetti, costituiscono la maggior parte del traffico di memoria.

Le GPU moderne sono fortemente parallele, come mostrato in Figura C.2.5; per esempio, la GeForce 8800 è in grado di elaborare 32 pixel per ciclo di clock, a 600 MHz. Ogni pixel tipicamente richiede la lettura e la scrittura del colore e della profondità, rappresentate complessivamente su 4 byte. Solitamente per generare il colore di un pixel vengono letti in media due o tre *texel* (elementi di tessitura), ciascuno di 4 byte. Vengono quindi richiesti, in media, 28 byte/pixel per ciascuno dei 32 pixel, pari a 896 byte per ciclo di clock. Chiaramente la banda richiesta al sistema di memoria è enorme.

Per soddisfare questi requisiti, i sistemi di memoria delle GPU possiedono le seguenti caratteristiche:

- Ampiezza elevata, nel senso che è presente un elevato numero di piedini per trasferire i dati tra la GPU e i suoi dispositivi di memoria, e che la struttura a matrice della memoria stessa è composta da molti chip di DRAM, in modo da poter sfruttare tutta la larghezza del bus dati.
- Elevata velocità, nel senso che sono implementate tecniche aggressive di segnalazione per massimizzare la velocità di trasferimento dei dati (bit/s) per ogni piedino del chip.
- Le GPU cercano di utilizzare ogni ciclo di clock disponibile per trasferire dati da o verso la matrice della memoria. Per raggiungere questo obiettivo,

nelle GPU non si cerca di minimizzare la latenza del sistema di memoria: un throughput elevato (efficienza di utilizzo) e una latenza bassa sono intrinsecamente in conflitto.
– Vengono impiegate tecniche di compressione, sia con perdita di informazioni (eventualità di cui il programmatore deve essere ben consapevole) sia senza perdita, le quali vengono applicate in maniera opportunistica e risultano trasparenti alle applicazioni.
– Cache e strutture che fondono le richieste di trasferimento vengono utilizzate per ridurre la quantità di traffico richiesto fuori dal chip e per garantire che i cicli di clock spesi per il trasferimento dei dati vengano sfruttati nella maniera più completa possibile.

Considerazioni sulle DRAM

Le GPU devono tener conto delle caratteristiche particolari delle DRAM. I chip di DRAM sono organizzati internamente in banchi multipli (tipicamente da quattro a otto), dove ciascun banco contiene un numero di righe pari a una potenza di 2 (tipicamente 16 384) e ciascuna riga contiene di solito un numero di bit che è potenza di 2 (tipicamente 8192). Le DRAM impongono al processore che le controlla una serie di requisiti di temporizzazione. Per esempio, vengono richieste dozzine di cicli di clock per attivare una riga ma, una volta attivata, i bit in essa contenuti sono accessibili, nello stesso tempo, attraverso il loro indirizzo di colonna, e si può leggere uno di questi bit ogni quattro cicli. Le DRAM sincrone DDR (*Double Data Read*) trasferiscono dati sul fronte sia di salita sia di discesa del clock dell'interfaccia (si veda il Capitolo 5). Una DRAM DDR con clock a 1 GHz trasferisce quindi dati al ritmo di 2 Gb/s per ogni piedino del chip. Le DRAM DDR per la grafica hanno di solito 32 piedini per il trasferimento bidirezionale dei dati e si possono quindi leggere o scrivere in una DRAM fino a otto byte per ciclo di clock.

Le GPU internamente possiedono un gran numero di generatori di traffico di memoria. Ciascuno dei diversi stadi della pipeline grafica logica genera il proprio flusso di richieste: prelievo di comandi e di attributi dei vertici, prelievo e lettura/scrittura della tessitura per gli shader, lettura/scrittura della profondità e del colore dei pixel. Per ogni stadio logico, ci sono spesso unità multiple indipendenti che producono risultati in parallelo. Ognuno di questi è un generatore indipendente di richieste di trasferimento con la memoria. Dal punto di vista del sistema di memoria, si osserva un numero enorme di richieste indipendenti in arrivo. Ciò risulta essere in naturale contrasto con l'organizzazione di riferimento preferita dalle DRAM. Una possibile soluzione è che il controllore di memoria mantenga heap separati per il traffico collegato ai differenti banchi di DRAM; il controllore attenderà che ci sia una quantità sufficiente di traffico pendente per una specifica riga di DRAM, per poi attivare la riga e trasferire tutti i dati in una volta. Si noti che accumulare richieste pendenti è vantaggioso per la località di riga della DRAM, e quindi per l'uso efficiente del bus dati, ma causa latenze medie più lunghe verso i componenti che hanno effettuato le richieste di trasferimento, poiché le richieste devono aspettare che arrivino altre richieste. Nella progettazione del sistema occorre tener conto della necessità che nessuna richiesta debba aspettare troppo a lungo, altrimenti è possibile che alcune unità di elaborazione finiscano per provocare l'inattività anche dei processori vicini.

I sottosistemi di memoria delle GPU sono organizzati in *partizioni multiple* di memoria, ciascuna delle quali comprende un controllore della memoria completamente indipendente e uno o due dispositivi DRAM che appartengono esclusivamente a quella partizione. Per ottenere il miglior bilanciamento del carico e avvicinarsi quindi alle massime prestazioni teoriche di n parti-

zioni, gli indirizzi vengono interlacciati in modo fine e uniforme su tutte le partizioni di memoria. Il passo di interlacciamento di ciascuna partizione è tipicamente un blocco di qualche centinaio di byte. Il numero di partizioni di memoria viene progettato in modo tale da bilanciare il numero dei processori e degli altri dispositivi che generano richieste di trasferimento dati con la memoria.

Le memorie cache

I carichi di lavoro delle GPU presentano insiemi di lavoro di grandi dimensioni, dell'ordine delle centinaia di megabyte, per generare una singola immagine grafica. A differenza delle CPU, non è pratico costruire una cache sul chip grande abbastanza da contenere qualcosa che si avvicini all'intero insieme di lavoro di un'applicazione grafica. Mentre le CPU possono contare su altissimi tassi di hit della cache (99,9% e oltre), le GPU sperimentano tassi di hit vicini al 90%, e devono perciò fare i conti con molti casi di miss durante l'elaborazione. Mentre una CPU può ragionevolmente essere progettata per mettersi in attesa quando si presenta una delle rare miss, una GPU ha la necessità di proseguire anche in presenza di miss e hit mescolate fra loro. Un'architettura di questo tipo è detta *architettura cache a flusso continuo*.

Le cache delle GPU devono fornire una banda notevolmente larga ai loro utilizzatori. Consideriamo il caso di una cache di tessitura. Una tipica unità di tessitura è in grado di calcolare due interpolazioni bilineari per ciascun gruppo di quattro pixel, per ogni ciclo di clock; una GPU può contenere molte di queste unità di tessitura, tutte operanti in modo indipendente. Ciascuna interpolazione bilineare richiede quattro texel separati, e ogni texel potrebbe essere un numero su 64 bit; quattro componenti da 16 bit sono tipici. Ne risulta una larghezza di banda totale di $2 \times 4 \times 4 \times 64 = 2048$ bit per ciclo di clock. Ogni singolo texel di 64 bit ha un indirizzo indipendente, quindi la cache deve gestire 32 indirizzi indipendenti per ciclo di clock. Ciò favorisce naturalmente un'organizzazione multibanco e/o multiporta delle matrici di SRAM.

L'unità di gestione della memoria (MMU)

Le moderne GPU sono in grado di mappare indirizzi virtuali su indirizzi fisici. Nella GeForce 8800, tutte le unità di elaborazione generano indirizzi di memoria in uno spazio di indirizzamento virtuale a 40 bit. Per il calcolo, le istruzioni dei thread per l'accesso a memoria (load e store) utilizzano indirizzi su 32 bit, i quali vengono estesi a indirizzi virtuali di 40 bit aggiungendo un offset su 40 bit. Un'unità di gestione della memoria effettua la traduzione degli indirizzi virtuali nei corrispondenti indirizzi fisici; l'hardware legge le tabelle delle pagine dalla memoria locale per rispondere alle miss per conto di una gerarchia di buffer di traduzione degli indirizzi (chiamati TLB, *Transition Lookaside Buffer*) sparsa tra i processori e i motori di rendering. Oltre ai bit delle pagine fisiche, gli elementi di una tabella delle pagine di una GPU specificano l'algoritmo di compressione utilizzato in ogni pagina. Le dimensioni delle pagine vanno da 4 a 128 KB.

Le aree di memoria

Come accennato nel Paragrafo C.3, CUDA utilizza differenti spazi di memoria per permettere al programmatore di memorizzare i dati nella maniera ottimale dal punto di vista delle prestazioni. La descrizione seguente si riferisce alle GPU con architettura Tesla di NVIDIA.

La memoria globale

La memoria globale (*global memory*) risiede nella DRAM esterna e non è locale a nessuno dei multiprocessori a flusso continuo (SM), poiché è pensata per la comunicazione tra i diversi CTA (blocchi di thread) di griglie diverse. Di fatto, i numerosi CTA che accedono a una stessa locazione della memoria globale potrebbero essere eseguiti nella GPU in tempi differenti; in CUDA, il programmatore non conosce la sequenza con cui vengono eseguiti i CTA (si tratta di un vincolo progettuale). Poiché lo spazio di indirizzamento è distribuito uniformemente fra tutte le partizioni di memoria, deve essere presente un cammino di lettura/scrittura che collega ogni processore a flusso continuo a ogni partizione della DRAM.

Non è garantito che la sequenza degli accessi alla memoria globale da parte di thread differenti (e da differenti processori) sia consistente. I programmi dei thread vedono un modello di ordinamento della memoria rilassato. All'interno di un thread viene preservato l'ordine delle letture e delle scritture a uno stesso indirizzo, mentre l'ordine degli accessi a indirizzi differenti può non essere mantenuto. Le letture e scritture della memoria richieste da thread differenti non mantengono l'ordinamento temporale. All'interno di un CTA, l'istruzione di sincronizzazione a barriera `bar.sync` può essere utilizzata per ottenere una sequenza temporale rigorosa dei thread del CTA, e l'istruzione di thread `membar` è un'operazione di «barriera/recinto» della memoria che registra gli accessi precedenti alla memoria e li rende visibili agli altri thread prima di procedere. I thread possono anche avvalersi delle operazioni atomiche sulla memoria, descritte nel Paragrafo C.4, per coordinare il lavoro sulla memoria da essi condivisa.

La memoria condivisa

La memoria condivisa (*shared memory*), dedicata a ciascun CTA, è visibile soltanto ai thread che appartengono a quel CTA e alloca spazio di memoria solo dall'istante in cui il CTA viene creato fino all'istante in cui termina; per questo motivo la memoria condivisa può risiedere sul chip. Questo approccio presenta molti vantaggi: anzitutto il traffico della memoria condivisa non ha bisogno di competere con la limitata banda di trasferimento verso l'esterno del chip, utilizzata per l'accesso alla memoria globale; in secondo luogo, risulta efficiente costituire strutture di memoria con banda elevatissima sul chip per supportare le richieste di lettura/scrittura di ciascun multiprocessore a flusso continuo. Di fatto, la memoria condivisa è strettamente accoppiata al multiprocessore a flusso continuo.

Ogni multiprocessore a flusso continuo contiene otto processori di thread. Durante un ciclo di clock della memoria condivisa, ciascun processore può elaborare le istruzioni di due thread, per cui in ogni ciclo di clock devono essere gestite le richieste alla memoria condivisa di 16 thread. Poiché ogni thread può generare i propri indirizzi (e normalmente tali indirizzi sono unici), la memoria condivisa è costruita utilizzando 16 banchi di SRAM indirizzabili in maniera indipendente. Per gruppi di indirizzi comuni, 16 banchi sono sufficienti per sostenere il flusso degli accessi, ma si possono verificare dei casi patologici «anomali»: per esempio, potrebbe accadere che tutti i 16 thread accedano a indirizzi differenti dello stesso banco di SRAM; deve essere possibile trasferire una richiesta proveniente da ciascuna corsia di thread a un qualsiasi banco di SRAM, per cui è necessaria una rete di interconnessione 16 × 16.

La memoria locale

La memoria locale (*local memory*), dedicata a ciascun thread, è una memoria privata visibile soltanto al singolo thread. Dal punto di vista architetturale,

è più grande dell'insieme dei registri del thread. Per permettere di allocare vaste aree di questa memoria (ricordiamo che lo spazio totale da allocare è lo spazio di allocazione di ciascun thread moltiplicato per il numero di thread attivi), si utilizza la DRAM esterna.

Sebbene la memoria globale e quella locale di ciascun thread risiedano all'esterno del chip, ben si prestano ad avere una cache sul chip.

La memoria delle costanti

La memoria delle costanti (*constant memory*) è una memoria di sola lettura per i programmi che vengono eseguiti sul multiprocessore SM, e può essere scritta mediante opportuni comandi inviati alla GPU. Essa risiede nella DRAM esterna e utilizza la cache del SM. Dato che comunemente tutti o la maggior parte dei thread di un warp SIMT leggono le costanti dallo stesso indirizzo, è sufficiente l'accesso a un unico indirizzo per ciclo di clock. La cache delle costanti è progettata per distribuire valori scalari a tutti i thread di ogni warp.

La memoria di tessitura

La memoria di tessitura (*texture memory*) contiene matrici di dati di grandi dimensioni ed è a sola lettura. Le tessiture utilizzate nel calcolo generico hanno gli stessi attributi e funzionalità delle tessiture utilizzate per la grafica 3D. Sebbene le tessiture siano costituite di solito da immagini bidimensionali (matrici 2D di valori di pixel), sono disponibili anche tessiture 1D (lineari) e 3D (di volume).

Un programma di calcolo referenzia una tessitura utilizzando l'istruzione `tex`. Gli operandi comprendono un identificatore del nome della tessitura e 1, 2 o 3 coordinate a seconda del numero di dimensioni della tessitura. Nelle coordinate (espresse in virgola mobile), la parte frazionaria specifica la posizione, spesso situata tra due texel. Le coordinate non intere richiamano un'interpolazione bilineare pesata (per una tessitura 2D) dei quattro campioni più vicini per poi restituire il risultato al programma.

La tessitura prelevata viene messa in una cache gerarchica a flusso continuo, progettata per ottimizzare un flusso continuo di prelievi di tessitura da parte di migliaia di thread concorrenti. Alcuni programmi utilizzano il prelievo della tessitura come metodo per portare in cache la memoria globale.

Le superfici

Superficie è un termine generico per indicare una matrice mono-, bi- o tridimensionale di valori di pixel e un formato a essi associato. Si possono definire diversi formati: per esempio, un pixel può essere definito come quaterna di componenti intere di 8 bit RGBA, oppure come quaterna di componenti in virgola mobile di 16 bit. Un kernel di programma non ha bisogno di conoscere il tipo di superficie. L'istruzione `tex` converte il risultato in numeri in virgola mobile, in funzione del formato della superficie.

Lettura e scrittura in memoria

Le istruzioni di lettura e scrittura con indirizzamento intero al byte permettono lo sviluppo e la compilazione di programmi in linguaggi convenzionali, quali il C e il C++. I programmi CUDA utilizzano istruzioni di load/store per accedere alla memoria.

Per aumentare la larghezza di banda del trasferimento con la memoria e ridurre il carico di lavoro aggiuntivo, le istruzioni di load/store della memoria locale e globale fondono le richieste individuali dei thread paralleli dello stesso warp in un'unica richiesta di blocco di memoria, a condizione che gli indirizzi siano riferiti allo stesso blocco e soddisfino opportuni criteri di allineamento. La fusione di tante piccole richieste individuali di memoria nella richiesta di un blocco di grandi dimensioni (coalescenza) porta a un significativo incremento delle prestazioni rispetto alle richieste separate. Il grande numero di thread dei multiprocessori unito al supporto a un numero elevato di richieste di lettura pendenti, aiuta a riempire le latenze di lettura della memoria locale e globale, costituita da DRAM esterna.

I ROP

Come mostrato in figura C.2.5, le GPU con architettura Tesla di NVIDIA contengono una schiera scalabile di processori a flusso continuo (SPA, *Streaming Processor Array*) che svolge tutti i calcoli programmabili della GPU, e un sistema di memoria scalabile che comprende il controllo della memoria DRAM esterna e i processori a funzione prefissata per le operazioni di rasterizzazione (ROP, *Raster Operation Processors*), i quali effettuano le operazioni sul colore e sulla profondità nel frame buffer, operando direttamente sulla memoria. Ciascuna unità ROP è accoppiata a una specifica partizione di memoria e ciascuna partizione ROP viene alimentata dai multiprocessori SM attraverso una rete di interconnessione. Ogni ROP è responsabile dei controlli sulla profondità e sugli stencil, nonché della miscelatura del colore. I ROP e i controllori della memoria collaborano all'implementazione della compressione senza perdita (fino a 8:1) del colore e della profondità, per ridurre la richiesta di banda di trasferimento verso l'esterno. Le unità ROP svolgono anche operazioni atomiche sulla memoria.

