Gpu Computing

GPU-DBSCAN

Luca Milanesi 55053A
luca.milanesi1@studenti.unimi.it

A.A. 2025-2026

Indice

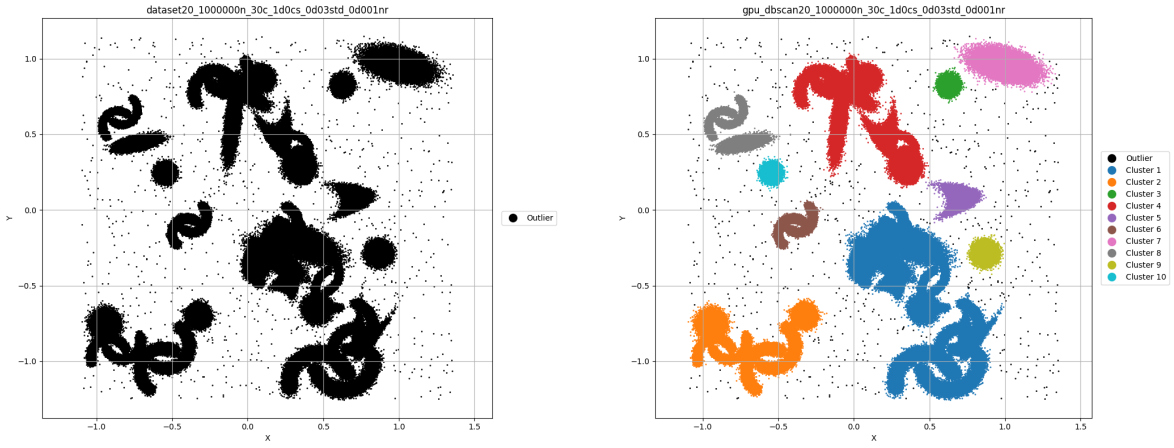
1 Descrizione

DBSCAN (*Density-Based Spatial Clustering of Applications with Noise*) è un algoritmo di clustering basato sulla densità che raggruppa punti spazialmente vicini e identifica gli outlier come rumore. L'algoritmo richiede due parametri:

- ϵ, che definisce il raggio di vicinanza tra i punti.
- *minPts*, il numero minimo di punti necessari per formare un cluster denso.

Un punto p è definito *core point* se nel suo ϵ-vicinato sono presenti almeno *minPts* punti. I punti raggiungibili da un core point attraverso una catena di core point appartengono allo stesso cluster, mentre i restanti vengono classificati come rumore.

Per semplicità, da ora in avanti indicheremo con n la cardinalità del dataset. Inoltre, un generico punto p del dataset viene incluso nel conteggio dei punti presenti nel suo ϵ-vicinato.



(a) Dataset originale con 1 000 000 di punti e presenza di rumore.

(b) Risultato del clustering DBSCAN con identificazione di 10 cluster.

Figura 1: Visualizzazione del processo di clustering su un dataset sintetico complesso: a sinistra la distribuzione spaziale iniziale, a destra i cluster densi identificati dall'algoritmo con $\epsilon = 0.03$ e $minPts = 8$.

2 Implementazione parallela su GPU

L'algoritmo DBSCAN presenta un elevato grado di parallelismo, poiché molte delle operazioni fondamentali possono essere eseguite indipendentemente sui singoli punti del

dataset. L'implementazione descritta in questo lavoro è riferita al caso bidimensionale (2D), nel quale i punti sono definiti da coordinate spaziali (x, y). L'approccio adottato è composto da tre fasi principali, ciascuna delle quali è parallelizzata a livello di thread.

1. **Spatial binning**: ogni punto viene associato a una cella di una griglia uniforme che partiziona lo spazio dei dati. La griglia è composta da celle quadrate di lato ϵ, e la cella di appartenenza di un punto è determinata a partire dalle sue coordinate spaziali. Questa strategia, illustrata nella sezione 2.2, consente l'organizzazione dei punti in una struttura dati che ottimizza la scoperta dei potenziali vicini.

