Università degli Studi di Milano

Facoltà di Scienze e Tecnologie
Dipartimento di Fisica

Corso di Metodi Computazionali della Fisica

Riduzione di Gauss-Jordan su GPU

Studente:
Gianluca Fugante

Docente del corso:
Alessandro Vicini

Anno Accademico 2018/2019

Capitolo 1

Introduzione

In questo breve elaborato viene presentato il progetto personale realizzato per l'esame del corso di Metodi Computazionali della Fisica dell'a.a. 2018-2019. Obbiettivo del progetto è la risoluzione di un generico sistema lineare $N \times N$ (con N molto grande) utilizzando una scheda grafica Nvidia.
Questa relazione si articola in tre parti: nella prima parte verrà presentata la teoria inerente al problema, mentre nella seconda e nella terza saranno descritti rispettivamente gli algoritmi implementati su CPU e GPU (scheda grafica).

Capitolo 2

Premessa teorica

Un generico sistema lineare $N \times M$ definito come:

$$\begin{cases} a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + ... + a_{0,N-1}x_{N-1} = b_0 \\ a_{10}x_0 + a_{11}x_1 + a_{12}x_2 + ... + a_{1,N-1}x_{N-1} = b_1 \\ ... \\ a_{M-1,0}x_0 + a_{M-1,1}x_1 + ... + a_{M-1,N-1}x_{N-1} = b_{M-1} \end{cases}$$

può essere scritto in notazione compatta come:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.1}$$

dove sono stati definiti la matrice dei coefficienti A e i vettori delle soluzioni x e dei termini noti b:

$$\begin{aligned} A &= a_{ij} \\ x &= x_i \\ b &= b_i \end{aligned}$$

dove i e j sono indici che vanno da 0 a $M-1$ e $N-1$ rispettivamente. Definiamo inoltre la matrice completa come:

$$(\mathrm{A|b}) = \begin{pmatrix} a_{00} & a_{01} \ \dots & | \ b_0 \\ a_{10} & a_{11} \ \dots & | \ b_1 \\ & \vdots & \\ a_{M-1,0} & a_{M-1,1} \dots & | \ b_{N-1} \end{pmatrix}$$

In questo progetto ci occuperemo solo di sistemi di N equazioni in N incognite, quindi di sistemi in cui $M = N$. Da qui in avanti considereremo solo questo caso.

Dal teorema di Rouchè-Capelli è noto che:

1. se $rango(A) < rango(A|b)$ allora il sistema non ammette soluzioni.
2. se $rango(A) = rango(A|b) = N$ allora il sistema ammette una e una sola soluzione.
3. se $rango(A) = rango(A|b) < N$ allora il sistema ammette $\infty^{N-rango(A)}$ soluzioni, ovvero una soluzione che dipende da $N - rango(A)$ parametri.

Un sistema lineare non è quindi sempre risolvibile, ma solo quando il rango delle matrice completa è minore o uguale al numero di incognite. Un metodo di risoluzione per tali sistemi è il metodo di riduzione di Gauss-Jordan, in cui si utilizzano opportune combinazioni lineari di equazioni per ridurre la matrice (A|b) a triangolare superiore, ovvero nella forma:

$$(\mathrm{A|b}) = \begin{pmatrix} a'_{00} & a'_{01} & \dots & a'_{0,N-1} & | \, b'_0 \\ 0 & a'_{11} & \dots & a'_{1,N-1} & | \, b'_1 \\ 0 & 0 & \dots & a'_{2,N-1} & | \, b'_2 \\ & \vdots & & & \\ 0 & 0 & \dots & a'_{N-1,N-1} & | \, b'_{N-1} \end{pmatrix}$$

in modo che dall'ultima equazione sia immediato ricavare l'ultima incognita e sostituendo nelle equazioni precedenti si ricavino tutte le altre (*backsubstitution*). Ogni incognita è quindi definita dalla seguente formula:

$$x_i = \frac{1}{a'_{ii}} \left(b'_i - \sum_{j=i+1}^{N-1} a'_{ij} x_j \right) \tag{2.2}$$

Se il sistema è risolvibile abbiamo quindi un modo semplice per ottenere la soluzione. Computazionalmente tuttavia si presentano altri due problemi:

1. due o più equazioni linearmente indipendenti possono avere coefficienti talmente vicini da venire rappresentati dalla macchina come lo stesso numero, rendendo così le equazioni linearmente dipendenti e facendo fallire l'algoritmo.
2. gli errori di arrotondamento commessi durante il procedimento incidono sulla soluzione finale, che può essere molto diversa dalla vera soluzione. L'algoritmo in questo caso converge, ma la soluzione è sbagliata.

