Barbieri Manuel - 0001189733
Catellani Alberto - 0001194644

Progetto Sistemi Digitali M

*Si tenga a mente che alcuni codici allegati a questo documento generano file .txt di report nella cartella dentro cui sono lanciati.

1. Studio del Problema

1.1 Contesto

Dato un fenomeno, se si effettua una serie indefinita di misurazioni dello stesso lungo un breve lasso di tempo, queste possono essere raccolte in un *cluster*. Ripetendo il procedimento ad intervalli regolari, è logico supporre che i valori registrati dai *cluster* siano tra loro simili, in quanto provenienti dallo stesso fenomeno. Questo non rispecchia sempre la realtà: fattori esterni, anche detti **anomalie**, possono influenzare i risultati delle misurazioni.

Per verificare l'omogeneità di un insieme di *cluster* di misurazioni, ovvero il grado di incidenza delle anomalie, è possibile avvalersi della correlazione lineare. Questo è vero in qualsiasi contesto in cui si effettui una trattazione ed analisi di dati sperimentali. Alcuni esempi:

- Rilevazioni geologiche o geotermiche
- Analisi del mercato
- Sviluppo di nuovi medicinali

Lo studio di omogeneità può essere realizzato avvalendosi del *coefficiente di correlazione di Pearson*, strumento statistico usato per confrontare due variabili casuali che restituisce un valore numerico compreso tra +1 a -1, che viene interpretato come segue:

- Tanto più il valore tende ad uno degli estremi, maggiore sarà il grado di relazione lineare tra le variabili. Due variabili in relazione lineare si influenzano a vicenda: il diminuire di una porta al diminuire dell'altra e l'aumentare dell'una conduce all'aumentare dell'altra
- Tanto più il valore si avvicina a 0, tanto minore sarà il grado di relazione lineare tra le variabili. Il comportamento che una variabile assume a seguito della variazione dell'altra non è da interpretarsi come indice di legame tra le variabili

Si noti che il *coefficiente di Pearson*:

- E' un valore adimensionale non influenzato dalle unità di misura in cui le variabili sono espresse
- Non è influenzato dall'eventuale dipendenza tra le variabili, né ne è un indicatore, a contrario della *regressione lineare*. Ciò significa che il confronto tra la variabile *X* e *Y* produce lo stesso coefficiente di Pearson del confronto tra *Y* e *X*.

Formula del coefficiente di Pearson tra le serie numeriche $\{X_1, X_2, \ldots, X_n\}$ e $\{Y_1, Y_2, \ldots, Y_n\}$:

$$r = \frac{\sum_{i=1}^{n}\left(X_i - \overline{X}\right)\left(Y_i - \overline{Y}\right)}{\sqrt{\sum_{i=1}^{n}\left(X_i - \overline{X}\right)^2} \cdot \sqrt{\sum_{i=1}^{n}\left(Y_i - \overline{Y}\right)^2}}$$

1.1 Comportamento di base

Un algoritmo che voglia confrontare tra loro coppie di *cluster di misurazioni* tramite coefficiente di Pearson è intrinsecamente sequenziale: non è possibile eseguire il passaggio *i-esimo* senza aver eseguito gli *i-1* passaggi precedenti, in quanto i risultati dei secondi sono necessari al primo per effettuare le proprie computazioni.

Il processo di ottenimento dei coefficienti di Pearson è scomponibile in:

1. Realizzare le serie numeriche composte da valori plausibili
2. Ottenere la media aritmetica per ogni *cluster*
3. Computare la sommatoria $\sqrt{\sum_{i=1}^{n}\left(X_i - \overline{X}\right)^2}$ degli $x_1 \ldots x_n$ valori che compongono ciascuno dei *cluster*
4. Calcolare la sommatoria $\sum_{i=1}^{n}\left(X_i - \overline{X}\right)\left(Y_i - \overline{Y}\right)$ per ogni coppia di *cluster*
5. Combinare i risultati dei punti precedenti per comporre la formula dei coefficienti

Assimilando i *cluster* a serie numeriche, ognuna di queste deve essere confrontata con tutte le altre per garantire una precisa analisi di omogeneità. Ergo, se sono presenti 10 serie di misurazioni, ciascuna di queste sarà confrontata con le altre 9, producendo un coefficiente di Pearson per ogni confronto.

Il numero delle computazioni necessarie è considerevole: se supponessimo che il programma, dopo aver calcolato ciascun coefficiente, lo posizioni all'interno di una matrice, le dimensioni di questa sarebbero pari a $m \times (m-1)$, dove m rappresenta il numero di serie considerate.

$$\begin{bmatrix} r_{1,2} & r_{1,3} & r_{1,4} & r_{1,5} \\ r_{2,1} & r_{2,3} & r_{2,4} & r_{2,5} \\ r_{3,1} & r_{3,2} & r_{3,4} & r_{3,5} \\ r_{4,1} & r_{4,2} & r_{4,3} & r_{4,5} \\ r_{5,1} & r_{5,2} & r_{5,3} & r_{5,4} \end{bmatrix}$$

Rappresentazione della matrice menzionata nel caso $N = 5$

Il numero di operazioni effettivamente da eseguire è inferiore in quanto $r_{xy} = r_{yx}$. Dunque, per determinare quanti coefficienti sarà necessario calcolare, ci si avvale della formula di combinazione statistica di due elementi:

$$C(n,\ 2) = \frac{m!}{2!\ (n-2)!} = \frac{m \cdot (m-1)}{2}$$

Se immaginassimo nuovamente la matrice, supponendo di non effettuare computazioni superflue, questa avrebbe ancora dimensioni pari a $m \times (m-1)$, ma metà delle sue celle sarebbe vuota.

$$\begin{bmatrix} r_{1,2} & r_{1,3} & r_{1,4} & r_{1,5} \\ 0 & r_{2,3} & r_{2,4} & r_{2,5} \\ 0 & 0 & r_{3,4} & r_{3,5} \\ 0 & 0 & 0 & r_{4,5} \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Rappresentazione della nuova matrice nel caso $N = 5$

Nota
In questo progetto, ci si è concentrati soltanto sui **primi due passaggi** dei cinque descritti. Nonostante questo, si è comunque finora tentato di dare un contesto completo. Dalla sezione successiva, gli ultimi tre passaggi non verranno più nominati, nonostante questi si predispongano alla parallelizzazione tanto quanto i passaggi scelti.

1.2 Implementazione Sequenziale

A fini implementativi, i cluster di misurazioni utilizzati sono rappresentati come array di valori reali. In particolare, si fa uso del tipo di dato **float**. In quanto programmare in CUDA comporta l'utilizzo di un'architettura basata sul linguaggio C, questi valori numerici saranno codificati secondo lo standard IEEE *single-digit precision*.

Dunque, per ogni valore float saranno utilizzati 32 bit, di cui:
- 1 per rappresentarne il segno
- 8 per rappresentarne l'esponente
- 23 per rappresentarne la mantissa

Questo dovrebbe garantire un'approssimazione sufficientemente attendibile fintanto che non si superano le sei cifre decimali.

Sia noto che le serie numeriche non sono popolate in maniera puramente casuale, bensì si è preferito renderle "verosimili":
- Tutte le serie partono dallo stesso valore iniziale V
- Il secondo elemento di ogni serie è calcolato come $V \pm \varepsilon$, dove ε è una quantità percentuale.
- Il terzo elemento è ottenuto dalla medesima formula, ma usando il secondo elemento invece che V
- …

Dunque, ogni numero che compone la serie (eccezione per il primo) è ottenuto variando il precedente di una percentuale ε. Il massimo valore che ε può assumere è impostabile nelle variabili globali del codice.

1.2.1 Risultati dell'implementazione sequenziale

Per ottenere un punto di partenza più accurato, il codice sequenziale è stato eseguito molteplici volte, rigenerando i valori delle serie in ognuna. Quanto segue è sono i tempi di computazione che, in media, il processo ha impiegato:

report.txt

```
1   Number of series: 8192
2   Number of elements per series: 8192
3   Number of tests: 500
4
5   Average generation time: 1.20337 sec
6   Average mean calculation time: 0.26660 sec
7   Average sqrt calculation time: 0.27366 sec
8   Average numerator calculation time: 0.26797 sec
9   Average coefficient calculation time: 0.00002 sec
10  Average total calculation time: 2.01166 sec
11
```

Output del file "sequenzialeMultipleProve.cu"

Come si può osservare, la maggior parte del runtime viene dedicata all'inizializzazione delle serie. In un contesto reale, questo passaggio non verrebbe effettuato: le misurazioni ottenute sperimentalmente sarebbero già disponibili (in file esterno, ad esempio) e il sistema dovrebbe unicamente recuperarle.

