Modelli comportamentali su GPU con CUDA

Bernardino Frola

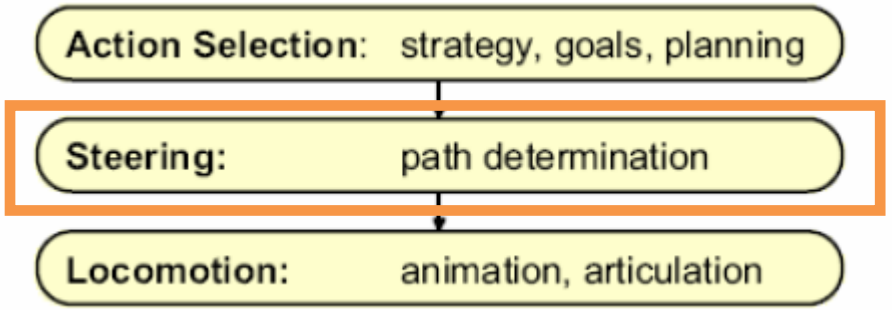
17 novembre 2008

Sommario

- Introduzione
- Database spaziale
- Comportamenti
- Prestazioni
- Conclusioni
- Appendici

Introduzione

- **Obbiettivo:** creare un ambiente di simulazione orientato ai grandi numeri
  - Decine di migliaia di agenti
- **Idea**: partire da un sistema esistente ed ottimizzarlo
  - Il sistema è OpenSteer
  - “L’ottimizzazione” è CUDA
- **Ambito (per ora):** Intelligenza artificiale “a basso livello”
  - Calcolo sterzata

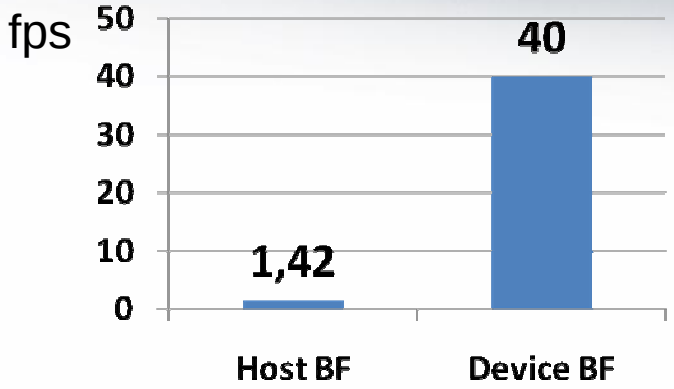


Database spaziale

- Il maggiore problema è il *solito* **calcolo del vicinato**
- OpenSteer fornisce un database spaziale
  - Efficiente se non si parla di decine di migliaia di elementi
  - Basato sulla suddivisione dello spazio in griglie
  - Cerca gli elementi in un certo raggio
  - **Nussun limite superiore al numero di vicini**
    - Punto debole!

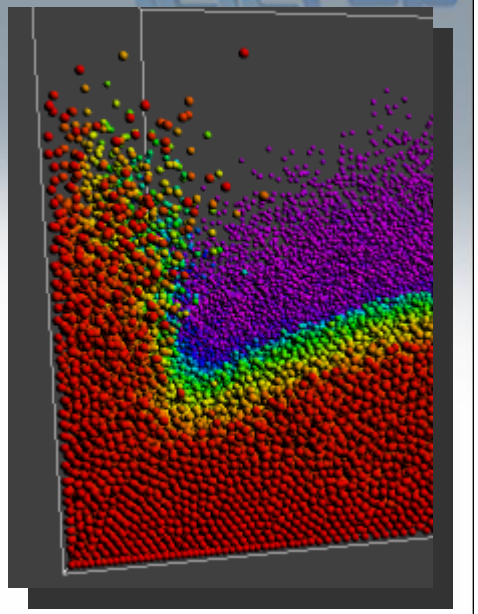
Calcolo di distanze in CUDA

- Il primo passo del lavoro è stato quello di realizzare la ricerca del vicinato mediante l'algoritmo forza bruta
- Risultati indicativi
  - 3.8k boids
  - CPU BF: 700ms (1.42fps)
  - CUDA BF: 25ms (40fps)
- L'incremento è notevole



- Controllo esaustivo, 3.8k^2 calcoli di distanze
  - Host: **22M** calcoli di distanze/s
  - Device: **600M** calcoli di distanze/s
- Come vedremo, il problema non è il calcolo..ma *l'accesso alla memoria…*

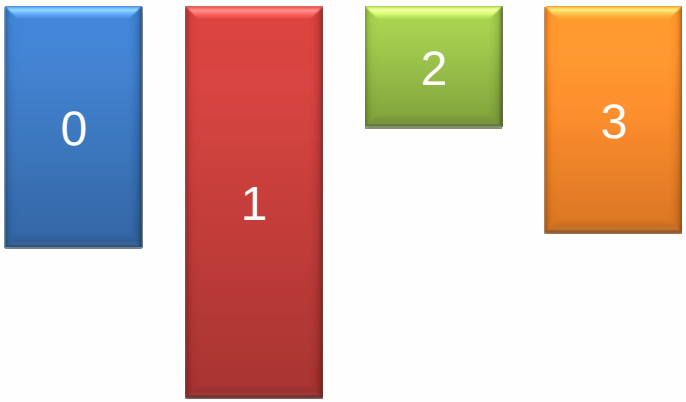
Collide: un algoritmo efficiente



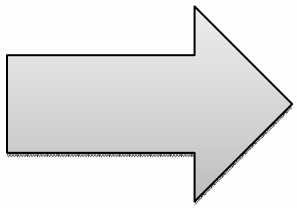
- Tra le demo fornite con l'SDK di CUDA ne è presente una interessante: *particles*
  - Calcola collisioni e semplici interazioni fisiche tra palline in un box
  - Si basa su un algoritmo di ordinamento realizzato dalla nvidia e descritti in *GPUGems 3*
    - Basato su prefix-sum
- Molto efficiente: calcola le collisioni tra 131k particelle in 10ms (100fps)
  - Mediante un uso estremamente efficiente della memoria
  - Godendo del fatto di dover scrivere poco…

