UNIVERSITÀ DEGLI STUDI DI NAPOLI
"PARTHENOPE"

FACOLTÀ DI SCIENZE E TECNOLOGIE

Corso di Laurea in Informatica

Elaborato di Laurea

Simulazioni Stocastiche di Moti Collettivi "Fish-Schooling" con GPU

Relatore:

Prof. Giulio Giunta

Presentata da:

Rita Mele

LI/476

Anno Accademico 2007-2008

Alla mia famiglia

senza la quale nulla sarebbe stato possibile

Al mio grande amore

arcobaleno della mia vita

Agli amici

che mi hanno sempre sostenuto

"La fortuna aiuta le menti preparate"

(Zadig, Voltaire)

[1]Powered By Latex

Abstract

In questo lavoro di tesi è stato analizzato e implementato, in ambiente GPU, un modello numerico finalizzato all'investigazione di fenomeni di moto collettivo nelle scienze naturali e biologiche, in particolare mirato allo studio dell'auto-organizzazione e del movimento di un banco di pesci. Molti sistemi naturali sono composti da un elevato numero di agenti in comunicazione locale, la cui azione individuale produce un comportamento macroscopico organizzato. Questo fenomeno si riscontra, per esempio, nel moto collettivo di stormi di uccelli o banchi di pesci, nello sviluppo di organismi biologici multicellulari e nel funzionamento di reti genetiche. In tutti i casi si tratta di sistemi ove un comportamento emergente di auto-organizzazione scaturisce a partire da semplici leggi di coordinazione locale tra le singole unità.

Sistemi di questo tipo, in cui si mira all'osservazione di un comportamento globale a partire da un insieme di organismi, sono tutti descritti tramite modelli matematici deterministici o stocastici.

Il *Fish-Schooling* è un modello di tipo stocastico che consente di effettuare simulazioni numeriche di elevata affidabilità, includendo una forte componente non deterministica e rappresenta uno strumento effettivo per la comprensione del comportamento degli organismi coinvolti e delle strutture dinamiche che ne derivano.

Nel modello si definisce quantitativamente la tendenza da parte di ciascun pesce ad allineare la propria posizione e il proprio orientamento, evitando le collisioni con gli altri individui che si muovono nel banco.

La ripetizione delle simulazioni al variare delle condizioni iniziali permette di estrarre quantità statistiche che caratterizzano il comportamento organizza-

to.

In tale contesto, è importante sviluppare una implementazione molto efficiente del modello, al fine di dimostrare i tempi di esecuzione della singola realizzazione stocastica o di un insieme di realizzazioni con scelta omogenea dei parametri. A tale scopo è stato utilizzato, come ambiente di esecuzione, una GPU, cioè un processore grafico. L'implementazione viene descritta con riferimenti approfonditi all'architettura della GPU utilizzata, che si è rivelata uno strumento molto efficiente e ben adatto per la risoluzione di questo tipo di simulazioni stocastiche.

Questo lavoro di tesi è così suddiviso: il capitolo 1 consiste in una introduzione generale alla problematica della modellazione dei fenomeni di moti collettivi, con riferimento ai principi teorici che stanno alla base del modello studiato e implementato.

Nel capitolo 2 si discutono gli argomenti relativi ai modelli *self-organized* e al *Fish-Schooling*.

Nel capitolo 3 vengono presentati nel dettaglio gli strumenti harware e software utilizzati per la realizzazione del progetto di tesi, ovvero le Graphics Processing Units e l'ambiente CUDA.

Nel capitolo 4 si discute il particolare modello di Fish Schooling considerato, si introduce l'implementazione del modello sviluppato nella tesi (chiamato FS-Parth) e si analizzano i risultati delle simulazioni stocastiche effettuate.

Nel capitolo 5 si riportano oltre alle conclusioni alcuni possibili sviluppi futuri di questo lavoro di tesi.

Ringraziamenti

Per la realizzazione di questo lavoro di tesi desidero ringraziare ed esprimere la mia riconoscenza nei confronti di tutte le persone che mi sono state vicine e hanno permesso e incoraggiato sia i miei studi che la realizzazione e la stesura di questa tesi. I miei più sentiti ringraziamenti vanno a tutto il team del Laboratorio del Dipartimento Di Scienze Applicate di Modellistica Numerica e Calcolo Parallelo, ed in particolare al Relatore Prof. Giulio Giunta per la fiducia fin da subito dimostratami e per avermi seguito durante lo svolgimento del lavoro con consigli e confronti che mi hanno aiutato ad intraprendere, ogni volta, le scelte più appropriate. In questo Dipartimento ho trovato sempre professori disponibili al dialogo e a confrontarsi con le idee altrui, pertanto un ringraziamento speciale a loro che mi hanno permesso di raggiungere questo traguardo molto importante per me e per la mia famiglia.

Indice

Capitolo 1

Introduzione

Lo scopo di questo lavoro di tesi è comprendere come gruppi di agenti mobili possano svolgere autonomamente missioni come il movimento in gruppo, l'allineamento delle proprie posizioni, la variazione delle proprie velocità, evitando collisioni con altri agenti, descrivendo in modo compatto ed unificato le dinamiche di interazione sia spaziale che temporale tra gli stessi.
Molte comunità di organismi presentano in modo evidente un ordine strutturale, un moto collettivo ordinato con un numero di individui che può essere molto elevato, e tale che il gruppo sembra muoversi come un singolo organismo, nonostante ciascun membro possa cambiare direzione o velocità, anche in moto casuale.
Molti comportamenti collettivi possono essere compresi considerando le interazioni locali individuali tra i membri del gruppo; per tali motivi le simulazioni al computer sono lo strumento fondamentale per analizzare il comportamento di un insieme di organismi di detta natura.
Lo studio matematico del moto collettivo, esaminato da un punto di vista soprattutto biologico, ha generalmente proceduto su due fronti. In primo luogo,

si può realizzare un modello di tipo *individual*, definendo tutte le regole che specificano le dinamiche e le interazioni tra un individuo e il restante gruppo; in secondo luogo, si può pensare di definire modelli cosiddetti *continuum* per descrivere nella loro totalità le dinamiche degli organismi coinvolti.

I modelli continui sono spesso poco affidabili, in quanto non riescono a catturare tutti i dettagli, che possono essere inclusi in un modello del primo tipo, spesso compromettendo il realismo biologico.

I modelli a scala di individui consistono normalmente in un ambiente in cui si verificano le interazioni e nei quali un certo numero di agenti sono definiti in base ai loro comportamenti (norme procedurali) e ad alcuni parametri caratteristici.

Nel capitolo 2, viene presentato un riferimento approfondito ai modelli a scala di individui e ai modelli aggregati o continui, approfondendo la loro definizione e illustrando i vantaggi e gli svantaggi nell'utilizzo dell'uno e dell'altro in merito allo studio di sistemi complessi di differente natura.

Nel caso specifico di questa tesi, è stato considerato un modello basato sugli individui relativo al Fish-Schooling; esso incorpora una tendenza da parte di ciascun pesce ad allineare la propria posizione e il proprio orientamento in funzione delle posizioni e degli orientamenti dei propri vicini, e in aggiunta una tendenza da parte di ciascun individuo ad evitare le collisioni, sulla base di alcuni parametri.

Nel modello è inclusa una componente di casualità, per tener conto delle imperfezioni che risultano dalla raccolta delle informazioni e dalla modalità con cui l'individuo agisce in funzione delle informazioni stesse. Per determinare accuratamente le proprietà statistiche del moto collettivo, le cui dinamiche

sono descritte da un modello basato sull'individuo, sono richieste molte realizzazioni, e quindi è necessaria una grande quantità di calcolo.

Nel capitolo 3, si discute delle GPU, ovvero di Graphics Processing Units, e di come esse possono essere usate in modo molto efficiente per ottenere delle simulazioni parallele di questo modello.

Grazie al loro basso costo e alle loro elevate prestazioni computazionali, le GPGPU sono diventate un campo di ricerca molto attivo con una varietà molto ampia di applicazioni scientifiche di cui le dinamiche sui fluidi, i sistemi di particelle, le reti neurali e la geometria computazionale, oltre ai modelli individual-based, sono un esempio.

GPGPU, acronimo di General Purpose on Graphics Processing Units, da un paio di anni a questa parte caratterizza molte discussioni legate alle schede grafiche, soprattutto nel momento in cui vengono presentate nuove generazioni di architetture video.

GPGPU è, in estrema sintesi, quella combinazione tra componente hardware e software che permette di utilizzare una tradizionale GPU per elaborazioni non di tipo grafico, che siano intrinsecamente parallelizzabili e che siano estremamente esigenti in termini di potenza di elaborazione.

Le computazioni intrinsecamente parallele sono in grado di beneficiare in modo sorprendentemente efficiente dell'architettura che i produttori hanno implementato all'interno delle GPU più recenti. A questo si aggiunga la loro semplice programmabilità: la grafica 3D videoludica si è recentemente trasformata, passando da un insieme di istruzioni fisse e predefinite ad un approccio per il quale le GPU vengono programmate utilizzando gli shader e quindi aprendosi virtualmente a qualsiasi tipo di applicazione.

Le precedenti generazioni di GPU richiedevano computazioni che dovevano essere integrate in una API, come OpenGL, che ha rappresentato una sfida significativa nell'ambito della programmazione con GPU per applicazioni di tipo non grafico.

In questa tesi si è scelto di utilizzare CUDA, Compute Unified Device Architecture, sviluppata dalla NVIDIA, e che rappresenta una nuova tecnolgia che consente l'implementazione diretta di programmi paralleli scritti nel linguaggio C, utilizzando un'API progettata per computazioni di tipo generale.

Nel capitolo 3 si parla nel dettaglio sia di GPU che di CUDA e si mostra il significativo vantaggio nell'utilizzo di queste due componenti del sistema, rispettivamente hardware e software, per l'esecuzione di simulazioni parallele relative al *Fish-Schooling Model*.

Capitolo 2

Modellazione di moti collettivi

Questo capitolo è diviso in quattro parti: nella prima parte si discute del concetto di simulazione, chiarendone dunque la definizione e l'importanza per lo sviluppo di particolari sistemi; nello specifico, si confronteranno le simulazioni riferite a modelli deterministici e riferite a modelli stocastici; nella seconda parte si presenta lo studio dei moti collettivi, dei quali il Fish-Schooling è un esempio; nella terza si argomenta il concetto di *Self-Organization*, o auto-organizzazione, relativo ad alcuni organismi che in natura evidenziano questa caratteristica; nella quarta parte si introduce il modello considerato in questa tesi, discutendone le proprietà teoriche.

2.1 Le Simulazioni deterministiche e stocastiche

Una simulazione è uno strumento computazionale sperimentale molto potente che può essere definito come la trasposizione in termini logico-matematici-procedurali di un "modello concettuale" della realtà; tale modello concettuale può essere a sua volta concepito come l'insieme di processi che hanno luogo

nel sistema valutato e il cui insieme permette di comprenderne le logiche di funzionamento.

I modelli computazionali di simulazione si caratterizzano per l'uso del calcolatore come strumento di calcolo e di rappresentazione degli elementi che costituiscono la realtà in esame. La corrispondenza tra realtà e modello computazionale non è basata su una riduzione proporzionale delle dimensioni, ma è di tipo funzionale: ad ogni elemento del sistema reale corrisponde un oggetto informatico che ne svolge la funzione nel modello. Essi sono particolarmente flessibili consentendo di rappresentare e di studiare sistemi molto complessi, e dei quali si conoscono talvolta solo alcune caratteristiche, attraverso analisi di tipo statistico. In questo ambito si definiscono due tipi di modelli: modello deterministico e modello stocastico.

Un modello deterministico è un modello che tenta di prevedere numericamente l'evoluzione di un sistema nello spazio-tempo, attraverso la risoluzione numerica delle equazioni matematiche che descrivono le leggi che governano il sistema in esame.

Per ottenere questo risultato è necessaria la conoscenza dello stato di partenza, ovvero delle condizioni iniziali, attraverso il quale è possibile fornire i valori di inizializzazione delle variabili indipendenti del sistema di equazioni del modello stesso. Una volta completato il processo di inizializzazione viene risolto il sistema di equazioni del modello deterministico, ottenendo un risultato unico, numerico, per ogni punto nello spazio e ad ogni istante temporale.

Un modello stocastico è un modello costituito da un insieme finito di variabili casuali che dipendono da un parametro t, con il quale si indica generalmente il tempo, e dai valori che le singole variabili casuali hanno assunto nel passa-

to, cioè con riferimento ad una base statistica di partenza. L'inizializzazione delle variabili casuali avviene mediante l'identificazione della distribuzione di probabilità che caratterizza ogni singola variabile, attraverso l'analisi statistica di una base di dati raccolti nel passato, che rappresenta lo spazio probabilistico dei valori che la variabile casuale può assumere. Una volta ricostruita la distribuzione di probabilità iniziale delle singole variabili casuali è possibile simulare, attraverso il modello stocastico, la variazione nel tempo della distribuzione di probabilità delle variabili casuali, ottenendo come risultato un nuovo spazio probabilistico di valori per ogni variabile casuale.
E' interessante puntualizzare le principali differenze che intercorrono tra un modello deterministico ed un modello stocastico. La prima differenza sostanziale fra i due modelli risiede nelle variabili utilizzate, in quanto, un modello deterministico utilizza variabili che possono assumere, in un determinato istante, uno ed un solo valore, mentre un modello stocastico utilizza variabili casuali, cioè che possono assumere un insieme di valori, in un determinato istante, aventi differente probabilità di manifestazione. La seconda differenza risiede nel fatto che un modello deterministico è basato sulla risoluzione di un sistema di equazioni matematiche deterministiche mentre un modello stocastico è composto da un sistema di equazioni stocastiche o da un insieme di procedure statistiche. La terza differenza riguarda la particolarità che un modello deterministico necessita della conoscenza dei valori effettivi per l'inizializzazione delle variabili indipendenti che caratterizzano il sistema di equazioni, all'istante iniziale T_0, mentre un modello stocastico necessita di una base di dati appartenenti al passato per l'inizializzazione delle variabili casuali, all'istante iniziale. La quarta differenza fra i due modelli è relativa

al fatto che un modello deterministico fornisce in output, all'istante T_n, un unico valore per ogni variabile inizializzata all'istante T_0, mentre un modello stocastico fornisce all'istante T_n, una distribuzione di probabilità per ogni variabile casuale inizializzata all'istante T_0, cioè evolve in un intervallo di valori probabili.

Gli approcci deterministici tradizionali per le simulazioni di reazioni chimiche o per la descrizione di modelli biologici e ambientali falliscono nella capacità di cogliere la casualità insita in sistemi di detta natura; pertanto, nonostante l'approccio deterministico sia efficace in molti casi, esso fallisce nella descrizione della maggioranza dei sistemi formati da organismi viventi, nei quali una piccola popolazione di alcune specie può risultare in un comportamento stocastico [1] e discreto.

[1] Molto spesso si utilizzano i termini *random*, con il significato di casuale, e *stochastic*, con il significato di stocastico, senza conoscere effettivamente la differenza tra i due.

Dal vocabolario *The new shorter Oxford English Dictionary (1993)* risulta:

Random: Guidato da un processo delle pari opportunità che coinvolgono sia reali che ipotetici membri di una popolazione o ottenuto da un processo di questo tipo e comunque completamente inprevedibile.

Stochastic: Casualmente determinato, che segue alcune distribuzioni di probabilità casuali cosicchè il comportamento potrebbe essere analizzato statisticamente ma non precisamente determinato.

Queste definizioni suggeriscono che i due vocaboli sono sinonimi ma che il primo è riferito ad un contesto di natura fisica mentre il secondo ad un contesto più matematico. Questo è confermato apertamente nella prefazione di Doob (1953) in cui viene affermato che un processo stocastico è l'astrazione matematica di un processo empirico il cui sviluppo è governato da leggi di probabilità. In particolare, nel 1906 Markov formulò il processo che fu solo dopo la sua morte chiamato con il suo nome: esso può essere definito come il futuro che è indipendente dal passato ma che tiene conto del presente, in altre parole passato e

2.2 Moti Collettivi

I modelli più semplici di moto collettivo assumono che gli individui si muovano ad una velocità costante e una direzione che è fortemente dipendente dai loro vicini e includono una componente stocastica che si concretizza nella simulazione del comportamento degli individui risultante da forme di repulsione locale, allineamento e orientamento del singolo agente in funzione della totalità che costituisce il gruppo.

Nella formazione dell'insieme e nella coesione tra i diversi individui, quindi sono proprio le simulazioni che aiutano a mostrare il cambiamento dei loro comportamenti sulla base di alcuni parametri, che possono mutare nel tempo, e che mostrano come essi possano influenzare lo stato dei singoli organismi all'interno del gruppo. Tra le prime domande riguardanti il meccanismo del trasferimento delle informazioni in gruppo, è inclusa quella relativa al come gli individui riconoscano coloro che sono informati e quella relativa al modo in cui i gruppi possano prendere una decisione collettiva in funzione anche di una mancata preferenza espressa da parte di qualche individuo.

E' noto che diverse specie di animali abbiano dei riferimenti comportamentali legati a segnali specifici, come la danza delle api, la quale, innescata da un individuo singolo, è in grado di indicare a tutti gli altri individui la necessità di muoversi verso una particolare fonte di nutrimento.

Sebbene sia probabile che alcune specie abbiano una propensione determinata geneticamente nel migrare verso una particolare direzione o a rispondere a

futuro sono statisticamente indipendenti quando il presente è noto. Nella teoria di Markov, quindi, un processo stocastico ha memoria zero del passato e il futuro è espresso come una funzione del presente per alcuni dati statistici nella transizione temporale degli eventi.

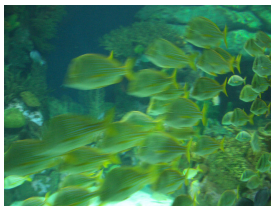

Figura 2.1: Banco di pesci

Figura 2.2: Stormo di uccelli

segnali abiotici, come gradienti termici, che possono influenzare i movimenti, è vero anche per molte specie che alcuni membri all'interno del gruppo giochino un ruolo da *leader*, fondamentale per guidare coloro che non hanno esperienza oppure non godono di quelle caratteristiche necessarie per dirigere una collettività.

In figura 2.1 e in figura 2.2 sono mostrati due esempi di moto collettivo presenti in natura. Il modello considerato in questa tesi è un *individual based model*, basato quindi sul comportamento individuale, ed include una componente stocastica che tiene conto delle imperfezioni legate alla raccolta delle informazioni e all'azione da compiere in virtù di queste.

Per alcuni gruppi di animali, come i banchi di pesci o gli sciami di insetti, potrebbe essere irragionevole pensare che i membri di un determinato gruppo abbiano la capacità del riconoscimento individuale; pertanto simili specie si riuniscono senza il bisogno di avere una guida all'interno del gruppo di cui fanno parte oppure senza avere la necessità di un riferimento esterno.

2.2.1 Individual-based models and continuum models

Nei modelli di comunità aggregata, o modelli continui, le popolazioni sono modellate come un unico elemento, attraverso una variabile di stato che rappresenta le caratteristiche "medie" della popolazione. I modelli matematici ecologici classici sono rappresentati con questo approccio ed è possibile introdurre alcune complessità nell'approccio modellistico di comunità aggregata rappresentando ad esempio i modelli ad età, in cui gli individui sono raggruppati in classi. Si analizzano di seguito i pro e i contro di questa categoria di modelli.

Pro:

- Semplice da implementare, utilizzando strumenti di sistema dinamici o librerie numeriche.
- Disponibilità di numerose classi di modelli già studiati.
- La complessità computazionale del modello non dipende dal numero degli individui della comunità (possono quindi modelizzare comunità con un gran numero di individui).
- I risultati sono semplici da analizzare, sia nel caso deterministico sia nel caso stocastico.

Contro:

- Il sistema biologico viene "costretto" in un modello spesso inadatto per rappresentare una complessità biologica.
- E' assunto un semplice ciclo di vita, in cui il modello descrive solo il comportamento medio dell'individuo.
- La distribuzione spaziale delle risorse non è modellizzata.
- I ricercatori sono costretti ad aggregare (in valori medi) le loro conoscenze e informazioni.

I modelli a scala di individuo sono stati introdotti nel 1988 da Huston (individual-based model, IBM). Essi descrivono la popolazione come un insieme di individui con proprietà singole e potenzialmente differenti.
Il comportamento dinamico di tutto il sistema deriva dalla interazione della dinamica dei singoli individui e viene studiato utilizzando un approccio modellisitco bottom-up (o reduzionistico). Si esaminano di seguito i pro e i contro di questo tipo di modelli.
Pro:

- La proprietà emergente del modello è data dalla integrazione delle interazioni tra gli individui.
- Ogni individuo è descritto dalle sue caratteristiche.
- La distribuzione spaziale delle risorse viene considerata.
- E' possibile inserire le conoscenze direttamente nel modello.
- Dal punto di vista implementativo il modello è intrinsecamente parallelo.

Contro:

- L'implementazione richiede elevate potenze di calcolo.
- L'implementazione è effettiva solo in calcolatori paralleli.
- Esistono pochissimi tool user friendly per semplificare il processo di implementazioni.

In seguito al raffronto tra i due differenti tipi di modelli e di approcci, si può affermare che i modelli individual-based permettono di simulare la dinamica delle popolazioni in maniera più realistica rispetto ai modelli di comunità aggregata, in particolare considerando la dinamica della comunità come una proprietà emergente del sistema, ottenuta attraverso l'interazione dei singoli individui, tenendo conto delle interazioni spaziali, dell' analisi del disturbo e degli eventuali cambiamenti dell'ambiente circostante.
Finora i modelli relativi ai banchi di pesci sono stati valutati per la loro abilità nel predire coesione e polarizzazione, e per tale motivo molto studiati. Una delle piu' affascinanti caratteristiche del comportamento collettivo dei pesci è la loro abilità nel muoversi in gruppi ben formati e chiusi. I pesci possono costituire insiemi strutturati chiamati banchi, ovvero strutture organizzate con movimenti sincronizzati. I banchi di pesci (fish schools) possono essere concepiti come sistemi auto-organizzati, dal momento che non hanno bisogno di un leader o di stimoli esterni per evitare la scissione, per muoversi in maniera coesa e per adottare una direzione comune. Le dinamiche di questi banchi emerge dalle numerose interazioni reciproche tra gli individui che si trovano all'interno di un range di percezione limitato e il movimento del singolo individuo dipende solo dalle posizioni dei propri vicini.

2.3 Self-Organization

I sistemi tecnologici diventano organizzati tramite comandi provenienti dall'esterno mentre quelli naturali divengono strutturati tramite i loro processi interni. Questi ultimi sono sistemi auto-organizzati e l'emergenza di ordine al loro interno è un fenomeno complesso che suscita l'interesse degli scienziati di varie discipline. L'auto-organizzazione si riferisce ad una vasta gamma di modelli in sistemi fisici e biologici; una caratteristica fondamentale di questi diversi sistemi e' la modalità attraverso cui essi riescono ad acquisire il loro ordine e la loro struttura, ovvero tramite interazioni interne senza l'intervento diretto di influenze esterne.

