Università della Calabria

Dipartimento di Matematica e Informatica

Corso di Laurea in Informatica

Tesi di laurea magistrale

Parallelizzazione CUDA della libreria per automi cellulari OpenCAL

Relatori:

Prof. Donato D'Ambrosio

Prof. William Spataro

Tesista:

Carmelo La Gamba

Matricola 160252

Anno accademico 2013 - 2014

Dedica

Indice

Elenco delle figure

Elenco delle tabelle

Sommario

In questo lavoro di tesi ho progettato e implementato una versione parallela della libreria per automi cellulari OpenCAL.
OpenCAL, si propone di facilitare l'implementazione di sistemi complessi basati su automi cellulari offrendo funzionalità complete per progettare un modello e simulare la sua evoluzione nel tempo. Il mio lavoro è consistito nella progettazione, e successiva implementazione, della parallelizzazione di OpenCAL utilizzando le schede grafiche per il calcolo general-purpose (General Purpose Computation with Graphics Processing Units - GPGPU), adottando il Compute Unified Device Architecture (CUDA) framework di NVIDIA con lo scopo di migliorare le performance.

TODO

- Risultati ottenuti
- validità della gpgpu programming

Introduzione

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Il primo capitolo offre una visione d'insieme della storia di LaTeX e ne vengono presentate le idee di fondo.

Il secondo capitolo spiega le operazioni, veramente semplici, per installare LaTeX sul proprio calcolatore.

L'appendice A descrive sinteticamente le principali norme tipografiche della lingua italiana, utili nella composizione di articoli, tesi o libri.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Capitolo 1

Il calcolo parallelo

1.1 Introduzione

Sin dalla nascita dei primi calcolatori, la velocità di calcolo è stata sempre oggetto di ricerche e studi ai fini di migliorare le performance. La central processing unit, più comunemente conosciuta come **CPU**, nel corso degli anni è stata migliorata notevolmente, aumentando il potere di calcolo e nello stesso tempo riducendo sempre di più i costi.

L'obiettivo primario dei produttori di CPU è stato quello di aumentare il tasso di esecuzione di FLOPS (floating point operations per second), in modo da poter sviluppare applicazioni in grado di produrre risultati soddisfacenti in tempi brevi. Tuttavia, l'aumento della potenza di calcolo e la velocità hanno incrementato i costi relativi all'energia spesa e la dissipazione di calore dei processori basati su singola CPU. Per questo si è pensata una nuova architettura hardware basata sull'aggiunta di più unità di calcolo (cores), definendo dunque i processori di ultima generazione **multicores**.

Oggi alle comuni CPU si sono affiancate con prepotenza le GPU (graphics processing unit). Inizialmente le GPU erano solamente utilizzate per il rendering grafico, campo che tuttora occupano con ottimi risultati. Si pensi infatti che tutto il mondo dei videogames ad alta definizione è basato sulla potenza di calcolo delle nuove generazioni di GPU sempre più performanti e veloci. Nel corso del tempo però si è pensato di sfruttare la loro potenza di calcolo anche nel mondo del parallel computing e in altri campi quali il clustering, l'audio signal processing e la bioinformatica.

Un altro dato importante da cui dipende la velocità di un calcolatore è la velocità con cui si accede alla memoria. Il gap presente tra la velocità di calcolo e la velocità di accesso alla memoria può influire negativamente sulla performance generale del calcolatore. Dopo diversi studi si è risolto questo problema grazie ad un dispositivo di memoria presente nelle architetture hard-

ware moderne, la **cache**. La cache è una memoria gestita dall'hardware che mantiene i dati utilizzati di recente della memoria principale, grazie a questo suo funzionamento il gap tra la velocità di calcolo e quella di accesso alla memoria si riduce migliorando le performance del sistema.

Sotto questo punto di vista, l'aspetto vincente dei sistemi dotati di più unità di calcolo, è dato dal fatto che ogni core ha in dotazione una memoria cache e dunque si può accedere con più rapidità ai dati utilizzati frequentemente.

1.2 Tassonomia di Flynn

Michael J. Flynn è un ingegnere informatico statunitense, la sua carriera iniziò con lo sviluppo dei primi computer per conto di **IBM**. Flynn nel 1966 pubblicò un articolo scientifico che diede i natali alla tassonomia di Flynn (Fig. 1.1), per poi completarne la pubblicazione nel 1972. La tassonomia di Flynn è una classificazione delle architetture dei calcolatori, prevedendo 4 diverse tipologie di architetture:

SISD (Single Instruction stream Single Data stream): è un sistema monoprocessore (architettura di Von Neumann) con un flusso di istruzioni singolo e un flusso di dati singolo

SIMD (Single Instruction stream Multiple Data stream): è un architettura in cui tante unità di elaborazione eseguono contemporaneamente la stessa istruzione lavorando però su insiemi di dati differenti.

MISD (Multiple Instruction stream Single Data stream): è un architettura in cui tante unità di elaborazione eseguono contemporaneamente diverse istruzioni operando però su un insieme di dati singolo.

MIMD (Multiple instruction stream Multiple Data stream): è un architettura in cui tante unità di elaborazione eseguono contemporaneamente diverse istruzioni operando su più insiemi di dati.

I computer attualmente in commercio sono basati sull'architettura di Von Neumann (SISD), cioè un architettura in cui non è presente nessun tipo di parallelismo e le operazioni vengono eseguite sequenzialmente su un flusso di dati singolo. Sia le architetture SIMD (Single Instruction stream Multiple Data stream) che le architetture MIMD (Multiple instruction stream Multiple Data stream) descritte in precedenza, si basano sulla filosofia parallela.

Una sottocategoria delle architetture MIMD (Multiple instruction stream Multiple Data stream) è l'architettura SPMD (Single Program Multiple Data). La sua tecnica è programmata per raggiungere il parallelismo. Si tratta di lanciare più istanze dello stesso programma su diversi insiemi di dati.

Le GPU (graphics processing units), richiamate in precedenza, sono l'esempio di architetture SIMD, mentre i processori più comuni sono un esempio di architettura MIMD.

1.3 Modelli di comunicazione

Tra le basi del parallelismo esiste l'opportunità di far comunicare i diversi *tasks* paralleli. Esistono due forme diverse di comunicazione:

- accesso ad uno spazio di memoria condivisa
- scambio di messaggi

1.3.1 Memoria condivisa

Questo tipo di architetture fanno si che tutte le unità di calcolo presenti accedono allo stesso spazio di memoria. I cambiamenti eseguiti da una singola unità di calcolo devono essere visibili anche dalle altre unità di calcolo. Possiamo distinguere due diversi tipi di accesso alla memoria:

- UMA (Uniform Memory Access): tutti i processori accedono allo spazio di memoria condivisa allo stesso tempo. In questo caso l'hardware deve assicurare la coerenza della cache in modo tale che tutte le unità di calcolo possano vedere le modifiche eseguite dagli altri processori, così da evitare accessi ai dati non aggiornati. Questo meccanismo è chiamato ***cache coherence***. (Fig.1.2a)
- NUMA (Non Uniform Memory Access): tutti i processori possono accedere alla loro memoria locale in modo estremamente rapido, tuttavia accedono più lentamente alla memoria condivisa e alla memoria degli altri processori. Anche in questo caso troviamo il meccanismo di *cache coherence* per garantire l'accesso coerente ai dati in memoria. (Fig. 1.2b)

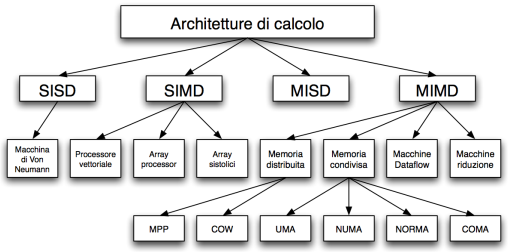
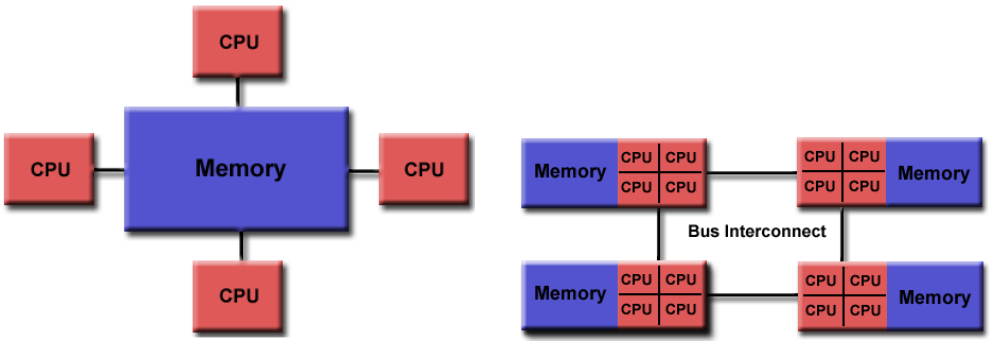


Figura 1.1: La tassonomia di Flynn



(a) UMA. (b) NUMA.

