UNIVERSITÀ DEGLI STUDI DI NAPOLI
"PARTHENOPE"
FACOLTÀ DI SCIENZE E TECNOLOGIE
Corso di Laurea in Informatica

Elaborato di Laurea

Simulazione di Processi di Dispersione in Atmosfera Utilizzando GPU in Ambiente CUDA

Relatori:
Prof. Giulio Giunta
Prof. Angelo Riccio

Candidato:
Alfredo Starace
matr. 108/566

Anno Accademico 2007-2008

Abstract

Introdotta ufficialmente nel 2006, la *Compute Unified Device Architecture* (d'ora in poi CUDA™) è una tecnologia realizzata dalla $NVIDIA^{®}$[1]. Questa ha lo scopo di estendere le capacità di elaborazione delle schede video, a compiti che vanno oltre la grafica computazionale.

Rispetto alla prima legge di Moore, secondo la quale le prestazioni dei processori raddoppiano ogni 18 mesi, le Graphics Processing Unit (GPU) delle schede video hanno dimostrato un raddoppio di prestazioni ogni 6 mesi. Ovvero la prima legge di Moore elevata al cubo. Questo è stato possibile grazie all'intenso sviluppo delle architetture parallele sulle quali si basano queste unità. Le loro capacità di calcolo hanno ormai raggiunto l'ordine del Teraflop/s (10^{12} operazioni floating point al secondo) contro gli attuali 100 GFlop/s della Central Processing Unit (CPU) per personal computer più "performante" sul mercato[2]. É chiaro quindi l'interesse ad apprendere il funzionamento e le caratteristiche di questo nuovo tipo di unità di calcolo, al fine di sfruttarne pienamente le potenzialità offerte.

L'obiettivo di questo elaborato di Tesi di Laurea è quello di esaminare i diversi aspetti della tecnologia CUDA, considerandola come una conseguenza diretta del processo evolutivo delle schede video. Viene quindi affrontata la realizzazione di una componente software che utilizza questa tecnologia, al fine di apprenderne l'utilizzo e valutarne l'efficacia. Tale componente consiste nell'implementazione di un modello di dispersione di particelle nell'atmosfera basato su metodi stocastici, un tipo di applicazione molto esigente in termini

[1] NVIDIA Corporation è una azienda leader nella produzione di processori grafici, schede madri e in generale di prodotti multimediali per personal computer e console.

[2] I dati si riferiscono alla GPU di NVIDIA GT200 e alla CPU Intel Xeon™ Harpertown.

di risorse di calcolo e che quindi si presta molto bene a testare le effettive potenzialità delle GPU per il calcolo ad alte prestazioni.

Il presente elaborato di tesi è stato strutturato come segue: dopo il primo capitolo introduttivo nel quale viene presentata una panoramica generale sull'argomento, nel capitolo 2 vengono illustrati i vari modelli stocastici, proposti negli anni dai ricercatori, per la diffusione di particelle negli strati dell'atmosfera. Nel capitolo 3 vengono descritte le GPU, analizzando la loro evoluzione nel tempo, la loro architettura e le loro potenzialità per il calcolo ad alte prestazioni. Nel quarto capitolo viene descritto dettagliatamente il modello dell'applicazione sviluppata evidenziando le varie scelte implementative. Infine, nel capitolo 5, verranno esaminati alcuni possibili sviluppi futuri, e verranno tratte le conclusioni.

"Dal più antico
degli argomenti
trarremo la più nuova
delle scienze"

Hermann Ebbinghaus

Powered by LaTeX

Indice

1 Introduzione **1**

2 Modelli di dispersione atmosferica **8**

2.1 Atmosfera 8

2.2 Modelli di dispersione 13

3 Calcolo scientifico con processori grafici **19**

3.1 GPU 19

3.1.1 Evoluzione delle GPU 22

3.2 Architettura GPU 27

3.2.1 Architettura TESLA 31

3.2.2 Architettura SIMT 37

3.3 Il GPU computing 39

3.4 CUDA 41

4 Implementazione del modello di dispersione nell'atmosfera **56**

4.1 Introduzione al modello 56

4.1.1 Moto Browniano 56

4.1.2 Diffusione turbolenta 59

4.2 Il modello di Luhar e Britter 63

4.2.1 Funzione di distribuzione di probabilità per velocità verticali 65
4.2.2 Derivazione delle equazioni del modello 67
4.3 Generazione di sequenze di numeri casuali 69
4.3.1 Il generatore Mersenne Twister 73
4.3.2 Il generatore Mersenne Twister in CUDA 74
4.4 Implementazione del modello in matlab 75
4.5 Implementazione del modello in CUDA 78
4.6 Analisi delle simulazioni 85

5 Conclusioni e Sviluppi Futuri 92

A Codici sorgente 94
A.1 sorgenti MATLAB 94
A.2 sorgenti CUDA 99

Bibliografia 128

Capitolo 1

Introduzione

La simulazione può essere definita come l'atto di "*imitare*" processi o aspetti della vita reale, per *riprodurli* in via sperimentale. La simulazione viene utilizzata nei più svariati contesti, incluso quello della modellazione dei sistemi naturali ed umani, in modo da poter studiare i meccanismi di funzionamento del sistema, oppure per mostrare i possibili effetti in conseguenza di condizioni differenti.

Le fasi principali di un processo di simulazione comprendono:

- l'acquisizione di fonti di informazioni valide sul soggetto in esame,
- l'individuazione e la selezione delle caratteristiche e dei comportamenti principali da riprodurre nel sistema,
- l'utilizzo di semplificazioni ed assunzioni per effettuare approssimazioni del sistema,
- il controllo della validità dei risultati ottenuti dalla simulazione.

Le simulazioni sono uno strumento sperimentale molto potente e si avvalgono della potenza di calcolo offerta dall'informatica; la simulazione, infatti, non è altro che la trasposizione in termini *logico-matematici-procedurali* di un modello concettuale della realtà; tale modello può essere definito come l'insieme di processi che hanno luogo nel sistema valutato e che può essere espresso in forma di algoritmo. Tradizionalmente, la modellazione formale di un sistema avviene attraverso un modello matematico, che cerca di trovare soluzioni analitiche che consentenono di prevedere il comportamento del sistema rispetto ad una serie di parametri e condizioni iniziali. Poiché un modello matematico è costituito da un numero di equazioni e di parametri che può essere anche molto alto, è di fondamentale importanza trovare il giusto compromesso tra precisione e semplicità del modello. Quando invece per i sistemi da modellare non è possibile trovare una soluzione analitica semplice, si ricorre a simulazioni al computer.

In linea di principio è sempre possibile descrivere i fenomeni della realtà, naturali o artificiali, in termini di sistemi e di processi. Tradizionalmente i modelli di rappresentazione dei fenomeni reali vengono classificati, in funzione di alcune loro caratteristiche notevoli, *statiche* o *dinamiche*, *continue* o *discrete* e *deterministiche* o *stocastiche*. Quasi sempre un modello rappresenta l'evoluzione nel tempo di un sistema reale ed è quindi *dinamico*; inoltre i modelli *continui* si distinguono dai modelli *discreti* per il carattere delle variabili in gioco.

Una differenza concettualmente importante è quella che intercorre tra processi *deterministici*, nei quali, indipendentemente dalla complessità, ogni evento è determinato da una causa o da un insieme di cause che si verificano

contemporaneamente o in sequenza temporale (principio di *causa-effetto* o di causalità), e processi *stocastici*, nei quali un evento è determinato da una causa, o da un insieme di cause, che si realizza con una certa probabilità e in alternativa ad effetti diversi aventi proprie probabilità di realizzazione. Spesso si ricorre ad un modello stocastico per descrivere un sistema complesso che è in realtà deterministico al fine di evitare una rappresentazione dettagliata e proibitiva dei numerosi oggetti, o eventi, e delle loro innumerevoli interazioni. Questa metodologia, permette in alcune situazioni di evitare di riprodurre la complessità di un sistema ricorrendo alla casualità.

Un insieme di particelle rilasciate nell'aria è messo in movimento dagli urti con le molecole d'aria. Queste ultime sono perennemente impegnate in un moto disordinato generato dai loro urti reciproci, un moto incoerente chiamato *agitazione termica*, tale che quanto più è elevata la temperatura dell'aria, tanto maggiore è la velocità delle molecole. La somma di tutti i contributi incoerenti dovuti agli urti è il risultato dello spostamento delle particelle.

Quando però si cerca di stimare l'importanza di questo effetto di diffusione molecolare, ci si rende conto che è del tutto trascurabile su scala macroscopica in condizioni normali. Infatti le molecole impiegherebbero svariate ore per percorrere circa un metro. Oltre al moto individuale, e casuale, di agitazione termica, le molecole sono coinvolte in un movimento collettivo. Quest'ultimo è organizzato in modo coerente su scale spaziali che possono essere dell'ordine di un millimetro, e quindi di svariati ordini di grandezza maggiori di quelli molecolari che sono intorno al miliardesimo di metro. Grazie a questa proprietà non è necessario mettere a fuoco il moto delle singole molecole, ma

è sufficiente seguire la traiettoria di una piccola porzione d'aria, per esempio delle dimensioni di un millimetro cubo, la cui traiettoria è rappresentativa del moto medio delle molecole cha la compongono.

Uno dei fenomeni più affascinati che è possibile osservare nel moto dei fluidi, e quindi anche dell'aria, è sicuramente la turbolenza, cioè il moto altamente irregolare e imprevedibile del fluido che lo rende estremamente efficace nel disperdere le particelle. Tenendo conto del moto irregolare dell'aria, si può stimare che un gruppo di molecole inizialmente localizzate in una regione di pochi millimetri cubi si separeranno fino a distanze di circa un metro nell'arco di secondi. Se le molecole in esame sono costituite da polveri inquinanti, o da atomi radioattivi oppure da batteri nocivi, si comprende l'importanza pratica di avere una descrizione precisa della dispersione turbolenta, che ha un ruolo chiave in un numero impressionante di fenomeni.

Lo studio sperimentale del moto delle particelle in un flusso turbolento si è però dimostrato particolarmente complesso e difficile. Diversi studi sono stati condotti seguendo le traiettorie di traccianti in atmosfera. Ma questo approccio presenta diversi svantaggi, principalmente dovuti alla grande variabilità della circolazione atmosferica e al costo elevato delle sonde. Ne consegue che è praticamente impossibile effettuare esperimenti con un elevato numero di traccianti in condizioni controllate e ripetibili. Per queste ragioni si è tentata la via degli esperimenti di laboratorio, che possono essere condotti a costi più contenuti.

In questo caso si possono seguire le traiettorie di minuscole sfere di metallo o *plexiglas* che riflettono la luce emessa da un laser. Tuttavia, non è possibile seguire il percorso compiuto dalle particelle con un'elevata risoluzione tem-

porale e per intervalli di tempo sufficientemente lunghi. Inoltre, non appena ci si pone l'obiettivo di seguire più di un gruppo di particelle simultaneamente diventa molto difficile attribuire senza ambiguità le traccie luminose alle corrispondenti particelle.

Per superare questa situazione di stallo si ricorre allo studio dell'evoluzione del fluido simulandola al computer. Tuttavia anche in quest'ultimo caso le difficoltà non mancano. Per calcolare l'evoluzione di una porzione di fluido di un metro cubo, composto cioè da un miliardo di particelle fluide di un millimetro cubo di volume ciascuna, per una durata di tempo apprezzabile, il computer deve effettuare circa un milione di miliardi di operazioni. Diventa quindi necessario fare ricorso a supercalcolatori.

La simulazione numerica è ormai da tempo una terza via di indagine scientifica, accanto ai tradizionali metodi della ricerca teorica e sperimentale, tanto che si parla ormai di *scienze computazionali*. Il principale vantaggio della simulazione numerica è la sua versatilità, che consente di esplorare regimi di comportamento spesso inaccessibili alle teorie analitiche e difficilmente realizzabili per via sperimentale, oppure semplicemente troppo costosi in termini economici o temporali. Oggigiorno la ricerca scientifica, di base e applicativa, in campi quali la meteorologia, la geofisica e la modellizzazione industriale, viene condotta con un forte supporto numerico-computazionale all'attività sperimentale. Questo motiva la progettazione e l'uso di strumenti di calcolo sempre più potenti, in grado di eseguire più attività contemporaneamente, in parallelo.

L'affacciarsi sul mercato di schede grafiche programmabili, sta aprendo una nuova strada nell'ambito del calcolo scientifico ad alte prestazioni, por-

tando ad una crescita d'interesse nell'applicare alle *Graphic Processing Unit* (GPU) algoritmi general purpose. La combinazione di memorie a banda larga ed hardware dotato di aritmetica *floating-point*, con prestazioni maggiori rispetto ad una CPU convenzionale, rendono i processori grafici strumenti potenti per l'implementazione di algoritmi di calcolo. Gli strumenti utilizzati nel campo del GPGPU (*General-Purpose Computation using Graphics Processing Units*) sono quelli tipici della programmazione in ambiente grafico. Strumenti molto potenti, portabili e con un livello di sviluppo e stabilità ormai assodati.

Nelle istanze moderne le GPU si presentano come dispositivi estremamente performanti, infatti utilizzando una GPU di fascia media (NVIDIA 7800 GTX 512) si raggiungono prestazioni dell'ordine dei 200 *Gflops/s*, nettamente superiori ai 12 *Gflop/s* ottenibili su un *Pentium4 3 GHz*. I punti di forza con cui le GPU riescono a ottenere picchi di tale potenza possono essere sostanzialmente identificati in:

- frequenza di clock;
- velocità della memoria primaria;
- volume delle informazioni processate in parallelo.

Nel corso degli anni si è assistito a un vertiginoso miglioramento dei tre parametri appena elencati, con un drastico aumento delle possibilità offerte da questi dispositivi. Attualmente, il picco massimo di prestazioni raggiungibili è dell'ordine di un Teraflop/s, grazie alla GPU NVIDIA GeForce GTX 280. Prestazioni nettamente superiori a quelle delle tradizionali CPU,

paragonabili a quelle di alcuni supercomputer e cluster di computer e quindi sufficienti per simulazioni legate alla dispersione atmosferica.

Capitolo 2

Modelli di dispersione atmosferica

2.1 Atmosfera

La Terra è circondata e protetta da un involucro gassoso, composto da azoto, ossigeno, anidride carbonica ed altri gas, che prende il nome di atmosfera. L'atmosfera è formata da diversi strati, che presentano caratteristiche fisiche e chimico-fisiche variabili con la quota, quali ad esempio la temperatura, la pressione e la densità. A causa della forza di gravità l'atmosfera è stratificata, con strati più densi in prossimità della superficie. L'andamento che indica le variazioni di temperatura ΔT in rapporto alla differenza di quota Δz è detto *gradiente termico verticale*; se la temperatura cresce con la quota il gradiente è positivo, se invece decresce è negativo. L'atmosfera terrestre ha una struttura piuttosto complessa ed è divisa in più strati:

- *troposfera*

è lo strato in cui si verificano quasi tutti i fenomeni meteorologici, causati dalla circolazione delle masse d'aria che danno vita ai venti, alle nuvole e alle precipitazioni atmosferiche. Contiene l'80% della massa gassosa e il 99% del vapore acqueo di tutta l'atmosfera. L'aria della troposfera è riscaldata dalla superficie terrestre ed ha una temperatura media globale di 15 °C al livello del mare, che diminuisce con l'altitudine (0,65 °C ogni 100m di quota) fino ai circa -60 °C della *tropopausa*. L'aria degli strati più bassi genera grandi correnti convettive da cui hanno origine i venti equatoriali e le perturbazioni atmosferiche. La troposfera ha uno spessore variabile a seconda della latitudine: ai poli è spessa solamente $8km$ mentre raggiunge i 17 km all'equatore. La pressione atmosferica decresce con l'altitudine secondo una legge esponenziale; oltre i 7-8 km di quota la pressione è tanto bassa che non è più possibile respirare senza l'uso di maschere collegate a bombole di ossigeno. Salendo in quota, oltre a pressione e temperatura, diminuisce anche il contenuto di vapore acqueo dell'aria. Ad un certo punto la temperatura si stabilizza a -60 °C circa: è la *tropopausa*, la zona di transizione fra *troposfera* e *stratosfera*.

- *stratosfera*

  È lo strato atmosferico che sta al di sopra della *troposfera* ed arriva ad un'altezza di 50-60 km. Qui avviene il fenomeno chiamato dell'inversione termica: cioè, mentre nella *troposfera* la temperatura diminuisce con l'altezza, nella *stratosfera* aumenta, fino alla temperatura di 0 °C. Questo fenomeno è dovuto alla presenza di uno strato di ozono, l'*ozonosfera*, che assorbe la maggior parte (circa il 99%) delle radi-

azioni solari ultraviolette. In alcuni punti dell'*ozonosfera* lo strato di ozono si è assottigliato (fenomeno del buco nell'ozono) al punto tale che non offre più un efficace protezione ai raggi ultravioletti che, in queste condizioni, riescono a giungere a terra. Questi raggi causano seri danni alle piante e a tutti gli esseri viventi. I danni all'uomo possono essere tumori alla pelle e cecità, a causa di danni irreversibili alla retina. Il buco nell'ozono è un fenomeno particolarmente accentuato nella zona antartica. Nella stratosfera i componenti si presentano sempre più rarefatti, il vapore acqueo e il pulviscolo diminuiscono, anche se esistono ancora alcuni rari fenomeni meteorologici e certi particolari tipi di nubi detti cirri.

- *mesosfera*

  In questa zona, che va dai 50 agli 80 km di quota, l'atmosfera non subisce più l'influsso della superficie terrestre ed è costante a tutte le latitudini. Non ci sono più né venti o correnti ascensionali, né nubi o perturbazioni, l'aria è completamente calma. In queste condizioni i gas si stratificano per diffusione e la composizione chimica media dell'aria inizia a variare a mano a mano che si sale. Il biossido di carbonio scompare rapidamente, il vapore acqueo ancora più in fretta e anche la percentuale di ossigeno inizia a diminuire con la quota. Aumentano le percentuali di gas leggeri come elio e idrogeno. L'effetto riscaldante dell'ozono cessa e la temperatura diminuisce sempre più con la quota fino a stabilizzarsi al limite superiore della *mesosfera* (-80 °C nella *mesopausa*).

  In questo strato hanno origine le stelle cadenti, cioè i piccoli meteoriti

che di solito non riescono a raggiungere la superficie terrestre e bruciano prima di raggiungere la Terra, lasciando scie luminose. Oltre la *mesopausa*, alla quota di circa 100 km, l'aria è tanto rarefatta da non opporre una resistenza tangibile al moto dei corpi, e diventa possibile muoversi con il moto orbitale. Per questo motivo, in astronautica la *mesopausa* viene considerata il confine con lo spazio.

- *termosfera*

  La *termosfera* è lo strato di atmosfera in cui i gas atmosferici sono fortemente ionizzati: è costituita dagli strati esterni dell'atmosfera, esposti alla radiazione solare diretta che strappa gli elettroni dagli atomi e dalle molecole. Contiene, nel suo insieme, una frazione minima della massa gassosa atmosferica, circa l'1%, ma ha uno spessore di alcune centinaia di chilometri e assorbe buona parte delle radiazioni ionizzanti provenienti dallo spazio. La temperatura in questo strato sale con l'altitudine, per l'irraggiamento solare, ed arriva ai 1700 °C al suo limite esterno.

  Ha una struttura a bande, divise durante il giorno dalla forte radiazione solare che ionizza preferenzialmente gas diversi a quote diverse: durante la notte alcune di queste bande si fondono insieme, aumentando la riflettività radio della *termosfera*.

  Al confine fra *mesopausa* e *termosfera* hanno luogo le aurore boreali.

  La composizione chimica è ancora simile a quella media, con una predominanza di azoto e ossigeno, ma cambia sempre più con l'altitudine. A circa 550 km di quota, questi due gas cessano di essere i componenti principali dell'atmosfera, e vengono spodestati da elio e idrogeno.

La termosfera riveste una grande importanza nelle telecomunicazioni perché è in grado di riflettere le onde radio, aiutandole a propagarsi oltre la portata visibile: tra i 60 e gli 80km vengono riflesse le onde lunghe, tra i 90 e i 120 le onde medie, tra i 200 e i 250 le onde corte, tra i 400 e i 500km le onde cortissime.

- *esosfera*

  È la parte più esterna dell'atmosfera terrestre, dove la composizione chimica cambia radicalmente. L'esosfera non ha un vero limite superiore, arrivando a comprendere anche *le fasce di Van Allen*. I suoi costituenti sono soprattutto idrogeno ed elio, in maggioranza particelle del vento solare catturate dalla magnetosfera terrestre. Tramite metodi di osservazione indiretti e da calcoli teorici si ricava che la temperatura dell'esosfera aumenta con l'altezza fino a raggiungere, se non addirittura superare, i 2000 °C.

Gli strati al di sopra della *mesosfera*, sono poco significativi nella modellazione della dispersione atmosferica. La parte più bassa della *troposfera* si chiama *strato limite atmosferico* (*Atmosheric Boundary Layer* o *ABL*) e si estende dalla superficie terrestre fino a circa 1.5-2.0 km di altezza. Lo strato limite atmosferico è il più importante per quanto riguarda le emissioni, il trasporto e la dispersione di sostanze inquinanti, infatti la maggior parte dei modelli esistenti sono riferiti proprio a questo strato. Quando le condizioni meteo sono instabili lo strato limite atmosferico prende il nome di strato limite convettivo (*Convective Boundary Layer* o *CBL*).

2.2 Modelli di dispersione

La modellazione della dispersione atmosferica è la simulazione matematica di come le particelle o gli inquinanti dell'aria si disperdono negli strati atmosferici. Viene eseguita con appositi software che risolvono le equazioni matematiche ed eseguono gli algoritmi che simulano la dispersione. I modelli di dispersione sono usati per stimare o prevedere la concentrazione sottovento degli inquinanti dell'aria prodotti da fonti come impianti industriali o dal traffico veicolare. Questi modelli sono importanti per le agenzie governative che hanno il compito di proteggere e gestire la qualità dell'aria dell'ambiente. I modelli vengono tipicamente utilizzati per determinare se un impianto industriale esistente o di prossima costruzione sarà conforme agli standard per la qualità dell'aria come il NAAQS (*National Ambient Air Quality Standards*). Tali modelli servono anche per assistere alla progettazione di edifici e di strategie di controllo per la riduzione dell'emissione di inquinanti nocivi. Mediante l'utilizzo dei modelli di dispersione si può inoltre analizzare l'efficacia dei diversi scenari di riduzione, modifica, rilocalizzazione delle emissioni, tenendo conto delle complesse interrelazioni tra i diversi fenomeni che avvengono in atmosfera, comparando poi i risultati ottenuti con i costi stimabili per i diversi scenari di intervento.

Simulare il comportamento di particelle, rilasciate in atmosfera, significa determinare il campo di concentrazione da esse prodotto in qualunque punto dello spazio e in qualunque istante successivo all'emissione. Esistono numerosi modelli previsionali che si differenziano, oltre che per i presupposti teorici e per la struttura più o meno sofisticata, anche per gli ambiti in cui possono essere applicati. I modelli di rappresentazione dei fenomeni di

diffusione atmosferica si dividono fondamentalmente in due diverse tipologie:

- *modelli fisici*

  modelli tipici per la rappresentazione in scala ridotta (scala di laboratorio) di fenomeni in microscala o scala locale (ad esempio gallerie del vento o canalette idrauliche) ma anche per la caratterizzazione di fenomeni a scala regionale (tavole idrauliche rotanti, per lo studio dei fenomeni rotazionali, effetto di Coriolis);

- *modelli matematici*

  algoritmi numerici o analitici rappresentativi dei fenomeni fisici, chimici e biologici che prendono parte alla diffusione e trasporto degli inquinanti in atmosfera e alla loro interazione con i sistemi, risolvendo un insieme di equazioni. I modelli numerici permettono di ottenere soluzioni approssimate mediante metodologie di integrazione numerica, a differenza di quelli analitici che consentono di ottenere soluzioni analitiche esatte o approssimate mediante l´impiego di una serie di equazioni di tipo parametrico.

