ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

GENERAL PURPOSE COMPUTING ON GRAPHIC PROCESSING UNIT (GPGPU): METODI E TECNOLOGIE PER LO SVILUPPO DI APPLICAZIONI

Elaborata nel corso di: Sistemi Operativi LA

Tesi di Laurea di:
TOMMASO ARMEO SOFI

Relatore:
Prof. ALESSANDRO RICCI

ANNO ACCADEMICO 2011-2012
SESSIONE III

PAROLE CHIAVE

GPGPU

Sistemi Eterogenei

Cuda

OpenCL

Programmazione Parallela

Indice

Introduzione

Microprocessori basati su singolo processore (CPU), hanno visto una rapida crescita di performances ed un abbattimento dei costi per circa venti anni. Questi microprocessori hanno portato una potenza di calcolo nell'ordine del GFLOPS (Giga Floating Point Operation per Second) sui PC Desktop e centinaia di GFLOPS su clusters di server. Questa ascesa ha portato nuove funzionalità nei programmi, migliori interfacce utente e tanti altri vantaggi. Tuttavia questa crescita ha subito un brusco rallentamento nel 2003 a causa di consumi energetici sempre più elevati e problemi di dissipazione termica, che hanno impedito incrementi di frequenza di clock. I limiti fisici del silicio erano sempre più vicini.

Per ovviare al problema i produttori di CPU (Central Processing Unit) hanno iniziato a progettare microprocessori multicore, scelta che ha avuto un impatto notevole sulla comunità degli sviluppatori, abituati a considerare il software come una serie di comandi sequenziali. Quindi i programmi che avevano sempre giovato di miglioramenti di prestazioni ad ogni nuova generazione di CPU, non hanno avuto incrementi di performance, in quanto essendo eseguiti su un solo core, non beneficiavano dell'intera potenza della CPU. Per sfruttare appieno la potenza delle nuove CPU la programmazione concorrente, precedentemente utilizzata solo su sistemi costosi o supercomputers, è diventata una pratica sempre più utilizzata dagli sviluppatori.

Allo stesso tempo, l'industria videoludica ha conquistato una fetta di mercato notevole: solo nel 2013 verranno spesi quasi 100 miliardi di dollari fra hardware e software dedicati al gaming[2]. Le software houses impegnate nello sviluppo di videogames, per rendere i loro titoli più accattivanti, puntano su motori grafici sempre più potenti e spesso scarsamente ottimizzati, rendendoli estremamente esosi in termini di performance. Per questo motivo i produttori di GPU (Graphic Processing Unit), specialmente nell'ultimo

decennio, hanno dato vita ad una vera e propria rincorsa alle performances che li ha portati ad ottenere dei prodotti con capacità di calcolo vertiginose. Ma al contrario delle CPU che agli inizi del 2000 intrapresero la strada del multicore per continuare a favorire programmi sequenziali, le GPU sono diventate manycore, ovvero con centinaia e centinaia di piccoli cores che eseguono calcoli in parallelo. Questa immensa capacità di calcolo può essere utilizzata in altri campi applicativi? La risposta è si e l'obiettivo di questa tesi è proprio quello di constatare allo stato attuale, in che modo e con quale efficienza può un software generico, avvalersi dell'utilizzo della GPU invece della CPU.

Capitolo 1: GPU e CPU: architetture a confronto

In questo capitolo vengono descritte e messe a confronto, le architetture hardware di GPU e CPU evidenziando i componenti e i modelli di esecuzione delle istruzioni.

Capitolo 2: Programmazione

In questo capitolo sono esposti i principi della programmazione a basso livello delle GPU ed un'introduzione sugli attuali strumenti di sviluppo ad alto livello.

Capitolo 3: CUDA

In questo capitolo viene descritto l'ambiente di programmazione NVIDIA CUDA (Compute Unified Device Architecture), per la programmazione di sistemi eterogenei basati esclusivamente su CPU e GPU NVIDIA

Capitolo 4: OpenCL e AMD App Acceleration

In questo capitolo viene descritto l'ambiente di programmazione OpenCL (Open Computing Language), per la programmazione di sistemi eterogenei basati su differenti tipi di dispositivi come CPU, GPU, DSP (Digital Signal Processor) e FPGA (Field Programmable Gate Array)

Capitolo 5: Testing

In questo capitolo sono stati effettuati dei test comparativi fra un'applicazione standard che utilizza solo la CPU e un'applicazione che utilizza sia CPU che GPU

Capitolo 6: Conclusioni

In questo capitolo sono presenti considerazioni sullo attuale della programmazione eterogenea, considerando le architetture hardware e gli ambienti di sviluppo a disposizione dei programmatori. Inoltre sono presenti delle riflessioni sugli eventuali sviluppi futuri della programmazione eterogenea.

Capitolo 1

GPU e CPU: architetture a confronto

In questo capitolo si fornisce un'overview sulle architetture di questi microprocessori.

1.1 Architettura GPU

1.1.1 Cenni storici

Il concetto moderno di GPU è stato sviluppato per lo più negli ultimi 15 anni [1]. Precedentemente l'elaborazione grafica era affidata a dispositivi chiamati VGA (Video Graphic Array), che erano semplicemente dei controller di memoria dotati di DRAM interfacciati con un uscita video. Essenzialmente ricevevano i dati delle immagini, le elaboravano e le davano in uscita su un dispositivo, tipicamente un monitor. Negli anni '90 questi controller iniziarono ad essere dotati di acceleratori grafici, in grado di effettuare operazioni di rasterization, texture mapping, e shading. Nel 1999 la GeForce 256 di Nvidia debutta sul mercato e può essere considerata la prima vera GPU. Essenzialmente il suo compito era quello di prendere un set di coordinate poligonali, colori, texture e specifiche di luminosità in modo da creare efficientemente un output video. Col passare del tempo le GPU sono diventate sempre più programmabili, fino a riuscire ad eseguire veri e propri programmi (shaders) per ogni singolo triangolo, vertice e pixel elaborato, aumentando considerevolmente la varietà e la qualità degli effetti

ottenibili come ombre e riflessi. Il passaggio progressivo da una computazione ad aritmetica indicizzata, ad una a virgola mobile con doppia precisione, ha permesso un incremento di precisione notevole. Oggi le GPU, dotate di centinaia di cores e di threads, sono strumenti formidabili nella computazione parallela e negli ultimi anni le ISA sono state estese per supportare linguaggi come C, C++, Python, Java e Fortran.

1.1.2 Struttura e funzionamento

La veloce evoluzione dell'hardware e le differenti tecniche progettuali dei produttori, non permettono di generalizzare un concetto di architettura GPU, tuttavia le somiglianze sono parecchie. Analizziamo come esempio una famiglia specifica, la RV700 prodotta da AMD. In figura 1.1 è possibile vedere la struttura logica della GPU. I registri sono di quattro tipi ed utilizzano la convenzione little-endian per l'ordinamento dei bit:

- GPRs (General Purpose Registers): 128 GPRs da 128bit, ognuno organizzato in 4 blocchi da 32bit
- CRs (Constant Registers): 512 CRs da 128bit, ognuno organizzato in 4 blocchi da 32bit
- AR (Address register)
- Loop Index: un registro inizializzato dal software ed incrementato dall'hardware ad ogni iterazione

Sono presenti inoltre un DPP (Data-Parallel Processor) array con 800 core, un command processor, un controller di memoria e altre logiche. Il command processor legge i comandi che la CPU ha scritto nei registri mappati nella memoria di sistema. Quando il comando è concluso il command processor invia alla CPU un interrupt. Il controller di memoria ha accesso a tutta la memoria della GPU ed anche alle regioni di memoria di sistema specificate dalla CPU. Per effettuare processi di lettura e scrittura, il controller di memoria utilizza un controller DMA (Direct-Memory Access). Un'applicazione non ha accesso diretto alla memoria locale della GPU, ma può chiedere alla GPU di copiare programmi e dati fra la memoria locale e la memoria di sistema. Quindi la CPU ha due metodi per scrivere sulla

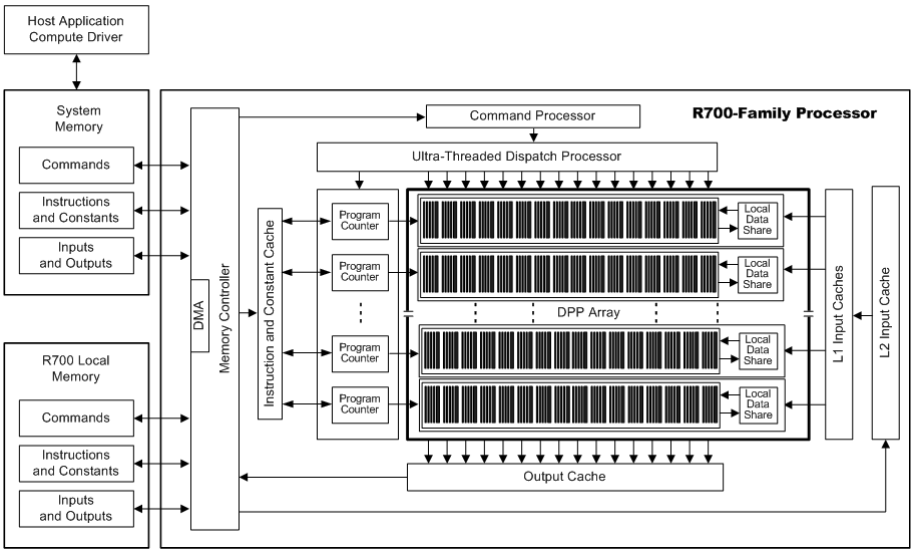


Figura 1.1: Diagramma a blocchi dell'architettura R700 di AMD

memoria della GPU: richiedere al controller DMA di effettuare l'operazione, oppure far eseguire alla GPU un programma che esegue il medesimo scopo.

I programmi che girano sulla GPU sono controllati dalla CPU in questa sequenza: configura i registri e imposta gli indirizzi, specifica il data domain dove deve operare la GPU, cancella la cache e fa eseguire il progamma.