1.3 Decomposizione e Struttura Parallela

L'algoritmo si presta per un considerevole livello di parallelizzazione. Dalla semplice osservazione delle strutture dati con cui si lavora, ovvero gli array, è intuitivo accoppiare ogni cella di questi ad un *GPU-thread* che operi con il valore ivi contenuto.

Supponendo che le capacità computazionali del device utilizzato siano illimitate e tali accoppiamenti siano possibili, ogni thread potrebbe efficientemente:

- Durante il passaggio di creazione delle serie, produrre un numero che si discosta di ε dal precedente
- Durante il calcolo della media, sommare il valore a cui è assegnato agli altri valori della medesima serie

Realisticamente, la GPU utilizzata non ha capacità illimitata, specialmente nelle dimensioni dei blocchi. Seguono le specifiche dell'host e del device utilizzato:

Processore	AMD Ryzen 5 3600 6-Core Processor 3.59 GHz
RAM installata	16,0 GB

```
CUDA Runtime Version: 12.60
Number of CUDA Devices: 1

Device 0: NVIDIA GeForce RTX 2060
1. Compute Capability: 7.5
2. Total Global Memory: 6.00 GB
3. Number of Multiprocessors: 30
4. Clock Core: 1755 MHz
5. Memory Clock: 7001 MHz
6. Memory Bus Width: 192 bit
7. L2 Cache Size: 3072 KB
8. Shared Memory per Block: 48 KB
9. Maximum Threads per Block: 1024
10. Maximum Grid Dimensions: (2147483647, 65535, 65535)
11. Maximum Block Dimensions: (1024, 1024, 64)
12. Warp Size: 32
13. Total Constant Memory: 65536 bytes
14. Texture Alignment: 512 bytes

15. Memory bandwidth: 336.05 GB/s
18. SM number: 30
```

1.3.1 Dati utilizzati e Dati prodotti

Detto M il numero delle serie ed N la lunghezza di ciascuna di queste:

	Descrizione Passaggio	Dati in ingresso	Dati in uscita
1	Si popolano le serie numeriche	-	$N \cdot M$
2	Per ogni serie, se ne calcola la media aritmetica	$N \cdot M$	M

1.3.2 Complessità delle computazioni effettuate

Detto M il numero delle serie ed N la lunghezza di ciascuna di queste, nella implementazione **sequenziale**:

	Descrizione Passaggio	Complessità del Lavoro	Complessità dello Step
1	Si popolano le serie	$O(NM)$	$O(NM)$
2	Per ogni serie, se ne calcola la media aritmetica	$O(NM)$	$O(NM)$*

*La complessità dello step dipende da come la somma di tutti gli elementi dell'array viene implementata. Nel codice sequenziale realizzato, si fa uso di una computazione sequenziale semplice. Nel caso di una riduzione iterativa a coppie, la complessità si riduce a $O\left(M\ log_2 N\right)$.

1.3.3 La natura monodimensionale del problema

Prima di approcciarsi a tematiche come dimensionamenti di blocchi e griglie, è opportuno comprendere se la natura del problema sia monodimensionale o bidimensionale. Per fare ciò, si è usato un processo di “scomposizione bottom-up”.

Generalizzando:

1. E’ sensato che la minima unità computazionale disponibile, ovvero il *GPU-thread*, lavori al più basso livello possibile:

   1 Thread ≡* 1 cella di uno degli M array

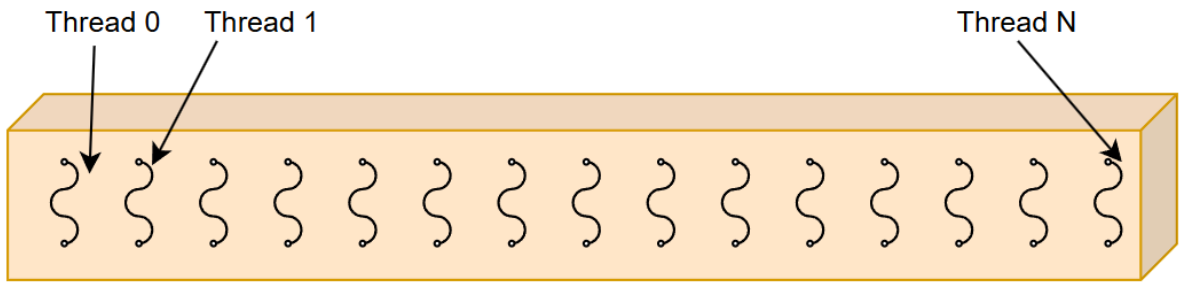
   *Questo è vero unicamente se gli array contengono al più 1024 elementi per via dei limiti attuali sulle dimensioni dei blocchi nelle architetture NVIDIA. Realisticamente, 1 thread potrebbe essere associato a molteplici celle.
2. Salendo di un livello, si giunge alla logica assunzione che:

   1 Block ≡ N Thread ≡ N celle di uno degli M array

   ↓

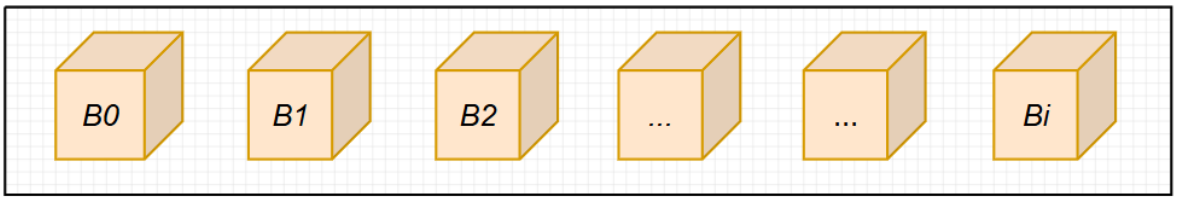
   1 Block ≡ 1 Array

Quanto detto è sufficiente per concludere che si lavorerà con **blocchi monodimensionali**.



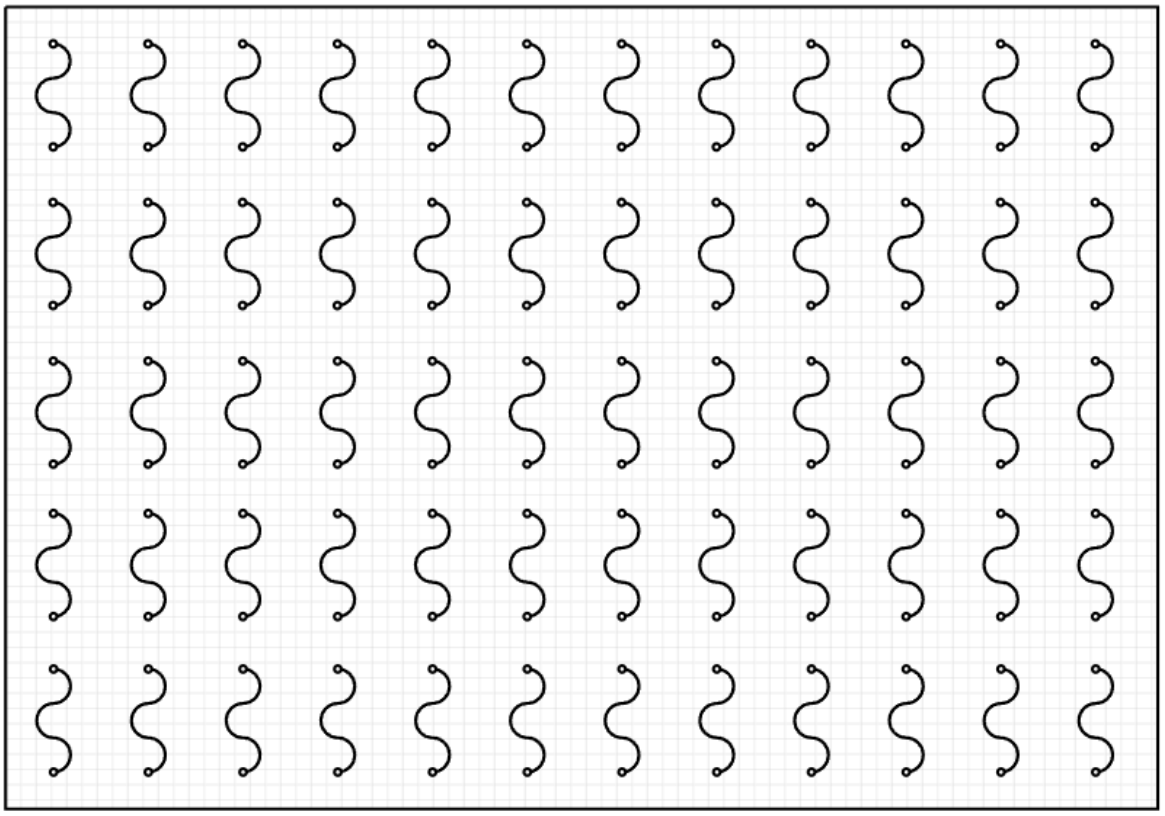
Rappresentazione di un blocco 1D, come quelli usati nel progetto

I cluster di misurazione saranno contenuti in un “array di M serie numeriche”. Avendo, per ora, supposto che 1 blocco coincida con 1 array, la griglia si presenterebbe come un “array di blocchi”.