In generale, i modelli matematici, relativi alla dispersione in atmosfera, possono essere divisi, a loro volta, in due grandi categorie:

- *modelli deterministici*

  che cercano di seguire il fenomeno del trasporto delle particelle in atmosfera mediante trattazione teorica dei fenomeni connessi alla diffusione atmosferica. L'approccio teorico al problema ha portato alla predisposizione di modelli Euleriani, Lagrangiani e modelli cinematici Gaussiani o analitici;

- *modelli stocastici*

  per lo più utilizzati in fase di descrizione e gestione dei dati misurati dalle reti di monitoraggio della qualità dell'aria, si basano sull'analisi delle serie storiche dei dati misurati. Lo studio statistico dei dati permette di verificare relazioni tra grandezze relative sia agli inquinanti che alla meteorologia, risultando quindi di grande aiuto nell'interpretazione dei fenomeni che governano la diffusione delle particelle e nell'analisi dei dati stessi.

Esistono inoltre modelli *misti*, in parte deterministici e in parte statistici. Tali modelli adottano metodi semi empirici o filtri in tempo reale che aggiustano le previsioni di un modello deterministico, a mano a mano che misure reali diventano disponibili.

Per quanto riguarda i modelli deterministici le due tecniche per la modellazione della dispersione turbolenta più utilizzate sono:

- *approccio Euleriano*

  si basa su un sistema di coordinate fissato nello spazio. Le proprietà di un fluido come la densità, la temperatura e la velocità sono indicate in uno specifico punto (x, y, z) in un dato tempo t. Il modello descrive la concentrazione della specie rispetto ad un volume di controllo inserito nel sistema di coordinate.

- *approccio Lagrangiano*

  utilizza un sistema coordinate basato sulla posizione (x, y, z) di una particella al tempo t relativo alla propria posizione (a, b, c) rispetto al tempo t_0. Descrive i cambiamenti di concentrazione rispetto al fluido in movimento, cioè la traiettoria delle particelle del fluido.

Ai modelli stocastici invece sono stati dati diversi nomi, per esempio *random displacement* o *Langevin model*. Nel modello di random displacement, la traiettoria delle particelle è il vettore somma degli spostamenti incrementali casuali. Nel modello di Langevin, invece, i cambiamenti stocastici incrementali della velocità Lagrangiana sono integrati rispetto al tempo per definire la traiettoria delle particelle nello spazio. In alternativa si può considerare il modello di *accelerazione casuale*, nel quale i cambiamenti incrementali stocastici della velocità sono integrati per ottenere la velocità e la disposizione. I metodi *Monte Carlo* o quelli basati su *catene di Markov*, sono invece dizioni generiche in quanto possono essere basati sia sull'equazione di *Langevin*, sia su tecniche di *random displacement* correlate nel tempo.

I vari modelli di dispersione richiedono l'immissione di dati che comprendono:

- condizioni meteorologiche, quali velocità e direzione del vento, l'energia cinetica turbolenta e la temperatura dell'aria.
- parametri di emissione, come l'altezza e la posizione della fonte, diametro, velocità e tipo[1] di diffusione.
- descrizione dei punti di elevazione del terreno
- la posizione, l'altezza e la larghezza di eventuali ostacoli nel percorso delle emissioni.

Uno dei moti più interessanti dei fluidi dal punto di vista pratico è la turbolenza, una forma di movimento apparentemente disordinato il cui es-

[1] i tipi di diffusione modellati in genere sono due: a rilascio istantaneo (*puff*) e a rilascio continuo (*plume*) (fig 2.1)

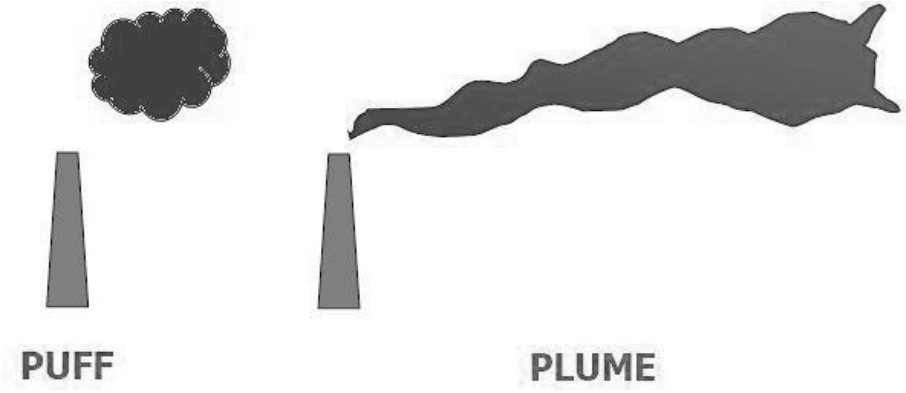

Figura 2.1: I due tipi di rilascio più diffusi.

ito diviene rapidamente imprevedibile. Nel fenomeno della turbolenza, si ritrovano altri due fenomeni fisici, il *moto Browniano* e la *diffusione turbolenta*, che devono essere tenuti in considerazione nella realizzazione del modello.

Come osservato dal botanico Robert Brown nel 1827, e spiegato teoricamente da Einstein nel 1905, una particella microscopica immersa in un fluido a riposo è soggetta ad un moto erratico conseguente alle collisioni con le molecole. La dispersione di un insieme di particelle soggette a moto Browniano, aumenta linearmente nel tempo secondo un fattore di proporzionalità D_0, detto coefficiente di diffusione, il cui valore è inversamente proporzionale al diametro delle particelle e proporzionale alla temperatura del fluido. Per situazioni macroscopiche, come la dispersione di inquinanti in atmosfera, è in pratica irrilevante a causa del ridotto valore di D_0. Nel 1926 il meteorologo Lewis Fry Richardson ottenne su basi empiriche, la legge di diffusione turbolenta che prende il suo nome. A partire da una serie di osservazioni in atmosfera, Richardson intuì che in casi macroscopici il coefficiente D_0 va rimpiazzato con un coefficiente di diffusione turbolento D_T. L'osservazione

sperimentale mostra che non solo D_T è molto maggiore di D_0, ma anche che il suo valore aumenta al crescere della dimensione della *macchia* di particelle nel tempo. Come conseguenza, la diffusione turbolenta è un processo esplosivo, per cui la nuvola di particelle si allarga tanto più velocemente quanto più è grande, generando una struttura ramificata frattale. Il quadrato della dimensione R della nuvola cresce secondo il cubo del tempo, cioè molto più velocemente che nel moto Browniano. La legge di Richardson ha una validità universale dalle più piccole scale di turbolenza (pochi millimetri) fino a scale di centinaia di chilometri in atmosfera.

Poiché la diffusione turbolenta è caratterizzata da velocità puntuali casuali, nei modelli si introduce il concetto di probabilità di trovare la particella in uno specifico volumetto, utilizzando la funzione di densità di probabilità (*Probability Density Function, PDF*).

I processi di dispersione nell'atmosfera sono quindi intrinsecamente complessi e di natura difficilmente determinabile, infatti la maggior parte dei modelli sviluppati si basano su un approccio di tipo *stocastico* utilizzando una distribuzione di probabilità casuale uniforme o Gaussiana.

Capitolo 3

Calcolo scientifico con processori grafici

3.1 GPU

La *Graphics Processing Unit* (GPU) è il microprocessore di una scheda video per *personal computer*, *workstation* o *console* dedicato ai calcoli di *Computer Graphics* (CG) ed in particolare alle operazioni di *rendering*.

La Grafica Computazionale Tridimensionale, o *Computer Graphics 3D*, è una branca della CG che usa dati geometrici tridimensionali per la produzione di immagini bitmap[1]. Questo processo è suddiviso in due fasi: la prima, in cui vengono definiti gli oggetti, le prospettive e le illuminazioni; la seconda, in cui si produce l'immagine. Fra queste, la seconda fase, chiamata fase di *rendering*, è molto più onerosa in termini di prestazioni rispetto alla prima.

Nei personal computer, quando si tratta di produrre una singola immagine, entrambe le fasi possono essere assegnate alla CPU. Invece, quando il

[1]Immagine bidimensionale definita da una matrice (mappa) di bit.

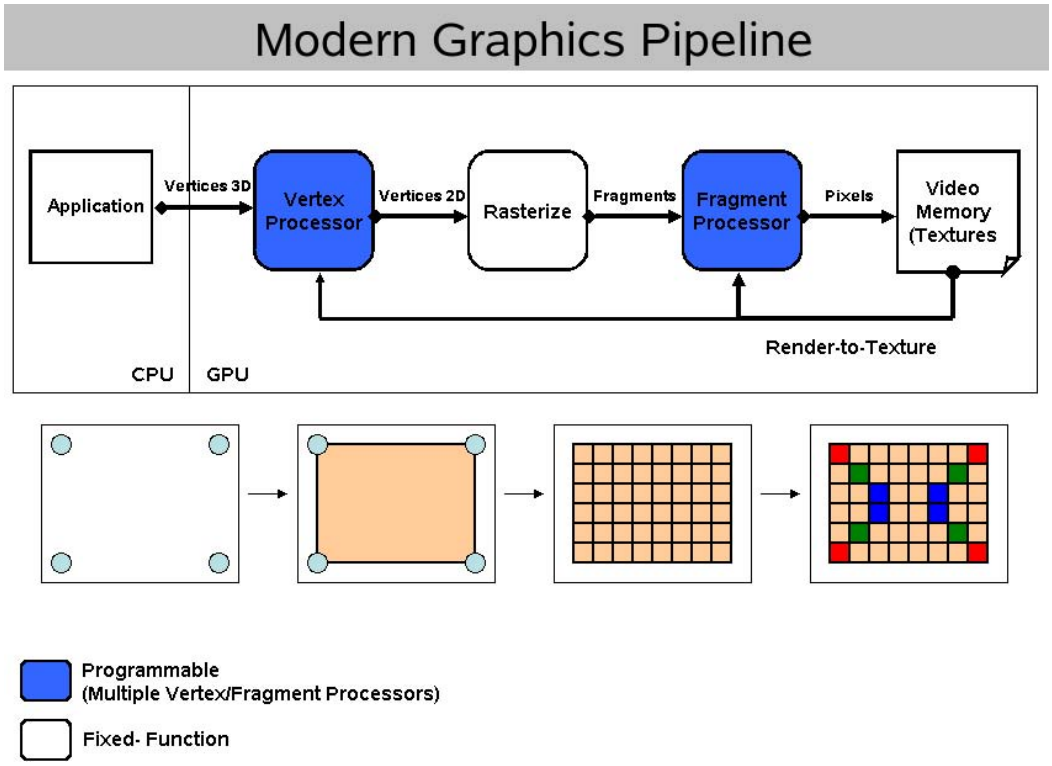

Figura 3.1: Pipeline di *rendering* di una scena tridimensionale.

compito è quello di produrre dalle 30 alle 60 immagini al secondo, la capacità di calcolo della CPU non è più sufficiente. Tali compiti sono soprattutto richiesti dai giochi per computer, i quali delegano il rendering alle schede video. Negli anni, grazie allo sviluppo dell'industria videoludica, le GPU di queste schede si sono evolute fino a diventare dei potenti calcolatori paralleli, sono state dotate di una propria memoria e hanno acquisito la capacità di svolgere velocemente le complesse operazioni del rendering.

L'intera operazione di rendering comprende diverse fasi, che, raggruppate insieme, formano la pipeline grafica (Figura 3.1).

1. Si inizia con il trasferimento della descrizione di una scena, dalla memoria centrale della CPU, alla memoria della scheda video. La descrizione

di una scena comprende: l'insieme di vertici che definiscono gli oggetti, le texture[2] che vi verranno applicate, i dati sull'illuminazione, ed il punto di osservazione della scena.

2. La seconda fase è quella in cui vengono eseguite le trasformazioni dei vertici. Le rotazioni, gli scaling e le traslazioni degli oggetti sono fondamentali per la definizione di una scena. Ad esempio, un *designer* può realizzare contemporaneamente sia il modello di un'automobile sia il modello di un tavolo, senza preoccuparsi del rapporto fra le loro dimensioni. In questa fase il primo oggetto sarà dimensionato in modo da essere più grande del secondo.

3. Alla precedente, segue un'altra fase di trasformazione, necessaria per l'aggiunta della prospettiva. In questa fase vengono anche eliminati dalla scena tutte le parti degli oggetti che, per via della *camera*[3], non compariranno nell'immagine finale.

4. La quarta fase è quella dedicata all'illuminazione degli oggetti. Nel rendering è una delle fasi più impegnative dal punto di vista computazionale. Per questo motivo, quando si parla di rendering in tempo reale, l'illuminazione di un oggetto, in base alle sorgenti di luce, viene calcolata soltanto ai vertici dei suoi poligoni. Durante le fasi successive, i valori dell'illuminazione ai vertici di un oggetto vengono usati per interpolare i valori di illuminazione della sua superficie.

5. Si passa quindi alla fase di *rasterization*, ovvero la trasformazione dei dati elaborati in un'immagine bitmap. Durante questa fase, le coordi-

[2]Una texture è l'immagine *bitmap* usata per rappresentare la superficie di un oggetto.
[3]*Camera* è il termine usato per indicare il punto di vista della scena.

nate tridimensionali vengono trasformate in coordinate bidimensionali e le texture vengono applicate sugli oggetti. Al termine vengono anche applicati eventuali effetti grafici come ombre, nebbia o antialiasing[4].

3.1.1 Evoluzione delle GPU

I primi *chip* grafici risalgono all'inizio degli anni '80 e le loro funzioni erano molto limitate; mancavano totalmente di funzioni per l'elaborazione di scene tridimensionali e avevano un insieme limitato di funzioni per il disegno di scene bidimensionali.

All'inizio degli anni '90 invece i chip grafici furono sostituiti con vere e proprie CPU, opportunamente modificate e programmate per fornire le funzioni di disegno richieste soprattutto da applicazioni di *Computer Aided Design* (CAD). In questi anni, anche molte stampanti della Apple hanno utilizzato processori di questo tipo che, a partire da un documento PostScript, producevano un'immagine *bitmap*.

Successivamente è stato possibile sviluppare dei chip grafici integrati, i quali fornivano tutte le funzioni di accelerazione richieste dalle applicazioni di disegno bidimensionale. Erano meno flessibili delle CPU programmabili, ma il costo inferiore, la maggiore semplicità di fabbricazione e l'avvento di $Microsoft^{®}$ $Windows^{TM}$ ne facilitarono la diffusione. Quest'ultimo *software*, infatti, stimolò significativamente l'interesse per la grafica bitmap e, pochi anni dopo, la $S3$ $Graphics^{©}$ introdusse sul mercato il primo chip di

[4]L'antialiasing è una tecnica per ridurre l'effetto di *aliasing* (scalettamento) che si ha quando un segnale a bassa risoluzione viene mostrato ad alta risoluzione. In questo caso l'antialiasing ammorbidisce le linee, smussando i bordi e omogeneizzando l'immagine finale.

accelerazione 2D; in breve tempo le più costose CPU grafiche furono ritirate dal mercato e uscirono dal commercio.

A metà degli anni '90 l'uso della grafica tridimensionale iniziò a diffondersi sia nel mercato delle *console* di gioco che in quello dei *personal computer*, spingendo i produttori ad integrare nei chip grafici diverse funzioni di accelerazione 3D. Una forte influenza all'implementazione di queste ultime è stata data dalla libreria grafica *Open Graphics Library* (OpenGL), già apparsa nei primi anni '90. Durante questo periodo, le implementazioni software dei metodi di OpenGL si semplificarono drasticamente grazie ad un supporto hardware da parte dei chip in continua crescita. Un'altra libreria che, più tardi negli anni, ha influenzato l'evoluzione delle GPU è stata la DirectX™ di Microsoft. L'approccio meno popolare del supporto *scheda-per-scheda*, inizialmente, ne rallentò lo sviluppo. Questo riprese vigore quando Microsoft iniziò a lavorare affianco ai progettisti hardware, programmando le nuove versioni di DirectX contemporaneamente con nuove schede video compatibili. La versione 7.0 delle librerie DirectX introdusse il primo supporto per l'accelerazione hardware alle trasformazioni ed alle illuminazioni delle scene. Fu uno dei primi passi a segnare la trasformazione della pipeline grafica all'interno della GPU, che avrebbe visto a breve l'introduzione dello shading.

Tradizionalmente uno *shader* è uno strumento della *computer grafica 3D* ed è generalmente utilizzato per determinare l'aspetto finale della superficie di un oggetto. Dal punto di vista software consiste essenzialmente in un insieme di istruzioni, ossia di un programma sottoposto ad alcune restrizioni secondo il paradigma dello *stream processing*. Le librerie OpenGL e DirectX utilizzano tre tipologie di shader, che sfruttano le capacità di shading delle

GPU:

- *Vertex shader*

  è adibito alla gestione della posizione dei vertici di un oggetto, può pertanto alterarne la forma,

- *Geometry shader*

  viene utilizzato per combinare una serie di vertici in modo da formare un oggetto più complesso al quale poi applicare effetti di pixel shading,

- *Pixel shader*

  elabora i singoli pixel di un oggetto, per applicare texture o effetti di *bump mapping* o nebbia.

NVIDIA è stata la prima azienda a produrre un chip con capacità di shading programmabili e successivamente $ATI^{®}$[5] introdusse il supporto a shader contenenti istruzioni di ciclo e di calcolo in virgola mobile.

In breve tempo le GPU sono divenute flessibili come le CPU completamente programmabili ed estremamente "performanti" nell'esecuzione di operazioni con molti dati. Rispetto alla prima legge di Moore, secondo la quale le prestazioni dei processori raddoppiano ogni 18 mesi, le GPU delle schede video hanno dimostrato un raddoppio di prestazioni ogni 6 mesi. Ovvero la prima legge di Moore elevata al cubo. Infatti come si può notare anche dal grafico in Fig. 3.2, le GPU pur operando a frequenze più basse delle CPU, riescono ad eseguire le operazioni più velocemente. Ciò è dovuto alla diversa natura dei due processori. Il compito della CPU è quello di leggere i dati e le

[5]ATI è stata per molti anni la principale società concorrente di NVIDIA. Nel 2006 è stata acquistata dalla $AMD^{®}$, azienda nota principalmente per la produzione di CPU.

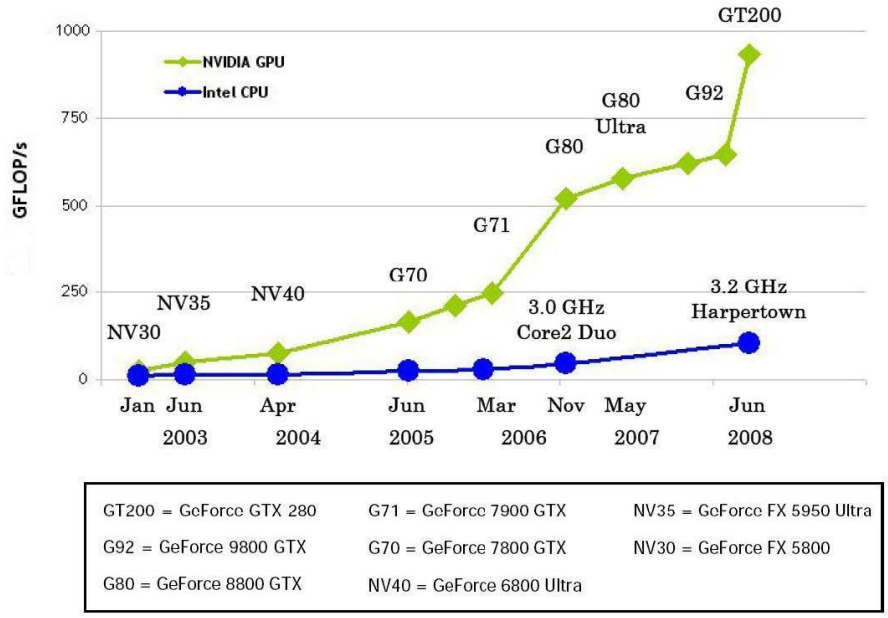

Figura 3.2: Velocità di calcolo di CPU e GPU a confronto

istruzioni dalla memoria per eseguirle, portando a termine un *qualsiasi* task nel minor tempo possibile. La GPU invece nasce come dispositivo *dedicato* alla manipolazione e alla visualizzazione di immagini di *computer graphics*, e deve quindi eseguire la stessa istruzione per ogni elemento dell'immagine, lavorando quindi su una grande mole di dati. Per questo motivo, le GPU sono dotate di hardware apposito che permette un accesso estremamente veloce ai dati (Fig. 3.3). Nell'esecuzione delle istruzioni, la GPU elabora i singoli pixel dell'immagine, quindi non necessita di molta memoria cache ed ha a disposizione più transistors dedicati all'elaborazione dei dati piuttosto che alla loro memorizzazione (Fig. 3.4).

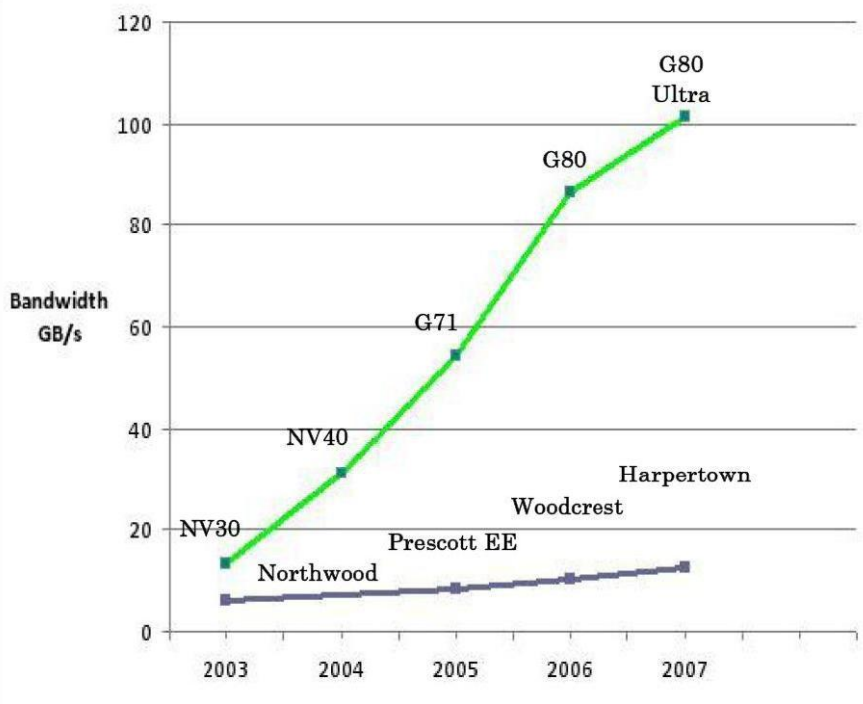

Figura 3.3: Velocità di accesso alla memoria di CPU e GPU a confronto

Figura 3.4: Architettura di una CPU e una GPU a confronto

3.2 Architettura GPU

Nonostante le GPU debbano svolgere gli stessi compiti, le scelte architetturali delle principali aziende del settore grafico rimangono profondamente diverse. Per confrontarne le caratteristiche principali si illustra la tassonomia di Flynn, la quale classifica i sistemi di calcolo in base al numero dei flussi di istruzioni e dei dati che sono in grado di gestire. Le categorie principali sono rappresentate in figura 3.5 e si distinguono in:

- SISD *Single Instruction on Single Data*.
  É la categoria contenente l'architettura tradizionale della macchina di *Von Neumann*(Fig. 3.6). Non implementa nessun tipo di parallelismo, ed è utilizzata da tutti i calcolatori convenzionali, in cui un solo processore opera su un singolo flusso di istruzioni (programma sequenziale) ed esegue queste istruzioni ogni volta su un singolo flusso di dati.