Il DDP array è il cuore pulsante della GPU. L'array è organizzato come un set di pipeline SIMD, ognuna indipendente dall'altra, che operano in parallelo su flussi di dati interi o in virgola mobile. La CPU richiede ad una pipeline di eseguire un kernel, passandogli un valore condizionale, una coppia di identificatori (x, y) e la posizione in memoria del codice del kernel. Ricevuta la richiesta, la pipeline carica le istruzioni e i dati dalla memoria, inizia l'esecuzione e continua fino al completamento. Durante l'esecuzione dei kernel, la GPU automaticamente preleva dati dalla memoria e li sposta nella cache, operazione totalmente indipendente dal software. Concettualmente ogni pipeline mantiene un'interfaccia di memoria separata, consistente in coppie di indici e un campo che identifica i tipi di richiesta (istruzione di un programma, costante in virgola mobile, costante booleana, lettura di un input...). I programmi che girano su GPU non possono in alcun modo interrompere l'operazione di una pipeline, per questo motivo

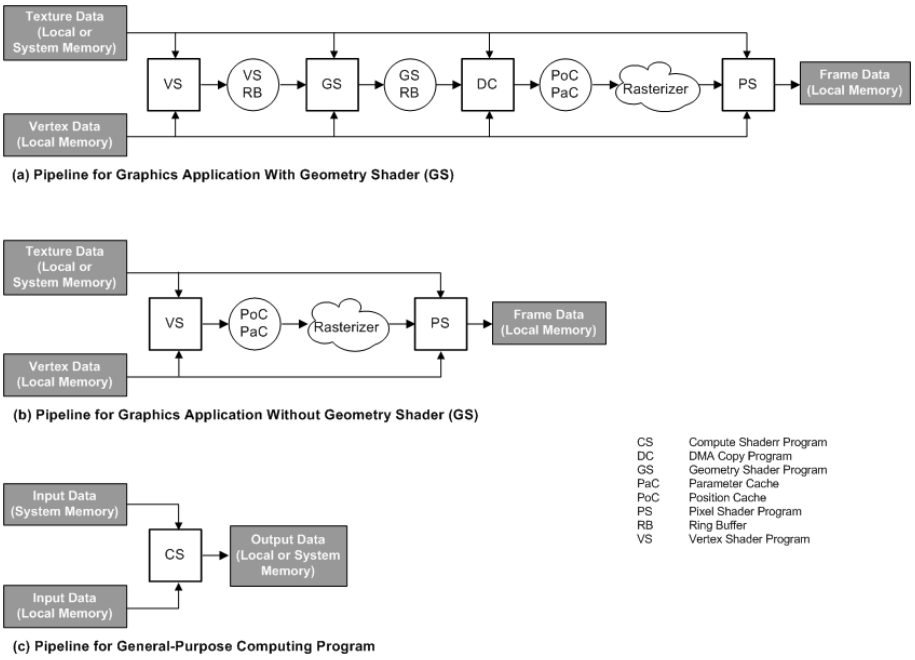


Figura 1.2: Dataflow di tre diverse applicazioni

non sono supportati interrupt, eccezioni, errori o altri eventi. Le uniche interruzioni possibili sono generate dall'hardware.

Nella figura 1.2 sono mostrati i dataflow di tre diverse applicazioni. Il primo caso (a) è un'applicazione grafica che include uno shader geometrico e un programma per la copia di dati tramite DMA. Il secondo caso (b) è analogo al primo ma senza lo shader. Il terzo caso (c) è un'applicazione general purpose. I blocchi quadrati rappresentano programmi eseguiti dal DDP array, mentre i cerchi e le nuvole rappresentano funzioni hardware non programmabili. Per le applicazioni grafiche, ogni blocco della catena elabora un particolare tipo di dato e invia i risultati al blocco successivo, mentre nell'applicazione general purpose un solo blocco esegue tutte le computazioni.

Figura 1.3: Kepler Chip

1.1.3 Il presente: Nvidia Kepler

Una delle GPU di riferimento del mercato attuale è senza dubbio il GK110 di Nvidia della famiglia Kepler. Nel die di questo chip troviamo 15 unità SMX (Streaming Multiprocessor) e 6 controller di memoria a 64bit, come mostrato in figura 1.3. La struttura interna dei singoli SMX è mostrata in figura 1.4. Ogni unità SMX contiene 192 cores(per un totale di 2880 cores e 7 miliardi di transistors) ed è in grado di gestire 2048 threads, il che fa intendere l'immensa capacità di calcolo in parallelo. Questa GPU che lavora a 900MHz, offre fino a 4,5 TFLOPs in singola precione e 1,5 TFLOPs in doppia precisione.

Figura 1.4: SMX Architecture

1.2 Architettura CPU

1.2.1 Struttura e funzionamento

A differenza delle GPU, l'architettura delle CPU si basa su uno standard vecchio 40 anni, l'x86. Durante gli anni sono stati aumentati i registri, allungate le pipeline, integrate nuove logiche ma sempre mantenendo gli stessi principi di funzionamento. Gli stadi di funzionamento di un ciclo di clock di una generica CPU possono essere sintetizzati come segue:

- Fetch: viene prelevata dalla memoria l'istruzione il cui indirizzo è contenuto nel program counter
- Decodifica: l'istruzione viene decodificata e vengono prelevati dati da eventuali registri sorgente
- Esecuzione: l'istruzione viene eseguita dalla
  e ALU
- Retire: i risultati della computazione vengono memorizzati nella memoria principale o nei registri interni

In figura 1.5 è illustrata la struttura logica interna di un microprocessore storicamente famoso: l'Intel Pentium 4. Si nota la presenza di due livelli di cache, che durante la fase di fetch memorizzano i dati dalla memoria centrale prima di inviarli al processore. Anche una trace cache è presente in modo da riutilizzare decodifiche precedentemente effettuate. I calcoli sono affidati a tre ALU, due per il calcolo veloce di operazioni semplici con clock raddoppiato e una lenta per i calcoli più complessi (virgola mobile, moltiplicazioni). Le pipeline sono a venti stadi suddivisi nella seguente seguenza: trace ->fetch ->queue ->schedulars ->dispatch ->execute ->retire. Si nota che non è presente il controller della memoria, solo anni dopo verrà integrato nel die del processore[3].

Anche per le CPU i registri sono di quattro tipi ed utilizzano la convenzione little-endian per l'ordinamento dei bit:

- GPRs (General Purpose Registers): 8 GPRs da 32bit
- Segment Registers: 6 SRs da 16bit
- EFLAGS Register: unico da 32bit
- EIP (Instruction Pointer register): unico a 32bit

I GPR sono utilizzati per memorizzare operandi di operazioni logiche e aritmetiche, operandi per il calcolo di indirizzi e puntatori a memoria. I Segment Register contengono i segment selectors, che sono puntatori speciali che identificano segmenti di memoria. Il registro EFLAGS contiene flags di stato, di controllo e di sistema utilizzate durante l'esecuzione delle istruzioni. L'EIP non è nient'altro che il program counter, continene l'offset, nel segmento di codice corrente, rispetto alla successiva istruzione da eseguire. Non è accessibile direttamente dal software, invece è controllato implicitamente da istruzioni di trasferimento di controllo (JUMP, CALL), interrupts ed eccezioni.

1.2.2 Il presente: Intel Ivy Bridge

Nella fascia alta del mercato CPU troviamo l'architettura Ivy Bridge di Intel. Come è possibile vedere è decisamente diversa dall'architettura di una GPU (fig. 1.6). Sul die del modello i7-3770k troviamo 4 cores (1,4 miliardi di transistors) che lavorano a 3,5GHz, in grado di gestire 2 threads ciascuno con un livello di cache per le istruzioni da 128 KB e tre livelli di cache per i dati rispettivamente da 128KB, 1MB e 8MB

1.3 GPU vs. CPU

Quali sono quindi pregi e difetti di queste due architetture? La CPU nonostante il parallelismo nettamente inferiore, è in grado di eseguire una varietà di tasks più ampia, opera a frequenze più alte e dispone di una cache più veloce e di maggiori dimensioni garantendo latenze ridotte, il che la

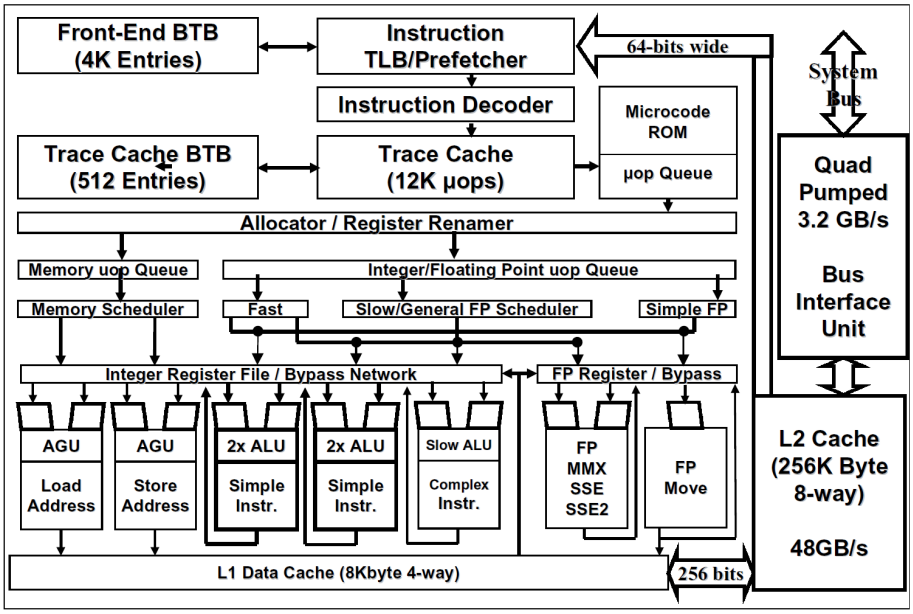


Figura 1.5: Diagramma a blocchi del Pentium 4

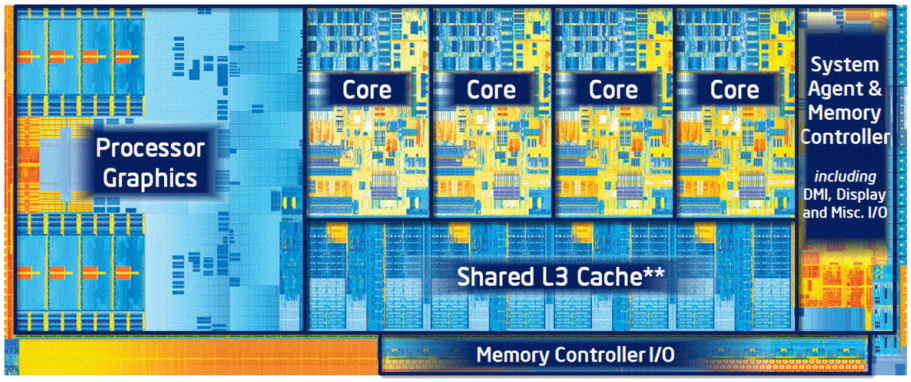


Figura 1.6: Intel i7-3770k Architecture

rende preferibile per operazioni sequenziali. La GPU invece grazie all'elevato numero di core e al troughput notevole, mira a svolgere parallelamente una moltitudine di tasks primitivi (vertici, triangoli, pixels). Nei prossimi capitoli verranno analizzati gli strumenti software per sfruttare queste tecnologie.