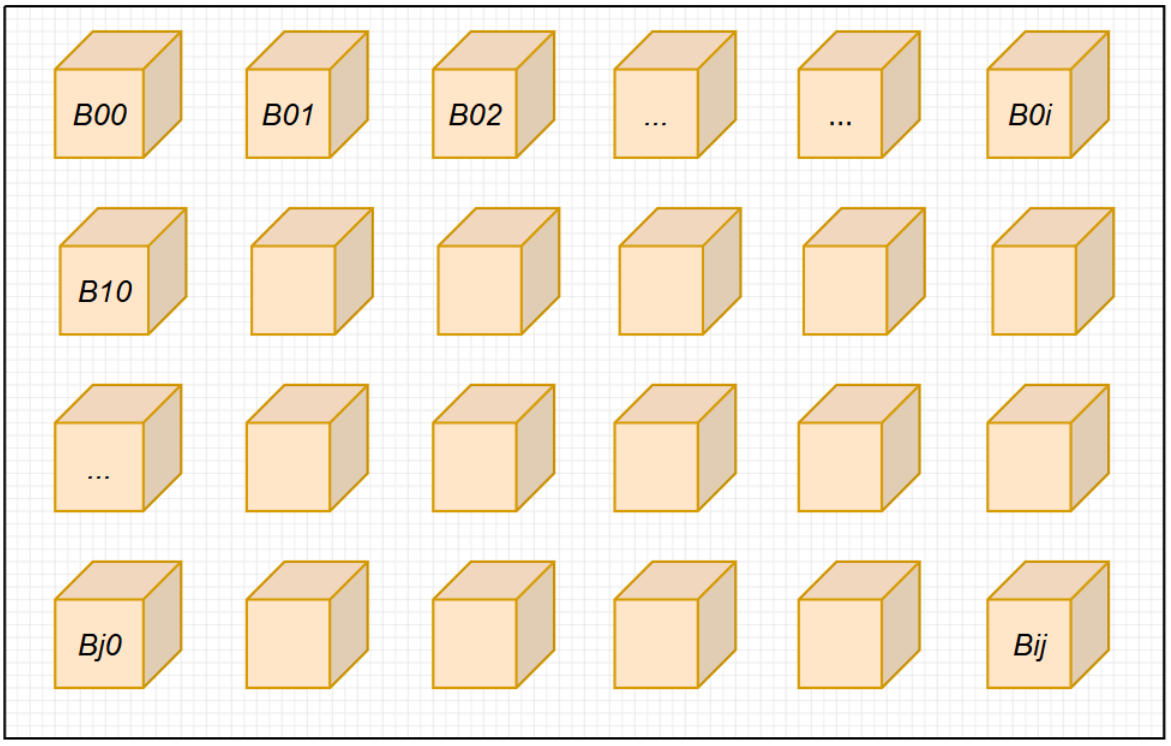
Rappresentazione di una griglia 1D

Se si aprissero i blocchi che compongono questa griglia, quello che risulterebbe è una matrice di dimensione $N \times M$ composta da thread.



La matrice ottenuta “aprendo” i blocchi.

La matrice che si utilizzerebbe, dunque, sarebbe diversa da quella predisposta per un problema bidimensionale.



Rappresentazione esemplificativa di una griglia 2D

Confrontandole, si nota che l'una sarebbe composta da NM thread, mentre l'altra da NM blocchi di thread. Si può concludere che in questo progetto **la griglia di blocchi è 1D**.

3. Step 1 - La generazione delle Serie

Secondo la filosofia di generazione adottata, per ogni serie, produrre il valore *i-esimo* (con *i* diverso da 0), richiede la conoscenza degli *i-1* valori ad esso precedenti. Il processo di generazione lavora con dati dipendenti tra loro, il che limita il grado di parallelizzazione.

Per meglio comprendere, si faccia riferimento alla seguente immagine:

```
for (int i = 0; i < numberOfSeries; i++) {
    for (int j = 0; j < elementsInArray; j++) {
        //generate a random number
        //save the number within the array
    }
}
```

Ciclo rappresentativo per descrivere il processo di generazione dei dati iniziali

Dei due cicli innestati, quello interno genera i valori di una serie, mentre quello esterno forza la ripetizione del processo per tutte le serie. Soltanto il ciclo esterno è parallelizzabile. Se, invece, tutti i valori che compongono le serie fossero generati in maniera indipendente dai precedenti, allora entrambi i cicli sarebbero parallelizzabili.

In questo primo step, i *GPU-thread* verranno utilizzati in maniera "naive", ovvero come unità di calcolo indipendenti e non-cooperanti, dediti alla generazione sequenziale degli elementi della serie. Sarà generato 1 elemento per ciascuna serie ad ogni passaggio. Sebbene questo non sfrutti al massimo le capacità di parallelismo della GPU, dovrebbe comunque condurre ad un runtime migliore dell'implementazione sequenziale.

Per permettere la generazione di variazioni percentuali casuali device-side, si è fatto uso della libreria *curand*.

3.1 Analisi preliminare

3.1.1 Dati utilizzati e dati prodotti

Nessun dato viene mandato in ingresso al device. Nell'invocazione del kernel sono unicamente forniti i dettagli necessari per la generazione dei numeri semi-casuali (come il valore massimo di ε) e la struttura dati per la memorizzazione delle serie.

Al termine della computazione, il kernel dovrebbe aver prodotto un numero di dati sufficienti a riempire completamente gli N slot di ciascuno degli M array: NM.

3.1.2 Complessità

Variare un valore di una percentuale generata casualmente ε è una computazione che non dipende dai dati in ingresso. Popolare un'intera serie richiede N computazioni, pertanto la complessità totale di un'implementazione sequenziale è $O(NM)$.
In un contesto parallelo, sono generati M valori ad ogni iterazione, 1 per ciascuna serie. Pertanto, soltanto N passaggi sono necessari per riempire le serie: $O(N)$

3.1.3 Grid, Blocks and Threads

Si devono quindi avviare M entità computazionali, ciascuna delle quali con il compito di popolare interamente una serie. Dunque, le dimensioni della griglia di questo kernel saranno determinate da:

$$numero\ di\ blocchi\ = \frac{M + numero\ di\ thread\ per\ blocco - 1}{numero\ di\ thread\ per\ blocco}$$

Non si hanno vincoli riguardo le dimensioni del blocco. Arbitrariamente, è stato scelto di lavorare con il massimo numero di thread contenibili in un blocco: 1024.
La formula per le dimensioni della griglia quindi diviene:

$$\frac{M + 1023}{1024}$$

3.1.4 Associazione data - thread

All'interno della funzione kernel sarà necessario determinare l'identificativo globale del thread. Di norma, questo è ottenuto dalla formula:

```
int idx = blockIdx.x * blockDim.x + threadIdx.x
```

Ogni thread deve popolare un'intera serie numerica, cioè generare N elementi e posizionarli entro un indirizzo di memoria adibito al contenimento del risultato.
Ricordando che la struttura dati che memorizza le serie altro non è che un array e chiamando questo S, si avrà che:

- Il thread con idx 0 genererà i valori che andranno a finire nelle celle S[0…N-1]
- Il thread con idx 1 genererà i valori che andranno a finire in S[N…2N-1]
- …

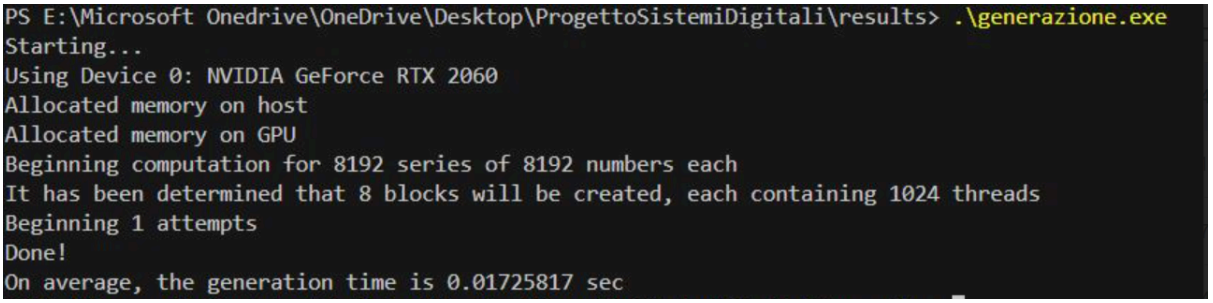
Detto N = 1024:

- Il thread con idx 0 genererà i valori che andranno a finire nelle celle S[0…1023]
- Il thread con idx 1 genererà i valori che andranno a finire in S[1024…2047]
- …