Il primo problema è chiaramente risolvibile aumentando la precisione sui coefficienti, utilizzando ad esempio variabili di tipo *double* invece che di tipo *float*.
Il secondo problema, invece, può essere ottimizzato facendo in modo che ad ogni passaggio sensibile agli errori di arrotondamento si commetta l'errore minore possibile. Ricordando che la rappresentazione numerica è più densa vicino allo 0, si può minimizzare l'errore cercando tra le righe che seguono la i-esima (dove $i = 0, ..., N$ è il numero di righe già ridotte) quella che ha l'i-esimo elemento massimo in valore assoluto (detto *pivot*), per poi scambiarla con la i-esima riga. Quando si procederà alla sottrazione di una riga successiva con la i-esima, per annullare il primo elemento basterà scegliere un coefficiente moltiplicativo (da applicare alla i-esima riga) che abbia al numeratore l'i-esimo elemento della riga scelta e al denominatore il pivot, garantendo che questo coefficiente sia il minore possibile. Tale pratica viene detta *pivoting*.

Capitolo 3

Riduzione di Gauss-Jordan su CPU

3.1 Descrizione dell'algoritmo

L'algoritmo su CPU si articola nei seguenti passaggi:

- Creazione della matrice
- Riduzione
- Calcolo delle soluzioni
- Stima dell'errore

La prima fase è quella di **creazione**, in cui viene creata la matrice $N \times N$ con elementi random in un intervallo fissato utilizzando il generatore pseudo-random uniforme della Standard Template Library. In questa fase viene creato anche il vettore dei termini noti, riempito con N elementi uguali. È necessario salvare una copia sia della matrice che del vettore dei termini noti, che verrano usati per il calcolo dell'errore. Le due matrici sono allocate sotto forma di vettore, in quanto l'accesso ai loro elementi sarà più veloce.

Una volta creati gli oggetti necessari si può passare alla **riduzione** della matrice. Questa fase consiste in un ciclo di $N - 1$ iterazioni in cui:

– si cerca la riga con i-esimo elemento massimo in valore assoluto (dove $i = 0, ..., N - 1$ è l'indice di iterazione)

– si scambiano gli elemnti della riga del pivot con quelli della i-esima riga

– sia nella matrice che nel vettore di termini noti si sottrae la i-esima riga, moltiplicata per un coefficiente opportuno, a tutte le righe successive

– si annullano tutti gli elemnti delle righe successive che hanno indice di colonna strettamente minore di i

dove l'ultimo passaggio è giustificato dal fatto che è noto a priori che gli elemnti del triangolo inferiore saranno nulli, ma numericamente la sottrazione di due numeri uguali può restituire un valore non nullo.

Una volta ottenuta la matrice in forma triangolare superiore il **calcolo delle soluzioni** è eseguito con la (2.2).

Infine la **stima dell'errore** è fatta ricalcolando i termini noti, quindi eseguendo il prodotto matrice-vettore tra la matrice iniziale (di cui è stata salvata una copia) e il vettore delle soluzioni. Questo nuovo vettore di termini noti viene sottratto ai termini noti iniziali e l'errore è quotato come elemento massimo di questa differenza.

3.2 Risultati

Per valutare le prestazioni dell'algoritmo sono stati misurati l'errore e il tempo di esecuzione in funzione di N al variare di diversi parametri. In particolare vengono modificati il range degli elementi della matrice (da $0-10$ a $0-10^6$) e il valore dei termini noti (da 1 a 10^6) considerando tutte le possibili combinazioni. Viene inoltre cambiato anche il tipo di dato utilizzato (*double* o *float*).

