WarpSort: sorting efficiente basato sui warp

Daniele Cazzato, Davide Franchi

daniele.cazzato1@studenti.unimi.it, 982410
davide.franchi@studenti.unimi.it, 960803

Indice

1 Introduzione

Il progetto consiste nell'implementazione di [1]. L'algoritmo propone come aspetto fondamentale l'utilizzo dei *warp* come unità di parallelizzazione fondamentale. Infatti, i 32 thread all'interno di un warp non necessitano di barriere di sincronizzazione. L'algoritmo prevede quattro fasi, tra le quali invece la sincronizzazione deve necessariamente essere forzata. La prima fase prevede l'ordinamento di piccole sequenze di dimensione costante, tramite *BitonicSort*. La seconda consiste in un merge delle sequenze appena ordinate, che procede finchè il numero di sequenze presenti è abbastanza elevato da garantire un buon livello di parallelismo. La terza fase fa fronte alla necessità di ristabilire un buon grado di parallelismo, dividendo nuovamente le sequenze in più parti, garantendo un'importante proprietà d'ordine. Infine, la quarta ed ultima fase procede nuovamente con il merge, in maniera simile (ma non uguale) alla fase due, per ottenere l'array ordinato.

2 Architettura utilizzata

Nel corso del progetto è stata utilizzata una GPU *NVIDIA GeForce GTX 1660 Ti*, avente le seguenti caratteristiche:

CUDA Driver Version / Runtime Version	11.2 / 11.2
CUDA Capability Major/Minor version number:	7.5
Total amount of global memory:	5937 MBytes
(24) Multiprocessors, (64) CUDA Cores/MP:	1536 CUDA Cores
L2 Cache Size:	1572864 bytes
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Total number of registers available per block:	65536
Warp size:	32
Maximum number of threads per multiprocessor:	1024
Maximum number of threads per block:	1024
Device supports Unified Addressing (UVA):	Yes

3 Implementazione

In questo capitolo sono elencati i dettagli relativi alle quattro fasi che compongono l'algoritmo. Si noti che nei paragrafi che seguono sono descritte esclusivamente le scelte implementative adottate. Non si affronta la logica dell'algoritmo, che rimane descritta in [1].

Si assume che la sequenza di input `a` sia composta da N interi di 32 bit, dove N è una potenza di 2.

3.1 Step 1

Nella prima fase la sequenza `a` viene suddivisa in $\frac{N}{\texttt{W}}$ sottosequenze di ugual dimensione, ciascuna ordinata tramite *BitonicSort* da un singolo warp. Si noti che `W` è una costante, è una potenza di 2 e deve essere ≥ 128, per garantire l'assenza di thread idle. In particolare, il kernel `bitonicSort` implementa una sorting network, attraverso le procedure `asc` (comparatore *ascendente*) e `desc` (comparatore *discendente*). La sottosequenza di `W` interi viene inizialmente caricata in shared memory (matrice `buffer`, indicizzata per righe da `WARP_NUM`: numero del warp nel blocco). Si noti che il caricamento della sottosequenza in shared memory avviene attraverso accessi a memoria *coalescenti*, dal momento che i thread appartenenti al warp hanno *id* consecutivi e ogni thread copia elementi a distanza `WARP_SIZE` $= 32$. Affinchè `buffer` non ecceda 49152 byte (dimensione della shared memory per blocco), `W` può essere al massimo 256, dovendo essere una potenza di 2.
La costante `FROM` indica l'inizio all'interno di `a` della sottosequenza assegnata al warp corrente. Dal momento che ad ogni warp è assegnata una sottosequenza e dal momento che queste sono tutte di ugual dimensione, per calcolare `FROM` è sufficiente moltiplicare il numero del warp (globale, all'interno della grid) per `W`.
Infine, la funzione *device* `sortBuff` (che ordina una sequenza bitonica) è stata estratta, poichè utilizzata anche da fasi successive.