Implementativamente, i due estremi che compongono l'intervallo entro cui il thread deve scrivere sono stati espressi in funzione del suo identificativo:

```
int startIndex = idx * N;
int endIndex = startIndex + N;
```

3.2 Prima Versione del Kernel K1



Risultati ottenuti dal file “generazioneParallela.cu” variando il numero di blocchi

Per ora, basandosi sui meri CPU timer, con il semplice utilizzo della GPU si è osservato uno speedup di:

$$\frac{1.20337}{0.0172581} \sim 70$$

Provando ad alzare il numero di prove e produrre un primo report di tempistiche, si è ottenuto quanto segue:

NVIDIA Nsight Systems				
Name	Time %	Total Time	Avg Time	Number of Calls
CudaDeviceSynchronize	43%	973.36 ms	9.74 ms	100
K1	43%	965.25 ms	9.65 ms	100
CudaMemcpy	6%	148.5 ms	74.2 ms	2
CudaLaunchKernel	0%	9.996 ms	99.96 µs	100
CudaFree	0%	1.139 ms	1.139 ms	1
CudaMalloc	0%	800.870 µs	800.870 µs	1

La quasi totalità del tempo di esecuzione viene occupata dalle operazioni di sincronizzazione host-device e dal kernel stesso. Presumibile, considerando che sono state effettuate 100 prove.

Nell’ordine di impatto sul tempo di computazione, seguono le operazioni di trasferimento dati in memoria. Considerando però che il numero di esecuzioni di queste ultime è indipendente dal numero di prove effettuate, a queste spetterebbe l’impatto maggiore se si fosse effettuata soltanto una prova.

Visti i considerevoli tempi di profiling che NVIDIA Nsight Compute richiede, si è deciso di effettuare il profiling con un solo tentativo e non 100.

NVIDIA Nsight Compute					
Kernel Name	Thread Lanciati	Istruzioni Eseguite	Memory Through.	Branch Efficiency	Duration
K1	8192	34.6 M	29.7 GB/s	100%	12.21 ms

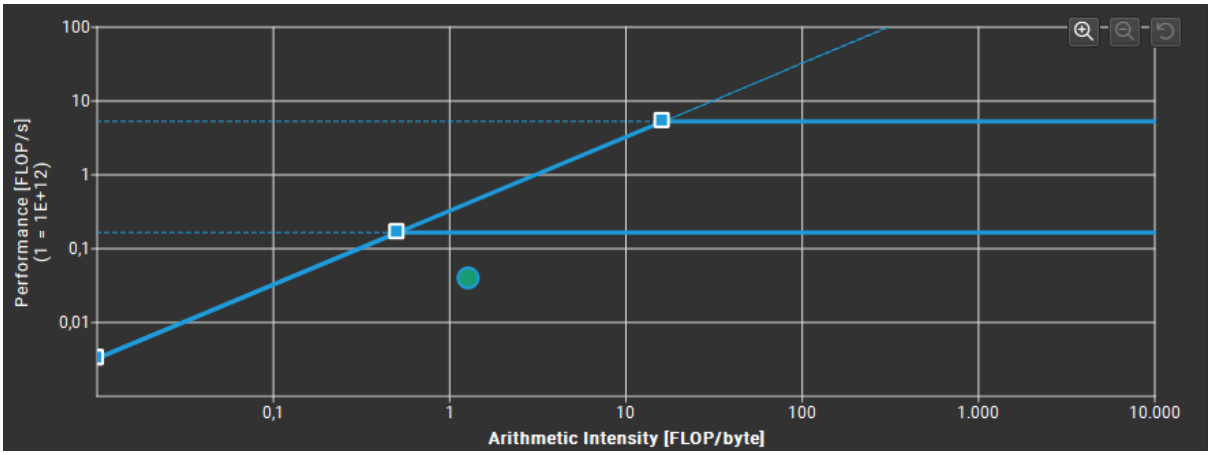


Diagramma Roofline del kernel K1

Dallo strumento roofline, è evidente vi sia un margine di miglioramento. Tuttavia, ci si trova entro la sezione in pendenza. Pertanto, il kernel **K1 è memory bound**. Questo è in linea con le tempistiche viste da Nsight Systems.

L'analisi con gli strumenti del *CUDA-Toolkit* ha rivelato una branch efficiency perfetta, infatti il kernel non presenta alcuna istruzione condizionale che possa scatenare warp divergence. Non è stato ritenuto necessario, pertanto, agire per migliorare questo aspetto.

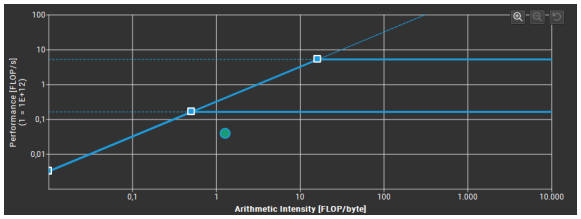
3.3 Seconda Versione del Kernel K1

Prima di procedere a ottimizzazioni mirate, si è utilizzato Nsight Compute per verificare se la decisione arbitraria di lanciare 1024 thread per blocco sia quella corretta. Il kernel è stato quindi eseguito con 512, 256, 128, 64 e 32 thread per blocco.

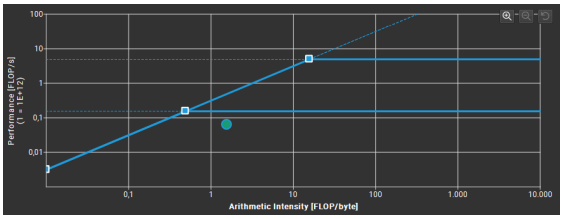
Al calare delle dimensioni del blocco, si è ottenuto un valore di Duration decrescente a scapito di un allontanamento dal tetto nel *roofline diagram*. Solo nelle casistiche di 64 e 32 thread per blocco, la Duration è calata drasticamente senza distanziarsi dal tetto.

Thread per blocco	Duration
1024	12.21 ms
64	8.32 ms
32	7.57 ms

Nel caso dei 32 thread, il punto che descrive il kernel nel diagramma roofline si è avvicinato al tetto di memoria. Visto i notevoli risultati, si è deciso di mantenere i 32 thread per blocco per K1.



Roofline Diagram per 1024 thread per blocco



Roofline Diagram per 32thread per blocco

3.4 Considerazioni di Ottimizzazione su K1

Nsight Compute ha individuato due pecche principali:

- La presenza di accessi non coalescenti
- Un alta percentuale di stallo per i thread in esecuzione

Come il software suggerisce, la seconda causa di bassa efficienza è presumibilmente dovuta alla combinazione della prima e dal basso hit-rate delle cache, quest'ultimo dovuto al pattern di accesso non-interleaved che il kernel utilizza nel scrivere le serie sull'array di output.

Il primo punto non è risolvibile per via della natura della generazione: in quanto due thread genereranno serie completamente diverse, questi scriveranno in celle non-adiacenti, nonostante appartengano allo stesso warp. **La non-coalescenza è inevitabile**.

Il secondo problema sarebbe risolvibile cambiando la logica di individuazione degli indici in cui il thread deve scrivere. Attualmente, il thread dedicato alla K-esima serie scrive nelle celle:

[K, . . . , K + N - 1]

Dunque, l'array di output si presenta tutti gli elementi della stessa serie adiacenti:

[(N elementi della serie 0) , (N elementi della serie 1) , . . . , (N elementi della serie M)]

Un altro approccio, di stampo più interleaved, potrebbe vedere gli elementi dello stesso indice adiacenti tra loro:

[(M elementi di indice 0) , (M elementi della serie 1) , . . . , (M elementi di indice N)]

E' una soluzione possibile e più efficiente, ma che rende delineare un algoritmo per sommare gli elementi di ogni serie più complesso, contorto e, potenzialmente, inefficiente. Si tratta dunque di compiere una **scelta**: rendere il kernel K1 leggermente più efficiente ma complicare la realizzazione del kernel K2 oppure accettare i limiti di ottimizzazione del kernel K1 per concentrarsi sul kernel K2, sicuramente più ottimizzabile.

La seconda opzione tra quelle descritte è sembrata la più adatta.

Si è comuqnue tentato di diminuire gli stalli a cui i thread del kernel K1 vanno incontro. Le misure adottate riguardano il posizionamento delle variabili che il kernel utilizza in differenti porzioni di memoria:

1. Le costanti di generazione di curand sono state salvate come __const__
2. L'array mandato in ingresso al kernel è stato salvato in memoria Zero-Copy (prima) e in UVA (dopo)

Non è stato possibile osservare un miglioramento. Le considerazioni riguardo la Zero-Copy Memory e la Unified Virtual Addressing si sono rivelate particolarmente infruttuose: la durate delle operazioni di memoria presenti nel codice originale sono leggermente diminuite, ma il tempo di esecuzione complessivo è aumentato in quanto il device si è dovuto sincronizzare più spesso con l'host.