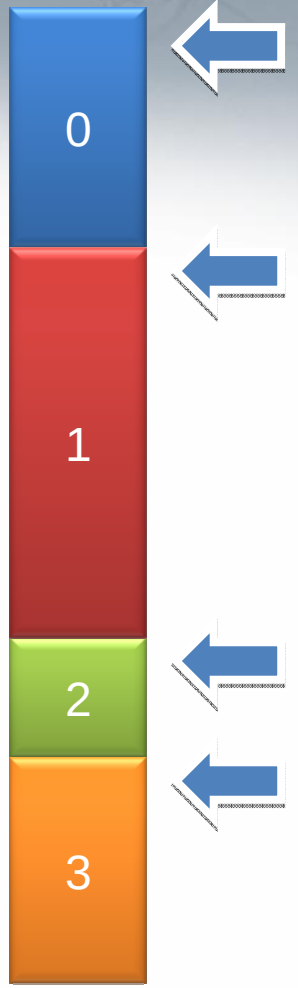
Collide: metodo

- L’idea è quella di non utilizzare liste di apparteneza separate per ogni cella, bensi usarne una sola
  - Segnandosi il punto di inizio di ogni cella (chiamati indici **cellStarts**)



Metodo classico





Metodo adottato

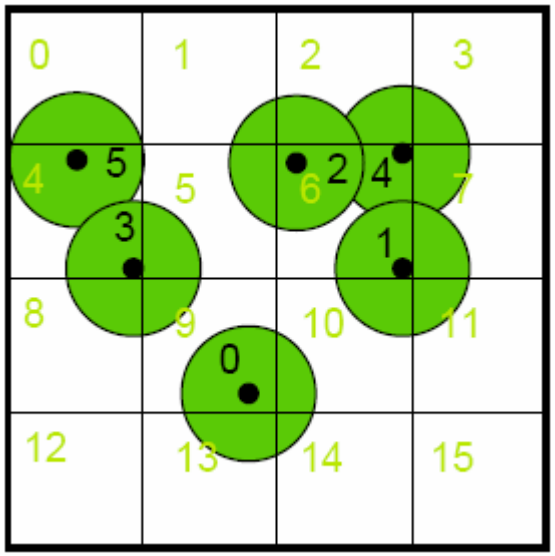
Collide: flusso delle operazioni

- Ogni iterazione prevede:
  - Le liste sono del tipo (Hash, Particle ID)

Calcolo **hash** (cella di appartenenza)

Ordinamento secondo l'hash

Calcolo **cellStarts**



(9, 0)
(6, 1)
(6, 2)
(4, 3)
(6, 4)
(4, 5)

(4, 3)
(4, 5)
(6, 1)
(6, 2)
(6, 4)
(9, 0)

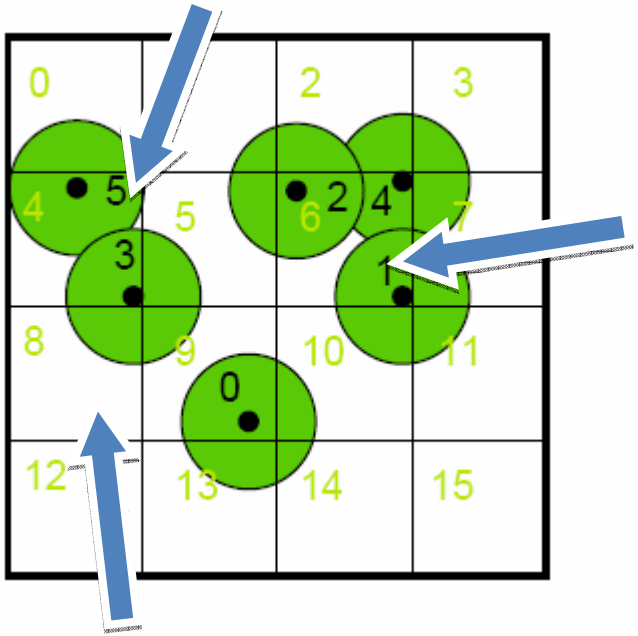
0	
1	
2	
3	
4	0
5	
6	2
7	
8	
9	5
10	
11	
12	
13	
14	
15	

Se non è presente l'indice di una cella nella lista cellStarts, vuol dire che la cella è vuota

Collide: ricerca collisioni

- La ricerca delle particelle vicine avviene nel modo seguente [particella 0]
  - Si lavora sulla lista ordinata secondo l'hash

cellStart[4] = 0
Inizia la ricerca da posizione 0



cellStart[6] = 2
Inizia la ricerca da posizione 2

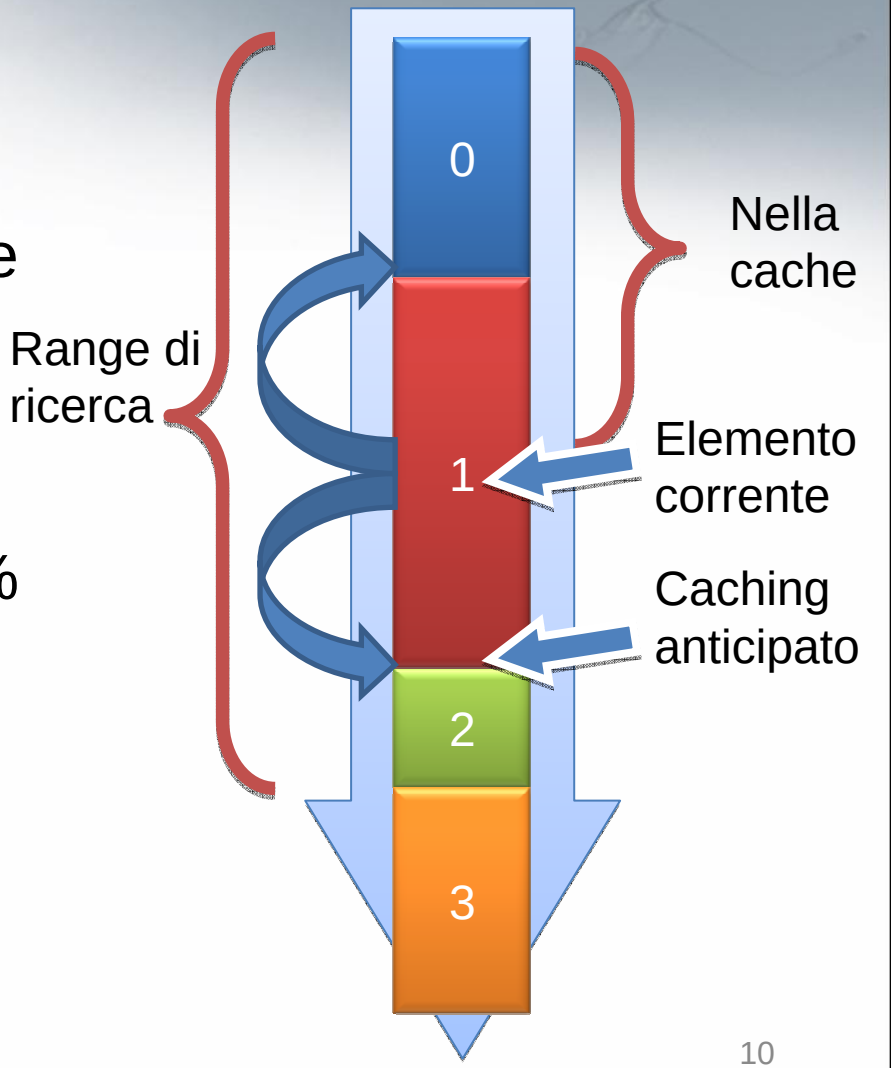
(4, 3)	4
(4, 5)	
(6, 1)	6
(6, 2)	
(6, 4)	
(9, 0)	9