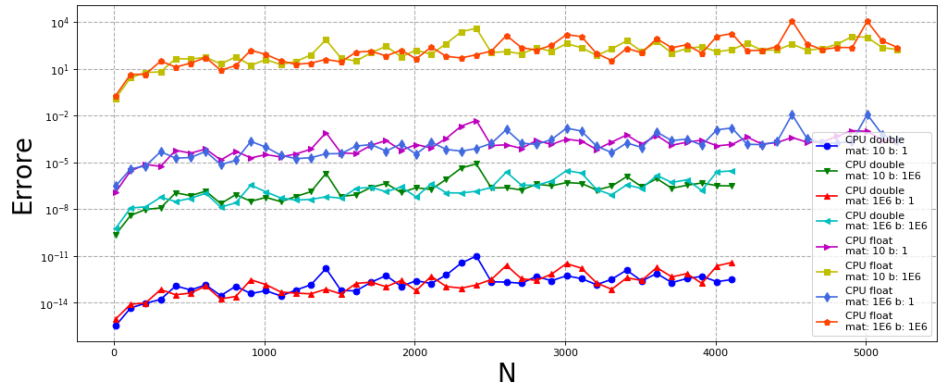

Figura 3.1: Grafico dell'errore in scala logaritmica

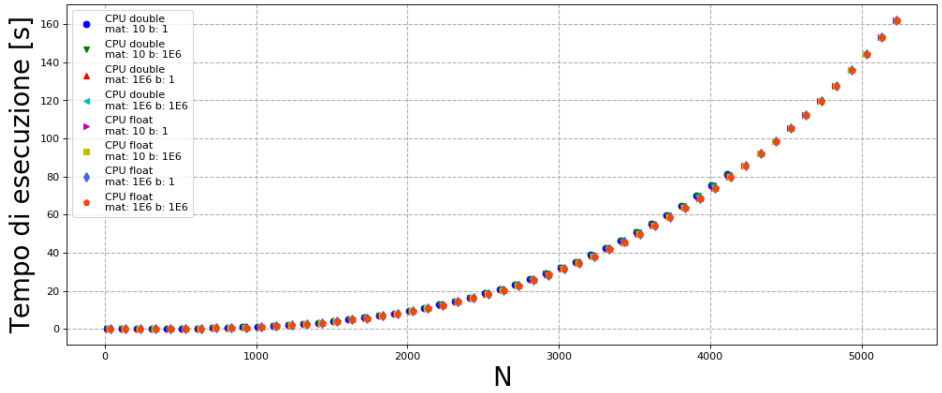

Figura 3.2: Grafico del tempo di esecuzione, le diverse serie sono rappresentate aggiungendo un offset sull'asse delle x per visualizzarli separatamente

Dal primo grafico si deduce che l'errore dipende sensibilmente dall'ordine di grandezza dei termini noti e dal tipo di dato utilizzato, come ci si aspetta, ma non dal range degli elementi della matrice.

Sui dati ottenuti per il tempo di esecuzione è stato inizialmente eseguito un fit con un polinomio di secondo grado, che però restituisce un χ^2_{rid} di $9,00$. È stato quindi eseguito un secondo fit con un polinomio di terzo grado che restituisce i seguenti parametri:

x^3	x^2	x^1	x^0	χ^2_{rid}
$1,10 \cdot 10^{-9}$	$3,24 \cdot 10^{-7}$	$-4,00 \cdot 10^{-4}$	$1,09 \cdot 10^{-1}$	$0,07$

In figura sono riportati i due fit:

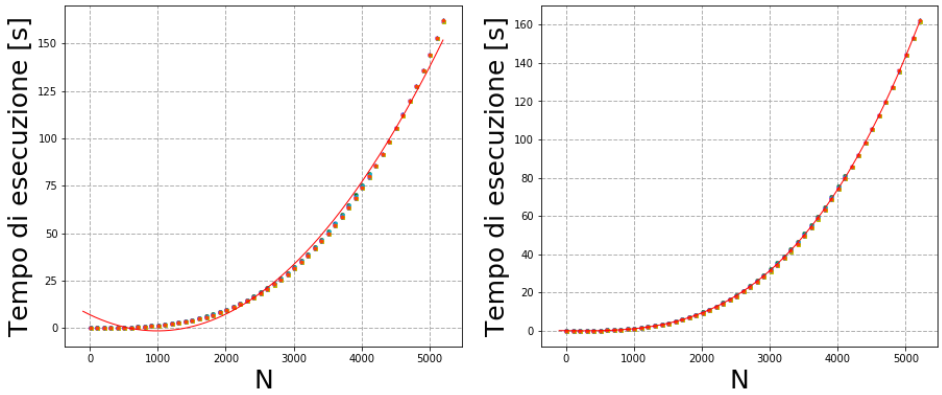

Figura 3.3: Fit del tempo di esecuzione: a sinistra con un polinomio di secondo grado, a destra con un polinomio di terzo grado

Il processo di sottrazione precedentemente descritto è infatti di ordine 3, all'i-esima iterazione ($i = 0, ..., N-1$) si scorrono $N - i - 1$ righe di N elementi e sommando:

$$\sum_{i=0}^{N-1} N(N-1-i) = N \cdot \sum_{k=0}^{N-1} k = N \cdot \frac{(N-1)N}{2} = \frac{N^3}{2} - \frac{N^2}{2} \qquad (3.1)$$

Capitolo 4

Riduzione di Gauss-Jordan su GPU

4.1 Scheda grafica

Una scheda grafica (o GPU, da *Graphics Processing Unit*) è un co-processore che viene utilizzato per ridurre il tempo di esecuzione di alcuni programmi, in particolare trova importanti applicazioni nella gestione della grafica video, ma viene utilizzata anche in campo scientifico.

Una tipica scheda grafica è composta da un circuito integrato che comprende una memoria dedicata e centinaia o migliaia di core, che permettono di parallelizzare determinati processi e quindi di guadagnare tempo. In figura è rappresento un confronto tra la struttura hardware rispettivamente di CPU e GPU.

Figura 4.1: Schema della struttura hardware di CPU e GPU

Il linguaggio utilizzato per programmare sulle schede NVIDIA a nostra

disposizione è il linguaggio CUDA (*Compute Unified Device Architecture*). Questo linguaggio definisce una struttura virtuale basata sui *threads*, ovvero oggetti separati che processano la stessa porzione di codice parallelamente. I threads sono indicizzati e raggruppati in blocchi, in cui tipicamente si possono istanziare fino a 1024 threads per blocco, per decine di migliaia di blocchi, a loro volta indicizzati e raccolti in una griglia. Sfruttando l'indicizzazione si può indirizzare diversi threads a diverse aree di memoria o all'esecuzione di porzioni di codice diverse. Tale indicizzazione può essere 1D, 2D o 3D per i threads e 1D o 2D per i blocchi.

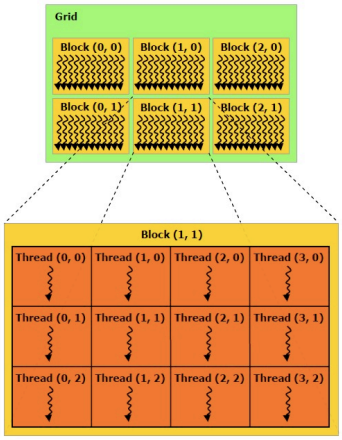

Figura 4.2: Schema della struttura di threads e blocchi

La scheda grafica dispone inoltre di diversi tipi di memoria:

- memoria di registro: di pochi kB, ma di rapido accesso. È visibile solo dal thread che l'ha allocata.
- memoria locale: usata quando la memoria di registro è esaurita, ha lo stesso scope, ma l'accesso è più lento.
- memoria condivisa: visibile a tutti i thread in un blocco. Di rapido accesso, ma limitata a pochi kB (più grande rispetto a quella di registro).
- memoria globale: visibile a tutti i threads dell'applicazione, ha diversi GB di memoria, ma l'accesso è molto lento.
- memoria costante: è una memoria come quella globale, ma è cached e di sola lettura.

– memoria texture: è una memoria come quella costante, ma ottimizzata per l'accesso 2D.

