UNIVERSITÀ DEGLI STUDI DI ROMA TOR VERGATA
MACROAREA DI INGEGNERIA

CORSO DI STUDIO IN
Ingegneria Informatica

TESI DI LAUREA IN
Sistemi di calcolo parallelo e applicazioni

TITOLO
Algoritmi in precisione mista per sistemi lineari su acceleratori GPU

Relatore:
Chiar.mo Prof.
Salvatore Filippone

Laureando:
matricola: 0324525
Simone Staccone

Anno Accademico 2023/2024

Indice

Abstract

Nel calcolo scientifico moderno è ormai consueto l'utilizzo di algoritmi iterativi per la risoluzione di sistemi lineari. In particolare, l'attenzione nell'ambito di ricerca si è spostata nel ottimizzazione di questi algoritmi, data la necessità sempre maggiore di risolvere sempre più sistemi e sempre più grandi. La tendenza delle aziende più grandi è quella di utilizzare hardware dedicati come FMA (Fused Multiply-Add) o TPU (Tensor Processing Unit), andando ad utilizzare formati di rappresentazione dei dati con meno bit per rendere la computazione più veloce a discapito dell'accuratezza del risultato. Risulta quindi interessante studiare come l'utilizzo di formati di rappresentazione dati diversa sia applicabile all'hardware più comune per il calcolo scientifico in virgola mobile, ovvero per le GPU (Graphical Processing Unit). L'idea sviluppata in questa tesi è quella di utilizzare un approccio misto dei formati di rappresentazione dei dati, andando a studiare come questo renda l'utilizzo dei solutori iterativi più veloce, cercando di mantenere un'accuratezza quantificabile.

Capitolo 1

Introduzione

1.1 Analisi Numerica

1.1.1 Panoramica generale

L'analisi numerica è un ramo della matematica che si occupa dello sviluppo, analisi e applicazione di algoritmi per risolvere problemi matematici in forma approssimata [1]. A differenza dei metodi analitici, che forniscono soluzioni esatte, l'analisi numerica si concentra sull'ottenimento di soluzioni numeriche che approssimano il risultato con un grado di accuratezza prefissato. Questo approccio è essenziale quando le soluzioni esatte non possono essere trovate facilmente, o sono impossibili da ottenere, come nel caso di equazioni differenziali complesse, sistemi di equazioni lineari di grandi dimensioni o integrali non risolvibili analiticamente.

Definizione 1.1.1 (Analisi Numerica)**.** L'analisi numerica è un settore disciplinare che studia le tecniche e le procedure di calcolo (dette complessivamente calcolo numerico) per la soluzione approssimata (detta anche soluzione numerica) di problemi ambientati in un insieme numerico continuo

Esiste una vasta gamma di problemi affrontati con questa metodologia, come la risoluzione di equazioni algebriche e trascendenti, il calcolo di derivate e integrali, la soluzione di sistemi di equazioni lineari e non lineari, la ricerca di autovalori e autovettori, e l'approssimazione di funzioni. Al centro di questa disciplina ci sono due concetti chiave: la precisione dei calcoli e la stabilità degli algoritmi. La precisione si riferisce a quanto un risultato approssimato si avvicini alla soluzione esatta, mentre la stabilità descrive il comportamento dell'errore durante il processo di calcolo, in particolare come piccoli errori iniziali possono propagarsi e influenzare il risultato finale.

Questa disciplina gioca un ruolo cruciale nell'informatica scientifica e ingegneristica, poiché permette di eseguire simulazioni e modelli matematici su computer per risolvere problemi reali, come la dinamica dei fluidi, l'analisi strutturale, la modellazione finanziaria e molti altri. Con lo sviluppo delle tecnologie computazionali, l'importanza

dell'analisi numerica è cresciuta notevolmente, poiché fornisce gli strumenti necessari per sfruttare appieno le potenzialità dei calcolatori moderni.

Un aspetto fondamentale è l'errore numerico, che può derivare da diversi fattori come la discretizzazione dei problemi continui, l'arrotondamento delle operazioni aritmetiche e la limitata capacità di rappresentazione dei numeri nei calcolatori. L'analisi numerica, quindi, non solo sviluppa algoritmi efficienti, ma si preoccupa anche di stimare e ridurre tali errori, garantendo che i risultati ottenuti siano sufficientemente accurati per l'applicazione pratica.

1.1.2 Storia dell'analisi numerica

L'analisi numerica ha origini antiche [2], sviluppandosi ben prima dell'avvento dei calcolatori elettronici. Già nell'antichità, matematici come Archimede, Eulero e Newton avevano bisogno di trovare soluzioni approssimate a problemi complessi che non potevano essere risolti con strumenti analitici esatti. Archimede, per esempio, utilizzava metodi di approssimazione per stimare il valore di π, sfruttando un processo di calcolo iterativo basato su poligoni inscritti e circoscritti a un cerchio. Newton, invece, fu uno dei pionieri nell'introduzione di metodi numerici per la risoluzione di equazioni, con il metodo di Newton-Raphson, ancora oggi largamente utilizzato per trovare approssimazioni delle radici di funzioni non lineari.