cellStart[8] = -1
Cella vuota

La ricerca termina quando cambia l'hash dell'elemento della lista ordinata

Collide: considerazioni

- Perché l'algoritmo è veloce?
- La ricerca è preceduta dal binding dei dati su texture, un tipo di memoria dotata di cache
- Il segreto è dunque nell'ordinamento, che combinato all'uso delle texture produce uno speed-up del 45%
  - Questo perché i collisori (o vicini) si trovano nella stessa zona della lista
- Questa intuizione è estendibile a tutti i dati della simulazione

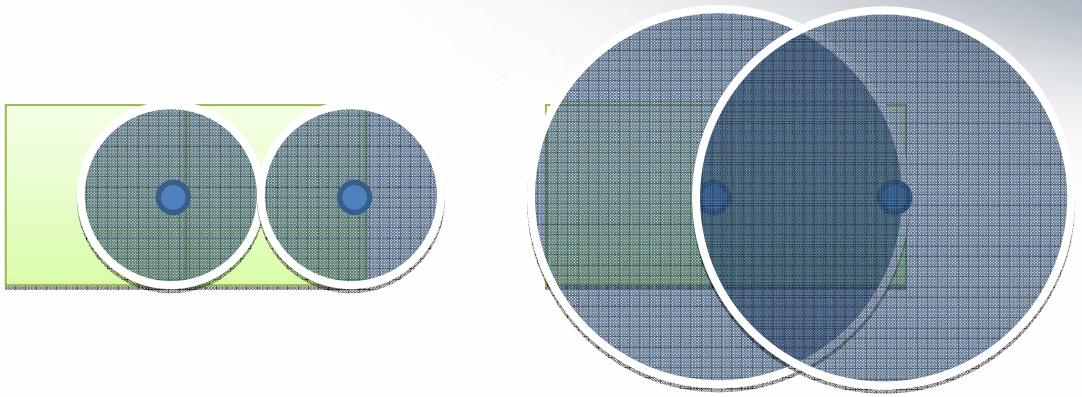


Collide: considerazioni

- La transizione da algoritmo per il calcolo di collisioni a calcolo del vicinato è breve
  - La differenza più sostanziale è nel fatto che le particelle possono sovrapporsi
  - Si perde l'upper bound sul numero di particelle che possono entrare in una cella, che era fissato a 4
- Vantaggi/svantaggi:
  - 👍 Efficiente
  - 👍 La griglia è ricalcolata ad ogni iterazione
    - *Teoricamente* adattabile, perlomeno pel centro e la dimensione
  - 👎 Il raggio massimo di ricerca è limitato proporzionalmente alla dimensione della cella

PDB: Integrazione in OS

- La dimensione della cella è stata scelta *accuratamente*
  - Dimensione cella = raggio (e non diametro) particella