Capitolo 2

Programmazione

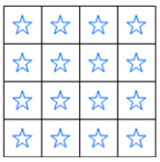
In questo capitolo si tratta l'approccio software al GPGPU.

2.1 Programmazione a basso livello

Diventando le GPU sempre più programmabili col passare degli anni, i loro set di istruzioni sono stati estesi per permettere l'esecuzione di un sempre maggior numero di tasks. Ad oggi è possibile eseguire sulle GPU le classiche istruzioni tipiche della programmazione su CPU, come cicli e condizioni, lettura arbitraria alla memoria e calcoli in virgola mobile. I set di istruzioni sono diventati inoltre sempre più ottimizzati per eseguire codice scritto in C: i due maggiori produttori di schede video discrete, Nvidia e AMD, hanno sviluppato da qualche anno le loro GPU con architetture GPGPU friendly, fornendo agli sviluppatori anche i relativi ambienti di sviluppo che permettono la programmazione in diversi linguaggi, oltre al C, fra cui C++, Python, Fortran e Java, che sono stati estesi per essere utilizzati con le GPU.

Le istruzioni della GPU sono essenzialmente Single-Instruction, Multiple-Data (SIMD), ovvero una singola istruzione viene eseguita su molteplici dati; basti pensare ad uno shader che esegue la stessa routine su migliaia di triangoli per applicare filtri, calcolare riflessi ecc... Volendo fare una transizione di codice da CPU a GPU si può considerare il seguente esempio: ipotizziamo di dover incrementare di 1 tutti gli elementi di un array bidimensionale 4 * 4. Utilizzando il classico approccio iterativo, il problema si affronta con 2 loop innestati, che scorreranno una cella alla volta l'array in-

```
block = 1:4 by 1:4
if y[i][j] = within block
   y[i][j] = y[i][j] + 1
```



```
for i = 1 to 4
   for j = 1 to 4
      y[i][j] = y[i][j] + 1
```

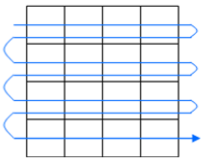


Figura 2.1: Differenze fra le operazioni di incremento di un array fra una GPU(sinistra) e una CPU(destra)

crementandone il valore; invece su una GPU, una singola operazione SIMD è in grado di modificare tutte le celle contemporaneamente. In figura 2.1 è possibile vedere come i 2 script funzionano su una CPU e una GPU con 16 cores. Una debolezza invece dei programmi per GPU sono i branch(salti), in quanto nel caso in cui in un blocco computazionale, non si vuole far eseguire lo stesso tipo di salto a tutti i singoli elementi, l'operazione diventa seriale rallentando drasticamente l'esecuzione del programma.

2.2 Programmazione ad alto livello

Allo stato attuale, gli sviluppatori dispongono di validi strumenti per la programmazione di software che si avvale dell'utilizzo della GPU in contesti non puramente grafici. Fra i principali ambienti di sviluppo troviamo CUDA e OpenCL. CUDA[5] è un'architettura hardware proprietaria di NVIDIA, che da il nome anche al relativo ambiente di sviluppo. Offre estensioni per i più utilizzati linguaggi di programmazione ed integrazione negli IDE (Integrated Development Environment) più diffusi come Eclipse e Visual Studio. Nel 2012 CUDA vantava un bacino di circa 120000 sviluppatori attivi[5], dua anni prima erano circa un terzo[4], il che sta a dimostrare il sempre maggiore interesse che questa tecnologia sta sviluppando nell'ambiente della programmazione parallela. Il secondo protagonista nel calcolo parallelo è OpenCL, che a differenza della controparte NVIDIA è uno standard aperto e può essere utilizzato, non solo con GPU di differenti produttori, ma anche con microprocessori di differenti tipologie. Sebbene OpenCL sia un soluzio-

ne più completa e versatile non vanta la maturità e la semplicità di utilizzo di CUDA.

CUDA e OpenCL sebbene siano i più importanti,non sono gli unici attori nel panorama della computazione parallela e nuovi ambienti di sviluppo emergono costantemente: uno di questi è Microsoft DirectCompute. Nonostante non sia di recente sviluppo[6], questa API (Application Programming Interface) presente nelle librerie DirectX è rimasta nell'ombra e solo ultimamente sta destando interesse nella community degli sviluppatori, grazie alle meccaniche di programmazione molto simili a quelle dei due principali competitors.

Nei prossimi capitoli verranno analizzati i modelli di programmazione di CUDA e OpenCL e i loro punti di forza.

Capitolo 3

CUDA

Questo capitolo tratta l'ambiente di sviluppo CUDA.

3.1 Cos'è CUDA

Come accennato precedentemente, Nvidia utilizza da qualche anno l'architettura CUDA (Compute Unified Device Architecture) per realizzare le proprie GPU, mettendo a disposizione anche il relativo ambiente di sviluppo per la computazione general purpose. Il linguaggio maggiormente usato è CUDA-C che, come suggerisce il nome, è un sottoinsieme del liguaggio C con estensioni NVIDIA, sottoinsieme in quanto manca di ricorsione, puntatori a funzione ed altre estensioni. CUDA-C quindi consente la programmazione eterogenea e fornisce le API per la gestione diretta di dispositivi e memoria senza ricorrere ad API grafiche specifiche come DirectX.

3.2 Programmazione eterogenea

Cosa si intende per programmazione, o meglio computazione eterogenea? Ormai è chiaro che le CPU eccellono nel calcolo seriale con bassa latenza, mentre le GPU nel calcolo in parallelo, quindi il metodo più performante per l'esecuzione di un programma è quello di ripartire il carico di lavoro fra i due processori sfruttando i loro punti di forza. Nella figura 3.1 è possibile vedere un esempio di computazione eterogenea, con la relativa suddivisione dei carichi fra CPU e GPU nell'esecuzione del codice.

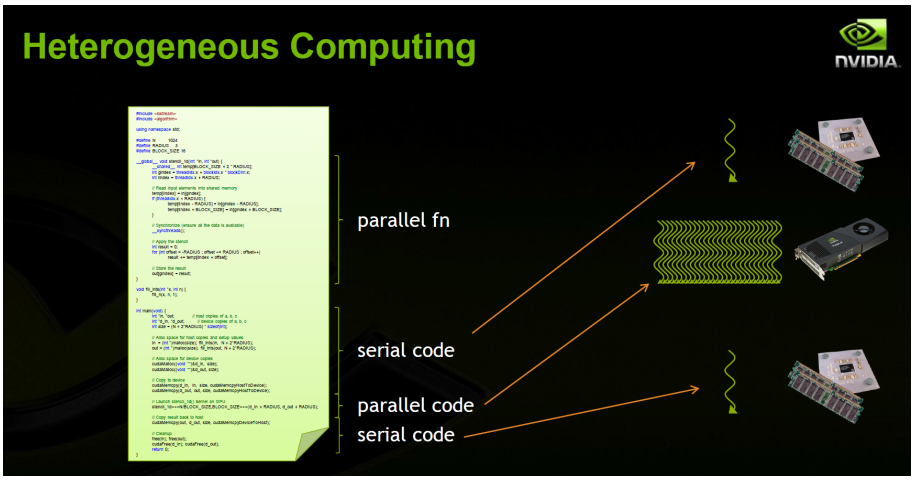


Figura 3.1: Computazione eterogenea

Purtroppo l'elaboratore non è così intelligente da capire autonomamente come ripartire l'esecuzione del codice, quindi è a carico dello sviluppatore indicare quali parti vanno fatte eseguire dalla CPU e quali dalla GPU. Per questo è necessario identificare in maniera differente CPU e GPU(e relative memorie) che chiameremo rispettivamente HOST e DEVICE per fini pratici. L'esecuzione di un programma eterogeneo è cosi strutturata:

1. caricamento dei dati nella memoria principale di sistema
2. invio dei dati alla memoria della GPU
3. invio alla GPU da parte della CPU del programma da eseguire
4. copia dei risultati dalla memoria della GPU alla memoria di sistema

Iniziamo col vedere il classico programma "Hello World!":

```
int main(void) {
  printf("Hello World! \n");
  return 0;
}
```

Essendo un codice standard C gira sull'host. Nonostante non sia presente codice destinato al device, il compilatore NVIDIA (NVCC) può essere usato

per compilare questo frammento ed avrà esattamente lo stesso risultato, ovvero la visualizzazione della stringa "Hello World!".

Vediamo ora la versione del precedente programma destinata all'esecuzione sul device:

```
__global__void mykernel(void) {
}

int main(void) {
  mykernel<<<1,1>>>();
  printf("Hello World! \n");
  return 0;
}
```

Si nota l'introduzione di nuovi elementi, il qualificatore *__global__* e il simbolo <<<>>>chiamato anche kernel launch.

Il qualificatore *__global__*, chiamato dal codice dell'host, indica che la funzione deve essere eseguita sul device. Il NVCC divide quindi il codice sorgente in 2 parti, quella destinata al device che verrà compilata da se stesso, e l'altra che verrà affidata al compilatore standard dell'host. In questo caso *mykernel()* verrà processato dal compilatore NVIDIA, mentre il *main()* da quello standard.

Le triple parentesi angolari, rappresentano una chiamata dal codice host al codice device. Questa operazione viene definita kernel lauch. Torneremo in seguito sul significato dei parametri al suo interno. Questi due operatori sono sufficenti per eseguire un programma su una GPU con architettura CUDA.

