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

A mio nonno, una stella che brilla.

Indice

Elenco delle figure

Elenco delle tabelle

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 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. 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. Dato il successo delle GPU come potenza di calcolo, si è introdotto nel parallel computing il termine GPGPU programming, che rappresenta tutti gli utilizzi delle GPU che non comprendono 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 [8].

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 che hanno portato diverse features innovative rendendo l'implementazione di algoritmi paralleli 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 [10].

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 [13].

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 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 [12].

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.

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.

Infine,

Il sesto capitolo mostrerà i risultati finali che validano il lavoro di tesi attraverso tempi di esecuzione e grafici rappresentanti le misure di performance.

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.

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:

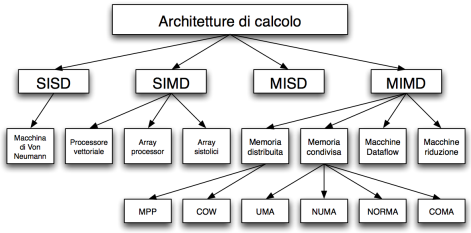
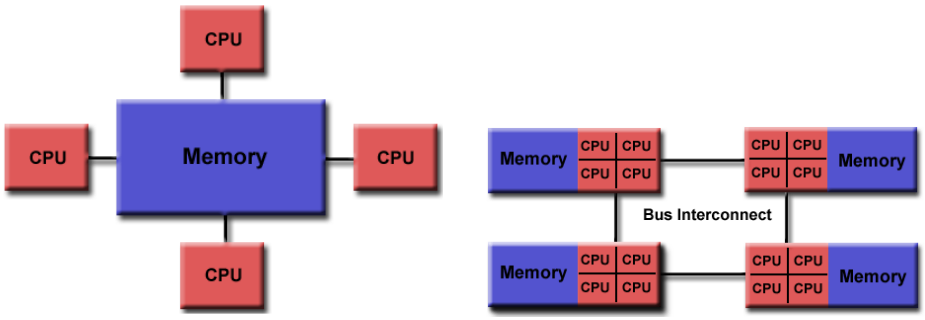


Figura 1.1: La tassonomia di Flynn



(a) UMA. (b) NUMA.

Figura 1.2: UMA e NUMA.

- 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 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

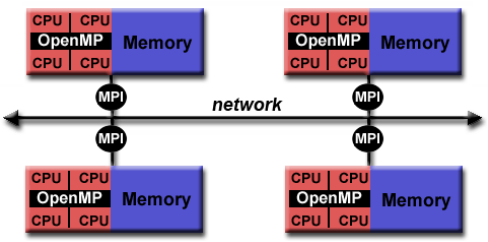


Figura 1.3: Esempio di sistema ibrido.

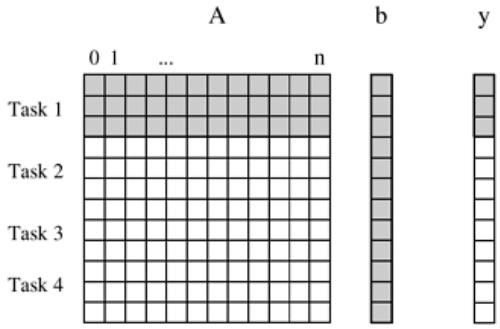


Figura 1.4: Suddivisione di una moltiplicazione tra una matrice e un vettore in 4 diversi task.

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 mes-

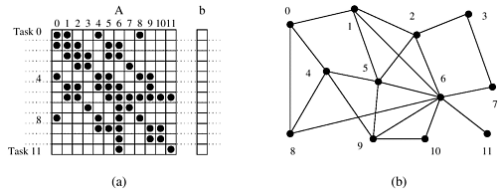


Figura 1.5: Esempio di un grafo delle interazioni tra i task.

saggi 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.

Decomposizione ricorsiva

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

Decomposizione dei dati

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

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

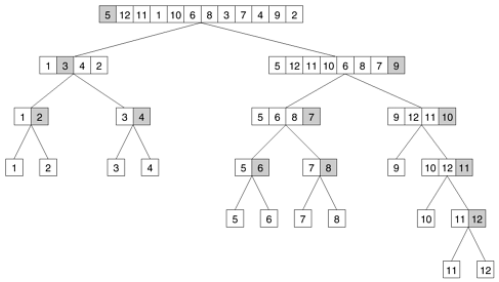


Figura 1.6: Esempio di decomposizione ricorsiva: il quicksort.

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

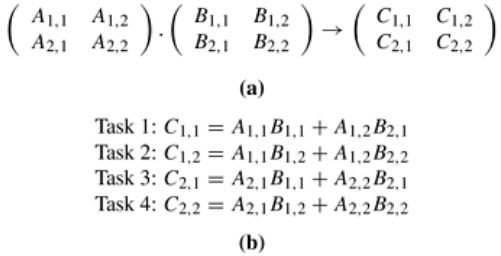


Figura 1.7: Esempio di decomposizione dei dati.

in più step, dove per ogni step si applica una tecnica di decomposizione diversa.

1.4.2 Tecniche di mapping

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

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

- Mapping statico
- Mapping dinamico

Descriviamo brevemente i due differenti approcci.

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

La tecnica di mapping dinamico invece distriuisce il lavoro durante l'esecuzione del programma. Scegliamo questa tecnica quando la dimensione dei task è sconosciuta e non si possono prevedere dunque le possibilità per un mapping ottimale [1].

1.4.3 Modelli di un algoritmo parallelo

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

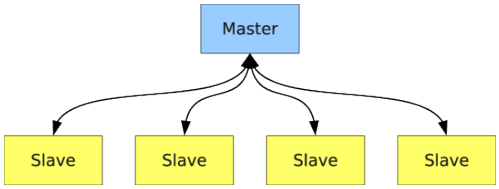


Figura 1.8: Il modello Master-Slave.

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

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

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

1.5 Misure di performance

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

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

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

Andiamo a descrivere ora, il significato di queste misure

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

L'**overhead** totale nel parallel computing è il tempo di esecuzione impiegato collettivamente da tutti i processori rispetto al tempo richiesto dal più veloce algoritmo sequenziale per risolvere il problema [1].

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

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*. Valutando un algoritmo spesso c'è bisogno di conoscere a quanto ammonta il guadagno effettivo, 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.

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

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.

1.6.2 MPI

MPI (Message Passing Interface) è uno standard utilizzato per il *message-passing* dagli sviluppatori di codice parallelo. MPI non è una vera e propria libreria ma un insieme di specifiche fissate da seguire all'interno di una libreria per il message passing. I principali orientamenti tale per cui un'interfaccia possa rispettare tali specifiche sono:

- La praticità
- La portabilità
- L'efficienza
- La flessibilità

Il più recente dei numeri di revisione di questo standard è MPI-3. A livello pratico, MPI, fornisce le più comuni funzionalità per consentire uno scambio di messaggi tra unità di calcolo. Rispetto ad OpenMP, ad

esempio, MPI consente il pieno controllo del parallelismo, in particolare è possibile conoscere l'ID univoco dei thread in esecuzione e dare precise direttive ad ognuno di loro. Proprio grazie allo scambio di messaggi è possibile implementare il modello `MASTER-SLAVE`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4  #include <math.h>
5
6  int main(int argc, char* argv)
7  {
8          int myid, numprocs;
9
10         MPI_Init(&argc, &argv); // This is mandatory at the start of the
               parallel part of program
11
12         MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
13         MPI_Comm_rank(MPI_COMM_WORLD, &myid);
14
15         /* Print out my rank and this run's PE size */
16         printf("Hello World from %d\n",myid);
17         printf("The number of procs is %d\n",numprocs);
18
19         MPI_Finalize(); // This is mandatory at the end of the parallel
               part of the program
20
21 }
```

Codice 1.2: Esempio di utilizzo di MPI.

Il codice 1.2 mostra un esempio basico dell'utilizzo di MPI.

```
MPI_Init(&argc, &argv);
```

e

```
MPI_Finalize();
```

sono delle direttive obbligatorie per l'utilizzo di MPI, in particolare contengono al loro interno la parte di codice da eseguire il parallelo.

```
MPI_Comm_size(MPI_COMM_WORLD , &numprocs);
```

e

```
MPI_Comm_rank(MPI_COMM_WORLD , &myid);
```

invece sono funzioni utilizzate per prendere le informazioni riguardo il numero di processi attualmente in esecuzione e l'id univoco per ogni thread. Grazie a queste informazioni è possibile gestire le differenti direttive parallele.

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <omp.h>

int main(int argc, char *argv[]) {

        const char Usage[] = "Usage: pi <steps> <repeats> (try 1000000 4)";
        if (argc < 3) {
                printf("%s \n", Usage);
                return (1);
        }
        int num_steps = atoi(argv[1]);
        int num_repeats = atoi(argv[2]);
        int my_rank, p, range, y, low, high, i, j, source, count = 0;
        int local_count = 0;
        int root = 0, tag = 0;
        double start, end, total;
        MPI_Status status; /*         return status for receive */

        //A little throwaway parallel section just to show num threads
        /* Start up MPI */
        MPI_Init(&argc, &argv);

        /* Find out process rank  */
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

        /* Find out number of processes */
        MPI_Comm_size(MPI_COMM_WORLD, &p);

        if (my_rank == 0) {
                printf(
                                "Computing pi via Monte Carlo using %d steps, repeating %d time \n",
                                num_steps, num_repeats);
        }

        start = omp_get_wtime();
        if (p <= num_repeats) {
                range = num_repeats / p;
                low = my_rank * range;
                high = low + range;
        } else {
                low = 0;
                high = num_repeats;
        }
        for (i = low; i < high; i++) {
                count = 0;
                local_count = 0;
                for (j = 0; j < num_steps; j++) {
                        double x = (double) rand() / RAND_MAX;
                        double y = (double) rand() / RAND_MAX;
                        if (x * x + y * y <= 1)
                                local_count++;
                }

                printf("Rank %d Local count %d \n", my_rank, local_count);
```

```
57
58                  if (my_rank == 0) {
59                          count = local_count;
60                          for (source = 1; source < p; source++) {
61                                  MPI_Recv(&local_count, 1, MPI_INT,
                                        source, tag, MPI_COMM_WORLD,
62                                              &status);
63                                  count = count + local_count;
64                          }
65
66                  } else {
67                          MPI_Send(&local_count, 1, MPI_INT, root, tag,
                                MPI_COMM_WORLD);
68                  }
69
70          }
71          if (my_rank == 0) {
72                  double pi = (double) count / (num_steps * p) * 4;
73                  printf("pi = %g\n", pi);
74                  end = omp_get_wtime();
75                  total = end - start;
76                  printf("Time: %1.24f seconds \n", total);
77          }
78          MPI_Finalize();
79          exit(0);
80  }
```

Codice 1.3: Determinazione di pigreco con il metodo montecarlo parallelizzato in MPI.

Il codice 1.3 è un esempio completo di utilizzo di MPI. E' l'implementazione parallela del metodo Montecarlo per determinare il valore di pigreco.

La due funzioni più interessanti sono le funzioni di invio e ricezione dei messaggi:

```
MPI_Recv(&local_count, 1, MPI_INT, source, tag,
            MPI_COMM_WORLD,&status);
```

e

```
MPI_Send(&local_count , 1, MPI_INT , root, tag,
            MPI_COMM_WORLD);
```

In particolare queste funzioni prendono in input:

- La variabile da trasferire attraverso i thread
- La dimensione della quantità di dati trasportati
- Il tipo di dati

- Chi invia e chi riceve il messaggio (ID dei thread)

- Il tag che indica l'ID del messaggio

In particolare la funzione adibita alla ricezione dei messaggi prende in input un valore in più, `status`, che consente la gestione dello stato del messaggio in modo da gestire eventuali errori e deadlock che possono accadere.

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 non venivano usate per scopi diversi dal rendering grafico e per accedere a questa tipologia di device i programmatori avevano a disposizione solamente librerie orientate al rendering grafico come OpenGL [15]. GPGPU (general purpose computing on graphic processing unit) è il termine che viene utilizzato per indicare l'utilizzo delle GPU in contesti differenti dal rendering grafico. Questa tecnica si diffuse nel 2007 grazie al rilascio di CUDA [9] da parte NVIDIA, che forniva ai programmatori un architettura completa capace di far 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.

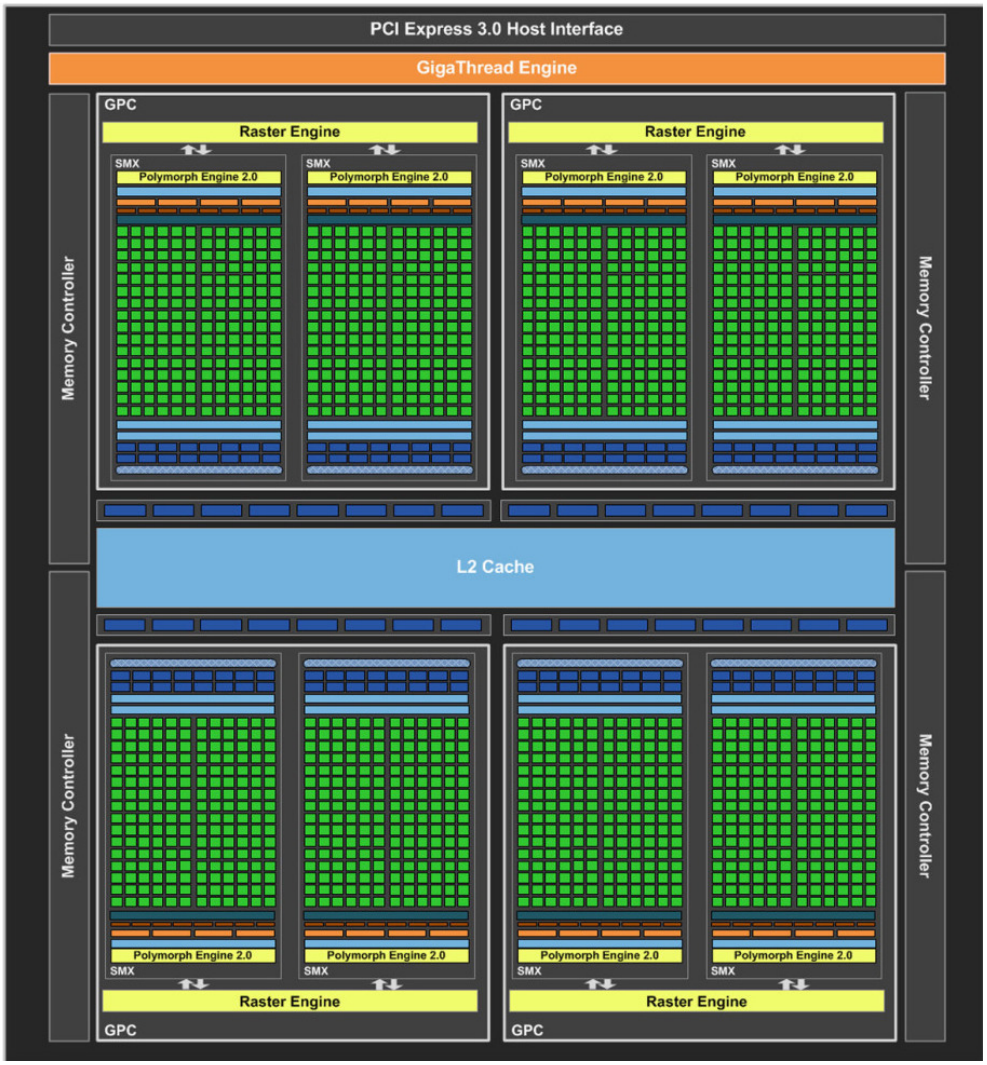


Figura 1.9: Architettura Kepler (GTX 680).