C.6 Aritmetica in virgola mobile

Le GPU attuali effettuano la maggior parte delle operazioni aritmetiche nei processori core programmabili utilizzando operazioni in virgola mobile su 32 bit a singola precisione, compatibili con lo standard IEEE 754 (si veda il Capitolo 3). L'aritmetica in virgola fissa delle GPU precedenti è stata soppiantata da quella in virgola mobile su 16 bit, poi su 24 bit, su 32 bit, e infine dall'aritmetica in virgola mobile su 32 bit compatibile con l'IEEE 754. Alcune funzioni logiche della GPU, come l'hardware per il filtraggio della tessitura, continuano ad avvalersi di formati numerici proprietari. Le GPU recenti forniscono anche istruzioni in virgola mobile su 64 bit a doppia precisione compatibili con l'IEEE 754.

I formati supportati

Lo standard IEEE 754 per l'aritmetica in virgola mobile (2008) definisce i formati di base e di memorizzazione. Le GPU utilizzano due dei formati di base per il calcolo: il formato binario a virgola mobile su 32 e 64 bit, comunemente chiamati singola precisione e doppia precisione. Lo standard specifica anche un formato a virgola mobile per la memorizzazione di numeri binari su 16 bit, denominato **mezza precisione**. Le GPU e il linguaggio di shading Cg utilizzano questo formato compatto su 16 bit per ottenere una memorizzazione e un trasferimento efficiente, pur mantenendo un'elevata gamma dinamica. Le GPU effettuano un gran numero di operazioni di filtraggio della tessitura e di miscelazione dei pixel su valori in mezza precisione all'interno delle unità di filtraggio della tessitura e delle operazioni di rasterizzazione. Il formato di file per le

Mezza precisione: un formato binario su 16 bit per i numeri in virgola mobile con 1 bit di segno, 5 bit di esponente e 10 bit di parte frazionaria; prevede un uno implicito prima della virgola.

immagini ad alta gamma dinamica OpenEXR, sviluppato da Industrial Light and Magic (2003), utilizza questo formato a mezza precisione per i valori delle componenti di colore in applicazioni di sintesi digitale di immagini e film.

L'aritmetica di base

Moltiplica e somma (MAD): una singola istruzione in virgola mobile che esegue un'operazione composta da una moltiplicazione seguita da una somma.

Le operazioni più comuni in virgola mobile in singola precisione nei core programmabili delle GPU comprendono l'addizione, la moltiplicazione, la moltiplicazione e somma (*multiply-add*), il minimo, il massimo, il confronto, l'impostazione di un predicato e la conversione tra numeri interi e in virgola mobile. Spesso le istruzioni in virgola mobile forniscono modificatori degli operandi sorgente per ottenere la loro negazione o il loro valore assoluto.

Nella maggior parte delle GPU attuali, le operazioni di addizione e moltiplicazione in virgola mobile sono compatibili con lo standard IEEE 754 per i numeri in virgola mobile a singola precisione, compresi i valori NaN e infinito. Le operazioni di addizione e moltiplicazione in virgola mobile utilizzano come modalità predefinita di arrotondamento l'«arrotondamento al numero pari più vicino» (*round to nearest even*) dello standard IEEE. Per aumentare il throughput dell'esecuzione delle istruzioni in virgola mobile, le GPU utilizzano spesso un'istruzione composta da moltiplicazione e somma (`mad`). L'operazione **moltiplica e somma** (**MAD**) esegue una moltiplicazione in virgola mobile con troncamento seguita da un'addizione in virgola mobile con arrotondamento al numero pari più vicino. Essa esegue due istruzioni in virgola mobile in un solo ciclo di clock, senza bisogno che lo scheduler lanci due istruzioni separate, ma le due operazioni non vengono fuse assieme e il troncamento del prodotto viene effettuato prima dell'addizione. Questo rende l'istruzione diversa dall'istruzione di moltiplicazione e somma integrate, illustrata nel terzo capitolo e utilizzata più avanti in questo Paragrafo. Le GPU tipicamente convertono in zero gli operandi sorgente denormalizzati preservando il segno, e convertono a zero preservando il segno dopo l'arrotondamento anche i risultati che si trovano al di sotto della gamma degli esponenti rappresentabili.

L'aritmetica specializzata

Unità per le funzioni speciali (SFU): un'unità hardware che calcola funzioni speciali e interpola attributi bidimensionali.

Le GPU sono provviste di hardware per accelerare il calcolo di alcune funzioni speciali, l'interpolazione degli attributi e il filtraggio della tessitura. Le funzioni speciali comprendono coseno, seno, esponenziale e logaritmo binario (in base 2), reciproco e reciproco della radice quadrata. Le istruzioni di interpolazione degli attributi consentono di calcolare in modo efficiente gli attributi dei pixel mediante la valutazione dell'equazione associata sul piano. L'**unità per le funzioni speciali** (**SFU**, *Special Function Unit*), presentata nel Paragrafo C.4, calcola le funzioni speciali e interpola gli attributi sul piano (Oberman e Siu, 2005).

Esistono diversi metodi per calcolare le funzioni speciali in hardware. È dimostrato che l'interpolazione quadratica basata su approssimazioni migliorate di tipo minimax è un metodo molto efficiente per approssimare alcune funzioni in hardware, tra cui il reciproco, il reciproco della radice quadrata, il $\log_2 x$, la funzione 2^x, il seno e il coseno.

Descriviamo ora sinteticamente il metodo di interpolazione quadratica utilizzato dalle SFU. Per un operando binario in ingresso X caratterizzato da un significando di n bit, il significando viene suddiviso in due parti: chiamiamo X_s la parte superiore contenente m bit e X_i la parte inferiore contenente $n - m$ bit. Gli m bit superiori, X_s, vengono utilizzati per consultare un insieme di tre tabelle di corrispondenza le quali forniscono tre coefficienti su un

numero finito di bit: C_0, C_1 e C_2. Ogni funzione da approssimare richiede un proprio insieme di tabelle. Questi coefficienti vengono utilizzati per approssimare nell'intervallo $X_s \le X < X_i + 2^{-m}$ la funzione data, f(X), con la seguente funzione polinomiale:

$$f(X) = C_0 + C_1 X_1 + C_2 X_1^2$$

L'accuratezza delle stime della funzione varia da 22 a 24 bit del significando. Alcune valutazioni dell'accuratezza dell'approssimazione per alcune funzioni sono riportate in Figura C.6.1.

Lo standard IEEE 754 specifica i requisiti richiesti dall'arrotondamento esatto per la divisione e la radice quadrata; tuttavia per molte applicazioni della GPU i requisiti possono non essere rispettati. Per tali applicazioni, un maggior throughput dei calcoli è più importante dell'accuratezza dell'ultimo bit. Per le funzioni speciali della SFU, la libreria matematica di CUDA fornisce sia una versione con la massima accuratezza sia una funzione veloce con l'accuratezza dell'istruzione della SFU.

Un'altra operazione aritmetica specializzata di una GPU è l'interpolazione degli attributi. Gli attributi principali sono solitamente specificati per i vertici delle primitive che costituiscono la scena da generare. Esempi di attributo sono il colore, la profondità e le coordinate di tessitura. Tali attributi devono essere interpolati nello spazio bidimensionale dello schermo, dovendo determinare il valore degli attributi per ogni pixel. Il valore di un certo attributo, U, sul piano (x, y) può essere espresso utilizzando una funzione della forma

$$U(x, y) = A_u x + B_u y + C_u$$

dove A, B e C sono i parametri di interpolazione associati a ogni attributo U. Tali parametri sono tutti rappresentati come numeri in virgola mobile a singola precisione.

Poiché ciascun processore di shading dei pixel deve sia valutare delle funzioni sia interpolatore degli attributi, si può progettare una singola SFU che faccia entrambe le cose in modo efficiente: le due funzioni, infatti, utilizzano un'operazione di somma di prodotti per interpolare i risultati; inoltre, il numero di termini da sommare è quasi lo stesso.

Le operazioni sulla tessitura

La mappatura e il filtraggio della tessitura costituiscono un altro insieme fondamentale di operazioni aritmetiche in virgola mobile specializzate di una GPU. Le operazioni utilizzate per la mappatura della tessitura comprendono:

1. Ricevere l'indirizzo della tessitura (s, t) per il pixel corrente (x, y), dove s e t sono valori in virgola mobile a singola precisione.

Funzione	Intervallo di supporto	Accuratezza (bit validi)	Errore misurato in ULP*	% arrotondamento esatto	Monotonicità
1/x	[1, 2)	24,02	0,98	87	Sì
1/sqrt(x)	[1, 4)	23,40	1,52	78	Sì
2^x	[0,1)	22,51	1,41	74	Sì
$\log_2 x$	[1,2)	22,57	N/A**	N/A	Sì
sin/cos	[0,π/2)	22,47	N/A	N/A	No

* ULP: unità in ultima posizione (si veda il Capitolo 3).
** N/A: non applicabile

Figura C.6.1. Statistiche sull'approssimazione delle funzioni speciali. Sono riferite all'unità per le funzioni speciali (SFU) della scheda GeForce 8800 di NVIDIA.

2. Calcolare il livello di dettaglio per identificare il corretto livello **MIP-map** della tessitura.
3. Calcolare i coefficienti frazionari dell'interpolazione trilineare.
4. Scalare l'indirizzo della tessitura (s, t) a seconda del livello MIP-map selezionato.
5. Accedere alla memoria e recuperare i texel richiesti.
6. Eseguire le operazioni di filtraggio dei texel.

MIP-map: dalla frase latina *multum in parvo*, molto in poco spazio. Una MIP-map contiene immagini precalcolate a differenti risoluzioni che vengono utilizzate per aumentare la velocità di rendering e per ridurre gli artefatti.

La mappatura della tessitura richiede un numero elevato di operazioni in virgola mobile quando viene effettuata alla massima velocità, e la gran parte di queste operazioni viene effettuata su valori in mezza precisione su 16 bit. Per esempio, la scheda grafica GeForce 8800 Ultra è in grado di eseguire circa 500 GFLOPS in virgola mobile su variabili in formato proprietario per ciascuna istruzione di mappatura della tessitura, oltre alle convenzionali istruzioni IEEE in virgola mobile a singola precisione. Per maggiori dettagli sulla mappatura e filtraggio della tessitura, si consulti Foley e van Dam (1995).

Le prestazioni

L'hardware aritmetico per l'addizione e la moltiplicazione in virgola mobile è organizzato in una struttura completamente a pipeline e la latenza è ottimizzata per bilanciare i ritardi e l'occupazione di area. Nonostante siano organizzate in pipeline, il throughput delle funzioni speciali è inferiore a quello delle addizioni e delle moltiplicazioni in virgola mobile: un throughput pari a un quarto della velocità è oggi una misura tipica per le funzioni speciali di una moderna GPU, contenente una SFU condivisa da quattro core SP. D'altra parte, le CPU presentano tipicamente un throughput inferiore per le funzioni analoghe, come la divisione e la radice quadrata, sebbene siano in grado di calcolarle con un'accuratezza maggiore. L'hardware di interpolazione degli attributi, tipicamente, presenta una struttura completamente a pipeline per ottenere lo shading dei pixel alla massima velocità possibile.

La doppia precisione

Le GPU più recenti, come la Tesla T10P, supportano in hardware anche le operazioni in virgola mobile in doppia precisione su 64 bit secondo lo standard IEEE 754. Le operazioni aritmetiche standard in virgola mobile in doppia precisione comprendono addizione, moltiplicazione e conversione tra i differenti formati interi e in virgola mobile. Lo standard per la virgola mobile IEEE 754 del 2008 contiene anche le specifiche per l'istruzione di moltiplicazione e somma integrate (FMA) descritta nel Capitolo 3. L'istruzione FMA esegue una moltiplicazione in virgola mobile seguita da un'addizione, con un solo arrotondamento al termine delle due operazioni; le operazioni di moltiplicazione e addizione integrate mantengono la piena accuratezza nei calcoli intermedi. Questa caratteristica consente di ottenere una maggiore accuratezza nei calcoli in virgola mobile che prevedono l'accumulo di prodotti parziali, come per esempio i prodotti scalari, il prodotto di matrici e la valutazione dei polinomi. L'istruzione FMA consente anche di implementare in software in modo efficiente la divisione e la radice quadrata, arrotondate in modo esatto, eliminando la necessità di disporre di un'unità hardware per le divisioni e per il calcolo delle radici quadrate.

Un'unità FMA hardware a doppia precisione implementa su 64 bit l'addizione, la moltiplicazione, le conversioni e l'operazione FMA stessa. L'architettura di una FMA a doppia precisione fornisce il supporto alla massima velocità consentita dei numeri denormalizzati sia sugli input sia sugli output. La Figura C.6.2 mostra uno schema a blocchi di un'unità FMA.

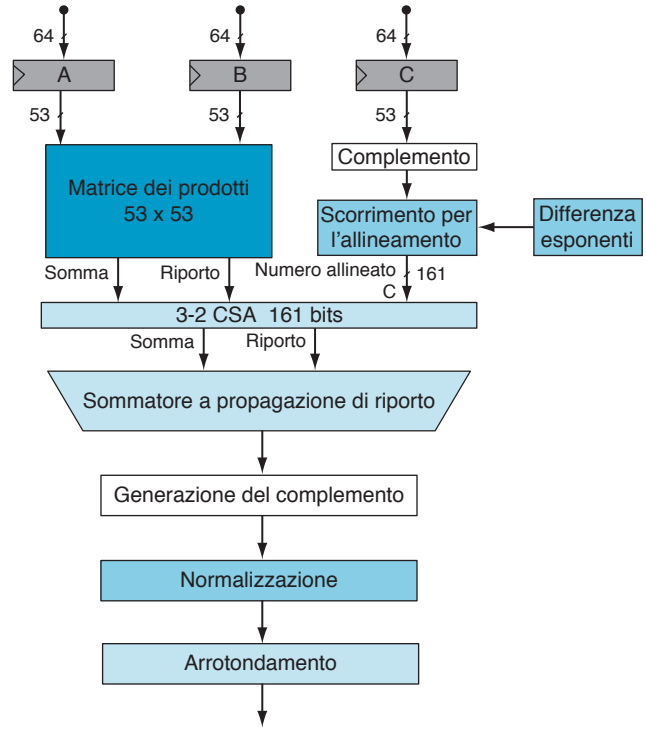


Figura C.6.2. Unità di moltiplicazione e somma fuse (FMA). Hardware utilizzato per implementare l'operazione A × B + C in doppia precisione.

Come mostrato in Figura C.6.2, i significandi di A e B vengono moltiplicati formando un prodotto su 106 bit, e il risultato viene scritto con l'ultimo bit di riporto salvato (*carry-save*). In parallelo, viene calcolato il complemento dell'addendo C, se necessario, su 53 bit e allineato al prodotto su 106 bit. Il risultato della somma e il bit di riporto del prodotto su 106 bit vengono poi sommati all'addendo allineato per mezzo di un sommatore di tipo carry-save (CSA, si veda l'Appendice C) a 161 bit. L'uscita in formato carry-save viene quindi sommata da un sommatore a propagazione di riporto, producendo il risultato non arrotondato in complemento a 2. Se necessario, viene calcolato il complemento del risultato in modo da rappresentare il risultato in formato modulo e segno e viene quindi normalizzato e successivamente arrotondato per adattarlo al formato previsto.

C.7 Un caso reale: la GeForce 8800 di NVIDIA

La GPU NVIDIA GeForce 8800, presentata nel novembre 2006, è un'architettura unificata per l'elaborazione di vertici e di pixel che supporta anche applicazioni di calcolo parallelo scritte in C utilizzando il modello di programmazione parallela CUDA. Si tratta della prima implementazione dell'architettura unificata di grafica e calcolo Tesla, descritta nel Paragrafo C.4 e in Lindholm, Nickolls, Oberman e Montrym (2008). La famiglia di GPU basate su Tesla consente di soddisfare le diverse esigenze dei computer portatili, dei desktop, delle workstation e dei server.