Nel corso dei secoli, l'analisi numerica si è evoluta per includere una vasta gamma di tecniche che permettevano di risolvere problemi matematici complessi, come la risoluzione di sistemi di equazioni lineari e non lineari, il calcolo di derivate e integrali, l'approssimazione di funzioni e la ricerca di autovalori e autovettori. In epoca pre-calcolatore, questi metodi richiedevano calcoli manuali lunghi e laboriosi, il che limitava notevolmente la scala dei problemi che potevano essere affrontati. Spesso, matematici e ingegneri dovevano accontentarsi di soluzioni semplificate, data la difficoltà di eseguire calcoli complessi su larga scala.

Con l'invenzione dei calcolatori elettronici nella metà del XX secolo, l'analisi numerica ha subito una trasformazione radicale. I calcolatori hanno reso possibile l'automatizzazione dei calcoli, consentendo di risolvere problemi di dimensioni e complessità molto superiori rispetto al passato. Problemi che in precedenza richiedevano giorni o settimane di lavoro manuale potevano ora essere risolti in pochi secondi o minuti. Questo ha aperto nuove frontiere in numerosi campi della scienza e dell'ingegneria, permettendo la simulazione di sistemi fisici complessi, l'analisi di grandi quantità di dati e la risoluzione di problemi di ottimizzazione su larga scala.

Con l'aumento della potenza di calcolo, l'analisi numerica è diventata una disciplina essenziale per affrontare problemi in fisica, ingegneria, economia, biologia, e molte altre scienze applicate. Ad esempio, la risoluzione di equazioni differenziali parziali,

fondamentali per modellare fenomeni come la propagazione delle onde o il flusso di fluidi, è diventata accessibile grazie all'utilizzo di metodi numerici. Inoltre, il campo dell'informatica scientifica si basa fortemente sull'analisi numerica per sviluppare simulazioni di fenomeni naturali, dalla dinamica dei fluidi in aerodinamica alla modellazione del clima globale.

Un altro aspetto importante di questa branca della matematica è la gestione degli errori. Poiché ogni calcolo numerico introduce un certo margine di errore, dovuto principalmente alla rappresentazione finita dei numeri nei calcolatori e ai processi di arrotondamento, si ripone notevole importanza nello sviluppare algoritmi che minimizzano e controllano tali errori. Metodi come il condizionamento dei problemi e la stabilità degli algoritmi sono fondamentali per garantire che i risultati ottenuti siano accurati e affidabili, anche in presenza di piccole perturbazioni nei dati di ingresso.

In sintesi, mentre l'analisi numerica esisteva già da secoli come strumento per ottenere soluzioni approssimate, l'avvento dei calcolatori elettronici ha trasformato radicalmente il campo, rendendolo fondamentale per il progresso della scienza e della tecnologia moderne. Le tecniche numeriche permettono di affrontare problemi che sarebbero irrisolvibili con metodi analitici, offrendo soluzioni pratiche ed efficienti a problemi complessi in un'ampia varietà di discipline.

1.2 Problema d'interesse

Uno dei problemi di maggiore interesse nell'analisi numerica riguarda la risoluzione di sistemi lineari.

Definizione 1.2.1. Un sistema di equazioni lineari è un insieme di m equazioni lineari in n incognite, che può essere scritto nel modo seguente:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n & = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n & = b_2 \\ \vdots & \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n & = b_m \end{cases} \tag{1.2.1}$$

Scritto in forma matriciale

$$Ax = b \tag{1.2.2}$$

Dove $A \in \mathbb{R}^{m \times n}$ è la matrice dei coefficienti, $b \in \mathbb{R}^m$ è il vettore dei termini noti e $x \in \mathbb{R}^n$ è il vettore delle incognite da trovare per risolvere il sistema.