La figura 1.9 mostra la tipica architettura di una GPU CUDA. La struttura è composta da un insieme di *streaming multiprocessor* (SM) divisi in blocchi. Ogni SM è composto ulteriormente da un insieme di *streaming processors* che condividono la memoria cache. Ogni GPU ha a disposizione una determinata quantità di gigabytes di DRAM, diversamente detta memoria globale. Questo tipo di memoria è diversa dalla normale memoria DRAM poiché è progettata per contenere dati relativi alla grafica. Quando si tratta di di applicazioni orientate alla grafica, questo tipo di memoria contiene informazioni relative ad immagini e texture usate per il rendering 3D. In ambito GPGPU viene sfruttata per

la sua larghezza di banda (molto ampia) al costo di una latenza più alta rispetto alla normale memoria DRAM. Più è alta la disponibilità di memoria delle GPU prodotte più le applicazioni tendono a memorizzare i dati nella memoria globale minimizzando così le interazioni con la memoria del sistema.

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.

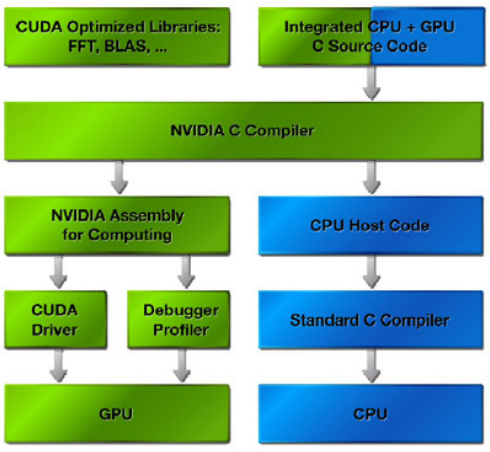


Figura 2.1: Struttura di Nvidia C Compiler.

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 diversi blocchi di shader cores, 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.

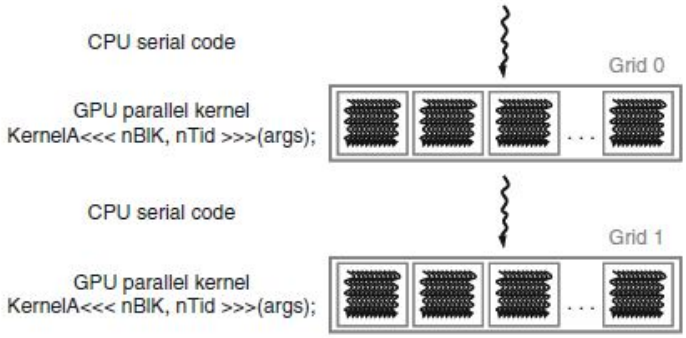


Figura 2.3: Esecuzione di un programma CUDA.

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.

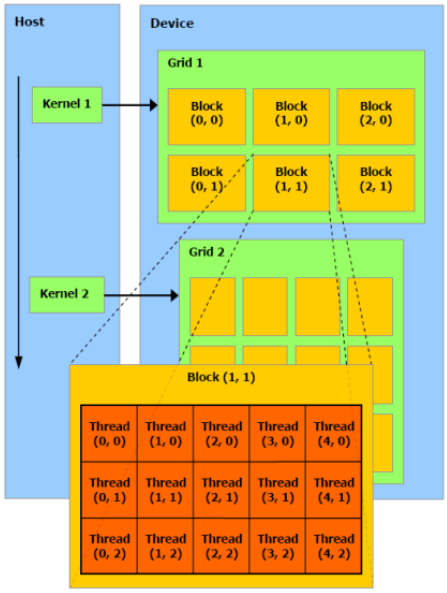


Figura 2.4: Esempio generico di griglie e blocchi in un programma CUDA.

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 alle loro

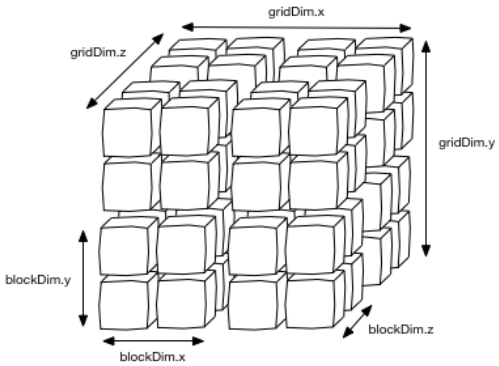


Figura 2.5: Un esempio di configurazione di griglie e blocchi tridimensionale in CUDA.

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**.

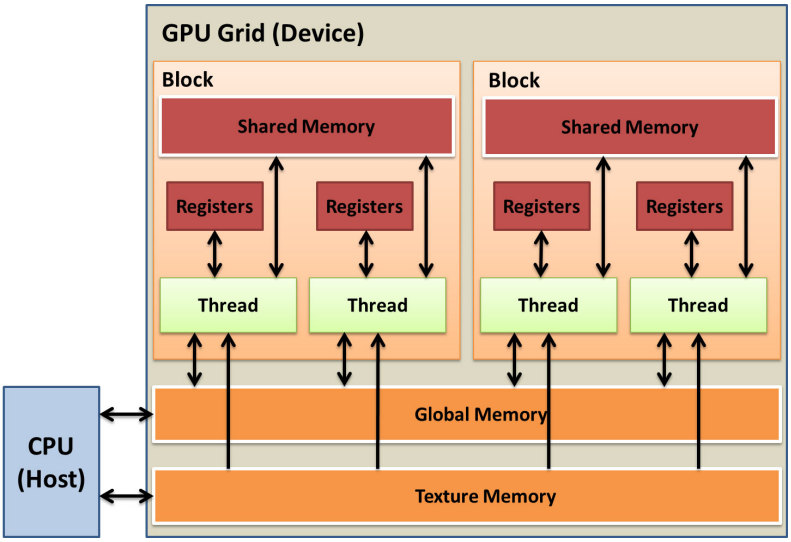


Figura 2.6: La struttura della memoria gestita dal sistema CUDA.

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.

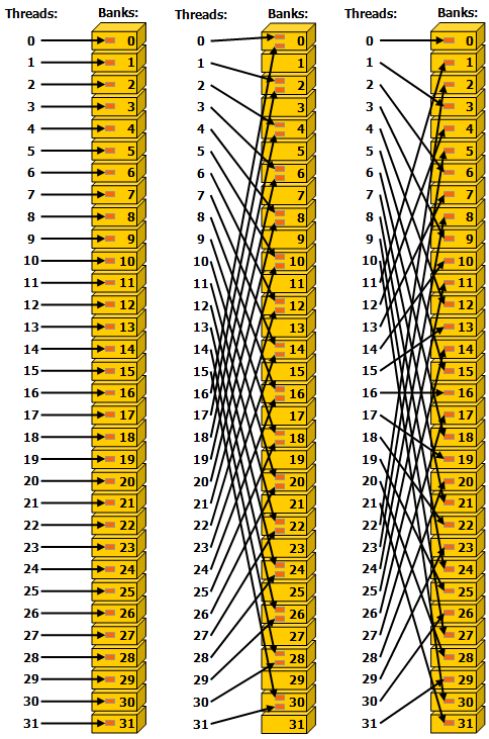


Figura 2.7: Shared memory divisa in banks.

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.

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. [7]

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.

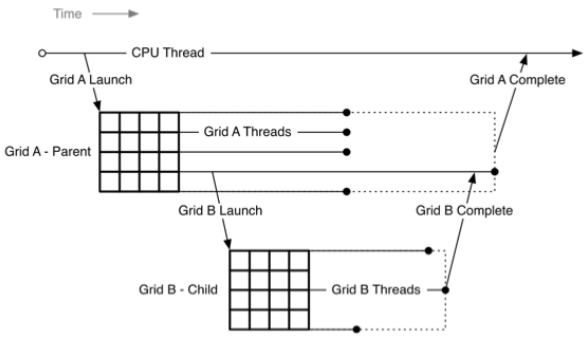


Figura 2.8: Dynamic Parallelism.

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 con-

sistenti: 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()`.

```
dim3 grid(3,2,2);
dim3 block(4,2);

__global__ void child(/* arguments */) {

        /* algorithm */

}

__global__ void kernel(/* arguments */) {

        child<<<grid, block>>>(/* arguments */);
}

int main() {

        kernel<<<grid, block>>>(/* arguments */);
}
```

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**) [5], tool di NVIDIA, che consente il debugging delle applicazioni CUDA. Un altro tool degno di menzione è **CUDA-MEMCHECK** [5], 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.

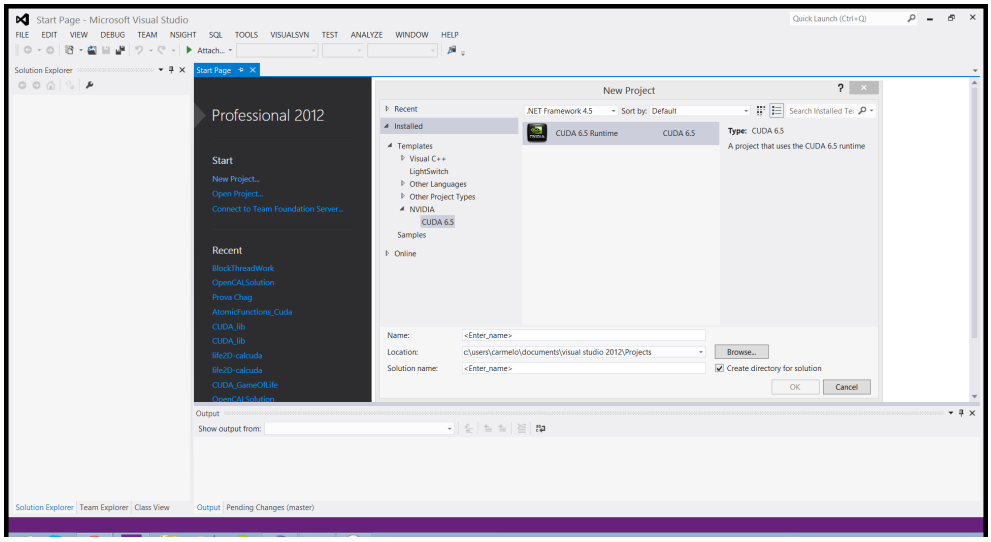


Figura 2.9: Creazione di un progetto CUDA 6.5 su Visual Studio.

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.

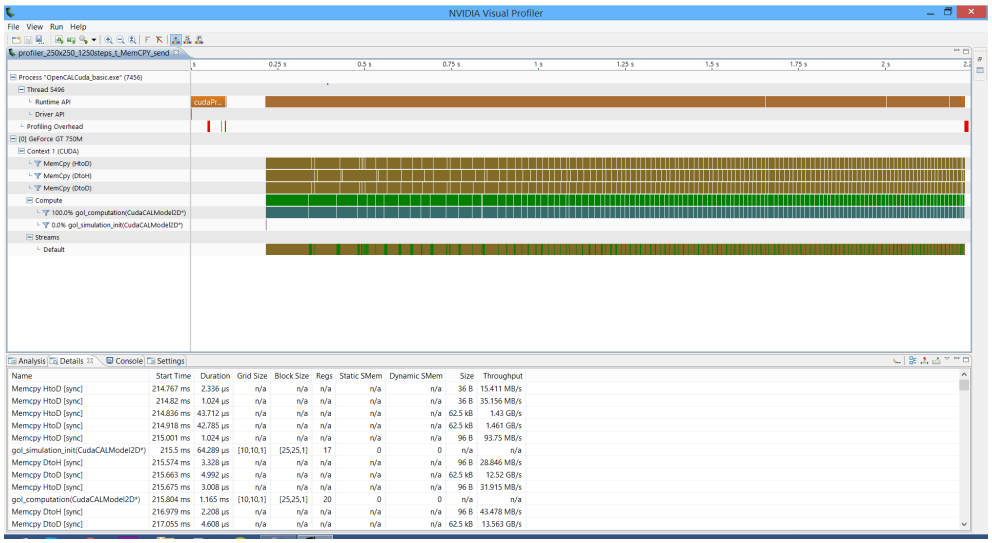


Figura 2.10: Esempio di progetto analizzato su VIsual Profiler.

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.

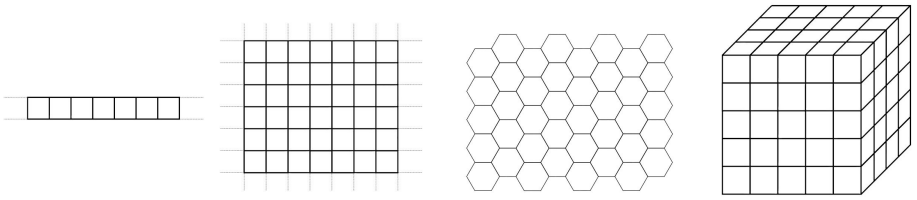
Capitolo 3

Automi Cellulari

3.1 Introduzione

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 auto-riprodursi, introdusse gli **Automi Cellulari**. Gli Automi Cellulari (AC) sono un insieme di regole logico-matematiche capaci di simulare sistemi complessi. Un AC è uno spazio d-dimensionale suddiviso in celle regolari alle quali è associato uno **stato** facente parte di un insieme finito di stati. La **funzione di transizione** (uguale per tutto l'insieme di celle) determina il valore dello stato di una cella in un determinato istante t. Il **vicinato**, cioè una relazione all'interno di uno spazio cellulare, influisce sullo stato di una cella. 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 determinato istante t definisce la configurazione dell' Automa Cellulare in quel preciso istante.



(a) (b) (c) (d)

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

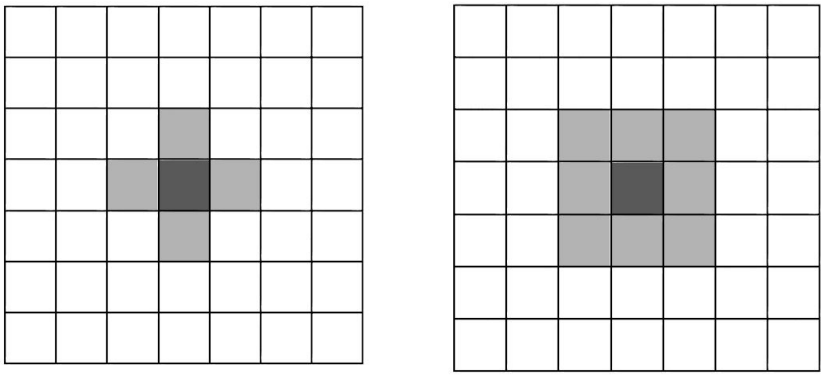
3.2 Definizione di Automa Cellulare

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 costituito 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 legge chiamata "funzione di transizione";
- lo stato di una cella al tempo $t + 1$ dipende dal suo stato corrente e dallo stato di tutto il suo vicinato al tempo t;
- la relazione di vicinanza è comune per tutto lo spazio cellulare e non cambia durante l'evoluzione dell'Automa Cellulare. Unica particolarità è che deve coinvolgere un numero di celle limitato.

