RELAZIONE

{INTRODUZIONE}

Uno fra i trend più prorompenti al giorno d'oggi è l'esplosione della produzione e disponibilità di dati in larga scala, a conferma di tale evento sono nate vere e proprie discipline atte ad estrarre conoscenza da questa enorme mole di dati.

In questo contesto di abbondanza di dati, spesso e volentieri l'hardware tradizionale ,e gli algoritmi disegnati su di esso, stenta a tenere il passo dal punto di vista delle performance.

Da questo e altri pretesti, è nata l'esigenza di maggiore potenza di elaborazione e di conseguenza di hardware più potente, questo problema è spesso e volentieri arginato spostando il calcolo dalla CPU alla GPU, infatti questa tecnologia è attualmente molto usata nel Calcolo ad Alte Prestazioni.

Quest'ultimo passaggio non è automatico e necessità di algoritmi scritti appositamente per le GPU, di conseguenza negli ultimi anni c'è un grande fermento della comunità scientifica da questo punto di vista; un esempio può essere il motore per database scritto da *Yong et al. [1],* la particolarità di questo motore è il supporto ,per alcuni componenti, all'esecuzione su GPU.

Per i motivi appena discussi, si è voluto sfruttare il progetto di questa materia per implementare una fra le operazioni più frequenti nell'elaborazione dei dati, ovvero l'ordinamento, sfruttando un hardware estremamente performante come la GPU con l'obbiettivo sia di incrementare le performance rispetto all'implementazione canonica che di comprendere il più possibile le "meccaniche" e il paradigma di programmazione di questo hardware.

Nello specifico, è stato scelto il QuickSort come algoritmo di ordinamento.

{FUNZIONAMENTO DEL QUICKSORT}

Il QuickSort è un algoritmo di ordinamento in loco basato sui confronti e sull'approccio *Divide et Impera,* quest'ultimo approccio prevede la scomposizione ricorsiva dei dati da processare (il problema iniziale) in sotto-processi (o sotto-problemi), in particolar modo la procedura che scompone ,ricorsivamente, i dati si chiamata ***partition***.

Entrando nello specifico, il funzionamento del QuickSort è il seguente:

- Lancio la procedura ***partition*** sull'array iniziale
- Nomino un elemento dell'array come ***pivot***
- Inserisco gli elementi minori o uguali al pivot all'inizio dell'array mentre gli elementi più grandi di quest'ultimo li inserisco alla fine.
- Fatto ciò, inserisco il pivot "al centro dell'array" ovvero in quella locazione di memoria tale che: tutte le celle ,dell'array, antecedenti a quella locazione conterranno dati più piccoli o uguali al pivot mentre tutte le celle che verranno dopo quella locazione conterranno dati maggiori del pivot.
- Così facendo è stata generata una partizione dell'insieme di partenza; in sostanza la partizione contiene 3 insiemi: i dati più piccoli del pivot, il pivot stesso e i dati più grandi del pivot.
- In sostanza l'operazione appena effettuata ha generato 2 sottoproblemi ,dello stesso tipo, a partire dal problema originario.
- A questo punto si riapplica ricorsivamente la partition ai due sottoproblemi appena creati fino a quando tutti i sottoproblemi non saranno stati risolti, ovvero quando tutti i sottoproblemi avranno cardinalità 1.
- Appena tutti i sottoproblemi saranno stati risolti, ci ritroveremo già con l'array ordinato dato che l'algoritmo opera in loco.

Prima di passare alle sezioni successive, è doveroso fare un cenno sulla struttura dati "indotta" dal QuickSort e sulla complessità di questo Algoritmo.

Al livello di struttura dati, le chiamate del QuickSort ai singoli sottoproblemi possono essere visualizzate come un Albero Binario di Ricerca; nello specifico l'albero è binario perché ogni sottoproblema può scindersi al più in 2 parti e di Ricerca perché i sottoproblemi dell'Albero sono ordinati a causa del modo in cui è costruita la logica della procedura ***partition***.

Dal punto di vista della complessità temporale il QuickSort ha un caso medio di $O(n \log_2 n)$ e un caso pessimo di $O(n^2)$.

Da precisare che queste complessità si riferiscono al modello teorico RAM, ovvero al modello che prevede un singolo processore che opera su una singola memoria.

{IDEA DI BASE PER LA PARALLELIZZAZIONE}

Il QuickSort fa parte della classe di algoritmi che sono non imbarazzantemente paralleli, ovvero è un algoritmo la cui implementazione parallela non è immediata.

Entrando nello specifico del progetto, è importante dire che tutte le implementazioni di questo progetto sfruttano un'idea base: ovvero l'Albero delle Chiamate esplicito.

Nella sezione precedente è stato già detto che le chiamate del QuickSort ai singoli sottoproblemi inducono un Albero Binario di Ricerca; in questa sezione si vuole aggiungere un ulteriore dettaglio implementativo, ovvero che nella versione seriale del

QuickSort si può scegliere se scrivere la catena di chiamate in modo esplicito (a questo punto l'implementazione del QuickSort diventa Iterativa) oppure in modo implicito (sfruttando la ricorsione).

Questa scelta ,compatibilmente con l'hardware e il linguaggio scelto per questo progetto, non c'è su GPU dato che il linguaggio scelto per questo progetto non supporta la ricorsione.

Di conseguenza, tutte le implementazioni di questo progetto sfrutteranno l'approccio iterativo.

Nello specifico ,in tutte le implementazioni la procedura ***partition*** viene eseguita su GPU mentre i calcoli per decidere se lanciare o meno la procedura vengono fatti dall'host.

A conclusione di questa sezione è doveroso fare una precisazione in merito alla pipeline di esecuzione, in sostanza ad ogni chiamata alla ***partition*** vengono elaborati tutti i sottoproblemi del livello attuale dell'albero al momento della chiamata (a differenza del classico approccio su CPU dove ogni chiamata alla partition elabora un singolo sottoproblema).

{API SCELTA E HARDWARE DI TEST}

L'API scelta per questo progetto è stata OpenCL (la versione è la 1.2), il motivo di tale scelta è totalmente arbitrario e quindi non ha un argomentazione scientifica.

In ogni caso il motivo della scelta è che ,secondo il parare di chi scrive, OpenCL è un API che mette in risalto meglio il funzionamento della GPU dato che maschera meno cose possibili; questa caratteristica dal punto di vista dell'apprendimento aiuta.

Il rovescio della medaglia è la poca maturità dell'API dal punto di vista della capacità di scalare su Hardware di vendor diversi, infatti molti codici di questo progetto hanno seri problemi di funzionamento sulle GPU non NVIDIA, mentre sulle GPU NVIDIA non hanno alcun problema; nello specifico hanno problemi solamente i codici che sfruttano i Work-Group.

É per questo motivo che è stata presa la decisione ,a malincuore, di escludere l'hardware che creava problemi in modo da uniformare i test.

Conseguentemente a questo discorso, i test sono stati effettuati usando il seguente Hardware:

Maca	**Serie**	**Modello**	**Architettura**	**Multiprocessori**	**Processing Elements**
NVIDIA	GeForce GTX	780	Kepler	12	2304 (192 a Multiprocessore)

{METODI DI VALUTAZIONE}

Dal punto di vista dei metodi di valutazione sono state fate varie scelte, in generale ogni implementazione di questo progetto avrà il compito di ordinare un insieme di input costruito attraverso un generato di numeri pseudo-causali (nello specifico si è sfruttato la primitiva ***rand*** del linguaggio C), inoltre per tutte le implementazioni il ***SEED*** è stato impostato a 20 in modo da garantire l'uniformità dei test.

Oltre a ciò, ogni programma costruito è stato sottoposto a 2 test che si differenziano nella grandezza dell'insieme di dati da ordinare, nello specifico ecco i dettagli dei test:

Test 1	**Test 2**
524.288 elementi da ordinare	1.048.576 elementi da ordinare

Le cardinalità degli insiemi contenuti nei test hanno una particolarità, infatti queste cardinalità sono entrambe un multiplo di 32.

Quest'ultima argomentazione è cruciale dal punto di vista delle prestazioni, infatti 32 è proprio la dimensione dell'Unità Minima di Esecuzione Hardware (o Warp) sulle GPU NVIDIA, che ricordiamo è la GPU utilizzata per il progetto.

Di conseguenza, se diamo in input alla GPU un numero di elementi che è multiplo di 32 segue che il driver può "distribuire equamente i Work-Item nei Warp", se così non fosse si rischierebbe di avere Warp in esecuzione dove pochi Work-Item lavorano a causa dei restanti ***Numero_Elementi mod Dimensione_Warp*** elementi, tutto ciò porta ad un rallentamento delle prestazioni.

A conferma di questo discorso, sono stati fatti vari test su varie implementazioni inserendo un numero di elementi da ordinare che non è multiplo del Warp, il primo risultato da costatare è stata la perdita di performance in tutte le implementazioni di questo progetto.

Inoltre il risultato ,su molte implementazioni,è stato disastroso; addirittura in alcuni casi si è passati da un tempo di esecuzione di 5 secondi ad un tempo di esecuzione di oltre due minuti.

Per tanto si è esclusa quest'opzione di ordinamento dai test a causa dell'elevatissima differenza nel tempo di esecuzione.

Prima di chiudere questa sezione bisogna fare 2 precisazioni: la prima è in merito alla numerosità dei Test e la seconda è in merito alle metriche per valutare le implementazioni.

Dal punto di vista dei test, si è volontariamente evitato di aggiungere ulteriori test con insiemi di cardinalità superiore al Test 2.

Questa scelta è stata presa per motivi di uniformità dei test; infatti solamente due implementazioni di questo progetto riescono ad ordinare più di 3 milioni di elementi in tempi ragionevoli, le restanti implementazioni hanno tempi di esecuzione lunghissimi.

L’ultimo tassello di questa sezione sono le metriche di valutazione, infatti ogni implementazione di questo progetto verrà valutata secondo due metriche: il ***Tempo di Esecuzione*** (o Runtime) in millisecondi e gli ***Elementi Computati*** in Giga Elementi al Secondo

Nello specifico, la metrica ***Elementi Computati*** è data dalla divisione fra il numero di elementi da ordinare e il ***Tempo di Esecuzione*** che ci è voluto per ordinarli.

