**ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA**

---

**FACOLTA' DI INGEGNERIA**

**CORSO DI LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA**
*Dipartimento di Informatica – Scienza e Ingegneria*

**TESI DI LAUREA**

in
Processi e tecniche di Data Mining

# IMPLEMENTAZIONE DI ALGORITMI DI DATA MINING IN ARCHITETTURE A ELEVATO PARALLELISMO

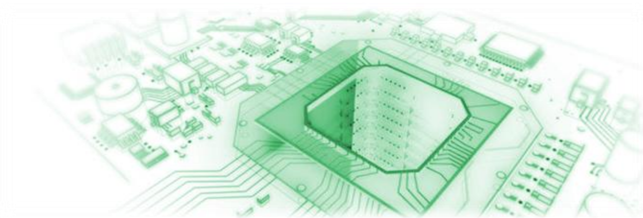
CANDIDATO
Matteo Zanirati

RELATORE:
Chiar.mo Prof. Ing. Claudio Sartori

CORRELATORI
Prof. Ing. Stefano Lodi
Ing. Stefano Basta

Anno Accademico 2011/12
Sessione III

*Dedicato a mia nonna*
*Maria Teresa*



# Sommario

Capitolo 1: Introduzione ........ 9

Capitolo 2: Outlier Detection Problem, algoritmi ODP Nested Loop e ODP Solving Set ........ 13

2.1 Principali tecniche di Outlier Detection ........ 14

2.2 Definizione del problema ........ 20

2.3 Algoritmo ODP Nested Loop ........ 21

2.4 Algoritmo ODP Solving Set ........ 25

2.5 Risultati sperimentali ........ 31

2.5.1 Cardinalità del solving set ........ 32

2.5.2 Confronto ODP Solving Set – ODP Nested Loop ........ 34

Capitolo 3: GPU Computing e architettura NVIDIA CUDA ........ 37

3.1 Evoluzione dell'architettura delle GPU ........ 38

3.2 NVIDIA CUDA (Compute Unified Device Architecture) ........ 43

Capitolo 4: Tecniche di GPU Computing per il problema *k-NN* ........ 53

4.1 Soluzioni GPGPU note in letteratura per l'approccio BF al *k-NN* problem ........ 54

4.2 Nuove soluzioni GPGPU proposte per l'approccio BF al *k-NN* problem ........ 60

4.3 Confronti teorici e risultati sperimentali ........ 65

4.3.1 Calcolo delle distanze ........ 66

4.3.2 Selezione delle minori *k* distanze ........ 69

Capitolo 5: Implementazione in CUDA di ODP Nested Loop ........ 79

5.1 Calcolo delle distanze ........ 81

5.2 Selezione delle minori *k* distanze ........ 83

5.2.1 Metodo Single heap ........ 83

5.2.2 Metodo di Kato ........ 84

5.2.3 Metodo Heap complete reduction ........ 85

5.3 Trasposta di un chunk ........ 87

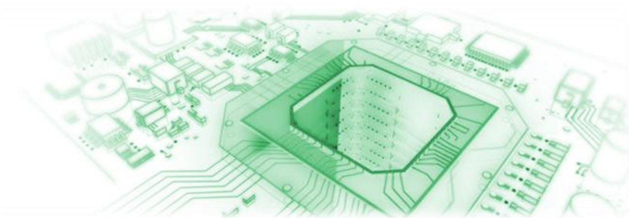
5.4 Selezione dei top *n* outlier ........ 89
5.5 Complessità computazionale ........ 92
5.6 Risultati sperimentali ........ 93
5.6.1 Confronto con ODP Nested Loop ........ 94
5.6.2 Confronto con ODP Solving Set ........ 96
Capitolo 6: Implementazione in CUDA di ODP Solving Set ........ 99
6.1 Fase di inizializzazione ........ 101
6.2 Fase di confronto *C* con *C* ........ 103
6.3 Fase di confronto *D* con *C* - upMin ........ 104
6.3.1 Modalità con matrice delle distanze ........ 104
6.3.2 Modalità senza matrice delle distanze ........ 113
6.4 Fase di confronto *D* con *C* - upMax ........ 118
6.5 Fase di selezione dei nuovi candidati ........ 120
6.6 Complessità computazionale ........ 120
6.7 Risultati sperimentali ........ 122
6.7.1 Confronto sui dataset tradizionali ........ 123
6.7.2 Confronto su dataset gaussiani di grandi dimensioni ........ 126
Capitolo 7: Algoritmo distribuito ODP Distributed Solving Set e implementazione in architetture distribuite multi-GPU ........ 129
7.1 Algoritmo distribuito ODP Distributed Solving Set ........ 131
7.1.1 Variante ODP Lazy Distributed Solving Set ........ 138
7.2 Implementazione dell’algoritmo distribuito ODP Distributed Solving Set in una architettura multi-GPU ........ 142
7.2.1 Procedura CUDA_NodeInit ........ 143
7.2.2 Procedura CUDA_NodeComp ........ 144
7.2.3 Complessità computazionale ........ 152
7.3 Risultati sperimentali ........ 154
7.3.1 Performance di ODPLazyDistributedSolvingSet ........ 155

7.3.2 Performance di CUDA_ODPLazyDistributedSolvingSet ........................ 157

Capitolo 8: Conclusioni e sviluppi futuri ................................................................... 163

Ringraziamenti ........................................................................................................... 165

Bibliografia ................................................................................................................. 167



# Capitolo 1: Introduzione

Il problema dell'individuazione delle anomalie in un insieme di dati (*Outlier Detection Problem - ODP*) costituisce un ramo molto importante nell'ambito delle tecniche di *Data Mining*. Questo problema presenta numerose applicazioni in diversi contesti, come nell'individuazione delle frodi, nella rivelazione di intrusioni in sistemi informatici, nei sistemi di supporto alle diagnosi mediche, nel marketing e in molti altri ancora. La comunità di ricerca ha proposto molte soluzioni, alcune più specifiche per determinati campi applicativi, altre più generiche [1].

Le tecniche supervisionate si basano sulla definizione di un modello, creato a partire da un insieme campione di dati detto *training set* (i cui oggetti siano stati preventivamente classificati come normali o anomali), tramite il quale valutare un nuovo oggetto come normale o anomalo (*outlier*), a seconda di come questo si presenti conforme al modello.

Le tecniche non supervisionate, invece, non richiedono la presenza di un training set e si basano sull'assunzione implicita che gli oggetti normali siano in numero nettamente superiore rispetto a quelli anomali. Tra gli approcci non supervisionati, i metodi *distance-based* distinguono un oggetto come outlier o normale sulla base delle distanze verso i rispettivi oggetti più vicini (*nearest neighbor*), nell'insieme di dati. Dato un dataset $D$, è possibile associare ad ogni oggetto $p$ del dataset un punteggio $w_k(p, D)$, detto *peso* o *anomality score*, che è funzione delle distanze dai $k$ oggetti più vicini a $p$ in $D$ ($k$-nearest neighbor). Tale valore permette di misurare quanto l'oggetto $p$ sia *dissimilare* dai rispettivi $k$-nearest neighbor in $D$. Tra le varie tecniche distance-based proposte in letteratura sono state fornite diverse possibili definizioni di anomality score. In questo lavoro utilizziamo la definizione fornita da *Angiulli et. al.* in [2]. Fissato un numero $n$, sia $w^*$ l'$n$-esimo maggior peso di un oggetto in $D$. Possiamo definire come outlier rispetto a $D$ un oggetto che presenti un anomality score maggiore o uguale a $w^*$. Inoltre, come peso di un oggetto $p$ in $D$, consideriamo la somma delle distanze di $p$ dai rispettivi $k$-nearest neighbor.

Il più semplice approccio per individuare gli $n$ oggetti dotati del maggior anomality score consiste nell'utilizzare un algoritmo di tipo *nested loop*. L'idea di base è quella di calcolare, per ogni oggetto $p \in D$, tutte le distanze dagli altri oggetti del dataset e di

quantificare il rispettivo peso $w_k(p, D)$ come somma delle minori $k$. Infine si dichiarano come outlier gli $n$ oggetti di peso maggiore. Tale algoritmo richiede di calcolare una quantità molto elevata di distanze (proporzionale a $O(|D|^2)$) e per questo motivo può non essere applicabile nella pratica a dataset di dimensioni molto elevate.

Un diverso approccio, molto più efficiente e sofisticato, è quello proposto da *Angiulli, Basta e Pizzuti* in [3], che si basa sul concetto di *Outlier Detection Solving Set* (o detto più brevemente *solving set*). Il *solving set* $S$ è un sottoinsieme del dataset $D$, che include un numero sufficiente di oggetti appartenenti a $D$, tali da permettere di considerare solamente le distanze tra le coppie in $S \times D$, per ottenere i top $n$ outlier. L'algoritmo proposto permette di determinare il solving set $S$ e risolvere l'ODP, evitando il calcolo di tutte le distanze, per ogni coppia di oggetti in $D$.

Nel caso di dataset di dimensioni molto elevate però neanche un approccio così sofisticato può risultare sufficiente, soprattutto nel caso in cui si vogliano tempi di risposta al problema ODP particolarmente brevi. Inoltre, nei sistemi informativi reali, molto spesso i dataset sono distribuiti su diversi nodi della rete, che possono essere anche molto numerosi. Una soluzione tradizionale, che preveda il trasferimento di tutti i dati provenienti dai diversi nodi all'interno di un'unica unità di storage centralizzata (come un *data warehouse*), su cui successivamente eseguire algoritmi di data mining, può non essere la scelta migliore. Nell'ambito dell'outlier detection *Angiulli, Basta, Lodi* e *Sartori* in [4] hanno presentato un efficace metodo distribuito per dataset di grandi dimensioni, basato su una generalizzazione del concetto di *solving set* al caso dei dataset distribuiti. Tale algoritmo permette di suddividere il carico di lavoro tra i diversi nodi, sfruttando l'azione contemporanea di più processori. Sperimentalmente si è mostrato come questo approccio fornisca dei tempi di esecuzione in grado di scalare quasi linearmente al crescere del numero di nodi utilizzati.

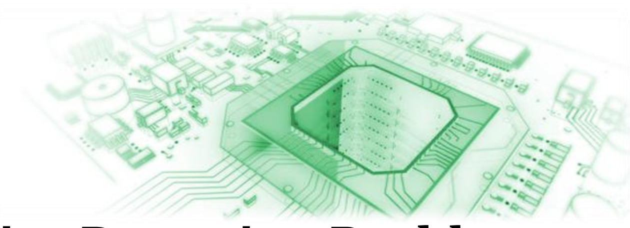
Una nuova e innovativa soluzione, per cercare di ridurre notevolmente i tempi di esecuzione, consiste nel combinare questi due approcci con tecniche di *GPU computing*. Con il passare degli anni e guidate dalla forte domanda di mercato per la grafica *3D*, le *GPU* (*Graphic Processor Unit*) si sono evolute in processori programmabili fortemente paralleli, multithreaded e multicore, dotati di un'impressionante potenza di calcolo e di un'elevata banda nei trasferimenti di memoria. La rapida crescita della potenzialità delle GPU e l'aumento della possibilità

di programmabilità, ha spinto molte comunità di ricerca nel cercare di mappare diverse tipologie di problemi, richiedenti una forte carico computazionale, su tali processori. Questo fenomeno prende il nome di *general purpouse computing on the GPU* (*GPGPU*), o più semplicemente *GPU Computing*. Le GPU sono adatte a risolvere problemi che possono essere espressi come operazioni di calcolo parallele, in cui lo stesso programma possa essere eseguito su più dati in parallelo, con un'alta intensità delle operazioni aritmetiche, rispetto a trasferimenti di memoria. Per permettere agli sviluppatori software di programmare con facilità tali dispositivi, sono nati diversi linguaggi di alto livello. Tra di essi spicca l'architettura *NVIDIA CUDA*, che offre un modello di programmazione basato su un linguaggio C/C++, per lo sviluppo di applicazioni GPGPU su dispositivi *NVIDIA*.

Gli algoritmi nested loop e solving set presentano per loro natura una buona propensione ad una efficace implementazione parallela su GPU. Il nostro obiettivo è quello di fornire una nuova rivisitazione di tali algoritmi, sfruttando la potenza di calcolo delle GPU per ottenere performance molto elevate. Dai risultati sperimentali emerge come sia possibile ottenere, tramite una soluzione GPGPU (con una GPU *NVIDIA Tesla M2070*), dei tempi di esecuzione 80-100 volte inferiori rispetto a quelli di una tradizionale soluzione basata su CPU. Inoltre, utilizzando la versione distribuita dell'algoritmo, è stato possibile raggiungere uno speedup pari a 450, combinando la potenza di calcolo di 10 GPU, rispetto all'algoritmo centralizzato eseguito su una sola CPU.

Questo lavoro è organizzato nel modo seguente.
Nel capitolo 2 è mostrata una panoramica dei principali metodi di individuazione delle anomalie noti in letteratura e sono descritti in maniera dettagliata gli algoritmi nested loop e solving set.
Nel capitolo 3 è illustrata l'evoluzione dell'architettura delle GPU e introdotta la tecnologia *NVIDIA CUDA*.
Nel capitolo 4 sono descritte varie tecniche GPGPU proposte in letteratura per risolvere un problema strettamente legato agli approcci distance-based per l'ODP, ovvero il *k-NN problem*.
Nel capitolo 5 e nel capitolo 6 sono proposte le nostre soluzioni GPGPU per eseguire in maniera parallela, rispettivamente, l'algoritmo nested loop e l'algoritmo solving set.
Nel capitolo 7 è mostrata la versione distribuita dell'approccio basato sul solving set e la nostra relativa implementazione GPGPU.

Infine, nel capitolo 8 sono presentati dei possibili sviluppi futuri di questo lavoro e nuovi potenziali campi di applicazione.



# Capitolo 2: Outlier Detection Problem, algoritmi ODP Nested Loop e ODP Solving Set

Il problema della rivelazione delle anomalie (*Outlier Detection Problem*) consiste nell'individuare dei pattern, in un insieme di dati, che differiscono dal normale comportamento aspettato [1]. Questi pattern discordanti possono essere indicati con diversi nomi, a seconda del dominio di riferimento: anomalie (*anomaly* o *outlier*), osservazioni discordanti, eccezioni, aberrazioni, peculiarità, agenti contaminanti, ecc. La rivelazione delle anomalie trova applicazioni molto importanti in diversi contesti. Nell'individuazione delle frodi, i sistemi di rilevazione delle anomalie vengono spesso utilizzati per individuare possibili furti di carte di credito. In caso di furto, il malintenzionato inizia tipicamente ad effettuare un numero molto elevato di acquisti, il che si differenza fortemente dal normale comportamento di un possessore di carta di credito. L'individuazione di *buying patterns* inconsueti può indicare un potenziale furto. In ambito medico, il presentarsi di sintomi o di risultati di test non ordinari può evidenziare possibili problemi di salute nel paziente. Nei sistemi di rivelazione delle intrusioni nei computer e nelle reti, è possibile individuare potenziali attacchi monitorando le attività correnti e confrontandole con dei pattern che rappresentano il normale comportamento del sistema. L'azione di programmi pirata o di hacker che si intromettono in una rete è tipicamente molto discordante dal comportamento ordinario ed un sistema di outlier detection può quindi riconoscere l'attacco. Altre applicazioni importanti si hanno nell'individuazione di problemi negli impianti industriali, nella sicurezza militare, nell'*image processing*, nelle reti di sensori, ecc.

**Figura 2-1 Rilevazione delle anomalie per l'individuazione del furto di una carta di credito**

*Hawkins* [5] definisce un'anomalia (*anomaly* o *outlier*) come un'osservazione che devia in maniera così evidente rispetto alle altre osservazioni, da alimentare sospetti che sia generata tramite un differente meccanismo. Questa definizione è basata su considerazioni statistiche ed assume che gli oggetti normali seguano un comune "meccanismo di generazione", come un determinato processo statistico. Gli oggetti che si discostano da questo meccanismo vengono considerati outlier. Possiamo generalizzare tale definizione e definire le anomalie come dei pattern che non rispecchiano il normale comportamento atteso. Un primo approccio alla rivelazione delle anomalie è quindi quello di definire una regione che rappresenti il comportamento normale e dichiarare come outlier ogni osservazione che non appartenga a tale regione. Diversi fattori rendono però questo approccio particolarmente complesso. Definire una regione normale, che rappresenti ogni possibile comportamento ordinario, è molto difficile e tipicamente il confine tra comportamento normale e anomalo non è così netto. Inoltre, in molti casi, i pattern che rappresentano il comportamento normale sono in continua evoluzione e una rappresentazione attuale di comportamento normale può non essere valida nel futuro. Infine, la definizione di anomalia può essere differente a seconda dello specifico dominio. Ad esempio, in ambito medico una piccola deviazione dalle normali condizioni (come le fluttuazioni della temperatura corporea) possono indicare un'anomalia, mentre nei mercati azionari le fluttuazioni dei prezzi possono essere considerati normali. A questo si aggiunge la disponibilità o meno di dati già classificati come normali o outlier, per poter essere utilizzati come modelli di training o di validazione, per le tecniche di rilevazione delle anomalie. A causa di tutti questi problemi, sono nate diverse tecniche e diverse definizioni dell'outlier detection problem, alcune più generali, altre più specifiche per particolari domini. Le diverse formulazioni del problema possono dipendere dalla natura dei dati, dalla disponibilità o meno di dati già classificati, dal tipo di anomalie che devono essere ricercate e da molti altri fattori.

## 2.1 Principali tecniche di Outlier Detection

A seconda della disponibilità o meno di dati già classificati come normali o anomali, possiamo dividere le tecniche di outlier detection in supervisionate, semi-supervisionate e non supervisionate.

Le tecniche *supervisionate* (*supervised outlier detection*) assumono di avere disponibile un *training set,* le cui istanze siano state divise in almeno due distinte

classi: *normal* e *outlier*. L'idea alla base di queste tecniche è di costruire un modello di predizione, basato sul training set, per determinare se un futuro elemento possa essere considerato come normale o come outlier. Si possono avere inoltre classi multiple per le due categorie, in modo da ottenere una suddivisione più accurata. Le tecniche semi-supervisionate (*semi-supervised outlier detection*) assumono invece la disponibilità di un *training set*, le cui istanze appartengano solamente alla classe normale. Siccome non richiedono la presenza di istanze classificate come outlier, sono maggiormente applicabili rispetto a quelle supervisionate. Gli approcci supervisionati e semi-supervisionati si basano tipicamente sulle classiche tecniche di classificazione, come reti neurali, classificatori bayesiani, e support vector machines (SVM). Questi approcci hanno il vantaggio di potersi basare su potenti algoritmi per la distinzione tra le diverse classi ed inoltre la fase di testing di nuove istanze (dopo l'addestramento del classificatore) è molto veloce. Purtroppo però non è spesso possibile ottenere un training set abbastanza ampio che possa rappresentare fedelmente le due distinte classi, in particolar modo per quella delle anomalie (solo negli approcci supervisionati). Questo rappresenta un grosso limite di tali approcci, soprattutto in particolari domini.

Le tecniche non supervisionate (*unsupervised outlier detection*) non richiedono la presenza di un traning set e sono quindi, in generale, le più applicabili. Si basano sull'assunzione implicita che le istanze normali siano in numero nettamente superiore rispetto a quello delle anomalie. Se tale ipotesi non risulta essere veritiera, queste tecniche possono soffrire di un numero molto elevato di rivelazioni errate. Dalla comunità di ricerca sono stati sviluppati moltissimi approcci non supervisionati, che si possono raggruppare in diverse categorie.

Gli approcci di tipo *statistico* assumono la presenza di un modello statistico, che descriva la distribuzione dei dati (come la distribuzione normale) e poi applicano dei test di inferenza statistica per determinare se un'istanza possa appartenere o meno a tale modello. Le istanze con una bassa probabilità di poter essere state generate da quel modello, secondo il test effettuato, sono dichiarate come outlier. Si hanno molti approcci che differiscono dal tipo e dal numero di distribuzioni assunte, dal numero di variabili (*uni variate/multivariate*) e se si tratta di tecniche parametriche o non parametriche.

Le tecniche parametriche assumono la conoscenza della sottostante distribuzione e dei corrispondenti parametri, mentre quelle non parametriche non effettuano tale

assunzione. Come esempio di tecnica parametrica possiamo citare il *test di Grubb*. Tale approccio assume la distribuzione normale e permette di determinare potenziali outlier in un dataset univariato. Per ogni istanza $X$ viene calcolato un punteggio $z$ nel seguente modo:

$$z = \frac{|X - \bar{X}|}{s},$$

dove $\bar{X}$ e $s$ rappresentano rispettivamente la media e la deviazione standard.
L'istanza è considerata un outlier se:

$$z > \frac{N-1}{\sqrt{N}} \sqrt{\frac{t^2_{\alpha/(2N),N-2}}{N-2+t^2_{\alpha/(2N),N-2}}}$$

dove $N$ è la cardinalità del dataset e $t^2_{\alpha/(2N),N-2}$ una soglia utilizzata per dichiarare l'istanza come anomala o meno (ottenuta dalla distribuzione $t$ al livello di significatività $\frac{\alpha}{2N}$).
Un esempio invece di tecnica non parametrica è l'approccio *histogram-based*. Nel caso di dati univariati, viene costruito un istogramma basandosi sui differenti valori di una *feature* e ad ogni istanza viene assegnato un punteggio, inversamente proporzionale all'altezza (cioè la frequenza) del *bin* a cui essa appartiene. Le istanze con un alto punteggio vengono considerate come outlier.
Gli approcci di tipo statistico hanno il vantaggio di basarsi su modelli statistici e di fornire soluzioni giustificabili da un punto di vista statistico. Inoltre, tipicamente, l'*anomaly score* fornito è associato con un intervallo di confidenza, che può fornire un'informazione importante. Hanno però il principale svantaggio che, non in tutti i casi, basarsi sull'assunzione che i dati siano generati da una particolare distribuzione può essere sempre corretto. C'è infine da sottolineare che in molti scenari non è semplice individuare la giusta tecnica statistica da utilizzare, soprattutto nel caso di dataset ad alta dimensionalità.

I metodi *clustering-based* sfruttano le note tecniche di clustering, per l'individuazione degli outlier. Si basano sull'assunzione che i dati normali appartengano ai cluster più grandi e densi, mentre gli outlier a cluster molto piccoli e poco densi. Vengono quindi dichiarate come anomale le istanze che appartengono a cluster la cui dimensione e/o densità sia al di sotto di una particolare soglia. Dalla comunità di ricerca sono stati proposti diversi metodi per poter incrementare l'efficienza delle tecniche di clustering, nel caso specifico di utilizzo per l'outlier detection. Gli approcci clustering-based hanno il vantaggio di poter essere applicati anche a dataset molto complessi,

scegliendo degli algoritmi di clustering che supportino tali dati. Inoltre la fase di testing, successiva alla generazione dei cluster su un training set, è tipicamente molto veloce, in quanto il numero dei cluster è inferiore al numero delle singole istanze. Soffrono però del problema di essere strettamente legati alle performance degli algoritmi di clustering e di essere difficilmente ottimizzati per l'outlier detection.

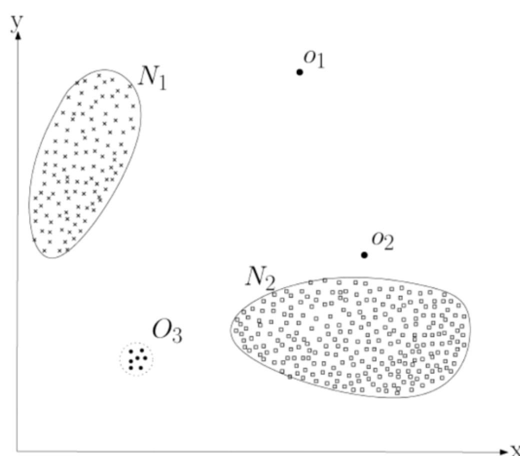



**Figura 2-2 Un esempio di anomalie in un dataset bi-dimensionale. I cluster $N1$ e $N2$ possono essere considerati come classi normali, mentre $O1$, $O2$ e $O3$ possono essere considerati come outlier**

Gli approcci *nearest-neighbor based* sfruttano il concetto di *neighborhood* (vicinato) di un punto per determinare le anomalie, tipicamente analizzando i $k$ punti più vicini (*$k$-nearest-neighbors – $k$-NN*) ad ogni punto. L'assunzione alla base di tali metodi è che i dati ordinari siano caratterizzati dalla presenza di neighborhood densi, mentre gli outlier siano relativamente più distanti dai propri vicini e quindi dotati di neighborhood meno densi. Le tecniche facenti parte di questa categoria possono essere a loro volta divise in due gruppi: tecniche *distance-based*, che assegnano ad ogni elemento un punteggio (*anomaly score*) che dipende dalle distanze dei *$k$-nearest-neighbors* e tecniche *density-based*, che utilizzano una misura della densità relativa di ogni istanza come anomaly score.

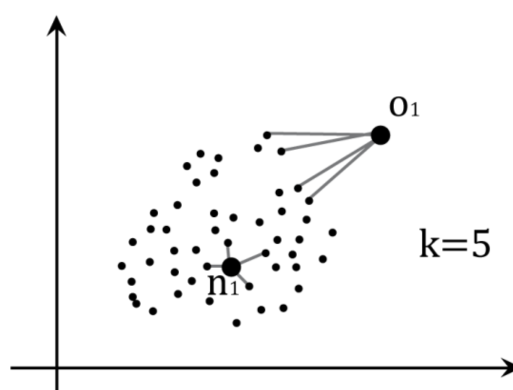



**Figura 2-3 *$k$-NN* di un punto normale e di un punto anomalo**

Nei metodi *distance-based* si sono susseguite diverse definizioni di outlier. *Knorr* e *Ng* [6] hanno definito gli outlier come i punti per cui si hanno meno di $k$ punti nel dataset, all'interno di un raggio $\delta$. Questa definizione non permette però di effettuare un ranking degli outlier e necessita di un dimensionamento corretto del valore $\delta$, per l'ottenimento di risultati significativi. I due autori hanno proposto due algoritmi, uno di tipo *nested-loop* richiedente un tempo $O(dN^2)$, con $d$ pari alla dimensionalità dei punti ed $N$ al numero di punti del dataset, ed un altro *ceil-based*, lineare rispetto ad $N$, ma esponenziale rispetto a $d$, rivelatosi efficace solo per $d \leq 4$. Successivamente *Ramaswamy et al.* [7] hanno formulato una diversa definizione di outlier, considerando come anomalie i primi $n$ punti, la cui distanza dal $k$*-esimo* nearest neighbor è maggiore. Per la rivelazione degli outlier viene utilizzato un algoritmo *partition-based*, che suddivide i punti tramite un algoritmo di clustering e poi elimina le partizioni che non possono contenere outlier. Gli outlier vengono quindi identificati tramite uno step finale, che considera solamente le istanze rimaste. Gli esperimenti hanno dimostrato una buona scalabilità di questo algoritmo, rispetto sia alla dimensione dei punti, che alla cardinalità del dataset. Tale definizione però non tiene conto delle informazioni contenute nell'intero $k$-neighborhood di ogni punto e può quindi presentare problemi nella distinzione tra neighborhood densi e neighborhood poco densi. *Angiulli et al.* [2] hanno introdotto una nuova definizione di outlier, che considera l'intero neighborhood. Ad ogni punto $p$ del dataset viene associato un peso $w_k(p)$, pari alla somma delle distanze dei $k$-nearest neighbors di $p$. Vengono quindi considerati come outlier i punti con il maggior peso $w_k(p)$. Questa definizione porta ad una misura più accurata di quanto un punto $p$ possa essere considerato come un'anomalia, in quanto tiene maggiormente conto della densità del neighborhood del punto. Per poter trovare i $k$*-NN* di ogni punto e calcolare i pesi, gli autori hanno proposto una tecnica che si basa sulla linearizzazione dello spazio di ricerca, tramite la *Hillbert space filling curve*. Questo algoritmo permette di lavorare bene in dataset ad alta dimensionalità e scala linearmente rispetto sia alla dimensionalità, che alla cardinalità del dataset. Più recentemente *Bay e Schwabacher* [8] hanno proposto un algoritmo di tipo nested-loop, che trova i $k$*-NN* di ogni punto nel dataset, sfruttando una semplice regola di pruning e introducendo un certo grado di "randomizzazione". Questa tecnica ha permesso di ottenere tempi in grado di scalare in modo quasi lineare su una serie di test effettuati su dataset reali, ad alta dimensionalità e dotati di un numero molto elevato di elementi.

I metodi *density-based* stimano la densità del neighborhood di ogni punto del dataset e dichiarano come outlier i punti che risiedono in neighborhood a bassa densità. Nascono dall'osservazione che le tecniche distance-based tendono a presentare dei problemi nel caso siano presenti più zone di dati a differenti densità. Per poter fronteggiare questo problema, sono nate diverse tecniche. *Breuning et al.* [9] assegnano ad ogni istanza un anomaly score, noto come *Local Outlier Factor* (*LOF*). Dato un punto $p$ del dataset, il *LOF score* di $p$ è definito come il rapporto tra la densità media locale dei $k$-NN di $p$ e la densità locale del punto stesso. Per il calcolo della densità locale di $p$, viene trovato il raggio dell'ipersfera centrata nel punto contenente i k-NN e posto il valore della densità locale pari al volume della ipersfera diviso $k$. Per un'istanza normale che appartiene ad una regione densa, la densità locale del punto sarà simile a quella dei propri nearest neighbors ($LOF \cong 1$), mentre per un outlier tale valore sarà inferiore a quello dei vicini ($LOF \gg 1$). In letteratura sono state inoltre proposte diverse varianti, per poter ottenere minori tempi di computazione o risultati più efficaci (come *COF - Connectivity-based Outlier Factor*, *INFLO - Influenced Outlierness* e diverse altre).

I metodi *nearest-neighbor based* hanno il vantaggio, oltre ad essere non supervisionati e non dipendenti da assunzioni sulla distribuzione statistica dei dati, di potersi adattare a differenti tipologie di dati, richiedendo semplicemente la definizione di una misura di distanza tra le istanze. Inoltre tipicamente non è richiesto che tale funzione di distanza sia una *metrica* in senso stretto; nella maggior parte dei metodi si assume solamente che sia *definita positiva* e *simmetrica*, ma è non obbligatorio il soddisfacimento della *disuguaglianza triangolare*. In alcuni contesti però, con dati particolarmente complessi, la definizione di misure di distanza può essere molto complicato (come, ad es, per i grafici). C'è infine da sottolineare che i metodi *nearest-neighbor based* presentano problemi nel caso in cui i dati normali non abbiano un numero sufficiente di vicini rispetto alle anomalie, o che gli outlier presentino comunque un certo numero di vicini. Inoltre, nel caso di dataset ad alta dimensionalità, il noto problema della *maledizione della dimensionalità* porta diversi inconvenienti a questi approcci: le differenze tra le distanze delle varie coppie punti si assottigliano, i dati diventano più sparsi, il concetto di neighborhood diventa poco significativo e quasi tutti i punti possono essere considerati come outlier.

Si hanno infine molti altri approcci di tipo non supervisionato. Le tecniche basate sulla *teoria dell'informazione* analizzano il contenuto informativo del dataset, utilizzando

delle misure come l'*entropia*, l'*entropia relativa* e la *Kolomogorov Complexity*. Tali metodi assumono che le anomalie introducano delle irregolarità nel contenuto informativo del dataset. Le tecniche basate sull'*analisi spettrale* sono specifiche per dataset ad alta dimensionalità ed eseguono una riduzione della dimensionalità, cercando un'approssimazione dei dati tramite una combinazione di attributi, che meglio catturano la variabilità delle istanze. L'idea è quella di determinare dei sottospazi in cui gli outlier siano più facili da identificare.

## 2.2 Definizione del problema

Gli approcci all'outlier detection problem, che sono oggetto di questo lavoro, rientrano nella categoria delle tecniche non supervisionate distance-based. Impostiamo il problema in maniera formale. Assumiamo che il dataset sia un sottoinsieme finito di un dato spazio metrico.

Dato un dataset $D$ di $d$ oggetti, un oggetto $p$ , una funzione di distanza $dist$ su $D \cup \{p\}$ ed un intero positivo $i$, l'$i$-*esimo nearest neighbor* $nn_i(p, D)$ di $p$ rispetto a $D$ è l'oggetto $q \in D$ tale che esistano esattamente $1 - i$ oggetti $r \in D$ (se $p \in D$, allora è necessario considerare anche $p$ stesso) tali che $dist(p, q) \geq dist(p, r)$. Di conseguenza, se $p \in D$ allora $nn_1(p, D) = p$, ovvero $nn_1(p, D)$ è l'oggetto in $D$ più vicino a $p$.

Come definizione di outlier e di anomaly score, utilizziamo quella fornita da *Angiulli et. al.* in [2]. Dato un dataset $D$ di $d$ oggetti, un oggetto $p \in D$, una funzione di distanza $dist$ su $D$ ed un intero positivo $k$ (con $1 \leq k \leq d$), il *peso* $w_k(p, D)$ di $p$ in $D$ è pari alla somma delle distanze di $p$ dai suoi $k$ *nearest neighbor* in $D$, ovvero:

$$w_k(p, D) = \sum_{i=1}^{k} dist(p, nn_i(p, D))$$

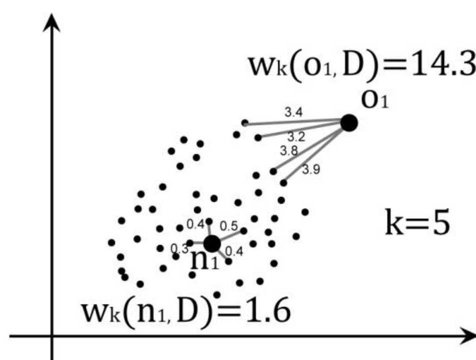



**Figura 2-4 Peso degli oggetti $n_1$ e $o_1$ nel dataset $D$, utilizzando la distanza euclidea**

Indichiamo con $D_{i,k}$ $(1 \leq i \leq d)$ l'oggetto in $D$ con l'$i$*-esimo* maggior peso in $D$, rispetto a $k$. Si ha quindi che $w_k(D_{1,k}, D) \geq w_k(D_{2,k}, D) \geq \cdots \geq w_k(D_{d,k}, D)$.

Dato un dataset $D$ di $d$ oggetti, un oggetto $p \in D$, una funzione di distanza $dist$ su $D$ e due interi positivi $n$ e $k$ (con $1 \leq k, n \leq d$), l'*Outlier Detection Problem* $ODP < D, dist, n, k >$ è definito nel modo seguente: trovare gli $n$ oggetti in $D$ di maggior peso, rispetto a $k$, ovvero l'insieme $D_{1,k}$, $D_{2,k}$, …, $D_{n,k}$. Tale insieme è detto *solution set* del problema o *insieme dei top* $n$ *outlier* in $D$. Un oggetto in $D$ il cui peso rispetto a $k$ sia maggiore o uguale di $w_k(D_{n,k}, D)$ è detto *outlier* rispetto a $D$ ($D$*-outlier*); viceversa è detto *inlier* rispetto a $D$ ($D$*-inlier*).

## 2.3 Algoritmo ODP Nested Loop

L'approccio più semplice per risolvere l'*ODP* è utilizzare un algoritmo *naif*, di tipo *nested loop*. L'idea di base è quella di calcolare, per ogni punto $p \in D$, tutte le distanze con gli altri punti del dataset e di quantificare il rispettivo peso $w_k(p, D)$ come somma delle minori $k$. Infine si determina il *solution set*, selezionando gli $n$ punti di peso maggiore.

Per la ricerca delle minori $k$ distanze, associamo ad ogni punto $p \in D$ una struttura dati $NN[p]$, i cui elementi siano le coppie $\langle q, \delta \rangle$, con $q \in D$ e $\delta = dist(p, q)$. $NN[p]$ mantiene le coppie aventi le $k$ minori distanze da $p$, ovvero i $k$ nearest neighbor di $p$. Per poter effettuare tale operazione di ricerca in maniera efficiente, strutturiamo $NN[p]$ come *heap binario* di tipo *max-heap*.

Gli *heap binari* (*binary heap*) sono delle strutture dati utilizzate comunemente in molti algoritmi di ordinamento o di selezione, in quanto permettono di trovare in maniera efficiente i minori (o maggiori) $k$ valori di un insieme. Formalmente si definisce come *heap binario* di tipo *max-heap* una struttura dati, organizzata ad albero binario, che soddisfa la seguente proprietà (*max-heap property*): ogni nodo $i$ diverso dalla radice è tale che $key[parent[i]] \geq key[i]$, con $parent[i]$ nodo padre di $i$ e $key[p]$ valore della *chiave* del nodo $p$ (ovvero un attributo del nodo $p$, utilizzato per l'ordinamento). Per un max-heap valgono due proprietà:

I. L'elemento con il massimo valore di chiave di un max-heap viene sempre memorizzato nella radice.
II. Un sottoalbero di un nodo $u$ di un max-heap contiene nodi il cui valore non è mai maggiore del valore di chiave del nodo $u$.

Similarmente, si definisce come *heap binario* di tipo *min-heap* una struttura dati, organizzata ad albero binario, che soddisfa la seguente proprietà (*min-heap property*): ogni nodo $i$ diverso dalla radice è tale che $key[parent[i]] \leq key[i]$, con $parent[i]$ nodo padre di $i$ e $key[p]$ valore della *chiave* del nodo $p$. Per un min-heap valgono due proprietà (reciproche delle precedenti):

I. L'elemento con il minimo valore di chiave di un min-heap viene sempre memorizzato nella radice
II. Un sottoalbero di un nodo $u$ di un min-heap contiene nodi il cui valore non è mai minore del valore di chiave del nodo $u$.

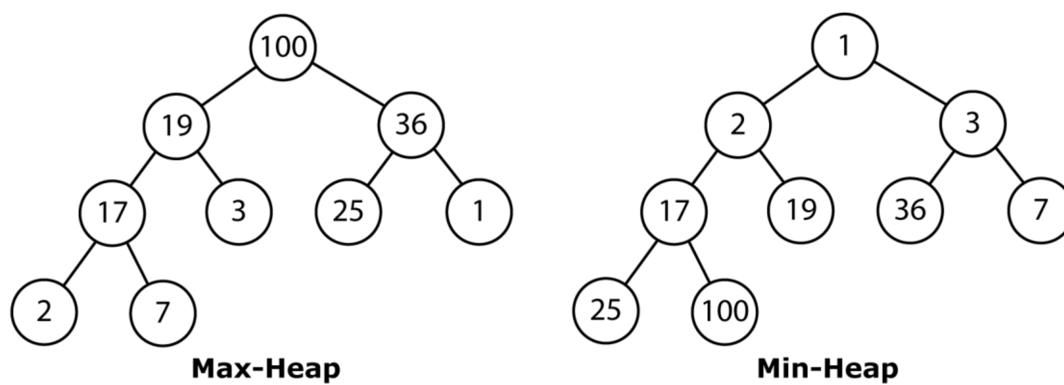



**Figura 2-5 Max-Heap e Min-Heap**

Gli heap binari sono tipicamente implementati tramite degli *array*, di lunghezza pari al numero massimo di elementi del heap. L'estrazione dell'elemento massimo (ovvero l'elemento con il massimo valore di chiave) da un max-heap, o del minimo da un min-heap, richiede un tempo pari $O(1)$, in quanto tale elemento è presente nella radice. L'operazione di inserimento di un nuovo elemento può essere effettuata in maniera efficiente con un algoritmo che richiede un tempo $O(\log(k))$, con $k$ pari al numero di elementi dell'heap. Un'altra importante operazione sugli heap è chiamata *heapify*. Tale operazione assume che i due nodi figli sinistro e destro, $left[root]$ e $right[root]$, del nodo radice $root$ siano dei max-heap (o min-heap), ma che $key[root]$ sia minore (o maggiore) del valore di chiave di almeno uno dei figli, violando così la proprietà di max-heap (o min-heap). L'azione *heapify* permette di correggere tale situazione, ripristinando la proprietà di max-heap (o min-heap). Anche tale operazione richiede un tempo pari a $O(\log(k))$.

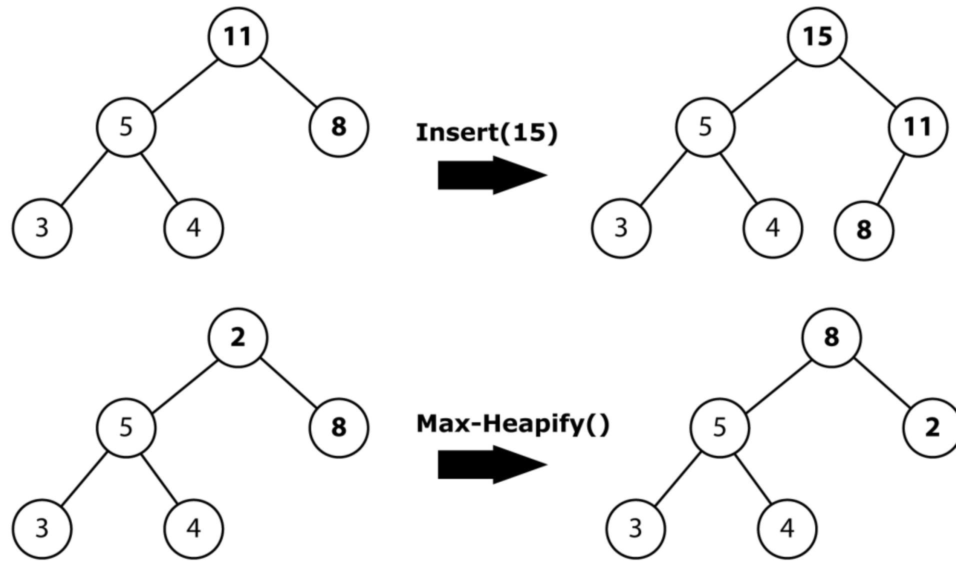



**Figura 2-6 Operazioni Insert e Max-Heapify su un Max-Heap**

Per poter trovare in maniera efficiente le minori $k$ distanze per ogni $p$, definiamo l'operazione $updateMin(NN[p], \langle q, \delta \rangle)$, che permette di inserire nel max-heap $NN[p]$ la nuova coppia $\langle q, \delta \rangle$, se $q$ risulta tra i correnti $k$ nearest neighbor di $p$. Se $|NN[p]| < k$, si effettua un semplice inserimento di $\langle q, \delta \rangle$ in $NN[p]$. Altrimenti sia $\langle s, \sigma \rangle$ la radice di $NN[p]$, con $\sigma = dist(p, s)$. Se $\delta < \sigma$, allora si sostituisce alla radice la nuova coppia $\langle q, \delta \rangle$ e si effettua una operazione di heapify su $NN[p]$. La funzione $updateMin$ richiede quindi, nel caso peggiore, un tempo pari a $O(\log(k))$, contro un tempo di $O(k)$ che si sarebbe avuto nel caso in cui $NN[p]$ fosse stato organizzato come un semplice array ordinato.

Inoltre, per la ricerca efficiente degli $n$ punti di maggior peso, manteniamo in memoria un min-heap $Top$, i cui elementi siano le coppie $\langle q, w_k(q, D) \rangle$. Definiamo inoltre l'operazione $updateMax(Top, \langle q, w_k(q, D) \rangle)$, che permette di inserire nel min-heap $Top$ la nuova coppia $\langle q, w_k(q, D) \rangle$ se $q$ risulta tra i correnti $n$ punti di maggior peso. Se $|Top| < n$, allora si effettua un semplice inserimento di $\langle q, w_k(q, D) \rangle$ in $Top$. Altrimenti sia $\langle s, w_k(s, D) \rangle$ la radice di $Top$. Se $w_k(q, D) > w_k(s, D)$, allora si sostituisce alla radice la nuova coppia $\langle q, w_k(q, D) \rangle$ e si effettua una operazione di heapify su $Top$. La funzione $updateMax$ richiede quindi, nel caso peggiore, un tempo pari a $O(\log(n))$.

Indichiamo con $Sum(NN[p]) = \sum_{\langle q, \delta \rangle \in NN[p]} \delta$, cioè la somma delle distanze dei correnti $k$ nearest neighbors di $p$ e quindi pari al valore corrente di $w_k(p, D)$. Lo pseudo-codice dell'algoritmo *ODPNestedLoop* è il seguente:

```
ODPNestedLoop(D, dist, n, k){
        Top := ∅
        for each p_i in {p_0, p_1, …, p_{d−1}} = D {
                for each p_j in {p_i, p_{i+1}, …, p_{d−1}} ⊆ D
                        if (i = j) {
                                updateMin(NN[p_i], ⟨p_i, 0⟩ )
                        } else {
                                δ := dist(p_i, p_j)
                                updateMin(NN[p_i], ⟨p_j, δ⟩ )
                                updateMin(NN[p_j], ⟨p_i, δ⟩ )
                        }
                }
                updateMax(Top, ⟨p_i, Sum(NN[p_i])⟩ )
        }
}
```

Consideriamo la matrice delle distanze $[\delta_{i,j}]$, con $\delta_{i,j} = dist(p_i, p_j)$, dove $p_i, p_j \in D$. Tale matrice ha dimensione $d \times d$ (con $d = |D|$) ed ogni elemento di riga $i$ e colonna $j$ rappresenta la distanza tra $p_i$ e $p_j$. Siccome $dist$, essendo una funzione di distanza, gode della proprietà simmetrica, allora si ha che $[\delta_{i,j}]$ è simmetrica. Di conseguenza, per ogni coppia $(p_i, p_j)$ con $i \neq j$, è possibile sfruttare questa proprietà, calcolando la distanza $dist(p_i, p_j)$ solo se $i > j$ ed effettuando la corrispondente operazione $updateMin$ su entrambi gli heap $NN[p_i]$ e $NN[p_j]$.

$$
\begin{array}{cc} & \begin{array}{ccccc} p_0 & p_1 & p_2 & \cdots & p_{d-1} \end{array} \\ [\delta_{i,j}] = \begin{array}{c} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{d-1} \end{array} & \begin{bmatrix} 0 & \alpha & \beta & \vdots & \gamma \\ \alpha & 0 & \delta & \vdots & \epsilon \\ \beta & \delta & 0 & \vdots & \vartheta \\ \cdots & \cdots & \cdots & \ddots & \omega \\ \gamma & \epsilon & \vartheta & \omega & 0 \end{bmatrix} \end{array}
$$

Sia $a$ la dimensionalità dei punti del dataset, ovvero il numero di coordinate (o di attributi) di essi. La complessità temporale dell'algoritmo ODPNestedLoop è pari a:

$$O\left(\frac{d^2}{2} \cdot (\mathrm{a} + 2\log(k))\right)$$

L'operazione di calcolo della distanza tra due punti $p_i, p_j \in D$, richiede un tempo proporzionale ad $O(a)$, mentre le operazioni di $updateMin$ e $updateMax$ sugli heap richiedono un tempo proporzionale a $O(\log(k))$ e $O(\log(n))$. Per la simmetria della matrice delle distanze, ogni calcolo di distanza viene eseguito $d^2/2$ volte, ogni operazione $updateMin$ $d^2/2 \cdot 2$ volte ed ogni $updateMax$ $d$ volte. La complessità dell'algoritmo è quindi pari a $O\left(\frac{d^2}{2} \cdot (\mathrm{a} + 2\log(k)) + d \cdot \log(n)\right)$. Siccome per $d$

sufficientemente grande $\frac{d^2}{2} \cdot (\mathrm{a} + 2\log(k)) \gg d \cdot \log(n)$, si può trascurare il secondo termine.

La complessità temporale di questo algoritmo è molto elevata, essendo proporzionale a $O(d^2)$. Inoltre tale algoritmo non sfrutta alcun meccanismo di pruning per cercare di limitare il numero di distanze calcolate e risulta quindi essere molto inefficiente e non ottimizzato per l'ODP.

## 2.4 Algoritmo ODP Solving Set

Un diverso approccio al problema *ODP*, molto più efficiente e sofisticato, è quello proposto da *Angiulli, Basta e Pizzuti* in [3], che si basa sul concetto di *Outlier Detection Solving Set* (o detto più brevemente *solving set*). Il *solving set* $S$ è un sottoinsieme del dataset $D$, che include un numero sufficiente di punti appartenenti a $D$, tali da permettere di considerare solamente le distanze tra le coppie in $S \times D$, per ottenere i top $n$ outlier. L'algoritmo proposto permette di determinare il solving set $S$ e risolvere l'ODP, evitando il calcolo di tutte le distanze, per ogni coppia di punti in $D$.

Formalizziamo il concetto di solving set. Dato un dataset $D$ di $d$ oggetti, una funzione di distanza $dist$ su $D$ e due interi positivi $n$ e $k$ (con $1 \leq k, n \leq d$), un *Outlier Detection Solving Set* per l'$ODP < D, dist, n, k >$ è un sottoinsieme $S \subseteq D$ tale che:

I. $|S| \geq \max\{n, k\}$
II. sia $lb(S)$ l'$n$-esimo elemento di $\{w_k(p, D) | p \in S\}$. Allora, per ogni $q \in (D - S)$, $w_k(q, S) < lb(S)$

Intuitivamente, un solving set $S$ è un sottoinsieme di $D$ tale che le distanze $\{dist(p, q) | p \in S, q \in D\}$, sono sufficienti a garantire che $S$ contenga il solution set dell'ODP. Sia $n^* \geq n$ l'intero positivo tale che $w_k(D_{n,k}, D) = w_k(D_{n^*,k}, D)$ e $w_k(D_{n,k}, D) > w_k(D_{n^*+1,k}, D)$. Indichiamo con *extended solution set* dell' l'$ODP < D, dist, n, k >$ l'insieme $\{D_{1,k}, \dots, D_{n^*,k}\}$. Proviamo la seguente proposizione:

"se $S$ è un solving set per l'$ODP < D, dist, n, k >$, allora $\{D_{1,k}, \dots, D_{n^*,k}\} \subseteq S$ e $lb(S) = w_k(D_{n,k}, D)$".

Assumiamo per assurdo che esista un intero $i$ $(1 \leq i \leq n^*)$, tale che $D_{i,k} \notin S$. Allora $D_{i,k} \in (D - S)$ e $w_k(D_{i,k}, D) \geq lb(S)$. Quindi $S$ non è un solving set, il che contraddice l'ipotesi e la tesi è provata.

Dato un dataset $D$ e fissati due parametri $n$ e $k$, in generale si possono avere più possibili solving set per il corrispondente ODP. Chiaramente anche $D$ stesso è un solving set e tipicamente il solo *extended solution set* può non essere sufficiente per formare un solving set.

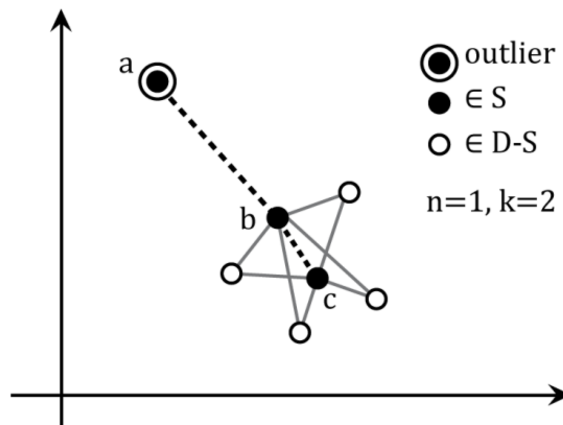



**Figura 2-7 Esempio di solving set, per $n$=1 e $k$=2**

Per poter risolvere l'Outlier Detection Problem e individuare un solving set per esso, gli autori presentano un algoritmo denominato *ODPSolvingSet*. L'algoritmo calcola il peso degli oggetti del dataset $D$ in più iterazioni, confrontando, ad ogni iterazione, ogni oggetto con un sottoinsieme molto ridotto del dataset, denominato *insieme dei candidati* e indicato con $C$. Per ogni oggetto in $D$ vengono memorizzati i $k$ nearest neighbors visitati fino all'iterazione corrente, rispetto a $C$. Di conseguenza, il peso corrente di un oggetto rappresenta un *upper bound* del suo peso reale. Gli oggetti, il cui peso corrente risulta essere inferiore all'$n$*-esimo* maggior peso finora calcolato, vengono indicati come *non attivi*, mentre i restanti vengono indicati come *attivi*. Alla fine di ogni iterazione vengono selezionati, tra i soli oggetti in $D$ dichiarati attivi, quelli aventi il massimo peso corrente ed utilizzati per formare il nuovo insieme di candidati $C$ per l'iterazione successiva. L'algoritmo si ferma quando non si hanno più oggetti dichiarati come attivi. Il solving set è dato dall'unione dei vari insiemi $C$ utilizzati ad ogni iterazione.

```
ODPSolvingSet(D, dist, n, k, m){
        S := ∅
        Top := ∅
        C := RandomSelect(D,m)

        while (C ≠ ∅) {
                S := S ∪ C
                D := D − C
                for each q in C {LNNC[q] := NN[q]}
                for each q_i in {q_0, q_1, ..., q_|C|−1} = C {
                        for each q_j in {q_i, q_i+1, ..., q_|C|−1} ⊆ C {
                                if (i = j) {
                                        updateMin(LNNC[q_i], ⟨q_i, 0⟩ )
                                } else {
                                        δ := dist(q_i, q_j)
                                        updateMin(LNNC[q_i], ⟨q_j, δ⟩ )
                                        updateMin(LNNC[q_j], ⟨q_i, δ⟩ )
                                }
                        }
                }
                LC := ∅
                for each p in D {
                        for each q in C {
                                if (max{Sum(NN[p]), Sum(LNNC[q])} ≥ Min(Top)) {
                                        δ := dist(p, q)
                                        updateMin(NN[p], ⟨q, δ⟩ )
                                        updateMin(LNNC[q], ⟨p, δ⟩ )
                                }
                        }
                        if (p.active = true AND Sum(NN[p]) ≥ Min(Top)) {
                                updateMax(LC, ⟨p, Sum(NN[p])⟩ )
                        }else {
                                p.active := false
                        }
                }

                for each q in C {
                        updateMax(Top, ⟨q, Sum(LNNC[q])⟩ )
                }
                C := candSelect(LC)
        }
}
```

L'algoritmo *ODPSolvingSet* prende in input un dataset $D$ contenente $d$ oggetti, una funzione di distanza $dist$ su $D$, un parametro $k$ che rappresenta il numero di neighbors da considerare per il calcolo del peso degli oggetti, un parametro $n$ che rappresenta il numero di top outlier da trovare e un parametro $m$ che rappresenta il numero di oggetti da selezionare, ad ogni iterazione, per determinare il nuovo insieme di candidati $C$. In maniera analoga all'algoritmo *ODPNestedLoop*, associamo ad ogni punto $p \in D$ un *max-heap* $NN[p]$, i cui elementi siano le coppie $\langle x, \delta \rangle$, con $x \in S$ e $\delta = dist(p, x)$. $NN[p]$ mantiene i correnti $k$ nearest neighbors di $p$, rispetto al corrente solving set $S$. Ricordiamo che $S$ è formato dall'unione dei vari insiemi $C$, individuati ad ogni iterazione. Similarmente, associamo ad ogni punto $q \in C$ un *max-heap* $LNNC[q]$, con elementi le coppie $\langle y, \delta \rangle$, con $y \in D$ e $\delta = dist(q, y)$, contenente i correnti $k$ nearest

neighbors di $q$ rispetto a $D$. Indichiamo con $Sum(NN[p]) = \sum_{\langle x,\delta\rangle \in NN[p]} \delta$, cioè la somma delle distanze dei correnti $k$ nearest neighbors di $p$ e con $Sum(LNNC[q]) = \sum_{\langle y,\delta\rangle \in LNNC[q]} \delta$. Inoltre manteniamo in memoria due ulteriori *min-heap* indicati come $Top$ e $LC$, rispettivamente di $n$ ed $m$ elementi. Gli heap memorizzano coppie di tipo $\langle q, Sum(LNNC[q])\rangle$ e $\langle p, Sum(NN[p])\rangle$, con $q \in S, p \in D$. $Top$ mantiene i correnti $n$ oggetti $q \in S$ con i maggiori pesi finora trovati, $LC$ i correnti $m$ oggetti $p \in D$ dichiarati come attivi e dotati dei maggiori *weight upper bound* (cioè $Sum(NN[p])$). Definiamo per gli heap $NN[p]$ e $LNNC[q]$ l'operazione $updateMin$ e per gli heap $Top$ e $LC$ l'operazione $updateMax$, come per l'algoritmo *ODPNestedLoop*.

Indichiamo con $Min(Top)$ il valore corrente del minor peso delle coppie in $Top$. Tale valore rappresenta il lower bound del peso dell'$n$*-esimo* outlier in $D$. Questo implica che gli oggetti $p \in D$ con $Sum(NN[p]) < Min(Top)$ non possono sicuramente far parte del solution set e per questo vengono etichettati dall'algoritmo come *non attivi*.

L'insieme $C$ viene inizialmente formato da $m$ elementi del dataset selezionati in maniera casuale, tramite la funzione $RandomSelect(D, m)$. Ad ogni iterazione, gli oggetti in $C$ vengono rimossi da $D$ ed inseriti in $S$. Ogni oggetto $q \in C$ è confrontato con tutti gli oggetti in $D \cup C$ e viene calcolato con esattezza il peso di $q$. Per effettuare i confronti in maniera efficiente, questa operazione è spezzata in due step. Nel primo step si confronta ogni oggetto $q_i \in C$ con tutti gli altri oggetti $q_j \in C$, calcolando per ogni coppia di punti la rispettiva distanza $dist(q_i, q_j)$ e chiamando la funzione $updateMin$ sugli rispettivi heap $LNNC[q_i]$ e $LNNC[q_j]$. Essendo la matrice delle distanze $[\delta_{i,j}]$, con $\delta_{i,j} = dist(q_i, q_j)$, simmetrica per ogni coppia $(q_i, q_j)$ con $i \neq j$, è possibile sfruttare questa proprietà, calcolando la distanza $dist(q_i, q_j)$ solo se $i > j$ ed effettuare la corrispondente operazione $updateMin$ su entrambi gli heap. Nel secondo step si confronta ogni oggetto $p \in D$ con gli oggetti $q \in C$. Per ogni coppia di oggetti, il calcolo della distanza e le rispettive operazioni $updateMin$ sugli heap sono effettuate solo se almeno uno dei due oggetti può essere considerato un outlier, ovvero se $\max\{Sum(NN[p]), Sum(LNNC[q])\} \geq Min(Top)$. Inoltre, se $Sum(NN[p]) \geq Min(Top)$, il punto $p$ viene dichiarato come attivo e viene effettuata l'operazione $updateMax$ sull'heap $LC$, altrimenti viene dichiarato come non attivo. Alla fine dell'iterazione, ogni candidato $q \in C$ ha il suo weight upper bound equivalente al proprio peso e viene effettuata quindi l'operazione di $updateMax$ della coppia $\langle q, Sum(LNNC[q])\rangle$ su $Top$, $\forall\, q \in C$ . Infine vengono selezionati i nuovi candidati

per la successiva iterazione, prendendo gli $m$ oggetti nell'heap $LC$, tramite la funzione $candSelect$. Se non ci sono più punti dichiarati come attivi, e di conseguenza $LC$ è vuoto, allora $candSelect$ restituisce un insieme vuoto e l'algoritmo si ferma. $Top$ contiene il solution set ed $S$ è un solving set per l'ODP.

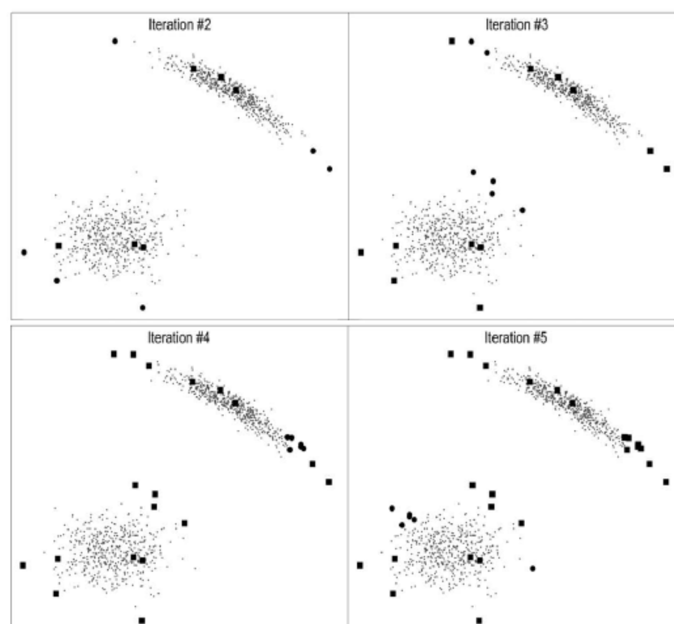



**Figura 2-8 Esempio di esecuzione di ODPSolvingSet. I quadrati neri costituiscono i punti del solving set $S$, i cerchi neri i punti candidati in $C$ dell'iterazione corrente**

Consideriamo ora la complessità temporale. Sia $a$ la dimensionalità dei punti del dataset, ovvero il numero di coordinate (o di attributi) di essi. Nel caso peggiore, la complessità temporale dell'algoritmo ODP SolvingSet è pari a:

$$O\left(d^2 \cdot (\mathrm{a} + 2\log(k))\right)$$

L'operazione di calcolo della distanza tra due punti richiede un tempo proporzionale ad $O(a)$, mentre le operazioni di $updateMin$ su $NN[p]$ e $LNNC[q]$ e $updateMax$ su $LC$ richiedono un tempo proporzionale a $O(\log(k))$ e $O(\log(m))$. Ad ogni iterazione, la fase più onerosa è quella del confronto $D$ con $C$, dove avvengono $m \cdot D$ calcoli di distanze, $2 \cdot m \cdot D$ operazioni $updateMin$ sugli heap $NN[p]$ e $LNNC[q]$ e $d$ operazioni $updateMax$ sulla heap $LC$. Ogni iterazione ha quindi complessità pari a $O(d \cdot m \cdot (\mathrm{a} + 2\log(k)) + d \cdot \log(m)) \cong O\left(d \cdot m \cdot (\mathrm{a} + 2\log(k))\right)$, per $d$ sufficientemente elevato. Nel caso peggiore vengono inseriti nel solving set tutti i punti del dataset, di conseguenza si hanno $\lceil d/m \rceil$ iterazioni. La complessità dell'algoritmo è quindi $O\left(\left\lceil \frac{d}{m} \right\rceil \cdot d \cdot m \cdot (\mathrm{a} + 2\log(k))\right) \cong O\left(d^2 \cdot (\mathrm{a} + 2\log(k))\right)$.