Vediamo ora un esempio più concreto dell'utilizzo di una GPU, visto che nel codice precedente il metodo eseguito dal device era vuoto:

```
__global__void add(int *a, int *b, int *c) {
  *c = *a + *b;
}
```

Semplicemente questo programma effettua la somma di due valori e immagazzina il risultati in un terzo. Notare che le variabili sono dei puntatori, ed essendo *add()* eseguito sul device, anche i puntatori devono puntare alla memoria del device. Sebbene sia possibile copiare un puntatore da un codice all'altro, non è possibile dereferenziare un puntatore che punta alla memoria del device nel codice host e vice versa. Per gestire la memoria della GPU, CUDA mette a disposizione delle API in cui troviamo i metodi *cudaMalloc()*, *cudaFree()* e *cudaMemcpy()*,quasi omonimi dei ben noti me-

todi presenti in C *malloc()*, *free()* e *memcpy()*. Vediamo ora di scrivere la funzione *main()*:

```
int main(void) {
  int a, b, c;   // copie host di a, b, c.
  int *d_a, *d_b, *d_c; // copie device di a, b, c.

  // Allocazione dello spazio nella memoria della GPU per le copie di a, b, c.
  cudaMalloc((void **)&d_a, size);
  cudaMalloc((void **)&d_b, size);
  cudaMalloc((void **)&d_c, size);

  // Impostazione dei valori iniziali
  a=2;
  b=7;

  // Copia dei valori nella memoria del device
  cudaMemcpy (d_a, &a, size, cudaMemcpyHostToDevice);
  cudaMemcpy (d_b, &b, size, cudaMemcpyHostToDevice);

  // Esecuzione di add() sulla GPU
  add<<<1,1>>>(d_a, d_b, d_c);

  // Copia dei risultati nell'host
  cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

  // Cleanup
  cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
  return 0;
}
```

Il codice è di semplice comprensione. Tuttavia questo programma effettua una sola operazione, mentre come abbiamo precedentemente evidenziato, il punto di forza della computazione su GPU è il parallelismo.

3.3 Blocks e Threads

Torniamo quindi ad esaminare i parametri contenuti fra le tripe parentesi angolari, che utilizziamo per il lancio di una funzione:

```
add<<<1,1>>>

add<<<N,1>>>
```

Il primo parametro indica quante volte deve essere eseguita la funzione *add()*, quindi con la sostituzione effettuata, si indica che la funzione deve essere eseguita N volte. Riuscendo ad eseguire *add()* in parallelo, possiamo effettuare una somma di vettori. Introduciamo quindi due nuovi termini: block e grid. Ogni invocazione parallela della funzione viene chiamata

block ed un gruppo di block viene chiamata grid. Ogni invocazione ha un identificatore di block univoco *blockIdx.x*

```
__global__void add(int *a, int *b, int *c) {
  c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

Con l'utilizzo di *blockIdx.x* per indicizzare gli array, ogni invocazione di *add()* gestisce un indice differente, quindi otterremo un grid così fatta:

- Block 0: *c[0] = a[0] + b[0];*
- Block 1: *c[1] = a[1] + b[1];*
- Block 2: *c[2] = a[2] + b[2];*
- ...

Di seguito viene riportato il codice completo con N = 512

```
#define N 512   // definizione della dimensione degli array
int main(void) {
  int *a, *b, *c;   // copie host di a, b, c.
  int *d_a, *d_b, *d_c;  // copie device di a, b, c.
  int size = N * sizeof(int);

  // Allocazione dello spazio nella memoria della GPU per le copie di a, b, c.
  cudaMalloc((void **)&d_a, size);
  cudaMalloc((void **)&d_b, size);
  cudaMalloc((void **)&d_c, size);

  // Allocazione dello spazio nella memoria per le copie di a, b, c e inizializzazione dei valori
  a = (int *)malloc(size); random_ints(a, N);
  b = (int *)malloc(size); random_ints(b, N);
  c = (int *)malloc(size);

  // Copia dei valori nella memoria del device
  cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

  // Esecuzione di add() sulla GPU con N blocks
  add<<<N,1>>>(d_a, d_b, d_c);

  // Copia dei risultati nell'host
  cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

  // Cleanup
  free(a); free(b); free(c);
  cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
  return 0;
}
```

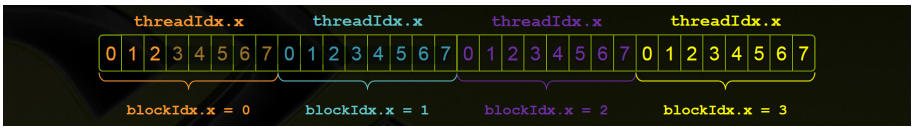


Figura 3.2: Vettore suddiviso in blocks e threads

Il block tuttavia non è un'unità primitiva, infatti esso può essere diviso in molteplici thread paralleli. Vediamo come scrivere la nostra funzione *add()* in modo che utilizzi threads invece di blocks:

```
__global__void add(int *a, int *b, int *c) {
  c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

e la funzione sarà invocata così:

```
add<<<1,N>>>(d_a, d_b, d_c);
```

Abbiamo scoperto quindi che il secondo parametro all'interno delle triple parentesi angolari indica il numero di threads da eseguire. Esistono differenze fra le 2 funzioni *add()* proposte? Praticamente no, infatti la vera differenza la abbiamo con l'uso congiunto di blocks e thread, per alcune proprietà che hanno esclusivamente i threads che vedremo successivamente. L'array in figura 3.2 è diviso in blocks da 8 threads ed ogni thread avrà un indice unico ottenuto da:

```
int index = threadIdx.x + (blockIdx.x * blockDim.x);
```

con *blockDim.x* uguale al numero di threads per block, nel nostro caso 8. Le parentesi sono superflue, ma offrono una migliore visibilità delle priorità.

Quindi la funzione *add()* che utilizza blocks e threads deve essere così definita:

```
__global__ void add(int *a, int *b, int *c) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  c[index] = a[index] + b[index]; }
```

ed anche il main deve subire qualche modifica:

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
  int main(void) {
  int *a, *b, *c;   // copie host di a, b, c
```

```
  int *d_a, *d_b, *d_c;  // copie device a, b, c
  int size = N * sizeof(int);

  // Allocazione dello spazio nella memoria della GPU per le copie di a, b, c.
  cudaMalloc((void **)&d_a, size);
  cudaMalloc((void **)&d_b, size);
  cudaMalloc((void **)&d_c, size);

  // Allocazione dello spazio nella memoria per le copie di a, b, c e inizializzazione dei valori
  a = (int *)malloc(size); random_ints(a, N);
  b = (int *)malloc(size); random_ints(b, N);
  c = (int *)malloc(size);

  // Copia dei valori nella memoria del device
  cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

  // Esecuzione di add() sulla GPU con N blocks
  add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

  // Copia dei risultati nell'host
  cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

  // Cleanup
  free(a); free(b); free(c);
  cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
  return 0;
}
```

dove N in questo caso rappresenta il numero di threads totali.

Sorgono dei problemi nel caso in cui il numero di threads totali non sia multiplo di *blockDim.x*, in quanto la funzione tenterà di effettuare un operazione fra valori inesistenti! Per evitare errori lo sviluppatore può modificare il numero di blocks e threads per block, in modo che il loro prodotto sia esattamente uguale al numero di threads totali, altrimenti è necessario escludere l'ultimo block (non completo) dal calcolo come nel codice seguente:

```
__global__ void add(int *a, int *b, int *c, int n) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  if (index < n) c[index] = a[index] + b[index];
}
```

```
add<<<(N + M - 1) / M,M>>>(d_a, d_b, d_c, N);
```

dove M è il numero di threads per block ed N il numero di threads totali.

A questo punto viene da chiedersi il perchè dell'utilizzo dei threads, in quanto fino ad ora hanno semplicemente aumentato il livello di complessità del codice, quindi sembrando più uno svantaggio che un vantaggio. La risposta si trova in due qualità che i blocks paralleli non hanno: i threads

possono comunicare e sincronizzarsi fra loro, caratteristiche molto utili come vedremo in seguito.

3.4 Memoria condivisa e sincronizzazione threads

Consideriamo di avere due array monodimensionali, in cui i valori di ogni cella del secondo sono dati dalla somma della cella nel primo array nella medesima posizione, sommata ad un range di caselle circostanti. Per esempio se abbiamo due array A e B ed un range uguale a 3, la decima cella del vettore B sarà uguale alla decima cella del vettore A sommata alle tre celle precedenti e alle tre celle successive. Ogni thread elabora il valore di una cella nell'array di uscita leggendo N valori (7 nel nostro esempio) nell'array di input, quindi la stessa cella viene utilizzata per il calcolo di N celle nell'array di uscita. Questo comporta N letture in memoria, cosa estremamente antiperformante, in quanto genera latenza e come abbiamo visto nel capitolo 1, la cache della GPU non eccelle in velocità.

Introduciamo allora la memoria condivisa (shared memory). All'interno di un block, i thread condividono i dati attraverso la memoria condivisa. Questa risiede su un chip dedicato ad alte prestazioni ed è gestibile direttamente dall'utente tramite il qualificatore *shared*. La shared memory di un block non è visibile ai threads di altri blocks.

Quindi l'operazione descritta precedentemente sarà cosi strutturata:

- Lettura di *blockDim.x + 2 * range* elementi nell'array d'ingresso nella memoria principale.
- Copia dei dati nella memoria condivisa
- Computazione di *blockDim.x* elementi in uscita
- Scrittura di *blockDim.x* elementi nella memoria principale

ed il codice sarà così composto:

```
__global__ void stencil(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RANGE];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RANGE;
```

```
  // Scrittura dei dati nella memoria condivisa
  temp[lindex] = in[gindex];
  if (threadIdx.x < RANGE) {
  temp[lindex − RANGE] = in[gindex − RANGE];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
  }

  // Esecuzione dei calcoli
  int result = 0;
  for (int offset = −RANGE ; offset <= RANGE ; offset++)
  result += temp[lindex + offset];

  // Scrittura del risultato
  out[gindex] = result;
}
```

Questo codice sembra ben fatto, ma in realtà può incappare in una race condition, infatti la memoria condivisa di un block potrebbe non essere stata scritta completamente prima che un thread vi acceda, senza trovare quindi il dato che sta cercando. Per evitare questo problema è necessario introdurre una nuova procedura: *syncthreads()*. Questa consente di sincronizzare tutti i threads di un block ed evitare così errori di tipo RAW (Read after Write), WAR(Write after Read) e WAW (Write after Write). Viene inserito quindi *syncthreads()* nel codice precedente subito dopo operazione di scrittura nella memoria condivisa, in modo che questa sia completata prima che i threads inizino ad accedervi.