Il fisico tedesco, Hermann Haken, nel 1977, evidenziò questo aspetto con un esempio basato su un'attività umana: *"Si considera un gruppo di lavoratori: si parlerà di organizzazione oppure, più precisamente, di comportamento organizzato se ciascun lavoratore agisce in un modo ben definito sulla base di ordini esterni dati, per esempio, dal capo, oppure si definirà lo stesso processo come* ***auto-organizzato*** *se non vi sono ordini esterni ma i lavoratori riescono comunque a svolgere i propri compiti con reciproca comprensione".*

Per esprimere nel modo più chiaro possibile ciò che si intende per auto-organizzazione nel contesto del modello di formazione nei sistemi biologici, si forniscono le seguenti definizioni: l'auto-organizzazione è un processo in cui un modello a livello globale di un sistema emerge esclusivamente da numerose interazioni tra le componenti di più basso livello del sistema stesso. Inoltre, le regole che specificano le interazioni tra le componenti vengono eseguite utilizzando solo informazioni a livello locale, senza riferimento al modello globale. In breve, il modello è una proprietà emergente del sistema, piuttosto

che una proprietà imposta sul sistema per un'influenza esterna predefinita. Tale proprietà emergente, inoltre, non deve essere percepita semplicemente esaminando quelle delle componenti ma richiede una considerazione delle interazioni tra queste ultime. E' importante puntualizzare che le componenti non necessariamente devono interagire direttamente.

E' bene anche chiarire il significato del termine modello, inteso come un insieme organizzato di oggetti (individui) collocati in uno spazio ed in un tempo; esempi di modelli biologici includono banchi di pesci, stormi di uccelli, il lampeggiare sincrono delle lucciole e altri simili. In un banco di pesci, per esempio, ogni individuo basa il proprio comportamento sulla percezione della posizione e della velocità del suo vicino più prossimo, piuttosto che sulla conoscenza di un comportamento globale dell'intero insieme. I meccanismi di auto-organizzazioe nei sistemi biologici differiscono da quelli fisici per due motivi di base. Il primo è la maggiore complessità dell'individuo nei sistemi biologici. Essi, in quelli fisici, sono rappresentate da oggetti inanimati come granelli di sabbia o reagenti chimici, mentre nei sistemi biologici, in virtù della maggiore complessità intrinsecsa, essi sono definite a partire da organismi viventi come pesci, formiche, batteri, cellule,...

La seconda differenza riguarda la natura delle regole che governano l'interazione tra le componenti del sistema; in quelli chimici e fisici, il modello è creato attraverso interazioni basate soltanto sulle leggi fisiche. Naturalmente, i sistemi biologici obbediscono alle leggi della fisica, ma in aggiunta a queste leggi esistono le interazioni comportamentali e fisiologiche tra i componenti viventi. In particolare, gli individui nei sistemi biologici acquisiscono le informazioni relative alle proprietà locali dell'insieme e agiscono in base a partico-

lari programmi genetici che sono stati sottoposti alla selezione naturale. Un ulteriore interessante aspetto riguardante l'emergere di un comportamento auto-organizzato nei sistemi biologici è la possibilità di modellizzare questi sistemi in modo semplice, sfruttando l'osservazione delle interazioni elementari tra i singoli individui, anche quando i comportamenti sono estremamente complessi, come i movimenti coordinati di un banco di pesci.

2.4 Fish Schooling

Nel corso dell'evoluzione gli organismi hanno adottato diverse strategie per la lotta quotidiana alla sopravvivenza: da una parte i predatori che affinano le tecniche per la cattura delle prede; dall'altra queste ultime che tentano di sfuggire ai continui attacchi dei cacciatori. Ammassati in enormi formazioni compatte, i pesci possono scoraggiare eventuali aggressori, o far fronte, con maggiori possibilità di sopravvivenza, alle pressioni e all'attacco dei predatori. Un banco di pesci (il corrispondente del branco per le specie terrestri) si viene a formare anche durante le migrazioni, ed il fine è sempre lo stesso, quello della difesa.

L'origine del termine "banco" va ricercato nella parola inglese shoal che vuol dire appunto banco, inteso sia come banco di sabbia che come moltitudine di pesci che nuotano nella stessa direzione. Infatti si usa il termine shoal per gli individui adulti, e il termine school, scuola, per gli individui giovani. Shoaling e schooling indicano i fattori e i processi che determinano le aggregazioni. La vita di relazione del banco è molto più complessa del branco terrestre, in quanto il pericolo può arrivare da ogni direzione, e si basa su un'intesa eccezionale tra i singoli individui del gruppo. Le migliaia, e a volte

decine o centinaia di migliaia di unità che lo compongono, si muovono con una apparente perfetta sincronia, tanto da apparire agli occhi del predatore, ed anche ai nostri, come un unico corpo di enormi dimensioni che fluttua nell'acqua, ora a destra, ora a sinistra, ora spezzandosi in due o tre gruppi che tornano poi a riunirsi in una caratteristica nube densa e scura, come una sorta di groviglio vivente che nuota con un sincronismo stupefacente, schivando i decisi attacchi dell'animale cacciatore.

Un ruolo altrettanto importante nella comunicazione di massa del banco è svolto dal sistema sensoriale, che si avvale della cosiddetta linea laterale, capace di fornire all'animale informazioni utili al mantenimento del ritmo all'interno del gruppo. Il rilevamento di variazioni di direzione e di velocità delle correnti locali, generate dai movimenti dei pesci vicini, permette ad ogni individuo del banco di regolare la propria posizione, evitando confusioni o collisioni in caso di repentini cambiamenti di direzione o di improvvise accelerazioni, come avviene in presenza di un predatore. I banchi rappresentano l'espressione di questo spirito gregario, che nell'ambiente acquatico è molto diffuso, visto che la metà delle specie ittiche lo adotta nella prima fase di vita, ed un quarto delle specie lo adotta poi anche come modello di vita permanente. Non tutti i banchi di pesci mantengono sempre la formazione compatta di banco, ma i componenti possono essere disposti in maniera più o meno casuale gli uni rispetto agli altri, ed organizzarsi in shoals solo in determinate occasioni. Un banco è costituito sempre da un considerevole numero di individui che normalmente appartengono alla stessa specie (i banchi di certe specie pelagiche arrivano a superare i 3 miliardi di componenti). Ci possono essere anche banchi costituiti da individui appartenenti

a specie differenti (banchi eterospecifici). La caratteristica che accomuna gli individui di un banco di pesci della stessa specie o di specie differenti è la conformità nella taglia dei componenti, in modo tale da permettere un perfetto allineamento dei singoli pesci, determinando così la formazione di una massa impenetrabile. Ritornando al caso specifico della tesi, si considera un modello bidimensionale basato sugli individui facenti parte di un banco di pesci studiando le interazioni comportamentali locali. Questo modello rientra nella categoria dei moti collettivi relativi a quegli organismi che non agiscono seguendo un leader di esperienza e nei quali lo stato del singolo dipende dai pesi di orientamento e di attrazione verso i restanti membri del banco stesso. Il modello è basato sulle seguenti ipotesi di base:

1. Ogni pesce all'interno del banco si muove senza che abbia un particolare punto di riferimento, anche se sembra esserci una temporanea presenza di un capo-gruppo durante le simulazioni.

2. Il movimento di ciascun individuo è solo influenzato dal suo vicino più prossimo.

3. Una componente casuale influenza il comportamento dei singoli individui.

Capitolo 3

Calcolo scientifico su processori grafici

Nel seguente capitolo sono descritti nel dettaglio i principali strumenti hardware e software utilizzati per la realizzazione del progetto di tesi, ovvero si parla di GPU e CUDA. Per ciascuna di tali tematiche seguono degli approfondimenti che aiutano a comprendere l'evoluzione nel tempo degli stessi strumenti e la modalità di impiego nei differenti ambiti di sviluppo.

3.1 GPU

Una GPU (Graphics Processing Unit) è un dispositivo hardware dedicato al rendering grafico impiegato in personal computer, workstation o console. Nota anche come Video Display Processor, ha iniziato a diffondersi alla fine degli anni '80 come chip integrato per poi evolversi, negli ultimi anni, fino a diventare una scheda di espansione su bus preferenziale.
Una GPU fornisce al programmatore la possibilità di risolvere un sottoinsieme più o meno ampio di problemi di diversa natura, usufruendo delle potenzialità dell'accelerazione hardware. Attualmente le GPU si presentano

come dispositivi estremamente performanti in grado di elaborare con picchi fino a 500 Gflops/s (ATI X1900XT) che paragonati ai 12 Glops/s ottenibili su un Pentium4 3 GHz rappresentano un dato meritevole di attenzioni da parte di coloro che si occupano di calcolo scientifico ad alte prestazioni. I punti di forza con cui le GPU riescono a ottenere picchi di tale consistenza possono essere sostanzialmente identificati in:

- Frequenza di clock;
- Velocità della memoria primaria;
- Volume delle informazioni processate in parallelo.

Per comprendere invece quali siano i benefici dell'utilizzo di una GPU per elaborazioni che non siano legate alla grafica 3D, è opportuno considerare:

- Vantaggi prestazionali: con applicazioni che sono intrinsecamente parallele è possibile, attraverso l'ottimizzazione del software, ottenere incrementi prestazionali rilevanti, con prestazioni sino a 100 volte superiori. L'ordine di grandezza, quindi, non è quindi quello dell'incremento di alcune decine di punti percentuali, o di un raddoppio della potenza elaborativa, ma di scenari per i quali se una elaborazione con una CPU richiede una unità di tempo, nella stessa unità di tempo è possibile eseguire 100 di queste elaborazioni con una GPU.
- Costo d'acquisto: i prezzi delle GPU sono allineati a quelli dei processori; è sempre vero che per utilizzare una GPU per elaborazioni di calcolo parallelo è richiesto comunque un processore, ma considerando

il costo di una GPU top di gamma rispetto per esempio ad un cluster di processori che permettono di ottenere la stessa potenza elaborativa con una specifica applicazione, il beneficio economico dell'utilizzo di una GPU è evidente.

- Tasso di aggiornamento tecnologico: i produttori sviluppano nuove GPU con un tasso di aggiornamento ben più rapido di quanto non avvenga nel mercato dei processori. Se per le CPU si possono assistere ad incrementi delle frequenze di clock con una periodicità predefinita, l'architettura di base tende a restare immutata per periodi di tempo di 2 anni come minimo; ogni 12-18 mesi i produttori di GPU rilasciano una nuova architettura top di gamma e un suo aggiornamento, con prestazioni più elevate, seguito da una nuova architettura che rivoluziona i livelli prestazionali della precedente.

Nel corso degli anni si è assistito ad un vertiginoso miglioramento dei parametri sopra elencati, con un drastico aumento delle possibilità offerte da questi dispositivi.

NVIDIA ha presentato un proprio nuovo marchio, Tesla, per indicare una nuova completa famiglia di soluzioni per il calcolo parallelo, incentrate sull'utilizzo di GPU completamente programmabili. Cuore di questa iniziativa è l'architettura G80, utilizzata per questo progetto di tesi e alla base della quale si conferma la volontà di utilizzare la stessa architettura video per differenti tipologie di implementazioni.

3.2 Evoluzione

Le moderne GPU discendono dai chip grafici monolitici di fine anni '70 e '80. Esse venivano spesso implementate con microprocessori non specializzati ad alta velocità e non erano dotate di funzioni per il disegno di forme.
Man mano che la tecnologia di costruzione di chip migliorava, è diventato possibile integrare sulla stessa scheda (e in seguito sullo stesso chip) sia le funzioni di disegno che la funzione BitBLT (acronimo di Bit-block Transfer) che permette il trasferimento di una sezione rettangolare di un'immagine in un'altra della stessa forma e dimensione, e consente di eseguire tutte le operazioni tipiche delle applicazioni grafiche. Questi acceleratori 2D semplificati non erano così flessibili come le GPU basate su microprocessore ma erano molto piu' facili da costruire e da vendere. L'Amiga è stato il primo computer per il mercato di massa a includere un blitter nel suo hardware video, cioè un co-processore dedicato al rapido trasferimento dei dati all'interno della memoria del computer, mentre il sistema grafico IBM 8514 è stato uno delle prime schede video per PC ad implementare in hardware le primitive 2D. All'inizio degli anni 1990, la diffusione di Microsoft Windows ha generato un grande interesse verso la grafica raster 2D ad alta velocità e alta risoluzione; nel 1993, S3 Graphics ha introdotto il primo acceleratore 2D su singolo chip, l'S386C911; nel 1995 tutti i principali produttori di chip grafici per PC avevano aggiunto il supporto per l'accelerazione 2D ai loro chip. Nel 1999 fu introdotta la prima generazione di schede grafiche GeForce prodotte dalla NVIDIA, pensate ad hoc per la grafica 3D. Un anno dopo ATI Technologies introdusse il primo chip della famiglia Radeon. Queste due aziende acquistarono sempre più quote di mercato e ancora oggi sono in concorrenza nella

vendita delle schede grafiche per videogiochi. Con l'avvento dell'API DirectX versione 8 e di simili funzionalità in OpenGL, le GPU hanno aggiunto l'ombreggiatura (shading) programmabile alle loro funzioni. Ogni pixel e ogni vertice geometrico potevano essere elaborati da un programma che riceve in input dati sotto forma di texture e li elabora prima di proiettarli sullo schermo.

Nel 2003, con l'introduzione della NVIDIA GeForce FX (detta anche NV30), gli ombreggiatori (shader) di pixel e di vertici potevano realizzare dei cicli, delle lunghe sequenze di operazioni a virgola mobile, e in generale stavano diventando flessibili quanto una CPU dedicata all'elaborazione di immagini raster. L'ultima versione oggi disponibile di DirectX è la 10, integrata in Windows Vista, che rappresenta un ulteriore passo avanti nella programmabilità delle GPU, grazie all'introduzione di shader unificati. Inoltre, con l'avvento dei Blu-ray Disc la GPU dovrà essere progettata per gestire il segnale e le chiavi AACS che servono per verificare l'autenticità di un film. Oggi le GPU parallele hanno cominciato a contendere alla CPU funzioni computazionali, e un sottosettore della ricerca, chiamato GPGPU (acronimo di General Purpose Computing on GPU), ha trovato impiego in vari campi applicativi del calcolo scientifico, come l'esplorazione petrolifera, l'elaborazione di immagini, la determinazione del prezzo delle opzioni sulle azioni di borsa,...

3.3 Architettura hardware

La moderna operazione di rendering per immagini 3D è interamente basata su texturing: nel corso degli anni è diventata sempre più complessa e ci si è avvalsi di shader, programmi costituiti da decine di istruzioni matemati-

che per l'accesso a tali strutture dati, e per la loro elaborazione inducendo a pensare ad una gestione più efficiente delle texture e concentrandosi sulla potenza di calcolo. I problemi da risolvere non sono semplici, per due motivazioni principali: la prima riguarda il fatto che l'accesso alla texture è spesso associata ad un'operazione di filtering della stessa e, quando quest'ultima è di alta qualità, impiega diversi cicli per essere eseguita; la seconda, molto più importante, riguarda l'architettura.

L'accesso alla memoria grafica può richiedere fino a più di 100 cicli. Se si dovessero aspettare 100 cicli per passare all'istruzione successiva l'efficienza complessiva sarebbe disastrosa. La GPU è in grado di effettuare il precaricamento dei dati in una memoria cache, al fine di ridurre il costo di accesso alla memoria. Quando le texture erano semplici decorazioni, questo compito era svolto facilmente, perché la GPU sapeva a priori quale area di memoria doveva essere occupata. In seguito, con l'evoluzione delle tecniche di rendering, le textures sono state destinate a contenere vari tipi di dati e il loro accesso non era più così chiaramente determinato.

Tuttavia, le GPU delle nuove generazioni si basano su un'architettura unificata che consente l'utilizzo delle stesse unità per processare tutti i tipi di elementi grafici, sia che si tratti di pixel, sia che si tratti di vertici.

3.3.1 NVIDIA G8 series

Con il rilascio delle schede video serie G8, NVIDIA ha rivoluzionato completamente il design dell'architettura GPU, basandola sull'utilizzo di shaders unificati; in figura 3.1 è mostrato il diagramma a blocchi per le G8.

Gli shaders al suo interno (vertex, geometry e pixels) sono in grado di ope-

rare sullo stesso insieme di risorse di esecuzione, prestandosi, in tale modo, ad applicazioni di vario genere relative a molteplici scenari.

Figura 3.1: Diagramma a blocchi G8

L'architettura è in grado di utilizzare l'hardware relativo alla gestione dei thread per la spedizione dei diversi tipi di istruzioni allo shader core. Pertanto, l'output del Vertex shader viene indirizzato come input del Geometry shader posizionandosi nella parte alta dello shader core; in seguito l'output del Geometry shader diviene l'input per il Pixel shader: una rappresentazione concettuale del tutto è visibile nello schema in figura 3.2.

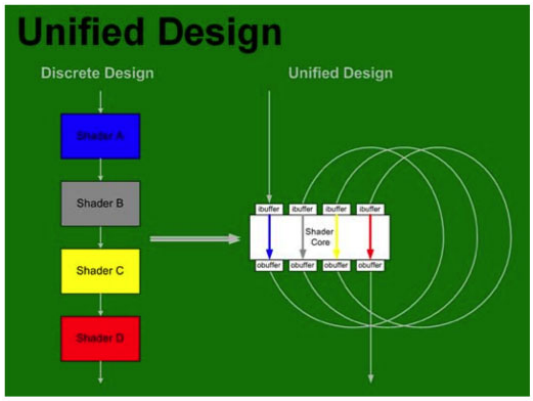

Figura 3.2: Shader unificati

Il chip G80 è costituito da 681 milioni di transistor, più di un single-core Itanium 2. Come riferimento si considerano da un lato l'X1900 di ATI Ra-

deon XTX basato sulla GPU R580, con 384 milioni di transistor disponibili e, dall'altro, la precedente GPU di fascia alta della NVIDIA, G71, basata su GeForce 7900 GTX, con 278 milioni di transistor.

In media, un sistema G8 utilizza circa l'8% di potenza in più rispetto ad una ATI Radeon X1950 XTX e, per comprenderne l'evoluzione, basta ossevare con attenzione lo shader core.

Esso è composto da 128 processori semplici, chiamati Stream Processors (SP), che operano ad una frequenza di 1.35 GHz. Gli SP sono fondamentalmente delle pipeline e sono in grado di operare solo su valori scalari; la semplificazione di questi processori e l'estensione delle loro pipeline ha consentito di raggiungere le prestazioni delle G8. Le GPU sono generalmente progettate tramite un linguaggio di descrizione hardware (HDL), che è una sorta di linguaggio di programmazione ad alto livello utilizzato per tradurre il codice in un layout di transistor che può essere impiegato nella costruzione di un chip. In passato, gli elementi costitutivi di una GPU sono stati progettati a livello di transistor e le interfacce di memoria, circuiti analogici e registri sono stati realizzati a partire da un livello di progettazione personalizzato mentre le componenti restanti della pipeline e gli shaders sono stati realizzati scrivendo codice ad alto livello HDL e basandosi su un layout automatico. Poichè gran parte del lavoro svolto dall'hardware relativo alla grafica è fortemente basato su vettori, diventa naturale programmare codice se si lavora con un insieme di processori paralleli, independenti e scalari e risulta anche più efficiente per ottenere unità separate per l'indirizzamento delle textures e l'operazione di filtering.

NVIDIA ha separato le unità di textures dal loro hardware shader, utilizzan-

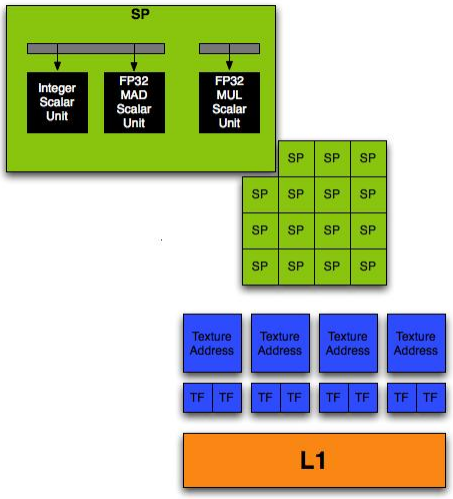

Figura 3.3: Stream Processors

do un insieme di processori scalari, ovvero gli Stream Processors, per eseguire le operazioni sui vettori, gestendo tutte le computazioni di tipo matematico tramite lo shader core.

Nel dettaglio, in figura 3.3, è possibile notare la struttura di un blocco di Stream Processors: 16 SPs per quad, o blocco; ciascun blocco di 16 SPs condivide 4 unità di indirizzamento per texture, 8 unità di filtraggio per texture e una cache L1. L'esecuzione di qualsiasi tipo di codice, utilizzando questa struttura hardware, rende questo shader molto potente e applicabile in ogni ambito di sviluppo; appena un thread viene processato da uno SP, G8 è in grado di scrivere l'output dello shader nella memoria in qualsiasi momento, a differenza della tecnologia hardware precedente, nella quale i dati dovevano passare attraverso l'intera pipeline seguendo diverse fasi per far sì che un valore fosse scritto nel frame buffer. In figura 3.4 è mostrata la tabella delle specifiche tecniche della scheda video NVIDIA Geforce 8400 GS, utilizzata per l'implementazione del modello descritto in questa tesi.

	GeForce 8400 GS
Clock core (MHz)	450
Clock shader (MHz)	900
Clock memoria (MHz)	400
Quantità memoria	256MB
Interfaccia di memoria	64-bit
Banda di memoria (GB/s)	6.4
Fill Rate texture (miliardi/s.)	3.6

Figura 3.4: Specifiche tecniche NVIDIA Geforce 8400 GS

3.3.2 SIMD

SIMD è un'architettura in cui più unità elaborano dati diversi in parallelo: è utilizzata da processori vettoriali o da processori che funzionano in parallelo. Il modello SIMD è composto da un'unica unità di controllo che esegue una istruzione alla volta controllando più ALU che operano in maniera sincrona. Ad ogni passo, tutti gli elementi eseguono la stessa istruzione scalare, ma ciascuno su un dato differente. Un elaboratore basato su questo modello è anche detto Array Processor. In passato venivano prodotti un numero elevato di dispositivi dedicati allo svolgimento di compiti specifici usualmente di tipo DSP (Digital Signal Processor) opportunatamente programmati. La differenza fondamentale tra le istruzioni SIMD e i DSP è che questi ultimi sono dotati di un set di istruzioni completo e quindi sono in grado di svolgere teoricamente qualsiasi compito. Al contrario, le istruzioni SIMD sono progettate per manipolare elevate quantità di dati in parallelo e per le usuali operazioni si appoggiano ad un altro insieme di istruzioni gestito dal processore. Inoltre i DSP tendono a includere un certo numero di istruzioni dedicate ad elaborare tipi specifici di dati come possono essere i dati audio o video mentre le istruzioni SIMD vengono utilizzate per elaborare dati generici.