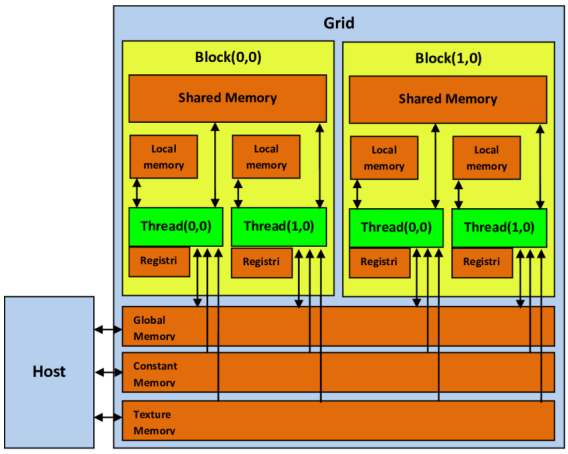

Figura 4.3: Schema delle diverse memorie

La velocità di accesso alle varie memorie è un parametro molto importante per minimizzare il tempo di esecuzione di un algoritmo, è quindi conveniente utilizzare le memorie ad accesso rapido, ma soprattutto è importante evitare il più possibile copie da o verso l'host, che ha tempi di accesso molto maggiori di quelli sopra elencati. Spesso è quindi conveniente continuare a lavorare sulla scheda anche quando si svolgono compiti non parallelizzabili e più in generale è buona norma copiare tutti i dati necessari nella memoria opportuna prima di eseguire i vari kernel.

In questo progetto è stata utilizzata una scheda GeForce GTX 480.

4.2 Descrizione dell'algoritmo

L'algoritmo su GPU segue gli stessi passaggi di quello su CPU, ma è necessario prima allocare lo spazio necessario sulla scheda per le matrici, i vettori e le variabili che serviranno durante la riduzione. Questo spazio viene allocato nella memoria globale.

La **creazione** della matrice e del vettore di termini noti (e relative copie) è eseguito a blocchi: vengono definiti N blocchi di 1024 threads, ogni blocco gestisce una singola riga e ogni thread carica un elemento della riga con un numero random ottenuto con il generatore uniforme cuRAND. Se il

numero di elementi di una riga eccede 1024, allora l'indice del thread viene incrementato di 1024, così che, ad esempio, il trhead di indice 0 gestisce non solo il primo elemento della riga, ma anche il 1025-esimo e così via. Facendo lavorare due volte un thread si ha ovviamente un tempo di esecuzione doppio e in generale si ottiene un grafico a gradini. Utilizzando più blocchi si pone anche il problema di poterli sincronizzare. CUDA, infatti, prevede un comando solo per la sincronizzazione dei threads in un blocco, ma non uno per sincronizzare diversi blocchi. È quindi necessario creare più kernel (e.g. uno per la creazione, uno per il pivoting etc.), così una volta usciti dal singolo kernel si è sicuri che i vari blocchi siano sincronizzati.

La fase di **riduzione** della matrice consiste ancora in un ciclo di $N-1$ iterazioni, in cui prima si applica il pivoting e poi le sottrazioni necessarie. Per la ricerca del pivot vengono istanziati N threads distribuiti in più blocchi (in modo che ogni thread venga eseguito il minor numero di volte possibile), ogni thread gestirà un singolo elemento della colonna in esame. Tali elementi vengono opportunamente caricati nelle varie memorie shared (i primi 1024 nel primo blocco, i successivi nel secondo e così via) e ogni blocco opera un confronto telescopico: al primo passaggio si confrontato parallelamente i primi 512 elementi con i successivi 512, scambiandoli se necessario, in modo da ottenere nelle prime 512 posizioni i nuovi candidati massimi per poi dimezzare di nuovo il numero di elementi (questa volta 256) e ripetere l'operazione. Alla fine il candidato massimo di ogni blocco si troverà nel primo elemento dei vettori di ogni memoria shared e il kernel termina, tra questi candidati si trova il massimo assoluto con un confronto semplice. A questo punto le righe vengono scambiate da un kernel si N threads divisi in $1024//N+1$ blocchi (dove il simbolo $//$ indica la divisione intera), appoggiandosi a variabili temporane allocate nella memoria di registro. Le sottrazioni sono invece effettuate da un kernel con $N-i-1$ blocchi (dove i è l'indice di iterazione) in cui ogni blocco gestisce una singola riga. Come nell'algoritmo su CPU in questa fase si mettono a 0 gli elementi del triangolo inferiore.