Va da se che a parità di elementi da ordinare, le metriche ***Elementi Computati*** e ***Tempo di Esecuzione*** sono inversamente proporzionali; nello specifico noi vorremmo sempre minimizzare il tempo di esecuzione e di conseguenza massimizzare gli ***Elementi Computati***.

{IMPLEMENTAZIONE 1}

L’idea di base per questa implementazione è stata la parallelizzazione del QuickSort soltanto dal punto di vista dei sottoproblemi.

Infatti, in questa implementazione viene lanciato 1 Work-Item per ogni sottoproblema.

All’interno della risoluzione del singolo sottoproblema la risoluzione resta seriale.

Nello specifico ecco la pipeline di esecuzione di questa implementazione:

- Inizialmente l’Host organizza le strutture dati da passare al device: nello specifico l’Host crea le seguenti strutture dati per il device:

    - ***Delimiters***, questa struttura dati conterrà una lista contigua dell’inizio e della fine di ogni sottoproblema che ancora bisogna risolvere.
      Inizialmente questo buffer viene inizializzato con l’inizio e la fine dell’array di input.
      Inoltre, questo Buffer avrà dimensione 2×2^h, *dove* ***h*** è l’altezza dell’Albero Binario di Ricerca (BST) indotto dal QuickSort.

      La spiegazione di questa scelta è presto detta:
      Dalla teoria sappiamo sia che l’altezza di un Albero Binario di Ricerca è $log_2(numero\_di\_elementi)$ sia che il numero massimo di nodi ad un determinato livello dell’Albero è $2^{livello\_albero}$.

      Unendo queste definizioni con l’esigenza di creare una struttura dati che abbia spazio sufficiente per mantenere in memoria tutti i sottoproblemi di un qualunque livello dell’Albero, segue che il buffer sarà dimensionato come 2×2^h dove il 2 a moltiplicare viene inserito perché ogni sottoproblema occupa due elementi dell’array, ossia l’inizio e la fine del sottoproblema.

- ***App***, questo è il buffer dove verranno scritti i nuovi sottoproblemi che si generano; lo spazio occupato da questo buffer è ovviamente identico a quello occupato dal buffer ***delimiters***.

- ***Input***, questo è il buffer che conterrà l'array di input inizialmente e l'array ordinato alla fine.

- Una volta create le strutture dati da passare al device, viene eseguito il codice su GPU, che in sostanza effettua la ***partition*** di tutti i sottoproblemi contenuti nell'array ***Delimiters***.
  Chiaramente, al primo lancio del kernel si avrà 1 solo Work-Item in esecuzione dato che in questo caso c'è 1 solo sottoproblema da partizionare.

- Esaminando il lato GPU, ogni Work-Item preleva le informazioni (inizio e fine) del proprio sottoproblema dall'array ***Delimiters*** in Global Memory.

- Ogni Work-Item imposta il pivot come l'ultimo elemento del proprio sottoproblema

- Ogni Work-Item imposta un indice ,chiamato indice di scambio, ad un numero che è uguale "all'inizio del sottoproblema – 1"

- Ogni Work-Item cicla sul proprio sottoproblema escludendo il pivot
  - All'interno del ciclo, il Work-Item verifica se l'elemento corrente è minore o uguale al pivot
    - Se l'if da esito positivo, il Work-Item aggiorna l'indice di scambio aggiungendogli 1, successivamente scambia l'elemento corrente del ciclo con l'elemento posizionato al nuovo valore dell'indice di scambio.
      In sostanza questo if favorisce la creazione di due sottoinsiemi dell'array di partenza: l'insieme dei più piccoli (o uguali) del pivot e quello dei più grandi del pivot.

- Alla fine del ciclo, il Work-Item scambia il pivot col primo elemento dell'insieme dei maggiori andando ,di fatto, a terminare la ***partition*** del sottoproblema iniziale.

- In ultima analisi, il Work-Item si calcola inizio e fine dei sottoproblemi che ha generato e li scrive nell'array ***App*** in modo da permettere ai Work-Item del prossimo lancio di individuare correttamente il proprio sottoproblema.
  Ovviamente può capitare che un Work-Item risolva definitivamente un sottoproblema oppure che il sottoproblema si scinda in una solo parte anziché due (ad esempio se il pivot si trovava all'inizio); in questo caso il Work-Item scriverà il valore ***-1*** all'inizio e alla fine del sottoproblema risolto.
  Un ulteriore precisazione va fatta sulla scrittura sull'array ***App,*** nello specifico la scrittura viene fatta su questo array e non sull'array **Delimiters** per motivi di concorrenza fra le scritture dei Work-Item

- Arrivati a questo punto termina il kernel e rientra in gioco il lato host dove viene invocata la funzione ***clean_vector*** che non fa altro che trasferire le informazioni dei sottoproblemi dall'array ***App*** all'array ***Delimiters*** e determinare se rimangono ancora sottoproblemi da risolvere o meno.

- Se restano sottoproblemi da risolvere si lancia ancora una volta la ***partition*** su GPU, altrimenti la computazione dell'ordinamento è terminata.

Le performance di queste implementazione sono mostrate nella tabella sottostante:

Runtime Test 1	Elementi Computati Test 1	Runtime Test 2	Elementi Computati Test 2
1260.7 ms	415.87 GE/s	1451.7 ms	722.307 GE/s

A conclusione di questa sezione è doveroso fare un commento su questi risultati, in sostanza le prestazioni sono discrete; nonostante ciò dal punto di vista dello sfruttamento dell'Hardware ,e quindi anche delle performance, si può fare di meglio.

Del resto, questo l'approccio è parallelo solamente dal punto di vista dei sottoproblemi.

{IMPLEMENTAZIONE 2}

Il difetto principale della prima implementazione era lo scarso sfruttamento dell'Hardware. Con l'obbiettivo di arginare questo fenomeno, si è pensato di scrivere una nuova implementazione che mitighi questo problema; questa nuova implementazione lancia un numero di Work-Item pari al numero di elementi che restano da ordinare anziché 1 solo Work-Item come l'***Implementazione 1***; in sostanza abbiamo un associazione 1:1 fra Work-Item ed elementi dell'array da ordinare.

Di seguito è riportata la spiegazione dettagliata dell'Implementazione:

- Inizialmente l'Host organizza le strutture dati da passare al device: nello specifico l'Host crea le seguenti strutture dati per il device:
    - ***Delimiters***, questa struttura dati conterrà una lista contigua dell'inizio e della fine di ogni sottoproblema che ancora bisogna risolvere.
      Inizialmente questo buffer viene inizializzato con l'inizio e la fine dell'array di input.
      Inoltre, questo Buffer avrà dimensione ***2 x 2^h***, *dove* ***h*** è l'altezza dell'Albero Binario di Ricerca (BST) indotto dal QuickSort.

      La spiegazione di questa scelta è presto detta:
      Dalla teoria sappiamo sia che l'altezza di un Albero Binario di Ricerca è ***$\log_2$ (numero_di_elementi)*** sia che il numero massimo di nodi ad un determinato livello dell'Albero è ***$2^{livello\_albero}$***.

      Unendo queste definizioni con l'esigenza di creare una struttura dati che abbia spazio sufficiente per mantenere in memoria tutti i sottoproblemi di un qualunque livello dell'Albero, segue che il buffer sarà dimensionato come ***2 x 2^h***

dove il 2 a moltiplicare viene inserito perché ogni sottoproblema occupa due elementi dell'array, ossia l'inizio e la fine del sottoproblema.

- ***App***, questo è il buffer dove verranno scritti i nuovi sottoproblemi che si generano; lo spazio occupato da questo buffer è ovviamente identico a quello occupato dal buffer ***delimiters***.

- ***Input***, questo è il buffer che conterrà l'array di input inizialmente e l'array ordinato alla fine.

- ***Map***, questa struttura dati permette di associare ogni Work-Item al giusto dato, quest'esigenza nasce dal fatto che i sottoproblemi possono essere sparsi nell'array di input mentre l'indicizzazione dei Work-Item che si lancia è consecutiva, di conseguenza si è sentito il bisogno di creare questa mappa che sancisce in maniera univoca questa associazione.
  Detto ciò, questo buffer verrà dimensionato con le stesse dimensioni dell'array di input.

- ***Smaller_Pivot***, questo buffer serve per separare gli elementi più piccoli o uguali al pivot da quelli più grandi, nello specifico in questo buffer andranno gli elementi più piccoli o uguali del pivot.
  Al livello di spazio occupato, questo buffer verrà dimensionato con le stesse dimensioni dell'array di input e inizializzato con tutti i valori uguali a -1.

- ***Greater_Pivot***, questo buffer serve per separare gli elementi più piccoli o uguali al pivot da quelli più grandi, nello specifico in questo buffer andranno gli elementi più grandi del pivot.
  Inoltre, questo buffer verrà dimensionato con le stesse dimensioni dell'array di input e inizializzato con tutti i valori uguali a -1.

- Una volta create le strutture dati da passare al device, viene eseguito il codice su GPU, che in sostanza effettua la ***partition***; in questo caso il numero di Work-Item lanciati è pari alla dimensione dell'array di ***Input.***

- Esaminando il lato GPU, ogni Work-Item, sfruttando l'array ***Map***, preleva gli estremi (inizio e fine) del sottoproblema che gli è stato assegnato.

- Fatto ciò, ogni Work-Item nomina pivot l'ultimo elemento del proprio sottoproblema.

- A questo punto, ogni Work-Item di ogni sottoproblema preleva ,sfruttando la mappa, l'elemento che gli è stato assegnato e verifica se quest'ultimo è più piccolo o uguale al pivot del sottoproblema.

- Se l'if da esito positivo allora l'elemento viene scritto nell'array ***Smaller_Pivot*** sfruttando lo stesso indice di lettura dell'elemento.

- In maniera speculare, se l'if non va a buon fine avviene lo stesso tipo di scrittura del punto precedente però stavolta nell'array ***Greater_Pivot***.

