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.

Introduzione

In un epoca in cui le alte prestazioni sono drasticamente essenziali nei più disparati campi scientifici, ci troviamo spesso a dover fronteggiare problematiche di inefficienza o esigenze di miglioramento con molteplici mezzi e soluzioni presenti nelle teorie informatiche moderne. Il termine **velocità** è diventato sinonimo di successo in diversi contesti ed è il protagonista principale di molti obiettivi progettuali dei nostri tempi. Tra i tanti fattori che comportano il successo di un calcolatore, di un software o di un'applicazione per dispositivi mobili possiamo distinguere naturalmente anche la velocità di risposta.

Sin dai primi calcolatori, le migliorie apportate alle macchine furono progettate e implementate quasi sempre con lo scopo di incrementare la la velocità. Negli ultimi vent'anni in particolar modo l'aumento prestazionale è stato e continua ad essere un esigenza. Dal punto di vista hardware c'è stata una vera rivoluzione che nel corso degli anni ha portato ad avere le più innovative tecnologie apportate ai processori che oggi popolano i nostri calcolatori. La richiesta prestazionale ha inciso in maniera dirompente nel mercato dello sviluppo del software portando così alla progettazione di tecniche innovative per migliorare sempre di più l'esperienza utente e le performance. Proprio i miglioramenti apportati hanno stravolto l'esperienza di utilizzo giornaliero integrando sempre di più l'utilizzo dei computer e altri dispositivi nella vita di tutti i giorni. Un esempio banale potrebbe essere un'applicazione mobile che ha il compito di fornire informazioni ai cittadini relative ai mezzi pubblici di trasporto. Se l'applicazione ha un tempo di risposta molto lenta, dovuta per esempio alla mole di dati da dover processare, potrebbe risultare inutilizzabile.

Ecco perché oggi il calcolo parallelo è molto utilizzato nello sviluppo del software. I notevoli miglioramenti lato hardware, che hanno dato i natali ai processori di ultima generazione dotati di più core (i cosìdetti *multicore*), hanno comportato lo sviluppo di tecniche innovative proprio per poter utilizzare a pieno questa nuova tipologia di processori. Il calcolo parallelo oggi ha un grosso impatto in diverse aree informatiche, dalla ricerca scientifica fino allo sviluppo di software commerciale. Grazie a questa branca informatica oggi è possibile sfruttare a pieno le nuove tecnologie e tutti i vantaggi dei nuovi dispositivi ultraveloci. Nel corso degli anni sono stati sviluppati e migliorati anche diverse metodologie e modelli paralleli, a partire dal miglioramento delle architetture all'arrivo dei nuovi framework. I due modelli di architettura più utilizzati oggi si dividono in modelli basati su memoria condivisa e modelli basati sul paradigma dello scambio di messaggi.

La GPU (Graphics Processing Unit), inizialmente utilizzata solamente per il rendering grafico delle immagini, negli ultimi anni è stata scoperta come potenza di calcolo con velocità teoriche quasi dieci volte superiori alle normali CPU. Questo ha rivoluzionato le teorie sul calcolo parallelo che in particolare ha tratto vantaggi anche in termini di costi. Dato l'iniziale successo delle GPU si è introdotto nel parallel computing il termine GPGPU programming, che rappresenta tutti gli utilizzi delle GPU che non comprendano il rendering grafico. Oggi la GPGPU programming è utilizzata in decine di campi scientifici,

dalla bio-informatica all'analisi finanziaria, coprendo anche campi come la fluidodinamica computazionale, l'apprendimento automatico e la scienza dei dati [7].

Nvidia Corporation, società che opera nello sviluppo delle GPU ormai da tantissimi anni, ha puntato molto sul calcolo parallelo producendo device sempre più adatti per uno scopo computazionale. **CUDA** è un architettura completa (hardware e software) creata proprio da Nvidia che abilita alla GPGPU programming sfruttando proprio le schede grafiche Nvidia oramai molto diffuse. In particolare sin dal 2007, anno del suo lancio, CUDA ha subito numerosi aggiornamenti in cui ognuno di loro ha portato diverse features innovative che rendono la GPGPU programming sempre più semplice ed estremamente comoda. L'unico neo di questa potentissima architettura è la portabilità. Non è infatti possibile eseguire programmi scritti in CUDA su schede video diverse dalla Nvidia. Per questo è stato creato nel 2008 OpenCL che risulta però leggermente diverso sia nel suo utilizzo che nella sua architettura.

Tra i sistemi complessi maggiormente noti in campo scientifico possiamo trovare gli **automi cellulari**. Gli automi cellulari sono insiemi di regole logico-matematiche capaci di descrivere sistemi complessi e rappresentarne la loro evoluzione nel tempo. Un AC può essere descritto come uno spazio suddiviso in celle regolari ognuna delle quali può trovarsi in un numero finito di stati. La legge che detta la sua evoluzione nel tempo è chiamata funzione di transizione comune per tutte le celle. Uno degli input per ogni cella è l'insieme dei suoi vicini configurati da una relazione di vicinanza che non varia nel tempo e nello spazio [9].

L'applicazione degli automi cellulari trova spazio in diversi campi di ricerca come la simulazione del comportamento dei pedoni nei centri commerciali [14], fino alla simulazioni di fenomeni naturali come frane [18] e colate laviche [12].

OpenCAL (Open Cellular Automata Library) è una libreria open source per la modellazione e la simulazione di modelli basati su automi cellulari, in particolare su automi cellulari complessi (CCA). La libreria nasce per rendere l'implementazione degli automi cellulari più semplice e immediata. Infatti grazie al suo utilizzo, l'utente potrà concentrarsi completamente sulla progettazione del modello senza dover dare particolari attenzioni ai dettagli implementativi. Nel corso del tempo sono state sviluppate diverse versioni della libreria utilizzando il calcolo parallelo, dalla versione in OpenMP alle versioni OpenCL e CUDA. Grazie alla sua comodità e le sue performance può essere considerata una valida alternativa a software per la creazione di modelli basati su automi cellulari come Camelot.

Questo lavoro di tesi ha comportato la progettazione e la successiva implementazione di **OpenCAL-CUDA**. E' stata utilizzata l'architettura CUDA per la progettazione di un ulteriore versione parallela della libreria sfruttando la GPGPU programming.

La tesi è suddivisa come segue: **??**

Il primo capitolo offre una visione generale sul parallel computing. Approfondiremo l'argomento con la descrizione delle più famose tecniche e metodologie attualmente utilizzate, compresi i modelli di comunicazione e un esempio di progettazione di un algoritmo parallelo. Ci sarà un breve cenno sul paradigma di programmazione OpenMP seguito da una parte introduttiva della GPGPU programming.

Il secondo capitolo descrive l'architettura CUDA, utilizzata nel lavoro di tesi. In particolare, dopo una breve introduzione, verrà descritta l'architettura hardware e la sua interfaccia di programmazione. Nello stesso capitolo verranno introdotti i tools di sviluppo messi a disposizione per operare con questo tipo di archietettura.

Il terzo capitolo introduce la teoria basata sugli automi cellulari e in particolare nel paragrafo 3.4 si introdurranno gli automi cellulari complessi.

Il quarto capitolo introduce la libreria OpenCAL descrivendo in dettaglio la definizione di un modello e di una simulazione, in particolare verranno proposti degli esempi di utilizzo.

Infine,

Il quinto capitolo dopo una breve introduzione, descrive le scelte progettuali intraprese nella parallelizzazione della versione CUDA di OpenCAL. Nei paragrafi successivi, si descriverà la nuova forma strutturale che rappresenta questa nuova implementazione e infine, come per il quarto capitolo, ci saranno degli esempi di utilizzo abbinati ad alcuni test effettuati per validare il progetto svolto e gli obiettivi di tesi.

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 ha 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), dando i natali ai processori di ultima generazione: i processori **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 hardware 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 del parallelismo.

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.

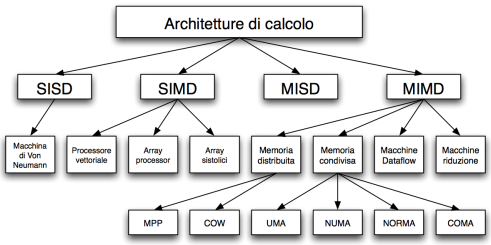
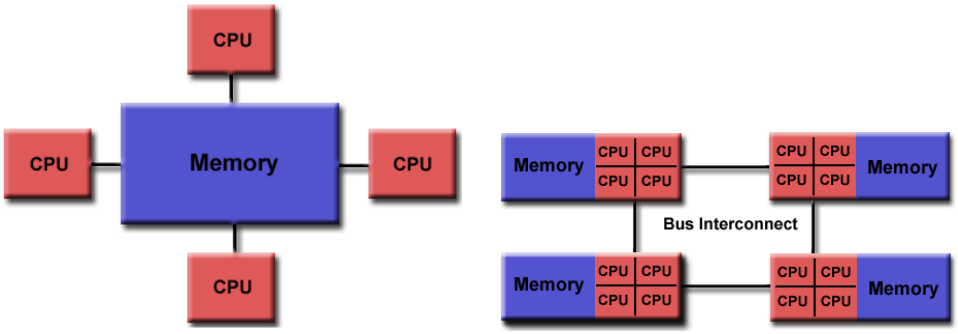


Figura 1.1: La tassonomia di Flynn



(a) UMA. (b) NUMA.

Figura 1.2: UMA e NUMA.

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)

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 distribuita 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 sono 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

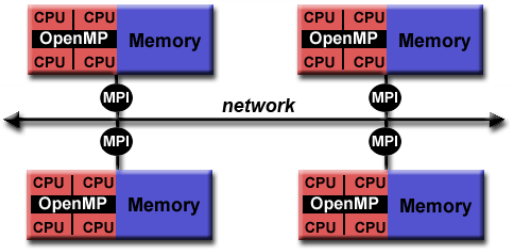


Figura 1.3: Esempio di sistema ibrido.

guide di calcolo parallelo evidenziano le seguenti problematiche per la progettazione di un algoritmo parallelo *nontrivial* [1]:

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

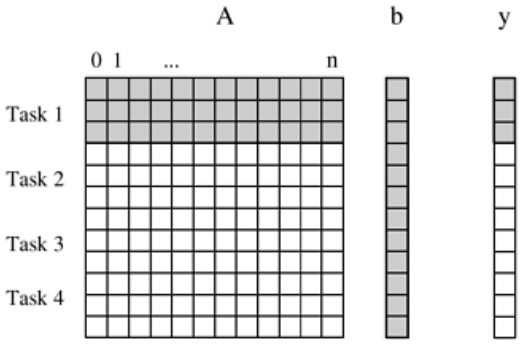


Figura 1.4: Suddivisione di una moltiplicazione tra una matrice e un vettore in 4 diversi task.

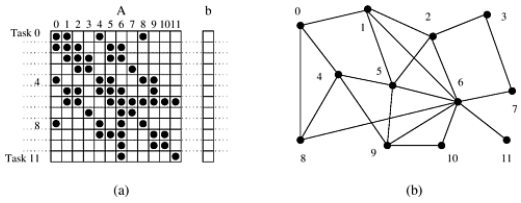


Figura 1.5: Esempio di un grafo delle interazioni tra i task.

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.

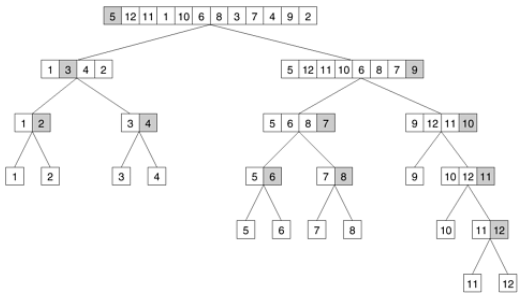


Figura 1.6: Esempio di decomposizione ricorsiva: il quicksort.

- 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.

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

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

(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.

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 [1].

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.

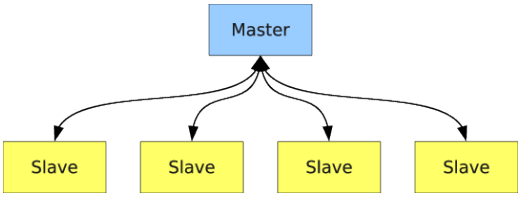


Figura 1.8: Il modello Master-Slave.

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 [15]). 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 [8] 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.

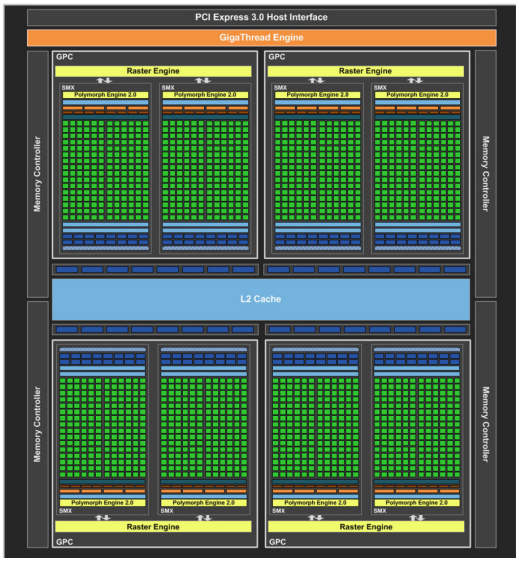


Figura 1.9: Architettura Kepler (GTX 680).

La figura 1.9 rappresenta l'architettura di una tipica GPU CUDA. La struttura è composta da un insieme di *streaming multiprocessor* (SM) divisi in blocchi (Nella figura 1.9 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 [13]. Generalmente lo stesso program-

ma viene eseguito per ogni dato da elaborare e la latenza dovuta agli accessi alla memoria globale viene “nascosta” dalla grande quantità di calcoli.

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 permette 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.