- SIMD *Single Instruction on Multiple Data*.
  Alla categoria SIMD appartengono le architetture composte da molte unità di elaborazione che eseguono contemporaneamente la stessa istruzione su un insieme di dati differenti, realizzando un parallelismo di tipo spaziale. Generalmente il modo di implementare queste architetture consiste nell'avere un'unica unità di controllo (CU) che invia le istruzioni in parallelo ad un insieme di unità di elaborazione, chiamate ALU *(Aritmetic Logic Unit)*, le quali provvedono ad eseguirle. Una di queste implementazioni è il processore vettoriale. I processori vettoriali sono unità di calcolo che, dopo aver letto e decodificato un'istruzione, la eseguono su più dati prima di passare all'istruzione successiva (Figura

3.7). L'unità di esecuzione lavora su registri vettoriali[6], il cui contenuto è gestito dall'unità vettoriale di *load-store*. Questa è l'unità che si occupa di leggere e scrivere dalla memoria centrale ai registri vettoriali. É in grado di operare su più dati contemporaneamente e, in un processore vettoriale, ve ne possono essere molteplici. A causa del funzionamento di quest'ultima componente, i processori vettoriali hanno lo svantaggio di essere poco adatti all'elaborazione di dati non distribuiti in modo costante. Quindi le loro reali prestazioni variano a seconda della tipologia del programma che si vuole eseguire.

- MISD *Multiple Instruction on Single Data*.

  É una classe di architetture in cui diverse unità effettuano diverse elaborazioni sugli stessi dati. Non è mai stata sviluppata sul piano commerciale ma solo in alcuni progetti di ricerca per l'elaborazione di segnali.

- MIMD *Multiple Instruction on Multiple Data*.

  É la categoria che comprende le architetture in grado di eseguire più flussi di istruzioni su più flussi di dati contemporaneamente, implementando un parallelismo di tipo asincrono, rappresenta una evoluzione della categoria SISD. Infatti la realizzazione di queste architetture avviene attraverso l'interconnessione di un numero elevato di elaboratori convenzionali, facendo sì che a questa classe appartengano sistemi di calcolo multiprocessore e sistemi di calcolo distribuiti.

[6] All'interno di un processore, i registri sono piccole quantità di memoria nelle quali vengono trasferiti i dati, dalla memoria principale, prima di operarvi. Un registro vettoriale è un insieme di registri sui quali si opera contemporaneamente con una singola istruzione.

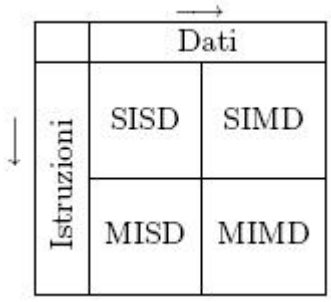

Figura 3.5: Tassonomia di Flynn.

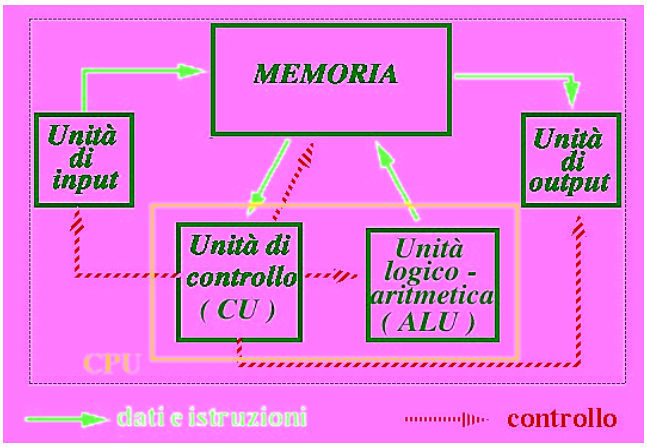

Figura 3.6: Schema funzionale della *macchina di Von Neumann*.

In particolare, le categorie SIMD e MIMD, costituiscono la classe delle architetture parallele. Di queste fanno parte le GPU, le quali, come già detto, si sono specializzate nell'esecuzione concorrente dello stesso flusso di istruzioni su molti dati.

A livello implementativo, sia l'architettura di AMD che quella di NVIDIA appartengono alla classe dei sistemi SIMD, mentre a livello funzionale NVIDIA ha scelto un approccio differente. Infatti se nella propria GPU, AMD usa dei processori vettoriali, NVIDIA adotta degli *Stream Multiprocessor* (SM). I

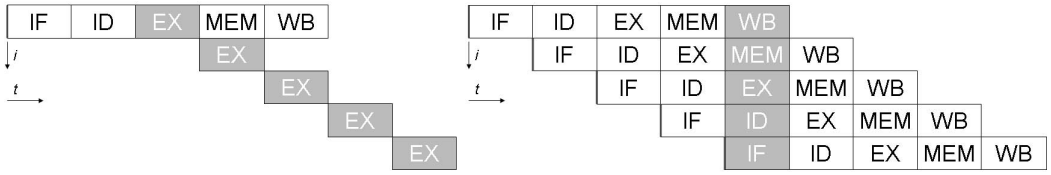

Figura 3.7: *Pipeline* di un processore vettoriale a confronto con la *pipeline* di un processore scalare.

dettagli degli SM e dell'architettura di NVIDIA saranno mostrati nei prossimi paragrafi, mentre per adesso è sufficiente sapere che questi non sono altro che processori multicore, ovvero processori contenenti più unità di elaborazione le quali sono in grado di lavorare in parallelo e comunicare tra di loro.

La scelta di tale soluzione è stata motivata da NVIDIA attraverso lo studio di centinaia di shader appartenenti a programmi di grafica e giochi per i computer. Gli shader, negli ultimi anni, hanno man mano adoperato un numero sempre maggiore di operazioni scalari, a scapito delle operazioni vettoriali. Soprattutto con gli shader più complessi, gli SM hanno dimostrato di essere più efficienti rispetto ai processori vettoriali nell'eseguire questo tipo di operazioni.

Nei paragrafi successivi si studia la soluzione per il GPU computing proposta da NVIDIA. Le ragioni di questa decisione dipendono in parte dall'hardware che si è avuto a disposizione per lo svolgimento di questo studio, ed in parte dal grande lavoro che NVIDIA ha svolto riguardo alla documentazione della propria piattaforma. Inoltre, quest'ultima, grazie all'uso degli SM, prometteva una maggiore flessibilità rispetto alla soluzione concorrente offerta da AMD.

3.2.1 Architettura TESLA

Tesla, schematizzata in Figura 3.8 , è il nome dell'architettura che sta alla base delle GPU della serie G80 o della più recente GT200. Le sue componenti principali consistono in una memoria *Dynamic Random Access Memory* (DRAM) e in uno *Scalable Processor Array* (SPA), ovvero la componente che si occupa di eseguire tutte le operazioni programmabili sulla GPU. Nello schema il lavoro fluisce dall'alto verso il basso, attraversando tutte le componenti, alcune delle quali operano esclusivamente nel contesto del *rendering* e rimangono quindi inutilizzate per le operazioni di *computing*. Di queste componenti si citeranno soltanto le più importanti, e ci si soffermerà maggiormente su quelle che hanno un ruolo determinante nell'uso della GPU a fini computazionali.

L'*Host Interface* è la componente che si occupa della comunicazione tra *Host* e *Device*, ovvero tra il *personal computer* e la periferica fisicamente connessa a questo, sulla quale risiede la GPU. Fra i suoi compiti principali c'è quello di iniziare i trasferimenti dei dati da e verso la memoria della CPU, l'interpretazione dei comandi dell'Host ed il controllo della loro consistenza. Successivamente il *Compute work distribution* si occupa della distribuzione sullo SPA del flusso di istruzioni generato dall'esecuzione dei kernel, ovvero le funzioni che vengono eseguite sulla GPU. Questo compito è analogo a quello dei *Pixel* e *Vertex work distribution* i quali, invece, distribuiscono il lavoro di *shading* nelle fasi del *rendering* grafico. Al termine dell'elaborazione, nel caso che fossero state eseguite operazioni di *rendering*, i risultati dei calcoli passano ai *Raster Operation Processor* (ROP) attraverso una rete di connessioni. I ROP hanno il compito di eseguire le funzioni non programmabili

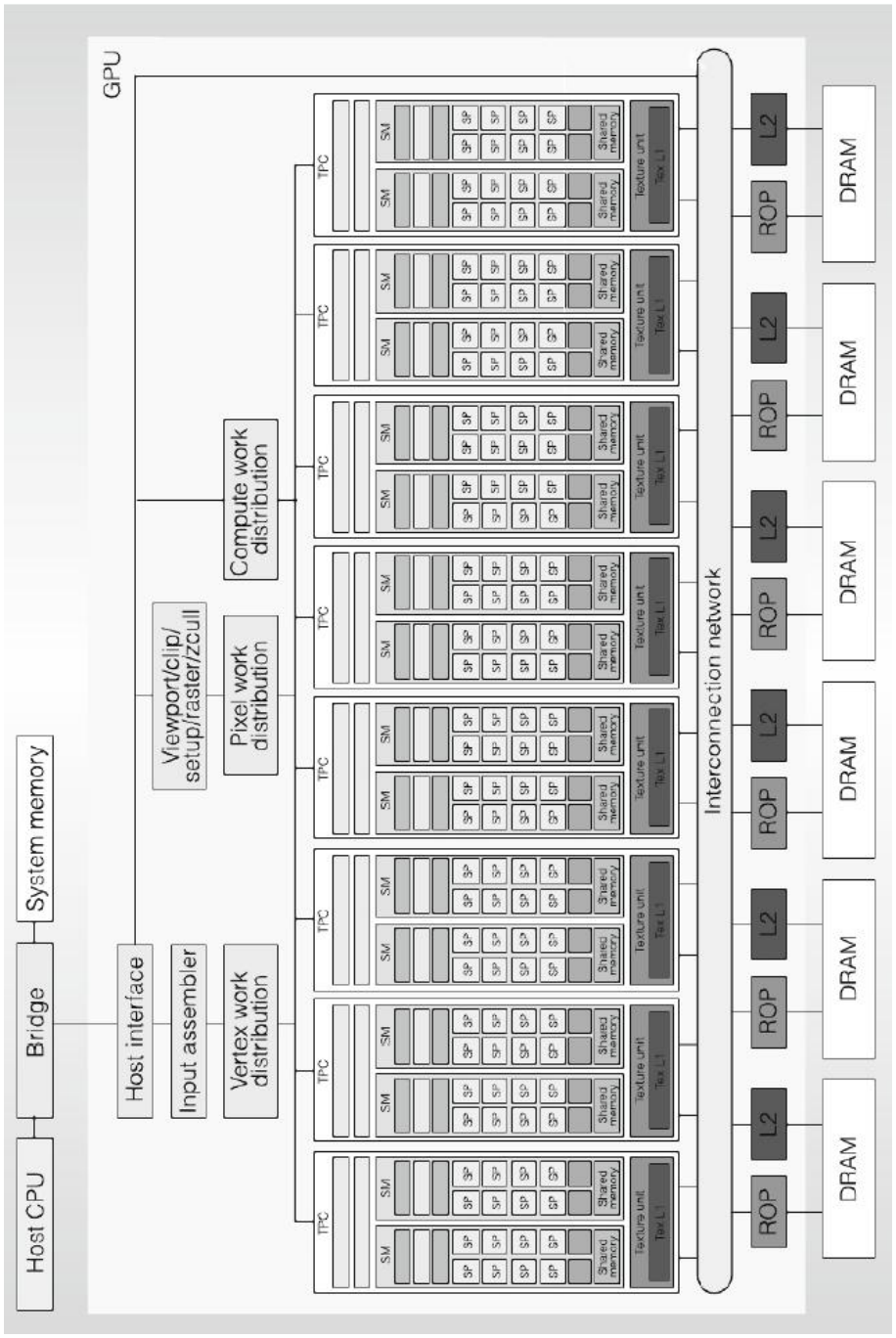

Figura 3.8: Architettura delle GPU serie G80 di NVIDIA

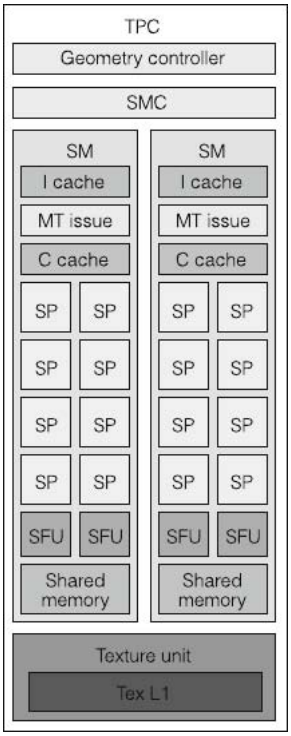

Figura 3.9: *Texture/Processor Cluster* di una GPU G80

di colorazione dei *pixel*, ed operano direttamente sulla memoria DRAM alla quale sono connessi. Più nel dettaglio le componenti fondamentali possono essere riassunte in:

- *Scalable Processor Array*

  Lo SPA è composto da diversi *Texture Processor Cluster* (TPC) (Figura 3.9) che, in base al livello di performance, possono scalare da uno, nelle GPU di fascia bassa, fino ad otto o più nelle GPU di fascia alta. Un TPC contiene un *controller* di geometria, uno *Streaming Multiprocessor Controller* (SMC), due *Stream Multiprocessor* ed un'unità di *texture.*

- *Stream Multiprocessor*

  Lo SM (Figura 3.10) è un processore in grado di eseguire istruzioni

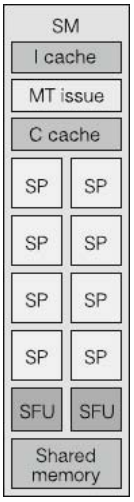

Figura 3.10: *Stream Multiprocessor*

inerenti sia allo shading sia al GPU computing. Ogni SM contiene 8 *Streaming Processor* (SP) *cores*, 2 S*pecial Function Unit* (SFU), una *MultiThreaded instruction fetch and Issue unit* (MT Issue), una *cache* delle istruzioni, una *cache* per la memoria costante, 16KByte di memoria condivisa ed 8192 registri da 32 bit ciascuno. Ogni SP contiene un'unità scalare Multiply-Add (MAD), dando quindi 8 unità MAD ad ogni SM. Le due SFU sono usate per il calcolo delle funzioni elaborate (esponenziali, logaritmiche, trigonometriche) ed inoltre ogni SFU contiene 4 unità di moltiplicazione per numeri in virgola mobile. Le operazioni di load e store dalla memoria principale sono implementante dall'SMC, mentre le operazioni di accesso alla memoria condivisa, residente sullo SM, sono effettuate direttamente.

- *Unità delle texture*

  Nel contesto del GPU computing, l'unità delle *texture* è una componente che permette di accedere in sola lettura della memoria DRAM.

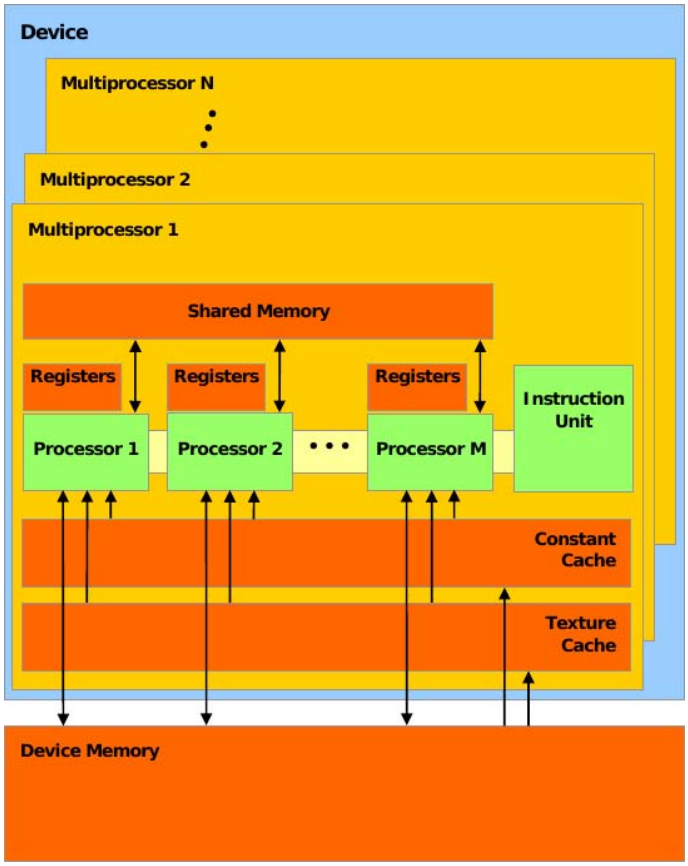

Figura 3.11: Schema degli accessi alla memoria

Questa è dotata di un meccanismo di *caching*[7] tale da privilegiare gli accessi localizzati ai dati. All'interno di un TPC, ogni unità serve contemporaneamente due SM, fornendo un'alternativa all'uso dello SMC per la lettura dei dati.

La mancanza di un sistema di *caching* generale, della memoria DRAM, favorisce l'ampiezza di banda passante dei dati tra GPU e memoria. Sul chip G80 questa supera gli 80 GByte/s contro gli 8-10 GByte/s della banda

[7]Un meccanismo di caching è un meccanismo che si appoggia generalmente ad una memoria piccola e veloce (cache) per ridurre i tempi di accesso alla memoria globale.

passante fra una CPU e la sua memoria globale. La mancanza di una cache però comporta un'alta latenza per le operazioni sui dati in memoria, superiore di 400-600 volte al tempo di accesso ai registri locali. Tuttavia grazie ad uno *scheduling* intelligente delle operazioni, la latenza di accesso alla DRAM viene nascosta quando si ha un numero di *thread*[8] per SM sufficientemente alto.

Infatti, quando il numero di *thread* è abbastanza grande, lo *scheduler* gestisce le operazioni da eseguire, in modo da ridurre o eliminare i tempi di inattività degli SM. Durante il tempo necessario affinché un'operazione sulla memoria venga eseguita, lo *scheduler* mette in attesa il *thread* che ha richiesto tale operazione. Si liberano così delle risorse, che vengono utilizzate per l'esecuzione del flusso di istruzioni di un altro *thread*, rimasto inattivo fino a quel momento.

I tempi di accesso alla memoria condivisa, al contrario di quanto appena visto per la memoria globale, sono di poco superiori a quelli dei registri. La bassa latenza è dovuta al fatto che i 16Kbyte di memoria condivisa sono situati all'interno di ogni SM e non esternamente al *die* (superficie) della GPU. Tuttavia questi 16Kbyte di memoria condivisa sono partizionati in 16 banchi da 1Kbyte, ciascuno soggetto a conflitti di accesso da parte degli SP *cores* componenti l'SM. Infatti nel caso ci siano più SP *cores* ad accedere allo stesso banco, le operazioni di ciascuno di questi vengono serializzate, con un conseguente degrado delle prestazioni dovuto all'aumento della latenza.

[8]Un thread è un flusso di esecuzione all'interno di un'applicazione. Quando il flusso è unico si parla di applicazione singlethread, altrimenti, quando i flussi sono più di uno, si parla di applicazioni multithread.

3.2.2 Architettura SIMT

Mentre al livello implementativo NVIDIA usa delle unità SIMD composte da 8 *cores* ciascuno, al livello funzionale queste non operano eseguendo 8 volte la stessa istruzione, per un solo flusso di esecuzione, come farebbero delle unità SIMD tradizionali. Esse, infatti, gestiscono 8 flussi di esecuzione differenti, eseguendo un'istruzione per ciascuno di essi. In questo modo, i singoli SP *cores* si comportano come delle unità scalari e non come unità di esecuzione di un processore vettoriale. Per sottolineare questa differenza, NVIDIA parla di un'architettura *SIMT*, ovvero *Single Instruction Multiple Threads*. A differenza delle unità SIMD, questa è in grado di massimizzare l'uso delle proprie unità di calcolo, soprattutto quando l'applicazione è composta da molti *thread*. La GPU infatti viene utilizzata appieno solo quando i *thread* sono nell'ordine delle migliaia, grazie al fatto che questi hanno un costo di creazione e gestione praticamente nullo.

L'architettura lavora assegnando ad ogni flusso di istruzioni un SP core, in modo che ogni *thread* venga eseguito indipendentemente dagli altri, conservando il proprio stato dei registri e indirizzo delle istruzioni. Ogni SM gestisce ed esegue simultaneamente fino a 24 gruppi, chiamati *Warps*, di 32 *thread* ciascuno, per i quali la *MT Issue* si occupa di selezionare e inoltrare le istruzioni. Durante l'esecuzione di un programma è possibile che, in presenza di un'istruzione condizionale (`if-else`, `switch-case`, ecc ...), due o più *thread*, all'interno dello stesso *Warp*, seguano percorsi differenti. Quando tutti i 32 *thread* di un *Warp* seguono lo stesso ramo di esecuzione, questa architettura raggiunge il massimo dell'efficienza per il fatto che tutti gli SP *cores* di un SM eseguono la medesima istruzione. Invece, se i *thread*

all'interno di un *Warp* divergono, le prestazioni decadono. Infatti, quando si presenta quest'ultimo caso, la MT Issue esegue in serie le istruzioni per i diversi flussi, fintanto che questi rimangono separati. Scrivere un software che tenga conto di questa caratteristica è fondamentale se si vuole ottenere il massimo delle prestazioni dalla GPU di NVIDIA.

Nonostante la maggiore efficienza, NVIDIA non vede in un'architettura SIMT la soluzione definitiva per le proprie GPU, poiché essa introduce diversi problemi rispetto ad architetture più semplici. Facendo un confronto con la classe dei sistemi SIMD è infatti richiesta una maggiore logica di controllo, che comporta un aumento della superficie del chip e un maggior consumo di energia. É sufficiente osservare i dettagli tecnici delle architetture, prima di AMD, e poi di NVIDIA, per vedere l'enorme differenza nella superficie dei chip che, nel primo caso, si limita a $192mm^2$ contro i $484mm^2$ del secondo.

L'architettura Tesla contiene un vasto set di istruzioni per operazioni su numeri interi, numeri in virgola mobile, singoli bit, conversioni di tipo, funzioni matematiche, controllo di flusso, lettura/scrittura dalla memoria e operazioni sulle texture.

Di maggiore interesse per il calcolo scientifico sono le istruzioni che operano sui numeri in virgola mobile, che seguono lo standard di rappresentazione *IEEE754*. Nell'elenco seguente si mostrano le differenze principali tra l'implementazione interna di Tesla e le specifiche ufficiali dello standard. La lista completa di queste differenze si può consultare nella documentazione della NVIDIA.

- Moltiplicazione e addizione possono essere combinate in una sola istruzione (FMAD) che non tiene conto del risultato intermedio della moltipli-

cazione.

- La divisione è implementata attraverso moltiplicazione per il reciproco.
- L'operazione di radice quadrata è implementata attraverso il reciproco della radice quadrata stessa.
- La politica di recovery adottata per le condizioni di errore è la store 0. Nel caso un'operazione incorra in un errore di *underflow*, questa tecnica ne azzera il risultato. Al contrario del *gradual underflow*, adottato nella maggior parte delle CPU per personal computer, si mantiene così invariata la precisione macchina.

Queste ed altre piccole differenze possono portare ad una discordanza con i risultati dei calcoli eseguiti da una CPU. Tuttavia gli errori di moltiplicazione e addizione non superano mai 0,5 *ulp* (*Unit in the Last Place* o precisione macchina), ed errori maggiori si possono verificare solo nel calcolo di funzioni particolarmente elaborate, i quali però restano dell'ordine di 2-3 *ulp*.

3.3 Il GPU computing

L'idea di sfruttare la capacità di calcolo parallelo delle GPU, per compiti differenti dal rendering, ha portato alla nascita del *GPU computing*. Già negli anni novanta si utilizzavano gli acceleratori grafici per effettuare calcoli matematici anche se limitatamente ad alcune funzionalità hardware come la rasterizzazione o i calcoli Z-buffers.

