UNIVERSITÀ
degli STUDI
di CATANIA

Merge Sort Parallelizzato in CUDA

Progetto Programmazione Parallela su Architetture GPU

Nome: Bruno Cognome: Montalto Matricola: 1000016231

Anno Accademico 2022/2023

1 Introduzione

L'efficace ordinamento di grandi volumi di dati è un requisito cruciale in molte applicazioni informatiche. L'algoritmo di ordinamento Merge Sort, noto per la sua stabilità e complessità temporale di O(n log n), offre una base solida per questo compito, tuttavia, nelle implementazioni sequenziali, potrebbe non essere sfruttato appieno il potenziale computazionale offerto dalle moderne unità di elaborazione grafica.

Questo progetto esplora l'accelerazione del Merge Sort attraverso la parallelizzazione su GPU tramite il framework CUDA. Confrontando implementazioni sequenziali e parallele, verranno analizzate diverse strategie di utilizzo della memoria condivisa, un elemento cruciale nella progettazione di algoritmi paralleli.

L'obiettivo di questo lavoro è non solo evidenziare le potenzialità della parallelizzazione su GPU per algoritmi di ordinamento, ma anche esplorare le dinamiche della memoria condivisa, fornendo spunti utili per ottimizzazioni future.

2 Merge Sort Sequenziale

Nella versione sequenziale di questo algoritmo, l’array di input viene diviso ricorsivamente (o iterativamente) a metà, fino ad arrivare a blocchi di singoli elementi, che vengono poi uniti tramite inserimento ordinato (merge) per ricostruire l’array di partenza nell’ordine corretto.
Il “punto debole” dal punto di vista della complessità temporale di questo algoritmo è l’operazione di merge, con complessità O(n).

L’algoritmo di merge sort sequenziale è stato incluso nel codice come linea base di confronto con le versioni parallelizzate.

3 Parallelizzazione su GPU

*Per evitare ambiguità, ci riferiremo ai blocchi su cui opera l'algoritmo di merge con il termine "**chunk**", useremo invece il termine "**blocco**" per indicare quelli della griglia di lancio.*

La strategia di base è la seguente:
effettuare $\log_2(n)$ iterazioni, ogni iterazione **i** ($i \in [0, \log_2(n) - 1]$) esegue il merge su coppie di chunk di 2^i elementi (`chunkSize`) adiacenti. L'obiettivo è eseguire l'operazione di merge parallelamente per ogni iterazione in tempo logaritmico, sfruttando la ricerca binaria.

3.1 Merge Parallelo - Pseudocodice

Ogni thread può determinare l'indice finale dell'elemento ad esso associato per l'i-esima operazione di merge, indipendentemente dagli altri thread.
Consideriamo i seguenti passaggi, relativamente a un generico thread con indice globale `globalIndex`:

- Il thread identifica se il chunk in cui si trova ha indice pari o dispari.

```
chunkIndex = globalIndex / chunkSize
odd = chunkIndex % 2
```

- in questo modo può identificare il primo elemento del chunk vicino coinvolto nello stesso merge `otherChunkStart`.

```
otherChunkStart = (chunkIndex+1-2*odd)*chunkSize
```

- In entrambi i casi (pari o dispari) è necessario conoscere l'indice di partenza del blocco pari della coppia (essendo questo l'indice di partenza del chunk finale di grandezza 2*`chunkSize`), viene quindi calcolato come:

```
evenChunkStart = (chunkIndex - odd) * chunkSize
```

- A questo punto è possibile calcolare il numero di elementi che precedono `thisElem` nell'ordinamento in entrambi i chunk, sfruttando il fatto che questi sono già ordinati:
    - Nel caso del chunk corrente, basta calcolare l'indice di `thisElem` relativo all'inizio del chunk, essendo questo già ordinato.
    - Nel caso del chunk vicino, invece, tale valore può essere ottenuto tramite una ricerca binaria (bSearch).