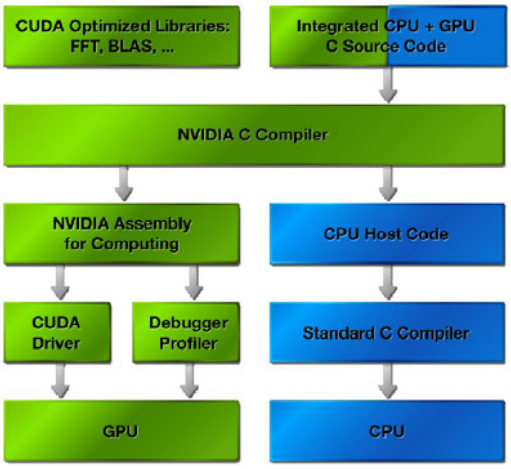


Figura 2.1: Struttura di Nvidia C Compiler.

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 CUDA. L'implementazione di un programma parallelo avviene utilizzando codice sorgente sia per CPU che per GPU.

Il compilatore NVIDIA C (*nvcc*) identificando il tipo di istruzione richiama i compilatori di riferimento, così da gestire nel miglior modo possibile la presenza di istruzioni per le differenti architetture (CPU e GPU).

2.2 Architettura hardware

Oggi sul mercato delle schede video possiamo trovare innumerevoli tipi di device e i computer di ultima generazione posseggono quasi sempre una scheda video dedicata. In particolare la **Nvidia Corporation** ha creato 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. Se devices diversi hanno il più alto revision number uguale sta a significare che 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 ad un miglioramento incrementale dell'architettura di base che spesso può comportare nuove funzionalità.

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 fu 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 numero di threads eseguiti in parallelo per raggiungere il data parallelism. Ad esempio per la somma di due matrici può essere implementata in 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 tralasciare questo tipo di problema.

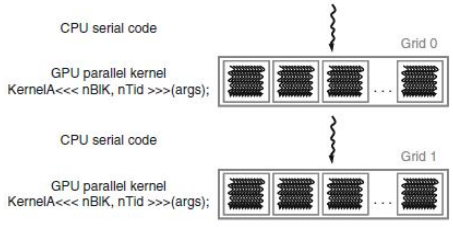


Figura 2.3: Esecuzione di un programma CUDA.

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 sezioni da eseguire in parallelo. 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__**". La chiamata ad un kernel, obbligatoriamente richiamata lato host (a meno che non ci sia un ambiente addatto per potere utilizzare il parallelismo dinamico 2.3.4), genererà una griglia di threads sul device. CUDA genera threads suddivisi in blocchi, ed ogni blocco appartiene ad una griglia. Lo schema è mostrato in figura 2.4.

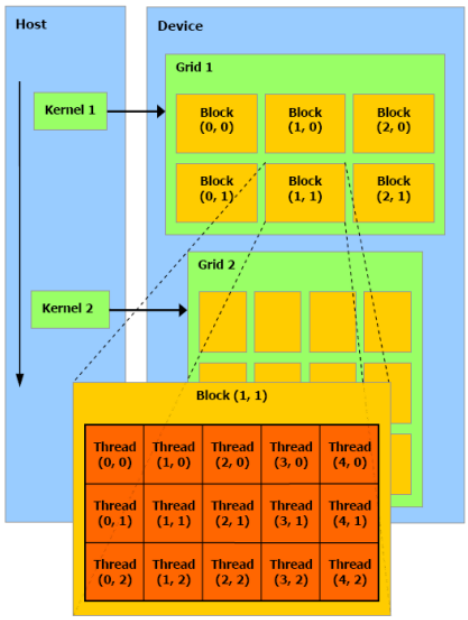


Figura 2.4: Esempio generico di griglie e blocchi in un programma CUDA.

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 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. L'identificazione di un thread è strettamente necessario nel calcolo parallelo, per questo c'è bisogno di un meccanismo per distinguere diversi threads in modo da poter dare direttive precise e diverse ad ognuno di loro. Come per i threads anche i blocchi hanno delle specifiche keywords per risalire

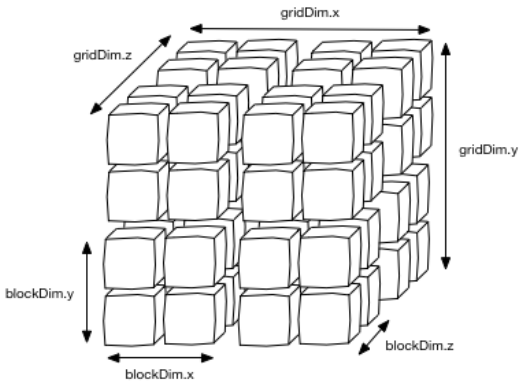


Figura 2.5: Un esempio di configurazione di griglie e blocchi tridimensionale in CUDA.

alle loro coordinate. blockIdx.x e blockIdx.y hanno il compito di restituire il valore delle coordinate per ogni blocco. Ogni blocco ha 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 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**.

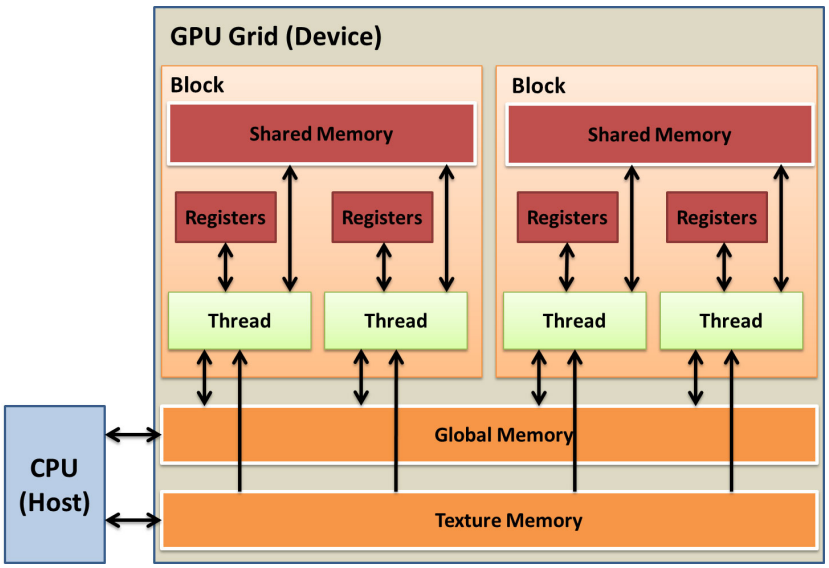


Figura 2.6: La struttura della memoria gestita dal sistema CUDA.

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.

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 alcuna operazione sulla shared memory di un altro blocco. CUDA offre un ottimo meccanismo per consentire una comunicazione e cooperazione dei threads veloce.

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 miglioramento di performance. Un altra differenza rispetto alla global memory è che al termine delle operazioni del kernel la shared memory terminerà il suo lavoro rilasciando i dati salvati in precedenza mentre la global memory mantiene le informazioni fino alla fine di tutto il programma.

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 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. Proprio per questa causa le API di CUDA forniscono diverse funzioni che favoriscono la mutua esclusione per l'accesso dei threads ai dati.

Le più note operazioni implementate dalle API di CUDA sono quelle relative alle operazioni aritmetiche. Facciamo un breve elenco delle funzioni più conosciute:

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.

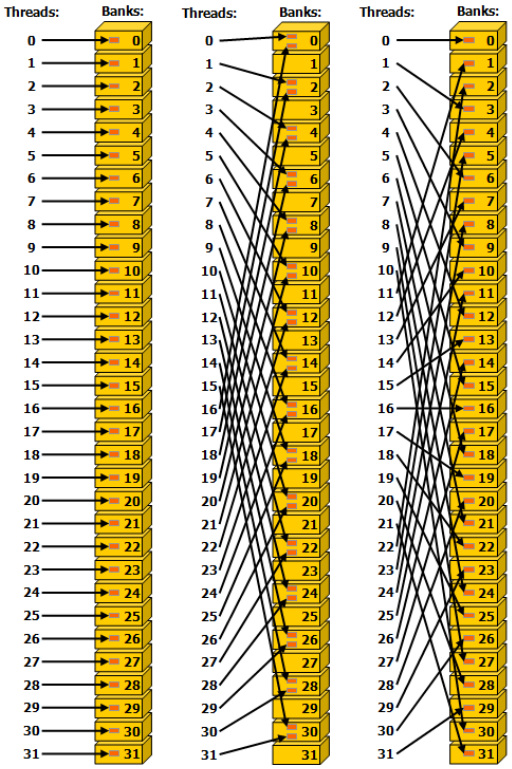


Figura 2.7: Shared memory divisa in banks.

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.

Creare un kernel direttamente da GPU può ridurre il bisogno di trasferire dati tra host e device così come riduce il controllo dell'esecuzione e della sincronizzazione dei threads. In particolare questa nuova feature consente al programmatore di gestire la configurazione

dei threads anche a runtime direttamente dal device. La stessa opportunità si ha per il parallelismo dei dati che può essere generato direttamente all'interno di un kernel, così da trarre beneficio dei vantaggi che l'hardware della GPU offre (scheduling, load balancing etc.).

Il parallelismo dinamico è supportato dai device con una compute capability pari a 3.5 o superiore. [6]

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.

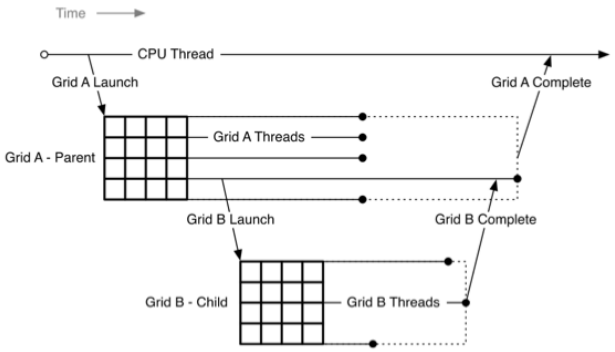


Figura 2.8: Dynamic Parallelism.

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 fondamentale tra i due 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**) [4], tool di NVIDIA, che consente il debugging delle applicazioni CUDA. Un altro tool degno di menzione è **CUDA-MEMCHECK** [4], 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.

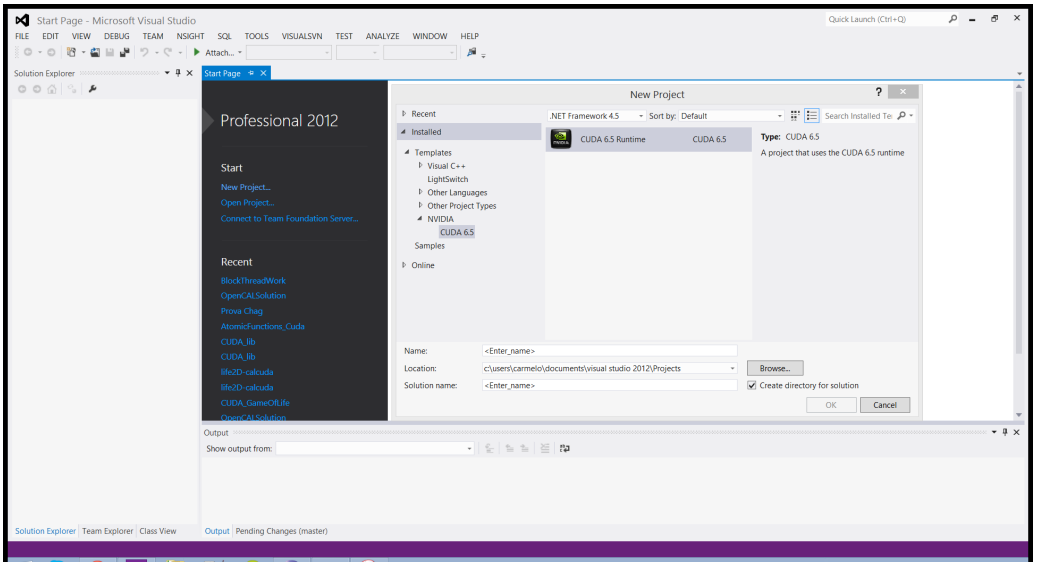


Figura 2.9: Creazione di un progetto CUDA 6.5 su Visual Studio.

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.

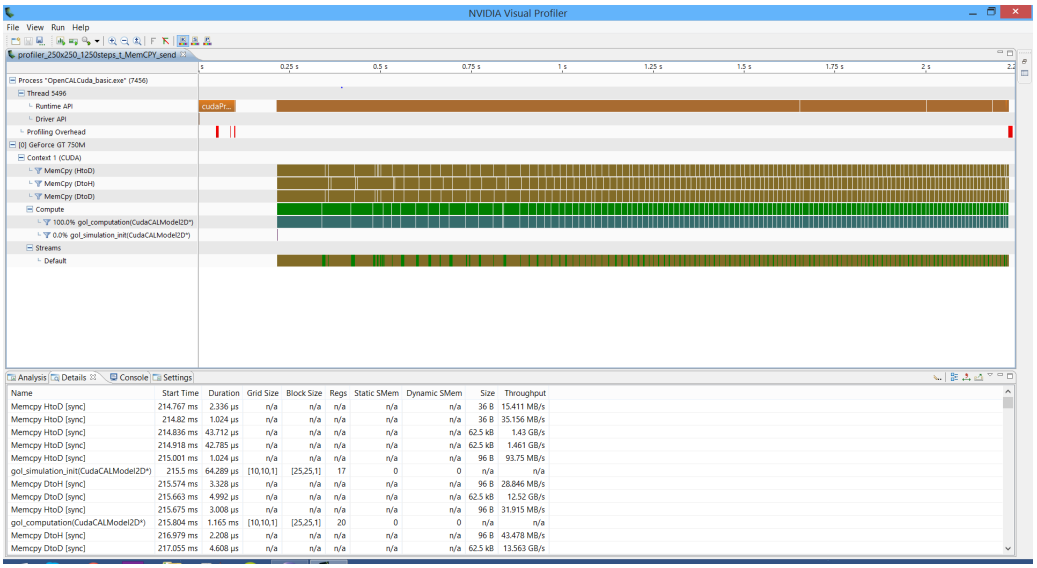


Figura 2.10: Esempio di progetto analizzato su VIsual Profiler.