```
__global__ void stencil(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RANGE];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RANGE;

  // Scrittura dei dati nella memoria condivisa
  temp[lindex] = in[gindex];
  if (threadIdx.x < RANGE) {
  temp[lindex − RANGE] = in[gindex − RANGE];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
  }

  //Sincronizzazione dei threads
  __syncthreads();

  // Esecuzione dei calcoli
  int result = 0;
  for (int offset = −RANGE ; offset <= RANGE ; offset++)
  result += temp[lindex + offset];

  // Scrittura del risultato
  out[gindex] = result;
}
```

3.5 Coordinamento CPU e GPU

Tutte le operazioni di kernel launch sono asincrone: appena vengono lanciate, il controllo torna subito alla CPU. Quindi è necessario sincronizzare la CPU in modo che non vada a leggere in memoria il risultato della funzione eseguita dalla GPU, prima che questa abbia finito. Le librerie CUDA mettono a disposizione delle funzioni per la differente gestione della CPU: Tutte le operazioni di kernel launch sono asincrone, appena vengono lanciate, il controllo torna subito alla CPU. Quindi è necessario sincronizzare la CPU in modo che non vada a leggere in memoria il risultato della funzione eseguita dalla GPU, prima che questa abbia finito. Le librerie CUDA mettono a disposizione delle funzioni per la differente gestione della CPU:

- cudaMemcpy()
- cudaMemcpyAsync()
- cudaDeviceSynchronize()

CudaMemcpy() eseguito dopo il kernel launch, blocca la CPU fino al completamento della copia dei risultati, che inizia quando tutte le chiamate CUDA sono completate. *CudaMemcpyAsync()* invece effettua la copia in memoria in modo asincrono, non bloccando in alcun modo la CPU. *CudaDeviceSynchronize()* blocca la CPU ma solo fino al completamento delle chiamate CUDA e non durante la copia in memoria dei risultati.

Ad ogni modo se vengono generati errori, tutte le chiamate CUDA ritornano un codice d'errore (*cudaError_t*), sia per problemi nelle chiamate CUDA stesse, sia per errori di sincronizzazione, come per esempio dopo un kernel launch. In caso di errore, è possibile ottenere il codice dell'ultimo generato

```
cudaError_t cudaGetLastError(void)
```

oppure avere una stringa con la descrizione dettagliata dell'errore oppure avere una stringa con la descrizione dettagliata dell'errore

```
char *cudaGetErrorString(cudaError_t)
```

ed effettuarne la visualizzazione:
ed effettuare una visualizzazione:

```
printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```

Con il diffondersi della tecnologia SLI (Scalable Link Interface), che permette l'utilizzo in parallelo di più GPU per un miglioramento delle performance nel gaming e dei sistemi multimonitor, i sistemi multi GPU sono diventati sempre più frequenti, e nell'esecuzione di software eterogeneo le API CUDA mettono a disposizione delle funzioni per gestire device molteplici:

Con il diffondersi della tecnologia SLI, che permette l'utilizzo in parallelo di più GPUs per un miglioramento delle performance nel gaming, e dei sistemi multimonitor, i sistemi multiGPU sono sempre più frequenti, e nell'esecuzione di software eterogeneo le API CUDA mettono a disposizione delle funzioni per gestire device molteplici:

- cudaGetDeviceCount(int *count)
- cudaSetDevice(int device)
- cudaGetDevice(int *device)
- cudaGetDeviceProperties(cudaDeviceProp *prop, int device)

Con *cudaGetDeviceCount(int *count)* si ottiene il numero di GPU presenti sulla macchina, *cudaSetDevice(int device)* sceglie quale o quali GPU devono eseguire la chiamata, *cudaGetDevice(int device)* indica quale o quali GPU sono state scelte con la funzione precedente e *cudaGetDeviceProperties(cudaDeviceProp *prop, int device)* permette di leggere le proprietà delle GPU come la memoria totale, il suo clock, il bus, il numero massimo di threads per block e molto altro.

Threads multipli della CPU possono condividere lo stesso device e un singolo thread della CPU può gestire più devices, quindi nel caso di un sistema a 2 GPU un singolo thread della CPU può lavorare su entrambe. Per fare ciò chiamiamo *cudaSetDevice(i)* dove i è il codice della prima GPU, allochiamo la memoria, copiamo i dati nella sua memoria dalla memoria principale, lanciamo il kernel e poi possiamo ripetere il tutto sulla seconda GPU. Se supportato dalle GPU e se il sistema operativo lo permette, *cudaMemcpy* può essere effettuato fra le memorie di due GPU velocizzando l'operazione descritta in precedenza.

Capitolo 4

AMD App Acceleration e OpenCL

Questo capitolo tratta l'ambiente di sviluppo OpenCL.

4.1 AMD App Acceleration

In fase iniziale chiamato ATI Stream, AMD App Acceleration è un ambiente di sviluppo diretto concorrente di CUDA. Si basava sul linguaggio di programmazione a basso livello Close to Metal, ma dopo scarsi risultati l'azienda adottò OpenCL[6].

4.2 Cos'è OpenCL

OpenCL è un framework per la programmazione eterogenea. A differenza della soluzione di NVIDIA, OpenCL è uno standard aperto, mantenuto dal consorzio no-profit tecnologico Khronos Group. In quanto aperto, grande punto di forza di questa tecnologia è l'ampia selezione di hardware supportato che non si limita a CPUs e GPUs, ma include anche altri tipi di processori come per esempio i DSP. Inoltre il codice scritto con OpenCL può essere eseguito su più sistemi senza essere modificato.

OpenCL include un linguaggio di programmazione (C99) per la scrittura di funzioni e APIs per l'interfacciamento con i dispositivi.

4.3 Modello di esecuzione

Si hanno 4 famiglie di oggetti in OpenCL:
Setup

- Devices: CPU, GPU, FPGA, DSP...
- Contexts: gruppi di devices
- Queues: ordinano l'esecuzione dei kernels nei devices

Memory

- Buffers: blocchi di memoria monodimensionali
- Images: array a due o tre dimensioni assimilabili a textures

Execution

- Kernels: funzioni e istanze d'esecuzione
- Programs: gruppi di kernels

Sincronizzazione/profiling

- Events: istanze per la sincronizzazione

Il Kernel è un'unità basica di codice eseguibile, assimilabile ad una funzione C. Può essere data-parallel o task-parallel ad ogni modo il cardine di OpenCL è lo sfruttamento del parallelismo. Il seguente concetto fondamentale è il Programma, una collezione di kernels e altre funzioni, analoghi a librerie dinamiche. Grazie ai programmi possiamo avere operazioni di tipo SPMD(Single Program Multiple Data) oltre a quelle di tipo SIMD viste in precedenza. Quindi possiamo raggruppare delle istruzioni in un kernel e raggruppare diversi kernel in un programma. I programmi possono essere chiamati dalle applicazioni. Infine abbiamo le Code di esecuzione che indicano l'ordine in cui vanno eseguiti i kernel, tuttavia in alcuni casi possono essere lanciati non seguendo l'ordine originale.

4.4 Parallelismo in OpenCL

Vediamo ora come esprimere il parallelismo in OpenCL. Definiamo un dominio computazionale a N dimensioni (N = 1, 2 o 3), N = 1 rappresenta un array monodimensionale, N = 2 una matrice bidimensionale ed N = 3 uno spazio tridimensionale. La dimensionsione totale del dominio è data quindi da N * D dove D rappresenta la grandezza di una singola dimensione. Ogni elemento indipendente in esecuzione nel dominio è chiamato Work-item, quindi la dimensione del dominio definisce il numero totale di work-items che viene eseguito in parallelo. Per esempio se dobbiamo processare un immagine (N = 2) 1024 * 1024, il numero di work-items è uguale al numero totale dei pixel, ovvero 1.048.576, che corrisponde anche al numero di kernel che vengono eseguiti.

Di seguito riportiamo un codice di esempio scritto in C che effettua una semplice moltiplicazione fra vettori:

```
void scalar_mul(int n, const float *a, const float *b, float *result) {
  int i;
  for (i=0; i<n; i++)
    result[i] = a[i] * b[i];
}
```

invece di seguito abbiamo una versione del codice precedente che lavora in parallelo:

```
_kernel void dp_mul(_global const float *a, _global const float *b, _global float *result) {
  int id = get_global_id(0);
  result[id] = a[id] * b[id];
}

// esegue dp_mul su "n" work-items
```

Invece del ciclo *for* abbiamo un'unica computazione parallela dove *_kernel* indica che la funzione è un kernel (come *__global__* in CUDA) e *get_globa_id(0)* indica la grandezza di una dimensione del dominio, quindi il numero di ripetizioni. Approfondiamo un attimo *get_globa_id(D)*. Abbiamo detto che il dominio può avere da una a tre dimensioni, e *get_globa_id(D)* restituisce il valore della dimensione D, quindi per esempio in un dominio tridimensionale, utilizzando coordinate cartesiane, la funzione per D = 0 restituisce il valore di x, per D = 1 il valore di y e per D = 2 il valore di z. Nel codice precedente, essendoci array monodimensionali si avrà che anche il dominio è monodimensionale, quindi totalmente rappresentato da una sola dimensione. Come si può notare, anche in OpenCL è presente un qualifi-

catore *_global*, ma si utilizza per la dichiarazione di variabili e costanti per specificare che risiedono nella memoria globale.

I kernel sono eseguiti su un dominio globale di work-items e questo dominio definisce il range della computazione. I work-items sono raggruppati in Workgroups, la loro dimensione è definita localmente e vengono eseguiti contemporaneamente, condividono oltretutto la stessa porzione di memoria e possono essere sincronizzati. Tuttavia i singoli work-items sebbene possano essere sincronizzati all'interno dello stesso workgroup non possono essere sincronizzati con work-items appartenenti ad altri workgroups. La sincronizzazione fra work-items avviene attraverso memory-fences, del tutto assimilabili al *syncthreads()* in CUDA.

4.5 Struttura dei dispositivi

Anche OpenCL utilizza un sistema di computazione eterogenea dove troviamo un host e uno o più devices. Nel caso di OpenCL un device non è necessariamente una GPU, ma può essere una qualsiasi unità elaborativa come un DSP un FPGA o addirittura un'altra CPU. Quì tratteremo solo device GPU in quanto gli altri non sono inerenti all'argomento di questa tesi. Nell'immagine 4.1 vediamo come sono interfacciati i dispositivi, dove i PE (processing elements) sono unità computazionali generiche come gli streaming processors NVIDIA. Notiamo inoltre la gerarchia dei tipi di memoria utilizzabili dal programmatore: global, constant, local e private. La memoria globale è allocata dinamicamente dall'host e può essere letta e scritta dai device per leggere dati oppure scrivere risultati. La memoria costante invece è sempre allocata dinamicamente dall'host, ma i device possono esclusivamente leggerla. La memoria locale è la controparte OpenCL alla memoria condivisa in CUDA, quindi non è accessibile all'host ed offre supporto in lettura e scrittura a tutti i work-items di un singolo workgroup. La memoria privata è accessibile solo ed esclusivamente da un singolo work-item, che non può quindi accedere alla memoria privata di un'altro work-item.