I vicinati più utilizzati nel caso di Automi Cellulari con dimensione 2, sono quelli di von Neumann (figura 3.2 a) e quello di Moore (figura 3.2 b). Il primo è composto dalla cella centrale (cioè la cella corrente) e dalle celle adiacenti a nord, sud, est, e ovest, mentre il secondo comprende in più le celle adiacenti a nord-est, sud-est, nord-ovest e sud-ovest formando un vero e proprio quadrato. E' possibile comunque definire altri tipi di vicinato customizzati e non è necessario che le celle che lo costituiscono siano strettamente adiacenti tra di loro. La funzione di transizione



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

Figura 3.2: Esempi di relazioni di vicinanza.

cambia lo stato di ogni cella in base agli stati delle celle del vicinato.

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

La funzione 3.1 rappresenta in modo formale una tipica funzione di transizione di un Automa Cellulare, dove S rappresenta l'insieme degli 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 ad un altro allora la funzione si dice deterministica, mentre se la transizione avviene da uno stato a un insieme di stati allora la funzione si dice 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 diversi tipi di configurazioni diverse.

3.2.2 Definizione formale di Automa Cellulare

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

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

- Z^d è l'insieme delle celle facenti parte di uno spazio cellulare di dimensione d, nel quale ogni cella è un automa elementare. In particolare è uno spazio euclideo che fornisce le coordinate ad ogni cella;
- S è un insieme finito degli stati della cella, detto *spazio degli stati*;

- X è un insieme finito di vettori $d-$dimensionali. Esso definisce l'insieme dei vicini $N(X,i)$ di una generica cella $i =< i_1, i_2, \ldots, i_d >$ in Z^d:
  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, cioè la funzione di transizione dell'automa elementare, con $n = \#X$.

Dato un automa cellulare bidimensionale una possibile configurazione può essere la seguente:

- l'insieme delle celle $Z^2 = \{(i,j)|i,j \in N, m \geq 0, n \geq 0\}$ 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 $t+1$.

Data la definizione corrente di AC, definiamo 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, quelli che suscitano particolare interesse sono gli Automi Cellulari 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. Abbiamo $k = 2$ (ovvero 0 e 1) stati che possono essere assunti

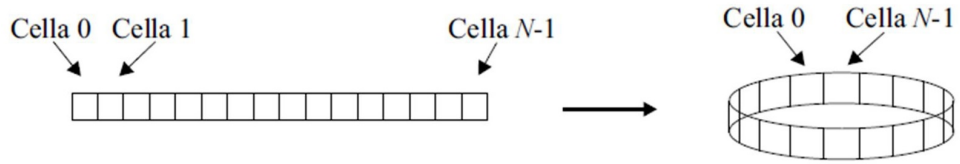


Figura 3.3: Esempio di automa cellulare unidimensionale

dalla cella e il raggio del vicinato è $r = 1$. Se prendiamo in considerazione la cella x il suo vicinato è formato dalla terna $< x-r, x, x+r >$. Dunque nel caso degli AC unidimensionali, siccome ogni cella può assumere $k = 2$ stati e ogni vicinato è composto da tre celle ci sono $2^3 = 8$ differenti combinazioni di vicinato e $2^8 = 256$ differenti 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

Ad ogni configurazione dell'AC corrisponde una funzione di transizione che determina lo stato corrente delle celle al passo $t+1$ ed è rappresentata da una sequenza formata dagli stati delle celle stesse. Generalmente le regole si indicano tramite numeri interi, dunque si converte la sequenza di caratteri in un numero intero. Nel caso dell'esempio nella tabella 3.1 la regola è definita dalla stringa di bit $s = 00000101$ è quindi dal numero intero $n = 5$.

Secondo Wolfram [20] [19] possiamo raggruppare in quattro classi i comportamenti differenti degli Automi Cellulari:

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 definizione di Automa Cellulare. Questo modello numerico si adatta perfettamente per la rappresentazione di fenomeni naturali in cui le 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 [13], 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:

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

Dove Z^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;
- $G = G_1 \cup G_2 \cup \ldots \cup G_l \subseteq Z^d$ è il sottoinsieme di celle di Z^d soggette ad influenza esterna.
- $\gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_t\}$ è l'insieme di funzioni che definiscono le influenze esterne

3.5 SCIARA-fv2

SCIARA è un modello computazionale basato su Automi Cellulari Complessi (CCA, 3.4) che simula il fenomeno naturale di una colata lavica. Già da qualche anno è utilizzato per numerose simulazioni di casi realmente accaduti, tra i più famosi l'eruzione del Monte Etna nell'area di Nicolosi del 2001 [13] e nell'area di Valle del Bove nel 1991 [2]. Possiamo formalizzare l'automa cellulare complesso che definisce SCIARA nel seguente modo:

$$SCIARA =< Z^d, S, X, G, P, \tau, \gamma >$$

- Z^d è uno spazio bi-dimensionale;
- $S = S_z \times S_h \times S_t \times S_f^8$ è l'insieme finito degli stati che può assumere una cella ottenuto dal prodotto cartesiano dei sottostati. Il loro significato è rispettivamente: quota (altitudine) della cella, spessore della lava, temperatura della lava, flussi uscenti dalla cella centrale verso le celle del vicinato (Nord, Ovest, Est, Sud, Nord-Ovest, Sud-Ovest, Sud-Est, Nord-Est);
- X è la relazione di vicinanza di Moore;
- P è l'insieme dei parametri globali usati per calibrare il modello. In particolare questi parametri non variano nel tempo;
- $\tau : S^9 \rightarrow S$ è la funzione di transizione deterministica, probabilistica o mista dell'automa;
- $\gamma : S_h \times \mathbb{N} \rightarrow S_h$ è la funzione che rappresenta le influenze esterne e in particolare l'emissione della lava dalle celle sorgenti.

La funzione di transizione di SCIARA è composta da quattro processi elementari:

- **calcolo dei flussi uscenti**: determina la fuoriuscita di lava dalla cella centrale verso le celle del vicinato applicando l'algoritmo di minimizzazione delle differenze.
- **calcolo della quantità di lava**: determina la quantità di lava considerando i flussi uscenti dalle celle.
- **calcolo della temperatura**: determina la temperatura della lava considerando la temperatura dei flussi entranti e la perdita di energia termica dalla superficie.
- **solidificazione**: determina la solidificazione della lava quando la temperatura scende al di sotto di un determinato valore.

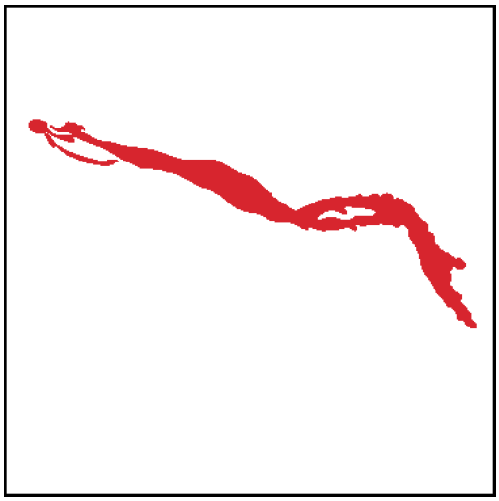


Figura 3.4: Evento reale rappresentato tramite il software Qgis

Parte di questo lavoro di tesi è stato parallelizzare questo modello utilizzando la libreria OpenCAL-CUDA che verrà descritta nel capitolo 5. Nella figura è mostrata la simulazione della colata lavica datata 2006.

Capitolo 4

OpenCAL

4.1 Liberia per Automi Cellulari

La modellistica è molto utilizzata negli ambienti di ricerca in diversi settori, dalla Biologia alla Geologia, dall'Ingegneria alla Bioinformatica. Per questo motivo, negli anni, sono state sviluppate diverse metodologie per la realizzazione di sistemi automatici e di supporto alle decisioni per la creazione di modelli e della loro simulazione: un esempio è CAMELot [12], un ambiente di sviluppo basato su Automi Cellulari per la simulazione di processi fisici. Al contrario di CAMELot, OpenCAL (Open Cellular Automata Library) è una libreria Open Source, capace di definire modelli di simulazione basati su Automi Cellulari complessi (CCA).

Alla base della nascita di OpenCAL troviamo la necessità di possedere una libreria open source, facilmente utilizzabile, che permetta all'utente di dare completa attenzione alla definizione dell'automa cellulare trascurando il più possibile i dettagli implementativi. Le funzioni, le strutture e i tipi di dato all'interno della libreria permettono di definire un modello di Automi Cellulari con uno spazio cellulare bidimensionale. OpenCAL supporta anche la definizione di modelli con uno spazio cellulare tridimensionale. Tuttavia le funzioni, le strutture e i tipi di dato usati per la definizione di un modello 2D hanno il loro corrispettivo nella versione 3D della libreria.

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` tramite 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.

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 (**double precision 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, la libreria mette a disposizione una funzione per la definizione della simulazione: `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.

Quando che viene eseguita un processo elementare o una funzione di supporto (init, steering, etc...) appartenente al ciclo di esecuzione, il contenuto delle matrici *next* dei sottostati viene copiato nelle matrici *current*. Nel secondo caso viene gestita direttamente dall'utente la gestione dell'aggiornamento dei sottostati. L'utente ha la possibilità di definire il proprio ciclo di esecuzione e la modalità di aggiornamento dei sottostati. Questo è reso possibile dalla funzione `calRunAddGlobalTransitionFunc2D` che riceve in input un puntatore alla funzione che definisce il ciclo di esecuzione dell'utente. Per aggiornare i sottostati possiamo utilizzare due diverse funzioni: se si vogliono aggiornare tutti i sottostati utilizziamo `calUpdate2D` se invece si vuole aggiornare solo un numero ristretto di sottostati (o uno solo) si utilizza `calUpdateSubstate2D(b|i|r)`. La funzione `calRun2D` permette di eseguire una simulazione, e infine la funzione `calRunCAStep2D` esegue un singolo passo di calcolo della simulazione per volta.

```
1  CALbyte calRunCAStep2D(struct CALRun2D* simulation)
2  {
3   //execute user transition function if defined
4   if (simulation->globalTransition)
5   {
6           simulation->globalTransition(simulation->ca2D);
7           if (simulation->UPDATE_MODE == CAL_UPDATE_IMPLICIT)
8                   calUpdate2D(simulation->ca2D);
9   }
10  else
11  //execute all elementary processes defined by the user
12  calGlobalTransitionFunction2D(simulation->ca2D);
13
14  //execute steering function if defined
15  if (simulation->steering)
16  {
17          simulation->steering(simulation->ca2D);
18          if (simulation->UPDATE_MODE == CAL_UPDATE_IMPLICIT)
19                  calUpdate2D(simulation->ca2D);
20  }
21
22  //check stop condition if defined
23  if (simulation->stopCondition)
24          if (simulation->stopCondition(simulation->ca2D))
25                  return CAL_FALSE;
26
```

```
27  return CAL_TRUE;
28 }
```

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 [11] 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*).

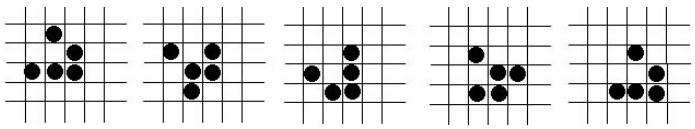


Figura 4.1: L'evoluzione del gioco della vita con la configurazione Glider

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>

#define ROWS 100
#define COLS 100
#define STEPS 1000
#define PATH gol_result

//gol substate
struct CALSubstate2Di * gol_substate;

//gol transition function
void gol_transition_function(struct CALModel2D* gol, int i, int j)
{
 int sum = 0, n;
 for (n=1; n<gol->sizeof_X; n++)
  sum += calGetX2Di(gol, gol_substate, i, j, n);

 if ((sum == 3) || (sum == 2 && calGet2Di(gol, gol_substate, i, j) == 1)
     )
  calSet2Di(gol, gol_substate, i, j, 1);
 else
  calSet2Di(gol, gol_substate, i, j, 0);
}

//initialization function
void init(struct CALModel2D* gol){
 calInitSubstate2Di(gol,gol_substate,0);
 calInit2Di(gol, gol_substate, 0, 2, 1);
 calInit2Di(gol, gol_substate, 1, 0, 1);
 calInit2Di(gol, gol_substate, 1, 2, 1);
 calInit2Di(gol, gol_substate, 2, 1, 1);
 calInit2Di(gol, gol_substate, 2, 2, 1);
}

int main() {

 //gol model definition
 struct CALModel2D * model = calCADef2D(ROWS, COLS,
     CAL_MOORE_NEIGHBORHOOD_2D, CAL_SPACE_TOROIDAL, CAL_NO_OPT);
 gol_substate = calAddSubstate2Di(model);
 calAddElementaryProcess2D(model,gol_transition_function);

 //gol execution definition
 struct CALRun2D * run = calRunDef2D(model, 1, STEPS,
     CAL_UPDATE_IMPLICIT);
 calRunAddInitFunc2D(run,init);
 calRun2D(run);

 //save computation result on a file
 calSaveSubstate2Di(model, gol_substate, PATH);
 return 0;
}
```

Codice 4.6: Il Game of Life in OpenCAL.

Nell'esempio 4.6, in circa 50 righe di codice, si implementa sia il modello che la simulazione del Game of Life. L'estrema semplicità dell'im-

plementazione mette in risalto dunque il punto di forza di OpenCAL. La libreria permette all'utente di concentrarsi sulla definizione dell'Automa Cellulare gestendo completamente tutti i dettagli implementativi. L'implementazione è divisa sostanzialmente in due fasi ed in particolare nella prima fase viene definito il modello. Usando la funzione `calCADef2D`, si crea un'istanza di `CALModel2D` con spazio cellulare bidimensionale toroidale e vicinato di Moore. Al modello viene aggiunto anche un sottostato, rappresentante l'insieme degli stati delle celle e la funzione di transizione definita dalle regole precedentemente elencate. La seconda fase comprende la definizione del ciclo di esecuzione. Al ciclo di esecuzione viene aggiunta una funzione di inizializzazione che definisce la configurazione iniziale di tutte le celle. Richiamando la funzione `calRun2D` si avvia la simulazione e i risultati della sua esecuzione vengono infine salvati su un file dalla funzione `calSaveSubsate2Di`.

4.4 SCIARA-fv2 in OpenCAL

Il modello computazionale più conosciuto, e quasi certamente il più semplice a livello computazionale, è Game of life (4.3). La sua implementazione è stata utile per i primi test e per i numerosi check di correttezza della libreria. Uno degli obiettivi di questo lavoro di tesi è stato tuttavia l'implementazione di modelli computazionalmente più complessi in modo da verificare la validità di OpenCAL e successivamente di OpenCAL-CUDA. Il modello proposto e implementato è **SCIARA**.

La descrizione formale di SCIARA si trova al paragrafo 3.5, in questa sezione si mostrerà l'implementazione tramite la libreria OpenCAL.