La schiera di processori a flusso continuo (SPA, *Streaming Processor Array*)

La GPU GeForce 8800 mostrata in Figura C.7.1 contiene 128 processori core a flusso continuo (SP, *Streaming Processor*) organizzati come 16 multiprocessori a flusso continuo (SM). Coppie di SM condividono un'unità di tessitura all'in-

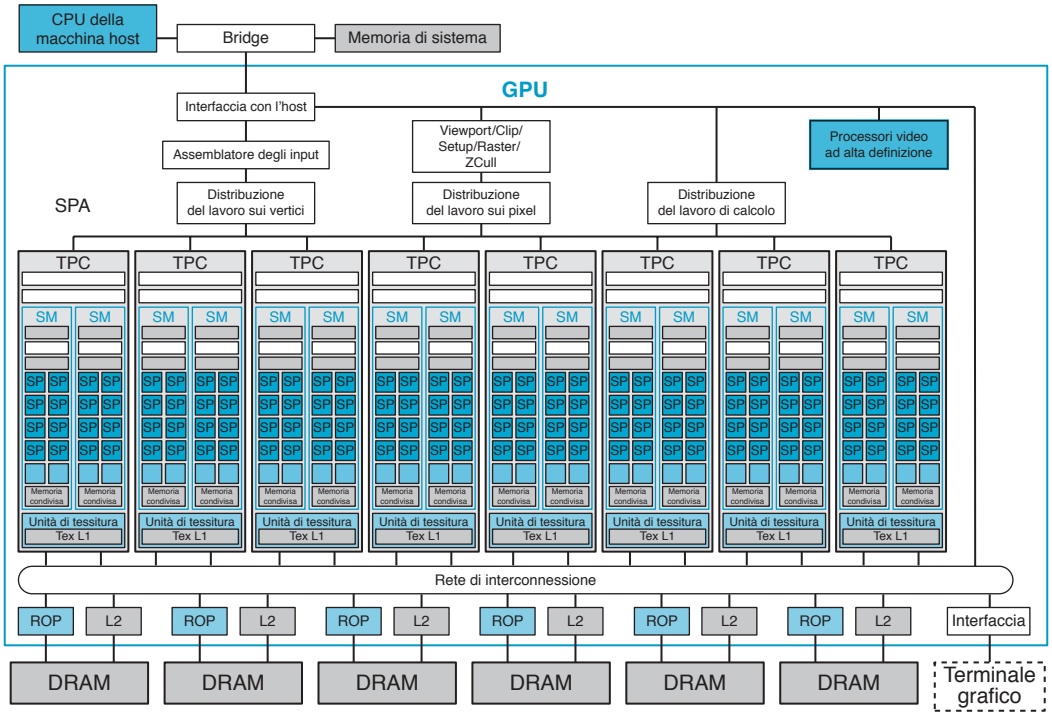


Figura C.7.1. Architettura di GPU unificata per calcolo e grafica Tesla di NVIDIA. Questa scheda grafica GeForce 8800 contiene 128 processori core a flusso continuo (SP) distribuiti su 16 multiprocessori a flusso continuo (SM), organizzati in otto nuclei tessitura/processore (TPC). I processori sono collegati a sei partizioni di DRAM con ampiezza di 64 bit attraverso una rete di interconnessione. In altre GPU che implementano l'architettura Tesla varia il numero dei core SP, dei SM, delle partizioni DRAM e delle altre unità.

terno di un nucleo processore/tessitura (TPC, *Texture/Processor Cluster*). Un insieme di otto TPC costituisce la schiera di processori a flusso continuo (SPA) che esegue tutti i programmi, sia di shading sia di calcolo.

L'unità di interfacciamento con l'host comunica con la CPU della macchina attraverso il bus PCI-Express, controlla la consistenza dei comandi ed effettua gli scambi di contesto. L'assemblatore degli input aggrega le primitive geometriche (punti, linee, triangoli); i blocchi di distribuzione del lavoro distribuiscono vertici, pixel e gruppi di thread di calcolo ai TPC appartenenti alla schiera degli SPA. I TPC possono eseguire sia programmi di shading di geometria e di pixel sia programmi di calcolo. I dati geometrici in uscita vengono inviati al blocco *viewport/clip/setup/raster/zcull* (inquadratura/taglio/impostazione/rasterizzazione/selezione mediante Z-buffer) per essere rasterizzati trasformandoli in frammenti di pixel che vengono poi ridistribuiti agli SPA per eseguire i programmi di shading dei pixel. I pixel generati vengono quindi inviati attraverso la rete di interconnessione per essere elaborati dalle unità ROP. La rete instrada anche verso la DRAM le richieste di lettura della memoria di tessitura da parte della SPA e legge i dati dalla DRAM trasferendoli alla SPA mediante una cache di livello 2.

Il nucleo tessitura/processore (TPC)

Ogni TPC contiene un controllore di geometria, un controllore di multiprocessore (SMC), due multiprocessori a flusso continuo (SM) e un'unità di tessitura, come mostrato in Figura C.7.2.

Il controllore della geometria mappa la pipeline logica grafica dei vertici sui processori fisici SM, indirizzando tutto il flusso degli attributi e della topologia dei vertici e delle primitive al TPC.

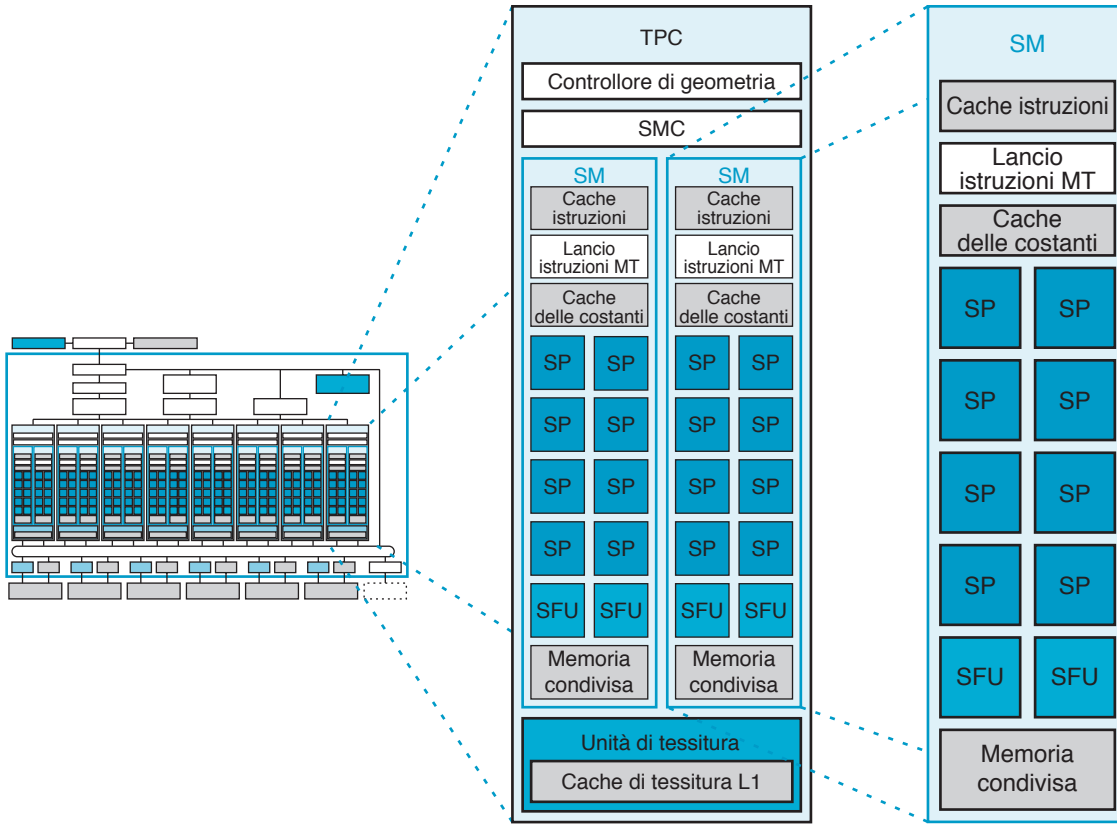


Figura C.7.2. Nucleo tessitura/processore (TPC) e multiprocessore a flusso continuo (SM). Ciascun SM contiene otto processori core a flusso continuo (SP), due SFU e una memoria condivisa.

Un controllore SMC controlla diversi multiprocessori SM, gestendo l'arbitraggio dell'unità di tessitura condivisa e dei percorsi di load/store e di I/O. Il controllore SMC gestisce contemporaneamente tre carichi di lavoro grafico: vertici, geometria e pixel.

L'unità di tessitura elabora, a ogni ciclo di clock, un'istruzione di tessitura per ogni vertice, di primitiva geometrica o di quadrilatero di pixel, oppure quattro thread di calcolo. I dati sorgente delle istruzioni di tessitura sono le coordinate di tessitura, mentre le uscite sono campioni pesati tipicamente codificati mediante le quattro componenti RGBA, espresse in virgola mobile. L'unità di tessitura è strutturata in una pipeline profonda. Nonostante contenga una cache a flusso continuo per catturare la località del filtraggio, essa prevede la presenza di hit e miss mescolate nel flusso degli accessi alla cache senza causare stalli.

Il multiprocessore a flusso continuo (SM)

L'SM è un multiprocessore unificato per grafica e calcolo che esegue sia programmi di shading di vertici, geometria e pixel sia programmi di calcolo parallelo. Il multiprocessore SM è costituito da otto processori core SP di thread, due SFU, un'unità di prelievo e lancio in esecuzione delle istruzioni multithread, una cache per le istruzioni, una cache a sola lettura per le costanti e una memoria condivisa di lettura/scrittura da 16 kB. Esso esegue istruzioni scalari sui singoli thread.

La scheda GeForce 8800 Ultra sincronizza i core SP e le unità SFU con un clock a 1,5 GHz, per una prestazione di picco di 36 GFLOP per singolo SM.

Per ottimizzare l'efficienza energetica e di occupazione di area, alcune unità dell'SM ausiliarie, non incluse nel flusso di elaborazione dei dati, lavorano a una frequenza di clock dimezzata.

Per eseguire in modo efficiente centinaia di thread paralleli di molti programmi diversi, il multiprocessore SM gestisce il multithreading in hardware; in questo modo è in grado di gestire ed eseguire fino a 768 thread concorrenti in hardware, senza alcun sovraccarico nella gestione della pianificazione. Ogni thread possiede un proprio stato di esecuzione e può seguire un cammino di elaborazione indipendente.

Un warp consiste di un massimo di 32 thread dello stesso tipo: vertice, geometria, pixel o calcolo. L'architettura SIMT, descritta nel Paragrafo C.4, condivide in modo efficiente l'unità di prelievo e lancio delle istruzioni, ma richiede un warp completo di thread attivi per raggiungere le prestazioni massime.

L'SM pianifica ed esegue molteplici tipi di warp in modo concorrente: in ogni istante in cui vengono lanciate istruzioni in esecuzione, lo scheduler seleziona uno dei 24 warp, lancia in esecuzione l'istruzione di tipo SIMT. L'istruzione del warp viene eseguita su quattro insiemi di 8 thread in quattro cicli di clock del processore. Le unità SP e SFU eseguono istruzioni in modo indipendente e lo scheduler riesce a mantenere entrambe le unità pienamente occupate lanciando istruzioni alternativamente sulle due unità. Una tabella assegna un punteggio a ogni warp pronto per il lancio a ogni ciclo di clock. Lo scheduler delle istruzioni rende prioritari tutti i warp pronti e seleziona per il lancio in esecuzione quello a priorità più elevata. Il calcolo della priorità è basato sul tipo di warp, sul tipo di istruzione e su un criterio di equità tra tutti i warp in esecuzione sull'SM.

L'SM esegue vettori di thread cooperativi (CTA) come warp multipli concorrenti che accedono a un'area di memoria condivisa allocata dinamicamente per il CTA stesso.

L'insieme delle istruzioni

I thread eseguono istruzioni scalari, a differenza delle precedenti architetture GPU a istruzioni vettoriali. Le istruzioni scalari sono più intuitive e semplici da compilare. Le istruzioni per le tessiture rimangono basate su uno schema vettoriale: ricevono un vettore di coordinate sorgente e restituiscono un vettore di colore filtrato.

L'insieme delle istruzioni basate su registri comprende tutte le istruzioni aritmetiche, intere e in virgola mobile, le funzioni trascendenti, logiche, di controllo di flusso, di lettura/scrittura e di tessitura riportate nella tabella delle istruzioni PTX di Figura C.4.3. Le istruzioni di lettura/scrittura utilizzano l'indirizzamento intero al byte che viene formato sommando al contenuto di un registro uno spiazzamento. Per i programmi di calcolo, le istruzioni di lettura/scrittura possono accedere in lettura e in scrittura ai tre spazi di memoria: la memoria locale per i dati privati e temporanei di ogni thread, la memoria condivisa, a bassa latenza, contenente i dati condivisi tra i diversi thread dello stesso CTA, e la memoria globale per i dati condivisi da tutti i thread. I programmi di calcolo utilizzano l'istruzione di sincronizzazione veloce a barriera, `bar.sync`, per sincronizzare in modo veloce i thread dello stesso CTA che comunicano tra loro per mezzo della memoria condivisa o della memoria globale. Le GPU più recenti con architettura Tesla implementano operazioni PTX atomiche sulla memoria che facilitano la parallelizzazione del codice e la gestione parallela delle strutture dati.

Il processore a flusso continuo (SP)

Il core SP multithread del processore è principalmente un processore di thread, come descritto nel Paragrafo C.4. Il suo insieme di registri è costituito

da 1024 registri scalari a 32 bit, a disposizione di un massimo di 96 thread, un numero maggiore di thread rispetto all'SP considerato nell'esempio del Paragrafo C.4. Le operazioni di addizione e moltiplicazione in virgola mobile che esegue sono compatibili con lo standard IEEE 754 per i numeri in virgola mobile a singola precisione, compresi i valori NaN e infinito. Le operazioni di addizione e moltiplicazione utilizzano il criterio di arrotondamento al numero pari più vicino specificato da IEEE come criterio di arrotondamento predefinito. Il core SP implementa anche tutte le istruzioni PTX di aritmetica intera, di confronto di conversione, e le istruzioni logiche su 32 e 64 bit riportate in Figura C.4.3. Il processore è completamente strutturato in pipeline e la latenza è ottimizzata per bilanciare ritardi e area occupata.

L'unità per funzioni speciali (SFU)

La SFU supporta sia il calcolo di funzioni trascendenti sia l'interpolazione planare di attributi. Come descritto nel Paragrafo C.6, essa sfrutta l'interpolazione quadratica basata su uno schema evoluto di approssimazione minimax per approssimare le funzioni reciproco, reciproco della radice quadrata, $\log_2 x$, 2^x, seno e coseno, ed è in grado di produrre un valore per ciclo di clock. La SFU supporta inoltre l'interpolazione di attributi di pixel, tra cui il colore, la profondità e le coordinate di tessitura al ritmo di quattro campioni per ciclo di clock.

La rasterizzazione

Le primitive geometriche passano, nello stesso ordine sequenziale con cui sono state ricevute in ingresso, dagli SM al blocco *viewport/clip/setup/raster/zcull*. Le unità di *viewport* e di *clipping* (taglio) eliminano le primitive al di fuori del tronco di piramide visualizzato e di eventuali piani di taglio definiti dall'utente, e trasformano poi la posizione dei vertici dallo spazio tridimensionale allo spazio dell'immagine, cioè in coordinate di pixel.

Le primitive sopravvissute al taglio entrano nell'unità di setup, la quale genera le equazioni di bordo per la rasterizzazione. Un primo stadio di rasterizzazione grossolana genera tutte le finestrelle di pixel che si trovano anche solo parzialmente all'interno della primitiva. L'unità di taglio lungo la profondità (*zcull*) conserva una gerarchia delle superfici lungo l'asse *z* della profondità ed elimina, in modo conservativo, le finestrelle di pixel che risultano già occupate da pixel disegnati in precedenza: un numero massimo di 256 pixel possono essere eliminati a ogni ciclo di clock. I pixel che sopravvivono al *zculling* entrano in uno stadio di rasterizzazione fine che genera informazioni dettagliate sull'aspetto della superficie e sulla profondità.

Il controllo sulla profondità e l'aggiornamento possono essere effettuati prima o dopo lo shading del frammento, a seconda dello stato corrente. Il controllore SMC aggrega i pixel sopravvissuti in warp da far elaborare a un SM che sta eseguendo lo shader sul pixel corrente. L'SMC invia quindi tali pixel e i dati loro associati al ROP.