Il termine GPGPU, *General Purpose computing using GPU*, venne coniato per la prima volta con l'arrivo di GPU che supportavano gli shader,

grazie alle quali era possibile effettuare calcoli matriciali convertendo i dati da elaborare in *texture*. Le unità di shading, in particolar modo quelle dedicate ai vertici, sono infatti costruite per l'esecuzione in parallelo di semplici applicazioni (*gli shader*), su una grande quantità di dati. Diversi progetti, utilizzano le *Application Programming Interface* (API) di disegno 3D come strumenti per la programmazione delle GPU. Attraverso queste API è possibile definire uno shader su insiemi di vertici e texture, in modo tale che questo corrisponda alla definizione di un'applicazione parallela su un insieme di dati. In questo modo la GPU, eseguendo lo shader, è in grado di assolvere a compiti general purpose; ovvero non strettamente legati ad applicazioni di grafica computazionale.

Con l'ultima generazione di GPU le cose sono ulteriormente migliorate. I diversi tipi di shader, che in passato venivano eseguiti su unità diverse e separate fra di loro, adesso vengono eseguiti su una sola unità di calcolo. Questa unità è capace di elaborare l'*Unified Shader Model*, ovvero un modello di shading che utilizza lo stesso insieme di istruzioni per la definizione di *vertex*, *geometry* o *pixel* shader. Questa soluzione presenta molti vantaggi per le applicazioni di rendering, a cominciare dalla distribuzione del carico di lavoro. Infatti, se nella generazione precedente di GPU si poteva presentare il caso in cui un tipo di unità di shading era sovraccaricata rispetto alle altre, adesso diviene possibile allocare dinamicamente più risorse al compito più impegnativo.

Oltre ai vantaggi per le applicazioni di rendering, grazie a questo radicale cambiamento è stato possibile sviluppare piattaforme di GPU computing come lo *Stream SDK* di AMD, *Larrabee* di Intel (ancora in fase di

realizzazione) e *CUDA* di NVIDIA, le quali non sono altro che ambienti di sviluppo per la programmazione della GPU. Questi hanno la caratteristica di non dover più ricorrere all'uso delle API di disegno 3D per elaborare i dati attraverso la scheda video. Si servono, infatti, di API appositamente sviluppate per la programmazione di applicazioni general purpose, ovvero di uso generico.

3.4 CUDA

CUDA è un modello di programmazione parallela ed un'architettura *hardware* e *software* progettata per sfruttare le capacità dei nuovi sistemi paralleli *multicore* e, più in particolare, delle GPU ad architettura Tesla.

Gli elementi principali sui quali questa piattaforma si basa sono:

- modello di esecuzione basato su Thread,
- meccanismi di condivisione della memoria,
- meccanismi di sincronizzazione.

Questi vengono controllati dal programmatore mediante alcune estensioni al linguaggio di programmazione C, facilitando così l'apprendimento per chi ha già familiarità con questo ambiente ed aprendo le porte del GPU computing ad una vasta cerchia di programmatori.

Le estensioni indirizzano lo sviluppatore a partizionare un problema in più sottoproblemi risolvibili in maniera indipendente. Tali problemi, a loro volta, vengono risolti mediante la cooperazione di più thread, ovvero, di più flussi d'esecuzione concorrenti dello stesso programma. Un insieme lineare, *2D* o

3D di *thread*, fino ad un massimo di 512, costituisce un *blocco* che viene assegnato durante l'esecuzione ad un unico multiprocessore. Ogni *blocco* possiede una memoria condivisa dai *thread* fino al termine della loro esecuzione e che può essere utilizzata per operazioni di comunicazione o sincronizzazione fra i *thread* dello stesso blocco. Se un blocco è costituito da un numero di *thread* maggiore del numero di processori, i thread verranno eseguiti in *time-slice*, con un conseguente calo delle prestazioni. Infine è possibile mappare un insieme di *blocchi* in una *griglia* monodimensionale, bidimensionale, o tridimensionale, che ha lo scopo di far eseguire a tutti i *thread* dei *blocchi* una stessa porzione di codice (*kernel*), rendendo in questo modo più semplice la scomposizione dei dati e del problema. Questa soluzione permette sia la cooperazione fra gruppi di thread per la risoluzione di ogni sottoproblema, sia l'assegnazione di un sottoproblema indipendente ad uno qualunque dei processori disponibili. La piattaforma è quindi scalabile sia rispetto al numero di SM presenti nel sistema, sia al numero di *thread* che questi possono gestire.

Dal punto di vista software, CUDA consiste in un ambiente di programmazione derivato dal C, e mette a disposizione librerie numeriche standard per *FFT* (*F*ast *F*ourier *T*ransform) e *BLAS* (*B*asic *L*inear *A*lgebra *S*ubroutines). Tra le estensioni al linguaggio C sono fondamentali i qualificatori da applicare alle funzioni:

- `__global__`

  quando applicato ad una funzione indica che quest'ultima è un *kernel*, una funzione che sarà chiamata dalla CPU ed eseguita dalla GPU.

- `__device__`

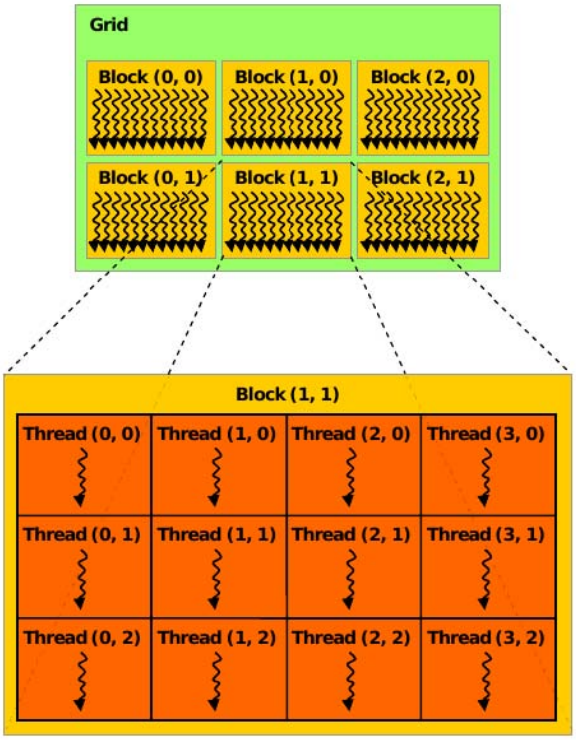

Figura 3.12: Gerarchia dei *threads* in CUDA

designa una funzione che sarà eseguita dalla GPU (*device*), ma che può essere richiamata solo dalla GPU, quindi da un *kernel* o da un'altra funzione __device__

- `__host__`

  opzionale, designa una funziona invocata dalla CPU ed eseguita dalla CPU stessa, quindi si tratta di una tradizionale funzione.

Per invocare un kernel bisogna utilizzare una sintassi del tipo:

```
Kernel_Name<<<n_blocks, n_threads>>>(Parameters);
```

che configura la chiamata al *kernel* utilizzando una griglia costituita da un numero di blocchi pari a *n_blocks* ognuno con *n_threads* thread, come rappresentato in Figura 3.12.

Ogni thread possiede una variabile `threadIdx`, accessibile all'interno del *kernel* ed inizializzata univocamente rispetto alla stessa variabile degli *n_threads* thread dello stesso blocco. `threadIdx` è una variabile avente 3 componenti intere, così che i thread possano essere organizzati in spazi monodimensionali, bidimensionali o tridimensionali. Questo permette al programmatore di scegliere la struttura di esecuzione del kernel più conveniente, in base al problema da risolvere.

Il codice di esempio 3.1 mostra come si effettua la somma tra due vettori A e B, di lunghezza M, ponendo il risultato in un terzo vettore C. Ogni thread somma una componente di A con la rispettiva componente di B e ne scrive il risultato in C. Essendo i thread organizzati in uno spazio monodimensionale, la componente dei vettori sulla quale opera ognuno di questi è data dal primo

(x) dei tre valori di *threadIdx*. Durante l'esecuzione, per ogni thread, questo valore sarà distinto e compreso tra 0 e M-1.

Listing 3.1: Somma di due vettori con un solo *blocco* di *threads*

```
__global__ void vecAdd(float* A, float* B, float* C){
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}
int main(){
    vecAdd<<<1, M>>>(A, B, C);
}
```

Nell'esempio sono dichiarate due funzioni. All'interno della funzione main, il punto di entrata, si invoca il *kernel* vecAdd, il quale riceve in input 3 riferimenti a vettori. Lo specificatore __global__ e ed i parametri contenuti tra i caratteri <<< e >>>, indicano rispettivamente al compilatore che la funzione vecAdd verrà eseguita sulla GPU in un solo blocco composto da M thread concorrenti.

Analogamente a quanto detto per `threadIdx`, si ha che ai thread appartenenti allo stesso blocco è assegnata la variabile `blockIdx`, accessibile all'interno del kernel e contenente il numero di blocco a cui appartiene il thread stesso. Questa variabile è composta da una o due componenti, a seconda che i blocchi siano organizzati logicamente in uno spazio monodimensionale o bidimensionale (vedi Fig. 3.12).

Il codice di esempio 3.2 estende il codice 3.1, introducendo l'uso di più blocchi organizzati in uno spazio monodimensionale. In questo esempio la dimensione dei vettori A e B è di M x N elementi, ed i parametri contenuti

tra i caratteri <<< e >>> indicano al compilatore che verranno eseguiti N blocchi composti da M thread ciascuno. Al fine di assegnare a tutti i thread la somma di una componente univoca dei due vettori, l'assegnazione di tale componente non dipenderà più solo dalla variabile threadIdx, ma dipenderà anche dalla variabile blockIdx. In questo modo gli M thread appartenenti agli N blocchi differenti, sono in grado di sommare distintamente tutti gli elementi dei vettori A e B.

Listing 3.2: Somma di due vettori multiblock

```
__global__ void vecAdd(float* A, float* B, float* C){
  int i = blockDim.x*blockIdx.x+threadIdx.x;
  C[i] =A[i] + B[i];
}
int main(){
  vecAdd<<<N, M>>>(A, B, C);
}
```

Il numero dei blocchi che verranno eseguiti sulla GPU, ed il numero dei thread per ogni blocco, determinano la configurazione di esecuzione di un kernel. All'interno di un kernel, è possibile conoscere tale configurazione accedendo alle variabili `blockDim` e `gridDim`: la prima contenente il numero di thread per blocco e la seconda contenente il numero di blocchi per il kernel in esecuzione.

L'architettura Tesla della GPU G80 limita il numero di thread per blocco a 512, e prevede al massimo la gestione di 65536 blocchi contemporaneamente. Lo scheduling dei blocchi di un kernel è gestito via hardware, sulla base di un algoritmo che di volta in volta assegna ogni blocco ad un solo SM,

e ad ogni SM assegna contemporaneamente fino ad 8 blocchi, la cui dimensione complessiva non supera però i 768 thread (24 warps). Questi blocchi vengono eseguiti in concorrenza fino a che lo scheduler decide di sospenderne l'esecuzione e assegnare allo SM dei nuovi blocchi. Per calcolare l'efficienza di una configurazione si introduce il concetto di occupazione, il quale è espresso come il rapporto tra il numero di warps attivi e il numero massimo di warps gestibili da un SM,

$$occupacy = \frac{active\ warps}{max\ active\ warps}.$$

Configurazioni che offrono l'occupazione massima possono essere facilmente costruite. Le più efficienti per le GPU della serie G80 sono quelle che hanno almeno 192 thread per blocco. In questo modo lo scheduler è in grado di mascherare buona parte della latenza di lettura e scrittura, sia della memoria, sia dei registri. Configurazioni con molti thread per blocco, però, introducono delle limitazioni sul numero dei registri utilizzabili all'interno di un kernel, ad esempio: se si decide per una configurazione di 256 thread per blocco, scrivere un kernel che usa 10 registri invece di 11 permette di passare da 2 a 3 blocchi (da 16 a 24 warps) attivi contemporaneamente per ogni singolo SM. Infatti, considerando 3 blocchi in esecuzione per ogni SM, nel caso ogni thread usi 10 registri, i registri utilizzati in totale sono 3 x 256 x 10 = 7680; un valore entro il limite degli 8192 disponibili. Invece, considerando 3 blocchi per ogni SM, ma assumendo che ogni thread usi 11 registri, questa volta i registri utilizzati in totale sono 3 x 256 x 11 = 8448, ovvero troppi affinché 3 blocchi possano essere attivi contemporaneamente su un solo SM.

Ogni thread ha accesso, in lettura e scrittura, a più spazi di memoria durante la sua esecuzione:

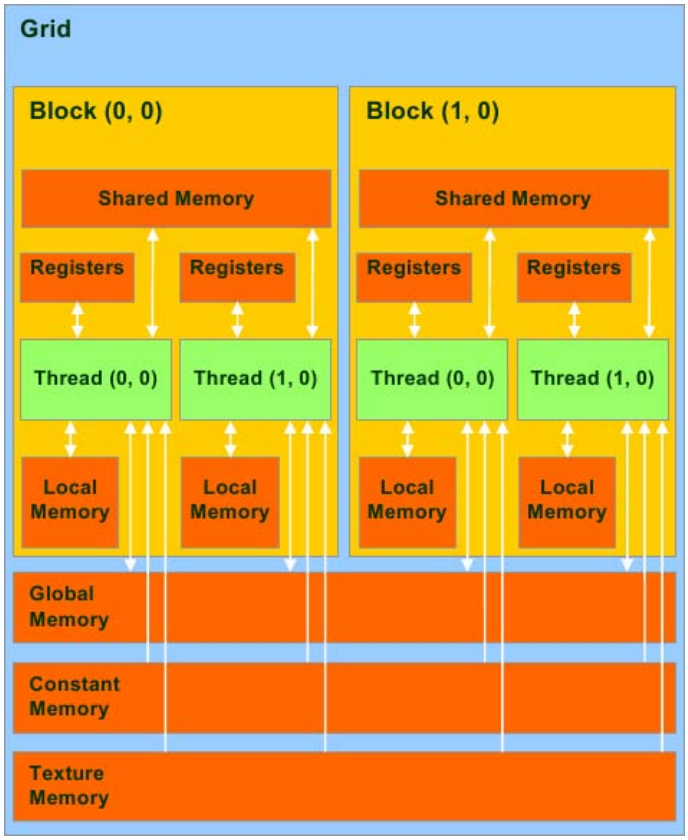

Figura 3.13: Gerarchia della memoria

- **Registri**

  Su ogni SM sono allocati un certo numero di registri e, ad ogni thread, ne è concesso solo un numero limitato. Le operazioni sui registri hanno latenza praticamente nulla, anche se possono avere dei ritardi di lettura dopo una scrittura dovuti all'aggiornamento del loro contenuto. Il numero di registri disponibili per ogni thread varia in base alla configurazione di esecuzione che si sceglie. Fissata quest'ultima, il loro numero può essere calcolato mediante la formula:

  $$\frac{R}{B \times \lceil T, 32 \rceil}$$

  dove R è il numero di registri per ogni SM, B è il numero di blocchi attivi per SM e T è il numero di thread per blocco, che, come si vede, viene approssimato per eccesso con il più piccolo multiplo di 32.

- **Memoria globale**

  Questa è la memoria che i thread utilizzano, sia per leggere i dati da elaborare, sia per scrivere i risultati delle loro operazioni. Essa è implementata nella DRAM, quindi i suoi tempi di accesso, come già visto durante lo studio dell'architettura Tesla, sono centinaia di volte superiori a quelli dei registri: è importante ridurne l'utilizzo il più possibile.

  Quando l'accesso alla memoria globale è inevitabile, è conveniente usare delle tecniche per leggere o scrivere i dati in blocchi (accessi *coalesced*). Nelle GPU della serie G80, un'operazione di lettura o scrittura coalesced si ha quando almeno la metà dei thread di un *warp* (*half-warp*) effettua un accesso alla memoria, in modo tale che il thread i-esimo acceda alla locazione i-esima di uno spazio allineato ad un indirizzo multiplo delle dimensioni di un half-warp, ovvero 16.

Operazioni di lettura e scrittura coalesced hanno tempi inferiori di circa 10 volte rispetto ad operazioni di lettura e scrittura non strutturate.

- **Memoria locale privata**

  In questa memoria vengono allocate le variabili locali dei singoli thread, che non possono essere contenute nei registri. Anche questo è uno spazio di memoria implementato nella DRAM e, di conseguenza, i tempi di accesso sono gli stessi della memoria globale.

- **Memoria condivisa**

  Questa memoria è accessibile a tutti i thread dello stesso blocco e, generalmente, è utilizzata per condividere i risultati intermedi dei processi di calcolo. É implementata mediante i banchi di memoria condivisa contenuti all'interno di ogni SM, quindi la latenza è leggermente maggiore di quella dei registri. Spazi contigui di memoria condivisa sono allocati su banchi differenti in modo da ridurre i conflitti di accesso e, di conseguenza, anche i tempi di latenza.

I seguenti spazi di memoria sono invece accessibili in sola lettura:

- **Memoria costante**

  É uno spazio di memoria implementato nella memoria globale, e contiene quei valori che restano costanti durante tutta l'esecuzione del kernel. I tempi di latenza sono al pari di quanto visto per la memoria globale; tuttavia questo spazio di memoria dispone di una cache. La sua presenza riduce drasticamente i tempi di attesa nel caso si acceda molte volte allo stesso elemento.

- **Memoria delle texture**

  Questo spazio di memoria è accessibile attraverso l'unità delle *texture*. Rappresenta quindi un'interfaccia di sola lettura alla memoria globale, che offre un meccanismo di *caching* ottimizzato per la località bidimensionale e, facoltativamente, un diverso tipo di indirizzamento ai dati. Nonostante le latenze siano superiori agli accessi di tipo *coalesced* verso la memoria globale, l'uso della memoria delle texture può essere molto vantaggioso. Tramite questo spazio di memoria infatti, si possono eseguire operazioni di lettura che, o non possono essere rese coalesced, o coinvolgono un insieme di dati sufficientemente piccolo da essere contenuto interamente nella cache.

Gli spazi di memoria *globale*, *costante* e delle *texture* sono consistenti tra la chiamata di un kernel e un altro. Al contrario, i contenuti della memoria *locale* e della memoria *condivisa* sono eliminati al termine di ogni esecuzione. Per allocare e deallocare memoria è necessario utilizzare le funzioni opportune:

- `CudaError_t cudaMalloc(void **devPtr, size_t count);`

  per allocare un'area di memoria di *count* byte.

- `CudaError_t cudaFree(void* devPtr);`

  per liberare l'area di memoria puntata da *devPtr*.

Entrambe restituiscono 0 in caso di successo o un errore di tipo `CudaError_t` altrimenti.

Thread all'interno dello stesso blocco possono cooperare tra di loro, condividendo dati attraverso la memoria condivisa, e sfruttando la primitiva di

sincronizzazione a barriera `__syncthreads()`. Quest'ultima, dopo essere stata chiamata da parte di un thread, ne blocca l'esecuzione, fino a che tutti gli altri thread appartenenti allo stesso blocco non l'hanno invocata a loro volta. L'uso di `__syncthreads()` assume un ruolo determinante quando, ad un dato momento del programma, si vuol essere sicuri che tutti i processi di un blocco abbiano terminato il loro compito; ad esempio la scrittura dei propri dati in memoria. Questa funzione è l'unico meccanismo di sincronizzazione offerto da CUDA. La mancanza di una primitiva di sincronizzazione globale costringe ad attendere il ritorno della chiamata ad un kernel, per avere la certezza che tutti i blocchi abbiano terminato il proprio compito. Tuttavia, già dalla seconda versione della GPU G80, è stata prevista la possibilità di effettuare operazioni atomiche sulla memoria globale, dando quindi agli sviluppatori la possibilità di implementare primitive a semaforo per la sincronizzazione globale dei thread.

In questo capitolo si sono osservate, sia tutte le caratteristiche dell'ambiente CUDA, sia i limiti che vi impone la GPU G80. Ovvero i limiti sulla configurazione di esecuzione dei kernel, il numero di registri disponibili, l'uso di funzioni atomiche, ecc .... Questi limiti non sono gli stessi per ogni GPU, pertanto NVIDIA assegna un valore chiamato *compute capability*, al quale sono associati i limiti e le caratteristiche generali della GPU in questione. La lista completa delle compute capability di ogni GPU è consultabile nella documentazione fornita dalla NVIDIA.

Prima di terminare questa sezione, è importante sottolineare come CUDA sia stato concepito per sfruttare le capacità di calcolo parallelo delle GPU come se queste fossero dei coprocessori della CPU principale. Pertanto

l'*Host* si occupa sia della configurazione e dell'esecuzione dei kernels, sia dei trasferimenti dei dati da e verso la DRAM del *Device* tramite la funzione:

```
cudaMemcpy(void *dest,const void *src,
        size_t count,enum cudaMemcpyKind kind);
```

dove:

- `src`

  è il puntatore all'area di memoria che contiene i dati da trasferire.

- `dest`

  è il puntatore all'indirizzo di memoria in cui salvare i dati.

- `count`

  è il numero di byte da trasferire.

- `kind`

  può assumere i valori:

  - `cudaMemcpyHostToDevice`

    per indicare un trasferimento dati da CPU a GPU.

  - `cudaMemcpyDeviceToHost`

    per indicare un trasferimento dati da GPU a CPU.

I file sorgente di un programma CUDA hanno estensione .cu e devo essere compilati utilizzando il compilatore fornito da NVIDIA secondo la sintassi:

```
nvcc sorgente.cu [-o <Nome_eseguibile>] [opzioni]
```

Il compilatore *nvcc* ha il compito di separare il codice che deve essere eseguito dalla CPU da quello che deve essere eseguito dalla GPU. Un'opzione molto utile è *-deviceemu*, che permette di far eseguire il programma alla CPU. Ovviamente, il flusso delle istruzioni dell'algoritmo sarà diverso in quanto la CPU non eseguirà tutti i thread in parallelo, ma in questa modalità è possibile richiamare in un kernel alcune funzioni che sarebbero inaccessibili alla GPU come per esempio la *printf()*, permettendo in molti casi di effettuare un semplice debugging.

Un esempio del flusso di un'applicazione CUDA è mostrato in Figura 3.14, dove si osserva l'alternanza tra il codice seriale eseguito sull'*Host*, e l'esecuzione parallela di uno o più kernel sul *Device*.

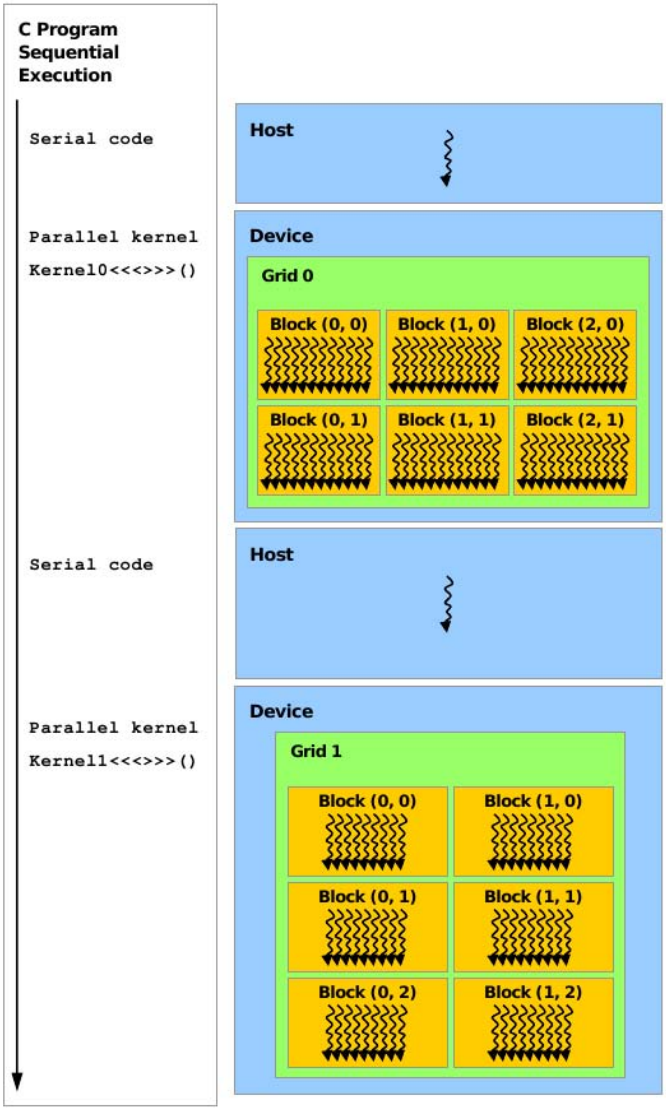

Figura 3.14: Flusso delle operazioni di un'applicazione CUDA

Capitolo 4

Implementazione del modello di dispersione nell'atmosfera

4.1 Introduzione al modello

Il fenomeno fisico più difficile da modellare per i processi di dispersione atmosferica è la turbolenza, una forma di movimento il cui esito diviene rapidamente imprevedibile. Nel fenomeno delle turbolenze, sono presenti altri due fenomeni fisici, il *moto Browniano* e la *diffusione turbolenta*, che devono essere tenuti in considerazione nella realizzazione del modello. Di seguito vengono brevemente riportati gli sforzi dei ricercatori che negli anni hanno tentato di modellare questi due fenomeni.