```
1  void initSciara(char * demPath, int steps) {
2
3   sciara = new Sciara;
4
5   /*some initializations*/
6
7   //model definition
8   sciara->model = calCADef2D(sciara->rows, sciara->cols,
        CAL_MOORE_NEIGHBORHOOD_2D, CAL_SPACE_FLAT, CAL_NO_OPT);
9
10  //substates definitions
11  sciara->substates = new SciaraSubstates();
12
13  sciara->substates->Sz = calAddSubstate2Dr(sciara->model);
14  sciara->substates->Slt = calAddSubstate2Dr(sciara->model);
15  sciara->substates->St = calAddSubstate2Dr(sciara->model);
16
17  sciara->substates->Mb = calAddSingleLayerSubstate2Db(sciara->model);
18  sciara->substates->Mv = calAddSingleLayerSubstate2Di(sciara->model);
```

```
19  sciara->substates->Msl = calAddSingleLayerSubstate2Dr(sciara->model);
20  sciara->substates->Sz_t0 = calAddSingleLayerSubstate2Dr(sciara->model);
21
22  //substates initializations
23  calInitSubstate2Dr(sciara->model, sciara->substates->Sz, 0);
24  calInitSubstate2Dr(sciara->model, sciara->substates->Slt, 0);
25  calInitSubstate2Dr(sciara->model, sciara->substates->St, 0);
26
27  for (int i = 0; i < sciara->rows * sciara->cols; ++i) {
28   sciara->substates->Mb->current[i] = CAL_FALSE;
29   sciara->substates->Mv->current[i] = 0;
30   sciara->substates->Msl->current[i] = 0;
31   sciara->substates->Sz_t0->current[i] = 0;
32  }
33
34  for (int i = 0; i < NUMBER_OF_OUTFLOWS; ++i) {
35   sciara->substates->f[i] = calAddSubstate2Dr(sciara->model);
36   calInitSubstate2Dr(sciara->model, sciara->substates->f[i], 0);
37  }
38
39  //elementary processes definition
40  calAddElementaryProcess2D(sciara->model, updateVentsEmission);
41  calAddElementaryProcess2D(sciara->model, empiricalFlows);
42  calAddElementaryProcess2D(sciara->model, width_update);
43  calAddElementaryProcess2D(sciara->model, updateTemperature);
44
45  //run definition
46  sciara->run = calRunDef2D(sciara->model, 1, steps, CAL_UPDATE_IMPLICIT)
        ;
47
48  calRunAddInitFunc2D(sciara->run, simulationInitialize);
49  calRunAddSteeringFunc2D(sciara->run, steering);
50  calRunAddStopConditionFunc2D(sciara->run, stopCondition);
51
52 }
```

Codice 4.7: Definizione del modello SCIARA in OpenCAL

Il codice 4.7 mostra la definizione del modello SCIARA implementato utilizzando la libreria OpenCAL. Secondo la definizione del modello viene creato uno spazio cellulare a due dimensioni con vicinato di Moore. Una volta aggiunti tutti i sottostati elencati nella descrizione formale dell'automa cellulare vengono aggiunti anche ulteriori sottostati a singola matrice utilizzati come supporto alla computazione. Infine, vengono definiti i processi elementari e il ciclo di esecuzione. Il codice 4.8 mostra l'implementazione dei processi elementari.

```
1  //first elementary process
2  void updateVentsEmission(struct CALModel2D * model, int i, int j) {
3   double emitted_lava = 0;
4   for (unsigned int k = 0; k < sciara->vent.size(); k++) {
5    int xVent = sciara->vent[k].x();
6    int yVent = sciara->vent[k].y();
7    if (i == yVent && j == xVent) {
8     emitted_lava = sciara->vent[k].thickness(sciara->elapsed_time, sciara
          ->Pclock, sciara->emission_time, sciara->Pac);
```

```
9     if (emitted_lava > 0) {
10     calSet2Dr(model, sciara->substates->Slt, yVent, xVent, calGet2Dr(
           sciara->model, sciara->substates->Slt, yVent, xVent) +
           emitted_lava);
11     calSet2Dr(model, sciara->substates->St, yVent, xVent, sciara->PTvent
           );
12    }
13   }
14  }
15 }
16
17 //second elementary process
18 void empiricalFlows(struct CALModel2D * model, int i, int j) {
19
20  if (calGet2Dr(model, sciara->substates->Slt, i, j) > 0) {
21   CALreal f[MOORE_NEIGHBORS];
22   outflowsMin(model, i, j, f);
23
24   for (int k = 1; k < MOORE_NEIGHBORS; k++)
25    if (f[k] > 0) {
26     calSet2Dr(model, sciara->substates->f[k - 1], i, j, f[k]);
27     if (active)
28      calAddActiveCellX2D(model, i, j, k);
29    }
30  }
31 }
32
33 //third elementary process
34 void width_update(struct CALModel2D* model, int i, int j) {
35  CALint outFlowsIndexes[NUMBER_OF_OUTFLOWS] = { 3, 2, 1, 0, 6, 7, 4, 5
        };
36  CALint n;
37  CALreal initial_h = calGet2Dr(model, sciara->substates->Slt, i, j);
38  CALreal initial_t = calGet2Dr(model, sciara->substates->St, i, j);
39  CALreal residualTemperature = initial_h * initial_t;
40  CALreal residualLava = initial_h;
41  CALreal h_next = initial_h;
42  CALreal t_next;
43
44  CALreal ht = 0;
45  CALreal inSum = 0;
46  CALreal outSum = 0;
47
48  for (n = 1; n < model->sizeof_X; n++) {
49   CALreal inFlow = calGetX2Dr(model, sciara->substates->f[
         outFlowsIndexes[n - 1]], i, j, n);
50   CALreal outFlow = calGet2Dr(model, sciara->substates->f[n - 1], i, j);
51   CALreal neigh_t = calGetX2Dr(model, sciara->substates->St, i, j, n);
52   ht += inFlow * neigh_t;
53   inSum += inFlow;
54   outSum += outFlow;
55  }
56  h_next += inSum - outSum;
57  calSet2Dr(model, sciara->substates->Slt, i, j, h_next);
58  if (inSum > 0 || outSum > 0) {
59   residualLava -= outSum;
60   t_next = (residualLava * initial_t + ht) / (residualLava + inSum);
61   calSet2Dr(model, sciara->substates->St, i, j, t_next);
62  }
63 }
```

```
64
65 //fourth elementary process
66 void updateTemperature(struct CALModel2D* model, int i, int j) {
67  CALreal nT, h, T, aus;
68  CALreal sh = calGet2Dr(model, sciara->substates->Slt, i, j);
69  CALreal st = calGet2Dr(model, sciara->substates->St, i, j);
70  CALreal sz = calGet2Dr(model, sciara->substates->Sz, i, j);
71
72  if (sh > 0 && !calGet2Db(model, sciara->substates->Mb, i, j)) {
73   h = sh;
74   T = st;
75   if (h != 0) {
76    nT = T;
77    aus = 1.0 + (3 * pow(nT, 3.0) * sciara->Pepsilon * sciara->Psigma *
          sciara->Pclock * sciara->Pcool) / (sciara->Prho * sciara->Pcv * h
           * sciara->Pac);
78    st = nT / pow(aus, 1.0 / 3.0);
79    calSet2Dr(model, sciara->substates->St, i, j, st);
80
81   }
82   if (st <= sciara->PTsol && sh > 0) {
83    calSet2Dr(model, sciara->substates->Sz, i, j, sz + sh);
84    calSetCurrent2Dr(model, sciara->substates->Msl, i, j, calGet2Dr(model
          , sciara->substates->Msl, i, j) + sh);
85    calSet2Dr(model, sciara->substates->Slt, i, j, 0);
86    calSet2Dr(model, sciara->substates->St, i, j, sciara->PTsol);
87   } else
88    calSet2Dr(model, sciara->substates->Sz, i, j, sz);
89  }
90 }
```

Codice 4.8: Definizione dei processi elementari del modello SCIARA 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). In questa tesi è stata sviluppata una versione CUDA della parte 2D di OpenCAL, che prende il nome di OpenCAL-CUDA, di seguito descritta 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 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 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.

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 1 e 2, 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

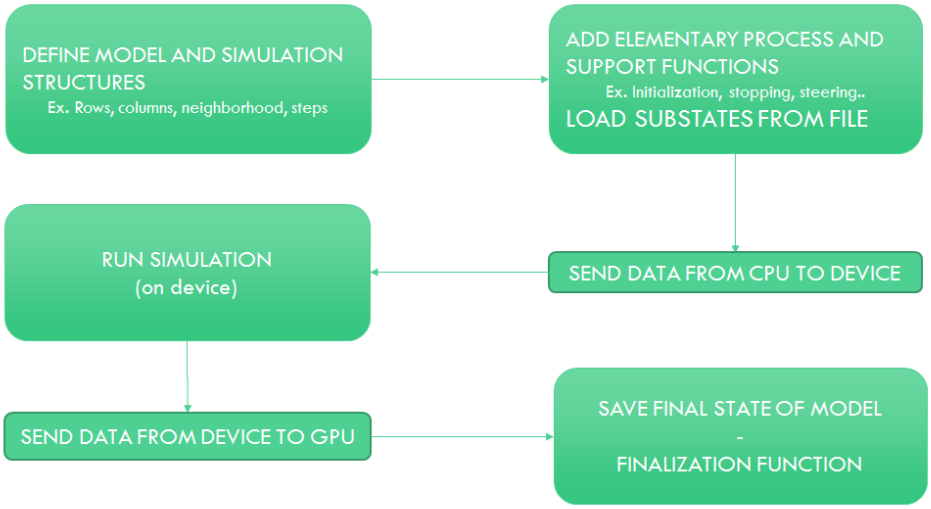


Figura 5.1: Diagramma del ciclo di vita del software OpenCAL-CUDA

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. 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`, tra le diverse implementazioni si rappresenta il modello in maniera del tutto simile. 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.

```
1  struct CudaCALModel2D {
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         //!< Array of flags having the substates' dimension: flag is
               CAL_TRUE if the corresponding cell is active, CAL_FALSE
               otherwise.
16         CALbyte* activecell_flags;
17
18         //!< Number of CAL_TRUE flags.
19         int activecell_size_next;
20
21         //!< i-Array of computational active cells.
22         int *i_activecell;
23
24         //!< j-Array of computational active cells.
25         int *j_activecell;
26
27         //!< Number of active cells in the current step.
28         int activecell_size_current;
29
30
31         //!< Array of cell coordinates defining the cellular automaton
               neighbourhood relation.
32         int *i;
33         int *j;
34
35         //!< Number of cells belonging to the neighbourhood. Note that
               predefined neighbourhoods include the central cell.
36         int sizeof_X;
37
38         //!< Neighbourhood relation's id.
39         enum CALNeighborhood2D X_id;
40
41         //!< Current linearised matrix of the substate, used for reading
                purposes.
42         CALbyte* pQb_array_current;
43         //!< Next linearised matrix of the substate, used for writing
               purposes.
44         CALbyte* pQb_array_next;
45
46         //!< Current linearised matrix of the substate, used for reading
                purposes.
47         CALint* pQi_array_current;
48         //!< Next linearised matrix of the substate, used for writing
               purposes.
```

```
49          CALint* pQi_array_next;
50
51          //!< Current linearised matrix of the substate, used for reading
                purposes.
52          CALreal* pQr_array_current;
53          //!< Next linearised matrix of the substate, used for writing
                purposes.
54          CALreal* pQr_array_next;
55
56          //!< Number of substates of type byte.
57          int sizeof_pQb_array;
58          //!< Number of substates of type int.
59          int sizeof_pQi_array;
60          //!< Number of substates of type real (floating point).
61          int sizeof_pQr_array;
62
63          //!< 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 sulla memoria del device. Proprio per questa motivazione è presente una struttura dati 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-CUDA, infatti l'utente non verrà mai a conoscenza della sua presenza poiché è utilizzata solo nel core della libreria. Questo non accade per `device_ca2D`. L'utente ha il compito di dichiarare un istanza del modello anche per la memoria sul device nel main principale, lasciando il compito della sua 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.

```
struct CudaCALRun2D
{
        //!< Pointer to the cellular automaton structure.
        struct CudaCALModel2D* ca2D;

        //!< Pointer to the cellular automaton structure on device.
        struct CudaCALModel2D* device_ca2D;

        //!< Pointer to the cellular automaton structure for data
            passing.
        struct CudaCALModel2D* h_device_ca2D;

        //!< Stream compaction data structure
        unsigned int * device_array_of_index_dim;

        //!< Current simulation step.
        int step;

        //!< Initial simulation step.
        int initial_step;

        //!< Final simulation step; if 0 the simulation becomes a loop.
        int final_step;

        //!< Callbacks substates' update mode; it can be
            CAL_UPDATE_EXPLICIT or CAL_UPDATE_IMPLICIT.
        enum CALUpdateMode UPDATE_MODE;

        //!< Simulation's initialization callback function.
        void (*init)(struct CudaCALModel2D*);

        //!< CA's globalTransition callback function. If defined, it is
            executed instead of cal2D.c::calGlobalTransitionFunction2D.
        void (*globalTransition)(struct CudaCALModel2D*);

        //!< Simulation's steering callback function.
        void (*steering)(struct CudaCALModel2D*);

        //!< Simulation's stopCondition callback function.
        void (*stopCondition)(struct CudaCALModel2D*);

        //!< Simulation's finalize callback function.
        void (*finalize)(struct CudaCALModel2D*);
};
```

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 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 **hybrid_model**)

4. Copiare con una semplice `memcpy` le variabili scalari di host_model su hybrid_model.

5. Copiare, utilizzando la funzione fornita dalle API di Cuda `cudaMemCpy`, il contenuto delle strutture dati (gestite tramite puntatori) di host_model su hybrid_model

6. Copiare, utilizzando la funzione fornita dalle API di Cuda `cudaMemCpy`, tutto l'oggetto hybrid_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 (hybrid_model). Una copia corretta del codice 5.2.3 potrebbe essere:

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

```
1  CALbyte calInitializeInGPU2D(struct CudaCALModel2D* model, struct
       CudaCALModel2D *d_model){
2
3          CALbyte result = CAL_TRUE;
4
5          calCudaAllocatorModel(model);
6
7          cudaMemcpy(copy_model->i,model->i, sizeof(CALint)*model->
               sizeof_X, cudaMemcpyHostToDevice);
8          cudaMemcpy(copy_model->j,model->j, sizeof(CALint)*model->
               sizeof_X, cudaMemcpyHostToDevice);
9
10         if(model->OPTIMIZATION == CAL_OPT_ACTIVE_CELLS){
11                 cudaMemcpy(copy_model->activecell_flags,model->
                       activecell_flags, sizeof(CALbyte)*model->rows*model
                       ->columns, cudaMemcpyHostToDevice);
12                 cudaMemcpy(copy_model->activecell_index,model->
                       activecell_index, sizeof(CALint)*model->rows*model->
                       columns, cudaMemcpyHostToDevice);
```

```
13                  cudaMemcpy(copy_model->array_of_index_result,model->
                        array_of_index_result, sizeof(CALint)*model->rows*
                        model->columns, cudaMemcpyHostToDevice);
14          }
15
16          if(model->sizeof_pQb_array > 0){
17                  cudaMemcpy(copy_model->pQb_array_current,model->
                        pQb_array_current, model->sizeof_pQb_array*model->
                        rows*model->columns*sizeof(CALbyte),
                        cudaMemcpyHostToDevice);
18                  cudaMemcpy(copy_model->pQb_array_next,model->
                        pQb_array_next, model->sizeof_pQb_array*model->rows*
                        model->columns*sizeof(CALbyte),
                        cudaMemcpyHostToDevice);
19          }
20          if(model->sizeof_pQi_array > 0){
21                  cudaMemcpy(copy_model->pQi_array_current,model->
                        pQi_array_current, model->sizeof_pQi_array*model->
                        rows*model->columns*sizeof(CALint),
                        cudaMemcpyHostToDevice);
22                  cudaMemcpy(copy_model->pQi_array_next,model->
                        pQi_array_next, model->sizeof_pQi_array*model->rows*
                        model->columns*sizeof(CALint),
                        cudaMemcpyHostToDevice);
23          }
24          if(model->sizeof_pQr_array > 0){
25                  cudaMemcpy(copy_model->pQr_array_current,model->
                        pQr_array_current, model->sizeof_pQr_array*model->
                        rows*model->columns*sizeof(CALreal),
                        cudaMemcpyHostToDevice);
26                  cudaMemcpy(copy_model->pQr_array_next,model->
                        pQr_array_next, model->sizeof_pQr_array*model->rows*
                        model->columns*sizeof(CALreal),
                        cudaMemcpyHostToDevice);
27          }
28          cudaMemcpy(d_model, copy_model, sizeof(struct CudaCALModel2D),
                cudaMemcpyHostToDevice);
29
30          return result;
31  }
```