- Da sottolineare che se un elemento viene inserito nell'array ***Smaller_Pivot*** *,* allora l'array ***Greater_Pivot*** si ritroverà nella stessa posizione un valore uguale a -1 e viceversa; questo comportamento implica la possibile creazione di buchi nei due array, chiaramente quei valori settati a -1 rappresentano dei valori non validi per la logica del programma.

- A questo punto termina l'esecuzione del kernel e si ritorna all'Host.

- Quest'ultimo lancia un ulteriore kernel ,diverso dal primo, con un numero di Work-Item pari al numero di sottoproblemi attivi al lancio del precedente kernel.

- A questo punto, ogni Work-Item si occupa di 1 sottoproblema.

- Nello specifico, ogni Work-Item si occupa di effettuare una ***Stream Compaction*** *del risultato della* **partition***, in sostanza ecco cosa accade in questa fase*:

    - Scrittura in maniera ordinata dei valori validi (diversi da -1) dall'array ***Smaller_Pivot*** all'array di ***Input***, inoltre dopo ogni scrittura sull'***Input*** verrà ripristinato il valore originario (-1) nell'array ***Smaller_Pivot*** per non inficiare le esecuzioni successive dei kernel.

    - Scrittura del Pivot nella posizione corretta nell'Array di ***Input***.

    - Scrittura in maniera ordinata dei valori validi (diversi da -1) dall'array ***Greater_Pivot*** all'array di ***Input***, inoltre dopo ogni scrittura sull'***Input*** verrà ripristinato il valore originario (-1) nell'array ***Greater***_**Pivot** per non inficiare le esecuzioni successive del kernel.

- Fatto ciò ogni Work-Item registra i nuovi sottoproblemi ,creati, sul buffer ***App*** con lo stesso identico approccio dell'***Implementazione 1***.

- A questo punto il flusso di esecuzione torna sull'Host dove parte le funzione ***clean_vector*** che ,usando la stessa identica logica dell'***Implementazione 1***, copia i nuovi sottoproblemi da risolvere ,eliminando i "buchi", nel buffer ***Delimiters*** e restituisce il numero di sottoproblemi da risolvere.

- Successivamente viene chiamata la funzione ***find_next_gws*** che non fa altro che calcolare quanti Work-Item lanciare alla prossima iterazione, questo calcolo viene fatto semplicemente calcolando la dimensionalità di ogni sottoproblema. Un'ulteriore capacità di questa funzione è riempire correttamente il buffer ***Map*** per l'eventuale prossima iterazione.

- Al termine di quest'ultima funzione, avviene un controllo sul numero di sottoproblemi ancora da risolvere.
    - Se non ci sono sottoproblemi da risolvere allora l'ordinamento è completato
    - Altrimenti, si rilancia la ***partition*** e in sostanza si rifanno tutte le procedure elencate sopra.

Prima di analizzare le prestazioni è doveroso fare una precisazione sul perché ,su quest'implementazione, vengono lanciati 2 kernel anziché 1 solo.

Il motivo di questa scelta è che spesso e volentieri ci sono sottoproblemi che non rientrano ,al livello di dimensione, in un singolo Work-Group per motivi hardware; di conseguenza i Work-Item vengono suddivisi in più Work-Group.

A causa di questa dispersione del sottoproblema ,in tanti Work-Group, si perde la possibilità di sincronizzare i Work-Item dello stesso sottoproblema, infatti non esiste in OpenCL una primitiva che sincronizza due o più Work-Group e il motivo per cui non esiste è un motivo di scalabilità su GPU che hanno caratteristiche hardware differenti.

Di conseguenza ,in questo caso, l'unico modo di sincronizzare i Work-Item dello stesso sottoproblema è attendere la fine di esecuzione del kernel e di conseguenza lanciare un ulteriore kernel per "continuare" la computazione.

Detto ciò, le performance di queste implementazione sono mostrate nella tabella sottostante:

Runtime Test 1	Elementi Computati Test 1	Runtime Test 2	Elementi Computati Test 2
16058.1 ms	32.6494 GE/s	10685.1 ms	98.1345 GE/s

A conclusione di questa sezione è doveroso fare un commento su questi risultati.

Come si può intuire dai numeri c'è stato un peggioramento marcato delle prestazioni nonostante lo sfruttamento migliore dell'Hardware.

Il motivo della perdita di performance è presto detto, in generale c'è più lavoro seriale rispetto a prima nonostante il maggior uso dell'Hardware, nello specifico questo lavoro seriale in più è localizzato nel secondo kernel.

Inoltre, questo lavoro seriale porta ad un uso massiccio della Memoria RAM della GPU, infatti ogni Work-Item ,nel primo kernel, deve comunque fare una scrittura in Global Memory anche se il suo elemento è più grande del pivot e sopratutto nel lancio del secondo kernel c'è una pesante fase di ***Stream Compaction*** degli Array ***Smaller e Greater_Pivot*** senza dimenticare che devono essere fatte delle scritture sull'array di ***Input*** per aggiornarlo.

Nello specifico ad ogni lancio di entrambi i kernel vengono fatti i seguenti accessi alla memoria per un generico sottoproblema di dimensionalità ***n***:

- ***n*** letture dall'array di ***Input***
- ***n-1*** scritture distribuite fra gli array ***Smaller e Greater_Pivot***
- ***4(n-1)*** accessi agli array ***Smaller e Greater_Pivot***, questi accessi vengono fatti nella fase di ***Stream Compaction*** ossia nella fase in cui i dati correttamente suddivisi vengono scritti nell'array di input.
- ***n*** scritture sull'array di ***Input,*** queste vengono fatte per aggiornare correttamente l'array di ***Input***.

Come si può intuire da questo resoconto, c'è un pesante fase di accesso in Global Memory che "ammazza" le prestazione di questa coppia di kernel.

Un ulteriore motivo di perdita di performance è l'utilizzo del buffer ***Map*** che oltre che a consumare tempo CPU per essere costruito, aggiunge un'indirettezza negli accessi alla Global Memory; fattore che non aiuta di certo le performance.

Come se non bastasse, c'è ancora un ulteriore fattore frenante per questa implementazione, ovvero il lancio di 2 kernel per risolvere i sottoproblemi di un livello dell'Albero mentre nell'***Implementazione 1*** ne bastava 1.

In generale lanciare tanti kernel impone un overhead dal punto di vista delle performance, sia chiaro l'overhead è insignificante rispetto al salasso di accessi alla Global Memory descritto sopra però comunque c'è e se è possibile bisognerebbe evitarlo.

{IMPLEMENTAZIONE 3 – VERSIONE BASE}

L' ***Implementazione 3*** basa la maggior parte della sua logica sull'***Implementazione 2***.

L'unica differenza è nella strategia di lancio dei kernel, infatti quest'implementazione lancia gli stessi due kernel dell'***Implementazione 2*** quando esiste almeno un sottoproblema che ha cardinalità maggiore del Numero Massimo di Work-Item in un Work-Group per il device corrente.

Quando invece tutti i sottoproblemi hanno cardinalità minore o uguale al Numero Massimo di Work-Item in un Work-Group per il device corrente, allora ogni sottoproblema viene inglobato in un Work-Group e lo stesso identico lavoro che facevano i due kernel dell'***Implementazione 2*** viene fatto fare ad un singolo kernel evitando quando possibile di lanciare 2 kernel per ogni livello dell'albero delle chiamate.

Inoltre ,quando si suddividono i sottoproblemi nei Work-Group, si evita anche di usufruire dell'array ***Map*** a causa della relazione 1:1 fra sottoproblemi e Work-Group, questo porta a velocizzare gli accessi in Global Memory dei Work-Item.

In ultima analisi, i Work-Group lanciati avranno tutti la stessa dimensione, che è uguale alla dimensione del sottoproblema più grande nel livello attuale dell'Albero delle Chiamate; di conseguenza ci saranno alcuni Work-Item in un Work-Group che saranno inoperosi.

Detto ciò, le performance di queste implementazione sono mostrate nella tabella sottostante:

Runtime Test 1	Elementi Computati Test 1	Runtime Test 2	Elementi Computati Test 2
15910 ms	32.9535 GE/s	8821.5 ms	118.866 GE/s

A conclusione di questa sezione è doveroso fare un commento su questi risultati, infatti l'evitato lancio di 2 kernel conferisce un boost di 1 e 2 secondi circa nei Test 1 e 2 rispetto all'***Implementazione 2***, sia chiaro restano tutti i problemi di accesso alla memoria raccontati nell'***Implementazione 2*** ma questo è comunque un miglioramento.

{IMPLEMENTAZIONE 3 – RAFFINAMENTO 1}

Questo è il primo raffinamento dell'***Implementazione 3***, la logica è del tutto identica a quest'ultima tranne che per il numero di elementi che gestisce ogni Work-Item nella fase del programma in cui tutti i sottoproblemi rientrano in ogni Work-Group.

Infatti ,quando siamo in questa fase, ogni Work-Item del Work-Group elabora 4 elementi alla volta anziché 1 solo come nell'***Implementazione 3 Base***.

Detto ciò, le performance di queste implementazione sono mostrate nella tabella sottostante:

Runtime Test 1	Elementi Computati Test 1	Runtime Test 2	Elementi Computati Test 2
15895.1 ms	32.9842 GE/s	8810.31 ms	119.017 GE/s

A conclusione di questa sezione è doveroso fare un commento su questi risultati, qua il miglioramento rispetto alla versione base c'è stato ma in sostanza è stato impercettibile, questo perché continuano a pesare di più l'enorme mole di accessi in Global Memory anche perché questo raffinamento entra in gioco solo nelle parti conclusive della computazione.

{IMPLEMENTAZIONE 3 – RAFFINAMENTO 2}

Questo è un ulteriore raffinamento dell'***Implementazione 3,*** anche qua la logica rimane quasi invariata.

L'unica novità è che quando è possibile assegnare ogni sottoproblema ad un Work-Group viene usata la Local Memory per smistare gli elementi dei sottoproblemi.

Nello specifico si sfruttano 2 array in Local Memory che in sostanza fanno il lavoro che facevano gli array ***Smaller e Greater Pivot*** nelle precedenti implementazioni col vantaggio di avere una memoria meno costosa dal punto di vista degli accessi rispetto alla Global Memory.