Il processore delle operazioni di rasterizzazione (ROP) e il sistema di memoria

Ogni ROP è associato a una specifica partizione di memoria. Per ogni frammento di pixel emesso da un programma di shading, i ROP effettuano il controllo su profondità e stencil, l'aggiornamento e, in parallelo, la miscelazione del colore e gli aggiornamenti relativi. Per ridurre la banda di trasferimento con la DRAM, viene adottata una compressione senza perdita di dati del colore (fino a 8:1) e della profondità (fino a 8:1). Ogni ROP ha una velocità di picco

di quattro pixel per ciclo di clock e supporta i formati HDR in virgola mobile a 16 e 32 bit. I ROP supportano il calcolo della profondità a velocità doppia quando la scrittura del colore è disabilitata.

Il supporto dell'antialiasing comprende fino a 16 operazioni di campionamento multiplo e di sovracampionamento (*supersampling*). L'algoritmo di antialiasing basato sul campionamento dell'aspetto della superficie (CSAA) calcola e memorizza valori booleani su un massimo di 16 campioni e comprime l'informazione ridondante sul colore, sulla profondità e sugli stencil nello spazio di memoria per una banda di trasferimento compresa tra quattro e otto campioni, con lo scopo di ottimizzare le prestazioni.

L'ampiezza del bus dati della memoria DRAM è di 384 piedini, organizzati in sei partizioni indipendenti di 64 piedini ciascuna. Ogni partizione supporta il protocollo a doppio trasferimento dei dati DDR2 e il protocollo per la grafica GDDR3 a una frequenza massima di 1.0 GHz, ottenendo una larghezza di banda di 16 GB/s per partizione per un totale di 96 GB/s.

I controllori della memoria supportano una vasta gamma di frequenze di clock, protocolli, densità dei dispositivi e ampiezze del bus dati delle DRAM. Le richieste di tessitura e di load/store possono arrivare da qualsiasi TPC a qualsiasi partizione di memoria, per cui una rete di interconnessione instrada sia le richieste sia le risposte.

Scalabilità

L'architettura unificata Tesla è stata progettata per essere scalabile. Variando il numero di SM, TPC, ROP, memorie cache e partizioni di memoria, si ottiene un bilanciamento corretto per differenti obiettivi in termini di costo e prestazioni per i diversi segmenti di mercato delle GPU. Il dispositivo di interconnessione scalabile *Scalable Link Interconnect* (SLI) permette di connettere GPU multiple fornendo ulteriore scalabilità.

Le prestazioni

La GeForce 8800 Ultra temporizza i processori SP dei thread e le SFU con un clock a 1,5 GHz per una prestazione di picco teorica di 576 GFLOPS. La GeForce 8800 GTX utilizza un clock del processore di 1,35 GHz e ha una prestazione di picco di 518 GFLOPS.

I tre paragrafi seguenti confrontano le prestazioni di una GPU GeForce 8800 con una CPU a core multipli per tre differenti applicazioni: algebra lineare densa, trasformate di Fourier veloci (FFT) e ordinamento. I programmi e le librerie per la GPU sono costituiti da codice C compilato con CUDA. Il codice per CPU utilizza la libreria multithread MKL 10,0 a singola precisione di Intel per trarre vantaggio dalle istruzioni SSE e dai core multipli.

Le prestazioni sull'algebra lineare densa

I calcoli di algebra lineare densa sono fondamentali in molte applicazioni. Volkov e Demmel (2008) hanno confrontato le prestazioni ottenute con una GPU e una CPU nella moltiplicazione di matrici dense in singola precisione (procedura SGEMM) e di fattorizzazione matriciale, LU, QR e di Cholesky. La Figura C.7.3 confronta la misura dei GFLOPS per la moltiplicazione matrice-matrice di matrici dense utilizzando la procedura SGEMM di una GPU GeForce 8800 GTX con quella di una CPU quad-core. La Figura C.7.4 confronta i valori di GFLOPS di una GPU con quelli di una CPU quad-core nella fattorizzazione di matrici.

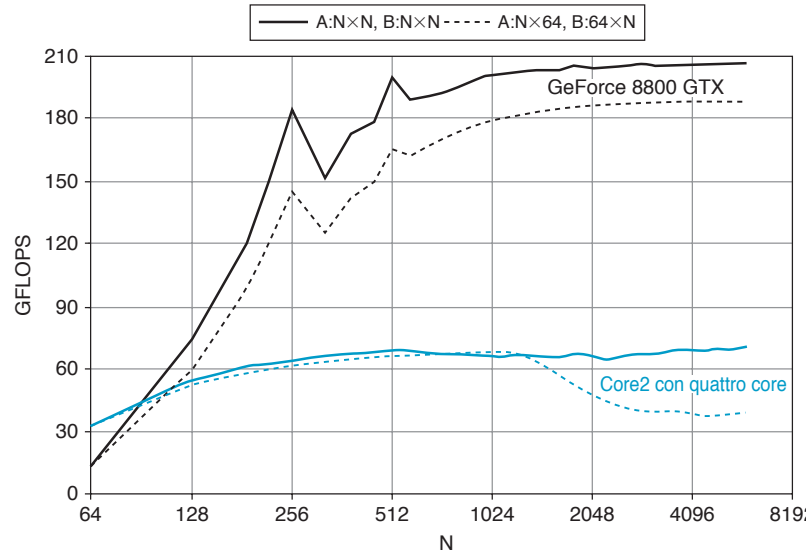


Figura C.7.3. Prestazioni del prodotto SGEMM tra due matrici dense. Il grafico mostra il numero di GFLOPS ottenuti nel prodotto in singola precisione di matrici quadrate N × N (linee continue) e di matrici sottili di N × 64 e 64 ×nella memoria della CPU.

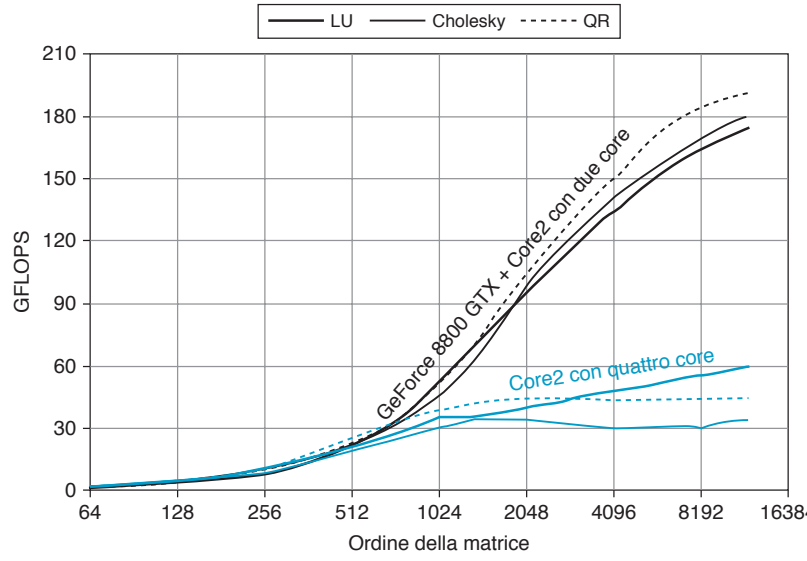


Figura C.7.4. Prestazioni della fattorizzazione di matrici dense. Il grafico mostra il numero di GFLOP ottenuti nelle fattorizzazioni di una matrice integrando la GPU e utilizzando la CPU da sola. I dati sono stati adattati dalla figura 7 di Volkov e Demmel (2008). Le linee nere corrispondono a una GeForce 8800 GTX che esegue il codice compilato mediante CUDA 1.1 su Windows XP, inserita in un host Core2 Intel con due processori core, Duo E6700, a 2,67 GHz. I GFLOPS tengono conto anche di tutti i tempi di trasferimento dati tra CPU e GPU. Le linee blu corrispondono a un Intel Core2 con quattro processori core, Core2 Quad Q6600, a 2,4 GHz, con sistema operativo Linux a 64 bit e la libreria MKL 10,0 di Intel.

Poiché il prodotto matrice-matrice SGEMM e le funzioni simili della libreria BLAS3 costituiscono il nucleo delle operazioni richieste per la fattorizzazione delle matrici, le loro prestazioni marcano un limite superiore della velocità di fattorizzazione. Quando l'ordine della matrice aumenta oltre un fattore compreso tra 200 e 400, il problema della fattorizzazione diventa sufficientemente grande da far sì che la funzione SGEMM sfrutti appieno il parallelismo della GPU e superi i sovraccarichi di gestione e di trasferimento GPU-CPU. La moltiplicazione matrice-matrice SGEMM di Volkov raggiunge i 206 GFLOPS, circa il 60% della velocità di picco della GeForce 8800 GTX per le operazioni di moltiplicazione e somma, mentre la fattorizzazione QR raggiunge i 192 GFLOPS, circa 4,3 volte la prestazione della CPU a core quadruplo.

Le prestazioni sulle FFT

Le trasformate veloci di Fourier (FTT) sono utilizzate in molte applicazioni. Le trasformate di grandi dimensioni e quelle multidimensionali vengono partizionate in lotti di trasformate monodimensionali più piccole.

In Figura C.7.5 sono confrontate le prestazioni su una FFT 1-D complessa a singola precisione di una GeForce 8800 GTX a 1,35 GHz (fine 2006) con quelle di uno Xeon Intel con quattro core della serie E5462 (nome in codice «Harpertown», fine 2007). Le prestazioni della CPU sono state misurate utilizzando la FFT della libreria *Math Kernel Library* (MKL) 10,0 di Intel con quattro thread. Le prestazioni della GPU sono state misurate utilizzando la libreria CUFFT 2.1 di NVIDIA effettuando gruppi di FFT monodimensionali *radix*-16, a decimazione di frequenza. Il throughput della GPU e della CPU è stato valutato effettuando le FFT su gruppi di dati, ciascuno di dimensioni pari a $2^{24}/n$, dove n è la dimensione della trasformata. Ne consegue che il carico di lavoro era pari a 128 MB per ciascuna trasformata. Per determinare il numero di GFLOPS, si è assunto un numero di operazioni pari a $5n \log_2 n$.

Le prestazioni sull'ordinamento

A differenza delle applicazioni appena descritte, l'ordinamento richiede un coordinamento molto maggiore tra i thread paralleli, di conseguenza è più difficile ottenere che le prestazioni scalino con il grado di parallelismo. Ciononostante, una varietà di algoritmi di ordinamento ben noti può essere resa parallela in modo efficiente per essere eseguita sulla GPU. Satish et al. (2008) descrivono in dettaglio la progettazione di algoritmi di ordinamento in CUDA, e i risultati da loro ottenuti per l'algoritmo Radix Sort sono riassunti in questo paragrafo.

In Figura C.7.6 sono messe a confronto le prestazioni di ordinamento parallelo su una GeForce 8800 Ultra con quelle ottenute su un sistema a otto core Clovertown di Intel, entrambi del 2007. I core della CPU sono distribuiti fisicamente su due connettori, dove a ciascun connettore è collegato un modulo a chip multiplo contenente due core gemelli Core2; ogni chip è dotato di 4 MB di cache L2. Tutte le procedure di ordinamento sono state progettate per ordinare coppie chiave-valore in cui sia le chiavi di ordinamenti sia i valori sono numeri interi a 32 bit. L'algoritmo principale qui considerato è Radix Sort, anche se i risultati vengono confrontati anche con la procedura basata su Quick Sort, `parallel_sort()`, fornita da Intel nei *Threading Building Blocks*. Delle due procedure Radix Sort scritte per la CPU, una è stata implementata utilizzando soltanto l'insieme delle istruzioni scalari e l'altra utilizzando procedure in linguaggio assembler messe a punto a mano, le quali sfruttano le istruzioni vettoriali SIMD dell'SSE2.

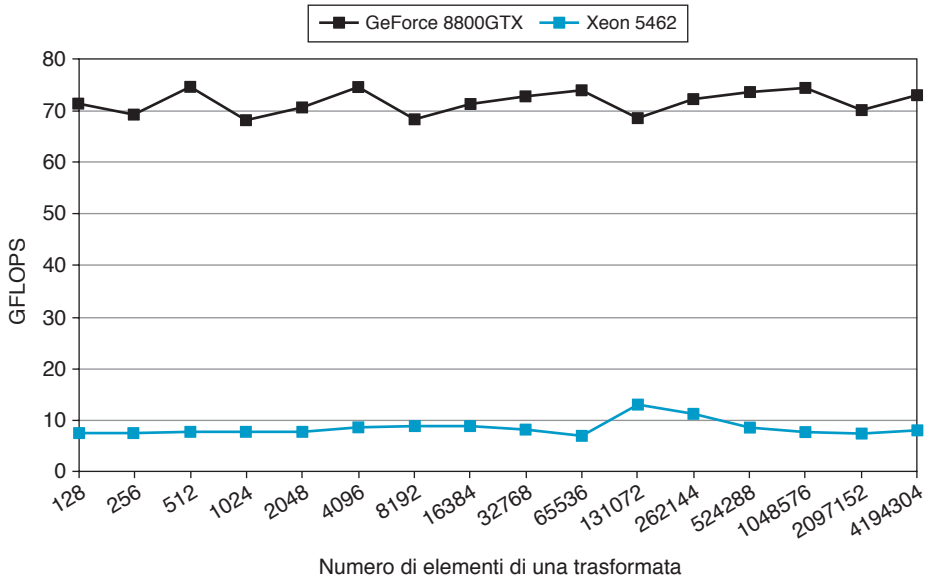


Figura C.7.5. Prestazioni del throughput della trasformata veloce di Fourier. Il grafico mette a confronto le prestazioni di FFT monodimensionali complesse «sul posto», cioè calcolate nella stessa memoria contenente i dati, su una GeForce 8800 GTX a 1,35 GHz e su uno Xeon Intel con quattro core, della serie E5462 (nome in codice «Harpertown») con 6 MB di cache L2, 4 GB di memoria, FSB 1600, sistema operativo Red Hat Linux e libreria MKL 10,0 di Intel.

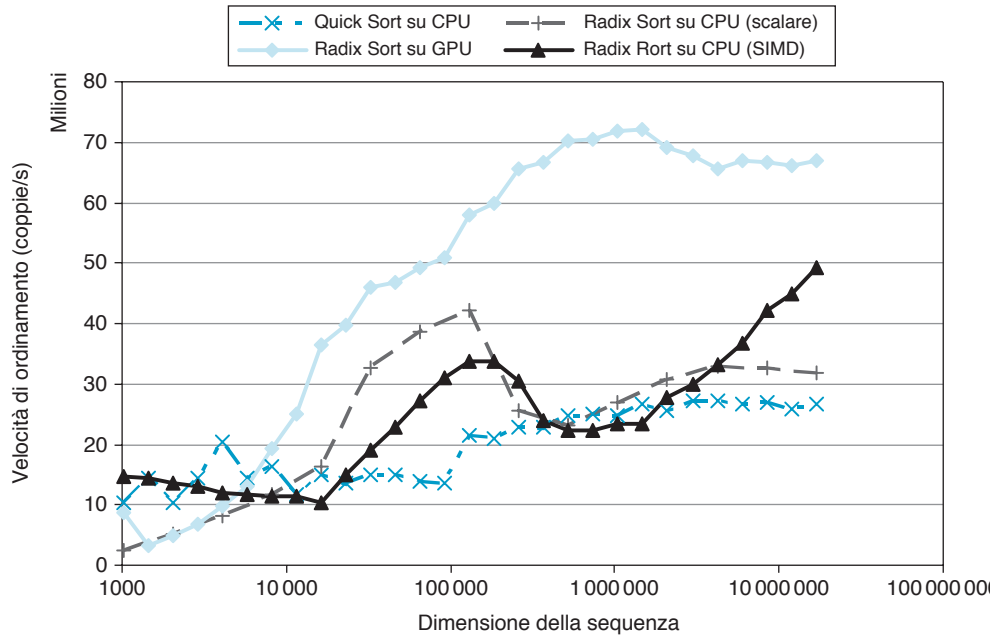


Figura C.7.6. Prestazioni dell'ordinamento parallelo. Questo grafico mette a confronto la velocità di ordinamento dell'implementazione parallela di Radix Sort su una GeForce 8800 Ultra con quella dell'implementazione su un sistema a 8 core Core2 Xeon E5345 a 2,33 GHz di Intel.

Il grafico in Figura C.7.6 mostra la velocità di ordinamento ottenuta, definita come numero di elementi ordinati diviso per il tempo necessario a ordinarli, in funzione delle dimensioni delle sequenze. Si evince chiaramente che il Radix Sort su GPU ottiene la più alta velocità di ordinamento per tutte le sequenze con più di 8000 elementi. In questo intervallo l'algoritmo è in media 2,6 volte più veloce della procedura basata su Quick Sort e circa 2 volte più veloce delle procedure Radix Sort, le quali impiegano tutti gli otto core disponibili. Le prestazioni del Radix Sort su CPU varia ampiamente, probabilmente a causa della scarsa località di cache nelle sue permutazioni globali.