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

3.1 Introduzione

Oggigiorno, dopo decadi di sviluppo e ricerca, la scienza ricopre un ruolo importante in ogni settore. Le pubbliche istituzioni e le aziende private sentono il bisogno di utilizzare metodi e nuove tecniche ai fini di scoprire e interpretare diversi casi di studio, a partire dai fenomeni naturali fino alla statistica e all'economia. Soprattutto i fenomeni naturali, suscitano un particolare interesse nell'ambito della ricerca scientifica, poiché dal loro studio è possibile fornire informazioni significative sul loro comportamento futuro e le eventuali precauzioni (nei casi di fenomeni più o meno disastrosi).

Nel 1947 da John von Neumann, con lo scopo di ideare un sistema in grado di autoriprodursi, introdusse gli **Automi Cellulari**. Gli Automi Cellulari (AC) sono modelli matematici usati per la simulazione di sistemi complessi. Un AC è sostanzialmente uno spazio d-dimensionale suddiviso in celle regolari alle quali è associato uno **stato** facente parte di un insieme finito di stati. Il valore dello stato di una cella in un determinato istante è dato da una **funzione di transizione** uguale per tutte le celle. La funzione di transizione calcola lo stato di una cella in base allo stato del suo **vicinato**, ossia un insieme di celle legate da una prefissata relazione. Tutto ciò avviene in passi discreti e lo stato di tutte le celle è aggiornato in modo simultaneo. L'insieme degli stati di tutte le celle in un istante t determina la configurazione dell' Automa Cellulare in quell' istante.

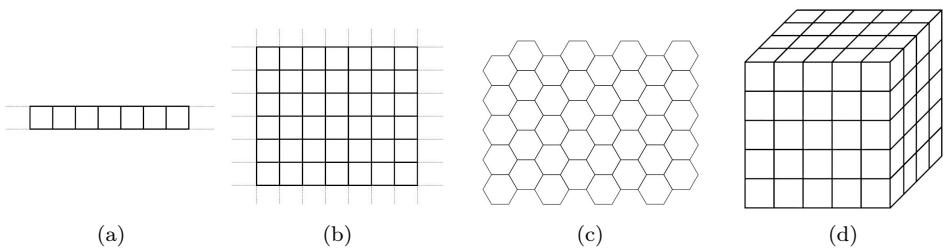


Figura 3.1: Esempi di spazi cellulari (a) unidimensionale, (b) e (c) bidimensionale, (d) tridimensionale.

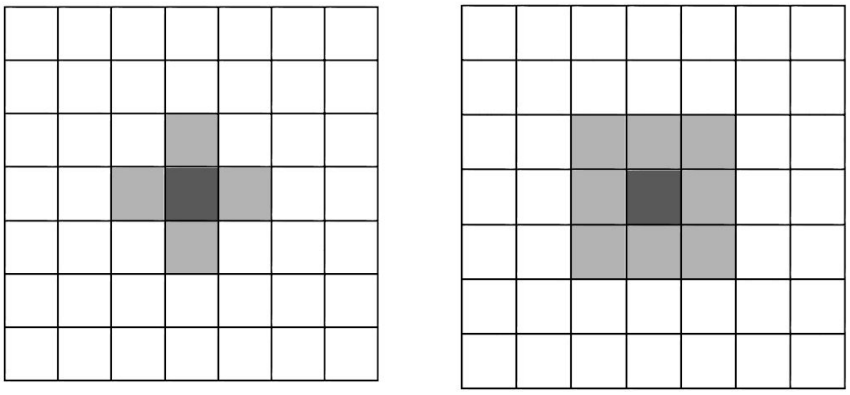
3.2 Definizione di Automa Cellulare

3.2.1 Definizione informale di Automa Cellulare

Informalmente possiamo dare una definizione di Automa Cellulare e distinguere le proprietà fondamentali come di seguito:

- lo spazio è formato da un insieme di celle regolari;
- l'insieme degli stati che possono essere assunti dalla cella è finito;
- l'evoluzione dell'automa avviene a passi discreti;
- le celle si evolvono in modo simultaneo in base ad un'unica funzione di transizione;
- lo stato di ogni cella nell'istante $t+1$ dipende dal suo stesso stato e dallo stato delle celle vicine al tempo t;
- la relazione di vicinanza è uguale per tutte le celle e non varia durante l'evoluzione dell'Automa Cellulare. Inoltre deve coinvolgere un numero limitato di celle.

Nel caso di un automa cellulare bidimensionale, i vicinati più comuni sono quelli di von Neumann (figura 3.2 a) e quello di Moore (figura 3.2 b). Il primo è composto dalla cella presa in esame e dalle celle adiacenti a nord, sud, est, e ovest, mentre il secondo comprende anche le celle adiacenti a nord-est, sud-est, nord-ovest e sud-ovest. E' possibile comunque definire altri tipi di vicinato e non è necessario che le celle che lo costituiscono siano prossime tra loro. La funzione di transizione cambia lo stato di ogni cella in base



(a) *Vicinato di von Neumann* (b) *Vicinato di Moore*

Figura 3.2: Esempi di relazioni di vicinanza.

agli stati delle celle del vicinato.

$$\sigma : S_n \to S \tag{3.1}$$

La funzione 3.1 rappresenta la funzione di transizione di un Automa Cellulare, dove S rappresenta l'insieme dei possibili stati che possono essere assunti da una cella, mentre S_n rappresenta l'insieme degli stati delle n celle del vicinato. Se la transizione avviene da uno stato a un unico altro stato allora la funzione è deterministica. Se invece la transizione può avvenire da uno stato a un insieme di stati allora la funzione è non deterministica. Nel

primo caso, dato un Automa Cellulare all'istante $t = 0$, la sua configurazione all'istante successivo è unica, mentre nel secondo caso ci possono essere diverse configurazioni.

3.2.2 Definizione formale di Automa Cellulare

Un Automa Cellulare può essere formalizzato come una quadrupla [16]

$$< E^d, S, X, \sigma >$$

- E^d è l'insieme delle celle individuate da punti a coordinate intere nello spazio euclideo reale $d-$dimensionale, nel quale ogni cella è un automa elementare;
- S è l'insieme finito degli stati della cella, detto *spazio degli stati*;
- X è un insieme finito di vettori $d-$dimensionali, detto indice di vicinanza, il quale definisce l'insieme dei vicini $N(X, i)$ di una generica cella $i =< i_1, i_2, \ldots, i_d >$ in E^d come segue:
  sia $X = \{z_1, z_2, \ldots z_n\}$ con $n = \#X$ e $z_j =< x_{j1}, x_{j2}, \ldots, x_{jd} >$ per $i \leq j \leq n$ allora:
  $$N(X, i) = \{i + z_1, i + z_2, \ldots, i + z_n\} \tag{3.2}$$
- $\sigma : S^n \to S$ è la funzione di transizione elementare, ossia la funzione di transizione dell'automa elementare, con $n = \#X$.

In particolare, nel caso di un automa cellulare bidimensionale avremo:

- l'insieme delle celle $E^2 = \{(i, j)|i, j \in N, m \geq 0, n \geq 0\}$ è una matrice $n \times m$, dove (i, j) è la cella posta nella riga $i-$esima e nella colonna $j-$esima;
- lo stato della cella (i, j) al tempo t sara $s^t(i, j)$, essendo $S = \{s_1, s_2, \ldots, s_r\}$ l'insieme dei possibili stati della cella;
- $N(i, j)$ è il vicinato della cella (i, j) al tempo t, composto dallo stato della cella stessa $s^t(i, j)$ e dagli stati delle celle con essa connesse;
- $\sigma(N(i, j))$ è la funzione di transizione che dà il nuovo stato $s^{t+1}(i, j)$ della cella (i, j) al tempo $t + 1$.

Con questa definizione di AC possiamo definire i vicinati di von Neumann e di Moore come segue:

$$V_{vN}(i, j) = \{s^t(k, l) \in L | \|k - i\| + \|l - j\| \leq 1\}$$
$$V_M(i, j) = \{s^t(k, l) \in L | \|k - i\| \leq 1, \|l - j\| \leq 1\}$$

3.3 Automi cellulari unidimensionali

Tra gli Automi Cellulari sono di particolare interesse quelli unidimensionali. Gli AC unidimensionali sono costituiti da n celle disposte su una griglia di dimensioni $1 \times n$ dove la prima e l'ultima cella sono adiacenti, cioé la griglia può essere considerata come un anello. Gli stati che possono essere assunti dalla cella sono $k = 2$ (ovvero 0 e 1) e il raggio del vicinato è $r = 1$. Se prendiamo in considerazione la cella x il suo vicinato è formato dalla terna $< x - r, x, x + r >$. Pertanto nel caso degli AC unidimensionali, poiché ogni cella può assumere $k = 2$ stati e poiché ogni vicinato è composto da tre celle ci sono $2^3 = 8$

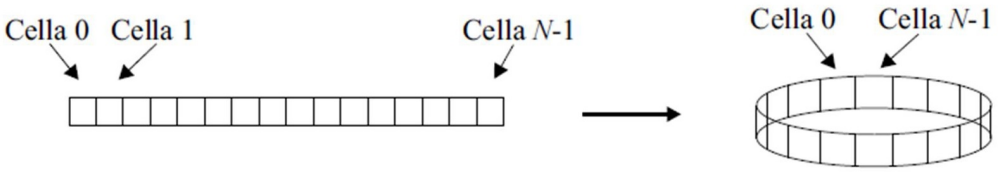


Figura 3.3: Esempio di automa cellulare unidimensionale

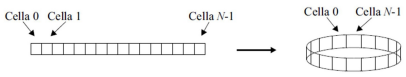


Figura 3.4: Evoluzione di un automa cellulare unidimensionale che ha regola di transizione 90. Ogni riga rappresenta un'iterazione dell'automa cellulare. Il colore bianco rappresenta le celle che sono nello stato 0 mentre il colore nero quelle che sono nello stato 1.

possibili combinazioni di vicinato e $2^8 = 256$ possibili configurazioni dell'automa cellulare. La regola di transizione può essere espressa con una tabella elencando in ordine crescente le configurazioni del vicinato e gli stati delle celle centrali all'iterazione successiva (Tabella 3.1).

Tabella 3.1: Nella prima riga della tabella sono elencate in ordine crescente le configurazioni dei vicinati al tempo t_0. Nella seconda riga sono elencati gli stati delle celle centrali al tempo t_1.

t_0	000	001	010	011	100	101	110	111
t_1	0	0	0	0	0	1	0	1

A ogni configurazione dell'AC corrisponde una legge di transizione che determina lo stato delle celle al passo successivo ed è rappresentata da una stringa di 8 bit formata dagli stati delle celle stesse. Dato che generalmente le regole sono indicate con un numero intero, si converte la stringa di 8 bit in un numero intero. Nel caso dell'esempio nella tabella 3.1 la regola è identificata dalla stringa di bit $s = 00000101$ è quindi dal numero intero $n = 5$.

I diversi comportamenti degli AC unidimensionali dovuti alle diverse funzioni di transizione possono essere raggruppati secondo Wolfram [20] [19] in quattro classi:

Classe 1 L'automa converge dopo poche iterazioni verso una configurazione uniforme indipendentemente da quale sia la configurazione iniziale.

Classe 2 L'automa converge verso una configurazione in cui alcune strutture si ripetono ciclicamente durante l'evoluzione dell'automa cellulare.

Classe 3 L'automa si evolve con un comportamento caotico.

Classe 4 L'automa si evolve con un comportamento sia caotico che ordinato. Inoltre è costituito da strutture semplici che interagendo tra loro creano strutture più complesse.

3.4 Automi Cellulari Complessi (CCA)

Gli Automi Cellulari Complessi (CCA) sono un'estensione della classica definizione di Automa Cellulare. In genere questo modello numerico si adatta alla rappresentazione di fenomeni le cui dinamiche possono essere descritte in termini di interazioni locali ad un livello macroscopico. Tra le applicazioni degli Automi Cellulari Complessi troviamo simulazioni di flussi lavici [12], flussi di detriti [18], incendi [21], traffico stradale [17], ecc. Estendendo la classica definizione di CA, gli Automi Cellulari Complessi possono facilitare la definizione di molti aspetti rilevanti per la modellazione dei sistemi complessi. In particolare:

- lo stato della cella è costituito da un insieme di sottostati. Il prodotto cartesiano dei valori assunti dai sottostati definisce lo stato della cella;
- la funzione di transizione viene suddivisa in processi elementari;
- il modello viene calibrato con un insieme di parametri "globali" che permettono di variare il comportamento del fenomeno preso in considerazione;
- un sottoinsieme di celle è soggetto anche ad influenze esterne. Lo stato della cella, oltre a dipendere dalla funzione di transizione e dallo stato delle celle del vicinato, dipende anche da un'altra funzione che rappresenta fenomeni esterni all'Automa Cellulare.