2. **Identificazione dei core point**: per ciascun punto viene calcolato il numero di punti presenti nel suo ϵ-vicinato. Grazie alla suddivisione dello spazio in celle, la ricerca dei vicini è limitata ai punti contenuti nella cella di appartenenza e nelle celle adiacenti. Un punto viene classificato come *core point* se il numero di vicini individuati è maggiore del parametro *minPts*.

3. **Espansione dei cluster**: una volta identificati i core point, i cluster vengono espansi tramite BFS (*Breadth-First Search*). A partire da un core point non ancora assegnato, l'espansione del cluster procede esplorando iterativamente tutti i punti raggiungibili da esso. Ogni livello della BFS può essere elaborato in parallelo processando simultaneamente tutti i punti appartenenti alla frontiera corrente.

L'implementazione descritta è attualmente limitata al caso bidimensionale (2D). Tuttavia, l'approccio è concettualmente estendibile a uno spazio di dimensione m, purché m rimanga piccolo. La suddivisione dello spazio in una griglia uniforme (*grid-based indexing*) consente infatti di ridurre drasticamente lo spazio di ricerca dei vicini, evitando la scansione completa del dataset per ogni punto con un numero di operazioni pari a $O(n^2)$, impraticabile per dataset di grandi dimensioni.

All'aumentare della dimensionalità, però, il numero di celle da ispezionare cresce esponenzialmente con m. Infatti, in uno spazio m-dimensionale è necessario controllare 3^m celle adiacenti per ciascun punto: ad esempio 9 celle in 2D, 27 in 3D, 81 in 4D, e così via.

2.1 Parametri di input e output dell'algoritmo

Il metodo DBSCAN prende in input i seguenti parametri:

- x, y: array di coordinate dei punti nel piano bidimensionale;
- n: numero totale di punti;

- ϵ: distanza massima per considerare due punti vicini;
- *minPts*: numero minimo di vicini per classificare un punto come core;

Mentre restituisce come output:

- *cluster*: array di dimensione n contenente l'ID del cluster di ciascun punto. Punti non assegnati a nessun cluster (outlier) hanno valore `NO_CLUSTER_LABEL`;
- *clusterCount*: numero totale di cluster trovati;

2.2 Grid-based indexing

Questa tecnica consiste nel partizionare lo spazio bidimensionale in una griglia uniforme con celle di lato ϵ. Dato un punto p in posizione (x, y), i suoi potenziali vicini entro distanza ϵ possono trovarsi al massimo nelle 9 celle adiacenti (3×3) centrate sulla cella contenente p. Questo perché la massima distanza tra due punti in celle non adiacenti supera sempre ϵ.

Formalmente, per un punto con coordinate (x, y), la cella di appartenenza è calcolata come:

$$c_x = \left\lfloor \frac{x - x_{\min}}{\epsilon} \right\rfloor, \quad c_y = \left\lfloor \frac{y - y_{\min}}{\epsilon} \right\rfloor$$

Per ottimizzare la computazione, al posto di usare l'operazione di divisione per ϵ, si utilizzerà la moltiplicazione per $\frac{1}{\epsilon}$, essendo più efficiente in CUDA.

La cella viene poi linearizzata in un ID univoco dato da:

$$cellId = c_y \cdot w + c_x$$

dove w è la larghezza della griglia. Questa struttura dati riduce drasticamente lo spazio di ricerca: invece di confrontare ogni punto con tutti gli altri $n - 1$ punti, si confronta solo con i punti nelle 9 celle adiacenti, riducendo la complessità media a $O(nk)$, dove k è il numero medio di punti nelle 9 celle adiacenti per ogni punto.

Ribadendo ciò che è già stato accennato precedentemente, lo stesso approccio è direttamente estendibile a uno spazio m-dimensionale: la griglia diventa ipercubica con celle di lato ϵ, ogni punto deve confrontarsi con al massimo 3^m celle adiacenti, e l'ID della cella si ottiene tramite una linearizzazione multidimensionale delle coordinate. La complessità media resta $O(nk)$, con k dipendente dalla densità locale e da m, ma indipendente da n.