Questo è un classico sistema lineare che può essere risolto con metodi esatti calcolando l'inversa della matrice A (se esiste) ottenendo in maniera analitica

$$x = A^{-1}b \tag{1.2.3}$$

Metodo	**Prodotti**	**Somme**	**Rad. Quad.**
Gauss	$O(\frac{n^3}{3})$	$O(\frac{n^3}{3})$	
Cholesky	$O(\frac{n^3}{6})$	$O(\frac{n^3}{6})$	O(n)
Gauss-Jordan	$O(\frac{n^3}{2})$	$O(\frac{n^3}{2})$	
Givens	$O(\frac{4 \cdot n^3}{3})$	$O(\frac{2 \cdot n^3}{3})$	$O(\frac{2 \cdot n^3}{2})$

Tabella 1.1: Complessità compuazionale

Quindi l'obiettivo è quello di trovare un vettore $x : x = A^{-1}b$. Questo però non è sempre realizzabile in pratica, sia perché la matrice A potrebbe essere non singolare (ovvero non invertibile) sia perché invertire la matrice A potrebbe richiede troppo tempo a livello computazionale. Infatti, nel mondo odierno i calcolatori elettronici hanno introdotto nuovi aspetti da tenere in considerazione, dall'approssimazione introdotta dalla rappresentazione finita dei dati, all'importanza della complessità computazionale dell'algoritmo utilizzato.

Analizzando la tabella (1.1) si può capire come già in linea teorica, l'implementazione di un algoritmo che ricalchi i metodi di risoluzione analitica sia possibile, ma computazionalmente poco efficiente. In questo scenario quindi ci poniamo il problema di risolvere sistemi lineari mantenendo particolare attenzione all'errore di approssimazione e all'implementazione algoritmica. In particolare, tanti campi applicativi si riducono alla discretizzazione di equazioni alle derivate parziali risultanti in sistemi lineari sparsi. In letteratura è ormai affermata la risoluzione di sistemi lineari sparsi attraverso l'utilizzo di metodi iterativi basati sui sottospazi di Krylov [3]. Il nuovo campo di ricerca invece riguarda quale formato di dati utilizzare per velocizzare la computazione, al pari dello stesso algoritmo utilizzato.

1.3 Principali campi applicativi

Parlando di analisi numerica è sempre importante enfatizzare la necessità di un fine pratico per cui sviluppare la teoria, infatti riprendendo Giuseppe Peano nella prefazione al volume *Tavole Numeriche* :

> «La Matematica insegna a calcolare il valore numerico delle incognite che si presentano nei problemi pratici. Questo è lo scopo ultimo di tutte le teorie, analitiche, geometriche e meccaniche. Queste, nelle nostre scuole, devono essere coronate dal calcolo numerico, onde porne in luce il significato e l'applicazione.»

Risulta importante avere una breve panoramica sui principali ambiti applicativi in cui utilizzare la risoluzione di sistemi lineari sparsi, che per quanto sia evidentemente

un aspetto di base nell'algebra lineare, non sempre viene associato agli innumerevoli risvolti applicativi, non comprendendo appieno l'importanza della ricerca su questo punto di vista in diversi ambiti, non solo dal punto di vista matematico.

Ingegneria e Fisica

- **Simulazioni strutturali**: Nella progettazione di edifici, ponti e infrastrutture, la risoluzione di sistemi lineari è cruciale nell'analisi delle forze e degli spostamenti delle strutture, attraverso l'analisi agli elementi finiti (FEA).
- **Dinamica dei fluidi**: Le equazioni di Navier-Stokes, discretizzate in sistemi lineari, permettono di simulare il comportamento dei fluidi in applicazioni come l'aerodinamica o l'oceanografia.
- **Elettromagnetismo**: Le equazioni di Maxwell, utilizzate nella progettazione di circuiti o antenne, generano sistemi lineari per modellare il comportamento del campo elettromagnetico.
- **Simulazioni fisiche**: I movimenti realistici in simulazioni di fluidi, gas o tessuti coinvolgono la risoluzione di sistemi lineari.

Informatica e Machine Learning

- **Regressione lineare**: La risoluzione di sistemi lineari è fondamentale per determinare i parametri nella regressione lineare, uno degli algoritmi più usati in machine learning.
- **Support Vector Machines (SVM)**: La risoluzione di problemi di ottimizzazione in SVM spesso si riduce a sistemi lineari.
- **Reti neurali**: Alcuni passaggi della propagazione dei gradienti nelle reti neurali comportano la risoluzione di sistemi lineari, specialmente in architetture avanzate.

Economia e Finanza

- **Modelli econometrici**: I modelli input-output di Leontief utilizzano sistemi lineari per analizzare le interconnessioni tra diverse industrie all'interno di un'economia.
- **Ottimizzazione dei portafogli**: La risoluzione di sistemi lineari è impiegata per minimizzare il rischio in un portafoglio di investimenti.

Grafica Computerizzata

- **Rendering e animazione**: Algoritmi di illuminazione globale, come il ray tracing, risolvono sistemi lineari per calcolare la luce e le ombre su superfici complesse.

Statistica e Analisi dei Dati

- **Analisi delle componenti principali (PCA)**: La PCA richiede la risoluzione di problemi di autovalori legati a sistemi lineari.
- **Curve fitting e interpolazione**: La risoluzione di sistemi lineari è utilizzata per approssimare dati sperimentali con modelli matematici.