Figura 1.2: UMA e NUMA.

Grazie alla presenza di memorie condivise, risulta molto semplice programmare algoritmi paralleli. Tuttavia ci sono dei punti critici da gestire, come ad esempio il meccanismo di lettura e scrittura. Per quanto riguarda il meccanismo di lettura, può avvenire in modo del tutto trasparente poiché non apporta inconsistenze nella memoria condivisa, ciò non accade per la scrittura, dove si ha bisogno di ulteriori meccanismi per l'accesso **esclusivo**. I paradigmi che supportano il modello di comunicazione a memoria condivisa (e.g. POSIX threads, OpenMP) forniscono strutture per la sincronizzazione come *lock, barriere, semafori* e così via.

1.3.2 Memoria distribuita

Le architetture a memoria condivisa prevedono diverse unità di calcolo, ognuno dei quali possiede un proprio spazio di memoria. Le unità di calcolo possono essere composte da un singolo processore o da un sistema multiprocessore con uno spazio di memoria condiviso. I processi in esecuzione comunicano attraverso uno scambio di messaggi. Grazie a questa interazione, i processi possono scambiarsi dati, assegnare task e sincronizzare i processi. L'architettura MIMD viene supportata da questo modello di comunicazione, ma nella maggior parte dei casi, le implementazioni basate sullo scambio dei messaggi sono implementati con l'approccio SPMD.

Le operazioni di base che un processo può eseguire sono l'invio e la ricezione dei messaggi. Nello scambio di messaggi è necessario anche specificare chi è il mittente e chi il destinatario del messaggio, per questo il sistema offre un meccanismo di assegnazione di un ID univoco ad ogni processo, in modo da distinguerlo da tutti gli altri. Altre funzionalità presenti in questo paradigma sono il *whoami* e il *numProc*. Il primo permette ad ogni processo di conoscere il proprio ID univoco, mentre il secondo consente ad ogni processo di conoscere il numero di processi in esecuzione.

Oggi ci sono diversi framework che consentono lo scambio di messaggi. Uno di questi è MPI (Message Passing Interface) che supporta tutte le operazioni citate in precedenza.

1.3.3 Sistemi ibridi

Le architetture basate sui sistemi ibridi non è nient'altro che un mix delle due architetture viste in precedenza. Immaginiamo di avere un numero N di processi. Solo un sottoinsieme di processi avranno accesso alla memoria condivisa. Per accedervi possono utilizzare ad esempio un paradigma di programmazione parallela a memoria condivisa (e.g OpenMP). Ogni processo che ha accesso alla memoria condivisa, può comunicare i dati tramite il paradigma del Message Passing agli altri processi che non vi hanno accesso. In questo modo entrano in gioco le due diverse architetture sfruttando i vantaggi di entrambe.

1.4 Progettazione di un algoritmo parallelo

Fino ad ora si è descritto in modo generico le strutture, le basi dei paradigmi e le architetture per sistemi paralleli ma la progettazione di un algoritmo parallelo è la parte che interessa di più un programmatore. Progettare un algoritmo parallelo implica uno studio totalmente diverso dalla progettazione di un algoritmo sequenziale. Come abbiamo già visto, entrano in gioco diverse operazioni per raggiungere l'output desiderato. Molte guide di calcolo parallelo evidenziano le seguenti problematiche per la progettazione di un algoritmo parallelo *nontrivial*:

- Identificazione della porzione di lavoro che può essere eseguita concorrentemente.
- Mapping dei task su più processi in parallelo
- Assegnare i dati relativi al programma.

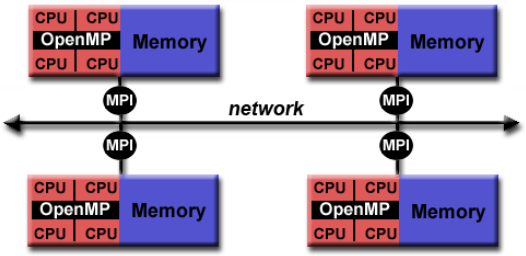


Figura 1.3: Esempio di sistema ibrido.

- Gestire gli accessi alla memoria condivisa
- Sincronizzare le unità di calcolo durante l'esecuzione.

Di solito ci sono diverse scelte da fare durante la progettazione, ma spesso si possono prendere decisioni progettuali anche basandosi sull'architettura a disposizione o in base al paradigma di programmazione utilizzato.

1.4.1 Tecniche di decomposizione

La decomposizione è il processo di dividere la computazione in piccole parti che potenzialmente possono essere eseguite in parallelo. I task sono unità di computazione nei quale la computazione principale viene suddivisa. Ci sono casi in cui alcuni task per poter iniziare la propria attività hanno bisogno dell'output di altri task, così da formare una relazione di dipendenza. Questo genere di relazione di dipendenza nel parallel computing viene rappresentata dal ***task-dependency graph***. Il grafo delle dipendenze è un grafo diretto e aciclico nel quale ogni nodo rappresenta un task e gli archi rappresentano la dipendenza tra i nodi. Quest'ultimo risulterà molto utile nei casi in cui si debbano prendere alcune scelte di progettazione dell'algoritmo, in particolare fornirà informazioni importanti sulla strategia da utilizzare per la suddivisione dei tasks. Un altro importante concetto per la suddivisione dei task è la **granularità**. Distinguiamo due tipi di granularità:

Suddivisione a granularità fine quando la decomposizione produce un numero consistente di task ma di piccola dimensione.

Suddivisione a granularità grossa quando la decomposizione produce un basso numero di task ma di grande dimensione.

Il numero di task che possono essere eseguiti in parallelo invece è detto **grado di concorrenza**.

Gli esempi più comuni di suddivisione dei task è rappresentato dai calcoli eseguiti su matrici. Supponiamo di avere a disposizione 4 unità di calcolo, e il task principale da eseguire è una semplice somma di tutte le celle della matrice. Possiamo decomporre la nostra matrice in 4 parti uguali (se è possibile), e assegnarne una per ogni processo a disposizione. Ipoteticamente l'algoritmo sarà 4 volte più veloce rispetto alla versione sequenziale.

Spesso anche il fattore di interazione tra i processi è un dato da non sottovalutare in una buona progettazione di un algoritmo parallelo. Come nel caso della fig. 1.4 tutti i task hanno bisogno di accedere all'intero vettore b, e nel caso in cui si ha una sola copia del vettore, i task devono obbligatoriamente

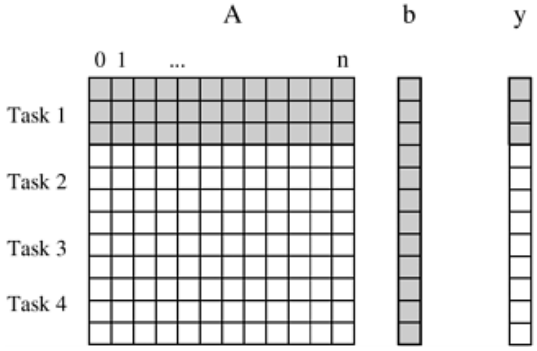


Figura 1.4: Suddivisione di una moltiplicazione tra una matrice e un vettore in 4 diversi task.

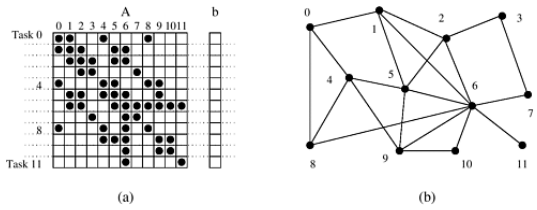


Figura 1.5: Esempio di un grafo delle interazioni tra i task.

iniziare a comunicare tra di loro tramite messaggi per accedere alle informazioni. Questa relazione tra i task viene rappresentata da un altro grafo: il ***task-interaction graph***.

L'interazione tra task è un fattore che limita molto la speedup di un algoritmo parallelo.

Vediamo insieme ora le cinque differenti tecniche di decomposizione.

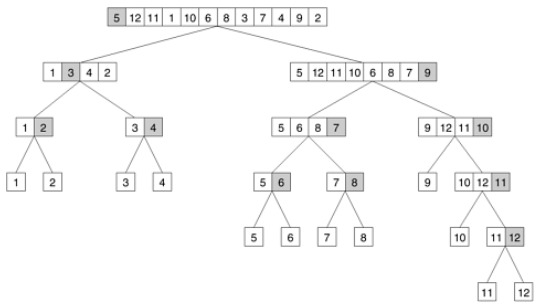


Figura 1.6: Esempio di decomposizione ricorsiva: il quicksort.

Decomposizione ricorsiva

La decomposizione ricorsiva è una tecnica per applicare la concorrenze in problemi che possono essere risolti tramite la strategia del divide-et-impera. La prima divisione consiste nel dividere il problema principale in sottoproblemi indipendenti. Ognuno dei sottoproblemi generati viene risolto ricorsivamente applicando la stessa tecnica.