  La somma dei due risultati rappresenta l'indice finale relativo a `evenChunkStart`, basta quindi sommare quest'ultimo al risultato per ottenere l'indice finale.

```
thisChunkPrevElems = globalIndex - chunkIndex*chunkSize
if (odd == 0):
    res = bSearchFirst(thisElem,otherChunkStart,chunkSize)
else:
    res = bSearchLast(thisElem,otherChunkStart,chunkSize)
otherChunkPrevElems = res
finalIndex =
 evenChunkStart + thisChunkPrevElems + otherChunkPrevElems
```

`bSearchFirst` interrompe la ricerca alla prima (eventuale) occorrenza di `thisElem`, `bSearchLast` all'ultima. Questa distinzone granatisce il corretto funzinamento e la stabilità dell'algoritmo nel caso in cui siano presenti elementi duplicati.

3.2 Vincoli

- Ogni iterazione dipende dalla precedente, sarà quindi necessario imporre una barriera per tutti i blocchi dopo ogni iterazione.
- L'algoritmo di merge, per funzionare correttamente, ha bisogno che n sia una potenza di 2, se così non fosse, una soluzione potrebbe essere quella di aggiungere un pad che porti la dimensione alla più vicina potenza di 2 maggiore di n.

3.3 Ottimizzazioni con Shared Memory

Viste le numerose letture in memoria globale richieste dalle varie ricerche binarie, un miglioramento notevole nei tempi di esecuzione potrebbe essere raggiunto caricando i dati interessati in memoria condivisa.

Distinguiamo 3 casi:

1. Se `chunkSize < blockSize`, l'ordinamento dei singoli blocchi viene completamento effettuato in shared memory.

2. Se `chunkSize = blockSize`, sarà sufficiente che ogni blocco scriva nella propria local memory i valori dei dati presenti nel blocco ad esso vicino.

3. Se `chunkSize > blockSize`, la strategia precedente funziona ancora, ma la presenza di più di un blocco per chunk, porta a ridondanze sempre più gravi all'aumentare di chunkSize.
   Una possibilità è quella di sacrificare del parallelismo (che in realtà viene preservato se si riesce a impiegare il massimo delle risorse) per eliminare le ridondanze tra i diversi blocchi. Si potrebbe quindi lanciare un solo blocco per ogni chunk e farlo operare sequenzialmente sui vari sotto-chunk di grandezza pari a quella del blocco. Ovviamente questo è possibile fino a quando lo spazio occupato in memoria da un chunk rientra nella memoria condivisa.

3.4 Implementazione

Sono state implementate le 4 seguenti versioni dell'algoritmo
Tutte le versioni utilizzano `blockSize=256` e restituiscono un ordinamento discendente.