Sistemi di Controllo

- **Robotica e automazione**: I sistemi di controllo per la robotica e l'automazione industriale utilizzano la risoluzione di sistemi lineari per ottimizzare traiettorie e comandi.
- **Ottimizzazione dei processi industriali**: La gestione ottimale di materiali e produzione si riduce spesso alla risoluzione di sistemi lineari.

Telecomunicazioni e Reti

- **Analisi di rete**: La distribuzione ottimale delle risorse in reti di telecomunicazione implica la risoluzione di sistemi lineari per gestire traffico e larghezza di banda.
- **Codifica e compressione**: Sistemi lineari sono utilizzati negli algoritmi di compressione dati e codifica dei segnali, come i codici convoluzionali.

Chimica Computazionale e Biologia

- **Modellazione molecolare**: La simulazione di molecole richiede la risoluzione di sistemi lineari per modellare la distribuzione elettronica e prevedere le proprietà chimiche.
- **Genetica e biostatistica**: Nella bioinformatica, i modelli di interazione genica e analisi statistica richiedono la risoluzione di sistemi lineari.

Criptoanalisi

- **Attacchi a cifrari**: Nella criptoanalisi, alcuni attacchi su cifrari a blocchi o flusso utilizzano sistemi lineari per individuare correlazioni tra chiavi segrete e testi cifrati.
- **Codici lineari**: Sistemi lineari sono utilizzati per decodificare messaggi cifrati in crittografia basata su codici, come i codici di Goppa.
- **Curve ellittiche**: Alcuni attacchi crittografici su curve ellittiche si riducono a risolvere sistemi lineari, utilizzati per analizzare la sicurezza dei protocolli basati su curve.

Con questa visione d'insieme in mente ci è possibile comprendere l'importanza di questo campo di ricerca e la sua natura interdisciplinare, dove ogni ambito della ricerca può portare un suo contributo importante affinché l'analisi numerica rispecchi le motivazioni per cui è nata, ovvero la risoluzione di problemi applicativi reali.

1.4 Formati di rappresentazione dati

In particolare, dal punto di vista della Computer Science, è fondamentale trovare metodologie implementative sempre nuove al fine di ottimizzare la computazione in questione. Parlando di metodi iterativi per la risoluzione di sistemi lineari è sempre più in uso l'utilizzo di formati di rappresentazione dati sempre più compatti. Lo scopo è quello di velocizzare la computazione dato che a livello hardware su FPU (Floating Point Unit) effettuare un calcolo a 32 bit risulta in uno speedup ≈ 2 rispetto ad un calcolo a 64 bit. Inoltre una rappresentazione a 32 bit ha un'occupazione minore della memoria, favorendo anche la velocità di trasferimento dati a parità di larghezza di banda. Alcuni esempi di studi sull'utilizzo di diverse precisioni nella rappresentazione dei dati sono i seguenti:

- Simulazioni: Nel mondo delle simulazioni risolta sempre presente una discretizzazione di equazioni differenziali, come viene mostrato nel simulatore del reattore nuclare ITER dove Idomura, Ina, Ali e Imamura [4] hanno utilizzato un float a 16 bit per ottenere uno speedup nell'utilizzo di metodi di Krylov.

- Previsioni meteo: Questo è un problema che richiede una grande mole di computazioni in poco tempo, rendendo importantissimo l'utilizzo di metodi con tempi di esecuzioni sempre minori, come dimostrato da N. Palmer [5] utilizzando float a 32 bit per ottenere speedup nella computazione.

- Machine Learning: Anche in questo ambito che è ormai diventato uno dei più fiorenti nel panorama della ricerca in Computer Science è stato da subito pensato come l'utilizzo di virgola mobile a 32 bit potesse non solo essere usata durante l'esecuzione del modello addestrato, ma anche nell'addestramento stesso del modello [6].

Risulta quindi interessante andare a studiare come l'utilizzo di una rappresentazione dei dati con un minor numero di bit possa essere utilizzata al fine di ottenere una computazione che tiene conto sia di aspetti di performance sia di aspetti di accuratezza del calcolo.

Definizione 1.4.1. (Accuratezza) Nella teoria degli errori, l'accuratezza è il grado di corrispondenza del dato teorico, desumibile da una serie di valori misurati (campione di dati), con il dato reale o di riferimento, ovvero la differenza tra valore medio campionario e valore vero o di riferimento. Indica la vicinanza del valore trovato a quello reale. È un concetto qualitativo che dipende sia dagli errori casuali che da quelli sistematici.

Avendo chiaro il punto di partenza, l'obiettivo di questa tesi sarà quello di sperimentare l'utilizzo di formati di dati a rappresentazioni compatte tentando di trovare un'accuratezza migliore utilizzando un approccio con precisioni miste.