4.1.1 Moto Browniano

Nel 1827 Robert Brown studiò il fenomeno che venne poi denominato come *moto Browniano*. Egli dapprima osservò dei piccoli granelli di polline sospesi

in acqua che mostravano un moto irregolare e molto agitato. Essendo un botanico fu naturale per lui chiedersi se questo moto fosse caratteristico solo della vita organica. Continuò quindi i propri studi utilizzando altri tipi di particelle puntiformi, organiche ed inorganiche, e osservò lo stesso fenomeno. Concluse quindi che tale moto non è di natura organica.

Einstein pubblicò la prima spiegazione soddisfacente del *moto Browniano* nel 1905; quest'opera è considerata il punto d'inizio della modellazione stocastica dei fenomeni naturali. Egli considerò il moto Browniano come un processo di *cammino casuale*, come fece Smoluchowski nel 1906 e negli anni seguenti. Markov pubblicò un'opera basilare sul cammino casuale nel 1912.

Langevin propose una soluzione alternativa per il *moto Browniano* nel 1908, nella quale introdusse l'equazione che prese il suo nome. L'equazione fu scritta nella forma:

$$du/dt = -a_1 + b\xi(t), \tag{4.1}$$

dove u è la velocità della particella, t è il tempo e a_1 è un coefficiente d'attenuazione associato alla resistenza viscosa sulla particella. Il prodotto del coefficiente b e la funzione casuale $\xi(t)$ è una componente di rapida accelerazione fluttuante dovuta al bombardamento molecolare irregolare e asimmetrico della particella. Nei successivi studi del moto Browniano nei campi elettrici e gravitazionali, fu aggiunto all'equazione base di Langevin un termine aggiuntivo di velocità a_0 (Uhlenbeck e Ornstein 1930; Wang e Uhlenbeck 1945; Chandrasenkhar 1943):

$$du/dt = a_0 - a_1 + b\xi(t). \tag{4.2}$$

Va sottolineato che la forma in cui sono scritte le equazioni (4.1) e (4.2) è piuttosto primitiva poiché sono descritte come termini della funzione casuale

$\xi(t)$. L'equazione di Langevin del 1908 fu il primo esempio di equazione stocastica differenziale, ma un'adeguata base matematica non fu disponibile fino a più di 40 anni dopo quando Ito formulò le proprie teorie per le equazioni differenziali stocastiche. Questo perché $\xi(t)$ è una funzione stocastica ed ha definite solo alcune proprietà statistiche.

L'equazione di Langevin è un'equazione Lagrangiana; successivamente venne studiata anche la sua versione Euleriana. Questo perché le coordinate Euleriane sono più facili da utilizzare rispetto a quelle Lagrangiane. L'equivalente Euleriano dell'equazione di Langevin è un equazione differenziale alle derivate parziali, l'equazione di Fokker-Planck. Il nome le fu attribuito dopo che Fokker pubblicò un articolo sul moto Browniano nel 1914 e Planck generalizzò la discussione nel 1917. Viene erroneamente chiamata anche equazione di Smoluchowski, a causa dell'opera che pubblicò nel 1915-1916, ma comunemente al suo nome si fa corrispondere un'altra equazione. Kolmogorov nel 1931 stabilì una base rigorosa per l'equazione di Fokker-Planck, quindi quando si parla di equazione di Fokker-Planck si fa riferimento a due varianti, l'equazione *precedente* e quella *successiva*. Una versione introduttiva dell'equazione di Fokker-Planck può essere scritta nella forma:

$$\frac{\partial P(x,u,t)}{\partial t} + u\frac{\partial P(x,u,t)}{\partial x} = -\frac{\partial}{\partial u}[(a_0 - a_1 u)P(x,u,t)] + \frac{\partial^2}{2\partial u^2}[b^2 P(x,u,t)]. \tag{4.3}$$

Qui a_0, a_1, e b hanno la stessa definizione dell'equazione (4.2), e $P(x,u,t) = P(x,u,t|x_0,u_0,t_o)$, dove $P(x,u,t|x_0,u_0,t_o)$ è funzione di distribuzione di probabilità condizionale o di transizione nella posizione x, a velocità u al tempo t, partendo da x_0 a velocità u_0 al tempo t_0.

Il lavoro sull'equazione di Langevin e di Fokker-Planck come modello di

moto Browniano continuò, e fra le pubblicazioni successive, due delle più importanti sono quelle di Uhlenbeck e Ornstein (1930) e Wang e Uhlenbeck (1945). L'importanza del primo è indicata col termine di processo di *Ornstein-Uhlenbeck* (un moto stazionario modellato dalle equazioni (4.1)-(4.3) con a_0 e a_1 costanti e $P(x, u, t)$ indipendente dal tempo), usato nei testi di Arnold (1974), Gardiner (1983; 1990) e van Kampen (1981; 1992). Infine va ricordata la revisione completa e dettagliata di handrasekhar (1943).

4.1.2 Diffusione turbolenta

Taylor (1921) fu il primo ad applicare la statistica del moto Lagrangiano delle particelle al problema della diffusione turbolenta. Egli non fece nessun riferimento esplicito ai lavori precedenti di Einstein, Langevin, o degli altri sul *moto Browniano*, ma notò alcune somiglianze fra la diffusione turbolenta e il cammino casuale. Introdusse l'idea di una funzione di autocorrelazione della velocità e ottenne due risultati fondamentali:

1. il valore quadratico medio dello spostamento nella diffusione turbolenta è inizialmente proporzionale al tempo

2. successivamente diventa proporzionale alla radice quadrata del tempo.

L'interesse specifico nelle equazioni di Langevin e Fokker-Planck ha origine grazie a Obukhov (1959) che propose di utilizzarle per modellare la diffusione turbolenta. Il suo suggerimento fu accolto con scetticismo come mostrato dalle discussioni pubblicate nei suoi articoli. Tuttavia, la sua proposta portò ad una nuova linea di sviluppo per i modelli di diffusione turbolenta. Molti ricercatori non usavano l'equazione di Fokker-Planck come suggerito

da Obukhov, ma implicitamente o talvolta anche esplicitamente utilizzavano l'equazione di Langevin. Il primo articolo teorico su quest'argomento fu pubblicato nel 1960, mentre la prima simulazione numerica avvenne nel 1970; da allora vi sono stati notevoli progressi sia nella teoria che nelle applicazioni fino al 1980.

Il primo ad adottare l'idea di Obukhov fu Lin che pubblicò due articoli nel 1960. Seguì Novik(1962), e Krasnoff e Peskin (1971) che pubblicarono quella che si può definire una vera e propria critica al modello di Lin. Questi modelli si preoccupavano della teoria e non delle applicazioni pratiche.

Hanna (1979), non menzionò nessuno dei precedenti modelli, e mostrò che l'equivalente delle differenze finite dell'equazione di Langevin era approssimativamente valido per le osservazioni Lagrangiane ed Euleriane della velocità del vento nello strato atmosferico. Gifford (1982), citò i risultati di Hanna così come gli autori precedenti nel proporre che l'equazione di Langevin venisse utilizzata per modellare la diffusione orizzontale nell'atmosfera dove il tempo di diffusione varia dai pochi secondi a svariati giorni. La proposta di Gifford fu criticata da Sawford (1984). Egli notò che l'equazione di Langevin era applicabile alle turbolenze tridimensionali soltanto laddove la diffusione relativa su scala globale era quasi bidimensionale.

Le prime simulazioni numeriche che utilizzavano l'equazione di Langevin (eq. 4.1) non fornirono risultati fisici realistici per campi di turbolenze non omogenei. Nello specifico, pur partendo da particelle uniformemente distribuite, la concentrazione delle particelle tendeva ad aumentare nelle regioni con una ridotta varianza della velocità, violando quindi il secondo principio della termodinamica. Wilson et al. (1981) furono i primi a proporre

di aggiungere un fattore di *correzione*, analogo ad a_0 nell'equazione (4.2). Seguirono Legg e Raupach (1982) con una differente proposta per a_0. Ogni correzione era un'aggiunta *ad hoc*, seppur parzialmente basata su un ragionamento fisico, all'equazione di Langevin. Nonostante le differenze fra le due proposte, esse fornivano risultati simili in sistemi in cui il gradiente della varianza della velocità era piccolo. Wilson dal primo gruppo e Legg dal secondo, si unirono a quello di Thompson (1983) per mostrare perché i due modelli, sotto certe condizioni, davano risultati simili. Dimostrarono inoltre che una versione modificata del modello di Wilson dava i migliori risultati in caso di turbolenza non omogenea. In chiusura sottolinearono riguardo a quest'ultimo modello: *"... benché non possiamo fornire alcuna prova che sia corretto, sembra degno di essere approfondito"*.

Thompson (1984) usò le funzioni generatrici dei momenti della densità di probabilità delle fluttuazioni della velocità e la funzione casuale $\xi(t)$ per derivare il modello di Wilson. Van Dop et al. (1985) trasformarono l'equazione di Langevin nel suo equivalente Euleriano, l'equazione di Fokker-Planck, e determinarono i coefficienti dell'equazione di Langevin riferendo il modello di cammino casuale all'equazione di Eulero per la conservazione della massa e della specie. Sawford (1986) notò che i loro risultati erano equivalenti a quelli di Thompson e dimostrò che "... solo il modello di Wilson et al. (1983) soddisfa il secondo principio della termodinamica in condizioni di turbolenza non omogenea".

L'articolo di Thompson (1987) è un classico punto di riferimento. Egli usò una forma più generale delle equazioni di Langevin e di Fokker-Planck nel valutare diversi criteri di progettazione dei modelli stocastici Lagrangiani

di diffusione turbolenta. Provò che il criterio di "*mescolamento uniforme*" (*well-mixed*) era equivalente se non superiore a tutti gli altri criteri. Mostrò che la distribuzione casuale deve essere Gaussiana se il processo stocastico deve essere continuo. Infine, dichiarò, ma non dimostrò, che non esiste un'unica soluzione al problema della correzione della direzione nel caso multidimensionale. Luhar e Britter (1989) e Weil (1990), utilizzarono in maniera indipendente il criterio di "*mescolamento uniforme*" di Thompson per derivare una variante molto complessa dell'equazione (4.2) per la diffusione verticale negli strati convettivi con turbolenza non Gaussiana. Wilson e Flesch (1993) confinarono questo problema nel contesto dell'equazione di Langevin che soddisfaceva il criterio di "*mescolamento uniforme*". Esaminarono inoltre l'argomento dei passi temporali non infinitesimi.

Tutti i riferimenti precedenti a partire da Lin e Reid (1963), riguardano il modello basato sull'equazione di Langevin (*Langevin Equation Model, LEM*) in una forma o in un'altra. Il modello di spostamento casuale (*Random Displacement Model RDM*) è menzionato brevemente solo da alcuni; per esempio, van Dop et al. (1985) e Thompson (1987), che discussero l'equazione differenziale stocastica dello spostamento casuale come limite dell'equazione di Langevin. Questo limite è illustrato da Arnold (1974) dalla trasformazione da processo di velocità stocastica (l'equazione di Langevin) a modello di spostamento casuale. Il risultato di tale trasformazione è un'equazione che è equivalente alla classica equazione di diffusione. L'RDM è stata discussa in dettaglio da Durbin (1980) e applicata da Durbin e Hunt (1980). La trasformazione matematica da LEM a RDM fu dimostrata da Durbin (1983,1984), Boughton et al. (1987) e da Bass e Troen (1989).

4.2 Il modello di Luhar e Britter

Sullivan (1971), Hall (1975), Reid (1979), Wilson et al. (1981) e Ley (1982), sono stati i primi a formulare dei modelli di cammino casuale basati sull'equazione di Langevin, per applicazioni con campi di turbolenza disomogenei. Questi modelli, ad ogni modo, erano limitati a problemi dove solo la scala temporale varia con la posizione e non l'energia della turbolenza. Legg e Raupach (1982), Ley e Thompson (1983) realizzarono successivamente dei modelli di cammino casuale che incorporavano la media della curvatura della velocità in flussi turbolenti disomogenei. Utilizzando numeri casuali con distribuzione non Gaussiana, Thompson (1984; 1986) e van Dop et al. (1985), estesero i modelli per turbolenze fortemente disomogenee e non stazionarie. Benché i modelli con forzante casuale non Gaussiana fornissero risultati ragionevolmente comparabili con le misure sperimentali, gli studi di Thompson (1987) hanno dimostrato che questi modelli non sono fisicamente ben posti, nel senso che non soddisfano alcuni requisiti fisici di base come il criterio di *mescolamento uniforme*. Questa è una condizione generale che porta al soddisfacimento degli altri criteri fisici. Inoltre è stato dimostrato che la forzante casuale in un modello di cammino casuale deve essere Gaussiana, altrimenti la violazione di questo requisito porta la velocità Lagrangiana ad essere una funzione discontinua nel tempo. Luhar e Britter hanno sviluppato un modello di cammino casuale non lineare con forzante casuale Gaussiana che soddisfa il criterio di mescolamento uniforme e che è applicabile a turbolenze disomogenee sia Gaussiane che non.

Il modello considera il caso monodimensionale di dispersione verticale in turbolenza stazionaria ed orizzontalmente omogenea senza flusso medio. Se

$z(t)$ e $w(t)$ sono rispettivamente la posizione e la velocità di un elemento del fluido o di una particella o di un tracciante , allora l'evoluzione di (w, z) può essere descritta dall'equazione differenziale stocastica non lineare:

$$\left.\begin{aligned} dw &= a(z,w)dt + b(z,w)d\xi \\ dz &= wdt \end{aligned}\right\} \tag{4.4}$$

Dove $d\xi$ sono gli incrementi di velocità casuali con $d\xi(t_1)$ e $d\xi(t_2)$ indipendenti per $t_1 \neq t_2$ e con media zero e varianza dt.

Per alti numeri di *Reynolds* le accelerazioni di un elemento del fluido o di una particella o di un tracciante può essere considerato un processo di *Markov* poiché l'accelerazione Lagrangiana è piccola rispetto ai passi temporali, molto più della scala temporale di *Kolmogorov*.

Se $P(z, w)$ è la funzione di di distribuzione di probabilità (PDF) di tutte le particelle, deve soddisfare l'equazione di Fokker-Planck (4.4). Questa è data da:

$$w\frac{\partial P}{\partial z} + \frac{\partial}{\partial w}(aP) = \frac{1}{2}\frac{\partial^2}{\partial w^2}(b^2 P) \tag{4.5}$$

Il criterio di mescolamento uniforme richiede che se le particelle del tracciante sono inizialmente ben mescolate esse devono rimanere in tale stato. Pertanto se $P_a(z, w)$ è la PDF degli elementi del fluido, $P_a(\equiv P)$ deve soddisfare anche l'equazione di Fokker-Plank (equazione 4.5). L'equazione risultante può essere scritta per semplicità in due parti:

$$aP_a = \frac{\partial}{\partial w}\left(\frac{1}{2}b^2 P_a\right) + \phi \tag{4.6a}$$

dove ϕ soddisfa:

$$\frac{\partial \phi}{\partial w} = -w\frac{\partial P_a}{\partial z} \tag{4.6b}$$

e $\phi \to 0$ con $|w| \to \infty$.

Questo è il criterio di mescolamento uniforme. Nel caso monodimensionale stazionario ed omogeneo nell'equazione (4.4) $a = -w/\tau$ e $b = (2w^2\tau)^{\frac{1}{2}}$, che è l'equazione di base di Langevin derivata da Wilson et al. (1983), che soddisfa l'equazione (4.6).

In flussi fortemente disomogenei, come in uno stato convettivo, il momento terzo della fluttuazione della velocità, $\overline{w^3}$, è non nullo. I modelli di cammino casuale che utilizzano una forzante di tipo Gaussiano incorporano la disomogeneità, ma non soddisfano la condizione di mescolamento uniforme. Thomson (1987) sviluppò un modello di cammino casuale utilizzando una PDF 'skewed' per le velocità verticali, valido soltanto nel *surface layer*, e quindi non applicabile all'intero strato convettivo (*convective boundary layer*, CBL).

4.2.1 Funzione di distribuzione di probabilità per velocità verticali

Nello strato limite convettivo la dispersione è fortemente influenzata dalla probabilità di trovarsi in correnti ascensionali o discensionali. Le prime hanno velocità verticali elevate ma occupano poca area orizzontale, mentre le seconde occupano aree ampie ma hanno una velocità inferiore. Quindi la PDF delle velocità verticali ad una certa altezza ha una 'skewness' positiva, cioè la moda è negativa ed è diversa dalla media. Questa proprietà porta alla discesa delle particelle verso il centro del boundary layer quando la sorgente si trova ad altezze elevate. Questo fenomeno, che non è descritto dal modello convenzionale Gaussiano, è stato confermato numericamente (Lamb, 1978) e attraverso esperimenti di laboratorio. La discesa delle particelle porta ad

una grande concentrazione al livello del terreno.

Il modello di cammino casuale di Luhar e Britter richiede una rappresentazione matematica della PDF, i cui parametri statistici sono stimati in base alle osservazioni dei valori dei momenti della componente fluttuante della velocità verticale $\overline{w^2}$ e $\overline{w^3}$, che sono stati misurati da svariati ricercatori in simulazioni in atmosfera e in laboratorio. Come descritto da Baerentsen e Berkowicz (1984) una PDF skewed $P_a(w,z)$ può essere costruita a partire da due distribuzioni gaussiane come segue:

$$P_a(w,z) = A(z)P_A(w,z) + B(z)P_B(w,z), \tag{4.7}$$

dove $A(z)$ e $B(z)$ sono le probabilità dell'occorrenza di correnti verso l'alto o verso il basso. A e B possono essere considerate come l'area occupata da queste correnti. $P_A(w,z) = (\sqrt{2\pi}\sigma_A)^{-1}\exp\left(-0.5[(w-\overline{w}_A)/\sigma_A]^2\right)$ è una PDF Gaussiana, relativamente alle correnti ascensionali, con deviazione standard σ_A e media $\overline{w}_A$, mentre $P_B(w,z) = (\sqrt{2\pi}\sigma_B)^{-1}\exp\left(-0.5[(w-\overline{w}_B)/\sigma_B]^2\right)$ è la PDF Gaussiana relativamente alle correnti discensionali con deviazione standard σ_B e media $\overline{w}_B$. Per la precisione $\overline{w}_B$ rappresenta il valore assoluto della velocità negativa della corrente verso il basso.

A questo punto, ad una qualunque altezza z, $P_a(w,z)$ dovrebbe soddisfare la seguente equazione per i quattro momenti per $n = 0, 1, 2, 3$:

$$A(z)\int_{-\infty}^{\infty} w^n P_A(w,z)dw + B(z)\int_{-\infty}^{\infty} w^n P_B(w,z)dw = \overline{w^n}(z) \tag{4.8}$$

dove $\overline{w}(z) = 0$.

Baerentsen e Berkowicz (1984) hanno posto $\sigma_A = \overline{w}_A$ e $\sigma_B = \overline{w}_B$ per risolvere le quattro equazioni rappresentate dall'equazione 4.8 che ha sei variabili incognite $\sigma_A, \sigma_B, \overline{w}_A, \overline{w}_B, A$ e B. Di seguito si riportano le espressioni per il

calcolo delle altre variabili

$$\left.\begin{aligned} \overline{w}_B &= (\sqrt{(\overline{w^3})^2 + 8(\overline{w^2})^3} - \overline{w^3})/4\overline{w^2} \\ \overline{w}_A &= \overline{w^2}/2\overline{w}_B \\ A &= \overline{w}_B/(\overline{w}_A + \overline{w}_B) \\ B &= \overline{w}_A/(\overline{w}_A + \overline{w}_B) \end{aligned}\right\} \tag{4.9}$$

L'assunzione menzionata precedentemente è basata sul fatto che l'energia turbolenta nelle correnti ascensionali è maggiore di quelle discensionali e la magnitudine della velocità media delle correnti ascensionali è più larga che in quelle ascensionali.

4.2.2 Derivazione delle equazioni del modello

Nel caso monodimensionale ϕ è univocamente determinato a partire dall'equazione (4.6b). Invece nel caso multidimensionale ϕ potrebbe non essere unico e, come osservato da Sawford (1988), i risultati ottenuti da forme di ϕ differenti possono essere diversi. Ma dato che il modello in esame considera la turbolenza soltanto nel monodimensionale (verticale), la soluzione dell'equazione (4.6b) è unica. Una volta determinato ϕ, un espressione per a può essere ottenuta dall'equazione (4.6a) se b è noto. Se si compara la funzione di struttura della velocità Lagrangiana ottenuta dall'equazione (4.4), con quella determinata dalla teoria di Kolmogorov sull'isotropia locale nell'intervallo inerziale, si può trovare che

$$b = \sqrt{C_0 \varepsilon} \tag{4.10a}$$

dove C_0 è una costante che si assume valga 2.0 e ε il tasso medio di dissipazione dell'energia cinetica turbolenta. In alternativa è possibile ottenere:

$$b = \sqrt{\frac{2\overline{w^2}}{\tau}} \tag{4.10b}$$

come avviene nell'equazione di Langevin di base. La scala temporale τ dovrebbe essere tale che l'equazione (4.10b) sia consistente con la (4.10a). Questo implica che $\tau = \overline{w^2}/\varepsilon$

L'equazione (4.6b) può essere risolta per trovare ϕ dopo aver sostituito l'espressione di P_a dall'equazione (4.7) nell'equazione (4.6b).

Nell'equazione (4.6a):

$$\frac{\partial}{\partial w}\left(\frac{1}{2}b^2 P_a\right) = -\frac{\overline{w^2}}{\tau}Q.$$

Pertanto,

$$a = \frac{\left(-\frac{\overline{w^2}}{\tau}Q + \phi\right)}{P_a} \tag{4.10c}$$

dove

$$Q = \frac{A(w - \overline{w}_A)}{\overline{w}_A^2}P_A + \frac{B(w + \overline{w}_B)}{\overline{w}_B^2}P_B$$

e P_a, P_A, P_B sono dati dall'equazione (4.7). Quindi l'equazione di cammino casuale (4.4) diviene:

$$dw = \frac{-(\overline{w^2}/\tau)Q + \phi}{P_a}dt + \left(\frac{2\overline{w^2}}{\tau}dt\right)^{\frac{1}{2}} d\mu \tag{4.11}$$

dove $d\mu$ è una funzione casuale gaussiana a media zero e varianza uno.