Detto ciò, le performance di queste implementazione sono mostrate nella tabella sottostante:

Runtime Test 1	Elementi Computati Test 1	Runtime Test 2	Elementi Computati Test 2
15923.1 ms	32.9263 GE/s	8823.88 ms	118.834 GE/s

La conclusione di questa sezione è analoga alla precedente, addirittura qua i risultati sono leggermente peggiori dell'***Implementazione 3 Base*** ma resta il fatto che il principale collo di bottiglia di queste prime Implementazioni (eccetto la 1) è il massivo accesso in Global Memory.

{IMPLEMENTAZIONE 4 BASE}

L' ***Implementazione 4*** basa quasi del tutto la sua logica sull'***Implementazione 3***.

L'unica differenza è l'eliminazione dell'array ***Map*****,** infatti eliminando il suddetto array si elimina un'indirettezza in molti accessi alla Global Memory nella fase in cui si lanciano 2 kernel per ogni livello dell'albero.

Per attuare questa rimozione, nella fase dei lanci dei due kernel, si lancia un numero di Work-Item pari al numero di elementi dell'array di ***Input***.

Questa strategia porta ad avere sempre più Work-Item inoperosi al progredire della computazione.

Detto ciò, le performance di queste implementazione sono mostrate nella tabella sottostante:

Runtime Test 1	Elementi Computati Test 1	Runtime Test 2	Elementi Computati Test 2
3467.77 ms	151.189 GE/s	7856.2 ms	133.471 GE/s

A conclusione di questa sezione è doveroso fare un commento su questi risultati, in sostanza il miglioramento nell'accesso alla Global Memory è stato sensibile, infatti si sono guadagnati parecchi secondi rispetto alle Implementazioni precedenti.

{IMPLEMENTAZIONE 4 – RAFFINAMENTO 1}

Questa Implementazione prende la maggior parte della logica dell'***Implemntazione 3 – Raffinamento 1***, in più gli aggiunge la rimozione dell'array ***Map*** avvenuta nell'***Implementazione 4 Base***.

Detto ciò, le performance di queste implementazione sono mostrate nella tabella sottostante:

Runtime Test 1	Elementi Computati Test 1	Runtime Test 2	Elementi Computati Test 2
3479.72 ms	150.67 GE/s	7869.52 ms	133.245 GE/s

A conclusione di questa sezione è doveroso fare un commento su questi risultati, in sostanza le performance sono restate tutto sommato identiche all'***Implementazione 4 Base***, questo a testimonianza che usare una pseudo-vettorializzazione (il fatto che gli elementi sono presi a 4 a 4) nella parte finale della computazione aiuta poco rispetto agli accessi massicci in Global Memory che si fanno all'inizio.

{IMPLEMENTAZIONE 4 – RAFFINAMENTO 2}

L' ***Implementazione 4 – Raffinamento 2*** basa la maggior parte della sua logica sull'***Implementazione 3 – Raffinamento 2***.

L'unica differenza è che ,come l'***Implementazione 4 Base***, viene eliminato l'utilizzo dell'array ***Map*** quando si è nella fase in cui si lanciano 2 kernel per ogni livello dell'albero delle chiamate.

Detto ciò, le performance di queste implementazione sono mostrate nella tabella sottostante:

Runtime Test 1	Elementi Computati Test 1	Runtime Test 2	Elementi Computati Test 2
3481.22 ms	150.605 GE/s	7885.14 ms	132.981 GE/s

A conclusione di questa sezione è doveroso fare un commento su questi risultati, in sostanza non c'è stato miglioramento nell'usare la Local Memory rispetto

all'***Implementazione 4 Base***, e i motivi sono del tutto simili al raffinamento 1 di quest'ultima.

{IMPLEMENTAZIONE 5 – VERSIONE BASE}

L'***Implementazione 5*** cambia totalmente approccio rispetto al passato e cerca di trovare una buona soluzione per rimuovere l'onerosa fase di Stream Compaction che abbiamo avuto fin'ora.

Per la stesura di quest'implementazione si è preso spunto dal paper GPU-QuickSort di *Tsigas et al. [2]*.

Di seguito è riportata la spiegazione dettagliata dell'Implementazione:

- Inizialmente l'Host organizza le strutture dati da passare al device: nello specifico l'Host crea le seguenti strutture dati per il device:

  - ***Delimiters***, questa struttura dati conterrà una lista contigua dell'inizio e della fine di ogni sottoproblema che ancora bisogna risolvere.
    Inizialmente questo buffer viene inizializzato con l'inizio e la fine dell'array di input.
    Inoltre, questo Buffer avrà dimensione 2×2^h, *dove* ***h*** è l'altezza dell'Albero Binario di Ricerca (BST) indotto dal QuickSort.

    La spiegazione di questa scelta è presto detta:
    Dalla teoria sappiamo sia che l'altezza di un Albero Binario di Ricerca è $log_2(numero\_di\_elementi)$ sia che il numero massimo di nodi ad un determinato livello dell'Albero è $2^{livello\_albero}$.

    Unendo queste definizioni con l'esigenza di creare una struttura dati che abbia spazio sufficiente per mantenere in memoria tutti i sottoproblemi di un qualunque livello dell'Albero, segue che il buffer sarà dimensionato come 2×2^h dove il 2 a moltiplicare viene inserito perché ogni sottoproblema occupa due elementi dell'array, ossia l'inizio e la fine del sottoproblema.

  - ***App***, questo è il buffer dove verranno scritti i nuovi sottoproblemi che si generano; lo spazio occupato da questo buffer è ovviamente identico a quello occupato dal buffer ***delimiters***.

  - ***Input***, questo è il buffer che conterrà l'array di input inizialmente e l'array ordinato alla fine.

  - ***Map***, questa struttura dati permette di associare ogni Work-Item al giusto dato, quest'esigenza nasce dal fatto che i sottoproblemi possono essere sparsi nell'array di input mentre l'indicizzazione dei Work-Item che si lancia è consecutiva, di conseguenza si è sentito il bisogno di creare questa mappa che sancisce in maniera univoca questa associazione.

Detto ciò, questo buffer verrà dimensionato con le stesse dimensioni dell'array di input.

  - ***Pivots***, questo buffer contiene la lista dei pivot relativi all'ultima partition effettuata, quest'ultima sarà indispensabile per aggiornare l'array ***Delimiters*** con i nuovi sottoproblemi creati.

  - ***Input_App***, questa nuova implementazione non è in loco come le altre e di conseguenza questo buffer rappresenta un array di appoggio per le computazioni.

- Una volta create le strutture dati da passare al device, viene eseguito il codice su GPU, che in sostanza effettua la ***partition***; inoltre il numero di Work-Item lanciati è pari al numero di elementi da ordinare.

- Esaminando il lato GPU, ogni Work-Item, sfruttando l'array ***Map***, preleva il proprio sottoproblema (indice di inizio e di fine del sottoproblema).

- Fatto ciò, ogni Work-Item nomina pivot l'ultimo elemento del proprio sottoproblema.

- A questo punto, ogni Work-Item conta ,all'interno del suo sottoproblema, quanti elementi vogliono scrivere alla sua sinistra se è più piccolo o uguale al pivot, altrimenti conta quanti elementi vogliono scrivere alla sua destra.

- A questo punto ogni Work-Item conosce la posizione corretta di scrittura e per tanto può scrivere serenamente il suo valore nell'array di appoggio ***Input_App*** *senza* avere nessun tipo di problema di concorrenza.

- Inoltre, il Work-Item che ha assegnato il pivot scrive anche l'indice corretto di quest'ultimo nell'array ***Pivots***, per accedere al suddetto array viene usato l'indice globale del Work-Item.

- A questo punto termina l'esecuzione del kernel e si ritorna all'Host.

- Quest'ultimo copia il buffer di appoggio nell'array di ***Input*** e contestualmente lancia un ulteriore kernel ,diverso dal primo, con un numero di Work-Item pari al numero di sottoproblemi attivi al lancio del precedente kernel.

- A questo punto, ogni Work-Item si occupa di 1 sottoproblema.

- Fatto ciò ogni Work-Item registra i nuovi sottoproblemi creati sul buffer ***App*** con lo stesso identico approccio dell'***Implementazione 1***, l'unica differenza è che gli indici dei pivot vengono prelevati dall'omonimo array.

- A questo punto il flusso di esecuzione torna sull'Host dove parte le funzione ***clean_vector*** che ,usando la stessa identica logica dell'***Implementazione 1***, copia i nuovi sottoproblemi da risolvere ,eliminando i "buchi", nel buffer ***Delimiters*** e restituisce il numero di sottoproblemi da risolvere.

- Successivamente viene chiamata la funzione ***find_next_gws*** che non fa altro che calcolare quanti Work-Item lanciare alla prossima iterazione, questo calcolo viene fatto semplicemente calcolando la dimensionalità di ogni sottoproblema.
  Un'ulteriore capacità di questa funzione è riempire correttamente il buffer ***Map*** per l'eventuale prossima iterazione.

- Da premettere che quando si verifica la condizione che ogni sottoproblema rientra ,dal punto di vista della dimensionalità, in un singolo Work-Group allora ,come nell'***Implementazione 3***, verrà effettuata una singola chiamata di kernel per ogni livello dell'Albero e di conseguenza verrà eliminato dalla computazione l'array ***Map*** andando ad incrementare le performance di accesso alla memoria.
  In ultima analisi, la copia del buffer di appoggio ***Input_App*** nel buffer originario ***Input*** rimane ugualmente.

- Al termine di quest'ultima funzione, avviene un controllo sul numero di sottoproblemi ancora da risolvere.
  - Se non ci sono sottoproblemi da risolvere allora l'ordinamento è completato
  - Altrimenti, si rilancia la ***partition*** e in sostanza si rifanno tutte le procedure elencate sopra.

Detto ciò, le performance di queste implementazione sono mostrate nella tabella sottostante:

Runtime Test 1	Elementi Computati Test 1	Runtime Test 2	Elementi Computati Test 2
59270.9 ms	8.84562 GE/s	80374.3 ms	13.0462 GE/s

A conclusione di questa sezione è doveroso fare un commento su questi risultati.