Nelle elaborazioni di dati multimediali spesso si incontrano algoritmi che

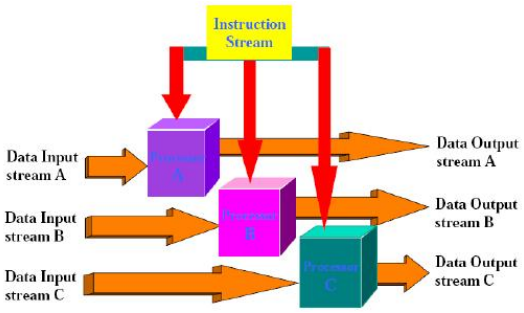

Figura 3.5: SIMD Instruction Stream

possono avvantaggiarsi di un'architettura SIMD (Figura 3.5). Per esempio per cambiare la luminosità di un'immagine il processore dovrebbe caricare ogni pixel che compone l'immagine nei suoi registri, effettuare la modifica della luminosità e poi salvare i risultati in memoria. Un processore SIMD eseguirebbe prima un'istruzione che caricherebbe con un'unica operazione un certo numero di pixel (il numero preciso dipende dall'architettura) poi modificherebbe tutti i dati in parallelo e in seguito li salverebbe tutti contemporaneamente in memoria. Eseguire le operazioni a blocchi invece che agire sui singoli pixel rende la computazione molto più efficiente, considerata la migliore efficienza del trasferimento dei dati a blocchi rispetto alla sequenza di singoli trasferimenti in memoria.

Un altro vantaggio deriva dal fatto che tipicamente le istruzioni SIMD sono sempre in grado di manipolare tutti i dati caricati contemporaneamente: quindi se un processore SIMD è in grado di caricare 8 dati, questo sarà anche in grado di processarli tutti contemporaneamente. La prima architettura SIMD a essere disponibile commercialmente fu l'architettura MMX realizzata da Intel. Le istruzioni MMX tuttavia fallirono l'obiettivo di portare i benefici del paradigma SIMD nei comuni personal computer; questo si era avuto per

due motivi principali: lo scarso supporto fornito da Intel agli sviluppatori e la mancanza di istruzioni utilizzabili nell'emergente mondo della grafica 3D. Le istruzioni MMX manipolano numeri interi e quindi non sono in grado di gestire le trasformazioni geometriche dei videogiochi perchè in questo compito sono richieste operazioni floating point. Inoltre l'utilità di un set di istruzioni capace di accelerare tali calcoli si è resa evidente in maniera chiara con l'affermazione di API dedicate alla gestione del 3D quali le Direct 3D e OpenGL. Questo fatto ha portato Intel a sviluppare una estensione del set di istruzioni, in maniera analoga a quanto fatto con gli interi, anche per le operazioni in virgola mobile a singola precisione chiamata Internet SSE (Internet Streaming SIMD Extension) o più semplicemente SSE.

Nel contempo si decise di estendere la tecnologia MMX (quale ad esempio istruzioni per facilitare le codifiche in tempo reale di tipo MPEG-2) e di introdurre istruzioni per mascherare la latenza che deriva dalle notevoli dimensioni, in termini di memoria, dei dati implicati in applicazioni video. Il termine Streaming si riferisce appunto alla presenza di istruzioni che permettono il prefetch di dati simultaneamente all'elaborazione di altri già disponibili velocizzando il flusso (stream) dei dati in ingresso e in uscita dal processore nascondendo nel tempo di esecuzione la latenza del fetch. Una delle scelte basilari nella definizione di un'architettura SIMD consiste nello stabilire su quanti dati contemporaneamente si vuole operare, in modo da raggrupparli in un vettore di dimensioni adeguate, che costituirà il nuovo tipo di dato cui faranno riferimento le istruzioni SIMD.

Il team di sviluppatori di Intel ritenne che la computazione parallela di 4 dati floating point a singola precisione (32 bit) e quindi di un data-type

SSE da 128 bit consentisse un raddoppio complessivo delle performance, senza aggiungere eccessiva complessità all'architettura essendo tendenzialmente ottenibile con un doppio ciclo della esistente architettura a 64 bit. La scelta di operare su 2 floating point non avrebbe consentito di ottenere prestazioni paragonabili mentre l'adozione di un datapath di 256 bit (8 FP da 32 bit) avrebbe determinato un impatto maggiore in termini di complessità. Mentre i 128 bit possono essere separati in 2 istruzioni da 64 bit che possono essere eseguite in un ciclo, con 256 bit, per mantenere lo stesso throughput, si sarebbe dovuto raddoppiare la larghezza delle unità di esecuzione e quindi la banda di memoria per alimentarle. Stabilito il datapath di 128 bit si poneva la domanda se implementare i registri a 128 bit nei registri MMX/x87 esistenti oppure definire un nuovo stato con registri appositi. La prima scelta, analoga a quella attuata con l'estensione alla tecnologia MMX, avrebbe comportato il vantaggio della piena compatibilità con il sistema operativo ma lo svantaggio di dover condividere i registri della architettura IA-32. La seconda scelta avrebbe comportato il problema di dover adattare i sistemi operativi, ma avrebbe avuto il vantaggio di facilitare i programmatori e la possibilità di eseguire contemporaneamente istruzioni MMX, x87 o SIMD-FP. I progettisti Intel optarono per aggiungere un nuovo stato architetturale, con la definizione di 8 nuovi registri da 128 bit (chiamati registri XMM), cosa che non era stata fatta con l'introduzione delle istruzioni MMX che operavano sugli stesi registri fisici della Floating Point Unit. Benchè le istruzioni SSE abbiano introdotto un utile parallelismo delle istruzioni floating point incrementi di prestazioni in molti settori del processing multimediale, ci sono ancora tipologie di operazioni non supportate come le moltiplicazioni tra

interi a 32 bit importanti per il processing audio di qualità. Per risolvere tutti questi problemi, Intel ha introdotto nel suo Pentium4 ben 144 nuove istruzioni SIMD che prendono il nome di SSE2, che hanno colmato buona parte delle mancanze presenti in SSE.
Sempre Intel ha introdotto, a partire dalla generazione Prescott, le istruzioni SSE3 che senza andare a incrementare sostanzialmente le potenzialità delle architetture SIMD ne hanno semplificato l'utilizzo.

3.4 CPU vs GPU

Per un confronto tra CPU e GPU, si considera il seguente esempio: si supponga di avere a disposizione due array di 1000 elementi tra i quali effettuare l'operazione di somma. Un programma CPU accederebbe iterativamente agli elementi dei due array, e per 1000 elementi, eseguirebbe dunque 1000 iterazioni. Lo stesso programma su GPU genera 1000 thread somma e se si utilizzasse una Geforce GTX con 240 core, determinando un totale di 240 thread per ogni clock, per 1000 elementi basterebbero cinque cicli di esecuzione. Il programmatore non deve preoccuparsi di creare, gestire e terminare i thread e ciò offre la possibilità di compilare il programma sorgente una sola volta per poi eseguirlo su differenti tipi di GPU, ciascuno con un diverso numero di core.
Si possono schematizzare le differenze tra le due componenti hardware nel seguente modo:

- Scopo di progettazione: Un core CPU è progettato per eseguire un flusso di istruzioni il più velocemente possibile mentre le GPU sono proget-

tate per eseguire il più velocemente possibile molti flussi di istruzioni paralleli.

- Uso dei transistor: La CPU usa i transistor su caratteristiche hardware come il riordinamento delle istruzioni nel buffer che hanno lo scopo di velocizzare l'esecuzione di un singolo thread, mentre la GPU sfrutta i transistors per gli array di processori, hardware multithreading, memoria condivisa e controllori multipli di memoria. Tutte queste operazioni non sono mirate a velocizzare l'esecuzione di un particolare thread, quanto piuttosto consentono al chip di supportare decine di migliaia di thread concorrentemente sostenendo un'ampia larghezza di banda di memoria.

- Ruolo della cache: La CPU utilizza la cache per migliorare le prestazioni riducendo la latenza degli accessi alla memoria. La GPU, invece, per amplificare l'ampiezza della banda.

- Gestione della latenza: La CPU gestisce la latenza utilizzando una cache molto ampia necessitando spesso di maggiore potenza; le GPU gestiscono la latenza sostenendo migliaia di thread alla volta e, se un particolare thread sta aspettando per essere caricato in memoria, la GPU può passare da un thread all'altro senza registrare ritardi o rallentamenti.

- Multithreading: Le CPU supportano uno o due thread per core ad un costo di centinaia di cicli mentre le GPU sono in grado di supportare 1024 thread per multiprocessore e non registrano alcun costo aggiuntivo significativo nel passaggio tra un thread all'altro.

Figura 3.6: CPU vs GPU

- Controllore della memoria: Le CPUs Intel non hanno controllore di memoria; le GPU ne possono impiegare fino ad un massimo di otto con conseguente ampiezza di banda, che risulta 10 volte maggiore di quella delle tradizionali CPU.

Attualmente le architetture di CPU e GPU sono completamente diverse. Le GPUs odierne eccedono la complessità delle tradizionali CPUs in termini di numero di transistors e stanno diventando sempre più programmabili in modo da rendere possibile il loro impiego in applicazioni diverse dalla grafica. Nelle GPU la maggior parte della circuiteria è dedicata al data processing, mentre nelle CPU buona parte è legata al flow control e al data caching; di conseguenza le CPU non sono ideali nell'elaborazione di processi che possono essere divisi in piccoli elementi processabili in parallelo, ambito di elaborazione nel quale invece le GPU possono eccellere in quanto si tratta, per loro natura, del tipico pacchetto di dati elaborati in un'applicazione grafica 3D. Confrontando le strutture di base di CPU e GPU, da notare in figura 3.6, si evidenzia chiaramente come la maggior parte dei transistor implementati all'interno di una GPU siano dedicati alle unità di esecuzione, mentre in una CPU la parte di circuiteria dedicata sia al controllo che alla cache è ben più importante. Del resto, un processore deve eseguire un singolo task il più

velocemente possibile, mentre una GPU deve cercare di eseguire il maggior numero di processi in parallelo, minimizzando per ciascuno il tempo di esecuzione e massimizzando il quantitativo di dati analizzati nell'unità di tempo. Tutte le applicazioni che sono basate su insiemi di dati molto ampi ed omogenei, per le quali si richiede l'esecuzione di un elevato numero di operazioni aritmetiche, possono beneficiare dell'utilizzo di una GPU in modo massiccio. In questo scenario, infatti, uno stesso programma può essere ripetuto su vari pacchetti di dati in parallelo, con un elevato rapporto di operazioni aritmetiche eseguite rispetto a quelle sulla memoria.

Le GPU si differenziano anche per un altro aspetto, vale a dire la coerenza nell'accesso alla memoria: quando un texel viene letto, pochi cicli dopo verrà letto il texel vicino, e quando un pixel viene scritto, pochi cicli dopo verrà scritto il pixel vicino. Organizzando la memoria in maniera intelligente, le prestazioni si possono avvicinare molto al bandwidth teorico. Questo significa che una GPU, diversamente da una CPU, non ha bisogno di molta cache. Tuttavia, il rapporto tra CPU e GPU è sempre in evoluzione e per ognuna delle due componenti aumentano le prestazioni e le funzionalità. In passato, se un utente voleva un PC più veloce, aveva bisogno di un processore più veloce ma nell' attuale epoca del 3D, con lo sviluppo delle interfacce grafiche innovative, l'alta risoluzione per i giochi, l'accelerazione video HD e la parallelizzazione a livello di thread, una CPU tradizionale non è in grado di soddisfare le richieste degli utenti e, in molti casi, sono proprio le GPU aggiornate che hanno un maggiore impatto sulle prestazioni del sistema.

3.5 CUDA

CUDA (Compute Unified Device Architecture), rilasciata da NVIDIA, è una piattaforma software realizzata per la risoluzione ad elevate prestazioni di problemi intrinsecamente paralleli sfruttando la potenza di calcolo delle GPU. Formalmente introdotta nel Novembre 2006, dopo un anno di testing in versione beta, Cuda è stata immediatamente apprezzata per lo sviluppo di applicazioni in campo scientifico e tecnologico. Allo stesso tempo, Nvidia riprogettava le sue GPU in modo da renderle dei dispositivi versatili, utilizzabili cioè in vari ambiti, oltre a quello naturale della grafica 3D.

L'idea di NVIDIA è stata di migliorare il modello di programmazione basato sulle GPU concentrandosi in particolare sugli shaders. Come è mostrato dalla figura 3.7, CUDA include tool in linguaggio C/C++ relativi allo sviluppo software, librerie di funzioni ed un meccanismo di astrazione hardware che permette di nascondere i dettagli hardware delle GPU agli sviluppatori. Sebbene CUDA richieda ai programmatori di scrivere un codice speciale per l'elaborazione parallela, non è necessario gestire esplicitamente i thread in senso convenzionale, opportunità questa che semplifica enormemente il modello di programmazione. I tool di sviluppo CUDA operano tramite un compilatore C/C++, cosicchè i programmatori possono integrare codice scritto per le GPU con un codice scritto per la CPU.

Il modello di programmazione CUDA differisce in maniera significativa dal codice basato su thread singoli della CPU e perfino dal codice parallelo per le GPU prima dell'avvento di CUDA. In un modello basato sul singolo thread, la CPU è in grado di recuperare un singolo flusso di istruzioni che operano in modo seriale sui dati e una CPU superscalare è in grado di instradare

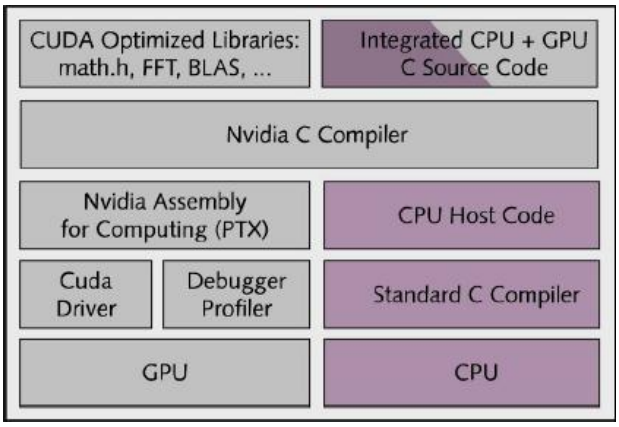

Figura 3.7: Piattaforma NVIDIA CUDA

tale flusso attraverso delle pipeline multiple, ma è ancora previsto un singolo flusso di istruzioni e il grado di parallelismo è fortemente limitato dai dati e dalle loro dipendenze logiche.

CUDA è altamente parallelo e divide l'insieme dei dati in piccole parti conservati in memoria sul chip GPU consentendone la condivisione ai processori multipli di thread. Tale meccanismo di memorizzazione a livello locale riduce di molto la necessità di accedere al di là della memoria del chip, migliorando pertanto le prestazioni.

Nel modello CUDA, un thread in stallo entra in una coda inattiva ed è sostituito da un altro thread che è pronto per essere eseguito: quando i dati relativi al thread precedentemente in stallo diventano nuovamente disponibili, allora il thread entra in un'altra coda la quale segnala che esso è pronto per essere avviato. I gruppi di thread sono gestiti tramite un modello di schedulazione round-robin, assicurando che ciascuno entri in esecuzione senza ritardare gli altri.

Un'ulteriore caratteristica di CUDA riguarda la gestione automatica dei th-

read che risulta estremamente importante quando il loro numero supera le migliaia; nonostante i thread siano leggeri, nel senso che ciascuno opera su una quantità ristretta di dati, essi sono da concepire nel senso classico del termine: ciascun thread infatti è dotato di un proprio stack, dei file di registro, di un program counter e di una memoria locale. Si è sottolineato il fatto che la gestione dei thread è automatizzata da CUDA, ma questo non significa che il programmatore sia completamente esonerato dal pensare ai thread; più precisamente, è necessario analizzare il problema da risolvere per determinare il modo migliore di distribuire i dati tra i diversi multiprocessori della GPU, quindi pensare al numero di thread e di blocchi che terrà la GPU completamente impegnata nell'esecuzione. L'unico limite relativo al numero di thread da coinvolgere, oltre che al numero di thread per processore, è il numero di registri che ogni singolo thread richiede. Il compilatore CUDA è in grado automaticamente di determinare il numero ottimale di registri per ciascun thread e, per raggiungere il massimo possibile pari a 12.288 thread in una Geforce della serie 8 con 128 processori il compilatore non può assegnare più di 10 registri per thread.

3.5.1 Background

Nel 2003 AMD e Intel erano in concorrenza per sviluppare processori sempre più potenti. In pochi anni le velocità di clock aumentarono vertiginosamente, specialmente quando Intel presentò i processori Pentium 4. La corsa delle frequenze però raggiunse presto i suoi limiti: tra il 2001 e il 2003 i Pentium 4 passarono da 1.5 Ghz a 3.0 GHz mentre negli ultimi anni si sta assistendo ad incrementi minimi (tra il 2003 e il 2005 la velocità massima è passata

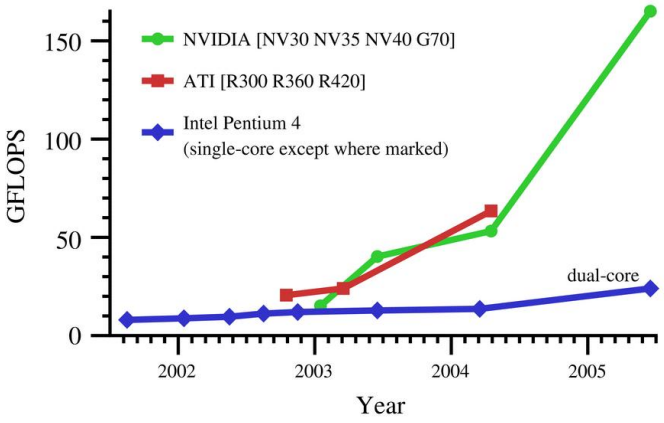

Figura 3.8: Legge di Moore

da 3 a 3.8 GHz). Anche le architetture ottimizzate per elevate velocità di clock, come la Prescott, si trovarono in difficoltà, questa volta per ragioni al di là del mercato; alcuni avevano anche profetizzato la fine della legge di Moore, ma erano ben lontani dalla realtà, a causa di interpretazioni sbagliate della legge, che si basa sulla quantità di transistor presenti in una data superficie di silicio. La confusione è dovuta al fatto che, per molto tempo, più transistor significava più prestazioni. Successivamente, però, la situazione divenne più complessa. I progettisti di CPU dovevano affrontare la legge detta "diminishing returns" (principio in cui, dopo un certo punto, aumentare solo il fattore di produzione, lasciando inalterate le altre caratteristiche, non avrebbe portato a miglioramenti dei risultati); l'effetto concreto era che ci sarebbero voluti troppi transistor per mantenere la proporzione dell'aumento delle prestazioni, e la produzione era semplicemente impossibile. Nel frattempo, mentre i produttori di CPU cercavano di trovare una soluzione ai loro problemi, i produttori di GPU continuavano a beneficiare sempre di più della legge di Moore che si vede illustata nella figura 3.8.

Il motivo per il quale i problemi riscontrati con le CPU non riguardavano le GPU è molto semplice: le CPU sono progettate per offrire le massime prestazioni gestendo un flusso di istruzioni che operano su dati differenti.
Per gestire questa mole di lavoro i progettisti si orientavano all'esecuzione di più istruzioni in parallelo per ottimizzare al massimo l'uso delle unità di calcolo. Il problema è che c'è sempre un limite al parallelismo in quanto esiste un unico flusso di istruzioni, e l'incremento del numero di unità di calcolo potrebbe far rimanere queste ultime inutilizzate per la maggior parte del tempo.

3.5.2 Le API CUDA

Come si può notare dal grafico raffigurato in figura 3.9, CUDA offre due API di rilievo:

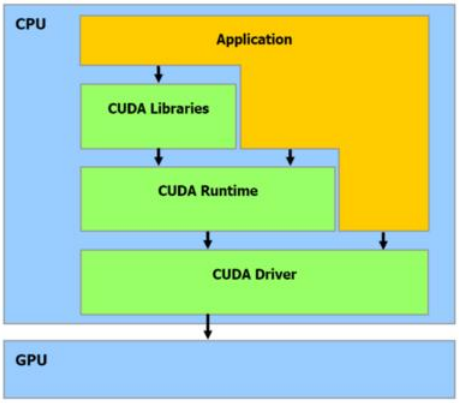

Figura 3.9: API CUDA

- Una API di alto livello: la CUDA Runtime API
- Una API di basso livello: la CUDA Driver API

Poichè la API di alto livello si trova nel livello superiore dell'API di basso livello, ogni chiamata alla funzione Runtime è divisa in due istruzioni fondamentali, gestite dall'API Driver. Queste ultime si escludono mutuamente, nel senso che il programmatore può vedere l'uno o l'altra. La API Runtime offre funzioni molto pratiche per l'inizializzazione e il context management, mentre la API Driver è più complessa ed ha il vantaggio della flessibilità che fornisce ai programmatori maggiore controllo.

CUDA può servire per generare risorse (trasformazioni geometriche, procedural textures e altro), che possono essere passate alle API grafiche, o al contrario, è possibile che una API 3D invii i suoi risultati di rendering a CUDA, che può essere usato come scopo di post-processing. Ci sono numerosi esempi di integrazioni, e il vantaggio è che le risorse rimangono immagazzinate nella RAM della GPU, senza dover passare tramite il bus PCI-Express, che farebbe da collo di bottiglia.

La condivisione di risorse potrebbe presentare alcuni problemi. Per esempio, per cambiare la risoluzione o la profondità dei colori, i dati grafici hanno la priorità, quindi, se le risorse del frame buffer devono essere aumentate, il driver non esiterà a prendere le risorse allocate da un'applicazione che usa CUDA creando crash di varia natura. Infine, il terzo livello software è il set di librerie CUBLAS, che dispone di una serie di parti fondamentali per i calcoli di algebra lineare e CUFFT, che calcola trasformate di Fourier, un algoritmo molto utilizzato nel campo del signal processing.

3.5.3 Alcune definizioni

Nella presentazione dell'ambiente CUDA, è fondamentale definire alcuni termini che ricorrono spesso nella documentazione di NVIDIA e che si utilizzano nella programmazione in CUDA. Innanzitutto si definisce il thread: esso non è sinonimo di thread per CPU ed è un elemento composto dai dati base di una elaborazione; in aggiunta i thread, diversamente da quelli delle CPU, sono notevolmente più leggeri, per cui il cambio di contesto tra due thread coinvolti nella stessa operazione non è molto impegnativa.