Una volta finito il ciclo la matrice si presenta in forma triangolare superiore e si può passare al **calcolo delle soluzioni**. Questo compito non è parallelizzabile, infatti è necessario conoscere le incognite precedenti per trovare la successiva, ma viene comunque eseguito sulla scheda per evitare di copiare la matrice sull'host.

Note le soluzioni il **calcolo dell'errore** è effettuato come su CPU: si ricalcolano i termini noti e si quota come errore la massima differenza tra questi e quelli iniziali. Il calcolo dei nuovi termini noti viene eseguito da N blocchi di 1024 threads, in cui ogni blocco gestisce una riga della matrice per farne il prodotto riga per colonna con il vettore delle incognite. Come

già fatto per la ricerca del pivot si carica il valore assoluto della differenza tra il vettore dei termini noti appena ottenuto e quello iniziale nelle memorie condivise di $1024//N + 1$ blocchi, e in ogni blocco si applica un confronto telescopico.

L'algoritmo così implementato ha una sola limitazione: N non può eccedere il massimo numero di blocchi istaziabile, che nel nostro caso è 65536. Tuttavia vedremo che l'algoritmo si ferma molto prima di $N = 65536$ per via della saturazione della memoria.

4.3 Risultati

Le performance dell'algoritmo sono state valutate con lo stesso metodo utilizzato per la CPU, sono quindi stati fatti variare il range degli elementi di matrice, il valore dei termini noti e il tipo di dato.

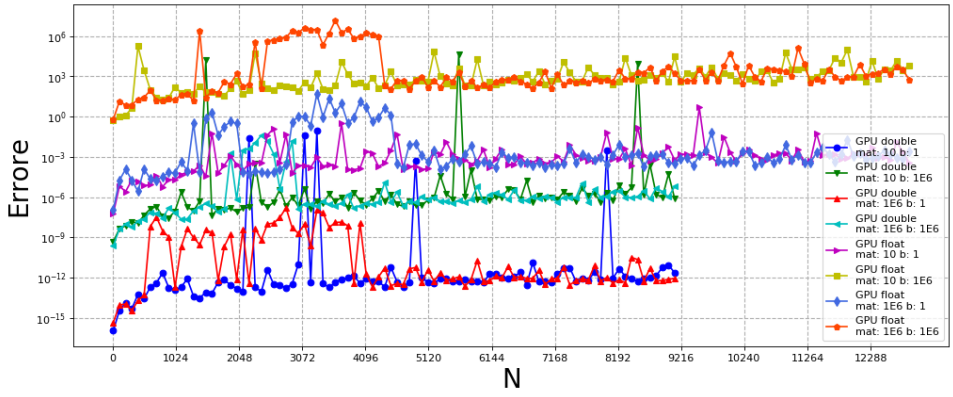

Figura 4.4: Grafico dell'errore in scala logaritmica

La precisione è coerente con quanto atteso: l'andamento è simile a quello ottenuto su CPU, ma molto più variabile.

I seguenti grafici rappresentano i tempi di esecuzioni tipici di ogni passaggio descritto precedentemente, in particolare i dati sono ottenuti con variabili di tipo float, con elementi di matrice nel range $0 - 10^6$ e termini noti pari a 1:

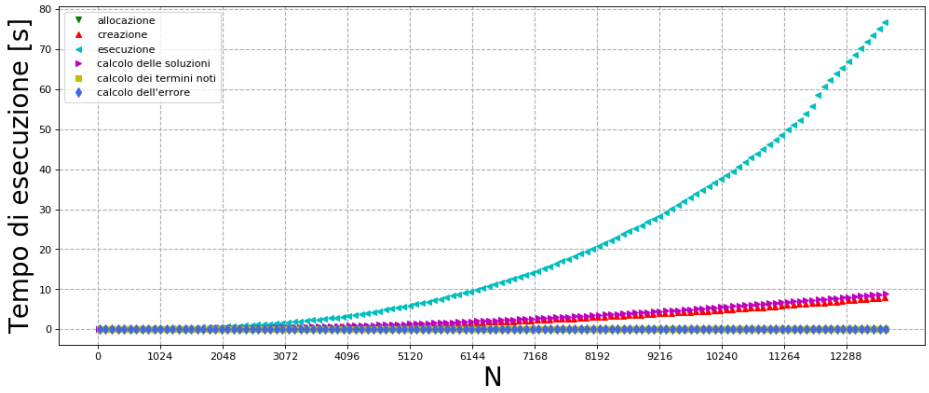

Figura 4.5: Grafico del tempo di esecuzione di ogni kernel

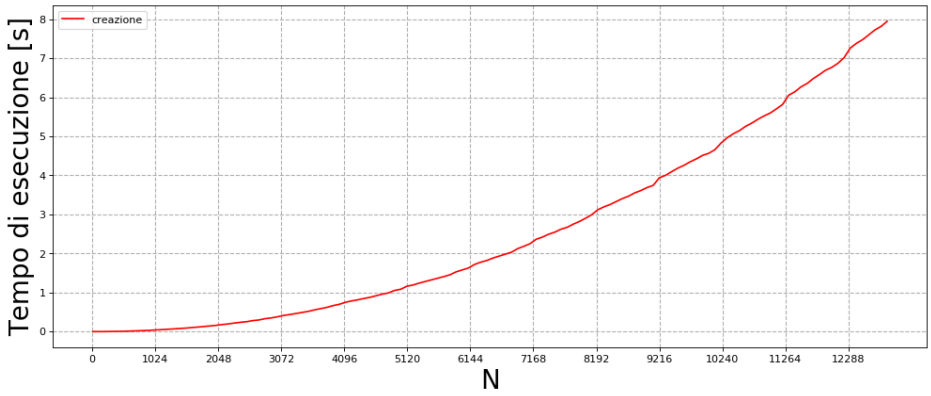

Figura 4.6: Grafico del tempo di esecuzione del kernel per la creazione della matrice

Dalla figura 4.5 si deduce che il tempo di esecuzione è essenzialmente quello necessario per ridurre la matrice, mentre dalla figura 4.6 si nota che l'andamento non è a gradini come ci si aspetta. Probabilmente aumentando il numero di threads aumenta anche il tempo massimo di esecuzione e quindi i gradini non sono ben definiti.

Il seguente grafico mostra i tempi di esecuzione totali per le stesse combinazioni di parametri usate su CPU:

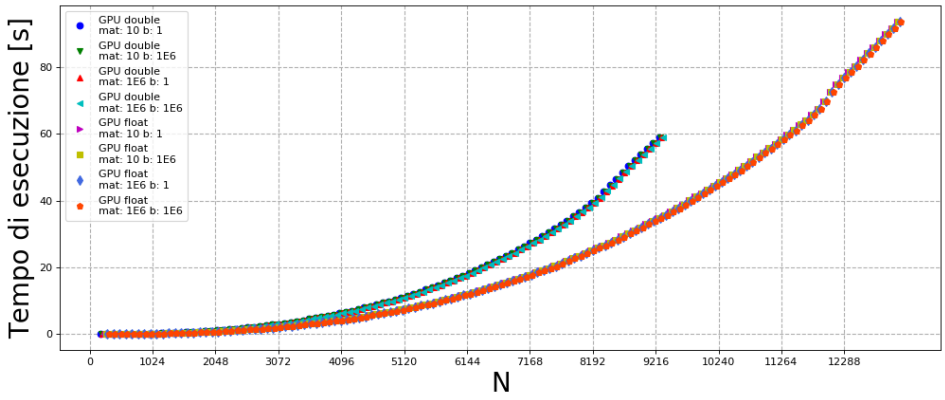

Figura 4.7: Grafico del tempo di esecuzione totale per varie combinazioni di parametri