Più formalmente un CCA è rappresentato da una 7-tupla:

$$< E^d, X, Q, P, \sigma, G, \gamma >$$

Dove E^d, X e σ hanno lo stesso significato che hanno per gli AC classici (vedo paragrafo 3.2.2) e

- $Q = Q_{s1} \times Q_{s2} \times \ldots \times Q_{sn}$ è l'insieme dei sottostati della cella;
- $P = p_1, p_2, \ldots, p_p$ è l'insieme dei parametri globali usati per calibrare il modello;
- $G = G_1 \cup G_2 \cup \ldots \cup G_l \subseteq E^d$ è il sottoinsieme di celle di E^d soggette ad influenza esterna.
- $\gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_t\}$ è l'insieme di funzioni che definiscono le influenze esterne

Capitolo 4

OpenCAL

4.1 Liberia per Automi Cellulari

Oggigiorno la modellistica è molto utilizzata negli ambienti di ricerca, tra i dipartimenti universitari di Geologia, Biologia, Ingegneria e Bioinformatica. Per questo motivo negli anni sono state sviluppate diverse metodologie per la realizzazione di sistemi automatici per la creazione di modelli e della loro simulazione come CAMELot [11]. OpenCAL (Open Cellular Automata Library) è una libreria Open Source, capace di definire modelli di simulazione basati su Automi Cellulari complessi (CCA).

OpenCAL nasce dalla necessità di avere una libreria open source e di facile utilizzo, che permetta all'utente di concentrarsi su ciò che riguarda la definizione dell'automa cellulare trascurando il più possibile i dettagli implementativi. Le funzioni, le strutture e i tipi di dato descritti nei prossimi paragrafi permettono di definire un modello di AC con uno spazio cellulare bidimensionale. Tuttavia OpenCAL supporta anche la definizione di modelli tridimensionali e le funzioni, le strutture e i tipi di dato usati per la definizione di un modello 2D hanno la loro controparte per la definizione di un modello 3D.

4.2 Utilizzare OpenCAL

Un vantaggio dell'utilizzo di OpenCAL si trova proprio sulla sua facilità di comprensione e di utilizzo, infatti in pochi passi è possibile definire un modello. La gestione del modello e della simulazione sono compito delle due struct principali: CALModel2D e CALRun2D. La libreria fornisce anche funzionalità per le operazioni di Input, Output e Buffer per la gestione dei file (ad esempio i dati sulla morfologia). Nelle prossime due sezioni si specificheranno la definizione di un modello e di una simulazione nei dettagli.

4.2.1 Definizione di un modello

In una prima fase di implementazione, il programmatore deve prendersi cura della definizione del modello. Come detto in precedenza, arrivati a questa fase l'utente ha già ben chiara la progettazione dell'automa cellulare e della sua evoluzione. Si tratta dunque di scrivere in codice le regole già progettate. Grazie ad OpenCAL questo può essere svolto in pochi e brevi passi. E' molto facile capire quanto possa essere oneroso impiegare del tempo per implementare tutte le strutture necessarie ai fini di completare un programma

in C/C++ adatto per automi cellulari. Curarsi solamente della progettazione del modello, lasciando ad OpenCAL il compito di gestire il *core* del problema, è senza dubbio il punto di forza della libreria.

Come anticipato in precedenza la libreria offre una struct (CALModel2D) per contenere le informazioni del modello. La funzione calCADef2D permette di ottenere un'istanza di CALModel2D definendone le caratteristiche. La funzione prende in input quattro diversi tipi di parametri:

- le dimensioni dello spazio cellulare
- la relazione di vicinanza delle celle (vicinato)
- la condizione ai bordi dello spazio cellulare
- la possibilità di utilizzare un tipo di ottimizzazione

Le dimensioni dello spazio cellulare sono semplicemente le righe e le colonne della matrice. La relazione di vicinanza delle celle è definita da un enumerativo CALNeighborhood2D con cui è possibile scegliere tra i vicinati più noti come Von Neumann, Moore ed il vicinato esagonale, questo non preclude la possibilità all'utente di definire una relazione di vicinato *custom* grazie alla funzione calAddNeighbor2D che riceve in input le coordinate relative del vicino che si vuole aggiungere rispetto ad una cella centrale.

```
/* ...  */

//MODEL
CALModel2D* model = calCADef2D(ROWS, COLUMNS, CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
                CAL_SPACE_TOROIDAL, CAL_NO_OPT);

/* ... */
```

Codice 4.1: Esempio della definizione di un modello con vicinato di Von Neumann.

In questo esempio (vedi 4.1), possiamo osservare la definizione di un modello con vicinato di Von Neumann, che utilizza uno spazio di celle toroidale e non utilizza nessuna tecnica di ottimizzazione.

```
/* ...  */

//MODEL
CALModel2D* model = calCADef2D(ROWS, COLUMNS, CAL_CUSTOM_NEIGHBORHOOD_2D,
                CAL_SPACE_TOROIDAL, CAL_NO_OPT);

// Neighborhood definition
calAddNeighbor2D (model , 0, 0);
calAddNeighbor2D (model , - 1, 0);
calAddNeighbor2D (model , 0, - 1);
calAddNeighbor2D (model , 0, + 1);
calAddNeighbor2D (model , + 1, 0);

/* ...  */
```

Codice 4.2: Esempio della definizione di un modello con vicinato custom definito dall'utente tramite la funzione calAddNeighbor2D.

L'ultima immagine invece mostra la definizione di un modello con un vicinato customizzato, utilizzando la funzione calAddNeighbor2D.

Le condizioni ai bordi sono definite da un altro enumerativo CALSpaceBoundaryCondition. Le due condizioni che possono essere scelte sono: CAL_SPACE_TOROIDAL e

CAL_SPACE_FLAT. Il primo permette di scegliere uno spazio toroidale il secondo invece uno spazio non toroidale.

L'ultima condizione riguarda la possilità di utilizzare un'ottimizzazione ai fini di migliorare le performance del programma. Si può scegliere se utilizzare le "celle attive" (con l'opzione CAL_OPT_ACTIVE_CELLS o meno.

Naturalmente la nostra definizione di modello non può fermarsi a questo punto. Come abbiamo visto nel capitolo 3 un modello è composto anche da stati. In particolare nel caso degli automi cellulari complessi (CCA) gli stati delle celle possono essere suddivisi in sottostati. Dunque, OpenCAL prevede tre tipi di sottostati:

CALSubstate2Dr sottostati di tipo reale (**floating point** in C)

CALSubstate2Di sottostati di tipo intero (**int** in C)

CALSubstate2Db sottostati di tipo byte (**char** in C)

Ogni sottostato ha due matrici linearizzate: matrice *current* e matrice *next*. La prima matrice è utilizzata per leggere i valori correnti dei sottostati mentre la seconda viene utilizzata per memorizzare i nuovi valori calcolati. Dopo ogni step della simulazione il contenuto della matrice *next* viene copiato sulla matrice *current* in modo da ottenere il parallelismo implicito cosicché i cambiamenti effettuati sui sottostati non modifichino lo stato corrente delle celle finché non si va al passo di calcolo successivo.

Per allocare nuovi sottostati si utilizza la funzione calAddSubstate2D(b|i|r) che restituisce un puntatore al sottostato appena creato. Ci sono casi in cui un sottostato non deve obbligatoriamente avere la doppia matrice, per questo c'è anche la possibilità di allocare sottostati con un singolo layer (dunque con la sola matrice *current*) con la funzione calAddSingleLayerSubstate2D(b|i|r).

```
1  CALSubstate2Db* gol_substate;
2
3  int main()
4  {
5
6          /* ...  */
7
8          //Model
9          CALModel2D* GameOfLife = calCADef2D(ROWS, COLUMNS, CAL_MOORE_NEIGHBORHOOD_2D,
               CAL_SPACE_FLAT, CAL_NO_OPT);
10
11         // add substates
12         gol_substate = calAddSubstate2Db(GameOfLife);
13
14         // load substate from file
15         calLoadSubstate2Db(GameOfLife, gol_substate, PATH);
16
17         /* ... */
18
19         return 0;
20 }
```

Codice 4.3: Esempio di creazione e inizializzazione di un sottostato.

In realta quest'esempio mostra solo una parte di funzionalità che in questa fase si possono utilizzare. Ad esempio la libreria offre una serie di funzioni per facilitare l'accesso ai sottostati e inizializzare le celle a valori stabiliti.

4.2.2 Definizione del ciclo di esecuzione

Il ciclo di esecuzione comprende tutto il processo di definizione e successivo avvio della simulazione. Tramite la libreria OpenCAL è possibile infatti aggiungere al ciclo di esecuzione le seguenti funzioni:

- una funzione di inizializzazione che verrà richiamata all'inizio del ciclo di esecuzione.
- una funzione di steering che verrà richiamata alla fine di ogni passo di calcolo.
- una funzione che definisce la condizione di stop e può interrompere il ciclo di esecuzione.

Per creare un istanza della simulazione dobbiamo utilizzare la struct CALRun2D. Questa struct oltre a contenere tutte le informazioni relative alla simulazione, racchiude le funzioni citate in precedenza per avviare un ciclo di esecuzione.

Così come per il modello, anche per la simulazione esiste una funzione per definirla: calRunDef2D. Questa funzione prende in input il numero dei passi di calcolo da effettuare e la modalità di aggiornamento dei sottostati. Dal punto di vista del numero dei passi sostanzialmente troviamo due valori da dare in input alla funzione: il passo iniziale e il passo finale. Se il passo finale viene impostato al valore predefinito CAL_RUN_LOOP la simulazione non avrà mai termine. In questo caso in particolare, di solito è definita dall'utente la condizione di stop (ad esempio quando un cratere non emette più lava etc.). Per quanto riguarda l'aggiornamento degli stati questa può avvenire in due modi diversi: implicita CAL_UPDATE_IMPLICIT o esplicita CAL_UPDATE_EXPLICIT. Nel primo caso l'aggiornamento dei sottostati viene gestito dal ciclo di esecuzione di OpenCAL.

```
1  /* ... */
2
3  //add transition function's elementary processes
4  calAddElementaryProcess2D(model, transition_function);
5
6  //Add init function
7  calRunAddInitFunc2D( simulation, init_function);
8  calRunAddStopConditionFunc2D( simulation, stop_condition_function);
9  calRunAddSteeringFunc2D( simulation, steering_function);
10
11 //Start simulation
12 calRun2D( simulation);
13
14 //saving configuration
15 calSaveSubstate2Db( model, substate, PATH_FINAL);
16
17 //finalizations
18 calRunFinalize2D( simulation);
19 calFinalize2D( model);
20
21 /* ... */
```

Codice 4.4: Esempio di definizione di una simulazione.

Ogni volta che viene eseguita una funzione appartenente al ciclo di esecuzione, il contenuto delle matrici *next* di tutti i sottostati viene copiato nelle matrici current. Nel secondo caso la gestione dell'aggiornamento dei sottostati viene lasciata gestire all'utente.

L'utente può infatti definire il proprio ciclo di esecuzione e la politica di aggiornamento dei sottostati. Per fare ciò è necessario usare la funzione calRunAddGlobalTransitionFunc2D che riceve in input un puntatore alla funzione che definisce il ciclo di esecuzione dell'utente. Per aggiornare i sottostati si possono usare le funzioni calUpdate2D (per aggiornarli tutti) e calUpdateSubstate2D(b|i|r) (per aggiornarne uno in particolare). La funzione calRun2D

permette infine di eseguire una simulazione, mentre la funzione calRunCAStep2D permette di eseguire un singolo passo di calcolo.

```
CALbyte calRunCAStep2D(struct CALRun2D* simulation)
{
 //execute user transition function if defined
 if (simulation->globalTransition)
 {
        simulation->globalTransition(simulation->ca2D);
        if (simulation->UPDATE_MODE == CAL_UPDATE_IMPLICIT)
                calUpdate2D(simulation->ca2D);
 }
 else
 //execute all elementary processes defined by the user
 calGlobalTransitionFunction2D(simulation->ca2D);

 //execute steering function if defined
 if (simulation->steering)
 {
        simulation->steering(simulation->ca2D);
        if (simulation->UPDATE_MODE == CAL_UPDATE_IMPLICIT)
                calUpdate2D(simulation->ca2D);
 }

 //check stop condition if defined
 if (simulation->stopCondition)
        if (simulation->stopCondition(simulation->ca2D))
                return CAL_FALSE;

 return CAL_TRUE;
}
```

Codice 4.5: La gestione del ciclo di esecuzione di OpenCAL.

4.3 Game of Life in OpenCAL

Il Game of Life è un automa cellulare ideato dal matematico inglese Conway nel 1970. Conway con la progettazione di questo automa cellulare voleva simulare le dinamiche base della vita e capirne la loro evoluzione nel tempo. Il gioco della vita in particolare è un automa cellulare ripetitivo, cioè dopo cinque step ritorna alla sua configurazione iniziale per poi riprendere la sua evoluzione. Lo spazio di celle del Game of Life è bidimensionale con il vicinato definito da Moore. Una cella può assumere due diversi stati: viva o morta. [10] La funzione di transizione è costituita dalle seguenti semplici regole:

1. Una cella viva, rimane viva se ha esattamente due o tre celle vive nel suo vicinato.
2. Una cella viva, muore per isolamento se ha meno di due celle vive nel suo vicinato.
3. Una cella viva, muore per sovraffollamento se ha più di tre celle vive nel suo vicinato.
4. Una cella morta, torna in vita se ha esattamente tre celle vive nel suo vicinato.

Nella figura 4.1 si mostra l'evoluzione del gioco della vita di Conway con la famosa configurazione dell'aliante (*glider*).