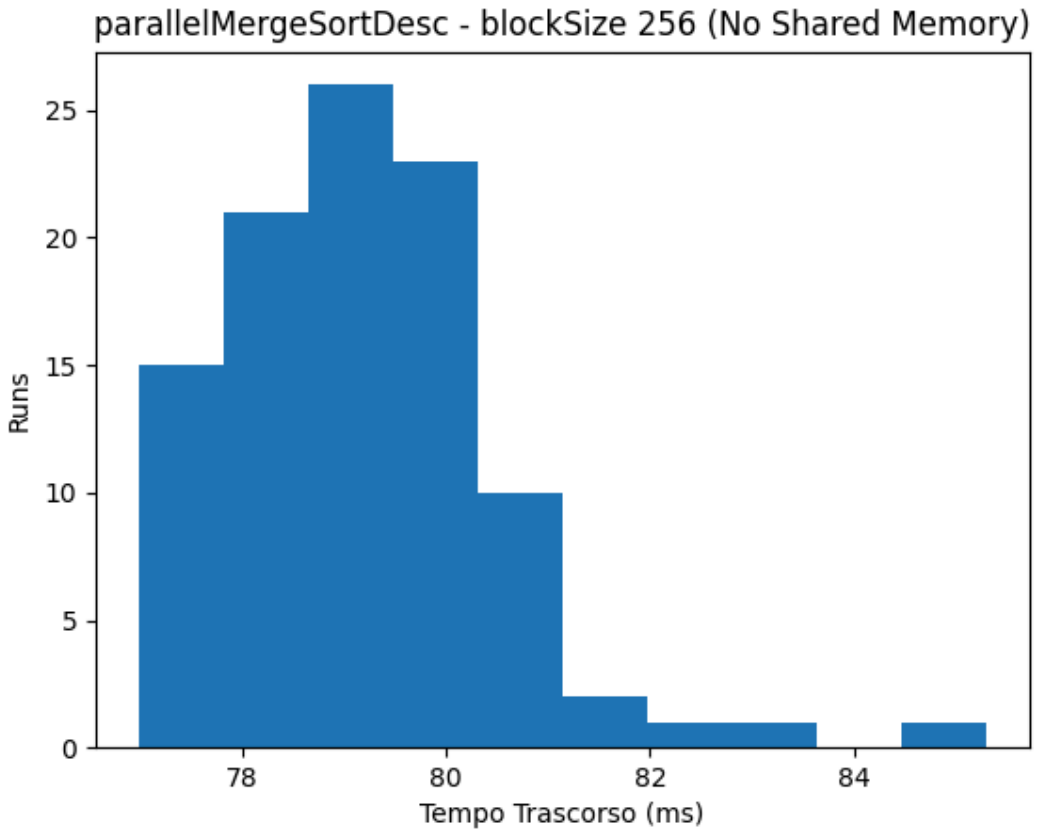
- **`parallelMergeSortDesc`**
  - Esegue il merge parallelo senza uso di shared memory.
- **`parallelMergeSortDescSM8`**
  - Utilizza la prima ottimizzazione con shared memory vista nella sezione precedente per un massimo di 8 iterazioni, le eventuali successive non fanno uso di shared memory.
- **`parallelMergeSortDescSM9`**
  - Uguale alla versione precedente, ma permette di utilizzare la shared memory anche alla 9ª iterazione, cioè il caso in cui `chunkSize = blockSize`.
- **`parallelMergeSortDescSM`**
  - Utilizza tutte le 3 strategie della sezione precedente.

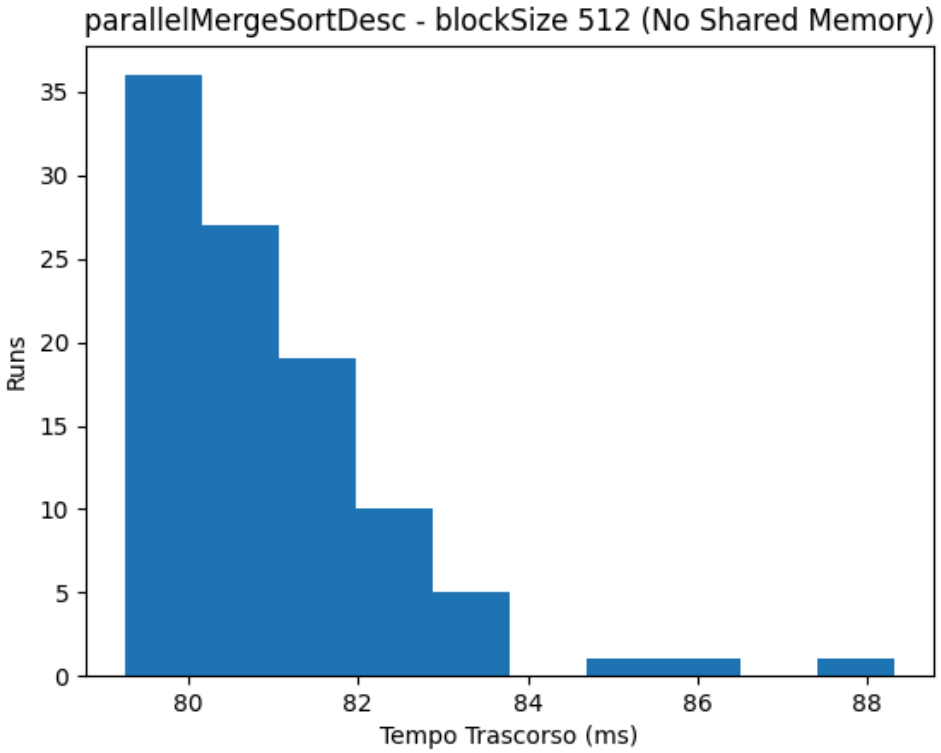
- Calcola in anticipo il numero di iterazioni con shared memory possibili in base alla grandezza della memoria condivisa per blocco.