Un'altra definizione importante è quella di warp: un gruppo di 32 thread, che è la minima dimensione di dati elaborati in SIMD da un multiprocessore CUDA. Tuttavia, in CUDA, anzichè manipolare direttamente warp si lavora con block (blocchi) che possono contenere dai 64 ai 512 thread. Un insieme di blocchi forma un grid, ovvero una griglia: il vantaggio di questo raggruppamento è che il numero di blocchi elaborati simultaneamente dalla GPU è vicino al limite delle risorse hardware. Si noti la figura 3.10 per avere un dettaglio aggiuntivo sulla struttura di un CUDA device.

Il numero di blocchi all'interno di una griglia rende possibile l'applicazione del kernel ad una grande quantità di thread con una singola chiamata, senza doversi preoccupare delle risorse. Se l'hardware dispone di poche risorse, allora eseguirà i blocchi sequenzialmente; viceversa, potrà elaborare i dati in parallelo. Ciò significa che lo stesso codice può essere gestito, in maniera adeguata, sia da GPU entry-level sia da GPU high-end e anche dai modelli che seguiranno.

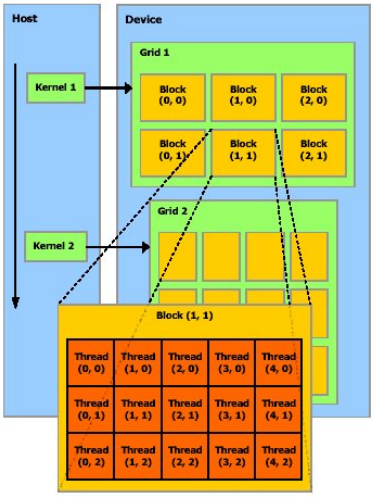

Figura 3.10: Struttura CUDA Device

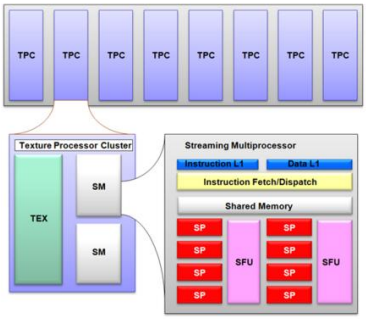

Figura 3.11: Shader core NVIDIA

3.5.4 CUDA Hardware

Il Shader Core NVIDIA, illustrato in figura 3.11, è costituito da diversi cluster, che sono chiamati Texture Processor Clusters: ogni cluster è a sua volta costituito da un'unità texture e due multiprocessori streaming (SM).
Questi processori sono costituiti da un front end che lancia le istruzioni, e da un back end, costituito da un gruppo di otto unità di calcolo e due SFU (Super Funcion Units), che eseguono l'istruzione in modalità SIMD. La stes-

sa istruzione è applicata a tutti i thread del warp. Nvidia chiama questa modalità SIMT (Single Instruction Multiple Threads). E' importante puntualizzare che il back end opera a frequenza doppia rispetto al front end.
La modalità operativa dei multiprocessori streaming è la seguente: ad ogni ciclo si effettua un'operazione di lettura di un warp dal front end, che lancia l'esecuzione di un'istruzione. Per applicare l'istruzione a tutti i 32 thread del warp, il back end impiega quattro cicli, ma siccome opera a frequenza doppia del front end, possiamo dire che ne impiega solo due. Quindi, per evitare che il front end rimanga inutilizzato per un ciclo e per massimizzare l'uso dell'hardware, l'ideale è alternare il tipo di istruzione a ogni ciclo - un'istruzione classica per un ciclo e un'istruzione SFU per l'altro. Ogni multiprocessore dispone di una certa quantità di risorse e di una piccola quantità di memoria chiamata Shared Memory, equivalente a 16 Kb per multiprocessore: non si tratta di una memoria cache in senso classico, perchè il programmatore ha pieno controllo di gestione su questa memoria. Questo dettaglio è particolarmente interessante, e dimostra che CUDA è veramente un set di tecnologie hardware e software. Quest'area di memoria permette ai thread presenti nello stesso blocco di comunicare tra loro. E' importante enfatizzare questo concetto: tutti i thread presenti nello stesso blocco sono eseguiti dallo stesso multiprocessore. L'assegnamento dei blocchi ai vari multiprocessori, invece, è del tutto separato, il che significa che due thread di blocchi differenti non possono comunicare tra loro durante la loro esecuzione. L'uso della memoria, quindi è piuttosto complicato, ma non per questo è poco efficiente. Eccezione fatta per i casi in cui diversi thread cerchino di accedere allo stesso banco di memoria, creando conflitti, l'accesso alla memoria condivisa è veloce quanto

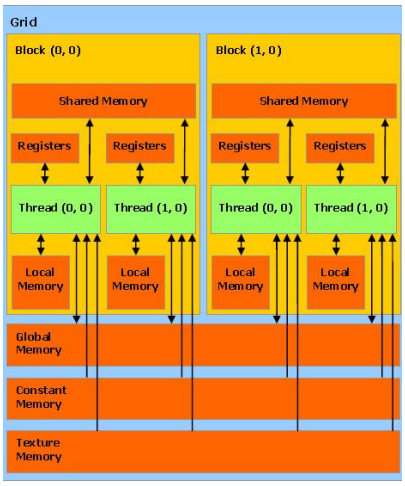

Figura 3.12: Accesso alla memoria e ai registri

l'accesso ai registri. Si veda la figura 3.12.

I multiprocessori hanno anche 8192 registri condivisi tra tutti i thread di tutti i blocchi attivi in un multiprocessore. Il numero dei blocchi attivi per multiprocessore non può essere maggiore di otto, e il numero dei warp attivi è limitato a 24 (768 thread). Ottimizzare un programma CUDA, quindi, vuol dire inizialmente trovare il miglior rapporto tra il numero di blocchi e la loro dimensione - più thread per blocco saranno utili per mascherare la latenza delle operazioni in memoria, ma allo stesso tempo il numero di registri disponibili per ogni thread è ridotto. Un blocco di 512 thread, inoltre, risulterebbe particolarmente inefficiente, poichè su un multiprocessore può essere attivo un solo blocco, e potenzialmente si sprecherebbero 256 thread. Quindi, Nvidia consiglia di usare blocchi composti da 128 a 256 threads, dimensione che offre il miglior compromesso per mascherare la latenza e utilizzare il numero di registri necessario per la maggior parte dei kernel.

3.5.5 CUDA Software

Da un punto di vista software, CUDA consiste in un set di estensione del linguaggio C, nel quale sono definiti i qualificatori da applicare alle funzioni e alle variabili. Più precisamente, le funzioni prendono il nome di **kernel** che devono essere dichiarati con un valore di ritorno di tipo *void* e che sono richiamabili ed eseguibili in differenti modalità dalla CPU host e dal dispositivo CUDA. Tra queste si ha:

- **global**, che quando applicata a una funzione indica che l'ultima è un kernel, una funzione cioè che sarà chiamata dalla CPU ed eseguita dalla GPU.
- **device** designa una funzione che sarà eseguita dalla GPU e che può essere chiamata solo dalla GPU (cioè da un'altra funzione identificata device o global).
- **host**, opzionale, designa una funziona chiamata dalla CPU ed eseguita solo dalla CPU (una funzione tradizionale).

E' fondamentale chiarire come l'host possa invocare un kernel, cioè come specificare il nome del kernel e la configurazione di esecuzione, ovvero, definire il numero di thread paralleli all'interno di un blocco e il numero di blocchi da usare quando viene eseguito un kernel sul dispositivo CUDA. Inoltre, è necessario comprendere come sincronizzare i vari kernel e il codice sul dispositivo host CPU. La chiamata ad un generico kernel viene effettuata specificando sia il nome del kernel stesso sia la configurazione di esecuzione, racchiusa nelle triplici parentesi angolari. I parametri che giocano un forte ruolo in tale configurazione sono per l'esattezza due, ovvero, **nBlocks** e **blocksize**;

nBlocks specifica il numero di blocchi nella griglia mentre blocksize definisce il numero di threads per ciascun blocco. Nel calcolo del numero di blocchi, nel caso in cui N non sia un numero perfettamente divisibile per il blocksize, l'ultimo termine nel calcolo di nBlocks aggiunge un blocco extra il quale implica che alcuni thread nel blocco non eseguiranno molto probabilmente alcun lavoro utile. Gli argomenti di chiamata del kernel vengono specificati in parentesi tonda.

Segue un generico esempio di invocazione di kernel in CUDA:

```
KernelName<<<nBlocks, blocksize>>>(input parameters);
int nBlocks=N/blocksize + (N%blocksize==0?0:1)
```

Il dispositivo CUDA risulta inattivo nel momento in cui viene richiamato il kernel sulla base del modello di esecuzione specificato e sulla base dei parametri di input coinvolti. Nel frattempo, l'host continua le sue istruzioni; a questo punto, sia il dispositivo CUDA che l'host sono simultaneamente in esecuzione con i rispettivi programmi separati. Il ragionamento legato alla scelta di nBlocks e di blocksize è abbastanza elegante perchè permette allo sviluppatore di tener conto delle limitazioni hardware senza richiedere la ricompilazione dell'applicazione. Nel kernel associato al dispositivo CUDA, diverse variabili built-in sono coinvolte nella configurazione di esecuzione:

- **blockIdx** che contiene l'indice del blocco all'interno della griglia.
- **threadIdx** che contiene l'indice del thread all'interno del blocco.
- **blockDim** che contiene il numero di thread all'interno del blocco.

Per spiegare l'utilizzo di queste 3 variabili, si mostra un esempio del calcolo dell'*idx*, ovvero dell'indice identificativo di ciascun thread all'interno di un

blocco, relativo al calcolo di un array i cui valori sono incrementati di una unità.

In ambiente CUDA, il kernel che risolve questo problema è il seguente:

```
__global__ void incrementArrayOnDevice(float *a, int N)
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  if (idx<N)
    a[idx] = a[idx]+1.f;
}
```

Il kernel prende in input l'array a ed N, che ne rappresenta la dimensione, calcola l'indice relativo a ciascun thread controllando che esso risulti minore di N, accede al contenuto dell'elemento $a[idx]$ e lo incrementa di un'unità. Se consideriamo a un vettore di $N = 10$ elementi, per comprendere come viene interpretato il singolo idx con $blockDim = 4$, la situazione sarebbe quella visualizzata in figura 3.13. In particolare si considerano 3 blocchi ognuno di 4 elementi (si prende la parte intera di 10/4), per un totale di 12 thread. Poichè gli elementi dell'array sono 10, due thread risultano inattivi; pertanto il controllo sull'idx e la scelta del numero di blocchi diviene fondamentale per la gestione del problema da risolvere in parallelo e può notevolmente influire sull'efficienza complessiva della computazione. Infine è opportuno osservare che ogni chiamata CUDA restituisce un codice di errore di tipo cudaError.

Uno dei risultati più importanti raggiunto da CUDA è il migliore uso delle risorse della memoria locale del multiprocessore; è importante notare che i thread all'interno di un blocco sono in grado di comunicare gli uni con gli

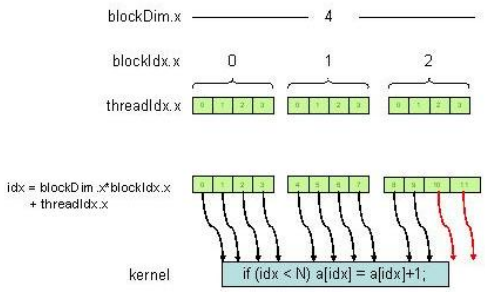

Figura 3.13: Calcolo dell'idx

altri attraverso tali risorse, perché il modello di esecuzione CUDA specifica che un blocco può essere processato solo su un unico multi-processore. In altre parole, i dati scritti in memoria condivisa all'interno di un blocco sono accessibili da tutti gli altri thread all'interno dello stesso blocco, ma non da un thread facente parte di un altro blocco. La memoria condivisa con queste caratteristiche può essere implementata in modo molto efficiente dal punto di vista hardware, il che si traduce in una elevata velocità di accesso.

Altre chiamate API, come estensioni C, sono la cudaMalloc() e cudaFree() per l'allocazione di memoria specifica per CUDA, e cudaMemcpy per il trasferimento dei dati dalla CPU alla GPU e viceversa. Nel caso di trasferimento dati dalla CPU alla GPU si ha:

```
CudaError_t cudaMemcpy(void *dest,
                       const void * src,
                       size_t count,
                       cudaMemcpyHostToDevice);
```

Nel caso di trasferimento dati dalla GPU alla CPU si ha:

```
CudaError_t cudaMemcpy(void *dest,
                       const void * src,
                       size_t count,
                       cudaMemcpyDeviceToHost);
```

In entrambi i casi alla cudaMemcpy viene passato l'indirizzo puntato da dest, che specifica dove viene copiato il dato, l'indirizzo dell'elemento puntato da src che specifica la sorgente della copia, la dimensione del dato da trasferire, ovvero il size e il tipo di copia tra i dispositivi coinvolti. Infine una API speciale, **syncthreads** fornisce una barriera di sincronizzazione esplicita che impedisce ai thread di agire sui dati che un altro thread sta simultaneamente usando.

3.5.6 Compilazione CUDA

La compilazione di un programma CUDA richiede fasi aggiuntive, sia perché il programma tiene conto di due architetture di processori differenti, (GPU e CPU), sia a causa dell'astrazione hardware di CUDA.
Come è mostrato in figura 3.13, lo stesso codice sorgente include codice C/C++ scritto sia per la GPU che per la CPU.
Il primo passo è quello di separare il codice sorgente per ogni architettura: a tal fine, NVIDIA fornisce un preprocessore front-end C++ sviluppato dall' Edison Design Group (EDG). Questi analizza il codice sorgente e crea diversi file per le due architetture.
Per la CPU, EDG crea dei file sorgenti di estensione .cpp, pronti per la compilazione sia con Microsoft che con il compilatore GNU C/C++; per i

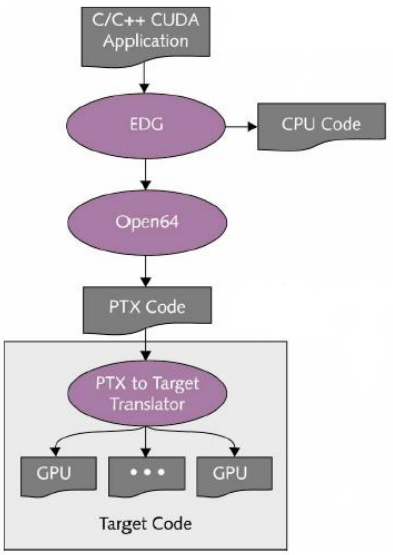

Figura 3.14: Accesso alla memoria e ai registri

processori grafici NVIDIA, EDG crea una diversa serie di file .cpp che vengono poi elaborati utilizzando Open64, un compilatore open-source C/C+ originariamente pensato per l'architettura Intel Itanium. L'implementazione di Open64 della NVIDIA genera un output scritto in un linguaggio simile all'assembler all'interno di file in formato Parallel Thread Execution (PTX); quesi file non sono specifici per una particolare GPU della NVIDIA ed è proprio grazie ad essi che lo stesso codice sorgente potrà essere eseguito sulle future GPU progettate, per esempio, per avere una maggiore quantità di memoria e un maggior numero di thread disponibili per ogni processore. Infine, l'ultima fase traduce questo linguaggio intermedio in comandi specifici per la GPU che poi viene incapsulato in forma binaria nell'eseguibile.

Il compilatore CUDA prende il nome di **nvcc**; le fasi che caratterizzano una compilazione di un programma realizzato in ambiente CUDA, sono espresse in pratica tramite una combinazione di opzioni fornite a riga di comando,

specificando il nome del file di input comprensivo di suffisso **.cu** e altre opzioni.

nvcc non effettua alcuna distinzione tra i file oggetto, di libreria oppure di risorsa; semplicemente passa i file di questi tipi al linker quando la fase di linking viene eseguita.

```
nvcc SourceFile.cu -[options]
```

Tra le opzioni più comuni di nvcc si ricordano in particolare -o[filename] per memorizzare l'eseguibile con il nome specificato da filename, e -deviceemu per emulare l'esecuzione del codice senza sfruttare la GPU, utile per effettuare operazioni di debugging sul dispositivo host.

Capitolo 4

Implementazione di un modello di Fish-Schooling

L'osservazione di gruppi di animali, di cui un esempio è il banco di pesci, oggetto di studio in questo lavoro di tesi, è un fenomeno affascinante perchè gli individui che costituiscono l'insieme si muovono come se avessero un'unica mente, suscitando un interesse che colpisce non solo gli studiosi ma anche le persone comuni.

Nel capitolo, organizzato in 6 sezioni, si discuterà in maniera approfondita del modello oggetto di studio di questa tesi e, in particolare nella prima sezione si parla dell'evoluzione del modello, facendo riferimento ad alcune versioni di particolare rilievo; nella seconda sezione si tratta il problema della generazione dei numeri pseudo casuali, approfondendo la tecnica del *Mersenne-Twister*; nella terza sezione si presenta il modello recentemente sviluppato da Linda Petzold e il suo gruppo; nella quarta e nella quinta si discute l'implementazione del modello realizzato durante il lavoro di tesi, chiamato FS-Parth (acronimo di Fish-Schooling Parthenope), rispettivamen-

te nel linguaggio MATLAB e CUDA; nella quinta ed ultima sezione, invece, si analizzano le simulazioni ottenute dall'esecuzione del programma CUDA.

4.1 Modellazione del Fish-Schooling - Cenni storici

Il primo riferimento al problema risale al 1927, quando Hantarou Nagaoka, un fisico famoso per la teoria del modello atomico diede a Morisaburo e Tauti un riconoscimento per aver effettuato un'analisi fisica nell'esposizione del fish-schooling.

Nel 1973, Onaka Sumiko e Saburo Sakai ipotizzarono le dinamiche dei pesci da un punto di vista sempre fisico ma eseguendo simulazioni del loro comportamento utilizzando il computer e mostrando che un banco è formato e mantenuto da una attrazione e un allineamento reciproco determinati da una velocità di moto. Dopo queste ricerche pioneristiche, negli anni 90́, le analisi e le simulazioni relative al fenomeno del banco di pesci evolsero radicalmente e portarono alla formulazione di modelli più complessi; l'interazione locale tra ogni individuo (ovvero il singolo pesce) provoca il costituirsi di un ordine globale all'interno del gruppo. A partire da questo principio nacque il concetto di sistema auto-organizzato di gruppo senza un leader.

La ricerca in questo ambito ha raggiunto poi un nuovo stadio grazie all'accelerazione e ai vantaggi tecnologici computazionali, consentendo di eseguire un gran numero di simulazioni per analizzare una varietà di condizioni realistiche. Nel 1986, Reynolds elaborò un modello computazionale relativo al comportamento animale, in particolare riferito a stormi di uccelli o banchi di pesci. Di seguito, i dettagli di tale modello.

4.1.1 Craig Reynolds - Sistema Boid

Craig Reynolds è un esperto di vita artificiale e di grafica computazionale. Ha realizzato software per le simulazioni di differenti comportamenti animali e umani, legati principalmente al movimento.
Nel 1986 realizzò un modello computazionale riferito al movimento coordinato degli animali; il modello, inizialmente concepito per la descrizione degli animali in volo, si è rivelato successivamente perfettamene adattabile alle specie marine, per cui può senz'altro essere integrato nella storia del modello di Fish-School. La simulazione fu chiamata dallo stesso Reynolds "Boids" e fu sviluppata come applet Java per illustrare come i movimenti del singolo individuo erano influenzati dalle posizioni e dalle velocità degli altri vicini. In particolare si introdussero tre concetti:

1. **Separazione**: per evitare affollamenti locali negli allineamenti tra gli individui.

2. **Allineamento**: orientarsi nella direzione dei vicini più prossimi in moto coeso.

3. **Coesione**: orientarsi verso le posizioni degli individui locali.

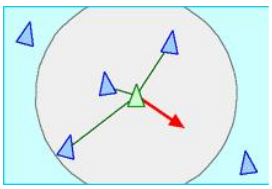

Figura 4.1: Separazione

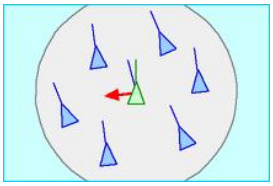

Figura 4.2: Allineamento

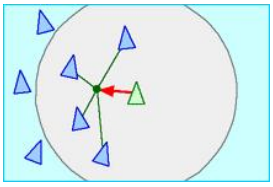

Figura 4.3: Coesione

In figura 4.1, 4.2 e 4.3 sono rappresentate le tre situazioni.
Ciò era equivalente ad affermare i seguenti principi validi per ciascun individuo considerato nel modello:

1. Mantenere una distanza mininma da tutti gli altri individui.

2. Mantenere una velocità dipendente da quella dei propri vicini.

3. Muoversi verso il centro dei Boids nelle vicinanze.

Ciascun boid, come descritto dal modello di Reynolds (fare riferimento al sito http://www.red3d.com/cwr/boids/ per visualizzare l'applet Java) è in grado di osservare in maniera diretta l'intera geometria del sistema, quindi la propria e quella dei suoi vicini; il vicino è caratterizzato da una distanza e da un angolo, data dalla direzione di volo (nel caso degli uccelli) e dalla direzione di nuoto (nel caso dei pesci) e tutti quelli che si trovano al di fuori delle vicinanze locali sono ignorati (figura 4.4). In genere, i comportamenti

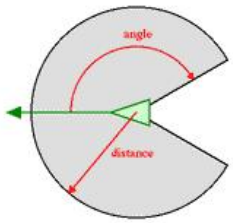

Figura 4.4: Direzione di un individuo nel banco

di gruppo sono intrinsecamente non lineari e le dinamiche emergenti hanno un aspetto localmente caotico ma globalmente ordinato. Una significativa proprietà del comportamento degli organismi viventi è l'imprevedibilità su piccole scale temporali. Ad esempio, in un particolare istante di tempo, i boid potrebbero inizialmente volare/nuotare da sinistra verso destra; potrebbe essere del tutto impossibile prevedere in quale direzione si spostino cinque minuti dopo. In una scala di tempi molto brevi il movimento è abbastanza prevedibile: l'individuo è portato a muoversi nella stessa direzione del suo vicino. Tale proprietà è unica nei sistemi complessi e va in contrasto sia con il comportamento caotico (per i quali sia per tempi brevi che lunghi non si riesce a prevedere nulla) sia comportamenti ordinati (statici o periodici). Tale principio è ben sintetizzato da una osservazione di Langton (1990) che sostiene che "i fenomeni legati agli esseri viventi sono in bilico sul bordo del caos". Il modello boid è un esempio di *individual-based model*: una classe di simulazioni impiegata per catturare il comportamento globale di un vasto numero di interazioni tra agenti autonomi. Questo tipo di modelli è stato utilizzato in biologia, ecologia, economia e altri campi applicativi. La complessità di tempo dell'algoritmo implementato da Reynolds è $O(n^2)$ per ogni passo temporale; ciascun boid ha bisogno di considerare l'altro simile, anche

se solo per determinare se si trova nelle sue vicinanze. E' tuttavia possibile ridurre questo costo a circa $O(n)$ tramite l'utilizzo di una struttura dati spaziale che consenta a ciascun boid di essere ordinato nella propria locazione in modo da esaminare solo la porzione dell'insieme che è in media nelle vicinanze. Nell'ambito degli IBE (individual-based ecology), le dimostrazioni sono solo il primo passo dal quale si vuole procedere per imparare qualcosa inerente al mondo reale, il che è riportato nel prossimo esempio di fish-school model, sviluppato da Huth e Wissel.