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

```
1   CALbyte calSendDataGPUtoCPU(struct CudaCALModel2D* model, struct
        CudaCALModel2D *d_model){
2
3           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);
        }
        if(model->sizeof_pQi_array > 0){
                cudaMemcpy(model->pQi_array_current,copy_model->
                    pQi_array_current,model->sizeof_pQi_array*model->
                    rows*model->columns*sizeof(CALint),
                    cudaMemcpyDeviceToHost);
                cudaMemcpy(model->pQi_array_next,copy_model->
                    pQi_array_next,model->sizeof_pQi_array*model->rows*
                    model->columns*sizeof(CALint),
                    cudaMemcpyDeviceToHost);
        }
        if(model->sizeof_pQr_array > 0){
                cudaMemcpy(model->pQr_array_current,copy_model->
                    pQr_array_current,model->sizeof_pQr_array*model->
                    rows*model->columns*sizeof(CALreal),
                    cudaMemcpyDeviceToHost);
                cudaMemcpy(model->pQr_array_next,copy_model->
                    pQr_array_next,model->sizeof_pQr_array*model->rows*
                    model->columns*sizeof(CALreal),
                    cudaMemcpyDeviceToHost);
        }

        calCudaFinalizeModel();

        return result;
}
```

L'oggetto *copy_model* presente nel codice è l'oggetto che in questo paragrafo abbiamo chiamato hybrid_model. 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`:

```
struct CudaCALModel2D* calCudaAllocatorModel(struct CudaCALModel2D *
    model){

        cudaMallocHost((void**)&copy_model, sizeof(struct CudaCALModel2D
            ), cudaHostAllocPortable);

        memcpy(copy_model,model,sizeof(struct CudaCALModel2D));

        cudaMalloc((void**)&copy_model->i,model->sizeof_X*sizeof(int));
        cudaMalloc((void**)&copy_model->j,model->sizeof_X*sizeof(int));

        if(model->OPTIMIZATION == CAL_OPT_ACTIVE_CELLS){
                cudaMalloc((void**)&copy_model->activecell_flags,model->
                    rows*model->columns*sizeof(CALbyte));
                cudaMalloc((void**)&copy_model->activecell_index,model->
                    rows*model->columns*sizeof(CALint));
                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 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 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.

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**[4].

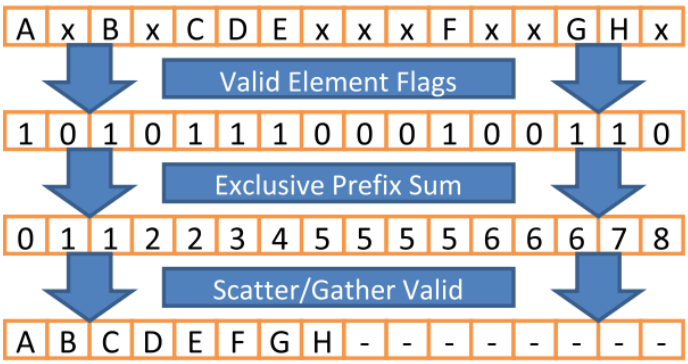


Figura 5.2: Esempio di stream compaction

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 [4] [3].

Inizialmente si era utilizzata la libreria **thrust** [6] 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 noiosi 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
```

```
23 #define STEPS_THRESHOLD 200
24 #define CONFIGURATION_PATH "./data/map_1000x1000.txt"
25 #define OUTPUT_PATH "./data/final.txt"
26
27 #define NUMBER_OF_SUBSTATE_BYTE 1
28 #define NUMBER_OF_NEIGHBORHOOD 9
29
30 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);
101
102         //add transition function's elementary processes
103         calCudaAddElementaryProcess2D(gol, gol_computation);
104
105         printf ("Starting alloc...\n");
106         start_time = time(NULL);
107
108         //add substates
109         calCudaAddSubstate2Db(gol, NUMBER_OF_SUBSTATE_BYTE);
110
111         //load configuration
112         calCudaLoadSubstate2Db(gol, CONFIGURATION_PATH, ALIVE);
113
114         //send data to GPU
115         calInitializeInGPU2D(gol, device_gol);
116
117         end_time = time(NULL);
118         printf ("Alloc terminated.\nElapsed time: %d\n\n", end_time-
                start_time);
119
120         cudaErrorCheck("Data initialized on device\n");
121
122         //simulation configuration
123         calCudaRunAddInitFunc2D(gol_simulation, gol_simulation_init);
124         calCudaRunAddStopConditionFunc2D(gol_simulation,
                gol_simulation_stop);
125
126         printf ("Starting simulation...\n");
127         start_time = time(NULL);
128
129         //simulation run
130         calCudaRun2D(gol_simulation,grid,block);
131
132         //send data to CPU
133         calSendDataGPUtoCPU(gol,device_gol);
134
135         cudaErrorCheck("Final configuration sent to CPU\n");
```

```
136
137         end_time = time(NULL);
138         printf ("Simulation terminated.\nElapsed time: %d\n\n", end_time
                -start_time);
139
140         cudaProfilerStop();
141
142         //saving configuration
143         calCudaSaveSubstate2Db(gol, OUTPUT_PATH, ALIVE);
144
145         cudaErrorCheck("Data saved on output file\n");
146
147         //finalizations
148         calCudaRunFinalize2D(gol_simulation);
149         calCudaFinalize2D(gol, device_gol);
150
151         system("pause");
152         return 0;
153 }
```

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 chiaro 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 "SCIARA-fv2" in OpenCAL-CUDA

Come specificato nel paragrafo 4.4 è stato implementato un modello più complesso a livello computazionale per eseguire diversi stressing test esaminando l'effettiva validità del lavoro di tesi. SCIARA è un modello basato su Automi Cellulari Complessi (CCA, 3.4) che descrive il fenomeno di una colata lavica (Per ulteriori dettagli [2] [13] o par. 4.4).

Il tempo totale impiegato per implementare la versione parallelizzata in OpenCAL-CUDA a partire dalla versione OpenCAL è sicuramente un fattore determinante per la riuscita del lavoro di tesi. Se il tempo per elaborare una versione parallela di un automa cellulare in OpenCAL-CUDA supera il tempo di implementazione dello stesso automa cellulare in CUDA C, allora utilizzare la libreria non comporterebbe nessun valore aggiuntivo. Nelle prove effettuate non è risultato così. In particolare è avvenuto il contrario: in poche ore, sia per SCIARA che per altri modelli di test, si è implementata una versione correttamente parallelizzata e performante. L'immediatezza del passaggio alla GPGPU programming tramite la libreria OpenCAL-CUDA è dunque uno dei punti di forza della libreria stessa.

```
1  //------------------------------------------------------------
2  //                      THE Sciara-FV2 CELLULAR AUTOMATON
           -
3  //------------------------------------------------------------
4  #define maximum_steps                          0
5  #define stopping_threshold                     0.00
6  #define refreshing_step                 0
7  #define thickness_visual_threshold      0.00
8  #define Pclock                          60.00
9  #define PTsol                           1143.00
10 #define PTvent                          1360.00
11 #define Pr_Tsol                         0.0750
12 #define Pr_Tvent                        0.90
13 #define Phc_Tsol                        60.00
14 #define Phc_Tvent                       0.4
15 #define Pcool                           9.0
16 #define Prho                            2600.00
17 #define Pepsilon                        0.90
18 #define Psigma                          5.68E-08
19 #define Pcv                             1150
20 #define algorithm                       MIN
21 #define layers                                      40
22 #define rows                                        378
23 #define cols                                        517
24 #define cell_size                                   10.000000
25 #define nodata_value                           0
26 #define num_emission_rate                      15
27 #define num_total_emission_rates        2
28 #define xllcorner                                   499547.500000
29 #define yllcorner                                   4174982.500000
30 #define rad2                                        1.41421356237
```

```
31
32 __device__ CALreal a,b,c,d;
33
34 //Number of steps
35 #define STEPS 3200
36
37 //Files path of event.
38 #define DEM_PATH "data/2006/2006_000000000000_Morphology.txt"
39 #define VENTS_PATH "data/2006/2006_000000000000_Vents.txt"
40 #define EMISSION_RATE_PATH "data/2006/2006_000000000000_EmissionRate.txt
       "
41 #define TEMPERATURE_PATH "data/2006/2006_000000000000_Temperature.txt"
42 #define THICKNESS_PATH "data/2006/2006_000000000000_Thickness.txt"
43 #define SOLIDIFIED_LAVA_THICKNESS_PATH "data/2006/2006
       _000000000000_SolidifiedLavaThickness.txt"
44 #define REAL_EVENT_THICKNESS_PATH "data/2006/2006_000000000000_RealEvent
       .txt"
45
46 #define O_DEM_PATH "data/2006_SAVE/2006_000000000000_Morphology.txt"
47 #define O_VENTS_PATH "data/2006_SAVE/2006_000000000000_Vents.txt"
48 #define O_EMISSION_RATE_PATH "data/2006_SAVE/2006
       _000000000000_EmissionRate.txt"
49 #define O_TEMPERATURE_PATH "data/2006_SAVE/2006_000000000000_Temperature
       .txt"
50 #define O_THICKNESS_PATH "data/2006_SAVE/2006_000000000000_Thickness.txt
       "
51 #define O_SOLIDIFIED_LAVA_THICKNESS_PATH "data/2006_SAVE/2006
       _000000000000_SolidifiedLavaThickness.txt"
52
53 //Use active_cells optimization, comment for not use.
54 #define ACTIVE_CELLS
55
56 //Define values of outflows, dimension of neighbors and substates
57 #define NUMBER_OF_OUTFLOWS 8
58 #define MOORE_NEIGHBORS 9
59 #define VON_NEUMANN_NEIGHBORS 5
60
61 #define NUMBER_OF_SUBSTATES_REAL 13
62 #define NUMBER_OF_SUBSTATES_INT 1
63 #define NUMBER_OF_SUBSTATES_BYTE 1
64
65 //Enumerative for increase readability of code.
66 enum SUBSTATES_NAMES_REAL{
67         ALTITUDE=0,THICKNESS,TEMPERATURE,PRE_EVENT_TOPOGRAPHY,
             SOLIDIFIED, FLOWN,FLOWO,FLOWE,FLOWS, FLOWNO, FLOWSO, FLOWSE,
             FLOWNE
68 };
69 enum SUBSTATES_NAMES_INT{
70         VENTS=0,
71 };
72 enum SUBSTATES_NAMES_BYTE{
73         TOPOGRAPHY_BOUND=0,
74 };
75
76 //Grids and blocks configuration
77 CALint N = 21;
78 CALint M = 47;
79 dim3 block(N,M);
80 dim3 grid(cols/block.x, rows/block.y);
```