Capitolo 2

Metodi di risoluzione di sistemi lineari

La risoluzione di sistemi lineari è uno dei problemi fondamentali dell'algebra lineare, con applicazioni che spaziano dall'ingegneria all'economia, dall'analisi dei dati alla fisica computazionale. Un sistema lineare si presenta nella forma:

$$Ax = b$$

dove $A \in \mathbb{R}^{m \times n}$ è una matrice quadrata di coefficienti, $x \in \mathbb{R}^n$ è il vettore delle incognite, e $b \in \mathbb{R}^m$ è il vettore dei termini noti. L'obiettivo è trovare il vettore delle soluzioni x, che soddisfa l'equazione data. Esistono diversi metodi per risolvere sistemi lineari, che possono essere classificati in due categorie principali:

- **Metodi diretti**: Trovano la soluzione esatta del sistema (entro i limiti dell'errore numerico) in un numero finito di operazioni. Questi metodi includono l'*eliminazione di Gauss*, la *fattorizzazione LU* e la *decomposizione ai valori singolari (SVD)*.
- **Metodi iterativi**: Forniscono una successione di approssimazioni alla soluzione esatta, migliorando progressivamente la stima iniziale. Questi metodi sono particolarmente utili per sistemi di grandi dimensioni o sparsi, dove i metodi diretti diventano computazionalmente costosi. Tra i più noti ci sono il *metodo di Jacobi*, il *metodo di Gauss-Seidel* e il *metodo del gradiente coniugato (CG)*.

2.1 Metodi Diretti

I metodi diretti per la risoluzione di un sistema lineare $Ax = b$ trovano la soluzione esatta (entro errori di arrotondamento numerico) in un numero finito di operazioni. Questi metodi sono efficaci per sistemi di dimensioni moderate e ben condizionati, in quanto garantiscono la soluzione esatta. Il condizionamento in matematica, in particolare nel calcolo numerico, riguarda il rapporto tra errore commesso sul risultato di un calcolo e incertezza sui dati in ingresso.

Definizione 2.1.1. (Condizionamento) Un sistema è ben condizionato quando la soluzione del sistema con delle piccole variazioni, non differisce molto dalla soluzione del sistema originale; al contrario, un sistema mal condizionato è un sistema dove le soluzioni sono molto sensibili a piccole perturbazioni dei dati iniziali.

Quindi risulta fondamentale quantificare la capacità di invarianza rispetto a piccoli cambiamenti nella matrice dei coefficienti per poter applicare un metodo diretto. In particolare, capire se la matrice dei coefficienti è mal condizionata è possibile attraverso il numero di condizionamento.

Definizione 2.1.2. Numero di condizionamento Data una matrice $A \in \mathbb{R}^{n\times n}$ non singolare, si dice numero di condizionamento della matrice

$$K(A) = ||A||^{-1}||A|| \tag{2.1.1}$$

dove per $||A||$ si intende la norma matriciale indotta.

Una volta capito per quali classi di sistemi lineari si possono usare i metodi diretti, possiamo descriverne indicativamente il funzionamento: si cerca di utilizzare trasformazioni matriciali per ridurre il sistema ad una forma più semplice, come una matrice triangolare, e successivamente risolvono il sistema tramite sostituzione in avanti o all'indietro. Essi forniscono la soluzione in un numero predeterminato di passi e sono ideali per sistemi di dimensioni moderate con matrici ben condizionate. Tuttavia, per sistemi di grandi dimensioni o mal condizionati, i metodi diretti possono soffrire di instabilità numerica o richiedere una complessità computazionale elevata. Le caratteristiche principali di questi metodi sono:

- **Soluzione esatta**: Forniscono la soluzione in un numero finito di operazioni, salvo errori numerici dovuti all'arrotondamento.
- **Efficienza per dimensioni moderate**: Sono efficaci per sistemi di dimensioni moderate, ma possono risultare computazionalmente costosi per sistemi di grandi dimensioni.
- **Fattorizzazione**: Spesso si basano sulla decomposizione della matrice A in prodotti di matrici triangolari, che facilitano la risoluzione del sistema.

Di seguito si possono osservare i principali metodi diretti conosciuti in letteratura.

2.1.1 Eliminazione di Gauss

L'Eliminazione di Gauss è uno dei metodi più semplici per risolvere un sistema lineare. Consiste nell'applicare trasformazioni elementari di riga per ridurre la matrice A in forma triangolare superiore:

$$A \to U$$

Una volta che la matrice è triangolare superiore, si risolve il sistema tramite **sostituzione all'indietro**. Il metodo è generalmente stabile, ma può essere necessario applicare il **pivoting parziale** per evitare instabilità numeriche.