2.3 Ottimizzazioni generali dei kernel

Tutti i kernel CUDA implementati adottano alcune ottimizzazioni generali comuni, volte a massimizzare l'efficienza e la scalabilità:

- **Dimensionamento automatico di grid e block**: ogni kernel viene lanciato tramite la funzione `launchKernel`, che calcola automaticamente il numero ottimale di blocchi e thread per massimizzare l'occupancy della GPU, basandosi sulla dimensione del dataset, sulla eventuale **memoria condivisa** richiesta e sulle caratteristiche del kernel;
- **Pattern di striding**: nel caso in cui il numero totale di thread disponibili sia inferiore al numero di punti da elaborare, ogni thread processa più punti separati da uno *stride* pari al numero totale di thread. Questo consente di scalare l'elaborazione a qualsiasi dimensione del dataset senza dover assegnare un thread per ogni punto;
- **Uso della memoria costante**: costanti globali lette frequentemente da tutti i thread come n, ϵ, $\frac{1}{\epsilon}$, ϵ^2, $minPts$, dimensioni della griglia e valori min/max del dataset, sono memorizzati in **memoria costante** per garantire un accesso rapido e broadcast efficiente a tutti i thread;

Questo approccio garantisce che l'algoritmo sia scalabile a dataset di grandi dimensioni e sfrutti al massimo le capacità della GPU.

3 Pipeline dell'algoritmo

L'algoritmo si articola in sei fasi principali, ciascuna implementata come kernel CUDA separato, che saranno eseguite sequenzialmente:

1. **Calcolo dei limiti del dataset**: si determinano $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$ del dataset attraverso una riduzione parallela. Questi valori servono a definire le dimensioni della griglia delle celle;
2. **Binning dei punti nelle relative celle**: ogni punto calcola indipendentemente le coordinate della propria cella e viene associato al corrispondente $cellId$;
3. **Ordinamento per cella**: i punti vengono riordinati in base al loro $cellId$. Questo passaggio raggruppa in memoria tutti i punti appartenenti alla stessa cella, così da avere un accesso sequenziale per ogni cella nelle fasi successive;

4. **Calcolo degli estremi delle celle**: si identificano gli indici di inizio (*cellStart*) e fine (*cellEnd*) di ogni cella nell'array ordinato della mappa degli indici. Questi indici permettono di iterare rapidamente su tutti i punti di una cella senza ricerche binarie molto costose.;

5. **Identificazione dei core points**: per ogni punto si contano i vicini nelle 9 celle adiacenti. Se un punto ha almeno *minPts* vicini, viene marcato come core point;

6. **Espansione dei cluster tramite BFS**: si itera su tutti i core points non ancora assegnati. Per ognuno si crea un nuovo cluster e lo si espande tramite una BFS parallela: ad ogni livello, tutti i nodi della frontiera corrente vengono processati concorrentemente, aggiungendo i loro vicini core alla frontiera successiva, fino alla completa espansione del cluster;

3.1 Calcolo dei limiti del dataset

Il calcolo dei limiti del dataset $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$ è necessario per determinare le dimensioni della griglia di celle per il *grid-based indexing*. Questo passaggio è implementato tramite il kernel `computeBounds`, che utilizza una strategia di riduzione gerarchica a più livelli:

1. **Riduzione intra-thread**: Ogni thread elaborando un sottoinsieme di punti con pattern di *striding*, calcola i propri limiti locali;

2. **Riduzione intra-warp**: utilizzando le istruzioni `__shfl_down_sync`, ogni warp riduce i valori dei 32 thread che lo compongono, trovando il valore minimo e massimo locale di x e y per ogni warp. Infine, il primo thread di ogni warp scrive i risultati parziali trovati in **memoria condivisa**;

3. **Riduzione inter-warp (blocco)**: il primo warp di ogni blocco esegue una seconda riduzione sui risultati parziali di tutti i warp del blocco trovati precedentemente;

4. **Riduzione multi-kernel su GPU** Poiché ogni blocco scrive i propri risultati in **memoria globale**, il kernel alla fine darà come risultato degli array di limiti locali con cardinalità pari al numero di blocchi del kernel appena eseguito. Dunque `computeBounds` può essere lanciato più volte. Ad ogni iterazione:

    - Si riduce il numero di elementi da processare, che sarà pari al numero di blocchi del kernel eseguito al passo precedente;

- L'input da processare sarà l'output del passo precedente;
- Si riesegue `computeBounds` fino a quando il numero di elementi da ridurre è inferiore a una soglia `MIN_BOUND_INPUT_SIZE` o fino al raggiungimento di un numero massimo di iterazioni `MAX_BOUND_ITERATION`;

5. **Riduzione finale su CPU**: quando il numero di valori parziali diventa sufficientemente piccolo, viene eseguita una riduzione finale in memoria host per ottenere i valori finali di $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$;

Questi valori vengono poi utilizzati per calcolare le dimensioni della griglia:

$$gridWidth = \left\lceil \frac{x_{\max} - x_{\min}}{\epsilon} + 1 \right\rceil, \quad gridHeight = \left\lceil \frac{y_{\max} - y_{\min}}{\epsilon} + 1 \right\rceil$$

La **memoria condivisa** è allocata dinamicamente con:

$$4 \cdot blockSize/warpSize \cdot sizeof(float)$$

byte per contenere i quattro limiti locali (min/max per x e y) di ogni warp.

3.2 Binning dei punti nelle relative celle

Una volta determinati i limiti del dataset, si costruisce una griglia uniforme con celle di lato ϵ. Il kernel `computeBinning` assegna ogni punto alla cella di appartenenza calcolando le coordinate di cella c_x e c_y e linearizzandole in un singolo ID secondo la logica vista nella sezione 2.2.

L'output del kernel consiste in due array di lunghezza n:

- $cellId[i]$: ID linearizzato della cella del punto i-esimo;
- $cellPoints[i]$: indice originale del punto inizializzato con:

  $$\forall i \in \{1, \ldots, n-1\} : cellPoints[i] = i$$

  ed è necessario per mantenere il riferimento ai punti originali nella successiva fase di ordinamento;

3.3 Ordinamento per cella

L'ordinamento viene effettuato utilizzando la funzione `thrust::sort_by_key` della libreria **Thrust** di NVIDIA, che ordina $cellId$ e permuta coerentemente l'array satellite $cellPoints$. Al termine dell'operazione, gli indici dei punti appartenenti alla stessa cella risultano memorizzati in un intervallo contiguo nell'array $cellPoints$.

Dopo l'ordinamento, sarà quindi possibile individuare per ciascuna cella l'intervallo di indici dei punti associati ad essa, consentendo una scansione efficiente dei candidati vicini durante il calcolo dei core points e l'espansione dei cluster illustrati rispettivamente nelle sezioni 3.5 e 3.6.

3.4 Calcolo degli estremi delle celle

Il calcolo degli indici di inizio ($cellStart$) e fine ($cellEnd$) di ciascuna cella all'interno dell'array ordinato dei punti viene effettuato dal kernel `buildBinExtreme`. Il kernel prende in input l'array $cellId$ precedentemente ordinato, in modo tale che i punti appartenenti alla stessa cella risultino contigui in memoria.

Per ogni punto di indice globale i, il kernel confronta il valore $cellId[i]$ con quelli dei punti adiacenti per individuare i confini delle celle:

$$\forall i \in \{0, \ldots, n-1\}, c = cellId[i] : \begin{cases} cellStart[c] = i & i = 0 \ \vee \ c \neq cellId[i-1] \\ cellEnd[c] = i+1 & i = n-1 \ \vee \ c \neq cellId[i+1] \end{cases}$$

Per ridurre il numero di accessi alla memoria globale, il kernel utilizza le **warp shuffle operations**, che consentono di accedere ai valori dei thread adiacenti all'interno dello stesso warp, evitando accessi inutili alla memoria globale:

- `__shfl_up_sync`: consente a un thread i di ottenere il $cellId[i-1]$ del thread precedente nel warp;
- `__shfl_down_sync`: consente a un thread i di ottenere il $cellId[i+1]$ del thread successivo nel warp;

Solo i thread ai margini del warp (lane 0 e lane 31) devono accedere direttamente alla memoria globale per recuperare il valore del punto precedente o successivo. Questa strategia genera una singola scrittura per $cellStart$, una per $cellEnd$ ed evita completamente l'uso di operazioni atomiche.