C.8 Un caso reale: come adattare le applicazioni alla GPU

L'avvento delle CPU multicore e delle GPU a moltissimi core ha fatto sì che i chip dei processori principali diventassero sistemi paralleli. Inoltre, il loro grado di parallelismo continua a crescere seguendo la legge di Moore. La sfida attuale consiste nello sviluppo di applicazioni di elaborazione visuale e di calcolo ad alte prestazioni che siano in grado di scalare il loro parallelismo in modo trasparente per trarre vantaggio dal crescente numero di core di elaborazione, come le applicazioni di grafica 3D riescono a scalare il loro parallelismo su GPU con un numero di core altamente variabile.

Questo paragrafo presenta alcuni esempi di come sia possibile adattare un'applicazione scalabile di calcolo parallelo su una GPU utilizzando CUDA.

Matrici sparse

In CUDA è possibile scrivere svariati algoritmi paralleli in modo molto semplice, anche nel caso in cui le strutture dati coinvolte non siano semplici griglie regolari. Il prodotto matrice-vettore con matrici sparse (SpVM) costituisce un buon esempio di calcolo numerico che può essere parallelizzato abbastanza

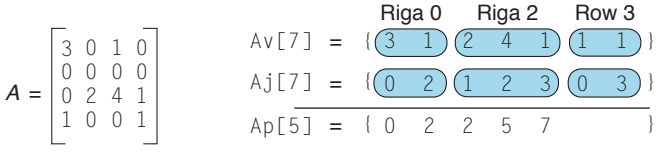
facilmente utilizzando le astrazioni fornite da CUDA. I kernel qui descritti, se combinati con le procedure sui vettori rese disponibili in CUBLAS, permettono di scrivere semplici algoritmi di ottimizzazione iterativi, come il metodo del gradiente coniugato.

Una matrice $n \times n$ sparsa è una matrice nella quale il numero m di elementi non nulli è solo una piccola parte del numero totale di elementi. Le rappresentazioni delle matrici sparse tendono a memorizzare soltanto gli elementi non nulli della matrice. Poiché una matrice sparsa $n \times n$ tipicamente contiene solamente $m = O(n)$ elementi non nulli, questa scelta porta a un consistente risparmio di spazio in memoria e di tempo di elaborazione.

Una delle più comuni rappresentazioni delle metrici sparse generiche non strutturate è la rappresentazione a righe sparse compresse (CSR, *Compressed Sparse Row*). Gli m elementi non nulli della matrice `A` vengono memorizzati per righe in un vettore `Av`. Un secondo vettore `Aj` contiene l'indice di colonna corrispondente a ogni elemento di `Av`. Infine, un vettore `Ap` di $n + 1$ elementi memorizza la lunghezza di ogni riga all'interno dei vettori precedenti: gli elementi della riga i-esima della matrice `A` saranno così individuati dagli indici memorizzati in `Aj` e `Av` tra l'indice contenuto in `Ap[i]` e l'indice immediatamente precedente al valore contenuto in `Ap[i+1]`; ciò significa che `Ap[0]` conterrà sempre 0 e `Ap[n]` sarà sempre pari al numero di elementi non nulli. La Figura C.8.1 mostra un esempio di rappresentazione CSR di una semplice matrice.

Data una matrice `A` in forma CSR e un vettore `x`, possiamo calcolare una singola riga del prodotto `y = Ax` utilizzando la procedura `moltiplica_riga()` riportata in Figura C.8.2. Per calcolare l'intero prodotto è quindi sufficiente eseguire un ciclo su tutte le righe e calcolare il risultato per ciascuna riga utilizzando `moltiplica_riga()`, come avviene nel codice C sequenziale riportato in Figura C.8.3.

Questo algoritmo può essere tradotto molto facilmente in un kernel CUDA parallelo: è sufficiente distribuire il ciclo contenuto in `moltcrs_sequenz()` su più thread paralleli, dove ciascun thread calcolerà esattamente una riga del vettore risultato `y`. Il codice di tale kernel è riportato in Figura C.8.4. Si noti



a. Matrice di esempio *A* b. Rappresentazione CSR della matrice

Figura C.8.1. Matrice a righe sparse compressa (CSR).

```
float moltiplica_riga(unsigned int dimriga,
                      unsigned int *Aj,     // indici di colonna della riga
                      float *Av,            // elementi non nulli della riga
                      float *x)             // il vettore da moltiplicare
{
   float somma = 0;

   for(unsigned int colonna=0; colonna<dimriga; colonna++)
       somma += Av[colonna] * x[Aj[colonna]];

   return somma;
}
```

Figura C.8.2. Il codice C sequenziale per una singola riga di prodotto matrice-vettore per matrici sparse.

Figura C.8.3. Il codice sequenziale per il prodotto matrice-vettore sparso.

```
void moltcsr_sequenz(unsigned int *Ap, unsigned int *Aj,
                             float *Av, unsigned int num_righe,
                             float *x, float *y)
{
   for(unsigned int riga=0; row<num_righe; ++riga)
   {
       unsigned int riga_inizio = Ap[riga];
       unsigned int riga_fine  = Ap[riga+1];

       y[riga] = moltiplica_riga(riga_fine, riga_inizio,
       Aj+riga_inizio, Av+riga_inizio, x);
   }
}
```

Figura C.8.4. La versione CUDA del prodotto matrice-vettore sparso.

```
__global__
void moltcsr_kernel(unsigned int *Ap, unsigned int *Aj,
                             float *Av, unsigned int num_righe,
                             float *x, float *y)
{
   unsigned int righe = blockIdx.x*blockDim.x + threadIdx.x;

   if(riga<num_righe)
   {
       unsigned int riga_inizio = Ap[riga];
       unsigned int riga_fine  = Ap[riga+1];

       y[row] = moltiplica_riga(riga_fine, riga_inizia,
       Aj+riga_inizia, Av+riga_inizia, x);
   }
}
```

che è molto simile al ciclo sequenziale utilizzato nella procedura `moltcrs_sequenz()`, e in effetti ci sono soltanto due differenze. Anzitutto, l'indice di riga, `riga`, per ogni thread viene calcolato a partire dall'indice di thread e di blocco assegnati a ciascun thread, eliminando così il ciclo *for*. Secondo, è presente un'istruzione di test che fa sì che il prodotto associato a una riga venga calcolato soltanto se l'indice di riga si trova all'interno dei limiti della matrice; ciò si rende necessario perché il numero di righe, n, può non essere un multiplo della dimensione del blocco definita quando il kernel viene lanciato in esecuzione.

Assumendo che le strutture dati delle matrici siano già state copiate nella memoria della GPU, i kernel possono essere lanciati in esecuzione in questo modo:

```
unsigned int dimblocco = 128;
 // o qualunque altra dimensione fino a 512
unsigned int numblocchi = (num_righe + dimblocco – 1)/
                           dimblocco;
moltcsr_kernel<<<numblocchi, dimblocco>>>
 (Ap, Aj, Av, num_righe, x, y);
```

La struttura di codice qui descritta è molto diffusa. L'algoritmo sequenziale originario è organizzato in un ciclo le cui iterazioni sono indipendenti fra

loro; i cicli di questo tipo possono essere parallelizzati molto facilmente, assegnando una o più iterazioni del ciclo a ogni thread parallelo. Il modello di programmazione fornito da CUDA rende particolarmente semplice esprimere questo tipo di parallelismo.

La strategia generale di scomporre i calcoli in blocchi di elaborazione indipendenti oppure, in alcuni casi, di spezzettare iterazioni di ciclo indipendenti non è caratteristica solamente di CUDA, ma si tratta di un approccio comune utilizzato nelle varie forme da diversi sistemi di programmazione parallela, quali OpenMP e Threading Building Blocks di Intel.

Utilizzo della memoria condivisa come cache

Gli algoritmi di prodotto matrice-vettore sparsi descritti nel paragrafo precedente sono abbastanza semplificati: ci sono molte ottimizzazioni che si possono adottare sia nel codice per la CPU sia in quello per la GPU per produrre un incremento delle prestazioni, come lo srotolamento dei cicli, il riordinamento delle matrici e l'impiego dei blocchi di registri. I kernel paralleli possono anche essere implementati da capo in termini di operazioni di scansione parallele sui dati, come mostrato da Sengupta et al. (2007).

Una delle caratteristiche importanti dell'architettura CUDA è la presenza della memoria condivisa per ciascun blocco, una piccola memoria sul chip caratterizzata da una bassissima latenza. Lo sfruttamento di questa memoria può produrre notevoli miglioramenti delle prestazioni; per esempio, si può utilizzare questa memoria condivisa come una cache gestita dal software per contenere i dati riutilizzati frequentemente. Le modifiche del codice precedente per utilizzare la memoria condivisa sono riportate in Figura C.8.5.

Nell'ambito del prodotto di matrici sparse, si può osservare che parecchie righe di A potrebbero utilizzare lo stesso elemento `x[i]`. In molti casi comuni, e in particolare quando la matrice è stata riordinata, le righe che utilizzano `x[i]` risulteranno vicine alla riga i. Possiamo quindi implementare un semplice meccanismo di utilizzo di una cache e aspettarci di ottenere un incremento delle prestazioni. Il blocco di thread che elabora le righe da i a j caricherà gli elementi compresi tra `x[i]` e `x[j]` nella memoria condivisa. Srotoleremo quindi il ciclo `moltiplica_riga()` e preleveremo gli elementi di `x` dalla cache, quando possibile. Il codice risultante è riportato in Figura C.8.5. La memoria condivisa può essere anche utilizzata per fare altre ottimizzazioni, come prelevare `Ap[riga+1]` da un thread adiacente, anziché prelevarlo di nuovo dalla memoria.

Poiché l'architettura Tesla fornisce una memoria condivisa su chip gestita in modo esplicito (anziché una cache hardware gestita in modo implicito), questa ottimizzazione viene di solito applicata. Sebbene questo possa imporre al programmatore del lavoro aggiuntivo durante lo sviluppo del codice, si tratta di un lavoro modesto se comparato al potenziale beneficio in termini di prestazioni. Nell'esempio descritto in precedenza questo semplice utilizzo della memoria condivisa fornisce già un incremento di prestazioni attorno al 20% su matrici di dimensioni significative, derivanti da *mesh* di superfici 3D. La disponibilità di una memoria gestita esplicitamente invece di una cache implicita presenta anche il vantaggio che le politiche di utilizzo della cache e di precaricamento dei dati possono essere adattate alle esigenze dell'applicazione.

Quelli appena mostrati sono kernel abbastanza semplici, il cui scopo principale è di illustrare le tecniche di base della programmazione CUDA e non di mostrare come ottenere le prestazioni massime. Esistono svariati modi per ottimizzare il codice, molti dei quali sono stati analizzati da Williams et al. (2007) utilizzando diverse architetture multicore. Ciononostante, è comun-

```
__global__
void moltcsr_cache(unsigned int *Ap, unsigned int *Aj,
                           float *Av, unsigned int num_righe,
                           float *x, float *y)
{
   // Poni in cache le righe di x[] corrispondenti a questo blocco.
   __shared__ float cache[dimblocco];

   unsigned int blocco_inizio = blockIdx.x * blockDim.x;
   unsigned int blocco_fine  = blocco_inizio + blockDim.x;
   unsigned int riga     = blocco_inizio + threadIdx.x;

   // Preleva e poni in cache la nostra finestra di x[].
   if( riga<num_righe )  cache[threadIdx.x] = x[riga];
   __syncthreads();

   if( riga<num_righe )
   {
       unsigned int riga_inizio = Ap[riga];
       unsigned int riga_fine  = Ap[riga+1];
       float somma = 0, x_j;

       for(unsigned int col=riga_inizio; col<riga_fine; ++col)
       {
           unsigned int j = Aj[col];

           // Preleva x_j dalla nostra cache, quando possibile
           if( j>=blocco_inizio && j<blocco_fine )
             x_j = cache[j-blocco_inizio];
           else
             x_j = x[j];

           somma += Av[col] * x_j;
       }

       y[riga] = somma;
   }
}
```

Figura C.8.5. **La versione del prodotto vettore-matrice sparsa che utilizza la memoria condivisa.**

que istruttivo esaminare in modo comparato le prestazioni anche per questi semplici kernel. Su un processore Core2 Xeon E5335 a 2 GHz di Intel, il kernel `moltcsr_sequenz()` elabora circa 202 milioni di elementi non nulli al secondo, per un insieme di matrici laplaciane derivate da superfici 3D a mesh triangolari. La parallelizzazione di questo kernel mediante il costrutto `parallel_for` fornito nei *Threading Building Blocks* di Intel produce un'accelerazione di un fattore 2,0, 2,1 e 2,3 quando il codice viene eseguito rispettivamente su una macchina con due, quattro o otto core. Su una GeForce 8800 Ultra, i kernel `moltcsr_kernel()` e `moltcsr_cache()` raggiungono rispettivamente una velocità di elaborazione di circa 772 e 920 milioni di elementi non nulli al secondo, che corrispondono a un aumento di velocità pari a 3,8 e 4,6 volte rispetto al codice sequenziale eseguito sulla CPU senza accelerazioni.

Scansione e riduzione

La *scansione* (*scan*) parallela, detta anche *somma a prefisso* (*prefix sum*) parallela, è uno dei componenti più importanti degli algoritmi che lavorano sui dati in parallelo (Blelloch, 1990). Data una sequenza a di n elementi

$$[a_0, a_1, \ldots, a_{n-1}]$$

e un operatore associativo binario $\oplus$, la funzione *scan* calcola la sequenza:

$$\texttt{scan}(a, \oplus) = [a_0, (a_0 \oplus a_1), \ \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$$

Per esempio, se assumiamo che $\oplus$ sia il normale operatore somma, l'applicazione della scansione al vettore di ingresso

$$a = [3\ 1\ 7\ 0\ 4\ 1\ 6\ 3]$$

produrrà la sequenza di somme parziali:

$$\text{scan}(a, \oplus) = [3\ 4\ 11\ 11\ 15\ 16\ 22\ 25]$$

Questo operatore di scansione corrisponde a una scansione *inclusiva*, nel senso che l'i-esimo elemento della sequenza prodotta comprende l'elemento a_i in ingresso. L'inclusione solo degli elementi precedenti corrisponde all'operatore di scansione *esclusiva*, conosciuto anche come operatore di *somma a prefisso*.

L'implementazione sequenziale di tale operazione è molto semplice. Si tratta di un ciclo che itera una volta sull'intera sequenza, come mostrato in Figura C.8.6.

```
template<class T>
__host__ T plus_scan(T *x, unsigned int n)
{
   for(unsigned int i=1; i<n; ++i)
       x[i] = x[i-1] + x[i];
}
```

Figura C.8.6. Modello della funzione plus_scan seriale.

A prima vista potrebbe sembrare che questa operazione sia per natura sequenziale. È possibile, invece, implementarla in parallelo in modo efficiente. Il punto cruciale è che, essendo l'addizione associativa, si può cambiare l'ordine con cui gli elementi vengono sommati tra loro. Per esempio, possiamo immaginare di sommare coppie di elementi consecutivi in parallelo per poi sommare le somme parziali risultanti, e così via.

Uno schema semplice per ottenere ciò è stato proposto da Hillis e Steele (1989) e un'implementazione del loro algoritmo in CUDA è riportato in Figura C.8.7.

```
template<class T>
__host__ T plus_scan(T *x)
{
   unsigned int i = threadIdx.x;
   unsigned int n = blockDim.x;

   for(unsigned int offset=1; offset<n; offset *= 2)
   {
       T t;

       if( i>=offset )      t = x[i-offset];
       __syncthreads();

       if( i>=offset )      x[i] = t + x[i];
       __syncthreads();
   }
   return x[i];
}
```

Figura C.8.7. Modello CUDA della funzione plus_scan parallela.

Si assume che il vettore d'ingresso, `x[]`, contenga esattamente un elemento per ciascun thread di un blocco di thread e che vengano effettuate $\log_2 n$ iterazioni del ciclo raccogliendo le somme parziali.

Per comprendere il funzionamento di questo ciclo si consideri la Figura C.8.8, che illustra il semplice caso con $n = 8$ thread ed elementi. Ogni livello del diagramma rappresenta un passo del ciclo. Le linee indicano la locazione dalla quale il dato viene prelevato. Per ogni elemento dell'uscita, corrispondente all'ultima riga del diagramma, si costruisce un albero delle somme degli elementi d'ingresso. Le linee evidenziate in blu mostrano la struttura dell'albero delle somme per l'elemento finale. Le foglie sono tutti gli elementi iniziali. Procedendo a ritroso, a partire da qualsiasi elemento dell'uscita, *i*, si vede che l'albero incorpora tutti i valori di ingresso fino all'*i*-esimo elemento incluso.