In OpenCL la gestione della memoria è esplicita, quindi è compito dello sviluppatore scegliere come allocarla e sincronizzarla.

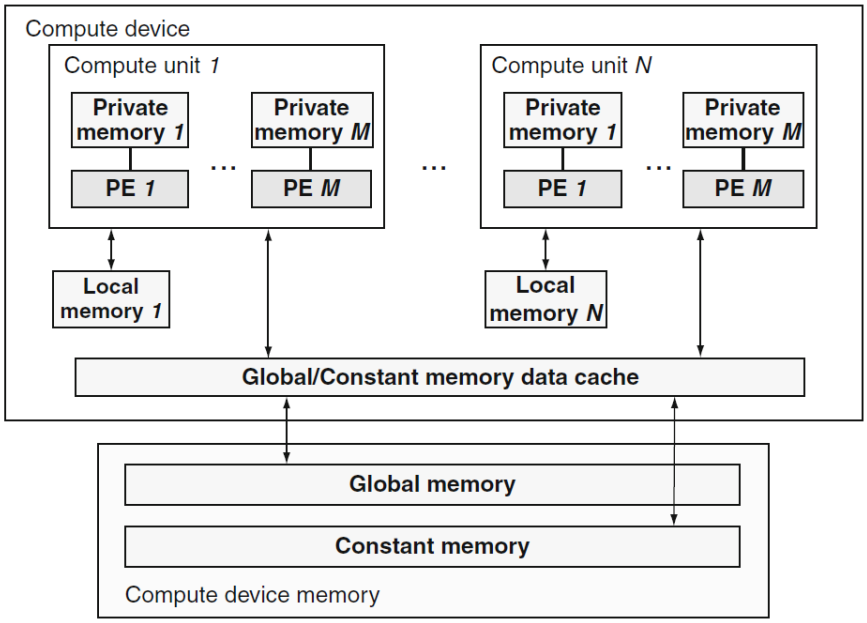


Figura 4.1: Architettura concettuale dei devices in OpenCL (host non presente)

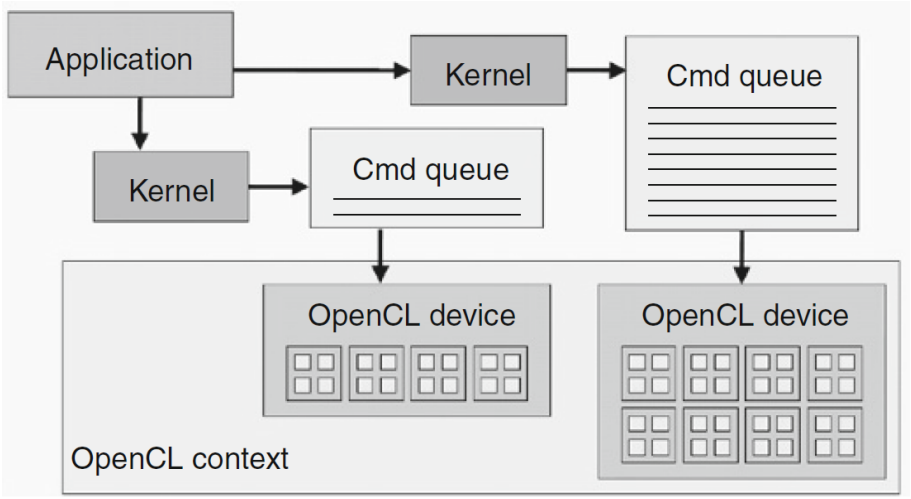


Figura 4.2: Gestione dei devices tramite contexts

4.6 Gestione dei dispositivi

OpenCL definisce un modello di gestione dei device più complesso di quello visto in CUDA, in quanto si trova a dover supportare diversi tipi di hardware. In OpenCL i device sono gestiti attraverso contexts (contesti) come mostrato in figura 4.2. Il programmatore per gestire differenti devices, deve prima creare dei contexts che li contengono, utilizzando le funzioni *clCreateContext()* oppure *clCreateContextFromType()*. Grazie all'utilizzo di *clGetDeviceIDs()* si possono ottenere il numero ed i tipi di devices prenti nel sistema, in modo da passare i corretti parametri durante la creazione dei contexts. I contexts oltre ad essere dei contenitori di devices, sono anche il loro strumento di comunicazione, sia fra di essi che verso l'esterno, ed è sempre il context che gestisce le risorse dei devices, le code di esecuzione ecc...

Per inviare un lavoro da eseguire ad un device, l'host deve prima creare una coda di comandi per il device. Questo può essere fatto chiamando la funzione *clCreateCommandQueue()*. Creata la coda, il codice host può inserirvi una serie di kernel che verranno eseguiti in ordine. Quando il device è disponibile elimina il primo kernel dalla coda e lo esegue. Di seguito è riportato un semplice codice che crea un context per un device e una coda

d'esecuzione:

```
...
cl_int clerr = CL_SUCCESS;
cl_context clctc = clCreateContextFromType(0, CL_DEVICE_TYPE_ALL, NULL, NULL, &clerr);
size_t parmsz;
clerr = clGetContextInfo(clctx, CL_CONTEXT_DEVICES, 0, NULL, &parmsz);
cl_device_id* cldevs = (cl_device_id*) malloc(parmsz);
clerr = clGetContextInfo(clctx, CL_CONTEXT_DEVICES, parmsz, cldevs, NULL(;
cl_command_queue clcmdq = clCreateCommandQueue(clctx, cldevs[0], 0, &clerr)
```

La seconda linea, senza scendere nei dettagli dei parametri accettati da *clCreateContextFromType()*, mostra la creazione di un context che include tutti i device disponibili nel sistema. La quarta linea chiama *clGetContextInfo()* che restituisce il numero di devices nel context. Siccome nella seconda linea la funzione chiede di creare un context con tutti i devices disponibili, l'applicazione non sa esattamente da quanti devices è composto il context. Il secondo argomento di *clGetContextInfo()* specifica che l'informazione richiesta è il numero di devices nel context, mentre il quarto argomento è un puntatore che punta alla cella di memoria dove si vuole inserire il risultato. Tuttavia questo puntatore è vuoto, in quanto la chiamata in realtà non è stata fatta per ottenere il numero di devices presenti nel contesto, ma per conoscere lo spazio in memoria necessario per inserivi il valore. L'applicazione ora conosce lo spazio necessario grazie al quinto parametro della funzione, il puntatore alla variabile *parmsz* che ora contiene questo valore. Ora è possibile allocare la memoria necessaria, e nella quinta linea viene effettuata l'operazione, assegnando l'indirizzo allocato al puntatore *cldevs*. Nella linea 6 viene chiamata nuovamente la funzione *clGetContextInfo()*, questa volta per ottenere effettivamente la lista dei devices presenti nel context, che verrà depositata nella zona di memoria puntata da *cldevs*. Nella settima linea viene creata la coda dei comandi per il primo dispositivo nella lista definita dall'array *cldevs*. Si nota subito una differenza di complessità fra il codice precedentemente analizzato ed i frammenti visti nel capitolo precedente, questo perchè l'API di CUDA offre un livello di astrazione più alto, nascondendo al programmatore le parti più macchinose quando non necessarie.

Come detto in precedenza, OpenCL supporta una gran varietà di microprocessori, ed essendo la scelta del device sul quale far girare un determinato kernel a carico dello sviluppatore, è necessario che egli conosca le caratteristiche dei devices in modo da optare per quello che può eseguire al meglio l'algoritmo interessato. Torna utile la funzione *clGetDeviceInfo(device, pa-*

*ram_name, *value)* che restituisce informazioni sul device interpellato, come il numero di unità computazionali, frequenza di clock, dimensioni della memoria ecc...

4.7 Oggetti in Memoria

In memoria si hanno due tipi di oggetti in OpenCL: buffers e images. Un buffer è un semplice spazio di memoria monodimensionale (vettore), accessibile dal kernel in qualsiasi modo(array, puntatore, struct...). Un buffer può essere sia letto che scritto da un kernel. Una image è una struttura bidimensionale o tridimensionale comparabile ad una texture e a differenza del buffer non offre libero accesso ai kernel. Per interagire con una image il kernel deve usare chiamate specifiche: *read_image()* per la lettura e *write_image()* per la scrittura. inoltre una singola image, può essere esclusivamente letta o scritta dallo stesso kernel. Le images sono importanti in quanto strutture già definite, che essendo assimilabili a textures, vengono elaborate in maniera familiare dalle GPU aventi harware specifico per la gestione di questi oggetti. Vediamo un esempio di allocazione:

```
cl_image_format format;
format.image_channel_data_type = CL_FLOAT;
format.image_channel_order = CL_RGBA;

cl_mem input_image;
input_image = clCreateImage2D(context, CL_MEM_READ_ONLY, &format, image_width,
      image_height, 0, NULL, &err);
cl_mem output_image;
output_image = clCreateImage2D(context, CL_MEM_WRITE_ONLY, &format, image_width,
      image_height, 0, NULL, &err);

cl_mem input_buffer;
input_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(cl_float)*4*image_width*
      image_height, NULL, &err);
cl_mem output_buffer;
output_buffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(cl_float)*4*image_width*
      image_height, NULL, &err);
```

Nelle prime tre righe di codice viene dichiarato il tipo e il formato dell'image. Il data-type è impostato come float ed il channel order come RGBA (Red, Green, Blue, Alpha), quindi quattro canali. Successivamente vengono create due images bidimensionali. Il primo parametro definisce il context, che come visto in precedenza è obbligatorio, mentre il secondo parametro indica l'utilizzo che si intende fare dell'oggetto. Nel terzo parametro si

indica il tipo di formato e nei successivi le dimensioni dell'image. Quindi si ottengono una image chiamata *input_image* di sola lettura e un'altra chiamata *output_image* di sola scrittura. Nelle righe successive vengono creati due buffers, ma a differenza delle images, hanno una sola dimensione. Quindi affinchè abbiano la stessa dimensione delle images, la calcoliamo moltiplicando numero di canali, altezza e larghezza dell images.