Decomposizione dei dati

La decomposizione dei dati è una tecnica che può essere applicata seguendo diversi approcci.

- Partizione dell'output dei dati: si sceglie questa tecnica nel caso in cui gli output possono essere calcolati indipendentemente uno dall'altro, senza aver bisogno di rielaborare il risultato finale. Ogni problema viene suddiviso in task, dove ad ognuno viene assegnato il compito di calcolare esattamente una porzione di output. (Fig. 1.7)

- Partizione dell'input dei dati: si sceglie questa tecnica nel caso in cui il risultato atteso è un dato singolo (eg. minimo, somma tra numeri). Si creano task per ogni partizione dell'input, ed ognuno di loro proseguono nella computazione nel modo più indipendente possibile. E' quasi sempre necessario dunque ricombinare i risultati alla fine della computazione.

Decomposizione esplorativa

La decomposizione esplorativa è una tecnica utilizzata per decomporre problemi nei quali per trovare la soluzione viene generato uno spazio di ricerca. Lo spazio di ricerca è suddiviso in diverse parti e in ciascuna di queste in parallelo si cerca la soluzione. Quando un processo trova la soluzione, tutti gli altri processi si interrompono.

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

(a)

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$
Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$
Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

(b)

Figura 1.7: Esempio di decomposizione dei dati.

Decomposizione speculativa e ibrida

La decomposizione speculativa è usata quando un programma può prendere diverse scelte che dipendono dall'output dello step precedente. Un esempio lampante è il caso dell'istruzione *switch* in C, prima che l'input per lo switch sia arrivato. Mentre un task computa un ramo dello switch, gli altri task in parallelo possono prendere a carico gli altri rami dello switch da computare. Nel mondo in cui l'input arriva allo switch viene preso in considerazione solamente il ramo corretto mentre gli altri possono essere scartati.

La decomposizione ibrida invece, si occupa di combinare diverse tecniche ai fini di migliorare le performance ulteriormente. E' struttrata in più step, dove per ogni step si applica una tecnica di decomposizione diversa.

1.4.2 Tecniche di mapping

Una volta decomposto il problema in task, c'è la necessità di creare un mapping tra i task e i processi. Il mapping è una fase molto importante e delicata ai fini di una buona performance. L'obiettivo da raggiungere è minimizzare in modo consistente l'overhead che si crea nell'esecuzione dei task in parallelo. Tra le principali fonti di **overhead** troviamo l'interazione tra i processi durante il periodo di esecuzione e il tempo in cui diversi processi non effettuano nessuna operazione. Frequentemente, per limitare la comunicazione tra i processi, nel caso in cui ci troviamo di fronte a task di piccole dimensioni, si può scegliere di accorpare più task assegnandole ad un unico processo. Questa può sembrare una scelta logica, a volte potrebbe anche essere la scelta corretta ma, creare un processo più corposo di un altro potrebbe scalfire il *load balancing*.

Proprio per questo la scelta di un corretto mapping potrebbe contrastare questo genere di problematiche, così da diventare determinante ai fini del raggiungimento di una buona performance. Distinguiamo due tipi di tecniche di mapping:

- Mapping statico
- Mapping dinamico

Descriviamo brevemente i due differenti approcci.

La tecnica di mapping statico assegna i task ai processi prima dell'inizio di esecuzione dell'algoritmo. In genere questa tecnica è utilizzata quando l'euristica dei task non è computazionalmente costosa, dunque gli algoritmi sono più facili da progettare e implementare.

La tecnica di mapping dinamico invece distriuisce il lavoro durante l'esecuzione del programma. Scegliamo questa tecnica quando la dimensione dei task è sconosciuta e non si possono prevedere dunque le possibilità per un mapping ottimale.

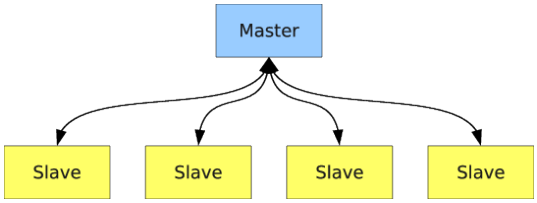


Figura 1.8: Il modello Master-Slave.

1.4.3 Modelli di un algoritmo parallelo

In questo paragrafo si mostreranno i differenti modelli utilizzati per implementare un algoritmo parallelo.

Dati in parallelo E' il più sempice dei modelli. Questo tipo di parallelismo è il risultato di operazioni identiche applicate concorrentemente in diversi elementi di dati. Si può reallizzare questo modello sia con un architettura a memoria condivisa sia utilizzando il paradigma del message-passing.

Task graph E' un modello basato sul concetto del task-dependency graph. A volte il grafo delle dipendenze può essere banale o non banale, e le interazioni tra i processi sono numerose. Questo modello è utilizzato per risolvere i problemi in cui la quantità di dati associata ai task è più grande rispetto alla quantità di calcolo ad essi associato. Un esempio basato sul questo modello comprende il quicksort parallelo come tanti altri algoritmi basati sul divide-et-impera.

Master-Slave E' uno dei più famosi modelli per progettare un algoritmo parallelo. Con questo modello uno o più processi vengono identificati come *master* e hanno il compito di distribuire il lavoro agli altri processi, definiti *slave*. Questo modello può essere accompagnato sia da una memoria condivisa che dal paradigma del message-passing. Spesso si usa questo modello quando si ha bisogno di gestire le diverse fasi di un algoritmo, in particolare per ogni fase un compito del master potrebbe comportare la sincronizzazione di tutti gli slaves. Bisogna essere comunque parsimoniosi se si decide di utilizzare questo modello, poiché può comportare facilmente colli di bottiglia che porterebbero ad una bassa performance.

1.5 Misure di performance

Fino ad ora si è parlato di perfomance, di parallelizzare un algoritmo in modo da renderlo più veloce. Nel parallel computing per definire il concetto di velocità e di performance migliore si utilizzano diverse misure, che analizzano e

permettono di valutare gli algoritmi, le architetture utilizzate e i benefici del parallelismo. Intendiamo misura di perfomance:

- Il tempo di esecuzione
- L'overhead totale
- Lo speedup
- L'efficienza

Andiamo a descrivere ora, il significato di queste misure

Il **tempo di esecuzione** T è il tempo effettivo che passa tra il momento in cui viene lanciato l'algoritmo e il momento in cui termina. Per gli algoritmi paralleli il tempo di esecuzione il tempo che passa tra il momento in cui inizia la computazione parallela fino al momento in cui l'ultimo processore termina la computazione. Questa può essere considerata come una prima valutazione del parallelismo.

L'**overhead** totale nel parallel computing è il tempo di esecuzione impiegato collettivamente da tutti i processori rispetto al tempo richiesto dal più veloce algoritmo sequenziale per risolvere il problema.

$$T_o = pT_p - T_s \tag{1.1}$$

dove p è il numero di unità di calcolo, T_p è il tempo parallelo e T_s è il tempo sequenziale.

Le due misure più importanti tra quelle citate sono la *speedup* e l'*efficienza*. Nel valutare un algoritmo spesso si vuole sapere qual è il guadagno, in termini di performance, di un'implementazione parallela rispetto ad un'implementazione seriale. Lo *speedup* quantifica i benefici nel risolvere un problema in parallelo e può essere definito come il rapporto tra il tempo T_s necessario per risolvere il problema su una singola unità di calcolo e il tempo T_p per risolvere lo stesso problema su un calcolatore parallelo con n identiche unità di calcolo.

$$S = \frac{T_s}{T_p} \tag{1.2}$$

In genere T_s è il tempo di esecuzione del più veloce algoritmo sequenziale conosciuto, in grado di risolvere il problema dato. In teoria, lo speedup non supera mai il numero di unità di calcolo n. Se T_s rappresenta il tempo del miglior algoritmo sequenziale, per ottenere uno speedup pari a n, avendo a disposizione n unità di calcolo, nessuna di esse deve impiegare un tempo maggiore di $\frac{T_s}{n}$. Uno speedup maggiore di n è possibile solo se tutte le unità di calcolo hanno un tempo di esecuzione minore di $\frac{T_s}{n}$. In questo caso una singola unità di calcolo potrebbe emulare le n unità di calcolo e risolvere il problema con un

tempo minore di T_s. Questa è una contraddizione poiché T_s è il tempo di esecuzione del miglior algoritmo sequenziale. In pratica, è però possibile avere uno speedup maggiore di n (speedup superlineare). Generalmente questo è dovuto a caratteristiche dell'hardware che mettono l'implementazione sequenziale in svantaggio rispetto a quella parallela. Ad esempio, è possibile che la cache di una singola unità di calcolo non sia abbastanza grande da contenere tutti i dati da elaborare, quindi, le sue scarse prestazioni sono dovute all'utilizzo di una memoria con un accesso lento rispetto a quello della memoria cache. Nel caso dell'implementazione parallela i dati vengono partizionati e ogni parte è abbastanza ridotta da entrare nella memoria cache dell'unità di calcolo alla quale è stata assegnata. Questo spiega come in pratica sia possibile avere uno speedup superlineare.