4. Step 2 - Il calcolo della media aritmetica

4.1 Analisi preliminare

Una media aritmetica presuppone che tutti gli elementi siano, preliminarmente, sommati tra loro. Detti N elementi, sequenzialmente, ciò impiegherebbe N-1 passaggi, essendo N-1 le operazioni di somma necessarie.

Passando a un contesto parallelo, ci si potrebbe chiedere a quanti iterazioni sequenziali corrisponde una iterazione parallela. La risposta dipende:

- Da quante somme per iterazione effettua ciascun thread lanciato
- Da quanti thread sono stati lanciati

Se, ad esempio, si assume che ogni thread esegua 1 somma e si lancino $\frac{N-1}{2}$ thread, allora in soltanto 2 iterazioni l'implementazione parallela ha concluso la computazione.

Tuttavia, il contesto dello step 2 è leggermente più complesso: si hanno M gruppi di N elementi. Pertanto, rispondere alla domanda "a quante iterazioni sequenziali corrisponde un'iterazione parallela?" si complica. Occorre difatti tenere in conto di:

- Quante somme per iterazione effettua ciascun thread lanciato
- Quanti thread sono stati lanciati per ogni serie numerica

Assumendo che, nuovamente, ad ogni iterazione ogni thread effettui 2 somme ed un totale di 2 thread siano stanziati per ogni serie, allora serviranno soltanto $\frac{N-1}{4}$ iterazioni per completare le $M(N-1)$ somme necessarie.

Per quanto detto, il minimo numero di somme che si possono effettuare in contemporanea è M, 1 per ciascuna serie. Questo rappresenta il caso peggiore, in cui saranno necessari $N-1$ passaggi per concludere la computazione.

4.1.1 Dati utilizzati

Dati gli N elementi che compongono ciascuna delle M serie, vi saranno NM dati in ingresso ed M dati in uscita.

4.1.2 Complessità

Sequenzialmente, la complessità del lavoro è di $O(NM)$, in quanto saranno necessarie un numero di somme pari agli elementi delle serie prima di poter calcolare la media. In un contesto parallelo, ponendosi nel caso peggiore (1 somma per serie ad ogni iterazione) questo valore scende a $O(N)$.

Nel caso sequenziale, la complessità dello step può variare tra $O(NM)$ o $O\left(Mlog_2 N\right)$ in base a come viene realizzata la somma tra gli elementi di una serie.
Un approccio "ad accumulatore", in cui si effettua una somma ad ogni passaggio, rappresentabile con la formula:

$$(((a + b) + c) + d)$$

Richiede un numero di passaggi pari alle dimensioni dell'array: $O(MN)$.
Una processo di somma iterativa in cui, ad ogni passaggio, si effettui un numero di passaggi pari alla metà delle somme effettuate nel passaggio precedente, richiede un totale di $Mlog_2 N$ passaggi:

$$(a + b) + (c + d) + (d + e)$$

Soltanto il secondo di questi approcci può essere parallelizzato, in quanto il primo presuppone l'attesa di ogni risultato intermedio prima di calcolare il successivo.

La complessità dello step di un'implementazione parallela con somma iterativa, considerando il caso più sfavorevole di 1 somma per serie ad iterazione, scende a $O\left(log_2 N\right)$.

4.1.3 Grid, Blocks and Threads

Appurato che le entità che effettueranno le somme tra cella e cella degli array saranno *GPU-thread*, resta da determinare come questi siano organizzati. Per fare ciò, occorre definire a quanto ammonti il quantitativo di dati con cui si lavora.

Se si ponesse NM come tale valore e si utilizzasse la formula generica vista nel primo step, si distribuirebbe in maniera omogenea i dati da calcolare lungo la griglia. Tuttavia, questo condurrebbe a uno scenario non desiderabile: due thread dello stesso blocco potrebbero lavorare con dati appartenenti a serie numeriche differenti.

Se il contesto del passaggio fosse giungere ad una sola media, ovvero produrre una sommatoria di tutti gli NM elementi, la situazione appena descritta non rappresenterebbe un problema: non ci sarebbe motivo per assicurarsi che elementi appartenenti alla serie A non vengano sommati con elementi della serie B.
Tuttavia, lo scenario è differente: se un blocco calcolasse una somma usando dati appartenenti alle serie A e B, la media che ne risulterebbe non sarebbe né la media di A, né quella di B. Conseguentemente, **si rende necessario introdurre una Regola: tutti i thread di un blocco operino con dati della stessa serie**.

Affinché ci si trovi in quello che in precedenza è stato chiamato "caso peggiore", serve lanciare un numero di blocchi almeno pari al numero delle serie con cui si sta lavorando. Se così non fosse, al completamento della computazione, si avrebbero meno di M medie. Per

mantenersi in questa casistica “peggiore”, si è arbitrariamente deciso che, almeno nella prima implementazione del kernel K2, saranno lanciati soltanto M blocchi.

Resta da chiarire le dimensioni di ogni blocco.

La somma è un’operazione che prende 2 addendi e restituisce 1 risultato. Dunque, se i dati in ingresso fossero N, allora un blocco composto da $\frac{N}{2}$ thread sarebbe sufficiente per concludere la computazione in maniera efficiente.
Oltretutto, questo permetterebbe di sfruttare gli altri $\frac{N}{2}$ thread rimasti inattivi per computare la somma dei valori che compongono un’altra serie. Con questa accortezza, si permetterebbe allo stesso blocco di computare su due serie, se solo non fosse che questo **infrange la Regola**.

L’ottimizzazione descritta non è possibile, in quanto tutti i thread di un blocco lavorano con dati della stessa serie. Volendo rendere il codice funzionante indipendentemente da N, si è deciso di **sfruttare al massimo** le potenzialità di ogni blocco. In generale, viste le limitazioni attuali di 1024 thread per blocco, si è scelto quest’ultimo come numero di thread per blocco per K2.

Se N fosse stabilito in partenza, sarebbe sufficiente seguire le due considerazioni sottostanti per stabilire l’uno dei due parametri di lancio del kernel:

- Se $N > 1024$ → blockDim = 1024
- Se $N \leq 1024$ → blockDim = $\frac{N}{2}$

4.2 Prima Versione del Kernel K2

4.2.1 Le due parti dell’algoritmo

L’algoritmo ha l'obiettivo di calcolare le medie aritmetiche tra gli elementi di molteplici serie numeriche. I risultati finali, come anche i risultati intermedi, saranno memorizzati all’interno dello stesso array in ingresso al kernel (*memorizzazione in-place*).

Si parte ricordando che:

- Soltanto 1 blocco lavora su ciascuna serie
- Non si hanno limitazioni su N, purché questo sia una potenza di 2

L’algoritmo di K2 si complica in base al valore del seguente rapporto:

$$R = \frac{N}{blockDim}$$

Se $R \leq 1$ → Ci sono più thread che dati da computare
Se $R > 1$* → Ci sono meno thread che dati da computare

Quando R < 1 o R = 1
C’è una sovrabbondanza di thread, pertanto non tutti effettueranno una somma. In particolare, poiché si è supposto che N sia una potenza di 2, la percentuale dei thread del

blocco che computeranno sarà $\frac{N/2}{blockDim} \cdot 100$. Ci troviamo, quindi, nel caso della *Riduzione Parallela*.

La *Riduzione Parallela* può implementata in due maniere differenti, in base al pattern di accesso ai dati che i thread utilizzano: Nieghbored o Interleaved. In entrambe, però:

1. Il numero di somme da computare dimezza ad ogni iterazione
2. Il numero di thread che partecipano alle somme si dimezza ad ogni iterazione
3. Alla conclusione di entrambi gli approcci di riduzione, nella cella di indice 0 dell'array iniziale si troverà il risultato della somma degli N elementi.

Neighbored	Interleaved
Si sommano celle contigue. La cella p sarà sommata con la cella $p + S$, con S detto **stride**.	Si sommano celle distanti. In particolare, è come se sezionasse l'array iniziale in due, esattamente a partire da metà della sua lunghezza, e si sommassero le celle che occupano le posizioni di uguale indice. La cella p della prima metà dell'array sarà sommata con la cella p della seconda metà, ovvero la cella $p + S$, con S detto **stride**.
Lo **stride** rappresenta la distanza tra le due celle sommate. Questo parte da 1 e raddoppia ad ogni iterazione.	Lo **stride** rappresenta l'offset necessario per porsi all'inizio della seconda metà dall'array. Questo parte da $\frac{N}{2}$ e si dimezza ad ogni iterazione.

Nel contesto del progetto, non verrà effettuata soltanto 1 riduzione parallela alla volta, ma M, essendo quest'ultimo il numero delle serie con cui lavorare. Si è scelto, in quanto conduce a un numero minore di accessi non coalescenti, di utilizzare una Riduzione Parallela Interleaved.