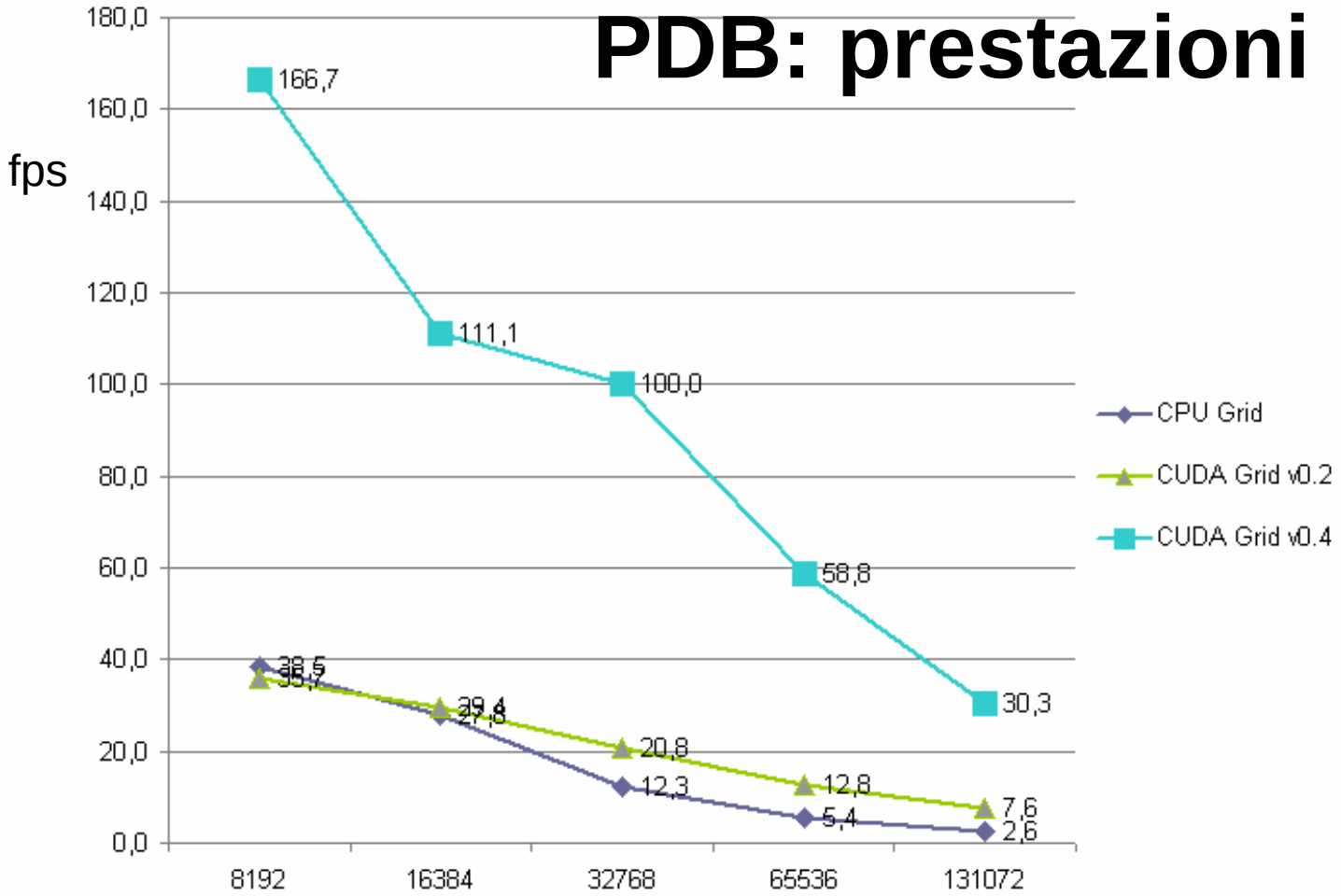


- Si è optato per una struttura dati contenente per ogni elemento (facile da implementare, ma non velocissima)
  - Numero vicini;
  - array dei vicini (impostato ad una dimensione *abbastanza* grande)

PDB: Configurazione e test

- Configurazione
  - Raggio boid: 0.5
  - Raggio di ricerca: 9
  - Dimensione cella: 9
- Dimensione ambiente e griglia proporzionali al numero di boid
  - Il numero medio di agenti per cella è mantenuto costante
- Le misurazioni sono state prese attendendo che si creassero dei gruppi e si stabilizzassero i tempi
  - Per effetto del comportamento di coesione
- Configurazione Hardware
  - CPU Intel Core Duo 2180 (2x2.0Ghz)
  - nVidia GeForce 8800GTX (128SP, 1.5Ghz)

PDB: prestazioni



- Versioni CUDA Grid:
  - v0.2: versione iniziale
  - v0.4: Dati ordinati secondo texture, nessuno trasferimento CPU-GPU

PDB: considerazioni

- Anche in questo caso l'ordinamento secondo l'hash porta benefici
  - Vettore di strutture neighbors
    - 8k boid: +43%
    - 16k biud: +36%
- Tenere presente che:
  - Non è utilizzata, nella ricerca del vicinato, quella che è ritenuta *il punto forte* di CUDA, ovvero la shared memory
  - Possibile aumento ulteriormente le prestazioni
    - Limitando il vicinato

Comportamenti: introduzione

Dettagli nell'Appendice D

- I comportamenti realizzati in CUDA sono quattro:
  - Separation
  - Cohesion
  - Alignment
  - Seeking (gira indietro quando al limite dello scenario)
- È stato portato su GPU anche l'aggiornamento di posizione, velocità, direzione e accellerazione in base al vettore forza calcolato
- Sono stati realizzati in CUDA, operazioni comuni come
  - Interpolazione tra vettori 3D (interpolazione)
  - Taglio della dimensione (truncatedLength)
  - Limitazione della forza di sterzata a basse velocità (vecLimitDeviationAngleUtility)

Comportamenti: considerazioni

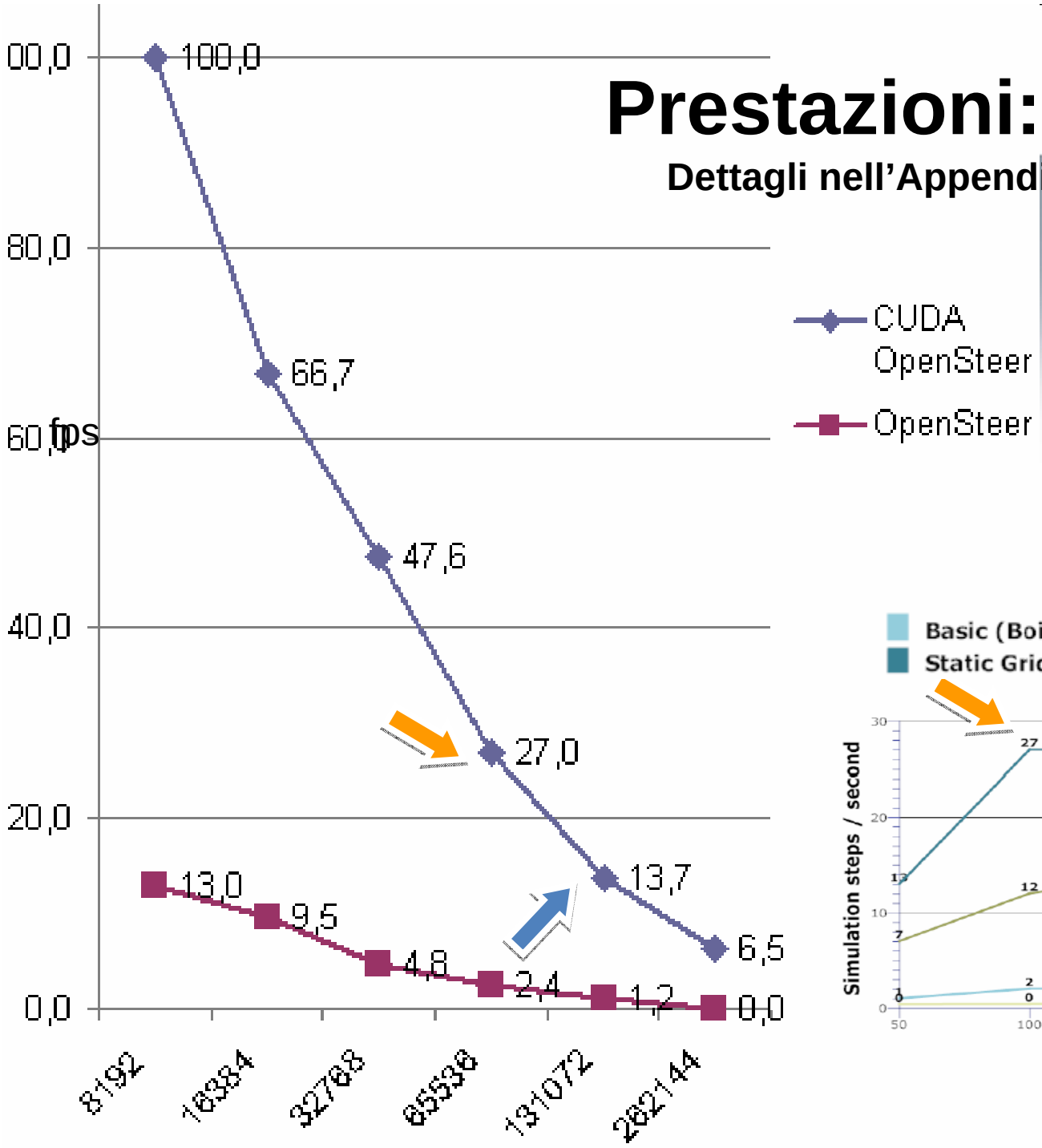
- La validazione è avvenuta confrontando i vettori 3D prodotti di ogni singolo comportamento e aggiornamento
- È garantito un errore massimo sulla differenza e sul dotProduct tra vettori host e device di $\mathbf{10^{-6}}$

```
Vec3 diff = hostForce - deviceForce;
float dotProduct = hostForce.dot(deviceForce);
float divergence = diff.length();
```