3.2 Step 2

Nella seconda fase le $\frac{N}{\texttt{W}}$ sottosequenze ordinate nello Step 1 devono essere *fuse*. Questo avviene a più riprese, fondendo coppie contigue: ad ogni iterazione, ogni coppia viene assegnata ad un singolo warp. Tuttavia, questa fase

non prosegue fino ad ottenere una singola sequenza ordinata, ma termina raggiunta una soglia `L` (costante, ≥ 64 e potenza di 2) di sequenze ordinate. Questo avviene per garantire un buon grado di parallelismo, che verrebbe invece a mancare con un numero ridotto di coppie di sequenze da fondere. Infatti si avrebbe un numero di coppie inferiore al numero di warp che possono lavorare in parallelo, lasciando inattivi dei multiprocessori.
Il kernel `merge` si occupa di fondere una coppia di sottosequenze contigue, denominate `A` e `B`. Ad ogni warp del blocco corrisponde una riga della matrice `buffer` in shared memory, chiamata `buff`. Di volta in volta vengono prelevati `TO_PICK` = `T`/2 elementi da `A` oppure da `B` (`T` è una costante multipla di 64), seguendo l'ordine. I `TO_PICK` elementi prelevati vengono disposti in `buff` affinchè la sequenza ottenuta risulti essere bitonica. Successivamente `buff` viene ordinato dalla procedura `sortBuff` ed i suoi `TO_PICK` elementi più piccoli vengono scritti nella sequenza di output. Il prelievo degli elementi avviene mediante la procedura `pick`, che prevede sei parametri. Tra questi, `rev` è un valore booleano che indica se gli elementi prelevati da `src` devono essere disposti in `dst` in ordine inverso, oppure no. Si noti che l'accesso a `src` è sempre coalescente, mentre quello a `dst` può non esserlo (se `rev` = *true*). Questo è preferibile rispetto al contrario, dal momento che tutte le volte in cui `rev` è *true* vale che la sorgente è in global memory, mentre la destinazione è in shared memory. Queste hanno infatti tempi di accesso differenti. In particolare, gli accessi alla prima sono molto più lenti. Infine, `off` è l'offset da aggiornare (passato per riferimento), a volte riguardante `src` (se si sta caricando `buff`), a volte invece `dst` (se si sta scrivendo in output).

3.3 Step 3

La fase tre consiste nello "spezzare" le `L` sottosequenze ordinate, ottenute dallo Step 2, in `S` + 1 sottosequenze, aventi la seguente proprietà (assumendo che le sottosequenze siano indicizzate da 0):

Proprietà 1 $\forall x, y \in \{0, ..., \mathtt{L}-1\}$ *e* $\forall i, j \in \{0, ..., \mathtt{S}\}$ tale che $i < j$*:*

$$\forall a \in \textit{sottosequenza}(x, i), \forall b \in \textit{sottosequenza}(y, j) : \ a \leq b$$

Si noti che le sottosequenze a seguito dello *split* possono avere dimensioni diverse ed in particolare essere vuote.
Lo split avviene memorizzando `S` elementi ordinati (estratti casualmente dalla sequenza di input) nell'array `splitters` e costruendo la matrice `splitPoints`.

Quest'ultima (implementata come matrice linearizzata) contiene `L` righe ed `S` + 1 colonne. In particolare, $\forall i \in \{0, ..., \texttt{L} - 1\}$ e $\forall j \in \{0, \ldots, \texttt{S} - 1\}$, `splitPoints`$[i][j]$ contiene l'indice dell'ultimo elemento nella sottosequenza i-esima ad essere $\leq$ `splitters`$[j]$. Invece, `splitPoints`$[i][\texttt{S}]$ contiene sempre l'indice dell'ultimo elemento della sequenza, ovvero l'ultimo ad essere $\leq$ `splitters`$[\texttt{S}]$, che non esiste e che assumiamo essere $+\infty$. Nel kernel `split` ad ogni warp è assegnata una delle `L` sequenze. Ogni thread si occupa quindi di individuare `S`/32 punti di split (come definiti nella matrice `splitPoints`) e per ciascuno applica una ricerca dicotomica (dato che le `L` sequenze sono ordinate). Essendo `L` una costante ridotta ed avendo a disposizione 24 multiprocessori, per massimizzare il parallelismo il kernel `split` viene invocato con `L` blocchi e un singolo warp per blocco.
Infine, nella fase tre si introduce del *padding* (costituito da valori `INF`), per fare in modo che ogni sottosequenza abbia una dimensione multipla di `T`/2. Questa operazione viene effettuata prima calcolando la dimensione dell'array ottenuto introducendo il padding, poi creando un array di tale dimensione (riempito con `INF`) ed infine copiando nel nuovo array le sottosequenze *opportunamente* (kernel `copyKeepingPadd`), come mostrato in Figura 1.