La complessità temporale, nel caso peggiore, è proporzionale a $O(d^2)$ e paragonabile a quella dell'algoritmo *ODPNestedLoop*. Nella pratica però, la cardinalità del solving set trovato è tipicamente di vari ordini di grandezza inferiore alla cardinalità del dataset. Di conseguenza, l'algoritmo ha una complessità pratica pari a $O(d^{1+\beta})$, con $\beta < 1$. Sia $S$ il solving set fornito in output dall'algoritmo. L'algoritmo esegue $d^{1+\beta} = d \cdot |S|$ calcoli di distanze (e rispettive operazioni sulle heap), di conseguenza $\beta = \frac{\log(|S|)}{\log(d)}$ (con $\frac{\log(|S|)}{\log(d)} < 1$ in quanto $|S| \leq d$). Se $\beta$ è sufficientemente piccolo $(1 + \beta \cong 1)$, l'algoritmo si comporta in modo quasi lineare, rispetto a $d$.

Il concetto di solving set, oltre a fornire un metodo efficiente per la risoluzione dell'ODP, può essere visto come un ottimo modello per la predizione degli outlier. L'*Outlier Prediction Problem* (*OPP*) consiste nel determinare se un nuovo oggetto non appartenente a $D$ possa essere un outlier, ovvero se il suo peso, rispetto a $D$ e $k$, sia maggiore o uguale del peso dell'$n$*-esimo* top outlier trovato in $D$. Formalmente, dato un dataset di oggetti $U$, un sottoinsieme $D \subseteq U$ di $d$ oggetti, un oggetto $q \in U$ detto *query object*, una funzione di distanza $dist$ su $U$ e due interi positivi $n$ e $k$ (con $1 \leq k, n \leq d$), l'*Outlier Prediction Problem* $OPP < D, q, dist, n, k >$ è definito nel modo seguente: $w_k(q, D) \geq w_k(D_{n,k}, D)$ ?

Il problema *OPP* può essere risolto in un tempo $O(d)$, confrontando il query object $q$ con tutti gli altri punti in $D$, ipotizzando che siano stati preventivamente trovati gli $n$-top outlier in $D$. Nelle applicazioni reali si presentano tipicamente dei dataset contenenti un numero molto elevato di istanze ed inoltre si possono avere dei requisiti che obbligano a risolvere l'OPP in tempi molto ristretti. In tali situazioni un tempo $O(d)$ può essere non accettabile. E' possibile utilizzare il solving set come un modello approssimato dell'intero dataset per l'OPP e classificare un query object $q$ come outlier o meno, approssimando il suo peso rispetto a $D$ con quello rispetto ad $S$. Questa operazione può essere svolta in un tempo $O(|S|)$, dove tipicamente $|S| \ll d$. Diciamo che $q$ è un *S-outlier* se $w_k(q, S) \geq lb(S)$, un *S-inlier* nel caso contrario. In questo approccio non è possibile avere falsi negativi (ovvero query object *D-outlier*, erroneamente classificati come normali da $S$), in quanto per ogni query object $q$ si ha che $w_k(q, S) \geq w_k(q, D)$, con $S \subseteq D$. Viceversa, il solving set può classificare erroneamente un query object come outlier, anche se in realtà non lo è rispetto a $D$ (falsi positivi). Gli autori in [3] forniscono inoltre la definizione di *robust solving set* ed un algoritmo per calcolarlo, che permette di eliminare anche i possibili falsi positivi, classificando sempre correttamente il query object. Inoltre il solving set può

essere visto come una rappresentazione compressa del dataset e può quindi essere utilizzato anche per altri scopi più generali, ad esempio a supporto dell'estrazione di feature vettoriali per i processi di classificazione.

## 2.5 Risultati sperimentali

Per valutare le performance dell'algoritmo ODPSolvingSet e confrontarlo con l'approccio *naif* seguito da ODPNestedLoop, sono stati eseguiti una serie di test su diverse tipologie di dataset. Entrambi gli algoritmi sono stati implementati in *Java*, per poter garantire un buon livello di generalità. Come piattaforma di testing si è utilizzato un singolo nodo del supercomputer *IBM PLX* [10], un cluster di 274 nodi presente presso il centro di calcolo del *CINECA*. Ogni nodo dispone di due *six-cores Intel Westmere* a 2.40 GHz (*E5645*), di 48 Gb di RAM *DDR3* e di due GPU *NVIDIA Tesla M2070* (non utilizzate per queste versioni dei due algoritmi). I vari nodi sono interconnessi tramite una rete *InfiniBand*, dotata di switch di tipo *4x QDR*, per una banda di circa 40 Gb/s. Nel novembre 2011 l'IBM PLX è stato classificato all'82-esima posizione nella classifica dei 500 più potenti supercomputer al mondo.

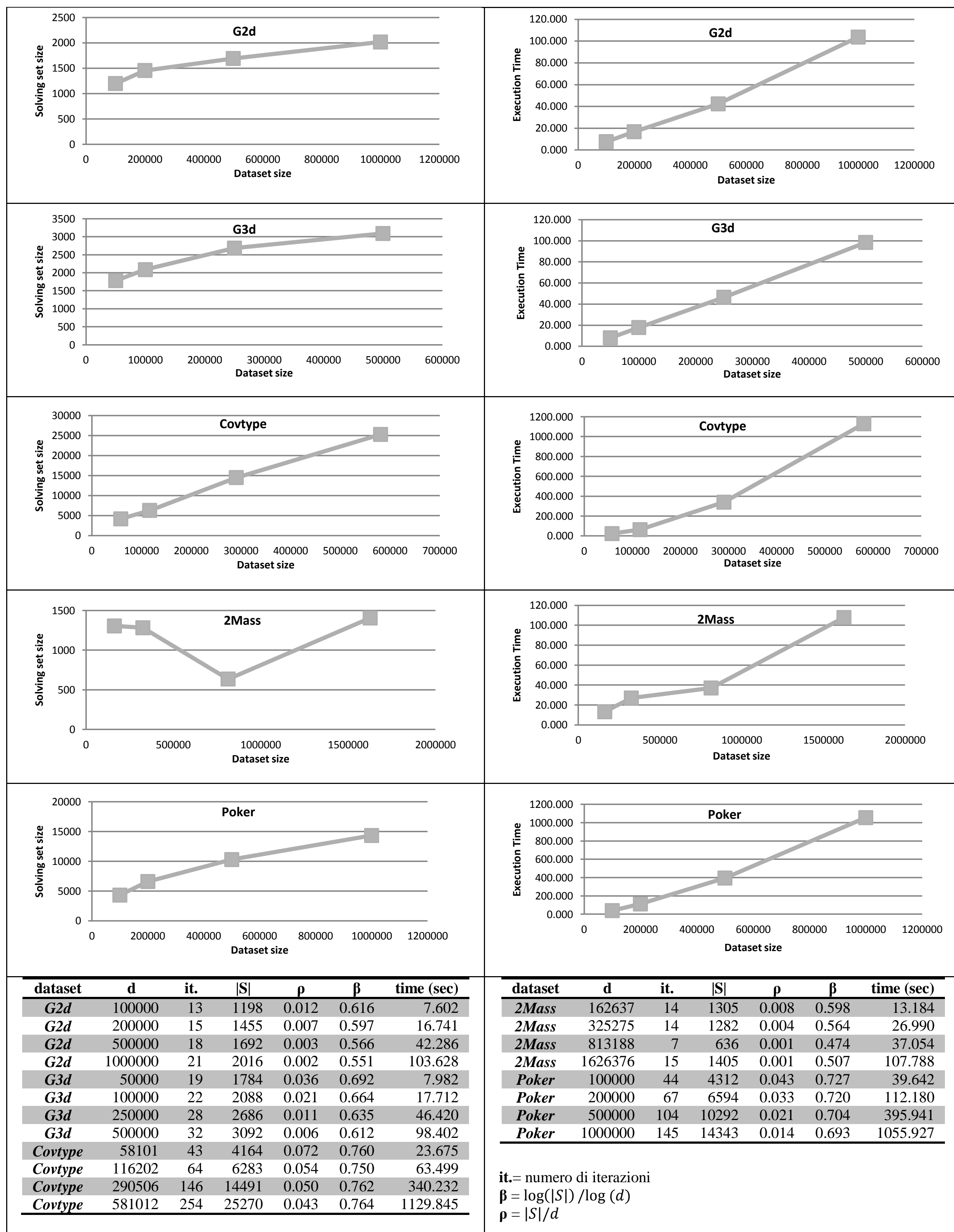
**Figura 2-9 IBM PLX**

### 2.5.1 Cardinalità del solving set

La prima batteria di test riguarda in maniera esclusiva l'algoritmo ODPSolvingSet e permette di osservare come variazioni della cardinalità del dataset in input si riflettano sulla cardinalità del solving set e sui tempi di esecuzione. Per questi esperimenti sono stati utilizzati cinque diversi dataset:

- *G2d*: un dataset artificiale contenente 1,000,000 di vettori bi-dimensionali, generati da una distribuzione normale *2-d*, ognuna avente l'origine come vettore delle medie e la matrice unitaria come matrice di covarianza
- *G3d*: un dataset artificiale contenente 500,000 vettori tri-dimensionali, ottenuto dall'unione di oggetti di tre differenti distribuzioni normali *3-d*, ognuna avente un differente vettore delle medie e la matrice unitaria come matrice di covarianza
- *Covtype*: un dataset reale di 581,012 istanze di 10 attributi, ottenuto selezionando gli attributi quantitativi del dataset *Covertype*, disponibile presso l'*UCI Machine Learning Repository* [11]
- *Poker*: un dataset reale di 1,000,000 istanze di 10 attributi, ottenuto rimuovendo la *class label* dal dataset *Poker-Hands* disponibile presso l'*UCI Machine Learning Repository*
- *2Mass*: un dataset reale di 1,623,376 istanze di 3 attributi, contenente dati ottenuti dal database *2MASS Survey Atlas Image Info* del *2MASS Survey Scan Working Databases catalog*, disponibile presso il *NASA/IPAC Infrared Science Archive* (*IRSA*) [12]. Ogni oggetto è formato da tre attributi quantitativi, associati a dei filtri *JHK*.

Per ogni esperimento sono stati utilizzati in input all'algoritmo un insieme di $d$ oggetti distinti, selezionati in maniera casuale da ogni dataset di riferimento, facendo variare di $d$ dal 100% al 10% della cardinalità del dataset originale. Come parametri dell'algoritmo sono stati posti $n = 10$, $m = 100$ e $k = 50$.




| dataset | d | it. | \|S\| | ρ | β | time (sec) |
|---|---|---|---|---|---|---|
| *G2d* | 100000 | 13 | 1198 | 0.012 | 0.616 | 7.602 |
| *G2d* | 200000 | 15 | 1455 | 0.007 | 0.597 | 16.741 |
| *G2d* | 500000 | 18 | 1692 | 0.003 | 0.566 | 42.286 |
| *G2d* | 1000000 | 21 | 2016 | 0.002 | 0.551 | 103.628 |
| *G3d* | 50000 | 19 | 1784 | 0.036 | 0.692 | 7.982 |
| *G3d* | 100000 | 22 | 2088 | 0.021 | 0.664 | 17.712 |
| *G3d* | 250000 | 28 | 2686 | 0.011 | 0.635 | 46.420 |
| *G3d* | 500000 | 32 | 3092 | 0.006 | 0.612 | 98.402 |
| *Covtype* | 58101 | 43 | 4164 | 0.072 | 0.760 | 23.675 |
| *Covtype* | 116202 | 64 | 6283 | 0.054 | 0.750 | 63.499 |
| *Covtype* | 290506 | 146 | 14491 | 0.050 | 0.762 | 340.232 |
| *Covtype* | 581012 | 254 | 25270 | 0.043 | 0.764 | 1129.845 |

| dataset | d | it. | \|S\| | ρ | β | time (sec) |
|---|---|---|---|---|---|---|
| *2Mass* | 162637 | 14 | 1305 | 0.008 | 0.598 | 13.184 |
| *2Mass* | 325275 | 14 | 1282 | 0.004 | 0.564 | 26.990 |
| *2Mass* | 813188 | 7 | 636 | 0.001 | 0.474 | 37.054 |
| *2Mass* | 1626376 | 15 | 1405 | 0.001 | 0.507 | 107.788 |
| *Poker* | 100000 | 44 | 4312 | 0.043 | 0.727 | 39.642 |
| *Poker* | 200000 | 67 | 6594 | 0.033 | 0.720 | 112.180 |
| *Poker* | 500000 | 104 | 10292 | 0.021 | 0.704 | 395.941 |
| *Poker* | 1000000 | 145 | 14343 | 0.014 | 0.693 | 1055.927 |

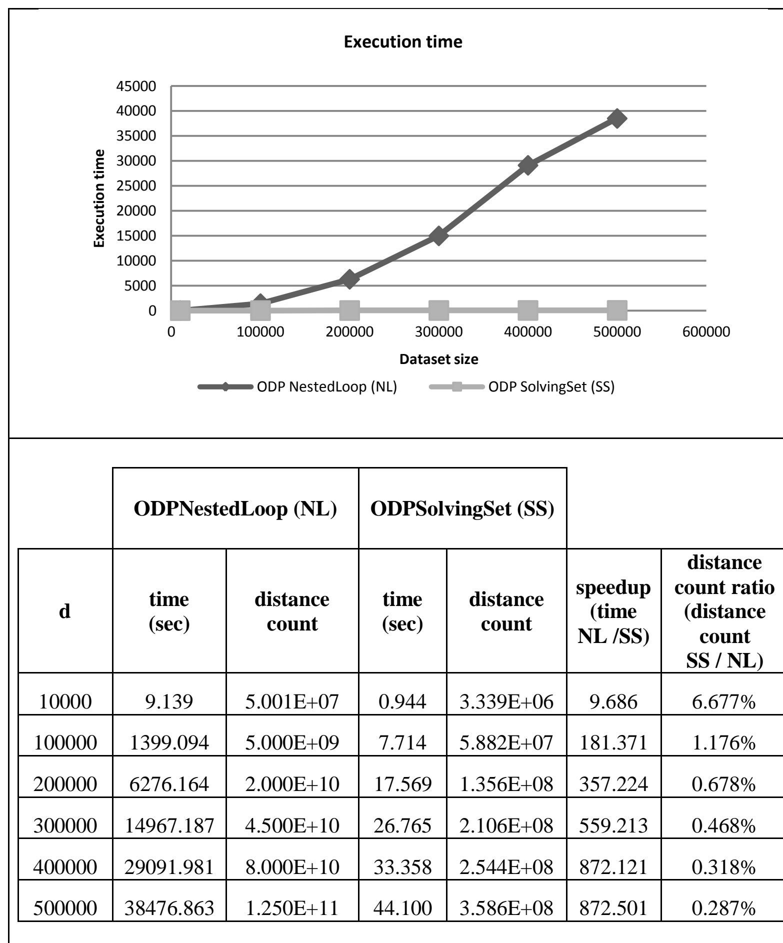
**it.**= numero di iterazioni
**β** = $\log(|S|)/\log(d)$
**ρ** = $|S|/d$

**Figura 2-10 Risultati sperimentali su ODP SolvingSet variando la cardinalità del dataset ($n$=10, $m$=100, $k$=50)**

I risultati mostrano chiaramente che la cardinalità del solving set $S$ individuato dall'algoritmo risulta essere, in tutti gli esperimenti, di vari ordini di grandezza inferiore alla cardinalità del dataset. Inoltre il rapporto $\rho = \frac{|S|}{d}$ tende fortemente a decrescere con l'aumentare di $d$, e di conseguenza anche il valore $\beta$ tende a calare. Osservando i tempi di esecuzione ci si accorge che questi tendono a crescere in modo nettamente sub-quadratico e non in modo quadratico, al variare di $d$, seguendo il prodotto $d \cdot |S|$. Questo conferma la complessità pratica pari a $O(d^{1+\beta})$, dove per $\beta$ piccolo si ha un comportamento quasi lineare rispetto a $d$.

### 2.5.2 Confronto ODP Solving Set – ODP Nested Loop

La seconda batteria di test ha l'obiettivo di confrontare le prestazioni dell'algoritmo ODPSolvingSet con quelle dell'algoritmo ODPNestedLoop. Per questi esperimenti sono stati utilizzati cinque dataset artificiali, contenenti un numero variabile (da 100,000 a 500,000) di vettori bi-dimensionali, generati da una distribuzione normale 2-d, con valor medio 100 e deviazione standard 50. Anche per questi test i parametri usati sono stati $n = 10$ e $k = 50$, con $m = 100$ per ODP SolvingSet.




| | ODPNestedLoop (NL) | | ODPSolvingSet (SS) | | | |
|---|---|---|---|---|---|---|
| **d** | **time (sec)** | **distance count** | **time (sec)** | **distance count** | **speedup (time NL /SS)** | **distance count ratio (distance count SS / NL)** |
| 10000 | 9.139 | 5.001E+07 | 0.944 | 3.339E+06 | 9.686 | 6.677% |
| 100000 | 1399.094 | 5.000E+09 | 7.714 | 5.882E+07 | 181.371 | 1.176% |
| 200000 | 6276.164 | 2.000E+10 | 17.569 | 1.356E+08 | 357.224 | 0.678% |
| 300000 | 14967.187 | 4.500E+10 | 26.765 | 2.106E+08 | 559.213 | 0.468% |
| 400000 | 29091.981 | 8.000E+10 | 33.358 | 2.544E+08 | 872.121 | 0.318% |
| 500000 | 38476.863 | 1.250E+11 | 44.100 | 3.586E+08 | 872.501 | 0.287% |

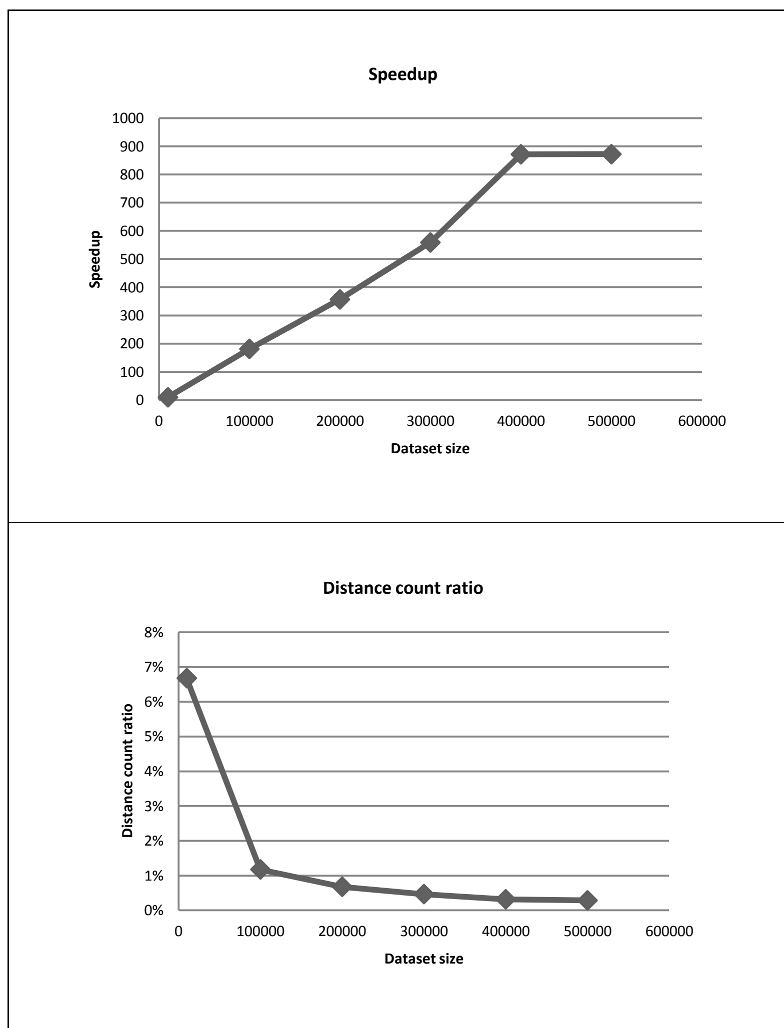
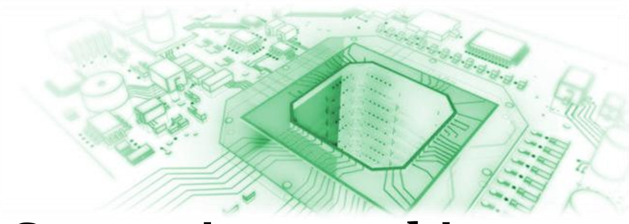



**Figura 2-11 Risultati sperimentali su ODPSolvingSet e ODPNestedLoop, variando la cardinalità del dataset ($n$=10, $m$=100, $k$=50)**

I tempi di esecuzione dell'algoritmo ODPSolvingSet risultano essere estremamente inferiori rispetto a quelli dell'algoritmo ODPNestedLoop, per $d$ sufficientemente grande. Se osserviamo lo *speedup* di ODPSolvingSet su ODPNestedLoop, definito come il rapporto tra i tempi di esecuzione di ODPNestedLoop e di ODPSolvingSet, tale numero arriva ad assumere un valore perfino superiore ad 800. Consideriamo ora il rapporto tra il numero di distanze calcolate da ODPSolvingSet e da quelle calcolate da ODPNestedLoop (*distance count ratio*). All'aumentare della cardinalità del dataset, tale rapporto tende ad un valore molto piccolo, prossimo a zero. Questi risultati dimostrano l'assoluta superiorità dell'algoritmo ODPSolvingSet, per $d$ sufficientemente grande.



# Capitolo 3: GPU Computing e architettura NVIDIA CUDA

Con il passare degli anni e guidate dalla forte domanda di mercato per la grafica *3D*, le *GPU* (*Graphic Processor Unit*) si sono evolute in processori programmabili fortemente paralleli, multithreaded e multicore, dotati di un'impressionante potenza di calcolo e di un'elevata banda nei trasferimenti di memoria. In particolare le GPU si sono specializzate nel calcolo parallelo intensivo, dedicando la maggior parte del silicio al *data processing*, piuttosto che a capienti strutture di caching o a sistemi complessi di gestione del flusso delle istruzioni. Le GPU sono adatte a risolvere problemi che possono essere espressi come operazioni di calcolo parallele, in cui lo stesso programma possa essere eseguito su più dati in parallelo, con un alta intensità delle operazioni aritmetiche, rispetto a trasferimenti di memoria. La rapida crescita della potenzialità delle GPU e l'aumento della possibilità di programmabilità hanno spinto molte comunità di ricerca a cercare di mappare diverse tipologie di problemi, richiedenti una forte carico computazionale, su tali processori. Questo fenomeno, che prende il nome di *general purpouse computing on the GPU* (*GPGPU*), o più semplicemente *GPU Computing*, ha posizionato le GPU nel mercato come una valida alternativa ai tradizionali microprocessori (*CPU*) nei sistemi *HPC* (*High Performance Computing*).

Le GPU sono particolarmente adatte per risolvere una particolare classe di problemi che presenti le seguenti caratteristiche [13]:

- *forti requisiti computazionali*: il rendering *3D* richiede di elaborare bilioni di pixel al secondo, e per ogni pixel sono necessitare migliaia di operazioni. Le GPU devono fornire delle alte performance di calcolo, per rispettare i requisiti real-time delle applicazioni.
- *parallelismo*: le operazioni sui singoli pixel possono essere effettuate in maniera indipendente ed in parallelo. La *graphic pipeline* delle GPU è basata su questo principio ed è possibile mappare su di essa problemi di differenti domini, ma comunque dotati di questo requisito di alto parallelismo.
- *lo throughput è più importante della latenza*: la percezione umana delle immagini opera su tempi dell'ordine dei millisecondi, mentre i processori moderni lavorano su tempi dell'ordine dei nanosecondi. La *graphic pipeline* è

ottimizzata per sfruttare al meglio questo gap: è dotata di molti stadi (migliaia contro le decine della pipeline delle CPU) ed incentrata sulla potenza di calcolo, con l'assenza di esecuzione speculative o gestioni complesse del flusso di istruzioni.

Le CPU sono ottimizzate per l'esecuzione di applicazioni formate da un numero molto limitato di thread, che esibiscano un alto grado di località, un mix di differenti tipi di operazioni e un'alta percentuale di branch. Le GPU, viceversa, sono adatte ad applicazioni formate da un alto numero di thread, dominate da un elevato numero di operazioni di calcolo sequenziali [14].

### 3.1 Evoluzione dell'architettura delle GPU

Fino all'anno 2006, le GPU erano dei processori non programmabili (*fixed-function processor*) basati sulla *graphics pipeline*, per il rendering della grafica tri-dimensionale [13]. In input alla *graphics pipeline* si hanno una serie di primitive geometriche, in un sistema di coordinate 3-D. Attraverso diversi step, alle primitive si applica l'ombreggiatura (*shading*) e gli elementi vengono assemblati per creare l'immagine finale. Si possono individuare cinque step principali della *graphics pipeline*:

- *Vertex Operations*: le primitive in input sono formate da un insieme di vertici. In questa fase ogni vertice viene mappato nello *screen space* e gli viene applicata l'ombreggiatura. Essendo le scene tipicamente formate da centinaia di migliaia di vertici e siccome le operazioni per ogni vertice possono essere eseguite in maniera indipendente, questa fase è particolarmente adatta ad un calcolo parallelo.
- *Primitive Assembly*: i vertici vengono assemblati in triangoli, la primitiva fondamentale per le GPU.
- *Rasterization*: per ogni triangolo si identifica quali pixel nello screen space sono coperti da esso, generando una primitiva che prende il nome di *fragment.*
- *Fragment Operations*: utilizzando le informazioni di colore dei vertici e accedendo a dati addizionali da una memoria globale in forma di *texture* (immagini che vengono mappate sulle superfici), ogni fragment viene ombreggiato per determinarne il colore finale. Questo stadio è tipicamente quello che richiede il maggior numero di calcoli.

- *Composition:* i fragment vengono assemblati nell'immagine finale, assegnando ad ogni pixel il proprio colore, ovvero il colore del fragment più vicino alla visuale della scena, a cui il pixel appartiene.

Nelle prime GPU, le azioni eseguibili negli stadi di *Vertex Operations* e *Fragment Operations* erano configurabili, ma non programmabili. Ad esempio, durante i calcoli sui vertici, una delle operazioni chiave è il calcolo del colore di ogni vertice, in funzione di alcune proprietà del vertice stesso e delle luci della scena. Il programmatore può controllare la posizione ed il colore dei vertici e delle luci, ma non come vengono determinate le ombreggiature, che dipendono solamente dall'interazione dei vertici con le luci. Il problema principale di questa *fixed-function pipeline* era la mancanza di poter esprimere delle operazioni di shading più sofisticate, per poter generare degli effetti più complessi (come esplosioni, fumo, riflessi dell'acqua, ecc).

Il passo successivo è stato quello di sostituire le fixed-function per le operazioni sui vertici e fragment, con dei programmi *user-specific*, eseguiti su ogni vertice o fragment. Nel 2001 la *NVIDIA* ha introdotto sul mercato consumer la *GeForce 3*, la prima GPU in grado di permettere la programmabilità delle funzioni di shading dei vertici e dei fragment [14]. All'inizio la programmabilità di questo chip era abbastanza limitata, ma con l'avanzamento tecnologico i programmi di shading poterono diventare sempre più complessi, grazie al supporto hardware di maggiori risorse di calcolo e nuovi instruction-set più completi, comprensivi di istruzioni di controllo di flusso più flessibili. Il culmine dell'evoluzione di questo modello si ebbe con la *NVIDIA GeForce 7800*, dotata di ben tre diversi motori di shading (detti *shader*) programmabili individualmente e che agiscono in diversi stadi della graphics pipeline (*vertex operations*, *geometric operations*, *fragment operations*).

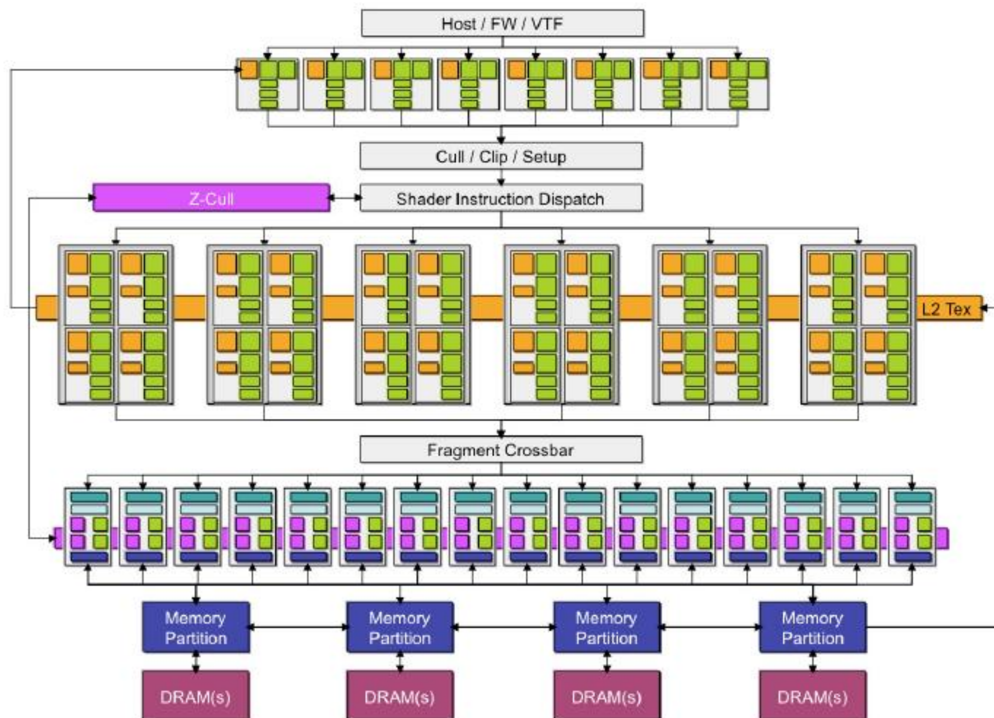



**Figura 3-1 Architettura della *NVIDIA GeForce 7800*. Si possono notare le tre diverse tipologie di motori di shading programmabili**

E' grazie a questa capacità di programmazione che è nato il GPU Computing. Alcuni pionieri iniziarono a cercare un modo per poter sfruttare l'elevato parallelismo e la grossa capacità di calcolo offerta dai vari motori di shading per scopi *general purpuse*, diversi dal rendering *3D*. L'idea è stata quella di alimentare la graphic pipeline con dati di input di natura non grafica, ma comunque strutturati come vertici o texture, di programmare le varie unità di shading per eseguire gli algoritmi voluti ed infine di recuperare i risultati dall'ultimo stage della pipeline. I primi risultati ottenuti furono molto buoni e produssero grandi speranze sul futuro di questa tecnica.

Il grosso problema della struttura di questa pipeline era il bilanciamento di carico. Come in tutte le pipeline, le performance dipendono dallo stadio più lento. Se, ad esempio, si ha un vertex program molto complesso ed un fragment program molto semplice, le performance dell'intera pipeline dipendono comunque dal vertex program. Il passo successivo è stato fatto nel 2006, con l'uscita della *NVIDIA GeForce 8 Series*, tramite l'introduzione di un'architettura a *shader unificato*. In questa nuova architettura, anziché avere tre diverse batterie di shader programmabili, ognuna dedicata ad uno specifico stage della pipeline, si ha un'unica batteria di shader. Ogni shader prende il nome di *streaming processor* o *core*, secondo la terminologia usata in maniera specifica da *NVIDIA* (da non confondere con il concetto

di *Core* delle CPU, completamente differente) e può essere paragonato ad una *ALU*. I *core* sono poi raggruppati in gruppi di dimensione fissa, denominati *streaming multiprocessor – SM*. Le operazioni sui vertici, di geometria o sui fragment vengono suddivise in tre diverse tipologie di thread ed uno scheduler si occupa di assegnare i vari thread (che vengono divisi in blocchi) agli SM liberi, secondo necessità. L'introduzione del modello a shader unificato permette quindi di risolvere il problema di bilanciamento di carico delle varie operazioni. La *GeForce 8800 Gtx*, ad esempio, è dotata di 16 SM, ognuno a sua volta provvisto di 8 core, per un totale di 128 unità di esecuzione. Ogni core è in grado di eseguire, in un solo ciclo di clock, una operazione semplice floating-point a precisione singola (32 bit) o intera (32 bit).

**Figura 3-2 Architettura della *NVIDIA GeForce 8800 Gtx*, dotata di 16 SM**

Con l'uscita della GeForce 8 Series è stato introdotto *CUDA* (*Compute Unified Device Architecture*), il primo ambiente di sviluppo per applicazioni GPU, basato su un linguaggio molto simile al *C*. CUDA ha fornito un modello di programmazione molto più semplice ed efficace, rispetto a tutti i precedenti approcci di GPGPU.

Un altro forte supporto al GPU Computing si ha avuto con l'uscita sul mercato, sempre da parte di *NVIDIA*, della linea di prodotti denominati *Tesla*, orientati al calcolo HPC. Queste schede sono caratterizzate dall'assenza di connettori video e da driver ottimizzati per i calcoli GPGPU, anziché per il rendering *3D*.

**Figura 3-3** ***NVIDIA Tesla M2070*****. Si può notare l'assenza di connettori video**

Le successive evoluzioni dell'architettura delle GPU hanno riguardato principalmente miglioramenti di tipo quantitativo, mantenendo sempre la struttura a shader unificato. Dopo la rivoluzionaria architettura (denominata anche con il termine *Tesla*) introdotta dalla *NVIDIA GeForce 8 Series*, si possono individuare due nuove principali architetture:

- architettura *Fermi*: nata nel 2010 con l'introduzione della *NVIDIA GeForce 400 Series*. Le GPU che seguono tale architettura sono dotate di 14,15 o 16 SM, ognuno provvisto di 32 core, per un totale di 448, 480 o 512 unita di elaborazione.
- architettura *Kepler*: nata nel 2012, con la *NVIDIA GeForce 600 Series*. Le GPU possiedono fino a 15 streaming multiprocessor (denominati *SMX* nell'architettura Kepler), ognuno provvisto di 192 core, per un totale di 2880 unità di esecuzione.

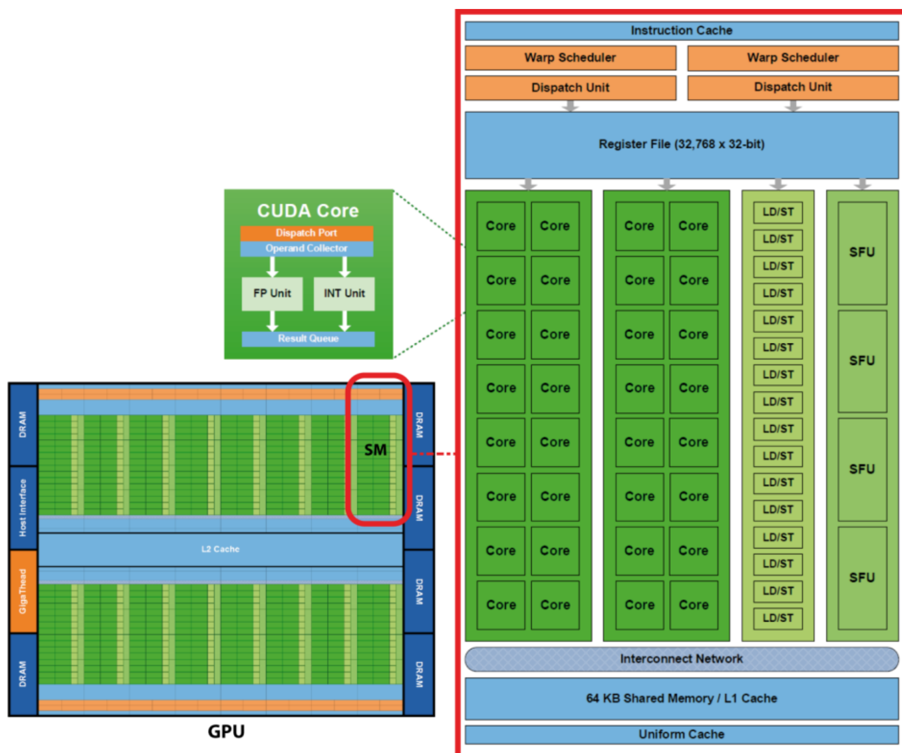



**Figura 3-4 Architettura** ***Fermi***

## 3.2 NVIDIA CUDA (Compute Unified Device Architecture)

La tecnologia *NVIDIA CUDA* (*Compute Unified Device Architecture*) è una architettura di tipo general-purpouse, che permette l'esecuzione parallela di programmi strutturati in più thread, sulle *GPU NVIDIA* [15]. La GPU viene considerata come un generico *device* in grado di effettuare dei calcoli; per questo motivo e per mantenere la massima conformità alla documentazione ufficiale *NVIDIA*, nel seguito verrà utilizzato il termine *device* come sinonimo di GPU. CUDA fornisce un modello di programmazione che permette di sviluppare software in grado di scalare, in maniera trasparente, rispetto alle caratteristiche hardware delle GPU su cui viene posto in esecuzione. Offre inoltre un *software environment* che permette ai programmatori di utilizzare un'estensione del linguaggio *C* (denominato *CUDA C*), come linguaggio di programmazione di alto livello. E' supportata inoltre l'integrazione con altri linguaggi di programmazione, sia in maniera nativa (*C++, FORTRAN, Direct Compute, OpenCL, OpenACC*), sia con l'ausilio di wrapper sviluppati da terze parti (*Java* con *JCuda*, *Phyton* con *PyCUDA*, ecc.).

Il modello di programmazione di CUDA si basa sulla definizione di funzioni, dette *kernel functions*, che vengono eseguite *N* volte in parallelo, da *N* differenti *CUDA threads*. I thread sono raggruppati e organizzati su 3 dimensioni, in insiemi di thread che prendono il nome di *blocchi*. A loro volta i diversi blocchi sono organizzati in una *griglia* su 3 dimensioni, venendo a creare una *gerarchia di thread*.

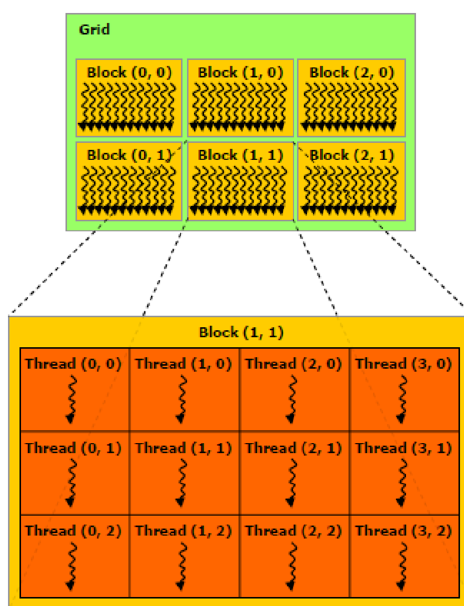



**Figura 3-5 Gerarchia di thread**

Ad ogni thread è associato un identificatore univoco (*threadID*) nel blocco a cui appartiene, accessibile all'interno della kernel function tramite la variabile *threadIdx*. La variabile *threadIdx* è organizzata come un vettore tri-dimensionale ed espone tre componenti intere: *threadIdx.x*, *threadIdx.y*, e *threadIdx.z*. Questo fornisce un metodo naturale per lavorare in domini come vettori, matrici o volumi. Secondo lo stesso principio, anche ai blocchi è associato un identificatore univoco (*blockID*) a cui corrisponde la variabile *blockIdx*, anch'essa organizzata a tre componenti (*blockIdx.x*, *blockIdx.y*, *blockIdx.z*). Sono inoltre fornite dall'enviroment alle kernel function due ulteriori variabili, sempre vettoriali a tre componenti: *blockDim* che dà la dimensione del blocco (il numero di thread per blocco) e *gridDim* che dà la dimensione della griglia (il numero di blocchi della griglia). Nel caso di domini mono-dimensionali, si può individuare in maniera univoca un thread all'interno di una griglia tramite la seguente espressione:

$$gid = blockIdx.x * blockDim.x + threadIdx.x$$

Supponiamo, ad esempio, di volere eseguire la somma di due vettori $A$ e $B$ di dimensione $N$ e di voler memorizzare il risultato in un vettore $C$. E' possibile effettuare tale operazione in *CUDA* tramite $N$ differenti thread, dove ogni $i$*-esimo* thread $T_i$ si occupa della somma degli $i$*-esimi* elementi di $A$ e $B$ ($C[i] = A[i] + B[i]$). Fissiamo la dimensione dei blocchi pari ad un valore $NT$ e utilizziamo $NB$ blocchi di $NT$ thread, con $NB = \left\lceil \frac{N}{NT} \right\rceil$. Possiamo definire la seguente kernel function per risolvere tale problema:

```
//kernel function
__global__ void VecAdd (float * A, float * B, float * C, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) { // N può non essere esattamente multiplo di NT
          C[i] = A[i] + B[i]
    }
}
```

A questo punto è necessario definire un normale programma per CPU che inizializzi l'ambiente CUDA, copi i vettori $A$ e $B$ dalla memoria *RAM* della CPU sulla memoria del device, lanci la griglia di $NB$ blocchi di $NT$ thread con la kernel function appena definita ed infine recuperi il vettore *C* di output, copiandolo dalla memoria del device alla memoria *RAM* della CPU. Tutte queste operazioni vengono effettuate sfruttando delle semplici API offerte dal *software environement* di CUDA.

Ad esempio, in C si può utilizzare il seguente codice:

```
int main() {
  ...

  int NT = 128; // == blockDim.x
  int NB = ceil(N/NT); // == gridDim.x

  // Copy data from HOST arrays to DEVICE arrays
  cudaMemcpy(d_A, A, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, B, N*sizeof(float), cudaMemcpyHostToDevice);

  // Invoke kernel with specified thread array
  VecAdd <<< NB, NT >>> (d_A, d_B, d_C, N);

  // Transfer result from DEVICE to HOST
  cudaMemcpy(C, d_C, N*sizeof(float), cudaMemcpyDeviceToHost);

  ...
}
```

I thread che appartengono ad uno stesso blocco possono inoltre utilizzare una memoria *on-chip* molto veloce, condivisa ed esclusiva per il blocco, detta *shared memory*, e possono coordinare la loro esecuzione negli accessi in memoria. In particolare è possibile definire dei punti di sincronizzazione all'interno della kernel function, tramite la funzione *__syncthreads()*. Tale funzione si comporta come una barriera ed il flusso di esecuzione di ogni thread dopo tale chiamata può proseguire solamente quando tutti i thread del blocco sono arrivati in quel punto.

Dal punto di vista hardware, l'architettura CUDA è basata su un array di *streaming multiprocessor*. Ogni blocco può essere schedulato per l'esecuzione su un qualsiasi SM libero della GPU, in modo che uno stesso programma CUDA possa essere eseguito su un qualsiasi numero di multiprocessori e che solo il *runtime* abbia la necessità di conoscere l'organizzazione fisica della GPU. Questo permette di ottenere trasparenza e scalabilità.

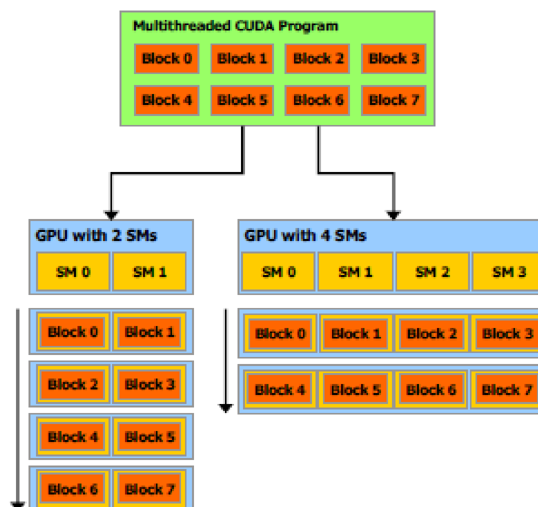



**Figura 3-6 Scheduling dei blocchi agli streaming multiproccesor**

I multiprocessori sono in grado di eseguire in maniera concorrente migliaia di thread e si basano sull'architettura chiamata *SIMT* (*Single-Instruction, Multiple-Thread*). I multiprocessori creano, gestiscono ed eseguono i thread a gruppi di 32 thread paralleli, detti *warp*. Quando ad un SM vengono associati uno o più blocchi di thread, l'hardware li partiziona in warp ed un *warp scheduler* si occupa del loro scheduling per l'esecuzione verso i *CUDA core*. Il metodo con cui un blocco è partizionato in warp è sempre il medesimo: ogni warp contiene thread con *threadID* consecutivi, dove il primo warp contiene il thread 0. Questo fattore è importante per poter operare alcune ottimizzazioni, come nel caso della tecnica di *riduzione parallela* (illustrata nel capitolo successivo). Per ogni warp, il multiprocessore esegue una singola istruzione comune a tutti i thread del warp, quindi la piena efficienza si ha solamente quando tutti i 32 thread del warp non *divergono* (e questo può capitare a causa di *branch* o di *loop*). In caso in cui i thread divergano in diversi *branch path*, il warp viene eseguito in maniera sequenziale per ogni branch path, limitando fortemente l'efficienza. Inoltre ogni multiprocessore, a differenza delle moderne CPU, esegue le istruzioni in ordine; non si ha predizione dei branch né esecuzione speculativa. Questo implica che per ottenere buone performance è necessario limitare il più possibile *branch* e *loop*, in particolare se potenzialmente divergenti per i vari thread appartenenti ad uno stesso warp. L'architettura SIMT ha molti aspetti comuni con l'architettura *SIMD* (*Single-Instruction*, *Multiple-Data*). Nel paradigma SIMD si hanno delle organizzazioni di esecuzione vettoriali parallele, in cui una singola istruzione controlla più unità di esecuzione in parallelo, che eseguono tutte la stessa operazione su differenti dati. Una grossa differenza tra le due architetture è che mentre le organizzazioni SIMD richiedono l'intera gestione del parallelismo lato software, il paradigma SIMT permette di specificare l'intero path di esecuzione e di branch di un singolo thread. Questo permette al programmatore di scrivere un codice quasi del tutto indipendente dal numero di thread e dalle caratteristiche hardware sottostanti.

I vari CUDA thread possono accedere a diversi spazi di memoria, durante la propria esecuzione. Tutti i thread di un *device* hanno accesso alla *global memory*, i thread di uno stesso blocco possono accedere alla *shared memory* dedicata a quel singolo blocco ed infine ogni singolo thread è dotato di uno spazio di memoria privato, detto *local memory*. La *global memory* risiede nella memoria *RAM* della scheda video, che prende il nome di *device memory*. Attualmente le schede grafiche più moderne utilizzano delle memorie *SDRAM GDDR5* e come massima banda teorica raggiungono quasi i 200 *GB/s*. La *shared memory* è una memoria *on-chip* gestita direttamente dal

programmatore (a differenza delle tradizionali *cache*); è l'applicazione stessa che esplicitamente si occupa dell'allocazione dei dati in essa e del relativo accesso. La shared memory ha una banda superiore e tempo di latenza molto inferiore rispetto alla global memory (dello stesso ordine di una cache *L1*), ma ha dimensioni limitate (16k nelle architetture *Tesla*, al più 48k nelle più recenti architetture *Fermi* e *Kepler*). La shared memory è presente direttamente all'interno di ogni streaming multiprocessor ed è quindi privata per ogni singolo SM (e di conseguenza privata per ogni singolo blocco). La *local memory* risiede sempre nella device memory e quindi ha la stessa latenza di accesso e la stessa banda della global memory. Viene utilizzata in automatico per memorizzare delle variabili che il compilatore decide di non porre all'interno dei registri, a causa di un numero non sufficiente di *registri* disponibili (o per motivi di ottimizzazione). Si hanno infine due ulteriori spazi di memoria accessibili a tutti i thread: la *constant memory* e la *texture memory*. La *constant memory* ha una dimensione di *64k*, è allocata nella device memory ed è *cached*, tramite la porzione di memoria on-chip dei singoli SM che non viene sfruttata come shared memory. Viene utilizzata per memorizzare delle costanti, è accessibile in sola lettura alle kernel function e può essere scritta solamente dalla CPU, tramite un'apposita *API*. La *texture memory* è anch'essa allocata nella device memory, *cached* e a sola lettura per le kernel function. E' strutturata in modo da ottimizzare gli accessi locali *2D*.

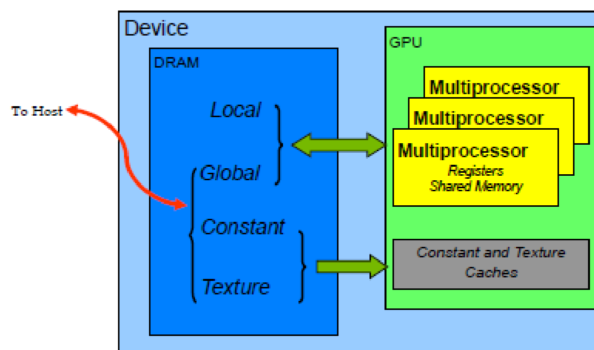



**Figura 3-7 I differenti spazi di memoria in CUDA**

C'è inoltre da sottolineare che, nelle GPU più moderne, parte della memoria on-chip degli SM è utilizzata anche come *cache L1* per gli accessi alla device memory, permettendo quindi di ottenere maggiori prestazioni nell'accesso alla global e local memory. In particolare le GPU basate sulle architetture *Fermi* e *Kepler* sono dotate di 64k di on-chip memory e permettono al programmatore di selezionare quanta memoria dedicare alla shared memory e quanta alla cache *L1*. Sono possibili due

scelte: *PreferShared* (48k di shared memory – 16k di cache *L1*) e *PreferL1* (16k di shared memory – 48k di cache *L1*). Oltre ad avere una cache L1, tali GPU dispongono anche di una *cache L2*, condivisa tra tutti i multiprocessori. Le GPU basate sulla precedente architettura *Tesla* non dispongono né di cache *L1*, né di cache *L2*.

A causa delle differenze tra le diverse architetture e delle feature di cui dispongono, le GPU *NVIDIA* che supportano CUDA vengono classificate tramite identificatore numerico, che prende il nome di *compute capability*. La compute capability di un device è formata da un *major revision number* ed un *minor revision number*. I device che presentano lo stesso *major revision number*, dispongono della medesima architettura di base. Il *minor revision number* indica miglioramenti incrementali rispetto ad essa. Attualmente si hanno 3 possibili *major revision number*: 1 per l'*architettura Tesla*, 2 per l'*architettura Fermi*, 3 per l'*architettura Kepler*.

| **Architecture specifications** | **Compute capability (version)** | | | | | | |
|---|---|---|---|---|---|---|---|
| | *1.0* | *1.1* | *1.2* | *1.3* | *2.0* | *2.1* | *3.0* |
| Number of cores for integer and floating-point arithmetic functions operations | 8 | | | | 32 | 48 | 192 |
| Number of special function units for single-precision floating-point transcendental functions | 2 | | | | 4 | 8 | 32 |
| Number of texture filtering units for every texture address unit or *render output unit* (ROP) | 2 | | | | 4 | 8 | 32 |
| Number of warp schedulers | 1 | | | | 2 | 2 | 4 |
| Number of instructions issued at once by scheduler | 1 | | | | 1 | 2 | 2 |

**Tabella 3-1 CUDA Compute capability**

Analizziamo infine una serie di fattori, che incidono sulle performance delle applicazioni CUDA. Si hanno tre elementi chiave che condizionano fortemente le prestazioni: il livello di utilizzo dei multiprocessori, le strategie di accesso alla global memory e il grado di divergenza dei warp [16].

Per poter ottenere buone performance, è opportuno mantenere i multiprocessori del device il più possibile occupati. Nel caso in cui il lavoro da eseguire non sia ben bilanciato tra gli SM, si ottengono delle prestazioni sub-ottimali. E' quindi fondamentale progettare i thread ed i blocchi di thread in modo tale da massimizzare l'utilizzazione dell'hardware e da permettere al runtime di CUDA la libera distribuzione del carico di lavoro ai diversi multiprocessori. Un concetto chiave relativo a questo problema è il quello di *occupancy* (o grado di occupazione dei multiprocessori). Le istruzioni dei vari thread sono eseguite in maniera sequenziale in CUDA e per questo motivo, quando un warp non può eseguire (ad esempio perché in attesa di dati provenienti dalla global memory o di una operazione __syncthreads()),

l'unico modo di mascherare le latenze è mantenere l'hardware occupato, rendendo disponibili al multiprocessore altri warp da eseguire (che possono far parte dello stesso blocco o di un differente blocco). Il principio di base è molto simile a quello del *multithreading* adottato da alcune CPU. Si definisce *occupancy* il rapporto tra il numero di warp attivi per multiprocessore e il massimo numero possibile di warp attivi. E' quindi importante mantenere un livello di occupancy sufficientemente elevato, per poter garantire buone performance. A seconda delle proprie risorse hardware e di quelle richieste dai thread, ogni GPU può mantenere attivi nei propri multiprocessori un certo numero massimo di blocchi. In particolare l'occupancy dipende da due fattori: il numero di *registri* richiesti e la quantità di *shared memory* richiesta da ogni blocco. Se le richieste sono superiori alle capacità di ogni SM, i nuovi blocchi rimangono sospesi e possono essere allocati in un multiprocessore solamente quando i precedenti hanno finito l'esecuzione. Registri e shared memory sono risorse limitate negli SM e per mantenere un elevato grado di occupancy, è opportuno limitarne la loro utilizzazione alla quantità minima necessaria per l'esecuzione. Inoltre è opportuno dimensionare i blocchi e la griglia per favorire l'ottenimento della massima occupancy possibile. La dimensione dei blocchi deve essere, se possibile, multipla di 32 (la dimensione dei warp) e sufficientemente elevata da garantire un numero sufficiente di warp, per ottenere una buona occupancy. Il numero di blocchi deve essere sufficientemente elevato da assicurare un'alta occupancy degli SM.

| **Compute Capability** | 2 |
|---|---|

**Resource Usage:**

| Threads Per Block | 256 |
|---|---|
| Registers Per Thread | 16 |
| Shared Memory Per Block (bytes) | 4096 |

**Physical Limits for Compute Capability 2.0**

| Threads per Warp | 32 |
|---|---|
| Warps per Multiprocessor | 48 |
| Threads per Multiprocessor | 1536 |
| Thread Blocks per Multiprocessor | 8 |
| Total # of 32-bit registers per Multiprocess | 32768 |
| Register allocation unit size | 64 |
| Register allocation granularity | warp |
| Registers per Thread | 63 |
| Shared Memory per Multiprocessor (bytes) | 49152 |
| Shared Memory Allocation unit size | 128 |
| Warp allocation granularity | 2 |
| Maximum Thread Block Size | 1024 |

**GPU Occupancy Data:**

| Active Threads per Multiprocessor | 1536 |
|---|---|
| Active Warps per Multiprocessor | 48 |
| Active Thread Blocks per Multiprocessor | 6 |
| Occupancy of each Multiprocessor | 1 |

| **Allocated Resources** | | **Per Block** | **Limit Per SM** | **= Allocatable Blocks Per SM** |
|---|---|---|---|---|
| Warps | (Threads Per Block / Threads Per Warp) | 8 | 48 | 6 |
| Registers | (≈Registers Per Thread * Threads Per Blo | 4096 | 32768 | 8 |
| Shared Memory (Bytes) | | 4096 | 49152 | 12 |

| **Maximum Thread Blocks Per Multiprocessor** | **Blocks/SM** | *** Warps/Block** | **= Warps/SM** |
|---|---|---|---|
| Limited by Max Warps or Max Blocks per Multiprocessor | 6 | 8 | 48 |
| Limited by Registers per Multiprocessor | 8 | 8 | 0 |
| Limited by Shared Memory per Multiprocessor | 12 | 8 | 0 |

**Physical Max Warps/SM = 48    Occupancy = 48 / 48 = 100%**

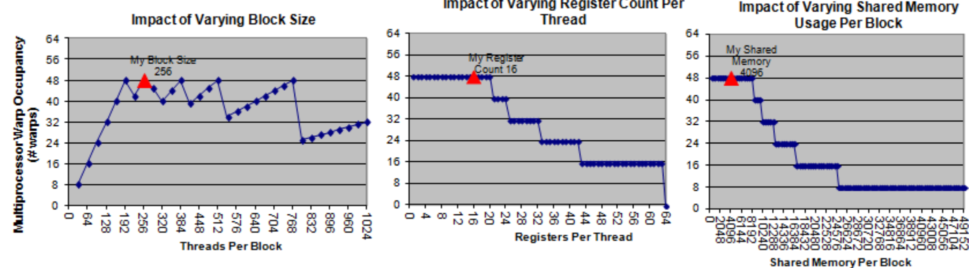



**Figura 3-8 Variazione dell'occupancy modificando la dimensione dei blocchi, il numero di registri per thread e la quantità di shared memory per blocco, in un device con compute capability 2.0**

Un altro importante fattore da considerare riguarda la modalità di accesso in global memory. Gli accessi in lettura e in scrittura in global memory da parte dei thread di un warp (o di un *half-warp* nei dispositivi con compute capability 1.x, ovvero 16 thread) vengono fusi in un'unica transazione quando possibile, se alcuni requisiti vengono soddisfatti. Si parla, in questo caso, di *coalesced access*. Per poter ottenere delle buone perfomance è opportuno progettare il codice e le strutture dati in modo da favorire questo tipo di accesso. I requisiti per permettere un coalesced access dipendono dalla CUDA compute capability della GPU e con i vari miglioramenti architetturali tali vincoli sono diventati sempre meno rigidi.

Per device con compute capability 1.0 e 1.1, gli accessi in global memory avvengono per half-warp (16 thread). Per poter effettuare un coalesced access si devono verificare le seguenti condizioni:

- la dimensione delle *memory word* a cui accedono i thread deve essere di 4, 8 o 16 bytes e l'accesso deve avvenire nello stesso segmento *allineato*
    - se le word sono di 4 byte, tutte le 16 word devono appartenere allo stesso segmento allineato di 64 byte (tipico caso delle variabili floating point a singola precisione o delle variabili intere)
    - se le word sono di 8 byte, tutte le 16 word devono appartenere allo stesso segmento allineato di 128 byte (tipico caso delle variabili floating point a doppia precisione)
    - se le word sono di 16 byte, le prime 8 word devono risiedere nello stesso segmento allineato di 128 byte e le ultime 8 nel successivo segmento di 128 byte
- i thread devono accedere alle memory word in sequenza: il *k-esimo* thread di un half-warp deve accedere alla *k-esima* word del segmento.

Se i requisiti vengono rispettati, viene effettuata una singola transazione in memoria da 64 byte o 128 byte (o doppia nel caso di word di 16 byte). In caso contrario, vengono effettuate 16 transazioni in memoria separate, da 32 byte ciascuna. Un coalesced access può essere effettuato anche quando il warp è divergente e ci sono alcuni thread inattivi che non richiedono un accesso in global memory.

Per le GPU con compute capability 1.2 e 1.3, tali condizioni sono un poco più rilassate. I vincoli di dimensione e di allineamento delle memory word rimangono i medesimi. I thread possono però accedere alle memory word in ogni ordine (anche alle stesse word) e la GPU effettua una singola transazione per ogni segmento a cui ha accesso l'half-warp.

Per le GPU con compute capability 2.x e 3.x, gli accessi in global memory sono *cached*. Inoltre, tramite un apposito flag in compilazione, è possibile specificare se utilizzare entrambe le cache *L1* ed *L2* (*-Xptxas -dlcm=ca* - condizione di default) oppure solamente la cache *L2* (*-Xptxas -dlcm=cg*). Una linea di cache è di 128 byte ed è mappata in un segmento allineato di memoria di 128 byte. Gli accessi contemporanei dei thread di un warp vengono serviti da un numero di transazioni in memoria pari al numero di linee di cache necessarie per servire tutti i thread del warp. Se si utilizzano entrambe le cache *L1* e *L2* gli accessi in memoria sono serviti da transizioni in memoria di 128 byte, mentre se si utilizza solamente la cache *L2* da transazioni di 32 byte.

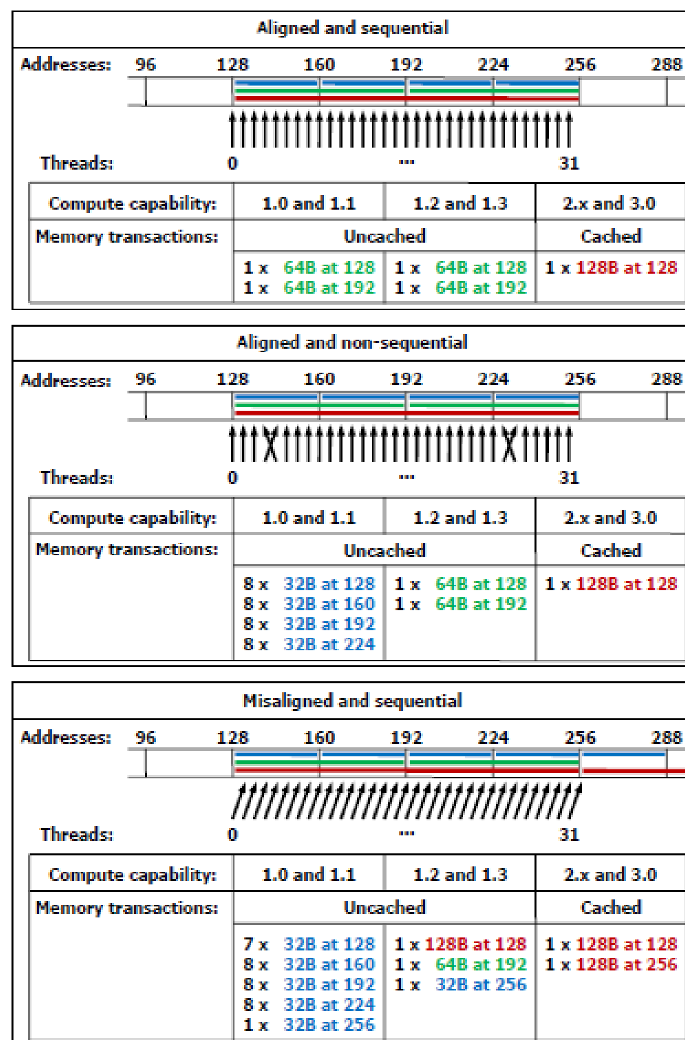
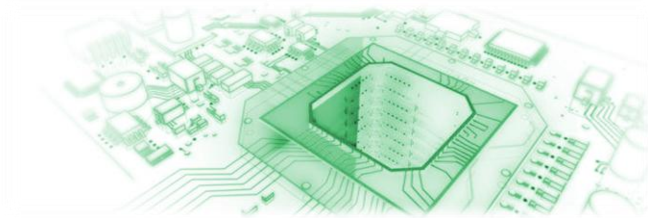



**Figura 3-9 Modalità di accesso in global memory**

Un altro fattore che incide fortemente sulle performance è il *grado di divergenza dei thread*. Come detto in precedenza, nel caso in cui i thread divergano in diversi *branch path* (a causa di *if* o *loop*), il warp viene eseguito in maniera sequenziale per ogni

branch path. Il codice delle kernel function deve essere strutturato in modo tale da limitare al massimo questi fenomeni.