2.1.2 Fattorizzazione Cholesky

La fattorizzazione di Cholesky è una variante della fattorizzazione LU per matrici simmetriche e definite positive. La matrice A è fattorizzata come:

$$A = LL^T$$

dove L è una matrice triangolare inferiore. Come nella fattorizzazione LU, la soluzione del sistema richiede due passaggi: la sostituzione in avanti per risolvere $Ly = b$ e la sostituzione all'indietro per risolvere $L^T x = y$. Questo metodo è più efficiente della fattorizzazione LU, ma può essere applicato solo a matrici con proprietà particolari.

2.1.3 Decomposizione ai Valori Singolari (SVD)

La **Decomposizione ai Valori Singolari (SVD)** è un metodo generale per risolvere sistemi lineari, specialmente quando la matrice A è mal condizionata o rettangolare. La decomposizione di una matrice A è:

$$A = U\Sigma V^T$$

dove U e V sono matrici ortogonali e Σ è una matrice diagonale contenente i valori singolari di A. La SVD è particolarmente utile per risolvere sistemi sovradeterminati o mal posti, e può essere utilizzata per il filtraggio di rumore nei dati.

2.1.4 Metodo di Eliminazione di Gauss con Pivoting

Il metodo di Gauss con **pivoting parziale** è una variante dell'eliminazione di Gauss che migliora la stabilità numerica. Durante il processo di eliminazione, si scambia la riga con il massimo elemento assoluto nella colonna corrente per minimizzare gli errori di arrotondamento:

$$A \to PA$$

dove P è una matrice di permutazione che rappresenta lo scambio di righe. Il metodo con pivoting è essenziale per migliorare la stabilità quando la matrice A è mal condizionata o contiene valori molto piccoli lungo la diagonale.

2.1.5 Fattorizzazione LU

La fattorizzazione LU decomposizione la matrice A in due matrici triangolari: una triangolare inferiore L e una triangolare superiore U:

$$A = LU$$

andando cosi a non dover calcolare esplicitamente A^{-1}, ma risolvendo due sistemi per sostituzione invece.

Dopo la fattorizzazione, risolvere il sistema $Ax = b$ richiede due semplici passaggi:

1. Risolvere $Ly = b$ tramite **sostituzione in avanti**.
2. Risolvere $Ux = y$ tramite **sostituzione all'indietro**.

La fattorizzazione LU è particolarmente efficiente per risolvere più sistemi lineari con la stessa matrice A, ma diversi vettori b, dato che al variare di b la fattorizzazione LU della matrice A rimane sempre la stessa.

Definizione 2.1.3 (Fattorizzazione LU)**.** Data una matrice $A \in \mathbb{R}^{n\times n}$ invertibile, si dice fattorizzazione LU della matrice A la decomposizione $A = LU : L \in \mathbb{R}^{n\times n}$ è una matrice triangolare inferiore e $U \in \mathbb{R}^{n\times n}$ è una matrice triangolare superiore

Esempio 2.1.1. Data una matrice 3x3, la sua decomposizione LU è la seguente

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \times \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix} = LU$$

Grazie a questa trasformazione possiamo risolvere il sistema lineare di partenza senza la necessità di calcolare l'inversa della matrice A, sfruttando la risoluzione per sostituzione all'indietro dei sistemi triangolari che abbiamo formato. Infatti:

$$Ax = b \rightarrow Ax = LUx = b \tag{2.1.2}$$

$$\begin{cases} Ly = b \\ Ux = y \end{cases} \tag{2.1.3}$$

Non sempre è possibile ottenere una decomposizione per una qualsiasi matrice A, quindi spesso si utilizza il pivoting, ovvero si considerano permutazioni di righe e colonne di A per ottenere una matrice che possiede una fattorizzazione LU.

Teorema 2.1.1. *Data una matrice $A \in \mathbb{R}^{n\times n}$ non singolare, allora esistono due matrici di permutazione P e Q, una matrice triangolare inferiore L con elementi diagonali uguali a uno e una matrice triangolare superiore $U : PAQ = LU$. Solo una fra P e Q è necessaria.*

Il teorema (2.1.1) ci assicura quindi l'esistenza della decomposizione, trasformando l'equazione (2.1.2) in

$$\begin{cases} Ly = Pb \\ Ux = y \end{cases} \tag{2.1.4}$$

dove P è la nostra matrice di permutazione.

Per trovare questa decomposizione si utilizza il metodo di Gauss, andando ad effettuare diverse moltiplicazioni tra matrici al fine di applicare operazioni elementari di trasformazione ad A avendo

$$U = T_1^{-1} \cdot T_2^{-1} \dots T_n^{-1} \cdot A = T \cdot A \tag{2.1.5}$$

potendo cosi calcolare facilmente l'inversa della matrice T

$$T^{-1} = T_1^{-1} \cdot T_2^{-1} \dots T_n^{-1} \tag{2.1.6}$$

$$L = T^{-1} \tag{2.1.7}$$

e ottenendo quindi

$$LU = T^{-1}TA \tag{2.1.8}$$

senza dover esplicitamente calcolare alcuna matrice inversa.