L'*efficienza* è una misura di prestazione legata allo speedup. Come menzionato precedentemente, la parallelizzazione di un'algoritmo introduce un overhead dovuto alla comunicazione tra i processi e ai processi che entrano in uno stato di idling. Per questo motivo è molto difficile raggiungere uno speedup pari al numero di unità di calcolo. L'efficienza quantifica la quantità di lavoro utile (tralasciando i tempi dovuti a overhead) effettuato dalle n unità di calcolo ed è definita come il rapporto tra lo speedup e n.

$$E = \frac{S}{n} \tag{1.3}$$

1.6 Linguaggi di programmazione

Esistono diversi linguaggi di programmazione e paradigmi di programmazione che consentono l'utilizzo del parallel computing durante l'implementazione di un algoritmo. Tra i più utilizzati troviamo sicuramente OpenMP e MPI. Nel prossimo paragrafo vedremo sommariamente come funziona OpenMP.

1.6.1 OpenMP

OpenMP è uno standard che offre funzionalità per creare algoritmi paralleli in uno spazio di memoria condiviso. Supporta dunque la concorrenza, la sincronizzazione e altre funzionalità utili per una corretta implementazione di un algoritmo parallelo su memoria condivisa. OpenMP per la sua semplicitè è molto usato, e qualche volta riesce a raggiungere risultati ottimi con speedup interessanti. Il suo utilizzo si basa sulla dichiarazione della seguente direttiva:

#pragma omp directive [clause list]

Il programma si esegue sequenzialmente finché non trova la direttiva ***parallel***. Questa direttiva è responsabile della creazione di un gruppo di *threads* che

devono eseguire in parallelo l'algoritmo. Il prototipo della direttiva parallel è il seguente:

#pragma omp parallel [clause list]

La lista di clausole è utile per aggiungere gradi di libertà all'utente nell'utilizzo della concorrenza. Ad esempio nel caso in cui la parallelizzazione e la conseguente creazione di più threads in parallelo debba avvenire solo in determinati casi, si può utilizzare la clausola:

if (*espressione*)

In questo caso solo se l'*espressione* è vera si userà la direttiva *parallel*. Un altra clausola utilizzata è

num_threads (int)

Questa specifica il numero di threads che devono essere creati ed eseguiti in parallelo. Nel caso in cui si vogliano utilizzare delle variabili private per ogni thread si può utilizzare la clausola:

private (lista delle variabili).

che specifica la lista delle variabili locali per ogni thread, cioà ogni thread possiede una copia di ognuna di queste variabili specificate in questa clausola. Le clausole che mette a disposizione OpenMP sono molteplici, tra queste troviamo la causola ***reduction(operazione: variabile)*** che come si può intuire applica una particolare operazione aritmetica ad una variabile.

Tra le direttive di OpenMP la più interessante è la direttiva **for**. La forma generale di questa direttiva à:

#pragma omp for [clause list] .
/* ciclo di for */

Questa è utilizzata per dividere lo spazio delle iterazioni parallele attraverso i threads a disposizione.

In generale OpenMP offre veramente decine di funzionalità da poter utilizzare e l'aspetto migliore di questo paradigma è sicuramente la semplicità della sua implementazione e l'integrazione con l'algoritmo parallelo. In ultimo, ecco un semplice esempio di parallelizzazione tramite OpenMP di un algoritmo sequenziale:

```
1 #include <omp.h>
2
3 float num_subintervals = 10000; float subinterval;
4 #define NUM_THREADS 5
5
6 void main ()
```

```
7  {
8          int i; float x, pi, area = 0.0;
9          subinterval = 1.0 / num_subintervls;
10
11         omp_set_num_threads (NUM_THREADS)
12         #pragma omp parallel for reduction(+:area) private(x)
13         for (i=1; i<= num_subintervals; i++) {
14                 x = (i-0.5)*subinterval;
15                 area = area + 4.0 / (1.0+x*x);
16         }
17         pi = subinterval * area;
18 }
```

Codice 1.1: Esempio di utilizzo di OpenMP.

1.7 Nuovi approcci al calcolo parallelo: GPGPU computing

La GPU (Graphics Processing Unit) è un processore grafico specializzato nel rendering di immagini grafiche. Viene utilizzata generalmente come coprocessore della CPU, infatti è tipicamente una componente della CPU in un circuito integrato, ma da alcuni anni la sua potenza di calcolo ha suscitato parecchio interesse nel campo scientifico. Le numerose ricerche hanno portato all'implementazione come circuito indipendente dotato di più *cores*. Sebbene le GPU operino a frequenze più basse rispetto alle CPU sin dai primi anni del nuovo millennio esse superano le CPU nel calcolo di operazioni in floating point (FLOPS) e, ad oggi la velocità di calcolo delle GPU è quasi dieci volte superiore quelle delle CPU. Prima del 2006, le GPU difficilmente venivano usate per scopi diversi dal rendering grafico poiché per accedere a questi dispositivi i programmatori avevano a disposizione soltanto API per la grafica (ad esempio OpenGL [4]). GPGPU (general purpose computing on graphic processing unit) è il termine che viene usato per indicare l'utilizzo delle GPU in ambiti diversi dal rendering grafico. Questa tecnica si diffuse nel 2007 grazie al rilascio di CUDA [2] da parte NVIDIA, che permetteva ai programmatori di sviluppare applicazioni parallele senza utilizzare le API grafiche. Oltre a questo NVIDIA iniziò ad inserire nei propri dispositivi delle componenti hardware apposite a supporto della programmazione parallela.

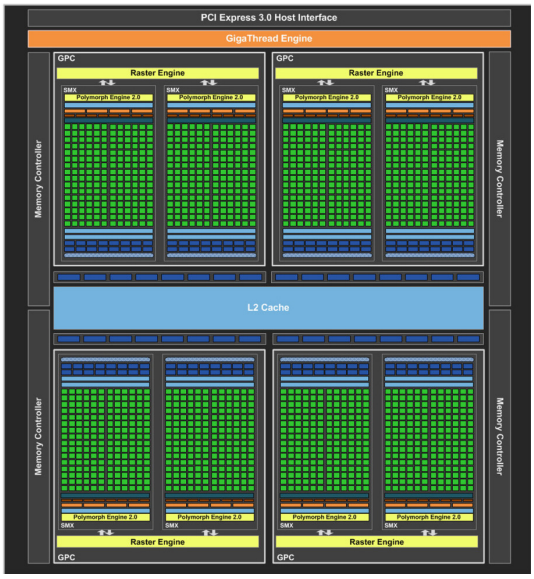


Figura 1.9: Architettura Kepler (GTX 680).

La figura **??** rappresenta l'architettura di una tipica GPU CUDA. La struttura è composta da un insieme di *streaming multiprocessor* (SM) divisi in blocchi (Nella figura **??** ci sono due SM per ogni blocco). Ogni SM è composto da un insieme di *streaming processors* che condividono la memoria cache. Ogni GPU ha a disposizione alcuni gigabytes di Graphic Double Data Rate DRAM, anche detta memoria globale. Questo tipo di memoria è diversa dalla normale memoria DRAM poiché è progettata per contenere dati relativi alla grafica. Nel caso di applicazioni grafiche contiene informazioni relative ad immagini e texture usate per il rendering 3D. In ambito GPGPU viene sfruttata per la sua ampia larghezza di banda al costo di una maggiore latenza rispetto alla normale memoria DRAM. Con l'aumentare della disponibilità di memoria delle GPU prodotte, le applicazioni tendono a memorizzare i dati nella memoria globale minimizzando le interazioni con la memoria del sistema. Le GPU sono particolarmente adatte a risolvere problemi che possono essere modellati con un approccio SIMD (vedi paragrafo 1.2) e che contengono una gran quantità di calcoli aritmetici [3]. Generalmente lo stesso programma viene eseguito per ogni dato da elaborare e la latenza dovuta agli accessi alla memoria globale viene "nascosta" dalla grande quantità di calcoli.

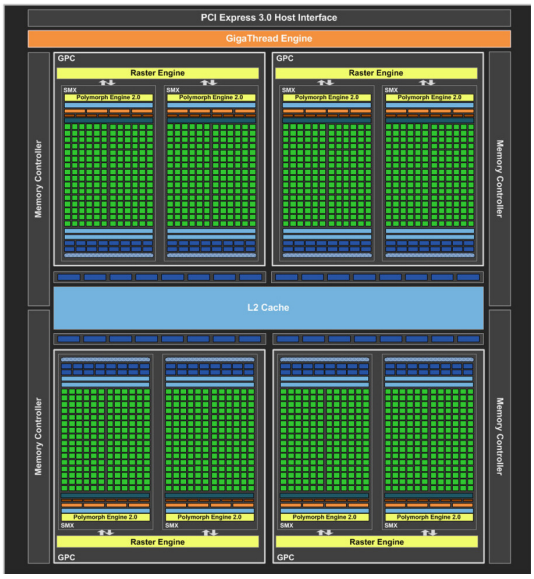


Figura 1.10: Architettura Kepler (GTX 680).