4.1.2 Huth & Wissel - Fish School

Il comportamento dei pesci illustrato nel modello di Huth e Wissel (1992,1993,1994; Huth 1992), Camazine 2001 (capitolo 11) è un altro esempio di sistema biologico capace di auto-organizzazione.
Il progetto di modellizzazione iniziò con un intenso lavoro di ricerca e una serie di osservazioni empiriche e teoriche relative al banco di pesci (Huth, 1992). Questa indagine aveva due principali obiettivi:

- Identificare le teorie alternative concernenti i comportamenti dei pesci nella formazione del banco.
- Identificare patterns che potevano essere utilizzati nella fase di testing in seguito all'implementazione del modello.

Le teorie testate dai due studiosi sostenevano che gli individui potessero avvertire solo una vicinanza limitata. Il range di percezione di un pesce è diviso in tre zone circolari, caratterizzate da un loro raggio r_1, r_2, r_3. Se un pesce individua un altro simile all'interno della sua zona di repulsione, ovvero la

distanza r dal vicino è più piccola del raggio r_1, il pesce ruota di 90 gradi per evitare la collisione con il suo vicino; il pesce ruota per nuotare in parallelo con gli altri vicini individuati nella zona parallela con $r_1 < r < r_2$; il pesce nuota nella direzione dei vicini nella zona di attrazione con $r_2 < r < r_3$.
Queste tre zone corrispondono ai tre principi generali di Reynolds e, in particolare, alla separazione, all'allineamento e alla coesione.
Tuttavia, le regole comportamentali presentate nel modello Huth-Wissel, a differenza del Boid, sono stocastiche: l'angolo di rotazione e la nuova velocità scelte da un pesce ad ogni passo temporale sono casualmente estratte da una distribuzione normale, la cui media è specifica per le tre diverse tipologie di movimento. Poichè il grado di stocasticità nel comportamento individuale non è a priori noto, le deviazioni standard di queste distribuzioni normali sono parametri del modello i cui valori devono essere calibrati da parte di ciascun pesce in funzione dell'influenza sulle proprietà del banco.
In sintesi, i principi sui quali si fonda tale modello, sono di seguito riportati:

- Posizione di preferenza: il singolo individuo non ha una particolare preferenza di posizione nello stare in banco.
- Distanza: la distanza da un pesce all'altro è determinata in funzione del più prossimo.
- Posizione dei vicini: ogni posizione è quantificata come l'angolo che si forma tra le direzioni di moto dei vicini sia in verticale che in orizzontale.

4.2 Generazione di numeri pseudo-casuali

I numeri pseudo casuali sono utilizzati per costruire simulazioni di natura probabilistica di fenomeni fisici (reattori nucleari, traffico stradale, aereodinamica), di problemi decisionali e finanziari (econometrica, previsione Dow-Jones), informatica (progettazione VLSI, rendering) o come semplice fonte di divertimento (videogiochi).

Il forte legame che esiste tra il gioco e le simulazioni probabilistiche è sottolineato dal fatto che a tali simulazioni viene generalmente dato il nome di metodi Monte Carlo (in onore del famoso casino a Monaco). L'idea di utilizzare in modo sistematico simulazioni di tipo probabilistico per risolvere un problema di natura fisica viene generalmente attribuita al matematico polacco Stanislaw Ulam (1909-1984) [1]. Può sembrare assurdo utilizzare un computer per generare numeri casuali: esso, infatti, è una macchina deterministica in grado di produrre un output perfettamente predicibile a priori. In effetti nessun calcolatore è in grado di generare numeri puramente casuali, ma solo numeri pseudo-casuali o quasi-casuali ossia numeri generati da algoritmi numerici deterministici in grado di superare una serie di test statistici che conferiscono a tali numeri una apparente casualità. Tale algoritmo è detto generatore di numeri pseudo-casuali (PRNG, dall'inglese pseudo-random

[1] Ulam fu uno dei personaggi chiave nel progetto americano per la costruzione della bomba atomica (Manhattan project) durante la seconda guerra mondiale tra il 1943 ed il 1945 a Los Alamos, New Mexico (dopo la guerra, Ulam diede contributi essenziali anche nello sviluppo della bomba a fusione di idrogeno o bomba H).

Il progetto Manhattan richiedeva infatti la risoluzione di un enorme numero di problemi incredibilmente complessi (nella sua autobiografia Ulam descrive come l'idea di utilizzare simulazioni casuali per risolvere tali problemi gli sia venuta mentre giocava a carte).

numbers generator).
Questi numeri si distinguono dalle sequenze casuali (random), ovvero sequenze di numeri prodotte da fenomeni fisici intrinsecamente aleatori (ad es.: tempo trascorso fra due impulsi successivi di un contatore), e devono soddisfare, al minimo, le seguenti proprietà statistiche:

- distribuzione degli elementi della sequenza secondo una funzione di distribuzione predefinita $f(x)$: di solito si richiede una distribuzione uniforme su un intervallo specificato (equidistribuzione), cioè $f(x) = \frac{1}{x_{max}-x_{min}}$ nell'intervallo $[x_{min}, x_{max}]$ e $f(x) = 0$ fuori da tale intervallo.
- indipendenza tra elementi successivi della sequenza: se la funzione di distribuzione per un singolo elemento è $f(x)$, la funzione di distribuzione per le coppie di elementi successivi deve essere $F(x, y) = f(x) \cdot f(y)$.

Considerando nello specifico il modello proposto nella tesi, si ha che per generare le condizioni iniziali e aggiungere la componente casuale al modello ad ogni istante temporale bisogna generare numeri casuali uniformemente distribuiti. In particolare, si utilizza un semplice generatore di numeri pseudo casuali su GPU; la scelta è ricaduta sul Mersenne Twister di cui si parla nella sezione che segue.

4.2.1 Mersenne Twister

Sviluppato nel 1997 da Makoto Matsumoto e Takuji Nishimura, il generatore MT provvede alla generazione veloce di numeri pseudo casuali di qualità elevata.
Le sue principali caratteristiche riguardano:

- Periodo più lungo e migliore equidistribuzione rispetto a qualsiasi altro generatore; questo significa che la correlazione fra valori successivi della sequenza è praticamente trascurabile.

- Generazione veloce: la velocità di generazione di numeri casuali è sostanzialmente uguale a quella di rand() presente nella libreria standard dell'ANSI C.

- Uso efficiente della memoria: la versione in C del generatore è costituita da 624 linee di codice.

Il generatore MT è stato progettato rispettando le condizioni pratiche che un buon generatore di numeri random deve soddisfare; non esistono infatti metodi matematici rigorosi per dimostrare l'assenza di difetti di un generatore e una della prove più forti per selezionare un buon generatore è il test spettrale.
MT è in grado di superare un test di questo tipo, ed in particolare il k-distribution test: se si osserva un periodo dell'uscita del generatore con 32 bit di accuratezza, le n-uple di numeri con $n = 623$, si distribuiscono uniformemente in uno spazio a 623 dimensioni. Analogamente, con un'accuratezza a 16 bit, le n-uple di numeri con $n = 1246$, si distribuiscono uniformemente in uno spazio a 1246 dimensioni.
Il Mersenne Twister è un algoritmo per la generazione di numeri pseudo-casuali di tipo lineare congruenziale. Un generatore di questo tipo è della forma:

$$Z_{i+1} = (a \cdot Z_i + c)(mod m)$$

dove:

- *mod* indica il resto della divisione intera tra $(a \cdot Z_i + c)$ e m;
- a e c sono due numeri prefissati detti rispettivamente moltiplicatore ed incremento;
- Z_0 è il primo valore della sequenza, detto seme.

Il generatore si dice moltiplicativo se $c = 0$; in questo caso la formula diventa $Z_{i+1} = (a \cdot Z_i)(modm)$. La scelta dei valori assegnati ai parametri a, c, m, e Z_0, è determinante per il livello di casualità della sequenza. Occorre inoltre ricordare che la sequenza generata è periodica, con periodo p, tale che $p \leq m$. Tutti i numeri generati saranno compresi tra 0 e $m - 1$; nel caso la sequenza numerica generata comprendesse tutti i numeri in questo intervallo, si ha il cosiddetto periodo pieno. Il generatore MT ha un periodo $p = 2^{19937} - 1$ ed è il migliore e più veloce generatore di numeri casuali attualmente conosciuto.

4.3 Modello stocastico di Petzold

Di seguito viene riportata la descrizione del modello stocastico di Linda Petzold del Dipartimento di Informatica dell'Università della California, Santa Barbara.

Si considera un modello bidimensionale a scala di individui per il moto collettivo di un banco di pesci sulla base di interazioni locali comportamentali. Il gruppo è costituito da N individui, ognuno con posizione $p_i(t)$, e direzione di velocità $\hat{v}_i(t)$, velocità costante s. Ad ogni passo temporale di dimensione τ, gli individui determinano simultaneamente una nuova direzione di moto considerando i vicini all'interno di due zone comportamentali. La prima zona, spesso denominata "zona di repulsione", è rappresentata da un cerchio

di raggio r_r, prefissato, relativo ad un individuo: i vari organismi coinvolti respingono i vicini che si trovano all'interno di questa zona.

La seconda zona, una "zona di orientamento e di attrazione", è rappresentata da un anello di raggio interno r_r e raggio esterno r_p relativo al singolo individuo; essa include un'area cieca, definita come un settore circolare con un angolo al centro pari a $(2\pi - \eta)$, in prossimità della quale i vicini all'interno della stessa non sono percepibili; la costante η è prefissata.

Queste zone sono utilizzate per definire le leggi comportamentali di ciascun pesce in moto all'interno del banco. Innanzitutto, se l'individuo i trova gli agenti all'interno della sua zona di repulsione, allora esso orienta la sua direzione lontano dalle direzioni medie relative a questi ultimi nella sua zona di repulsione. La sua direzione desiderata di moto al passo temporale successivo è data da

$$v_i(t+\tau) = -\sum_{j \neq i} \frac{p_j(t) - p_i(t)}{|p_j(t) - p_i(t)|} \tag{4.1}$$

Questo vettore è normalizzato come $\hat{v}_i(t+\tau) = \frac{v_i(t+\tau)}{|v_i(t+\tau)|}$ assumendo che $v_i(t+\tau) \neq 0$. Nel caso in cui $v_i(t+\tau) = 0$, l'agente i mantiene la sua direzione corrente di moto al passo successivo, pertanto $\hat{v}_i(t+\tau) = \hat{v}_i(t)$.

Viceversa, se non vengono trovati individui nella zona di repulsione dell'individuo i, allora quest'ultimo si allinea con gli individui vicini e, in particolare, si muove spinto da un'attrazione nei confronti di quelli che si trovano nella zona di orientamento e attrazione. La direzione di moto desiderata al passo che segue, sarà data dalla somma pesata dei due termini a destra:

$$v_i(t+\tau) = \omega_a \frac{\sum_{j \neq i} \frac{p_j(t) - p_i(t)}{|p_j(t) - p_i(t)|}}{\left| \sum_{j \neq i} \frac{p_j(t) - p_i(t)}{|p_j(t) - p_i(t)|} \right|} + \omega_o \frac{\sum_j \hat{v}_j(t)}{\left| \sum_j \hat{v}_j(t) \right|} \tag{4.2}$$

dove ω_a e ω_o sono rispettivamente i pesi di attrazione e orientamento.
Questo vettore è poi normalizzato come $\hat{v}_i(t+\tau) = \frac{v_i(t+\tau)}{|v_i(t+\tau)|}$, assumendo che $v_i(t+\tau) \neq 0$. Come prima, se $v_i(t+\tau) = 0$, allora l'agente i mantiene la sua direzione di moto corrente.
Si definisce $r = \omega_o/\omega_a$ come rapporto tra le tendenze di orientamento e attrazione di ciascun individuo. Quando $r = 0$ ($\omega_o = 0$), gli individui non sono orientati dai loro vicini; non appena il valore di r si avvicina ad 1, gli individui bilanciano il loro orientamento e manifestano la loro preferenza di attrazione. Per $r > 1$, gli individui sono più interessati all'orientamento verso i loro vicini piuttosto che all'attrazione.
Gli effetti stocastici sono incorporati nel modello ruotando la direzione desiderata $\hat{v}_i(t+\tau)$ dell'agente i di un angolo chiamato σ. Inoltre, dal momento in cui gli individui possono solo ruotare di $\theta\tau$ radianti in un passo temporale, se l'angolo tra $\hat{v}_i(t)$ e $\hat{v}_i(t+\tau)$ è maggiore di $\theta\tau$, gli individui non raggiungono la direzione desiderata e ruotano di $\theta\tau$ radianti. Infine, la posizione di ciascun individuo è aggiornata simultaneamente nel seguente modo:

$$p_i(t+\tau) = p_i(t) + s\hat{v}_i(t+\tau)\tau. \tag{4.3}$$

La Petzold ha anche presentato un'implementazione su GPU del suo modello.

4.4 Modello stocastico FS-Parth in Matlab

FS-Parth è l'implementazione sviluppata in questa tesi del modello proposto da Petzold. La prima implementazione che si discute è quella in Matlab.
L'algoritmo implementato nel linguaggio Matlab è costituito da 5 m-files principali:

1. **fish_schooling.m**
    - visualizza_grafici.m
    - non_visualizzare_grafici.m
2. **Norma.m**;
3. **distanza.m**
4. **normvitau.m**
5. **angvet.m**

I file sono eseguiti dallo script Matlab fish_schooling.m, il quale consente di gestire, tramite una interfaccia grafica, la scelta di visualizzare i grafici prodotti ad ogni passo temporale. In figura 4.5 è mostrata la finestra che appare nel momento in cui è mandato in esecuzione lo script fish_schooling.

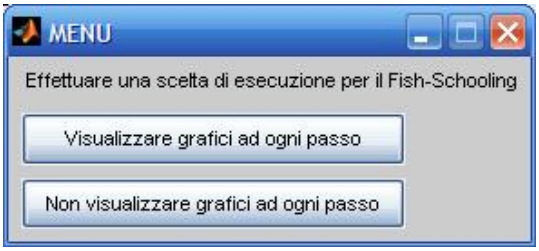

Figura 4.5: Interfaccia grafica

Nel caso in cui si sceglie di riprodurre i risultati grafici, viene richiamato lo script *visualizza_grafici.m*, viceversa, *non_visualizzare_grafici.m*. Ciascuno di essi richiama tutte le funzioni di cui sopra. In particolare, il programma coinvolge 4 strutture dati principali: la matrice delle posizioni iniziali, generata in maniera casuale tramite la *rand*, la matrice delle posizioni successive, inizializzata a 0, la matrice delle velocità iniziali, generata anch'essa casualmente

con la *rand* e la matrice delle velocità successive, inizializzata a 0. Tutte le matrici hanno dimensioni pari a $2 \cdot N$, dove N è il numero totale di individui. Le righe rappresentano rispettivamente la coordinata x e la coordinata y di ciascun individuo mentre il numero delle colonne è il numero di individui che si considerano nell'intera simulazione. Dopo l'inizializzazione di alcuni parametri costanti e la dichiarazione e la definizione delle matrici appena viste, viene chiamata la funzione Norma.m che prende in input la matrice di cui calcolare la norma dei suoi vettori colonna e il numero di colonne della matrice stessa.

All'interno di un costrutto ripetivo *for* che scorre sui passi temporali compresi nell'intervallo temporale di simulazione $[0, T]$, si richiama la funzione distanza.m che prende in input la matrice delle posizioni iniziali e il numero di colonne della stessa, per calcolare la distanza tra l' *i-simo* individuo e tutti gli altri; l'output è dato da una matrice delle distanze D in cui l'elemento (i, j) è la distanza tra l'i-simo e il j-simo individuo. Per tale motivo, la matrice D è quadrata, simmetrica, di dimensione $N \cdot N$.

Il gruppo è costituito da N individui con posizioni $p_i(t)$, direzioni di velocità $\hat{v}_i(t)$, velocità costante s. Ad ogni passo temporale di dimensione τ, in riferimento al modello di Petzold, gli individui determinano simultaneamente una nuova direzione di moto considerando i vicini all'interno delle due zone, "zona di repulsione" e "zona di orientamento e di attrazione".

In Matlab, la prima situazione, ovvero la zona di repulsione, è gestita calcolando un vettore che contenga gli indici degli elementi per i quali è rispettata la condizione:

```
indrep=find(D(i,:)<=rr & D(i,:)~=0);
```

ovvero, la distanza tra l'*i-simo* individuo e tutti gli altri deve risultare minore o uguale al raggio interno e deve essere diversa da 0: quest'ultimo caso, infatti individua lo stesso individuo. Se il vettore che contiene gli indici relativi alla zona di repulsione, non è nullo, allora per tutti i p_j, ovvero le posizioni degli individui corrispondenti agli indici determinati, viene calcolato il vettore $v_i(t+\tau)$ (4.1).

Questo vettore è normalizzato come $\hat{v}_i(t+\tau) = \frac{v_i(t+\tau)}{|v_i(t+\tau)|}$ assumendo che $v_i(t+\tau) \neq 0$. Nel caso in cui $v_i(t+\tau) = 0$, l'agente i mantiene la sua direzione corrente di moto al passo successivo, pertanto $\hat{v}_i(t+\tau) = \hat{v}_i(t)$.

Viceversa, se il vettore *indrep* ha dimensione nulla, si determina un nuovo vettore che contenga gli indici di attrazione nel seguente modo:

```
indattr=find(D(i,:)>rr & D(i,:)<=rp & D(i,:)~=0);
```

Questa volta la condizione è leggermente cambiata per quanto riguarda il valore del raggio da considerare in quanto la distanza tra l'*i-simo* individuo e i restanti deve essere maggiore del raggio interno e minore o uguale al raggio esterno, oltre che diverso da 0.

Come nel caso precedente, se il vettore *indattr* non è nullo, si procede al calcolo del vettore $v_i(t+\tau)$ come (4.2).

Questo vettore è poi normalizzato come $\hat{v}_i(t+\tau) = \frac{v_i(t+\tau)}{|v_i(t+\tau)|}$, assumendo che $v_i(t+\tau) \neq 0$. Come prima, se $v_i(t+\tau) = 0$, allora l'agente i mantiene la sua direzione di moto corrente.

Si considera inoltre la "zona di influenza" come la zona in cui sono presenti quegli individui che non si trovano nè nella zona di attrazione nè nella zona di repulsione; per essi la direzione di velocità al passo successivo $v_i(t+\tau)$ rimane invariata alla precedente, per cui si ha $v_i(t+tau) = v_i(t)$.

Nel modello MATLAB, gli effetti stocastici si realizzano mediante la rotazione della direzione desiderata $\hat{v}_i(t+\tau)$ dell'agente i di un angolo prefissato σ. La rotazione è un'operazione comune a tutti e tre i casi (attrazione, repulsione, non influenza), e si concretizza con la generazione di un angolo casuale che corrisponde alla perturbazione dell'angolo stesso, chiamato *phi*. Il massimo angolo di rotazione per passo temporale è una costante predefinita, denotata con θ. La matrice di rotazione standard è rappresentata da:

$$\begin{pmatrix} cos(\theta\tau) & sin(\theta\tau) \\ -sin(\theta\tau) & cos(\theta\tau) \end{pmatrix}$$

Un costrutto di selezione verifica se ci troviamo nella zona di influenza oppure no; nel primo caso l'angolo considerato z è uguale a 0, nel secondo caso, invece, si richiama la funzione angvet.m che prende in input i vettori $v_i(t)$ e $v_i(t+\tau)$ e ne determina l'angolo (con segno).

Inoltre, dal momento in cui gli individui possono solo ruotare di $\theta\tau$ radianti in un passo temporale, se l'angolo tra $\hat{v}_i(t)$ e $\hat{v}_i(t+\tau)$ è maggiore di $\theta\tau$, gli individui non raggiungono la direzione desiderata e ruotano di $\theta\tau$ radianti, ovvero viene utilizzata la matrice di rotazione standard. La nuova direzione di velocità $v_i(t+\tau)$ è data dal prodotto tra la matrice di rotazione calcolata e il vettore della direzione di velocità precedenti $v_i(t)$. Infine, la posizione di ciascun individuo è aggiornata simultaneamente come in 4.3. Segue la visualizzazione dei risultati utilizzando la function Matlab quiver sulle posizioni e sulle velocità calcolate per ogni individuo ad ogni passo. Prima di uscire dal ciclo ripetitivo sui passi temporali, si aggiornano le strutture dati delle posizioni e delle direzioni di velocità. Infine, si visualizza il grafico relativo alla polarizzazione, in funzione del tempo, ovvero il grado dell'allineamento

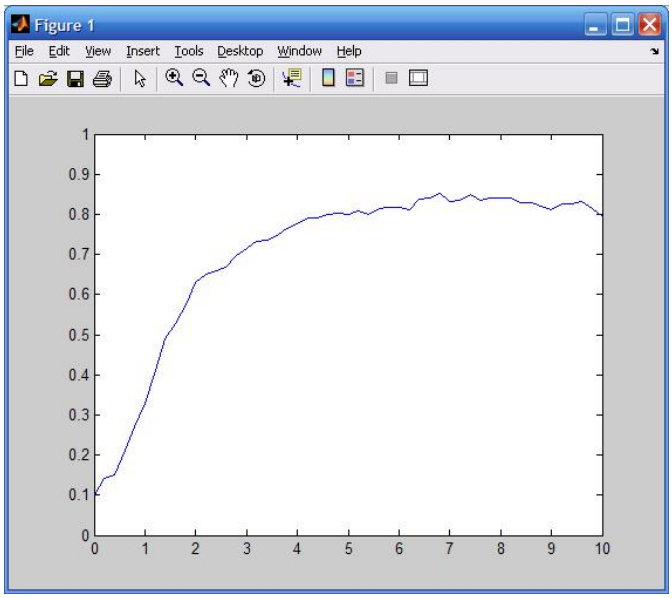

Figura 4.6: Polarizzazione in una realizzazione

degli individui in gruppo, misurato nel seguente modo:

$$P(t) = \frac{1}{N} \cdot |\sum_{i=1}^{N} v_i(t)| \tag{4.4}$$

La polarizzazione può assumere un valore compreso tra 0 e 1. L'implementazione utilizza un vettore POL, di T/τ elementi, che, ad ogni passo temporale da 0 a T, assume il valore calcolato dalla 4.4. In figura 4.6 è mostrata la polarizzazione ottenuta da una singola realizzazione dell'algoritmo. Infine, un ultimo script Matlab chiamato *Realization.m*, esegue per 1120 volte l'algoritmo fish_schooling e visualizza in un grafico le medie sulle realizzazioni dei valori della polarizzazione a ogni passo temporale. In particolare, viene utilizzata una matrice di 1120 righe e di $T/tau + 1$ colonne che viene aggiornata con il vettore delle Polarizzazioni calcolato ad ogni passo. Il grafico mostrato in figura 4.7 riporta sull'asse delle ascisse i passi temporali e sull'asse delle ordinate y le medie delle polarizzazioni ad ogni passo temporale.