3.5 Identificazione dei core points

L'identificazione dei core points viene effettuata dal kernel `computeIsCoreArr`, il cui obiettivo è determinare, per ciascun punto del dataset, se il numero di vicini entro distanza ϵ è maggiore o uguale al parametro *minPts*.

Per ogni punto i, il kernel calcola innanzitutto le coordinate e l'ID lineare della cella di appartenenza e successivamente esamina tutte le celle adiacenti nella griglia uniforme, ovvero le $3 \times 3 = 9$ celle centrate sulla cella del punto. Grazie al *grid-based indexing*, la ricerca dei vicini è quindi limitata a un insieme ristretto di candidati, evitando confronti inutili con punti distanti.

Per ogni cella adiacente, il kernel itera sugli indici dei punti compresi nell'intervallo $[cellStart, cellEnd)$, precedentemente calcolato. Per ogni punto candidato j, viene verificato se la distanza euclidea tra i e j è inferiore a ϵ: in tal caso, il contatore dei vicini viene incrementato. Il punto i viene infine marcato come core point se il numero totale di vicini soddisfa il vincolo imposto da *minPts*.

Il kernel adotta le seguenti ottimizzazioni:

- **Controllo dei limiti senza branch**: per verificare se una cella adiacente rientra nei limiti della griglia, viene utilizzata una maschera booleana invece di istruzioni condizionali. Celle fuori dai limiti producono intervalli vuoti e non generano iterazioni, riducendo la divergenza a livello di warp;
- **Confronto delle distanze al quadrato**: la funzione `isEpsNeighbor` confronta la distanza euclidea al quadrato con ϵ^2, evitando l'uso della radice quadrata e riducendo il costo computazionale del test di vicinanza;
- **Tipizzazione efficiente di** *isCoreArr*: l'array *isCoreArr* non utilizza il tipo `bool`, ma `uint8_t` (8 bit) per rappresentare il flag di core/non-core. Questa scelta riduce l'overhead di memoria e permette accessi più efficienti in GPU;

Il risultato del kernel è un array booleano *isCoreArr* in cui $isCoreArr[i] = 1$ se e solo se il punto i è un core point. Questo pre-calcolo consente alle fasi successive dell'algoritmo di accedere in tempo costante all'informazione di core/non-core, evitando ricalcoli ridondanti durante l'espansione dei cluster.

3.6 Espansione dei cluster tramite BFS

L'ultima fase dell'algoritmo DBSCAN consiste nell'espansione dei cluster a partire dai core points individuati in precedenza. Prima di avviare tale fase, l'array `cluster` viene inizializzato assegnando a ogni punto il valore `NO_CLUSTER_LABEL`, che identifica i punti non ancora associati ad alcun cluster.

L'espansione dei cluster è strutturata come un ciclo iterativo sul lato host, composto da due operazioni parallele sul device eseguite sequenzialmente:

1. ricerca del prossimo core point non ancora assegnato ad alcun cluster;
2. espansione del cluster associato al core point trovato tramite una BFS (*Breadth-First Search*);

Il ciclo viene ripetuto fino a quando non esistono più core points non clusterizzati. Ogni iterazione individua un nuovo core point, crea un nuovo cluster e ne espande l'insieme di punti densamente raggiungibili fino a saturazione. Al termine del processo, tutti i punti che risultano ancora etichettati con `NO_CLUSTER_LABEL` vengono automaticamente classificati come rumore.

3.6.1 Ricerca del prossimo core point

La selezione del prossimo core point da cui avviare l'espansione di un cluster è affidata al kernel `findNextCore`. L'obiettivo è individuare l'indice minimo di un punto che soddisfi contemporaneamente le seguenti condizioni:

- Sia un core point;
- Non sia ancora stato assegnato ad alcun cluster, ovvero:

$$cluster[i] = \texttt{NO\_CLUSTER\_LABEL}$$

- Abbia indice maggiore o uguale a $startIdx$, cioè un valore che indica da quale indice nell'array dei punti iniziare la ricerca per di evitare di riesaminare punti già controllati nelle iterazioni precedenti;