2.2 Metodi Iterativi

I **metodi iterativi** per la risoluzione di un sistema lineare $Ax = b$ sono approcci che convergono a una soluzione attraverso successive approssimazioni. A differenza dei metodi diretti, i metodi iterativi migliorano una soluzione approssimativa iniziale e convergono verso la soluzione esatta. Sono particolarmente utili per sistemi di grandi dimensioni o sparsi perché mantengono la proprietà di sparsità della matrice. I metodi iterativi partono da una stima iniziale della soluzione e migliorano progressivamente questa stima fino a quando la soluzione approssimata raggiunge una precisione desiderata. Sebbene i metodi iterativi non garantiscano sempre la convergenza, sono spesso preferiti per la loro efficienza computazionale in contesti di grandi dimensioni. Volendo analizzare uno schema di massima di questa tipologia di metodi possiamo considerare lo pseudo-codice dell'algoritmo (1).

Algoritmo 1 Metodo Iterativo

Input: $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, $x_0 \in \mathbb{R}^n$, ϵ, $k_{\max}$
Output: x
$k \leftarrow 0$
$r_0 \leftarrow b - Ax_0$ ▷ Initial residual
while $\|r_k\| > \epsilon$ and $k < k_{\max}$ **do**
    Compute a correction Δx_k
    Update the solution: $x_{k+1} \leftarrow x_k + \Delta x_k$
    Update the residual: $r_{k+1} \leftarrow b - Ax_{k+1}$
    $k \leftarrow k + 1$
end while
return x_k

Per costruire un metodo iterativo, possiamo considerare:

$$b - Ax = 0$$

$$Mx + (b - Ax) = Mx$$

dove M è una matrice scelta in modo che sia non singolare. A questo punto, il sistema lineare può essere riscritto come:

$$M^{-1}(Mx + (b - Ax)) = M^{-1}Mx$$

o, in forma equivalente:

$$x = M^{-1}(Mx + (b - Ax))$$
$$x = M^{-1}Mx + M^{-1}b - M^{-1}Ax$$
$$x = x(I - M^{-1}A) + M^{-1}b$$

e quindi definendo

$$B = I - M^{-1}Ac = M^{-1}b$$

si ottiene

$$x^{k+1} = Bx^k + c \tag{2.2.1}$$

dove viene definita matrice di iterazione Questa equazione suggerisce un processo iterativo per risolvere il sistema, dove si parte da una stima iniziale $x^{(0)}$ e si aggiorna la soluzione utilizzando il passo iterativo. Questo fornisce un'approssimazione successiva della soluzione esatta.

Dopo aver costruito un metodo iterativo è opportuno domandarsi se la scelta di M è stata opportuna, cioè se dopo infinite iterazioni la soluzione ottenuta è realmente quella del sistema (la proprietà di convergenza di cui sopra).

Condizione necessaria e sufficiente affinché il metodo converga alla soluzione del sistema per ogni scelta di x^0 è che il raggio spettrale (cioè il più grande autovalore in modulo) di B sia strettamente inferiore a 1, in formule:

$$\lim_{k \to \text{inf}} x^k = x, \forall x^0 \iff \rho(B) < 1$$

Il principale vantaggio dei metodi diretti è la loro precisione e determinismo: essi forniscono la soluzione esatta in un numero finito di operazioni. Tuttavia, per sistemi molto grandi, come quelli che emergono da simulazioni fisiche o da modelli di rete, i metodi iterativi offrono un'alternativa più efficiente, soprattutto quando le matrici coinvolte sono sparse o mal condizionate. Di seguito sono riportati i principali metodi iterativi utilizzati nel calcolo scientifico.

2.2.1 Metodo di Jacobi

Il metodo di Jacobi separa la matrice A nella parte diagonale D e nella parte rimanente R. L'iterazione aggiorna la soluzione come segue:

$$x^{(k+1)} = D^{-1}(b - Rx^{(k)})$$

Ogni componente della nuova approssimazione $x^{(k+1)}$ viene calcolata usando i valori della precedente iterazione $x^{(k)}$. La convergenza è garantita se A è diagonalmente dominante o definita positiva.

2.2.2 Metodo di Gauss-Seidel

Il metodo di Gauss-Seidel migliora il metodo di Jacobi usando immediatamente i valori aggiornati della soluzione:

$$x_i^{(k+1)} = \frac{1}{A_{ii}} \left(b_i - \sum_{j<i} A_{ij} x_j^{(k+1)} - \sum_{j>i} A_{ij} x_j^{(k)} \right)$$

Spesso converge più velocemente rispetto al metodo di Jacobi, soprattutto quando A è ben condizionata.