La prima parte di codice riguarda principalmente la definizione di tutti i valori numerici relativi al fenomeno naturale. Tra queste definizioni troviamo anche la dimensione del modello, il numero e il nome dei sottostati. Da notare che, come per Game of Life, gli enumerativi sono utilizzati solo per la leggibilità e per un implementazione più semplice del modello.

```
1  //Elementary processes and support functions
2
3  __global__
4          void updateVentsEmission(struct CudaCALModel2D * model) {
5
6                  CALint offset = calCudaGetIndex(model), i =
                       calCudaGetIndexRow(model, offset), j=
                       calCudaGetIndexColumn(model, offset);
7                  CALreal emitted_lava;
8
9                  if(calCudaGet2Di(model,offset,VENTS) == 1)
10                 {
11                         emitted_lava = 1.806732;
12                         if (emitted_lava > 0) {
13                                 calCudaSet2Dr(model, offset,
                                       calCudaGet2Dr(model, offset,
                                       THICKNESS) + emitted_lava, THICKNESS
                                       );
14                                 calCudaSet2Dr(model, offset, PTvent,
                                       TEMPERATURE);
15                         }
16                 }
17
18                 if(calCudaGet2Di(model,offset,VENTS) == 2)
19                 {
20                         emitted_lava = 1.806732;
21                         if (emitted_lava > 0) {
22                                 calCudaSet2Dr(model, offset,
                                       calCudaGet2Dr(model, offset,
                                       THICKNESS) + emitted_lava, THICKNESS
                                       );
23                                 calCudaSet2Dr(model, offset, PTvent,
                                       TEMPERATURE);
24                         }
25                 }
26 }
27
28 __device__
29         double powerLaw(double k1, double k2, double T) {
30                 double log_value = k1 + k2 * T;
31                 return pow(10.0, log_value);
32 }
33
34 __device__
35         void outflowsMin(struct CudaCALModel2D * model, int offset,
               CALreal *f) {
36
37                 bool n_eliminated[MOORE_NEIGHBORS];
38                 double z[MOORE_NEIGHBORS];
39                 double h[MOORE_NEIGHBORS];
40                 double H[MOORE_NEIGHBORS];
```

```
41                  double theta[MOORE_NEIGHBORS];
42                  double w[MOORE_NEIGHBORS];              //Distances
                        between central and adjecent cells
43                  double Pr_[MOORE_NEIGHBORS];    //Relaiation rate arraj
44                  bool loop;
45                  int counter;
46                  double avg, _w, _Pr, hc, sum, sumZ;
47
48                  CALreal t = calCudaGet2Dr(model, offset, TEMPERATURE);
49
50                  _w = cell_size;
51                  _Pr = powerLaw(a, b, t);
52                  hc = powerLaw(c, d, t);
53
54                  for (int k = 0; k < MOORE_NEIGHBORS; k++) {
55
56                          h[k] = calCudaGetX2Dr(model, offset, k,
                                THICKNESS);
57                          H[k] = f[k] = theta[k] = 0;
58                          w[k] = _w;
59                          Pr_[k] = _Pr;
60                          CALreal sz = calCudaGetX2Dr(model, offset, k,
                                ALTITUDE);
61                          CALreal sz0 = calCudaGet2Dr(model, offset,
                                ALTITUDE);
62                          if (k < VON_NEUMANN_NEIGHBORS)
63                                  z[k] = calCudaGetX2Dr(model, offset, k,
                                        ALTITUDE);
64                          else
65                                  z[k] = sz0 - (sz0 - sz) / rad2;
66                  }
67
68                  H[0] = z[0];
69                  n_eliminated[0] = true;
70
71                  for (int k = 1; k < MOORE_NEIGHBORS; k++)
72                          if (z[0] + h[0] > z[k] + h[k]) {
73                                  H[k] = z[k] + h[k];
74                                  theta[k] = atan(((z[0] + h[0]) - (z[k] +
                                        h[k])) / w[k]);
75
76                                  n_eliminated[k] = true;
77                          } else
78                                  n_eliminated[k] = false;
79
80                          do {
81                                  loop = false;
82                                  avg = h[0];
83                                  counter = 0;
84                                  for (int k = 0; k < MOORE_NEIGHBORS; k
                                        ++)
85                                      if (n_eliminated[k]) {
86                                              avg += H[k];
87                                              counter++;
88                                      }
89                                      if (counter != 0)
90                                              avg = avg / double(
                                                    counter);
91                                      for (int k = 0; k <
                                            MOORE_NEIGHBORS; k++)
```

```
92                                                  if (n_eliminated[k] &&
                                                        avg <= H[k]) {
93                                                      n_eliminated[k]
                                                            = false;
94                                                      loop = true;
95                                                  }
96                          } while (loop);
97
98                          for (int k = 1; k < MOORE_NEIGHBORS; k++) {
99                                  if (n_eliminated[k] && h[0] > hc * cos(
                                        theta[k])) {
100                                         f[k] = Pr_[k] * (avg - H[k]);
101                                 }
102                         }
103 }
104
105 __global__
106         void empiricalFlows(struct CudaCALModel2D * model) {
107
108                 CALint offset = calCudaGetIndex(model);
109
110                 if (calCudaGet2Dr(model, offset, THICKNESS) > 0) {
111                         CALreal f[MOORE_NEIGHBORS];
112                         outflowsMin(model, offset, f);
113
114                         if (f[1] > 0) {
115                                 calCudaSet2Dr(model, offset, f[1],
                                        FLOWN);
116 #ifdef ACTIVE_CELLS
117                                 calCudaAddActiveCellX2D(model, offset,
                                        1);
118 #endif
119                         }
120
121                         if (f[2] > 0) {
122                                 calCudaSet2Dr(model, offset, f[2],
                                        FLOWO);
123 #ifdef ACTIVE_CELLS
124                                 calCudaAddActiveCellX2D(model, offset,
                                        2);
125 #endif
126                         }
127
128                         if (f[3] > 0) {
129                                 calCudaSet2Dr(model, offset, f[3],
                                        FLOWE);
130 #ifdef ACTIVE_CELLS
131                                 calCudaAddActiveCellX2D(model, offset,
                                        3);
132 #endif
133                         }
134
135                         if (f[4] > 0) {
136                                 calCudaSet2Dr(model, offset, f[4],
                                        FLOWS);
137 #ifdef ACTIVE_CELLS
138                                 calCudaAddActiveCellX2D(model, offset,
                                        4);
139 #endif
140                         }
```

```
141
142                         if (f[5] > 0) {
143                                 calCudaSet2Dr(model, offset, f[5],
                                        FLOWNO);
144 #ifdef ACTIVE_CELLS
145                                 calCudaAddActiveCellX2D(model, offset,
                                        5);
146 #endif
147                         }
148
149                         if (f[6] > 0) {
150                                 calCudaSet2Dr(model, offset, f[6],
                                        FLOWSO);
151 #ifdef ACTIVE_CELLS
152                                 calCudaAddActiveCellX2D(model, offset,
                                        6);
153 #endif
154                         }
155
156                         if (f[7] > 0) {
157                                 calCudaSet2Dr(model, offset, f[7],
                                        FLOWSE);
158 #ifdef ACTIVE_CELLS
159                                 calCudaAddActiveCellX2D(model, offset,
                                        7);
160 #endif
161                         }
162
163                         if (f[8] > 0) {
164                                 calCudaSet2Dr(model, offset, f[8],
                                        FLOWNE);
165 #ifdef ACTIVE_CELLS
166                                 calCudaAddActiveCellX2D(model, offset,
                                        8);
167 #endif
168                         }
169
170                 }
171 }
172
173 __global__
174         void width_update(struct CudaCALModel2D* model) {
175                 CALint outFlowsIndexes[NUMBER_OF_OUTFLOWS] = { 3, 2, 1,
                        0, 6, 7, 4, 5 };
176                 CALint n;
177                 CALint offset = calCudaGetIndex(model);
178                 CALreal initial_h = calCudaGet2Dr(model, offset,
                        THICKNESS);
179                 CALreal initial_t = calCudaGet2Dr(model, offset,
                        TEMPERATURE);
180                 CALreal residualTemperature = initial_h * initial_t;
181                 CALreal residualLava = initial_h;
182                 CALreal h_next = initial_h;
183                 CALreal t_next;
184
185                 CALreal ht = 0;
186                 CALreal inSum = 0;
187                 CALreal outSum = 0;
188
189                 CALreal inFlow = 0, outFlow = 0, neigh_t = 0;
```

```
190
191                 // n = 1
192                 inFlow = calCudaGetX2Dr(model, offset, 1, FLOWS);
193                 outFlow = calCudaGet2Dr(model, offset, FLOWN);
194                 neigh_t = calCudaGetX2Dr(model, offset, 1, TEMPERATURE);
195                 ht += inFlow * neigh_t;
196                 inSum += inFlow;
197                 outSum += outFlow;
198
199                 // n = 2
200                 inFlow = calCudaGetX2Dr(model, offset, 2, FLOWE);
201                 outFlow = calCudaGet2Dr(model, offset, FLOWO);
202                 neigh_t = calCudaGetX2Dr(model, offset, 2, TEMPERATURE);
203                 ht += inFlow * neigh_t;
204                 inSum += inFlow;
205                 outSum += outFlow;
206
207                 // n = 3
208                 inFlow = calCudaGetX2Dr(model, offset, 3, FLOWO);
209                 outFlow = calCudaGet2Dr(model, offset, FLOWE);
210                 neigh_t = calCudaGetX2Dr(model, offset, 3, TEMPERATURE);
211                 ht += inFlow * neigh_t;
212                 inSum += inFlow;
213                 outSum += outFlow;
214
215                 // n = 4
216                 inFlow = calCudaGetX2Dr(model, offset, 4, FLOWN);
217                 outFlow = calCudaGet2Dr(model, offset, FLOWS);
218                 neigh_t = calCudaGetX2Dr(model, offset, 4, TEMPERATURE);
219                 ht += inFlow * neigh_t;
220                 inSum += inFlow;
221                 outSum += outFlow;
222
223                 // n = 5
224                 inFlow = calCudaGetX2Dr(model, offset, 5, FLOWSE);
225                 outFlow = calCudaGet2Dr(model, offset, FLOWNO);
226                 neigh_t = calCudaGetX2Dr(model, offset, 5, TEMPERATURE);
227                 ht += inFlow * neigh_t;
228                 inSum += inFlow;
229                 outSum += outFlow;
230
231                 // n = 6
232                 inFlow = calCudaGetX2Dr(model, offset, 6, FLOWNE);
233                 outFlow = calCudaGet2Dr(model, offset, FLOWSO);
234                 neigh_t = calCudaGetX2Dr(model, offset, 6, TEMPERATURE);
235                 ht += inFlow * neigh_t;
236                 inSum += inFlow;
237                 outSum += outFlow;
238
239                 // n = 7
240                 inFlow = calCudaGetX2Dr(model, offset, 7, FLOWNO);
241                 outFlow = calCudaGet2Dr(model, offset, FLOWSE);
242                 neigh_t = calCudaGetX2Dr(model, offset, 7, TEMPERATURE);
243                 ht += inFlow * neigh_t;
244                 inSum += inFlow;
245                 outSum += outFlow;
246
247                 // n = 8
248                 inFlow = calCudaGetX2Dr(model, offset, 8, FLOWSO);
249                 outFlow = calCudaGet2Dr(model, offset, FLOWNE);
```

```
250                 neigh_t = calCudaGetX2Dr(model, offset, 8, TEMPERATURE);
251                 ht += inFlow * neigh_t;
252                 inSum += inFlow;
253                 outSum += outFlow;
254 
255                 h_next += inSum - outSum;
256                 calCudaSet2Dr(model, offset, h_next, THICKNESS);
257                 if (inSum > 0 || outSum > 0) {
258                         residualLava -= outSum;
259                         t_next = (residualLava * initial_t + ht) / (
                                residualLava + inSum);
260                         calCudaSet2Dr(model, offset, t_next, TEMPERATURE
                                );
261                 }
262 }
263 
264 __global__
265         void updateTemperature(struct CudaCALModel2D* model) {
266                 CALreal nT, h, T, aus;
267                 CALint offset = calCudaGetIndex(model);
268                 CALreal sh = calCudaGet2Dr(model, offset, THICKNESS);
269                 CALreal st = calCudaGet2Dr(model, offset, TEMPERATURE);
270                 CALreal sz = calCudaGet2Dr(model, offset, ALTITUDE);
271 
272                 if (sh > 0 && !calCudaGet2Db(model, offset,
                        TOPOGRAPHY_BOUND)) {
273                         h = sh;
274                         T = st;
275                         if (h != 0) {
276                                 nT = T;
277 
278                                 aus = 1.0 + (3 * pow(nT, 3.0) * Pepsilon
                                         * Psigma * Pclock * Pcool) / (Prho
                                        * Pcv * h * cell_size * cell_size);
279                                 st = nT / pow(aus, 1.0 / 3.0);
280                                 calCudaSet2Dr(model, offset, st,
                                        TEMPERATURE);
281 
282                         }
283 
284                         //solidification
285                         if (st <= PTsol && sh > 0) {
286                                 calCudaSet2Dr(model, offset, sz + sh,
                                        ALTITUDE);
287                                 calCudaSetCurrent2Dr(model, offset,
                                        calCudaGet2Dr(model, offset,
                                        SOLIDIFIED) + sh, SOLIDIFIED);
288                                 calCudaSet2Dr(model, offset, 0,
                                        THICKNESS);
289                                 calCudaSet2Dr(model, offset, PTsol,
                                        TEMPERATURE);
290 
291                         } else
292                                 calCudaSet2Dr(model, offset, sz,
                                        ALTITUDE);
293                 }
294 }
295 
296 __global__
297         void removeActiveCells(struct CudaCALModel2D* model) {
```

```
298                 CALint offset = calCudaGetIndex(model);
299                 CALreal st = calCudaGet2Dr(model, offset, TEMPERATURE);
300                 if (st <= PTsol && !calCudaGet2Db(model, offset,
                        TOPOGRAPHY_BOUND))
301                     calCudaRemoveActiveCell2D(model, offset);
302 }
303
304 __global__
305         void steering(struct CudaCALModel2D* model) {
306
307                 CALint offset = calCudaGetIndex(model);
308
309                 calCudaInit2Dr(model, offset, 0, FLOWN);
310                 calCudaInit2Dr(model, offset, 0, FLOWO);
311                 calCudaInit2Dr(model, offset, 0, FLOWE);
312                 calCudaInit2Dr(model, offset, 0, FLOWS);
313                 calCudaInit2Dr(model, offset, 0, FLOWNO);
314                 calCudaInit2Dr(model, offset, 0, FLOWSO);
315                 calCudaInit2Dr(model, offset, 0, FLOWSE);
316                 calCudaInit2Dr(model, offset, 0, FLOWNE);
317
318                 if (calCudaGet2Db(model, offset, TOPOGRAPHY_BOUND) ==
                        CAL_TRUE) {
319                     calCudaSet2Dr(model, offset, 0, THICKNESS);
320                     calCudaSet2Dr(model, offset, 0, TEMPERATURE);
321                 }
322 }
323
324 __device__
325         void evaluatePowerLawParams(CALreal value_sol, CALreal
                value_vent, CALreal &k1, CALreal &k2) {
326                 k2 = (log10(value_vent) - log10(value_sol)) / (PTvent -
                        PTsol);
327                 k1 = log10(value_sol) - k2 * (PTsol);
328 }
329
330 __device__
331         void MakeBorder(CudaCALModel2D* model) {
332
333                 CALint i, j, offset = calCudaGetIndex(model);
334
335                 i = calCudaGetIndexRow(model, offset);
336                 j = calCudaGetIndexColumn(model, offset);
337
338                 //prima riga
339                 if(i == 0){
340                         if (calCudaGet2Dr(model, offset, ALTITUDE) >= 0)
                                {
341                             calCudaSetCurrent2Db(model, offset,
                                    CAL_TRUE, TOPOGRAPHY_BOUND);
342 #ifdef ACTIVE_CELLS
343                             calCudaAddActiveCell2D(model, offset);
344 #endif
345                         }
346                 }
347
348                 //ultima riga
349                 if(i == rows - 1){
350
351                         if (calCudaGet2Dr(model, offset, ALTITUDE) >= 0)
```

```
                                {
352                                 calCudaSetCurrent2Db(model, offset,
                                        CAL_TRUE, TOPOGRAPHY_BOUND);
353 #ifdef ACTIVE_CELLS
354                                 calCudaAddActiveCell2D(model, offset);
355 #endif
356                         }
357
358
359                 }
360
361
362                 //prima colonna
363                 if( j == 0 ){
364                         if (calCudaGet2Dr(model, offset, ALTITUDE) >= 0)
                                {
365                                 calCudaSetCurrent2Db(model, offset,
                                        CAL_TRUE, TOPOGRAPHY_BOUND);
366 #ifdef ACTIVE_CELLS
367                                 calCudaAddActiveCell2D(model, offset);
368 #endif
369                         }
370                 }
371
372
373                 //ultima colonna
374                 if(j == cols - 1){
375                         if (calCudaGet2Dr(model, offset, ALTITUDE) >= 0)
                                {
376                                 calCudaSetCurrent2Db(model, offset,
                                        CAL_TRUE, TOPOGRAPHY_BOUND);
377 #ifdef ACTIVE_CELLS
378                                 calCudaAddActiveCell2D(model, offset);
379 #endif
380                         }
381
382                 }
383
384                 if( i > 0 && j > 0 && i < rows - 1 && j < cols - 1){
385                         if (calCudaGet2Dr(model, offset, ALTITUDE) >= 0)
                                {
386                                 for (int k = 1; k < model->sizeof_X; k
                                        ++)
387                                         if (calCudaGetX2Dr(model, offset
                                                , k, ALTITUDE) < 0) {
388                                                 calCudaSetCurrent2Db(
                                                        model, offset,
                                                        CAL_TRUE,
                                                        TOPOGRAPHY_BOUND);
389 #ifdef ACTIVE_CELLS
390                                                 calCudaAddActiveCell2D(
                                                        model, offset);
391 #endif
392                                                 break;
393                                         }
394                         }
395                 }
396
397 }
398
```

```
399 __global__
400         void simulationInitialize(struct CudaCALModel2D* model) {
401
402                 //dichiarazioni
403                 unsigned int maximum_number_of_emissions = 0;
404
405                 CALint offset = calCudaGetSimpleOffset();
406
407                 calCudaInit2Dr(model, offset, 0.000000, THICKNESS);
408
409                 //TODO single layer initialization
410                 calCudaInit2Db(model, offset, CAL_FALSE,
                        TOPOGRAPHY_BOUND);
411
412                 calCudaInit2Dr(model, offset, 0, FLOWN);
413                 calCudaInit2Dr(model, offset, 0, FLOWO);
414                 calCudaInit2Dr(model, offset, 0, FLOWE);
415                 calCudaInit2Dr(model, offset, 0, FLOWS);
416                 calCudaInit2Dr(model, offset, 0, FLOWNO);
417                 calCudaInit2Dr(model, offset, 0, FLOWSO);
418                 calCudaInit2Dr(model, offset, 0, FLOWSE);
419                 calCudaInit2Dr(model, offset, 0, FLOWNE);
420
421                 //definisce il bordo della morfologia
422                 MakeBorder(model);
423
424                 //calcolo a b (parametri viscosità) c d (parametri
                        resistenza al taglio)
425                 evaluatePowerLawParams(Pr_Tsol, Pr_Tvent, a, b);
426                 evaluatePowerLawParams(Phc_Tsol, Phc_Tvent, c, d);
427
428 #ifdef ACTIVE_CELLS
429
430                 if(calCudaGet2Di(model,offset,VENTS) == 1){
431                         calCudaAddActiveCell2D(model, offset);
432                 }
433                 if(calCudaGet2Di(model,offset,VENTS) == 2){
434                         calCudaAddActiveCell2D(model, offset);
435                 }
436
437 #endif
438
439
440 }
```

Il cuore dell'implementazione riguarda la scrittura dei kernel e delle funzioni device. In particolare i kernel rappresentano le funzioni elementari dell'automa cellulare. Rispetto alla versione sequenziale mostrata al paragrafo 4.4 si notano molte differenze, così come accade per Game of Life (5.4).