Il kernel realizza una riduzione parallela gerarchica su più livelli:

1. **Riduzione intra-thread**: ogni thread scansiona il proprio sottoinsieme di punti tramite *striding* e memorizza il primo core point valido trovato come candidato locale;
2. **Riduzione intra-warp**: i thread all'interno di ciascun warp eseguono una riduzione tra i loro minimi locali trovati al passo precedente utilizzando l'operazione di `__shfl_down_sync`, che consentono lo scambio diretto dei valori tra registri dei thread minimizzando la latenza;

3. **Riduzione inter-warp (blocco)**: i minimi calcolati da ciascun warp vengono scritti in **memoria condivisa**. Il primo warp di ogni blocco esegue una riduzione finale su questi valori parziali, ottenendo il minimo a livello di blocco;

4. **Riduzione globale**: i minimi di ciascun blocco vengono combinati in memoria globale dal primo thread del blocco tramite `atomicMin`, garantendo correttezza anche in presenza di più blocchi concorrenti;

La **memoria condivisa** è allocata dinamicamente con:

$$blockSize/warpSize \cdot sizeof(float)$$

byte per contenere i minimi locali trovati da ogni warp di uno stesso blocco.

3.6.2 Espansione BFS del cluster

L'espansione vera e propria del cluster è implementata dal kernel `bfsExpand`, che realizza una BFS parallela. L'esplorazione avviene a livelli, partendo dal core point radice e propagandosi ai punti densamente raggiungibili.

Al livello iniziale, il primo thread del kernel assegna il punto radice al cluster corrente e lo inserisce nella frontiera iniziale. Successivamente, a ogni iterazione:

1. Tutti i thread processano in parallelo i punti da esplorare appartenenti alla frontiera corrente;

2. Per ciascun punto, vengono analizzati i candidati vicini contenuti nelle 9 celle adiacenti della griglia;

3. I punti entro distanza ϵ vengono assegnati al cluster tramite `atomicCAS`, che effettua l'assegnazione in maniera atomica solo se il punto non era già stato assegnato, evitando così race condition e duplicati nella frontiera;

4. Solo i vicini che sono core point e non ancora assegnati vengono inseriti nella frontiera successiva tramite `atomicAdd`, che riserva in modo atomico una posizione nella nuova frontiera, permettendo scritture concorrenti senza conflitti;

Le ottimizzazioni implementate nel kernel coincidono con quelle descritte nella sezione 3.5 per l'identificazione dei core points, dato che hanno una logica simile. Tra le quali ci sono l'applicazione di maschere per il controllo dei limiti e l'eliminazione dell'uso dell'operazione `sqrt`.

Al termine di ogni livello BFS, la dimensione della nuova frontiera viene copiata sul lato host per verificare la condizione di arresto.

Il processo continua finché la frontiera risulta vuota, indicando che il cluster è stato completamente espanso. Lo scambio dei puntatori tra frontiera corrente e successiva evita copie di memoria superflue e migliora l'efficienza complessiva.

4 Analisi delle prestazioni

Le prestazioni dell'algoritmo GPU-DBSCAN sono state valutate confrontando i tempi di esecuzione dell'implementazione parallela su GPU (*GeForce RTX 5070*) con un'implementazione seriale su CPU (*AMD Ryzen 9 9900X*). I test sono stati condotti su dataset sintetici di dimensione n variabile da 1000 a 1 000 000 di punti.