In seguito verrà mostrato l'esempio in C dell'implementazione del *Game of Life* con la libreria OpenCAL.

```
#include <cal2D.h>
#include <cal2DRun.h>
#include <cal2DIO.h>

```

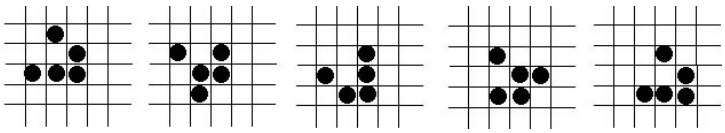


Figura 4.1: L'evoluzione del gioco della vita con la configurazione Glider

```
5  #define ROWS 100
6  #define COLS 100
7  #define STEPS 1000
8  #define PATH gol_result
9
10 //gol substate
11 struct CALSubstate2Di * gol_substate;
12
13 //gol transition function
14 void gol_transition_function(struct CALModel2D* gol, int i, int j)
15 {
16  int sum = 0, n;
17  for (n=1; n<gol->sizeof_X; n++)
18   sum += calGetX2Di(gol, gol_substate, i, j, n);
19
20  if ((sum == 3) || (sum == 2 && calGet2Di(gol, gol_substate, i, j) == 1))
21   calSet2Di(gol, gol_substate, i, j, 1);
22  else
23   calSet2Di(gol, gol_substate, i, j, 0);
24 }
25
26 //initialization function
27 void init(struct CALModel2D* gol){
28  calInitSubstate2Di(gol,gol_substate,0);
29  calInit2Di(gol, gol_substate, 0, 2, 1);
30  calInit2Di(gol, gol_substate, 1, 0, 1);
31  calInit2Di(gol, gol_substate, 1, 2, 1);
32  calInit2Di(gol, gol_substate, 2, 1, 1);
33  calInit2Di(gol, gol_substate, 2, 2, 1);
34 }
35
36 int main() {
37
38  //gol model definition
39  struct CALModel2D * model = calCADef2D(ROWS, COLS, CAL_MOORE_NEIGHBORHOOD_2D,
        CAL_SPACE_TOROIDAL, CAL_NO_OPT);
40  gol_substate = calAddSubstate2Di(model);
41  calAddElementaryProcess2D(model,gol_transition_function);
42
43  //gol execution definition
44  struct CALRun2D * run = calRunDef2D(model, 1, STEPS, CAL_UPDATE_IMPLICIT);
45  calRunAddInitFunc2D(run,init);
46  calRun2D(run);
47
48  //save computation result on a file
49  calSaveSubstate2Di(model, gol_substate, PATH);
50  return 0;
51 }
```

Codice 4.6: Il Game of Life in OpenCAL.

Nell'esempio 4.6 si implementa in circa 50 righe di codice sia il modello che la simulazione del Game of Life. L'estrema semplicità dell'implementazione mette in risalto dunque il punto di forza di OpenCAL. La libreria "nasconde" i dettagli implementativi permettendo all'utente di concentrarsi sulla definizione dell'Automa Cellulare. L'implementazione del programma è divisa in due fasi. Nella prima fase viene definito il modello su cui è basato il Gioco della Vita. Usando la funzione calCADef2D, viene creata un'istanza di CALModel2D con spazio cellulare bidimensionale toroidale e vicinato di Moore. La definizione del modello si conclude con l'aggiunta di un sottostato, che rappresenta

l'insieme degli stati delle celle e l'aggiunta di una funzione di transizione definita con le regole precedentemente elencate. La seconda fase prevede la definizione del ciclo di esecuzione. In questo caso viene usato il ciclo di esecuzione di default implementato in OpenCAL e viene aggiunta una funzione di inizializzazione che imposta lo stato iniziale di tutte le celle del sottostato aggiunto precedentemente. Infine viene richiamata la funzione calRun2D per eseguire una simulazione e i risultati della computazione vengono scritti su file dalla funzione calSaveSubsate2Di.

4.4 "Modello" in OpenCAL

Capitolo 5

OpenCAL-CUDA

5.1 Introduzione

OpenCAL si è rivelata completa e efficace per l'implementazione di automi cellulari. L'evoluzione nel tempo della libreria ha comportato anche diverse versioni e miglioramenti dal lato della performance. Proprio per questo si è pensato di sfruttare i vantaggi del calcolo parallelo (cap. 1) come ricerca e sviluppo di OpenCAL. Attualmente esistono diverse versioni della libreria, a partire da quella sequenziale alla versione parallela OpenCAL-OMP (implementazione in OpenMP), OpenCAL-CL (implementazione in OpenCL) e OpenCAL-CUDA. Proprio quest'ultima verrà introdotta nei dettagli progettuali e implementativi.

Progettare una versione parallela della libreria comporta non solo una fase di studio approfondito della tecnologia da utilizzare, ma anche una buona analisi del codice sequenziale. Lo studio della tecnologia utilizzata possiamo dividerlo in due momenti differenti:

- Scelta del linguaggio e dell'architettura da utilizzare
- Studio pratico della tecnologia scelta

Oggi ci sono decine di modi per parallelizzare un programma, ragion per cui a volte la scelta tra le diverse opportunità può essere decisiva ai fini della riuscita del progetto. Nel caso di OpenCAL-CUDA, la scelta dell'utilizzo dell'architettura CUDA ha trovato riscontro sui buoni risultati ottenuti da passate parallelizzazioni di Automi Cellulari su schede video NVIDIA. Anche la semplicità di CUDA C e della sua elasticità (in continuo aggiornamento) ha mostrato le potenzialità per un progetto a lungo termine e facilmente mantenibile. E' anche vero però, che a volte la scelta della tecnologia dipende anche strettamente dal progetto e dall'utilizzo futuro.

Lo studio del linguaggio CUDA C ha occupato circa un mese del tempo totale utilizzato per la riuscita del progetto. La parallelizzazione in GPU richiede anche tempo di comprensione delle diverse architetture spesso poco conosciute. Oggi, fortunatamente, le stesse case produttrici delle schede video offrono materiale in abbondanza per studiare approfonditamente architetture e linguaggi da utilizzare.

Tornando alla fase di progettazione l'evidenziazione delle sezioni *critiche* del codice, cioé le parti parallelizzabili, e la ricerca di una soluzione ottima è stata senza ombra di dubbio la parte più interessante del progetto.

OpenCAL è una soluzione generica, progettata per essere compatibile con svariati problemi matematici e diversi tipi di automi cellulari. A volte l'utilizzo del parallelismo complica alcuni aspetti implementativi e può comportare diversi cambiamenti progettuali. Per quanto riguarda questo lavoro di tesi, si è pensato di applicare il parallelismo nella completa trasparenza dell'utente ma con l'aggiunta di piccole limitazioni, dovuti alla filosofia del parallelismo in CUDA che non si sposavano a pieno con la versione sequenziale.

Nel resto di questo capitolo si affronteranno passo dopo passo le scelte progettuali che hanno condizionato la parallelizzazione in CUDA della libreria OpenCAL.

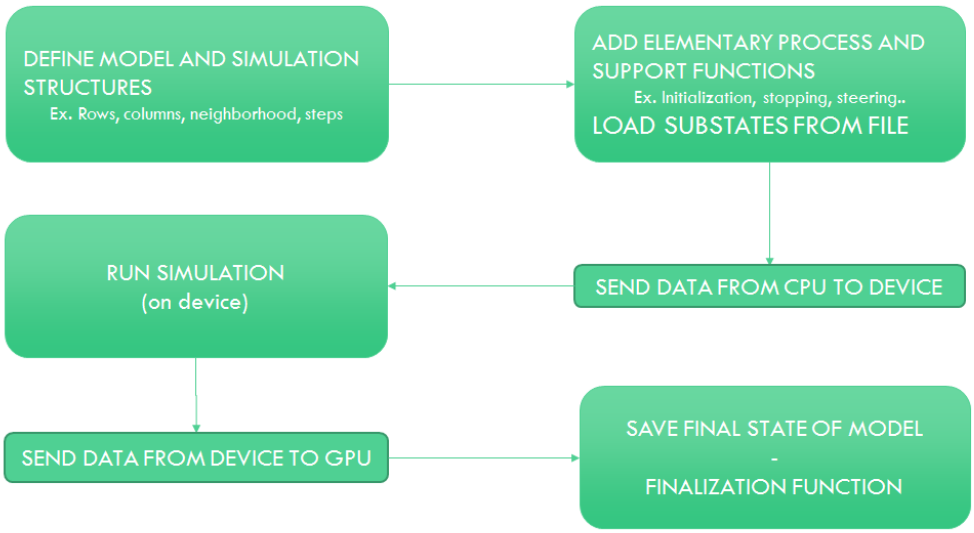


Figura 5.1: Diagramma del ciclo di vita del software OpenCAL-CUDA

5.2 Scelte progettuali

In questo paragrafo si mostreranno le scelte progettuali e le più importanti differenze implementative della versione sequenziale della libreria e della versione parallelizzata in CUDA.

5.2.1 CALModel2D e CudaCALModel2D

Per poter utilizzare la potenza delle GPU, come descritto nei capitoli 2 e 1, è essenziale trasportare i dati del programma sulla memoria del device. Il primo passo è stato appunto capire come poter trasportare (5.2.3) la struct CALModel2D sul device. All'inizio poteva sembrare semplice grazie alla funzione cudaMemcpy{ ... }, ma come vedremo non è stato possibile utilizzarla in questo caso, o meglio, non è stato possibile lasciare l'incarico della copia dell'oggetto al motore di CUDA. La presenza infatti di puntatori (e puntatori di puntatori) all'interno di CALModel2D è stata la causa di tutto ciò. Infatti, come ben sappiamo, la copia di un puntatore non è nient'altro che la copia dell'indirizzo di memoria dove è allocato il puntatore, ed un oggetto sul device non può avere all'interno puntatori allocati sulla memoria dell'host. Dunque il primo passo è stato rendere

più dettagliata possibile la struct CALModel2D. L'opzione adottata è stata scorporare le struct (e i puntatori a struct) interne, come CALCell2D e CALSubstate2D(b|i|r), rendendo il loro contenuto parte della struct principale CALModel2D, in questo modo si è perso un grado di astrazione ma rendendo vantaggioso il trasferimento dei dati da device a host e viceversa. Come si può notare nei codici 5.1 e 5.2, che mostrano le strutture CALModel2D e CudaCALModel2D, si notano implementazioni in genere molto diverse ma rappresentano un modello nella medesima maniera. Infatti l'utente non si renderà mai conto della differenza tra i due tipi di struttura.

```
1  struct CALModel2D {
2
3          //!< Number of rows of the 2D cellular space.
4          int rows;
5
6          //!< Number of columns of the 2D cellular space.
7          int columns;
8
9          //!< Type of cellular space: toroidal or non-toroidal.
10         enum CALSpaceBoundaryCondition T;
11
12         //!< Type of optimization used. It can be CAL_NO_OPT or CAL_OPT_ACTIVE_CELLS.
13         enum CALOptimization OPTIMIZATION;
14
15         //!< Computational Active cells object. if A.actives==NULL no optimization is
               applied.
16         struct CALActiveCells2D A;
17
18         //!< Array of cell coordinates defining the cellular automaton neighbourhood
               relation.
19         struct CALCell2D* X;
20
21         //!< Number of cells belonging to the neighbourhood. Note that predefined
               neighbourhoods include the central cell.
22         int sizeof_X;
23
24         //!< Neighbourhood relation's id.
25         enum CALNeighborhood2D X_id;
26
27         //!< Array of pointers to 2D substates of type byte
28         struct CALSubstate2Db** pQb_array;
29
30         //!< Array of pointers to 2D substates of type int
31         struct CALSubstate2Di** pQi_array;
32
33         //!< Array of pointers to 2D substates of type real (floating point)
34         struct CALSubstate2Dr** pQr_array;
35
36         //!< Number of substates of type byte.
37         int sizeof_pQb_array;
38
39         //!< Number of substates of type int.
40         int sizeof_pQi_array;
41
42         //!< Number of substates of type real (floating point).
43         int sizeof_pQr_array;
44
45         //!< Array of function pointers to the transition function's elementary
               processes callback functions. Note that a substates' update must be
               performed after each elementary process has been applied to each cell of
               the cellular space (see calGlobalTransitionFunction2D).
46         void (**elementary_processes)(struct CALModel2D* ca2D, int i, int j);
47
48         //!< Number of function pointers to the transition functions's elementary
               processes callbacks.
49         int num_of_elementary_processes;
50 };
```

Codice 5.1: La rappresentazione del modello in OpenCAL.

Un esempio lampante è la rappresentazione dei sottostati. CALSubstate2D(b|i|r) mentre in CALModel2D è rappresentato da un puntatore a struct, in CudaCALModel2D è rappresentato da una coppia di puntatori *next* e *current* per ogni tipo di sottostato.

```
struct CudaCALModel2D {

        //!< Number of rows of the 2D cellular space.
        int rows;

        //!< Number of columns of the 2D cellular space.
        int columns;

        //!< Type of cellular space: toroidal or non-toroidal.
        enum CALSpaceBoundaryCondition T;

        //!< Type of optimization used. It can be CAL_NO_OPT or CAL_OPT_ACTIVE_CELLS.
        enum CALOptimization OPTIMIZATION;

        //!< Array of flags having the substates' dimension: flag is CAL_TRUE if the
            corresponding cell is active, CAL_FALSE otherwise.
        CALbyte* activecell_flags;

        //!< Number of CAL_TRUE flags.
        int activecell_size_next;

        //!< i-Array of computational active cells.
        int *i_activecell;

        //!< j-Array of computational active cells.
        int *j_activecell;

        //!< Number of active cells in the current step.
        int activecell_size_current;


        //!< Array of cell coordinates defining the cellular automaton neighbourhood
            relation.
        int *i;
        int *j;

        //!< Number of cells belonging to the neighbourhood. Note that predefined
            neighbourhoods include the central cell.
        int sizeof_X;

        //!< Neighbourhood relation's id.
        enum CALNeighborhood2D X_id;

        //!< Current linearised matrix of the substate, used for reading purposes.
        CALbyte* pQb_array_current;
        //!< Next linearised matrix of the substate, used for writing purposes.
        CALbyte* pQb_array_next;

        //!< Current linearised matrix of the substate, used for reading purposes.
        CALint* pQi_array_current;
        //!< Next linearised matrix of the substate, used for writing purposes.
        CALint* pQi_array_next;

        //!< Current linearised matrix of the substate, used for reading purposes.
        CALreal* pQr_array_current;
        //!< Next linearised matrix of the substate, used for writing purposes.
        CALreal* pQr_array_next;

        //!< Number of substates of type byte.
        int sizeof_pQb_array;
        //!< Number of substates of type int.
        int sizeof_pQi_array;
        //!< Number of substates of type real (floating point).
        int sizeof_pQr_array;

        //!< Array of function pointers to the transition function's elementary
            processes callback functions. Note that a substates' update must be
```

```
            performed after each elementary process has been applied to each cell of
            the cellular space (see calGlobalTransitionFunction2D).
64          void (**elementary_processes)(struct CudaCALModel2D* ca2D);
65          //!< Number of function pointers to the transition functions's elementary
                processes callbacks.
66          int num_of_elementary_processes;
67  };
```

Codice 5.2: La rappresentazione del modello in OpenCAL-CUDA.