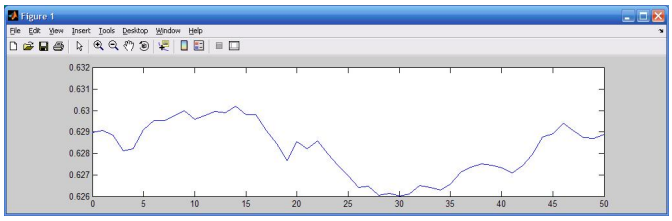

Figura 4.7: 1120 Realizzazioni

Per maggiori dettagli sull'implementazione in Matlab, si faccia riferimento all'appendice A.

4.5 Modello stocastico FS-Parth in CUDA

In questa sezione viene presentata l'implementazione di FS-Parth sviluppata nel linguaggio CUDA. L'implementazione consta di due file:

1. **MyCudaLib.h**

2. **Fish_Schooling.cu**

MyCudaLib.h è una libreria del Fish_Schooling.cu che contiene tutti i kernel e le function utili per l'applicazione. E' stata progettata per separare i kernel dal main, in modo da rendere più elegante e leggibile l'intero codice che così organizzato è di più immediata comprensione.
Le prime istruzioni riguardano la definizione e l'inizializzazione dei parametri costanti che sono utilizzati nel corso dell'esecuzione dei singoli kernel: la costante PIGRECO, l'angolo θ, il passo temporale τ, l'angolo σ espresso in radianti e corrispondente a 115°, la velocità costante s con cui si muovono gli individui all'interno del gruppo, il raggio interno r_r, il raggio esterno r_p, il peso di orientamento w_o, il peso di attrazione w_a e il numero di individui

M da considerare nella simulazione.

Come nell'implementazione Matlab, le strutture dati principali sono 4: la matrice delle posizioni iniziali, la matrice delle posizioni finali, la matrice delle velocità iniziali e la matrice delle velocità finali, tutte aventi 2 righe ed M colonne.

Fish_Schooling.cu è il main in cui sono incluse le librerie standard del C, la libreria sviluppata ad hoc per l'applicazione, ovvero *MyCudaLib.h*, e le librerie utili per la generazione dei numeri casuali con Mersenne Twister e per il calcolo dei tempi di esecuzione del programma, ovvero *cutil.h*.

Nel main sono ovviamente dichiarate tutte le variabili definite nei vari kernel, sono calcolate le dimensione della griglia, in termini di numero di blocchi e numero di threads, e sono generati i numeri casuali tramite la GPU corrispondenti alle posizioni iniziali degli individui, alle velocità degli stessi, (nei termini di coordinata x e y) ed infine relativi all'angolo *phi* utile nell'implementazione della rotazione dell'angolo tra due vettori.

Per fare in modo che ad ogni esecuzione del programma si abbia una sequenza di numeri casuali diversa, il seed della GPU viene inizializzato nel seguente modo:

```
srand(time(NULL));
seedMTGPU(rand());
```

Le istruzioni che, invece hanno permesso di generare i numeri random da parte della GPU sono state:

```
seedMTGPU(rand());
RandomGPU<<<1,2*M>>>(A_d,2*M);
seedMTGPU(rand());
```

```
RandomGPU<<<1,2*M>>>(V_d, 2*M);
calc_val_cas <<<1,2*M>>>(A_d, V_d);
RandomGPU<<<1,M>>>(angolo_d ,M);
BoxMullerGPU<<<1,M>>>(angolo_d ,M);
```

Il kernel *RandomGPU* richiede che siano specificati in input il nome della variabile in cui verranno memorizzati i valori generati, e la sua dimensione, mentre tra parentesi angolari bisogna precisare rispettivamente il numero di blocchi e il numero di thread da utilizzare per l'esecuzione del kernel.
Nel caso in cui si calcolano i valori per le posizioni *A_d* e le velocità *V_d*, come si può facilmente notare, non è richiamato il kernel *BoxMullerGPU*, il quale, invocato solo per la generazione dell'angolo casuale *angolo_d*, a seguito della *RandomGPU*, consente di trasformare i numeri da una distribuzione uniforme ad una distribuzione gaussiana sulla GPU.
Seguendo un approccio di tipo parallelo, si ha la necessità di calcolare il numero di blocchi e il numero di thread presenti in ciascun blocco, in modo da non superare la quantità massima supportata dalla GPU che è pari a 512.
Dopo la dichiarazione, l'inizializzazione e l'allocazione di tutte le variabili utili sia all'host che al device, il primo compito del programma è il trasferimento dei dati dalla CPU alla GPU utilizzando la *cudaMemcpy* dall'*HostToDevice*.
Dopo aver generato $2 \cdot M$ numeri casuali, viene richiamato il primo kernel CUDA, chiamato *calc_val_cas*, il quale prende in input la matrice delle posizioni iniziali e la matrice delle velocità inizali e fa in modo che ciascun thread, uno per ogni individuo, cioè uno per ogni colonna delle matrici, generi il proprio numero casuale compreso in un intervallo limitato: le posizioni assumono un valore compreso tra $]-10, 10[$ mentre le direzioni di velocità sono vettori uni-

tari con componenti comprese tra $[-1, 1]$, calcolate richiamando l'apposito kernel *Norma*; ogni singolo thread calcola la norma euclidea della velocità di un singolo individuo e, in seguito, un costrutto ripetitivo *for* sui passi temporali da 0 a T con passo τ gestisce l'intera simulazione. Viene calcolata la matrice delle distanze invocando il kernel *DistMat*, il quale prende in input la matrice delle posizioni, la matrice in cui memorizzare l'output ed M; questo kernel viene richiamato su dimGrid e dimBlock calcolati in precedenza, corrispondenti al numero di blocchi e al numero di thread, in base alla dimensione M del problema.

A questo punto si determinano gli indici relativi alla zona di attrazione e gli indici relativi alla zona di repulsione, a partire dalla matrice delle distanze. A tal proposito, sono stati sviluppati due kernel, rispettivamente chiamati *find_rep* e *find_attr* che prendono in input la matrice delle distanze, la dimensione M e generano un vettore in cui vengono memorizzati gli indici degli individui appartenenti alle due zone.

In particolare, nel caso della repulsione, si verifica se ogni elemento della matrice delle distanze risulti minore del raggio interno r_r e diverso da 0, mentre, nel caso dell'attrazione, si controlla se il singolo elemento della matrice delle distanze risulti compreso tra il raggio interno r_r e il raggio esterno r_p e sia diverso da 0.

L'idea per il calcolo dei vettori *indrep* e *indattr* è la seguente: ogni riga i della matrice delle distanze rappresenta le distanze dall'i-simo individuo; per ognuna di esse il singolo thread si preoccupa di determinare gli indici di repulsione e di attrazione che vengono contati e salvati nei rispettivi vettori di dimensione $(M+1) \cdot M$ in modo che la $M+1$ locazione rispetto all'i-sima riga

contenga il numero complessivo degli indici relativi alle due zone considerate. Dopo aver determinato gli indici della zona sia di repulsione che di attrazione, il kernel *Gestfind* si preoccupa di calcolare la matrice delle velocità finali sulla base delle considerazioni viste nell'implementazione del modello MATLAB. Il kernel prende in input il vettore in cui sono stati memorizzati gli indici di repulsione, il vettore in cui sono stati memorizzati gli indici di attrazione, la matrice delle posizioni iniziali, M, la matrice delle velocità successive che rappresenta l'output, la matrice delle velocità iniziali normalizzate, e un vettore contenente M angoli casuali generati nel main.
Anche in questo caso, ogni thread si occupa del singolo individuo coinvolto nell'intera simulazione e i passi gestiti dal kernel sono sintetizzati come segue:

1. Se il vettore contenente gli indici di repulsione non è vuoto, si calcolano le velocità successive come in (4.1); altrimenti, si verifica il vettore contenente gli indici di attrazione: se esso non è nullo, si determinano le velocità successive come in (4.2. Se non ci sono individui nè nella zona di repulsione nè nella zona di attrazione, la velocità resta invariata, a meno di effetti stocastici.

2. Le nuove velocità calcolate vengono normalizzate richiamando il kernel *normvitau*.

3. Viene generato per ogni individuo un angolo ϕ casuale, seguito dal calcolo dell'angolo che si forma tra i due vettori velocità iniziale e velocità successiva presunta, chiamato z. In particolare, se si considera la zona di influenza l'angolo $z = 0$, altrimenti viene richiamato il kernel *calc_angolo* che prende in input i due vettori tra i quali determinare l'angolo stesso, M, e z nel quale andrà memorizzato l'output.

4. Si calcola l'angolo $phiz = \phi + z$ per determinare le velocità finali a partire dal prodotto tra la matrice di rotazione e la velocità iniziale, nelle modalità viste prima nell'implementazione MATLAB.

L'ultimo kernel ad essere richiamato è *AggPos*, che prende in input la matrice delle posizioni iniziali, la matrice delle velocità successive e la matrice delle posizioni successive che ne rappresenta l'output. Si determinano le posizioni successive in base alla (4.3).

L'algoritmo termina con il trasferimento dei dati dalla GPU alla CPU, invocando la *cudaMemcpy* dal *DeviceToHost*, e con la stampa delle variabili relative alle posizioni e alle velocità finali degli M individui. Infine, si dealloca la memoria occupata dalle variabili stesse facendo differenza tra quelle relative all'host e quelle relative al device e si termina il main con un *return EXIT_SUCCESS*.

Tutti i kernel nel main, ovvero in *Fish_Schooling.cu*, vengono eseguiti $k = (T/\tau + 1)$ volte mediante un costrutto ripetitivo *for* che parte da 0 e con passo *tau* arriva fino a T. All'interno di questo ciclo, si aggiornano di volta in volta le variabili che devono essere richiamate dai vari kernel utilizzando la *cudaMemcpy* dal *DeviceToDevice*, e vengono memorizzati su file i valori relativi alle due matrici (matrice delle posizioni e matrice delle velocità), che rappresentano l'output da visualizzare.

Al termine dell'esecuzione dell'intero programma CUDA, per visualizzare i grafici, è stato realizzato uno script in Matlab chiamato *Grafici.m*, il cui codice è in Listing 4.1.

Listing 4.1: Grafici.m

```
function Grafici(T)
```

```
for i=0:T
    file1=['p' num2str(i) '.txt'];
    file2=['v' num2str(i) '.txt'];
    p=load(file1);
    v=load(file2);
    quiver(p(1,:),p(2,:),v(1,:),v(2,:));
    pause()
end
```

In input viene passato il numero complessivo di passi temporali con cui viene eseguito l'algoritmo; un ciclo for consente di caricare i file prodotti dall'esecuzione CUDA rispettivamente delle posizioni e delle velocità degli individui coinvolti. Il valore memorizzato nelle variabili relative al caricamento di detti file, p e v viene in seguito utilizzato per la rappresentazione grafica tramite *quiver*. E' importante chiarire che sebbene si faccia riferimento a strutture dati bidimensionali, in CUDA il problema è stato sviluppato dal punto di vista monodimensionale per cui se una matrice in Matlab è stata considerata come una $2 \cdot M$, in CUDA la stessa matrice è stata rappresentata come un vettore di $2 \cdot M$ elementi, gestito in maniera opportuna dai thread coinvolti nelle operazioni, facendo in modo che per M thread, ognuno si occupasse dell'elemento x e dell'elemento y di ciascun individuo, accedendo opportunamente agli elementi di detto vettore.

Per maggiori dettagli sulla implementazione del modello FS-Parth in CUDA, si faccia riferimento all'appendice B.

4.6 Analisi di Simulazioni stocastiche con FS-Parth

In seguito allo sviluppo dei due algoritmi FS-Parth rispettivamente in Matlab e in ambiente CUDA, si possono fare delle considerazioni utili sull'output prodotto dal modello e, in particolare, mettere in rilievo i risultati ottenuti, mostrando che essi simulano similmente il comportamento degli individui all'interno del banco.

Sono stati effettuati diversi test al variare di alcuni parametri in entrambi i codici e, nello specifico sono stati misurati i tempi di esecuzione dell'implementazione CUDA sia emulata, quindi sfruttando la CPU, sia eseguita direttamente sul device GPU.

Soffermandosi su quest'ultimo aspetto, si è considerato un numero complessivo di individui $N = 100$, ed un periodo compreso nell'intervallo $[0, T]$ partendo da un valore minimo di 50 passi temporali ad un massimo di 3000. I risultati delle simulazioni eseguite al variare di T hanno prodotto i risultati visibili nella tabella 4.1.

T	Tempi CPU in ms	Tempi GPU in ms	Steps	CPUtime/GPUtime
10	1569.772949	57.006001	50	27.54
50	7512.612793	296.052002	250	25.38
100	14965.041992	566.83308	500	26.40
200	30221.345703	1187.000977	1000	25.46
400	59904.882812	2306.558105	2000	26.00
600	90544.687500	3524.821045	3000	25.69

Tabella 4.1: Tempi di esecuzione CPU vs GPU

Le istruzioni che hanno permesso di ottenere i risultati mostrati, sono ripor-

tate in Listing 4.2.

Listing 4.2: Impostazioni timer

```
unsigned int timer=0;
cutCreateTimer(&timer);
cutStartTimer( timer );
...<istruzioni>...
cutStopTimer( timer );
printf(''CUDA execution time= %f ms\n'',
                cutGetTimerValue(timer));
```

Dopo aver inizializzato il timer al valore 0, esso viene impostato tramite il metodo *cutCreateTimer*, al seguito del quale parte il contatore del tempo iniziale di esecuzione invocando *cutStartTimer*. Eseguiti tutti i kernel, il timer viene fermato richiamando il metodo *cutStopTimer* mentre il *cutGetTimerValue* consente di restituire il valore stampato.

Nella prima colonna della tabella, sono stati riportati i valori di T, da un minimo di 10 ad un massimo di 600, tramite i quali è stato possibile determinare i passi temporali della quarta colonna.

La seconda colonna della tabella mostra i tempi di esecuzione dell'algortimo utilizzando la CPU e per ottenerli, da riga di comando è stata lanciata l'istruzione che segue:

```
nvcc <Fish_Schooling.cu> -o <file_exe> -deviceemu
```

T	Media Tempi CPU in ms	Media Tempi GPU in ms	Steps
10	1633.156982	58.441002	50
50	7591.4680176	295.7212005	250
100	14973.042091	574.424011	500
200	30567.813963	1190.608032	1000
400	60301.882812	2318.530029	2000
600	90873.271973	3587.726074	3000

Tabella 4.2: Media dei tempi di esecuzione

I tempi, invece, relativi all'esecuzione da parte della GPU sono stati presi compilando con *nvcc* ma naturalmente senza l'opzione *deviceemu*; come si può facilmente notare quelli relativi alla GPU sono molto ridotti rispetto a quelli registrati dalla CPU e questo mostra l'efficienza di questo strumento con il quale è possibile effettuare computazioni di calcolo scientifico ad elevate prestazioni.

Per avere un quadro ancor più generale relativo ai tempi di esecuzione dell'algoritmo si può pensare di calcolare la media dei tempi sia per la CPU che per la GPU.

Sulla base dei valori riportati in Tabella 4.1, avremo la situazione mostrata in Tabella 4.2, eseguendo l'algoritmo 10 volte per ogni T sia con CPU che con GPU. Per notare l'andamento grafico delle due situazioni mostrate in Tabella 4.2, è stato realizzato in Matlab un plot sulle medie relative ai tempi di esecuzione e i risultati ottenuti sono illustrati in figura 4.8.

E' interessante a questo punto osservare, riportando alcuni esempi, come varia l'output del programma FS-Parth sviluppato in Matlab ed in Cuda.

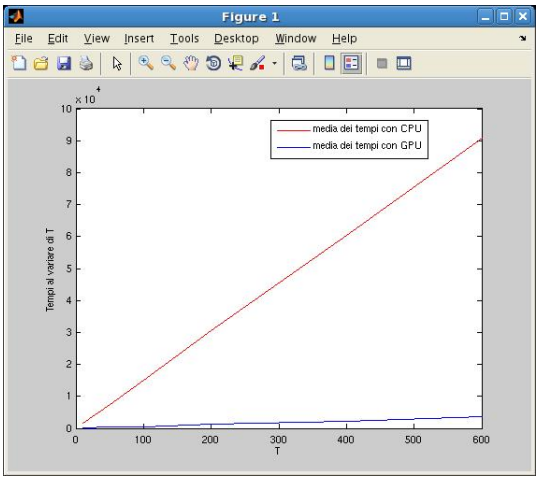

Figura 4.8: Grafico della media dei tempi di esecuzione CPU vs GPU

Sono stati effettuati diversi test modificando N, ovvero il numero di individui da considerare all'interno del modello, T, che è il periodo di esecuzione dell'algoritmo, e w_0 e w_a che rappresentano rispettivamente i parametri di orientamento e di attrazione che influenzano il comportamento degli individui in moto all'interno del banco.

Per $N = 10$, $T = 10$, $w_0 = 0.80$, $w_a = 0.20$, si ha la situazione illustrata in figura 4.9.

E' stata riportata la situazione iniziale come prima immagine, nella quale gli individui hanno posizioni e velocità generate in maniera casuale. Le immagini successive mostrano come essi si dispongano nello spazio, al variare del tempo, in modo da rispettare le condizioni di orientamento e di attrazione descritti nell'algoritmo.

Per $N = 100$, $T = 10$, $w_0 = 0.80$, $w_a = 0.20$ si ha quanto riportato in figura 4.10.

E'stato mostrato un caso in cui pesa maggiormente l'orientamento piutto-

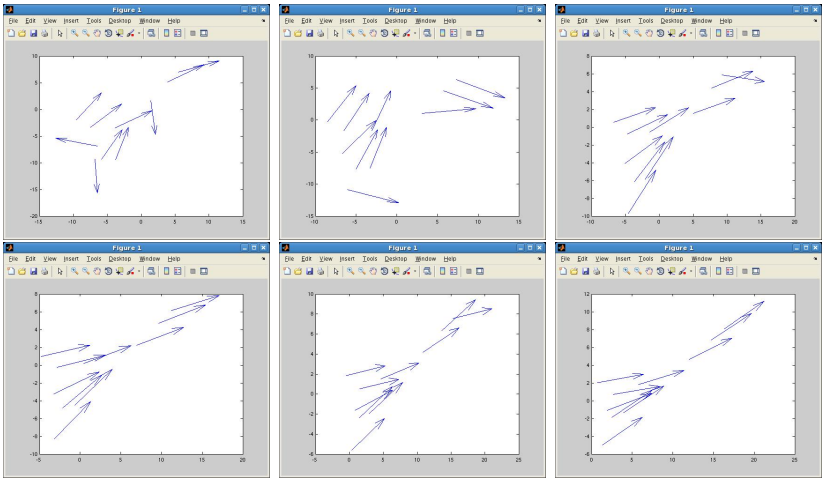

Figura 4.9: Passi successivi N=10, $T = 10$, $w_0 = 0.80$, $w_a = 0.20$

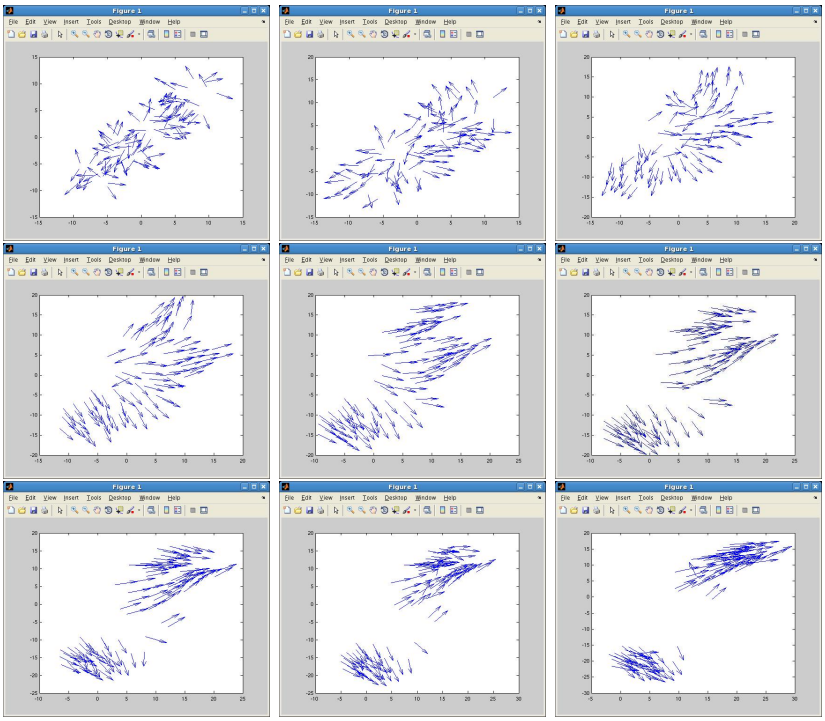

Figura 4.10: Passi successivi N=100, $T = 10$, $w_0 = 0.80$, $w_a = 0.20$

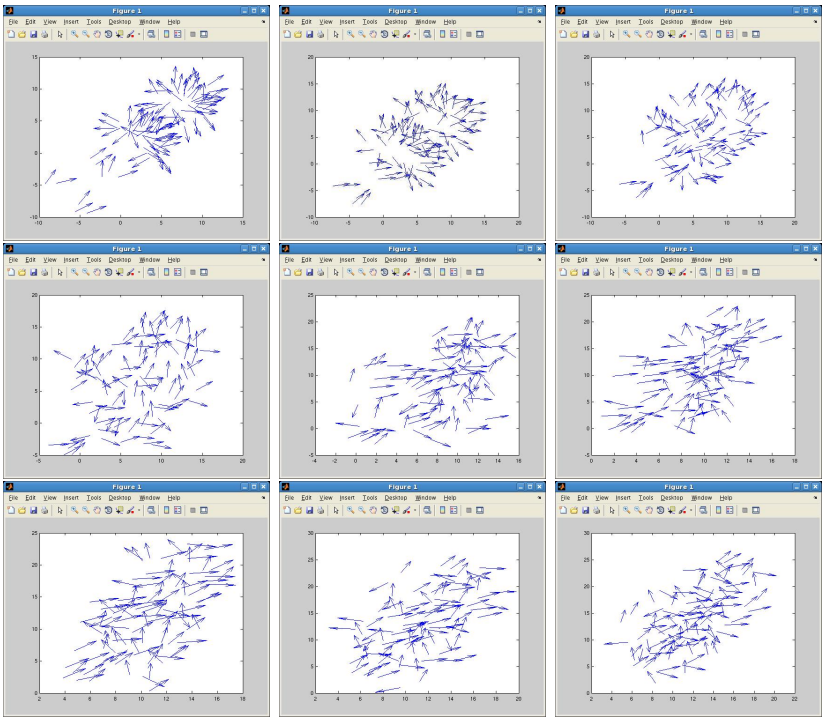

Figura 4.11: Passi successivi N=100, $T = 100$, $w_0 = 0.20$, $w_a = 0.80$

sto che l'attrazione, portando alla formazione di sottobanchi. Nella figura 4.11, è possibile visualizzare la situazione inversa per $N = 100$, $T = 100$, $w_0 = 0.20$, $w_a = 0.80$. La figura 4.12, invece, illustra la situazione in cui il peso di orientamento e il peso di attrazione assumono lo stesso valore ed influenzano similmente il comportamento degli individui coinvolti considerando dunque $N = 100$, $T = 100$, $w_0 = 0.50$, $w_a = 0.50$. Infine, si presenta il caso in cui il moto del banco di pesci è completamente influenzato dal peso di orientamento; in questa situazione, gli individui tendono a non toccarsi mai e a dirigersi verso una particolare direzione di moto. Ciò è rappresentato in figura 4.13 per $N = 100$, $T = 100$, $w_0 = 1.00$, $w_a = 0.00$. Nell'ambito delle simulazioni stocastiche con FS-Parth, rientrano le ultime ed interessanti considerazioni grafiche ottenute grazie all'utilizzo di uno strumento messo a