Come si può intuire dai numeri c'è stato un peggioramento marcato delle prestazioni anche rispetto alle implementazioni che usano Stream Compaction.

Questo perché ,banalmente, sono aumentati ancora di più gli accessi alla Global Memory.

Infatti ogni Work-Item ,all'interno del singolo sottoproblema, deve leggersi tutti gli elementi che lo precedono se è più piccolo o uguale al pivot oppure tutti gli elementi che lo seguono se è più grande, oltre a ciò bisogna fare anche la scrittura finale nell'***Input_App*** che è allocato sempre in Global Memory.

{IMPLEMENTAZIONE 5 – RAFFINAMENTO 1}

Questa implementazione è un leggero raffinamento dell'***Implementazione 5***.

In sostanza ,sulla falsa riga dell'***Implementazione 3 – Raffinamento 2***, si è spostato il calcolo dalla Global Memory alla Local Memory se e solamente se ogni sottoproblema da risolvere riesce ad essere contenuto in un Work-Group.

Detto ciò, le performance di queste implementazione sono mostrate nella tabella sottostante:

Runtime Test 1	Elementi Computati Test 1	Runtime Test 2	Elementi Computati Test 2
56253.8 ms	9.32004 GE/s	79236.9 ms	13.2334 GE/s

A conclusione di questa sezione è doveroso fare un commento su questi risultati, in sostanza il miglioramento c'è stato rispetto alla versione base (1 secondo circa nel Test 2), resta il fatto che le prestazioni non sono migliorate in maniera netta; anche qua a causa dei numerosi accessi in Global Memory.

{IMPLEMENTAZIONE 6}

Questa implementazione basa la maggior parte della sua logica sull'***Implementazione 5***.

L'unica differenza è che viene abolito l'utilizzo dell'array ***Map*** con l'obbiettivo di velocizzare gli accessi in Global Memory.

La logica di rimozione dell'array ***Map*** è identica all'***Implementazione 4 Base***.

Detto ciò, le performance di queste implementazione sono mostrate nella tabella sottostante:

Runtime Test 1	Elementi Computati Test 1	Runtime Test 2	Elementi Computati Test 2
5600.62 ms	93.6124 GE/s	15429.2 ms	67.9605 GE/s

A conclusione di questa sezione è doveroso fare un commento su questi risultati, in sostanza ,come negli altri casi in cui si è rimossa la Map, il miglioramento è stato prorompente; ciò non toglie che restano comunque troppi accessi in Global Memory dato che le prestazioni sono ancora scadenti e in ogni caso lontane dal risultato raggiunto dall'***Implementazione 1***.

{IMPLEMENTAZIONE 7}

L' ***Implementazione 7*** basa la maggior parte della sua logica sull'***Implementazione 6***.

L'unica differenza è che si lancia sempre 1 solo kernel per ogni livello dell'Albero delle Chiamate.

Per ottenere questo risultato è stato spostato il calcolo dell'aggiornamento dei sottoproblemi su host.

Detto ciò, le performance di queste implementazione sono mostrate nella tabella sottostante:

Runtime Test 1	Elementi Computati Test 1	Runtime Test 2	Elementi Computati Test 2
7041.35 ms	74.4584 GE/s	21129.8 ms	49.6256 GE/s

A conclusione di questa sezione è doveroso fare un commento su questi risultati, in sostanza quest'ultima idea non ha portato un miglioramento.

Nonostante ciò quest'ultimo approccio ci conferma ,quanto meno, che il secondo lancio di kernel è stato davvero necessario fin'ora.

{IMPLEMENTAZIONE 8}

Il responso principale di quasi tutte le implementazioni è stato l'eccessivo uso della Global Memory e di conseguenza le prestazione sono state più o meno sempre lontane dall'***Implementazione 1*** che ,nonostante sfrutti meno l'hardware, resta attualmente l'implementazione più performante del progetto fino a questo momento.

Cercando di tenere a mente questo discorso, si è pensato di studiare una soluzione che cerchi il più possibile di bilanciare accessi alla memoria, numero di lanci di kernel e sfruttamento dell'Hardware col solito obbiettivo di minimizzare i tempi di esecuzione.

Dal punto di vista della programmazione è stato decisivo il salto di paradigma dal punto di vista del parallelismo, infatti qua si è adottata una strategia che è una via di mezzo fra il solo Work-Item per sottoproblema dell'***Implementazione 1*** e i numerosissimi Work-Item lanciati nelle altre implementazioni.

Questa strategia è nota come ***Sliding Window***, e in senso generale ha le seguenti caratteristiche:

(1) Si applica un insieme di thread di esecuzione ad una porzione della struttura dati da elaborare
(2) Ogni thread effettua la sua computazione
(3) Opzionalmente, viene imposta una barriera ovvero si sincronizzano tutti i thread allo stesso punto.
(4) A questo punto ogni thread sposta la sua applicazione alla destra del punto di applicazione corrente, lo scostamento (o offset) è tanto grande quanto la numerosità del gruppo di thread; in sostanza in questo modo l'insieme dei thread scorre attraverso la struttura dati, da qui il nome ***Sliding Window***.
(5) Fatto ciò, si ritorna al punto 2 tranne che non siano terminati i dati contenuti nella struttura da elaborare.

Dal punto di vista del lancio del kernel, in questa implementazione vengono lanciati un numero di Work-Group pari al numero di sottoproblemi da risolvere; inoltre ogni Work-Group ha la stessa dimensionalità della dimensione della finestra che l'utente dovrà passare in input al programma.

Da queste premesse è nata l'ottava implementazione di questo progetto che è raccontata nel dettaglio in seguito:

- Inizialmente l'Host organizza le strutture dati da passare al device: nello specifico l'Host ,dopo aver prelevato la dimensione della finestra, crea le seguenti strutture dati per il device:

  - ***Delimiters***, questa struttura dati conterrà una lista contigua dell'inizio e della fine di ogni sottoproblema che ancora bisogna risolvere.
    Inizialmente questo buffer viene inizializzato con l'inizio e la fine dell'array di input.
    Inoltre, questo Buffer avrà dimensione 2×2^h, *dove* ***h*** è l'altezza dell'Albero Binario di Ricerca (BST) indotto dal QuickSort.

    La spiegazione di questa scelta è presto detta:
    Dalla teoria sappiamo sia che l'altezza di un Albero Binario di Ricerca è $log_2(numero\_di\_elementi)$ sia che il numero massimo di nodi ad un determinato livello dell'Albero è $2^{livello\_albero}$.

    Unendo queste definizioni con l'esigenza di creare una struttura dati che abbia spazio sufficiente per mantenere in memoria tutti i sottoproblemi di un qualunque livello dell'Albero, segue che il buffer sarà dimensionato come 2×2^h dove il 2 a moltiplicare viene inserito perché ogni sottoproblema occupa due elementi dell'array, ossia l'inizio e la fine del sottoproblema.

  - ***App***, questo è il buffer dove verranno scritti i nuovi sottoproblemi che si generano; lo spazio occupato da questo buffer è ovviamente identico a quello occupato dal buffer ***delimiters***.

- ***Input***, questo è il buffer che conterrà l'array di input inizialmente e l'array ordinato alla fine.

- ***Cache,*** questa struttura dati viene allocata in Local Memory a differenza delle altre; inoltre ha dimensione pari alla dimensione della finestra che per inciso è la stessa dimensione dei Work-Group.
  Questa struttura dati è necessaria per permettere al Work-Group di capire quali elementi ,fra quelli analizzati correntemente, devono essere scambiati.

- ***Scan_Cache,*** anche questa struttura dati è allocata in Local Memory e ha dimensione pari al Work-Group.
  L'Utilità di questa struttura dati è conservare lo Scan dell'array ***Cache***.

- **Ex_Index**, questo array ,allocato in Local Memory, ha un solo valore e solitamente è impostato ad un valore che inizialmente è -1 e che nel corso dell'esecuzione del Work-Group viene incrementato con lo scan effettuato dall'ultimo Work-Item del Work-Group sull'array **C*ache.***
  In sostanza è grazie a questo indice che il Work-Group può calcolare quali scambi deve fare in parallelo.

- Una volta create le strutture dati da passare al device, viene eseguito il codice su GPU, che in sostanza effettua la ***partition*****.**

- Esaminando il lato GPU, ogni Work-Item di ogni Work-Group ,sfruttando l'indicizzazione di quest'ultimo, preleva gli estremi del suo sottoproblema (inizio e fine).

- Fatto ciò, ogni Work-Item nomina pivot l'ultimo elemento del sottoproblema.

- A questo punto, il Work-Group controlla se la sua dimensione è più grande del sottoproblema da risolvere, in tal caso non ci sarà bisogno di "far scorrere il Work-Group attraverso il sottoproblema"; va da se che ,in questo caso, i Work-Item del Work-Group in eccesso ,rispetto alla numerosità del sottoproblema, resteranno inoperosi.

- Dopo questa operazione, il Work-Group calcola quanti "scorrimenti deve fare", ovviamente questa divisione può dare resto e in quel caso verrebbero effettuati un numero di scorrimenti uguale alla parte intera della divisione più il resto.

- Fatto ciò il Work-Group entra in un ciclo che in sostanza permette al Work-Group di scorrere attraverso la struttura dati:

  - All'interno del ciclo:
    - Il Work-Item numero 1 del Work-Group aggiorna l'***Ex_Index*** sommandogli l'ultimo scan contenuto nell'array **scan_cache** se c'è già stato almeno uno scorrimento altrimenti lo imposta a 1.
      Da precisare che i Work-Item diversi dal numero 1 possono tranquillamente eseguire il punto successivo mentre il Work-Item 1 aggiorna l'***Ex_Index***.