Per il caso delle variabili scalari e per i processi elementari invece, il codice è rimasto sostanzialmente uguale.

Questa prima parte di studio ha evidenziato come la differenza di architettura e il passaggio di dati tra la memoria device e host possano essere determinanti sia in termini di performance che di sviluppo del progetto. Non è l'unico caso in cui si è dovuto ricorrere ad un codice adattato per tradurre il codice sequenziale in parallelo.

5.2.2 CALRun2D e CudaCALRun2D

Rispetto a CALModel2D, la struct CALRun2D ha subito meno cambiamenti nella versione parallela della libreria, sia perché c'erano meno punti critici sia perché la maggior parte dei cambiamenti sono dovuti ad aggiunte di strutture dati. La loro implementazione è rappresentata dai codici 5.3 e 5.4 rispettivamente per CALRun2D e CudaCALRun2D.

Naturalmente nella versione parallela il modello è presente all'interno della simulazione, così come accade per CALRun2D, e in particolare troviamo tre diversi modelli.

Da premettere che, come descritto nel diagramma in fig. 5.1 tutta la parte di simulazione avviene lato device. Questo comporta dunque la presenza dei dati del modello sul device che spiega la presenza dunque di un modello in più (device_ca2D). La presenza di h_device_ca2D invece è richiesta per le operazioni di trasferimento dati tra l'host e il device. I dettagli implementativi relativi a questa scelta progettuale verranno spiegati nel paragrafo successivo (5.2.3). Naturalmente h_device_ca2D non incide assolutamente sull'utilizzo di OpenCAL, infatti l'utente non verrà mai a conoscenza della sua presenza poiché è utilizzata solo nel core della libreria. Invece device_ca2D comporta una piccola modifica di utilizzo di OpenCAL ed è l'utente stesso che deve dichiararla nel main principale e lasciare il compito della definizione alla libreria aggiungendola come parametro alla funzione calCudaRunDef2D.

Vediamo insieme ora le due diverse implementazioni della struct dedicata alla simulazione:

```
1   struct CALRun2D
2   {
3           //!< Pointer to the cellular automaton structure.
4           struct CALModel2D* ca2D;
5
6           //!< Current simulation step.
7           int step;
8
9           //!< Initial simulation step.
10          int initial_step;
11
12          //!< Final simulation step; if 0 the simulation becomes a loop.
13          int final_step;
14
15          //!< Callbacks substates' update mode; it can be CAL_UPDATE_EXPLICIT or
                CAL_UPDATE_IMPLICIT.
16          enum CALUpdateMode UPDATE_MODE;
17
18          //!< Simulation's initialization callback function.
19          void (*init)(struct CALModel2D*);
```

```
20
21          //!< CA's globalTransition callback function. If defined, it is executed
                instead of cal2D.c::calGlobalTransitionFunction2D.
22          void (*globalTransition)(struct CALModel2D*);
23
24          //!< Simulation's steering callback function.
25          void (*steering)(struct CALModel2D*);
26
27          //!< Simulation's stopCondition callback function.
28          CALbyte (*stopCondition)(struct CALModel2D*);
29
30          //!< Simulation's finalize callback function.
31          void (*finalize)(struct CALModel2D*);
32  };
```

Codice 5.3: La rappresentazione del modello in OpenCAL.

```
1   struct CudaCALRun2D
2   {
3           //!< Pointer to the cellular automaton structure.
4           struct CudaCALModel2D* ca2D;
5
6           //!< Pointer to the cellular automaton structure on device.
7           struct CudaCALModel2D* device_ca2D;
8
9           //!< Pointer to the cellular automaton structure for data passing.
10          struct CudaCALModel2D* h_device_ca2D;
11
12          //!< Stream compaction data structure
13          unsigned int * device_array_of_index_dim;
14
15          //!< Current simulation step.
16          int step;
17
18          //!< Initial simulation step.
19          int initial_step;
20
21          //!< Final simulation step; if 0 the simulation becomes a loop.
22          int final_step;
23
24          //!< Callbacks substates' update mode; it can be CAL_UPDATE_EXPLICIT or
                CAL_UPDATE_IMPLICIT.
25          enum CALUpdateMode UPDATE_MODE;
26
27          //!< Simulation's initialization callback function.
28          void (*init)(struct CudaCALModel2D*);
29
30          //!< CA's globalTransition callback function. If defined, it is executed
                instead of cal2D.c::calGlobalTransitionFunction2D.
31          void (*globalTransition)(struct CudaCALModel2D*);
32
33          //!< Simulation's steering callback function.
34          void (*steering)(struct CudaCALModel2D*);
35
36          //!< Simulation's stopCondition callback function.
37          void (*stopCondition)(struct CudaCALModel2D*);
38
39          //!< Simulation's finalize callback function.
40          void (*finalize)(struct CudaCALModel2D*);
41  };
```

Codice 5.4: La rappresentazione del modello in OpenCAL-CUDA.

5.2.3 Trasferimento dei dati tra Host e Device

Il trasferimento dei dati utili alla computazione tra GPU e CPU è sempre stato uno dei punti critici del parallelismo su dispositivi grafici. Perciò negli anni le architetture hanno sviluppato diverse tecniche performanti per migliorare questo aspetto. In CUDA

C utilizzare le funzioni fornite dall'API è molto conveniente perché sono ottimizzate. La conferma la si può benissimo trovare nel Visual Profiler (2.4.2) con dei semplici toy-problems.

Nel caso di OpenCAL-CUDA il trasferimento dei dati è stato più complesso del previsto. Come accennato in precedenza, le sole API di CUDA non sono bastate per trasferire un modello tra la GPU e la CPU. Per questo è stato utilizzata una procedura ad hoc per questo tipo di trasferimento.

Il problema principale del trasferimento dei dati da host a device è stato il passaggio di strutture dati dichiarate tramite puntatori. In generale lato host non si può accedere a blocchi di memoria su device e viceversa. Dunque copiare l'indirizzo di un puntatore non era la scelta corretta.

Per copiare un puntatore da host a device in CUDA bisogna copiare il contenuto della struttura dati puntata, all'interno di una nuova struttura dati allocata correttamente sul device. Il modello h_device_ca2D dichiarato all'interno di CudaCALRun2D ha il compito di fare da intermediario tra l'host e il device. Cioè, è un *oggetto* allocato sulla CPU ma con i puntatori a strutture dati (vicinato, sottostati etc.) allocati sul device. Se vogliamo copiare un modello di automa cellulare da host a device eseguiamo i seguenti passi:

1. Allocare e definire (popolare) sull'host un oggetto CudaCALModel2D (chiamiamolo **host_model**)
2. Allocare su device un oggetto CudaCALModel2D (chiamiamolo **device_model**)
3. Allocare su host un oggetto CudaCALModel2D con i puntatori alle strutture dati allocati sul device (chiamiamolo **ibrid_model**)
4. Copiare con una semplice memcpy le variabili scalari di host_model su ibrid_model.
5. Copiare, utilizzando la funzione fornita dalle API di Cuda cudaMemCpy, il contenuto delle strutture dati (gestite tramite puntatori) di host_model su ibrid_model
6. Copiare, utilizzando la funzione fornita dalle API di Cuda cudaMemCpy, tutto l'oggetto ibrid_model su device_model

Il processo sembra senza dubbio tortuoso ma in realtà è un passo obbligato se si vogliono utilizzare struct di questo genere. Il perché del suo funzionamento è semplice e lo insegnano gli errori di compilazione incontrati durante la fase implementativa.

Ad esempio se provassimo a copiare l'indice i del vicinato presente in CudaCALModel2D direttamente da host_model a device_model ci si troverebbe davanti il seguente codice:

```
1 //Using cudaMemcpy
2 cudaMemcpy(device_model->i,host_model->i, sizeof(CALint)*model->sizeof_X,
      cudaMemcpyHostToDevice);
```

Questo codice tuttavia risulta sbagliato poiché da host stiamo cercando di accedere direttamente alla memoria sul device (device_model->i) poiché device_model è allocato interamente su device. E' per questo che torna utile l'utilizzo di una struttura intermedia accennata in precedenza (ibrid_model). Una copia corretta del codice 5.2.3 potrebbe essere:

```
1 //Using cudaMemcpy
2 cudaMemcpy(ibrid_model->i,host_model->i, sizeof(CALint)*model->sizeof_X,
      cudaMemcpyHostToDevice);
3
4 //Whole model passed between host and device
5 cudaMemcpy(ibrid_model,host_model, sizeof(CudaCALModel2D), cudaMemcpyHostToDevice);
```

In questo modo c'è la certezza che non si tenta di accedere sul device dal codice compilato lato host e la copia va a buon fine. In particolare la seconda copia va a buon fine poiché quando cudaMemcpy andrà a copiare l'intero oggetto adesso può benissimo copiare l'indirizzo dei puntatori tra le due struct poiché entrambi gli indirizzi sono allocati sul device.

Un esempio completo della copia del modello da host a device è mostrato nel codice 5.2.3

```
CALbyte calInitializeInGPU2D(struct CudaCALModel2D* model, struct CudaCALModel2D *
    d_model){

        CALbyte result = CAL_TRUE;

        calCudaAllocatorModel(model);

        cudaMemcpy(copy_model->i,model->i, sizeof(CALint)*model->sizeof_X,
            cudaMemcpyHostToDevice);
        cudaMemcpy(copy_model->j,model->j, sizeof(CALint)*model->sizeof_X,
            cudaMemcpyHostToDevice);

        if(model->OPTIMIZATION == CAL_OPT_ACTIVE_CELLS){
                cudaMemcpy(copy_model->activecell_flags,model->activecell_flags,
                    sizeof(CALbyte)*model->rows*model->columns, cudaMemcpyHostToDevice
                    );
                cudaMemcpy(copy_model->activecell_index,model->activecell_index,
                    sizeof(CALint)*model->rows*model->columns, cudaMemcpyHostToDevice)
                    ;
                cudaMemcpy(copy_model->array_of_index_result,model->
                    array_of_index_result, sizeof(CALint)*model->rows*model->columns,
                    cudaMemcpyHostToDevice);
        }

        if(model->sizeof_pQb_array > 0){
                cudaMemcpy(copy_model->pQb_array_current,model->pQb_array_current,
                    model->sizeof_pQb_array*model->rows*model->columns*sizeof(CALbyte)
                    , cudaMemcpyHostToDevice);
                cudaMemcpy(copy_model->pQb_array_next,model->pQb_array_next, model->
                    sizeof_pQb_array*model->rows*model->columns*sizeof(CALbyte),
                    cudaMemcpyHostToDevice);
        }
        if(model->sizeof_pQi_array > 0){
                cudaMemcpy(copy_model->pQi_array_current,model->pQi_array_current,
                    model->sizeof_pQi_array*model->rows*model->columns*sizeof(CALint),
                     cudaMemcpyHostToDevice);
                cudaMemcpy(copy_model->pQi_array_next,model->pQi_array_next, model->
                    sizeof_pQi_array*model->rows*model->columns*sizeof(CALint),
                    cudaMemcpyHostToDevice);
        }
        if(model->sizeof_pQr_array > 0){
                cudaMemcpy(copy_model->pQr_array_current,model->pQr_array_current,
                    model->sizeof_pQr_array*model->rows*model->columns*sizeof(CALreal)
                    , cudaMemcpyHostToDevice);
                cudaMemcpy(copy_model->pQr_array_next,model->pQr_array_next, model->
                    sizeof_pQr_array*model->rows*model->columns*sizeof(CALreal),
                    cudaMemcpyHostToDevice);
        }
        cudaMemcpy(d_model, copy_model, sizeof(struct CudaCALModel2D),
            cudaMemcpyHostToDevice);

        return result;
}
```

Si può notare che il processo inverso (da device a host) è del tutto simile e applica la seguente procedura al contrario:

```
CALbyte calSendDataGPUtoCPU(struct CudaCALModel2D* model, struct CudaCALModel2D *
    d_model){

        CALbyte result = CAL_TRUE;
```

```
4
5           cudaMemcpy(copy_model, d_model, sizeof(struct CudaCALModel2D),
                cudaMemcpyDeviceToHost);
6
7           if(model->sizeof_pQb_array > 0){
8                   cudaMemcpy(model->pQb_array_current,copy_model->pQb_array_current,
                        model->sizeof_pQb_array*model->rows*model->columns*sizeof(CALbyte)
                        , cudaMemcpyDeviceToHost);
9                   cudaMemcpy(model->pQb_array_next,copy_model->pQb_array_next,model->
                        sizeof_pQb_array*model->rows*model->columns*sizeof(CALbyte),
                        cudaMemcpyDeviceToHost);
10          }
11          if(model->sizeof_pQi_array > 0){
12                  cudaMemcpy(model->pQi_array_current,copy_model->pQi_array_current,
                        model->sizeof_pQi_array*model->rows*model->columns*sizeof(CALint),
                         cudaMemcpyDeviceToHost);
13                  cudaMemcpy(model->pQi_array_next,copy_model->pQi_array_next,model->
                        sizeof_pQi_array*model->rows*model->columns*sizeof(CALint),
                        cudaMemcpyDeviceToHost);
14          }
15          if(model->sizeof_pQr_array > 0){
16                  cudaMemcpy(model->pQr_array_current,copy_model->pQr_array_current,
                        model->sizeof_pQr_array*model->rows*model->columns*sizeof(CALreal)
                        , cudaMemcpyDeviceToHost);
17                  cudaMemcpy(model->pQr_array_next,copy_model->pQr_array_next,model->
                        sizeof_pQr_array*model->rows*model->columns*sizeof(CALreal),
                        cudaMemcpyDeviceToHost);
18          }
19
20          calCudaFinalizeModel();
21
22          return result;
23  }
```

L'oggetto *copy_model* presente nel codice sarebbe il nostro ibrid_model dichiarato nello stesso file. In particolare il metodo calCudaAllocatorModel (cod. 5.2.3)prende in input il modello e ne copia gli scalari in copy_model, allocando in seguito tutti i puntatori all'interno sul device in modo da avere l'oggetto pronto alla nostra copia tra host e device.