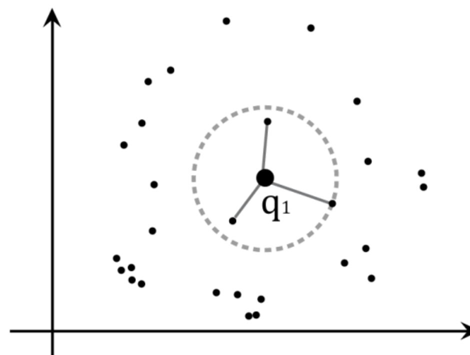
Oltre a questi fattori chiave, si hanno diversi altri elementi secondari che incidono sulle performance. Tra questi è opportuno citare il problema dei *conflitti tra i banchi di memoria della shared memory* (*bank conflict*). Per quanto riguarda gli accessi in shared memory, non si hanno particolari vincoli di accesso, a differenza della global memory. Ogni thread può accedere contemporaneamente a differenti dati, purché non nascano conflitti tra i banchi di memoria. Per poter ottenere un'alta banda di trasferimento, la shared memory è divisa in moduli di memoria della stessa dimensione, che prendono il nome di *banchi* (*banks*), a cui è possibile accedere simultaneamente. Nel caso si presentino *n* richieste di lettura o scrittura, a *n* indirizzi appartenenti a banchi diversi, è possibile servire contemporaneamente le *n* richieste, tramite la tecnica dell'*interleaving*. Se però due (o più) indirizzi di memoria appartengono allo stesso banco si crea un conflitto e i due accessi vengono serializzati. Per ottenere buone prestazioni è importante fare in modo che differenti thread di uno stesso warp accedano a banchi differenti. Nei dispositivi con compute capability 1.x, la shared memory è formata da 16 banchi, organizzati in modo tale che word successive di 32 bit siano mappate in banchi successivi differenti. Nei dispositivi con compute capability 2.x e 3.x, la shared memory è formata da 32 banchi e word successive di 32 bit sono sempre mappate in banchi successivi differenti. Per non generare conflitti, è sufficiente che i thread di uno stesso warp (o anche solo di un half-warp nel caso di compute capability 1.x) accedano alla shared memory con indirizzi consecutivi. Ad es, nel caso di $NT$ thread (con $NT \geq 32$) e di un array $x[]$ in shared memory, non si hanno mai conflitti se ogni thread $T_i$ accede all'elemento $x[i]$. Inoltre non si hanno conflitti anche nel caso di più thread che accedono in lettura allo stesso indirizzo di memoria.



# Capitolo 4: Tecniche di GPU Computing per il problema *k-NN*

Il primo obiettivo di questo lavoro è sfruttare la grande potenza di calcolo delle GPU per ottimizzare le performance degli algoritmi ODPNestedLoop e ODPSolvingSet. Le GPU sono particolarmente adatte ad eseguire applicazioni formate da un alto numero di thread, dominate da un alto numero di operazioni di calcolo. I due algoritmi sono per loro natura sequenziali, ma possono essere efficacemente ristrutturati per permettere una esecuzione fortemente parallela. Ad esempio, nell'algoritmo ODPNestedLoop viene calcolata la distanza tra ogni coppia di punti del dataset: è possibile effettuare questa operazione in parallelo, definendo un thread per ogni punto $p \in D$, che si occupi del calcolo delle distanze tra $p$ ed ogni altro punto del dataset. Per cercare di individuare le tecniche migliori per l'implementazione su GPU di tali algoritmi, analizziamo in primo luogo una serie di tecniche di GPU Computing proposte da diversi gruppi di ricerca per un problema fortemente legato a ODPNestedLoop e ODPSolvingSet, denominato *k-NN problem*.

Sia $D = \{p_1, \dots, p_d\}$ un insieme di $d$ *reference point* con valori in $\mathbb{R}^a$ e sia $C = \{q_1, \dots, q_m\}$ un insieme di $m$ *query point* nello stesso spazio. Il $k$ *nearest neighbor* (*k-NN) problem* consiste nel trovare i $k$ punti più vicini (*nearest neighbors*) ad ogni query point $q_j \in C$ in $D$, data una specifica funzione di distanza $dist$ definita su $D \cup C$.




**4-1 *k-NN* problem per *k*=3**

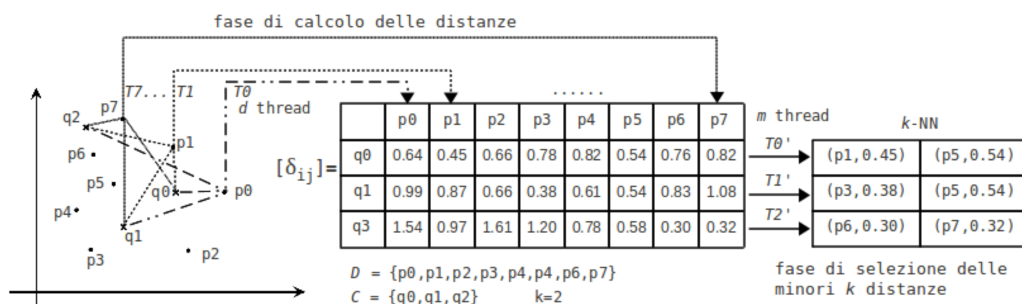
Una tecnica molto semplice per la risoluzione del *k-NN problem* è la *ricerca esaustiva* o *algoritmo a forza bruta* (*BF*). Per ogni query point $q_j \in C$, l'algoritmo $BF$ calcola tutte le distanze tra $q_j$ ed ogni $p_i \in D$ e seleziona i $k$ reference point, che corrispondono alle più piccole $k$ distanze. Se consideriamo ODPNestedLoop, tale metodo segue esattamente lo stesso procedimento dell'algoritmo BF, ponendo l'intero dataset sia come insieme dei reference point, sia come insieme dei query point. Se consideriamo ODPSolvingSet, ad ogni iterazione dell'algoritmo, nel loop di confronto $D$ con $C$ si utilizza un procedimento simile a quello di BF, ponendo il dataset come insieme dei reference point e l'insieme dei candidati come insieme dei query point.

Essendo il *k-NN problem* di importanza fondamentale in molti campi e l'algoritmo BF particolarmente adatto ad essere ristrutturato per una esecuzione parallela, vari gruppi di ricerca hanno individuato una serie di tecniche di GPU Computing per l'implementazione dell'algoritmo BF su GPU, sfruttando la tecnologia *NVIDIA CUDA*.

## 4.1 Soluzioni GPGPU note in letteratura per l'approccio BF al *k-NN* problem

*Garcia et al.* [17] hanno proposto il primo metodo basato su GPU, per l'esecuzione parallela dell'algoritmo *BF*. La loro soluzione, denominata *BF-CUDA-kNN*, prevede un procedimento in due fasi. Nella prima fase vengono calcolate, in parallelo, tutte le distanze tra i reference point e i query point e salvate in una matrice delle distanze (di dimensione $m \times d$, con $m = |C|,\ d = |D|$). Per questa operazione si definiscono $d$ thread $T_i$, dove $T_i$ calcola tutte le distanze tra $p_i \in D$ ed ogni $q_j \in C$ e le salva nella corrispondente colonna della matrice. Nella seconda fase, per ogni query point $q_j$, vengono ordinate le rispettive distanze (corrispondenti alla $j$-esima riga della matrice) e selezionate le prime $k$, tramite un singolo thread per ogni $q_j$. Come algoritmo di ordinamento è utilizzata una variante dell'*insertion sort* per $k$ piccolo, (indicato nell'articolo come $k < 120$ su un insieme di 4800 reference point) o il *comb sort* per $k$ grande. Il *comb sort* effettua l'ordinamento completo di tutte le $d$ distanze per ogni query point, mentre la variante proposta dell'*insertion sort* fornisce in output solamente le minori $k$ distanze ordinate. Nel caso dell'*insertion sort*, si definiscono $m$ thread $T_j$, ognuno dedicato al singolo query point $q_j$. Ogni thread $T_j$ mantiene in *global memory* un array $A_j$, contenente le $k$ minori coppie $\langle p_z, \sigma \rangle$, con $\sigma = dist(q_j, p_z)$, ordinate in ordine crescente secondo il valore delle distanze. Data in input una matrice delle distanze, $T_j$ confronta tutte le distanze tra $q_j$ ed ogni altro

reference point $p_z$, ed inserisce ogni coppia $\langle p_z, \sigma \rangle$ in $A_j$ tramite un algoritmo di tipo *insert-sort,* solo se $\delta > A_j[k-1]$. Un algoritmo di tipo *insert-sort*, eseguito in parallelo da più thread, permette di avere un basso grado di divergenza dei thread appartenenti allo stesso warp e permette un *coalesced access* alla global memory, nelle letture e scritture sugli array. I risultati sperimentali hanno fornito uno *speedup* di 10 sull'algoritmo sequenziale per CPU. Dalla loro analisi si evidenzia che, mentre negli algoritmi per CPU il grosso del tempo è speso nel calcolo delle distanze (91%), nell'algoritmo su GPU il tempo di ordinamento dei punti è molto significativo. Ad esempio, su di un insieme di 4800 punti (utilizzato sia come insieme reference point, che di query point) con dimensionalità 16 e $k = 20$, il 47% del tempo è speso nell'ordinamento delle distanze, il 51% dal calcolo delle distanze e il 2% per i trasferimenti in memoria. Questo è dovuto al fatto che, mentre il calcolo delle distanze su GPU può essere eseguito in parallelo in maniera particolarmente efficiente, l'ordinamento è un'operazione per sua natura poco adatta ad essere eseguita su GPU. Nell'articolo viene inoltre mostrata un'osservazione molto interessante. Aumentando il numero di coordinate dei punti, nell'implementazione su CPU i tempi di esecuzione crescono di un fattore pari 0.45, mentre nell'implementazione su GPU rimangono praticamente costanti, mostrando un fattore di 0.001. Questo evidenzia che la soluzione su GPU è particolarmente adatta a dataset con punti ad alta dimensionalità. Ipotizzando che la GPU sia in grado di eseguire contemporaneamente $d$ thread, la fase di calcolo delle distanze ha una complessità temporale pari a $O(m \cdot a)$, con $a$ pari alla dimensionalità (il numero di coordinate) dei reference point e query point. Rispetto ad una soluzione sequenziale, che richiede un tempo $O(d \cdot m \cdot a)$, è possibile scalare in via teorica di un fattore $d$, tramite una soluzione parallela. Per la fase di ordinamento delle distanze, ipotizzando questa volta che la GPU sia in grado di eseguire contemporaneamente $m$ thread, la complessità temporale è pari a $O(d \cdot k)$ nel caso dell'*insertion-sort* o $O(d \cdot \log(d))$ nel caso del *comb-sort*.




| | p0 | p1 | p2 | p3 | p4 | p5 | p6 | p7 |
|---|---|---|---|---|---|---|---|---|
| q0 | 0.64 | 0.45 | 0.66 | 0.78 | 0.82 | 0.54 | 0.76 | 0.82 |
| q1 | 0.99 | 0.87 | 0.66 | 0.38 | 0.61 | 0.54 | 0.83 | 1.08 |
| q3 | 1.54 | 0.97 | 1.61 | 1.20 | 0.78 | 0.58 | 0.30 | 0.32 |

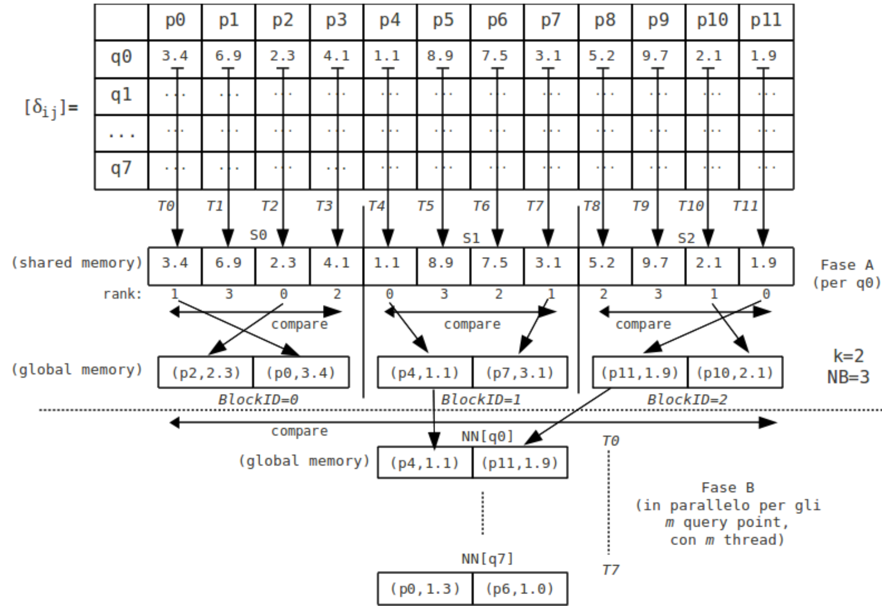
| | |
|---|---|
| (p1,0.45) | (p5,0.54) |
| (p3,0.38) | (p5,0.54) |
| (p6,0.30) | (p7,0.32) |

**4-2 Metodo di *Garcia***

*Quansheng et al.* [18] propongono un nuovo metodo, anch'esso suddiviso in una prima fase di calcolo delle distanze ed una seconda fase di ordinamento delle distanze calcolate, per l'individuazione delle minori $k$ per ogni query point. Nella prima fase viene costruita una matrice delle distanze di dimensione $m \times d$, scomponendola in piccole parti, dette $tile$, tali da poter essere allocate in *shared memory* durante la loro costruzione. Nella seconda fase, ogni riga della matrice viene ordinata sfruttando un'implementazione parallela dell'algoritmo *radix-sort*, che si basa sul concetto di *sorting network* e sulle note tecniche che permettono di eseguire le operazioni di *prefix-sum* in maniera parallela ed efficiente in CUDA. Con queste tecniche gli autori hanno ottenuto uno *speedup* intorno a 30, sull'algoritmo sequenziale per CPU. Ipotizzando che la GPU sia in grado di eseguire contemporaneamente $NB$ blocchi di $NT$ thread, la complessità della fase di selezione delle $k$ minori distanze è pari a $O\left(m \cdot \frac{d}{NT \cdot NB} \cdot w\right)$, con $w$ pari al numero di bit necessari per disintinguere i diversi $d$ valori delle distanze, per ogni query point. Ipotizzando di rappresentare le distanze con numeri floating point a precisione singola a 32 bit, assumendo indipendenti i valori delle $d$ distanze e ponendoci nel caso peggiore, possiamo porre $w = 32$.

*Liang et al.* [19] propongono una tecnica denominata *CUKNN*. Anche in questa soluzione si lavora in due fasi distinte, dove nella prima vengono calcolate in parallelo tutte le distanze tra reference point e query point e nella seconda, per ogni query point $q_j$, vengono selezionate le $k$ minori. Nella seconda fase si lavora su di un singolo query point $q_j$ alla volta, predisponendo $NB$ blocchi di $NT$ thread ciascuno (tutti dedicati a $q_j$), con $NB \cdot NT = d$. Si suddividono le distanze dei reference point rispetto a $q_j$, calcolate al passo precedente, tra i diversi blocchi e per ogni blocco ne viene caricata in *shared memory* la rispettiva porzione. Denominiamo la porzione di distanze spettante al blocco $z$ con $S_z$. Ogni thread $i$ del blocco $z$ si occupa dell'$i$*-esima* distanza di $S_z$ ed ottiene il rank di essa in $S_z$, confrontandola con tutte le altre in $S_z$. A questo punto, per ogni blocco, vengono selezionate le prime $k$ distanze. Infine un ulteriore thread si occupa di analizzare gli $NB$ set di $k$ distanze minime e di trovare i $k$ valori minimi globali. Gli autori sostengono di aver raggiunto uno speedup massimo di 15.2 rispetto ad una soluzione sequenziale su CPU, con un dataset di $524{,}000$ punti (con dimensionalità 8) e con $k = 7$. Ipotizzando che la GPU sia in grado di eseguire contemporaneamente $d$ thread, la complessità della fase di selezione delle $k$ minori distanze è pari a $O\left(m \cdot NT + \frac{d}{NT}\, k\right)$. Ogni thread, per determinare il rank della

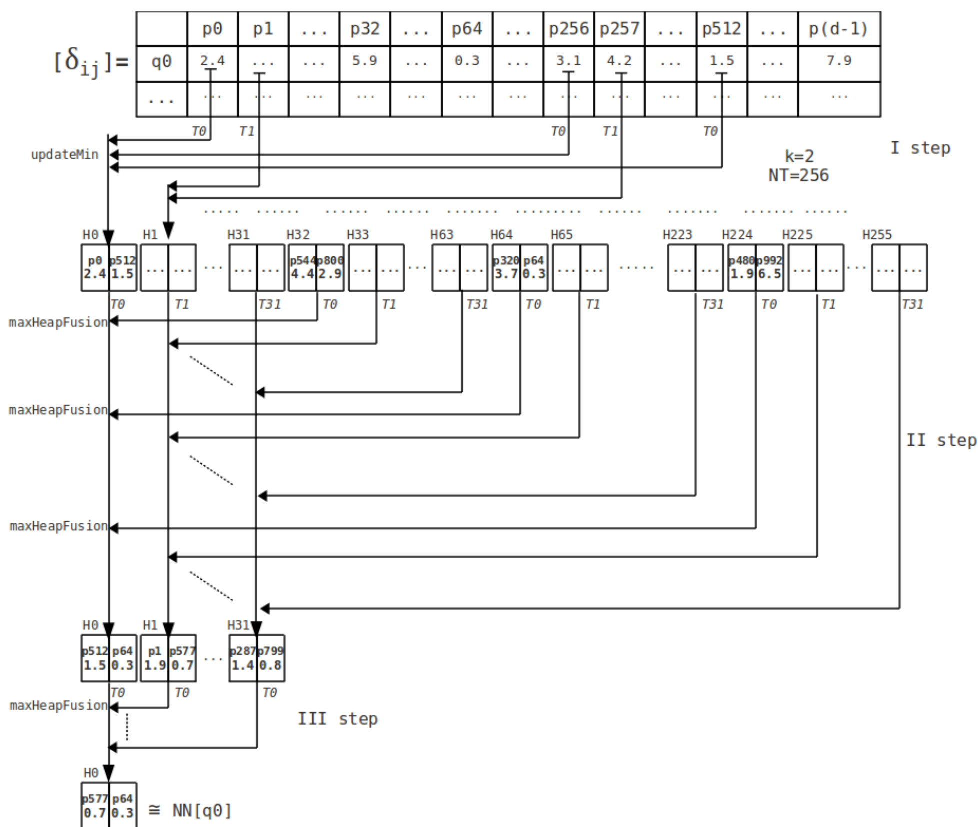
corrispondente distanza, esegue $NT$ confronti. La selezione finale delle minori $k$ distanze richiede $\frac{d}{NT}\,k$ confronti per ogni query point ed è possibile eseguirla in parallelo, tramite $m$ thread, ognuno dedicato ad un singolo query point.




**4-3 Metodo *CUKNN***

*Barrientos et al.* [20] criticano fortemente i metodi che si basano sull'ordinamento completo delle distanze calcolate (come quello di *Garcia* con *comb sort* e quello di *Quansheng*), in quanto essendo tipicamente $k \ll d$, è possibile utilizzare tecniche maggiormente ottimizzate, che sfruttino tale ipotesi per la selezione delle $k$ minori distanze. Propongono un approccio basato sugli heap, definito come *Heap-based Reduction*. Si definiscono $m$ blocchi di $NT$ thread, con ogni blocco dedicato ad un singolo query point $q_j \in \mathrm{C}$. L'insieme dei reference points viene suddiviso in $NT$ sottoinsiemi $S_i$. Nel primo step, ogni $i$*-esimo* thread del blocco $j$ calcola tutte le distanze tra $q_j \in \mathrm{C}$ ed ogni punto $p_z \in S_i$ e mantiene le minori $k$, salvando le coppie $\langle p_z, \sigma \rangle$ con $\sigma = dist(q_j, p_z)$, in un proprio max-heap di dimensione $k$ (sfruttando l'operazione $updateMin$, definita in precedenza in 2.3). A questo punto, per ogni blocco vengono fusi gli heap degli $NT$ thread, fino ad ottenerne uno singolo, tramite due ulteriori passi. Il secondo step è eseguito solamente dal primo *warp* del blocco (i primi 32 thread), dove ogni thread del primo warp si occupa di fondere $NT/32$ heap. Il terzo step è eseguito da un solo thread che unisce i 32 heap rimasti. Gli heap del primo step sono memorizzati in *global memory*, per non limitare troppo l'*occupancy* dei multiprocessori, mentre gli heap del secondo e terzo passo sono allocati in *shared memory*, essendo in numero inferiore. Per quanto riguarda i risultati sperimentali,

nell'articolo non è stato mostrato lo speedup rispetto ad una soluzione su CPU, ma è stato solamente esibito un confronto con una versione su GPU che utilizza l'algoritmo di ordinamento per GPU più veloce conosciuto (*quicksort* proposto da Cederman e Philips). Rispetto a tale soluzione con *quicksort* si è evidenziato uno speedup di 10, con un set di reference point e di query point di 300,000-1,000,000 punti e con $k = 8, 16, 32$. Ipotizzando che la GPU sia in grado di eseguire contemporaneamente $m \cdot NT$ thread, la complessità della selezione delle $k$ minori distanze è pari a $O\left(\frac{d}{NT}\log(k)\right)$. Ogni thread, nel primo step, analizza $\frac{d}{NT}$ distanze, eseguendo di conseguenza $\frac{d}{NT}$ operazioni di $updateMin$ sul proprio heap, che richiedono un tempo $O(\log(k))$. Il secondo ed terzo passo, per $d$ sufficientemente grande, richiedono un carico di lavoro molto inferiore rispetto al primo passo (rispettivamente $O\left(\frac{k \cdot NT}{32}\log(k)\right)$ e $O(32k \cdot \log(k))$) e possono quindi non essere considerati nel calcolo della complessità teorica.




**4-4 Metodo *Heap-based Reduction* di *Barrientos***

*Kato* e *Hosino* [21] propongono un approccio sempre diviso in due fasi, dove nella prima vengono calcolate in parallelo tutte le distanze tra reference point e query point e nella seconda, per ogni query point $q_j$, vengono selezionate le $k$ minori distanze. Per

quanto riguarda la seconda fase, viene utilizzato un algoritmo di *partial sorting* abbastanza originale, basato sugli heap. Si definiscono $m$ di $NT$ thread, ognuno dedicato ad un singolo query point $q_j$, e si associa ad ogni blocco un singolo max-heap di dimensione $k$. Ad ogni thread si associa un proprio buffer, di dimensione $buffsize$. I thread del blocco $j$ leggono in parallelo le distanze, calcolate e salvate nella fase precedente, ed inseriscono nel buffer solamente le coppie $\langle p_z, \delta \rangle$ (con $p_z \in$ D e $\delta = dist(q_j, p_z)$) tali che $\delta < \sigma$, dove $\langle \text{s}, \sigma \rangle$ è l'elemento radice dell'heap $j$ (cioè il punto con la massima distanza, tra i $k$ punti nell'heap), fino a riempire il buffer. A questo punto un solo thread si occupa di prelevare in sequenza gli elementi dai vari buffer e di inserirli nell'heap, con operazioni di tipo $updateMin$. Il procedimento viene iterato fino a quando i thread non hanno letto tutte le distanze. Per ogni blocco, la singola heap e gli $NT$ buffer vengono allocati in *shared memory* e, dimensionando opportunamente $buffsize$, è possibile ottenere una buona *occupancy* dei multiprocessori. Questa tecnica si basa sul principio che, se $k << d$, la maggior parte dei dati viene scartata e non inserita nei buffer. Inoltre, per quanto riguarda la prima fase, è proposto un metodo che permette di lavorare anche su reference set e/o query set di dimensioni molto elevate e di poter eventualmente sfruttare contemporaneamente più GPU. L'idea è quella di spezzare la matrice delle distanze in più sezioni, dette *chunk*, e di lavorare in sequenza o in parallelo (nel caso siano disponibili più GPU) sui vari chunk. E' necessaria infine un'azione eseguita dalla CPU, che si occupa di effettuare il *merge* dei risultati parziali riguardanti ogni chunk. Con un set di reference point e di query point di $80{,}000$ punti con dimensionalità $256$ e con $k = 100$, gli autori mostrano che si è riusciti ad ottenere uno speedup sulla CPU di $172.6$ con una sola GPU e di $331.7$ con 2 GPU. Ipotizzando che la GPU sia in grado di eseguire contemporaneamente $m \cdot NT$ thread, la complessità della fase di selezione delle $k$ minori distanze è pari a $O(\text{d} \cdot \log(k))$, in quanto il singolo thread di ogni blocco che lavora sull'heap esegue $d$ operazioni $updateMin$, nel caso peggiore. In realtà, nel caso medio, tramite il filtro che si adopera prima dell'inserimento delle coppie nel buffer dei vari thread, le operazioni sull'heap sono nettamente inferiori e quindi questa tecnica permette di ottenere comunque buone performance.

*Arefin et al.* [22] propongono infine un lavoro molto recente (2012) su questo problema, introducendo un metodo denominato GPU-FS-*K*NN. Anche in questo caso si opera in due fasi, dove nella prima vengono calcolate in parallelo tutte le distanze tra reference point e query point e nella seconda, per ogni query point $q_j$, vengono

selezionate le $k$ minori distanze. A tal fine, si definiscono $m$ thread, ognuno dedicato al singolo query point $q_j$. Ogni thread mantiene un array contenente i $k$ minori valori trovati ed un puntatore $Maxk$ all'elemento dell'array contenente il valore massimo dei $k$ valori. Per ogni distanza, ogni thread confronta tale valore con quello puntato da $Maxk$ e, nel caso sia inferiore, lo inserisce nella cella puntata da $Maxk$ e aggiorna $Maxk$, cercando il nuovo valore massimo nell'array. Anche in questo lavoro, per quanto riguarda la prima fase, è proposto un metodo che permette di lavorare anche su dataset di dimensioni molto elevate e di poter sfruttare contemporaneamente più GPU, spezzando la matrice delle distanze in più sezioni e poi eseguendo il *merge* finale dei risultati. Con un set di reference point e di query points di $1{,}533{,}876$ punti con dimensionalità $295$ e con $k = 20$, gli autori mostrano che si è riusciti ad ottenere uno speedup sulla CPU di $32$ con una sola GPU e di $57{,}8$ con due GPU. Per la fase di ordinamento delle distanze, ipotizzando che la GPU sia in grado di eseguire contemporaneamente $m$ thread, la complessità temporale è pari a $O(d \cdot k)$, in quanto, nel caso peggiore, ogni thread esegue $d$ inserimenti nell'array ed ogni inserimento richiede un tempo $O(k)$ per l'aggiornamento di $Maxk$.

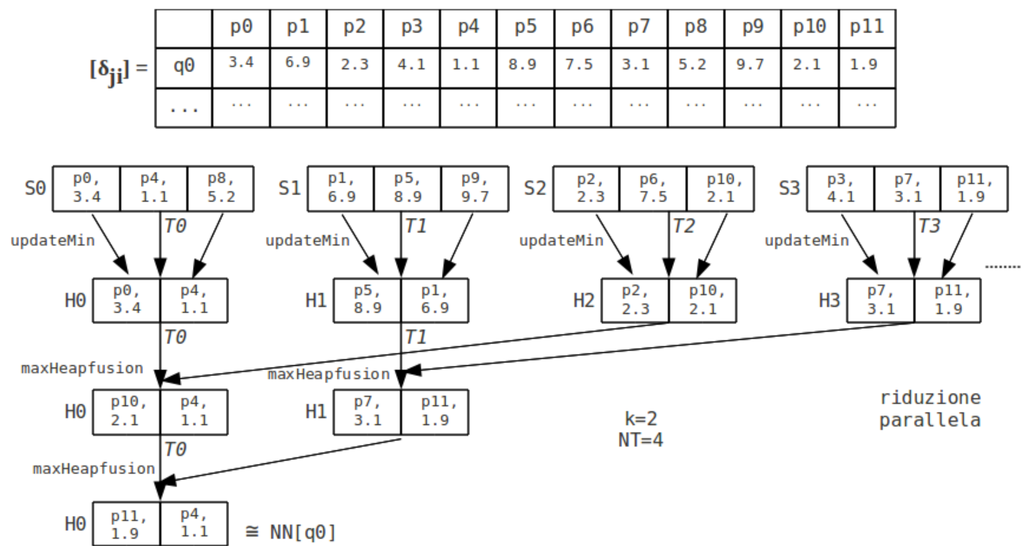
In conclusione, tutti i metodi presentati propongono un procedimento in due fasi: una prima fase di calcolo in parallelo delle distanze tra reference point e query points, con salvataggio dei valori in una matrice delle distanze, ed una seconda fase in cui, per ogni query point $q_j$, vengono selezionate le $k$ minori distanze. La prima fase è molto simile per tutti i metodi, in quanto può essere eseguita in modo parallelo, in maniera molto efficiente. Per la seconda fase, invece, si hanno molti diversi approcci, in quanto è particolarmente critica per soluzioni GPU.

### 4.2 Nuove soluzioni GPGPU proposte per l'approccio BF al *k-NN* problem

In aggiunta alla tecniche illustrate in precedenza, proponiamo una serie di metodi ulteriori, alcuni dei quali introducono maggiori elementi di novità, mentre altri possono essere considerati come varianti o ottimizzazioni ai precedenti. Per tali metodi focalizziamo l'attenzione solamente sulla fase più critica di selezione delle minori $k$ distanze.

Il primo metodo da noi proposto, è basato sulla nota tecnica della *riduzione parallela* [23] e può essere considerato un'ottimizzazione della tecnica esposta da *Barrientos*. Chiamiamo questo metodo *Heap complete reduction*. Come nella soluzione di *Barrientos*, definiamo $m$ blocchi di $NT$ thread, dove ogni blocco è dedicato ad un

query point $q_j \in$ C. L'insieme dei reference points viene suddiviso in $NT$ sottoinsiemi $S_i$, con $NT$ pari ad una potenza di 2. Prendendo in input la matrice delle distanze, ogni *i-esimo* thread del blocco $j$ seleziona i punti $p_z \in S_i$ con le minori $k$ distanze da $q_j$, salvando le rispettive coppie $\langle p_z, \sigma \rangle$ con $\sigma = dist(q_j, p_z)$, in un proprio max-heap di dimensione $k$, sfruttando l'operazione $updateMin$. A questo punto, per ogni blocco vengono fusi gli heap degli $NT$ thread, sfruttando la tecnica della riduzione parallela. La riduzione avviene in $\log_2 (NT)$ step: nel primo step vengono fusi a due a due $NT$ heap ottenendo $NT/2$ heap, nel secondo step vengono fusi a due a due i rimanenti $NT/2$ heap ottenendo $NT/4$ heap e si procede con questo processo fino a quando, nell'ultimo step, vengono fusi gli ultimi due heap rimasti, ottenendone uno solo. L'ultimo heap rimasto del blocco $j$ contiene i $k$ nearest neighbor del query point $q_j$.



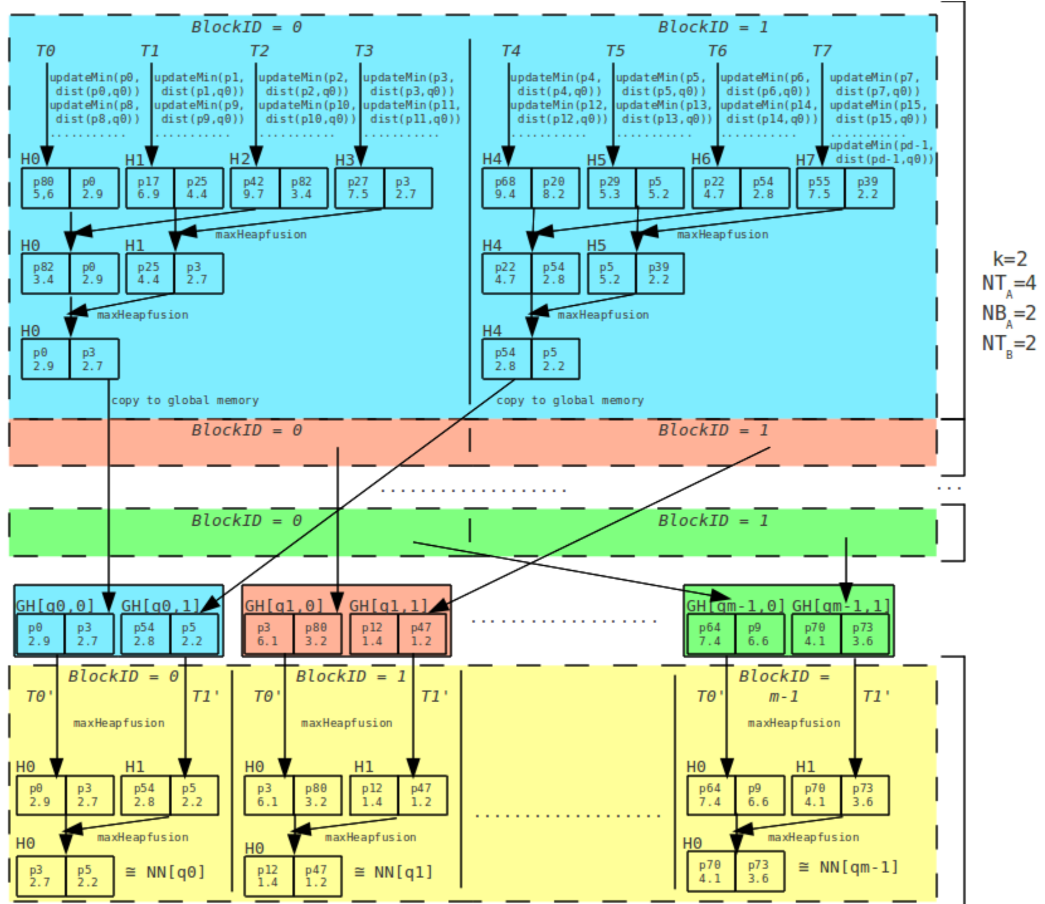

**4-5 Metodo *Heap complete reduction***

Affinché tale metodo possa essere eseguito in maniera efficiente, è necessario che gli heap siano allocati in shared memory. Questo non solo in quanto la shared memory ha un tempo di latenza inferiore a quello della global memory, ma soprattutto per un problema di accesso alla global memory. Nel caso gli heap siano allocati in global memory, durante le operazioni di fusione eseguite in parallelo dai thread, gli accessi a tali strutture dati risulterebbero essere di tipo casuale, senza seguire i requisiti richiesti per permettere un *coalesced access*. Purtroppo però, per $k$ elevato la dimensione degli heap può diventare molto significativa, e ridurre fortemente l'*occupancy* dei multiprocessori, a causa dalla grossa quantità di shared memory richiesta dai blocchi. Sperimentalmente si è riscontrato che per valori di $k$ inferiori a 100 si ottengono comunque risultati migliori allocando gli heap in shared memory, nonostante

l'occupancy molto bassa (intorno addirittura al 6% in una GPU con *compute capability* 2.0, per $k = 100$), a causa dei grossi costi dovuti ad un accesso in global memory non efficiente. Ipotizzando che la GPU sia in grado di eseguire contemporaneamente $m \cdot NT$ thread, la complessità di tale metodo è $O\left(\frac{d}{NT}\log(k)\right)$. Prima della riduzione ogni thread analizza $d/NT$ distanze ed esegue l'operazione $updateMin$ sul proprio heap (ogni $updateMin$ richiede $O(\log(k))$). Vengono poi effettuati $\log_2(NT)$ step di riduzione ed ogni operazione di fusione richiede un tempo $O(k\log(k))$. Di conseguenza la complessità è pari a $O\left(\frac{d}{NT}\log(k) + k \cdot \log(k)\log(NT)\right) \cong O\left(\frac{d}{NT}\log(k)\right)$ in quanto $d/NT \gg k \cdot \log(NT)$, essendo tipicamente $d \gg k, NT$.

La tecnica precedente, per poter essere efficiente, richiede che $m$ sia sufficientemente elevato. Questo in quanto il numero di blocchi di thread è posto uguale ad $m$ e nel caso in cui $m$ sia minore del numero minimo di blocchi richiesti per ottenere la massima occupancy dei multiprocessori, si ha un uso non ottimale della risorsa GPU. Per ovviare a questo problema, abbiamo introdotto un nuovo metodo da noi chiamato *Heap reduction multi-block*. A differenza della tecnica precedente, lavoriamo su di un singolo query point $q_j$ alla volta, predisponendo $NB_A$ blocchi di $NT_A$ thread ciascuno, tutti interamente dedicati a $q_j$. Data in input una matrice delle distanze, ogni thread $T_z$ (con $z = 0,1,\ldots,NB_A \cdot NT_A - 1$) si occupa della selezione delle $k$ minori distanze di $q_j$ con $d/(NB_A \cdot NT_A)$ punti $p_i \in D$, salvando le rispettive coppie $\langle p_i, \sigma \rangle$ (con $\sigma = dist(q_j, p_i)$) in un proprio max-heap $H_z$ (allocato in shared memory) di dimensione $k$, sfruttando l'operazione $updateMin$. Per il modello di esecuzione di CUDA, non è possibile effettuare in maniera efficiente una riduzione parallela *inter-blocco*, a causa della mancanza di primitive di sincronizzazione tra thread appartenenti a blocchi diversi e al fatto che solamente i thread appartenenti allo stesso blocco possono accedere alla shared memory esclusiva di quel blocco. E' stato quindi deciso di spezzare il procedimento in due distinte kernel function. La prima kernel function esegue, per ogni blocco, la fusione degli heap $H_z$ dei thread appartenenti al medesimo blocco, tramite la tecnica della riduzione, e salva il risultante heap di ogni blocco in global memory. La seconda kernel function si occupa quindi di eseguire le operazioni di fusione dei precedenti heap allocati in global memory, sempre tramite la tecnica di riduzione (sfruttando un'altra serie di heap temporanei in shared memory, durante i passi di riduzione). Inoltre, per una maggiore efficienza, si utilizzano $m$ blocchi di

$NT_B$ thread, un blocco per ogni candidato $q_j \in C$, in modo da effettuare le ultime operazioni di fusione in parallelo per gli $m$ query point, anziché eseguirle separatamente per ogni punto (come nella prima kernel function).




**4-6 Metodo *Heap reduction multi-block***

Nella prima fase, se $NB_A$ è sufficientemente elevato, i multiprocessori raggiungono la massima occupancy possibile, anche se si considera un solo query point alla volta. Nella seconda fase, per $m$ piccolo, si ha una minor occupancy, ma essendo la prima fase quella dominante, questo fenomeno non deve incidere particolarmente sulle performance. Ipotizzando che la GPU sia in grado di eseguire contemporaneamente $NB_A \cdot NT_A$ thread, la complessità di tale metodo è $O\left(\frac{dm}{NB_A \cdot NT_A}\log(k)\right)$. La prima kernel function, per le conclusioni effettuate nel metodo precedente, presenta una complessità pari a $O\left(\frac{d}{NB_A \cdot NT_A}\log(k)\right)$, in quanto ogni thread confronta $\frac{d}{NB_A \cdot NT_A}$ distanze. La seconda kernel function ha complessità $O(\frac{NB_A}{NT_B}k \cdot \log(k))$, in quanto ogni thread, prima della riduzione parallela, esegue l'operazione di fusione di $\frac{NB_A}{NT_B}$ heap su di un proprio heap temporaneo in shared memory, di dimensione $k$.

Siccome la prima kernel function viene eseguita $m$ volte, una per query point, la complessità totale dell'intero procedimento è $O\left(\frac{dm}{NB_A \cdot NT_A}\log(k) + \frac{NB_A}{NT_B}k \cdot \log(k)\right) \cong O\left(\frac{dm}{NB_A \cdot NT_A}\log(k)\right)$, in quanto il primo addendo domina nettamente il successivo, per $d$ sufficientemente elevato. Si può osservare che se si utilizza $NB_A = m$, si ottiene la stessa complessità del metodo precedente.

E' possibile applicare una simile variante anche al metodo di *Kato*, per renderlo efficiente anche per $m$ piccolo: chiamiamo questo metodo *Kato multi-block*. Come per la tecnica precedente, spezziamo il procedimento in due kernel function. Nella prima lavoriamo su di un singolo query point $q_j$ alla volta, predisponendo $NB_A$ blocchi di $NT_A$ thread ciascuno, tutti interamente dedicati a $q_j$. Associamo ad ogni blocco un max heap in global memory e, ad ogni thread, un proprio buffer in shared memory. Data in input una matrice delle distanze, ogni thread $T_z$ (con $z = 0,1,\ldots, NB_A \cdot NT_A - 1$) si occupa della selezione delle $k$ minori distanze di $q_j$ con $d/(NB_A \cdot NT_A)$ punti $p_i \in D$ (con $d = |D|$), sfruttando il metodo proposto da Kato. I thread inseriscono nel proprio buffer solamente le coppie $\langle p_z, \delta \rangle$ (con $p_z \in \text{D}$ e $\delta = dist(q_j, p_z)$) tali che $\delta < \sigma$, dove $\langle \text{s}, \sigma \rangle$ è l'elemento radice dell'heap $j$, fino a riempire il buffer. Un solo thread si occupa, in sequenza, di prelevare gli elementi dai vari buffer e di inserirli nell'heap, tramite l'operazione $updateMin$. Il processo viene iterato fino a quando non sono state considerate tutte le distanze. Nella seconda kernel function si utilizzano $m$ blocchi di $NT_B$ thread, un blocco per ogni candidato $q_j \in C$, ed ogni blocco determina i $k$ nearest neighbor di $q_j$, selezionando le prime $k$ distanze tra quelle presenti nell'heap in global memory, sfruttando sempre il medesimo procedimento della precedente kernel function. Ipotizziamo che la GPU sia in grado di eseguire contemporaneamente $NB_A \cdot NT_A$ thread. Anche in questo caso, la complessità temporale è data da quella della prima fase, dominante rispetto alla seconda, che risulta pari a $O\left(\frac{dm}{NB_A}\log(k)\right)$, in quanto il singolo thread di ogni blocco che lavora sull'heap esegue $\frac{d}{NB_A}$ operazioni $updateMin$ e tale kernel function viene ripetuta $m$ volte, una per query point.

Nel caso $m$ sia molto elevato (ad esempio $m \cong d$), è possibile adottare una tecnica molto semplice, che possiamo chiamare *Single heap*. Predisponiamo $m$ thread $T_j$ (con $j = 0,1,\ldots, \text{m} - 1$), ognuno dedicato ad un singolo query point $q_j$ ed associamo ad

ogni thread un proprio max-heap $H_j$ di dimensione $k$, allocato in shared memory. Data in input una matrice delle distanze, ogni thread $T_j$ considera le distanze tra $q_j$ ed ogni altro punto $p_z \in D$ e trova i $k$ nearest neighbor di $q_j$, inserendo le rispettive coppie $\langle p_z, \sigma \rangle$, con $\sigma = dist(q_j, p_z)$, nel proprio heap, sfruttando l'operazione $updateMin$. Siccome si ha un singolo thread $T_j$ dedicato ad ogni query point $q_j$, tale tecnica può essere considerata una sorta di variante ai metodi proposti da *Garcia* e da *Arefin*. Ipotizzando che la GPU sia in grado di eseguire contemporaneamente $m$ thread, la complessità temporale del metodo risulta essere pari a $O(d \cdot \log(k))$.

Infine, l'ultima tecnica che proponiamo è una combinazione del metodo di *Garcia* (con *insert-sort*) con la tecnica della *riduzione parallela*. Definiamo $m$ blocchi di $NT$ thread, dove ogni blocco è dedicato ad un singolo query point $q_j \in C$ e associamo ad ogni thread un array, contenente le $k$ minori coppie $\langle p_z, \sigma \rangle$, con $\sigma = dist(q_j, p_z)$, ordinate in ordine crescente secondo il valore delle distanze. Data in input una matrice delle distanze, ogni *i-esimo* thread del blocco *j* analizza $d/NT$ distanze e seleziona le minori $k$, applicando la tecnica usata da *Garcia* con *insert-sort* sul proprio array. A questo punto, per ogni blocco vengono fusi gli array degli $NT$ thread, sfruttando la tecnica della riduzione parallela. Ipotizzando che la GPU sia in grado di eseguire contemporaneamente $m \cdot NT$ thread, la complessità di tale metodo è $O\left(\frac{d}{NT}k\right)$, in quanto ogni thread, prima della fase di riduzione, analizza $\frac{d}{NT}$ distanze ed ogni inserimento nell'array richiede, nel caso peggiore, $O(k)$.

### 4.3 Confronti teorici e risultati sperimentali

Tutti gli approcci descritti prevedono un procedimento in due fasi, una di calcolo delle distanze ed una di selezione delle rispettive minori $k$ per ogni query point. Per valutare in maniera approfondita quanto le seguenti tecniche GPGPU possano apportare un incremento di performance, rispetto ad una soluzione sequenziale eseguita su CPU, si è deciso di analizzare in maniera separata le due fasi.

Anche in questo caso, come piattaforma di testing si è utilizzato un singolo nodo del supercomputer *IBM PLX* [10]. Ricordiamo che ogni nodo dispone di due *six-cores Intel Westmere* a 2.40 *GHz* (*E5645*), di 48 *GB* di RAM *DDR3* e di due GPU *NVIDIA Tesla M2070*. La GPU *NVIDIA Tesla M2070* è basata sull'*architettura Fermi* ed è classificata con *compute capability 2.0*. Possiede 14 streaming multiprocessor a 1.15 *GHz*, ognuno dotato di 32 CUDA core, per un totale di 448 core. E' in grado di

eseguire 1.03 trilioni di operazioni floating point a precisione singola al secondo (1.03 $TFLOPS$). Possiede inoltre 6 *GB* di memoria dedicata di tipo *GDDR5* (*device memory*), 64$k$ di memoria *on-chip* per ogni SM (16$k$ di cache *L1* e 48$k$ di *shared memory*, a default) e 768$k$ di cache *L2* (condivisa tra tutti gli SM).

```
Device 0: "Tesla M2070"
  CUDA Driver Version:                           4.2
  CUDA Capability Major/Minor version number:    2.0
  Total amount of global memory:                 5375 MBytes (5636554752 bytes)
  (14) Multiprocessors x ( 32) CUDA Cores/MP:   448 CUDA Cores
  GPU Clock rate:                                1147 MHz (1.15 GHz)
  Memory Clock rate:                             1566 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 786432 bytes
  Max Texture Dimension Sizes                    1D=(65536) 2D=(65536,65535) 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(16384) x 2048, 2D=(16384,16384) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Texture alignment:                             512 bytes
  Maximum memory pitch:                          2147483647 bytes
  Concurrent copy and execution:                 Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Concurrent kernel execution:                   Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support enabled:                Yes
  Device is using TCC driver mode:               No
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           20 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

**Figura 4-7 Caratteristiche della GPU *NVIDIA Tesla M2070***

### 4.3.1 Calcolo delle distanze

La fase di calcolo delle distanze avviene in maniera molto simile per tutti i metodi illustrati, essendo un'operazione molto efficiente e semplice da eseguire in parallelo. Le uniche differenze riguardano la possibilità di spezzare la matrice delle distanze in più sezioni, alfine di poter lavorare anche con dataset di dimensioni particolarmente elevate, oppure l'eventuale uso della shared memory, usata semplicemente come cache temporanea per dati intermedi. Per questo motivo, per valutare l'incremento di performance fornito da una soluzione parallela su GPU rispetto ad una soluzione sequenziale per CPU, si è deciso di considerare solamente un semplice metodo di GPGPU, alla base di tutte le tecniche mostrate in precedenza.

Dato un insieme di $d$ reference point $D$ ed un insieme di $m$ query point $C$, si definiscono $d$ thread $T_i$, dove $T_i$ calcola tutte le distanze tra $p_i \in D$ ed ogni $q_j \in C$ e le salva nella corrispondente $i$-esima colonna della matrice delle distanze $[\delta_{j,i}]$ (con $\delta_{j,i} = dist(p_i, q_j)$, di dimensione $m \times d$). Secondo il modello di programmazione di CUDA, fissiamo la dimensione dei blocchi pari ad un valore $NT$ e utilizziamo $NB$

blocchi di $NT$ thread, con $NB = \left\lceil \frac{d}{NT} \right\rceil$. Possiamo utilizzare la seguente kernel function per il calcolo delle distanze (in pseudo-codice):

```
calcDist_Kernel(D, C, dist){
        i := blockIdx.x * blockDim.x + threadIdx.x

        for each q_j in C{
                δ[j][i] := dist(p_i, q_j), con p_i ∈ D
        }
}
```

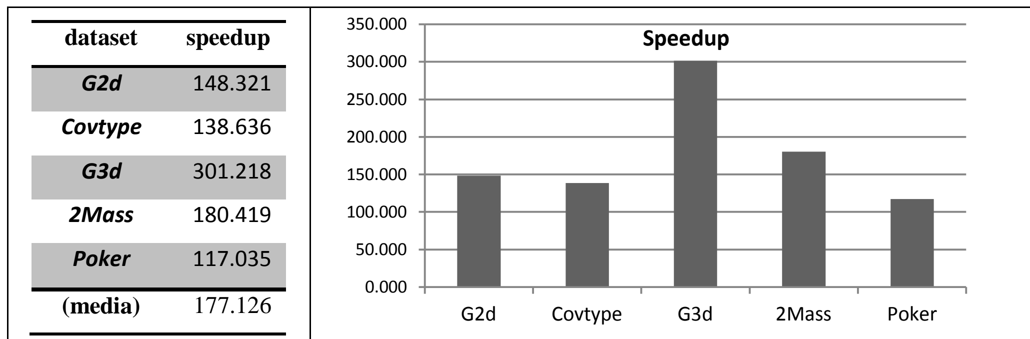
Come corrispondente algoritmo sequenziale per CPU, consideriamo un semplice algoritmo *nested loop* che calcoli in maniera iterativa, per ogni coppia $(p_i, q_j)$, con $p_i \in D$, $q_j \in C$, la rispettiva distanza $dist(p_i, q_j)$ e la salvi nella rispettiva cella della matrice delle distanze $[\delta_{j,i}]$.

```
calcDist_CPU(D, C, dist){
        for each p_i in D {
                for each q_j in C {
                        δ[j][i] := dist(p_i, q_j)
                }
        }
}
```

Ipotizzando che la GPU sia in grado di eseguire contemporaneamente $d$ thread, il metodo parallelo ha una complessità temporale pari a $O(m \cdot a)$, con $a$ pari alla dimensionalità (il numero di coordinate) dei reference point e query point. Rispetto alla soluzione sequenziale proposta, che richiede un tempo $O(d \cdot m \cdot a)$, è possibile scalare in via teorica di un fattore $d$. Da un punto di vista implementativo, relativo al modello di CUDA, è possibile ottenere un'ottima efficienza con questa soluzione. Non essendo richiesta shared memory è possibile ottenere la massima occupancy dei multiprocessori, dimensionando correttamente $NB$ e $NT$. Strutturando la matrice delle distanze e l'insieme dei reference point e query point (allocati in *global* memory) in maniera opportuna, è sempre possibile ottenere un $coalesced\ access$, per ogni accesso in lettura o in scrittura. Inoltre il grado di divergenza dei thread all'interno di uno stesso warp è praticamente nullo (a meno dell'ultimo warp dell'ultimo blocco, in quanto $d$ può non essere esattamente multiplo di $NT$).

Per gli esperimenti, come insieme di reference point, sono stati utilizzati i cinque diversi dataset mostrati nei test di ODP SolvingSet, ovvero *G2d*, *G3d*, *Covtype*, *Poker* e *2Mass*. Come rispettivi insiemi di query point, sono stati selezionati $m = 100$ punti in maniera casuale, appartenenti al corrispondente dataset. L'algoritmo sequenziale per CPU è stato implementato in *Java*. Per quanto riguarda invece l'algoritmo per GPU, è stato realizzato un modulo *CUDA cubin* [15] contenente la kernel fuction

(scritto tramite il linguaggio *CUDA C* e (v. 3.2) e poi compilato). La parte di comunicazione e di coordinamento con la GPU è stata implementata in *Java*, sfruttando la libreria *JCuda* (v. 0.4.2) [24].



| dataset | speedup |
|---|---|
| ***G2d*** | 148.321 |
| ***Covtype*** | 138.636 |
| ***G3d*** | 301.218 |
| ***2Mass*** | 180.419 |
| ***Poker*** | 117.035 |
| **(media)** | 177.126 |



**Figura 4-8 Risultati sperimentali su $calcDist$ ($m$=100)**

Consideriamo lo *speedup* della versione GPU sulla versione CPU, definito come il rapporto tra i tempi di esecuzione della versione GPU e della versione CPU. Tale grandezza presenta una valor medio pari a circa 177, con un minimo di 117 ed un massimo di 301.

Per valutare la bontà di tale risultato, utilizziamo come riferimento il massimo numero teorico di operazioni floating-point a precisione singola al secondo (*FLOPS - FLoating-point Operations Per Second*) che sono in grado di eseguire la CPU e la GPU, utilizzate negli esperimenti. Per quanto riguarda la CPU, un singolo Core della architettura *Intel Westmere* è in grado di eseguire fino a 8 operazioni floating point a precisione singola per ciclo di clock, in quanto dotato di 2 unità *SIMD*, dove ognuna può eseguire fino a 4 operazioni floating point a precisione singola contemporanee, per ciclo di clock [25]. Di conseguenza, un singolo Core dell'*Intel Westmere* (*E5645*) a 2.40 *GHz* è in grado di eseguire fino a 19.2 $GFLOPS = 2.4\ GHz \times 2\ SIMD\ UNIT \times 4\ FLOP$. La GPU *NVIDIA Tesla M2070* è dotata di 448 CUDA core (ovvero delle ALU, non sono da confondere con il concetto di Core delle CPU), ognuno in grado di eseguire fino a 2 operazioni floating-point a precisione singola contemporanee, nel caso di operazioni *micro fuse* (come *multiply-add*). Siccome i CUDA core lavorano a 1.15 *GHz*, la GPU è in grado di eseguire fino a $1.03\ TFLOPS = 1.15\ GHz \times 448\ core\ \times 2\ FLOPS$. Definiamo con $GSU_{bestCPU}$ il rapporto tra il massimo valore teorico dei $FLOPS$ della GPU e il massimo valore teorico dei $FLOPS$ della CPU, ovvero $GSU_{bestCPU} = FLOPS_{GPU}/FLOPS_{CPU}$. Di conseguenza, possiamo affermare che, per valori di speedup di una versione GPU rispetto ad una versione CPU intorno a $GSU_{bestCPU}$, il risultato ottenuto sia buono. In questo specifico caso si ha

$GSU_{bestCPU} = 53.64 = 1.03\ TFLOPS/19.2\ GFLOPS$, ed uno speedup medio di circa 177. C'è comunque da sottolineare che si tratta di valori teorici e che non sempre i compilatori riescono a sfruttare al meglio tutte le risorse. Inoltre l'algoritmo per CPU è stato implementato in *Java*, la *JVM* cerca di sfruttare il pieno parallelismo delle unità *SIMD* quando possibile, ma non sempre riesce ad ottimizzare il codice per poter eseguire contemporaneamente, in modo vettoriale, esattamente 4 operazioni floating point su ogni unità. Nel caso questa operazione di ottimizzazione non avvenga, viene eseguita una sola operazione floating point per unità *SIMD*, ad ogni ciclo di clock. In tal caso il numero di operazioni floating point a precisione singola al secondo scende a $4.8\ GFLOPS = 2.4\ GHz \times 2\ SIMD\ UNIT \times 1\ FLOP$. Definiamo con $GSU_{worstCPU}$ il rapporto tra il massimo valore teorico dei $FLOPS$ della GPU e il massimo valore teorico dei $FLOPS$ della CPU nel caso di una sola operazione floating point per unità *SIMD* al secodo. In questo specifico caso $GSU_{worstCPU} = 214.58 = 1.03\ TFLOPS/4.8\ GFLOPS$. Nel nostro caso si sono ottenuti valori di speedup compresi tra $GSU_{bestCPU}$ e $GSU_{worstCPU}$, a volte pure superiori a $GSU_{worstCPU}$, il che indica che il calcolo delle distanze può essere implementato in maniera molto efficiente con un approccio GPGPU.

### 4.3.2 Selezione delle minori $k$ distanze

La fase di selezione delle minori $k$ distanze per ogni query point, è molto critica e difficile da eseguire in parallelo in maniera davvero efficace, tramite il modello di esecuzione offerto da CUDA. Per tale fase sono state proposte tecniche molto differenti. Alcuni metodi puntano ad ottenere una bassa complessità teorica, ma non riescono a sfruttare appieno le risorse offerte dalla GPU, altri preferiscono cercare di utilizzare al massimo la potenza dell'hardware, a discapito però della complessità delle operazioni eseguite. Inoltre, a seconda dei parametri del problema ($k$,$m$,$d$), alcune tecniche possono essere molto più efficaci di altre.

Per effettuare un confronto, come corrispondente algoritmo sequenziale per CPU consideriamo un semplice algoritmo iterativo, che prenda in input una matrice delle distanze $[\delta_{j,i}]$ e selezioni, per ogni query point $q_j \in C$, le $k$ minori distanze, sfruttando l'operazione $updateMin$ su di un max-heap $NN[q_j]$ associato a $q_j$. Tale algoritmo ha complessità temporale $O(d \cdot m \cdot \log(k))$.