I sottostati in OpenCAL vengono gestiti tramite la struttura `CALSubstate2D(r|i|b)`, come spiegato nei paragrafi precedenti tutto ciò in OpenCAL-CUDA non accade. Tutti i sottostati, per ogni tipo di dato, sono rappresentati da un'unica struttura dati lineare. L'utilizzo di enumerativi, che rappresentano dunque l'indice per ogni sottostato nel-

la matrice linearizzata, garantisce non solo la leggibilità del codice ma favorisce la comprensione per un eventuale manutenzione al codice. L'enumerativo riesce a dare un'idea precisa del sottostato che viene chiamato in causa.

```
enum SUBSTATES_NAMES_REAL{
        ALTITUDE=0,THICKNESS,TEMPERATURE,PRE_EVENT_TOPOGRAPHY,
            SOLIDIFIED, FLOWN,FLOWO,FLOWE,FLOWS, FLOWNO, FLOWSO, FLOWSE,
            FLOWNE
};
enum SUBSTATES_NAMES_INT{
        VENTS=0,
};
enum SUBSTATES_NAMES_BYTE{
        TOPOGRAPHY_BOUND=0,
};

int main(){
        /* ... */

        calCudaGet2Dr(model, offset, THICKNESS);
        calCudaSet2Dr(model, offset, PTvent, TEMPERATURE);
        calCudaSet2Dr(model, offset, calCudaGet2Dr(model, offset,
            THICKNESS) + emitted_lava, THICKNESS);

        /* ... */
}
```

Codice 5.6: Esempio di utilizzo degli enumerativi in OpenCAL-CUDA con diversi sottostati

Infine il *main* in cui vengono definite tutte le configurazioni e le proprietà dell'automa cellulare. Viene definito il modello tramite la funzione `calCudaCADef2D` e avviata la simulazione tramite la funzione `calCudaRun2D`.

```
int main()
{
        time_t start_time, end_time;

        cudaProfilerStart();

        //Model and simulation definitions
        struct CudaCALModel2D* sciara_fv2;
        struct CudaCALRun2D* simulation_sciara_fv2;

#ifdef ACTIVE_CELLS
        sciara_fv2 = calCudaCADef2D (rows, cols,
            CAL_MOORE_NEIGHBORHOOD_2D, CAL_SPACE_TOROIDAL,
            CAL_OPT_ACTIVE_CELLS);
#else
        sciara_fv2 = calCudaCADef2D (rows, cols,
            CAL_MOORE_NEIGHBORHOOD_2D, CAL_SPACE_TOROIDAL, CAL_NO_OPT);
#endif

```

```
17          //Model allocated on device
18          struct CudaCALModel2D* device_sciara_fv2 = calCudaAlloc();
19
20          //Add transition function's elementary processes
21          calCudaAddElementaryProcess2D(sciara_fv2, updateVentsEmission);
22          calCudaAddElementaryProcess2D(sciara_fv2, empiricalFlows);
23          calCudaAddElementaryProcess2D(sciara_fv2, width_update);
24          calCudaAddElementaryProcess2D(sciara_fv2, updateTemperature);
25
26  #ifdef ACTIVE_CELLS
27          calCudaAddElementaryProcess2D(sciara_fv2, removeActiveCells);
28  #endif
29
30          //Add substates
31          calCudaAddSubstate2Dr(sciara_fv2,NUMBER_OF_SUBSTATES_REAL);
32          calCudaAddSubstate2Di(sciara_fv2,NUMBER_OF_SUBSTATES_INT);
33          calCudaAddSubstate2Db(sciara_fv2,NUMBER_OF_SUBSTATES_BYTE);
34
35          //Load configuration
36          calCudaLoadSubstate2Dr(sciara_fv2, DEM_PATH, ALTITUDE);
37          calCudaLoadSubstate2Di(sciara_fv2, VENTS_PATH, VENTS);
38
39          //calCudaLoadSubstate2Dr(sciara_fv2, THICKNESS_PATH, THICKNESS);
40          calCudaLoadSubstate2Dr(sciara_fv2, TEMPERATURE_PATH, TEMPERATURE
                );
41          calCudaLoadSubstate2Dr(sciara_fv2,
                SOLIDIFIED_LAVA_THICKNESS_PATH, SOLIDIFIED);
42
43          //Copy data from CPU to GPU
44          calInitializeInGPU2D(sciara_fv2,device_sciara_fv2);
45
46          //Check errors otherwise print the message in input
47          cudaErrorCheck("Data initialized on device\n");
48
49          simulation_sciara_fv2 = calCudaRunDef2D(device_sciara_fv2,
                sciara_fv2, 1, STEPS, CAL_UPDATE_IMPLICIT);
50
51          //Add init, steering and stop condition
52          calCudaRunAddInitFunc2D(simulation_sciara_fv2,
                simulationInitialize);
53          calCudaRunAddSteeringFunc2D(simulation_sciara_fv2, steering);
54          calCudaRunAddStopConditionFunc2D(simulation_sciara_fv2,
                stopCondition);
55
56          //Start simulation
57          printf ("Starting simulation...\n");
58          start_time = time(NULL);
59          calCudaRun2D(simulation_sciara_fv2, grid, block);
60
61          //Send data to CPU
62          calSendDataGPUtoCPU(sciara_fv2,device_sciara_fv2);
63
64          cudaErrorCheck("Final configuration sent to CPU\n");
65          end_time = time(NULL);
66          printf ("Simulation terminated.\nElapsed time: %d\n", end_time-
                start_time);
67
68          //Saving configuration
69          calCudaSaveSubstate2Dr(sciara_fv2, O_DEM_PATH, ALTITUDE);
70          calCudaSaveSubstate2Dr(sciara_fv2,
```

```
            O_SOLIDIFIED_LAVA_THICKNESS_PATH, SOLIDIFIED);
71          calCudaSaveSubstate2Dr(sciara_fv2, O_TEMPERATURE_PATH,
                TEMPERATURE);
72          calCudaSaveSubstate2Dr(sciara_fv2, O_THICKNESS_PATH, THICKNESS);
73          calCudaSaveSubstate2Di(sciara_fv2, O_VENTS_PATH, VENTS);
74
75          //Check errors
76          cudaErrorCheck("Data saved on output file\n");
77
78          //Finalizations
79          calCudaRunFinalize2D(simulation_sciara_fv2);
80          calCudaFinalize2D(sciara_fv2, device_sciara_fv2);
81          cudaProfilerStop();
82
83          system("pause");
84          return 0;
85  }
```

Capitolo 6

Test e analisi delle prestazioni della libreria

6.1 Introduzione

Nel settore del parallel computing, i termini di successo di un progetto sono le misure di performance. In questo lavoro di tesi è stata svolta un'analisi attenta dei tempi e delle nostre misure di performance per poter validare il lavoro. In particolare sono stati eseguiti differenti test su più modelli.

Un primo test è stato il confronto della simulazione sequenziale con quella parallela. Una volta accertata la loro stretta somiglianza, e in molti casi l'esatta coincidenza dei valori, sono stati eseguiti ulteriori test riguardo altre misure di performance e i tempi di esecuzione.

Questi ultimi test sono stati significativi poiché son serviti a validare il lavoro di tesi. Si mostreranno nei paragrafi successivi alcuni grafici che rappresentano i risultati dei test effettuati.

Di rilevante importanza è stata la scelta dei processori e delle schede video utilizzate nell'esecuzione di questi test. Per quanto riguarda la simulazione sequenziale sono stati utilizzati due processori Quad Xeon da 2.8 GHz con a disposizione 4 core ciascuno. Questi due processori sono attualmente montati sulla macchina *Stromboli* situata al centro di calcolo ad alte prestazioni (HPCC) dell'**Università della Calabria**. Per quanto riguarda la simulazione parallela sono state utilizzate due schede grafiche diverse: una GPU Nvidia Tesla K20c e una GPU Nvidia GeForce GT750M. La prima è situata, come per la CPU Quad Xeon, al centro di calcolo ad alte prestazioni dell'Unical, mentre la seconda è una delle più

comuni schede video dei moderni laptop.

In forma tabellare è interessante mostrare le caratteristiche tecniche delle due schede grafiche utilizzate:

Tabella 6.1: Specifiche tecniche della GPU Nvidia Tesla K20c e della GPU Nvidia GeForce GT750M.

Specifiche	Tesla K20c	GeForce GT750M
Nome della GPU	GK110	GK107
Data di rilascio	12 Nov 2012	9 Gen 2013
GPU Clock	706 MHz	941 MHz
Memory clock	1300 MHz 5200 MHz eff.	1000 MHz 4000 MHz eff.
Numero di SM	2496	384
Numero di compute units	13	2
Floating point performance	3,524 GFLOPS	722.7 GFLOPS
Dimensione di memoria	5120 MB	2048 MB
Tipo di memoria	GDDR5	GDDR5
Compute Capability	CUDA 3.5	CUDA 3.0

I test sono stati effettuati su diversi tipi di implementazioni di SCIARA-fv2. L'insieme dei dati utilizzato tuttavia non è cambiato. In particolare sono stati utilizzati i dati relativi all'attività svolta dal monte Etna nel 2006. Si è utilizzata la libreria OpenCAL per implementare la versione sequenziale del modello. In particolare è stata implementata anche una versione ottimizzata tramite le celle attive. Si è utilizzata invece, la libreria OpenCAL-CUDA per implementare la versione parallela del modello, e, anche in questo caso è stata implementata una versione ottimizzata tramite la tecnica delle celle attive.

6.2 Confronto con i risultati

Le immagini che verranno mostrate rappresentano i risultati delle simulazioni eseguite con differenti tipologie di configurazione. Nella prima coppia di immagini (6.1) si mostra la simulazione di SCIARA dopo 10000 step con una mappa composta da due crateri. La seconda coppia (6.2), invece si mostra la simulazione di SCIARA dopo 1000 step con una mappa composta da duecento differenti crateri.

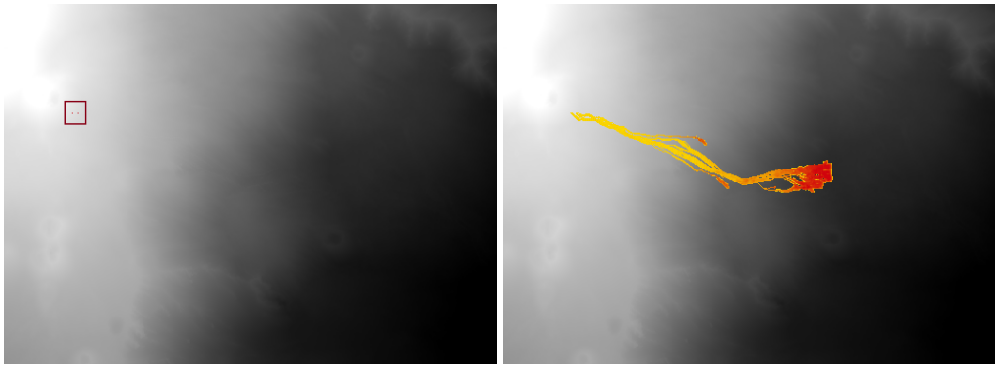


Figura 6.1: Simulazione dopo 10000 passi con due crateri rappresentata tramite il software Qgis

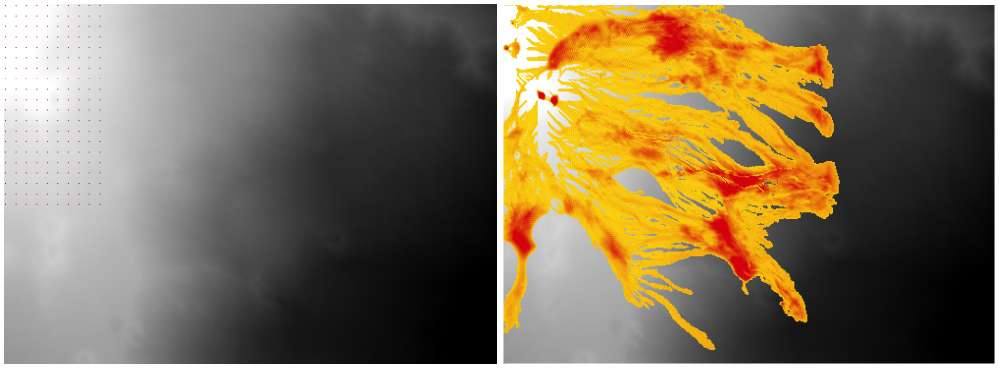


Figura 6.2: Simulazione dopo 1000 passi con duecento crateri rappresentata tramite il software Qgis

Mostriamo ora i grafici relativi ai test effettuati sulle diverse implementazioni. Confronteremo la versione sequenziale, implementata senza l'ottimizzazione delle celle attive, con la versione parallela, implementata anch'essa senza l'utilizzo di alcuna ottimizzazione.

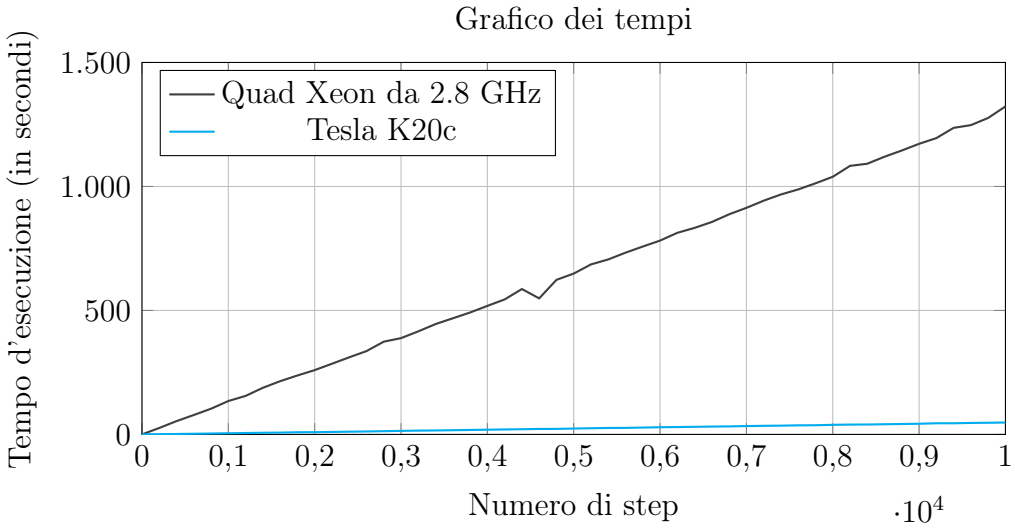


Figura 6.3: La versione sequenziale è stata implementata con la libreria OpenCAL, quella parallela con OpenCAL-CUDA.

Dal grafico si può notare come grazie all'architettura CUDA e la GPGPU programming si possono incrementare le performance drasticamente. Possiamo mostrare dunque, tramite il grafico 6.4, l'andamento della speedup (Vedi paragrafo 1.5) relativo al numero di passi.