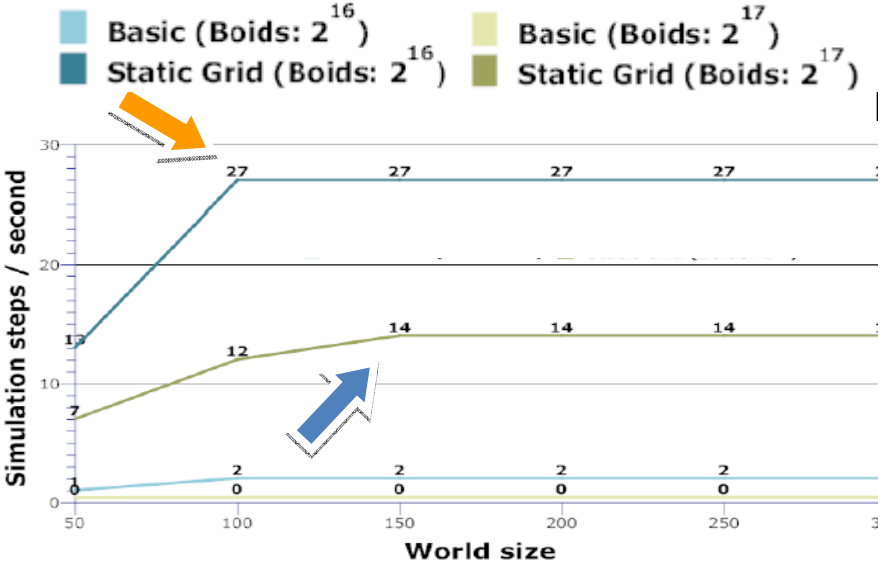
- Anche in questo caso, l'ordinamento delle informazioni in lettura ha portato benefici
  - Vettore Forward
    - 8k boid: +25%
    - 16k boid: +18%
  - Vettore di strutture neighbors
    - 8k boid: +33%
    - 16k boid: +33%

Prestazioni: fps

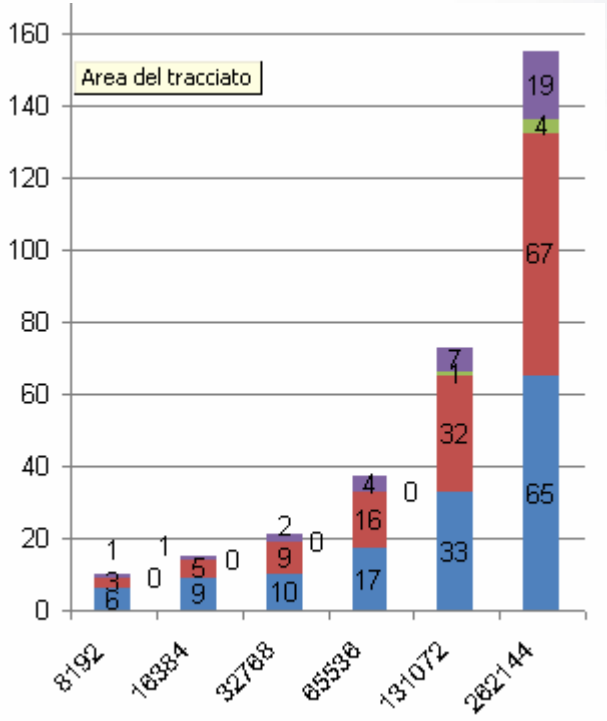
Dettagli nell'Appendice C-E



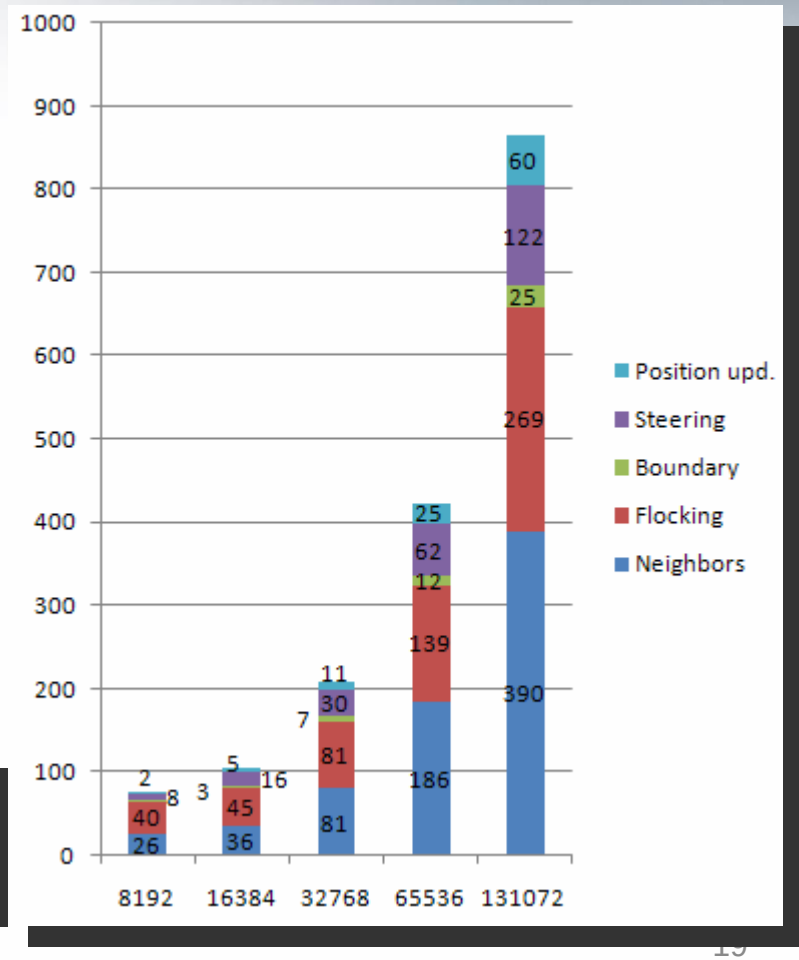
Jens Breitbart, Case studies on GPU usage and data structure design
KASSEL UNIVERSITY