Benché semplice, questo algoritmo non è così efficiente come vorremmo.

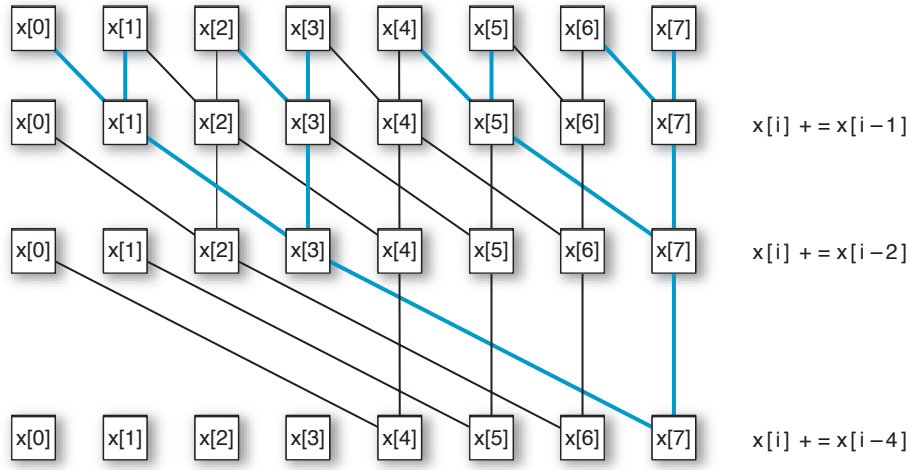


Figura C.8.8. Indirizzamento dei dati nella scansione parallela ad albero.

Esaminando l'implementazione sequenziale, notiamo che essa richiede *O(n)* addizioni. L'implementazione parallela, invece, effettua *O(n* log *n)* addizioni. Non è quindi efficiente, poiché svolge più lavoro rispetto alla versione sequenziale per calcolare lo stesso risultato. Fortunatamente esistono altre tecniche, più efficienti, per implementare la scansione. Ulteriori dettagli su tali tecniche, nonché l'estensione di questa procedura basata su un blocco a vettori multiblocco, si possono trovare in Sengupta et al. (2007).

In alcuni casi, potremmo essere interessati soltanto a calcolare la somma di tutti gli elementi di un array, invece che la sequenza di tutte le somme a prefisso fornite da `scan`. In questo caso si parla di *riduzione parallela*. Potremmo utilizzare un algoritmo di scansione per effettuare questo calcolo, ma la riduzione può essere in genere implementata più efficientemente della scansione.

La Figura C.8.9 riporta il codice per il calcolo di una riduzione utilizzando la somma. In questo esempio, ogni thread carica semplicemente un elemento della sequenza di ingresso (cioè somma inizialmente una sottosequenza di lunghezza 1). Alla fine della riduzione, vogliamo che il thread 0 contenga la somma di tutti gli elementi caricati inizialmente dai thread del suo blocco. Il ciclo in questo kernel costruisce implicitamente un albero di somme sugli elementi di ingresso, in modo molto simile al precedente algoritmo di scansione.

```
__global__
void plus_reduce(int *input, unsigned int N, int *totale)
{
   unsigned int tid = threadIdx.x;
   unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

   // Ogni blocco carica i suoi elementi in memoria condivisa,
   // riempiendo di 0 se N non è un multiplo di dimblocco
   __shared__ int x[dimblocco];
   x[tid] = (i<N) ? input[i] : 0;
   __syncthreads();

   // Ora ogni thread contiene un valore di ingresso in x[]
   //
   // Costruisci l'albero di somma sugli elementi.
   for(int s=blockdDim.x/2; s>0; s=s/2)
   {
      if(tid < s)    x[tid] += x[tid + s];
      __syncthreads();
   }

   // Il thread 0 contiene ora la somma di tutti i valori in ingresso
   // di questo blocco. Aggiungi ora tale somma al totale corrente.
   if( tid==0 )     atomicAdd(totale, x[tid]);
}
```

Figura C.8.9. Implementazione CUDA della riduzione «plus» (plus_reduce).

Al termine di questo ciclo, il thread 0 conterrà la somma di tutti i valori caricati da questo blocco. Se si desidera che il conenuto finale della locazione puntata da `total` sia il totale di tutti gli elementi del vettore, è necessario combinare le somme parziali di tutti i blocchi della griglia. Una strategia per ottenere questo risultato sarebbe quella di fare in modo che ogni blocco scriva la propria somma parziale in un secondo vettore e quindi lanci di nuovo il kernel di riduzione, ripetendo il processo fino a ridurre la sequenza a un unico valore. Un'alternativa più interessante supportata dall'architettura di GPU Tesla è quella di utilizzare la primitiva `atomicAdd()`, un'efficiente primitiva di lettura-modifica-scrittura atomica supportata dal sottosistema di memoria. L'impiego di questa funzione elimina la necessità di costruire vettori temporanei addizionali e di lanciare ripetutamente il kernel.

La riduzione parallela è una primitiva essenziale per la programmazione parallela ed evidenzia l'importanza della memoria condivisa di blocco e delle barriere di basso costo, con lo scopo di rendere efficiente la cooperazione tra i thread. Questo grado di rimescolamento dei dati fra i thread, infatti, avrebbe un costo proibitivo in termini di velocità se effettuato nella memoria globale fuori dal chip.

Radix Sort

Un'applicazione importante delle primitive di scansione è l'implementazione di procedure di ordinamento. Il codice in figura C.8.10 implementa un ordinamento Radix Sort di interi su un singolo blocco di thread. Esso riceve in ingresso un vettore di valori `valori` contenente un intero a 32 bit per ogni thread del blocco. Per questioni di efficienza, questo vettore dovrebbe essere memorizzato nella memoria condivisa di blocco, pur non trattandosi di un requisito per avere un ordinamento corretto.

Figura C.8.10. Il codice CUDA per l'ordinamento mediante algoritmo Radix Sort.

```
__device__ void radix_sort(unsigned int *valori)
{
   for(int bit=0; bit<32; ++bit)
   {
      partizione_per_bit(valori, bit);
      __syncthreads();
   }
}
```

Quella presentata è un'implementazione abbastanza semplice di Radix Sort. Si assume di avere a disposizione una procedura `partizione_per_bit()` che partiziona il vettore dato, in modo che tutti i valori con uno 0 in corrispondenza del bit designato vengano posti prima di tutti i valori che presentano un 1 in tale posizione. Per produrre un risultato corretto, tale partizionamento deve essere robusto.

L'implementazione della procedura di partizionamento è una semplice applicazione della scansione. Il thread i possiede il valore x_i e deve calcolare l'indice di uscita corretto, in corrispondenza del quale scrivere quel valore. Per fare ciò, è necessario calcolare: 1) il numero di thread $j < i$ per i quali il bit considerato è 1 e 2) il numero totale di bit per i quali il bit considerato è 0. Il codice CUDA per la funzione `partizione_per_bit()` è riportato in figura C.8.11.

Una strategia simile può essere applicata per implementare un kernel di Radix Sort che ordini vettori di grandi dimensioni, anziché soltanto il vettore di un blocco. L'aspetto cruciale rimane la procedura di scansione, sebbene, quando il calcolo è ripartito su kernel multipli, si debba impiegare un doppio buffer per i vettori dei valori piuttosto che effettuare partizioni di spazio. Ulteriori dettagli sull'ordinamento efficiente di grandi vettori con Radix Sort sono forniti da Satish, Harris e Garland (2008).

Figura C.8.11. Il codice CUDA per partizionare i dati in base al confronto bit a bit utilizzato nell'impementazione di Radix Sort.

```
__device__ void partizione_per_bit(unsigned int *valori,
                                   unsigned int bit)
{
   unsigned int i    = threadIdx.x;
   unsigned int dimens = blockDim.x;
   unsigned int x_i  = valori[i];
   unsigned int p_i  = (x_i >> bit) & 1;

   valori[i] = p_i;
   __syncthreads();

   // Calcola il numero T di bit fino a p_i incluso.
   // Salva anche il numero totale di bit F.
   unsigned int T_prima  = plus_scan(valori);
   unsigned int T_totale  = valori[dimens-1];
   unsigned int F_totale  = dimens - T_totale;
   __syncthreads();

   // Scrivi ogni x_i nella corretta posizione
   if( p_i )
      valori[T_prima - 1 + F_totale] = x_i;
   else
      valori[i - T_prima] = x_i;
}
```

Applicazioni del problema a *N* corpi su GPU[1]

Nyland, Harris e Prins (2007) descrivono un kernel di calcolo semplice ma molto utile, con prestazioni su GPU eccellenti: l'algoritmo di *tutte le coppie di N corpi*, che compare in molte applicazioni scientifiche ed è caratterizzato da un notevole costo computazionale. Le simulazioni a *N* corpi calcolano l'evoluzione di un sistema di corpi nei quali ogni elemento interagisce permanentemente con tutti gli altri. Un esempio è rappresentato da una simulazione astrofisica, in cui ogni corpo rappresenta una stella e i corpi si attraggono a vicenda grazie alla forza gravitazionale. Altri esempi sono il *folding* delle proteine, dove la simulazione a *N* corpi viene utilizzata per calcolare le forze elettrostatiche e di van der Waals, la simulazione dei flussi turbolenti nei fluidi e il calcolo dell'illuminazione globale nella computer grafica.

L'algoritmo di tutte le coppie di *N* corpi calcola la forza totale su ciascun corpo del sistema valutando ogni forza tra due coppie e sommando le forze che agiscono su ogni elemento. Molti scienziati considerano questo metodo il più accurato, dove l'unica perdita di precisione è dovuta alle operazioni in virgola mobile calcolate in hardware. Lo svantaggio è la sua complessità computazionale, dell'ordine di $O(n^2)$, di gran lunga troppo elevata per sistemi costituiti da più di 10^6 corpi. Per ovviare a questo costo eccessivo, sono state proposte svariate semplificazioni per ridurre la complessità degli algoritmi a $O(n \log n)$ o $O(n)$: l'algoritmo di Barnes-Hut, il metodo veloce dei Multipoli (FMM, *Fast Multipole Method*) e la sommatoria particella-reticolo di Ewald (PME, *Particle-Mesh-Ewald*) sono degli esempi. Tutti questi algoritmi veloci, però, si basano comunque su un metodo che considera *tutte le coppie* e che costituisce il fondamento per il calcolo accurato delle forze di interazione a breve distanza; tale algoritmo continua quindi ad essere importante.

La matematica dei problemi a *N* corpi

Per le simulazioni gravitazionali, la forza d'interazione tra due corpi si calcola utilizzando la fisica elementare. Tra due corpi i e j, il vettore tridimensionale che rappresenta la forza è dato da:

$$\mathbf{f}_{ij} = G\frac{m_i m_j}{\left\|\mathbf{r}_{ij}\right\|^2} \times \frac{\mathbf{r}_{ij}}{\left\|\mathbf{r}_{ij}\right\|}$$

L'intensità della forza è data dal termine a sinistra del secondo membro, mentre la direzione è rappresentata dal termine a destra (il versore è diretto lungo la congiungente i due corpi).

Dato un insieme di corpi interagenti tra loro (un sistema intero o un sottoinsieme), il calcolo è semplice: per tutte le coppie di elementi, occorre calcolare la forza e sommarla per ogni corpo. Dopo aver calcolato le forze risultanti, queste vengono utilizzate per aggiornare la posizione e la velocità di ciascun corpo a partire dai precedenti valori di posizione e velocità. Il calcolo delle forze ha complessità $O(n^2)$, mentre l'aggiornamento ha complessità $O(n)$.

Il codice sequenziale per il calcolo delle forze richiede due cicli *for* annidati che iterano sulle coppie di corpi. Il ciclo esterno seleziona il corpo sul quale viene calcolata la forza risultante e il ciclo interno itera su tutti i corpi. Quest'ultimo chiama una funzione che calcola la forza tra due corpi e somma poi la forza ottenuta alla somma corrente.

Per calcolare le forze in parallelo si può assegnare un thread a ogni corpo, poiché il calcolo della forza su un corpo è indipendente dal calcolo della forza

[1] Tratto da Nyland, Harris e Prins [2007], «Fast N-Body Simulation with CUDA», Capitolo 31 di *GPU Gems 3*.

sugli altri corpi. Una volta che tutte le forze sono state calcolate, la posizione e la velocità dei corpi possono essere aggiornate.

Il codice della versione sequenziale e parallela è riportato nelle Figure C.8.12 e C.8.13. La versione sequenziale presenta due cicli *for* annidati. La conversione a CUDA, come in molti altri casi, converte il ciclo seriale esterno in un kernel di thread, dove ogni thread calcola la forza totale su un singolo corpo. Il kernel CUDA assegna un identificativo globale a ciascun thread, il quale sostituisce la variabile di iterazione del ciclo esterno sequenziale. Entrambe le procedure terminano memorizzando l'accelerazione totale di ciascun corpo in un vettore globale, che viene utilizzato, in un passo successivo, per calcolare il nuovo valore di posizione e velocità. Il ciclo esterno viene sostituito da una griglia di kernel CUDA che lancia in esecuzione N thread, uno per ogni corpo.

Ottimizzazione per l'esecuzione su GPU

Il codice CUDA mostrato in Figura 8.13 è corretto dal punto di vista del funzionamento ma non è efficiente, poiché non considera alcune caratteristiche fondamentali dell'architettura. Si possono ottenere prestazioni migliori implementando tre ottimizzazioni principali. Anzitutto, la memoria condivisa può essere utilizzata per evitare la lettura delle stesse celle di memoria da parte di thread diversi. Secondo, per piccoli valori di N l'utilizzo di più thread per ciascun corpo migliora le prestazioni. Infine, lo srotolamento dei cicli riduce il carico aggiuntivo per la loro gestione.

Utilizzo della memoria condivisa

La memoria condivisa può contenere un sottoinsieme di posizioni dei corpi in modo molto simile a una cache, eliminando le richieste ridondanti alla me-

Figura C.8.12. Il codice sequenziale per calcolare tutte le forze fra le coppie di *N* corpi.

```
void accel_su_tutti_corpi()
{
   int i, j;
   float3 acc(0.0f, 0.0f, 0.0f);

   for(i = 0; i < N; i++) {
       for(j = 0; j < N; i++) {
         acc = interazione_corpo_corpo(acc, corpo[i],
                                       corpo[j]);
       }
       accel[i] = acc;
   }
}
```

Figura C.8.13. Il codice di un thread CUDA per calcolare la forza totale su un singolo corpo.

```
__global__ accel_su_un_corpo()
{
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   int j;
   float3 acc(0.0f, 0.0f, 0.0f);

   for(j = 0; j < N; i++) {
       acc = interazione_corpo_corpo(acc, corpo[i], corpo[j]);
   }
   accel[i] = acc;
}
```

moria globale da parte di thread diversi. Si può quindi ottimizzare il codice precedentemente riportato in modo tale che ciascuno dei *p* thread di un blocco carichi *una* posizione in memoria condivisa, per un totale di *p* posizioni. Una volta che tutti i thread hanno caricato una posizione nella memoria condivisa, il che è assicurato da una chiamata a `__syncthreads()`, ogni thread può effettuare *p* interazioni utilizzando i dati presenti nella memoria condivisa. Questo viene ripetuto *N/p* volte per completare il calcolo della forza su ogni corpo, per cui il numero delle richieste alla memoria viene ridotto di un fattore *p*, tipicamente compreso tra 32 e 128.

La funzione `accel_su_un_corpo()` richiede alcune modifiche per ottenere questa ottimizzazione, e il codice modificato è riportato in Figura C.8.14.

Il ciclo che prima iterava su tutti i corpi ora salta con passo pari alla dimensione del blocco, *p*. Ogni iterazione del ciclo esterno carica *p* posizioni successive nella memoria condivisa, una per ogni thread. I thread si sincronizzano, quindi ciascuno di essi calcola *p* forze. Una seconda sincronizzazione si rende necessaria per garantire che i nuovi valori non vengano caricati nella memoria condivisa prima che tutti i thread abbiano completato il calcolo delle forze con i dati attuali.