Dall'equazione (4.9) segue che nell'espressione di ϕ:

$$\left.\begin{aligned}
\frac{\partial \overline{w}_B}{\partial z} &= -\frac{1}{F}\left[\overline{w}_B\frac{\partial \overline{w^3}}{\partial z} + \frac{\partial \overline{w^2}}{\partial z}\left(\frac{\overline{w}_B F}{\overline{w^2}} - 3\overline{w^2}\right)\right] \\
\frac{\partial \overline{w}_A}{\partial z} &= \frac{1}{2\overline{w}_B}\frac{\partial \overline{w^2}}{\partial z} - \frac{\overline{w}_A}{\overline{w}_B}\frac{\partial \overline{w}_B}{\partial z} \\
\frac{\partial A}{\partial z} &= \frac{1}{\overline{w}_A + \overline{w}_B}\left(-A\frac{\partial \overline{w}_A}{\partial z} + B\frac{\partial \overline{w}_B}{\partial z}\right) \\
\frac{\partial B}{\partial z} &= \frac{\partial A}{\partial z}
\end{aligned}\right\} \tag{4.12}$$

dove $F = 4\overline{w^2}\overline{w}_b + \overline{w^3}$. Quindi l'equazione (4.11) insieme alla (4.10c) e alla (4.12) costituisce il modello di equazioni del cammino casuale che include

lo skewness nelle velocità verticali e soddisfa il criterio di mescolamento uniforme fornito dall'equazione di Fokker-Planck (4.6).

Nel caso di turbolenza Gaussiana, l'equazione (4.10c) e conseguentemente la (4.11) si riducono alla forma:

$$dw = \left[-\frac{w}{\tau} + \frac{1}{2}\left(\frac{w^2}{\overline{w^2}} + 1 \right) \frac{\partial \overline{w^2}}{\partial z} \right] dt + \left(\frac{2\overline{w^2}}{t} dt \right)^{\frac{1}{2}} d\mu \qquad (4.13)$$

4.3 Generazione di sequenze di numeri casuali

I modelli di simulazione sono creati per indagare le proprietà fondamentali che regolano il comportamento dei fenomeni oggetto di studio, e devono poter astrarre dal singolo caso. Ci si trova quindi ad affidare al caso il valore di alcuni parametri e variabili che, sono fondamentali ai fini della simulazione. Quando questi modelli vengono implementati al calcolatore, sorge la problematica questione di come generare tali valori utilizzando algoritmi deterministici.

Secondo una definizione empirica una sequenza di numeri si dice casuale se le sue proprietà statistiche sono le stesse di quelle di dati provenienti da un esperimento casuale. In maniera più intuitiva una sequenza di numeri si può definire casuale se un qualsiasi elemento della sequenza non può essere determinato a partire dagli altri elementi. Il computer è uno strumento deterministico e per tale motivo non può generare sequenze di numeri davvero casuali, tuttavia utilizzando particolari algoritmi è possibile generare delle sequenze di numeri che hanno le stesse proprietà di una sequenza di numeri casuali. Tali numeri prendono il nome di *pseudo-casuali*, in quanto non sono

effettivamente casuali, ma sono in grado di superare una serie di test statistici che conferiscono a tali numeri una apparente casualità. Gli algoritmi per la generazione di sequenze di numeri casuali prendono quindi il nome di *Pseudo-Random Number Generators* (PRNG).

Numeri casuali possono scaturire dall'osservazione di processi fisici. I cosiddetti generatori hardware di numeri casuali sono dispositivi che, sfruttando fenomeni fisici microscopici, sono in grado di restituire sequenze di numeri casuali. É il caso di dispositivi per l'analisi del rumore termico in semiconduttori o del decadimento atomico. Tra i generatori hardware di numeri casuali più vicini all'esperienza quotidiana possiamo citare i dadi o la roulette, tutti dispositivi che restituiscono risultati imprevedibili e non correlati se consideriamo una sequenza di lanci. I problemi principali di questi dispositivi risiedono nell'incapacità di riprodurre più di una volta una stessa sequenza di numeri, nel costo e nella lentezza, problemi che invece sono risolvibili utilizzando dei generatori software.

Una sequenza di numeri pseudo-casuali deve soddisfare almeno le proprietà statistiche di:

- *distribuzione*

  gli elementi della sequenza di numeri devono essere distribuiti secondo una funzione $f(x)$ in un intervallo $[x_{min},\ x_{max}]$. Un tipo di distribuzione molto frequente è quella uniforme, in cui $f(x) = \frac{1}{|x_{max} - x_{min}|}$ nell' intervallo e $f(x) = 0$ all'esterno.

- *indipendenza*

  i numeri della sequenza devono essere indipendenti tra di loro. L'i-esimo numero generato non deve quindi influire sull'i+1-esimo.

I *Pseudo-Random Number Generators* utilizzati in ambito statistico sono algoritmi che, nella maggior parte dei casi, seguono lo schema introdotto da *L'Ecuyer* nel 1994. Secondo tale schema, le parti fondamentali di un RNG sono:

$$(\delta, \mu, f, U, g).$$

Dove:

- δ rappresenta l'insieme finito di stati detto spazio degli stati,
- μ descrive la distribuzione di probabilità, utilizzata per selezionare dallo spazio degli stati δ il seme, detto stato iniziale,
- f è la funzione di transizione utilizzata per determinare lo stato al tempo $t+1$ dato lo stato al tempo t, espressa in forma di $S_{i+1} = f(S_i)$,
- U determina lo spazio degli output. Solitamente si utilizzano valori compresi nell'intervallo [0, 1], poiché da questo è possibile ottenere una sequenza di numeri casuali compresi in un qualsiasi intervallo $[a,\ b]$ mediante la formula $x = (b-a)t + a$ dove $t \in [0,\ 1]$ e $x \in [a,\ b]$,
- g, è la funzione di output. Dato un qualsiasi stato S_i, $u_i = g(S_i) \in U$. Gli output $u_0, u_1, u_2, \ldots$ sono i numeri casuali prodotti dallo Pseudo-Random Number Generators.

Dal momento che lo spazio degli stati δ è finito, selezionando un qualsiasi seme S_i, esisterà sempre un valore l per cui: $S_{i+l} = S_i$. In pratica, qualsiasi sia lo stato iniziale, dopo un numero l di iterazioni, il generatore torna inevitabilmente allo stato iniziale. Dal momento che sia la funzione di transizione che la funzione di output sono deterministiche, allora anche gli output generati torneranno inevitabilmente allo stato iniziale.

Il valore di l più piccolo per cui si realizza il ritorno allo stato iniziale è chiamato periodo del PRNG ed è indicato col simbolo ρ . Il valore di ρ è sempre minore o uguale alla cardinalità dello spazio degli stati δ . Nella pratica, essendo δ memorizzato in un calcolatore sotto forma di stringa binaria di lunghezza k, allora ρ avrà lunghezza:

$$\rho \leq 2^k.$$

Pertanto PRNG di buona qualità si distinguono per valori di ρ prossimi a $|\delta|$, in maniera tale che il sistema non compia cicli brevi e prevedibili.

Oltre al periodo, altri parametri di valutazione dei PRNG sono:

- *efficienza*

  Un buon generatore è un software che utilizza un quantitativo ridotto di risorse computazionali, misurate in termini di memoria allocata e tempo di calcolo.

- *ripetibilità*

  Gli algoritmi di generazione devono essere in grado di riprodurre esattamente la stessa sequenza di numeri pseudo-casuali partendo dallo stesso stato iniziale.

- *portabilità*

  i PRNG devono essere realizzati in maniera da essere il più possibile indipendenti dal contesto hardware e software in cui sono implementati.

Proprio a causa di quest'ultimo parametro, la scelta dell'algoritmo di generazione di sequenze di numeri casuali, è ricaduta sul generatore *Mersenne Twister*, poichè altri generatori, seppur di ottima qualità, richiedono per l'im-

plementazione una quantità di memoria o strutture dati complesse che non sono disponibili in CUDA.

4.3.1 Il generatore Mersenne Twister

Il Mersenne Twister è un generatore di numeri pseudo-casuali sviluppato nel 1997 da *Makoto Matsumoto* e *Takuji Nishimura*. Il nome dell'algoritmo deriva dal fatto che la lunghezza del periodo coincide con un numero primo di Mersenne[1].

La sua versione più recente chiamata anche *MT 19937* gode dei vantaggi:

- ha un periodo molto lungo, pari a $2^{19937} - 1$
- permette di generare punti equidistribuiti in uno spazio fino a 623 dimensioni.
- è molto veloce ed fa un uso efficiente della memoria
- numerosi test statistici confermano l'ottima qualità dei numeri generati.

L'algoritmo Mersenne Twister può essere scritto nella forma:

$$x_{k+n} = x_{k+m} + (x_k^{upper} | x_{k+1}^{lower}) * A \quad k = 0, 1, \ldots$$

dove:

- w rappresenta il numero di bit dei numeri da generare,
- n e m sono tali che $1 \leq m \leq n$ ed esprimono il grado di ricorrenza,

[1]Un numero primo di Mersenne è un numero primo esprimibile come $2^n - 1$, dove n è un numero intero a sua volta primo.

- r è tale che $0 \leq r \leq w$ ed indica il numero di bit per la concatenazione dei bit dei numeri,
- A costituisce una matrice di bit di dimensione $w \times w$
- $(x_k^{upper}|x_{k+1}^{lower})$ è il numero dato dalla concatenazione degli r bit più significativi di x_k e dai w-r bit meno significativi di x_{k+1}

4.3.2 Il generatore Mersenne Twister in CUDA

L'algoritmo Mersenne Twister, come la maggior parte dei generatori di numeri pseudo-casuali, è iterativo, pertanto è difficile da parallelizzare con più thread. Inoltre la GPU richiede una griglia d'esecuzione composta da un elevato numero di thread per poter essere sfruttata a pieno. La soluzione più veloce e semplice è allora quella di avere più processi Mersenne Twister in parallelo. Tuttavia valori iniziali anche molti diversi non prevengono l'emissione di sequenze correlate. Per risolvere questo problema e per un'implementazione più efficiente su architetture parallele *Makoto Matsumoto* e *Takuji Nishimura* hanno sviluppato una libreria apposita per l'inizializzazione e la creazione dei parametri.

La libreria accetta una stringa di 16 bit, l'id del thread, come input e la codifica con un approccio di tipo *per-thread* nei parametri del Mersenne Twister. In questo modo ogni thread può agire indipendentemente dagli altri.

La matrice $A(w \times w)$ è implementata per semplicità nella forma:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & & & \\ 0 & 0 & 1 & 0 & & \vdots & \\ 0 & 0 & 0 & 1 & & & \\ & & & & 1 & & \\ & & \cdots & & & \ddots & \\ & & & & & & 1 \\ a_{w-1} & a_{w-2} & & & & a_1 & a_0 \end{bmatrix}$$

in questo modo i thread possano avere gli stessi parametri, come ad esempio il seed, ad eccezione dell'ultima riga di A.

L'algoritmo genera sequenze di numeri pseudo-casuali equidistribuite nell'intervallo $(0, 1)$. Per ottenere invece una distribuzione Gaussiana si può ricorrere alla trasformazione di *Box-Muller*.

4.4 Implementazione del modello in matlab

Prima di implementare il modello in ambiente CUDA è stato necessario scrivere una prima versione non parallela del software. Si è scelto così *MATLAB*, un linguaggio che, oltre ad eseguire l'algoritmo, fosse anche in grado di realizzare dei grafici, in modo da poter controllare visivamente il risultato della simulazione.

MATLAB è un ambiente per il calcolo numerico e l'analisi statistica che comprende anche l'omonimo linguaggio di programmazione creato dalla *MathWorks*. MATLAB consente di manipolare matrici, visualizzare funzioni e dati, implementare algoritmi ed è disponibile per diversi sistemi operativi, tra cui GNU/Linux, Unix, Windows e Mac OS.

Il linguaggio di programmazione di Matlab è di tipo interpretato, quindi le righe di codice vengono direttamente eseguite, saltando la fase di compilazione. Questa soluzione permette ai software scritti in questo linguaggio di essere completamente indipendenti dalla piattaforma di esecuzione, rinunciando però ad una parte delle prestazioni.

Matlab offre la possibilità di scrivere script e programmi utilizzando le funzioni del linguaggio. In questo modo è possibile estenderne le funzionalità e costruire delle funzioni personalizzate di elaborazione ed analisi dei dati. I codici risultano particolarmente compatti grazie alla notazione vettoriale che permette di trattare interi vettori e matrici di dimensione arbitraria come variabili elementari, e all'operatore "." che consente di evitare di dover indicare esplicitamente le operazioni sui singoli elementi. Grazie a queste caratteristiche è quindi possibile utilizzare la stessa struttura dati per memorizzare ognuna delle caratteristiche di tutte le particelle.

I parametri di input per la simulazione sono l'altezza `zzi` della sorgente, che emette un numero `NP` di particelle, il tempo finale `TFin` e i valori δt δx e δz che rappresentano la risoluzione spaziale (`dx` e `dz`) e temporale (`dt`) della simulazione.

L'algoritmo alla base della simulazione può essere espresso nella forma:

$$\left.\begin{aligned} W_{n+1} &= W_n + \tfrac{1}{P_{a_n}} \left[-\alpha_n Q_n + \phi_n\right] \Delta T + \left(2\alpha_n \Delta T\right)^{\frac{1}{2}} \Delta\mu \\ Z_{n+1} &= Z_n + W_n \Delta T \end{aligned}\right\} \tag{4.14}$$

dove $Z_0 = Z_s$, cioè l'altezza della sorgente, W_0 è pari alla velocità Euleriana all'altezza della sorgente, e gli altri parametri sono stati descritti nei precedenti paragrafi e approfonditamente da Luhar e Britter (1989).

Per garantire una buona modularità del codice e per favorirne la riusabilità l'implementazione dell'algoritmo è stata realizzata suddividendo le varie

funzionalità in più file. Il codice sorgente è quindi composto dalle function:

1. `LB`

   la funcion principale dell'algoritmo che rappresenta il punto di partenza per la simulazione,

2. `initW`

   necessaria per l'inizializzazione delle velocità delle particelle,

3. `Pa`

   che è la vera e propria implementazione della formula 4.14, e che aggiorna la quota e la velocità delle particelle,

4. `getPar.m`

   per il calcolo dei parametri della simulazione,

5. `NormPDF`

   che implementa la funzione di distribuzione di probabilità.

La struttura dati che meglio si presta per la memorizzazione delle informazioni delle particelle è il vettore. In particolare nella function principale ne vengono utilizzati due, uno `z` relativo all'altezza a cui si trovano le particelle, e l'altro `w` relativo alla velocità delle particelle. Questi due vettori devono essere inizializzati, il primo al valore `zzi`, che rappresenta l'altezza a cui si trova la fonte, e l'altro utilizzando la function `initW`.

In base ai parametri `TFin` e `dt` è possibile, effettuando una semplice divisione, determinare il numero `N` di passi temporali necessari per completare la simulazione. L'algoritmo consiste quindi nell'aggiornare ad ogni passo la quota e la velocità delle particelle tramite la function `Pa`. Quest'ultima

richiama le function `getPar` e `NormPDF` per ottenere i parametri necessari per il calcolo della formula 4.14, ed infine controlla il numero di particelle la cui quota è minore di 0 o maggiore di 1. A queste particelle deve essere applicato un rimbalzo elastico che consiste nell'inversione della velocità e nell'aggiornamento della quota. Per le particelle che si trovavano al di sotto del livello 0 è sufficiente ribaltare la quota con una semplice inversione di segno mentre per le particelle che si trovano oltre la quota 1 questa viene aggiornata calcolandola come 2.0-`z[i]`, dove `i` è l'indice della `i`-esima particella.

Durante gli `N` passi temporali ogni `dx/dt` passi vengono calcolate le concentrazioni delle particelle alle `1/dz` possibili quote. Tali valori vengono salvati nella `k`-esima colonna della matrice `ZSave`, con `k` inizializzato a 0 ed incrementato di uno ogni volta che vengono calcolate le concentrazioni.

Trascorsi gli `N` passi temporali, la matrice `ZSave`, di dimensioni (1/dz) x (TFin/dx), conterrà i valori della concentrazione di particelle in base ai valori `dx` e `dz` della risoluzione spaziale, e può essere utilizzata per effettuare il grafico delle relative isolinee. In fig. 4.1 ne è mostrato un esempio ottenuto con i valori `zzi`=0.49, `NP`=2000, `dt`=0.01, `dx`=0.1, `dz`=0.05 e `TFin`=4.

4.5 Implementazione del modello in CUDA

Attualmente non esiste un vero e proprio standard in CUDA per la nomenclatura delle variabili. In questo lavoro di tesi si è utilizzata la notazione secondo cui al nome della variabile si aggiunge `_h` se questa viene allocata nella memoria dell'*host* cioè della CPU, e `_d` se invece la memoria da allocare e quella del *device* cioè della GPU.

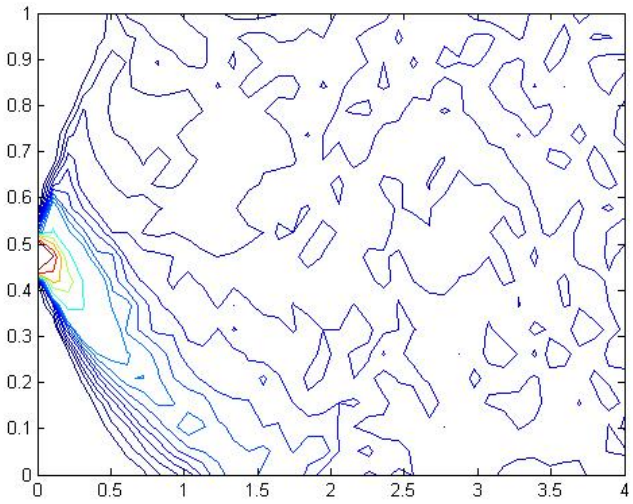

Figura 4.1: output di una simulazione in MATLAB

Per favorire la portabilità e la riusabilità del codice, è stato realizzato un *header* file che raccoglie i vari kernel. Il progetto risulta nel suo complesso composto dai file:

1. `LB.cu`

   che contiene il `main` e che rappresenta il punto di entrata del programma.

2. `myCudaLib.h`

   header file che comprende i kernel:

   - `__device__ void getw2(float *z, float *w2, int N)`

     per il calcolo di $\overline{w^2}/w_*^2$,

   - `__device__ void getw3(float *w2, float *w3, int N)`

     per il calcolo di $\overline{w^3}/w_*^3$,

- `__device__ void getwb(···)`

  per il calcolo di $\overline{w}_B$,

- `__device__ void getwa(···)`

  per il calcolo di $\overline{w}_A$,

- `__device__ void getb(···)`

  per il calcolo di B,

- `__device__ void geta(···)`

  per il calcolo di A,

- `__device__ void getf(···)`

  per il calcolo di F,

- `__device__ void getdw2dz(···)`

  per il calcolo di $\partial\overline{w^2}/\partial z$,

- `__device__ void getdw3dz(···)`

  per il calcolo di $\partial\overline{w^3}/\partial z$,

- `__device__ void getdwbdz(···)`

  per il calcolo di $\partial\overline{w}_B/\partial z$,

- `__device__ void getdwadz(···)`

  per il calcolo di $\partial\overline{w}_A/\partial z$,

- `__device__ void getPhi(···)`

  per il calcolo di ϕ,

- `__device__ void getQ(···)`

  per il calcolo di Q,

- `__device__ void getTau(···)`

  per il calcolo di τ,

- `__device__ void getP(···)`

  per il calcolo di P,

- `__device__ void getdW(···)`

  per il calcolo di W,

- `__device__ void getZ(···)`

  per il calcolo di Z,

- `__device__ void getWInit(···)`

  per il calcolo di W durante l'inizializzazione,

- `__global__ void normPDF(···)`

  per il calcolo della funzione di distribuzione di probabilità,

- `__global__ void normRnd(···)`

  per convertire un array di numeri casuali con distribuzione Gaussiana in una con media μ e varianza σ,

- `__global__ void initArray(···)`

  per l'inizializzazione di tutti gli elementi di un array ad uno stesso valore,

- `__device__ void checkBorder(···)`

  per l'eventuale rimbalzo elastico delle particelle,

- `__global__ void getPar(···)`

  per il calcolo dei vari parametri,

- `__global__ void initW(···)`

  per l'inizializzazione di W,

- `__global__ void Pa(···)`

  per il calcolo dell'equazione 4.14,

- `__global__ void histc(···)`

  per il calcolo della concentrazione di particelle,

- `__global__ void divideArray(···)`

  per dividere tutti gli elementi di un array per uno stesso valore,

- `__global__ void copyArrayInMat(···)`

  per copiare un array nella `k`-esima colonna di una matrice,

e la function:

- `void checkCUDAError(const char *msg)`

  che in caso di errore fornisce una descrizione dettagliata a riguardo ed arresta l'esecuzione del programma.

3. `CUDAscript.m`

   function matlab per la visualizzazione grafica dell'output

I parametri di input per la simulazione devono essere passati da linea di comando quando si esegue l'applicazione. In caso di omissione parziale o totale di tali valori, il programma provvede ad informare l'utente illustrandogli la sintassi corretta e mostrando un esempio.

L'algoritmo implementato in MATLAB, non può essere parallelizzato rispetto al tempo, infatti come indicato espressamente dall'equazione 4.14 la simulazione al passo i-esimo dipende direttamente da quello (i-1)-esimo. Tuttavia sfruttando l'indipendenza delle particelle che vengono rilasciate è possibile implementare un parallelismo di tipo spaziale, cioè è possibile applicare l'equazione 4.14 a più particelle parallelamente.

Per far ciò è necessario memorizzare le informazioni relative alle particelle in array separati, in cui l'i-esima componente si riferisce all'i-esima particella.

Con questo tipo di allocazione di memoria ed utilizzando un *thread* per ogni particella, è possibile sfruttare appieno le potenzialità di CUDA, ottenendo in pratica l'equivalente parallelo dell'operatore puntuale del MATLAB. Questo permette inoltre di avere una buona corrispondenza fra il codice MATLAB e quello CUDA (consultabili in appendice).

Come per ogni algoritmo, anche in CUDA, prima di passare alle fasi di calcolo, è necessario effettuare alcune inizializzazioni dopo aver allocato la memoria per le variabili da utilizzare. Quando si ha a che fare con le GPU però bisogna tener conto del fatto che queste ultime utilizzano uno spazio di indirizzamento differente da quello delle CPU. Pertanto in genere i dati vengono prima inizializzati sulla memoria della CPU e poi trasferiti a quelli della GPU utilizzando la function `cudaMemcpy`.

Questo tipo di approccio tuttavia comporta un *overhead* aggiuntivo, e va quindi evitato quando possibile. Nel nostro caso è necessario inizializzare tutti gli elementi di alcuni array con lo stesso valore, come avviene per esempio per l'array `z` relativo alla quota delle particelle che va inizializzato all'altezza della sorgente di emissione. In questo caso, piuttosto che inizializzare l'array nella memoria della CPU e trasferire i dati, è più conveniente utilizzare la GPU stessa facendo inizializzare ogni elemento da un thread, con conseguente risparmio di spazio di memoria e tempo di calcolo.

I kernel sono stati sviluppati facendo si che ogni thread si occupi di un elemento dell'array ed esegua la stessa operazione di tutti gli altri anche se su dati diversi, in modo da massimizzare l'efficienza del programma. Le uniche due eccezioni sono i kernel `checkBorder` e `histc`. Nel primo ogni thread controlla la quota di una particella ed eventualmente ne effettua il rimbalzo

elastico, tuttavia non necessariamente tutte le particelle devono rimbalzare e comunque non tutte rimbalzano in modo uguale o un ugual numero di volte. Per questo motivo può capitare che un thread porti a termine l'esecuzione del kernel prima degli altri e per evitare che prosegua l'esecuzione prima che tutte le particelle siano rimbalzate è necessario effettuare una sincronizzazione dei thread. La primitiva `__syncthreads()` del linguaggio CUDA, permette di effettuare una sincronizzazione a barriera, cioè blocca l'esecuzione di un thread che invoca la funzione fintanto che anche tutti gli altri thread non l'abbiano richiamata.