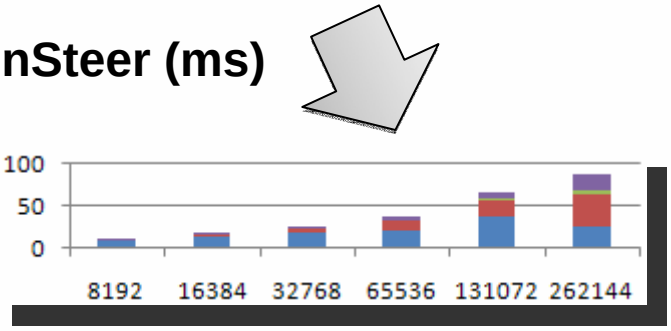
Conclusioni: tempi in ms



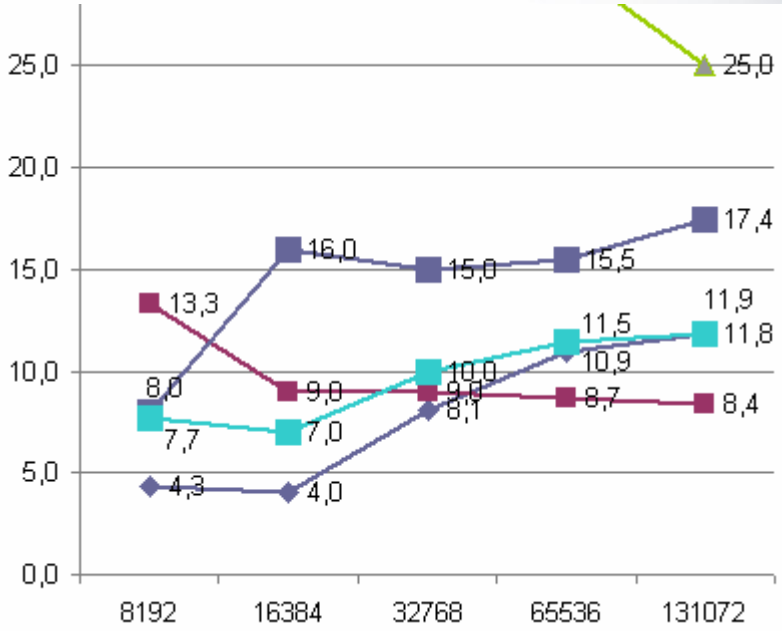
CPU OpenSteer (ms)

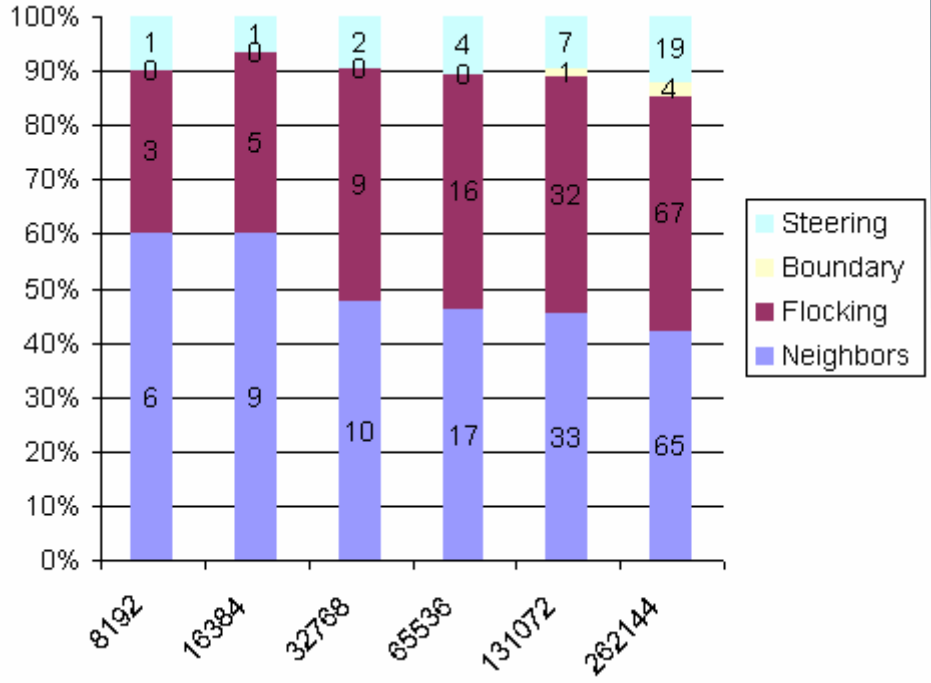


CUDA OpenSteer (ms)