- Ogni Work-Item del Work-Group inserisce 1 nell'array ***Cache*** ,usando il suo indice locale come pattern di accesso, se l'elemento associato al Work-Item è più piccolo (o uguale) del pivot, altrimenti inserisce uno 0 nell'array ***Cache***.
- Fatto ciò, interviene una barriera che sincronizza i Work-Item del Work-Group e in sostanza garantisce che tutte le scritture sull'array **Cache** siano terminate.
- Dopo la barriera ogni Work-Item del Work-Group esegue lo scan inclusivo dell'array ***Cache*** usando come indice di fine dello scan il loro indice locale rispettivamente.
- Finito lo scan, ogni Work-Item scrive il suo risultato nell'array ***Scan_Cache*** all'indice uguale al proprio indice locale.
- Fatto ciò avviene una barriera per assicurarsi che tutti gli scan siano terminati.
- Dopo questa barriera, il Work-Group sa esattamente quali scambi deve fare. Al di là di questo, le scritture sull'array ***Input*** vengono eseguite solamente dal Work-Item 1 mentre gli altri restano inoperosi.
  Nello specifico il Work-Item 1 scambia tutti gli elementi ,controllati attualmente dal Work-Group, che hanno il corrispettivo valore nell'array ***Cache*** impostato ad uno, più precisamente lo scambio viene effettuato con l'elemento alla posizione seguente:

  “””
  inizio_sottoproblema+ex_index+scan_relativo_all'i-esimo_elemento_analizzato_dal_Work_Group

  “””

- Dopo quest'operazione c'è un'ulteriore barriera che assicura che il Work-Item 1 abbia terminato le sue scritture, la presenza di questa barriera è necessaria alla corretta esecuzione delle scritture; questo perché il Work-Item uno necessità di leggere l'array ***Cache*** per capire se deve effettuare lo scambio, array che potrebbe venire alterato se non ci fosse la barriera.

- All'uscita dal ciclo, il Work-Group controlla se resta ancora uno scorrimento da fare, in tal caso lo effettua e applica la stessa logica usata nel ciclo descritto sopra; l'unica differenza è che quel procedimento viene eseguito una volta sola dato che si tratta dell'ultimo scorrimento.
  Chiaramente in questa fase avremo Work-Item della finestra inoperosi a causa della divisione con resto raccontata all'inizio.

- A questo punto la fase di partition è ultimata e solamente il primo Work-Item del Work-Group aggiorna l'array ***Delimiters*** con i nuovi sottoproblemi creati sfruttando quasi la stessa logica dell'***Implementazione 1,*** l'unica differenza è il calcolo dell'indice del pivot che comporta un ulteriore aggiornamento dell'indice ***Ex_Index*** con l'ultimo scan disponibile.

- A questo punto il flusso di esecuzione torna sull'Host dove parte le funzione ***clean_vector*** che ,usando la stessa identica logica dell***'Implementazione 1***, copia i nuovi sottoproblemi da risolvere ,eliminando i “buchi”, nel buffer ***Delimiters*** e restituisce il numero di sottoproblemi da risolvere.

- Al termine di quest'ultima funzione, avviene un controllo sul numero di sottoproblemi ancora da risolvere.
  - Se non ci sono sottoproblemi da risolvere allora l'ordinamento è completato
  - Altrimenti, si rilancia la ***partition*** e in sostanza si rifanno tutte le procedure elencate sopra.

Prima di analizzare le prestazioni è doverosa una delucidazione in merito al pattern di scrittura dei valori corretti sull'array di ***Input*** da parte dei Work-Item appartenenti ad un Work-Group.

Come si è potuto apprezzare dalla descrizione di sopra, questa scrittura viene fatta serialmente ed esiste un preciso motivo per questa scelta.

Il motivo è presto detto, la scrittura viene fatta serialmente per evitare scritture da parte di più Work-Item sulla stessa locazione di memoria.

Nello specifico esiste un modo in cui si possono susseguire gli scambi che inibisce l'idea di effettuare la scrittura parallela dei risultati, pena un risultato incoerente dell'ordinamento.

Entrando più nel dettaglio il problema si verifica nella seguente situazione:

- C'è già stato uno scorrimento
- L'indice ***Ex_Index*** è uguale:
  ➔ ""

  all'indice dell'elemento assegnato al primo Work-Item ,della finestra attuale, – lunghezza_finestra

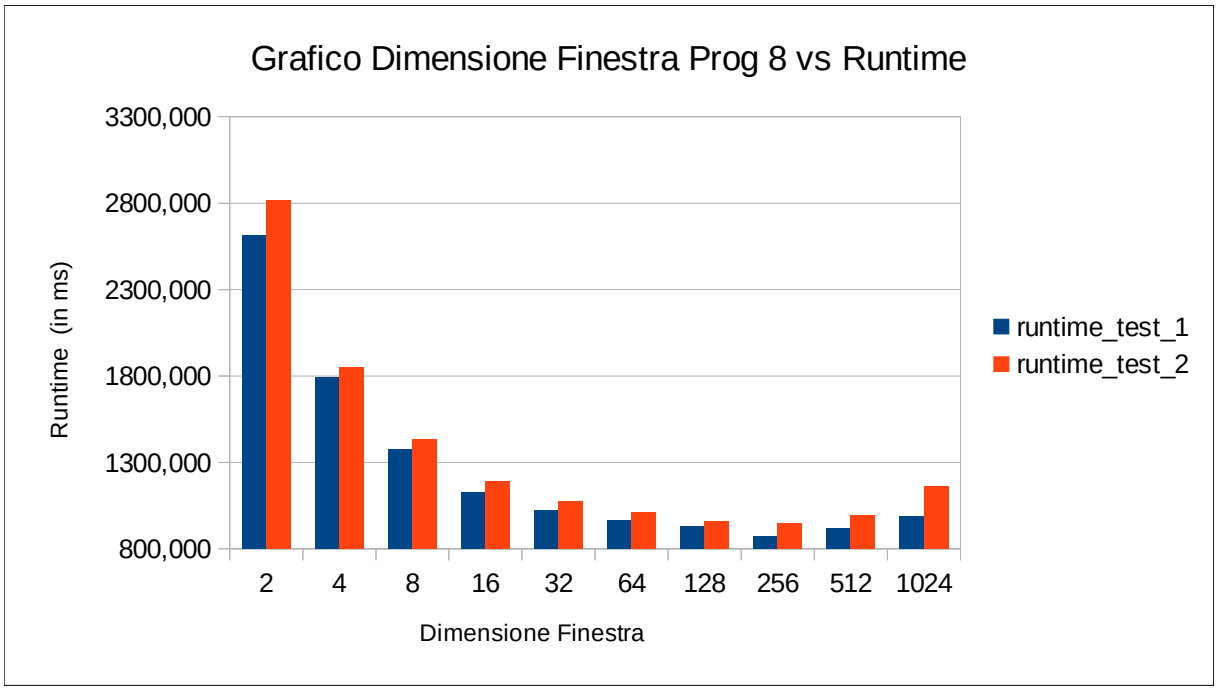
  ""

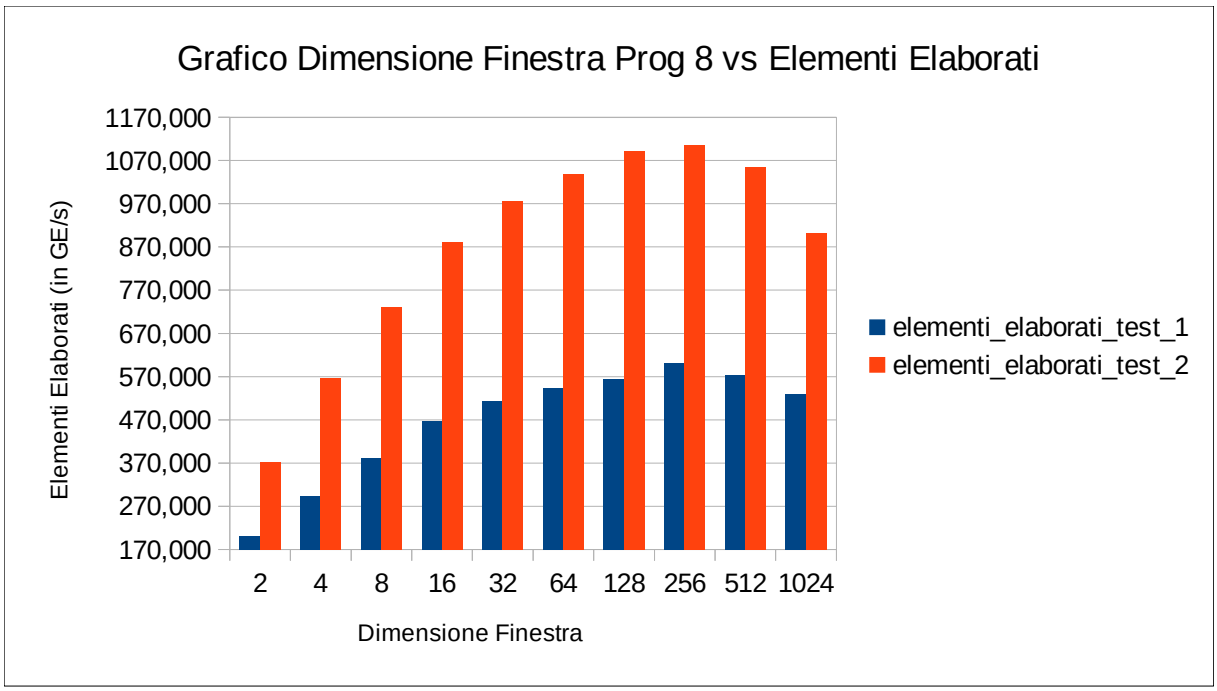
- Tutti gli elementi controllati dal Work-Group devono scambiare:
  ➔ Segue che il primo elemento controllato dal Work-Group deve essere scambiato con l'elemento antecedente mentre contemporaneamente il secondo elemento ,controllato dal Work-Group, deve essere scambiare col primo ecc……..
  Questo computazione porta ad un risultato impredicibile dato che ci sono più Work-Item che vogliono scrivere sulla stessa locazione di memoria.