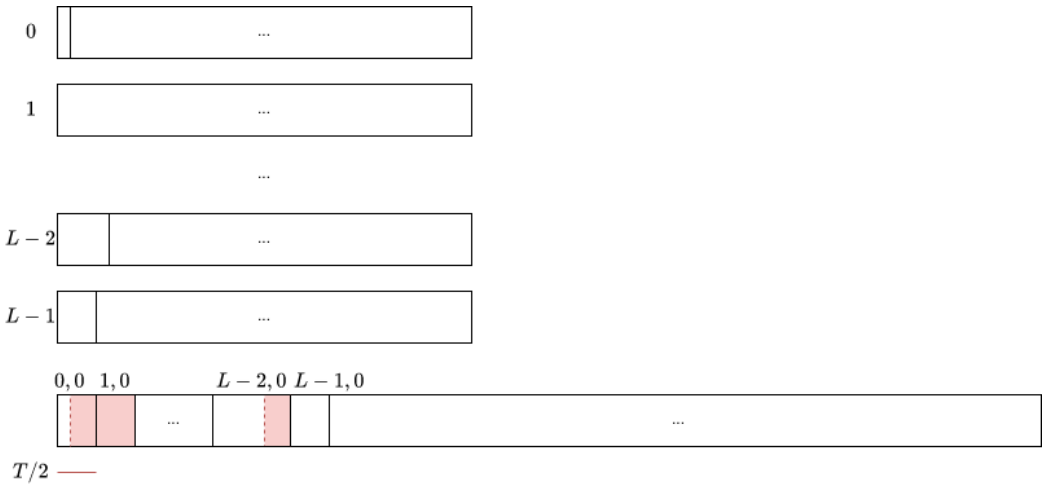


Figura 1: *Concatenazione di sottosequenza*$(0, 0), \ldots,$ *sottosequenza*$(\texttt{L} - 1, 0)$*, introducendo il padding (evidenziato in rosso). Si noti che sottosequenza*$(1, 0)$ *è vuota, mentre sottosequenza*$(\texttt{L} - 1, 0)$ *non necessita di padding. Le nuove sottosequenze hanno tutte dimensione non nulla e multipla di* `T`/2*.*

3.4 Step 4

Le S+1 sottosequenze possono adesso essere ordinate indipendentemente. Queste infatti godono della Proprietà 1, a meno degli INF, che verranno comunque facilmente rimossi alla fine. Le sottosequenze vengono quindi ordinate similmente a quanto fatto nello Step 2, tenendo conto però che adesso le parti da fondere potrebbero avere dimensioni diverse. Le operazioni appena descritte vengono svolte dal kernel `step4`. Infine, i valori INF vengono rimossi dal kernel `filterAll`, che fa uso del parallelismo dinamico. Infatti, ogni thread provvede a lanciare il kernel `filter` su una delle $\mathtt{S}+1$ sequenze da filtrare. Quest'ultimo poi assegna ad ogni warp W elementi da spostare, ammesso che vi siano, così da portare gli N elementi $\neq$ INF nelle prime N posizioni (dell'array che verrà restituito).

4 Analisi delle prestazioni

L'algoritmo WarpSort è stato implementato in CUDA C (scegliendo come costanti $\mathtt{W} = 2^7$, $\mathtt{T} = 2^6$, $\mathtt{L} = 2^6$ e $\mathtt{S} = 2^8$) e confrontato con l'algoritmo di ordinamento sequenziale presente nella libreria standard. Il grafico in Figura 2 mostra i tempi ottenuti (in ms) su istanze di dimensione N, generate casualmente scegliendo numeri interi nell'intervallo $[0, 2^{25}-1]$ con probabilità uniforme.