L'impiego della memoria condivisa riduce la banda di trasferimento con la memoria a meno del 10% della banda totale che la GPU è in grado di sostenere: sono necessari, infatti, meno di 5 GB/s. Con questa ottimizzazione, l'applicazione è impegnata a svolgere i calcoli, anziché a restare in attesa del caricamento dei dati dalla memoria (come succede se non si utilizza la memoria condivisa). Le prestazioni, per diversi valori di *N*, sono mostrate in Figura C.8.15.

Impiego di più thread per ciascun corpo

La Figura C.8.15 mostra come siano basse le prestazioni per problemi con *N* piccolo ($N < 4096$) su una GeForce 8800 GTX. Molte applicazioni scientifiche sono basate sul calcolo di problemi con piccoli valori di *N* (per tempi lunghi di simulazione), perciò è necessario dedicare i nostri sforzi di ottimizzazione a questi casi. Finora abbiamo supposto che le basse prestazioni fossero semplicemente dovute al fatto che, per *N* piccolo, non c'è abbastanza lavoro per

```
__shared__ float4 PosizioneCondivisa [256];
…
__global__ void accel_su_un_corpo()
{
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   int j, k;
   int p = blockDim.x;
   float3 acc(0.0f, 0.0f, 0.0f);
   float4 mioCorpo = corpo[i];

   for(j = 0; j < N; i += p) {       // Il ciclo esterno salta ogni volta di p
       PosizioneCondivisa[threadIdx.x] = corpo[j+threadIdx.x];
       __syncthreads();
       for(k = 0; k < p; k++) { // Il ciclo interno accede a p posizioni
         acc=interazione_corpo_corpo(acc, mioCorpo, PosizioneCondivisa[k]);
       }
       syncthreads();
   }
   accel[i] = acc;
}
```

Figura C.8.14. Il codice CUDA per calcolare la forza totale su ciascun corpo, utilizzando la memoria condivisa per migliorare le prestazioni.

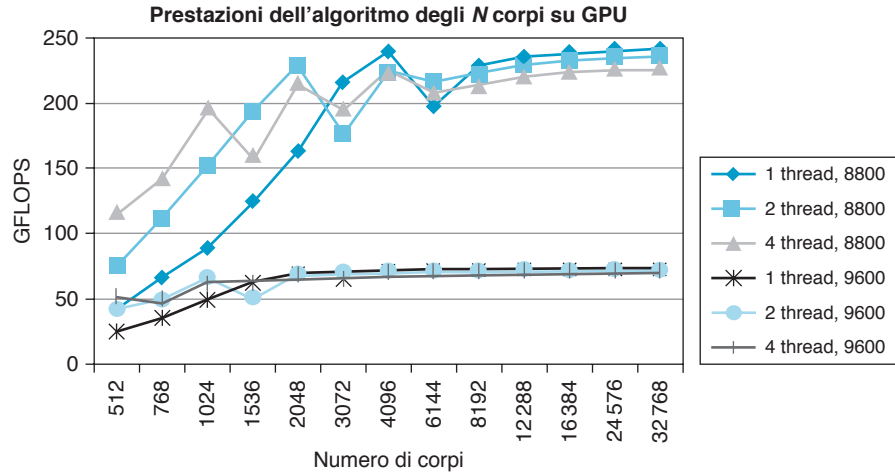


Figura C.8.15. Misura delle prestazione dell'applicazione degli *N* corpi di una GeForce 8800 GTX e di una GeForce 9600. La GeForce 8800 contiene 128 processori a flusso continuo che lavorano a 1,35 GHz, mentre la 9600 ne ha 64 che lavorano a 0,80 GHz (circa il 30% della 8800). Le prestazioni di picco sono di 242 GFLOPS. Per una GPU con più processori, le dimensioni del problema devono essere maggiori per poter ottenere le massime prestazioni: il picco per la 9600 si ha con circa 2048 corpi, mentre la 8800 non raggiunge il proprio picco fino a 16 384 corpi. Per *N* piccolo, l'impiego di più di un thread per corpo può migliorare nettamente le prestazioni, ma si può incorrere in un calo di prestazioni al crescere di *N*.

mantenere la GPU impegnata; la soluzione è quindi allocare più thread per ogni corpo. Cambiamo allora le dimensioni del blocco dei thread, da (p, 1, 1) a (p, q, 1), dove q thread si dividono in parti uguali il carico di lavoro relativo a un singolo corpo. Inserendo i thread addizionali all'interno di uno stesso blocco di thread, i risultati parziali possono essere salvati in memoria condivisa. Quando il calcolo di tutte le forze è stato completato, i q risultati parziali possono essere raccolti e sommati per ottenere il risultato finale. Come si può vedere in Figura C.8.15, l'utilizzo di due o quattro thread per ciascun corpo porta a miglioramenti notevoli, per piccoli valori di N.

Per esempio, le prestazioni della 8800 GTX aumentano del 110% per $N = 1024$: con un thread si ottengono 90 GFLOPS, mentre con quattro thread se ne ottengono 190. Le prestazioni si abbassano leggermente al crescere di N, per cui conviene utilizzare questa ottimizzazione per N minore di 4096. L'incremento delle prestazioni è mostrato in Figura C.8.15 per una GPU con 128 processori e per una GPU più piccola, con 64 processori che lavorano a due terzi della frequenza di clock della prima GPU.

Confronto delle prestazioni

Le prestazioni del codice per il problema a N corpi sono mostrate in Figura C.8.15 e C.8.16. La Figura C.8.15 mostra le prestazioni di due GPU (una di fascia media e una di fascia alta), oltre ai miglioramenti delle prestazioni ottenuti impiegando più thread per ciascun corpo. Le prestazioni della GPU più veloce vanno da 90 a poco meno di 250 GFLOPS.

La Figura C.8.16 mostra le prestazioni di una procedura quasi identica (in C++ rispetto al codice precedente in CUDA) che viene eseguita su una CPU Core2 di Intel. Le prestazioni della CPU sono circa l'1% di quelle della GPU in termini di GFLOPS (sono comprese tra lo 0,2% e il 2%), e rimangono pressoché costanti rispetto alle dimensioni del problema che invece variano moltissimo.

Il grafico mostra anche i risultati ottenuti compilando per la CPU la versione CUDA del codice, con cui si ottiene un aumento delle prestazioni del 24%.

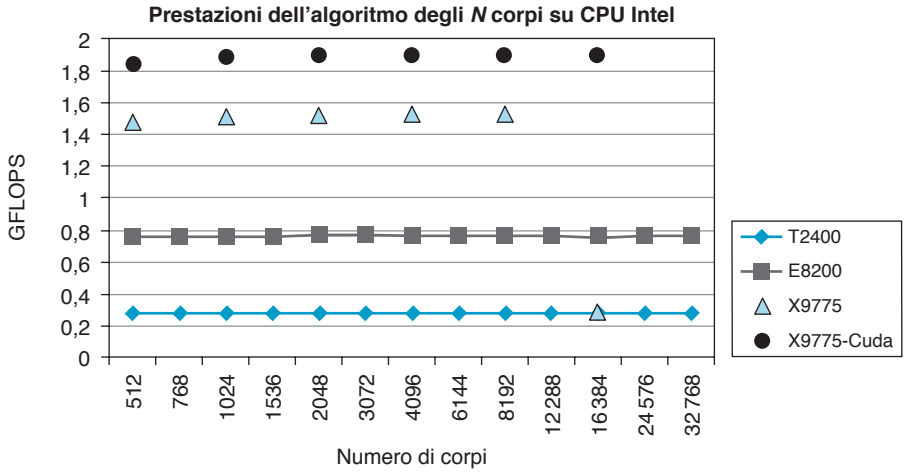


Figura C.8.16. Misura delle prestazione del codice del problema a *N* corpi su una CPU. Il grafico mostra le prestazioni del codice del problema a *N* corpi in singola precisione utilizzando alcune CPU Core2 di Intel; i diversi modelli sono indicati con un simbolo. Si noti la drastica riduzione di prestazioni in termini di GFLOP che mette in evidenza quanto la GPU sia più veloce della CPU. Le prestazioni della CPU sono in genere indipendenti dalla dimensione del problema, eccetto che in un caso anomalo in cui la CPU X9775 produce basse prestazioni per $N = 16\,384$. Il grafico, inoltre, mostra i risultati ottenuti eseguendo la versione CUDA del codice (utilizzando il compilatore *CUDA-for-CPU*) su una CPU a singolo core; questa versione supera le prestazioni del codice C++ del 24%. Come linguaggio di programmazione, CUDA rende esplicito il parallelismo e la località che possono essere sfruttati dal compilatore. Le CPU analizzate sono Intel: un Core2 Extreme X9775 (nome in codice «Penryn») a 3,2 GHz, un E8200 (nome in codice «Wolfdale») a 2,66 GHz, una CPU per desktop antecedente la Penryn, e un T2400 (nome in codice «Yonah») a 1,83 GHz, una CPU per portatili del 2007. La versione Penryn dell'architettura Core2 è particolarmente interessante per i problemi a *N* corpi grazie al suo divisore a 4 bit che permette di eseguire divisioni e radici quadrate quattro volte più velocemente delle precedenti CPU Intel.

CUDA, come linguaggio di programmazione, rende esplicito il parallelismo, permettendo così al compilatore di fare un uso migliore dell'unità vettoriale SSE sul singolo core. La versione CUDA del codice degli N corpi è mappata in modo naturale anche su CPU multicore, mediante griglie di blocchi, sulle quali le prestazioni scalano in modo quasi perfetto con il numero di core, ottenendo per $N = 4096$ un aumento delle prestazioni pari a 2,0, 3,97 e 7,94 con due, quattro e otto core rispettivamente.

Risultati

Con uno sforzo modesto, abbiamo sviluppato un kernel di calcolo che consente di ottenere prestazioni di GPU superiori fino a un fattore 157 rispetto alle prestazioni delle CPU multicore. Il tempo di esecuzione del codice del problema a N corpi eseguito su una recente CPU Intel (Penryn X9775 a 3,2 GHz, core singolo) è più di 3 secondi per immagine, ma lo stesso codice viene eseguito su una GeForce 8800 a una velocità di 44 immagini al secondo. Su una CPU pre-Penryn il codice richiedeva da 6 a 16 secondi, mentre sui meno recenti processori Core2 e Pentium IV il tempo richiesto era di 25 secondi. In realtà il guadagno nelle prestazioni va dimezzato, poiché la CPU esegue soltanto metà dei calcoli per ottenere lo stesso risultato, sfruttando il fatto che le forze su una coppia di corpi sono uguali in ampiezza e hanno verso opposto.

Come fa la GPU a rendere l'esecuzione del codice così veloce? Per rispondere bisogna esaminare l'architettura nel dettaglio. Il calcolo della forza tra una coppia di corpi richiede 20 operazioni in virgola mobile, di cui la maggior parte è costituita da somme e prodotti (alcune delle quali possono essere combinate in un'unica istruzione di moltiplicazione e somma), ma ci sono anche divisioni e radici quadrate richieste per la normalizzazione dei vettori.

Le CPU Intel impiegano molti cicli di clock per svolgere le divisioni e le radici quadrate in singola precisione[2], sebbene ci sia stato un miglioramento nella più recente famiglia di CPU (Penryn) grazie all'introduzione di un divisore veloce a 4 bit.[3] Inoltre, la capacità limitata dei registri costringe a inserire molte istruzioni MOV all'interno del codice x86, presumibilmente per trasferire dati da e verso la cache L1. Al contrario, la GeForce 8800 esegue un'istruzione di thread che calcola il reciproco della radice quadrata in quattro cicli di clock (si veda il Paragrafo C.6), possiede un insieme di registri per ogni thread più grande e una memoria condivisa dalla quale può leggere gli operandi richiesti dalle istruzioni. Infine, il compilatore CUDA produce 15 istruzioni per ciascuna iterazione del ciclo, rispetto alle oltre 40 istruzioni ottenute dai diversi compilatori per le CPU x86. Un maggiore parallelismo, l'esecuzione più veloce di istruzioni complesse, più spazio nei registri e un compilatore più efficiente sono le ragioni che, combinate insieme, spiegano il drastico aumento di prestazioni del codice per il problema a *N* corpi passando dalla CPU alla GPU.

Su una GeForce 8800, l'algoritmo di calcolo su tutte le coppie di *N* corpi fornisce prestazioni di oltre 240 GFLOPS, a fronte di meno di 2 GFLOPS sui processori sequenziali più recenti. La compilazione ed esecuzione della versione CUDA del codice su una CPU dimostra che il problema scala bene sulle CPU multicore, ma rimane decisamente più lento che su una singola GPU.

Abbiamo accoppiato la simulazione degli *N* corpi su GPU con la visualizzazione grafica del loro moto, riuscendo a visualizzare in modo interattivo 16 384 corpi interagenti tra loro alla velocità di 44 immagini al secondo. Questo permette di visualizzare ed esplorare eventi astrofisici e biofisici in modo interattivo. È inoltre possibile parametrizzare molte impostazioni, come la riduzione del rumore, il fattore di smorzamento e le tecniche di integrazione, visualizzando immediatamente il loro effetto sulla dinamica del sistema. Tutto ciò fornisce agli scienziati una formidabile capacità di visualizzazione che accresce enormemente il livello di comprensione di sistemi altrimenti invisibili, troppo grandi o troppo piccoli, troppo veloci o troppo lenti, e permette di creare modelli migliori dei fenomeni fisici.

La Figura C.8.17 presenta la visualizzazione di una sequenza temporale di una simulazione astrofisica di 16 384 corpi, dove ogni corpo rappresenta una galassia. La configurazione iniziale è un guscio sferico di corpi rotanti intorno all'asse *z;* fenomeni di interesse per gli astrofisici sono la formazione di ammassi e la fusione di galassie. Per il lettore interessato, il codice CUDA per questa applicazione è disponibile nel kit di sviluppo software (SDK) di CUDA, scaricabile da www.nvidia.com/CUDA.

C.9 Errori e trabocchetti

La rapidità con cui si sono evolute le GPU ha fatto nascere molti errori e trabocchetti. In questo paragrafo ne esaminiamo alcuni.

Errore: *le GPU sono soltanto multiprocessori vettoriali SIMD*

È facile trarre l'errata conclusione che le GPU siano semplicemente multiprocessori vettoriali SIMD. Le GPU hanno effettivamente un modello di

[2] Le istruzioni x86 SSE per il reciproco (RCP*) e il reciproco della radice quadrata (RSQRT*) non vengono considerate nel confronto, perché la loro accuratezza è troppo bassa.

[3] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Novembre 2007, Intel Corporation. Disponibile anche all'indirizzo: www3.intel.com/design/processor/manuals/248966.pdf.

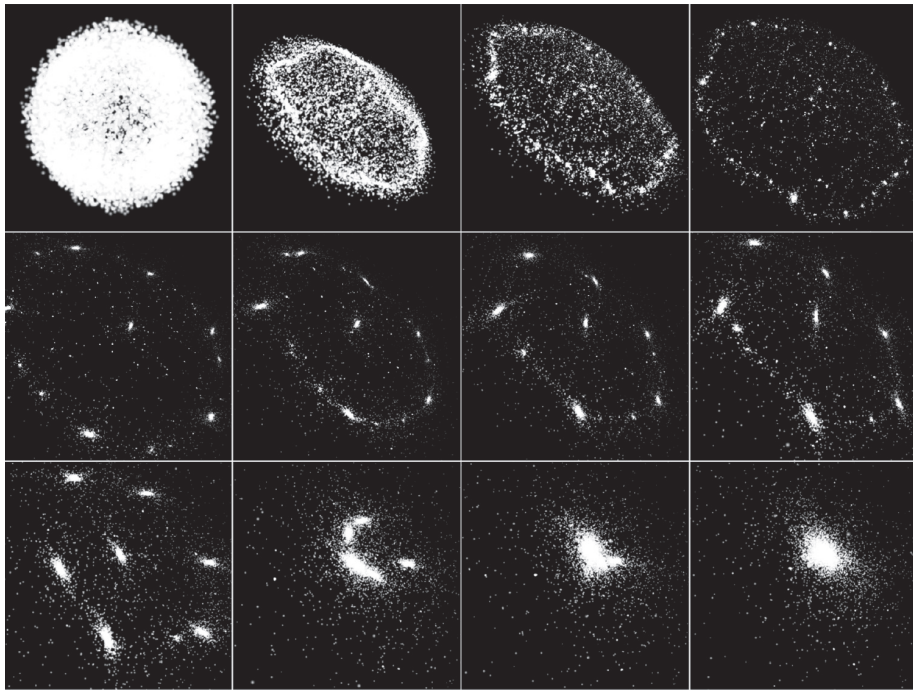


Figura C.8.17. 12 immagini catturate durante l'evoluzione di un sistema a *N* corpi costituito da 16 384 elementi.

programmazione in stile SPMD, con cui il programmatore può scrivere un singolo programma che viene eseguito da istanze di thread multiple, su dati multipli. Tuttavia, l'esecuzione di tali thread non è puramente SIMD o vettoriale, ma avviene secondo la modalità *singola istruzione, thread multiplo* (SIMT) descritta nel Paragrafo C.4. Ogni thread della GPU possiede propri registri scalari, una memoria privata, uno stato di esecuzione, un numero identificativo del thread, un percorso indipendente di esecuzione e una diramazione condizionata con un proprio program counter, e può indirizzare la memoria in modo indipendente. Anche se un gruppo di thread, come un warp di 32 thread, viene eseguito in modo più efficiente se il program counter dei diversi thread contiene lo stesso valore, questa condizione non è necessaria, per cui i multiprocessori non sono puramente SIMD. Il modello di esecuzione dei thread è MIMD con sincronizzazione a barriera e ottimizzazioni di tipo SIMT. L'esecuzione è più efficiente se gli accessi alla memoria dei singoli thread per la lettura/scrittura possono essere integrati nell'accesso a un unico blocco di dati (coalescenza), ma non è obbligatorio che questo si verifichi. In un'architettura vettoriale SIMD pura, gli accessi ai registri o alla memoria da parte dei diversi thread devono essere allineati secondo una struttura vettoriale regolare. Una GPU non impone tali restrizioni sugli accessi ai registri o alla memoria, anche se l'esecuzione risulta più efficiente se i thread di uno stesso warp accedono a blocchi locali di dati.