Quando R > 1

Non sono presenti abbastanza thread per i dati con cui si sta lavorando. Conseguentemente, il 100% dei thread del blocco dovranno essere utilizzati. In particolare, se R = 2, allora ciascun thread effettuerà 1 somma. Se R > 2, invece, i thread dovranno computare molteplici somme nella stessa iterazione.

Apportando una leggera modifica alla formula dell'indice R, questo può essere ricalcolato ad ogni iterazione:

$$R_i = \frac{VLTC_{i-1}/2}{blockDim}$$

Dove VLTC è una sigla per "values left to compute", pari a N nella prima iterazione. Quindi, se a runtime ci si trova nella situazione R > 1, allora si ha la certezza di giungere nella situazione R < 1, ovvero nella Riduzione Parallela.

Un’ipotetico scheletro del kernel è già delineabile:

```
__global__ void K2(double* inputData, double* outputData, int N) {

    //inizializzazione delle variabili necessarie, come R, VLTC e idx
    //...

    while (R > 1) {
        //Il 100% dei thread è al lavoro
        //Saranno eseguite 1 o più somme per ciascun thread

        valuesLeftToCompute /= 2;
        R = ... //ricalcolo R
    }

    //Qui, R è <= 1. Pertanto si avvia la Riduzione Parallela.
        //Meno del 100% dei thread è al lavoro
        //Dei thread disponibili, quelli attivi eseguiranno una somma
}
```

4.2.2 L’Algoritmo Completo

Di seguito è riportato il codice di K2, la cui struttura è simbolicamente identica all’ipotesi:

```
__global__ void K2(double* inputData, double* outputData, int N) {
    //Se fosse appurato N < blockDim.x, allora N sarebbe da sostituire con quest'ultimo in global_index
    int global_index = (blockIdx.x * N) + threadIdx.x;
    int blockValuesLeftToCompute = N;
    int sumsToComputeThisCyclePerThread;
    int iteration = 1, x, stride;

    //For che scorre fintanto che rimangono da computare tanti elementi quanti i thread disponibili
    for (stride = blockDim.x; blockValuesLeftToCompute > blockDim.x; stride *= 2) {
        //Se si è qui, il 100% dei thread lavora
        sumsToComputeThisCyclePerThread = (blockValuesLeftToCompute / 2) / blockDim.x;
        x = 0;
        //Ogni thread deve calcolare TOT somme in questo ciclo. Per farlo, si usa il seguente for.
        for (int sumsDone = 0; sumsDone < sumsToComputeThisCyclePerThread; sumsDone++) {
            /*
            * GLOBAL_INDEX -> individua il punto dell'array di input cui questo thread parte
            * STRIDE -> individua la distanza tra i due addendi
            * X -> individua il salto che un thread deve compiere per passare da coppia a coppia
            */
            inputData[global_index + x * blockDim.x] += inputData[global_index + x * blockDim.x + stride];
            x += powf(2, iteration);
        }
        //Aggiorno le variabili di controllo
        blockValuesLeftToCompute /= 2;
        iteration++;
    }
    //Giunti qui, ci sono esattamente tanti thread quanti blockValuesLeftToCompute
    if (threadIdx.x == 0) {
        InterleavedReduction <<<1, blockValuesLeftToCompute / 2 >>> (inputData, blockValuesLeftToCompute / 2, outputData, global_index, blockIdx.x);
    }
}
```

```
__global__ void InterleavedReduction(float* inputData, int strideI, float* outputData, int global_index, int blockId) {

    for (int stride = strideI;stride > 0; stride >>= 1) {
        // Ogni thread attivo somma due elementi separati dallo stride corrente
        if (threadIdx.x < stride) {
            // Il thread somma l'elemento alla sua posizione con quello a distanza 'stride'
            inputData[global_index + threadIdx.x] += inputData[global_index + threadIdx.x + stride];
        }
        __syncthreads(); // Sincronizzazione da parte dei thread che non lavorano
    }

    if (threadIdx.x == 0) {
        //salva il risultato finale
        outputData[blockId] = inputData[global_index];
    }
}
```

Partendo dall'alto:

1. Si inizializzano le variabili necessarie
2. Si esegue un ciclo for in cui, ad ogni giro, viene calcolato il numero di somme che ogni thread dovrà compiere
3. Tramite il secondo ciclo for, ci si assicura che tale numero di somme sia eseguito da ogni thread
4. Una volta giunti all'iterazione tale che il numero di addendi da sommare sia pari alle dimensioni del blocco, si procede ad invocare un kernel secondario per la Riduzione Parallela

Attraverso un esempio:

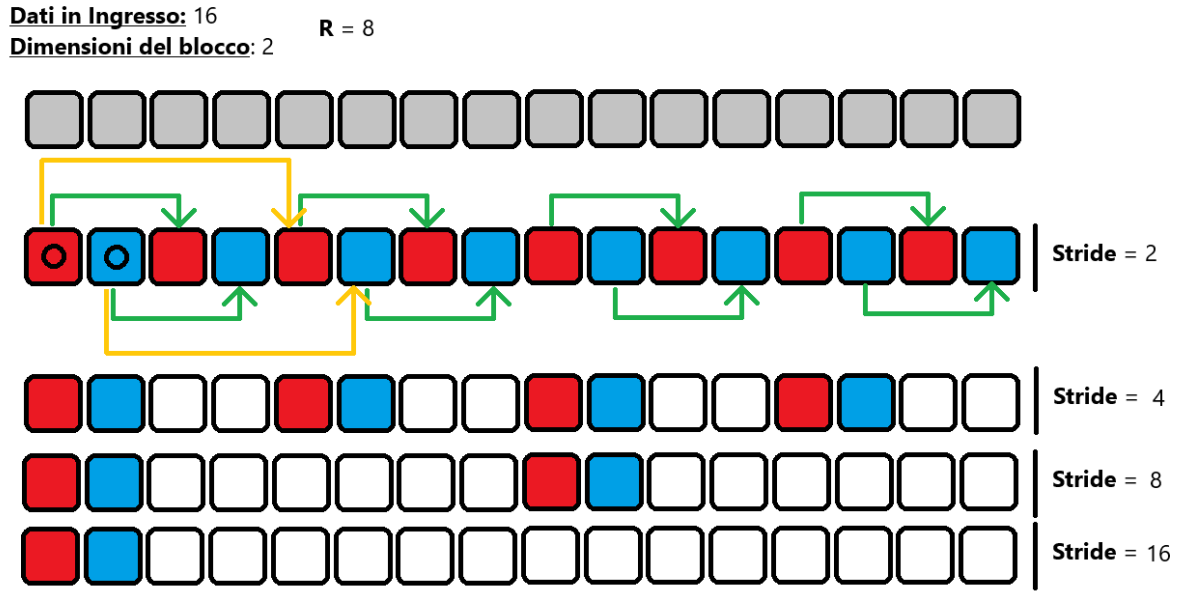


Immagine esplicativa del funzionamento dell'algoritmo "pre-riduzione" di K2. Le frecce verdi rappresentano lo stride, quelle gialle rappresentano il funzionamento di x.

4.3 Primi Risultati di K2

```
PS E:\Microsoft Onedrive\OneDrive\Desktop\ProgettoSistemiDigitali\results> .\K2Float.exe
Starting...
Using Device 0: NVIDIA GeForce RTX 2060
Series generated
Beginning computation for 8192 series of 8192 numbers each
It has been determined that 8192 blocks will be created, each containing 1024 threads
On average, the generation time is 0.00906849 sec
K2 elapsed 0.009068 sec
```

Risultato del file "K2Float.cu"