```
kSmallest_CPU(D, C, [δ_{j,i}]){
        NN[q_j] := ∅
        for each q_j in C {
                for each p_i in D {
                        updateMin(NN[q_j], ⟨p_i, δ[j][i]⟩ )
                }
        }
}
```

La seguente tabella riassume brevemente la situazione ed indica, per ogni metodo, la rispettiva complessità temporale e i tre fattori principali che permettono un'efficace implementazione in CUDA.

| **Metodo** | **Complessità temporale teorica** | **Grado di occupancy dei multiprocessori** | **Grado di efficienza negli accessi in global memory (*coalesced acceses*)** | **Grado di divergenza dei thread di un warp** |
|---|---|---|---|---|
| *CPU* | $O(d \cdot m \cdot \log(k))$ | - | - | - |
| *GPU Garcia - Combsort* | $O(d \cdot \log(d))$ (ipotizzando $m$ thread contemporanei) | basso, per *m* piccolo | basso | alto |
| *GPU Garcia - Insertsort* | $O(d \cdot k)$ (ipotizzando $m$ thread contemporanei) | basso, per *m* piccolo | alto | basso |
| *GPU Arefin* | $O(d \cdot k)$ (ipotizzando $m$ thread contemporanei) | basso, per *m* piccolo | alto | basso |
| *GPU Single Heap* | $O(d \cdot \log(k))$ (ipotizzando $m$ thread contemporanei) | basso, per *m* piccolo e per *k* grande (per gli heap in shared mem.) | alto | alto |
| *GPU Radix Sort* | $O\left(m \cdot \frac{d}{NT \cdot NB} \cdot w\right)$, con $w$ = numero di bit per distinguere le $d$ distanze (ipotizzando $NB \cdot NT$ thread contemporanei) | alto | alto | basso |
| *GPU CUKNN* | $O\left(m \cdot NT + \frac{d}{NT} k\right)$ (ipotizzando $d$ thread contemporanei) | alto | alto | basso |
| *GPU Barrientos Reduction* | $O\left(\frac{d}{NT} \log(k)\right)$ (ipotizzando $m \cdot NT$ thread contemporanei) | alto | basso (gli heap del primo passo sono allocate in global memory) | alto |

| | | | | |
|---|---|---|---|---|
| *GPU Kato* | $O(d \cdot \log(k))$ (ipotizzando $m \cdot NT$ thread contemporanei) | alto | alto | basso, durante l'inserimento nel buffer<br>alto, durante l'inserimento nell' heap |
| *GPU Heap Complete Reduction* | $O\left(\frac{d}{NT}\log(k)\right)$ (ipotizzando $m \cdot NT$ thread contemporanei) | basso, per $k$ grande (a causa degli heap in shared memory) | alto | alto |
| *GPU Insertsort Reduction* | $O\left(\frac{d}{NT}k\right)$ (ipotizzando $m \cdot NT$thread contemporanei) | alto | alto | basso |
| *GPU Heap Reduction Multi-block* | $O\left(\frac{dm}{NB_A \cdot NT_A}\log(k)\right)$ (ipotizzando $NB_A \cdot NT_A$thread contemporanei) | basso, per $k$ grande (a causa degli heap in shared memory) | alto | alto |
| *GPU Kato Multi-block* | $O\left(\frac{dm}{NB_A}\log(k)\right)$ (ipotizzando $NB_A \cdot NT_A$thread contemporanei) | alto | alto | basso, durante l'inserimento nel buffer<br>alto, durante l'inserimento nell' heap |

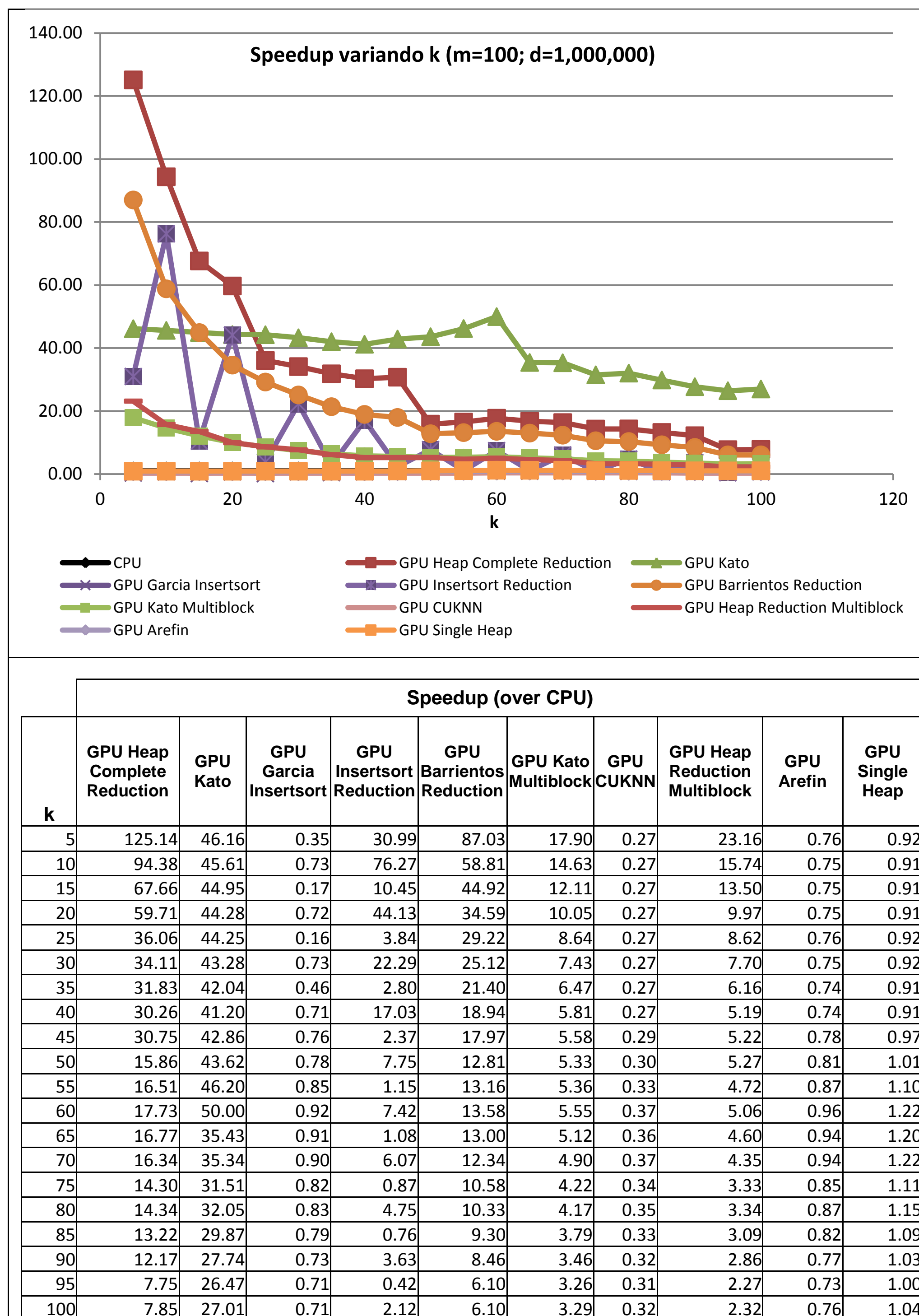
**Tabella 4-1 Confronto complessità e caratteristiche dell'implementazione CUDA dei metodi di selezione delle minori $k$ distanze**

Se consideriamo solamente la complessità computazionale, diversi metodi riescono a scalare di un fattore ideale rispetto alla soluzione sequenziale, come *Barrientos Reduction*, *Heap Complete Reduction, Heap Reduction Multi-block* ed anche *Single Heap*. Se invece consideriamo l'efficacia in una implementazione CUDA (dal punto di vista di massimo sfruttamento dell'hardware), tali metodi però non presentano il massimo che si può ottenere, ovvero alta occupancy dei multiprocessori, alto grado di efficienza negli accessi in global memory e basso grado di divergenza dei thread. Un tale grado di efficienza lo mostrano solamente i metodi *Garcia – Insertsort, Arefin, RadixSort*, *CUKNN* e *Insertsort Reduction*, che però presentano una complessità computazionale più elevata. *Kato* e *Kato Multi-block* possono essere considerati un compromesso. Inoltre *Single Heap*, *Garcia – Insertsort, Garcia – Combsort* e *Arefin* richiedono che $m$ sia estremamente elevato per ottenere la massima occupancy dei multiprocessori. Nel caso specifico della *NVIDIA Tesla M2070*, è necessiario che $m \geq 21{,}504$ (= $14\,SM \times 48\,max\ warp\ per\ SM \times 32\,warp\ size$), per un occupancy del 100%. *Barrientos Reduction*, *Heap Complete Reduction* e *Kato* richiedono anch'essi $m$ sufficientemente elevato per ottenere la massima occupancy, ma è necessario solamente un $m \geq 112 = 14\,SM \times 8\,max\ blocks\ per\ SM$, per

un'occupancy del 100%. Inoltre, a causa dei limiti imposti dal forte uso di shared memory, tipicamente *Barrientos Reduction*, e *Heap Complete Reduction* non possono raggiungere comunque un grado di occupancy 100% (anche se comunque sufficiente a mantenere impegnati tutti gli SM), di conseguenza il valore minimo di $m$ richiesto per raggiungere tale occupancy è inferiore a 112. *Heap Reduction Multi-block, Kato Multi-block, Radix Sort* e *CUKNN* riescono invece ad ottenere sempre la massima occupancy, non dipendendo da $m$ (a meno di limiti imposti da altri fattori, come la quantità di shared memory richiesta).

Per quanto riguarda i risultati sperimentali, sono state eseguite diverse serie di test, per studiare lo speedup delle tecniche parallele sulla soluzione sequenziale, al variare di $k, m$ e $d$. Anche in questo caso l'algoritmo sequenziale per CPU è stato implementato in *Java*, mentre per i metodi GPGPU è stato realizzato un modulo *CUDA cubin* contenente le kernel fuction, con la parte di comunicazione e di coordinamento con la GPU implementata in Java, sfruttando *JCuda*. Dagli esperimenti sono stati esclusi i metodi che effettuano l'ordinamento completo delle $d$ distanze (ovvero *Garcia – Combsort* e *Radixsort*), in quanto tali approcci possono essere significativi solo per valori di $d$ molto bassi, mentre in questo lavoro si focalizza l'attenzione solamente su dataset con un numero molto elevato di istanze. Nei vari test, la matrice delle distanze $m \times d$ in input è stata generata in maniera casuale, con valori di tipo floating point a precisione singola appartenenti all'intervallo $[0, 10]$. Per ogni batteria di test, sono stati effettuati 16 esperimenti utilizzando 4 diversi random seed per la generazione della matrice (con 4 ripetizioni per ogni random seed) e come valori di riferimento per i confronti si è presa la media dei tempi dei 16 esperimenti.

Per studiare le performance al variare di $k$, si è posto $d = 1{,}000{,}000$, $m = 100$ ed il parametro $k$ è stato fatto variare da 5 a 100, con incrementi di 5.



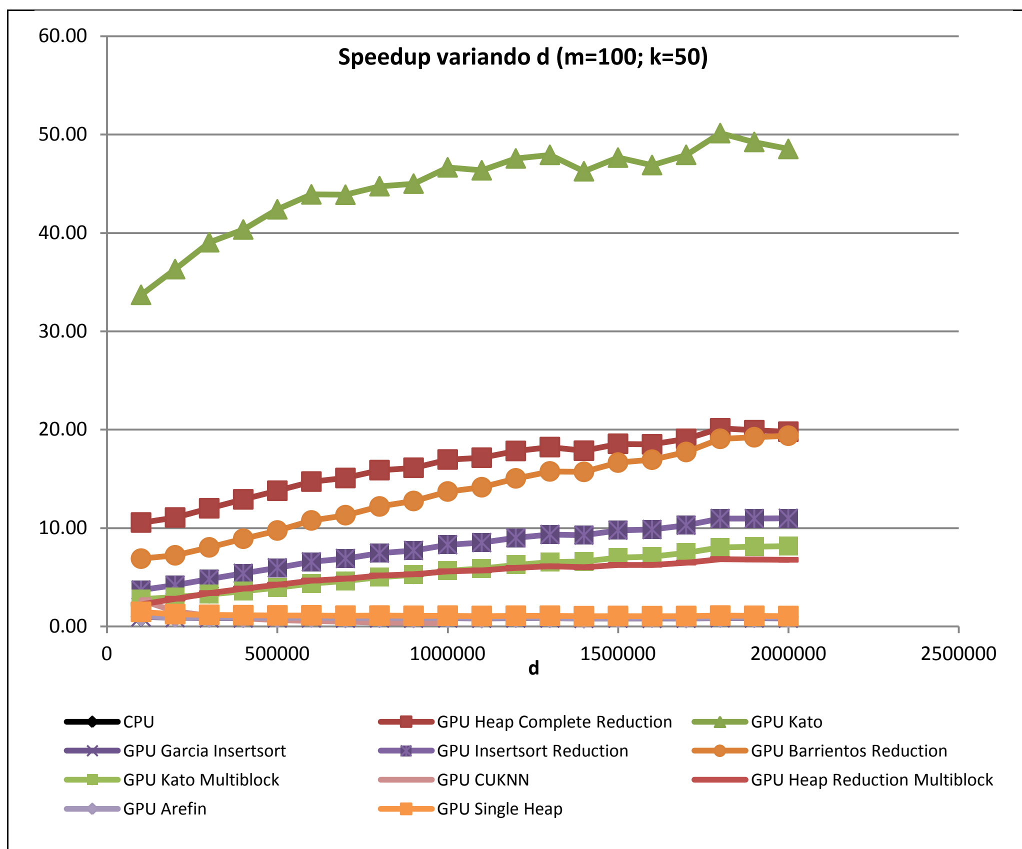

| k | Speedup (over CPU) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | GPU Heap Complete Reduction | GPU Kato | GPU Garcia Insertsort | GPU Insertsort Reduction | GPU Barrientos Reduction | GPU Kato Multiblock | GPU CUKNN | GPU Heap Reduction Multiblock | GPU Arefin | GPU Single Heap |
| 5 | 125.14 | 46.16 | 0.35 | 30.99 | 87.03 | 17.90 | 0.27 | 23.16 | 0.76 | 0.92 |
| 10 | 94.38 | 45.61 | 0.73 | 76.27 | 58.81 | 14.63 | 0.27 | 15.74 | 0.75 | 0.91 |
| 15 | 67.66 | 44.95 | 0.17 | 10.45 | 44.92 | 12.11 | 0.27 | 13.50 | 0.75 | 0.91 |
| 20 | 59.71 | 44.28 | 0.72 | 44.13 | 34.59 | 10.05 | 0.27 | 9.97 | 0.75 | 0.91 |
| 25 | 36.06 | 44.25 | 0.16 | 3.84 | 29.22 | 8.64 | 0.27 | 8.62 | 0.76 | 0.92 |
| 30 | 34.11 | 43.28 | 0.73 | 22.29 | 25.12 | 7.43 | 0.27 | 7.70 | 0.75 | 0.92 |
| 35 | 31.83 | 42.04 | 0.46 | 2.80 | 21.40 | 6.47 | 0.27 | 6.16 | 0.74 | 0.91 |
| 40 | 30.26 | 41.20 | 0.71 | 17.03 | 18.94 | 5.81 | 0.27 | 5.19 | 0.74 | 0.91 |
| 45 | 30.75 | 42.86 | 0.76 | 2.37 | 17.97 | 5.58 | 0.29 | 5.22 | 0.78 | 0.97 |
| 50 | 15.86 | 43.62 | 0.78 | 7.75 | 12.81 | 5.33 | 0.30 | 5.27 | 0.81 | 1.01 |
| 55 | 16.51 | 46.20 | 0.85 | 1.15 | 13.16 | 5.36 | 0.33 | 4.72 | 0.87 | 1.10 |
| 60 | 17.73 | 50.00 | 0.92 | 7.42 | 13.58 | 5.55 | 0.37 | 5.06 | 0.96 | 1.22 |
| 65 | 16.77 | 35.43 | 0.91 | 1.08 | 13.00 | 5.12 | 0.36 | 4.60 | 0.94 | 1.20 |
| 70 | 16.34 | 35.34 | 0.90 | 6.07 | 12.34 | 4.90 | 0.37 | 4.35 | 0.94 | 1.22 |
| 75 | 14.30 | 31.51 | 0.82 | 0.87 | 10.58 | 4.22 | 0.34 | 3.33 | 0.85 | 1.11 |
| 80 | 14.34 | 32.05 | 0.83 | 4.75 | 10.33 | 4.17 | 0.35 | 3.34 | 0.87 | 1.15 |
| 85 | 13.22 | 29.87 | 0.79 | 0.76 | 9.30 | 3.79 | 0.33 | 3.09 | 0.82 | 1.09 |
| 90 | 12.17 | 27.74 | 0.73 | 3.63 | 8.46 | 3.46 | 0.32 | 2.86 | 0.77 | 1.03 |
| 95 | 7.75 | 26.47 | 0.71 | 0.42 | 6.10 | 3.26 | 0.31 | 2.27 | 0.73 | 1.00 |
| 100 | 7.85 | 27.01 | 0.71 | 2.12 | 6.10 | 3.29 | 0.32 | 2.32 | 0.76 | 1.04 |

**Figura 4-9 Risultati sperimentali su selezione minori $k$ distanze variando $k$ ($m$=100; $d$=1,000,00)**

Dai risultati ottenuti appare evidente che si hanno grosse differenze prestazionali tra i vari metodi. Per $k \leq 20$, *Heap Complete Reduction* e *Barrientos Reduction* riescono a fornire le migliori prestazioni, che però tendono fortemente a degradare con l'aumento di $k$. Dei due metodi il primo risulta sempre più efficiente del secondo. Per $k$ più elevato, la tecnica di *Kato* risulta essere in assoluto la migliore. Inoltre lo speedup

fornito da tale soluzione tende a rimanere abbastanza costante, presentando una caduta prestazionale molto meno marcata rispetto ai precedenti due metodi, all'aumentare di $k$. Al quarto posto si trova *Insertsort Reduction*, con prestazioni altalenanti, ma comunque mediamente inferiori ai primi tre metodi. Questo approccio permette di avere un minor grado di divergenza dei thread rispetto a quelli basati su heap, però il fatto di avere una complessità più elevata nell'inserimento dell'array ($O(k)$ contro $O(\log(k))$ su heap) si riflette a livello prestazionale. Si trovano poi Heap *Reduction Multi-block* e *Kato Multi-block*, che forniscono prestazioni inferiori ai corrispettivi metodi con numero fisso di blocchi pari ad $m$. Si hanno infine, con prestazioni simili o perfino inferiori a quelle della CPU, gli altri metodi. *Single Heap*, *Garcia – Insertsort,* e *Arefin* risentono fortemente del fatto di richiedere un parametro $m$ estremamente elevato, per ottenere un grado sufficiente di occupancy. *CUKNN* è il metodo che risulta più lento; anche se gode di un'ottima efficienza nell'implementazione CUDA, ha una complessità molto elevata, che porta a prestazioni estremamente basse.

Per studiare le performance al variare del numero $d$ di reference point, si è posto $m = 100$, $k = 50$ ed il valore di $d$ è stato fatto variare da 100,000 a 2,000,000, con incrementi di 100,000.

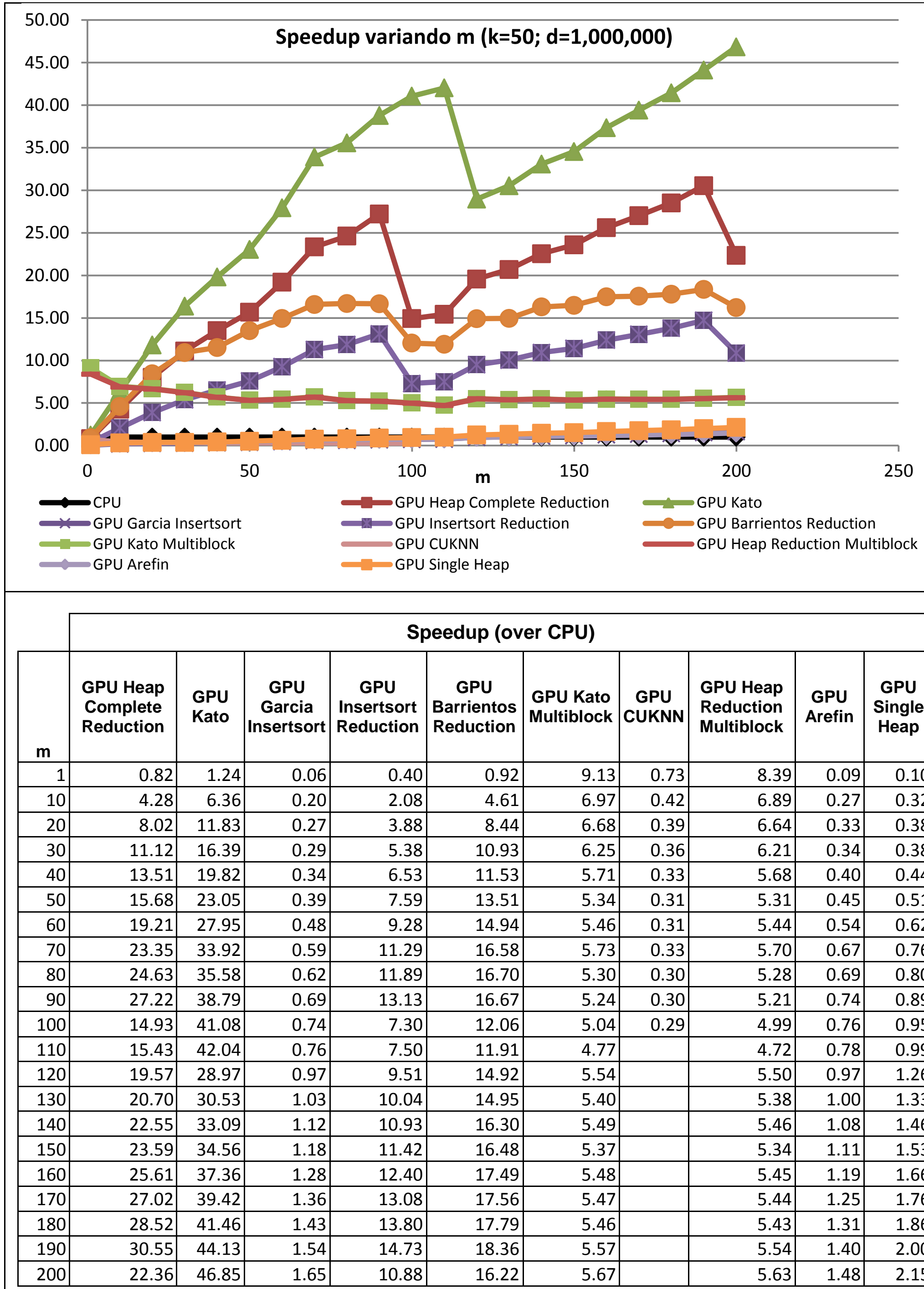


| | Speedup (over CPU) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| d | GPU Heap Complete Reduction | GPU Kato | GPU Garcia Insertsort | GPU Insertsort Reduction | GPU Barrientos Reduction | GPU Kato Multi block | GPU CUKNN | GPU Heap Reduction Multi block | GPU Arefin | GPU Single Heap |
| 100000 | 10.55 | 33.70 | 0.96 | 3.69 | 6.88 | 2.77 | 2.68 | 2.23 | 1.02 | 1.47 |
| 200000 | 11.06 | 36.31 | 0.87 | 4.18 | 7.22 | 2.98 | 1.52 | 2.81 | 0.91 | 1.24 |
| 300000 | 12.00 | 39.05 | 0.85 | 4.80 | 8.03 | 3.26 | 1.07 | 3.38 | 0.89 | 1.17 |
| 400000 | 12.90 | 40.34 | 0.84 | 5.39 | 8.91 | 3.58 | 0.80 | 3.84 | 0.87 | 1.13 |
| 500000 | 13.78 | 42.39 | 0.83 | 5.95 | 9.74 | 3.92 | 0.65 | 4.24 | 0.86 | 1.11 |
| 600000 | 14.72 | 43.91 | 0.84 | 6.56 | 10.76 | 4.37 | 0.56 | 4.66 | 0.89 | 1.11 |
| 700000 | 15.08 | 43.87 | 0.83 | 6.90 | 11.30 | 4.63 | 0.47 | 4.85 | 0.85 | 1.08 |
| 800000 | 15.88 | 44.74 | 0.83 | 7.43 | 12.19 | 5.01 | 0.40 | 5.17 | 0.87 | 1.09 |
| 900000 | 16.12 | 45.00 | 0.82 | 7.71 | 12.74 | 5.24 | 0.37 | 5.29 | 0.85 | 1.07 |
| 1000000 | 16.96 | 46.66 | 0.84 | 8.30 | 13.71 | 5.67 | 0.33 | 5.60 | 0.87 | 1.08 |
| 1100000 | 17.14 | 46.36 | 0.82 | 8.51 | 14.14 | 5.89 | | 5.69 | 0.84 | 1.05 |
| 1200000 | 17.83 | 47.56 | 0.84 | 9.00 | 15.04 | 6.26 | | 5.95 | 0.87 | 1.08 |
| 1300000 | 18.22 | 47.93 | 0.84 | 9.34 | 15.75 | 6.53 | | 6.10 | 0.86 | 1.08 |
| 1400000 | 17.85 | 46.26 | 0.80 | 9.27 | 15.71 | 6.59 | | 6.01 | 0.82 | 1.03 |
| 1500000 | 18.56 | 47.65 | 0.82 | 9.77 | 16.65 | 6.98 | | 6.25 | 0.85 | 1.05 |
| 1600000 | 18.50 | 46.89 | 0.81 | 9.84 | 16.94 | 7.11 | | 6.25 | 0.87 | 1.03 |
| 1700000 | 19.06 | 47.93 | 0.82 | 10.28 | 17.72 | 7.49 | | 6.47 | 0.84 | 1.05 |
| 1800000 | 20.13 | 50.14 | 0.85 | 10.96 | 19.06 | 8.03 | | 6.84 | 0.88 | 1.09 |
| 1900000 | 19.93 | 49.24 | 0.83 | 10.95 | 19.22 | 8.09 | | 6.80 | 0.86 | 1.06 |
| 2000000 | 19.78 | 48.53 | 0.82 | 10.98 | 19.38 | 8.17 | | 6.76 | 0.87 | 1.04 |

**Figura 4-10 Risultati sperimentali su selezione minori $k$ distanze variando $d$ ($m$=100; $k$=50)**

*Kato* risulta essere nettamente superiore a tutti gli altri metodi, per ogni valore di $d$ nell'intervallo utilizzato. Si susseguono poi *Heap Complete Reduction*, *Barrientos Reduction* e tutte le altre soluzioni, con lo stesso ordine della serie precedente di test. Per tutti i metodi, tranne che per *CUKNN*, si osserva un incremento dello speedup all'aumentare di $d$, più marcato in *Kato*. Le prestazioni di *CUKNN* non soltanto tendono a decrescere all'aumentare di $d$, ma lo rendono perfino non utilizzabile per $d > 1{,}000{,}000$, a causa di vincoli sul numero dei blocchi di thread e sulla dimensione di tali blocchi, che dipendono necessariamente da $d$.

Per studiare le performance al variare del numero $m$ di query point, sono state eseguite due differenti serie di test, la prima per valori relativamente bassi di $m$ ($10 \leq m \leq 200$), la seconda per valori di $m$ molto elevati, ponendo $m \equiv d$. Nella prima serie si è posto $d = 1{,}000{,}000$, $k = 50$ e il parametro $m$ è stato fatto variare da 10 a 200, con incrementi di 10.



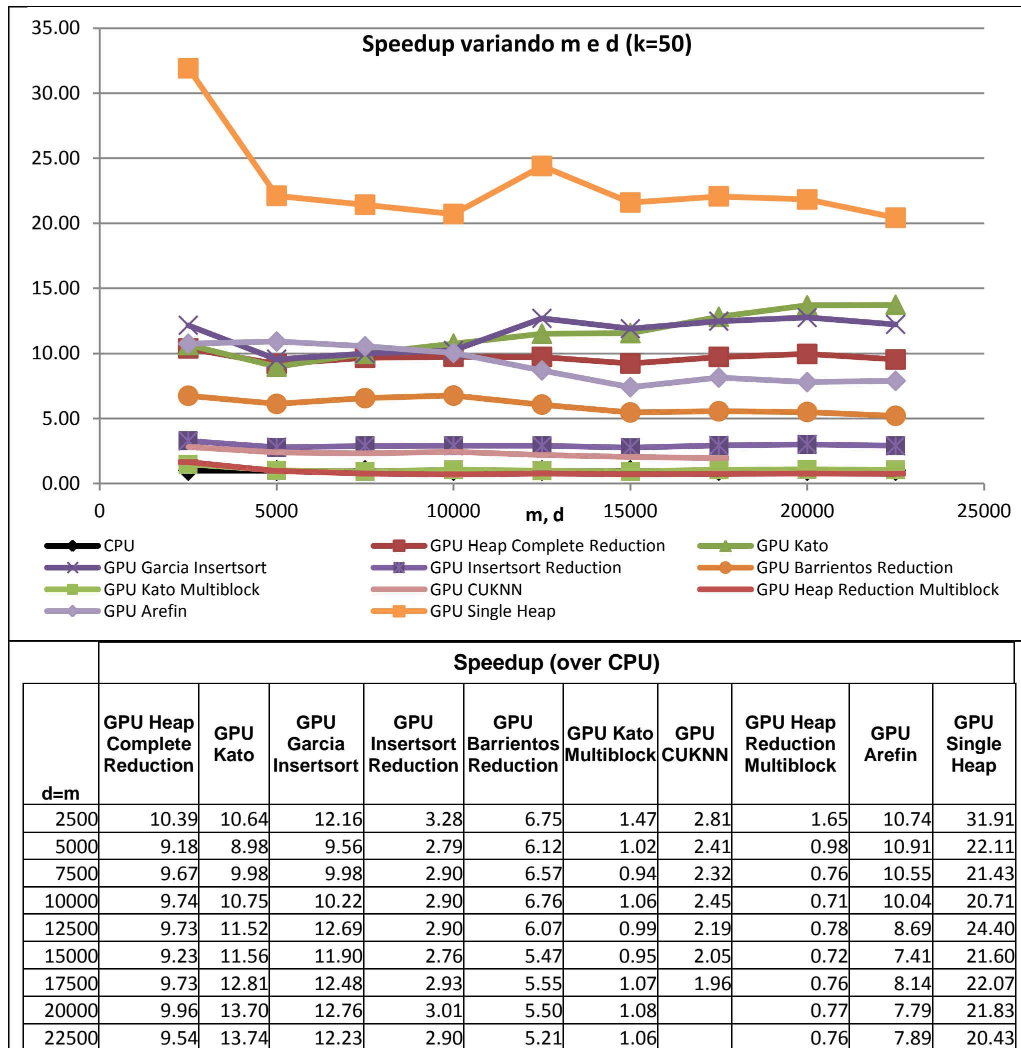

| m | Speedup (over CPU) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | GPU Heap Complete Reduction | GPU Kato | GPU Garcia Insertsort | GPU Insertsort Reduction | GPU Barrientos Reduction | GPU Kato Multiblock | GPU CUKNN | GPU Heap Reduction Multiblock | GPU Arefin | GPU Single Heap |
| 1 | 0.82 | 1.24 | 0.06 | 0.40 | 0.92 | 9.13 | 0.73 | 8.39 | 0.09 | 0.10 |
| 10 | 4.28 | 6.36 | 0.20 | 2.08 | 4.61 | 6.97 | 0.42 | 6.89 | 0.27 | 0.32 |
| 20 | 8.02 | 11.83 | 0.27 | 3.88 | 8.44 | 6.68 | 0.39 | 6.64 | 0.33 | 0.38 |
| 30 | 11.12 | 16.39 | 0.29 | 5.38 | 10.93 | 6.25 | 0.36 | 6.21 | 0.34 | 0.38 |
| 40 | 13.51 | 19.82 | 0.34 | 6.53 | 11.53 | 5.71 | 0.33 | 5.68 | 0.40 | 0.44 |
| 50 | 15.68 | 23.05 | 0.39 | 7.59 | 13.51 | 5.34 | 0.31 | 5.31 | 0.45 | 0.51 |
| 60 | 19.21 | 27.95 | 0.48 | 9.28 | 14.94 | 5.46 | 0.31 | 5.44 | 0.54 | 0.62 |
| 70 | 23.35 | 33.92 | 0.59 | 11.29 | 16.58 | 5.73 | 0.33 | 5.70 | 0.67 | 0.76 |
| 80 | 24.63 | 35.58 | 0.62 | 11.89 | 16.70 | 5.30 | 0.30 | 5.28 | 0.69 | 0.80 |
| 90 | 27.22 | 38.79 | 0.69 | 13.13 | 16.67 | 5.24 | 0.30 | 5.21 | 0.74 | 0.89 |
| 100 | 14.93 | 41.08 | 0.74 | 7.30 | 12.06 | 5.04 | 0.29 | 4.99 | 0.76 | 0.95 |
| 110 | 15.43 | 42.04 | 0.76 | 7.50 | 11.91 | 4.77 | | 4.72 | 0.78 | 0.99 |
| 120 | 19.57 | 28.97 | 0.97 | 9.51 | 14.92 | 5.54 | | 5.50 | 0.97 | 1.26 |
| 130 | 20.70 | 30.53 | 1.03 | 10.04 | 14.95 | 5.40 | | 5.38 | 1.00 | 1.33 |
| 140 | 22.55 | 33.09 | 1.12 | 10.93 | 16.30 | 5.49 | | 5.46 | 1.08 | 1.46 |
| 150 | 23.59 | 34.56 | 1.18 | 11.42 | 16.48 | 5.37 | | 5.34 | 1.11 | 1.53 |
| 160 | 25.61 | 37.36 | 1.28 | 12.40 | 17.49 | 5.48 | | 5.45 | 1.19 | 1.66 |
| 170 | 27.02 | 39.42 | 1.36 | 13.08 | 17.56 | 5.47 | | 5.44 | 1.25 | 1.76 |
| 180 | 28.52 | 41.46 | 1.43 | 13.80 | 17.79 | 5.46 | | 5.43 | 1.31 | 1.86 |
| 190 | 30.55 | 44.13 | 1.54 | 14.73 | 18.36 | 5.57 | | 5.54 | 1.40 | 2.00 |
| 200 | 22.36 | 46.85 | 1.65 | 10.88 | 16.22 | 5.67 | | 5.63 | 1.48 | 2.15 |

**Figura 4-11 Risultati sperimentali su selezione minori $k$ distanze variando $m$ ($k$=50; $d$=1,000,00)**

Per $m \leq 10$, i due metodi *Heap Reduction Multiblock* e *Kato Multiblock* risultano essere i più performanti, anche se con un valore di speedup relativamente basso, rispetto a quello massimo di 46.85 registrato in questi esperimenti. All'aumentare di $m$, i metodi *Kato, Heap Complete Reduction*, *Barrientos Reduction* e *Insertsort Reduction* superano le prestazioni dei primi due. *Kato* risulta sempre essere il metodo che offre le prestazioni migliori, per $m \geq 20$. Si può notare che tutti i metodi che

richiedono un numero fisso di blocchi pari ad $m$ mostrano un andamento a scaletta, con la presenza di picchi. Il primo picco si ha in corrispondenza del valore minimo di $m$ che permette di ottenere la massima occupancy possibile in tutti i multiprocessori ($m$=112 per *Kato; m*=98 per *Heap Complete Reduction* e *Barrientos Reduction*), i successivi picchi si presentano ad intervalli successivi, sempre di tale valore. Questo in quanto solamente in corrispondenza di tali picchi si ottiene la massima occupancy e una distribuzione completamente uniforme dei vari blocchi su tutti i multiprocessori. *Heap Reduction Multiblock* e *Kato Multiblock* non presentano tale fenomeno in quanto il numero di blocchi di thread non è dipendente da *m,* ma è scelto in modo tale da garantire sempre la massima occupancy di tutti i multiprocessori.

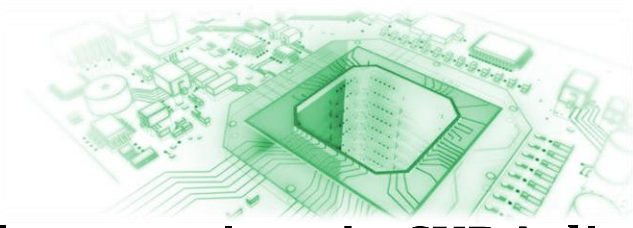
Nella ultima serie di esperimenti si è posto $k = 50$ ed $m \equiv d$, facendo variare tali valori da 2,500 a 22,500, con incrementi di 2,500.




| | Speedup (over CPU) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| d=m | GPU Heap Complete Reduction | GPU Kato | GPU Garcia Insertsort | GPU Insertsort Reduction | GPU Barrientos Reduction | GPU Kato Multiblock | GPU CUKNN | GPU Heap Reduction Multiblock | GPU Arefin | GPU Single Heap |
| 2500 | 10.39 | 10.64 | 12.16 | 3.28 | 6.75 | 1.47 | 2.81 | 1.65 | 10.74 | 31.91 |
| 5000 | 9.18 | 8.98 | 9.56 | 2.79 | 6.12 | 1.02 | 2.41 | 0.98 | 10.91 | 22.11 |
| 7500 | 9.67 | 9.98 | 9.98 | 2.90 | 6.57 | 0.94 | 2.32 | 0.76 | 10.55 | 21.43 |
| 10000 | 9.74 | 10.75 | 10.22 | 2.90 | 6.76 | 1.06 | 2.45 | 0.71 | 10.04 | 20.71 |
| 12500 | 9.73 | 11.52 | 12.69 | 2.90 | 6.07 | 0.99 | 2.19 | 0.78 | 8.69 | 24.40 |
| 15000 | 9.23 | 11.56 | 11.90 | 2.76 | 5.47 | 0.95 | 2.05 | 0.72 | 7.41 | 21.60 |
| 17500 | 9.73 | 12.81 | 12.48 | 2.93 | 5.55 | 1.07 | 1.96 | 0.76 | 8.14 | 22.07 |
| 20000 | 9.96 | 13.70 | 12.76 | 3.01 | 5.50 | 1.08 | | 0.77 | 7.79 | 21.83 |
| 22500 | 9.54 | 13.74 | 12.23 | 2.90 | 5.21 | 1.06 | | 0.76 | 7.89 | 20.43 |

**Figura 4-12 Risultati sperimentali su selezione minori *k* distanze variando *m* e *d* (*m*=*d*; *k*=50)**

In questa batteria di test, il metodo *Single Heap* risulta essere in assoluto il più performante. Gli altri due metodi che dedicano un singolo thread ad ogni query point, *Garcia – Insertsort* e *Arefin*, presentano prestazioni inferiori, paragonabili a quelle di *Kato* e di *Heap Complete Reduction*. E' interessante osservare che per $m$ molto elevato i metodi *Heap Reduction Multiblock* e *Kato Multiblock* presentano performance molto negative, pari o inferiori a quelli della CPU.



# Capitolo 5: Implementazione in CUDA di ODP Nested Loop

Per poter implementare l'algoritmo *ODPNestedLoop* in maniera efficiente in un'architettura *CUDA*, è necessario apportare una netta ristrutturazione, al fine di permettere un'esecuzione parallela. L'algoritmo deve essere suddiviso in diverse fasi, ognuna di esse formata da molteplici thread. I thread vengono eseguiti in parallelo sui diversi multiprocessori della GPU, tramite il supporto fornito da CUDA. Compito della CPU è invece il coordinamento dell'esecuzione delle diverse fasi sulla GPU.

In maniera simile agli approcci GPGPU per il *k-NN problem* (cap. 4), è necessaria una prima fase di calcolo delle distanze, seguita da una successiva fase di selezione delle minori $k$ distanze per ogni punto del dataset, al fine di determinare gli *anomaly score*. L'ultimo passo consiste nel determinare i primi $n$ punti dotati del maggior *anomaly score*.

La prima fase ha come obiettivo la creazione di una matrice delle distanze $[\delta_{j,i}]$, contenente le distanze tra ogni coppia di punti del dataset $D$, ovvero $\delta_{j,i} = dist(p_i, q_j)$, con $p_i, q_j \in D$. Siccome tale matrice ha dimensione $d \times d$ (con $d = |D|$) e nel nostro scenario di riferimento $d$ è molto elevato, non è possibile allocare nella global memory della GPU l'intera matrice. Ad esempio, per $d = 1.000.000$, sarebbe richiesto più di 1 TB di memoria. Per poter ovviare a questo problema, si può adottare la tecnica utilizzata da *Kato* in [21]. La matrice viene spezzata in più sottomatrici, che prendono il nome di *chunk*. Indichiamo con $CSIZE$ la dimensione dei lati dei chunk; di conseguenza ogni chunk avrà dimensione pari a $CSIZE \times CSIZE$. Il chunk $\Delta_{J,I}$ di $[\delta_{j,j}]$ è la sottomatrice di $[\delta_{j,i}]$ che ha come elementi i $\delta_{j,i}$ tali che $CSIZE \cdot J \leq j < CSIZE \cdot (J+1)$ e $CSIZE \cdot I \leq i < CSIZE \cdot (I+1)$. L'idea è quella di generare e mantenere in global memory un solo chunk alla volta, dimensionando $CSIZE$ in maniera opportuna.

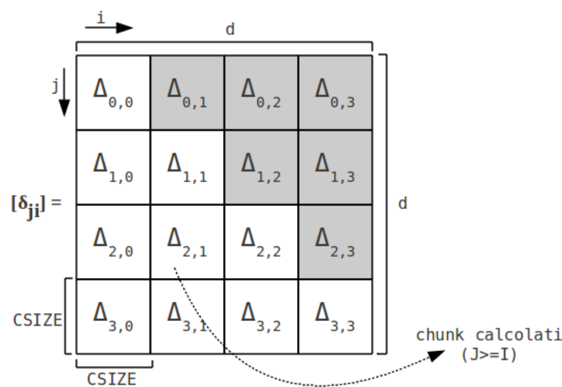



**Figura 5-1 Suddivisione della matrice delle distanze in chunk**

Associamo ad ogni punto $p \in D$ un max-heap $NN[p]$, i cui elementi siano le coppie $\langle q, \delta \rangle$, con $q \in D$ e $\delta = dist(p, q)$. Per ogni sezione della matrice delle distanze, si alterna una fase di generazione del chunk ed una fase di selezione delle minori $k$ distanze. In particolare, dato un chunk $\Delta_{J,I}$, per ogni punto $q_j$ con $CSIZE \cdot J \leq j < CSIZE \cdot (J + 1)$, si effettua l'operazione di $updateMin$ sul rispettivo max-heap $NN[q_j]$, per ogni distanza contenuta nella rispettiva riga di $q_j$ nel chunk. Una volta processati tutti i chunk, ogni $NN[p]$ contiene i $k$ nearest neighbor del punto $p \in D$. Essendo la matrice delle distanze simmetrica, è possibile sfruttare questa proprietà e generare solamente i chunk $\Delta_{J,I}$ con $J \geq I$. Dato un chunk $\Delta_{J,I}$, il rispettivo chunk $\Delta_{I,J}$ è corrispondente alla trasposta di $\Delta_{J,I}$, ovvero $\Delta_{I,J} = {\Delta_{J,I}}^T$. E' quindi sufficiente calcolare i chunk $\Delta_{J,I}$ con $J \geq I$, per ognuno di essi eseguire la fase di selezione delle minori $k$ distanze e, se $J > I$, eseguire solo quest'ultima fase anche per ${\Delta_{J,I}}^T$. Infine, per ogni punto $p \in D$, si calcola l'anomaly score come somma delle rispettive minori $k$ distanze salvate in $NN[p]$ e si selezionano i primi $n$ punti dotati del maggior punteggio.

Ad ogni fase dell'algoritmo corrispondono una o più kernel function, eseguite sulla GPU. La CPU ha invece il compito di caricare il dataset nella global memory della GPU, coordinare l'esecuzione delle kernel function sulla GPU e recuperare infine i risultati.

La procedura eseguita dalla CPU, è la seguente (in pseudo-codice):

```
CUDA_ODPNestedLoop(D, dist, n, k){
        allocateToGPU(D) //contenente il dataset
        allocateToGPU(NN) //contenente i max-heap NN[p] ∀ p ∈ D
        allocateToGPU(currentChunk) //contenente il chunk corrente
        allocateToGPU(Top) //min-heap con i top outlier
        allocateToGPU(GHMin) //contenente i min-heap temporanei
                //per findTopOutliers_phaseA, findTopOutliers_phaseB
        copyToGPU(D)
        numOfChunks := ⌈d/CSIZE⌉
        for I := 0 to numOfChunks{
                for J := I to numOfChunks {
                        launchOnGPU(calcDist_chunk(D, dist, J, I, currentChunk))
                        synchWithGPU()
                        launchOnGPU(k_smallest_*(k, NN, J, I, currentChunk))
                                //dipendente dalla tecnica adottata
                        synchWithGPU()
                        if(J > I) {
                                launchOnGPU(transponse(currentChunk))
                                synchWithGPU()

                launchOnGPU(k_smallest(k, NN, I, J, currentChunk))
                                synchWithGPU()
                        }
                }
        }
        launchOnGPU(findTopOutliers_phaseA(n, NN, GHMin))
        synchWithGPU()
        launchOnGPU(findTopOutliers_phaseB(n, GHMin, Top))
        synchWithGPU()
        copyFromGPU(Top)
}
```

### 5.1 Calcolo delle distanze

La fase di calcolo delle distanze, per il chunk $\Delta_{J,I}$, viene eseguita dalla GPU tramite la kernel function $calcDist_chunk()$. Il chunk viene diviso orizzontalmente in un insieme di segmenti, ognuno contenente $BSIZE$ righe di $\Delta_{J,I}$, e ad ogni segmento viene associato un blocco di thread. I thread del blocco si occupano del calcolo delle distanze per ogni coppia di punti corrispondente al segmento e del salvataggio delle distanze nelle rispettive celle del chunk. Per rendere più efficiente tale operazione, viene sfruttata la shared memory dei multiprocessori, in cui viene caricata temporaneamente una porzione dei punti del dataset.

Sia $SSIZE$ il numero di punti caricati da ogni blocco in shared memory, $X_I$ l'insieme dei punti in $D$ corrispondenti alle colonne del chunk $\Delta_{J,I}$, ovvero $X_I = \{p_{i_idx} \in D \mid CSIZE \cdot I \leq i_idx < CSIZE \cdot (I+1)\}$ e $Y_J$ l'insieme dei punti in $D$ corrispondenti alle righe del chunk $\Delta_{J,I}$, ovvero $Y_J = \{q_{j_idx} \in D \mid CSIZE \cdot J \leq j_idx < CSIZE \cdot (J+1)\}$. Per ogni blocco $B_z$ si definiscono $BSIZE \times SSIZE$ thread $T_{i,j}$, con $i = 0, \dots, SSIZE - 1$ e $j = 0, \dots, BSIZE - 1$, organizzati su due dimensioni. Dopo aver caricato in shared memory i primi $SSIZE$ punti di $X_I$, ogni thread $T_{i,j}$ di $B_z$ calcola la distanza tra

$q_{j^*} \in Y_J$ (con $j^* = z * BSIZE + j$) e il punto $p_i \in X_I$ in shared memory e salva tale valore nella corrispettiva cella del chunk. Vengono quindi caricati in shared memory i successivi $SSIZE$ punti di $X_I$ e ripetuta l'operazione fino a quando vengono considerati tutti i punti $p_i \in X_I$.

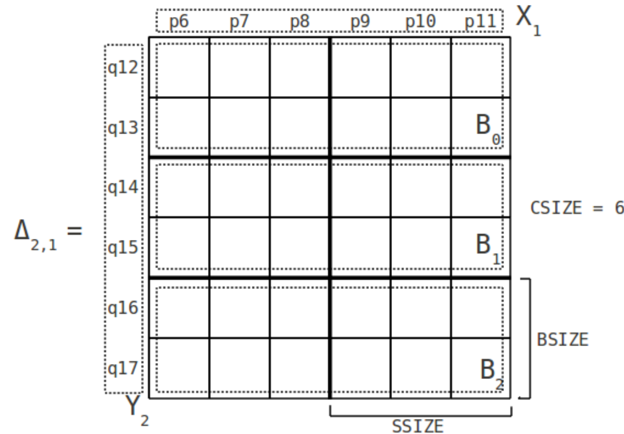



**Figura 5-2 Fase di calcolo delle distanze del chunk**

La procedura eseguita dalla kernel function è la seguente (in pseudo-codice):

```
calcDist_chunk(D, dist, J, I, currentChunk){
        j := threadIdx.y
        i := threadIdx.x
        bid := blockIdx.x
        j_idx := J * CSIZE + bid * BSIZE + j
        i_idx := I * CSIZE + i
        <shared> s_0, s_1,..., s_{SSIZE-1}  //s_0, s_1,..., s_{SSIZE-1} in shared memory

        while(i < CSIZE AND i_idx < d){
                if (j = 0) {
                        s_i := p_{i_idx}, con p_{i_idx} ∈ D //copia in shared mem.
                }
                __syncthreads()
                currentChunk[bid * BSIZE + j][i] := dist(s_i, q_{j_idx}), con q_{j_idx} ∈ D
                i := i + SSIZE
                i_idx := i_idx + SSIZE
        }
}
```

Sperimentalmente si è trovato che le migliori performance si hanno dimensionando $CSIZE$ con il maggior possibile valore che permette di allocare l'intero chunk in global memory. Inoltre, per favorire un *coalesced access* al contenuto del chunk, è opportuno che $CSIZE$ sia un multiplo di 32. $BSIZE$ e $SSIZE$ devono essere scelti in modo da garantire la massima occupancy dei multiprocessori e devono inoltre essere divisori di $CSIZE$.

## 5.2 Selezione delle minori $k$ distanze

Per la scelta della tecnica da adottare per la selezione delle minori $k$ distanze, possiamo effettuare un paragone con le tecniche GPGPU per il *kNN problem* (cap. 4). Dato un chunk $\Delta_{J,I}$, possiamo considerare $X_I$ come l'insieme dei *reference point* e $Y_J$ come l'insieme dei *query point*. Sotto tali ipotesi, abbiamo che la cardinalità $m^*$ dell'insieme dei query point e la cardinalità $d^*$ dell'insieme dei reference point sono entrambe pari a $CSIZE$, dove $CSIZE$ è un valore sufficientemente elevato. Nel caso di $m^*$ elevato, con $m^* \cong d$, il metodo per il *kNN problem* che si è rivelato più efficace è stato *single heap*. Per valori di $m^*$ inferiori (con $m \cong 100, 200$), le tecniche più performanti sono state quella di $Kato$ (per $k$ elevato) e *heap complete reduction* (per $k$ piccolo). Per la selezione delle minori $k$ distanze nell'algoritmo parallelo CUDA_ODPNestedLoop, abbiamo quindi deciso di implementare le tre tecniche, tramite tre distinte kernel function: $k_smallest_singleHeap()$, $k_smallest_kato()$ e $k_smallest_heapReduction()$.

### 5.2.1 Metodo Single heap

Predisponiamo $CSIZE$ thread $T_j$ (con $j = 0,1,\dots, CSIZE - 1$), ognuno dedicato ad un singolo punto $q_j \in Y_J$ e associamo ad ogni thread un proprio max-heap $H_j$ di dimensione $k$, allocato in shared memory. Come prima operazione, ogni $T_j$ inizializza $H_j$ con il contenuto del corrispondente heap di $q_j$, ovvero $NN[q_j]$. Successivamente ogni thread $T_j$ considera le distanze tra $q_j$ ed ogni altro punto $p_i \in X_I$ e trova i $k$ nearest neighbor di $q_j$, inserendo le rispettive coppie $\langle p_i, \delta \rangle$, con $\delta = dist(q_j, p_i)$, nel proprio heap, sfruttando l'operazione $updateMin$. Infine $T_j$ copia il contenuto di $H_j$ in $NN[q_j]$. Per rispettare i vincoli del modello di CUDA e per ottenere una distribuzione uniforme dei thread tra tutti i multiprocessori, è opportuno dividere i thread in $NB$ blocchi, ognuno composto di $NT$ thread (con $NB = \left\lceil \frac{CSIZE}{NT} \right\rceil$). Inoltre, per ottenere delle buone performance, è opportuno dimensionare $NT$ in modo tale da ottenere la massima occupancy possibile.

La procedura eseguita dalla kernel function è la seguente (in pseudo-codice):

```
k_Smallest_singleHeap(k, NN, J, I, currentChunk){
        tid := threadIdx.x
        j := blockIdx.x * blockDim.x + tid
        i := 0
        j_idx := J * CSIZE + j
        i_idx := I * CSIZE + i
        <shared> H_0, H_1, ..., H_{NT-1}

        H_tid := NN[q_{j_idx}], con q_{j_idx} ∈ D

        while(i < CSIZE AND i_idx < d){
                updateMin(H_tid, ⟨p_{i_idx}, currentChunk[j][i]⟩), con p_{i_idx} ∈ D
                i := i + 1
                i_idx := i_idx + 1
        }

        NN[q_{j_idx}] := H_tid, con q_{j_idx} ∈ D
}
```

### 5.2.2 Metodo di Kato

Definiamo $CSIZE$ blocchi di $NT$ thread, dove ogni blocco $B_j$ è dedicato ad un singolo punto $q_j \in Y_J$. Come prima operazione, ogni blocco $B_j$ si occupa di copiare temporaneamente il max-heap $NN[q_j]$ all'interno di un proprio max-heap $H_j$ di dimensione $k$, allocato in shared memory per rendere più efficienti le successive operazioni $updateMin$. Associamo ad ogni thread un buffer, di dimensione pari a $buffersize$. I thread di $B_j$ leggono in parallelo le distanze nel chunk ed inseriscono nel buffer solamente le coppie $\langle p_i, \delta \rangle$ (con $p_i \in X_I$ e $\delta = dist(q_j, p_i)$) tali che $\delta < \sigma$, dove $\langle s, \sigma \rangle$ è l'elemento radice di $H_j$ (cioè il punto con la massima distanza, tra i $k$ punti nell'heap), fino a riempire il buffer. A questo punto un solo thread si occupa in sequenza di prelevare gli elementi dai vari buffer e di inserirli in $H_j$, con operazioni di tipo $updateMin$. Il procedimento viene iterato fino a quando i thread non hanno letto tutte le distanze. Ad ogni iterazione vengono quindi caricate nei buffer al più $NT \cdot buffersize$ coppie ed effettuate altrettante operazioni sull'heap. Infine, quando tutta la rispettiva $j$-esima riga del chunk è stato letta, si ha l'aggiornamento di $NN[q_j]$ con il contenuto di $H_j$.

La procedura eseguita dalla kernel function è la seguente (in pseudo-codice):

```
k_Smallest_kato(k, NN, J, I, currentChunk){
        tid := threadIdx.x
        j := blockIdx.x
        i := tid
        j_idx := J * CSIZE + j
        i_idx := I * CSIZE + i

        <shared> H_j
        <shared> Buff_0, Buff_1,..., Buff_{NT-1},

        if (tid = 0) H_j := NN[q_{j_idx}]
        __syncthreads()

        Buff_tid := ∅
        while (i < CSIZE AND i_idx < d) {
                for e := 0 to bufferSize {
                        δ := currentChunk[j][i]
                        if (|H_j| < k OR δ < σ, con ⟨s,σ⟩ radice di H_j) {
                                addToBuffer(Buff_tid, ⟨p_{i_idx}, δ⟩), con p_{i_idx} ∈ D
                        }
                        i := i + NT
                        i_idx := i_idx + NT
                }
                __syncthreads()
                if (tid = 0) {
                        for t := 0 to NT {
                                for each ⟨p_{i_idx}, δ⟩ in Buff_t {
                                        updateMin(H_j, ⟨p_{i_idx}, δ⟩)
                                }
                                Buff_t := ∅
                        }
                }
                __syncthreads()
        }
        if (tid = 0) NN[q_{j_idx}] := H_j
}
```

E' importante dimensionare $NT$ e $buffersize$ in maniera opportuna, per poter assicurare un'alta occupancy dei multiprocessori e buone performance. Inoltre sperimentalmente si è trovato che, fissato un certo $buffersize$, le migliori prestazioni si hanno scegliendo $NT$ come il massimo valore che permette di avere il massimo numero possibile di blocchi attivi per multiprocessore. Nel caso specifico della GPU *NVIDIA TESLA M2070* è risultata come miglior coppia $NT = 64$ e $buffersize = 22$.

### 5.2.3 Metodo Heap complete reduction

Predisponiamo $CSIZE$ blocchi di $NT$ thread, dove ogni blocco è dedicato ad un singolo punto $q_j \in Y_J$ e associamo ad ogni thread $T_i$ un proprio max-heap $H_i$ di dimensione $k$, allocato in shared memory. Ogni thread $T_i$ del blocco $B_j$ considera le distanze tra $q_j$ e $CSIZE/NT$ punti $p_z \in X_I$ e trova le rispettive minori $k$ distanze, inserendo le rispettive coppie $\langle p_z, \delta \rangle$ in $H_i$, con $\delta = dist(q_j, p_z)$, mediante l'operazione $updateMin$. Per ogni blocco vengono quindi fusi gli heap degli $NT$

thread, sfruttando la tecnica della riduzione parallela, tramite $\log_2(NT)$ step. Come ultima operazione si ha la fusione di $NN[q_j]$ con l'heap risultante dalla riduzione.

E' possibile effettuare una piccola ottimizzazione, nel caso si utilizzino più chunk, sfruttando un'idea del metodo di *Kato*. Supponiamo che sia stata eseguita una precedente chiamata a questa kernel function per un differente chunk e che $NN[q_j]$ contenga già $k$ elementi, con $\langle s, \sigma \rangle$ come rispettivo elemento radice (con $\sigma = dist(q_j, s)$). In questo caso ogni thread $T_i$ del blocco $B_j$ può evitare di inserire in $H_i$ le coppie $\langle p_z, \delta \rangle$ con $\delta \geq \sigma$ (con $\delta = dist(q_j, p_z)$) in quanto verrebbero sicuramente scartate nella fase finale di fusione con l'heap $NN[q_j]$.

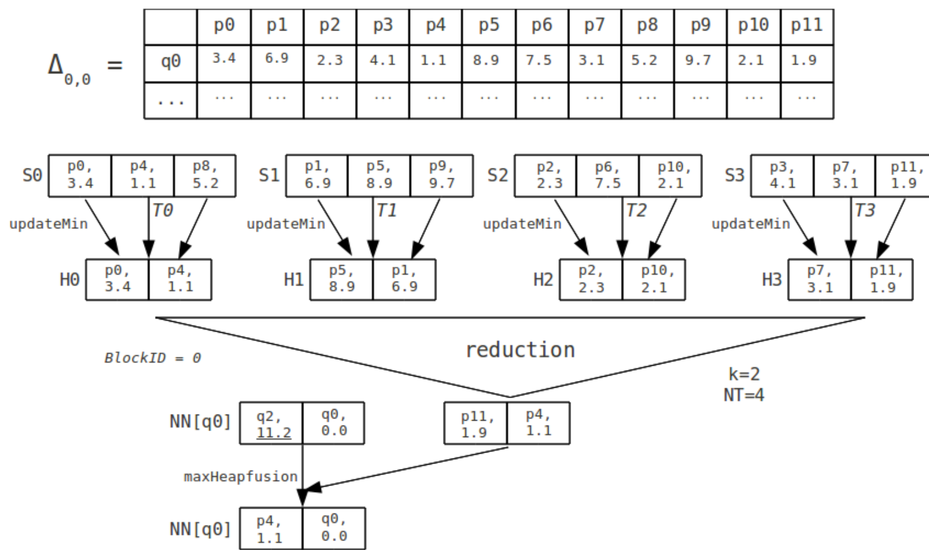



**Figura 5-3 Selezione delle minori *k* distanze con heap complete reduction**

La procedura eseguita dalla kernel function è la seguente (in pseudo-codice):

```
k_Smallest_heapReduction(k, NN, J, I, currentChunk){
        tid := threadIdx.x
        j := blockIdx.x
        i := tid
        j_idx := J * CSIZE + j
        i_idx := I * CSIZE + i

        <shared> H_0, H_1, ..., H_{NT-1}
        H_tid := ∅

        while (i < CSIZE AND i_idx < d) {
                δ := currentChunk[j][i]
                if(|NN[q_{j_idx}]| < k OR δ < σ, ⟨s,σ⟩ radice di NN[q_{j_idx}]){
                        updateMin(H_tid, ⟨p_{i_idx}, δ⟩), con p_{i_idx} ∈ D
                }
                i := i + NT
                i_idx := i_idx + NT
        }
        __syncthreads()
```

```
            //riduzione in shared mem
            if(NT ≥ 512) {
                    if(tid < 256) maxheapFusion(H_tid, H_tid+256)
                    __syncthreads()
            }
            if(NT ≥ 256) {
                    if(tid < 128) maxheapFusion(H_tid, H_tid+128)
                    __syncthreads()
            }
            if(NT ≥ 128) {
                    if(tid < 64) maxheapFusion(H_tid, H_tid+64)
                    __syncthreads()
            }
            if(NT ≥ 64) {
                    if(tid < 32) maxheapFusion(H_tid, H_tid+32)
                    //per tid < 32 i threads sono nello stesso warp
                    //e sempre sincronizzati
            }
            if(NT ≥ 32) {
                    if(tid < 16) maxheapFusion(H_tid, H_tid+16)
            }
            if(NT ≥ 16) {
                    if(tid < 8) maxheapFusion(H_tid, H_tid+8)
            }
            if(NT ≥ 8) {
                    if(tid < 4) maxheapFusion(H_tid, H_tid+4)
            }
            if(NT ≥ 4) {
                    if(tid < 2) maxheapFusion(H_tid, H_tid+2)
            }
            if(NT ≥ 2) {
                    if(tid < 1) maxheapFusion(H_tid, H_tid+1)
            }
            if (tid = 0) maxheapFusion(NN[q_j_idx], H_0)
    }

    con:
    maxheapFusion(HA, HB){
        for each ⟨p_i, δ⟩ in HB{
            updateMin(HA, ⟨p_i, δ⟩)
        }
    }
```

Anche in questo caso, per ottenere delle buone performance, è importante dimensionare $NT$ in modo tale da ottenere la massima occupancy possibile dei multiprocessori.

### 5.3 Trasposta di un chunk

La generazione della trasposta del chunk $\Delta_{J,I}$ viene eseguita dalla GPU tramite la kernel function $transpose()$. Come tecnica parallela per calcolare la matrice trasposta è stata utilizzata una leggera variante della soluzione proposta da *Ruetsch* e *Micikevicius* [26]. Il chunk viene suddiviso in tante sottomatrici, chiamate *tile*, ognuna di dimensione $TILEDIM \times TILEDIM$. Ogni blocco di thread si occupa di effettuare la trasposta di due tile opposti. Ad esempio, dividendo un chunk in $4 \times 4$ tile, il blocco (2,1) si occupa di effettuare la trasposta dei tile (2,1) e (1,2). Per permettere un *coalesced access* in lettura e in scrittura in global memory, per ogni blocco si allocano

in shared memory due matrici di dimensione $TILEDIM \times TILEDIM$, corrispondenti ai due tile. Ogni blocco copia temporaneamente in shared memory i due tile dal chunk e successivamente effettua la trasposta del tile, salvando i rispettivi elementi direttamente nel chunk.

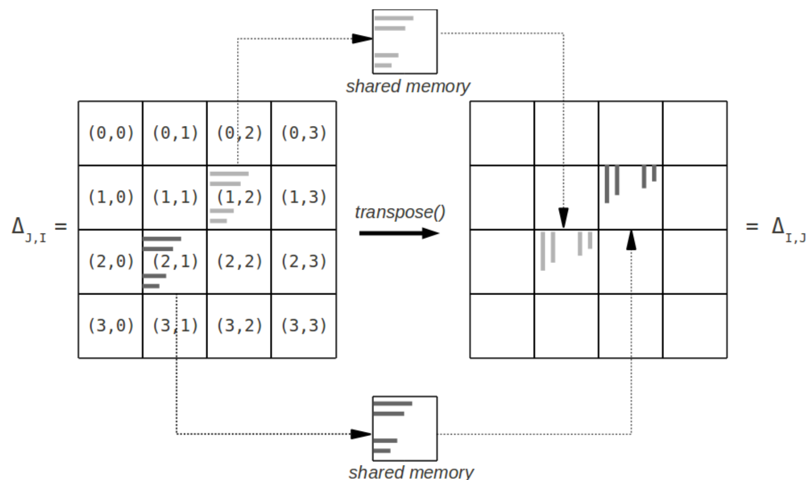



**Figura 5-4 Operazioni di trasposizione eseguite dal blocco (2,1)**

Per ogni blocco si hanno $TILEDIM \times BROWS$ thread, con $BROWS \leq TILEDIM$, dove ogni thread si occupa di $TILEDIM/BROWS$ elementi dei due tile. Inoltre, per permettere un accesso più efficiente in global memory ed evitare il fenomeno denominato *partition camping* [26], ovvero conflitti tra i banchi di memoria della global memory, si utilizza la tecnica del *diagonal reordering*. Ai thread vengono associati gli elementi del tile non secondo un sistema di coordinate cartesiane, ma tramite un sistema di coordinate diagonali, interpretando $blockIdx.y$ come un indice che determina la diagonale nel chunk e $blockIdx.x$ come la distanza dell'elemento dalla diagonale indicata da $blockIdx.y$.

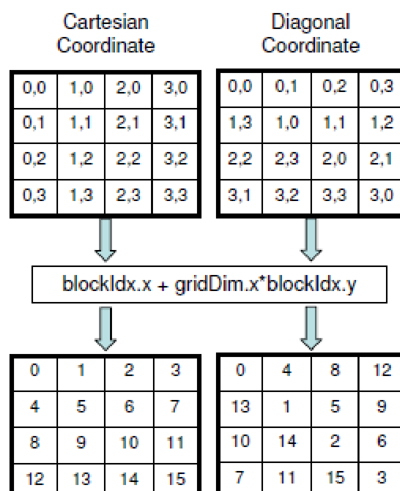



**Figura 5-5 Passaggio alle coordinate diagonali**

La procedura eseguita dalla kernel function è la seguente (in pseudo-codice):

```
transpose(currentChunk){
        <shared> tileA[TILEDIM][TILEDIM + 1]
        <shared> tileB[TILEDIM][TILEDIM + 1]
        //TILEDIM + 1 per evitare shared mem. bank conflicts