Prestazioni: miglioramento sulla CPU



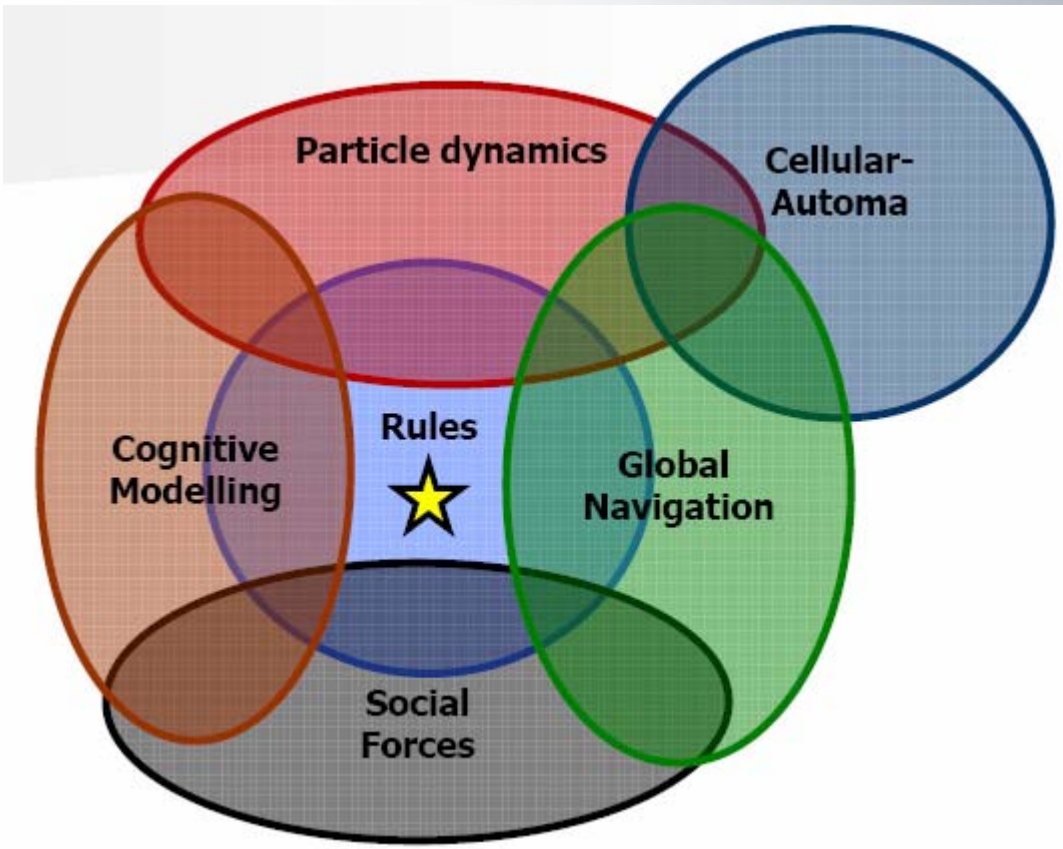


- Il margine maggiore si ha su steering e comportamenti
  - Questo perché sono operazioni che coivolgono molti calcoli matematici come interpolazioni, prodotti su vettori
- Il calcolo del vicinato rimane il peso maggiore, anche con CUDA
  - Con una percentuale del 50-75% del tempo di calcolo

International Conference on Computer Animation and Social Agents

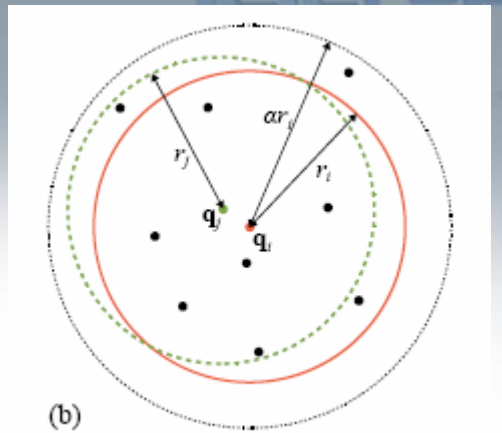
- L'edizione 2008 (settembre) ha introdotto una sezione dedicata al crowd simulation
  - **Adding Variation to Path Planning**
    - Ioannis Karamouzas (Utrecht University), Mark Overmars (Utrecht University)
  - **Managing Coherent Groups**
    - Renato Silveira (Universidade Federal do Rio Grande do Sul), Edson Prestes (Universidade Federal do Rio Grande do Sul), Luciana P. Nedel (Universidade Federal do Rio Grande do Sul)
  - **Shape Constrained Flock Animation**
    - Jiayi Xu (Zhejiang University), Xiaogang Jin (Zhejiang University), Yizhou Yu (University of Illinois at Urbana-Champaign), Tian Shen (Zhejiang University)
  - **A Social Agent Pedestrian Model**
    - Andrew Park (Simon Fraser University), Tom Calvert (Simon Fraser University)
  - **Agent-Based Human Behavior Modeling for Crowd Simulation**
    - Linbo Luo (Nanyang Technological University), Suiping Zhou (Nanyang Technological University), Wentong Cai (Nanyang Technological University), Malcolm Yoke Hean Low (Nanyang Technological University), Feng Tian (Nanyang Technological University)
  - **Dynamically Populating Large Urban Environments with Ambient Virtual Humans**
    - Murat Haciomeroglu (University of East Anglia), Robert Laycock (University of East Anglia), Andy Day (University of East Anglia)

Ambito


Particle dynamics
Cellular-
Automa
Rules
Cognitive
Modelling
Global
Navigation
Social
Forces

Conclusioni

- Ci sono ancora diverse ottimizzazioni da poter adottare
  - Scettering matrix (PDB)
  - Shared memory
    - Per l'accesso alle celle del PDB
    - A livello di calcolo dei comportamenti
  - Raggruppamento vicini (PDB)
    - Simon Heinzle, Gaël Guennebaud, Mario Botsch, Markus Gross. **A Hardware Processing Unit for Point Sets**. 2008



- Sulle prestazioni, Reynold ne parla a fine luglio 2008:
  - If you have not already, you should look at the PSCrowd system I wrote for the Cell processor. For a while it was the highest performance flock/crowd simulation, **until it was eclipsed by Bjoern's student's CUDA version**. Recent versions of PSCrowd can handle simple flocks with 15,000 to 20,000 members at 60 frames per second (with simple graphics). **One major differeence between the OpenSteer Boids plugin and PScrowd is the use of a maximum number of neighbors** ("N nearest neighbors" (aka "KNN")).

128.000 flocking boids with 16 simulation steps per second

Porterebbe molti benefici anche alla versione CUDA

Conclusioni (2)

- Quello che si può fare adesso è
  - Ampliare la gamma di comportamenti a disposizione, con la possibilità di adottare *qualcosa di più adatto alla GPU* come i potential field
  - Migliorare l'attuale versione: i comportamenti usano ancora alcuni dati non ordinati
  - Rendere la versione attuale di CUDA OS più compatibile con la versione standard
- Per le fasi successive (più ad alto livello)
  - È possibile conoscere quasi gratis le zone di alta densità (contando gli elementi presenti nelle celle)
  - La stessa nVidia ha pubblicato un lavoro su Path-finding con CUDA che potrebbe rivelarsi utile

Appendice A

Note su CUDA

- Le ottimizzazioni maggiori sono state ottenute mediante l'uso delle texture
  - **tex1Dfetch**: legge da texture
- Sono state usate inoltre:
  - **#pragma unroll**: "Srotola i cicli"
  - **__mul24**: esegue moltiplicazioni tagliando 8 bit, eseguendo moltiplicazioni in 4 cicli di clock anzicchè 16
  - **align 16**: usato per l'allineamento a 16 byte della struct contenente il vicinato
- È possibile ancora usare
  - **fdividef**: esegue divisioni più velocemente
    - Porterebbe benefici nella realizzazione dei comportamenti e nell'applicazione della forza di sterzata