Capitolo 2

CUDA - Compute Unified Device Architecture

2.1 Introduzione

Quasi nove anni fa, nel Novembre 2006 la **NVIDIA Corporation** ha rilasciato CUDA, una piattaforma (hardware e software insieme) che permettono di utilizzare linguaggi di programmazione ad alto livello (Ad es. **C**, **C++**, **Java**) per implementare codice parallelo per risolvere problemi molto complessi a livello computazionale in una maniera efficiente rispetto alle normali CPU.

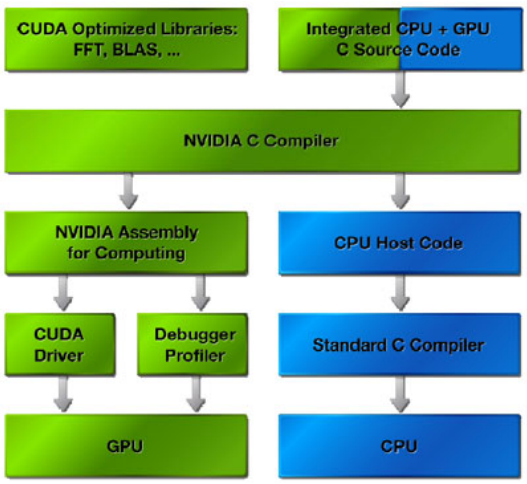


Figura 2.1: Struttura di Nvidia C Compiler.

CUDA è molto utilizzato poiché è un sistema completo e anche molto semplice da capire ed utilizzare. Sopratutto quest'ultimo particolare è di importante rilevanza, dato che attualmente le alternative a CUDA, come OpenCL, risultano essere molto più complesse a livello implementativo e di leggibilità del

codice. Come illustrato in figura 2.1, NVIDIA fornisce un compilatore capace di riconoscere le istruzioni CUDAma l'implementazione di un programma parallelo avviene utilizzando codice sorgente sia per CPU che per GPU. Il compilatore NVIDIA C (*nvcc*) dunque, identifica il tipo di istruzione richiamando i compilatori di riferimento gestendo così questa convivenza.

2.2 Architettura hardware

Oggi sul mercato delle schede video possiamo trovare innumerevoli tipi di device, e i computer moderni referibilmente posseggono una scheda video dedicata. In particolare la **Nvidia Corporation** ha creato anche diverse architetture hardware per soddisfare ogni tipo di richiesta. Quelle conosciute sono le architetture **Kepler**, **Fermi** e **Tesla**. L'architettura Kepler è quella più utilizzata nei computer in commercio con scheda grafica NVIDIA.

In generale, le architetture GPU NVIDIA, sono composte da un array di *Streaming Multiprocessors (SMs)*. Lo Streaming Multiprocessors è progettato per eseguire centinaia di threads in parallelo e contiene un determinato numero di Streaming Processors (SP). Gli Streaming processors sono anche chiamati *CUDA cores* e il loro numero dipende dalla capacità del device installato.

2.2.1 Compute capability

Ogni device possiede un *revision number* che possiamo definire come la **compute capability** del device, e determina l'insieme di funzionalità che possono essere usate nell'implementazione di codice parallelo in CUDA. La compute capability è definita dal più alto numero di revision number e il minor numero di revision number. Nel caso in cui devices diversi abbiano il più alto revision number posseggono la stessa architettura. Il più alto numero di revision number per le architetture Kepler è 3, per i devices basati su un'architettura Fermi è 2, mentre per i device con architettura Tesla 1. Il numero minore di revision number invece, corrisponde al miglioramento del core dell'architettura che spesso porta a nuove funzionalità da poter utilizzare tramite le API fornite appunto da NVIDIA.

2.2.2 Architettura Kepler

L'architettura Kepler è stata progettata e successivamente lanciata nel 2010 insieme all'architettura Fermi. La prima GPU basata sull'architettura Kepler si chiamava "GK104" in cui ogni unità interna è stata progettata ai fini di avere la miglior performance per watt (perf/watt). Alcuni esperti hanno affermato

che la GK104 Kepler è la GPU più potente per la computazione e il rendering grafico dei videogames.

Inizialmente la GPU utilizzata per questo lavoro di tesi è stata la NVIDIA GeForce GT 750M basata anch'essa su un architettura Kepler. Il core in particolare è il "GK107" che offre due shader di blocchi, chiamati **SMX**, ognuno dei quali ha 192 shaders per un totale di 384 shader cores con una velocità di 967 MHz.

Figura 2.2: La scheda video NVIDIA GT 750-M.

2.3 Interfaccia di programmazione

Un programma CUDA consiste in una o più fasi che sono eseguite sia lato host (**CPU**) che lato device (**GPU**). Le fasi in cui l'ammontare computazionale non è eccessivo, e dunque non siamo in presenza di parallelismo dei dati, vengono implementate lato host, mentre le fasi che richiedono un grosso ammontare di parallelismo dei dati sono implementate lato device. CUDA consente di creare un unico file sorgente con codice host e device insieme. Il compilatore NVIDIA C (**nvcc** fig. 2.1) separa le due diverse implementazioni durante il processo di compilazione.

Il linguaggio per scrivere codice sorgente lato device è ANSI C, esteso con particolari *keywords* per far comprendere al compilatore quali sono le funzioni con la presenza di parallelismo. Queste funzioni sono chiamate ***kernels***. Per utilizzare nvcc naturalmente dobbiamo essere in possesso di una GPU NVidia correttamente montata sulla propria macchina, ma se così non fosse si può emulare su CPU le features di CUDA per poter eseguire i kernels (MCUDA tool etc.).

Le funzioni kernel generano un determinato di threads eseguiti in parallelo per raggiungere il data parallelism. Ad esempio per la somma di due matrici può essere implementata come un kernel dove ogni threads computa un elemento dell'output. Il massimo del parallelismo si ha quando ad ogni threads è associata una cella della matrice. Se la dimensione della matrice è 1000 x

1000 servono 1 milione di threads per raggiungere il nostro scopo. Lato CPU per generare e eseguire lo scheduling di un enorme numero di threads è particolarmente oneroso, mentre in CUDA c'è un ottimo supporto hardware da questo punto di vista, dunque il programmatore può sorvolare su questo tipo di problema.

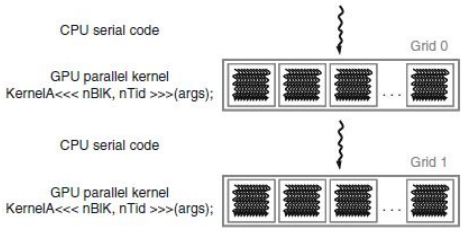


Figura 2.3: Esecuzione di un programma CUDA.

Una tipica esecuzione di un programma CUDA è mostrata nella Fig. 2.3. L'esecuzione viene eseguita a strati, la prima ad essere eseguita è la parte host (CPU) per poi susseguirsi un insieme di strati che possono comportare anche il lancio dei kernels nel caso ci siano parti parallelizzate. I threads sono inglobati all'interno di **blocchi** che a loro volta sono parte di una griglia di blocchi chiamata **grid**. Quando un kernel termina, il programma continua con l'esecuzione lato host fino a che un nuovo kernel viene lanciato.

2.3.1 I kernel

Come detto in precedenza, la funzione *kernel* specifica il codice che deve essere eseguito da tutti i threads lanciati nella fase parallela di un programma CUDA. Tutti i threads lanciati in parallelo eseguono lo stesso codice, infatti un programma CUDA non è nient'altro che l'applicazione pratica del modello Single-Program Multiple-Data (Tassonomia di Flynn 1.2). Questa tecnica è molto utilizzata nei sistemi paralleli.

Per poter dichiarare un kernel c'è una specifica keyword di CUDA da utilizzare: "**__global__**". Questa keyword indica che la funzione è un kernel e questa funzione richiamata dall'host genererà una griglia di threads sul device, in particolare può solamente essere richiamata lato host (a meno che non ci sia un ambiente addatto per potere utilizzare il parallelismo dinamico 2.3.4). CUDA genera threads suddivisi in blocchi, ed ogni blocco appartiene ad una griglia. Lo schema è mostrato in figura 2.4.