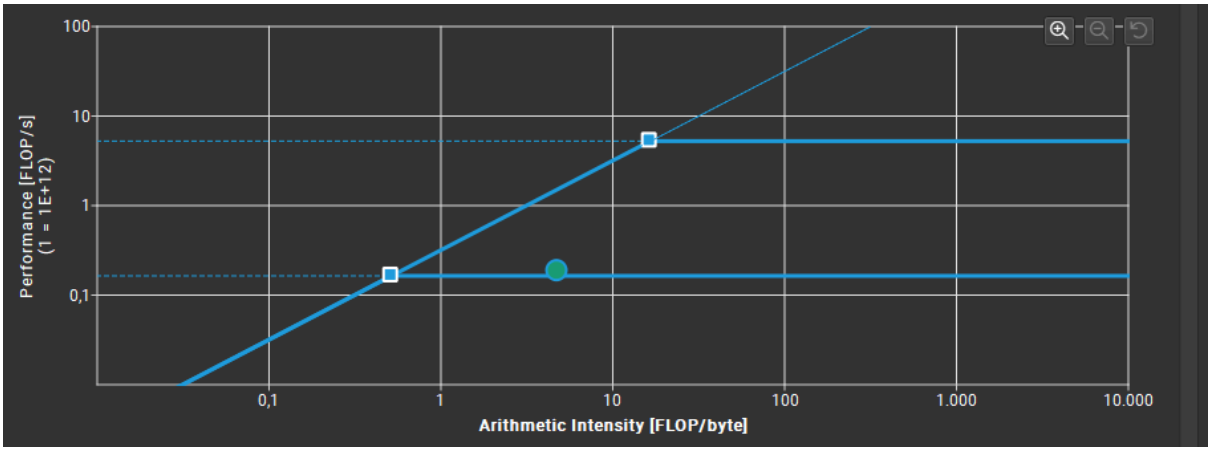
Basandosi sui CPU timer, già con il semplice utilizzo della GPU si è osservato uno speedup di:

$$\frac{0.2666}{0.009068} \sim 29$$

NVIDIA Nsight Systems				
Name	Time %	Total Time	Avg Time	Number of Calls
CudaDeviceSynchronize	44	594.87 ms	5.89 ms	100
K2	36	479.27 ms	4.79 ms	100
CudaMemcpy	9	120.53 ms	1.182 ms	102
CudaLaunchKernel	0	12.107 ms	119.87 µs	100
CudaFree	0	1.354 ms	677.1 µs	2
CudaMalloc	0	837.49 µs	418.74 µs	2

Come ci si potrebbe aspettare, la quasi totalità del tempo di esecuzione viene occupata da K2 e dalle operazioni di sincronizzazione host-device, in quanto il kernel viene lanciato 100 volte, al termine di ciascuna delle quali si sincronizza.
Effettuare soltanto 1 prova non sarebbe sufficiente a scalzare K2 e CudaDeviceSynchronize dalla cima della classifica di impatto sul tempo di computazione.

NVIDIA Nsight Compute					
Kernel Name	Thread Lanciati	Istruzioni Eseguite	Memory Through.	Branch Efficiency	Duration
K2	8.4 M	124.5 M	38.4 GB/s	99.85%	7.47 ms



Roofline Diagram di “K2Float.cu”

Dallo strumento roofline, si nota ampio margine di miglioramento. Tuttavia, si può già dire che il kernel **K2 è memory bound**.

L’analisi con gli strumenti del *CUDA-Toolkit* ha rivelato che la maggior parte dei miglioramenti applicabili riguardano i pattern di accesso alla memoria, l’utilizzo improprio della lettura a banchi delle cache e la latenza di accesso alla memoria globale.

4.4 K2 Full Shared e K2 Intense

Nel tentativo di risolvere la latenza dovuta all'utilizzo di memoria globale, si è fatto uso della **memoria shared**. Sorge però un problema: il kernel invocato da K2, che esegue la Riduzione Parallela, non ha visione della memoria shared allocata da K2.

Si è pertanto:

- Adoperato la memoria condivisa in K2
- Una volti giunti al punto di invocare il kernel figlio, copiato i valori necessari a questi per operare dalla shared memory alla memoria globale
- Dentro il figlio, copiato i valori appena inseriti nella memoria globale nella memoria shared del figlio ed adoperato questi ultimi

Per via di questa doppio caricamento-scaricamento di dati dalla memoria globale prima e dopo l'avvio del kernel innestato, tutti i vantaggi che si sarebbero ottenuti dall'utilizzo della memoria shared stessa sono stati vanificati. Questa versione del kernel K2 è stata chiamata *Full Shared K2*.

Per poter usare la memoria shared in maniera efficiente, si è abbandonata l'idea del kernel innestato. Questo ha aumentato il numero di istruzioni presenti nel codice di K2, che in questa sua versione è detto *Intense K2*. In quanto la differenza tra *Full Shared K2* e *Intense K2* è minima ed il primo di questi non ha ottenuto risultati differenti da K2, si è deciso di **non allegare al progetto il codice di *Full Shared K2***.

```
PS E:\Microsoft Onedrive\OneDrive\Desktop\ProgettoSistemiDigitali\results> .\K2IntenseFloat.exe
Starting...
Using Device 0: NVIDIA GeForce RTX 2060
Series generated
Beginning computation for 8192 series of 8192 numbers each
It has been determined that 8192 blocks will be created, each containing 1024 threads
K2 elapsed 0.001846 sec
```

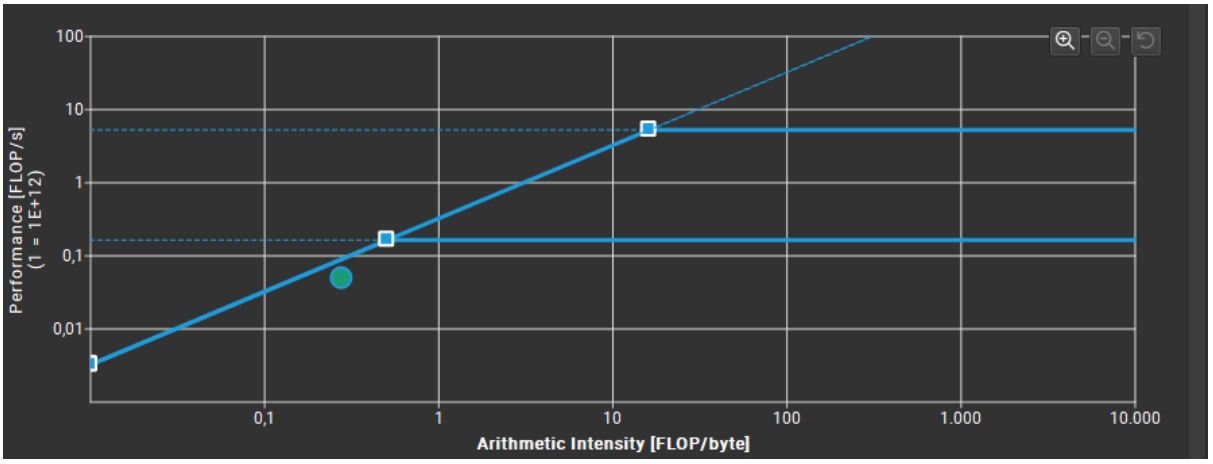
Output del file "K2IntenseFloat.cu"

Si è quindi osservato uno speedup rispetto alla prima versione di K2 pari a:

$$\frac{0.009068}{0.001846} \sim 5$$

Riguardo le performance di *Intense K2*, Nsight Compute riporta:

NVIDIA Nsight Compute					
Kernel Name	Thread Lanciati	Istruzioni Eseguite	Memory Through.	Branch Efficiency	Duration
K2	8.4 M	62.2 M	173 GB/s	100%	1.57 ms



Roofline Diagram di “K2IntenseFloat.cu”

Il punto sul grafico risulta essere più a destra e molto più vicino al tetto rispetto a quello visibile nel diagramma roofline della prima versione di K2.

4.5 K2 Intense Optimized

Nel tentativo di rendere ancora più efficiente il kernel *K2 Intense*, si è cercato di adottare alcune misure che potessero agire sui due cicli *for* che compongono la quasi totalità del codice.

Di questi:

- Il primo ciclo si occupa di caricare i dati in shared memory ed effettuare le computazioni necessarie per ricondursi al caso di Riduzione Parallela
- Il secondo ciclo effettua la Riduzione Parallela secondo il modello Interleaved

Data la natura del problema, secondo cui non è noto il numero di valori che compongono ciascuna serie, non è possibile dedurre a priori le iterazioni che il primo ciclo compierà. Di conseguenza, le ottimizzazioni che seguiranno sono state applicate unicamente al ciclo che effettua la Riduzione Parallela.