In seguito si mostra il contenuto della funzione calCudaAllocatorModel:

```
1   struct CudaCALModel2D* calCudaAllocatorModel(struct CudaCALModel2D *model){
2
3           cudaMallocHost((void**)&copy_model, sizeof(struct CudaCALModel2D),
                cudaHostAllocPortable);
4
5           memcpy(copy_model,model,sizeof(struct CudaCALModel2D));
6
7           cudaMalloc((void**)&copy_model->i,model->sizeof_X*sizeof(int));
8           cudaMalloc((void**)&copy_model->j,model->sizeof_X*sizeof(int));
9
10          if(model->OPTIMIZATION == CAL_OPT_ACTIVE_CELLS){
11                  cudaMalloc((void**)&copy_model->activecell_flags,model->rows*model->
                        columns*sizeof(CALbyte));
12                  cudaMalloc((void**)&copy_model->activecell_index,model->rows*model->
                        columns*sizeof(CALint));
13                  cudaMalloc((void**)&copy_model->array_of_index_result, model->rows*
                        model->columns*sizeof(CALint));
14          }
15
16          if(model->sizeof_pQb_array > 0){
17                  cudaMalloc((void**)&copy_model->pQb_array_current,model->
                        sizeof_pQb_array*model->rows*model->columns*sizeof(CALbyte));
18                  cudaMalloc((void**)&copy_model->pQb_array_next,model->sizeof_pQb_array
                        *model->rows*model->columns*sizeof(CALbyte));
19          }
20          if(model->sizeof_pQi_array > 0){
21                  cudaMalloc((void**)&copy_model->pQi_array_current,model->
                        sizeof_pQi_array*model->rows*model->columns*sizeof(CALint));
22                  cudaMalloc((void**)&copy_model->pQi_array_next,model->sizeof_pQi_array
                        *model->rows*model->columns*sizeof(CALint));
23          }
```

```
24          if(model->sizeof_pQr_array > 0){
25                  cudaMalloc((void**)&copy_model->pQr_array_current,model->
                        sizeof_pQr_array*model->rows*model->columns*sizeof(CALreal));
26                  cudaMalloc((void**)&copy_model->pQr_array_next,model->sizeof_pQr_array
                        *model->rows*model->columns*sizeof(CALreal));
27          }
28
29          return copy_model;
30  }
```

5.2.4 L'ottimizzazione delle celle attive

L'ottimizzazione è una keyword che si utilizza spesso quando si parla di parallelismo e tecniche di performance. Gli automi cellulari hanno diverse tecniche di ottimizzazione. Come ben sappiamo infatti alcuni di loro richiedono un tempo computazionale elevato, ragion per cui, nel tempo, sono state ideate delle tecniche di minimizzazione della complessità.

Una tecnica molto utilizzata e perfomante è la tecnica delle celle attive. Una cella si dice attiva quando non si trova in uno stato "quiescente" cioè ad un determinato tempo t la funzione di transizione cambia uno o più stati della cella.

Un esempio banale potrebbe essere una colata lavica rappresentata da un automa cellulare. Uno degli stati di una cella potrebbe essere la quantità di lava che deve ancora *colare*. Tutte le celle che contengono lava possono essere definite **celle attive**. Le funzioni di transizione spesso possono essere realmente complesse e richiedono un elevato tempo computazionale. Grazie a questa tecnica di ottimizzazione si può supporre che la funzione di transizione viene eseguita solamente dalle celle in cui avviene un'evoluzione al tempo t e nelle loro vicine.

Pensate bene che nel caso di una colata lavica una morfologia può generare una matrice con migliaia di celle tra cui nei primi step della computazione solo poche celle attive. Avviando la funzione di transizione solo per le celle realmente attive si può guadagnare dunque tanto carico di lavoro.

Quello descritto è un approccio intuitivo e se entriamo nei dettagli implementativi possiamo notare come questa ottimizzazione comporta l'aggiunta di strutture dati a supporto. Una di queste, la più importante, è sicuramente la matrice di FLAGS. Quest'ultima possiamo rappresentarla come una matrice booleana in cui il valore TRUE nella posizione *i-esima* sta a significare che la cella è attiva, FALSE altrimenti. Per poter accedere in scrittura alla matrice di flags in un programma parallelo bisogna considerare l'utilizzo dell'esclusività in quanto bisogna mantenere la lista di celle attive in uno stato coerente. In CUDA l'esclusività è gestita dalle funzioni atomiche descritte nel capitolo 2.

Capiamo bene che per stilare una lista delle celle in cui deve avvenire la computazione allo step successivo bisogna obbligatoriamente scorrere la matrice di flags e vedere quali sono attive e quali no. Scorrere tutta la matrice di flags però può diventare oneroso in termini di performance. Per questo esiste un'altra tecnica subordinata chiamata **stream compaction**[3].

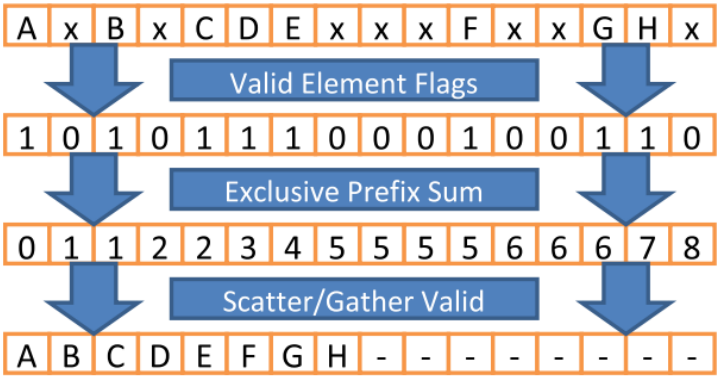


Figura 5.2: Esempio di stream compaction

Gli algoritmi di stream compaction attraverso semplici passaggi rimuovono gli elementi non utili da un insieme di dati sparsi. Nel nostro caso, questo genere di algoritmi, prendono in input la matrice di flags e restituiscono in output un array con in testa le celle in cui il flag è "true". In particolare, l'algoritmo si porta dietro l'indice delle celle attive poiché identifica la posizione in cui andare a lanciare la funzione di transizione al passo successivo.

In OpenCAL-CUDA è stata implementata tramite l'utilizzo della libreria chag creata appunto dagli autori dell'articolo scientifico "Efficient Stream Compaction" citato in precedenza [3] [2].

Inizialmente si era utilizzata la libreria **thrust** [5] ma dopo una serie di test risultava inefficiente per il nostro tipo di problema e dunque è stata scartata a favore della libreria chag::pp.

Mostriamo ora l'utilizzo dell'algoritmo di stream compaction all'interno della libreria:

```
1  struct Predicate
2  {
3          __host__ __device__
4                  bool operator()(unsigned int x) const
5          {
6                  return (x != -1);
7          }
8  };
9
10 /* ... */
11
12 if (simulation->ca2D->OPTIMIZATION == CAL_OPT_ACTIVE_CELLS){
13
14                 CALint SIZE = (simulation->ca2D->rows*simulation->ca2D->columns);
15
16                 if(simulation->ca2D->activecell_size_current != simulation->
                       h_device_ca2D->activecell_size_next){
17
18                         generateSetOfIndex<<<grid,block>>>(simulation->device_ca2D);
19
20                         cudaMemcpy(simulation->h_device_ca2D, simulation->device_ca2D,
                               sizeof(struct CudaCALModel2D), cudaMemcpyDeviceToHost);
21
22                         pp::compact(
23                                 /* Input start pointer */
24                                 simulation->h_device_ca2D->activecell_index,
25
26                                 /* Input end pointer */
27                                 simulation->h_device_ca2D->activecell_index+SIZE,
```

```
28
29                                  /* Output start pointer */
30                                  simulation->h_device_ca2D->array_of_index_result,
31
32                                  /* Storage for valid element count */
33                                  simulation->device_array_of_index_dim,
34
35                                  /* Predicate */
36                                  Predicate()
37                                  );
38
39                          cudaMemcpy(simulation->device_ca2D, simulation->h_device_ca2D,
                                sizeof(struct CudaCALModel2D), cudaMemcpyHostToDevice);
40
41                  }
42
43
44                  // resize of grid and blocks.
45                  simulation->ca2D->activecell_size_current = simulation->h_device_ca2D
                        ->activecell_size_next;
46                  CALint num_blocks = simulation->ca2D->activecell_size_current / ((
                        block.x) * (block.y));
47                  grid.x = (num_blocks+2);
48                  grid.y = 1;
49
50                  //elementary process run
51                  elementary_process<<<grid,block>>>(simulation->device_ca2D);
52          }
```

Codice 5.5: Stream compaction in OpenCAL-CUDA

Seguendo step by step il codice si mostra come l'algoritmo viene utilizzato solamente nel caso in cui l'utente abbia scelto di ottimizzare la simulazione tramite la tecnica delle celle attive. Nel caso in cui le celle attive si aggiornano viene avviato un kernel che ha il compito di aggiornare la lista di flags trasformandoli in indici. Questo perché all'algoritmo interessa quali sono gli indici (in matrici lineari) delle celle attive in modo da sapere su quali celle deve essere lanciata la funzione di transizione allo step successivo. La funzione cruciale è pp::compact{...} che prende in input:

1. Il range di celle dell'array di indici sparsi (aggiornato dal kernel descritto in precedenza)

2. L'array di output utile per ricavare le informazioni finali

3. La dimensione dell'array

4. Una funzione predicato

La funzione predicato è definita da una struct eseguendo un override dell'operatore (). Il predicato definisce la regola secondo cui l'algoritmo deve compattare la matrice. Nel nostro caso se trova nell'array un valore diverso da -1 lo inserisce al primo posto disponibile in testa, altrimenti va avanti.

L'utilizzo di questa tecnica ha portato ad un guadagno del 30% delle performance.

5.3 Struttura di OpenCAL-CUDA

In questa sezione vedremo insieme come si utilizza in maniera completa la libreria OpenCAL-CUDA.

5.3.1 Il *main*

L'obiettivo principale di OpenCAL-CUDA era quello di fornire una versione di OpenCAL completamente parallela mantenendo la stessa struttura, in modo da rendere il parallelismo completamente trasparente all'utente. Nonostante ciò, date alcune circostanze e la differenza di architetture, l'utilizzo della libreria è leggermente cambiato in base all'esigenze dell'architettura CUDA.

Per poter utilizzare la libreria OpenCAL-CUDA possiamo stabilire i seguenti passi:

- Definizione e allocazione del modello e della simulazione
- Definizione e allocazione dei kernel e dei sottostati
- Trasferimento dei dati dall'host al device
- Definizione del ciclo di esecuzione
- Trasferimento dei dati dal device all'host
- Operazioni di finalizzazione

Come si può notare è presente qualche passo in più rispetto all'utilizzo della versione sequenziale della libreria. Questo per consentire il passaggio di dati tra le memorie poiché la simulazione avviene totalmente su GPU.

Come ricordato in precedenza, date alcune limitazioni nel passaggio di memoria (5.2.3), si è perso un livello di astrazione rispetto ad OpenCAL. Questo ha comportato dei leggeri cambiamenti anche in alcune funzioni utilizzate nella libreria. Possiamo portare un esempio:

calAddSubstate(b|i|r) come mostrato nel capitolo 4 si implementava nel seguente modo:

```
1  int main()
2  {
3
4          /* ...  */
5
6          // add substates
7          substate1 = calAddSubstate2Db(model);
8          substate2 = calAddSubstate2Di(model);
9          substate3 = calAddSubstate2Dr(model);
10
11         // load substate from file
12         calLoadSubstate2Db(model, substate1, PATH_substate1);
13         calLoadSubstate2Dr(model, substate3, PATH_substate3);
14
15         /* ... */
16
17 }
```

In OpenCAL-CUDA l'implementazione cambia leggermente. Poiché i sottostati sono rappresentati da matrici lineari (una per ogni tipo supportato, *byte, integer, real...*), calCudaAddSubstate risulta leggermente diverso rispetto alla sua versione sequenziale. Vediamone un esempio:

```
1  int main()
2  {
3
4          /* ...  */
5
6          //add substates
7          calCudaAddSubstate2Dr(model, NUMBER_OF_SUBSTATES_REAL);
8          calCudaAddSubstate2Db(model, NUMBER_OF_SUBSTATES_BYTE);
```

```
9
10          //load configuration
11          calCudaLoadSubstate2Dr(model, SUBSTATE1_PATH, SUBSTATE1_INDEX);
12          calCudaLoadSubstate2Dr(model, SUBSTATE2_PATH, SUBSTATE2_INDEX);
13          calCudaLoadSubstate2Db(model, SUBSTATE3_PATH, SUBSTATE3_INDEX);
14
15          /* ... */
16
17  }
```

Con questa nuova versione la chiamata alla funzione è unica per ogni tipo di dato. Dunque l'allocazione in memoria viene effettuata una sola volta per tutti i sottostati di tipo uguale. La prima differenza sta nel fatto che l'utente deve stabilire a priori il numero di sottostati per ogni tipo che desidera. Questo non dovrebbe essere un grosso problema poiché si suppone che l'utente che decide di utilizzare OpenCAL ha già bene in mente quale sia il suo modello e già possiede queste informazioni. Nonostante ciò rimane libero di cambiare le sue informazioni in qualsiasi momento. Una seconda differenza sta nelle funzioni di *load*. Siccome la nostra struttura dati è ora una matrice lineare, ogni sottostato possiede un indice in modo tale da conoscere qual è il range di memoria che occupa. Questo per facilità può essere rappresentato da un enumerativo che rende il codice abbastanza chiaro e lineare.