        // diagonal reordering
        blockIdx_y := blockIdx.x
        blockIdx_x := (blockIdx.x + blockIdx.y) % gridDim.x

        if (blockIdx_x < blockIdx_y) {
                return
        }

        xIndex := blockIdx_x * TILEDIM + threadIdx.x
        yIndex := blockIdx_y * TILEDIM + threadIdx.y
        index_in := xIndex + yIndex * width

        xIndex := blockIdx_y * TILEDIM + threadIdx.x
        yIndex := blockIdx_x * TILEDIM + threadIdx.y
        index_out := xIndex + yIndex * width

        for j := 0 to TILEDIM step BROWS {
                tileA[threadIdx.y + j][threadIdx.x] := chunk_data[j][index_in]
                tileB[threadIdx.y + j][threadIdx.x] := chunk_data[j][index_out]
        }
        __syncthreads();
        for j := 0 to TILEDIM step BROWS {
                chunk_data[j][index_out] := tileA[threadIdx.x][threadIdx.y + j]
                chunk_data[j][index_in] := tileB[threadIdx.x][threadIdx.y + j]
        }
}
```

Per ottenere delle buone performance è opportuno dimensionare $TILEDIM$ e $BROWS$ in modo tale da ottenere la massima occupancy possibile. Sperimentalmente si è trovato che le migliori prestazioni, per la GPU *NVIDIATesla M2070*, si hanno con $TILEDIM = 32$ e $BROWS = 8$.

### 5.4 Selezione dei top *n* outlier

Anche per la fase di selezione degli $n$ punti dotati del maggior peso (ovvero l'anomaly score) possiamo effettuare un paragone con le tecniche GPGPU per il *kNN problem*. Se consideriamo la fase di selezione delle minori $k$ distanze nel kNN problem, questa consiste nel determinare, per ogni riga della matrice delle distanze, i $k$ minori elementi. Consideriamo ora il vettore riga

$$w = [Sum(NN[p_0]), Sum(NN[p_1]), \dots, Sum(NN[p_{d-1}])] \equiv$$
$$[w_k(p_0, D), w_k(p_1, D), \dots, w_k(p_{d-1}, D)],$$
$$\text{con } p_0, p_1, \dots, p_{d-1} \in D.$$

La fase selezione dei top $n$ outlier si può ricondurre al problema di individuare gli $n$ elementi in $w$ di maggiore valore. Possiamo quindi considerare questo problema, come il "reciproco" della selezione delle minori $k$ distanze, utilizzando come matrice di input il vettore riga $w$. Essendo la matrice in input dotata di una sola riga, dalle considerazioni effettuate nel capitolo 4 possiamo affermare che, sotto questa condizione, le tecniche più efficaci siano *Heap reduction multi-block* e *Kato multi-block*. Negli esperimenti le due tecniche portavano ad uno speedup molto simile, con un leggerissimo vantaggio della seconda sulla prima. Abbiamo scelto però di basarci su *Heap reduction multi-block*, in quanto tale approccio garantisce una minore complessità teorica nel caso peggiore ed il vantaggio registrato negli esperimenti del secondo metodo sul primo è stato al più pari all'8%.

Come per *Heap reduction multi-block* proposto nel capitolo 4 lavoriamo in due fasi, a cui corrispondono due distinte kernel function: $findTopOutliers_phaseA()$ e $findTopOutliers_phaseB()$. Per la prima fase predisponiamo $NB_A$ blocchi di $NT_A$ thread. Associamo ad ogni thread $T_z$ (con $z = 0,1,\ldots,NB_A \cdot NT_A - 1$) un proprio min-heap $H_z$ di dimensione $n$, allocato in shared memory, contenente le coppie $\langle p_i, w_k(p_i, D)\rangle$, con $p_i \in D$. Ogni $T_z$ si occupa inizialmente della selezione degli $n$ punti di maggior peso tra $d/(NB_A \cdot NT_A)$ elementi in $w$, sfruttando l'operazione $updateMax$ su $H_z$. Per ogni blocco viene quindi eseguita la fusione dei vari heap, tramite la tecnica della riduzione parallela. L'heap risultante per ogni blocco viene salvato in global memory. Per la seconda fase, utilizziamo un solo blocco di $NT_B$ thread ed anche in questo caso associamo ad ogni thread $T_z{'}$ (con $z = 0,1,\ldots,NT_B - 1$) un min-heap $H_z{'}$ di dimensione $n$, in shared memory. Ogni $T_z{'}$ esegue la fusione di $NB_A/NT_B$ heap salvati in global memory con il proprio heap $H_z{'}$ e successivamente vengono fusi gli heap $H_0{'}, \ldots, H_{NT_B-1}{'}$ tramite la riduzione parallela. L'heap risultante coincide con il min-heap $Top$ di ODPNestedLoop e contiene i top $n$ outlier.

Per ottenere delle buone performance è necessario dimensionare $NB_A$, $NT_A$ e $NT_B$ in modo tale da ottenere la massima occupancy possibile dei multiprocessori.

Le procedure eseguite dalle due kernel function sono le seguenti (in pseudo-codice):

```
findTopOutliers_phaseA(n, NN, GHMin){

    tid := threadIdx.x
    bid := blockIdx.x

    <shared> H_0, H_1,…, H_{NT_A−1}
    H_tid := ∅

    i := bid · NT_A + tid
    while (i < d) {
        updateMax(H_tid, ⟨p_i, Sum(NN[p_i])⟩),
                                        con p_i ∈ D
        i := i + NT_A · NB_A
    }
    __syncthreads()

    //riduzione in shared mem
    if(NT_A ≥ 512) {
        if(tid < 256)
minheapFusion(H_tid, H_tid+256)
        __syncthreads()
    }
    if(NT_A ≥ 256) {
        if(tid < 128)
          minheapFusion(H_tid, H_tid+128)
        __syncthreads()
    }
    if(NT_A ≥ 128) {
        if(tid < 64)
          minheapFusion(H_tid, H_tid+64)
        __syncthreads()
    }
    if(NT_A ≥ 64) {
        if(tid < 32)
          minheapFusion(H_tid, H_tid+32)
    }
    if(NT_A ≥ 32) {
        if(tid < 16)
          minheapFusion(H_tid, H_tid+16)
    }
    if(NT_A ≥ 16) {
        if(tid < 8)
          minheapFusion(H_tid, H_tid+8)
    }
    if(NT_A ≥ 8) {
        if(tid < 4)
          minheapFusion(H_tid, H_tid+4)
    }
    if(NT_A ≥ 4) {
        if(tid < 2)
          minheapFusion(H_tid, H_tid+2)
    }
    if(NT_A ≥ 2) {
        if(tid < 1)
          minheapFusion(H_tid, H_tid+1)
    }
    if (tid = 0) GHMin[bid] := H_0
}

con:
minheapFusion(HA, HB){
    for each ⟨p_i, Sum(NN[p_i])⟩ in HB{
        updateMax(HA, ⟨p_i, Sum(NN[p_i])⟩)
    }
}
```

```
findTopOutliers_phaseB(n, GHMin, Top)
{

    tid := threadIdx.x

    <shared> H_0, H_1,…, H_{NT_B−1}
    H_tid := ∅

    i := tid
    while (i < NB_A) {
        minheapFusion(H_tid, GHMin[i])
        i := i + NT_B
    }

    __syncthreads()

    //riduzione in shared mem
    if(NT_B ≥ 512) {
        if(tid < 256)
minheapFusion(H_tid, H_tid+256)
        __syncthreads()
    }
    if(NT_B ≥ 256) {
        if(tid < 128)
          minheapFusion(H_tid, H_tid+128)
        __syncthreads()
    }
    if(NT_B ≥ 128) {
        if(tid < 64)
          minheapFusion(H_tid, H_tid+64)
        __syncthreads()
    }
    if(NT_B ≥ 64) {
        if(tid < 32)
          minheapFusion(H_tid, H_tid+32)
    }
    if(NT_B ≥ 32) {
        if(tid < 16)
          minheapFusion(H_tid, H_tid+16)
    }
    if(NT_B ≥ 16) {
        if(tid < 8)
          minheapFusion(H_tid, H_tid+8)
    }
    if(NT_B ≥ 8) {
        if(tid < 4)
          minheapFusion(H_tid, H_tid+4)
    }
    if(NT_B ≥ 4) {
        if(tid < 2)
          minheapFusion(H_tid, H_tid+2)
    }
    if(NT_B ≥ 2) {
        if(tid < 1)
          minheapFusion(H_tid, H_tid+1)
    }
    if (tid = 0) Top := H_0
}
```

## 5.5 Complessità computazionale

Per calcolare la complessità computazionale teorica dell'algoritmo nel caso peggiore poniamo per semplicità $CSIZE = d$, ovvero il caso in cui si utilizzi un singolo chunk. In questa ipotesi non si sfrutta la simmetria delle matrice delle distanze ed è necessario il calcolo di tutte le $d^2$ distanze (anziché $d^2/2$ nel caso ideale). Ipotizziamo inoltre che la GPU sia in grado di eseguire contemporaneamente al più $NP$ thread e che, per semplicità, $d$ sia multiplo di $NP$. Nel caso in cui sia presente nella GPU un numero superiore di thread, l'esecuzione di essi viene sequenzializzata, suddividendoli in gruppi di $NP$ thread alla volta.

Per la fase di calcolo delle distanze si hanno $d/BSIZE$ blocchi di $BSIZE \cdot SSIZE$ thread ed ogni thread si occupa del calcolo di $\frac{d}{SSIZE}$ distanze. Ponendo $a$ pari alla dimensionalità dei punti del dataset (ovvero il numero di coordinate di ogni punto), ogni calcolo della distanza tra due punti richiede un tempo $O(a)$. Di conseguenza questa fase ha complessità pari a $O\left(\left\lceil\frac{\frac{d}{BSIZE} \cdot BSIZE \cdot SSIZE}{NP}\right\rceil \cdot \frac{d}{SSIZE} a\right) = O\left(\frac{d^2}{NP} a\right)$, nell'ipotesi precedente di $d$ multiplo di $NP$, in quanto i $\frac{d}{BSIZE} \cdot BSIZE \cdot SSIZE$ thread vengono eseguiti a gruppi di $NP$.

Per la fase di selezione delle minori $k$ distanze dobbiamo distinguere tra le tre diverse tecniche implementate. Nel caso di *single heap* si hanno $d$ thread, dove ognuno esegue $d$ operazioni $updateMin$, che richiedono ciascuna un tempo $O(\log(k))$. Possiamo quindi affermare che tale tecnica ha complessità pari a $O\left(\left\lceil\frac{d}{NP}\right\rceil \cdot d \cdot \log(k)\right) = O\left(\frac{d^2}{NP}\log(k)\right)$. Nel caso di *heap complete reduction* si hanno $d$ blocchi di $NT$ thread ciascuno. Per quanto discusso nel capitolo 4, possiamo asserire che tale procedura ha complessità pari a $O\left(\left\lceil\frac{d \cdot NT}{NP}\right\rceil \cdot \frac{d}{NT} \cdot \log(k)\right) = O\left(\frac{d^2}{NP}\log(k)\right)$, in quanto ogni thread prima della riduzione esegue al più $d/NT$ operazioni $updateMin$. Con la tecnica di *Kato* la complessità teorica è più elevata e pari a $O\left(\left\lceil\frac{d \cdot NT}{NP}\right\rceil \cdot d \cdot \log(k)\right) = O\left(\frac{d^2 \cdot NT}{NP}\log(k)\right)$. Questo è dovuto al fatto che nel caso peggiore si hanno $d$ operazioni *updateMin* sequenziali per ogni blocco, svolte dall'unico thread che lavora sull'heap, indipendentemente dal valore di $NT$. Nella pratica ovviamente la situazione è diversa e in realtà l'operazione di filtro eseguita dagli $NT$ thread permette di ridurre notevolmente il numero di operazioni sull'heap.

Per il calcolo della complessità della fase di selezione delle minori $k$ distanze assumiamo di utilizzare una delle due tecniche che forniscono la minor complessità teorica (*single heap* e *heap complete reduction*). Tale fase presenta quindi una complessità pari a $O\left(\frac{d^2}{NP}\log(k)\right)$.

Per la selezione dei top $n$ outlier si hanno $NB_A$ blocchi di $NT_A$ thread nel primo step ed un solo blocco di $NT_B$ thread nel secondo. Per le osservazioni effettuate nel capitolo 4 possiamo affermare che tale procedura ha complessità pari a $O\left(\frac{d}{NP}\log(n)\right)$, in quanto il vettore riga $w$ è formato da $d$ elementi e l'operazione $updateMax$ sui min-heap richiede un tempo O(log($n$)).

Nell'ipotesi di $CSIZE = d$ non è ovviamente necessaria la fase di generazione della trasposta del chunk.

La complessità dell'algoritmo CUDA_ODPNestedLoop è quindi pari a:

$$O\left(\frac{d^2}{NP}(\mathrm{a} + \log(\mathrm{k})) + \frac{d}{NP}\log(n)\right) \cong O\left(\frac{d^2}{NP}(\mathrm{a} + \log(\mathrm{k}))\right)$$

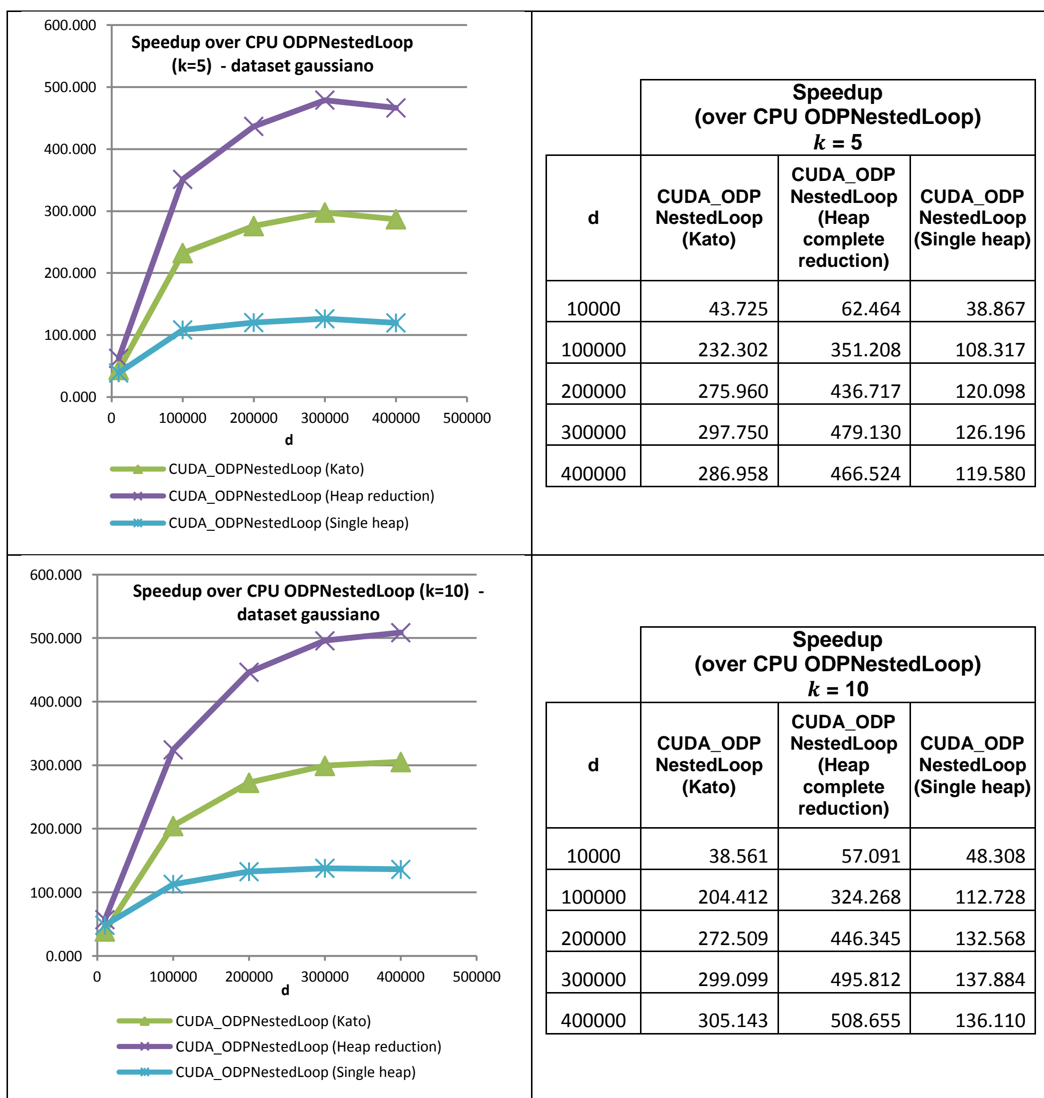
Se confrontiamo la complessità di CUDA_ODPNestedLoop con la complessità di ODPNestedLoop si può notare che, utilizzando una GPU in grado di eseguire contemporaneamente $NP$ thread, in via teorica è possibile scalare di un fattore pari a $NP$ rispetto alla soluzione sequenziale, a meno di un fattore 2 legato allo sfruttamento della simmetria della matrice delle distanze. Inoltre, spezzando la matrice in più chunk, è possibile risentire meno di questa lacuna, grazie alle ottimizzazioni apportate nella fase di calcolo delle distanze che sfruttano l'operazione di trasposizione dei chunk e permettono di generare solamente la metà dei chunk.

## 5.6 Risultati sperimentali

Per valutare le performance dell'algoritmo parallelo CUDA_ODPNestedLoop sono state eseguite due diverse serie di test, la prima orientata ad un confronto con l'algoritmo ODPNestedLoop sequenziale, la seconda con l'algoritmo ODPSolvingSet, molto più efficiente di ODPNestedLoop. Per le kernel function è stato realizzato un modulo CUDA *cubin*, scritto in *CUDA C* e compilato. La parte di comunicazione e coordinamento con la GPU, richiesta dall'algoritmo CUDA_ODPNestedLoop, è stata implementata in Java, sfruttando la libreria *JCuda*. Anche in questo caso per i test si è utilizzato un singolo nodo del supercomputer *IBM PLX*, dotato di due CPU *six-cores Intel Westmere* a 2.40 *GHz* (*E5645*) e di due GPU *NVIDIA Tesla M2070*.

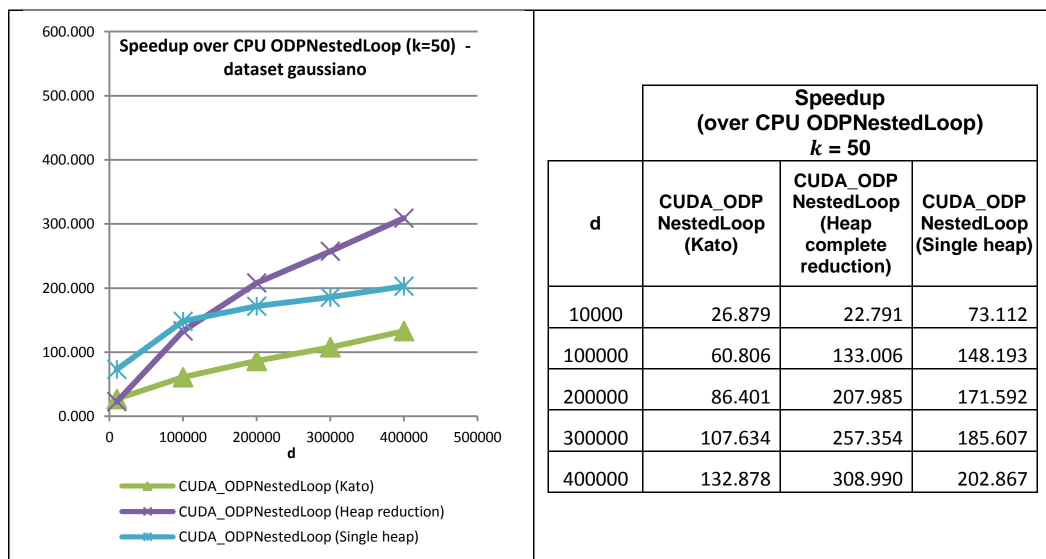
### 5.6.1 Confronto con ODP Nested Loop

La prima batteria di test ha l'obiettivo di confrontare le prestazioni dell'algoritmo parallelo CUDA_ODPNestedLoop eseguito su GPU, con l'algoritmo sequenziale ODPNestedLoop eseguito su CPU. Per questi esperimenti sono stati utilizzati cinque dataset artificiali, contenenti un numero variabile (da 10,000 a 400,000) di vettori bi-dimensionali, generati da una distribuzione normale 2-d con valor medio 100 e deviazione standard 50. Negli esperimenti si è posto $n = 10$ e sono state effettuate prove per $k = 5, 10$ e 50. Per CUDA_ODPNestedLoop è stato fissato $CSIZE = 16{,}384$, ovvero la massima dimensione possibile del chunk per renderlo allocabile nella global memory della GPU, in quanto sperimentalmente si è trovato che i migliori risultati si hanno dimensionando $CSIZE$ come massimo valore possibile. Inoltre anche tutti gli altri parametri specifici di CUDA sono stati appositamente scelti per garantire le migliori performance.




| | Speedup (over CPU ODPNestedLoop) $k = 5$ | | |
|---|---|---|---|
| d | CUDA_ODP NestedLoop (Kato) | CUDA_ODP NestedLoop (Heap complete reduction) | CUDA_ODP NestedLoop (Single heap) |
| 10000 | 43.725 | 62.464 | 38.867 |
| 100000 | 232.302 | 351.208 | 108.317 |
| 200000 | 275.960 | 436.717 | 120.098 |
| 300000 | 297.750 | 479.130 | 126.196 |
| 400000 | 286.958 | 466.524 | 119.580 |

| | Speedup (over CPU ODPNestedLoop) $k = 10$ | | |
|---|---|---|---|
| d | CUDA_ODP NestedLoop (Kato) | CUDA_ODP NestedLoop (Heap complete reduction) | CUDA_ODP NestedLoop (Single heap) |
| 10000 | 38.561 | 57.091 | 48.308 |
| 100000 | 204.412 | 324.268 | 112.728 |
| 200000 | 272.509 | 446.345 | 132.568 |
| 300000 | 299.099 | 495.812 | 137.884 |
| 400000 | 305.143 | 508.655 | 136.110 |




| | Speedup (over CPU ODPNestedLoop) $k$ = 50 | | |
|---|---|---|---|
| d | CUDA_ODP NestedLoop (Kato) | CUDA_ODP NestedLoop (Heap complete reduction) | CUDA_ODP NestedLoop (Single heap) |
| 10000 | 26.879 | 22.791 | 73.112 |
| 100000 | 60.806 | 133.006 | 148.193 |
| 200000 | 86.401 | 207.985 | 171.592 |
| 300000 | 107.634 | 257.354 | 185.607 |
| 400000 | 132.878 | 308.990 | 202.867 |

**Figura 5-6 Speedup di CUDA_ODPNestedLoop su ODPNestedLoop (test su dataset gaussiano, variando *d*)**

Come indicatore di performance consideriamo lo *speedup* di CUDA_ODPNestedLoop su ODPNestedLoop, definito come il rapporto tra i tempi di esecuzione di ODPNestedLoop e di CUDA_ODPNestedLoop. Dai risultati ottenuti si evidenzia come questo valore sia molto elevato, per $d$ sufficientemente grande. Nel caso di $k = 5$ e $k = 10$ l'utilizzo del metodo heap complete reduction per la selezione delle $k$ minori distanze risulta essere la scelta più efficace, presentando addirittura uno speedup intorno a 500 per $d = 400{,}000$. Seguono quindi le versioni che utilizzano il metodo di Kato e single heap, con un distacco molto marcato. Per $k = 50$ le differenze tra i diversi metodi si riducono, in quanto le prestazioni di heap complete reduction e di Kato peggiorano nettamente, mentre quelle di single heap addirittura tendono a migliorare. Heap complete reduction risulta comunque essere ancora la scelta migliore, presentando uno speedup intorno a 300 per $d = 400{,}000$. Il degrado delle prestazioni all'aumentare di $k$ è dovuto principalmente a due cause. Con $k$ elevato, per heap complete reduction la quantità di shared memory richiesta per gli heap dei diversi thread diventa molto significativa, dando luogo ad un minor grado di occupancy dei processori. Inoltre per $k$ elevato le azioni di *filtering* effettuate da Kato (tramite l'inserimento delle coppie nel buffer) e da heap complete reduction (tramite l'esecuzione delle operazioni $updateMin$ sugli heap temporanei solamente per le coppie con distanza inferiore a quella della radice dell'heap) diventano meno efficaci.

### 5.6.2 Confronto con ODP Solving Set

Negli esperimenti presentati nel paragrafo 2.5.2 si è evidenziato come l'algoritmo ODPSolvingSet presenti performance nettamente superiori a ODPNestedLoop. In particolare consideriamo lo speedup di ODPSolvingSet su ODPNestedLoop, nel caso dei test effettuati sui dataset gaussiani con $k = 50$ e confrontiamolo con i risultati ottenuti in 5.6.1, tramite CUDA_ODPNestedLoop.

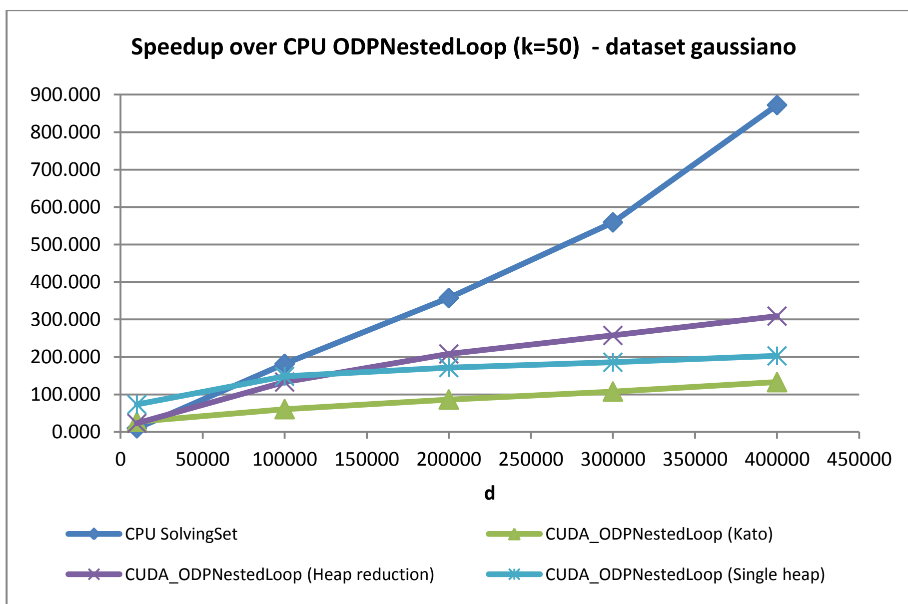
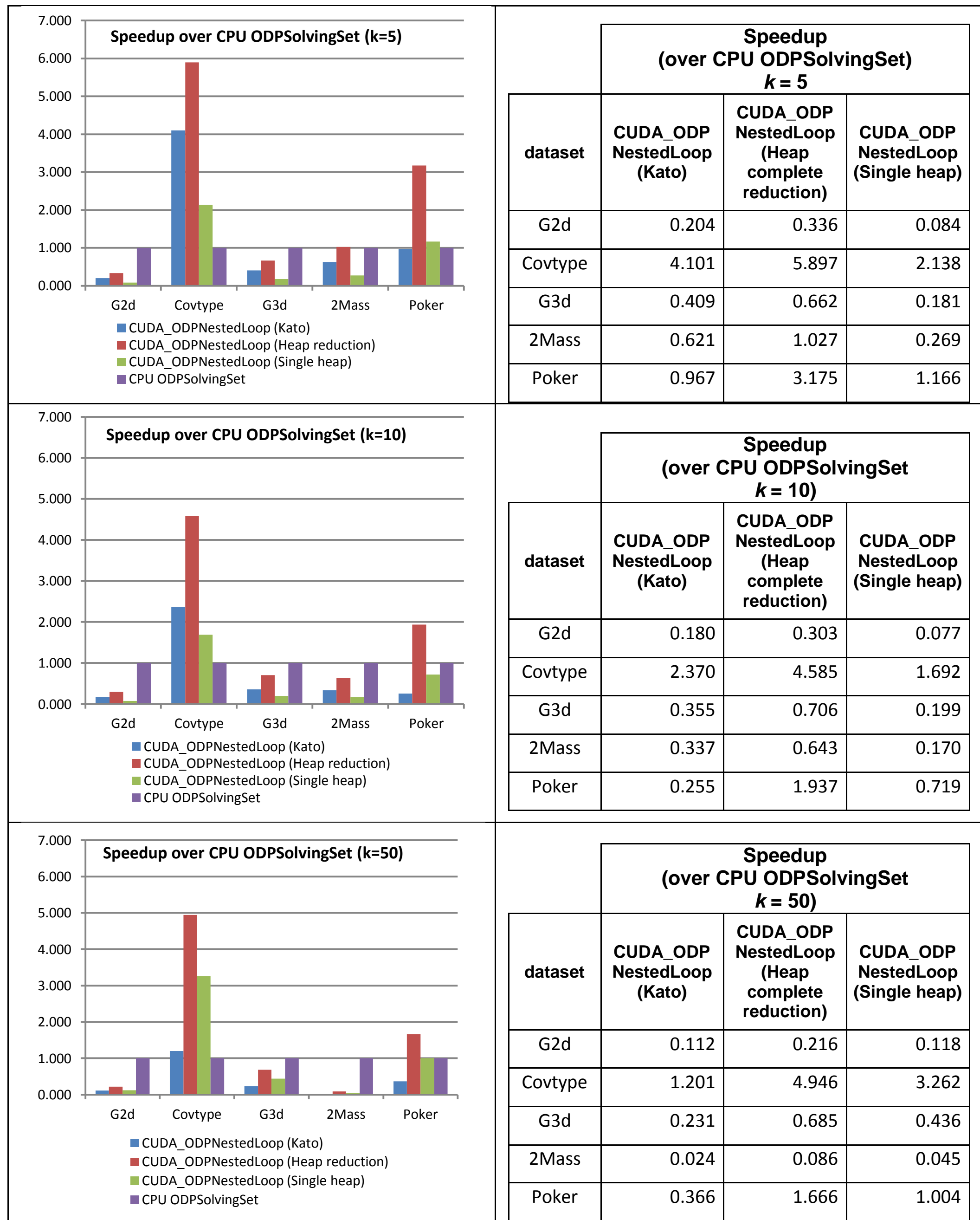



**Figura 5-7 Speedup di CUDA_ODPNestedLoop e ODPSolvingSet su ODPNestedLoop (test su dataset gaussiano, variando $d$)**

Per $d = 400{,}000$ ODPSolvingSet ha raggiunto uno speedup perfino superiore a 800, mentre CUDA_ODPNestedLoop ha fornito uno speedup intorno a 300 (nel caso di utilizzo di heap complete reduction). CUDA_ODPNestedLoop è riuscito ad avere prestazioni superiori a ODPSolvingSet solamente per $d = 10{,}000$.

Per effettuare un confronto più accurato, è stata eseguita un'altra batteria di test, utilizzando questa volta i cinque dataset presentati nel paragrafo 2.5.1. In questi test si è posto $n = 10$ e sono state effettuate prove per $k = 5, 10$ e $50$. Da questi esperimenti è stato escluso il metodo ODPNestedLoop sequenziale, in quanto non in grado di terminare l'esecuzione entro il limite di tempo prefissato a 24 ore, ed è stato preso come riferimento lo speedup di CUDA_ODPNestedLoop su ODPSolvingSet.



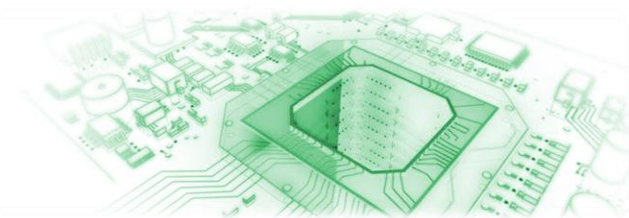

| | Speedup (over CPU ODPSolvingSet) k = 5 | | |
|---|---|---|---|
| dataset | CUDA_ODP NestedLoop (Kato) | CUDA_ODP NestedLoop (Heap complete reduction) | CUDA_ODP NestedLoop (Single heap) |
| G2d | 0.204 | 0.336 | 0.084 |
| Covtype | 4.101 | 5.897 | 2.138 |
| G3d | 0.409 | 0.662 | 0.181 |
| 2Mass | 0.621 | 1.027 | 0.269 |
| Poker | 0.967 | 3.175 | 1.166 |

| | Speedup (over CPU ODPSolvingSet k = 10) | | |
|---|---|---|---|
| dataset | CUDA_ODP NestedLoop (Kato) | CUDA_ODP NestedLoop (Heap complete reduction) | CUDA_ODP NestedLoop (Single heap) |
| G2d | 0.180 | 0.303 | 0.077 |
| Covtype | 2.370 | 4.585 | 1.692 |
| G3d | 0.355 | 0.706 | 0.199 |
| 2Mass | 0.337 | 0.643 | 0.170 |
| Poker | 0.255 | 1.937 | 0.719 |

| | Speedup (over CPU ODPSolvingSet k = 50) | | |
|---|---|---|---|
| dataset | CUDA_ODP NestedLoop (Kato) | CUDA_ODP NestedLoop (Heap complete reduction) | CUDA_ODP NestedLoop (Single heap) |
| G2d | 0.112 | 0.216 | 0.118 |
| Covtype | 1.201 | 4.946 | 3.262 |
| G3d | 0.231 | 0.685 | 0.436 |
| 2Mass | 0.024 | 0.086 | 0.045 |
| Poker | 0.366 | 1.666 | 1.004 |

**Figura 5-8 Speedup di CUDA_ODPNestedLoop su ODPSolvingSet**

Le prestazioni di CUDA_ODPNestedLoop risultano essere anche in questo caso mediamente inferiori a quelle di ODPSolvingSet, a parte alcune eccezioni. Per il dataset *Covtype* l'algoritmo parallelo ha ottenuto uno speedup sull'algoritmo sequenziale compreso tra 6 e 5 (con l'utilizzo di heap complete reduction) e per *Poker* compreso tra 3 e 2. In tutti gli altri casi ODPSolvingSet è risultato molto più veloce di CUDA_ODPNestedLoop, in particolare per il dataset *2Mass* (più rapido di quasi 10 volte per $k = 50$).



# Capitolo 6: Implementazione in CUDA di ODP Solving Set

Nel capitolo precedente abbiamo evidenziato la presenza di un discreto gap prestazionale tra l'algoritmo parallelo *CUDA_ODPNestedLoop* eseguito su GPU e l'algoritmo sequenziale *ODPSolvingSet* eseguito su CPU. Il passo successivo di questo lavoro è quindi quello di utilizzare le tecniche di GPU computing per permettere un'efficiente esecuzione parallela dell'algoritmo ODPSolvingSet.

Come nel caso dell'algoritmo di tipo *nested loop*, è necessario ristrutturare l'algoritmo *ODPSolvingSet*, al fine di permettere un'esecuzione parallela. L'algoritmo viene suddiviso in diverse fasi, ognuna delle quali è formata da molteplici thread, eseguiti in parallelo sui diversi multiprocessori della GPU. Compito della CPU è il coordinamento dell'esecuzione delle diverse fasi sulla GPU e lo svolgimento di ulteriori operazioni utili per la logica dell'algoritmo stesso, effettuate in parallelo al lavoro svolto dalla GPU.

Dopo una breve fase preliminare svolta dalla CPU, riguardante l'inizializzazione delle varie strutture dati necessarie e la selezione dei primi $m$ candidati, come nell'implementazione sequenziale l'algoritmo viene eseguito tramite diverse iterazioni. Ad ogni iterazione avviene il calcolo del peso di ogni punto candidato e la selezione dei nuovi candidati per l'iterazione successiva. Questo procedimento prosegue fino a quando non si hanno più punti dichiarati attivi, ovvero quando il nuovo insieme di candidati restituito risulta essere vuoto. La CPU ha la responsabilità di mantenere in memoria ed aggiornare ad ogni iterazione il solving set, tramite l'inserimento del nuovo insieme di candidati eletto, e il min-heap $Top$, contenente i correnti $n$ punti di maggior peso. La GPU si occupa invece di tutte le altre operazioni.

Le azioni svolte ad ogni iterazione, dalla CPU e dalla GPU, possono essere suddivise nelle seguenti fasi:

- *Inizializzazione*: la GPU effettua la rimozione dei punti candidati dal dataset $D$ ($D := D - C$) e l'inizializzazione degli heap in $LNNC$, tramite la copia del contenuto di $NN[q_j]$ in $LNNC[q_j]$, per ogni $q_j \in C$. Queste operazioni avvengono tramite la kernel function $init()$. In parallelo all'azione svolta dalla GPU, la CPU effettua l'inserimento di $C$ nel corrente solving set $S$.

- *Confronto* $C$ *con* $C$: la GPU calcola in parallelo le distanze per ogni coppia di punti $(q_i, q_j)$ in $C$ (con $q_i, q_j \in C$) ed esegue l'aggiornamento dei rispettivi heap $LNNC[q_i]$ e $LNNC[q_j]$. A tal fine viene utilizzata la kernel function $compareCwithC()$.
- *Confronto* $D$ *con* $C$: è la fase più critica che è opportuno suddividere a sua volta in più sottofasi, per poter operare efficientemente in parallelo tramite l'ambiente CUDA:
  - *upMin*: in cui la GPU calcola in parallelo le distanze per ogni coppia di punti $(p_i, q_j)$, con $p_i \in D$ e $q_j \in C$, ed esegue l'aggiornamento dei rispettivi heap $NN[p_i]$ e $LNNC[q_j]$. Questa sottofase è quella che richiede il maggior lavoro e la sua efficacia è dipendente dai parametri di input all'algoritmo. Per questo motivo sono state previste diverse tecniche, specifiche per determinate situazioni.
  - *upMax*: in cui la GPU esegue l'aggiornamento dell'heap $LC$, per permettere la successiva selezione dei nuovi punti candidati, e la dichiarazione dei punti $p_i \in D$ come attivi o non attivi. Per riuscire ad eseguire in maniera efficace queste azioni è necessario utilizzare due distinte kernel function: $compareDwithC_upMax_phaseA()$ e $compareDwithC_upMax_phaseB()$. Inoltre, in parallelo a queste operazioni svolte dalla GPU, la CPU recupera dalla memoria della GPU il vettore $W = [Sum(LNNC[q_0]), \dots, Sum(LNNC[q_{m-1}])]$, con $q_j \in C$, contenente il peso di ogni punto, necessario per la fase successiva.
- *Selezione dei nuovi candidati e aggiornamento dell'heap Top*: la GPU esegue la copia dei punti presenti nell'heap $LC$ in $C$, predisponendo il nuovo insieme di candidati per l'iterazione successiva, tramite la kernel function $candSelect()$. In parallelo a tale operazione, la CPU esegue l'aggiornamento dell'heap $Top$, effettuando l'operazione $updateMax$ su $Top$ per ogni coppia $\langle q_j, Sum(LNNC[q_j])\rangle$, con $q_j \in C$ e $Sum(LNNC[q_j]) \in W$. Infine la CPU recupera dalla memoria della GPU il nuovo insieme dei candidati, aggiornando la copia di $C$ nella propria memoria.

L'intero algoritmo, in termini di operazioni eseguite dalla CPU, può quindi essere ristrutturato nel modo seguente (in pseudo-codice):

```
CUDA_ODPSolvingSet(D, dist, n, k, m){
        allocateToGPU(D) //contenente il dataset
        allocateToGPU(C) //contenente i punti candidati
        allocateToGPU(NN) //contenente i max-heap NN[p] ∀ p ∈ D
        allocateToGPU(LNNC) //contenente i max-heap LNNC[q] ∀ q ∈ C
        allocateToGPU(LC) //min-heap per i nuovi candidati
        allocateToGPU(…) //strutture dati specifiche per la
                //tecnica adottata per la fase compareDwithC_upMin *
        allocateToGPU(GHMin) //contenente i min-heap temporanei
          //per compareDwithC_upMax_phaseA, compareDwithC_upMax_phaseB

        S := ∅
        Top := ∅
        C := RandomSelect(D, m)
        copyToGPU(D)
        copyToGPU(C)

        while (C ≠ ∅) {
                launchOnGPU(init(D, C, NN, LNNC))
                S := S ∪ C
                synchWithGPU()

                launchOnGPU(compareCwithC(k, dist, C, LNNC))
                synchWithGPU()

                compareDwithC_upMin * (k, dist, D, C, NN, LNNC, Min(Top), …)
                                //dipendente dalla tecnica adottata
                launchOnGPU(
                    compareDwithC_upMax_phaseA(m, D, NN, GHMin, Min(Top))
                copyFromGPU(w),con
                        w = [Sum(LNNC[q_0]), …, Sum(LNNC[q_{m-1}])],  q_j ∈ C
                synchWithGPU()
                launchOnGPU(compareDwithC_upMax_phaseB(m, LC, GHMin)
                synchWithGPU()

                launchOnGPU(candSelect(C, LC))
                for each q_j in C {
                        updateMax(Top, ⟨q_j, Sum(LNNC[q_j])⟩)
                }
                synchWithGPU()
                copyFromGPU(C)
        }
}
```

Presentiamo quindi in dettaglio le diverse operazioni svolte dalla GPU per ogni fase dell'algoritmo.

## 6.1 Fase di inizializzazione

In questa fase, tramite la kernel function $init()$, la GPU esegue la rimozione dei punti candidati dal dataset $D$ ($D := D - C$) e l'inizializzazione degli heap in $LNNC$, con la copia del contenuto di $NN[q_j]$ in $LNNC[q_j]$, per ogni $q_j \in C$. Dal punto di vista implementativo, il dataset $D$ viene allocato nella global memory della GPU come un vettore ordinato di $d$ punti. Per poter effettuare l'operazione $D := D - C$ in maniera

efficiente, viene salvato negli elementi in $C$ anche l'indice di ogni punto in $D$, in modo che non sia necessario effettuare una ricerca in $D$ per individuare la posizione dei punti durante l'operazione di rimozione.

Per la kernel function si definiscono $m$ thread $T_j$, un thread per ogni $q_j \in C$. Come prima operazione ogni $T_j$ si occupa dell'inizializzazione di $LNNC[q_j]$, copiando nell'heap il contenuto di $NN[q_j]$. Successivamente un singolo thread si occupa di trovare i possibili punti da sostituire in $D$, al posto dei punti candidati che devono essere rimossi. Per questa operazione effettua in primo luogo un sorting degli indici degli $m$ punti (tramite un algoritmo di tipo heapsort che richiede un tempo $O(m \cdot \log(m))$, salvando il risultante vettore $c_idx[]$ in shared memory. Per ogni candidato, procedendo in ordine crescente di indice in $D$, sceglie il rispettivo sostituto selezionandolo tra gli ultimi $m$ punti in $D$. I rispettivi indici dei sostituti vengono salvati all'interno di un vettore $subs_idx[]$, sempre allocato in shared memory. Infine ogni thread $T_j$ si occupa dell'effettiva copia dei sostituti in $D$.

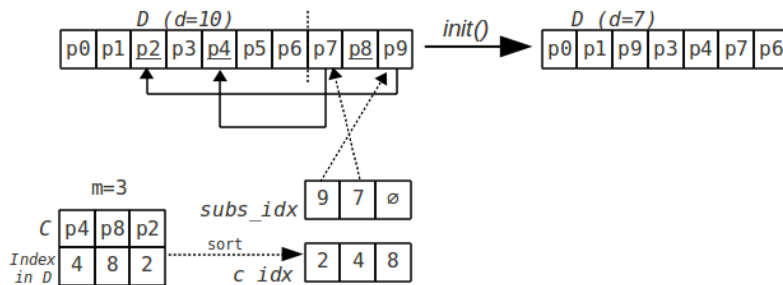



**Figura 6-1 Eliminazione dal dataset dei punti in $C$**

La procedura eseguita dalla kernel function, in pseudo-codice, è la seguente:

```
init(D, C, NN, LNNC) {
        <shared> c_idx[]
        <shared> subs_idx[]
        j := threadIdx.x

        LNNC[q_j] := NN[q_j]
        if(j = 0){
                c_idx := sort(C.indexInD) // O(m · log(m)) con heapsort
                subs_idx := findSubsitutes(|C|, D, c_idx)
                d := d − |C|
        }
        __syncthreads()
        if(subs_idx[j] ≠ −1){
                D[c_idx[j]] := D[subs_idx[j]]
        }
}
```

```
con:
subs_idx[] findSubstitutes(dim, D, c_idx){
        t := d − 1
        for j := 0 to dim{
                subs_idx[j] := −1
                while(t > c_idx[j] AND subs_idx[j] = −1){
                        if(t not in c_idx){ // O(log(m)) ricerca binaria
                                subs_idx[j] := t
                        }
                        t := t − 1
                }
        }
        return subs_idx
}
```

## 6.2 Fase di confronto $C$ con $C$

In questa fase la GPU, tramite la kernel function $compareCwithC()$, calcola le distanze per ogni coppia di punti $(q_i, q_j)$ in $C$ (con $q_i, q_j \in C$) ed esegue l'aggiornamento dei rispettivi heap $LNNC[q_i]$ e $LNNC[q_j]$. A tale scopo si definiscono $m$ thread $T_j$, ognuno dedicato ad singolo punto $q_j \in C$. Ogni thread $T_j$ calcola la distanza tra $q_j$ ed ogni altro punto $q_i \in C$ e chiama l'operazione $updateMin$ su $LNNC[q_j]$ per ogni coppia $\langle q_i, \delta \rangle$, con $\delta = dist(q_i, q_j)$. Per poter sfruttare al meglio il parallelismo della GPU, è opportuno dividere i thread in più blocchi di dimensione $NT$, dimensionando $NT$ in modo tale da poter suddividere il lavoro tra più multiprocessori.

La procedura eseguita dalla kernel function, in pseudo-codice, è la seguente:

```
compareCwithC(k, dist, C, LNNC){
        j := blockIdx.x · blockDim.x + threadIdx.x

        for each q_i in C{
                δ := dist(q_i, q_j), con q_j ∈ C
                updateMin(LNNC[q_j], ⟨q_i, δ⟩ )
        }
}
```

E' da osservare che, rispetto all'algoritmo sequenziale per CPU, si ha un numero maggiore di distanze calcolate, in quanto non si sfrutta la simmetria della matrice delle distanze tra i vari punti candidati (di dimensione $m \times m$). Siccome il parametro $m$ viene posto tipicamente ad un valore relativamente piccolo (comunemente compreso tra 10 e 100) se confrontato con il numero massimo $NP$ di thread contemporanei che può eseguire la GPU, non è conveniente adoperare una suddivisione di tale matrice in più sezioni per cercare di sfruttare la simmetria (come in 5.1). Di conseguenza è preferibile una soluzione che calcoli in maniera parallela tutte le $m^2$ distanze, cercando di sfruttare al meglio l'hardware della GPU.

### 6.3 Fase di confronto $D$ con $C$ - upMin

Nella prima fase del confronto $D$ con $C$ il compito della GPU è calcolare in parallelo le distanze per ogni coppia di punti $(p_i, q_j)$, con $p_i \in D$ e $q_j \in C$, ed eseguire l'aggiornamento dei rispettivi heap $NN[p_i]$ e $LNNC[q_j]$. Il più semplice approccio consisterebbe nel definire $d$ thread $T_i$, ognuno dedicato ad singolo punto $p_i \in D$, dove ogni $T_i$ calcoli la distanza tra $p_j$ ed ogni altro punto candidato $q_j \in C$ e chiami l'operazione $updateMin$ su $NN[p_i]$ e $LNNC[q_j]$ per ogni distanza. Purtroppo però con un'implementazione di questo tipo si creerebbero dei contrasti negli accessi agli heap $LNNC[q_j]$, in quanto si avrebbe la presenza di più thread contemporanei in attesa di effettuare l'operazione $updateMin$ su $LNNC[q_j]$. Se si definisse tale operazione come *atomica*, la soluzione parallela andrebbe di fatto a degenerare in una soluzione sequenziale, a causa dei conflitti di accesso su tali heap. Per poter ovviare a questo problema si possono applicare diverse tecniche.

#### 6.3.1 Modalità con matrice delle distanze

La prima possibile soluzione a questo problema consiste nello spezzare le operazioni in due ulteriori sottofasi. La prima fase A si compone di $d$ thread $T_i$, dove ogni $T_i$ calcola la distanza tra $p_i \in D$ ed ogni altro punto candidato $q_j \in C$, salva tali distanze nelle rispettive celle di una matrice delle distanze $[\delta_{j,i}]$, con $\delta_{j,i} = dist(p_i, q_j)$, e chiama l'operazione $updateMin$ sull'heap $NN[p_i]$ per ogni coppia $\langle q_j, \delta_{j,i} \rangle$. Nella seconda fase B si ha invece l'aggiornamento parallelo degli heap $LNNC$, considerando in input la matrice calcolata nella fase A. Grazie a questa tecnica è possibile evitare contrasti di accesso negli heap e quindi sfruttare al pieno il parallelismo dell'hardware della GPU.

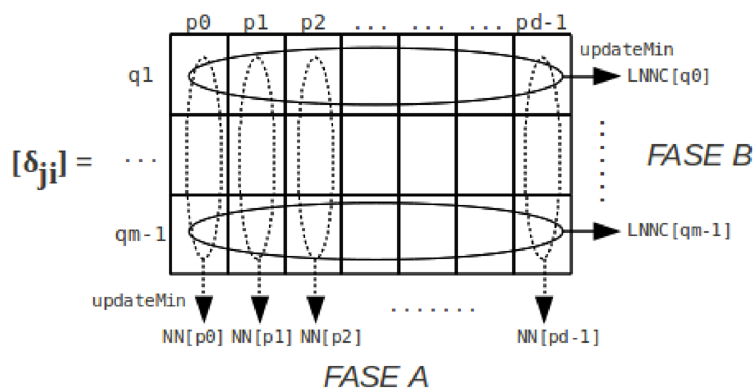



**Figura 6-2 Fase di confronto $D$ con $C$ - upMin con matrice delle distanze**

Se consideriamo la matrice delle distanze $[\delta_{j,i}]$, si ha che tale matrice ha dimensione $m \times d$. Il parametro $m$ dell'algoritmo viene posto tipicamente ad un valore relativamente piccolo, comunemente compreso tra 10 e 100. Viceversa, per dataset dotati di un alto numero di istanze, il valore $d$ può risultare molto elevato. In un caso del genere può non essere possibile allocare l'intera matrice delle distanze in global memory, se il device non è dotato di abbastanza memoria RAM dedicata. Per questo motivo è opportuno dividere la matrice delle distanze in più chunk, come nel capitolo 5. Ipotizzando $m$ relativamente piccolo, possiamo spezzare la matrice solamente in sezioni verticali, tramite dei chunk $\Delta_I$ di dimensione $m \times CSIZE$. Per ogni chunk è quindi necessario eseguire la fase A (per la generazione del chunk stesso e l'aggiornamento degli heap $NN$) e la fase B (per l'aggiornamento degli heap LNNC).

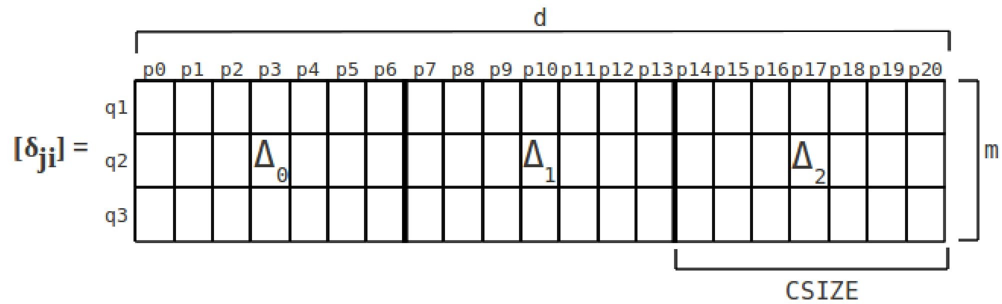



**Figura 6-3 Divisione delle matrice delle distanze in chunk**

La fase *upMin*, in termini di operazioni di coordinamento eseguite dalla CPU nel caso di utilizzo della modalità con matrice delle distanze, può quindi essere indicata tramite la seguente funzione (mostrata in pseudo-codice):

```
compareDwithC_upMin_dm(k, dist, D, C, NN, LNNC, Min(Top), currentChunk){
        numOfChunks := ⌈d/CSIZE⌉
        for I := 0 to numOfChunks{
                launchOnGPU(
                        compareDwithC_upMin_dm_phaseA(k, dist, D, C, NN, LNNC,
                                                Min(Top), I, currentChunk))
                synchWithGPU()
                launchOnGPU(compareDwithC_upMin_dm_phaseB_*(k, D, C,
                                        LNNC, Min(Top), I, currentChunk))
                                //dipendente dalla tecnica adottata
                synchWithGPU()
        }
}
```

Per la scelta della tecnica più adatta da utilizzare nella fase B, possiamo ancora effettuare un paragone con le tecniche GPGPU per il *kNN problem*, come nel paragrafo 5.2. Dato un chunk $\Delta_I$, possiamo considerare l'insieme $X_I = \{p_{i_idx} \in D \mid CSIZE \cdot I \leq i_idx < CSIZE \cdot (I+1)\}$ (ovvero l'insieme dei punti in $D$ corrispondenti alle colonne del chunk $\Delta_I$) come l'insieme dei *reference point* e $C$

come l'insieme dei *query point*. Sotto tali ipotesi abbiamo che la cardinalità dell'insieme dei query point è pari ad $m$, con $m \cong 100$. Per quanto osservato nel capitolo 4, per un tale valore di cardinalità dell'insieme dei query point le tecniche più performanti sono risultate essere quella di $Kato$ (per $k$ elevato) e *heap complete reduction* (per $k$ piccolo). Possiamo quindi utilizzare delle semplici varianti di questi metodi, per l'esecuzione della fase B.

#### 6.3.1.1 Modalità con matrice delle distanze – fase A

La fase A per il chunk $\Delta_I$ è eseguita dalla kernel function $compareDwithC_upMin_dm_phaseA()$ e si compone di $CSIZE$ thread $T_i$, uno per ogni punto $p_i \in X_I$. Ogni thread $T_i$ calcola la distanza tra $p_i$ ed ogni altro punto candidato $q_j \in C$, chiama l'operazione $updateMin$ sull'heap $NN[p_i]$ e salva tali distanze nelle rispettive celle del chunk. Inoltre, come avviene nell'algoritmo sequenziale, il calcolo della distanza e la chiamata di $updateMin$ sull'heap vengono eseguiti solamente se $\max\{Sum(NN[p_i]), Sum(LNNC[q_j])\} \geq Min(Top)$. Nel caso tale condizione non sia soddisfatta, viene salvato nella rispettiva cella del chunk il valore $+\infty$. Per rispettare i vincoli del modello di CUDA e per ottenere una distribuzione uniforme dei thread tra tutti i multiprocessori, è opportuno dividere i thread in $NB$ blocchi, ognuno composto di $NT$ thread (con $NB = \left\lceil \frac{CSIZE}{NT} \right\rceil$). In questa fase le operazioni $updateMin$ vengono eseguite direttamente sugli heap $NN[p_i]$ in global memory, senza effettuare una prima copia preliminare in shared memory. Questo in quanto tipicamente si ha che $m$ è di un ordine di grandezza simile a $k$ (comunemente $m = 100$ e $k = 50$) e in una tale situazione non sarebbe per nulla conveniente copiare gli heap $NN[p_i]$ temporaneamente in shared memory e poi aggiornare nuovamente la copia in global memory, dopo aver eseguito solamente al più $m$ operazioni $updateMin$ sull'heap. A questo poi si aggiunge l'importante fattore che gli heap in shared memory limitano fortemente l'occupancy dei multiprocessori per $k$ elevato ed è quindi opportuno scegliere questa strada solo quando è strettamente necessario.

La procedura eseguita dalla kernel function, in pseudo-codice, è la seguente:

```
compareDwithC_upMin_dm_phaseA(k, dist, D, C, NN, LNNC,
                                        Min(Top), I, currentChunk){
        i := blockIdx.x · blockDim.x + threadIdx.x
        i_idx := I * CSIZE + i

        for each q_j in C{
                if (max{Sum(NN[p_i_idx]), Sum(LNNC[q_j])} ≥ Min(Top),
                                                con p_i_idx ∈ D) {
                        δ := dist(p_i_idx, q_j)
                        updateMin(NN[p_i_idx], ⟨q, δ⟩ )
                }else{
                        δ := +∞
                }
                currentChunk[j][i] := δ
        }
}
```

### 6.3.1.2 Modalità con matrice delle distanze – fase B con metodo di Kato

La fase B per il chunk $\Delta_I$, nel caso di utilizzo del metodo di Kato, è eseguita tramite la kernel function $compareDwithC_upMin_dm_phaseB_kato()$. Tale funzione è molto simile a quella mostrata in 5.2.2, con l'aggiunta di qualche ottimizzazione per questo caso specifico. Definiamo $m$ blocchi di $NT$ thread, dove ogni blocco $B_j$ è dedicato ad un singolo punto $q_j \in C$. Nel caso in cui si verifichi la condizione $Sum(LNNC[q_j]) < Min(Top)$ si termina immediatamente la kernel function, in quanto sicuramente il punto $q_j$ non potrà essere inserito nell'heap $Top$, dato che durante l'esecuzione della funzione il peso di $q_j$ potrà solamente al più diminuire. Ogni blocco $B_j$, se viene superato questo filtro, effettua una prima copia dell'heap $LNNC[q_j]$ all'interno di un proprio max-heap $H_j$ di dimensione $k$, allocato in shared memory. Associamo ad ogni thread un buffer, di dimensione pari a $buffersize$. I thread di $B_j$ leggono in parallelo le distanze nel chunk ed inseriscono nel buffer solamente le coppie $\langle p_z, \delta \rangle$ (con $p_z \in X_I$ e $\delta = dist(q_j, p_z)$) tali che $\delta$ è stata calcolata durante la fase A (ovvero salvata nel chunk con un valore diverso da $+\infty$) e $\delta < \sigma$, dove $\langle s, \sigma \rangle$ è l'elemento radice di $H_j$, fino a riempire il buffer. A questo punto un solo thread si occupa di prelevare in sequenza gli elementi dai vari buffer e di inserirli in $H_j$, con operazioni di tipo $updateMin$. Il procedimento viene iterato fino a quando i thread non hanno letto l'intera $j$-esima riga del chunk oppure il peso di $q_j$ scende al di sotto del valore minimo di soglia $Min(Top)$. Infine si ha l'aggiornamento di $LNNC[q_j]$ con il contenuto di $H_j$.

La procedura eseguita dalla kernel function è la seguente (in pseudo-codice):

```
compareDwithC_upMin_dm_phaseB_kato(k, D, C, LNNC, Min(Top),
                                                  I, currentChunk){
        tid := threadIdx.x
        j := blockIdx.x
        i := tid
        i_idx := I * CSIZE + i

        <shared> H_j
        <shared> Buff_0, Buff_1,..., Buff_{NT-1},

        if (Sum(LNNC[q_j]) < Min(Top), con q_j ∈ C){
                return
        }

        if (tid = 0) H_j := LNNC[q_j]
        __syncthreads()

        Buff_tid := ∅
        while (i < CSIZE AND i_idx < d AND Sum(LNNC[q_j]) ≥ Min(Top)) {
                for e := 0 to bufferSize {
                        δ := currentChunk[j][i]
                        if (δ < +∞ AND (|H_j| < k OR δ < σ,
                                        con ⟨s,σ⟩ radice di H_j)) {
                                addToBuffer(Buff_tid, ⟨p_i_idx, δ⟩), con p_i_idx ∈ D
                        }
                        i := i + NT
                        i_idx := i_idx + NT
                }
                __syncthreads()
                if (tid = 0) {
                        for t := 0 to NT {
                                for each ⟨p_i_idx, δ⟩ in Buff_t {
                                        updateMin(H_j, ⟨p_i_idx, δ⟩)
                                }
                                Buff_t := ∅
                        }
                }
                __syncthreads()
        }
        if (tid = 0) LNNC[q_j] := H_j
}
```

#### 6.3.1.3 Modalità con matrice delle distanze – fase B con Heap complete reduction

La fase B per il chunk $\Delta_I$, nel caso di utilizzo del metodo heap complete reduction, è eseguita tramite la kernel function $compareDwithC_upMin_dm_phaseB_red()$. Anche in questo caso la funzione è molto simile a quella mostrata in 5.2.3, con l'aggiunta di qualche piccola ottimizzazione. Predisponiamo $m$ blocchi di $NT$ thread, dove ogni blocco è dedicato ad un singolo punto $q_j \in C$ e associamo ad ogni thread $T_i$ un proprio max-heap $H_i$ di dimensione $k$, allocato in shared memory. Come per la tecnica precedente, se si verifica la condizione $Sum(LNNC[q_j]) < Min(Top)$ si termina immediatamente la kernel function, in quanto sicuramente il punto $q_j$ non potrà essere inserito nell'heap $Top$. Se si supera tale filtro, ogni thread $T_i$ del blocco $B_j$ considera le distanze tra $q_j$ e $CSIZE/NT$ punti $p_z \in X_I$ e trova le rispettive minori $k$ distanze, inserendo le rispettive coppie $\langle p_z, \delta\rangle$ in $H_i$, con $\delta = dist(q_j, p_z)$, sfruttando l'operazione $updateMin$. Inoltre le operazioni $updateMin$ vengono effettuate solamente per le coppie per cui la rispettiva distanza $\delta$ è stata calcolata durante la fase A (ovvero salvata nel chunk con un valore diverso da $+\infty$) e per cui $\delta < \sigma$, dove $\langle s, \sigma\rangle$ è l'elemento radice di $LNNC[q_j]$. Per ogni blocco vengono quindi fusi gli heap degli $NT$ thread, sfruttando la tecnica della riduzione parallela, tramite $\log_2(NT)$ step. Come ultima operazione si ha la fusione di $LNNC[q_j]$ con l'heap risultante dalla riduzione.

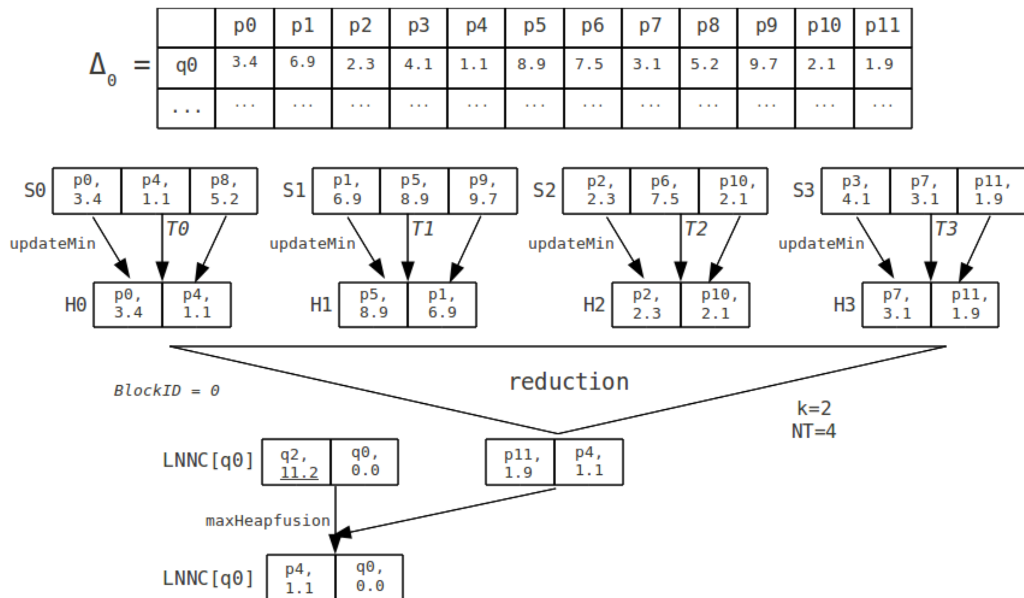



**Figura 6-4 Fase B con heap complete reduction**

La procedura eseguita dalla kernel function è la seguente (in pseudo-codice):

```
compareDwithC_upMin_dm_phaseB_red(k, D, C, LNNC, Min(Top),
                                                    I, currentChunk){
        tid := threadIdx.x
        j := blockIdx.x
        i := tid
        i_idx := I * CSIZE + i