In realtà, la dimensione della griglia e dei blocchi la decide il programmatore, che organizza le diverse dimensioni in base al problema e al suo effettivo utilizzo. Si può avere fino a tre dimensioni diverse (x,y,z) sia per la griglia che per i blocchi. Ad ogni blocco, come per ogni threads, è assegnato un indice che

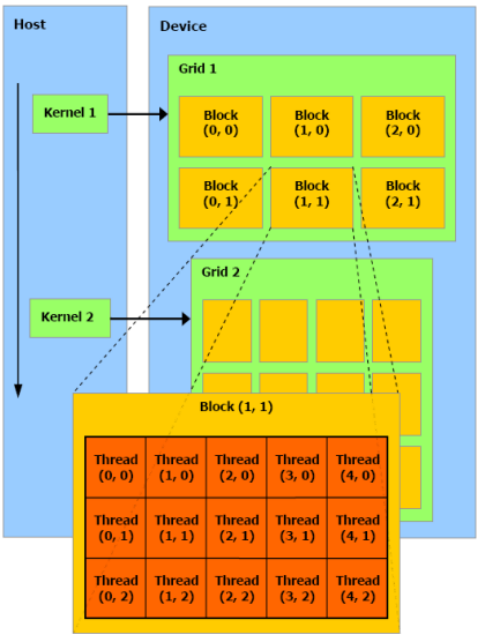


Figura 2.4: Esempio generico di griglie e blocchi in un programma CUDA.

può essere ottenuto tramite altre keywords. Le keywords threadIdx.x e threadIdx.y (e in caso anche threadIdx.z) si riferiscono all'indice dei threads all'interno di un blocco. Tutti i threads eseguono lo stesso codice presente all'interno di un kernel, quindi abbiamo bisogno di un meccanismo per distinguerli in modo da potergli dare direttive diverse o gestire il loro comportamento. Come per i threads anche i blocchi hanno delle specifiche keywords per risalire alle loro coordinate. blockIdx.x e blockIdx.y hanno il compito di ritornare il valore delle coordinate per ogni blocco. Ogni blocco deve avere lo stesso numero di threads.

Spesso i programmatori CUDA utilizzano la struct dim3 per dichiarare la dimensione di griglie e blocchi. E' una struttura che contiene tre diversi interi (le tre dimensioni). Ad esempio se dichiarassimo dim3 dimGrid(3,2,2) vogliamo far intendere al compilatore che la dimensione della griglia sarà tridimensionale, dove in particolare la x avrà valore 3, la y 2 e la z 2. Nel caso in cui invece dichiarassimo dim3 dimGrid(3) il compilatore comprende che vogliamo solamente utilizzare una dimensione e imposterà la y e la z ad 1 automaticamente.

Non dimentichiamo però che le dimensioni di griglie e blocchi vengono definite lato host e non all'interno dei kernels.

In ultimo è bene fare la distinzione tra i tre tipi di funzione che possono essere dichiarate in un programma CUDA. Il primo tipo sono i kernel accompagnati dalla keyword "__global__", descritti in questo paragrafo, gli altri due tipi sono "__device__" e "__host__". Come si può intuire una funzione

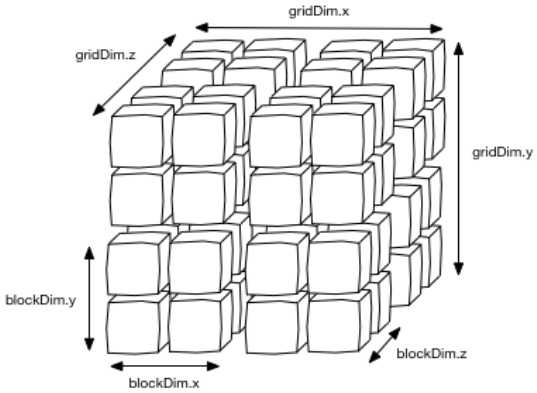


Figura 2.5: Un esempio di configurazione di griglie e blocchi tridimensionale in CUDA.

di tipo "__device__" può essere richiamata dai kernels e dunque verrà lanciata lato device, mentre "__host__" sarà una funzione che verrà richiamata lato host, in cui non avviene nessun parallelismo. Nel caso in cui una funzione viene accompagnata da "__host__" e "__device__" insieme, il compilatore genera due versioni della funzione diverse: una per il device e un'altra per l'host. Se una funzione invece non possiede nessuna keyword, implicitamente verrà compilata come una funzione host.

Per lanciare un kernel, bisogna aggiungere alla chiamata a funzione la sua configurazione definita all'interno di $\langle\langle\langle$ e $\rangle\rangle\rangle$. Al loro interno vanno definiti i parametri relativi alla dimensione di griglie e blocchi. Un esempio lo troviamo in 2.1.

Naturalmente la dimensioni di griglie e blocchi sono limitate in base alla scheda grafica presente sulla macchina. Ad esempio sulla scheda GTX 680 il massimo numero di threads per blocchi è 1024 e la dimensione massima di un blocco è $1024 \times 1024 \times 64$.

```
1 dim3 grid(3,2,2);
2 dim3 block(4,2);
3
4 kernel<<<grid, block>>>();
```

Codice 2.1: Esempio del lancio di un kernel con griglie e blocchi definiti con la struct dim3.

2.3.2 La memoria

In CUDA, host e device hanno spazi di memoria separati. L'hardware dei devices sono dotati di random memory access propri (DRAM). Quindi per eseguire

un kernel sul device, il programmatore ha bisogno di allocare la memoria sul device e trasferire le informazioni pertinenti ai dati sui cui si vuole agire parallelamente dalla memoria sull'host verso la memoria allocata sul device. Il sistema CUDA fornisce al programmatore, tramite le sue API, le funzioni per gestire le allocazioni e i trasferimenti tra le memorie sull'host e sul device.

Le funzioni C malloc(...) e memcpy(...) sono riproposte da CUDA C con la versione cudaMalloc(...) e cudaMemcpy(...) che eseguono rispettivamente un'allocazione sulla memoria device e una trasferimento di dati tra la memoria sull'host e la memoria sul device. In particolare cudaMemcpy(...) ha bisogno di ricevere in input anche la direzione del trasferimento dei dati (da host a device e viceversa). Ecco alcuni esempi delle due funzioni citate:

```
1 //Allocazione sul device
2 cudaMalloc((void**)&data, sizeof(...));
3
4 //Trasferimento dei dati da CPU a GPU
5 cudaMemcpy(void *dst, void *src, sizeof(...), cudaMemcpyHostToDevice);
6 //Trasferimento dei dati da GPU a CPU
7 cudaMemcpy(void *dst, void *src, sizeof(...), cudaMemcpyDeviceToHost);
```

Codice 2.2: Allocazione e trasferimenti dei dati tra CPU e GPU utilizzando CUDA C.

Questa è la prima teoria da conoscere ma, come vedremo, ci sono diversi tipi di memoria a cui un thread può accedere all'interno del device. I tipi di memoria possono essere classificate per grado di privacy oppure sulla loro velocità. Tutti i threads possono accedere liberamente alla **global memory** chiamata anche comunemente *device memory*. I threads all'interno dello stesso blocco possono accedere ad una memoria condivisa, chiamata **shared memory**, utilizzata per la loro cooperazione, ed infine tutti possiedono una memoria locale chiamata **registro**.

Ci sono anche due diversi tipi di spazi di memoria che possono essere utilizzati dai threads: la memoria costante e la texture memories. Ognuna di loro ha un uso particolare, ad esempio la constant memory viene utilizzata per salvare i dati che non cambieranno in tutto il ciclo di vita del kernel.

La global memory

Lo spazio di memoria più utilizzato per la lettura e la scrittura dei dati è la global memory, allocata e completamente gestita lato host. In particolare, in modo da ottimizzare l'accesso alla DRAM non c'è nessun controllo di consistenza e più threads possono scrivere e leggere allo stesso tempo senza nessun meccanismo di esclusività. Per questo le varie incoerenze devono essere completamente gestite dal programmatore.

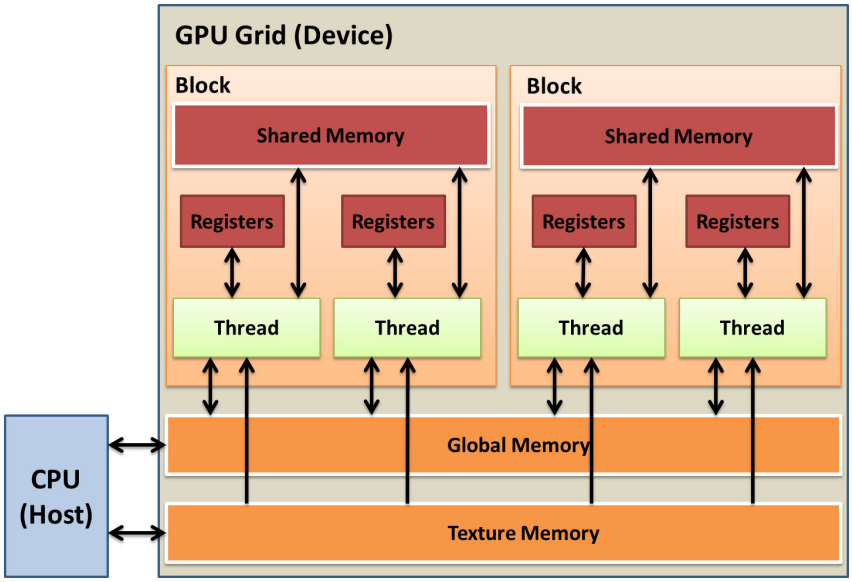


Figura 2.6: La struttura della memoria gestita dal sistema CUDA.