Nel kernel `histc` invece i threads calcolano le concentrazioni delle particelle alle varie altezze. Per far questo quindi ogni thread effettua l'intera scansione del vettore delle quote ed ognuno si occupa di calcolare la concentrazione in un intervallo differente. Quindi anche se la scansione dell'array avviene contemporaneamente, soltanto uno dei thread aggiorna il proprio contatore, ed anche in questo caso è necessaria una sincronizzazione con `__syncthreads()`.

Per la generazione di array di numeri casuali con distribuzione uniforme nell'intervallo $[0,\ 1]$, è stato utilizzato il kernel `getRandomNum` basato sull'algoritmo Mersenne-Twister dell'SDK della NVIDIA. Il modello implementato richiede sequenze di numeri casuali con distribuzione Gaussiana, pertanto è stato necessario effettuare la trasformazione di Box-Muller utilizzando il kernel `gasDev`. Nell'implementazione MATLAB le sequenze di numeri casuali vengono generate nel momento del loro effettivo utilizzo, in questo caso invece, per sfuttare appieno il parallelismo della GPU, viene generata in parallelo un'unica sequenza di lunghezza `NP*N` che sarà poi utilizzata per le

`NP` particelle durante gli `N` passi della simulazione. Per ottenere sequenze diverse ad ogni esecuzione è possibile legare il *seed* da passare al kernel ad un parametro che varia ad ogni esecuzione come per esempio l'ora locale, utilizzando l'istruzione:

```
srand(time(NULL));
```

e scegliendo poi come *seed* un numuero casuale calcolato dalla CPU tramite l'istruzione `rand()`.

L'esecuzione del programma avviene con un numero fisso di thread per blocco pari a 192. In base al numero `NP` di particelle è quindi possibile determinare il numero di blocchi da utilizzare calcolandolo come:

```
nBlock = NP/threadPerBlock + (NP%threadPerBlock==0?0:1)
```

La formula consiste semplicemente nell'effettuare la divisione fra `NP` ed il numero di thread per ogni blocco, se poi il resto di tale divisione è diverso da 0 è necessario utilizzare un ulteriore blocco.

Al termine della simulazione vengono salvati su file i parametri della simulazione ed i valori ottenuti. Il file viene salvato in formato `ASCII` per permettere di controllare più semplicemente l'esito della simulazione. Oltretutto è possibile ottenere un grafico della concentrazione delle particelle passando il file ottenuto come parametro alla function Matlab `CUDAscript`.

4.6 Analisi delle simulazioni

Poiché la simulazione si basa fortemente sulla casualità, i risultati ottenuti variano anche mantenendo fissi i parametri. In fig. 4.2 vengono illustrati due esempi di output per una delle possibili configurazioni dei parametri.

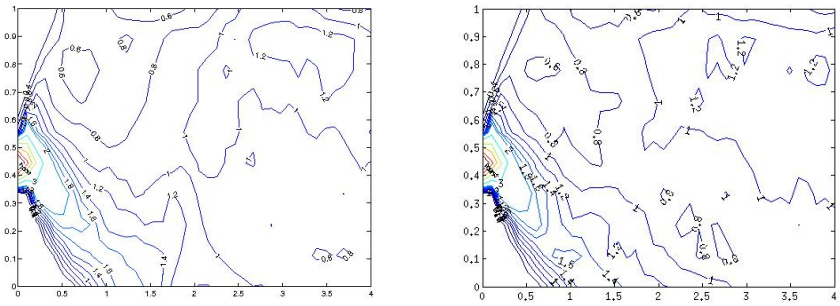

Figura 4.2: `zzi=0.49, NP=2048, dt=0.01, dx=0.1 dz=0.1 TFin=4`

Le librerie CUDA forniscono molte funzionalità fra cui risultano molto utili gli strumenti che consentono di misurare il tempo di esecuzione grazie alle funzioni:

- `cutCreateTimer(&timer);`

  per creare il timer associandolo alla variabile intera senza segno `timer`.

- `cutStartTimer(timer);`

  per avviare il conteggio del tempo di esecuzione

- `cutStopTimer(timer);`

  per arrestare il timer

- `cutGetTimeValue(timer);`

  per ottenere il tempo trascorso, espresso in millisecondi.

Grazie a queste funzioni è quindi possibile calcolare i tempi di esecuzione dell'algoritmo sia su CPU che su GPU. Tuttavia non si può prescindere dalla macchina che eseguirà il codice poiché l'hardware adoperato è determinante

per i risultati delle misurazioni; per tale motivo, di seguito, sono riportate le caratteristiche della piattaforma di lavoro utilizzata.

Il personal computer utilizzato è una workstation HP della serie wx4600 con una CPU Intel Core 2 Duo E6750 che lavora alla frequenza di 2,66 GHz. Quest'ultima ha una memoria cache da 4MByte ed ha a disposizione una memoria centrale che ammonta a 2 GByte. La scheda video, invece, è una semplice NVIDIA GeForce 8400 GS, composta da 2 SM ed avente una memoria di 256 MByte. GPU e memoria della scheda operano entrambe alla frequenza di 450 MHz. Il sistema operativo utilizzato è invece la distribuzione Scientific Linux derivata dalla Red Hat 5.

Il tempo di esecuzione dipende inoltre dalla configurazione di esecuzione, in base al numero di blocchi ed al numero di thread per blocco utilizzati. Va però sottolineato che la configurazione ottimale varia in base all'hardware utilizzato. Nel caso della GPU utilizzata, la configurazione che permette di avere il maggior numero di blocchi attivi per ogni Stream Multiprocessor è quella costituita da 196 thread per blocco.

Al variare dei parametri della simulazione varia anche il tempo di esecuzione. In particolare il tempo di calcolo cresce all'aumentare del numero di particelle, del tempo di simulazione o della risoluzione della simulazione che implica valori di δt piccoli.

Di seguito sono riportati i tempi medi relativi a dieci esecuzioni dell'algoritmo su CPU e GPU, con relativo guadagno prestazionale, al variare dei parametri.

Come si può notare dalle tabelle e dai grafici, il tempo di esecuzione su GPU è almeno 26 volte inferiore rispetto a quello su CPU. In particolare

Tabella 4.1: `NP=512, dx=0.1 dz=0.1 TFin=4`

`dt`	CPU time ms	GPU time ms	speed up
0.01	3145	117	26.88
0.005	6279	200	31.39
0.001	30879	882	35.01
0.0001	307788	8520	36.12

Tabella 4.2: `NP=512, dt=0.01 dx=0.1 dz=0.1`

`TFin`	CPU time ms	GPU time ms	speed up
4	3145	117	26.88
16	12451	465	26.77
32	24889	929	26.79
64	49859	1859	26.82

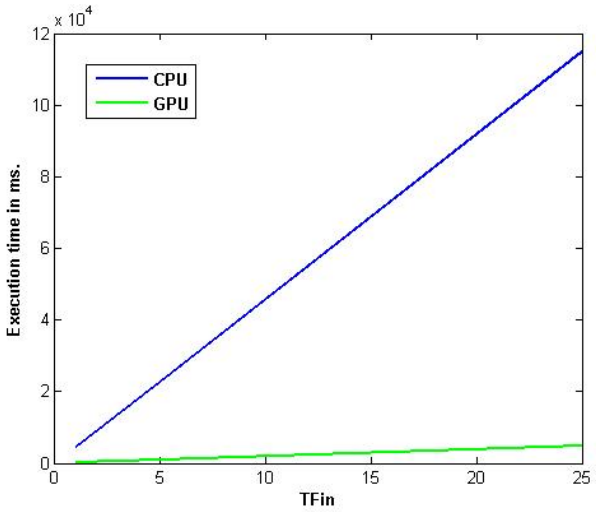

Figura 4.3: Tempi di esecuzione al variare del parametro `TFin`

Tabella 4.3: `dt=0.01 dx=0.1 dz=0.1, TFin=4`

NP	CPU time ms	GPU time ms	speed up
512	3145	117	26.88
1024	5226	181	28.87
2048	9068	289	31.37
4096	17088	521	32.79
8192	32632	954	34.20
16384	67951	1837	36.99
32768	148071	3579	41.37

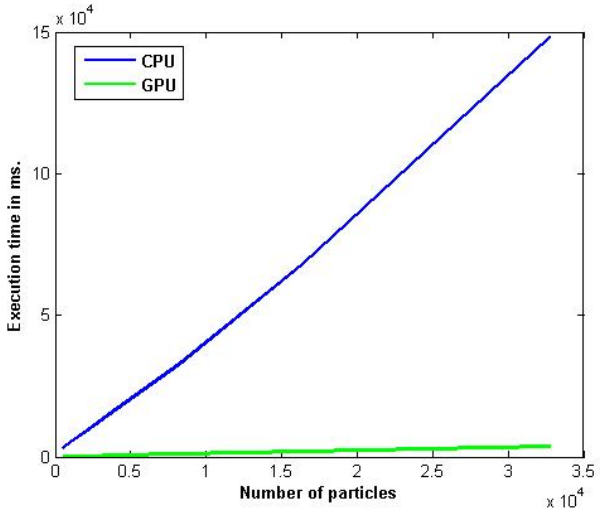

Figura 4.4: Tempi di esecuzione al variare del parametro `NP`

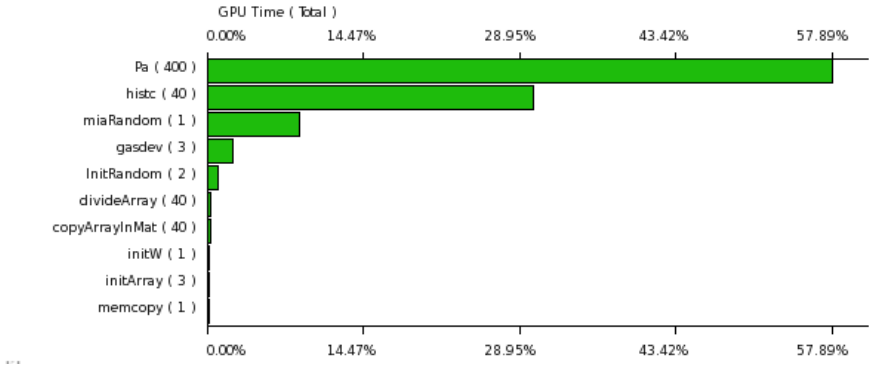

Figura 4.5: Percentuali dei tempi di esecuzione.

va notato che all'aumentare del numero di passi temporali lo *speed-up* resta pressochè costante, mentre aumentando il numero di particelle le prestazioni migliorano sempre più. Ciò è dovuto la fatto che l'algoritmo non è parallelizzato rispetto al tempo, e quindi il tempo di esecuzione dipende linearmente dal numero di passi temporali. Un numero maggiore di particelle invece influisce meno sul tempo di calcolo poichè queste vengono elaborate in parallelo.

Il grafico in fig. 4.5 indica in che percentuale incide sul tempo di calcolo ogni kernel, e come ci si aspetterebbe la parte più onerosa è quella relativa al calcolo della formula 4.14.

L'ultimo confronto di questo capitolo riguarda il rapporto fra prestazioni e prezzo. Il rapporto prestazioni/prezzo è una delle caratteristiche più importanti nel campo industriale, e permette di valutare la convenienza nell'investire in un sistema di calcolo basato su una GPU, piuttosto che su una CPU. Tuttavia questo rapporto è strettamente legato, sia al software realizzato, sia all'hardware utilizzato. Il seguente test va quindi considerato valido solo nel contesto di questo lavoro di tesi. In Tabella 4.4 si mostrano i dati riguardanti

Tabella 4.4: Confronto del rapporto prestazioni/prezzo fra CPU e GPU

	CPU €	$\frac{speedup}{prezzo} \times 10^3$	GPU €	$\frac{speedup}{prezzo} \times 10^3$	rapp. $\frac{GPU}{CPU}$
min.	150.00	$\frac{1}{150} \times 10^3 = 6.66$	60	$\frac{26}{60} \times 10^3 = 433.33$	65.06
max.	150.00	$\frac{1}{150} \times 10^3 = 6.66$	60	$\frac{41}{60} \times 10^3 = 683.33$	102.6

l'hardware utilizzato. I prezzi indicati sono ricavati dalla media dei prezzi raccolti dai listini di più rivenditori. Questi risultati evidenziano un vantaggio economico derivante dall'utilizzo della GPU, di almeno 65 volte, rispetto a quello ricavato da una CPU. Tuttavia questo vantaggio è solo una stima della reale convenienza economica che si ha nell'utilizzo di una GPU, poichè non sono stati considerati i costi della memoria, componente indispensabile all'interno del sistema.

Capitolo 5

Conclusioni e Sviluppi Futuri

Con il lavoro svolto in questa tesi si sono illustrate le caratteristiche principali delle GPU moderne e della tecnologia CUDA, descrivendone in dettaglio sia l'architettura, sia il modello di programmazione. Successivamente, sfruttando questa tecnologia, si è realizzata un'implementazione del modello di Luhar e Britter per la dispersione di particelle in atmosfera. In questo processo sono stati realizzati dei kernel in modo tale che questi sfruttassero a fondo l'architettura parallela delle GPU. Infine si è effettuato il confronto fra l'esecuzione dell'algoritmo da parte della CPU e della GPU. Il confronto ha evidenziato un notevole vantaggio, in termini di prestazioni, dell'architettura CUDA. Infatti, sull'hardware utilizzato, questa si è dimostrata capace di ridurre i tempi di elaborazione da 26 fino a 41 e più volte. Lo stesso confronto stima un significativo vantaggio economico, derivante dall'uso della tecnologia CUDA nel software realizzato. Per l'elaboratore su cui si sono eseguiti i test, questo vantaggio è stato stimato confrontando il rapporto prestazioni/prezzo della GPU con il rapporto prestazioni/prezzo della CPU. Si arriva così ad ottenere una convenienza economica della tecnologia CUDA

che va da 65 fino a 100 e più volte quella della tecnologia convenzionale, ovvero basata sul solo utilizzo della CPU . Questi aspetti costituiscono proprio lo spunto più interessante per un eventuale proseguimento del lavoro di tesi. Infatti, realizzando un'implementazione capace di sfruttare contemporaneamente le capacità di calcolo di più GPU, si potrebbe incrementare ulteriormente la convenienza economica. Ancor più interessante, potrebbe essere lo sviluppo di un'implementazione del modello che sfrutti congiuntamente sia la tecnologia classica, sia la tecnologia CUDA, per ottenere speed up ancora più significativi.

Appendice A

Codici sorgente

A.1 sorgenti MATLAB

Listing A.1: LB.m

```
function LB(zzi,NP,dt,dx,dz,TFin)
Nz=fix(1/dz);
Mz=fix(TFin/dx);
ZSave=zeros(Nz,Mz);
z=zzi*ones(NP,1);
w=InitW(z,NP);
N=TFin/dt;
k=0;
tic
for i=1:N
   [w, z]=Pa(w, z, dt);
   if ( mod(i, N/Mz) == 1 )
```

```
            k=k+1;
            ZZ=histc(z, linspace(0,1,Nz+1))/(NP/Nz);
            ZSave(:,k)=ZZ(1:Nz);
  end
end
toc
Clev=[0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2 3 4 5 6 7];
figure(1), clf, cla
contour(linspace(0,TFin,Mz),linspace(0,1,Nz),ZSave,Clev)
set(gcf, 'Color', [1 1 1])
view(2), shg
end
```

Listing A.2: initW.m

```
function w=InitW(z,NP)

[A,B,w2,w3,wa,wb,F,dw2dz,...
dw3dz,dwadz,dwbdz,dAdz,dBdz]=GetPar(z);

wa=normrnd(+wa.*ones(NP,1), wa);
wb=normrnd(-wb.*ones(NP,1), wb);

NA=fix(NP*0.01*fix(100*A(1)));

S=randperm(NP);
w=[wa(S(1:NA)); wb(S(NA+1:NP))];
```

```
end
```

Listing A.3: Pa.m

```
function [w, z]=Pa(w, z, dt)

[A,B,w2,w3,wa,wb,F,dw2dz,...
dw3dz,dwadz,dwbdz,dAdz,dBdz]=GetPar(z);

Pa=NormPDF(w, wa, wa);
Pb=NormPDF(w,-wb, wb);

phi=-0.5*(A.*dwadz+wa.*dAdz).*erf((w-wa)./ ...
    (sqrt(2.0)*wa)) + wa.*(A.*dwadz.*...
    ((w./wa).^2+1.0)+wa.*dAdz).*Pa + ...
    0.5*(B.*dwbdz+wb.*dBdz).*erf((w+wb)./ ...
    (sqrt(2.0)*wb)) + wb.*(B.*dwbdz.* ...
    ((w./wb).^2+1.0)+wb.*dBdz).*Pb;
Q=A.*(w-wa).*Pa./(wa.^2) + ...
  B.*(w+wb).*Pb./(wb.^2);
tau=w2./(1.5-1.2*z.^(1/3));
P=max(1.0e-10, A.*Pa + B.*Pb);

w=w + ((phi-w2.*Q./tau)./P)*dt + ...
  sqrt(2.0*w2*dt./tau).*randn(size(w));
```

```
w=min(3.0, w); w=max(-3.0, w);


z=z + w.*dt;


J=find(z<0 | z>1 );
while ( not(isempty(J)) )
    I=find(z<0);
    if ( not(isempty(I)) )
        w(I)=-w(I);
        z(I)=-z(I);
    end
    I=find(z>1);
    if ( not(isempty(I)) )
        w(I)=-w(I);
        z(I)=2.0-z(I);
    end
    J=find(z<0 | z>1 );
end


end
```

Listing A.4: getPar.m

```
function [A,B,w2,w3,wa,wb,F,dw2dz,...
dw3dz,dwadz,dwbdz,dAdz,dBdz]=GetPar(z);


w2=(1.1-1.1*(4*z-1.2)./...
```

```
(2+abs(z-0.3)).^2).*(z.*(1-z)).^(2/3);
w3=0.8*w2.^1.5;


wb=(sqrt(w3.^2+8*w2.^3)-w3)./(4*w2);
wa=0.5*w2./wb;
A=wb./(wa+wb);
B=wa./(wa+wb);


F=4.0*w2.*wb + w3;
dw2dz=(-4.4./(2+abs(z-0.3)).^2+2*(4.4*z-1.32)./...
(2+abs(z-0.3)).^3.*sign(z-0.3)).*...
(z.*(1-z)).^(2/3)+2/3*(1.1-(4.4*z-1.32)./  ...
(2+abs(z-0.3)).^2)./(z.*(1-z)).^(1/3).*(1-2*z);
dw3dz=1.2*dw2dz.*(w2.^0.5);


dwbdz=-(wb.*dw3dz+dw2dz.*(wb.*F./w2-3.0*w2))./F;
dwadz=0.5*dw2dz./wb-wa.*dwbdz./wb;
dAdz=zeros(size(A));
dBdz=-dAdz;


end
```

Listing A.5: NormPDF.m

```
function y=NormPDF(x, mu, sigma)


y = exp(-0.5*((x-mu)./sigma).^2)./(sqrt(2*pi).*sigma);
```

```
end
```

A.2 sorgenti CUDA

Listing A.6: LB.cu

```
#include "myCudaLib.h"


int main(int argc,char **argv)
{
  if(argc==8)
  {
    FILE *fp;
    float TFin=atof(argv[6]),
          dt=atof(argv[3]),
          dx=atof(argv[4]),
          dz=atof(argv[5]),
          zzi=atof(argv[1]);
    int Nz=fix(1.0f/dz),
        Mz=fix(TFin/dx),
        NP=atoi(argv[2]);
    int threadPerBlock=196;
    int nBlock = NP/threadPerBlock +
          (NP%threadPerBlock==0? 0:1);
    float *z_d,*w2_d,*w3_d,*wb_d,*wa_d,*a_d,
```

```
    *b_d,*f_d,*dw2dz_d,*dw3dz_d,*dwbdz_d,
    *dwadz_d,*dadz_d,*dbdz_d,*phi_d,*q_d,
    *tau_d,*p_d,*w_d,*pa_d,*pb_d,*zz_d,
    *randVec_d,*normRand1_d,*normRand2_d,
    *zSave_d,*waFin_d,*wbFin_d,*zSave_h;
float N=TFin/dt;
int Nrand=(int)N*NP;

/*series of cudaMalloc*/
int size=NP*sizeof(float);
cudaMalloc((void**)&z_d,size);
cudaMalloc((void**)&w2_d,size);
cudaMalloc((void**)&w3_d,size);
cudaMalloc((void**)&wb_d,size);
cudaMalloc((void**)&wa_d,size);
cudaMalloc((void**)&a_d,size);
cudaMalloc((void**)&dw2dz_d,size);
cudaMalloc((void**)&b_d,size);
cudaMalloc((void**)&f_d,size);
cudaMalloc((void**)&dw3dz_d,size);
cudaMalloc((void**)&dwbdz_d,size);
cudaMalloc((void**)&dwadz_d,size);
cudaMalloc((void**)&dadz_d,size);
cudaMalloc((void**)&dbdz_d,size);
cudaMalloc((void**)&q_d,size);
```

```
cudaMalloc((void**)&phi_d, size);
cudaMalloc((void**)&tau_d, size);
cudaMalloc((void**)&p_d, size);
cudaMalloc((void**)&w_d, size);
cudaMalloc((void**)&pa_d, size);
cudaMalloc((void**)&pb_d, size);
cudaMalloc((void**)&randVec_d, Nrand*sizeof(float));
cudaMalloc((void**)&normRand1_d, size);
cudaMalloc((void**)&normRand2_d, size);
cudaMalloc((void**)&waFin_d, size);
cudaMalloc((void**)&wbFin_d, size);
cudaMalloc((void**)&zSave_d, Nz*Mz*sizeof(float));
cudaMalloc((void**)&zz_d, Nz*sizeof(float));
checkCUDAError("CUDAMalloc");
/*calc. of randoms*/
srand(time(NULL));
InitRandom<<<1,1>>>(normRand1_d,NP,rand());
gasDev<<<1,1>>>(normRand1_d,NP,1);
checkCUDAError("randInit 1");
InitRandom<<<1,1>>>(normRand2_d,NP,rand());
gasDev<<<1,1>>>(normRand2_d,NP,1);
checkCUDAError("randInit 2");

/*series of init*/
initArray<<<nBlock,threadPerBlock>>>
```

```
    (z_d,NP,zzi);
checkCUDAError(``init Z'');
initArray<<<nBlock,threadPerBlock>>>
    (dadz_d,NP,0.0f);
checkCUDAError(``init DADZ'');
initArray<<<nBlock,threadPerBlock>>>
    (dbdz_d,NP,0.0f);
checkCUDAError(``init DBDZ'');
initArray<<<2,400>>>(zSave_d,Nz*Mz,0.0f);
initW<<<nBlock,threadPerBlock>>>
    (z_d,a_d,w2_d,w3_d,wa_d,wb_d,w_d,NP,normRand1_d,
     normRand2_d,waFin_d,wbFin_d,NP);
checkCUDAError(``init W'');

unsigned int timer;
cutCreateTimer(&timer);
cutStartTimer(timer);
getRandomNum<<<nBlock,threadPerBlock>>>
  (randVec_d,Nrand,NP,rand());
gasDev<<<nBlock,threadPerBlock>>>
  (randVec_d,Nrand,NP);
checkCUDAError("randInit_3");
int k=-1;
for(int i=1;i<=N;i++)
{
```

```
    Pa<<<nBlock,threadPerBlock>>>
      (w_d,z_d,0.01,pa_d,pb_d,phi_d,q_d,tau_d,
       p_d,a_d,dwadz_d,wa_d,dadz_d,b_d,dwbdz_d,
       wb_d,dbdz_d,w2_d,w3_d,f_d,dw2dz_d,dw3dz_d
       ,randVec_d,i-1,NP);
    checkCUDAError("Pa");
    int num=(int)(N/Mz);
    if((i%num)==1)
    {
      k++;
      histc<<<1,Nz>>>(z_d,zz_d,1.0f/Nz,NP,Nz);
      cudaThreadSynchronize();
      checkCUDAError("histc");
      divideArray<<<1,Nz>>>
        (zz_d,((float)NP/(float)Nz),Nz);
      checkCUDAError("divideArray");
      copyArrayInMat<<<1,Nz>>>(zSave_d,zz_d,k,Nz,Mz);
      checkCUDAError("copyArrayInMat");
    }
  }
  cutStopTimer(timer);
  zSave_h=(float*)malloc(Nz*Mz*sizeof(float));
  cudaMemcpy(zSave_h,zSave_d,
             Nz*Mz*sizeof(float),
             cudaMemcpyDeviceToHost);
```

```
    fp=fopen(argv[7],"w");
    fprintf(fp,"%f\n",TFin);
    fprintf(fp,"%f\n",dx);
    fprintf(fp,"%f\n",dz);
    for(int i=0;i<Nz;i++)
    {
      for(int j=0;j<Mz;j++)
      {
        fprintf(fp,``%f '',zSave_h[i*Mz+j]);
      }
      fprintf(fp,"\n");
    }
    fclose(fp);
    printf(``execution time %f ms\n'',
           cutGetTimerValue(timer));
    return 0;
  }
  else
  {
    printf(``usage: %s <zzi> <NP> <dt> <dx>
    <dz> <TFin> <outputFilename>\n'',argv[0]);
    printf(``\nwhere:\n <zzi> is the source height\n'');
    printf(`` <NP> is the number of particles\n'');
    printf(`` <dt> <dx> <dz> are
    deltaT deltaX and deltaZ parameters\n'');
```

```
    printf(`` <Tfin> is the final time\n'');
    printf(`` <outputFilename> is the name
    of the output file\n'');
    printf(``\n an example can be:\n %s 0.49 2000
    0.01 0.1 0.05 4 file.txt\n'',argv[0]);
    return -1;
  }
}
```