        <shared> H_0, H_1,..., H_{NT-1}
        if (Sum(LNNC[q_j]) < Min(Top), con q_j ∈ C){
                return
        }

        H_tid := ∅

        while (i < CSIZE AND i_idx < d) {
                δ := currentChunk[j][i]
                if (δ < +∞  AND (|LNNC[q_j]| < k OR δ < σ,
                                        con ⟨s,σ⟩ radice di LNNC[q_j])) {
                        updateMin(H_tid, ⟨p_i_idx, δ⟩), con p_i_idx ∈ D
                }
                i := i + NT
                i_idx := i_idx + NT
        }
        __syncthreads()

        //riduzione in shared mem
        if(NT ≥ 512) {
                if(tid < 256) maxheapFusion(H_tid, H_tid+256)
                __syncthreads()
        }
        if(NT ≥ 256) {
                if(tid < 128) maxheapFusion(H_tid, H_tid+128)
                __syncthreads()
        }
        if(NT ≥ 128) {
                if(tid < 64) maxheapFusion(H_tid, H_tid+64)
                __syncthreads()
        }
        if(NT ≥ 64) {
                if(tid < 32) maxheapFusion(H_tid, H_tid+32)
                //per tid < 32 i threads sono nello stesso warp
                //e sempre sincronizzati
        }
        if(NT ≥ 32) {
                if(tid < 16) maxheapFusion(H_tid, H_tid+16)
        }
        if(NT ≥ 16) {
                if(tid < 8) maxheapFusion(H_tid, H_tid+8)
        }
        if(NT ≥ 8) {
                if(tid < 4) maxheapFusion(H_tid, H_tid+4)
        }
        if(NT ≥ 4) {
                if(tid < 2) maxheapFusion(H_tid, H_tid+2)
        }
        if(NT ≥ 2) {
                if(tid < 1) maxheapFusion(H_tid, H_tid+1)
        }
        if (tid = 0) maxheapFusion(LNNC[q_j], H_0)
}
```

#### 6.3.1.4 Modalità con matrice delle distanze – fase B con Heap complete reduction multistep

E' possibile infine combinare alcuni vantaggi della tecnica di Kato con quelli di heap complete reduction. Il metodo di Kato, per un determinato blocco $B_j$, riesce a fermare preventivamente l'esecuzione della kernel function nel caso in cui il valore di $Sum(LNNC[q_j])$ scenda al di sotto del valore di soglia $Min(Top)$. La tecnica con riduzione, come implementata nel punto precedente, non può trarre vantaggio da questo filtro. Rispetto al metodo di Kato tale tecnica permette però un'esecuzione più efficiente delle operazioni $updateMin$ sull'heap $LNNC[q_j]$, grazie alla riduzione parallela. E' possibile combinare i due effetti positivi, grazie ad una nuova soluzione che possiamo definire come *heap complete reduction multistep*.

Predisponiamo, come nel caso precedente, $m$ blocchi di $NT$ thread, dove ogni blocco è dedicato ad un singolo punto $q_j \in C$ e associamo ad ogni thread $T_i$ un proprio max-heap $H_i$ di dimensione $k$, allocato in shared memory. Anziché far analizzare ai thread del blocco tutti i $CSIZE$ punti $p_z \in X_I$ e poi effettuare un unico processo di riduzione sugli heap, lavoriamo in più step. Ad ogni step i thread di $B_j$ considerano solamente $StepSIZE$ punti (con $StepSIZE \leq CSIZE$) alla volta, chiamando, se necessario, le rispettive operazioni $updateMin$ sui propri heap $H_i$. Dopo aver letto $StepSIZE$ distanze, viene effettuata la riduzione parallela degli heap ed infine fuso $LNNC[q_j]$ con l'heap risultante dalla riduzione. Nel caso in cui si verifichi la condizione $Sum(LNNC[q_j]) < Min(Top)$ si interrompe l'esecuzione della kernel function per i thread del blocco $B_j$, in caso contrario si prosegue analizzando i successivi $StepSIZE$ punti. Tale procedimento viene quindi iterato fino a quando non vengono analizzati tutti i $CSIZE$ punti in $X_I$, oppure il peso di $q_j$ scende al di sotto di $Min(Top)$.

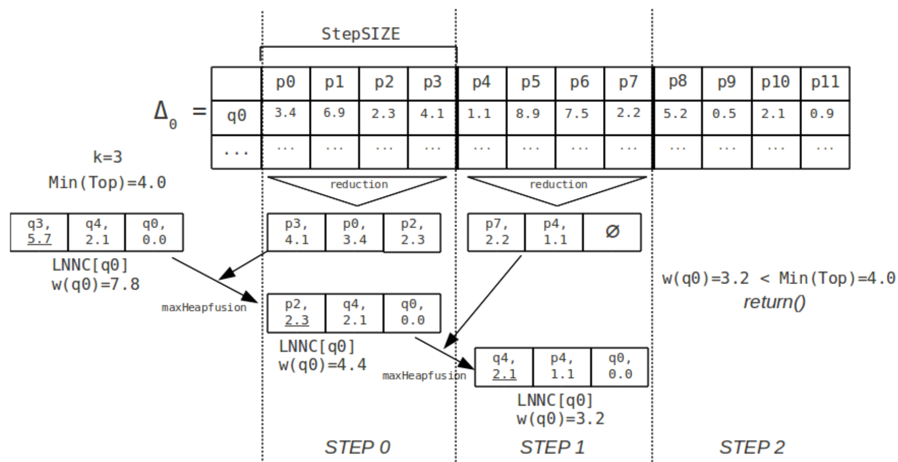



**Figura 6-5 Fase B con heap complete reduction multistep**

La procedura eseguita dalla kernel function è la seguente (in pseudo-codice):

```
compareDwithC_upMin_dm_phaseB_redMS(k, D, C, LNNC, Min(Top),
                                                    I, currentChunk){
        tid := threadIdx.x
        j := blockIdx.x
        i := tid
        i_idx := I * CSIZE + i

        <shared> H_0, H_1,..., H_{NT-1}

        step := 0
        numOfSteps := ⌈Min{CSIZE; d} / StepSIZE⌉
        while(step < numOfSteps AND Sum(LNNC[q_j]) ≥ Min(Top)){
            H_tid := ∅
            while (i < step · StepSIZE AND i < CSIZE AND i_idx < d) {
               δ := currentChunk[j][i]
               if (δ < +∞ AND (|LNNC[q_j]| < k OR δ < σ,
                                       con ⟨s,σ⟩ radice di LNNC[q_j])) {
                       updateMin(H_tid, ⟨p_i_idx, δ⟩), con p_i_idx ∈ D
               }
               i := i + NT
               i_idx := i_idx + NT
            }
            __syncthreads()

            //riduzione in shared mem
            if(NT ≥ 512) {
               if(tid < 256) maxheapFusion(H_tid, H_tid+256)
               __syncthreads()
            }
            if(NT ≥ 256) {
               if(tid < 128) maxheapFusion(H_tid, H_tid+128)
               __syncthreads()
            }
            if(NT ≥ 128) {
               if(tid < 64) maxheapFusion(H_tid, H_tid+64)
               __syncthreads()
            }
            if(NT ≥ 64) {
               if(tid < 32) maxheapFusion(H_tid, H_tid+32)
               //per tid < 32 i threads sono nello stesso warp
               //e sempre sincronizzati
            }
            if(NT ≥ 32) {
               if(tid < 16) maxheapFusion(H_tid, H_tid+16)
            }
            if(NT ≥ 16) {
               if(tid < 8) maxheapFusion(H_tid, H_tid+8)
            }
            if(NT ≥ 8) {
               if(tid < 4) maxheapFusion(H_tid, H_tid+4)
            }
            if(NT ≥ 4) {
               if(tid < 2) maxheapFusion(H_tid, H_tid+2)
            }
            if(NT ≥ 2) {
               if(tid < 1) maxheapFusion(H_tid, H_tid+1)
            }
            if (tid = 0) maxheapFusion(LNNC[q_j], H_0)

            step := step + 1
        }
}
```

### 6.3.2 Modalità senza matrice delle distanze

L'utilizzo della matrice delle distanze, con suddivisione in più chunk, presenta per contro alcuni problemi collaterali. Date le elevate dimensioni della matrice, sono necessari molti accessi in scrittura (durante la fase A) e molti accessi in lettura (durante la fase B), che costituiscono un overhead non indifferente rispetto ad una soluzione sequenziale che non richiede tali operazioni. Inoltre, nel caso in cui il valore del parametro $m$ sia relativamente piccolo, la fase B soffre di un potenziale problema di sottoutilizzo dell'hardware della GPU. Durante tale fase, per tutti i tre metodi proposti, si definiscono $m$ blocchi di thread, un blocco per ogni punto candidato. Nel caso in cui il valore di $m$ sia inferiore al numero minimo di blocchi richiesti per ottenere la massima occupancy dei multiprocessori, non si raggiunge la piena efficienza. Tale fenomeno diventa ancora più marcato nel caso in cui $m$ sia addirittura inferiore al numero di multiprocessori della GPU. Per cercare di risolvere questi due problemi, introduciamo un nuovo metodo, che deriva dall'approccio Heap reduction multi-block, presentato in 4.2. Come in tale soluzione, prevediamo due distinti step.

Nel primo step (fase A) lavoriamo considerando un singolo punto candidato $q_j \in C$ alla volta. Predisponiamo $NB_A$ blocchi di $NT_A$ thread e associamo ad ogni thread $T_z$ (con $z = 0,1,\ldots, NB_A \cdot NT_A - 1$) un proprio max-heap $H_z$ di dimensione $k$, allocato in shared memory, contenente le coppie $\langle p_i, \delta \rangle$, con $\delta = dist(q_j, p_i)$ e $p_i \in D$. Ogni $T_z$ calcola la distanza $\delta$ tra $q_j$ e $d/(NB_A \cdot NT_A)$ punti $p_i \in D$ ed esegue l'operazione $updateMin$ su $NN[p_i]$ e $H_z$, per le rispettive coppie $\langle q_j, \delta \rangle$ e $\langle p_i, \delta \rangle$. Come per l'algoritmo sequenziale, il calcolo della distanza e le operazioni sugli heap vengono eseguite solamente se vale la condizione $\max\{Sum(NN[p_i]), Sum(LNNC[q_j])\} \geq Min(Top)$. Inoltre l'$updateMin$ su $H_z$ avviene solamente per le coppie in cui $\delta < \sigma$, dove $\langle \mathrm{s}, \sigma \rangle$ è l'elemento radice di $LNNC[q_j]$. Per ogni blocco viene quindi effettuata la fusione dei vari heap $H_z$, tramite la tecnica della riduzione parallela. L'heap risultante per ogni blocco $B_t$ viene salvato a partire dalla riga $j$ e colonna $t$ di una matrice di heap $GHMax$, allocata in global memory.

Per il secondo step (fase B) utilizziamo $m$ blocchi di $NT_B$ thread, uno per ogni $q_j \in C$. Associamo ad ogni thread $T_z'$ (con $z = 0,1,\ldots, NT_B - 1$) del blocco $B_j'$ un max-heap $H_z'$ di dimensione $k$, in shared memory. Ogni $T_z'$ di $B_j'$ esegue inizialmente la fusione di $NB_A/NT_B$ heap salvati in global memory nella $j$-esima riga di $GHMax$ con il

proprio heap $H_z{}'$. Successivamente vengono fusi gli heap $H_0{}', \dots, H_{NT_B-1}{}'$ tramite la riduzione parallela ed infine l'ultimo heap risultante viene fuso con $LNNC[q_j]$.

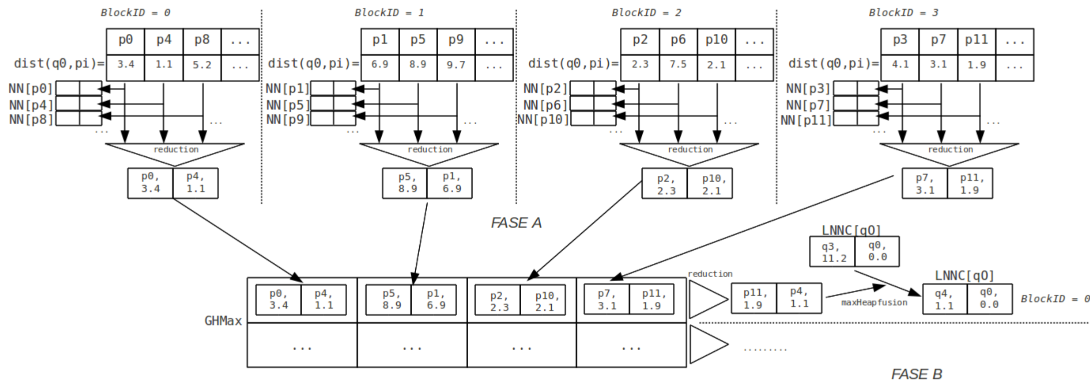



**Figura 6-6 Fase di confronto $D$ con $C$ - upMin senza matrice delle distanze**

$GHMax$ ha dimensione $m \times NB_A \times k$, mentre la matrice delle distanze del precedente approccio ha dimensione $m \times d$. Per $d$ sufficientemente elevato si ha che $NB_A \cdot k \ll d$, di conseguenza gli accessi in global memory (in lettura e in scrittura) effettuati da questa soluzione sono in numero molto inferiore. Inoltre nel primo step i multiprocessori riescono a raggiungere sempre la massima occupancy possibile, indipendentemente dal valore di $m$, a patto di dimensionare correttamente $NB_A$ e $NT_A$. Nel secondo step, per $m$ piccolo, si ha un minor sfruttamento dell'hardware, ma essendo la prima fase nettamente dominante sulla seconda, questo fenomeno non incide particolarmente sulle performance.

La fase *upMin*, in termini di operazioni di coordinamento eseguite dalla CPU nel caso di utilizzo della modalità senza matrice delle distanze, può quindi essere indicata tramite la seguente funzione (mostrata in pseudo-codice):

```
compareDwithC_upMin_ndm(k, dist, D, C, NN, LNNC, Min(Top), GHMax){
        for each q_j in C{
                launchOnGPU(
                        compareDwithC_upMin_ndm_phaseA(k, dist, D, C, NN,
                                                LNNC, Min(Top), j, GHMax))
                synchWithGPU()
        }
        launchOnGPU(compareDwithC_upMin_ndm_phaseB(k, LNNC, GHMax))
        synchWithGPU()
}
```

Le procedure eseguite dalle due kernel function sono le seguenti (in pseudo-codice):

```
compareDwithC_upMin_ndm_phaseA
(k, dist, D, C, NN, LNNC, Min(Top), j, GHMax) {
    tid := threadIdx.x
    bid := blockIdx.x
    <shared> H_0, H_1,..., H_{NT_A-1}
    H_tid := ∅

    i := bid · NT_A + tid
    while (i < d) {
     if(max{Sum(NN[p_i]), Sum(LNNC[q_j])} ≥
        Min(Top), con p_i ∈ D, q_j ∈ C) {
          δ := dist(p_i, q_j)
          updateMin(NN[p_i], ⟨q_j, δ⟩ )
          if ( |LNNC[q_j]| < k OR δ < σ,
            ⟨s,σ⟩ radice di LNNC[q_j]){
              updateMin(H_tid, ⟨p_i, δ⟩ )
          }
     }
     i := i + NT_A · NB_A
    }
    __syncthreads()
    //riduzione in shared mem
    if(NT_A ≥ 512) {
        if(tid < 256)
          maxheapFusion(H_tid, H_tid+256)
        __syncthreads()
    }
    if(NT_A ≥ 256) {
        if(tid < 128)
          maxheapFusion(H_tid, H_tid+128)
        __syncthreads()
    }
    if(NT_A ≥ 128) {
        if(tid < 64)
          maxheapFusion(H_tid, H_tid+64)
        __syncthreads()
    }
    if(NT_A ≥ 64) {
        if(tid < 32)
          maxheapFusion(H_tid, H_tid+32)
    }
    if(NT_A ≥ 32) {
        if(tid < 16)
          maxheapFusion(H_tid, H_tid+16)
    }
    if(NT_A ≥ 16) {
        if(tid < 8)
          maxheapFusion(H_tid, H_tid+8)
    }
    if(NT_A ≥ 8) {
        if(tid < 4)
          maxheapFusion(H_tid, H_tid+4)
    }
    if(NT_A ≥ 4) {
        if(tid < 2)
          maxheapFusion(H_tid, H_tid+2)
    }
    if(NT_A ≥ 2) {
        if(tid < 1)
          maxheapFusion(H_tid, H_tid+1)
    }
    if (tid = 0) GHMax[j][bid] := H_0
}
```

```
compareDwithC_upMin_ndm_phaseB
(k, LNNC, GHMax) {
    tid := threadIdx.x
    j := blockIdx.x
    <shared> H_0, H_1,..., H_{NT_B-1}
    H_tid := ∅

    i := tid
    while (i < NB_A) {
        maxheapFusion(H_tid, GHMax[j][i])
        i := i + NT_B
    }
    __syncthreads()

    //riduzione in shared mem
    if(NT_B ≥ 512) {
        if(tid < 256)
          maxheapFusion(H_tid, H_tid+256)
        __syncthreads()
    }
    if(NT_B ≥ 256) {
        if(tid < 128)
          maxheapFusion(H_tid, H_tid+128)
        __syncthreads()
    }
    if(NT_B ≥ 128) {
        if(tid < 64)
          maxheapFusion(H_tid, H_tid+64)
        __syncthreads()
    }
    if(NT_B ≥ 64) {
        if(tid < 32)
          maxheapFusion(H_tid, H_tid+32)
    }
    if(NT_B ≥ 32) {
        if(tid < 16)
          maxheapFusion(H_tid, H_tid+16)
    }
    if(NT_B ≥ 16) {
        if(tid < 8)
          maxheapFusion(H_tid, H_tid+8)
    }
    if(NT_B ≥ 8) {
        if(tid < 4)
          maxheapFusion(H_tid, H_tid+4)
    }
    if(NT_B ≥ 4) {
        if(tid < 2)
          maxheapFusion(H_tid, H_tid+2)
    }
    if(NT_B ≥ 2) {
        if(tid < 1)
          maxheapFusion(H_tid, H_tid+1)
    }
    if (tid = 0)
        maxheapFusion(LNNC[q_j], H_0)
}
```

#### 6.3.2.1 Modalità senza matrice delle distanze – variante hgm_a1a2

Nel caso in cui il parametro $k$ sia posto ad un valore elevato, il fatto di avere i max-heap $H_z$ allocati in shared memory può portare ad una grossa limitazione dell'occupancy dei multiprocessori della GPU, per l'elevata quantità di shared memory richiesta da ogni blocco di thread. Tale fenomeno diventa ancora più rilevante nel caso in cui le kernel function abbiano bisogno di effettuare molti accessi in global memory, in quanto l'unico modo che hanno gli SM di mascherare le latenze degli accessi è il fatto di avere molti thread disponibili per l'esecuzione. Nel caso di un basso livello di occupancy, il numero di thread presenti in ogni SM diventa insufficiente per nascondere le latenze. Nella fase A dell'approccio senza matrice delle distanze si ha un numero molto elevato di letture e scritture in global memory, sia per le operazioni di calcolo delle distanze, sia per le operazioni sugli heap $NN$, ed una bassa occupancy può quindi limitare fortemente le perfomance. Per $k$ elevato possiamo quindi proporre una semplice variante alla precedente soluzione, che tenga conto di questo problema.

Spezziamo la fase A in due distinte sottofasi A1 e A2, a cui corrispondo due diverse kernel function. Nella fase A1 predisponiamo $NB_{A1}$ blocchi di $NT_{A1}$ thread e associamo ad ogni thread $T_z$ (con $z = 0,1,\ldots, NB_{A1} \cdot NT_{A1} - 1$) un proprio max-heap $GH_z$ di dimensione $k$, allocandolo però in global memory. Ogni $T_z$ calcola la distanza $\delta$ tra $q_j$ e $d/(NB_{A1} \cdot NT_{A1})$ punti $p_i \in D$ ed esegue l'operazione $updateMin$ su $NN[p_i]$ e $GH_z$, per le rispettive coppie $\langle q_j, \delta \rangle$ e $\langle p_i, \delta \rangle$, similarmente alla fase A. Il processo di riduzione non viene però eseguito sugli heap in global memory, in quanto altrimenti risulterebbe molto inefficiente, poiché gli accessi in global memory non risponderebbero ai requisiti richiesti per un *coalesced access*. Di conseguenza si termina la fase A1 e si esegue la fase A2. Per questa fase predisponiamo $NB_{A2}$ blocchi di $NT_{A2}$ thread e associamo ad ogni thread $T_z$ (con $z = 0,1,\ldots, NB_{A2} \cdot NT_{A2} - 1$) un max-heap $H_z$ di dimensione $k$, allocato in shared memory. Ogni thread $T_z$ di A2 esegue inizialmente la fusione di $(NB_{A1}.NT_{A1})/(NB_{A2}.NT_{A2})$ heap salvati in global memory in A1 con il proprio heap $H_z$ in shared memory. Successivamente vengono fusi gli heap $H_z$ per ogni blocco, tramite la riduzione parallela e l'heap risultante per ogni blocco $B_t$ viene salvato a partire dalla riga $j$ e colonna $t$ di $GHMax$. Viene poi eseguita la fase B, con la stessa modalità della soluzione precedente.

Possiamo quindi riscrivere le operazioni di coordinamento eseguite dalla CPU, nel caso di utilizzo della variante hgm_a1a2, nel modo seguente (in pseudo-codice):

```
compareDwithC_upMin_ndm_hgmA1A2(k, dist, D, C, NN, LNNC,
                                              Min(Top), GH, GHMax){
        for each q_j in C{
                launchOnGPU(
                   compareDwithC_upMin_ndm_hgmA1A2_phaseA1(k, dist, D, C, NN,
                                                   LNNC, Min(Top), j, GH))
                synchWithGPU()
                launchOnGPU(
        compareDwithC_upMin_ndm_hgmA1A2_phaseA2(k, j, GH, GHMax))
                synchWithGPU()
        }
        launchOnGPU(compareDwithC_upMin_ndm_phaseB(k, LNNC, GHMax))
        synchWithGPU()
}
```

Le procedure eseguite da $compareDwithC_upMin_ndm_hgmA1A2_phaseA1$ e $compareDwithC_upMin_ndm_hgmA1A2_phaseA2$ sono le seguenti (in pseudo-codice):

```
compareDwithC_upMin_ndm_hgmA1A2_
phaseA1 (k, dist, D, C, NN, LNNC,
                         Min(Top), j, GH) {

   tid := threadIdx.x
   bid := blockIdx.x

   GH_tid := ∅

   i := bid · NT_A1 + tid
   while (i < d) {
     if(max{Sum(NN[p_i]), Sum(LNNC[q_j])} ≥
        Min(Top), con p_i ∈ D, q_j ∈ C) {
          δ := dist(p_i, q_j)
          updateMin(NN[p_i], ⟨q_j, δ⟩ )
          if ( |LNNC[q_j]| < k OR δ < σ,
           ⟨s, σ⟩ radice di LNNC[q_j]) {
               updateMin(GH_tid, ⟨p_i, δ⟩ )
          }
     }
        i := i + NT_A1 · NB_A1
   }
}
```

```
compareDwithC_upMin_ndm_hgmA1A2_
phaseA2 (k, j, GH, GHMax) {

   tid := threadIdx.x
   bid := blockIdx.x
   <shared> H_0, H_1, ..., H_(NT_B-1)
   H_tid := ∅

   i := tid
   while (i < NT_A1 · NB_A1) {
        maxheapFusion(H_tid, GH[i])
        i := i + NT_A2 · NB_A2
   }
   __syncthreads()

   //riduzione in shared mem
   if(NT_A2 ≥ 512) {
        if(tid < 256)
          maxheapFusion(H_tid, H_tid+256)
        __syncthreads()
   }
   if(NT_A2 ≥ 256) {
        if(tid < 128)
          maxheapFusion(H_tid, H_tid+128)
        __syncthreads()
   }
   if(NT_A2 ≥ 128) {
        if(tid < 64)
          maxheapFusion(H_tid, H_tid+64)
        __syncthreads()
   }
   .......
   if(NT_A2 ≥ 2) {
        if(tid < 1)
          maxheapFusion(H_tid, H_tid+1)
   }
   if (tid = 0) GHMax[j][bid] := H_0
}
```

### 6.4 Fase di confronto $D$ con $C$ - upMax

Nella seconda fase del confronto $D$ con $C$, compito della GPU è eseguire l'aggiornamento dell'heap $LC$, per permettere la successiva selezione dei nuovi punti candidati. Inoltre, nel caso in cui il peso di un punto $p_i \in D$ scenda al di sotto della soglia $Min(Top)$, tale punto viene dichiarato come non attivo e quindi non più considerato in futuro per l'elezione a punto candidato.

Per queste operazioni si può seguire un approccio molto simile a quello proposto nel paragrafo 5.4, che si basa sul metodo *Heap reduction multi-block*. Consideriamo ora il vettore riga

$$w = [Sum(NN[p_0]), Sum(NN[p_1]), \dots, Sum(NN[p_{d-1}])],$$

$$\text{con } p_0, p_1, \dots, p_{d-1} \in D.$$

La fase di aggiornamento dell'heap $LC$, ovvero di selezione degli $m$ punti $p_i \in D$ dotati del maggior weight upper bound $Sum(NN[p_i])$, si può ricondurre al problema di individuare gli $m$ elementi in $w$ di maggiore valore. Il problema da affrontare è il medesimo mostrato in 5.4, con l'unica differenza che in questo caso occorre aggiornare l'heap $LC$ (non l'heap $Top$) ed è necessaria qualche azione aggiuntiva per l'etichettamento dei punti come attivi o non attivi.

Anche in questo caso lavoriamo in due fasi, tramite due distinte kernel function: $compareDwithC_upMax_phaseA()$ e $compareDwithC_upMax_phaseB()$. Per la prima fase predisponiamo $NB_A$ blocchi di $NT_A$ thread e associamo ad ogni thread $T_z$ (con $z = 0,1, \dots, NB_A \cdot NT_A - 1$) un proprio min-heap $H_z$ di dimensione $m$, allocato in shared memory, contenente le coppie $\langle p_i, w_k(p_i, D)\rangle$, con $p_i \in D$. Ogni $T_z$ si occupa inizialmente della selezione degli $m$ punti di maggior peso tra $d/(NB_A \cdot NT_A)$ elementi in $w$, sfruttando l'operazione $updateMax$ su $H_z$. Durante tale operazione vengono considerati solamente i punti $p_i \in D$ dichiarati come attivi e per cui valga la condizione $Sum(NN[p_i]) \geq Min(Top)$. Inoltre, se il weight upper bound di $p_i$ si trova al di sotto della soglia $Min(Top)$, il punto viene etichettato come non attivo. Per ogni blocco viene quindi eseguita la fusione dei vari heap, tramite la tecnica della riduzione parallela e l'heap risultante per ogni blocco viene salvato in global memory. Per la seconda fase utilizziamo un solo blocco di $NT_B$ thread ed anche in questo caso associamo ad ogni thread $T_z{}'$ (con $z = 0,1, \dots, NT_B - 1$) un min-heap $H_z{}'$ di dimensione $m$, in shared memory. Ogni $T_z{}'$ esegue la fusione di $NB_A/NT_B$ heap salvati in global memory con il proprio heap $H_z{}'$ e successivamente vengono fusi gli heap $H_0{}', \dots, H_{NT_B-1}{}'$ tramite la riduzione parallela. L'heap risultante coincide con $LC$.

Le procedure eseguite dalle due kernel function sono le seguenti (in pseudo-codice):

```
compareDwithC_upMax_ phaseA
 (m, D, NN, GHMin, Min(Top)) {

   tid := threadIdx.x
   bid := blockIdx.x

   <shared> H_0, H_1,..., H_{NT_A−1}
   H_tid := ∅

   i := bid · NT_A + tid
   while (i < d) {
    if(p_i.active = true AND
     Sum(NN[p_i]) ≥ Min(Top), con p_i ∈ D){
        updateMax(H_tid, ⟨p_i, Sum(NN[p_i])⟩)
    }else{
        p_i.active := false
    }
        i := i + NT_A · NB_A
   }
   __syncthreads()

   //riduzione in shared mem
   if(NT_A ≥ 512) {
        if(tid < 256)
          minheapFusion(H_tid, H_{tid+256})
        __syncthreads()
   }
   if(NT_A ≥ 256) {
        if(tid < 128)
          minheapFusion(H_tid, H_{tid+128})
        __syncthreads()
   }
   if(NT_A ≥ 128) {
        if(tid < 64)
          minheapFusion(H_tid, H_{tid+64})
        __syncthreads()
   }
   if(NT_A ≥ 64) {
        if(tid < 32)
          minheapFusion(H_tid, H_{tid+32})
   }
   if(NT_A ≥ 32) {
        if(tid < 16)
          minheapFusion(H_tid, H_{tid+16})
   }
   if(NT_A ≥ 16) {
        if(tid < 8)
          minheapFusion(H_tid, H_{tid+8})
   }
   if(NT_A ≥ 8) {
        if(tid < 4)
          minheapFusion(H_tid, H_{tid+4})
   }
   if(NT_A ≥ 4) {
        if(tid < 2)
          minheapFusion(H_tid, H_{tid+2})
   }
   if(NT_A ≥ 2) {
        if(tid < 1)
          minheapFusion(H_tid, H_{tid+1})
   }
   if (tid = 0) GHMin[bid] := H_0
}
```

```
compareDwithC_upMax_ phaseB
 (m, LC, GHMin) {

   tid := threadIdx.x


   <shared> H_0, H_1,..., H_{NT_B−1}
   H_tid := ∅

   i := tid
   while (i < NB_A) {
        minheapFusion(H_tid, GHMin[i])
        i := i + NT_B
   }
   __syncthreads()

   //riduzione in shared mem
   if(NT_B ≥ 512) {
        if(tid < 256)
          minheapFusion(H_tid, H_{tid+256})
        __syncthreads()
   }
   if(NT_B ≥ 256) {
        if(tid < 128)
          minheapFusion(H_tid, H_{tid+128})
        __syncthreads()
   }
   if(NT_B ≥ 128) {
        if(tid < 64)
          minheapFusion(H_tid, H_{tid+64})
        __syncthreads()
   }
   if(NT_B ≥ 64) {
        if(tid < 32)
          minheapFusion(H_tid, H_{tid+32})
   }
   if(NT_B ≥ 32) {
        if(tid < 16)
          minheapFusion(H_tid, H_{tid+16})
   }
   if(NT_B ≥ 16) {
        if(tid < 8)
          minheapFusion(H_tid, H_{tid+8})
   }
   if(NT_B ≥ 8) {
        if(tid < 4)
          minheapFusion(H_tid, H_{tid+4})
   }
   if(NT_B ≥ 4) {
        if(tid < 2)
          minheapFusion(H_tid, H_{tid+2})
   }
   if(NT_B ≥ 2) {
        if(tid < 1)
          minheapFusion(H_tid, H_{tid+1})
   }
   if (tid = 0) LC := H_0
}
```

### 6.5 Fase di selezione dei nuovi candidati

L'ultima fase eseguita dalla GPU è quella di selezione dei nuovi punti candidati. Il suo compito consiste semplicemente nell'eseguire la copia dei punti presenti nell'heap $LC$ all'interno dell'insieme $C$, predisponendo il nuovo insieme di candidati per l'iterazione successiva. Tale operazione è svolta tramite la kernel function $candSelect()$, impiegando $m$ thread $T_j$, dove ogni thread $T_j$ si occupa semplicemente della copia del $j$-esimo elemento di $LC$ in $C$.

Implementando l'insieme $C$ come un vettore di $m$ punti allocato in global memory, la kernel function eseguita dalla GPU è la seguente (in pseudo-codice):

```
candSelect(C,LC) {
        j := threadIdx.x
        C[j] := getPoint(j,LC)
}
```

### 6.6 Complessità computazionale

Per il calcolo della complessità computazionale consideriamo innanzitutto le singole fasi eseguite dalla GPU ad ogni iterazione. Ipotizziamo che la GPU sia in grado di eseguire contemporaneamente al più $NP$ thread e che, per semplicità, $d$ sia multiplo di $NP$. Nel caso in cui siano presenti nella GPU un numero superiore di thread, l'esecuzione di essi viene sequenzializzata, suddividendoli in gruppi di $NP$ thread alla volta.

La fase di inizializzazione presenta una complessità pari a $O(m \cdot \log(m))$, in quanto la funzione $findSubsitutes()$ e il sorting degli indici dei punti in $C$ (con algoritmo di tipo heapsort) richiedono un tempo proporzionale a $O(m \cdot \log(m))$.

Per la fase di confronto $C$ con $C$ si hanno $m$ thread, dove ogni thread esegue $m$ calcoli di distanza ed $m$ operazioni di tipo $updateMin$. Ipotizzando $a$ pari alla dimensionalità dei punti del dataset (ovvero il numero di coordinate dei punti), tale fase presenta quindi una complessità pari a $O\left(\left\lceil\frac{m}{NP}\right\rceil \cdot m(a + \log(k))\right) = O(m(a + \log(k)))$, assumendo $NP > m$.

Per la fase di confronto $D$ con $C$ – upMin occorre distinguere tra le diverse modalità. Nel caso di scelta della tecnica con matrice delle distanze, ipotizziamo per semplicità l'utilizzo di un solo chunk di dimensione $m \times d$; le conclusioni effettuate sotto tale ipotesi sono comunque valide anche nel caso generale di suddivisione della matrice in più chunk. La fase A presenta una complessità pari a $O\left(\frac{d}{NP} \cdot m(a + \log(k))\right)$, in

quanto si hanno $d$ thread, dove ogni thread esegue $m$ calcoli di distanza ed $m$ operazioni di tipo $updateMin$. Per la fase B, nel caso di utilizzo della modalità heap complete reduction o heap complete reduction multistep, si definiscono $m$ blocchi di $NT$ thread dove ogni thread, prima della riduzione, effettua al più $d/NT$ operazioni $updateMin$ sul proprio heap. Di conseguenza, per le conclusioni effettuate nel capitolo 4, tale fase presenta una complessità pari a $O\left(\left\lceil\frac{m \cdot NT}{NP}\right\rceil \cdot \frac{d}{NT} \cdot \log(k)\right) = O\left(\frac{d}{NP} \cdot m \cdot \log(k)\right)$, ipotizzando $m \cdot NT$ multiplo di $NP$. Con la tecnica di Kato la complessità teorica è più elevata e pari a $O\left(\left\lceil\frac{m \cdot NT}{NP}\right\rceil \cdot d \cdot \log(k)\right)$, in quanto nel caso peggiore si hanno $d$ operazioni *updateMin* sequenziali per ogni blocco, svolte dall'unico thread che lavora sull'heap, indipendentemente dal valore di $NT$. Per il calcolo della complessità della fase B assumiamo di utilizzare una delle due tecniche che forniscono la minor complessità teorica (heap complete reduction o heap complete reduction multistep). Di conseguenza la complessità della fase di confronto $D$ con $C -$ upMin, nel caso di utilizzo della modalità con matrice delle distanze, è pari a

$$O\left(\frac{d}{NP} \cdot m(a + \log(k)) + \frac{d}{NP} \cdot m \cdot \log(k)\right) = O\left(\frac{d}{NP} \cdot m(a + 2\log(k))\right).$$

Con la modalità senza matrice delle distanze si hanno $NB_A$ blocchi di $NT_A$ thread nel primo step A ed $m$ blocchi di $NT_B$ thread nel secondo step B. Nel caso di utilizzo della variante hgm_a1a2 il primo step è spezzato in due sottofasi A1 e A2, formate rispettivamente da $NB_{A1} \cdot NT_{A1}$ e $NB_{A2} \cdot NT_{A2}$ thread. In entrambi i casi, per le osservazioni effettuate nel capitolo 4, possiamo affermare che la complessità è data dal primo step A (o A1), in quanto nettamente dominante in termini di numero di operazioni effettuate. La kernel function del primo step presenta una complessità pari a $O\left(\left\lceil\frac{NB_A \cdot NT_A}{NP}\right\rceil \cdot \frac{d}{NB_A \cdot NT_A}(a + 2\log(k))\right) = O\left(\frac{d}{NP}(a + 2\log(k))\right)$ ipotizzando $NB_A \cdot NT_A$ multiplo di $NP$ (con $NT_A = NT_{A1}$ e $NB_A = NB_{A1}$ nel caso di hgm_a1a2), in quanto ogni thread calcola al più $\frac{d}{NB_A \cdot NT_A}$ distanze ed esegue per ogni coppia l'operazione $updateMin$ sull'heap $NN$ e sul proprio heap $H_z$ (o $GH_z$), prima della riduzione. Siccome il primo step viene ripetuto $m$ volte, una per ogni punto candidato, la fase di confronto $D$ con $C -$ upMin senza matrice delle distanze presenta una complessità pari a $O\left(\frac{d}{NP} \cdot m(a + 2\log(k))\right)$, come nel caso con matrice.

Per la fase di confronto $D$ con $C -$ upMax si hanno $NB_A$ blocchi di $NT_A$ thread nel primo step ed un solo blocco di $NT_B$ thread nel secondo. Per quanto detto nel capitolo 4 e nel paragrafo 5.5, possiamo affermare che tale procedura ha complessità pari a

$O\left(\frac{d}{NP}\log(m)\right)$, in quanto il vettore $w$ è formato da $d$ elementi e l'operazione $updateMax$ su $LC$ richiede un tempo $O(\log(m))$.

La fase finale di selezione dei nuovi candidati richiede un tempo proporzionale a $O\left(\left\lceil\frac{m}{NP}\right\rceil \cdot a\right) = O(a)$ ipotizzando $NP > m$, in quanto tale operazione è eseguita da $m$ thread, dove ogni thread si occupa di copiare un singolo punto candidato.

Infine l'operazione più costosa eseguita dalla CPU ad ogni iterazione è l'aggiornamento dell'heap $Top$ che richiede un tempo $O(m \cdot \log(n))$.

Per quanto riguarda il calcolo della complessità temporale di una singola iterazione dell'algoritmo, si ha che la fase nettamente dominante è quella di confronto $D$ con $C -$ upMin; di conseguenza è possibile ignorare le tempistiche delle altri fasi in questo contesto. La complessità di una singola iterazione dell'algoritmo è quindi pari a $O\left(\frac{d}{NP} \cdot m(a + 2\log(k))\right)$.

Nel caso peggiore l'algoritmo inserisce l'intero dataset nel solving set e di conseguenza esegue $\lceil d/m \rceil$ iterazioni. La complessità dell'algoritmo CUDA_ODPSolvingSet è quindi pari a:

$$O\left(\left\lceil\frac{d}{m}\right\rceil \cdot \frac{d}{NP} \cdot m(a + 2\log(k))\right) \cong O\left(\frac{d^2}{NP} \cdot (a + 2\log(k))\right)$$

Se confrontiamo la complessità dell'algoritmo parallelo CUDA_ODPSolvingSet con la complessità dell'algoritmo sequenziale ODPSolvingSet si può notare che, utilizzando una GPU in grado di eseguire contemporaneamente $NP$ thread, in via teorica è possibile scalare di un fattore pari a $NP$ rispetto alla soluzione sequenziale. Anche per l'algoritmo ODPSolvingSet è quindi possibile implementare un algoritmo parallelo particolarmente efficiente.

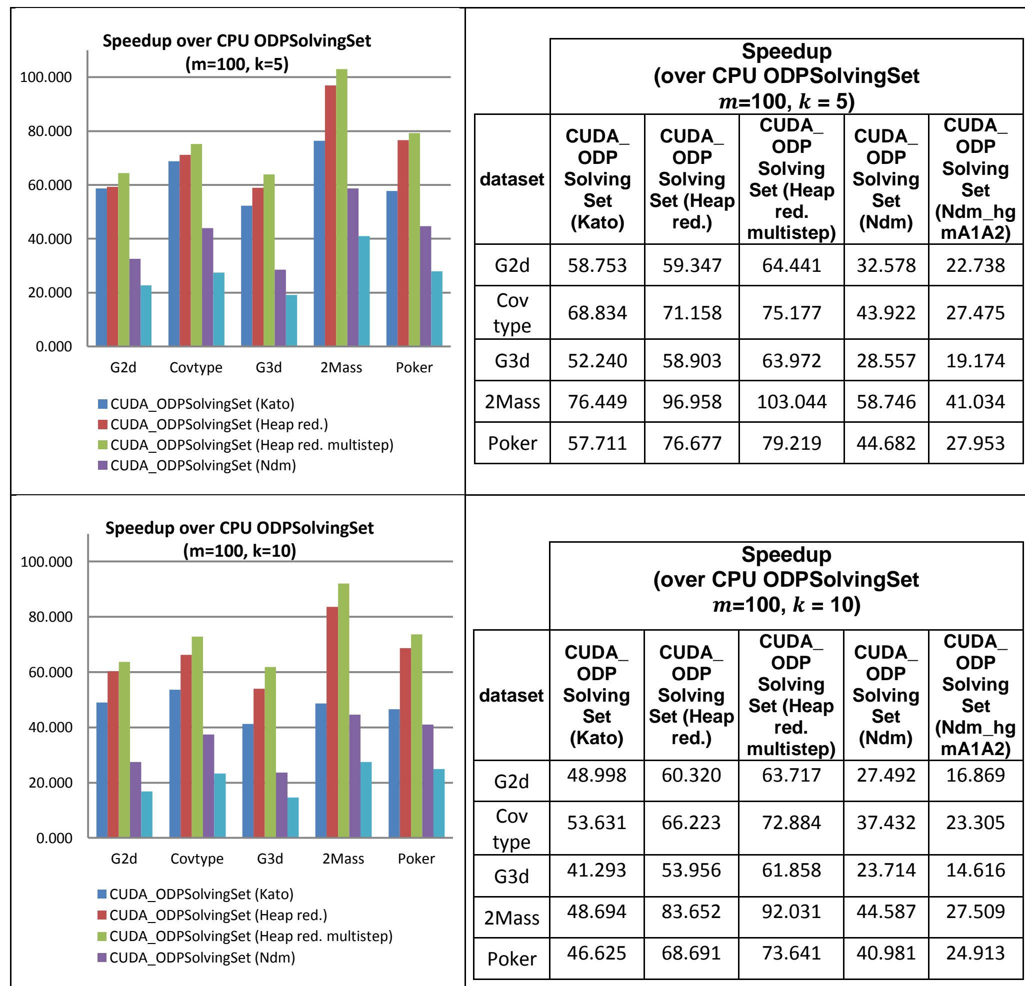
## 6.7 Risultati sperimentali

Per studiare le prestazioni dell'algoritmo parallelo CUDA_ODPSolvingSet sono state eseguite diverse serie di test. In tutti gli esperimenti come indicatore di performance è stato considerato lo *speedup* dell'algoritmo parallelo su quello sequenziale, definito come il rapporto tra i tempi di esecuzione di ODPSolvingSet e di CUDA_ODPSolvingSet. Anche in questo caso per le kernel function è stato realizzato un modulo CUDA *cubin* e la parte di comunicazione e coordinamento con la GPU è stata implementata in Java, sfruttando la libreria *JCuda*. Come piattaforma di testing si

è utilizzato un singolo nodo del supercomputer *IBM PLX*, dotato di due CPU *six-cores Intel Westmere* a 2.40 *GHz* (*E5645*) e di due GPU *NVIDIA Tesla M2070*.

### 6.7.1 Confronto sui dataset tradizionali

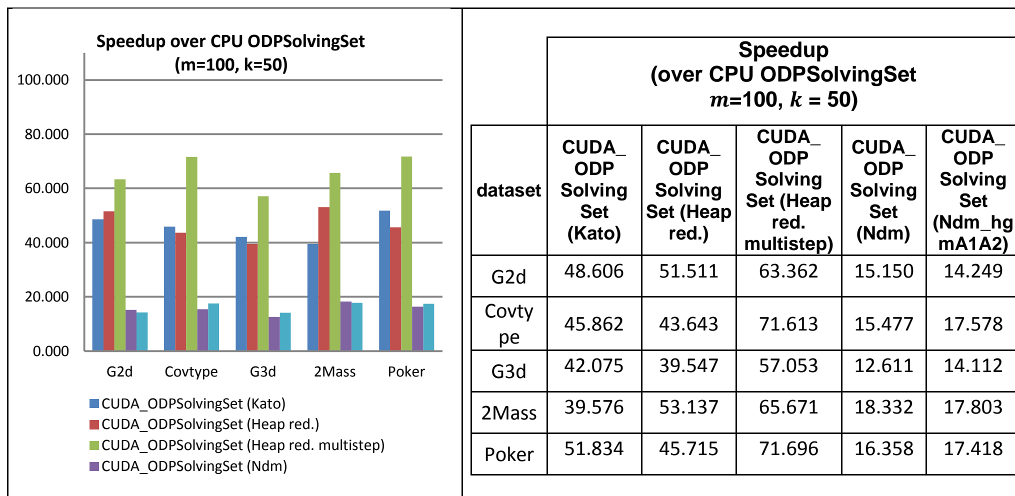
Per la prima batteria di test sono stati utilizzati i cinque dataset presentati nel paragrafo 2.5.1. Come parametri degli algoritmi è stato posto $n = 10$, $m = 100$ e sono state effettuate prove per $k = 5, 10, 50$. Per CUDA_ODPSolvingSet gli esperimenti hanno riguardato tutte le possibili modalità illustrate per la fase di confronto $D$ con $C -$ upMin. In particolare, nel caso di utilizzo della matrice con distanze, è stato utilizzato un solo chunk, in quanto per tutti i dataset è stato possibile allocare nella global memory della GPU l'intera matrice $m \times d$. I parametri specifici di CUDA per le diverse kernel function sono stati appositamente scelti per garantire le migliori performance.




| | Speedup (over CPU ODPSolvingSet $m$=100, $k$ = 5) | | | | |
|---|---|---|---|---|---|
| dataset | CUDA_ ODP Solving Set (Kato) | CUDA_ ODP Solving Set (Heap red.) | CUDA_ ODP Solving Set (Heap red. multistep) | CUDA_ ODP Solving Set (Ndm) | CUDA_ ODP Solving Set (Ndm_hg mA1A2) |
| G2d | 58.753 | 59.347 | 64.441 | 32.578 | 22.738 |
| Cov type | 68.834 | 71.158 | 75.177 | 43.922 | 27.475 |
| G3d | 52.240 | 58.903 | 63.972 | 28.557 | 19.174 |
| 2Mass | 76.449 | 96.958 | 103.044 | 58.746 | 41.034 |
| Poker | 57.711 | 76.677 | 79.219 | 44.682 | 27.953 |



| | Speedup (over CPU ODPSolvingSet $m$=100, $k$ = 10) | | | | |
|---|---|---|---|---|---|
| dataset | CUDA_ ODP Solving Set (Kato) | CUDA_ ODP Solving Set (Heap red.) | CUDA_ ODP Solving Set (Heap red. multistep) | CUDA_ ODP Solving Set (Ndm) | CUDA_ ODP Solving Set (Ndm_hg mA1A2) |
| G2d | 48.998 | 60.320 | 63.717 | 27.492 | 16.869 |
| Cov type | 53.631 | 66.223 | 72.884 | 37.432 | 23.305 |
| G3d | 41.293 | 53.956 | 61.858 | 23.714 | 14.616 |
| 2Mass | 48.694 | 83.652 | 92.031 | 44.587 | 27.509 |
| Poker | 46.625 | 68.691 | 73.641 | 40.981 | 24.913 |

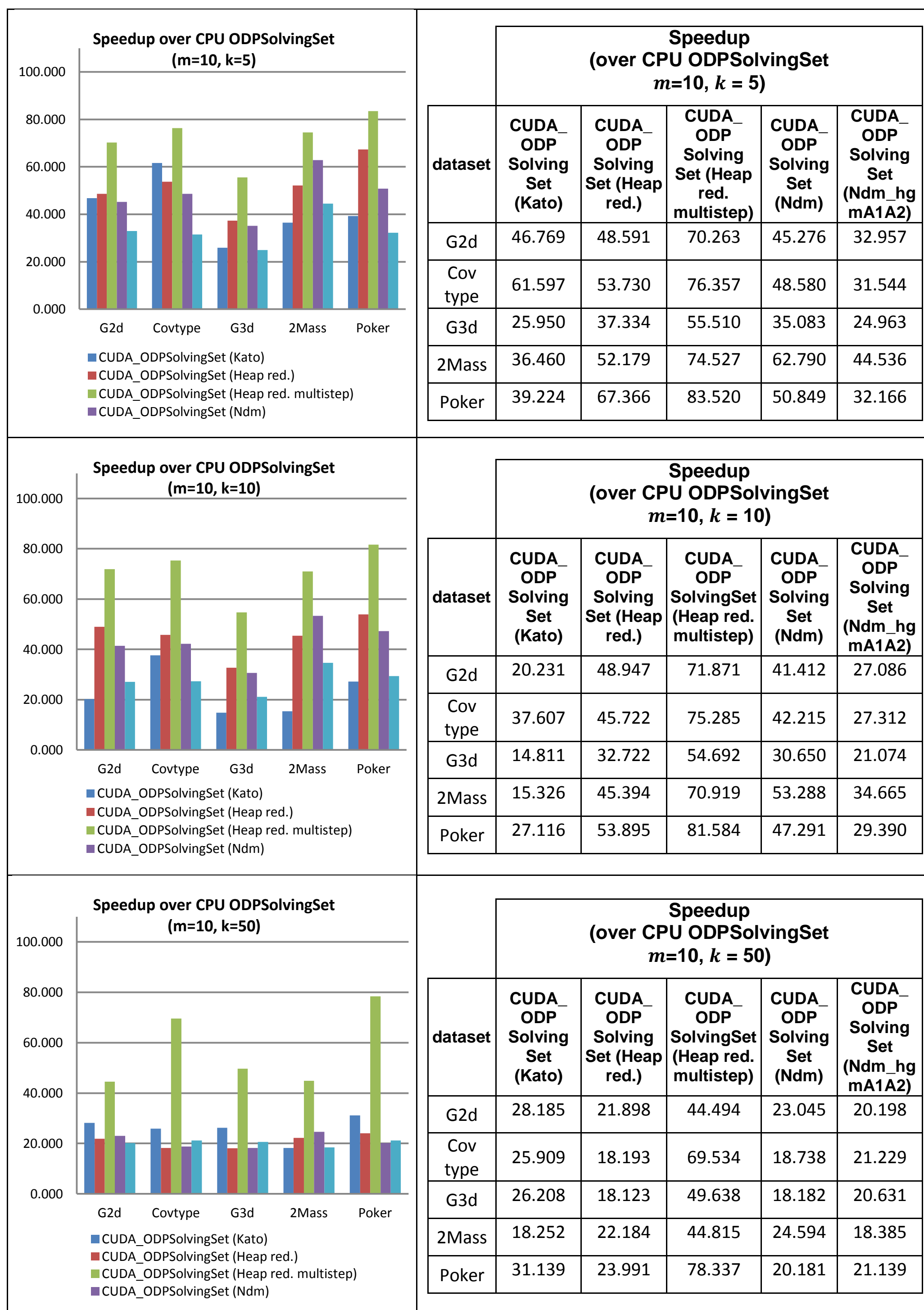



| | Speedup (over CPU ODPSolvingSet $m$=100, $k$ = 50) | | | | |
|---|---|---|---|---|---|
| dataset | CUDA_ ODP Solving Set (Kato) | CUDA_ ODP Solving Set (Heap red.) | CUDA_ ODP Solving Set (Heap red. multistep) | CUDA_ ODP Solving Set (Ndm) | CUDA_ ODP Solving Set (Ndm_hg mA1A2) |
| G2d | 48.606 | 51.511 | 63.362 | 15.150 | 14.249 |
| Covty pe | 45.862 | 43.643 | 71.613 | 15.477 | 17.578 |
| G3d | 42.075 | 39.547 | 57.053 | 12.611 | 14.112 |
| 2Mass | 39.576 | 53.137 | 65.671 | 18.332 | 17.803 |
| Poker | 51.834 | 45.715 | 71.696 | 16.358 | 17.418 |

**Figura 6-7 Speedup di CUDA_ODPSolvingSet su ODPSolvingSet, test sui 5 dataset tradizionali (*m*=100, *n*=10)**

Dai risultati dei test appare evidente che le modalità che utilizzano la matrice delle distanze (per la fase di confronto $D$ con $C$ – upMin) forniscono le migliori performance. Tra di esse la tecnica heap complete reduction multistep è quella che in assoluto garantisce il maggior speedup. Per valori piccoli di $k$ si raggiunge uno speedup massimo di 103 (per il dataset *2Mass*) con heap complete reduction multistep, seguono quindi heap complete reduction, Kato e più staccate le due modalità senza matrice delle distanze. Per valori più elevati di $k$ le performance dei metodi basati sulla riduzione parallela tendono a diminuire, registrando uno speedup al più intorno a 71, mentre il metodo di Kato presenta prestazioni più costanti. Questo è dovuto soprattutto alla maggiore quantità di shared memory richiesta per la memorizzazione degli heap temporanei, per poter attuare la riduzione, che limita l'occupancy dei multiprocessori. Inoltre, per $k$ elevato, si può osservare come aumenti il gap tra heap complete reduction multistep e heap complete reduction, in quanto in tali condizioni l'azione di filtering della tecnica multistep risulta essere più efficace.

Sempre utilizzando i medesimi cinque dataset, è stata effettuata una seconda batteria di test, utilizzando un valore nettamente inferiore per il parametro $m$. In particolare si è posto $n = 10,\ m = 10$ e sono state effettuate prove per $k = 5, 10{,}50$.




| dataset | Speedup (over CPU ODPSolvingSet $m$=10, $k$ = 5) CUDA_ ODP Solving Set (Kato) | CUDA_ ODP Solving Set (Heap red.) | CUDA_ ODP Solving Set (Heap red. multistep) | CUDA_ ODP Solving Set (Ndm) | CUDA_ ODP Solving Set (Ndm_hg mA1A2) |
|---|---|---|---|---|---|
| G2d | 46.769 | 48.591 | 70.263 | 45.276 | 32.957 |
| Cov type | 61.597 | 53.730 | 76.357 | 48.580 | 31.544 |
| G3d | 25.950 | 37.334 | 55.510 | 35.083 | 24.963 |
| 2Mass | 36.460 | 52.179 | 74.527 | 62.790 | 44.536 |
| Poker | 39.224 | 67.366 | 83.520 | 50.849 | 32.166 |

| dataset | Speedup (over CPU ODPSolvingSet $m$=10, $k$ = 10) CUDA_ ODP Solving Set (Kato) | CUDA_ ODP Solving Set (Heap red.) | CUDA_ ODP SolvingSet (Heap red. multistep) | CUDA_ ODP Solving Set (Ndm) | CUDA_ ODP Solving Set (Ndm_hg mA1A2) |
|---|---|---|---|---|---|
| G2d | 20.231 | 48.947 | 71.871 | 41.412 | 27.086 |
| Cov type | 37.607 | 45.722 | 75.285 | 42.215 | 27.312 |
| G3d | 14.811 | 32.722 | 54.692 | 30.650 | 21.074 |
| 2Mass | 15.326 | 45.394 | 70.919 | 53.288 | 34.665 |
| Poker | 27.116 | 53.895 | 81.584 | 47.291 | 29.390 |

| dataset | Speedup (over CPU ODPSolvingSet $m$=10, $k$ = 50) CUDA_ ODP Solving Set (Kato) | CUDA_ ODP Solving Set (Heap red.) | CUDA_ ODP SolvingSet (Heap red. multistep) | CUDA_ ODP Solving Set (Ndm) | CUDA_ ODP Solving Set (Ndm_hg mA1A2) |
|---|---|---|---|---|---|
| G2d | 28.185 | 21.898 | 44.494 | 23.045 | 20.198 |
| Cov type | 25.909 | 18.193 | 69.534 | 18.738 | 21.229 |
| G3d | 26.208 | 18.123 | 49.638 | 18.182 | 20.631 |
| 2Mass | 18.252 | 22.184 | 44.815 | 24.594 | 18.385 |
| Poker | 31.139 | 23.991 | 78.337 | 20.181 | 21.139 |

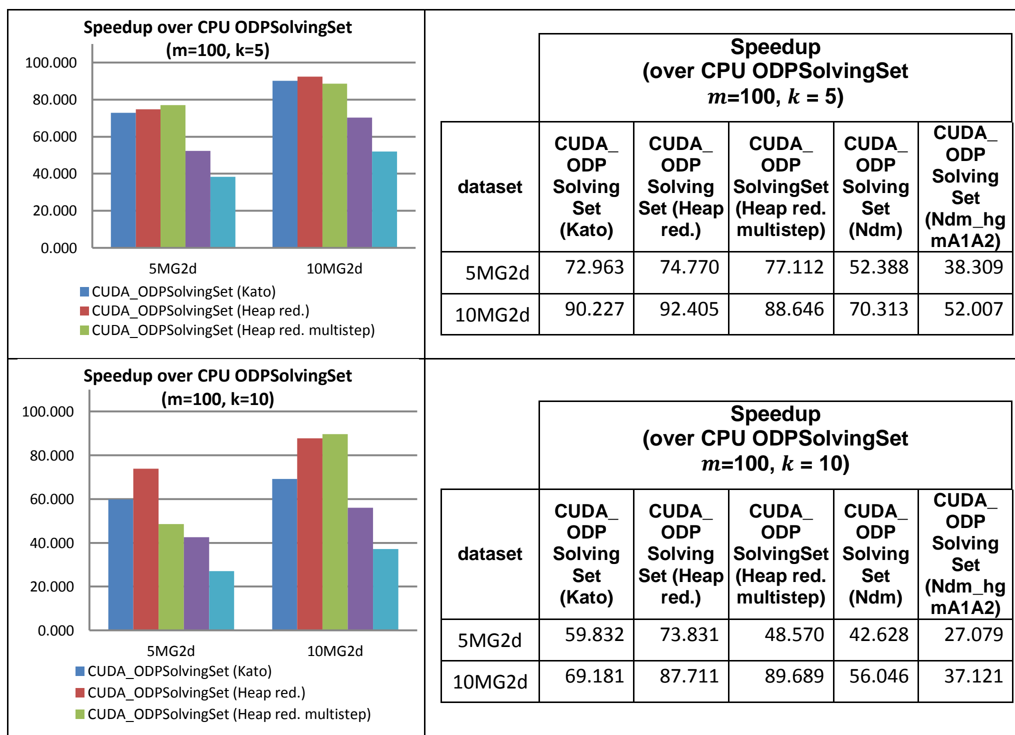
**Figura 6-8 Speedup di CUDA_ODPSolvingSet su ODPSolvingSet, test sui 5 dataset tradizionali ($m$=10, $n$=10)**

Per valori bassi di $m$ le prestazioni delle tecniche con matrice delle distanze tendono a diminuire, in quanto per lo step B della fase di confronto $D$ con $C$ – upMin utilizzano un numero fisso di blocchi pari ad $m$. Nel caso in cui $m$ sia inferiore al minimo numero di blocchi che garantisce la massima occupancy possibile dei multiprocessori, si ha uno sfruttamento non ideale dell’hardware della GPU. Le tecniche senza matrice

utilizzano invece un numero di blocchi indipendente da $m$ e riescono a fornire performance simili a quelle ottenute nella precedente batteria di test. La variante hgm_a1a2 appare utile solo nel caso di $k = 50$, dove la quantità di shared memory richiesta per gli heap temporanei diventa molto elevata, ma le differenze con la versione originale senza matrice risultano essere davvero minime. In ogni caso le modalità con matrice presentano comunque performance superiori a quelle senza matrice e l'utilizzo di heap complete reduction multistep si rivela essere ancora la scelta migliore.

### 6.7.2 Confronto su dataset gaussiani di grandi dimensioni

Per l'ultima serie di test sono stati utilizzati due dataset dotati di un numero molto elevato di istanze, generati artificialmente tramite una distribuzione gaussiana. I due insiemi di dati, *5MG2d* e *10MG2d*, contengono rispettivamente 5.000.000 e 10.000.000 vettori bi-dimensionali. Come parametri degli algoritmi è stato posto $n = 10$, $m = 100$ e sono state effettuate prove per $k = 5$ e 10. Non è stato possibile effettuare prove per $k = 50$, in quanto in tale condizione l'algoritmo sequenziale per CPU non è risultato in grado di terminare l'esecuzione entro il limite di tempo stabilito pari a 24 ore. Per CUDA_ODPSolvingSet, nel caso di utilizzo della modalità con matrice delle distanze per la fase di confronto $D$ con $C$ - upMin, è stato fissato il parametro $CSIZE = 2^{21} = 2.097.152$, spezzando quindi la matrice in più chunk.

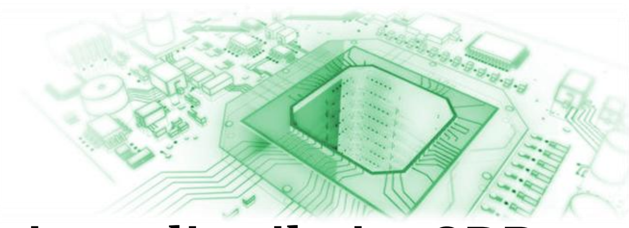



| | Speedup (over CPU ODPSolvingSet $m$=100, $k$ = 5) | | | | |
|---|---|---|---|---|---|
| dataset | CUDA_ ODP Solving Set (Kato) | CUDA_ ODP Solving Set (Heap red.) | CUDA_ ODP SolvingSet (Heap red. multistep) | CUDA_ ODP Solving Set (Ndm) | CUDA_ ODP Solving Set (Ndm_hgmA1A2) |
| 5MG2d | 72.963 | 74.770 | 77.112 | 52.388 | 38.309 |
| 10MG2d | 90.227 | 92.405 | 88.646 | 70.313 | 52.007 |

| | Speedup (over CPU ODPSolvingSet $m$=100, $k$ = 10) | | | | |
|---|---|---|---|---|---|
| dataset | CUDA_ ODP Solving Set (Kato) | CUDA_ ODP Solving Set (Heap red.) | CUDA_ ODP SolvingSet (Heap red. multistep) | CUDA_ ODP Solving Set (Ndm) | CUDA_ ODP Solving Set (Ndm_hgmA1A2) |
| 5MG2d | 59.832 | 73.831 | 48.570 | 42.628 | 27.079 |
| 10MG2d | 69.181 | 87.711 | 89.689 | 56.046 | 37.121 |

**Figura 6-9 Speedup di CUDA_ODPSolvingSet su ODPSolvingSet, test su ds gaussiani di grandi dimensioni ($m$=100, $n$=10)**

Confrontando i risultati dei test su *5MG2d* e *10MG2d* con quelli sul dataset gaussiano *G2d* (dotato di 1.000.000 di punti), si può notare un incremento di performance all'aumentare della dimensione dei dataset. Per $k = 5$ ed $m = 100$ si ha uno speedup massimo di 64.441 su *G2d*, di 77.112 su *5G2d* e di 92.405 su *10G2d*. Inoltre si può osservare che, spezzando la matrice delle distanze in più chunk, il metodo heap complete reduction riesca ad ottenere a volte migliori prestazioni della rispettiva variante di heap complete reduction multistep. Questo è dovuto al fatto che l'azione di filtering, che permette di terminare direttamente l'esecuzione dalla kernel function $compareDwithC_upMin_dm_phaseB_red$() nel caso in cui si verifica la condizione $Sum(LNNC[q_j]) < Min(Top)$, diventa più significativa per la divisione in chunk della matrice, assottigliando le differenze con la variante multistep.



# Capitolo 7: Algoritmo distribuito ODP Distributed Solving Set e implementazione in architetture distribuite multi-GPU

I grossi progressi tecnologici nelle tecniche di memorizzazione dei dati e di comunicazione nelle reti hanno portato alla nascita di molti sistemi di dati distribuiti. I sistemi bancari, i sistemi di prenotazione delle compagnie aeree, i database distribuiti e perfino Internet stesso possono esserne considerati un importante esempio. Tutti questi scenari sono caratterizzati dalla presenza di diverse sorgenti di informazione, organizzate su molteplici nodi di una rete. I singoli nodi possono essere molto numerosi e a loro volta gestire una quantità molto elevata di dati. L'analisi di sorgenti di dati distribuite richiede tecniche e tecnologie di *data mining* specializzate per questi ambiti distribuiti.

Una soluzione classica molto adottata consiste nell'utilizzare un'architettura *data warehouse-based*. Questo approccio prevede il trasferimento di tutti i dati provenienti dalle diverse sorgenti di dati distribuite, all'interno di una singola unità di storage centralizzata, per permettere quindi l'esecuzione di tradizionali tecniche di data mining sul singolo dataset risultante. Una soluzione centralizzata può non essere però la scelta più adatta in molti contesti. La grande quantità di risorse di memorizzazione locali richieste, gli elevati tempi di trasferimento dei dati e di esecuzione di algoritmi sequenziali su dataset di dimensioni così elevate, possono diventare fattori tali da rendere un approccio centralizzato inefficiente o perfino non applicabile.

Consideriamo ad esempio il sistema *NASA Earth Observing System* (*EOS*), che raccoglie le informazioni reperite da numerosi satelliti collocati in orbite polari e orbite terrestri basse. I dati raccolti sono suddivisi in 1450 data set distribuiti che sono gestiti da differenti *EOS Data and Information System* (*EOSDIS*), geograficamente dislocati in diverse località degli Stati Uniti. La quantità di dati gestiti è immensa, considerando che ogni satellite è in grado di produrre più di 350 GB di dati in un giorno. In un tale contesto una classica soluzione centralizzata per operazioni di data mining non è di certo ipotizzabile.

Un diverso approccio consiste invece nel prevedere delle architetture e degli algoritmi di data mining che prestino particolare attenzione alla distribuzione delle sorgenti dati, ai costi di trasmissione e alla possibilità di effettuare computazioni su più nodi distribuiti. Obiettivo del *Distributed Data Mining* (*DDM*) [27] è l'esecuzione di operazioni di data mining, ottimizzate a seconda della tipologia e della disponibilità delle risorse distribuite. Le tecniche DDM possono essere utili anche nel caso di sistemi dotati di diversi nodi specializzati nel calcolo intensivo, tipicamente interconnessi da reti ad alta velocità, come *cluster* o *supercomputer paralleli*. In tali situazioni diventa fondamentale un approccio distribuito, per ripartire in maniera efficace il calcolo computazionale tra le diverse unità di elaborazione.

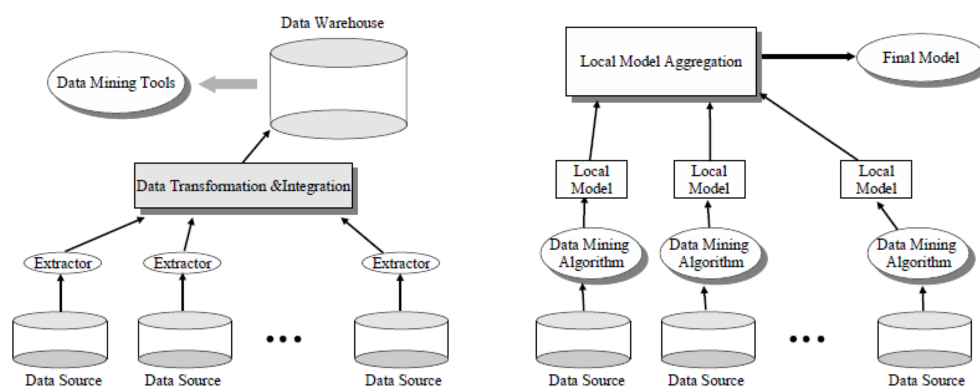



**Figura 7-1 Architettura data warehouse-based (a sinistra) e apprroccio DDM (a destra)**

Anche per il problema dell'outlier detection sono state proposte diverse soluzioni distribuite. In particolare *Angiulli, Basta, Lodi* e *Sartori* in [4] hanno presentato un efficace metodo distribuito per dataset di grandi dimensioni, basato su una generalizzazione del concetto di *solving set* al caso di dataset distribuiti. Obiettivo finale del nostro lavoro è la rivisitazione di tale algoritmo nell'ambito di una *architettura distribuita multi-GPU*, in cui siano presenti più nodi, ciascuno dotato di una propria CPU e di una propria GPU. Si viene quindi a creare un sistema a due livelli, in grado di combinare la capacità di calcolo parallelo delle GPU, con la possibilità di distribuire il lavoro su più nodi.

## 7.1 Algoritmo distribuito ODP Distributed Solving Set

Assumiamo di essere in presenza di un sistema distribuito, dotato di un *nodo supervisore* $N_0$ e di $l$ *nodi locali* $N_\gamma$ (con $\gamma = 1, \dots, l$), e ipotizziamo inoltre che il dataset $D$ sia partizionato in $l$ dataset locali $D_1, \dots, D_l$, con $D_\gamma$ allocato presso il nodo locale $N_\gamma$. L'obiettivo dell'algoritmo *ODPDistributedSolvingSet* è risolvere il problema $ODP < D, dist, n, k >$ e determinare il rispettivo solving set $S$, per il dataset distribuito $D$ [4].

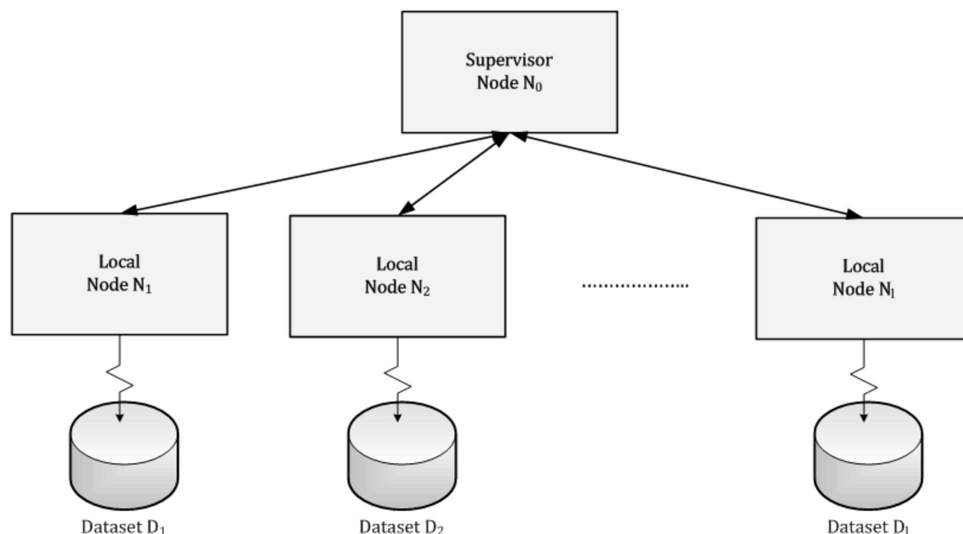



**Figura 7-2 Architettura distribuita**

L'algoritmo ODPDistributedSolvingSet adotta una strategia simile a quella di ODPSolvingSet. L'esecuzione avviene in più iterazioni, coordinate dal nodo supervisore, che si compongono di due step: un primo step di calcolo sui nodi locali ed un secondo step di sincronizzazione e analisi dei risultati parziali, forniti dai nodi locali. Nel primo step ogni nodo locale $N_\gamma$ riceve dal supervisor l'insieme corrente di punti candidati $C$ e il lower bound del peso dell'$n$-esimo top outlier (corrispondente a $Min(Top)$ di ODPSolvingSet). $N_\gamma$ confronta l'insieme $C$ con il proprio dataset locale $D_\gamma$ e ritorna al nodo supervisore la lista dei $k$-nearest neighbor locali in $D_\gamma$ rispetto a $C$, unitamente ad un insieme di nuovi candidati locali per l'iterazione successiva, appartenenti a $D_\gamma$. Nel secondo step il nodo supervisore utilizza i risultati ottenuti dai nodi locali per calcolare il peso esatto di ogni punto candidato in $C$, aggiornare l'insieme corrente dei top-$n$ outlier e generare un nuovo insieme di candidati per l'iterazione successiva.

L'algoritmo eseguito dal nodo supervisore è il seguente (in pseudo-codice):