Per quanto riguarda invece le operazioni di trasferimento dei dati, sono gestite automaticamente dalla libreria grazie a due funzioni: calInitializeInGPU2D e calSendDataGPUtoCPU. Queste due funzioni prendono in input il modello allocato sull'host e il modello allocato sul device e rispettivamente trasferiscono i dati da CPU a GPU e viceversa.

Un'altra aggiunta naturalmente riguarda le funzioni relative alla simulazione dell'automa cellulare. Mentre la definizione delle funzioni di inizializzazione e supporto rimangono sostanzialmente uguali alla versione sequenziale, la funzione calRun2D ha subito un leggero cambiamento. Come ben sappiamo CUDA utilizza una serie di threads suddivisi in griglie e blocchi. In OpenCAL-CUDA lasciamo al libertà all'utente di gestire questa configurazione a patto che il core della libreria venga informata della scelta. Per questo due valori di tipo dim3 devono essere incluse tra gli input della funzione calCudaRun2D.

Ecco un confronto tra la versione sequenziale e quella parallela:

```
1   /* ... */
2
3   CALint N = 16;
4   CALint M = 61;
5   dim3 block(N,M);
6   dim3 grid(COLS/block.x, ROWS/block.y);
7
8   /* ... */
9
10
11  //Start simulation in OpenCAL
12  calRun2D(simulation);
13
14  //Start simulation in OpenCAL-CUDA
15  calCudaRun2D(simulation, grid, block);
16
17  /* ... */
```

5.3.2 La dichiarazione dei *kernel*

I kernel per OpenCAL-CUDA sono tutte le funzioni che devono eseguire codice parallelo. La libreria richiede che la funzione di inizializzazione, la funzione di steering, la funzione di stop e i processi elementari devono essere definite come kernel. Questo perché sono le funzioni che verranno avviate in parallelo dalla libreria attraverso l'architettura CUDA.

Questa tipologia di funzioni sono dichiarate in maniera del tutto simile alla versione sequenziale ma con l'aggiunta della keyword __global__ che identifica un kernel. All'interno di queste funzioni l'utente deve progettare l'algoritmo parallelo a suo piacimento mettendo in pratica i concetti base di CUDA. OpenCAL-CUDA fornisce delle comode funzioni per ricevere delle informazioni molto utilizzate nei programmi scritti in CUDA C. Ad esempio, capita spesso che un programmatore debba ricavarsi le informazioni riguardo l'ID univoco dei threads. Questo comporta piccoli calcoli matematici che a lungo andare possono diventare annoianti e ripetitivi, inoltre ci si può imbattere in piccoli errori di calcolo. Per questo la libreria OpenCAL-CUDA esegue tutte queste operazioni di routine in automatico tramite alcune chiamate a funzione.

Mostriamo un esempio di processo elementare implementato tramite la libreria OpenCAL-CUDA:

```
__global__ void elementary_process(struct CudaCALModel2D* model)
{
        CALreal value;
        CALint n, offset = calCudaGetOffset();

        value = calCudaGet2Dr(model, offset, SUBSTATE1_INDEX);

        value += calCudaGetX2Dr(model, offset, n, SUBSTATE2_INDEX)
                - calCudaGet2Dr(model, offset, SUBSTATE3_INDEX);

        calCudaSet2Dr(model, offset, value, SUBSTATE1_INDEX);
}
```

5.4 Game of Life in OpenCAL-CUDA

Come descritto nel paragrafo 4.3 il Game of Life è il più famoso automa cellulare. Per questo possiamo prenderlo da esempio per la sua semplicità e la sua chiarezza. Vediamone insieme una sua implementazione tramite la libreria OpenCAL-CUDA.

```
#include ".\include\cal2D.cuh"
#include ".\include\cal2DIO.cuh"
#include ".\include\cal2DToolkit.cuh"
#include ".\include\cal2DRun.cuh"
#include <stdlib.h>
#include <time.h>

#include <iostream>
using namespace std;

#include "cuda_profiler_api.h"
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

//------------------------------------------------------------------------
//   THE GOL (Toy model) CELLULAR AUTOMATON
//------------------------------------------------------------------------

#define ROWS 1000
#define COLS 1000

#define STEPS 200
#define STEPS_THRESHOLD 200
#define CONFIGURATION_PATH "./data/map_1000x1000.txt"
#define OUTPUT_PATH "./data/final.txt"

#define NUMBER_OF_SUBSTATE_BYTE 1
#define NUMBER_OF_NEIGHBORHOOD 9

enum SUBSTATE_NAME{
```

```
31          ALIVE = 0
32  };
33
34  struct CudaCALRun2D* gol_simulation;
35
36  CALint N = 25;
37  CALint M = 5;
38  dim3 block(N,M);
39  dim3 grid(COLS/block.x, ROWS/block.y);
40
41  __global__ void gol_computation(struct CudaCALModel2D* gol)
42  {
43          CALint sum = 0, n, offset = calCudaGetIndex(gol);
44
45
46          CALint i = calCudaGetIndexRow(gol, offset),j = calCudaGetIndexColumn(gol,
                offset);
47
48          CALbyte myState = calCudaGet2Db(gol,offset, ALIVE);
49
50          for(n=1; n<NUMBER_OF_NEIGHBORHOOD; n++){
51                  sum += calCudaGetX2Db(gol,offset,n, ALIVE);
52          }
53
54          if(myState == CAL_TRUE){
55                  if(sum != 2 && sum != 3){
56                          calCudaSet2Db(gol, offset, CAL_FALSE, ALIVE);
57                  }
58          }else{
59                  if(sum == 3){
60                          calCudaSet2Db(gol, offset, CAL_TRUE, ALIVE);
61                  }
62          }
63  }
64
65  __global__ void gol_simulation_init(struct CudaCALModel2D* gol)
66  {
67          CALint offset = calCudaGetIndex(gol);
68
69          //initializing substates to 0
70          calCudaInit2Db(gol,offset,CAL_FALSE, ALIVE);
71
72          // Glider 1000x1000
73          if(offset == 1002 || offset == 2003 || offset == 3001 || offset == 3002 ||
                offset == 3003){
74                  calCudaInit2Db(gol,offset,CAL_TRUE, ALIVE);
75          }
76
77  }
78
79
80  __global__ void gol_simulation_stop(struct CudaCALModel2D* gol)
81  {
82          CALint offset = calCudaGetIndex(gol);
83          CALint i = calCudaGetIndexRow(gol, offset), j = calCudaGetIndexColumn(gol,
                offset);
84
85          if(i == 14 && j == 9 && calCudaGet2Db(gol,offset,0))
86                  calCudaStop(gol);
87
88  }
89
90  int main()
91  {
92          time_t start_time, end_time;
93          cudaProfilerStart();
94
95          //cadef
96          struct CudaCALModel2D* gol = calCudaCADef2D (ROWS, COLS,
                CAL_MOORE_NEIGHBORHOOD_2D, CAL_SPACE_TOROIDAL, CAL_NO_OPT);
97          struct CudaCALModel2D* device_gol = calCudaAlloc();
98
99          //rundef
100         gol_simulation = calCudaRunDef2D(device_gol, gol, 1, CAL_RUN_LOOP,
```

```
            CAL_UPDATE_IMPLICIT);

        //add transition function's elementary processes
        calCudaAddElementaryProcess2D(gol, gol_computation);

        printf ("Starting alloc...\n");
        start_time = time(NULL);

        //add substates
        calCudaAddSubstate2Db(gol, NUMBER_OF_SUBSTATE_BYTE);

        //load configuration
        calCudaLoadSubstate2Db(gol, CONFIGURATION_PATH, ALIVE);

        //send data to GPU
        calInitializeInGPU2D(gol, device_gol);

        end_time = time(NULL);
        printf ("Alloc terminated.\nElapsed time: %d\n\n", end_time-start_time);

        cudaErrorCheck("Data initialized on device\n");

        //simulation configuration
        calCudaRunAddInitFunc2D(gol_simulation, gol_simulation_init);
        calCudaRunAddStopConditionFunc2D(gol_simulation,gol_simulation_stop);

        printf ("Starting simulation...\n");
        start_time = time(NULL);

        //simulation run
        calCudaRun2D(gol_simulation,grid,block);

        //send data to CPU
        calSendDataGPUtoCPU(gol,device_gol);

        cudaErrorCheck("Final configuration sent to CPU\n");

        end_time = time(NULL);
        printf ("Simulation terminated.\nElapsed time: %d\n\n", end_time-start_time);

        cudaProfilerStop();

        //saving configuration
        calCudaSaveSubstate2Db(gol, OUTPUT_PATH, ALIVE);

        cudaErrorCheck("Data saved on output file\n");

        //finalizations
        calCudaRunFinalize2D(gol_simulation);
        calCudaFinalize2D(gol, device_gol);

        system("pause");
        return 0;
}
```

Quello mostrato è il classico esempio di implementazione di un modello e una simulazione in OpenCAL-CUDA. All'inizio del programma troviamo tutte le informazioni relative alle strutture dati, path dei file di configurazione e stampa, librerie incluse etc.

Da notare l'enumerativo SUBSTATE_NAME, utile per accedere alla matrice linearizzata dei sottostati (in questo caso di byte). Prima della dichiarazione del main troviamo tutti i kernel utili ai fini della simulazione. Questi sono implementati come delle normali funzioni con la differenza che il codice viene eseguito in parallelo da migliaia di threads. Una delle funzioni di supporto descritte nel paragrafo precedente è calCudaGetOffset che ha il compito di restituire l'ID univoco per ogni thread che accede al kernel corrente. In questo caso sono state utilizzate altre due funzioni di supporto: calCudaGetIndexRow e calCudaGetIndexColumn. Queste vengono utilizzate per risalire ai più comuni indici i e j di una matrice a partire dalla matrice linearizzata e dall'ID univoco del thread. Sono

state implementate poiché spesso ci si trova a dover gestire un determinato angolo di celle nella loro evoluzione e l'utilizzo di indici può essere molto utile.

Un ultimo commento è relativo alla leggibilità del codice che nonostante l'utilizzo della GPGPU programming è rimasto molto chiarp e ridotto. Questo può essere visto come un enorme potenzialità della libreria che evita dunque la pesantezza di leggibilità del codice parallelo per GPU.

5.5 "Modello" in OpenCAL-CUDA

Capitolo 6

Conclusioni

Ringraziamenti

Bibliografia

[1] Grama Ananth et al. *Introduction to Parallel Computing, Second Edition*. Addison Wesley, 2003.

[2] Markus Billeter, Ola Olsson e Ulf Assarsson. *Chag::pp website*. `http://www.cse.chalmers.se/~billeter/pub/pp`. [Online; accessed 09-April-2015]. 2009.

[3] Markus Billeter, Ola Olsson e Ulf Assarsson. «Efficient stream compaction on wide SIMD many-core architectures». In: *Proceedings of the Conference on High Performance Graphics 2009*. ACM. 2009.

[4] NVIDIA Corporation. *CUDA toolkit documentation*. `http://docs.nvidia.com/cuda/`. [Online; accessed 12-April-2015].

[5] NVIDIA Corporation. *Thrust documentation website*. `http://docs.nvidia.com/cuda/thrust/`. [Online; accessed 12-April-2015].

[6] Nvidia Corporation. *CUDA Dynamic Parallelism, Programming guide*.

[7] Nvidia Corporation. *GPU-Accelerated applications*. 2015.

[8] Nvidia Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*.

[9] D'Ambrosio D. «Automi Cellulari nella modellizzazione di fenomeni complessi macroscopici e loro ottimizzazione con Algoritmi Genetici». Tesi di dott. Universita' della Calabria, 2003.

[10] Martin Gardner. «Mathematical games: The fantastic combinations of John Conway's new solitaire game "life"». In: *Scientific American* (1970).

[11] Spezzano Giandomenico et al. «A parallel cellular tool for interactive modeling and simulation». In: *Computing in Science and Engineering* (1996).

[12] Crisci G.M. et al. «The simulation model SCIARA: the 1991 and 2001 at Mount Etna». In: *Journal of Vulcanogy and Geothermal Research* (2004).

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

[14] Jaroslaw Was Pawel Kleczek. «Simulation of Pedestrians Behavior in a Shopping Mall». In: (2014).

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

[16] Di Gregorio S. e Trautteur. G. «On reversibility in Cellular Automata». In: *Journal of Computer and System Science* (1975).

[17] Di Gregorio S. et al. «A microscopic freeway traffic simulator on a highly parallel system». In: *Parallel Computing: State-of-the-Art and Perspectives* (1996).

[18] Di Gregorio S. et al. «Mount Ontake landslide simulation by the cellular automata model SCIDDICA-3». In: *Physics and Chemistry of the Earth* (1999,).

[19] Wolfram S. *A new kind of Science*. Wolfram Media Inc, 2002.

[20] Wolfram S. «Statistical mechanics of cellular automata». In: *Reviews of Modern Physics* (1983).

[21] Giuseppe A Trunfio. «Predicting wildfire spreading through a hexagonal cellular automata model». In: *Cellular Automata*. Springer, 2004, pp. 385–394.