Listing A.7: myCUDALib.h

```
#include <stdio.h>
#include <cutil.h>
#include "MersenneTwister.h"
#include "MersenneTwister_kernel.cu"
#include <math.h>

__global__ void InitRandom( float *vec, int N,
                            unsigned long jran)
{
    unsigned long ia=4096;
    unsigned long im=714025;
    unsigned long ic=150889;

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

```
    for (int j=0; j<N; j++) {
     jran=(jran*ia+ic) % im;
     vec[j]=(float) jran / (float) im;
    }
}

__global__ void getRandomNum(float *vec,int N, int NP,
                             unsigned long jran)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<NP)
  {
        long idum = jran*-1 + index;
        int j;
        long k;
        unsigned long iy=0;
        unsigned long iv[NTAB];
    float temp;
        if (idum<=0 || !iy) {
        if (-idum < 1L) idum=1L;
        else idum = -idum;
        for (j=NTAB+7;j>=0;j--) {
            k=idum/IQ;
            idum=IA*(idum-k*IQ)-IR*k;
            if (idum < 0L) idum += IM;
```

```
            if (j < NTAB) iv[j] = idum;
        }
        iy=iv[0];
    }
        for ( int jj=0; jj<N; jj += NP )
        {
            k=idum/IQ;
            idum=IA*(idum-k*IQ)-IR*k;
        if (idum < 0) idum += IM;
        j=iy/NDIV;
        iy=iv[j];
        iv[j] = idum;
        if ((temp=AM*iy) > RNMX)
                vec[index+jj]= RNMX;
        else
                vec[index+jj]= temp;
    }
  }
}


#define PI 3.14159265358979f


__device__ void MyBoxMuller(float& u1,float& u2)
{
```

```
    float    r = sqrtf(-2.0f * logf(u1));
    float phi = 2 * PI * u2;
    u1 = r * __cosf(phi);
    u2 = r * __sinf(phi);

}

__global__ void gasDev(float *vec, int N, int NP)
{
int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<NP)
  {
    for ( int j=0; j<N-NP; j += 2*NP )
    MyBoxMuller(vec[index+j], vec[index+j+NP]);
  }
}

__device__ void getw2(float *z, float *w2, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    w2[index]=(1.1-1.1*(4*z[index]-1.2)/
    (pow((2+abs(z[index]-0.3)),2)))*
    (pow((z[index]*(1-z[index])),2.0f/3.0f));
```

```
  }
}

__device__ void getw3(float *w2, float *w3, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    w3[index]=0.8*pow(w2[index],1.5f);
  }
}

__device__ void getwb
  (float *w2, float *w3, float *wb, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    wb[index]=(sqrt(pow(w3[index],2)+
    8*pow(w2[index],3))-w3[index])/(4*w2[index]);
  }
}

__device__ void getwa
  (float *w2, float *wb, float *wa, int N)
```

```
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    wa[index]=0.5*w2[index]/wb[index];
  }
}

__device__ void geta
  (float *wa, float *wb, float *a, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    a[index]=wb[index]/(wa[index]+wb[index]);
  }
}

__device__ void getb
  (float *wa, float *wb, float *b, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    b[index]=wa[index]/(wa[index]+wb[index]);
```

```
  }
}

__device__ void getf
(float *wb, float *w2, float *w3, float *f, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    f[index]=4*w2[index]*wb[index]+w3[index];
  }
}

__device__ void getdw2dz1
(float *z, float *dw2dz, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    dw2dz[index]=(-4.4/pow((2+abs(z[index]-0.3)),2)
    +2*(4.4*z[index]-1.32)/pow((2+abs(z[index]-0.3)),3)*
    ((z[index]-0.3)>=0?1:-1));
  }
}
```

```
__device__ void getdw2dz2(float *z, float *dw2dz, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    dw2dz[index]=dw2dz[index]
    *pow((z[index]*(1-z[index])),(2.0f/3.0f))
    +2.0f/3.0f*(1.1-(4.4*z[index]-1.32)/
    pow((2+abs(z[index]-0.3)),2))
    /pow((z[index]*(1-z[index])),(1.0f/3.0f))*
    (1-2*z[index]);
  }
}

__device__ void getdw3dz
(float *dw2dz, float *w2, float *dw3dz, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    dw3dz[index]=1.2f*dw2dz[index]*pow(w2[index],0.5f);
  }
}
```

```
__device__ void getdwbdz
  (float *wb, float *dw2dz, float *dw3dz, float *w2,
   float *f, float *dwbdz, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    dwbdz[index]=-(wb[index]*dw3dz[index]+
    dw2dz[index]*(wb[index]*f[index]/
    w2[index]-3.0f*w2[index]))/f[index];
  }
}



__device__ void getdwadz
  (float *dw2dz, float *wa, float *wb,
   float *dwbdz, float *dwadz, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    dwadz[index]=0.5f*dw2dz[index]/
    wb[index]-wa[index]*dwbdz[index]/wb[index];
  }
}
```

```
__device__ void getPhi
  (float *a, float *dwadz, float *wa, float *dadz,
   float *w, float *pa, float *b, float *dwbdz,
   float *wb, float *dbdz, float *pb, float *phi,
   int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
     phi[index]=-0.5f*(a[index]*dwadz[index]+
     wa[index]*dadz[index])*
     erf((w[index]-wa[index])
    /(sqrt(2.0f)*wa[index])) +wa[index]*
    (a[index]*dwadz[index]*
    (pow((w[index]/wa[index]),2)+1.0f)+
    wa[index]*dadz[index])*pa[index]
    +0.5f*(b[index]*dwbdz[index]+wb[index]*
    dbdz[index])*erf((w[index]+wb[index])/
    (sqrt(2.0)*wb[index]))+wb[index]*(b[index]*
    dwbdz[index]*(pow((w[index]/wb[index]),2)+
    1.0f)+wb[index]*dbdz[index])*pb[index]
    ;
  }
```

```
}



__device__ void getQ
  (float *a, float *w, float *wa, float *pa,
   float *b, float *wb, float *pb, float *q, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    q[index]=a[index]*(w[index]-wa[index])*
    pa[index]/pow(wa[index],2)+b[index]*
    (w[index]+wb[index])*pb[index]/pow(wb[index],2);
  }
}



__device__ void getTau
  (float *w2, float *z, float *tau, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    tau[index]=w2[index]/(1.5-1.2*
    pow(z[index],1.0f/3.0f));
```

```
  }
}



__device__ void getP
  (float *a, float *pa, float *b,
   float *pb, float *p, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    p[index]=max(1.0e-10,a[index]*
    pa[index]+b[index]*pb[index]);
  }
}



__device__ void getW
  (float *phi, float *w2, float *q,
   float *tau, float *p, float *randNArray,
   float dt, float *w, int i,int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
```

```
    w[index]=w[index]+((phi[index]-w2[index]*
    q[index]/tau[index])/p[index])*dt +
    sqrt(2.0f*w2[index]*dt/tau[index])*
    randNArray[i*N+index];
    w[index]=min(3.0f, w[index]);
    w[index]=max(-3.0f, w[index]);
  }
}



__device__ void getZ
  (float *w, float dt, float *z, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
   z[index]=z[index]+w[index]*dt;
  }
}



__device__ void getWInit
(float *wa, float *wb, int NA, float *w, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
```

```
  if(index<N && index<NA)
  {
   w[index]=wa[index];
  }
  else
  {
    if(index<N)
    {
     w[index]=wb[index];
    }
  }
}



__device__ void normPDF
(float *x, float mu, float sigma, float *y, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    y[index]=exp(-0.5*pow(((x[index]-mu)/
      sigma),2))/(sqrt(2*M_PI)*sigma);
  }
}
```

```
__device__ void normRnd
(float *result, float mu, float sigma,
 float *randnVec, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    result[index]=randnVec[index]*sigma+mu;
  }
}



__device__ void normRnd2a
(float *out, float *in, float *randnVec, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    out[index]=randnVec[index]*in[index]+in[index];
  }
}



__device__ void normRnd2b
```

```
  (float *out, float *in, float *randnVec, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    out[index]=randnVec[index]*in[index]-in[index];
  }
}



__global__ void initArray
(float *array, int N, float initValue)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    array[index]=initValue;
  }
}



__device__ void checkBorder
(float *z, float *w, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
```

```
  if(index<N)
  {
    while(z[index]<0 || z[index]>1)
    {
        if(z[index]<0)
        {
            z[index]=-z[index];
            w[index]=-w[index];
        }
        if(z[index]>1)
        {
            z[index]=2-z[index];
            w[index]=-w[index];
        }
    }
  }
}


__device__ void getPar
  (float *z, float *a, float *b, float *w2,
   float *w3, float *wa, float *wb,
   float *F, float *dw2dz, float *dw3dz,
   float *dwadz, float *dwbdz,
   float *dAdz, float *dBdz , int N)
{
```

```
  getw2(z,w2,N);
  getw3(w2,w3,N);
  getwb(w2,w3,wb,N);
  getwa(w2,wb,wa,N);
  geta(wa,wb,a,N);
  getb(wa,wb,b,N);
  getf(wb,w2,w3,F,N);
  getdw2dz1(z,dw2dz,N);
  getdw2dz2(z,dw2dz,N);
  getdw3dz(dw2dz,w2,dw3dz,N);
  getdwbdz(wb,dw2dz,dw3dz,w2,F,dwbdz,N);
  getdwadz(dw2dz,wa,wb,dwbdz,dwadz,N);
}



__global__ void initW
  (float *z, float *a,float *w2, float *w3, float *wa,
  float *wb, float *w, int NP, float *randnVec1,
  float *randnVec2, float *waFin, float *wbFin, int N)
{
    getw2(z,w2,N);
    getw3(w2,w3,N);
    getwb(w2,w3,wb,N);
    getwa(w2,wb,wa,N);
    geta(wa,wb,a,N);
```

```
    normRnd2a(waFin,wa,randnVec1,N);
    normRnd2b(wbFin,wb,randnVec2,N);
    int NA=(int)(N*0.01*((int)(100*a[0])));
    getWInit(waFin,wbFin,NA,w,N);
}



__global__ void Pa
  (float *w, float *z, float dt, float *pa, float *pb,
   float *phi, float *q, float *tau, float *p,
   float *a, float *dwadz, float *wa, float *dadz,
   float *b, float *dwbdz, float *wb, float *dbdz,
   float *w2, float *w3, float *f, float *dw2dz,
   float *dw3dz, float *randnVec, int i, int N)
{
  getPar(z,a,b,w2,w3,wa,wb,f,dw2dz,
   dw3dz,dwadz,dwbdz,dadz,dbdz,N);
  normPDF(w,wa[0],wa[0],pa,N);
  normPDF(w,-wb[0],wb[0],pb,N);
  getPhi(a,dwadz,wa,dadz,w,pa,b,
    dwbdz,wb,dbdz,pb,phi,N);
  getQ(a,w,wa,pa,b,wb,pb,q,N);
  getTau(w2,z,tau,N);
  getP(a,pa,b,pb,p,N);
```

```
  getW(phi,w2,q,tau,p,randnVec,dt,w,i,N);
  getZ(w,dt,z,N);
  checkBorder(z,w,N);
  __syncthreads();
}



__global__ void histc
(float *in, float*out,
float delta, int Nin, int Nout)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  extern __shared__ int count[];
  if(index<Nout)
  {
    count[0]=0;
    for(int i=0;i<Nin;i++)
    {
      if(in[i]>=(delta*index) &&
        in[i]<(delta*(index+1)))
      {
        count[0]++;
      }
    }
    out[index]=count[0];
```

```
  }
}



__global__ void divideArray
  (float *array, float divisor, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<N)
  {
    array[index]/=divisor;
  }
}



__global__ void copyArrayInMat
(float *mat, float *array, int col, int M, int N)
{
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  if(index<M)
  {
    mat[index*N+col]=array[index];
  }
}
```

```
void checkCUDAError(const char *msg)
{
    cudaError_t err = cudaGetLastError();
    if( err != cudaSuccess )
    {
        fprintf(stderr, ''Cuda error: %s: %s.\n'',
                msg, cudaGetErrorString( err) );
        exit(EXIT_FAILURE);
    }
}

int fix(float num)
{
  return (int)num;
}
```

Listing A.8: CUDAscript.m

```
function CUDAscript(inputFilename)
if ( not(nargin) )
    inputFilename='file';
end
OK=exist(inputFilename, 'file');
if ( not(OK) )
    error(strcat(['Il file ''',
    inputFilename, ''' non esiste']))
end
```

```
fid=fopen(inputFilename);
TFin=fscanf(fid, '%f', 1);
dx=fscanf(fid, '%f', 1);
dz=fscanf(fid, '%f', 1);
Nz=fix(1/dz);
Mz=fix(TFin/dx);

ZSave=fscanf(fid, '%f', Mz*Nz);
fclose(fid);

ZSave=transpose(reshape(ZSave, Mz, Nz));

Clev=[0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2 3 4 5 6 7];
figure(1); clf, cla
contour(linspace(0,TFin,Mz),linspace(0,1,Nz),ZSave,Clev)
shg
end
```

Bibliografia

[1] Arnold. L. *Sthocastic Differential Equations: Theory and Application.*. John Wisley and Sons, 228.

[2] Baerentsen J. H., Berkowicz. R. *Monte Carlo simulation of plume dispersion in the convective boundary layer*. Atmos. Environ., 18 pp. 701-712.

[3] Brent. Richard P. *Uniform Random Number Generator for Vector and Parallel Computers*. Report TR-CS-92-02.

[4] Chandrasekhar. S. *Sthocastic problem in physics and astronomy*. Rev. Mod. Phys., 15 pp. 1-89.

[5] Che Shuai , Boyer Michael, Meng Jiayun, Tarajan David, Sheaffer W. Jeremy, Skadron Kevin. *A performance study of general-purpose applications on graphics processor using CUDA*. Journal of Parallel and Distributed Computing, 68(10) pp. 1370-1380.

[6] Cummins G., Adams R. , Newell. T. *Scientific computation through a GPU*. Southeastcon 2008 IEEE, pp. 244-246.

[7] de Baas A. F., Throen. I. *A sthocastic equation for diffusion in inhomogeneous conditions*. Phiysica Scripta, 40 pp. 64-72.

[8] Durbin. P. A. *Sthocastic differential equations and turbulent dispersion*. NASA Reference Publication, 1103(69).

[9] Durbin. P. A. *Comments on papers by Wilson et al. (1981) and Legg and Raupach (1982)*. Bound.-Layer Meteor., 29 pp. 409-411.

[10] Edgar Richard. *Advanced CUDA*. GPU workshop.

[11] Gardiner. C. W. *Handbook of Sthocastic Methods*. Springer-Verlag, 442.

[12] Gifford. F. A. *Horizontal diffusion in atmosphere: A Lagrangian-dynamical theory*. Atmos. Environ., 15 pp. 505-512.

[13] Halfhill. Tom R. *Parallel processing with CUDA*.Microprocessor 2008.

[14] Hall. C. D. *The simulation of particle motion in the atmosphere by a numerical random walk model*. Quart J. R. Soc., 101 pp. 235-244.

[15] Hanna. S. R. *Some Statistics of Lagrangian and Eulerian wind fluctuations*. J. Appl. Meteor., 20 pp. 242-249.

[16] Kaeli R. David, Leeser Miriam.*Special Issue: General-purpose processing using graphics processing units*. Journal of Parallel and Distributed Computing, 68(10) pp. 1305-1306.

[17] Kirk B. David. *10 important problems in computer architecture*. 2008 NVIDIA Chief Scientist slides.

[18] Kolgomorov. A. N. *Uber die analytischen Methoden in der Wahrscheinlichkeitsrechung (On anatytical methods in probability theory).* Math. Ann, 104 pp. 415-458.

[19] Legg B. J., Raupah. M. R. *Markov-chain simulation of particle diffusion in inhomogeneous flows: The mean drift velocity induced by a gradient in Eulerian velocity cariance.* Bound.-Layer Meteor, 24 pp. 3-13.

[20] Ley. A. J. *A random walk simulation of two-dimensional turbulent diffusion in the neutral surface layer.* Atmos. Environ., 16 pp. 2799-2808.

[21] Lin C. C., Reid. W. H. *Turbulent flow, theoretical aspects.* Hand Physik, VIII/2 pp. 438-523.

[22] Lindholm E.,Nickolls J., Oberman S., MontrymJ. . *NVIDIA Tesla: a unified graphics and computing architecture.*IEEE Micro, 28(2) pp. 39-55.

[23] Luebke David.*GPU architecture and implications.* 2007 NVIDIA research slides.

[24] Luhar Ashok K. , Britter 1989. Rex E. *A random walk model for dispersion in inhomogeneous turbulence in a convective boundarylayer.* Atmospheric Environment, 23(9) pp. 1911-1924.

[25] Macedonia Michael . *The GPU enters computing's mainstream.* Computer, 36(10) pp.106-108.

[26] Mathworks. *Matlab 7 Desktop Tools and Development Environment.* Mathworks Inc. 2007.

[27] Matsumoto Makoto , Nishimura Takuji. *Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator.* ACM Transaction on Modeling and Computer Simulation (TOMACS), 8(1) pp. 3-30.

[28] NVIDIA Corporation 2008. *CUDA Programming Guide 2.0.*

[29] NVIDIA Corporation 2008. *CUDA Reference Manual 2.0.*

[30] Obukhov. A. M. *Description of turbulence in terms of Lagrangian variables* . Adv. Geophys., 6 pp. 113-116.

[31] Owens J.D., Houston M., Luebke D., Green S., Stone J.E., Phillips. J.C. *GPU computing.* Proceedings of the IEEE, 95(5) pp. 879-899.

[32] Podlozhnyuk Victor. *Parallel Mersenne Twister.*

[33] Sawford. B. L. *The basis for, and some limitations of the Langevin equation in atmospheric relative diffusion modelling.* Atmos. Environ., 18 pp. 2405-2411.

[34] Sawford. B. L. *Generalized random forcing in random-walk turbulent diffusion dispersion.* Phys. Fluids, A 3 pp. 1577-1586.

[35] Schenk Olaf, Christen Matthias, Burkhart Helmar. *Algorithmic performance studies on graphics processing units.* Journal of Parallel and Distributed Computing, 68(10) pp. 1360-1369.

[36] Strzodka Robert, Doggett Michael, Kolb Andreas. *Scientific computing for simulations on programmable graphics hardware.* Simulation Modelling Practice and Theory, 13(8) pp. 667-680.

[37] Thomson. D. J. *Random walk modelling of diffusion in inhomogeneous turbulance*. The Quartely Journal of the Royal Metereological Society, 110(446) pp. 1107-1120.

[38] Uhlenbeck G. E., Ornstein. L. S. *On theory of the Brownian Motion*. Phys. Rev., 36(5) pp. 823-841.

[39] van Dop H., Nieuwstadt. F. T. M. *Random walk models for particle displacements in inhomogeneous unsteady turbulent flows*. Phys. Fluids, 28(6) pp. 1639-1653.

[40] Wilson J. D., Flesch. T. K. *The well-mixed constrain applied to random flight models with reflecting boundaries*. Proceedings of the 10th Symp. on Turbulence and Diffusion. Portland, Oregon.. Amer Meteor. Soc., pp. 222-225.

[41] Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt. *Dynamic warp formation and scheduling for efficient gpu control flow*. Micro '07: Proceedings of the 40*th* Annual IEEE/ACM International Symposium on Microarchitecture, pp. 407-420.

[42] Weil. J. C. *A diagnosis of the asymmmetry in top-down and bottom-up diffusion using a Lagrangian sthocastic model*. J. Atmos. Sci., 47 pp. 501-515.

[43] Wolley Cliff. *Efficient Data Parallel Computing on GPUs*. SIGGRAPH 2004 GPGPU Presentation.

[44] Won-Ki Jeong, Ross Whitaker. *High performance computing on the GPU: NVIDIA G80 and CUDA*. SCI Institute, University of Utah slides.

Ringraziamenti

Giunto al termine della tesi e del mio percorso di studi non posso non ringraziare tutte le persone che in questi anni mi sono state vicine nei momenti belli e soprattutto in quelli brutti.

Il mio primo GRAZIE va alla mia famiglia, ai miei genitori che sono sempre stati dalla mia parte e mi hanno permesso di arrivare dove sono ora, a mio fratello che mi ha aiutato a distrarmi nei momenti di stress, ai miei nonni che mi hanno sempre sostenuto e ai mie zii che sono che sono sempre stati fieri di me.

Un GRAZIE particolare lo devo ai miei relatori, al Prof. Giulio Giunta, vero modello di professionalità e competenza, che mi ha permesso di apprendere un'argomento tanto innovativo quanto entusiasmante e al Prof. Angelo Riccio, che mi ha introdotto gradualmente nel mondo delle scienze ambientali e mi ha trasmesso parte della sua passione.

Un GRAZIE speciale al mio "team": Mina e Ary. Grazie per tutte le indimenticabili giornate sia belle che brutte che abbiamo passato insieme. GRAZIE per tutte i sorrisi che mi avete regalato, grazie a voi adesso sono certo che ridere fa bene alla salute. GRAZIE per avermi reso meno timido e per avermi fatto conoscere tanti amici come Mariangela, Antonio, Carlo e tutti gli altri.

GRAZIE a tutti gli studenti che ho incontrato alla Parthenope, soprattutto a Mariangela, Antonio, Carlo, Salvatore, Francesco ed Enzo, coi quali ho

condiviso alcuni degli attimi più belli e più memorabili della mia vita. Con questo non escludo dai miei ringraziamenti tutte le altre persone che ho anche solo incrociato lungo il mio percorso di cui non riporto il nome solo per mancanza di tempo e di spazio.

Per ultimo, ma non per questo meno importante, un GRAZIE a tutti i membri del laboratorio di modellistica numerica e calcolo parallelo "LMNCP" coi quali ho condiviso gli ultimi momenti della mia carriera universitaria, in particolare un GRAZIE al Prof. Montella e a Giuseppe Agrillo che mi hanno aiutato e consigliato nei momenti di difficoltà.

I miei ringraziamenti sono rivolti a tutti voi, perché ognuno di voi a modo suo ha contribuito al raggiungimento di questa importante tappa.

A tutti voi GRAZIE.

Questa tesi è anche vostra.