Creati questi oggetti in memoria bisogna vedere come avviene lo spostamento dei dati. La funzione *clEnqueueReadBuffer(...)* permette di leggere i dati presenti in una regione dell'oggetto per poi copiarli nella memoria dell'host, mentre l'operazione inversa viene effettuata con *clEnqueueWriteBuffer(...)*. Salta subito all'occhio la parola enqueue all'interno dei nomi delle funzioni e infatti come ogni funzione eseguita su un device, verranno inserite nella coda dei comandi. Proprio il primo parametro delle funzioni appena viste indica la coda nella quale devono essere inserite e fra gli altri numerosi parametri troviamo l'oggetto di destinazione, se la chiamata è bloccante o meno e la dimensione. In maniera analoga funzionano *clEnqueueReadImage(...)* e *clEnqueueWriteImage(...)*. Si possono anche copiare regioni di memoria di oggetto in un altro con la funzione *clEnqueueCopyBuffer(...)* ma solo ed esclusivamente se appartenenti allo stesso context.

4.8 Lancio e sincronizzazione Kernel

Un oggetto kernel include una funzione kernel dichiarata tramite il qualificatore *_kernel* ed i relativi argomenti. Un oggetto kernel deve essere sempre contenuto in un programma, quindi non può essere creato se non è prima stato costruito un programma. Al momento della compilazione, il codice dei kernel all'interno di un programma viene diviso in parti, che vengono date in pasto a compilatori diversi in base al microprocessore che dovrà poi eseguirle. La stessa cosa accade in CUDA come visto nel capitolo precedente, dove la parte destinata all'host viene compilata dal compilatore CPU, mentre la parte destinata ai device dal compilatore NVCC. Tuttavia in OpenCL lo stesso codice di alto livello, può girare su sistemi differenti, in quanto verrà compilato ad hoc per l'hardware presente. Quando un programma viene lanciato, i kernel vengono messi nelle rispettive code d'esecuzione con il comando *clEnqueueNDRangeKernel(...)* in cui vengono specificati la coda di destinazione e il kernel da lanciare. I kernel verranno eseguiti in suc-

cessione ogni volta che il relativo device sarà disponibile. Questo avverrà in maniera totalmente asincrona, in quanto il programma si è limitato a inserire i kernel nelle code e non può sapere in che ordine i kernel su due code diverse verranno serviti. Per sincronizzare i kernel è necessario usare delle chiamate bloccanti, oppure eventi che traccino lo stato delle esecuzioni. La maggior parte delle funzioni in OpenCL possono restituire eventi e questi possono essere inseriti in code di eventi specifiche chiamate *Waitlists*. L'ordine delle code di esecuzione dei vari dispositivi è quindi subordinato alle waitlists che sincronizzano le varie esecuzioni. Tutti i comandi del tipo *clEnqueue...(..., num_events_in_waitlist, *event_waitlist, *event_out)* hanno questi tre argomenti dove il primo indica un numero di eventi nella waitlist che bisogna attendere, il secondo la waitilist di riferimento e per ultimo l'evento generato. Se un kernel per esempio necessita dei dati che deve generare un'altro kernel in un'altra coda d'esecuzione, attenderà un evento generato da quest'ultimo prima di inizare l'esecuzione.

4.9 OpenCL vs CUDA

OpenCL è un API standardizzata cross-platform per lo sviluppo di applicazioni che sfruttano il calcolo parallelo in sistemi eterogenei. Le somiglianze con CUDA sono notevoli, a partire dalla gerarchia della memoria, alla corrispondenza diretta fra threads e work-items. Anche a livello di programmazione si trovano molti aspetti simili, ed estensioni con le stesse funzionalità. Tuttavia OpenCL ha un modello di gestione dei device molto più complesso, dovuto alla sua capacità di supportare un'ampia varietà di hardware. D'altronde OpenCL è stato progettato per avere portabilità di codice fra prodotti di diversi produttori, e questa qualità incide parecchio sulla complessità. CUDA grazie alla sua maggior maturità e all'hardware dedicato, offre una gestione dei device semplificata ed API di più alto livello che lo rendono preferibile, ma se e solo se si ha a che fare con architetture specifiche. Riassumendo:

Punti di forza di OpenCL:

- Permette di utilizzare sistemi eterogenei con diversi tipi di microprocessori
- Lo stesso codice gira su sistemi diversi

Svantaggi di OpenCL:

- Gestione dei devices complessa
- API ancora giovane

Punti di forza di CUDA:

- API con livello di astrazione molto alto
- ha estensioni per molti linguaggi di programmazione
- documentazione abbondante e community molto vasta

Svantaggi di CUDA

- supporta come device esclusivamente GPU NVIDIA di ultima generazione
- l'eterogeneità è ridotta a CPU e GPU

Capitolo 5

Testing

In questo capitolo si sono effettuati dei test comparativi di performance fra CPU e GPU.

5.1 Sistema di prova

Per effettuare i test è stata usata la seguente macchina:

- CPU: Intel I7-3770k
- GPU: NVIDIA GeForce GTX 660 Ti
- RAM: Corsair Vengeance DDR3 PC12800 1600MHz 16Gb
- Hard Disk: Sandisk Extreme SSD 128Gb
- Scheda Madre: Asrock Extreme 4 Z77
- Sistema Operativo: Windows 8 64bit

Per lo sviluppo software sono stati utilizzati l'SDK CUDA 5 e Microsoft Visual Studio 2012 Express. Sia la CPU che la GPU sono di fascia prestazionale simile in modo da avere una comparativa equa.

5.2 Descrizione del test

Per capire quanto l'utilizzo di una GPU possa avvantaggiare la computazione parallela, si è preso come metro di misura, il tempo di computazione di una semplice operazione matematica: il prodotto di due matrici con elementi espressi in virgola mobile. Per effettuare la comparativa, la stessa operazione è stata implementata in due codici distinti, uno scritto in C# che si avvale del solo utilizzo della CPU, mentre il secondo è scritto in CUDA-C ed utilizza la GPU per il calcolo. I test sono stati eseguiti su matrici di dimensione variabile per verificare la variazione nella differenza delle prestazioni.

5.3 Implementazione

Di seguito viene riportato il frammento del codice destinato ad essere eseguito sulla sola CPU, per brevità è mostrata solo la parte relativa alla computazione:

```
...

public void MultiplyMatrix()
  {
  //Controllo delle dimensioni delle matrici
    if(a.GetLength(1)==b.GetLength(0))
    {

    //Start del timer
    tim_num = 0;
    timer_start = true;

      c=new float[a.GetLength(0),b.GetLength(1)];
      for(int i=0;i<c.GetLength(0);i++)
      {
        for(int j=0;j<c.GetLength(1);j++)
        {
          c[i,j]=0;
          for(int k=0;k<a.GetLength(1);k++)
          c[i,j]=c[i,j]+a[i,k]*b[k,j];
        }
      }
      timer_start = false;
    }
  }

...
```

A, *b* e *c* sono array di float dichiarati in precedenza, è presente un timer per calcolare il tempo della computazione ed una semplice funzione che effettua la classica moltiplicazione riga per colonna di due matrici.

Qui è mostrato il codice CUDA destinato alla computazione eterogenea:

```
...

__global__ void matrixMulCUDA(float *C, float *A, float *B, int wA, int wB)
{

...

//Indice dei block
int bx = blockIdx.x;
int by = blockIdx.y;

//Indice dei thread
int tx = threadIdx.x;
int ty = threadIdx.y;

//Indice della prima sottomatrice di A processata dal blocco
int aBegin = wA * BLOCK_SIZE * by;

//Indice dell'ultima sottomatrice di A processata dal blocco
int aEnd = aBegin + wA - 1;

//Passo utilizzato per scorrere le sotto-matrici di A
int aStep = BLOCK_SIZE;

//Indice della prima sottomatrice di B processata dal blocco
int bBegin = BLOCK_SIZE * bx;

//Passo utilizzato per scorrere le sotto-matrici di B
int bStep = BLOCK_SIZE * wB;

//Csub viene utilizzato per contere la computazione della sottomatrice
float Csub = 0;


...

//Esecuzione del calcolo
for (int k = 0; k < BLOCK_SIZE; ++k)
{
  Csub += As[ty][k] * Bs[k][tx];
  }

...

//Le sottomatrici vengono riassemblate per avere il risultato finale
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
```

Il codice è una versione leggermente modificata del sample fornito con il CUDA SDK, MatrixMul. Anche qui per brevità sono stati omessi vari passaggi. A e B sono le matrici di input e come vediamo, sono prima organizzate in sottomatrici per predisporre la computazione parallela e poi vengono riassemblate. *As* e *Bs* sono sono matrici nella memoria condivisa, dove vengono caricate le relative sottomatrici.

5.4 Test e risultati

La prima prova è stata effettuata calcolando la moltiplicazione di due matrici quadrate da 160*160 con i seguenti tempi di computazione:

- CPU: 4msec
- GPU: 0,2msec
- rapporto CPU/GPU: 20

La seconda prova su due matrici quadrate 320*320 con i seguenti risultati:

- CPU: 46msec
- GPU: 2msec
- rapporto CPU/GPU: 23

La terza prova su due matrici quadrate 480*480:

- CPU: 154msec
- GPU: 6msec
- rapporto CPU/GPU: 25

La quarta prova su due matrici quadrate 640*640:

- CPU: 440msec
- GPU: 13msec

- rapporto CPU/GPU: 34

L'ultima prova è effettuata prima su due matrici rettangolari 320*640 e poi su due matrici rettangolari 640*320 edi risultati sono molto interessanti:
Caso 1

- CPU: 92msec
- GPU: 7msec
- rapporto CPU/GPU: 13

Caso 2

- CPU: 185msec
- GPU: 4msec
- rapporto CPU/GPU: 46

Purtroppo le computazioni di matrici più grandi hanno provocato l'arresto dei driver video a causa di un errore sconosciuto.

Senza dubbio la computazione in parallelo beneficia drasticamente dell'architettura manycore della GPU, che ottiene risultati anche 30 volte migliori dello stesso calcolo effettuato sulla CPU. Inoltre è evidente il comportamento sempre migliore della GPU rispetto alla CPU, con l'aumentare della quantità dei calcoli e quindi del parallelismo. L'ultima prova riflette chiaramente questo andamento, infatti nel primo caso con la prima matrice avente un numero di colonne maggiore delle righe, si ha un parallelismo inferiore in quanto si eseguono pochi lunghi calcoli. Nel secondo caso invece si ha la situazione opposta, ovvero molti calcoli ma più semplici e qui la potenza della GPU si fa sentire.