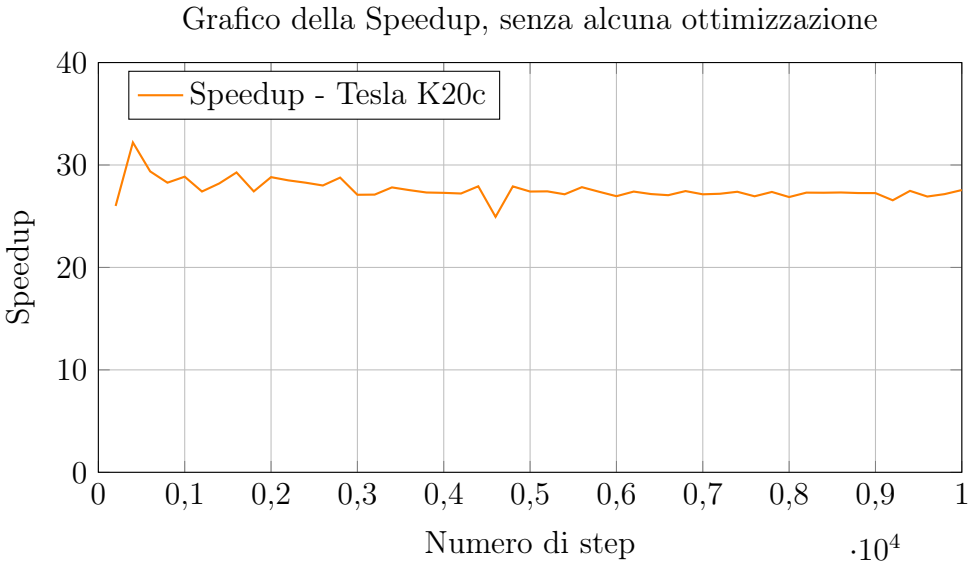


Figura 6.4: Il calcolo della speedup tra implementazione sequenziale e parallela senza ottimizzazioni.

Confrontiamo ora le due implementazioni, sequenziale e parallela, del modello tramite l'ottimizzazione delle celle attive.

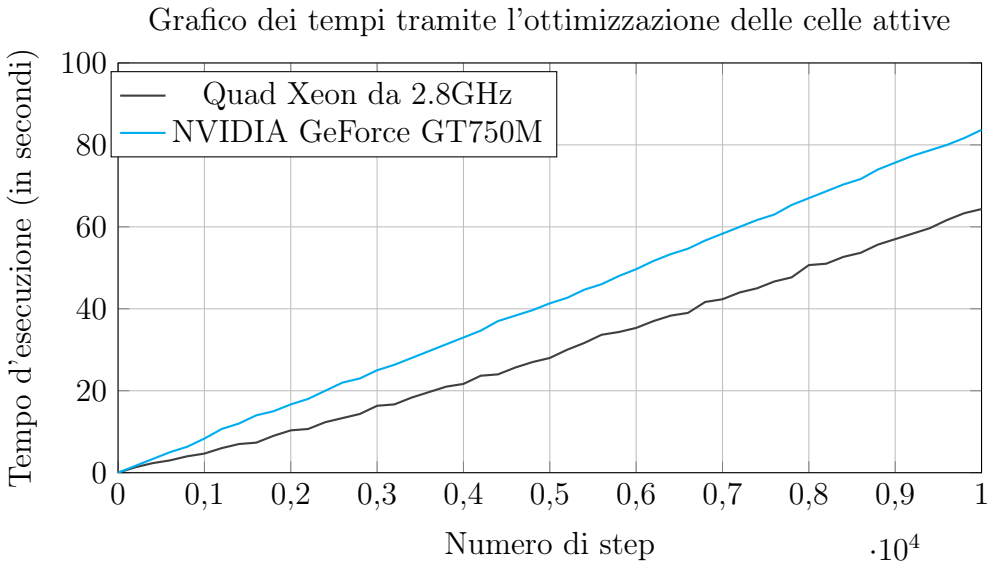


Figura 6.5: La versione sequenziale è stata implementata con la libreria OpenCAL, quella parallela con OpenCAL-CUDA, entrambe le versioni hanno utilizzato l'ottimizzazione delle celle attive.

Nel caso del grafico in fig. 6.5 possiamo notare come la versione sequenziale è più veloce rispetto a quella parallela dopo un certo numero di step. Naturalmente la prima spiegazione plausibile è che le due macchine a confronto non sono equiparabili a livello prestazionale. La Quad Xeon è un processore ad alte prestazioni con un numero di operazioni al secondo di gran lunga superiore ad una GPU come la GT 750M. Il secondo motivo è che tramite l'ottimizzazione delle celle attive, per quanto riguarda il modello SCIARA, vengono chiamate in causa per ogni step poche celle della nostra mappa. Il loro numero ridotto non consente alle tecniche di parallelismo utilizzate di mostrare la loro velocità computazionale.

Dato questo piccolo gap è stata implementata anche una seconda configurazione per testare la libreria. Questa configurazione contiene un solo cambiamento rispetto alla configurazione di base: la mappa dei crateri. Nella nuova mappa dei crateri sono presenti 200 crateri diversi che a loro volta genereranno un flusso più consistente di lava. In seguito mostreremo i grafici dei test eseguiti con questa configurazione e si noterà come la GPGPU programming tramite l'architettura CUDA ha apportato un grosso vantaggio in termini di performance.

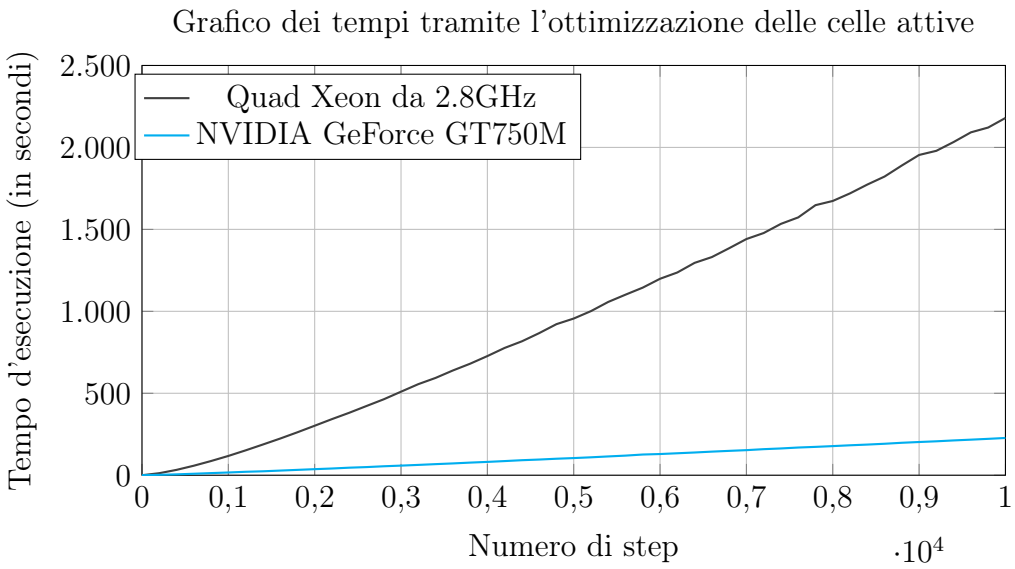


Figura 6.6: La versione sequenziale è stata implementata con la libreria OpenCAL, quella parallela con OpenCAL-CUDA, entrambe le versioni hanno utilizzato l'ottimizzazione delle celle attive. In questo caso sono stati utilizzati 200 crateri nella configurazione iniziale.

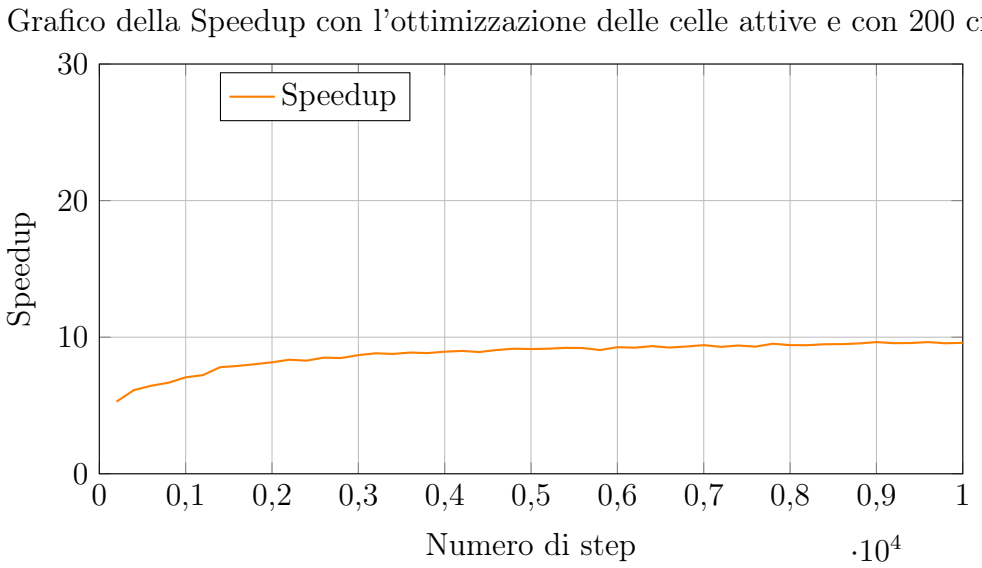


Figura 6.7: Il calcolo della speedup tra implementazione sequenziale e parallela con l'ottimizzazione delle celle attive e una configurazione contenente 200 crateri.

Capitolo 7

Conclusioni

Già da molto tempo l'approccio sistematico al parallel computing ha comportato miglioramenti generali nell'utilizzo dei sistemi informatici. La ricerca basata sull'incremento delle performance dei moderni computer e il calcolo ad alte prestazioni ha trovato campo fertile in numerosi settori dell'informatica tra i quali la modellistica e la simulazione.

L'obiettivo di questo lavoro di tesi è stata la parallelizzazione della libreria per lo sviluppo di modelli basati su Automi Cellulari OpenCAL. Gli Automi Cellulari per loro natura si prestano egregiamente ad un approccio parallelo, proprio per questo è stata immediata la scelta del parallel computing per migliorare le performance della libreria OpenCAL. La versione parallela OpenCAL-CUDA, come si intuisce, è stata implementata tramite l'archiettura CUDA sviluppata e rilasciata dalla società NVIDIA Corporation. In particolare è stato utilizzato il linguaggio CUDA C, estensione del linguaggio C, per l'implementazione del codice parallelo.

Tutte le caratteristiche appartenenti ad OpenCAL sono state mantenute, tuttavia l'implementazione dei modelli e dei loro processi elementari sono state adattate al tipo di architettura utilizzata. In particolare sono presenti alcuni cambiamenti dovuti ad una filosofia implementativa diversa tra l'approccio sequenziale e quello parallelo.

NVIDIA dal 2006 ai giorni nostri, ha rilasciato in maniera frequente aggiornamenti per l'architettura CUDA con numerosi miglioramenti relativi alla leggibilità del codice e alle performance. Le API di CUDA compatibili con i device NVIDIA hanno consentito la realizzazione del progetto.

Gli Automi Cellulari, come spiegato nel capitolo 3, evolvono basandosi sulla funzione di transizione, in particolare per gli Automi Cellulari

Complessi (CCA, 3.4) l'evoluzione dipende da più processi elementari. Questa funzione di transizione viene eseguita allo stesso modo su ogni cella dello spazio cellulare. Questo tipo di approccio viaggia in perfetta sintonia con la filosofia del parallel computing.

In OpenCAL-CUDA vengono creati un numero di blocchi e thread in base al numero di celle dello spazio cellulare in modo tale da assegnare un thread per ogni cella. In questo modo, tutti i thread eseguono la stessa operazione nello stesso momento su celle diverse incrementando le performance e minimizzando i tempi di risposta. Per l'implementazione di un automa cellulare si può utilizzare anche l'ottimizzazione delle celle attive. Questo approccio, che utilizza solamente le celle attive escludendo le celle in stato quiescente, è supportato dalla versione parallela grazie all'utilizzo della stream compaction (par. 5.2.4). La stream compaction ha il compito di elaborare e comprimere i dati sparsi. I dati sparsi nel nostro caso sono il numero di celle attive ad un determinato tempo t. Si istanzieranno dunque un determinato numero di thread in base al numero di celle effettivamente attive.

Dopo aver terminato la parallelizzazione di OpenCAL sono stati implementati diversi modelli con il fine di testare il lavoro di tesi. Tra i vari modelli implementati, quello utilizzato per confrontare i tempi di esecuzione e i vari miglioramenti di performance è stato SCIARA [2] [13]. I test eseguiti si sono basati sull'implementazione del modello SCIARA sia con l'ottimizzazione delle celle attive che senza alcun tipo di ottimizzazione.

Per raccogliere i dati relativi ai tempi di esecuzione per la versione parallela del modello è stata utilizzata la workstation "Stromboli" situata al centro di calcolo ad alte prestazioni dell'Unical, dotata di un processore Quad Xeon da 2.8GHz. Per quanto riguarda le schede grafiche utilizzate sono state scelte due differenti schede: la prima è una Tesla K20c, la seconda una GeForce GT750M entrambe di marca NVIDIA.

Con OpenCAL-CUDA per la versione implementata senza ottimizzazioni raggiunge una speedup di circa 29× in media, utilizzando la scheda Tesla K20c. Per quanto riguarda i test sulla versione con l'ottimizzazione delle celle attive con 200 crateri raggiunge una speedup di circa 10× utilizzando la scheda GeForce GT 750M.

Oggi OpenCAL è un progetto open source avviato. Sono presenti anche diverse implementazioni della libreria tra cui due versioni parallelizzate utilizzando i linguaggi di programmazione OpenCL e OpenMP. Un ulteriore versione della libreria integra OpenGL per la visualizzazione grafica.

Un possibile sviluppo futuro di OpenCAL-CUDA potrebbe essere

l'implementazione della versione 3D, mentre a scopo statistico e di ricerca sarebbe sicuramente interessante effettuare un confronto delle performance tra le varie implementazioni parallelizzate.

Ringraziamenti

Un ringraziamento particolare lo devo a mio padre e mia madre che in tutti gli anni universitari mi hanno sostenuto. Insieme ai miei fratelli, mi hanno stimolato allo studio ed educato alla cultura: il miglior regalo ricevuto. Ringrazio Eleonora per aver appoggiato le mie scelte durante gli ultimi anni universitari e per avermi sempre spronato per raggiungere i miei obiettivi.

Un ringraziamento doveroso lo devo ai professori William Spataro e Donato D'Ambrosio che mi hanno seguito con costanza, fornendomi consigli preziosi per il mio lavoro di tesi. Inoltre ringrazio il dott. Davide Spataro, il dott. Maurizio Macrì e il dott. Alessio De Rango per il loro supporto durante questi ultimi mesi.

Un ultimo ringraziamento va all'Akademia Gorniczo-Hutnicza - University of Science and Technology di Cracovia e al prof. Jaroslaw Was. Grazie alla loro accoglienza e disponibilità hanno reso indimenticabili i miei mesi di studio in Polonia.

Bibliografia

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

[2] D. Barca et al. «Cellular automata for simulating lava flows: A method and examples of the Etnean eruptions». In: *Transport Theory and Statistical Physics* 23.1-3 (1994), pp. 195–232.

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

[4] 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.

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

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

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

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

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

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

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

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

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

[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.