Appendice A

Cose strane in CUDA

- Una chiamata a kernel che non prevede sincronizzazione di thread oppure non è seguita da una copia dei dati da device a host risulta essere **asincrona**
- Se si usa uno spazio di memoria non allocato:
  - non viene generato nessun tipo di messaggio o errore
  - Se si sta usando i VBO, sarà generato un errore di tipo GL_INVALID_OPERATION (!)
  - Ogni lettura di memoria risulta invalidata
  - Su alcune schede si ha una schermata nera (tutta la schermata) per un istante

Appendice B

nVidia Particle: prestazioni

Dati ottenuti con CUDA Profiler

		Device (ns)	Host (ns)	Occupancy (%)
44105,2	memcopy	628,128		
65772,6	__globfunc__Z9integrateP6float4S0_S0_S0_f	212,704	12,681	0,667
66443,4	__globfunc__Z9calcHashDP6float4P5uint2	81,952	7,316	1
66972,7	__globfunc__Z8RadixSumP12KeyValuePairjjj	291,424	15,366	0,25
66993,5	__globfunc__Z14RadixPrefixSumv	57,536	443,792	0,25
67455	__globfunc__Z25RadixAddOffsetsAndShuffleP12KeyValuePairS0_jji	397,344	795,216	0,25
68260,9	__globfunc__Z8RadixSumP12KeyValuePairjjj	291,488	4,525	0,25
68270	__globfunc__Z14RadixPrefixSumv	56,096	4,37	0,25
68283,5	__globfunc__Z25RadixAddOffsetsAndShuffleP12KeyValuePairS0_jji	395,296	4,57	0,25
68297,5	__globfunc__Z8RadixSumP12KeyValuePairjjj	291,232	4,85	0,25
68306,2	__globfunc__Z14RadixPrefixSumv	55,04	4,38	0,25
68319,6	__globfunc__Z25RadixAddOffsetsAndShuffleP12KeyValuePairS0_jji	394,656	4,32	0,25
68341,3	__globfunc__Z8RadixSumP12KeyValuePairjjj	293,024	4,68	0,25
68349,8	__globfunc__Z14RadixPrefixSumv	61,792	4,205	0,25
68363,3	__globfunc__Z25RadixAddOffsetsAndShuffleP12KeyValuePairS0_jji	420	4,665	0,25
72213,7	__globfunc__Z28reorderDataAndFindCellStartDP5uint2P6float4S2_S2_S2_Pj	675,616	39,564	1
73402,4	__globfunc__Z8collideDP6float4S0_S0_S0_P5uint2PjP9neighType	6645,92	24,637	0,25
80527,3	memcopy	2,72		

Ordinamento

3ms

Shared mem

Totale 10ms

Appendice C

PDB CUDA OS: prestazioni

Method		GPU Time	CPU Time	Occupancy
__globfunc__Z9calcHashDP6float4P5uint2	Uguale	202,752	7,166	1
__globfunc__Z8RadixSumP12KeyValuePairjjj	Uguale	269,952	5,561	0,25
__globfunc__Z14RadixPrefixSumv	Uguale	50,368	4,155	0,25
__globfunc__Z25RadixAddOffsetsAndShuffleP12KeyValuePairS0_jji	Uguale	364,192	4,44	0,25
__globfunc__Z8RadixSumP12KeyValuePairjjj		270,08	4,435	0,25
__globfunc__Z14RadixPrefixSumv		51,808	3,985	0,25
__globfunc__Z25RadixAddOffsetsAndShuffleP12KeyValuePairS0_jji		362,272	4,08	0,25
__globfunc__Z8RadixSumP12KeyValuePairjjj		269,728	4,075	0,25
__globfunc__Z14RadixPrefixSumv		51,616	3,845	0,25
__globfunc__Z25RadixAddOffsetsAndShuffleP12KeyValuePairS0_jji		363,712	4,225	0,25
__globfunc__Z8RadixSumP12KeyValuePairjjj		270,624	4,035	0,25
__globfunc__Z14RadixPrefixSumv		50,656	3,86	0,25
__globfunc__Z25RadixAddOffsetsAndShuffleP12KeyValuePairS0_jji		394,976	4,085	0,25
__globfunc__Z28reorderDataAndFindCellStartDP5uint2P6float4S2_P6float3S2_S2_S4_Pj	Era 700	1630,53	5,291	0,667
__globfunc__Z13neighborhoodDP6float4S0_P5uint2PjP9neighType	Era 6500	14930,7	4,93	0,5
__globfunc__Z11separationDP6float4P5uint2P6float3S4_S4_P9neighTypeP9debugType		6670,43	5,141	0,5
__globfunc__Z10alignmentDP6float4P5uint2P6float3S4_S4_P9neighTypeP9debugType		7646,21	5,271	0,5
__globfunc__Z9cohesionDP6float4P5uint2P6float3S4_S4_P9neighTypeP9debugType		7205,47	5,236	0,5
__globfunc__Z19seekingWorldCenterDP6float4S0_P5uint2P6float3S4_S4_P9neighTypefP9debugType		1528,86	4,725	0,833
__globfunc__Z19applySteeringForceDP6float4S0_S0_S0_PfS1_P6float3S3_S3_S3_S3_P5uint2fP9debugType		6342,24	5,181	0,333

Ordinamento

3ms

Comportamenti

Appendice D

Comportamenti: implementazione

- Su CPU

```
for (AVIterator other = flock.begin(); other != flock.end(); other++)
    {
        if (inBoidNeighborhood (**other, radius()*3, maxDistance, cosMaxAngle))
        {
            steering += (**other).position();
            neighbors++;
        }
    }

    if (neighbors > 0)
         steering = ((steering / (float)neighbors) - position()).normalize()
```

- In CUDA

```
for (int i = 0; i < neighborhood.neighNum; i ++)
    {
         otherIndex = neighborhood.neighborsIndex[i];
         otherPos = make_float3((float4)FETCH(oldPos, otherIndex));

         if (inBoidNeighborhood(myPos, otherPos, 1.5, 9, -0.15, myForward, debugData))
         {
             steering = steering + otherPos;
             neighbors ++;
         }
    }
    if (neighbors > 0)
         steering = normalize((steering / (float)neighbors) - myPos);
```