La shared memory

La shared memory è una parte di memoria utilizzata per condividere dati tra threads all'interno dello stesso blocco. Ogni thread dunque può leggere, scrivere e modificare dati presenti sulla shared memory ma non può eseguire le stesse operazioni sulla shared memory di un altro blocco. CUDA offre un ottimo meccanismo per consentire una comunicazione e cooperazione dei threads ottimizzata. In particolare una motivazione per cui utilizzare la memoria condivisa è la differenza di velocità rispetto alla global memory, già con semplici esempi come la moltiplicazione tra matrici si può notare come l'utilizzo della shared memory rispetto alla global memory, comporta un ottimizzazione dell'algoritmo. Un altra differenza rispetto alla global memory è che la durata di vita della shared memory è uguale al ciclo di un kernel, cioè al termine del kernel anche la shared memory terminerà il suo lavoro rilasciando i dati salvati in precedenza.

In particolare la shared memory è suddivisa in banks, in cui ogni bank può eseguire solo una richiesta per volta.

La constant memory

Una parte della memoria sul device è la costant memory, che consente di salvare un limitato numero di simboli, precisamente 64KB. Si può accedere a questo tipo di memoria solo in modalità lettura. In particolare può essere utilizzata per aumentare le performance di accesso ai dati che devono essere condivisi

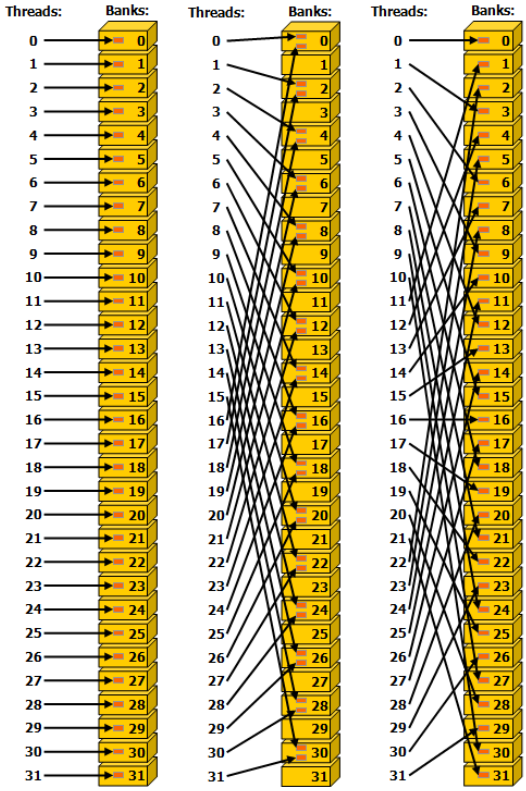


Figura 2.7: Shared memory divisa in banks.

da tutti i threads. La keyword utilizzata per salvare determinati dati sulla memoria costante è: "__constant__".

2.3.3 Atomicità

Come scritto in precedenza, la global memory non gestisce nessun tipo di inconsistenza dei dati. Per questo è il programmatore che deve gestire la scrittura e la lettura concorrente. Per fare ciò, le API di CUDA, forniscono diverse funzioni che favoriscono la mutua esclusione per i dati a cui i threads accedono. In particolare vengono fornite diverse funzioni, tra cui le più note sono quelle relative alle operazioni aritmetiche. Facciamo un breve elenco delle funzioni atomiche fornite dalle API di CUDA:

atomicAdd() gestisce l'esclusività per l'operazione somma.

atomicSub() gestisce l'esclusività per l'operazione sottrazione.

atomicMin() gestisce l'esclusività per il calcolo del minimo.

atomicMax() gestisce l'esclusività per il calcolo del massimo.

atomicInc() gestisce l'esclusività per l'operazione di incremento.

atomicDec() gestisce l'esclusività per l'operazione di decremento.

atomicAnd() gestisce l'esclusività per l'operazione *AND*.

atomicOr() gestisce l'esclusività per l'operazione *OR*.

atomicXor() gestisce l'esclusività per l'operazione *XOR*.

atomicCAS() gestisce l'esclusività per l'operazione di *compare and swap*.

Grazie a queste funzioni, un programmatore CUDA può gestire le concorrenze quando c'è strettamente bisogno della mutua esclusione.

2.3.4 Parallelismo dinamico

Il parallelismo dinamico è un estensione di CUDA, introdotta con CUDA 5.0, che consente la creazione e la sincronizzazione di un kernel direttamente dal device. Sfruttare questa opportunità comporta diversi vantaggi in termini di performance, infatti creare *lavoro* direttamente da GPU può ridurre il bisogno di trasferire dati tra host e device come il controllo dell'esecuzione e sincronizzazione. In particolare questa nuova feature consente al programmatore di gestire la configurazione dei threads anche a runtime direttamente dal device. Stessa opportunità si ha per il parallelismo dei dati che può essere generato direttamente all'interno di un kernel prendendo i vantaggi che l'hardware della GPU offre come lo scheduling e il load balancing.

Il parallelismo dinamico è supportato dai device con una compute capability pari a 3.5 o superiore. [1]

All'interno di un kernel, un thread può configurare e lanciare una nuova griglia di blocchi chiamata "child grid" mentre la griglia a cui appartiene il thread si chiamerà "parent grid". La sincronizzazione tra parent e grid è implicita nel caso in cui non viene espressamente definita. L'immagine 2.8 è un chiaro esempio di approccio al parallelismo dinamico.

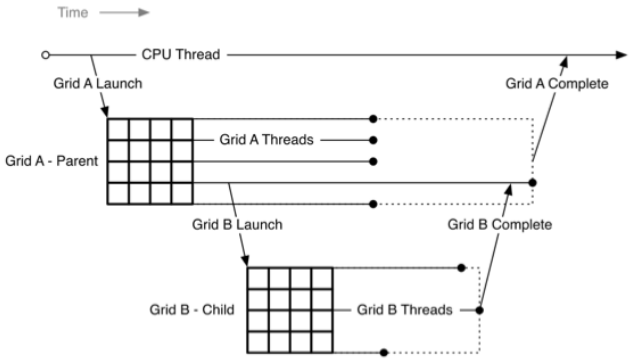


Figura 2.8: Dynamic Parallelism.

Le griglie parent and grid condividono la stessa memoria globale e la stessa memoria costante ma non la shared memory e la memoria locale (2.3.2). La coerenza e la consistenza possono diventare un problema nell'utilizzo del parallelismo dinamico, ragione per cui a volte è espressamente indicato l'utilizzo di una sincronizzazione esplicita. In generale ci sono due punti di esecuzione in cui c'è la sicurezza di avere dei dati consistenti: quando un thread invoca una nuova child grid e quando la child grid ha completato la sua esecuzione. Comunque sia, la sincronizzazione può avvenire in qualsiasi momento tramite due funzioni appartenenti alle API di CUDA: cudaDeviceSynchronize() e __syncthreads().

```
1  dim3 grid(3,2,2);
2  dim3 block(4,2);
3
4  __global__ void child(/* arguments */) {
5
6          /* algorithm */
7
8  }
9
10 __global__ void kernel(/* arguments */) {
11
12         child<<<grid, block>>>(/* arguments */);
13 }
14
15 int main() {
16
17         kernel<<<grid, block>>>(/* arguments */);
18 }
```

Codice 2.3: Esempio di un programma CUDA utilizzando il Dynamic parallelism.

2.4 Tools di sviluppo

Nsight Visual Studio e Nsight Eclipse Edition sono due ottime soluzioni per implementare un programma CUDA. La distinzione tra i due fondamentale è il sistema operativo in cui operano: il primo sul sistema Windows e il secondo sui sistemi Linux e MacOS.

Spesso, durante le fasi implementative di un programma parallelo, il programmatore ha bisogno di funzionalità per ottimizzare i tempi e le performance di un programma. Anche nelle applicazioni sequenziali ormai il Debug è diventato fondamentale per la corretta implementazione di un programma. In CUDA, come nel resto dei paradigmi per il parallelismo, non è scontato avere queste utilità nei software per lo sviluppo.

Fortunatamente, le soluzioni implementate per CUDA offrono al programmatore diverse features e tools per ottimizzare il codice e favorire la riuscita di una buona implementazione. Nei sistemi Linux e MAC troviamo **CUDA-GDB**), tool di NVIDIA, che consente il debugging delle applicazioni CUDA. Un altro tool degno di menzione è **CUDA-MEMCHECK**, incluso in CUDA Toolkit, che controlla l'accesso alla memoria e i vari errori che possono essere incontrati in corso di esecuzione (es. out of bounds, errori di accesso alla memoria etc.).