```
ODPDistributedSolvingSet(l, [d_1 ... d_l], dist, n, k, m) {
        DSS := ∅
        Top := ∅
        d := Σ_{γ=1}^{l} d_γ
        for each node N_γ {
                NodeInit_γ(⌈m d_γ/d⌉, C_γ)
        }
        C := ∪_{γ=1}^{l} C_γ
        act := d

        while (C ≠ ∅) {
                DSS := DSS ∪ C
                for each node N_γ {
                        NodeComp_γ(Min(Top), C, act, LNNC_γ, LC_γ, act_γ)
                }
                act := Σ_{γ=1}^{l} act_γ
                for each q in C {
                        NNC[q] := get_k_NNC(∪_{γ=1}^{l} LNNC_γ[q])
                        updateMax(Top, ⟨q, Sum(NNC[q])⟩ )
                }
                C := ∅
                for each p in ∪_{γ=1}^{l} LC_γ {
                        C := C ∪ {p}
                }
        }
}
```

L'algoritmo ODPDistributedSolvingSet prende in input il numero $l$ di nodi locali, i valori $d_\gamma$ rappresentanti le dimensioni dei dataset locali $D_\gamma$, una funzione di distanza $dist$ definita su $D$, il parametro $n$ indicante il numero di top outlier da trovare, il numero $k$ di nearest neighbor da considerare per il calcolo del peso dei punti ed un intero $m$ rappresentante il numero di oggetti che devono essere inseriti nel solving set ad ogni iterazione. Il nodo supervisore mantiene in memoria il solving set $DSS$, un min-heap $Top$ che memorizza i correnti $n$ punti $q \in DSS$ con i maggiori pesi finora trovati (tramite coppie di tipo $\langle q, w_k(p, D)\rangle$) ed un intero $act$ che rappresenta il numero totale di punti dichiarati come attivi in $D$. All'inizio dell'esecuzione vengono inizializzate queste tre strutture e viene scelto il primo insieme di punti candidati $C$, selezionando in maniera casuale $m$ punti dal dataset distribuito $D$. Questo avviene tramite la chiamata della procedura $NodeInit()$, effettuata dal supervisor su tutti gli $l$ nodi locali. L'algoritmo è quindi eseguito tramite diverse iterazioni, fino a quando l'insieme dei candidati $C$ eletto per l'iterazione successiva non risulta essere vuoto, ovvero quando non si hanno più punti in $D$ dichiarati come attivi. Ad ogni iterazione il supervisore aggiunge l'insieme $C$ al corrente solving DSS ed invoca l'operazione

$NodeComp()$ su ogni nodo locale, passando ai nodi l'insieme dei candidati $C$, il lower bound del peso dell'$n$-esimo top outlier $Min(Top)$ (corrispondente alla radice dell'heap $Top$) e il numero totale degli oggetti attivi $act$. Dopo aver terminato l'esecuzione, ogni nodo locale $N_\gamma$ ritorna al supervisore una struttura $LNNC_\gamma$, contenente i $k$ nearest neighbor nel dataset locale $D_\gamma$ per ogni punto $q \in C$ unitamente alle rispettive distanze da $q$, un max-heap $LC_\gamma$, contenente i punti in $D_\gamma$ che faranno parte dell'insieme di candidati nell'iterazione successiva, e un intero $act_\gamma$, pari al numero di punti rimasti attivi in $D_\gamma$ alla fine dei confronti. Ogni max-heap $LC_\gamma$ contiene rispettivamente gli $m_\gamma$ punti dichiarati come attivi in $D_\gamma$ con il maggior weight upper bound, dove $m_\gamma = \left\lceil m \frac{act_\gamma}{act} \right\rceil$. A questo punto il nodo supervisore calcola il peso $w_k(q, D)$ di ogni punto $q \in C$ come somma delle minori $k$ distanze dei punti appartenenti a $LNNC_1[q] \cup LNNC_2[q] \cup \ldots \cup LNNC_l[q]$ e chiama l'operazione $updateMax$ su $Top$ per ogni coppia $\langle q, w_k(q, D) \rangle$. Infine viene aggiornato il numero di punti attivi totali $act$ e selezionato il nuovo insieme di candidati $C$ come unione di tutti punti appartenenti in $LC_1 \cup LC_2 \cup \ldots \cup LC_l$. Se l'insieme $C$ non risulta essere vuoto viene eseguita una nuova iterazione dell'algoritmo, altrimenti vengono restituiti i punti in $Top$ come soluzione del problema ODP e $DSS$ come solving set.

Focalizziamo ora l'attenzione sulle operazioni eseguite dai nodi locali. Ogni nodo locale $N_\gamma$ mantiene in memoria un insieme di punti $C_\gamma$, contenente i punti candidati dell'iterazione corrente appartenenti al dataset locale $D_\gamma$, ed un intero $act_\gamma$, rappresentante il numero di punti in $D_\gamma$ dichiarati come attivi. Inoltre associamo ad ogni punto $p \in D_\gamma$ un max-heap $NN_\gamma[p]$, i cui elementi siano le coppie $\langle x, \delta \rangle$, con $x \in DSS$ e $\delta = dist(p, x)$, che mantiene i correnti $k$ nearest neighbors di $p$, rispetto al corrente solving set $DSS$. Similarmente associamo ad ogni punto $q \in C$ un max-heap $LNNC_\gamma[q]$, con elementi le coppie $\langle y, \delta \rangle$, con $y \in D_\gamma$ e $\delta = dist(q, y)$, contenente i correnti $k$ nearest neighbors di $q$ rispetto al dataset locale $D_\gamma$ (e a $DSS$ se $q \in C_\gamma$). Descriviamo le due procedure eseguite sui nodi locali $N_\gamma$:

- $NodeInit_\gamma()$: riceve in input un intero $m_\gamma$ e ritorna un insieme $C_\gamma$ di $m_\gamma$ elementi, scelti in maniera casuale nel dataset locale $D_\gamma$. Inoltre viene inizializzato $act_\gamma$, ponendolo pari alla cardinalità di $D_\gamma$.

```
NodeInitγ(mγ, Cγ){
        Cγ := RandomSelect(Dγ, mγ)
        actγ := |Dγ|
}
```

- $NodeComp_\gamma()$: può essere paragonata alle operazioni eseguite nei due loop di confronto dell'algoritmo ODPSolvingSet. La procedura riceve in input l'insieme dei candidati $C$, il lower bound del peso dell'$n$-esimo top outlier $Min(Top)$ e il numero totale $act$ degli oggetti attivi in $D$. Come prime operazioni si ha la rimozione dei candidati locali $C_\gamma$ dal dataset locale $D_\gamma$, l'aggiornamento di $act_\gamma$ e l'inizializzazione degli heap $LNNC_\gamma[q_j]$ con il contenuto di $NN_\gamma[q_j]$, per ogni $q_j \in C_\gamma$. Gli heap $LNNC_\gamma[q_i]$ per i punti $q_i$ non appartenenti all'insieme locale $C_\gamma$ vengono inizializzati come vuoti, in quanto solamente il nodo locale $N_\alpha$ che ha generato il candidato $q_i$ si deve preoccupare di memorizzare i $k$-nearest neighbor di $q_i$ rispetto ai precedenti set di candidati (memorizzati in $NN_\alpha[q_i]$). Ogni candidato $q_j \in C$ viene quindi confrontato con tutti i punti locali in $C_\gamma \cup D_\gamma$. Per poter effettuare questa operazione in maniera efficiente, il confronto è spezzato in tre diversi loop. Il primo loop confronta a due a due tutti i candidati locali in $C_\gamma$, aggiornando i rispettivi heap. Il secondo loop confronta ogni candidato locale in $C_\gamma$ con i punti candidati appartenenti a $C - C_\gamma$, aggiornando solamente gli heap $LNNC_\gamma$ dei candidati locali. Nel terzo loop si confrontano tutti i punti $p \in D_\gamma$ del dataset locale con i candidati $q \in C$. Il calcolo della distanza tra $p$ e $q$ e l'aggiornamento dei rispettivi heap $LNNC_\gamma[q]$ e $NN_\gamma[p]$ avviene solamente se si verifica $\max\{Sum(NN_\gamma[p]), Sum(LNNC_\gamma[q])\} \geq MinTop$, come in ODPSolvingSet. Per l'elezione dei nuovi candidati solamente i punti $p \in D_\gamma$ tali che $Sum(NN_\gamma[p]) \geq MinTop$ vengono inseriti nell'heap $LC_\gamma$ tramite l'operazione $updateMax$, gli altri punti vengono invece dichiarati come non attivi. Inoltre viene aggiornato il valore di $act_\gamma$, rispetto a questi ultimi confronti. Viene infine popolato $C_\gamma$ con il nuovo set di punti presenti in $LC_\gamma$, tramite la funzione $candSelect()$ e vengono ritornati al nodo supervisore le strutture $LNNC_\gamma$, $LC_\gamma$ e $act_\gamma$ aggiornate.

```
NodeCompγ (MinTop, C, act, LNNCγ, LCγ, actγ){
        Dγ := Dγ − Cγ
        actγ := actγ − |Cγ|
        for each qj in Cγ {LNNCγ[qj] := NNγ[qj]}

        for each qi in {q0, q1, …, q|Cγ|−1} = Cγ {
                for each qj in {qi, qi+1, …, q|Cγ|−1} ⊆ Cγ {
                        if (i = j) {
                                updateMin(LNNCγ[qi], ⟨qi, 0⟩ )
                        } else {
                                δ := dist(qi, qj)
                                updateMin(LNNCγ[qi], ⟨qj, δ⟩ )
                                updateMin(LNNCγ[qj], ⟨qi, δ⟩ )
                        }
                }
        }
        for each qi in Cγ {
                for each qj in C − Cγ {
                        δ := dist(qi, qj)
                        updateMin(LNNCγ[qi], ⟨qj, δ⟩ )
                }
        }

        LCγ := ∅ //LCγ ha dimensione pari a ⌈m actγ/act⌉
        actγ := 0
        for each p in Dγ {
                for each q in C {
                    if(max{Sum(NNγ[p]), Sum(LNNCγ[q])} ≥ MinTop) {
                                δ := dist(p, q)
                                updateMin(NNγ[p], ⟨q, δ⟩ )
                                updateMin(LNNCγ[q], ⟨p, δ⟩ )
                    }
                }
                if (p.active = true AND Sum(NNγ[p]) ≥ MinTop) {
                        updateMax(LCγ, ⟨p, Sum(NNγ[p])⟩ )
                        actγ := actγ + 1
                }else {
                        p.active := false
                }
        }
        Cγ := candSelect(LCγ)
}
```

Dimostriamo ora che l'algoritmo è corretto, ovvero che ODPDistributedSolvingSet determina i top $n$ outlier del dataset distribuito $D$.

Siccome i punti in $Top$ sono stati confrontati con tutti gli oggetti in $D$, il loro peso è corretto. Di conseguenza per dimostrare la tesi è sufficiente provare che, per ogni punto $p \in D - Top$, valga $w_k(p, D) \le Min(Top)$. Consideriamo un generico punto $p \in D$ e supponiamo che appartenga al dataset locale $D_\gamma$ (con $1 \le \gamma \le l$). $Sum(NN_\gamma[p])$ è un upper bound del peso reale $w_k(p, D)$ di $p$, in quanto $NN_\gamma[p]$ contiene i $k$-nearest neigbor rispetto a $DSS \subseteq D$. L'algoritmo termina quando tutti gli heap $LC_\gamma$, e di conseguenza il nuovo set di candidati $C$, risultano essere vuoti, ovvero

quando tutti i punti $p$ rimasti in $D_\gamma$ presentano un weight upper bound $Sum(NN_\gamma[p])$ non superiore a $Min(Top)$. In altre parole il set $Top$ ritornato dall'algoritmo è tale che per ogni $p \in D - Top$ vale $w_k(p, D) \leq Sum\big(NN_\gamma[p]\big) \leq Min(Top)$, il che conferma la tesi.

Consideriamo ora la complessità temporale dell'algoritmo. Sia $a$ la dimensionalità dei punti del dataset, ovvero il numero di coordinate (o di attributi) di essi.
Per quanto riguarda il calcolo affidato ai nodi locali, facendo un paragone con l'algoritmo ODPSolvingSet, possiamo asserire che la fase dominante del carico di lavoro è quella eseguita nel loop di confronto $D_\gamma$ con $C$ della procedura $NodeComp_\gamma()$. Tale fase presenta una complessità pari a $O\left(d_\gamma \cdot m \cdot (\mathrm{a} + 2\log(k))\right)$, in quanto si ha il calcolo di $m \cdot d_\gamma$ distanze, con rispettive operazioni $updateMin$ sugli heap $NN_\gamma$ e $LNNC_\gamma$. Supponiamo per semplicità che il dataset $D$ sia distribuito in maniera uniforme tra i vari nodi locali, ovvero che $d_\gamma = \frac{d}{l}$, per $\gamma = 1, \dots, l$. Siccome nel caso peggiore l'algoritmo esegue $\left\lceil \frac{d}{m} \right\rceil$ iterazioni, la complessità temporale del lavoro in carico ad un singolo nodo locale è pari a

$$O\left(\left\lceil \frac{d}{m} \right\rceil \cdot \frac{d}{l} \cdot m \cdot (\mathrm{a} + 2\log(k))\right) \cong O\left(\frac{d^2}{l} \cdot (\mathrm{a} + 2\log(k))\right).$$

Se consideriamo il lavoro svolto dal nodo supervisore, le operazioni dominanti consistono nella determinazione dei $k$-nearest neighbor di ogni punto candidato $q \in C$ tra le $k \cdot l$ distanze in $LNNC_1[q] \cup LNNC_2[q] \cup \dots \cup LNNC_l[q]$ e nell'aggiornamento dell'heap $Top$, che richiede $m$ operazioni $updateMax$. Assumiamo inoltre che le $k \cdot l$ distanze vengano ordinate con un algoritmo di tipo *heap-sort*, al fine di determinarne le $k$ minori. Siccome nel caso peggiore l'algoritmo esegue $\left\lceil \frac{d}{m} \right\rceil$ iterazioni, la complessità temporale del lavoro in carico ad nodo supervisore è pari a

$$O\left(\left\lceil \frac{d}{m} \right\rceil \cdot m \cdot (kl \cdot \log(kl) + \log(n))\right) \cong O\big(d \cdot (kl \cdot \log(kl) + \log(n))\big).$$

La complessità totale dell'algoritmo è quindi data dalla somma dei costi in capo ai nodi locali e al supervisore. Siccome per $d \gg l$, con $d$ sufficientemente grande, il costo dell'esecuzione su un singolo nodo locale è nettamente dominante rispetto al costo del lavoro eseguito dal supervisore, possiamo affermare che l'algoritmo ODPDistributedSolvingSet ha complessità temporale pari a :

$$O\left(\frac{d^2}{l} \cdot (\mathrm{a} + 2\log(k))\right).$$

Se confrontiamo la complessità della soluzione distribuita con quella presentata dall'algoritmo sequenziale ODPSolvingSet, si può notare come è possibile scalare di una fattore ideale pari ad $l$, utilizzando $l$ nodi locali. C'è comunque da sottolineare che nel caso di $l$ particolarmente elevato ($l \cong d$) i tempi di esecuzione del nodo supervisore e dei nodi locali diventano comparabili e lo speedup della versione distribuita tende fortemente a calare. In una situazione reale però tipicamente $d \gg l$ e quindi questo problema non si presenta.

Consideriamo ora il costo di trasmissione, in termini di quantità di dati che necessitano di essere trasferiti tra il nodo supervisore e i nodi locali. Ipotizziamo che l'algoritmo esegua $t$ iterazioni per determinare la soluzione. La procedura $NodeInit_\gamma()$ è eseguita una sola volta e richiede da parte del supervisore l'invio di un intero (ovvero $m_\gamma$) e la ricezione di $m_\gamma$ punti (dotati di $a$ coordinate), per ogni nodo locale. La procedura $NodeComp_\gamma()$ è invece eseguita $t$ volte. Da parte del supervisore richiede l'invio di un numero floating point $Min(Out)$, degli $m$ oggetti in $C$ e di un intero $act$. Da parte dei nodi locali richiede invece l'invio di $m \cdot k$ distanze in $LNNC_\gamma$, $m_\gamma$ punti in $LC_\gamma$ ed un intero $act_\gamma$. Il numero totale di dati trasferiti, in termini di numeri interi o floating point, è pari a:

$$TD = \sum_{\gamma=1}^{l} (1 + m_\gamma a) + t\left(1 + ma + 1 + \sum_{\gamma=1}^{l} (mk + m_\gamma a + 1)\right)$$
$$= l + ma + 2t + |DSS|(a + lk + a) + tl,$$

in quanto vale $\sum_{\gamma=1}^{l} m_\gamma = m$ e $tm = |DSS|$.

Siccome tipicamente $m \ll |DSS|$, possiamo approssimare $TD$ nel modo seguente:

$$TD \approx |DSS|(lk + 2a).$$

Indichiamo con $\rho$ la dimensione relativa del solving set rispetto al dataset distribuito $D$, ovvero $\rho = |DSS|/d$. Consideriamo il rapporto tra la quantità di dati trasferiti $TD$ e la dimensione del dataset in termini di numeri floating point (pari a $d \cdot a$):

$$TD\% = \frac{TD}{da} \approx \frac{\rho lk}{a}.$$

Possiamo notare che, a parità di valori $d, a$ e $k$, $TD\%$ è direttamente proporzionale alla dimensione relativa del solving set $\rho$ e al numero di nodi locali $l$. Dagli esperimenti mostrati nel paragrafo 2.5.1 si è però visto come $\rho$ decresca in modo più che lineare al crescere di $d$, in quanto la dimensione del solving set tende a stabilizzarsi. Inoltre la maggior parte dei dati trasmessi sono distanze e non punti, fattore che diventa rilevante nel caso di punti ad alta dimensionalità. Ipotizzando $l \ll d$, il costo di

trasmissione è quindi relativamente basso, se confrontato con quello di una soluzione centralizzata che preveda il trasferimento degli interi $l$ dataset locali su di un unico nodo, per poter successivamente eseguire sul singolo dataset risultante l'algoritmo centralizzato per il problema ODP. La dipendenza lineare dal numero di nodi locali $l$ può però risultare un fattore da non sottovalutare, soprattutto in scenari in cui il canale di comunicazione sia particolarmente lento, e può portare a notevoli deterioramenti delle performance all'aumentare del numero di nodi. Per cercare di eliminare questa dipendenza, introduciamo una variante all'algoritmo ODPDistributedSolvingSet di base, denominata *ODPLazyDistributedSolvingSet*.

### 7.1.1 Variante ODP Lazy Distributed Solving Set

La variante *ODPLazyDistributedSolvingSet* utilizza una strategia più sofisticata, atta a ridurre il numero di distanze inviate dai nodi locali. In particolare mira a far trasmettere ad ogni nodo $N_\gamma$ un numero $k_\gamma < k$ di distanze per ogni iterazione dell'algoritmo, per un totale di $l \cdot k_\gamma$ valori, tale che $l \cdot k_\gamma$ sia $O(k)$. Questo permette di sostituire il termine $l \cdot k$ con $k$ nel calcolo di $TD$ e di limitare la dipendenza da $l$ sull'ammontare di informazioni scambiate. La quantità relativa di dati trasferiti, ovvero il rapporto tra $TD$ e la dimensione del dataset, con questa variante può essere quindi approssimata nel modo seguente:

$$TD\% \approx \frac{\rho k}{a}.$$

Inoltre la complessità temporale delle operazioni svolte dal nodo supervisore diventa:

$$O\big(d \cdot (k \cdot \log(k) + \log(n))\big),$$

eliminando la dipendenza diretta da $l$.

Per poter limitare il numero di distanze trasmesse si utilizza una procedura incrementale, svolta in più iterazioni. Ad ogni iterazione il nodo supervisore riceve un sottoinsieme delle distanze di ogni candidato dai rispettivi $k$ nearest neighbor, le aggiunge a quelle eventualmente ricevute in precedenza e controlla se siano necessarie delle distanze addizionali per poter determinare il peso esatto del corrispondente candidato. Se si verifica tale situazione effettua una nuova iterazione ed una nuova richiesta ai nodi locali, altrimenti la procedura incrementale si ferma.

Per poter attuare tale strategia, è necessaria innanzitutto una piccola modifica alla procedura $NodeComp_\gamma()$. La funzione deve prendere in input un parametro aggiuntivo $k_0$, con $k_0 < k$, rappresentante il numero delle minori $k_0$ distanze dei $k$

nearest neighbor da inviare al supervisore come risultato di $NodeComp_\gamma()$. Il nodo locale deve svolgere tutte le operazioni previste nella precedente versione di $NodeComp_\gamma()$, più altre azioni aggiuntive. In particolare, per ogni punto candidato $q \in C$, deve operare il sorting (in ordine crescente) delle distanze presenti nell'heap $LNNC_\gamma[q]$ e mantenere in $LNNC_\gamma[q]$ solamente le minori $k_0$ distanze. Le rimanenti $k - k_0$ vengono trasferite in un nuovo vettore $rLNNC_\gamma[q]$, in modo da renderle disponibili per possibili richieste future da parte del supervisore. Il parametro $k_0$ viene posto pari a $\left\lceil \frac{k}{l} \right\rceil + 1$, per garantire al supervisore la ricezione di almeno $k$ distanze totali.

```
NodeCompγ (MinTop, C, act, k0, LNNCγ, LCγ, actγ){
        … //come NodeCompγ() di ODPDistributedSolvingSet
        for each q in C {
                LNNCγ[q] := sort(LNNCγ[q])
                rLNNCγ[q] := get_last(LNNC[q]γ, k − k0)
                LNNCγ[q] := get_first(LNNCγ[q], k0)
        }
}
```

Per ogni candidato $q$, dopo l'esecuzione della procedura $NodeComp_\gamma()$ su ogni nodo locale, il supervisore aggiorna il vettore ordinato $NNC[q]$ con le $l \cdot k_0 < l \cdot k$ distanze contenute in $LNNC_1[q] \cup \ldots \cup LNNC_l[q]$. Se tutte le distanze in $NNC[q]$ risultano essere minori dell'ultimo elemento di $LNNC_\gamma[q]$ (indichiamolo con $last(LNNC_\gamma[q])$), per $\gamma = 1, \ldots, l$, allora $NNC[q]$ contiene le distanze dai $k$ nearest neighbor di $q$. Questo in quanto le distanze non inserite in $LNNC_\gamma[q]$ (e salvate in $rLNNC_\gamma[q]$) sono sicuramente non minori di quelle in $LNNC_\gamma[q]$, per l'ordinamento effettuato in $NodeComp_\gamma()$. Altrimenti sia $dist_{min} = \min_\gamma\{last(LNNC_\gamma[q])\}$, ovvero la più piccola distanza tra le maggiori distanze di tutti gli heap $LNNC_\gamma[q]$. In questo caso $dist_{min}$ è presente in $NNC[q]$ in una certa posizione, supponiamo la $j$-esima. Allora le prime $j$ distanze in $NNC[q]$ sono quelle che separano $q$ dai rispettivi primi $j$ nearest neighbor, mentre le rimanenti $k - j$ rappresentano un upper bound delle vere distanze che separano $q$ dai successivi $k - j$ nearest neighbor. Per poter individuare il valore esatto delle rimanenti $k - j$ distanze è necessaria una nuova richiesta da parte del supervisore ai nodi locali, che avviene tramite la procedura $NodeReq_\gamma()$. Il supervisore salva in $u_NNC[q]$ il valore $k - j$, in $cur_last[q]$ l'upper bound $NNC[q][k]$ della distanza di $q$ dal $k$-esimo nearest neighbor, in $nodes[q]$ gli identificativi dei nodi $N_\gamma$ per cui $last(LNNC_\gamma[q]) \in NNC[q]$ (ovvero quei nodi che

possono fornire almeno una delle $k-j$ distanze “ignote”) e invia tali informazioni ai nodi locali.

La procedura $NodeReq_{\gamma}()$ copia in $LNNC_{\gamma}[q]$ le nuove distanze addizionali e le ritorna al supervisore. In particolare, per ogni punto candidato $q \in C$, nel caso in cui il nodo $N_{\gamma}$ possa fornire almeno una delle $k-j$ distanze ignote (ovvero se $N_{\gamma} \in nodes[q]$), si ha il trasferimento in $LNNC_{\gamma}[q]$ delle successive più piccole $\left\lceil \frac{u_LNNC[q]}{|nodes[q]|} \right\rceil + 1$ distanze rimaste salvate in $rLNNC_{\gamma}[q]$. Inoltre tra di esse si ha l’inserimento in $LNNC_{\gamma}[q]$ solamente di quelle inferiori a $cur_last[q]$, ovvero l’upper bound alla $k$-esima distanza da $q$.

```
NodeReqγ (u_LNNC, cur_last, nodes, LNNCγ){
        for each q : Nγ ∈ nodes[q]  {
                LNNCγ[q] := get_first(rLNNCγ[q],
                                        ⌈u_LNNC[q] / |nodes[q]|⌉ + 1, cur_last[q])
                rLNNCγ[q] := get_last(rLNNCγ[q],
                                        |rLNNCγ[q]| − |LNNCγ[q]|)
        }
}
```

Dopo aver ricevuto il nuovo insieme di distanze, il supervisore aggiorna $NNC[q]$ per ogni punto candidato $q$ e determina, tramite il medesimo procedimento precedente, se è necessaria una nuova iterazione di richiesta di distanze aggiuntive. In caso in cui non sia necessaria (ovvero quando $nodes[q]$ risulti essere vuoto per ogni $q \in C$) allora il processo si ferma, viene calcolato il peso esatto di $q$, aggiornato l’heap $Top$ e determinato il nuovo insieme di candidati per l’iterazione successiva dell’algoritmo.

L'algoritmo *ODPLazyDistributedSolvingSet* eseguito dal nodo supervisore è il seguente (in pseudo-codice):

```
ODPLazyDistributedSolvingSet(l, [d_1 ... d_l], dist, n, k, m){
    DSS := ∅
    Top := ∅
    d := Σ_{γ=1}^{l} d_γ
    for each node N_γ {
        NodeInit_γ(⌈m d_γ/d⌉, C_γ)
    }
    C := ∪_{γ=1}^{l} C_γ
    act := d

    while (C ≠ ∅) {
        DSS := DSS ∪ C
        for each node N_γ {
            NodeComp_γ(Min(Top), C, act, ⌈k/l⌉ + 1, LNNC_γ, LC_γ, act_γ)
        }
        act_γ := Σ_{γ=1}^{l} act_γ
        do {
            for each q in C {
              NNC[q] := get_k_NNC(NNC[q] ∪ (∪_{γ=1}^{l} LNNC_γ[q]))
              u_NNC[q] := 0
              nodes[q] := ∅
              if (∃ j : min_γ{last(LNNC_γ[q])} = NNC[q][j]) {
                u_NNC[q] := k − j
                cur_last[q] := NNC[q][k]
                nodes[q] := ∪_{γ=1}^{l} N_γ: last(LNNC_γ[q]) ∈ NNC[q]
              }
            }
            for each node N_γ in ∪_{q∈C} nodes[q] {
              NodeReq_γ(u_NNC, cur_last, nodes, LNNC_γ)
            }
        } while (∪_{q∈C} nodes[q] ≠ ∅)
        for each q in C {
            updateMax(Top, ⟨q, Sum(NNC[q])⟩ )
        }

        C := ∅
        for each p in ∪_{γ=1}^{l} LC_γ {
            C := C ∪ {p}
        }

    }
}
```

## 7.2 Implementazione dell'algoritmo distribuito ODP Distributed Solving Set in una architettura multi-GPU

Assumiamo ora che ogni nodo locale $N_\gamma$, con $l = 1, \dots, l$, sia dotato di una di propria GPU che supporti CUDA. Possiamo sfruttare la potenza di calcolo della GPU per accelerare le operazioni svolte singolarmente da ogni nodo locale. Si viene quindi a creare un *sistema di distribuzione a due livelli*, con un primo livello che prevede la sudddivisione del dataset distribuito $D$ tra i diversi nodi locali ed un secondo livello che prevede l'utilizzo dell'hardware della GPU per effettuare in parallelo, tramite più thread, le operazioni sui singoli dataset locali $D_\gamma$.

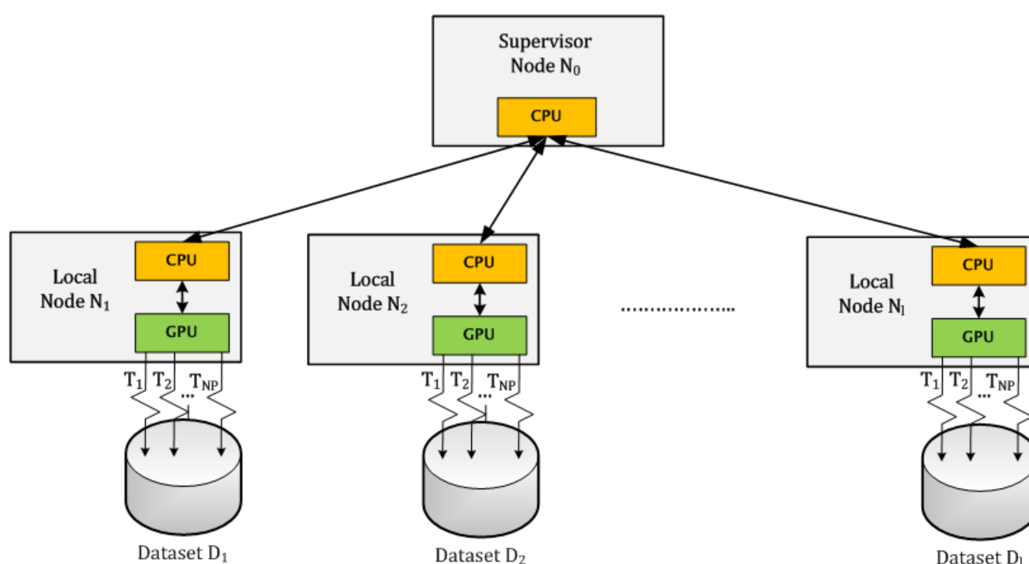



**Figura 7-3 Architettura distribuita multi-GPU**

Ogni nodo locale $N_\gamma$ contribuisce all'esecuzione dell'algoritmo distribuito attraverso l'attuazione delle procedure $NodeInit_\gamma()$, $NodeComp_\gamma()$ e $NodeReq_\gamma()$ (nel caso di utilizzo della variante *lazy*). Per poter sfruttare la potenza di calcolo della GPU è quindi necessario agire sull'implementazione di queste procedure. $NodeComp_\gamma()$ è l'operazione più onerosa in termini di lavoro svolto e viene eseguita dai nodi locali ad ogni iterazione dell'algoritmo. Per sua natura si presta molto bene ad una implementazione efficace in CUDA, in quanto richiede l'esecuzione di un numero elevato di calcoli, effettuati principalmente su dati locali ad ogni singolo nodo. Il tempo di computazione su GPU risulta essere di vari ordini di grandezza superiore a quello richiesto per i trasferimenti di dati (in input e in output) tra memoria della CPU e global memory della GPU. $NodeReq_\gamma()$ si occupa principalmente dell'invio delle distanze addizionali al supervisore e la quantità di calcoli richiesta per

l'aggiornamento delle strutture dati $LNNC_\gamma$ e $rLNNC_\gamma$ è relativamente molto bassa. Tale procedura non può quindi trarre reali vantaggi da un'implementazione su GPU e l'overhead dei trasferimenti CPU-GPU richiesti potrebbe perfino risultare superiore al tempo di calcolo su GPU. Di conseguenza conviene implementare questa procedura nel modo classico, con l'uso esclusivo della CPU. $NodeInit_\gamma()$ viene chiamata una sola volta durante l'esecuzione dell'algoritmo, per la selezione del primo insieme dei punti candidati e l'inizializzazione di altre strutture. E' opportuno quindi prevedere in questa fase anche l'allocazione delle varie strutture dati richieste dalla GPU ed il trasferimento del dataset e dei primi candidati sulla global memory della GPU.
Il nodo supervisore $N_0$ si occupa del coordinamento dell'esecuzione tra i vari nodi locali e dell'analisi dei risultati ricevuti. Anche in questo caso però la quantità di calcoli eseguita è relativamente bassa ed inoltre tali operazioni devono avvenire principalmente su dati ricevuti da altri nodi e non esclusivi al nodo stesso (come invece avviene ad esempio per $NodeComp_\gamma()$). Se si decidesse di utilizzare la GPU per rendere più rapidi i calcoli del supervisore, l'overhead dei trasferimenti CPU-GPU potrebbe diventare dominante e l'intera esecuzione potrebbe risultare più lenta rispetto a quella di un classico approccio basato solamente su CPU. Per questo motivo conviene mantenere di tipo sequenziale l'algoritmo eseguito dal nodo supervisore, che risulta quindi esssere invariato rispetto alla precedente implementazione.

Mostriamo quindi le modifiche da apportare alle due procedure $NodeInit_\gamma()$ e $NodeComp_\gamma()$, al fine di permettere l'utilizzo della GPU per rendere più veloci le operazioni eseguite sui nodi locali. Indichiamo con $CUDA_NodeInit_\gamma()$ e $CUDA_NodeComp_\gamma()$ le rispettive procedure modificate per l'ausilio della GPU, con *CUDA_ODPDistributedSolvingSet* e *CUDA_ODPLazyDistributedSolvingSet* i due algoritmi distribuiti che utilizzano $CUDA_NodeInit_\gamma()$ e $CUDA_NodeComp_\gamma()$, al posto $NodeInit_\gamma()$ e $NodeComp_\gamma()$.

### 7.2.1 Procedura CUDA_NodeInit

La procedura $NodeInit_\gamma()$ deve essere modificata per permettere l'allocazione e l'inizializzazione delle strutture dati richieste dalla GPU, nella global memory del device. Inoltre è necessario copiare nelle rispettive strutture della GPU l'intero dataset e il primo insieme di candidati locali, selezionato in modo casuale.

Le operazioni eseguite sono quindi le seguenti (in pseudo-codice):

```
CUDA_NodeInitγ(mγ, Cγ){
      allocateToGPU(D) //contenente il dataset
      allocateToGPU(C) //contenente tutti i candidati
      allocateToGPU(Cγ) //contenente i candidati locali
      allocateToGPU(NNγ) //contenente i max-heap NNγ[p] ∀ p ∈ Dγ
      allocateToGPU(LNNCγ) // i max-heap LNNCγ[q] ∀ q ∈ C
      allocateToGPU(LCγ) // min-heap i nuovi candidati
      allocateToGPU(…) //strutture dati specifiche per la
            //tecnica adottata per la fase compareDγwithC_upMin *
      allocateToGPU(GHMin) // contenente i min-heap temporanei
       //per compareDγwithC_upMax_phaseA, compareDγwithC_upMax_phaseB
      allocateToGPU(actγ) //per l'output del numero actγ
      allocateToGPU(Gcount) // array di interi temp.
       //per compareDγwithC_upMax_phaseA, compareDγwithC_upMax_phaseB

      Cγ := RandomSelect(Dγ, mγ)
      actγ := |Dγ|
      copyToGPU(D)
      copyToGPU(Cγ)
}
```

### 7.2.2 Procedura CUDA_NodeComp

Le modifiche alla procedura $NodeComp_{\gamma}()$ rappresentano il vero cuore della soluzione distribuita multi-GPU. Per poter essere efficacemente implementata in CUDA, la procedura deve essere suddivisa in diverse fasi, ognuna di esse formata da molteplici thread, eseguiti in parallelo sui diversi multiprocessori della GPU. Compito della CPU è il coordinamento dell'esecuzione delle diverse azioni sulla GPU e lo svolgimento di ulteriori operazioni utili per la logica dell'algoritmo stesso, effettuate in parallelo al lavoro svolto dalla GPU.

Possiamo quindi individuare le seguenti fasi:

- *Inizializzazione*: la CPU copia nella global memory della GPU il nuovo set di candidati $C$ ricevuto dal supervisore e aggiorna il numero $act_{\gamma}$ di punti attivi nel dataset locale. La GPU effettua la rimozione dei punti candidati locali dal dataset locale $D_{\gamma}$ ($D_{\gamma} := D_{\gamma} - C_{\gamma}$) e l'inizializzazione degli heap in $LNNC_{\gamma}$, tramite la copia del contenuto di $NN_{\gamma}[q_j]$ in $LNNC_{\gamma}[q_j]$, per ogni $q_j \in C_{\gamma}$. Queste operazioni avvengono tramite la kernel function $init()$.

- *Confronto $C_\gamma$ con $C$*: per ogni candidato locale $q_j \in C_\gamma$ la GPU calcola in parallelo le distanze da ogni $q_i \in C$ ed esegue l'aggiornamento del rispettivo heap $LNNC_\gamma[q_j]$, tramite la kernel function $compareC_\gamma withC()$.
- *Confronto $D_\gamma$ con $C$*: è la fase più critica che è opportuno suddividere a sua volta in più sottofasi, per poter operare efficientemente in parallelo tramite l'ambiente CUDA:
  - *upMin*: in cui la GPU calcola in parallelo le distanze per ogni coppia di punti $(p_i, q_j)$, con $p_i \in D_\gamma$ e $q_j \in C$, ed esegue l'aggiornamento dei rispettivi heap $NN_\gamma[p_i]$ e $LNNC_\gamma[q_j]$. Come per CUDA_ODPSolvingSet sono state previste diverse tecniche, specifiche per determinate situazioni.
  - *upMax*: in cui la GPU esegue l'aggiornamento dell'heap $LC_\gamma$, la dichiarazione dei punti $p_i \in D_\gamma$ come attivi o non attivi e il conteggio del nuovo numero $act_\gamma$ di punti rimasti attivi nel dataset locale. Queste azioni sono eseguite tramite due kernel function: $compareD_\gamma withC_upMax_phaseA()$ e $compareD_\gamma withC_upMax_phaseB()$.
- *Selezione dei nuovi candidati e invio risultati*: la GPU esegue la copia dei punti presenti nell'heap $LC_\gamma$in $C_\gamma$, tramite la kernel function $candSelect()$. La CPU recupera quindi dalla global memory della GPU il nuovo insieme dei candidati $C_\gamma$, gli heap $LNNC_\gamma$, $LC_\gamma$ e il valore $act_\gamma$. Nel caso in cui sia implementata la variante *lazy* effettua inoltre il sorting di $LNNC_\gamma[q]$ e trasferisce in $rLNNC_\gamma[q]$ le ultime $k - k_0$ distanze. Infine si ha l'invio al supervisore di $LNNC_\gamma$, $LC_\gamma$, e $act_\gamma$.

La procedura $CUDA_NodeComp_{\gamma}()$, in termini di operazioni eseguite dalla CPU, può quindi essere ristrutturata nel modo seguente (in pseudo-codice):

```
CUDA_NodeCompγ (MinTop, C, act, k0, LNNCγ, LCγ, actγ){
        copyToGPU(C)
        launchOnGPU(init(Dγ, C, Cγ, NNγ, LNNCγ))
        actγ := actγ − |Cγ|
        synchWithGPU()

        launchOnGPU(compareCγwithC(k, dist, C, Cγ, LNNCγ))
        synchWithGPU()

        compareDγwithC_upMin ∗ (k, dist, Dγ, C, NNγ, LNNCγ, MinTop, …)
                        //dipendente dalla tecnica adottata
        mγ := ⌈m actγ/act⌉
        launchOnGPU(
                compareDγwithC_upMax_phaseA(mγ, Dγ, NNγ, GHMin,
                                                MinTop, Gcount)
        synchWithGPU()
        launchOnGPU(
                compareDγwithC_upMax_phaseB(mγ, LCγ, GHMin,
                                                Gcount, actγ)
        synchWithGPU()

        launchOnGPU(candSelect(Cγ, LCγ))
        synchWithGPU()
        copyFromGPU(Cγ)
        copyFromGPU(LCγ)
        copyFromGPU(LNNCγ)
        copyFromGPU(actγ)

        //per la variante lazy:
        for each q in C {
                LNNCγ[q] := sort(LNNCγ[q])
                rLNNCγ[q] := get_last(LNNC[q]γ, k − k0)
                LNNCγ[q] := get_first(LNNCγ[q], k0)
        }
}
```

Si possono notare molti gradi di somiglianza con le operazioni eseguite nelle diverse fasi di CUDA_ODPSolvingSet, alcune operazioni risultano essere sostanzialmente identiche, altre richiedono qualche piccola modifica. Illustriamo quindi le diverse azioni eseguite dalla GPU ad ogni fase, cercando di rapportarle a quelle eseguite in CUDA_ODPSolvingSet.

#### 7.2.2.1 Fase di inizializzazione

In questa fase, tramite la kernel function $init()$, la GPU esegue la rimozione dei punti candidati locali dal dataset locale $D_{\gamma}$ ($D_{\gamma} := D_{\gamma} - C_{\gamma}$) e l'inizializzazione degli heap in $LNNC_{\gamma}$. A tal fine si definiscono $m$ thread $T_j$, un thread per ogni $q_j \in C$. Come prima operazione ogni $T_j$ si occupa dell'inizializzazione di $LNNC_{\gamma}[q_j]$, copiando nell'heap il contenuto di $NN_{\gamma}[q_j]$ se $q_j$ risulta essere un candidato locale al nodo $N_{\gamma}$ (ovvero se

$q_j \in C_\gamma$), altrimenti viene posto $LNNC_\gamma[q_j] = \emptyset$. Successivamente si ha la rimozione dal dataset locale $D_\gamma$ dei punti in $C_\gamma$, tramite la stessa modalità adottata per la kernel function $init()$ di CUDA_OPDSolvingSet, descritta nel paragrafo 6.1.

La procedura eseguita dalla kernel function, in pseudo-codice, è la seguente:

```
init(Dγ, C, Cγ, NNγ, LNNCγ) {
        <shared> c_idx[]
        <shared> subs_idx[]
        j := threadIdx.x

        if(qj ∈ Cγ){
                LNNC[qj] := NN[qj]
        }else{ //ovvero qj ∈ C − Cγ
                LNNC[qj] := ∅
        }
        if(j = 0){
                c_idx := sort(Cγ.indexInD)
                subs_idx := findSubsitutes(|Cγ|, D, c_idx)
                d := d − |Cγ|
        }
        __syncthreads()
        if(j < |Cγ| AND subs_idx[j] ≠ −1){
                D[c_idx[j]] := D[subs_idx[j]]
        }
}
con:
subs_idx[] findSubstitutes(dim, D, c_idx){
        t := d − 1
        for j := 0 to dim{
                subs_idx[j] := −1
                while(t > c_idx[j] AND subs_idx[j] = −1){
                        if(t not in c_idx){
                                subs_idx[j] := t
                        }
                        t := t − 1
                }
        }
        return subs_idx
}
```

### 7.2.2.2 Fase di confronto $C_\gamma$ con $C$

Durante questa fase, per ogni candidato locale $q_j \in C_\gamma$, la GPU calcola in parallelo le distanze da ogni $q_i \in C$ ed esegue l'aggiornamento del rispettivo heap $LNNC_\gamma[q_j]$, tramite la kernel function $compareC_\gamma withC()$. A tale scopo si definiscono $m_\gamma$ thread $T_j$, ognuno dedicato ad singolo punto $q_j \in C_\gamma$. Ogni thread $T_j$ calcola la distanza tra $q_j$ ed ogni altro punto $q_i \in C$ e chiama l'operazione $updateMin$ su $LNNC_\gamma[q_j]$ per ogni coppia $\langle q_i, \delta \rangle$, con $\delta = dist(q_i, q_j)$. Come nel caso della kernel function $compareCwithC()$ di CUDA_ODPSolvingSet definita nel paragrafo 6.2, per poter sfruttare al meglio il parallelismo della GPU è opportuno dividere i thread in più blocchi di dimensione $NT$, dimensionando $NT$ in modo tale da poter suddividere il lavoro tra più multiprocessori.

La procedura eseguita dalla kernel function, in pseudo-codice, è la seguente:

```
compareCγwithC(k, dist, C, Cγ, LNNCγ){
        j := blockIdx.x · blockDim.x + threadIdx.x

        for each q_i in C{
                δ := dist(q_i, q_j), con q_j ∈ Cγ
                updateMin(LNNCγ[q_j], ⟨q_i, δ⟩ )
        }
}
```

A differenza dell'algoritmo sequenziale per CPU non viene effettuata una distinzione tra candidati locali e candidati non locali, per cercare di limitare il più possibile il numero di distanze calcolate. Nel paragrafo 6.2 è stato mostrato come, ipotizzando $m$ relativamente piccolo se confrontato con il numero massimo $NP$ di thread contemporanei che può eseguire la GPU, dal punto di vista dell'esecuzione su GPU sia più conveniente non sfruttare la simmetria della matrice delle distanze tra i vari punti candidati e calcolare tutte le $m^2$ distanze. In questo contesto il motivo è molto simile. Nell'algoritmo sequenziale il confronto tra i punti candidati è spezzato in due distinti *loop*, in quanto la matrice delle distanze dei candidati locali in $C_\gamma$ risulta essere simmetrica, mentre l'intera matrice delle distanze tra i punti in $C_\gamma$ e i punti in $C$ normalmente no. Questo permette quindi di poter risparmiare il calcolo di metà delle distanze tra le coppie di candidati locali in $C_\gamma$, spezzando il confronto $C_\gamma$ con $C$ in due diversi step. Nel caso di esecuzione su GPU non è invece conveniente sfruttare la simmetria della matrice delle distanze dei candidati locali in $C_\gamma$, in quanto $m_\gamma$ risulta essere tipicamente molto inferiore ad $NP$, riconducendoci alle osservazioni effettuate in 6.2. E' quindi preferibile una soluzione che calcoli in maniera parallela tutte le $m_\gamma \cdot m$ distanze, cercando di sfruttare al meglio l'hardware della GPU.

### 7.2.2.3 Fase di confronto $D_\gamma$ con $C$ - upMin

In questa prima fase del confronto $D_\gamma$ con $C$ la GPU calcola in parallelo le distanze per ogni coppia di punti $(p_i, q_j)$, con $p_i \in D_\gamma$ e $q_j \in C$, ed esegue l'aggiornamento dei rispettivi heap $NN_\gamma[p_i]$ e $LNNC_\gamma[q_j]$. Possiamo osservare che le operazioni richieste sono esattamente le medesime eseguite nella fase di confronto $D$ con $C$ – upMin di CUDA_ODPSolvingSet, considerando il dataset locale $D_\gamma$ al posto di $D$. Di conseguenza è possibile utilizzare le diverse strategie (con matrice delle distanze o senza) proposte nel paragrafo 6.3 anche per la fase di confronto $D_\gamma$ con $C$ – upMin della procedura $CUDA_NodeComp_\gamma$().

#### 7.2.2.4 Fase di confronto $D_\gamma$ con $C$ - upMax

Nella seconda fase del confronto $D_\gamma$ con $C$ la GPU esegue l'aggiornamento dell'heap $LC_\gamma$, la dichiarazione dei punti $p_i \in D_\gamma$ come attivi o non attivi e il conteggio del nuovo numero $act_\gamma$ di punti rimasti attivi nel dataset locale. Le operazioni effettuate sono le medesime di quelle eseguite dalle due kernel function utilizzate nella fase di confronto $D$ con $C$ – upMax dall'algoritmo CUDA_ODPSolvingSet (paragrafo 6.4), a cui si aggiunge la necessità del calcolo di $act_\gamma$.

Anche in questo caso lavoriamo in due fasi, tramite due distinte kernel function: $compareD_\gamma withC_upMax_phaseA$() e $compareD_\gamma withC_upMax_phaseB$().
Per la prima fase predisponiamo $NB_A$ blocchi di $NT_A$ thread. Associamo ad ogni thread $T_z$ (con $z = 0,1,\dots,NB_A \cdot NT_A - 1$) un proprio min-heap $H_z$ di dimensione $m_\gamma$, contenente le coppie $\langle p_i, w_k(p_i, D_\gamma)\rangle$ con $p_i \in D_\gamma$ ed una variabile intera $count_z$, per il conteggio dei punti attivi locali, entrambi allocati in shared memory. Ogni $T_z$ si occupa inizialmente della selezione degli $m_\gamma$ punti di maggior peso tra $d_\gamma/(NB_A \cdot NT_A)$ elementi in $w = \left[Sum\left(NN_\gamma[p_0]\right), \dots, Sum(NN_\gamma[p_{d_\gamma - 1}])\right]$, sfruttando l'operazione $updateMax$ su $H_z$ e considerando solamente i punti $p_i \in D_\gamma$ attivi per cui valga la condizione $Sum\left(NN_\gamma[p_i]\right) \geq MinTop$. Se il weight upper bound di $p_i$ si trova al di sotto della soglia $MinTop$, il punto viene etichettato come non attivo. Inoltre ogni thread $T_z$ aggiorna la variabile $count_z$, ponendola pari al numero di punti $p_i$ analizzati da $T_z$ e rimasti attivi. Per ogni blocco viene quindi eseguita la fusione dei vari heap, tramite la tecnica della riduzione parallela e l'heap risultante per ogni blocco viene salvato in global memory. Sempre durante il medesimo processo di riduzione, avviene il calcolo della somma degli interi $count_z$ per ogni singolo blocco. Nel primo step si sommano in parallelo a due a due $NT_A$ interi ottenendo $NT_A/2$ somme, che vengono salvate nelle prime $NT_A/2$ variabili $count_z$ del blocco. Nel secondo si sommano a due a due tali somme, ottenendo questa volta $NT_A/4$ risultati parziali e si procede con questa tecnica fino a quando per ogni blocco si ha un'unica variabile, contenente la somma di tutti gli interi $count_z$ del blocco. Il risultato finale del blocco $B_j$ viene salvato nella $j$-esima posizione di un vettore di $NB_A$ interi $Gcount$, allocato in global memory.
Per la seconda fase utilizziamo un solo blocco di $NT_B$ thread ed anche in questo caso associamo ad ogni thread $T_z'$ (con $z = 0,1,\dots,NT_B - 1$) un min-heap $H_z'$ di dimensione $m_\gamma$ ed un intero $count_z$, in shared memory. Ogni $T_z'$ esegue la fusione di

$NB_A/NT_B$ heap salvati in global memory con il proprio heap $H_z'$ e la somma in $count_z$ dei rispettivi valori in $Gcount$. Successivamente vengono fusi gli heap $H_0', \ldots, H_{NT_B-1}'$ ed eseguita la somma dei $count_z$, tramite la riduzione parallela. L'heap risultante coincide con il min-heap $LC$ e la somma dei $count_z$ con $act_\gamma$.

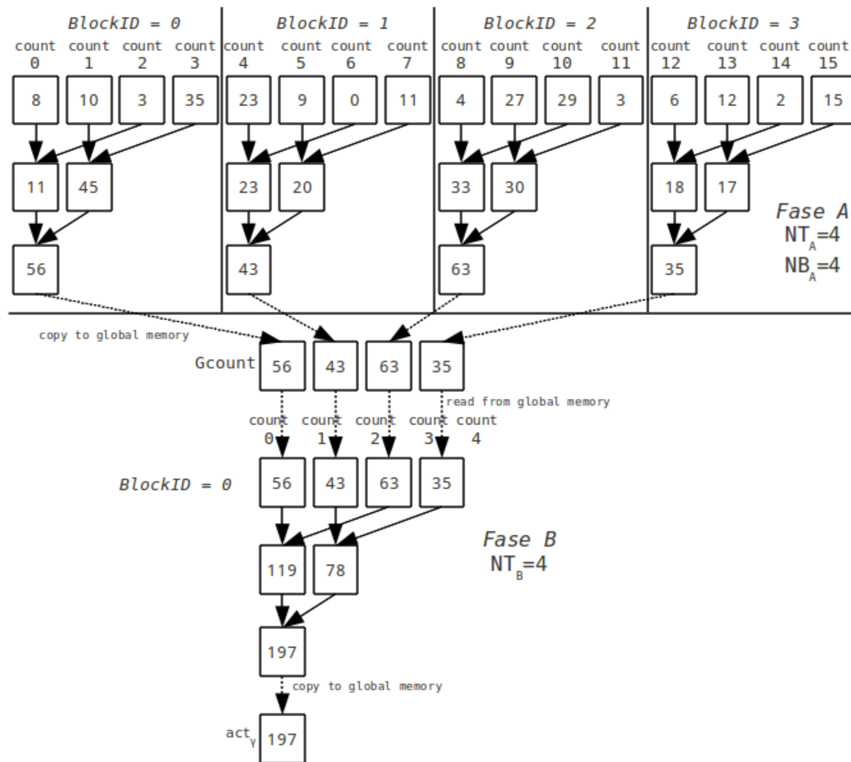



**Figura 7-4 Calcolo $act_\gamma$**

Le procedure eseguite dalle due kernel function sono le seguenti (in pseudo-codice):

```
compareDγwithC_upMax_phaseA
 (m_γ, D_γ, NN_γ, GHMin, MinTop, Gcount) {

   tid := threadIdx.x
   bid := blockIdx.x

   <shared> H_0, H_1,..., H_{NT_A-1}
   <shared> count_0, count_1,..., count_{NT_A-1}

   H_tid := Ø
   count_tid := 0

   i := bid · NT_A + tid
   while (i < d) {
    if(p_i.active = true AND
     Sum(NN[p_i]) ≥ Min(Top), con p_i ∈ D){
       updateMax(H_tid, ⟨p_i, Sum(NN[p_i])⟩)
       count_tid := count_tid + 1
    }else{
     p_i.active := false
    }
    i := i + NT_A · NB_A
   }
   __syncthreads()
```

```
compareDγwithC_upMax_phaseB
 (m_γ, LC_γ, GHMin, Gcount, act_γ) {

   tid := threadIdx.x


   <shared> H_0, H_1,..., H_{NT_B-1}
   <shared> count_0, count_1,..., count_{NT_B-1}