In questo caso, a differenza dell'esecuzione su CPU, l'andamento è di tipo quadratico, infatti nella riduzione lo scorrimento delle righe è parallelizzato. Resta tuttavia una significativa differenza tra *float* e *double*, dovuta al modo differente di gestire tipi di dato diversi sulla scheda. In tabella sono riportati i parametri dei fit:

dato	x^2	x^1	x^0	χ^2_{rid}
float	$7,55 \cdot 10^{-7}$	$-3,15 \cdot 10^{-3}$	$3,57$	$2,00$
double	$9,75 \cdot 10^{-7}$	$-3,06 \cdot 10^{-3}$	$2,50$	$1,66$

Dal grafico si deduce anche che la massima grandezza della matrice raggiungibile è molto minore del numero di blocchi istanziabili. Questo è dovuto al fatto che la memoria globale viene saturata, ovviemante più velocemente per i *double* che per i *float*.

Il grafico riporta la percentuale di memoria occupata e la percentuale di memoria attesa:

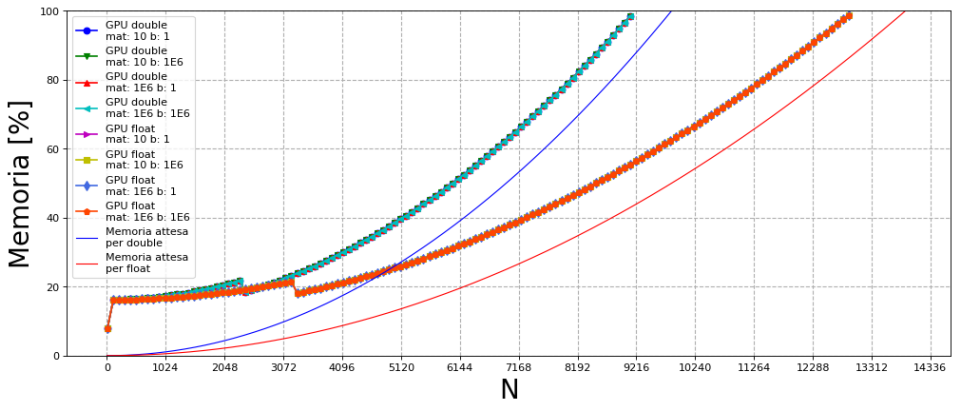

Figura 4.8: Grafico della memoria occupata su GPU

La differenza tra i dati e le percentuali attese è probabilmente dovuta a un'allocazione poco organizzata della memoria, ma il gap iniziale non è del tutto capito.

Capitolo 5

Conclusioni

La riduzione di Gauss-Jordan è stata implementata con successo sia su CPU che su GPU, ma con significative differenze. Prima di tutto il tempo di esecuzione, come atteso, è stato notevolmente ridotto con l'utilizzo della scheda grafica, ma risulta differente quando vengono usate variabili di tipo *float* o *double*. Il grafico riporta il confronto tra i diversi tempi di esecuzione:

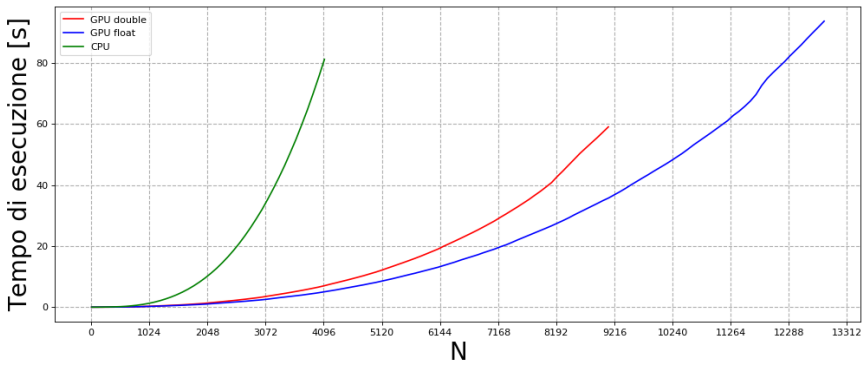

Figura 5.1: Confronto dei tempi di esecuzione

L'errore, invece, ha un andamento simile a quello della CPU, ma molto più variabile.

Concludiamo quindi che è possibile usare l'algoritmo sviluppato su GPU anche per sistemi lineari molto più grandi di quelli possibili su CPU, ma non è sempre garantita la stessa precisione.