Detto ciò, le performance di queste implementazione sono mostrate nella tabella sottostante:

Runtime Test 1 (Sliding Window = 256)	**Elementi Computati Test 1** (Sliding Window = 256)	**Runtime Test 2** (Sliding Window = 256)	**Elementi Computati Test 2** (Sliding Window = 256)
871.048 ms	601.905 GE/s	948.078 ms	1106 GE/s

Prima di passare al commento delle prestazioni è importante mostrare come variano le performance e gli elementi computati al variare della grandezza della finestra:





Come si può apprezzare dai grafici, la scelta migliore in merito alla dimensione della finestra è stata 256 dato che garantisce il miglior rapporto fra numero degli scorrimenti del Work-Group e numero degli scan da effettuare.
Andando ad analizzare le prestazioni di quest'implementazione possiamo dire ,numeri alla mano, che è nettamente la migliore di tutto il progetto.

Questo perché riesce a coniugare bene il parallelismo con un uso parsimonioso della Global Memory della GPU.

{IMPLEMENTAZIONE 9}

L'Implementazione 9 è in gran parte basata sull'Implementazione 2.

L'unica differenza è la fase di Stream Compaction, infatti tale fase viene svolta in parallelo a differenza del passato.

Di seguito vengono elencati i punti salienti della Stream Compaction Parallela:

- Al livello di lancio di kernel, vengono lanciati tanti Work-Group quanti sono i sottoproblemi ancora da risolvere.

- La dimensione dei Work-Group è dettata dalla Dimensione della Finestra che passerà l'utente in Input, di conseguenza quest'implementazione va a sfruttare l'approccio Sliding Window discusso nell'***Implementazione 8***.

- L'idea di base di questa Implementazione è lo scorrimento "continuo" degli array ***Smaller e Greater Pivot*** e lo scorrimento "quando è necessario" dell'array di ***Input***; chiaramente questa frase verrà "snocciolata" in seguito.

- Viene sfruttato un array ***Cache*** allocato in Local Memory, la sua dimensione è uguale a quella del Work-Group; questo array "segnalerà" al Work-Group dove sono posizionati gli elementi inconsistenti (o buchi) negli array ***Smaller e Greater Pivot***.

- Viene sfruttato un array ***Scan_Cache*** allocato in Local Memory, la sua dimensione è uguale a quella del Work-Group; questo array conterrà lo scan attuale dell'array **Cache**.

Quello appena descritto è un sommario dell'Implementazione 9, ora andiamo più in profondità nella descrizione:

- Dopo il lancio del kernel, Il Work-Group acquisisce gli estremi del proprio sottoproblema (inizio e fine), archivia l'inizio del sottoproblema nella variabile ***Inizio_Sottoproblema_Fittizio*** e calcola quante volte deve far scorrere la finestra sugli array ***Smaller e Greater Pivot***, questo calcolo è una divisione e di conseguenza può dare resto; in tal caso verrà effettuato un ultimo scorrimento per coprire i restanti **"(Cardinalità_Sotto_Problema-1) *mod* Dimensione_Finestra"** .

Il meno 1 è dovuto al fatto che, nell'insieme ***Smaller o Greater_Pivot*** ci possono stare ,potenzialmente, tutti gli elementi del sottoproblema – 1, che sarebbe il pivot.

- Detto ciò, ogni Work-Item del Work-Group entra nel ciclo degli scorrimenti dell'array ***Smaller_Pivot*** e analizza l'elemento che gli è stato assegnato all'interno dell'attuale scorrimento:

  - Se l'elemento è stato valorizzato (diverso da "-1") il Work-Item inserirà "0" nell'array ***Cache,*** nello specifico il Work-Item userà il suo indice locale come pattern di accesso alla struttura dati ***Cache***.
  - Altrimenti, il Work-Item inserirà "1" nell'array ***Cache*** con lo stesso pattern di accesso del punto precedente.

- A questo punto, interviene una barriera atta a sincronizzare le scritture del Work-Group sull'array ***Cache***.

- Al termine della barriera, ogni Work-Item del Work-Group fa uno scan inclusivo dell'array ***Cache*** sfruttando ,come "indice dello scan", il suo indice locale; il risultato dello scan verrà archiviato nell'array ***Scan_Cache*** all'indice locale del Work-Item.

- Successivamente interviene una barriera atta a sincronizzare gli scan.

- Fatto ciò, ogni Work-Item che possiede ,al proprio indice locale sull'array ***Cache***, uno "0" andrà a scrivere l'elemento attualmente controllato ,sull'array ***Smaller_Pivot,*** nell'array di ***Input***; l'indirizzo di scrittura sull'***Input*** è il seguente:

  - **Inizio_Sottoproblema_Fittizzio + Indice_Locale_Work_Item – Numero_Buchi_Prima_del_Work_Item_Attuale**

- A questo punto ogni Work-Item si calcola quante scritture ha effettuato il Work-Group all'interno dell'attuale scorrimento della finestra (il calcolo è effettuato sottraendo il numero di buchi dell'attuale scorrimento alla dimensione della finestra) e:

  - Sostituisce il valore ***Inizio_Sottoproblema_Fittizio*** con se stesso incrementato del ***Numero_di_Scritture_Scorrimento_Attuale***.
  - Con questo stratagemma è possibile spostata l'area di scrittura (o griglia) sull'***Input*** solamente del numero di valori realmente scritti su di esso; di conseguenza se in uno scorrimento dell'array ***Smaller_Pivot*** non ci fossero scritture da effettuare, segue che l'area di scrittura sull'***Input*** non subirebbe spostamenti.

- Arrivati a questo punto si cicla e ogni Work-Item del Work-Group fa scorrere il proprio punto di applicazione ,sull'array ***Smaller_Pivot***, della dimensione della finestra (che è uguale a quella del Work-Group chiaramente).

- Fatto ciò, il Work-Group ripete tutte le operazioni descritte sopra finché non ci sono più le condizioni per ciclare.

- Al termine del ciclo, gli eventuali ,restanti, elementi da analizzare (possono restare elementi in virtù della divisione fra la dimensione del sottoproblema e la dimensione della finestra) vengono smistati nell'array di ***Input*** con le stesse identiche modalità che si sono viste nel ciclo descritto precedentemente, l'unica differenza sta nel calcolo del numero di scritture effettuate attualmente dal Work-Group; infatti qua la sottrazione va fatta tra il resto della divisione fra ***Dimensione Sottoproblema e Dimensione Finestra*** e il numero di buchi presenti nell'attuale scorrimento; da precisare che quest'ultima informazione viene sempre prelevata dall'array ***Scan_Cache***, qua la differenza fra la parte attuale e quella col ciclo è che:

    - Nella parte "con ciclo", il numero di buchi attuali viene prelevato dall'ultimo elemento dell'array ***Scan_Cache***
    - Nella parte "del resto", il numero di buchi attuali viene prelevato dall'elemento di posto "***resto della divisione fra Dimensione Sottoproblema e Dimensione Finestra".***

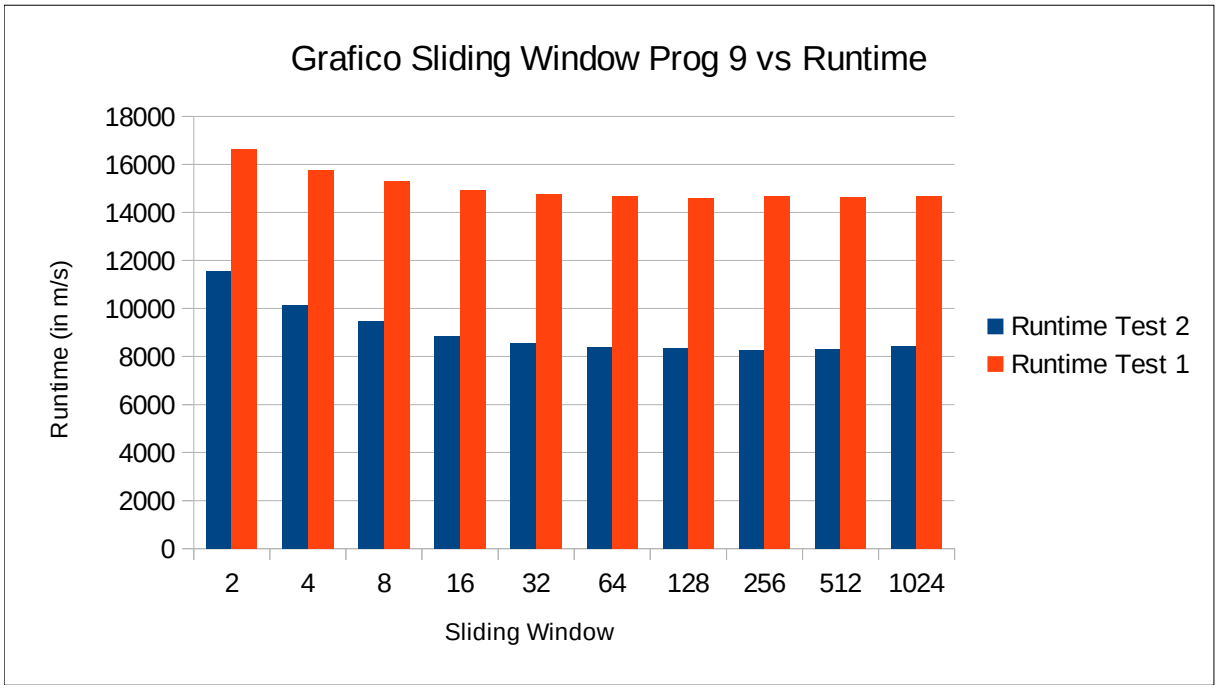
- Arrivati a questo punto, la variabile "***Inizio_Sottoproblema_Fittizio"*** conterrà il numero di scritture totali effettuate scorrendo tutto l'array ***Smaller_Pivot***, di conseguenza questo valore sarà uguale al corretto indice dove scrivere il pivot; infatti il passo successivo dell'esecuzione del programma è la scrittura del pivot ,sull'array di ***Input***, all'indirizzo "***Inizio_Sottoproblema_Fittizio"***. Questa scrittura viene fatta fare ad un solo Work-Item dato che si tratta di 1 solo elemento.