Gli ambienti Nsight per lo sviluppo di applicazioni offrono un sistema user friendly che facilità la compilazione dellle applicazioni CUDA. Visual Profiler invece risulta essere di vitale importanza ai fini della performance consentendo ai programmatori di comprendere e ottimizzare le applicazioni CUDA. La potenza del profiler è la facile comprensione del risultato, molto simile ad un diagramma di Gantt, che mostra a video le attività della CPU e della GPU includendo analisi automatiche sull'applicazione identificando opportunità di miglioramento della performance.

2.4.1 Nsight Visual Studio

Visual Studio è un ambiente di sviluppo molto conosciuto dai programmatori. E' sviluppato da Microsoft e supporta diversi linguaggi di programmazione quali C, C++, C#, ASP .Net. Inoltre è un ambiente di sviluppo multipiattaforma con cui poter realizzare applicazioni per PC, Server ma anche web applications e applicazioni per smartphone.

Nel suo più comune utilizzo offre in dotazione un debugger e un compilatore per il linguaggi citati.

La versione Nsight è utilizzata dagli sviluppatori CUDA e fornisce diversi strumenti per il Debug, il Profiler e la computazione eterogenea per applicazioni CUDA C/C++.

La sua installazione è semplice e la creazione di progetti è guidata per ogni tipo di esigenze. In ambiente Windows è veramente immediata l'installazione del toolkit fornito da NVIDIA, che consente di creare progetti NVIDIA CUDA direttamente da Visual Studio.

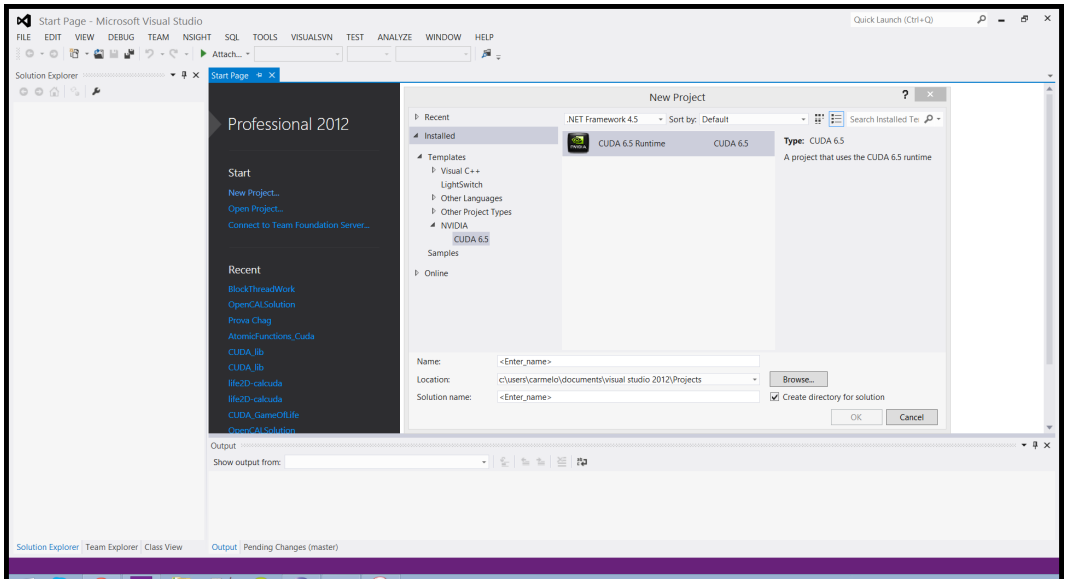


Figura 2.9: Creazione di un progetto CUDA 6.5 su Visual Studio.

2.4.2 Visual Profiler

Il Visual Profiler è un software secondario fornito da NVIDIA utile per un'analisi approfondita dell'utilizzo della memoria e delle performance in generale della GPU. E' un ambiente ricco di funzionalità e informazioni utili che il programmatore può utilizzare ai fini di migliorare il programma CUDA e migliorarne le prestazioni.

Il software si presenta come in figura 2.10.

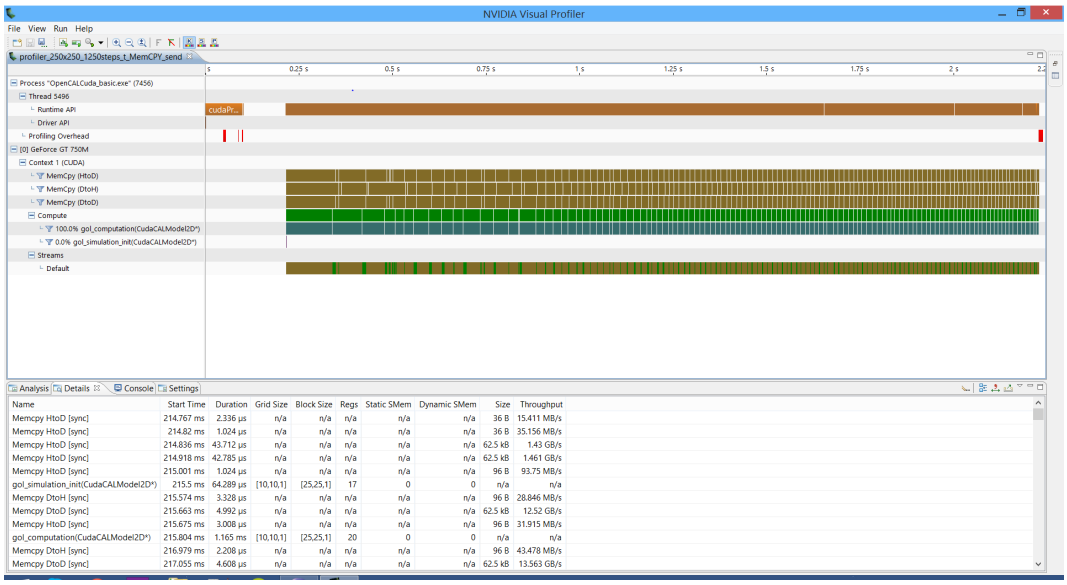


Figura 2.10: Esempio di progetto analizzato su VIsual Profiler.

Tra le tante analisi effettuate dal software, quelle che risultano più interessanti sono sicuramente le informazioni relative al trasferimento di dati tra GPU e CPU e le informazioni sui tempi impiegati dai kernel e dal loro effettivo utilizzo.

Sul trasferimento dei dati tra memoria è interessante conoscere anche la velocità di trasferimento che naturalmente cambia da scheda a scheda e da tipo di trasferimento. Il trasferimento dei dati più veloce avviene all'interno del device. Infatti una copia di memoria da device a device, su una scheda video NVIDIA GT-750M, può arrivare fino a 4,5 TB/s, con trasferimenti che impiegano nanosecondi.

Il profiler risulta molto utile in fase di programmazione poiché rende facile l'individuazione dei kernel "lenti". Spesso si abusa di chiamate ai kernel senza accorgersene e senza profiler è sicuramente più difficile individuare i punti critici del programma.

Nel lavoro di tesi è stato utilizzato il profiler parecchie volte in fase di programmazione proprio per implementare la versione più performante della libreria OpenCAL. Per poter utilizzare Visual Profiler bisogna creare un nuovo progetto che prende in input l'eseguibile del progetto CUDA compilato e la cartella dei dati che vengono utilizzati dal programma. E' anche possibile utilizzare altre funzionalità valide per le analisi ma possono essere attivate in fase di profiling. Naturalmente il programmatore potrebbe anche desiderare di analizzare solo parte del programma, per questo il profiler prende in considerazione

solamente il codice racchiuso tra le chiamate a funzione cudaStartProfiler() e cudaStopProfiler().

Il Visual Profiler è scaricabile facilemente dal sito di NVIDIA, e può essere utilizzato sia in ambienti Linux/Unix e MacOS che su ambienti Windows.

Capitolo 3

Automi Cellulari

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim.

Vestibulum pellentesque felis eu massa.

Capitolo 4

OpenCAL

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim.

Vestibulum pellentesque felis eu massa.

Capitolo 5

OpenCAL-CUDA

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim.

Vestibulum pellentesque felis eu massa.

Capitolo 6

Conclusioni

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim.

Vestibulum pellentesque felis eu massa.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis.
Curabitur dictum gravida mauris.

— Donald Ervin Knuth

Ringraziamenti

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Bibliografia

[1] *CUDA Dynamic Parallelism, Programming guide.*

[2] *NVIDIA CUDA Compute Unified Device Architecture Programming Guide.*

[3] Kartashev P. e Nazaruk V. *Analysis of GPGPU Platforms Efficiency in General-Purpose Computations.*

[4] Wright Richard S. e Lipchak Benjamin. *OpenGL SuperBible, Third Edition.* Sams Publishing, 2004.