Misure intraprese:

- Loop unrolling nel ciclo *for* di Riduzione Parallela al fine di ridurre il numero di volte in cui la condizione del *for* viene rivalutata

```
if(sizeBlock >= 1024 && threadIdx.x < 512){
    s[threadIdx.x] += s[threadIdx.x + 512];
}
__syncthreads();
if(sizeBlock >= 512 && threadIdx.x < 256){
    s[threadIdx.x] += s[threadIdx.x + 256];
}
__syncthreads();
if(sizeBlock >= 256 && threadIdx.x < 128){
    s[threadIdx.x] += s[threadIdx.x + 128];
}
__syncthreads();
if(sizeBlock >= 128 && threadIdx.x < 64){
    s[threadIdx.x] += s[threadIdx.x + 64];
}
```

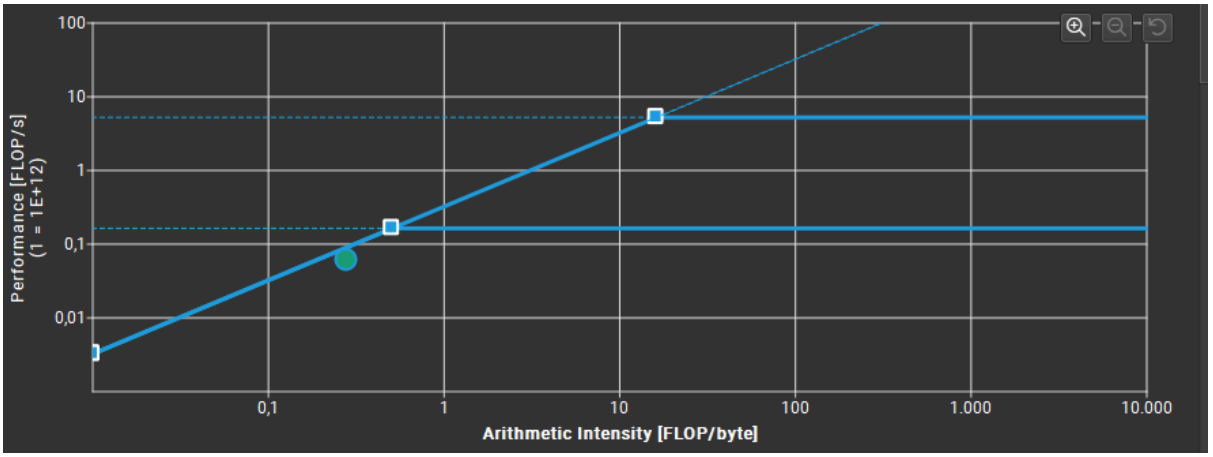
- Unrolling a livello di warp nel ciclo *for* di Riduzione Parallela per evitare di sincronizzare quando non necessario: se il numero di thread rimasti è racchiuso in soltanto un warp, questi sono già sincronizzati tra loro.

```
//Unrolling a livello di warp
if(threadIdx.x < 32){
    volatile float *v_s = s;
    v_s[threadIdx.x] += v_s[threadIdx.x + 32];
    v_s[threadIdx.x] += v_s[threadIdx.x + 16];
    v_s[threadIdx.x] += v_s[threadIdx.x + 8];
    v_s[threadIdx.x] += v_s[threadIdx.x + 4];
    v_s[threadIdx.x] += v_s[threadIdx.x + 2];
    v_s[threadIdx.x] += v_s[threadIdx.x + 1];

}
```

Riguardo le performance di *Intense Optimized K2*, Nsight Compute riporta:

NVIDIA Nsight Compute					
Kernel Name	Thread Lanciati	Istruzioni Eseguite	Memory Through.	Branch Efficiency	Duration
K2	8.4 M	42.2 M	213 GB/s	100%	1.28 ms



Roofline Diagram di “K2IntenseFloatOptimized.cu”

4.6 Appendice sui blocchi in K2

Come già accennato, nella realizzazione del kernel K2 si è supposto venisse lanciato un numero di blocchi pari al numero delle serie numeriche. Tuttavia, questa è una decisione puramente arbitraria. Si possono lanciare anche 2, 4 o 8 blocchi per ciascuna serie, purché questi rispettino la Regola.

Una versione di K2 con molteplici blocchi dedicati alla stessa serie non è stata implementata, ma è stata discussa in maniera teorica. Si è concluso che:

- Se fossero lanciati B blocchi per ciascuna serie, ognuno di questi lavorerebbe con $\frac{N}{B}$ valori di questa
- Se $N >$ blockDim, allora potrebbe accadere che $\frac{N}{B} <$ blockDim. Potenzialmente, quindi, usando abbastanza blocchi, sarebbe possibile che ciascuno di questi esegua direttamente la Riduzione Parallela

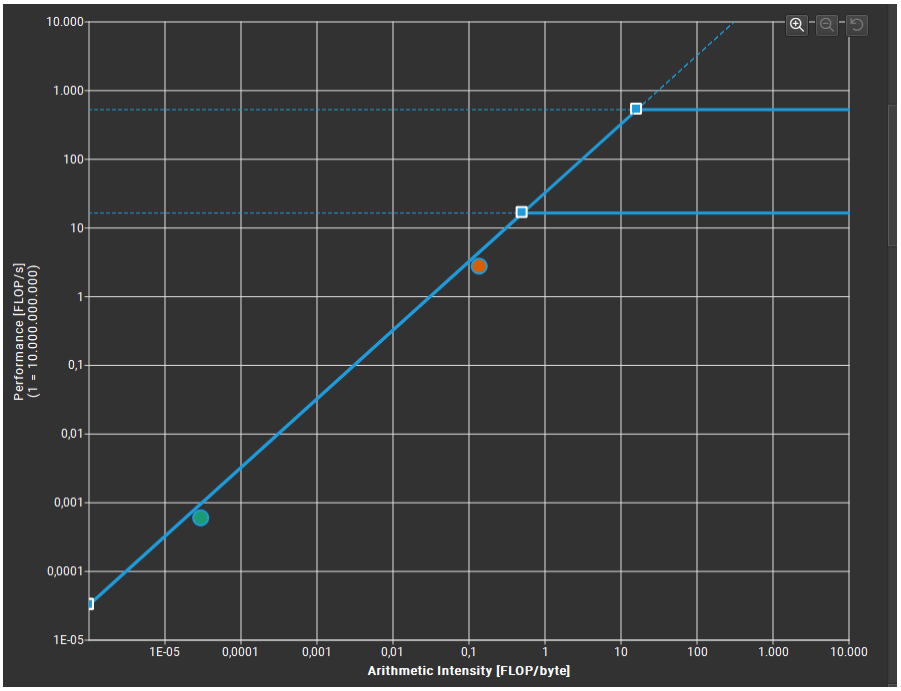
4.7 Appendice sui tipi di dato

Fino a questo momento si sono utilizzati dati di tipo **float**. Tuttavia, durante lo sviluppo, ci si è resi conto che questi limitano di molto le capacità computazionali del kernel. Soltanto con serie di lunghezza limitata il tipo di dato *float* si è rivelato preciso.

```
Series 11: 466808.218750 VS 466807.937500
Series 12: 594004.812500 VS 594005.000000
Series 13: 66812.062500 VS 66811.984375
Series 14: 32822.679688 VS 32822.671875
Series 15: 28324.171875 VS 28324.160156
Series 16: 72586.546875 VS 72586.656250
Series 17: 24697.519531 VS 24697.519531
Series 18: 11286.091797 VS 11286.109375
Series 19: 71078.062500 VS 71078.015625
Series 20: 23745.123047 VS 23745.085938
Series 21: 20104.917969 VS 20104.894531
Series 22: 101614.375000 VS 101614.437500
Series 23: 48876.878906 VS 48876.863281
Series 24: 76517.906250 VS 76517.726562
```

Output del calcolo delle somme di K2 con un N superiore a 4096. Come si vede, le somme non coincidono per via di un piccolo scarto dovuto ad approssimazioni varie

Si è pertanto provato ad utilizzare anche il formato dati **double**. In particolare, è stata realizzata una versione di *Intense K2* che utilizza quest'ultimo formato di dati. Di seguito è riportato a titolo informativo il roofline diagram:



Roofline Diagram di "*K2IntenseDouble.cu*". *Si faccia presente che in questo codice non si fa uso di Shared Memory. Utilizzandola, presumibilmente, si potrebbe persino toccare il tetto.*

Infine, la seguente tabella riassume i risultati delle sperimentazioni fatte con i due formati di dato. Si noti come, al variare del volume dei dati con cui si lavora e del margine di errore accettato, vari la colonna di accuracy.

K2									
Using Floats					Using Double				
M	N	Epsi	Accuracy	Runtime	M	N	Epsi	Accuracy	Runtime
2048	2048	$1.0e^{-1}$	74.41%	0.000520	2048	2048	$1.0e^{-3}$	100%	0.000753
2048	2048	$1.0e^{-2}$	16.65%	0.000513	2048	4096	$1.0e^{-3}$	100%	0.001261
2048	2048	$1.0e^{-3}$	3.86%	0.000538	2048	8192	$1.0e^{-3}$	100%	0.000988
2048	1024	$1.0e^{-3}$	7.37%	0.000291	2048	16384	$1.0e^{-3}$	100%	0.001244
2048	512	$1.0e^{-3}$	11.96%	0.000247	2048	32768	$1.0e^{-3}$	100%	0.002270
2048	256	$1.0e^{-3}$	18.36%	0.000252	2048	65536	$1.0e^{-3}$	100%	0.003607
2048	128	$1.0e^{-3}$	49.32%	0.000877	4096	65536	$1.0e^{-3}$	100%	0.009244
2048	64	$1.0e^{-3}$	89.16%	0.000805	8192	65536	$1.0e^{-3}$	100%	0.015798
2048	32	$1.0e^{-3}$	100%	0.001716	16384	65536	$1.0e^{-3}$	100%	0.519659
4096	32	$1.0e^{-3}$	100%	0.000462	16384	65536	$1.0e^{-6}$	99.77%	0.632978