4 Risultati Sperimentali

Le varie versioni dell'algoritmo sono state eseguite per 100 run su array di 2^{24} elementi generati casualmente, su una NVIDIA GeForce RTX 3060. Gli array sono popolati da istanze di Entity, una semplice struttura dati, la sua chiave per l'ordinamento è l'attributo distanceFromPlayer.

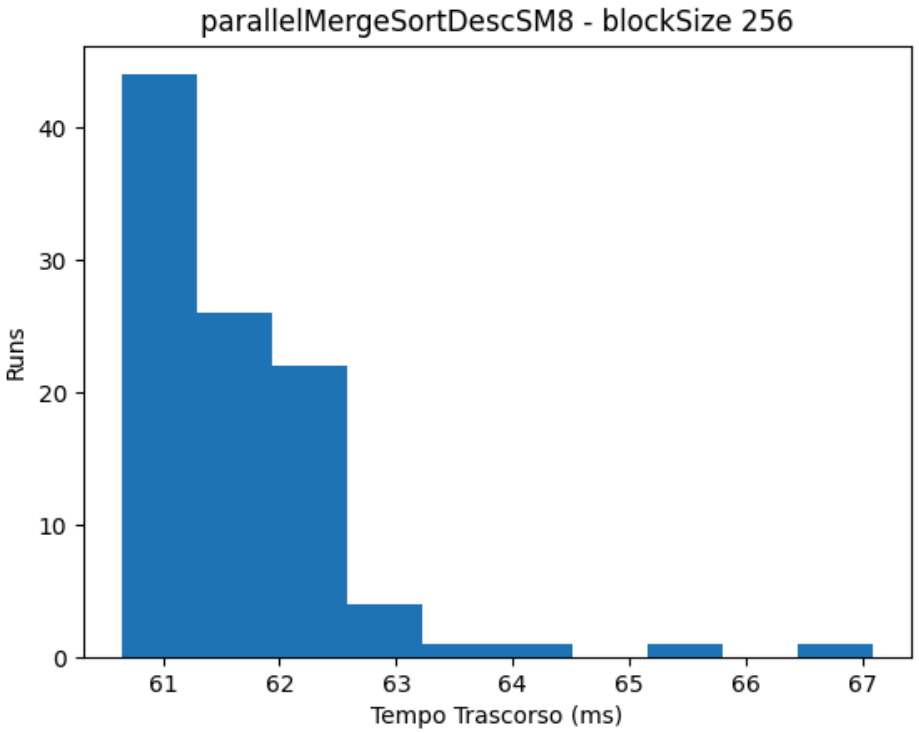
- Per prima cosa è stato effettuato un confronto per i valori di blockSize 256 e 512, senza memoria condivisa.