Un'ulteriore deviazione rispetto al modello SIMD puro consiste nel fatto che una GPU SIMT può eseguire più di un warp di thread in modo concorrente. Nelle applicazioni di grafica, ci possono essere molti gruppi di programmi di vertici, pixel e geometria che vengono eseguiti sulla schiera di multiprocessori in modo concorrente. Anche le applicazioni di calcolo possono eseguire programmi differenti in diversi warp in modo concorrente.

***Errore:** le prestazioni delle GPU non possono crescere più velocemente della legge di Moore*

La legge di Moore esprime semplicemente una velocità attesa, non definisce un limite massimo di velocità, come la velocità della luce. La legge di Moore

prevede che, in base al progresso della tecnologia dei semiconduttori e alla riduzione di dimensioni dei transistor, il costo di produzione del singolo transistor diminuisca esponenzialmente nel tempo. In altre parole, supponendo costante il costo di produzione, il numero di transistor cresce esponenzialmente. Gordon Moore (1965) previde che questo fenomeno avrebbe portato approssimativamente al raddoppio del numero di transistor ogni anno a parità di costo di produzione, e successivamente rivide la previsione riducendo la velocità di crescita a un raddoppio del numero di transistor ogni due anni. Nonostante la prima previsione fosse stata proposta da Moore nel 1965, quando il numero di componenti per circuito integrato era soltanto 50, essa si è dimostrata sorprendentemente vicina alla realtà. La riduzione delle dimensioni dei transistor ha storicamente portato anche altri benefici, come la riduzione del consumo del singolo transistor e l'aumento della velocità del clock, a parità di energia assorbita.

La crescente abbondanza di transistor serve ai progettisti dei chip per costruire processori, memoria e altri componenti. Per qualche tempo, i progettisti delle CPU hanno utilizzato i transistor in eccesso per aumentare le prestazioni dei processori a un ritmo simile a quello individuato dalla legge di Moore. In conseguenza di ciò, molti pensano che la crescita delle prestazioni dei processori (il doppio ogni 18-24 mesi) sia la legge di Moore, anche se in realtà non lo è.

I progettisti dei microprocessori utilizzano una parte dei transistor in più nei core dei processori per migliorarne l'architettura e si servono della pipeline per aumentare la frequenza di clock. Gli altri transistor in più sono impiegati per fornire una cache di dimensioni maggiori, in modo da rendere più veloce l'accesso alla memoria. I progettisti delle GPU, invece, non utilizzano quasi nessuno dei transistor in più per aumentare le dimensioni della cache: i nuovi transistor vengono utilizzati per migliorare i core dei processori e aggiungere nuovi core.

Le GPU diventano sempre più veloci grazie a quattro meccanismi. Primo, i progettisti delle GPU sfruttano abbondantemente la legge di Moore in modo diretto, utilizzando un numero sempre maggiore di transistor per costruire processori più paralleli e, quindi, più veloci. Secondo, i progettisti delle GPU sono riusciti a perfezionare l'architettura, aumentando l'efficienza di elaborazione. Terzo, la legge di Moore assume costi costanti, per cui il ritmo di crescita della legge di Moore può chiaramente essere superato se si spende di più per avere chip di dimensioni maggiori, contenenti ancora più transistor. Quarto, la banda di trasferimento effettiva dei sistemi di memoria delle GPU è cresciuta a un ritmo quasi comparabile a quello dell'aumento della velocità di elaborazione, grazie all'impiego di memorie più veloci e più ampie, alla compressione dei dati e al miglioramento delle cache. La combinazione di questi quattro aspetti ha permesso che le prestazioni delle GPU raddoppiassero regolarmente ogni 12-18 mesi. Questo ritmo di crescita, che supera la legge di Moore, è stato mantenuto per le applicazioni di grafica negli ultimi dieci anni e non dà segnali di rallentamento. Il limite alla crescita, che rende la sfida più impegnativa, sembra essere costituito dal sistema di memoria; tuttavia, l'innovazione sta facendo progressi altrettanto rapidamente in questo settore.

> ***Errore:*** *le GPU servono soltanto a gestire la grafica 3D e non a eseguire calcoli generali*

Le GPU sono costruite per generare grafica 3D, grafica 2D e video. Per soddisfare le esigenze degli sviluppatori di software grafico, espresse dalle specifiche delle interfacce e delle prestazioni/caratteristiche delle API per la grafica,

le GPU sono diventate processori in virgola mobile programmabili e massicciamente paralleli. Nel campo della grafica, questi processori vengono programmati attraverso le API grafiche, utilizzando arcani linguaggi di programmazione, quali GLSL, Cg e HLSL, costruiti con OpenGL e Direct3D. Tuttavia, nulla impedisce ai progettisti delle GPU di rendere direttamente accessibili ai programmatori i core di elaborazione paralleli senza dover passare attraverso le API grafiche o quei linguaggi di programmazione.

Di fatto, la famiglia delle GPU con architettura Tesla rende accessibili i processori attraverso un ambiente software chiamato CUDA, che permette ai programmatori di sviluppare programmi per applicazioni generiche utilizzando il linguaggio C e presto il C++. Le GPU sono elaboratori Turing completi, per cui possono eseguire qualsiasi programma che viene eseguito su una CPU, anche se forse non così bene, ma più velocemente.

Errore: *le GPU non possono eseguire velocemente i calcoli in virgola mobile a doppia precisione*

In passato, le GPU non erano in grado di eseguire programmi di calcolo in virgola mobile a doppia precisione se non passando attraverso l'emulazione software, un metodo che risultava tutt'altro che veloce. Nel corso degli anni, le GPU sono passate dalla rappresentazione aritmetica indicizzata (tabelle di corrispondenza dei colori) agli interi su 8 bit per componente di colore, all'aritmetica in virgola fissa, arrivando all'aritmetica in virgola mobile in singola precisione e, recentemente, alla doppia precisione. Le GPU moderne effettuano praticamente tutte le operazioni aritmetiche in virgola mobile in singola precisione secondo lo standard IEEE e stanno iniziando a supportare anche la doppia precisione.

A fronte di un piccolo costo addizionale, una GPU può supportare aritmetica in virgola mobile sia in doppia sia in singola precisione. Attualmente, la velocità di calcolo in doppia precisione è minore di quella in singola precisione (da cinque a dieci volte inferiore). Con un costo addizionale, le prestazioni in doppia precisione vengono incrementate, rispetto alla singola precisione, man mano che aumentano le applicazioni che la richiedono.

Errore: *le GPU non effettuano correttamente i calcoli in virgola mobile*

Le GPU, o almeno la famiglia di processori con architettura Tesla, effettuano le elaborazioni in virgola mobile in singola precisione al livello di accuratezza indicato dallo standard IEEE 754. In termini di accuratezza, quindi, le GPU sono equivalenti a ogni altro processore conforme allo standard IEEE 754.

Attualmente, le GPU non implementano alcune caratteristiche specifiche descritte nello standard, come la gestione dei numeri denormalizzati e la generazione di eccezioni precise in virgola mobile. D'altro canto, la GPU Tesla T10P, recentemente introdotta sul mercato, produce un arrotondamento completamente conforme allo standard IEEE, le operazioni moltiplicazione e somma integrate e il supporto dei numeri denormalizzati per la doppia precisione.

Trabocchetto: *basta utilizzare più thread per coprire latenze di memoria più lunghe*

I core di una CPU sono progettati per eseguire un singolo thread alla massima velocità. Perciò, ogni istruzione e i suoi dati devono essere disponibili quando arriva il momento di eseguire l'istruzione. Se l'istruzione successiva non è

pronta o se i dati richiesti per quell'istruzione non sono disponibili, l'istruzione non può essere eseguita e il processore viene messo in stallo. La memoria esterna è distante dal processore, per cui si sprecano molti cicli di esecuzione per prelevare i dati da questa memoria; di conseguenza, le CPU necessitano di cache locali capienti, con lo scopo di mantenere il flusso di esecuzione continuo e non andare in stallo. La latenza della memoria è lunga, quindi si cerca di evitare gli stalli facendo tutti gli sforzi possibili per eseguire le istruzioni con la cache. In certi casi, lo spazio di lavoro richiesto dal programma può essere più grande di qualsiasi cache. Alcune CPU hanno adottato un approccio multithreading per rendere tollerabile questa latenza, ma il numero di thread per core è in generale sempre limitato a poche unità.

La strategia adottata dalle GPU è differente. I core delle GPU sono progettati per eseguire molti thread in modo concorrente, ma soltanto un'istruzione alla volta per ciascun thread. In altre parole, una GPU esegue ogni thread lentamente ma, nel complesso, se consideriamo tutti i thread, li esegue in modo efficiente. Ogni thread può tollerare un certo livello di latenza della memoria, perché nel frattempo possono essere eseguiti gli altri thread.

L'aspetto negativo è che sono necessari moltissimi thread multipli per coprire la latenza della memoria. Inoltre, se gli accessi a memoria sono sparsi o non sono correlati tra i diversi thread, il sistema di memoria è destinato progressivamente a rallentare, perché dovrà rispondere separatamente a ogni singola richiesta; può quindi succedere che anche i thread multipli non siano in grado di coprire la latenza. Quindi, non bisogna puntare ad avere «più» thread, ma bisogna cercare di averne in numero sufficiente; i thread, inoltre, devono «comportarsi bene» in termini di località degli accessi alla memoria.

Errore: *gli algoritmi O(*n*) sono difficili da velocizzare*

Indipendentemente da quanto è veloce la GPU nell'elaborazione dei dati, le operazioni di trasferimento da e verso il dispositivo possono limitare le prestazioni degli algoritmi di complessità $O(n)$ che richiedono una piccola quantità di lavoro per ogni dato. La velocità massima di trasferimento attraverso il bus PCI-Express è di circa di 48 GB/s quando viene utilizzato il trasferimento mediante DMA, ed è leggermente inferiore per trasferimenti non DMA. La CPU, al contrario, presenta velocità tipiche di accesso alla memoria di 8-12 GB/s: alcuni algoritmi, come la somma di vettori, saranno quindi limitati dal trasferimento dei dati dalla memoria di sistema alla GPU e dei risultati dei calcoli dalla GPU alla memoria.

Ci sono tre modi per ovviare al costo del trasferimento dei dati. Il primo è cercare di lasciare i dati sulla GPU il più a lungo possibile, invece di spostare i dati avanti e indietro in corrispondenza delle diverse fasi di un algoritmo complesso: CUDA lascia deliberatamente i dati nella GPU tra il lancio di due programmi kernel.

Il secondo sfrutta il fatto che la GPU supporta la concorrenza nelle operazioni di copia in ingresso, copia in uscita e calcolo, per cui i dati possono essere fatti fluire dentro e fuori dal dispositivo mentre esso sta elaborando. Questo modello è utile per qualsiasi flusso di dati che debba essere elaborato non appena viene ricevuto, come nel caso dell'elaborazione video, dell'istradamento di pacchetti in rete, della compressione/decompressione di dati e di alcune elaborazioni molto più semplici, come le operazioni matematiche su vettori di grandi dimensioni.

Il terzo modo per far fronte al costo del trasferimento è quello di utilizzare la CPU e la GPU insieme, migliorando le prestazioni attraverso l'assegnazione di una parte del lavoro a ciascuna di esse, ossia considerando il sistema

come una piattaforma eterogenea di calcolo. Il modello di programmazione CUDA supporta l'assegnamento dell'attività a una o più GPU e il contemporaneo utilizzo continuato della CPU mediante funzioni asincrone di GPU, senza dover definire dei thread; è quindi relativamente semplice far lavorare in modo concorrente tutte le GPU e la CPU per incrementare ulteriormente le prestazioni.

C.10 Note conclusive

Le GPU sono processori massicciamente paralleli che si sono ormai diffusi non solo nell'ambito della grafica 3D, ma anche nel contesto di molte altre applicazioni. Una così vasta gamma di applicazioni è stata resa possibile dall'evoluzione dei dispositivi grafici in processori programmabili. Il modello di programmazione su GPU per le applicazioni grafiche è solitamente una API, come DirectX o OpenGL. Per le applicazioni di calcolo più generali, il modello di programmazione CUDA è basato sullo stile SPMD (singolo programma, dati multipli) con l'impiego di molti thread paralleli.

Il parallelismo delle GPU continuerà a crescere in accordo con la legge di Moore, principalmente grazie all'aumento del numero di processori. Soltanto i modelli di programmazione parallela che possono scalare in modo semplice su centinaia di processori e migliaia di thread avranno successo sulle GPU e CPU con moltissimi core. Inoltre, solo le applicazioni caratterizzate da molti compiti paralleli, in gran parte indipendenti, potranno essere accelerate dalle architetture a moltissimi core massicciamente parallele.

I modelli di programmazione delle GPU stanno diventando sempre più flessibili, sia per la grafica sia per il calcolo. Per esempio, CUDA sta evolvendo rapidamente per fornire le funzionalità complete del C/C++. Le API della grafica e i modelli di programmazione si adatteranno probabilmente alle possibilità offerte dal calcolo parallelo e da CUDA. CUDA, inoltre, è organizzato in thread di tipo SPMD, è in grado di scalare e rappresenta un modello pratico, sintetico e facile da capire per implementare un parallelismo elevato.

Trascinata da questi cambiamenti nei modelli di programmazione, l'architettura delle GPU, dal canto suo, sta diventando più flessibile e più programmabile. Le unità a funzione fissa delle GPU stanno diventando accessibili ai programmi generici, seguendo le stesse modalità con cui i programmi CUDA già utilizzano le funzioni intrinseche di tessitura per il campionamento della tessitura (attraverso le istruzioni GPU di gestione della tessitura e l'unità di tessitura stessa).

L'architettura delle GPU continuerà ad adattarsi alle richieste dei programmatori sia di applicazioni grafiche sia di altre applicazioni. Le GPU, inoltre, continueranno a svilupparsi fornendo potenze di calcolo sempre maggiori attraverso l'aumento dei core di elaborazione, della banda di trasferimento, dei thread e della memoria a disposizione dei programmi. Infine, anche i modelli di programmazione si evolveranno, per adattarsi ai sistemi eterogenei a moltissimi core che contengono sia GPU sia CPU.

Ringraziamenti

Quest'appendice è frutto del lavoro di diverse persone di NVIDIA. Esprimiamo la nostra gratitudine per il loro contributo a Michael Garland, John Montrym, Doug Voorhies, Lars Nyland, Erik Lindholm, Paulius Micikevicius, Massimiliano Fatica, Stuart Oberman e Vasily Volkov.

C.11 Inquadramento storico e approfondimenti

Questo paragrafo, che si trova nel CD, fornisce una panoramica storica sulle unità programmabili di elaborazione grafica in tempo reale (GPU) dagli inizi degli anni Ottanta a oggi, periodo in cui il prezzo delle GPU si è ridotto di due ordini di grandezza e le prestazioni sono cresciute altrettanto. Viene descritta l'evoluzione delle GPU dalle pipeline a funzione prefissata ai processori grafici programmabili, con alcuni accenni al calcolo su GPU, ai processori unificati per grafica e calcolo, all'elaborazione visuale e alle GPU scalabili.