Capitolo 6

Conclusioni

La situazione

CPU e GPU hanno architetture e funzionamenti molto diversi, ed entrambe eccellono in diversi contesti. Ovviamente le CPU sono state e continuano ad essere le principali unità di computazione di qualsiasi sistema, ma la continua evoluzione delle GPU sta rivelando il grande aiuto che questi processori possono portare nei calcoli complessi. Le GPU sicuramente non possono sostituire le CPU, ma un codice eterogeneo ben strutturato, in grado di sfruttare i punti di forza di entrambi i tipi di processori, può portare vantaggi notevoli.

Le CPU grazie ai set di istruzioni più completi possono eseguire una varietà di task enorme al contrario delle GPU che hanno set ridotti più specializzati. Inoltre le cache sempre più grandi e più veloci permettono alle CPU di eseguire operazioni sequenziali con latenze molto basse. Le GPU, essendo concepite come elaboratori grafici, solo negli ultimi anni hanno espanso i loro set di istruzioni per supportare l'elaborazione di funzioni generiche, ma la distanza dalle CPU è ancora elevata. Anche nella latenza le GPU soffrono a causa della cache limitata, il che ne fa delle pessime unità per la computazione sequenziale. Tuttavia le GPU grazie alla moltitudine di core disponibili e al throughput elevato, riescono ad avere performance notevoli nel calcolo parallelo e grazie alle istruzioni SIMD, riescono ad eseguire operazioni complesse in un numero di clock inferiore rispetto alle CPU. In presenza di un numero ingente di calcoli in parallelo, come l'esempio della moltiplicazione di matrici visto nel capitolo precedente, l'utilizzo della GPU può portare ad incrementi prestazionali notevoli, anche nell ordine di decine

di volte, ma dipendentemente dall'abilità dello sviluppatore e dal campo di applicazione, questo valore può ridursi o elevarsi anche drasticamente.

Esistono campi d'applicazione che sono particolarmente adatti per l'implementazione su GPU, in quanto un programma per sfruttare al meglio i suoi punti di forza dovrebbe avere le seguenti caratteristiche:

- grosse quantità di calcolo parallelo
- necessità di un throughput elevato
- esecuzione del codice lineare con meno salti possibili fra i threads
- operazioni matematiche complesse

Fra i maggiori esempi d'applicazione troviamo:

- algoritmi di ricerca
- simulazioni fisiche complesse
- rendering real-time della fisica nei videogiochi

Attualmente gli sviluppatori hanno a disposizione principalmente due framework: Nvidia CUDA e OpenCL. Il primo è uno standard proprietario vincolato ad essere utilizzato solo su hardware NVIDIA, tuttavia per questo gode di miglior ottimizzazione. OpenCL è uno standard aperto in grado funzionare non solo su CPU e GPU ma anche su altri tipi di processori come DSP e FPGA. Il supporto ad una categoria di hardware più vasta purtroppo ha un prezzo e OpenCL lo paga in termini di complessità del codice. A livello prestazionale le due tecnologie si equivalgono, ed il livello di maturità aumenta costantemente, quindi solo in base al contesto si può decidere qualle delle due sia meglio utilizzare.

Il futuro

Dal trend degli ultimi anni rappresentato in figura 6.1, si evince che le GPU incrementeranno sempre maggiormente il divario con le CPU per quello che riguarda la capacità di calcolo. Inoltre il sempre maggior interesse degli sviluppatori verso la computazione eterogenea, sicuramente favorirà l'orientamento delle architetture delle GPU al GPGPU, che anno dopo anno stanno

già facendo progressi non indifferenti. Un miglioramento delle performance della cache ed un set di istruzioni più esteso, renderebbe le GPU idonee per l'esecuzione di molte più operazioni e soprattutto si potrebbero vedere risultati tangibili anche per calcoli in parallelo meno esosi. Nel prossimo futuro si parla anche di CPU manycore, tuttavia considerando che la proporzione attuale del numero di core è di circa 300:1 in favore delle GPU, ed è in aumento, è lecito presumere che il trend non cambi. Dal lato software sia CUDA che OpenCL sono in costante crescita e da poco sta acquisendo visibilità Microsoft DirectCompute, un'API per lo sviluppo general purpose integrata in DirectX. Non si può escludere che nuovi framework si affianchino a quelli già presenti, arricchendo ulteriormente la collezione di strumenti a disposizione degli sviluppatori.

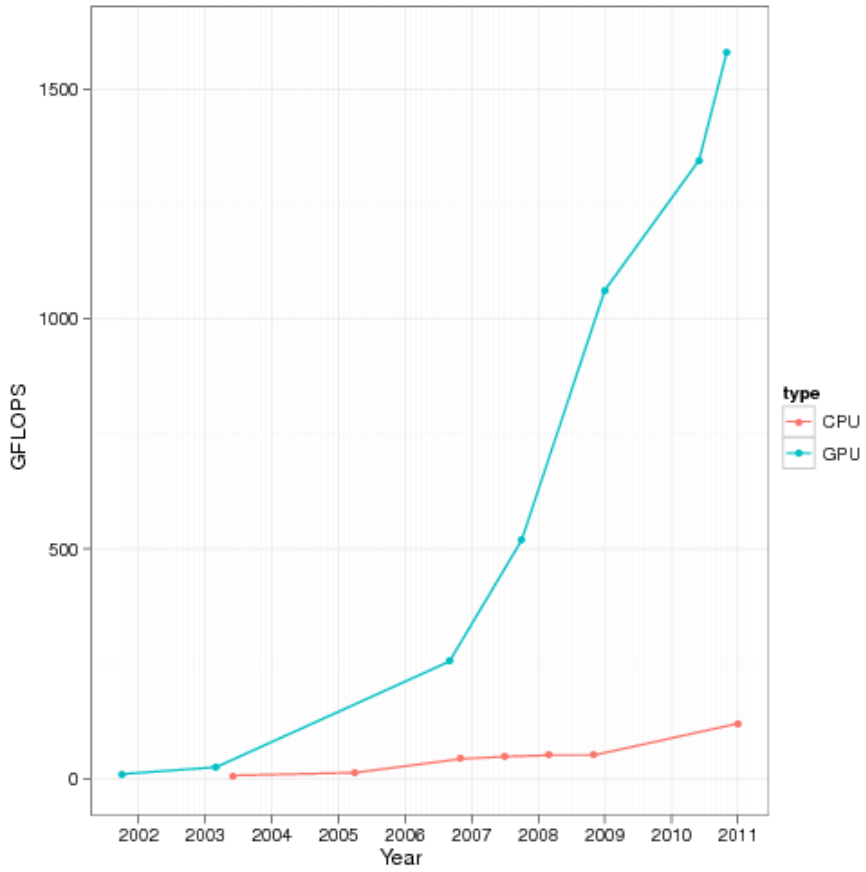


Figura 6.1: Trend negli anni della capacità di calcolo di CPU e GPU

Bibliografia

[1] J. H. D. Patterson. Computer organisation and design. Morgan Kaufmann, 2008.

[2] Gartner. Gartner says spending on gaming to exceed 74 billion dollars in 2011, 2011. `http://www.gartner.com/newsroom/id/1737414`.

[3] Intel. Intel, 2013. `http://www.intel.com`.

[4] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[5] Nvidia. Nvidia, 20123. `http://www.nvidia.com`.

[6] Wikipedia. Opencl, 2013. `http://www.wikipedia.org`.

Ringraziamenti

Prima di tutti ringrazio i miei genitori che mi hanno dato supporto in tutti questi anni, nonostante la scarsa fiducia che ormai riponevano nella realizzazione di questo traguardo.

Ringrazio il professor Ricci, che mi ha permesso di realizzare questa tesi molto stimolante ed interessante (non lo scrivo per fare il ruffiano sia ben chiaro).

Un sentito grazie alla Ua Ua Family, un gruppo di amici preziosissimo, che mi ha accompagnato costantemente in questi anni, a partire da Lorenzo, un amico, quasi un fratello, sicuramente una delle figure più importanti e presenti, con cui ho condiviso veramente tanto. Grazie a Maria per il suo sorriso costante e per aver sempre apprezzato tutti gli intrugli che ho cucinato. Grazie ad Antonello, fedele compagno di bolgia e mangiate sempre presente in ogni occasione, di festa e non. Grazie a Federico che ha sicuramente stimolato la mia creatività con le sue invenzioni meccanico-culinarie, dal trapano-frullatore alla pisellata. Grazie a Carla per le sue invenzioni culinarie non meccaniche. Grazie a Cristiano con cui ho condiviso studio ma soprattutto nerdate di ogni sorta. Grazie a Chiara per avermi sempre dato supporto morale in ogni occasione ed i babà al limoncello. Grazie ad Alice anche lei sempre presente, soprattutto se c'è qualcosa di fumante in pentola. Grazie a Marcello le orecchie di tutti i macchinisti fischiano ogni giorno. Grazie a Tany perchè 1000km non possono cancellare le belle esperienze passate insieme, e nemmeno il nostro primogenito: le Birriadi. Grazie ad Alex compagno di grandi riflessioni, sale e limone. Grazie a Riccardo ho scoperto che le mini-palle da basket sono strepitose. Grazie a Barbara ho scoperto che Riccardo si può sopportare. Grazie ad Endi fedele compagno in aula Beta, di pause caffè. Grazie a Luis che anche a 70 anni farà apparire Cesena ad ogni matricola come un luna park. Grazie a Dario la Rolex fallirebbe.

Un grande grazie a Patrizia che in questi ultimi mesi mi è stata vicinissima, sempre pronta al confronto e al dialogo nei momenti di necessità e non.

Grazie alle mie coinquiline Cecilia e Irene per avermi sostenuto e sopportato.

Grazie a Marco il mio primo vero amico a Cesena, con il quale ho iniziato l'università e finito tanto caffettone.

Grazie ad Ana, amica di sempre in momenti belli e brutti.

Grazie a Francesco, compagno di banco dalle superiori fino all'università

Grazie a Biste, Sergio, Ghetti, Balza e Roby perchè la pancetta...non è un optional.

Grazie anche a tutte le persone che ho conosciuto in questi anni, che a loro modo hanno contribuito al mio miglioramento.

Grazie a Cesena, una città che mi ha dato e a cui ho dato tanto.

E infine ringrazio me, per essere riuscito a trovare tanta determinazione dopo tanti anni.