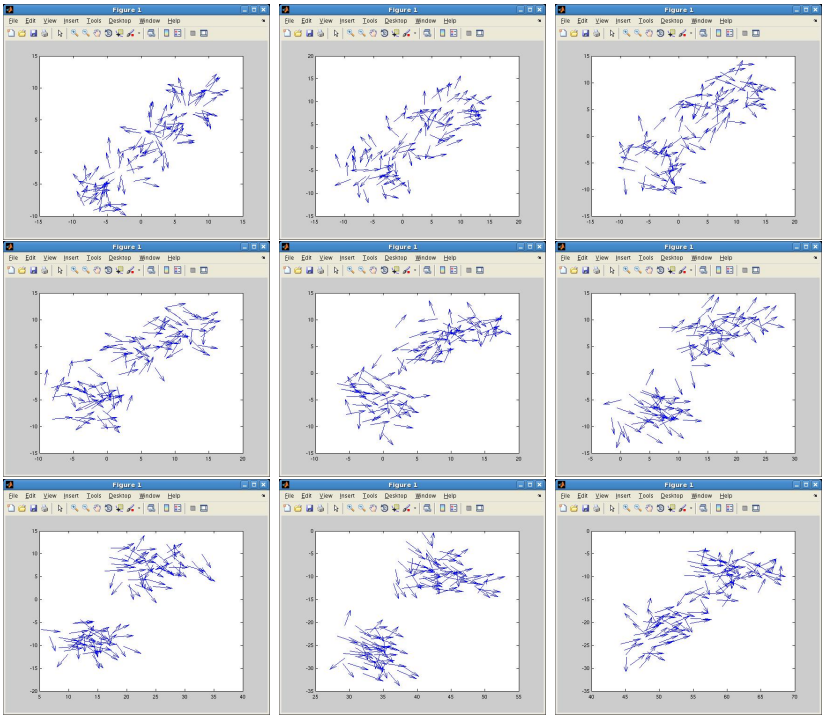

Figura 4.12: Passi successivi N=100, $T = 100$, $w_0 = 0.50$, $w_a = 0.50$

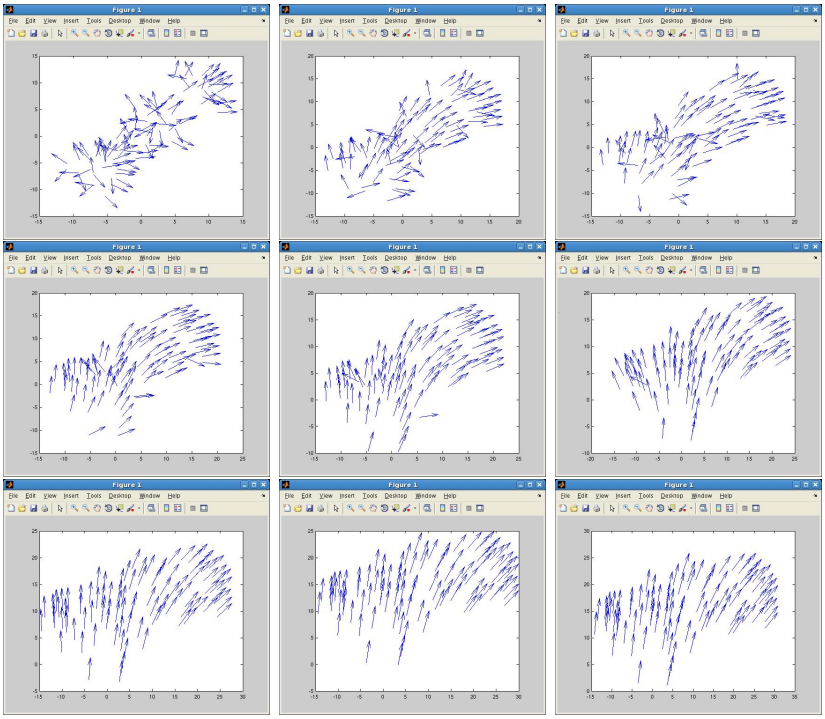

Figura 4.13: Passi successivi N=100, $T = 100$, $w_0 = 1.00$, $w_a = 0.00$

disposizione da NVIDIA che prende il nome di *CudaProfile*. Esso crea un progetto a partire dall'eseguibile GPU e genera diversi grafici in relazione all'insieme delle variabili e all'insieme dei kernel che la GPU stessa durante l'esecuzione richiama.

In sequenza sono mostrati i grafici di tipo *GPU Time Summary Plot*, i quali riportano in percentuale il tempo impiegato dalla GPU per eseguire i singoli kernel in ordine decrescente. Sulla sinistra, in corrispondenza dell'asse y, sono riportati i nomi dei kernel; in parentesi tonda viene indicato il numero di volte in cui quello stesso kernel viene richiamato.

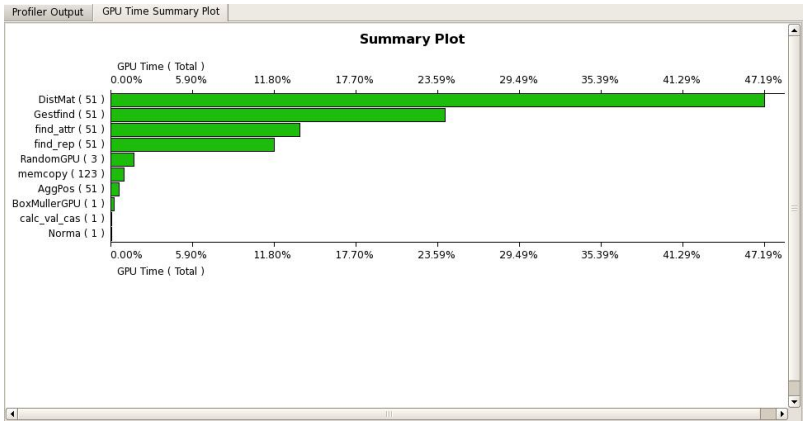

Figura 4.14: $T = 10$, $N = 100$

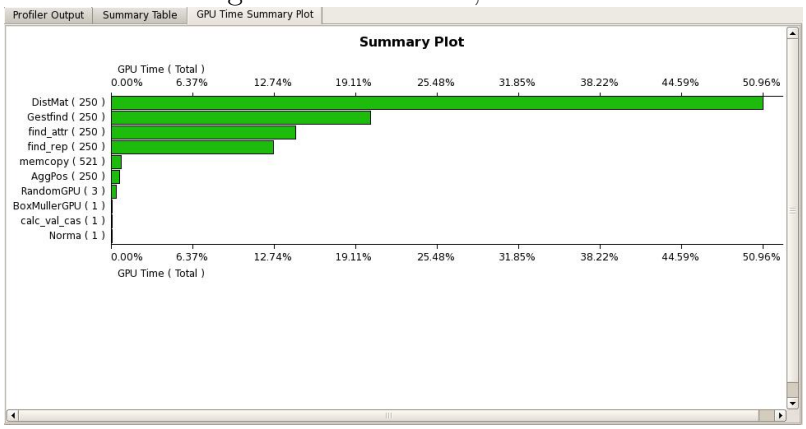

Figura 4.15: $T = 50$, $N = 100$

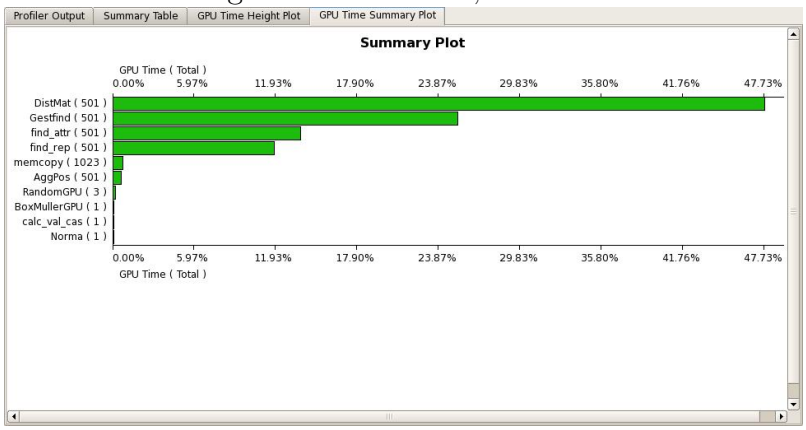

Figura 4.16: $T = 100$, $N = 100$

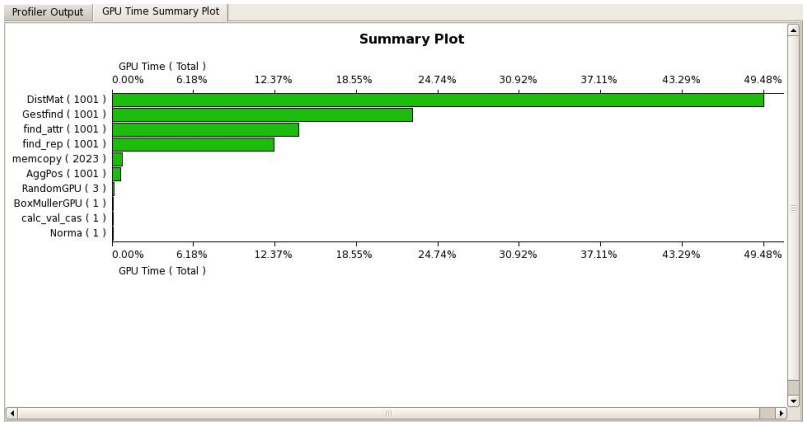

Figura 4.17: $T = 200$, $N = 100$

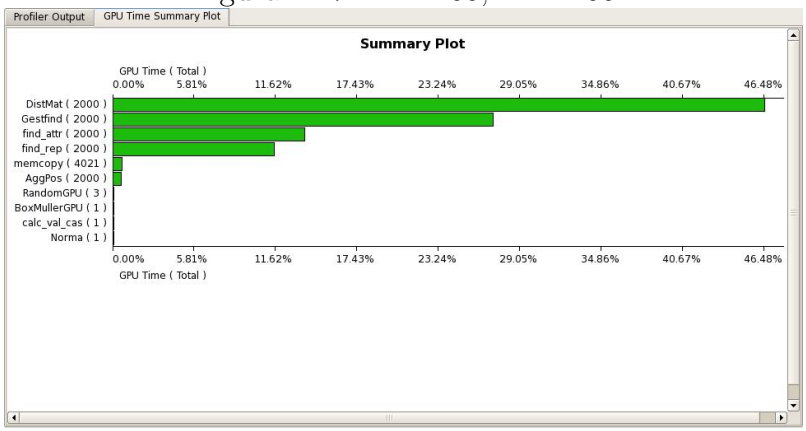

Figura 4.18: $T = 400$, $N = 100$

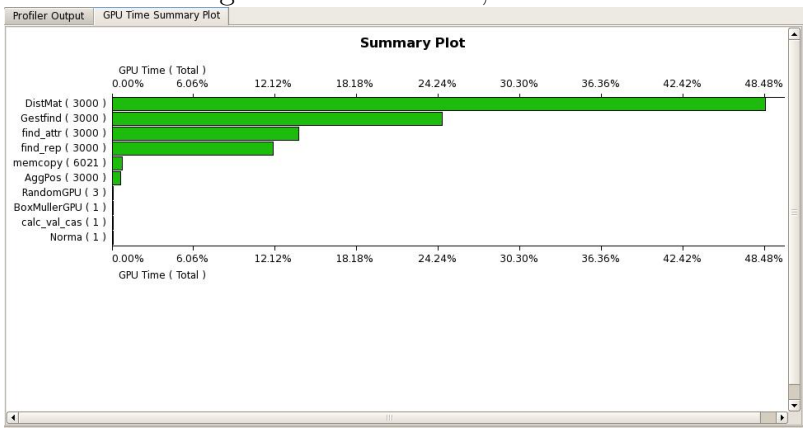

Figura 4.19: $T = 600$, N=100

Capitolo 5

Conclusioni e sviluppi futuri

In questa tesi è stato sviluppato un algoritmo parallelo utilizzando le Workstation HP della serie wx4600 equipaggiate con la scheda video NVIDIA Geforce 8400 GS. Le workstation sono equipaggiate con un processore Intel Core 2 Duo CPU E6750 2.66GHz e come sistema operativo la distribuzione ScientificLinux derivata dalla Red Hat 5, sulla quale sono stati opportunamente installati i driver della scheda video rilasciati dalla stessa NVIDIA.
Dal punto di vista software è stato installato e utilizzato CUDA, sviluppato da NVIDIA per l'architettura G80. I vantaggi prestazionali, in termini di efficienza, che sono stati riscontrati in questo ambiente sono stati notevoli, come è stato quantificato nel capitolo 4 riguardante le analisi delle simulazioni stocastiche con il programma FS-Parth sviluppato nella tesi.
Il futuro delle architettue CPU si sposta sempre più rapidamente verso soluzioni multicore, aprendo scenari interessanti su approcci potenzialmente vincenti tra gli attuali sviluppati da Intel e AMD, quest'ultima interessata su particolari architetture che abbinino core general purpose con acceleratori dedicati.

Si sa che il futuro dell'IT è proiettato verso il parallel computing: tutto, dal software all'hardware, si sta avviando in questa direzione e con CUDA, Nvidia ha dato un contributo significativo, riscuotendo un notevole interesse nella comunità scientifica. Gli obiettivi futuri concernenti CUDA riguardano in particolare la portabilità. NVIDIA afferma che nel mondo sono già presenti 70 milioni di GPU compatibili con CUDA, ma questo non è sufficiente per imporre ancora uno standard.

A differenza delle CPU, le GPU utilizzano un'architettura composta da decine di core, ciascuno dei quali è capace di eseguire migliaia di thread indipendenti e simultanei, per un totale di alcuni miliardi di operazioni in virgola mobile al secondo (gigaflops). La nuova proposta di Nvidia in ambito notebook sarà la tecnologia SLI ibrida (Hybrid SLI) in grado di modulare, a seconda delle necessità, diverse combinazioni fra un sottosistema grafico integrato (UMA) e una scheda grafica dedicata: il notebook sarà in grado di operare in due differenti modalità: la modalità Risparmio Energetico (Power Saving) che andrà a spegnere la scheda grafica dedicata per affidarsi completamente al sottosistema grafico integrato nel chipset, e la modalità Prestazioni Massime (Max Performance) che andrà ad attivare il chip grafico dedicato.

Sono diverse gli sviluppi futuri da considerare in relazione alle GPU e ai loro vantaggi prestazionali. Sarà interessante provare a combinare due schede video dell'ultima generazione per eseguire algoritmi paralleli, oppure si potrà pensare di effettuare simulazioni ad alto carico computazionale utilizzando cluster di GPU ove la parallelizzazione effettiva dipenderà dalla velocità di comunicazione del cluster stesso. Tra gli obiettivi futuri ancora, si spera

nella possibilità di espandere la memoria delle GPU, di consentire l'esecuzione contemporanea delle istruzioni sia da parte della CPU che della GPU in parallelo e di poter ridurre i tempi di trasferimento dei dati tra la CPU e la GPU quando viene mandato in esecuzione un programma sviluppato in ambiente CUDA.

Per quanto concerne il modello di Fish Schooling, il piano successivo è sviluppare una implementazione che sfrutti il parallelismo delle GPU a livello non della singola realizzazione (come è stato fatto nella tesi) ma a livello della totalità delle realizzazioni previste da una simulazione complessa.

Appendice A

FS-Parth MATLAB

Listing A.1: Fish_Schooling.m

```
scelta= menu('Effettuare una scelta di esecuzione per il
              Fish-Schooling','Visualizzare grafici ad
              ogni passo','Non visualizzare grafici ad
              ogni passo');
switch scelta
    case 1
       visualizza_grafici
    case 2
        non_visualizzare_grafici
end
```

Listing A.2: visualizza_grafici.m

```
rr=1.0;
rp=7;
teta=115*pi/180;
s=2.0;
sigma=0.01;
wa=0.20;
w0=0.80;
T=10;
tau=0.2;
N=100;
tetatau=teta*tau;
Mat_p_att=-10+20*rand(2,N);
Mat_v_att=-1+2*rand(2,N);
Mat_v_att=Norma(Mat_v_att,N);
Mat_v_succ=zeros(2,N);
figure(1)
quiver(Mat_p_att(1,:),Mat_p_att(2,:),Mat_v_att(1,:),
       Mat_v_att(2,:));
axis([-30 30 -30 30])
kk=0;
POL=zeros(1,T/tau+1);
for k=0:tau:T
    kk=kk+1;
    D=distanza(Mat_p_att,N);
```

```
    pol_temp=[0 0]';
    for i=1:N
        noinfluenza = 0;
        indrep=find(D(i,:)<=rr & D(i,:)~=0);
        p_i=Mat_p_att(:,i);
        vit=Mat_v_att(:,i);
        if(length(indrep)~=0)
            vitau=[0;0];
            for j=1:length(indrep)
                pj=Mat_p_att(:,indrep(j));
                vitau=vitau-((pj-p_i)/(norm(pj-p_i)));
            end
        else
            indattr=find(D(i,:)>rr & D(i,:)<=rp &
                         D(i,:)~=0);
            if(length(indattr)~=0)
                numvi=[0;0];
                vj=[0;0];
                for j=1:length(indattr)
                    pj=Mat_p_att(:,indattr(j));
                    numvi=numvi+((pj-p_i)/
                                 (norm(pj-p_i)));
                    vj=vj+Mat_v_att(:,indattr(j));
                end
                vitau=wa*numvi/(norm(numvi))+
                                 w0*vj/(norm(vj));
            else
                noinfluenza = 1;
                vitau = vit;
            end
        end
        vitau = normvitau(vit,vitau);
        phi=sigma*randn(1);
        mat_rot_std=[cos(tetatau) -sin(tetatau);
                     sin(tetatau) cos(tetatau)];
        if noinfluenza
            z = 0;
        else
            z=angvet(vit,vitau);
        end
        phiz=phi+z;
        if abs(phiz)<(tetatau/2)
            M_rot=[cos(phiz) -sin(phiz);
                   sin(phiz) cos(phiz)];
        else
            if phiz > 0
                M_rot=mat_rot_std;
            else
                M_rot=mat_rot_std';
            end
        end
        vitau=M_rot*vit;
        pol_temp=pol_temp+vitau;
        Mat_v_succ(:,i)= vitau;
        Mat_p_succ(:,i) = p_i + s*vitau*tau;
        figure(2)
        quiver(Mat_p_succ(1,:),Mat_p_succ(2,:),
               Mat_v_succ(1,:),Mat_v_succ(2,:))
        axis([-30 30 -30 30])
    end
    Mat_p_att=Mat_p_succ;
    Mat_v_att=Mat_v_succ;
```

```
    POL(kk)=norm(pol_temp)/N;
end
plot(0:tau:T,POL)
axis([0 T 0 1])
```

Listing A.3: non_visualizzare_grafici.m

```
rr=1.0;
rp=7;
teta=115*pi/180;
s=2.0;
sigma=0.01;
wa=0.20;
w0=0.80;
T=10;
tau=0.2;
N=100;
tetatau=teta*tau;
Mat_p_att=-10+20*rand(2,N);
Mat_v_att=-1+2*rand(2,N);
Mat_v_att=Norma(Mat_v_att,N);
Mat_v_succ=zeros(2,N);
kk=0;
POL=zeros(1,T/tau+1);
for k=0:tau:T
    kk=kk+1;
    D=distanza(Mat_p_att,N);
    pol_temp=[0 0]';
    for i=1:N
        noinfluenza = 0;
        indrep=find(D(i,:)<=rr & D(i,:)~=0);
        p_i=Mat_p_att(:,i);
        vit=Mat_v_att(:,i);
        if(length(indrep)~=0)
            vitau=[0;0];
            for j=1:length(indrep)
                pj=Mat_p_att(:,indrep(j));
                vitau=vitau-((pj-p_i)/(norm(pj-p_i)));
            end
        else
            indattr=find(D(i,:)>rr & D(i,:)<=rp &
                         D(i,:)~=0);
            if(length(indattr)~=0)
                numvi=[0;0];
                vj=[0;0];
                for j=1:length(indattr)
                    pj=Mat_p_att(:,indattr(j));
                    numvi=numvi+((pj-p_i)/
                          (norm(pj-p_i)));
                    vj=vj+Mat_v_att(:,indattr(j));
                end
                vitau=wa*numvi/(norm(numvi))+
                      w0*vj/(norm(vj));
            else
                noinfluenza = 1;
                vitau = vit;
            end
        end
        vitau = normvitau(vit,vitau);
        phi=sigma*randn(1);
        mat_rot_std=[cos(tetatau) -sin(tetatau);
                     sin(tetatau) cos(tetatau)];
```

```
        if noinfluenza
            z = 0;
        else
            z=angvet(vit,vitau);
        end
        phiz=phi+z;
        if abs(phiz)<(tetatau/2)
            M_rot=[cos(phiz) -sin(phiz);
                   sin(phiz) cos(phiz)];
        else
            if phiz > 0
                M_rot=mat_rot_std;
            else
                M_rot=mat_rot_std';
            end
        end
        vitau=M_rot*vit;
        pol_temp=pol_temp+vitau;
        Mat_v_succ(:,i)= vitau;
        Mat_p_succ(:,i) = p_i + s*vitau*tau;
    end
    Mat_p_att=Mat_p_succ;
    Mat_v_att=Mat_v_succ;
    POL(kk)=norm(pol_temp)/N;
end
plot(0:tau:T,POL)
axis([0 T 0 1])
```