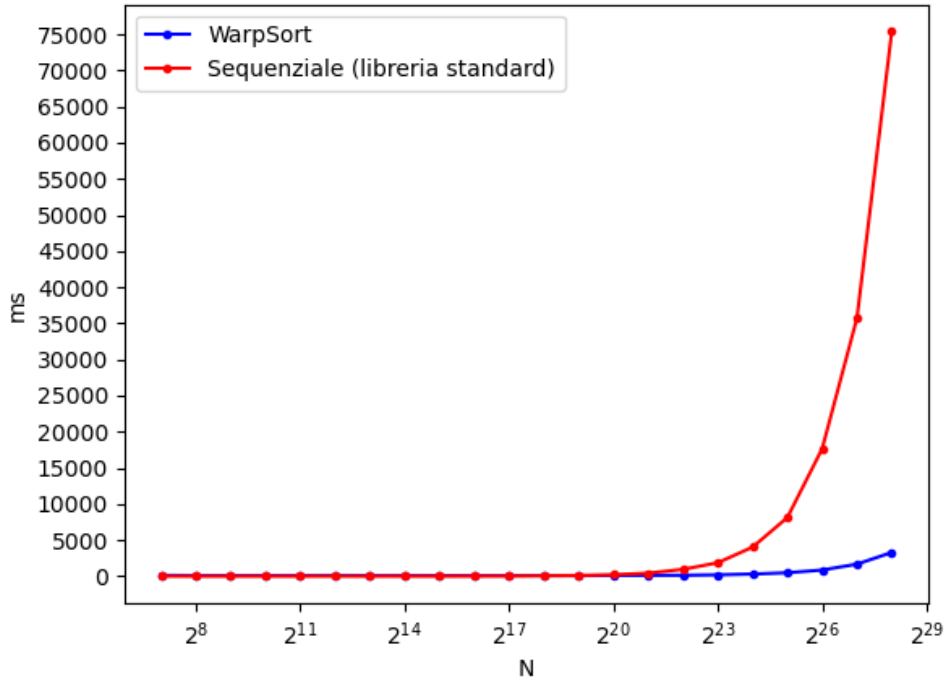


Figura 2: *Tempi di esecuzione a confronto. La scala dell'asse N è logaritmica, mentre quella dell'asse dei tempi non lo è.*

La Figura 3 mostra lo speed up registrato, al crescere di N.