   H_tid := Ø
   count_tid := 0

   i := tid
   while (i < NB_A) {
       minheapFusion(H_tid, GHMin[i])
       count_tid := count_tid + count_i
       i := i + NT_B
   }
   __syncthreads()
```

```
    //riduzione in shared mem
    if(NT_A ≥ 512) {
        if(tid < 256) {
            minheapFusion(H_tid, H_tid+256)
            count_tid := count_tid + count_tid+256
        }
        __syncthreads()
    }
    if(NT_A ≥ 256) {
        if(tid < 128) {
            minheapFusion(H_tid, H_tid+128)
            count_tid := count_tid + count_tid+128
        }
        __syncthreads()
    }
    if(NT_A ≥ 128) {
        if(tid < 64) {
            minheapFusion(H_tid, H_tid+64)
            count_tid := count_tid + count_tid+64
        }
        __syncthreads()
    }
    if(NT_A ≥ 64) {
        if(tid < 32) {
            minheapFusion(H_tid, H_tid+32)
            count_tid := count_tid + count_tid+32
        }
    }
    if(NT_A ≥ 32) {
        if(tid < 16) {
            minheapFusion(H_tid, H_tid+16)
            count_tid := count_tid + count_tid+16
        }
    }
    if(NT_A ≥ 16) {
        if(tid < 8) {
            minheapFusion(H_tid, H_tid+8)
            count_tid := count_tid + count_tid+8
        }
    }
    if(NT_A ≥ 8) {
        if(tid < 4) {
            minheapFusion(H_tid, H_tid+4)
            count_tid := count_tid + count_tid+4
        }
    }
    if(NT_A ≥ 4) {
        if(tid < 2) {
            minheapFusion(H_tid, H_tid+2)
            count_tid := count_tid + count_tid+2
        }
    }
    if(NT_A ≥ 2) {
        if(tid < 1) {
            minheapFusion(H_tid, H_tid+1)
            count_tid := count_tid + count_tid+1
        }
    }
    if (tid = 0) {
        GHMin[bid] := H_0
        Gcount[bid] := count_0
    }
}
```

```
    //riduzione in shared mem
    if(NT_B ≥ 512) {
        if(tid < 256) {
            minheapFusion(H_tid, H_tid+256)
            count_tid := count_tid + count_tid+256
        }
        __syncthreads()
    }
    if(NT_B ≥ 256) {
        if(tid < 128) {
            minheapFusion(H_tid, H_tid+128)
            count_tid := count_tid + count_tid+128
        }
        __syncthreads()
    }
    if(NT_B ≥ 128) {
        if(tid < 64) {
            minheapFusion(H_tid, H_tid+64)
            count_tid := count_tid + count_tid+64
        }
        __syncthreads()
    }
    if(NT_B ≥ 64) {
        if(tid < 32) {
            minheapFusion(H_tid, H_tid+32)
            count_tid := count_tid + count_tid+32
        }
    }
    if(NT_B ≥ 32) {
        if(tid < 16) {
            minheapFusion(H_tid, H_tid+16)
            count_tid := count_tid + count_tid+16
        }
    }
    if(NT_B ≥ 16) {
        if(tid < 8) {
            minheapFusion(H_tid, H_tid+8)
            count_tid := count_tid + count_tid+8
        }
    }
    if(NT_B ≥ 8) {
        if(tid < 4) {
            minheapFusion(H_tid, H_tid+4)
            count_tid := count_tid + count_tid+4
        }
    }
    if(NT_B ≥ 4) {
        if(tid < 2) {
            minheapFusion(H_tid, H_tid+2)
            count_tid := count_tid + count_tid+2
        }
    }
    if(NT_B ≥ 2) {
        if(tid < 1) {
            minheapFusion(H_tid, H_tid+1)
            count_tid := count_tid + count_tid+1
        }
    }
    if (tid = 0) {
        LC := H_0
        act_γ := count_0
    }
}
```

#### 7.2.2.5 Fase di selezione dei nuovi candidati locali

Durante quest'ultima fase la GPU effettua la copia dei punti presenti nell'heap $LC_\gamma$ all'interno dell'insieme $C_\gamma$. Tale operazione è svolta tramite la kernel function $candSelect()$, impiegando $m_\gamma$ thread $T_j$, dove ogni thread $T_j$ si occupa semplicemente della copia del $j$-esimo elemento di $LC_\gamma$in $C_\gamma$.

Implementando l'insieme $C_\gamma$ come un vettore di $m_\gamma$ punti allocato in global memory, la kernel function eseguita dalla GPU è la seguente (in pseudo-codice):

```
candSelect(Cγ,LCγ) {
        j := threadIdx.x
        Cγ[j] := getPoint(j, LCγ)
}
```

### 7.2.3 Complessità computazionale

Per il calcolo della complessità computazionale delle operazioni svolte da ogni nodo locale possiamo focalizzare l'attenzione solamente sull'esecuzione della procedura $CUDA_NodeComp_\gamma()$, essendo dominante sulle altre. Consideriamo quindi le singole fasi eseguite dalla GPU durante tale procedura. Ipotizziamo che la GPU sia in grado di eseguire contemporaneamente al più $NP$ thread e che, per semplicità, $d_\gamma$ sia multiplo di $NP$. Nel caso in cui sia presente nella GPU un numero superiore di thread, l'esecuzione di essi viene sequenzializzata, suddividendoli in gruppi di $NP$ thread alla volta.

La fase di inizializzazione presenta una complessità pari a $O(m_\gamma \cdot \log(m_\gamma))$, in quanto la funzione $findSubsitutes()$ e il sorting degli indici dei punti in $C_\gamma$ (con algoritmo di tipo heapsort) richiedono un tempo proporzionale a $O(m_\gamma \cdot \log(m_\gamma))$.

Per la fase di confronto $C_\gamma$ con $C$ si hanno $m_\gamma$ thread, dove ogni thread esegue $m$ calcoli di distanza ed $m$ operazioni di tipo $updateMin$. Ipotizzando $a$ pari alla dimensionalità dei punti del dataset (ovvero il numero di coordinate dei punti), tale fase presenta quindi una complessità pari a $O\left(\left\lceil\frac{m_\gamma}{NP}\right\rceil \cdot m(a + \log(k))\right) =$
$O(m(a + \log(k)))$, ipotizzando $NP > m_\gamma$.

Per la fase di confronto $D_\gamma$ con $C$ – upMin possiamo utilizzare una delle modalità proposte per la fase di confronto $D$ con $C$ – upMin dell'algoritmo CUDA_ODPSolvingSet (paragrafo 6.3). Per quanto osservato nel paragrafo 6.6, possiamo affermare che tale fase presenta una complessità pari a
$O\left(\frac{d_\gamma}{NP} \cdot m(a + 2\log(k))\right)$.

Per la fase di confronto $D_\gamma$ con $C$ – upMax si hanno $NB_A$ blocchi di $NT_A$ thread nel primo step ed un solo blocco di $NT_B$ thread nel secondo. Tale procedura presenta una complessità pari a $O\left(\frac{d_\gamma}{NP}\log(m)\right)$, per le osservazioni del paragrafo 6.6.

La fase finale di selezione dei nuovi candidati locali richiede un tempo proporzionale a $O\left(\left\lceil\frac{m_\gamma}{NP}\right\rceil \cdot a\right) = O(a)$ ipotizzando $NP > m_\gamma$, in quanto tale operazione è eseguita da $m_\gamma$ thread, uno per ogni singolo punto candidato locale.

Nel caso di utilizzo della variante *lazy* la CPU si occupa infine del sorting degli $m$ heap in $LNNC_\gamma$, che richiede un tempo proporzionale a $O(m \cdot k \cdot \log(k))$, e della suddivisione delle distanze tra $LNNC_\gamma$ e $rLNNC_\gamma$.

Possiamo quindi concludere che la fase dominante risulta essere quella di confronto $D_\gamma$ con $C$ – upMin e di conseguenza ignorare le tempistiche delle altri fasi, in questo contesto. La complessità di $CUDA_NodeComp_\gamma()$ è quindi pari a
$O\left(\frac{d_\gamma}{NP} \cdot m(a + 2\log(k))\right)$.

Supponiamo per semplicità che il dataset $D$ sia distribuito in maniera uniforme tra i vari nodi locali, ovvero che $d_\gamma = \frac{d}{l}$, per $\gamma = 1, \ldots, l$. Siccome nel caso peggiore l'algoritmo esegue $\left\lceil\frac{d}{m}\right\rceil$ iterazioni, la complessità temporale del lavoro in carico ad un singolo nodo locale è pari a $O\left(\left\lceil\frac{d}{m}\right\rceil \cdot \frac{d}{l \cdot NP} \cdot m \cdot (\mathrm{a} + 2\log(k))\right) \cong$
$O\left(\frac{d^2}{l \cdot NP} \cdot (\mathrm{a} + 2\log(k))\right)$. Le operazioni eseguite dal nodo supervisore rimangono invariate, in quanto l'ausilio della GPU è utilizzato solamente dai nodi locali, e di conseguenza presentano una complessità pari a $O\big(d \cdot (k \cdot \log(k) + \log(n))\big)$, nel caso di utilizzo della variante *lazy*. Ipotizzando $d \gg l \cdot NP$, con $d$ sufficientemente grande, il costo dell'esecuzione su un singolo nodo locale è ancora nettamente dominante rispetto al costo del lavoro eseguito dal supervisore e possiamo affermare che l'algoritmo ha complessità temporale pari a

$$O\left(\frac{d^2}{l \cdot NP} \cdot (\mathrm{a} + 2\log(k))\right).$$

Se confrontiamo tale complessità con quella dell'algoritmo distribuito ODPLazyDistributedSolvingSet che utilizza solamente la CPU sui nodi locali, si può

notare che in via teorica è possibile scalare di un fattore pari a $NP$. Inoltre questa soluzione risulta essere $l \cdot NP$ volte più veloce dell'algoritmo sequenziale ODPSolvingSet. C'è però infine da sottolineare che al crescere del fattore $l \cdot NP$ il tempo di esecuzione del nodo supervisore ed anche il tempo di comunicazione diventano sempre più rilevanti. Nel caso di utilizzo di molti nodi locali dotati di GPU particolarmente potenti, i tempi di computazione dei nodi locali tendono fortemente a decrescere, diventando comparabili con quelli di calcolo del supervisiore e di comunicazione, a meno dell'utilizzo di dataset di dimensioni molto elevate. Questo fenomeno può quindi diventare un importante elemento limitativo dello speedup raggiungibile con una soluzione distribuita multi-GPU, rispetto ad una soluzione centralizzata basata su una singola GPU.

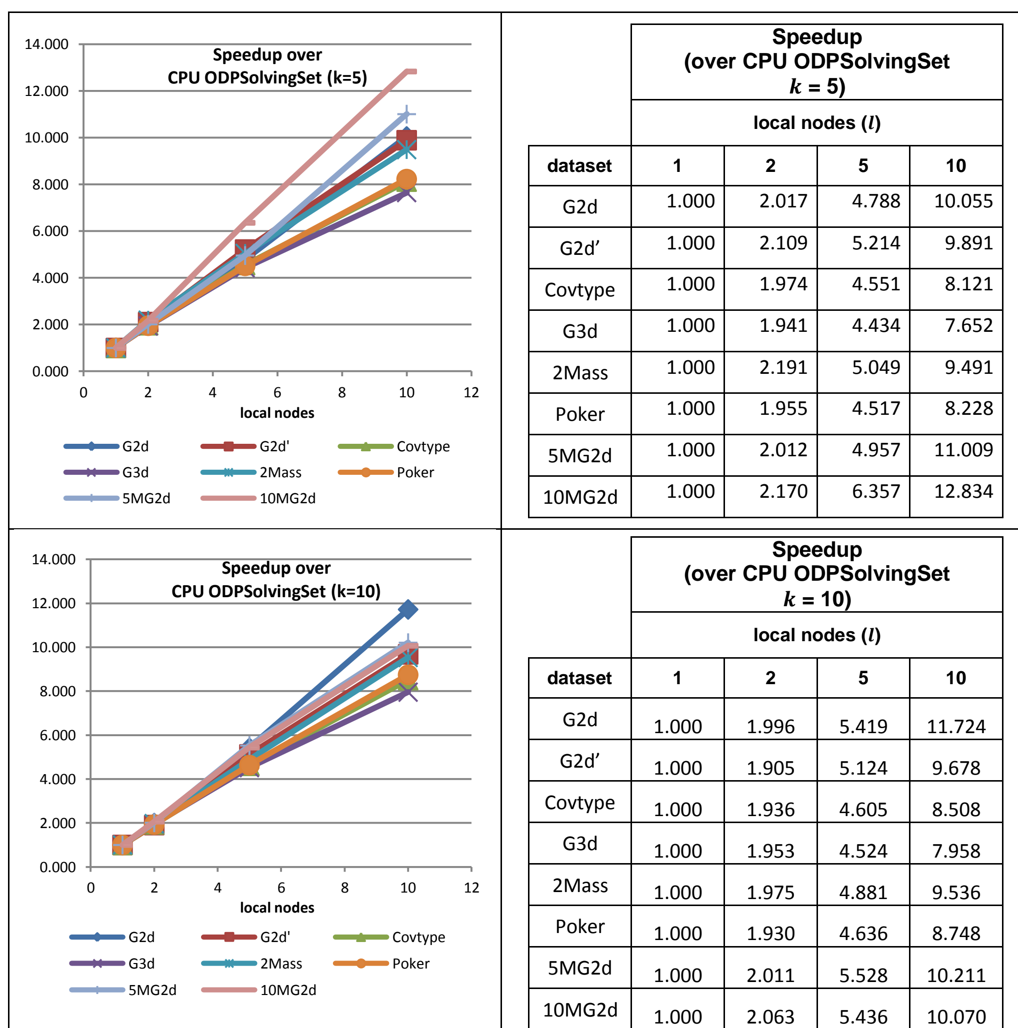
## 7.3 Risultati sperimentali

Per studiare le prestazioni di questi approcci distribuiti sono state eseguite due batterie di test, la prima riguardante la versione tradizionale basata su CPU e la seconda riguardante la versione multi-GPU. Come piattaforma di testing è stato utilizzato un numero variabile di nodi del supercomputer *IBM PLX*. Ricordiamo che ogni nodo è dotato di due CPU *six-cores Intel Westmere* a 2.40 *GHz* (*E5645*) e di due GPU *NVIDIA Tesla M2070*. I vari nodi sono interconnessi tramite una rete *InfiniBand*, dotata di switch di tipo *4x QDR*, per una banda teorica di circa 40 Gb/s, reale di al più 32 Gb/s. Gli esperimenti sono stati eseguiti facendo variare il numero di nodi locali dell'algoritmo da 1 a 10, allocando il nodo supervisore e i nodi locali su distinti nodi fisici del supercomputer. Come dataset di input sono stati utilizzati i cinque insiemi di dati presentati nel paragrafo 2.5.1, i due dataset gaussiani di 5 milioni e 10 milioni di punti illustrati in 6.7.2 ed un nuovo dataset *G2d'*. Ogni dataset (di dimensione $d$) è stato suddiviso in egual misura tra i diversi nodi locali, selezionando da esso per ogni nodo locale $d_\gamma = \frac{d}{l}$ punti, in maniera casuale, a meno che per *G2d'*. *G2d'* è un dataset derivato da *G2d*, contenente gli stessi elementi, ma distribuito tra i diversi nodi in maniera intenzionalmente polarizzata, in modo che tutti gli $n$ outlier siano allocati presso un unico nodo locale. *G2d'* rappresenta quindi il peggior scenario possibile. Come parametri degli algoritmi è stato posto $n = 10,\ m = 100$ e sono state effettuate prove per $k = 5, 10, 50$ per i primi cinque dataset, solamente per $k = 5, 10$ per i due dataset gaussiani da 5 e 10 milioni di punti (per $k = 50$ l'algoritmo per CPU con un singolo nodo locale non è risultato in grado di terminare l'esecuzione entro il limite di tempo prefissato per questi due dataset). Inoltre gli esperimenti sono stati eseguiti

utilizzando solamente gli algoritmi nella variante *lazy* (ovvero ODPLazyDistributedSolvingSet e CUDA_ODPLazyDistributedSolvingSet). Il codice riguardante le azioni svolte dal nodo supervisore e tutto il protocollo di comunicazione tra nodi locali e supervisor è stato implementato in *Java*. Le procedure eseguite dai nodi locali sono state realizzate sempre in *Java* nella versione tradizionale per CPU, mentre nella versione per GPU è stato realizzato un modulo CUDA *cubin* e la parte di comunicazione e coordinamento con la GPU è stata implementata in *Java*, sfruttando la libreria *JCuda*. Come supporto di trasmissione sono state utilizzate le classiche *Java socket*, utilizzando il protocollo *TCP/IP*.

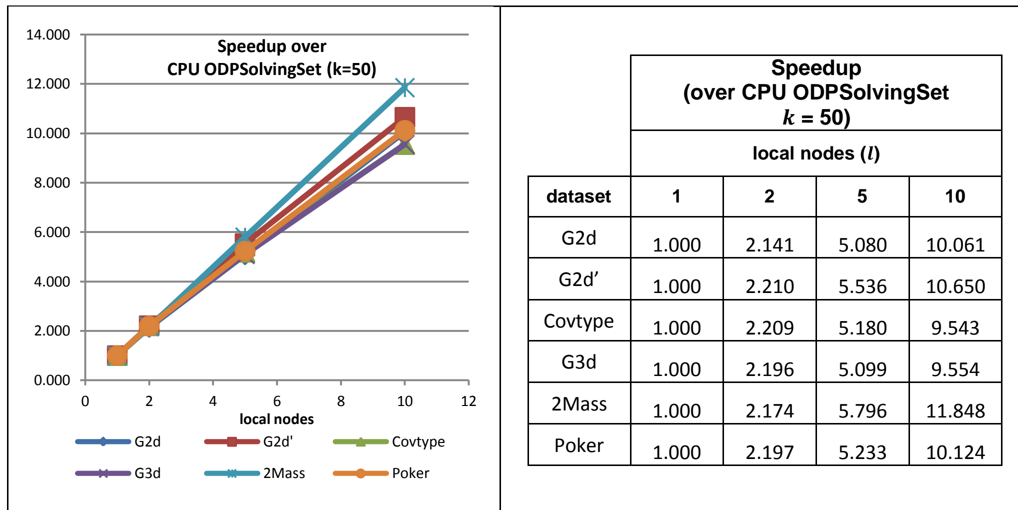
### 7.3.1 Performance di ODPLazyDistributedSolvingSet

In questa prima batteria di test è stato studiato lo speedup dell'algoritmo distribuito su quello centralizzato (nelle versioni CPU), definito come il rapporto tra i tempi di esecuzione di ODPSolvingSet e di ODPLazyDistributedSolvingSet, al variare del numero di nodi locali. La seguente figura mostra i risultati ottenuti.




| | Speedup (over CPU ODPSolvingSet *k* = 5) | | | |
|---|---|---|---|---|
| | local nodes (*l*) | | | |
| dataset | 1 | 2 | 5 | 10 |
| G2d | 1.000 | 2.017 | 4.788 | 10.055 |
| G2d' | 1.000 | 2.109 | 5.214 | 9.891 |
| Covtype | 1.000 | 1.974 | 4.551 | 8.121 |
| G3d | 1.000 | 1.941 | 4.434 | 7.652 |
| 2Mass | 1.000 | 2.191 | 5.049 | 9.491 |
| Poker | 1.000 | 1.955 | 4.517 | 8.228 |
| 5MG2d | 1.000 | 2.012 | 4.957 | 11.009 |
| 10MG2d | 1.000 | 2.170 | 6.357 | 12.834 |

| | Speedup (over CPU ODPSolvingSet *k* = 10) | | | |
|---|---|---|---|---|
| | local nodes (*l*) | | | |
| dataset | 1 | 2 | 5 | 10 |
| G2d | 1.000 | 1.996 | 5.419 | 11.724 |
| G2d' | 1.000 | 1.905 | 5.124 | 9.678 |
| Covtype | 1.000 | 1.936 | 4.605 | 8.508 |
| G3d | 1.000 | 1.953 | 4.524 | 7.958 |
| 2Mass | 1.000 | 1.975 | 4.881 | 9.536 |
| Poker | 1.000 | 1.930 | 4.636 | 8.748 |
| 5MG2d | 1.000 | 2.011 | 5.528 | 10.211 |
| 10MG2d | 1.000 | 2.063 | 5.436 | 10.070 |




| | Speedup (over CPU ODPSolvingSet $k$ = 50) | | | |
|---|---|---|---|---|
| | local nodes ($l$) | | | |
| dataset | 1 | 2 | 5 | 10 |
| G2d | 1.000 | 2.141 | 5.080 | 10.061 |
| G2d' | 1.000 | 2.210 | 5.536 | 10.650 |
| Covtype | 1.000 | 2.209 | 5.180 | 9.543 |
| G3d | 1.000 | 2.196 | 5.099 | 9.554 |
| 2Mass | 1.000 | 2.174 | 5.796 | 11.848 |
| Poker | 1.000 | 2.197 | 5.233 | 10.124 |

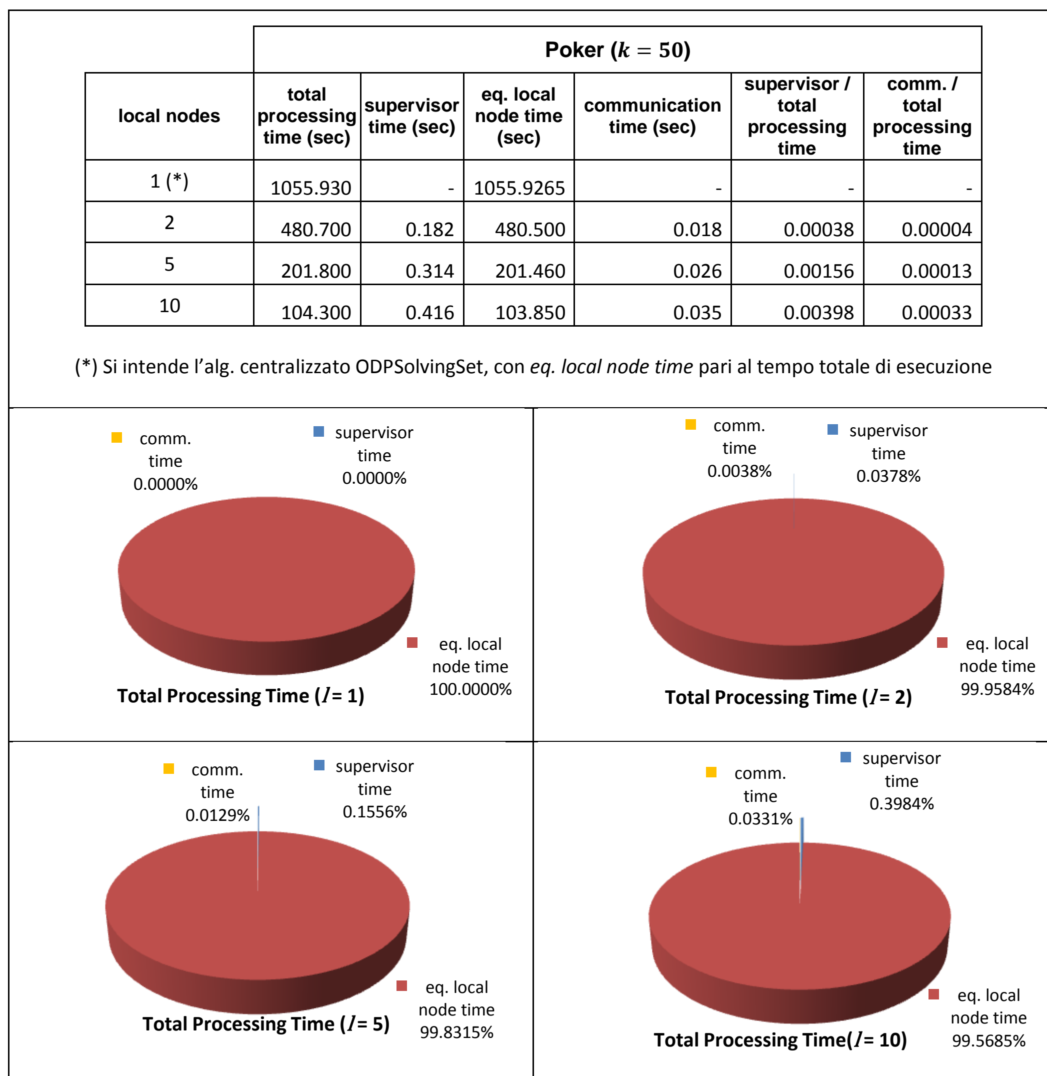
**Figura 7-5 Speedup di ODPLazyDistributedSolvingSet su ODPSolvingSet ($m$=100, $n$=10)**

E' possbile notare come, per tutti i dataset e per tutti i valori di $k$, l'algoritmo riesca a scalare molto bene all'aumentare del numero di nodi, mostrando un comportamento quasi lineare. Anche nel caso del dataset *G2d'* le performance sono molto buone.

Confrontiamo ora i tempi di esecuzione dei nodi locali con i tempi di comunicazione e di esecuzione del supervisore. Al fine di quantificare correttamente il tempo di esecuzione dei nodi locali, introduciamo il concetto di *equivalent local node processing time*. Sia $T_{i,\gamma}$ il tempo di esecuzione necessario per lo svolgimento delle procedure $Init_\gamma()$, $NodeComp_\gamma()$ e $NodeReq_\gamma()$ durante l'$i$-esima iterazione dell'algoritmo per il nodo locale $N_\gamma$ e sia $t$ il numero totale di iterazioni effettuate. Definiamo con *equivalent local node processing time* la somma:

$$\max(T_{1,1}, T_{1,2}, \dots, T_{1,l}) + \max(T_{2,1}, T_{2,2}, \dots, T_{2,l}) + \dots + \max(T_{t,1}, T_{t,2}, \dots, T_{t,l}).$$

Dai test è apparso in maniera evidente come il tempo di esecuzione sui nodi locali risulti essere in assoluto dominante. Il tempo di comunicazione ed il tempo di esecuzione sul nodo supervisore rappresentano una fetta molto piccola del tempo totale di esecuzione. In particolare consideriamo il caso specifico dei test effettuati sul dataset *Poker* con $k = 50$, i cui risultati sono mostrati nella figura 7.6. Si può notare come aumentando il numero di nodi locali il tempo di comunicazione e il tempo di esecuzione del nodo supervisore tendano a crescere. Se però confrontiamo tali valori con il tempo totale di esecuzione, si ha che, anche nel caso di utilizzo di 10 nodi locali, il tempo di esecuzione del supervisore non supera lo 0.4% del tempo totale e il tempo di comunicazione risulta essere perfino inferiore allo 0.04%. Questo fattore è quindi il principale motivo che giustifica le ottime performance dell'algoritmo ODPLazyDistributedSolvingSet.



| local nodes | Poker ($k$ = 50) | | | | | |
|---|---|---|---|---|---|---|
| | total processing time (sec) | supervisor time (sec) | eq. local node time (sec) | communication time (sec) | supervisor / total processing time | comm. / total processing time |
| 1 (*) | 1055.930 | - | 1055.9265 | - | - | - |
| 2 | 480.700 | 0.182 | 480.500 | 0.018 | 0.00038 | 0.00004 |
| 5 | 201.800 | 0.314 | 201.460 | 0.026 | 0.00156 | 0.00013 |
| 10 | 104.300 | 0.416 | 103.850 | 0.035 | 0.00398 | 0.00033 |

(*) Si intende l'alg. centralizzato ODPSolvingSet, con *eq. local node time* pari al tempo totale di esecuzione

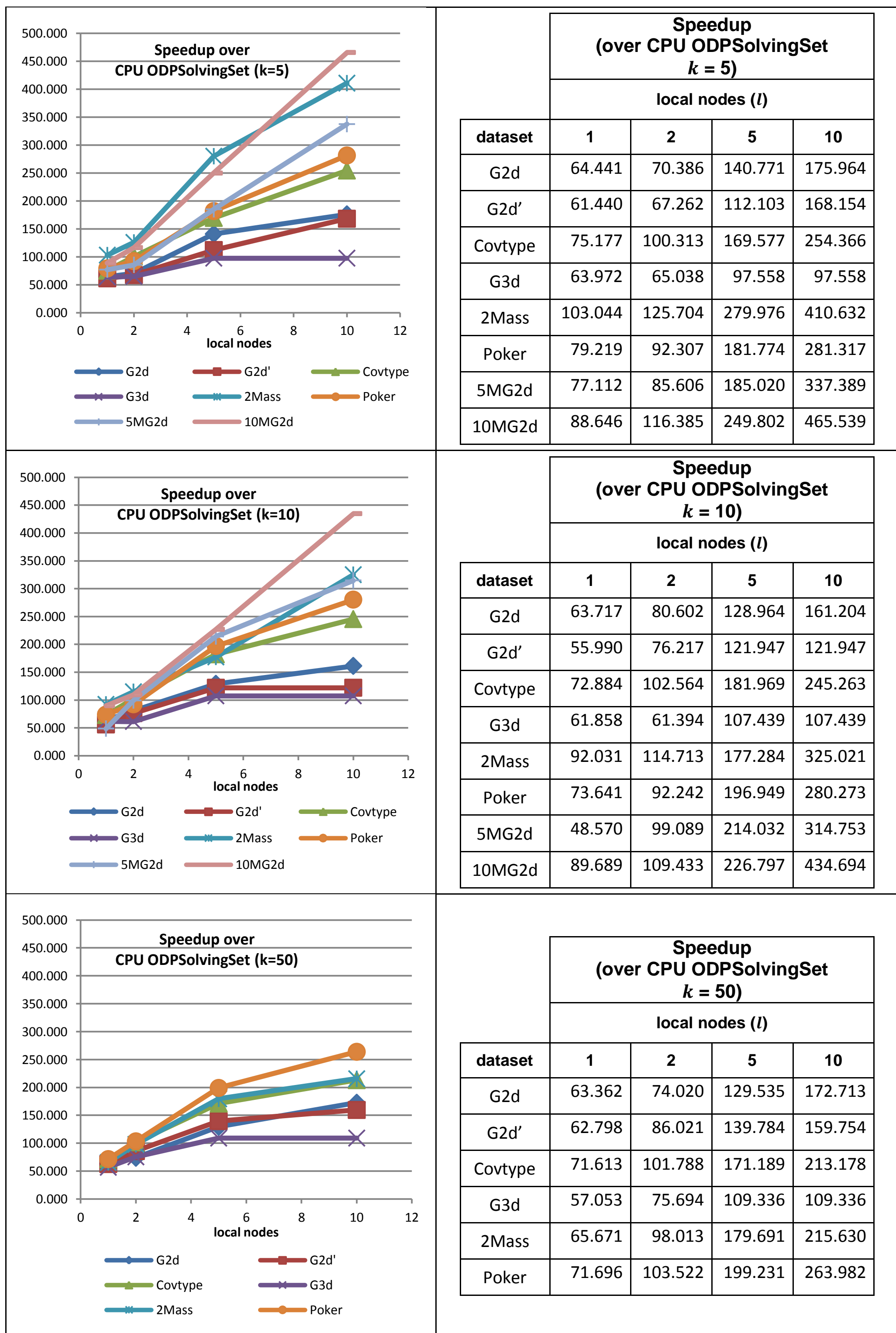


**Figura 7-6 Analisi dei tempi di esecuzione di ODPLazyDistributedSolvingSet per il dataset *Poker* (*k*=50, *m*=100, *n*=10)**

### 7.3.2 Performance di CUDA_ODPLazyDistributedSolvingSet

Per la seconda batteria di esperimenti è stata utilizzata invece la versione distribuita multi-GPU, ovvero la soluzione che abbiamo indicato con il termine CUDA_ODPLazyDistributedSolvingSet.

Consideriamo lo speedup dell'algoritmo distribuito multi-GPU su quello centralizzato nella versione CPU, definito come il rapporto tra i tempi di esecuzione di ODPSolvingSet e di CUDA_ODPLazyDistributedSolvingSet, al variare del numero di nodi locali. I risultati dei test dimostrano come sia possibile ottenere un buon incremento dello speedup utilizzando contemporaneamente più nodi locali, ognuno dotato di una propria GPU per l'esecuzione delle operazioni previste nella procedura $CUDA_NodeComp_{\gamma}()$.




| | Speedup (over CPU ODPSolvingSet $k$ = 5) | | | |
|---|---|---|---|---|
| | local nodes ($l$) | | | |
| dataset | 1 | 2 | 5 | 10 |
| G2d | 64.441 | 70.386 | 140.771 | 175.964 |
| G2d’ | 61.440 | 67.262 | 112.103 | 168.154 |
| Covtype | 75.177 | 100.313 | 169.577 | 254.366 |
| G3d | 63.972 | 65.038 | 97.558 | 97.558 |
| 2Mass | 103.044 | 125.704 | 279.976 | 410.632 |
| Poker | 79.219 | 92.307 | 181.774 | 281.317 |
| 5MG2d | 77.112 | 85.606 | 185.020 | 337.389 |
| 10MG2d | 88.646 | 116.385 | 249.802 | 465.539 |

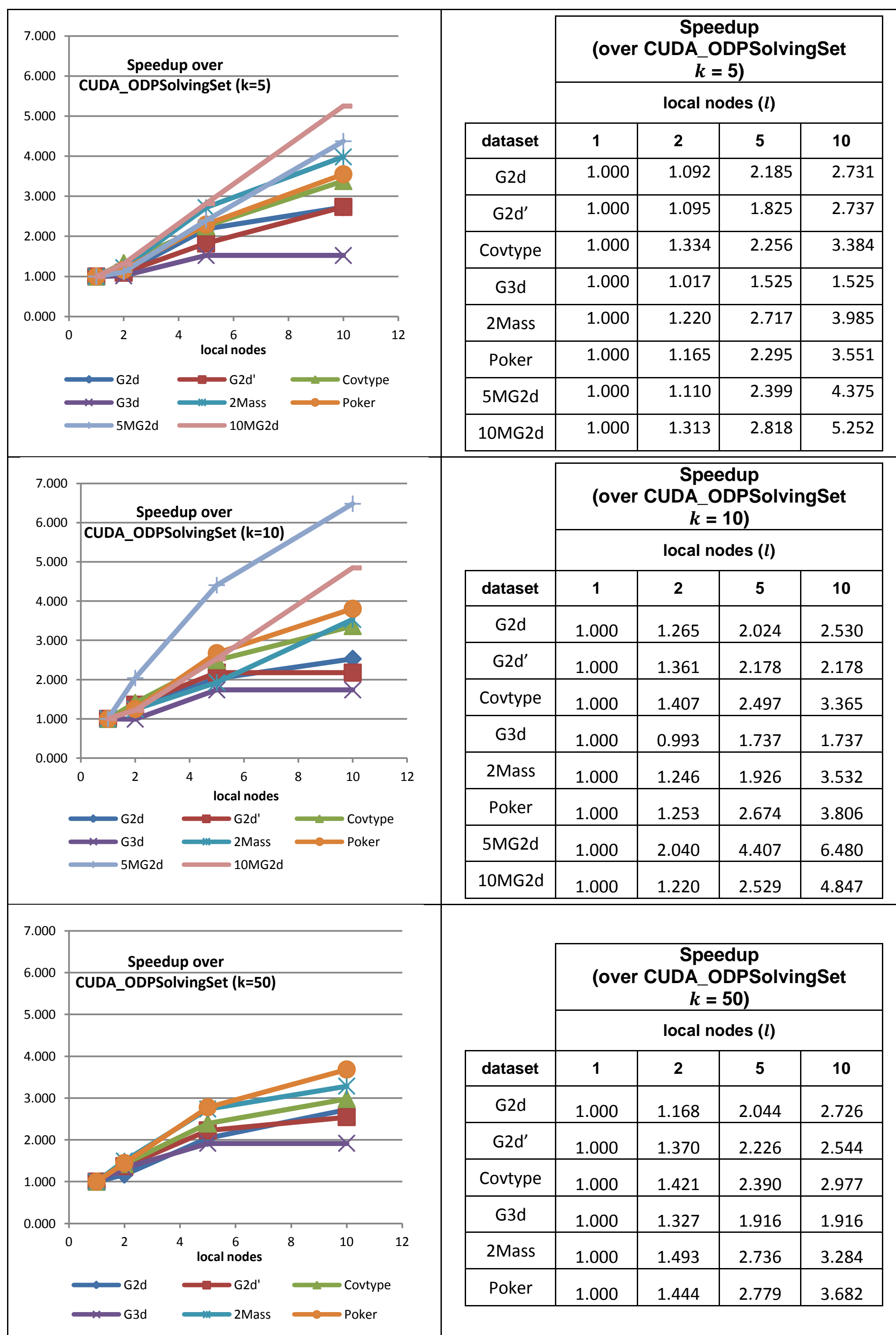
| | Speedup (over CPU ODPSolvingSet $k$ = 10) | | | |
|---|---|---|---|---|
| | local nodes ($l$) | | | |
| dataset | 1 | 2 | 5 | 10 |
| G2d | 63.717 | 80.602 | 128.964 | 161.204 |
| G2d’ | 55.990 | 76.217 | 121.947 | 121.947 |
| Covtype | 72.884 | 102.564 | 181.969 | 245.263 |
| G3d | 61.858 | 61.394 | 107.439 | 107.439 |
| 2Mass | 92.031 | 114.713 | 177.284 | 325.021 |
| Poker | 73.641 | 92.242 | 196.949 | 280.273 |
| 5MG2d | 48.570 | 99.089 | 214.032 | 314.753 |
| 10MG2d | 89.689 | 109.433 | 226.797 | 434.694 |

| | Speedup (over CPU ODPSolvingSet $k$ = 50) | | | |
|---|---|---|---|---|
| | local nodes ($l$) | | | |
| dataset | 1 | 2 | 5 | 10 |
| G2d | 63.362 | 74.020 | 129.535 | 172.713 |
| G2d’ | 62.798 | 86.021 | 139.784 | 159.754 |
| Covtype | 71.613 | 101.788 | 171.189 | 213.178 |
| G3d | 57.053 | 75.694 | 109.336 | 109.336 |
| 2Mass | 65.671 | 98.013 | 179.691 | 215.630 |
| Poker | 71.696 | 103.522 | 199.231 | 263.982 |

**Figura 7-7 Speedup di CUDA_ODPLazyDistributedSolvingSet su ODPSolvingSet ($m$=100, $n$=10)**

Con $l = 10$ è stato possibile ottenere uno speedup massimo intorno a 450, rispetto ad uno speedup di 100 nel caso della versione centralizzata. Possiamo notare che le prestazioni migliori si hanno in corrispondenza dei test sui dataset dotati del maggior numero di istanze (ovvero *Poker*, *2Mass*, *5MG2d*, *10MG2d*). Nei dataset di minori

dimensioni le performance sono nettamente inferiori. Per renderci meglio conto del problema, focalizziamo ora l'attenzione sullo speedup dell'algoritmo distribuito multi-GPU su quello centralizzato basato su GPU, ovvero il rapporto tra i tempi di esecuzione di CUDA_ODPSolvingSet e di CUDA_ODPLazyDistributedSolvingSet.




| | Speedup (over CUDA_ODPSolvingSet $k$ = 5) | | | |
|---|---|---|---|---|
| | local nodes ($l$) | | | |
| dataset | 1 | 2 | 5 | 10 |
| G2d | 1.000 | 1.092 | 2.185 | 2.731 |
| G2d' | 1.000 | 1.095 | 1.825 | 2.737 |
| Covtype | 1.000 | 1.334 | 2.256 | 3.384 |
| G3d | 1.000 | 1.017 | 1.525 | 1.525 |
| 2Mass | 1.000 | 1.220 | 2.717 | 3.985 |
| Poker | 1.000 | 1.165 | 2.295 | 3.551 |
| 5MG2d | 1.000 | 1.110 | 2.399 | 4.375 |
| 10MG2d | 1.000 | 1.313 | 2.818 | 5.252 |

| | Speedup (over CUDA_ODPSolvingSet $k$ = 10) | | | |
|---|---|---|---|---|
| | local nodes ($l$) | | | |
| dataset | 1 | 2 | 5 | 10 |
| G2d | 1.000 | 1.265 | 2.024 | 2.530 |
| G2d' | 1.000 | 1.361 | 2.178 | 2.178 |
| Covtype | 1.000 | 1.407 | 2.497 | 3.365 |
| G3d | 1.000 | 0.993 | 1.737 | 1.737 |
| 2Mass | 1.000 | 1.246 | 1.926 | 3.532 |
| Poker | 1.000 | 1.253 | 2.674 | 3.806 |
| 5MG2d | 1.000 | 2.040 | 4.407 | 6.480 |
| 10MG2d | 1.000 | 1.220 | 2.529 | 4.847 |

| | Speedup (over CUDA_ODPSolvingSet $k$ = 50) | | | |
|---|---|---|---|---|
| | local nodes ($l$) | | | |
| dataset | 1 | 2 | 5 | 10 |
| G2d | 1.000 | 1.168 | 2.044 | 2.726 |
| G2d' | 1.000 | 1.370 | 2.226 | 2.544 |
| Covtype | 1.000 | 1.421 | 2.390 | 2.977 |
| G3d | 1.000 | 1.327 | 1.916 | 1.916 |
| 2Mass | 1.000 | 1.493 | 2.736 | 3.284 |
| Poker | 1.000 | 1.444 | 2.779 | 3.682 |

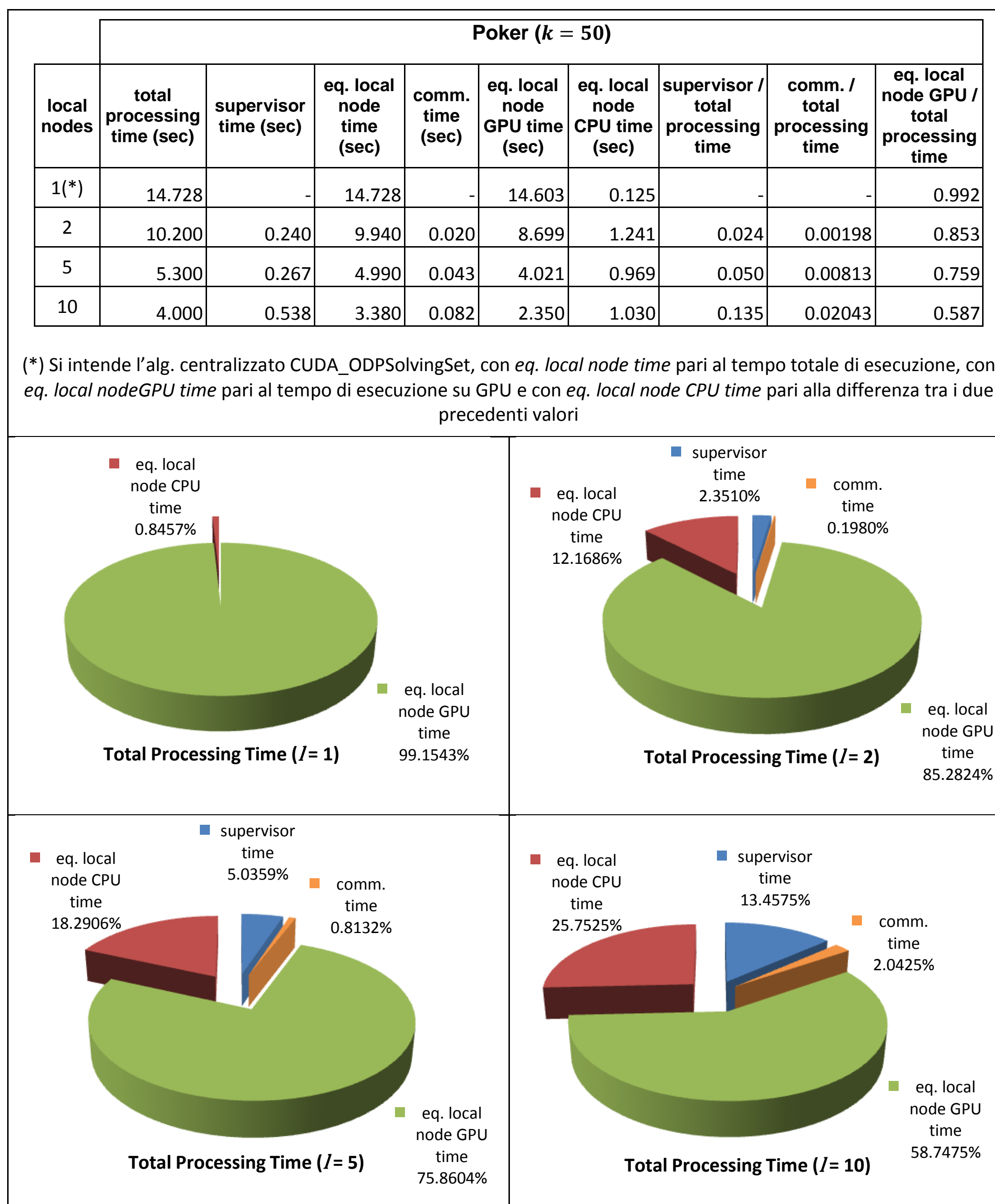
**Figura 7-8 Speedup di CUDA_ODPLazyDistributedSolvingSet su CUDA_ODPSolvingSet ($m$=100, $n$=10)**

Possiamo notare come in questo caso l'algoritmo riesca a scalare in maniera meno efficace all'aumentare del numero di nodi locali, rispetto alla versione distribuita basata esclusivamente su CPU. Nei dataset di dimensioni elevate si riesce a raggiungere uno speedup massimo di 4 - 6.5 con 10 nodi locali, quindi un valore pari a circa il 50% di quello ideale. Questo fenomeno però è particolarmente marcato nei dataset dotati di un numero minore di oggetti: con *G3d* addirittura si riese a raggiungere solamente uno speedup massimo di 1.9 (per $k = 50$) utilizzando 10 nodi locali.

Il motivo di questo gap prestazionale rispetto a ODPLazyDistributedSolvingSet appare chiaro se confrontiamo i tempi di esecuzione dei nodi locali con i tempi di comunicazione e di esecuzione del nodo supervisore. Grazie all'azione della GPU i tempi di esecuzione dei nodi locali tendono fortemente a decrescere, mentre i tempi di comunicazione e di esecuzione del nodo supervisore rimangono costanti. Ne consegue che queste due ultime tempistiche presentano un ruolo più incisivo sulle performance, a differenza di quanto avveniva per la soluzione CPU. Il tutto è aggravato dal fatto che non tutte le operazioni svolte dai nodi locali sono eseguite tramite GPU, ma è richiesto l'intervento della CPU per diverse funzionalità (come ad esempio per l'esecuzione delle procedure $NodeReq_{\gamma}()$) ed è inoltre necessario un maggior scambio di dati tra CPU e GPU. Per cercare di quantificare correttamente il tempo di esecuzione su GPU, introduciamo il concetto di *equivalent local node GPU time*. Sia ${T^*}_{i,\gamma}$ il tempo di esecuzione su GPU necessario durante l'$i$-esima iterazione dell'algoritmo per il nodo locale $N_{\gamma}$ e sia $t$ il totale numero di iterazioni effettuate. Definiamo con *equivalent local node GPU time* la somma:

$$\max\left({T^*}_{1,1}, {T^*}_{1,2}, \dots, {T^*}_{1,l}\right) + \max\left({T^*}_{2,1}, {T^*}_{2,2}, \dots, {T^*}_{2,l}\right) + \cdots + \max\left({T^*}_{t,1}, {T^*}_{t,2}, \dots, {T^*}_{t,l}\right).$$

Inoltre definiamo con *equivalent local node CPU time* la differenza tra *equivalent local node processing time* e *equivalent local node GPU time*, ovvero il tempo speso dai nodi locali per l'esecuzione su CPU e per lo scambio di dati tra CPU e GPU. Dai test è apparso in maniera evidente come l'*equivalent local node CPU time* costituisca una percentuale importante dell'*equivalent local node processing time*. In particolare consideriamo anche in questo caso le tempistiche dei test effettuati sul dataset *Poker* con $k = 50$. I risultati sono mostrati nella figura 7.9.



| | Poker ($k$ = 50) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| local nodes | total processing time (sec) | supervisor time (sec) | eq. local node time (sec) | comm. time (sec) | eq. local node GPU time (sec) | eq. local node CPU time (sec) | supervisor / total processing time | comm. / total processing time | eq. local node GPU / total processing time |
| 1(*) | 14.728 | - | 14.728 | - | 14.603 | 0.125 | - | - | 0.992 |
| 2 | 10.200 | 0.240 | 9.940 | 0.020 | 8.699 | 1.241 | 0.024 | 0.00198 | 0.853 |
| 5 | 5.300 | 0.267 | 4.990 | 0.043 | 4.021 | 0.969 | 0.050 | 0.00813 | 0.759 |
| 10 | 4.000 | 0.538 | 3.380 | 0.082 | 2.350 | 1.030 | 0.135 | 0.02043 | 0.587 |

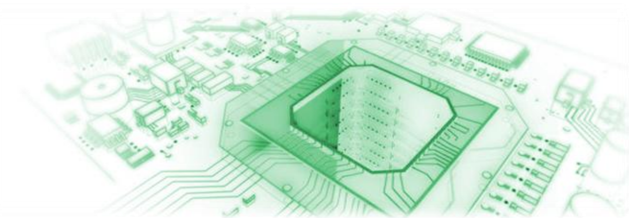
(*) Si intende l'alg. centralizzato CUDA_ODPSolvingSet, con *eq. local node time* pari al tempo totale di esecuzione, con *eq. local nodeGPU time* pari al tempo di esecuzione su GPU e con *eq. local node CPU time* pari alla differenza tra i due precedenti valori



**Figura 7-9 Analisi dei tempi di esecuzione di CUDA_ODPLazyDistributedSolvingSet per il dataset *Poker* ($k$=50, $m$=100, $n$=10)**

Possiamo notare come in questo caso il tempo di esecuzione sul nodo supervisore e il tempo di comunicazione rappresentino una fetta non indifferente del tempo totale di esecuzione, pari a circa il 15% con 10 nodi locali. Inoltre un altro ruolo importante è giocato dall' *equivalent local node CPU time*. Se consideriamo la soluzione concentrata basata su GPU, possiamo notare come il tempo di esecuzione della CPU e dei trasferimenti di dati tra CPU e GPU costituisca una percentuale minima del tempo totale di esecuzione (inferiore all'1%). Viceversa, nella versione distribuita possiamo notare come questa percentuale sia più elevata e tenda a crescere all'aumentare del numero di nodi, fino a raggiungere un 25% nel caso di utilizzo di 10 nodi locali.

Questi fattori diventano degli importanti elementi limitativi del massimo speedup raggiungibile, utilizzando un numero elevato di nodi locali. Per ottenere delle buone performance è necessario utilizzare dei dataset dotati di un numero molto elevato di oggetti, in quanto al crescere della dimensione del dataset i tempi di esecuzione su GPU tendono ad aumentare in maniera più rapida rispetto agli altri. Questo in quanto il tempo di esecuzione della procedura $CUDA_NodeComp_{\gamma}()$ (eseguita dalla GPU) presenta una dipendenza quadratica dalla dimensione del dataset $d$, a differenza dei tempi di $NodeReq_{\gamma}()$, delle operazioni eseguite dal nodo supervisore e di comunicazione (che dipendono in maniera quasi lineare da $d$). Ricordiamo che l'esecuzione di $CUDA_NodeComp_{\gamma}()$ è l'unica fase dell'algoritmo distribuito che risente in maniera positiva della presenza di un numero $l$ elevato di nodi locali, i tempi delle altre fasi invece tendono a crescere all'aumentare di $l$. Per questo motivo più dominante è la fase di esecuzione della procedura $CUDA_NodeComp_{\gamma}()$, maggiori sono le performance dell'algoritmo, se confrontate con la soluzione CPU.



# Capitolo 8: Conclusioni e sviluppi futuri

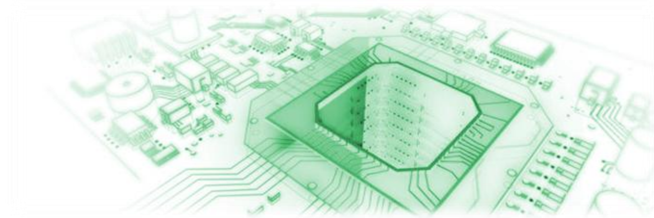
In questo lavoro abbiamo mostrato un approccio innovativo al problema dell'individuazione delle anomalie, in dataset di grandi dimensioni. Tramite l'utilizzo delle GPU è possibile ridurre i tempi di esecuzione degli algoritmi e accelerare la velocità di risoluzione del problema di ben due ordini di grandezza. L'elevata potenza di calcolo offerta dai chip grafici permette di ottenere delle performance estremamente elevate, per problemi facilmente scomponibili in un'esecuzione parallela. Le prestazioni di una singola GPU possono essere paragonabili a quelle di un supercomputer di medio taglio, che da un punto di vista economico presenta però un costo tremendamente più elevato. Queste caratteristiche economiche rendono l'utilizzo delle tecniche di GPU computing particolarmente adatte in ambito aziendale. Inoltre oggigiorno sono stati creati diversi supercomuter dotati di un numero elevatissimo di GPU, per permettere di combinare la potenza di calcolo di più processori grafici. Ne è un esempio il superpc *IBM PLX* locato presso il *CINECA*, che è stato utilizzato per effettuare i test del nostro lavoro.

Abbiamo rivisitato l'approccio proposto da *Angiulli et al.* [3] [4], uno dei più efficienti metodi di outlier detection di tipo *distance-based*, in un'ottica di GPU computing, sia nel caso di un ambiente centralizzato (con una sola GPU), che nel caso di un ambiente distribuito (con un'architettura multi-GPU). Dai risultati sperimentali è emerso come sia stato possibile ottenere, tramite una soluzione GPGPU (con una GPU *NVIDIA Tesla M2070*), dei tempi di esecuzione 80-100 volte inferiori rispetto a quelli di una tradizionale soluzione basata su CPU. Inoltre, utilizzando la versione distribuita dell'algoritmo, è stato possibile raggiungere uno speedup pari a 450, combinando la potenza di calcolo di 10 GPU, rispetto all'algoritmo centralizzato eseguito su una sola CPU.

Abbiamo infine mostrato come, nella versione distribuita multi-GPU proposta, l'ottenimento di alte prestazioni sia però vincolato alla disponibilità di dataset distribuiti dotati di un numero di oggetti molto elevato. Per dataset di dimensioni non sufficientemente elevate i tempi di esecuzione del nodo supervisore, i tempi di comunicazione tra i nodi e i singoli tempi di trasferimento di dati tra CPU e GPU all'interno dei nodi locali diventano significativi, se confrontati con il reale tempo di computazione su GPU. Per risolvere questo problema è opportuno agire su due fronti.

Occorre un primo intervento sulle tecnologie di comunicazione, per rendere la trasmissione di dati tra i diversi nodi più efficiente. Nel caso specifico di esecuzione su un cluster, dove il sistema di interconnessione tra i nodi è particolarmente veloce, l'utilizzo di socket *TCP/IP* può non essere la scelta migliore. Un'alternativa è quella di utilizzare il protocollo *MPI* (*Message Passing Interface*), standard *de facto* di comunicazione tra i diversi nodi di un cluster in sistemi a memoria distribuita, ottimizzato per sfruttare al meglio la specifica architettura sottostante.
Un secondo intervento riguarda invece una modifica strutturale dell'algoritmo stesso. Per eliminare i tempi di esecuzione del supervisore è opportuno prevedere un algoritmo che non necessiti di un nodo supervisore, in cui l'esecuzione avvenga in maniera esclusiva da parte dei singoli nodi locali, prevedendo comunicazioni dirette di tipo *peer-to-peer*. Inoltre abbiamo mostrato come anche l'azione locale eseguita dalla CPU di ogni nodo locale (che possiamo considerare come un *overhead*) diventi significativa. E' quindi opportuno cercare di abbatterla, delegando alla GPU la maggior parte possibile del lavoro, limitando i compiti della CPU ai soli scambi di dati tra la GPU locale e gli altri nodi. In quest'ottica ad ogni iterazione dell'algoritmo, per la formazione del nuovo insieme di candidati, ogni nodo locale dovrà ricevere tutti candidati locali provenienti dai diversi nodi, che potranno essere inviati in multicast sulla rete. Similmente, per il calcolo del peso di ogni candidato e l'aggiornamento dell'insieme corrente dei top-$n$ outlier, ogni nodo invierà a tutti gli altri gli heap $LNNC_{\gamma}$ ed ogni singolo nodo procederà in maniera autonoma al calcolo dei pesi e alla ricerca dei punti di maggior peso. Anche in questo caso è possibile utilizzare delle ottimizzazioni simili a quelle proposte nella variante *lazy* di ODPDistributedSolvingSet, per limitare la quantità di distanze trasferite.

Molte delle tecniche di GPU computing proposte in questo lavoro possono essere inoltre utili anche in contesti diversi. La tecnica della riduzione parallela basata sugli heap, *heap complete reduction* e relative varianti, si è rivelata essere particolarmente efficace anche per risolvere il tradizionale problema *k-NN*. Può essere utilizzata, ad esempio, per accelerare l'esecuzione di algoritmi di clustering basati sulla località, come *DBSCAN,* o più semplicemente per rendere più veloce la risposta a query *k-NN* nei database. Infine può essere combinata con strutture efficaci di indicizzazione, come gli *R-tree*, per fornire prestazioni ancora più elevate.

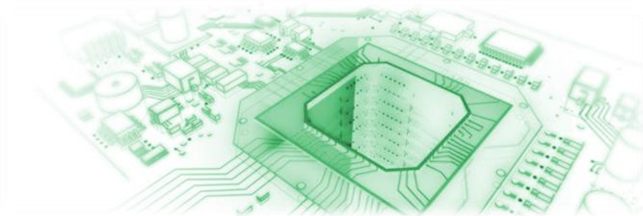


# Ringraziamenti

I miei più sentiti ringraziamenti li porgo al *Prof. Ing. Claudio Sartori*, all'*Ing. Stefano Basta* e al *Prof. Ing. Stefano Lodi* per l'opportunità offertami di svolgere questa tesi e per il loro costante aiuto durante l'intero svolgimento del lavoro.

Un particolare ringraziamento va alla mia famiglia, che mi è sempre stata vicina in ogni occasione, anche nei momenti più difficili.

Un ringraziamento speciale lo porgo alla mia ragazza, che è riuscita a sopportarmi durante tutti questi anni di studi (pur cercando di uccidermi più di una volta…). Ringrazio infine anche tutta la sua famiglia, per il loro costante supporto.



# Bibliografia

[1] V. Chandola, A. Banerjee e V. Kumar, «Anomaly detection: A Survey,» *ACM Computing Surveys,* vol. 41, n. 3, 2009.

[2] F. Angiulli e C. Pizzuti, «Outlier Mining in Large High-Dimensional Data Sets,» *IEEE Trans. Knowledge and Data Eng.,* vol. 2, n. 17, pp. 203-215, 2005.

[3] F. Angiulli, S. Basta e C. Pizzuti, «Distance-Based Detection and Prediction of Outliers,» *IEEE Transactions on Knowledge and Data Engineering,* vol. 18, n. 2, 2006.

[4] F. Angiulli, S. Basta, S. Lodi e C. Sartori, «Distributed Strategies for Mining Outliers in Large Data Sets,» *IEEE Transactions on Knowledge and Data Engineering,* vol. V, 2012.

[5] D. M. Hawkins, Identification of Outliers, London: Chapman and Hall, 1980.

[6] E. Knorr e R. Ng, «Algorithms for Mining Distance-Based Outliers in Large Datasets,» *Proc. of the VLDB Conference,* pp. 392-403, 1998.

[7] S. Ramaswamy, R. Rastogi e K. Shim, «Efficient Algorithms for Mining Outliers from Large Data Sets,» in *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, Dallas, TX, 2000.

[8] S. Bay e M. Schwabacher, «Mining Distance-Based Outliers in Near Linear Time with Randomization and a Simple Pruning Rule,» in *Proc. Int'l Conf. Knowledge Discovery and Data Mining (KDD)*, 2003.

[9] M. Breuning, H. Kriegel, R. Ng e J. Sander, «LOF: identifyng density-based local outliers,» in *In Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, Dallas, Texas, 2000.

[10] «IBM PLX,» [Online]. Available: http://www.hpc.cineca.it/hardware/ibm-plx.

[11] A. Asuncion e D. Newman, UCI machine learning repository, 2007.

[12] «NASA/IPAC Infrared Science Archive (IRSA),» [Online]. Available: http://irsa.ipac.caltech.edu/.

[13] J. D. Owens, M. Houston, D. Luebke e S. Green, «GPU Computing,» *Proceedings of the IEEE,* pp. Vol.96, No. 5, 2008.

[14] P. N. Glaskowksy, «NVIDIA'S Fermi: The First Complete GPU Computing Architecture».

[15] NVIDIA, NVIDIA CUDA C Programming Guide (v. 4.2), 2012.

[16] NVIDIA, CUDA C Best Practices Guide (v. 4.1), 2012.

[17] V. Garcia, E. Debreuve e M. Barlaud, «Fast k nearest neighbor search using GPU,» in *Computer Vision and Pattern Recognition Workshops, CVPRW '08*, 2008.

[18] K. Quansheng e Z. Lei, «A pratical GPU based kNN algorithm,» in *Proc. of the Second Symposium International Computer Science and Computational Technology (ISCSCT '09)*, 2009.

[19] S. Liang, C. Wang, Y. Liu e J. Jian, «CUKNN: A parallel implementation of K-nearest neighbor on CUDA-enabled GPU,» in *Information, Computing and Telecommunication, 2009. YC-ICT '09. IEEE Youth Conference on*, 2009.

[20] R. Barrientos, J. Gòmez, C. Tenllado e M. Prieto, «Heap Based k-Nearest Neighbor Search on GPUs,» in *XXI Jornadas de Paralelismo*, Valencia, Spagna, 2010.

[21] K. Kato e T. Hosino, «Solving k-Nearest Neighbor Problem on Multiple Graphics Processors,» in *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGrid 2010*, Melbourne, Victoria, Australia, 2010.

[22] A. S. Arefin, C. Riveros, R. Berretta e P. Moscato, «GPU-FS-kNN: A Software Tool for Fast and Scalable kNN Computation Using GPUs,» *PLoS ONE,* vol. 7, n. 8, 2012.

[23] M. Harris, «Optimizing Parallel Reduction in CUDA,» NVIDIA.

[24] «JCuda.org,» [Online]. Available: http://www.jcuda.de/.

[25] M. Thomadakis, The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms, Texas A&M University, 2011.

[26] G. Ruetsch e P. Micikevicius, «Optimizing Matrix Transpose in CUDA,» NVIDIA, 2009.

[27] B. Park e H. Kargupta, «Distributed Data Mining: Algorithms, Systems, and Applications,» 2002.