Listing A.4: distanza.m

```
function dist=distanza(P,N)
for i=1:N
    for j=1:N
        dist(i,j)=sqrt(((P(1,i)-P(1,j)).^2)+
                       ((P(2,i)-P(2,j)).^2));
    end
end
```

Listing A.5: Norma.m

```
function M_norm=Norma(M,N)
    for j=1:N
        M_norm(:,j)=M(:,j)/(norm(M(:,j)));
    end
```

Listing A.6: normvitau.m

```
function Vel_succ=normvitau(vit,vitau)
    norm_vett=norm(vitau);
    if(norm_vett~=0)
        Vel_succ=vitau/norm_vett;
    else
        Vel_succ=vit;
    end
```

Listing A.7: angvet.m

```
function y = angvet(p1,p2)
    if norm(p1)==0 || norm(p2)==0
        y=0;
```

```
    else
        v1=p1/norm(p1); v2=p2/norm(p2);
        y = acos(v1'*v2);
        if acos(v2'*[-v1(2);v1(1)])>pi/2
            y = -y;
        end
    end
```

Listing A.8: Realization.m

```
rr=1.0;
rp=7;
teta=115*pi/180;
s=2.0;
sigma=0.01;
wa=0.20;
w0=0.80;
T=10;
tau=0.2;
N=100;
tetatau=teta*tau;
Mat_p_att=-10+20*rand(2,N);
Mat_v_att=-1+2*rand(2,N);
Mat_p_succ=zeros(2,N);
Mat_v_att=Norma(Mat_v_att,N);
Mat_v_succ=zeros(2,N);
nstep=T/tau+1;
POL=zeros(1,nstep);
PR=zeros(1120,nstep);
for nr=1:1120
    kk=0;
    for k=0:tau:T
        kk=kk+1;
        D=distanza(Mat_p_att,N);
        pol_temp=[0 0]';
        for i=1:N
            noinfluenza = 0;
            indrep=find(D(i,:)<=rr &
                        D(i,:)~=0);
            p_i=Mat_p_att(:,i);
            vit=Mat_v_att(:,i);
            if(length(indrep)~=0)
                vitau=[0;0];
                for j=1:length(indrep)
                    pj=Mat_p_att(:,indrep(j));
                    vitau=vitau-((pj-p_i)/
                            (norm(pj-p_i)));
                end
            else
                indattr=find(D(i,:)>rr & D(i,:)<=rp &
                             D(i,:)~=0);
                if(length(indattr)~=0)
                    numvi=[0;0];
                    vj=[0;0];
                    for j=1:length(indattr)
                        pj=Mat_p_att(:,indattr(j));
                        numvi=numvi+((pj-p_i)/
                                (norm(pj-p_i)));
                        vj=vj+Mat_v_att(:,indattr(j));
                    end
                    vitau=wa*numvi/(norm(numvi))+
                          w0*vj/(norm(vj));
```

```
                else
                    noinfluenza = 1;
                    vitau = vit;
                end
            end
            vitau = normvitau(vit,vitau);
            phi=sigma*randn(1);
            mat_rot_std=[cos(tetatau) -sin(tetatau);
                         sin(tetatau) cos(tetatau)];
            if noinfluenza
                z = 0;
            else
                z=angvet(vit,vitau);
            end
            phiz=phi+z;
            if abs(phiz)<(tetatau/2)
                M_rot=[cos(phiz) -sin(phiz);
                       sin(phiz) cos(phiz)];
            else
                if phiz > 0
                    M_rot=mat_rot_std;
                else
                    M_rot=mat_rot_std';
                end
            end
            vitau=M_rot*vit;
            pol_temp=pol_temp+vitau;
            Mat_v_succ(:,i)= vitau;
            Mat_p_succ(:,i) = p_i + s*vitau*tau;
        end
        Mat_p_att=Mat_p_succ;
        Mat_v_att=Mat_v_succ;
        POL(kk)=norm(pol_temp)/N;
        PR(nr,:)=POL;
    end
    PRmedio=mean(PR);
    disp('passo')
    disp(nr)
end
plot(0:nstep-1,PRmedio)
```

Listing A.9: Grafici.m

```
function Grafici(T)
for i=0:T
    file1=['p' num2str(i) '.txt'];
    file2=['v' num2str(i) '.txt'];
    p=load(file1);
    v=load(file2);
    quiver(p(1,:),p(2,:),v(1,:),v(2,:));
    pause(1)
end
```

Appendice B

FS-Parth CUDA

Listing B.1: MyCudaLib.h

```
#define PIGRECO 4*atan(1.)
const float teta=2.0071;
const float tau=0.2;
const float tetatau=teta*tau;
const float sigma=0.01;
const float s=2.0;
const float rr=1.0;
const float rp=7.0;
const float w0=0.80;
const float wa=0.20;
const int M=100;
__global__ void DistMat(float *A,float *B,int M)
{
  int idx=blockIdx.x * blockDim.x + threadIdx.x;
  int jdx=blockIdx.y * blockDim.y + threadIdx.y;
  if(idx<M && jdx<M)
  {
    B[jdx*M+idx]=sqrt(pow((A[idx]-A[jdx]),2)+
                      pow(A[M+idx]-A[M+jdx],2));
  }
}
__global__ void find_rep(int *indrep,float *B,int M)
{
  int idx=blockIdx.x * blockDim.x + threadIdx.x;
  int cont=0;
  if(idx<M)
  {
    for(int i=0;i<M;i++)
    {
      if(B[idx*M+i]<=rr && B[idx*M+i]!=0)
      {
        cont++;
        indrep[idx*(M+1)+cont]=i;
      }
    }
    indrep[idx*(M+1)]=cont;
  }
}
__global__ void find_attr(int *indattr,float *B,int M)
{
  int idx=blockIdx.x * blockDim.x + threadIdx.x;
  int cont=0;
```

```
  if(idx<M)
  {
    for(int i=0;i<M;i++)
    {
      if(B[idx*M+i]>rr && B[idx*M+i]<=rp &&
                                  B[idx*M+i]!=0)
      {
        cont++;
        indattr[idx*(M+1)+cont]=i;
      }
    }
   indattr[idx*(M+1)]=cont;
  }
}
__device__ void calc_angolo(float *Vp,float *Vs,
                                    int M,float *z)
{
  int idx=blockIdx.x * blockDim.x + threadIdx.x;
  float v1riga,v1col,v2riga,v2col;
  if(idx<M)
  {
    float normp1=sqrt((pow(Vp[idx],2))+
                      (pow(Vp[M+idx],2)));
    float normp2=sqrt((pow(Vs[idx],2))+
                      (pow(Vs[M+idx],2)));
    if(normp1==0 || normp2==0)
    {
      *z=0;
    }
    else
    {
      v1riga=Vp[idx]/normp1;
      v1col=Vp[M+idx]/normp1;
      v2riga=Vs[idx]/normp2;
      v2col=Vs[M+idx]/normp2;
      float prod=(v1riga*v2riga)+
                 (v1col*v2col);
      *z=acos(prod);
      float prod2=(v2riga*(-v1col)+
                   (v2col*v1riga));
      float z2=acos(prod2);
      if(z2 > PIGRECO/2)
      {
        *z=-*z;
      }
    }
  }
}
__global__ void Norma(float *p,int M,float *p_norm)
{
  int idx=blockIdx.x * blockDim.x + threadIdx.x;
  float calc_norm,val_norm_riga,val_norm_col;
  if(idx<M)
  {
    calc_norm=sqrt((pow(p[idx],2))+(pow(p[M+idx],2)));
    val_norm_riga=p[idx]/calc_norm;
    val_norm_col=p[M+idx]/calc_norm;
    p_norm[idx]=val_norm_riga;
    p_norm[M+idx]=val_norm_col;
  }
```

```
}
__device__ void normvitau(float *V_new, float *V_iniz)
{
  int idx=blockIdx.x * blockDim.x + threadIdx.x;
  if(idx<M)
  {
    float normvitau=sqrt((pow(V_new[idx],2))+
                         (pow(V_new[M+idx],2)));
    if(normvitau!=0)
    {
      V_new[idx]=V_new[idx]/normvitau;
      V_new[M+idx]=V_new[M+idx]/normvitau;
    }
    else
    {
      V_new[idx]=V_iniz[idx];
      V_new[M+idx]=V_iniz[M+idx];
    }
  }
}
__global__ void Gestfind(int *indrep, int *indattr,
                         float *A, int M, float *V_s,
                         float *V,float *ang)
{
  int idx=blockIdx.x * blockDim.x + threadIdx.x;
  int j2;
  int noinfluenza=0;
  if(idx<M)
  {
    int dim_rep=indrep[idx*(M+1)];
    int dim_attr=indattr[idx*(M+1)];
    if(dim_rep!=0)
    {
      for(int j=1;j<=dim_rep;j++)
      {
        j2=indrep[idx*(M+1)+j];
        float norm=sqrt((pow(A[j2]-A[idx],2))+
                        (pow(A[M+j2]-A[M+idx],2)));
        V_s[idx]=V_s[idx]-((A[j2])-(A[idx]))/norm;
        V_s[M+idx]=V_s[M+idx]-((A[M+j2])-(A[M+idx]))/
                                                 norm;
      }
    }
    else
    {
      float numviriga=0.0,numvicol=0.0,
            vjriga=0.0,vjcol=0.0;
      if(dim_attr!=0)
      {
        for(int j=1;j<=dim_attr;j++)
        {

          j2=indattr[idx*(M+1)+j];
          float norm=sqrt((pow(A[j2]-A[idx],2))+
                          (pow(A[M+j2]-A[M+idx],2)));
          numviriga=numviriga+((A[j2]-A[idx]))/norm;
          numvicol=numvicol+((A[M+j2]-A[M+idx]))/norm;
          vjriga=vjriga+V[j2];
          vjcol=vjcol+V[M+j2];
        }
```

```
        float normnumvi=sqrt((pow(numviriga,2))+
                             (pow(numvicol,2)));
        float normvj=sqrt((pow(vjriga,2))+
                          (pow(vjcol,2)));
        V_s[idx]=wa*numviriga/(normnumvi)+
                             w0*vjriga/normvj;
        V_s[M+idx]=wa*numvicol/(normnumvi)+
                             w0*vjcol/normvj;
      }
      else
      {
        noinfluenza=1;
        V_s[idx]=V[idx];
        V_s[M+idx]=V[M+idx];
      }
    }
    normvitau(V_s,V);
    float z;
    float phi=sigma*ang[idx];
    if (noinfluenza==1)
    {
      z=0;
    }
    else
    {
      calc_angolo(V,V_s,M,&z);
    }
    float phiz=phi+z;
    float M_Rot[4];
    if(abs(phi+z) < (tetatau/2) )
    {
      M_Rot[0]=(cos(phi+z));
      M_Rot[1]=(-sin(phi+z));
      M_Rot[2]=(sin(phi+z));
      M_Rot[3]=(cos(phi+z));
    }
    else
    {
      if(phiz > 0)
      {
        M_Rot[0]=cos(tetatau);
        M_Rot[1]=-sin(tetatau);
        M_Rot[2]=sin(tetatau);
        M_Rot[3]=cos(tetatau);
      }
      else
      {
        M_Rot[0]=cos(teta*tau);
        M_Rot[1]=sin(teta*tau);
        M_Rot[2]=-sin(teta*tau);
        M_Rot[3]=cos(teta*tau);
      }
    }
    V_s[idx]=(M_Rot[0]*V[idx])+
             (M_Rot[1]*V[M+idx]);
    V_s[M+idx]=(M_Rot[2]*V[idx])+
               (M_Rot[3]*V[M+idx]);
  }
}
```

```
__global__ void AggPos(float *A, float *V_succ,
                                    float *A_s)
{
  int idx=blockIdx.x * blockDim.x + threadIdx.x;
  if(idx<M)
  {
    A_s[idx]=A[idx]+s*(V_succ[idx]*tau);
    A_s[M+idx]=A[M+idx]+s*(V_succ[M+idx]*tau);
  }
}
__global__ void calc_val_cas(float *A, float *V)
{
  int idx=blockIdx.x * blockDim.x + threadIdx.x;
  if(idx<M)
  {
    A[idx]=-10+(20*A[idx]);
    A[M+idx]=-10+(20*A[M+idx]);
    V[idx]=-1+(2*V[idx]);
    V[M+idx]=-1+(2*V[M+idx]);
  }
}
void CheckCUDAError(const char *msg)
{
  cudaError_t err=cudaGetLastError();
  if(cudaSuccess !=err)
  {
    fprintf(stderr,``Cuda error: %s: %s.\n'',msg,
                        cudaGetErrorString(err));
    exit(EXIT_FAILURE);
  }
};
```

Listing B.2: Fish_Schooling.cu

```
#include <stdio.h>
#include <math.h>
#include "/home/LI476/NVIDIA_CUDA_SDK/
                    common/inc/cutil.h"
#include "/home/LI476/NVIDIA_CUDA_SDK/
              projects/MersenneTwister/
              MersenneTwister_kernel.cu"

#include "MyCudaLib.h"
#define  MT_RNG_COUNT 1
const float T=10;
int main(int argc, char *argv[])
{
  float *A_h,*A_d,*B_h,*B_d,*V_h,*V_d,
        *V_succ_h,*V_succ_d;
  int *ind_h,*ind_d,*inda_h,*inda_d;
  float *angolo_h,*angolo_d;
  float *A_succ_h,*A_succ_d;
  float *Vnorm_h,*Vnorm_d;
  FILE *fp1,*fp2;
  char vel[10],pos[10];
  size_t size=M*M*sizeof(float);
  size_t sizeA=2*M*sizeof(float);
  size_t sizev=(M+1)*M*sizeof(int);
  srand(time(NULL));
  A_h=(float*)malloc(sizeA);
  B_h=(float*)malloc(size);
```

```
V_h=(float*)malloc(sizeA);
V_succ_h=(float*)malloc(sizeA);
Vnorm_h=(float*)malloc(sizeA);
ind_h=(int*)malloc(sizev);
inda_h=(int*)malloc(sizev);
A_succ_h=(float*)malloc(sizeA);
angolo_h=(float *)malloc(M*sizeof(float));
cudaMalloc((void **)&inda_d,sizev);
cudaMalloc((void **)&ind_d,sizev);
cudaMalloc((void **)&A_d,sizeA);
cudaMalloc((void **)&B_d,size);
cudaMalloc((void **)&V_d,sizeA);
cudaMalloc((void **)&V_succ_d,sizeA);
cudaMalloc((void **)&Vnorm_d,sizeA);
cudaMalloc((void **)&A_succ_d,sizeA);
cudaMalloc((void **)&angolo_d,M*sizeof(float));
cudaMemcpy(A_d,A_h,sizeA,cudaMemcpyHostToDevice);
cudaMemcpy(ind_d,ind_h,sizev,cudaMemcpyHostToDevice);
cudaMemcpy(inda_d,inda_h,sizev,
           cudaMemcpyHostToDevice);
cudaMemcpy(V_d,V_h,sizeA,cudaMemcpyHostToDevice);
cudaMemcpy(V_succ_d,V_succ_h,sizeA,
           cudaMemcpyHostToDevice);
cudaMemcpy(A_succ_d,A_succ_h,sizeA,
           cudaMemcpyHostToDevice);
dim3 dimGrid(8,8);
dim3 dimBlock(13,13);
seedMTGPU(rand());
RandomGPU<<<1,M>>>(angolo_d,M);
BoxMullerGPU<<<1,M>>>(angolo_d,M);
seedMTGPU(rand());
RandomGPU<<<1,2*M>>>(A_d,2*M);
seedMTGPU(rand());
RandomGPU<<<1,2*M>>>(V_d,2*M);
calc_val_cas<<<1,2*M>>>(A_d,V_d);
Norma<<<1,M>>>(V_d,M,Vnorm_d);
int ind=0;
unsigned int timer=0;
cutCreateTimer(&timer);
cutStartTimer(timer);
for (float k=0;k<=T;k+=tau)
{
  DistMat<<<dimGrid,dimBlock>>>(A_d,B_d,M);
  find_rep<<<1,M>>>(ind_d,B_d,M);
  find_attr<<<1,M>>>(inda_d,B_d,M);
  Gestfind<<<1,M>>>(ind_d,inda_d,A_d,M,V_succ_d,
                    Vnorm_d,angolo_d);
  AggPos<<<1,M>>>(A_d,V_succ_d,A_succ_d);
  cudaMemcpy(A_d,A_succ_d,sizeA,
             cudaMemcpyDeviceToDevice);
  cudaMemcpy(Vnorm_d,V_succ_d,sizeA,
             cudaMemcpyDeviceToDevice);
  cudaMemcpy(V_succ_h,V_succ_d,sizeA,
             cudaMemcpyDeviceToHost);
  cudaMemcpy(A_succ_h,A_succ_d,sizeA,
             cudaMemcpyDeviceToHost);
  sprintf(vel,''v%d.txt'',ind);
  sprintf(pos,''p%d.txt'',ind);
  fp1=fopen(vel,''w'');
```

```
    fp2=fopen(pos,''w'');
    for(int i=0;i<2;i++)
    {
      for(int j=0;j<M;j++)
      {
        fprintf(fp1,''%f'',V_succ_h[(i*M)+j]);
      }
      fprintf(fp1,"\n");
    }
    for(int i=0;i<2;i++)
    {
      for(int j=0;j<M;j++)
      {
        fprintf(fp2,''%f'',A_succ_h[(i*M)+j]);
      }
      fprintf(fp2,"\n");
    }
    ind++;
  }
  CheckCUDAError("errore");
  cutStopTimer( timer );
  cudaMemcpy(B_h,B_d,size,cudaMemcpyDeviceToHost);
  cudaMemcpy(ind_h,ind_d,sizev,cudaMemcpyDeviceToHost);
  cudaMemcpy(inda_h,inda_d,sizev,
             cudaMemcpyDeviceToHost);
  cudaMemcpy(V_h,V_d,sizeA,cudaMemcpyDeviceToHost);
  cudaMemcpy(V_succ_h,V_succ_d,sizeA,
             cudaMemcpyDeviceToHost);
  cudaMemcpy(A_h,A_d,sizeA,cudaMemcpyDeviceToHost);
  cudaMemcpy(Vnorm_h,Vnorm_d,sizeA,
             cudaMemcpyDeviceToHost);
  cudaMemcpy(angolo_h,angolo_d,M*sizeof(float),
             cudaMemcpyDeviceToHost);
  cudaMemcpy(A_succ_h,A_succ_d,sizeA,
             cudaMemcpyDeviceToHost);
  printf(''CUDA execution time= %f ms\n'',
         cutGetTimerValue( timer ));
  for(int k=0;k<2*M;k++)
  {
    printf(''V_succ[%d]=%f\n'',k,V_succ_h[k]);
  }
  for(int i=0;i<2*M;i++)
  {
    printf(''P_succ[%d]=%f\n'',i,A_succ_h[i]);
  }
  free(A_h);
  cudaFree(A_d);
  free(B_h);
  cudaFree(B_d);
  free(V_h);
  cudaFree(V_d);
  free(ind_h);
  cudaFree(ind_d);
  free(inda_h);
  cudaFree(inda_d);
  free(V_succ_h);
  cudaFree(V_succ_d);
  free(Vnorm_h);
  cudaFree(Vnorm_d);
```

```
    free(angolo_h);
    cudaFree(angolo_d);
    free(A_succ_h);
    cudaFree(A_succ_d);
    return EXIT_SUCCESS;
}
```

Bibliografia

[1] S. Camazine, J. L. Deneubourg, N. R. Franks, J. Sneyd, G. Theraulaz, and E. Bonabeau.*Self-Organization in Biological Systems*. Princeton University Press, Princeton, 2003.

[2] I. D. Couzin, J. Krause, N. R. Franks, and S. A. Levin. *Effective leadership and decision making in animal groups on the move*. Nature, 2005.

[3] I. D. Couzin, J. Krause, R. James, G. D. Ruxton, and N. R. Franks. *Collective memory and spatial sorting in animal groups*. J. theor. Biol., 2002.

[4] R. M. May. *Flight formations in geese and other birds*. Nature, 1979.

[5] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 2007. (http://developer.download.nvidia.com).

[6] Huth, A. & Wissel, C. *The simulation of the movement of fish schools*. J. theor. Biol. 156,(1992).

[7] Krause, J. and Ruxton, G. D. *Living in Groups*. Oxford University Press, (2002).

[8] Camazine, Deneubourg, Franks, et al.: *Self-Organization in Biological Systems*. Princeton University Press, 2001.

[9] Hong Liy, Yang Caoz, Linda R. Petzoldx, Daniel T. Gillespie. *Algorithms and Software for Stochastic Simulation of Biochemical Reacting Systems*, 2007.

[10] Sabine Stocker. *Models for tuna school formation.* Politecnico di Torino, Dipartimento di Matematica, 1998.

[11] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. *A Survey of General-Purpose Computation on Graphics Hardware.* EUROGRAPHICS 2005.

[12] Darryl Gates. Demonstrating Emergent Behaviour With Schools of Fish, 2004.

[13] DAY, R. L., T. MACDONALD,C. BROWN, K. N. LALAND AND S. M. READER. *Interactions between shoal size and conformity in guppy social foraging.* Animal Behaviour, 2001.

[14] R. K. BUTLIN, N. PEUHKURI AND V. L. PRITCHARD. *The social organization of fish shoals: a test of the predictive power of laboratory experiments for the field.* Biol. Rev., 2000.

[15] PEUHKURI. *Size-assortative shoaling in fish: the effect of oddity on foraging behaviour.* Animal Behaviour, 1997.

[16] H. Li, 1 L. Petzold. *Efficient Parallelization of stochastic simulation alggorithm for chemically reacting systems on the GPU.* Department of Computer Science, University of California, Santa Barbara, CA 93106, U.S.A, 2007.

[17] Eung Kon Kim1 , Jong Chan Kim. *Fish Schooling Behavior Simulator for Cyber Aquarium*. Dept. of Computer Engineering, Sunchon National University Jeonnam, Korea, 2007.

[18] C. W. Reynolds. *Flocks, Herds, and Schools: A Distributed Behavioral model*. ACM SIGGRAPH 87, Vol. 21(4), July 1987, pp. 25-34.

[19] X. Tu, and D. Terzopoulos. *Perceptual Modeling for Behavioral Animation of Fishes*. Second Pacific Conference on Computer Graphics, 1994, pp. 185-200.

[20] Linda Petzold. *Multiscale Simulation of Biochemical Systems*. University of California Santa Barbara, 2007.