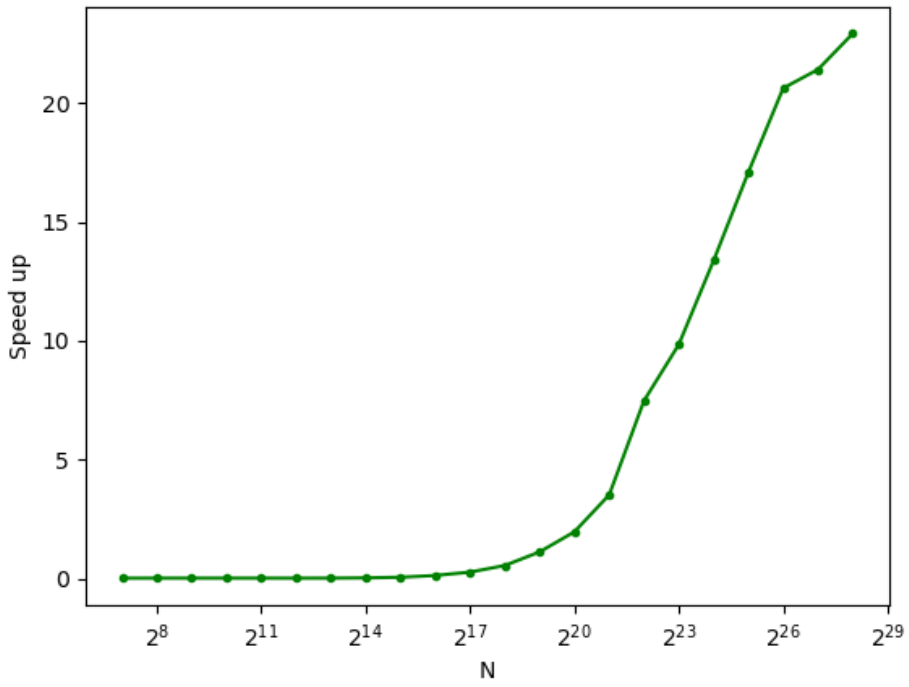
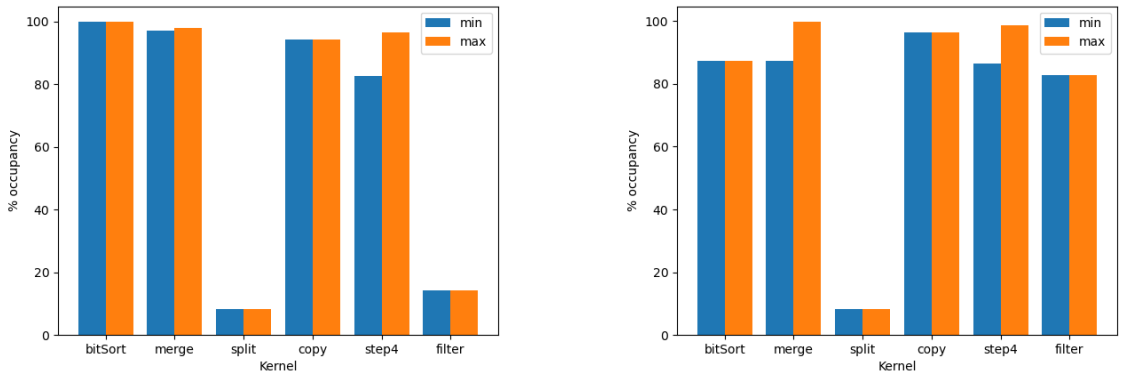


Figura 3: *Andamento del rapporto tra il tempo richiesto dall'algoritmo sequenziale e quello richiesto dal WarpSort, al variare di N.*

Si noti che l'implementazione del WarpSort proposta assume che N sia una potenza di 2 maggiore o uguale a `W` ed `L`.
L'assunzione che N sia una potenza di 2 consente di semplificare l'implementazione e può essere facilmente ottenuta aggiungendo un numero opportuno di `INF` alla sequenza. Tuttavia, il codice necessario a questo *pre-processing* non è stato aggiunto, pertanto N deve essere scelto a priori come indicato. L'assunzione di $N \geq \texttt{W}, \texttt{L}$ invece è dettata dal fatto che non avrebbe senso introdurre del codice di controllo aggiuntivo per gestire istanze tanto piccole, per le quali l'algoritmo sequenziale è di fatto equivalente a quello parallelo in termini di performance, come mostrato dalla Figura 2.
Infine, la Figura 4 evidenzia i valori minimi e massimi di occupancy riscontrati durante l'esecuzione dei vari kernel, in corrispondenza di un'istanza "piccola" ($N = 2^{15}$) e di una "grande" ($N = 2^{25}$). Le rilevazioni sono state effettuate

servendosi di NVIDIA Nsight Compute e settando come metrica (opzione `--metrics`) `sm__warps_active.avg.pct_of_peak_sustained_active`.



(a) $N = 2^{15}$ (b) $N = 2^{25}$

Figura 4: *Minimi e massimi livelli di occupancy, ottenuti a seguito delle diverse esecuzioni dei vari kernel.*

Si noti che i kernel `bitonicSort` (*bitSort*), `split`, `copyKeepingPadd` (*copy*) e `filterAll` (*filter*) presentano sempre lo stesso valore di minimo e di massimo, essendo eseguiti una sola volta. L'unico kernel a presentare una occupancy ridotta risulta essere `split`. Questo infatti coinvolge un ridotto numero di warp (che non dipende dalla dimensione dell'input), trattandosi di un'operazione di per sè rapida.

5 Fonti

[1] Ye, Xiaochun and Fan, Dongrui and Lin, Wei and Yuan, Nan and Ienne, Paolo. High performance comparison-based sorting algorithm on many-core GPUs. *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 1–10, 2010.