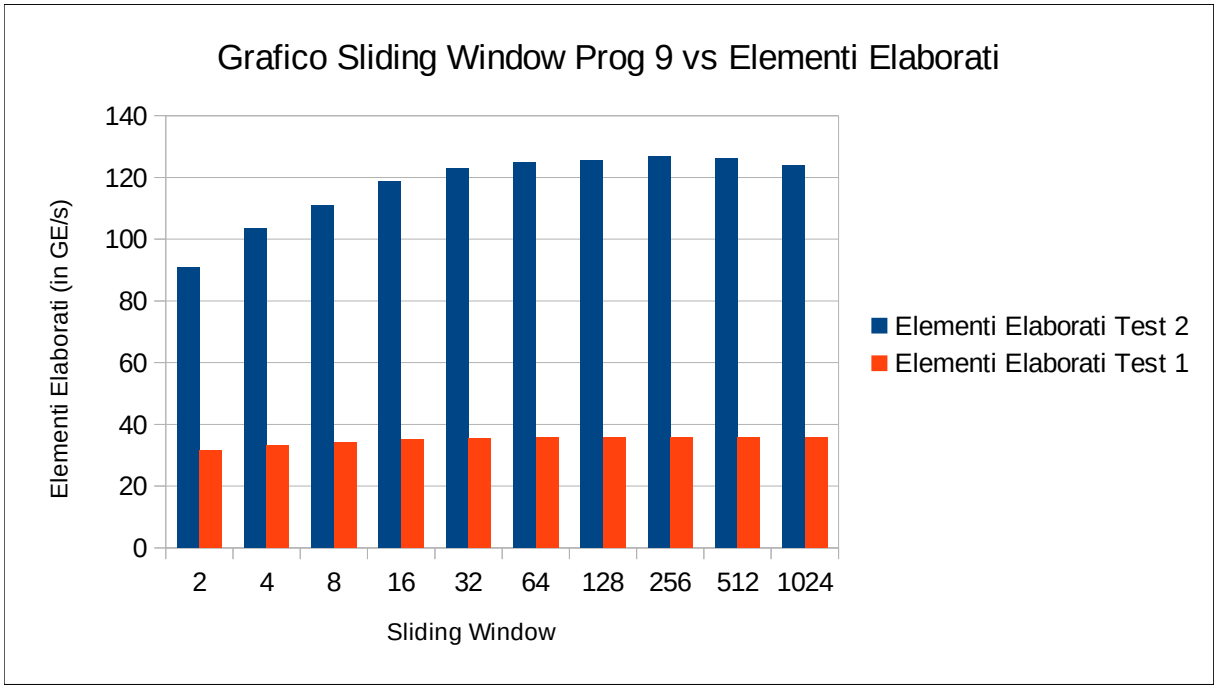
- A questo punto, vengono replicate le stesse identiche operazioni effettuate prima, l'unica differenza è che stavolta andrà "spazzolato" l'array ***Greater_Pivot***.

- Dopo quest'ultima fase, vengono scritti i nuovi sottoproblemi con la stessa logica dell'***Implementazione 2***, nello specifico questa scrittura viene fatta fare ad un solo Work-Item per ogni sottoproblema.

Runtime Test 1 (Sliding-W = 256)	**Elementi Computati Test 1 (Sliding-W = 256)**	**Runtime Test 2 (Sliding-W = 256)**	**Elementi Computati Test 2 (Sliding-W = 256)**
14751 ms	35,5425 GE/s	8307,96 ms	126,213 GE/s

Prima di chiudere questa sezione, è corretto mostrare le differenze di performance del programma al variare della dimensione della Sliding-Window:





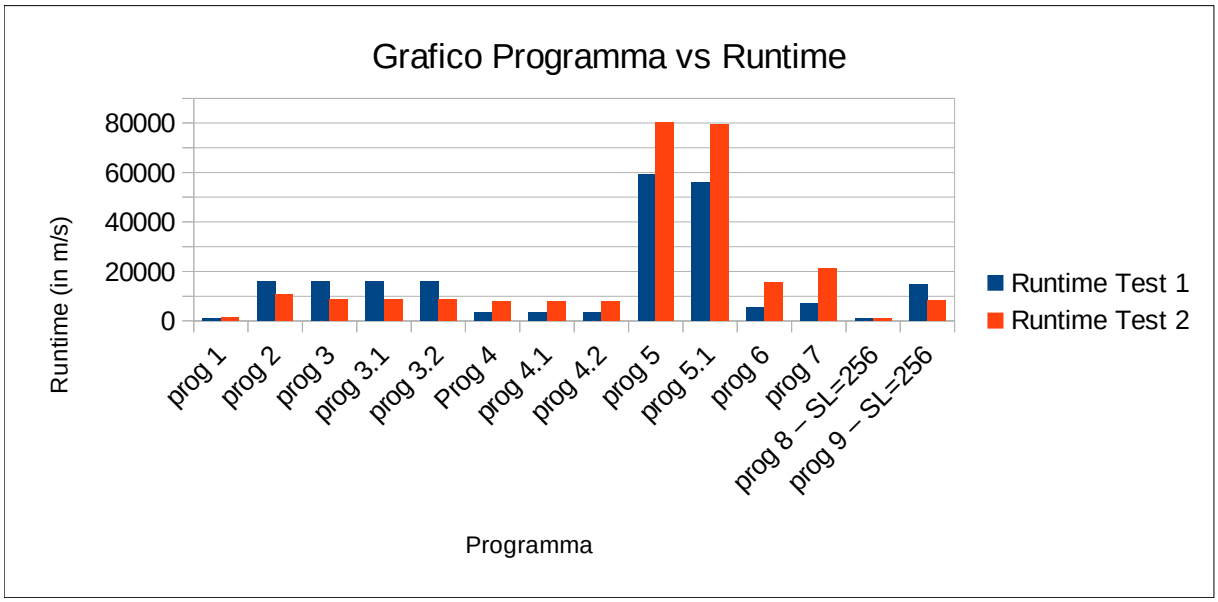
A conclusione di questa sezione è doveroso fare un commento su questi risultati, qua il miglioramento rispetto all'***Implementazione 2*** c'è stato ed è anche marcato.

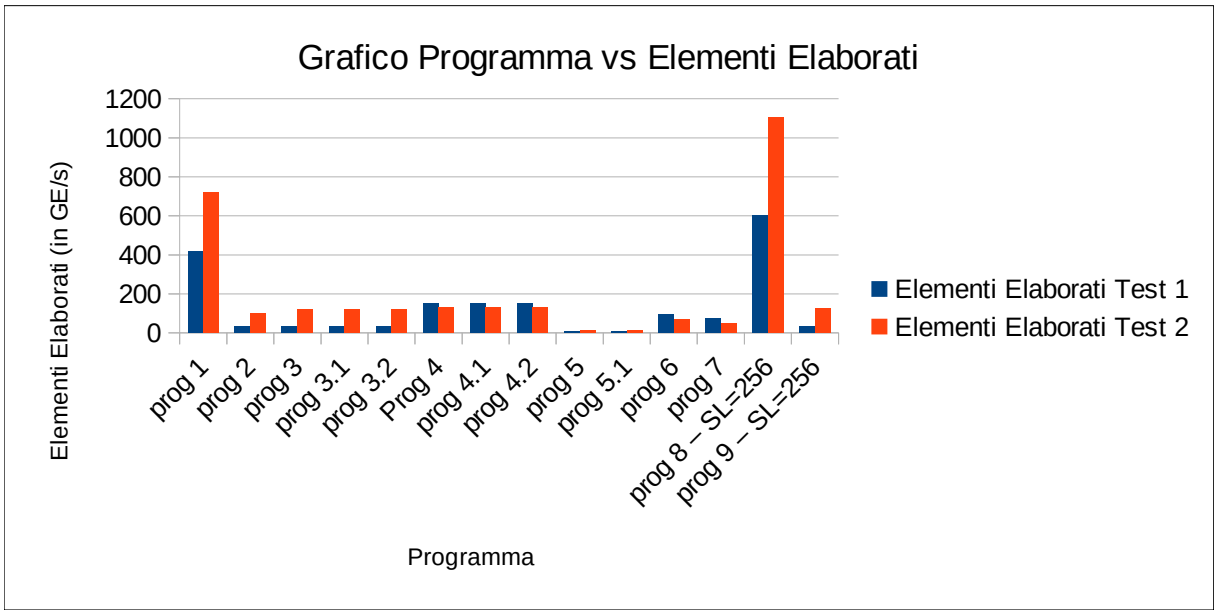
Infatti, sono stati guadagnati ben 2 secondi nel Test 2.

Nonostante ciò, continuano comunque a pesare i massicci accessi in Global Memory ed è per questo che siamo lontani ,dal punto di vista delle prestazioni, dall'***Implementazione 8***.

{GRAFICI}

In questa sezione verranno mostrati dei grafici riepilogativi delle prestazioni di tutte le implementazioni del Progetto:





{CONCLUSIONI}

Quest'esperienza ha portato alla ribalta numerosi temi caldi dal punto di vista dello stile di programmazione sulla GPU, uno fra tutti è l'impatto ,dal punto di vista delle performance, degli accessi alla Global Memory.

A conferma di ciò, questo progetto dimostra che si deve fare un uso davvero parsimonioso della Global Memory se non si vuole incorrere in forti perdite di performance, al più si potrebbe cercare di coprire le latenze di accesso con del calcolo, strategia non sempre attuabile purtroppo.

Un altro tema portato alla ribalta da questo progetto è ,secondo chi scrive, la poca maturità dell'API OpenCL dal punto di vista della coerenza di comportamento fra hardware di vendor diverso.

Infatti molto codice di questo progetto si comporta in modo diverso su piattaforme diverse come già spiegato all'inizio di questa trattazione, in particolar modo questo comportamento si manifesta quando si imposta un Local Work-Size manuale.

Al di là di questo, quest'esperienza mi ha permesso di esplorare in pratica le conoscenze del corso e di questo ne sono davvero felice :)

{REFERENZE}

[1] - *GPU SQL Query Accelerator*

Link Paper: http://www.intjit.org/cms/journal/volume/22/1/221_3.pdf

[2] - *GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors*

Link Paper: *http://www.cse.chalmers.se/~tsigas/papers/GPU-Quicksort-jea.pdf*

{AUTORE}

Rosario Scalia