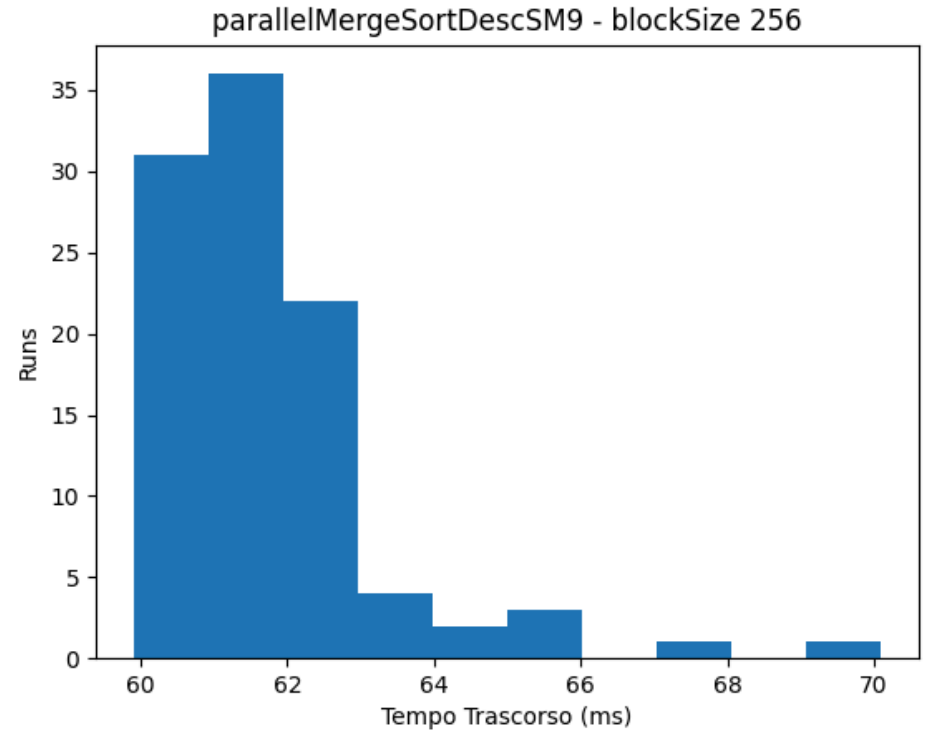


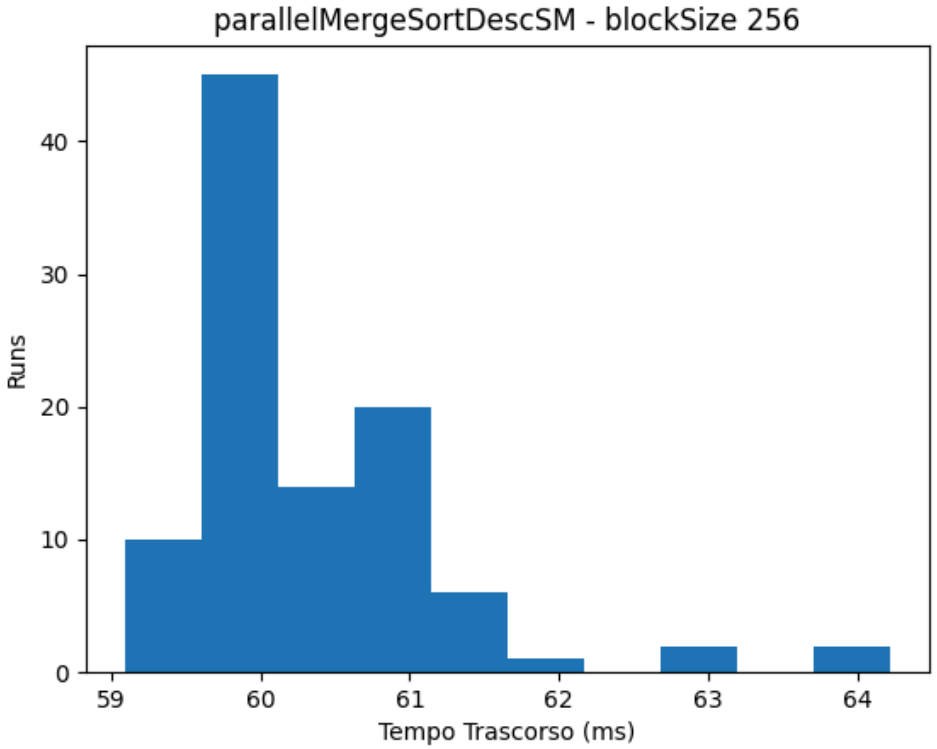


Il valore di blockSize consigliato per la GPU utilizzata è 256, coerente con i risultati ottenuti.

- Di seguito i risultati delle rimanenti versioni, con blockSize 256

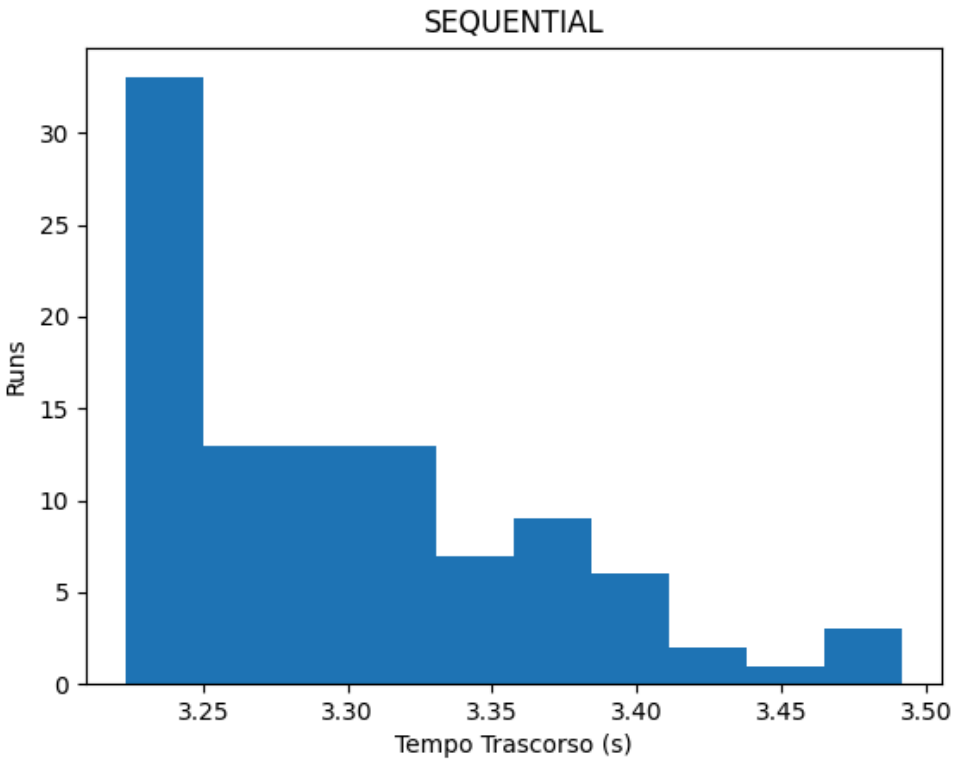






In questo caso la shared memory è stata usata fino alla 11ª iterazione.

Risultati per l’algoritmo sequenziale:



5 Conclusioni

I risultati ottenuti mostrano come un uso corretto della memoria condivisa per ridurre le letture in memoria globale possa portare a visibili miglioramenti nel tempo di esecuzione.

Tuttavia, è importante sottolineare che l'efficacia di queste ottimizzazioni può variare in base alle dimensioni del problema, l’ultima in particolare. Ulteriori sperimentazioni potrebbero essere condotte per valutare le prestazioni su dati di dimensioni diverse e per esplorare ulteriori ottimizzazioni specifiche del contesto.