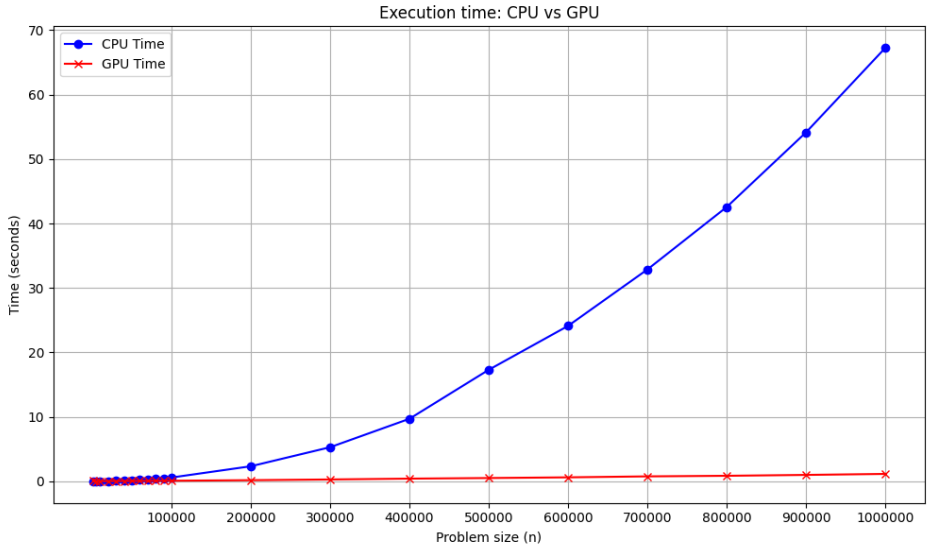


Figura 2: Confronto dei tempi di esecuzione (in secondi) tra CPU e GPU al variare della dimensione del problema.

La Figura 2 evidenzia il drastico abbattimento dei tempi di calcolo ottenuto tramite la parallelizzazione su GPU. Mentre il tempo di esecuzione su CPU cresce esponenzialmente all'aumentare di n, raggiungendo circa 68 secondi per n pari a 1 000 000, il tempo su GPU si mantiene lineare ed estremamente ridotto, circa 1 secondo o poco più per ogni dataset.

Il guadagno prestazionale è illustrato più dettagliatamente nella Figura 3, che mostra l'andamento dello *speedup*, calcolato come rapporto tra il tempo CPU e il tempo GPU. Per valori bassi di n ($< 100\,000$), lo speedup è contenuto a causa dell'overhead fisso necessario per il trasferimento dei dati tra host e device e l'avvio dei kernel. Tuttavia,

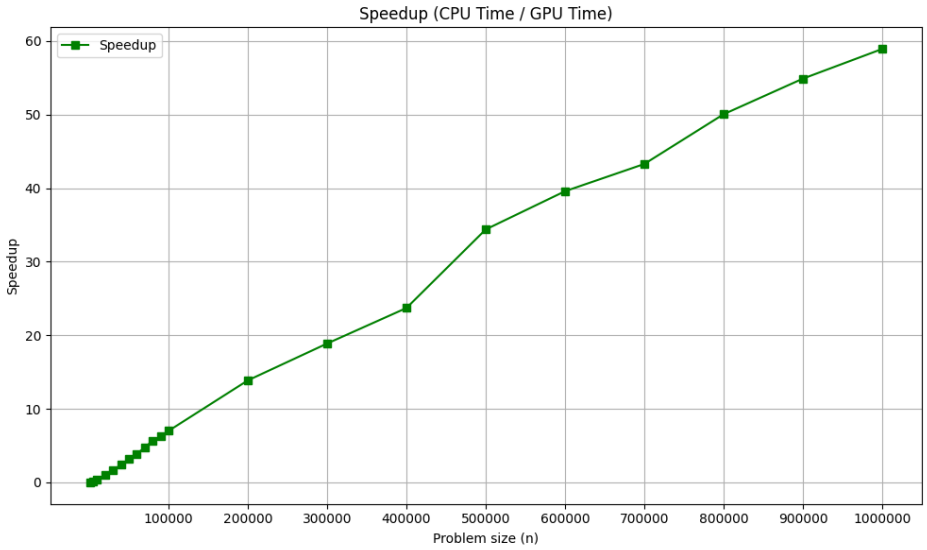


Figura 3: Speedup ottenuto al variare della dimensione del problema.

al crescere della dimensione del dataset, il vantaggio dell'esecuzione parallela su GPU diventa dominante, con uno speedup che scala linearmente fino a raggiungere un fattore di circa $\mathbf{59}\times$ per 1 000 000 di punti.

In conclusione, l'approccio parallelo su GPU risulta nettamente superiore per dataset di grandi dimensioni, rendendo trattabili in poco tempo problemi che richiederebbero tempi proibitivi su CPU.