2.2.3 Convergenza dei Metodi Iterativi

La convergenza dipende principalmente da:

- **Condizionamento della matrice** A: Matrici ben condizionate favoriscono la convergenza.
- **Proprietà della matrice**: La convergenza è garantita se A è diagonalmente dominante o definita positiva.

I metodi iterativi sono fondamentali per risolvere sistemi lineari di grandi dimensioni, con una vasta gamma di applicazioni in ingegneria, fisica computazionale e analisi dei dati. In particolare, il metodo del Gradiente Coniugato (CG) e la Decomposizione ai Valori Singolari (SVD) sono strumenti essenziali in questi contesti.

2.2.4 Metodo del Gradiente Coniugato (CG)

Il metodo del gradiente coniugato è efficace per matrici simmetriche e definite positive. Minimizza una funzione quadratica associata al sistema lineare:

$$r^{(k)} = b - Ax^{(k)}$$

Il metodo genera una sequenza di approssimazioni $x^{(k)}$ migliorando la soluzione in modo che il residuo $r^{(k)}$ diminuisca progressivamente. È noto per la sua efficienza, spesso convergendo in meno iterazioni rispetto alla dimensione n del sistema. Nel prossimo capitolo andremo ad analizzare nel dettaglio questo metodo per capirlo a fondo, in modo da avere una visione d'insieme utile durante l'implementazione dell'algoritmo.

Capitolo 3

Gradiente Coniugato

Per analizzare l'algoritmo del gradiente coniugato (GC) dobbiamo prima capire la sua origine, ovvero passare per l'algoritmo della discesa del gradiente [7]. Infatti, il nostro scopo è quello di capire affondo le singole operazioni svolte all'interno dell'algoritmo, in modo da poter garantire la migliore implementazione possibile. Il metodo del GC è applicabile a sistemi lineari la cui matrice dei coefficienti A è simmetrica e definita semi-positiva, questo è dovuto dalla funzione che stiamo cercando di ottimizzare.

3.1 Discesa del Gradiente

Prima di entrare nel dettaglio dell'algoritmo della discesa del gradiente, è necessario capire l'importanza delle proprietà richieste dalla matrice per far convergere questo algoritmo andando a richiamre la basi teoriche che vi sono dietro.

3.1.1 Basi teoriche

L'obiettivo che si pone questo algoritmo è quello di minimizzare una forma quadratica del tipo

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c \tag{3.1.1}$$

L'equazione che minimizza questa forma quadratica è proprio $Ax = b$ ed è importante avere un intuizione grafica di questo fenomeno osservando le figure 3.1 e 3.2
Poiché A è definita positiva, la superficie definita da $f(x)$. ha la forma di un paraboloide. Questo ci garantisce alcune proprietà riguardanti il gradiente della funzione stessa.

$$f(x) = \begin{bmatrix} \frac{\delta f(x)}{\delta x_1} \\ \frac{\delta f(x)}{\delta x_1} \\ \vdots \\ \frac{\delta f(x)}{\delta x_n} \end{bmatrix} \tag{3.1.2}$$

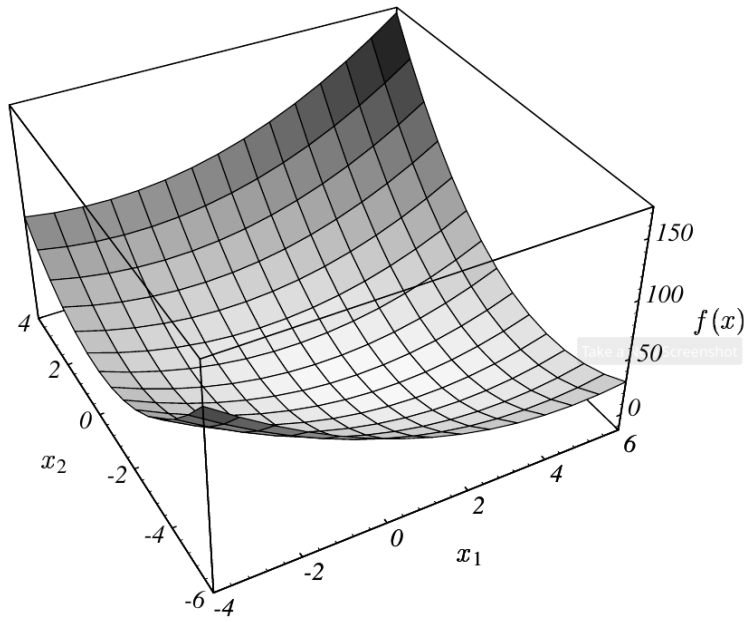


Figura 3.1: Iperpiano di della forma quadratica 3.1.1

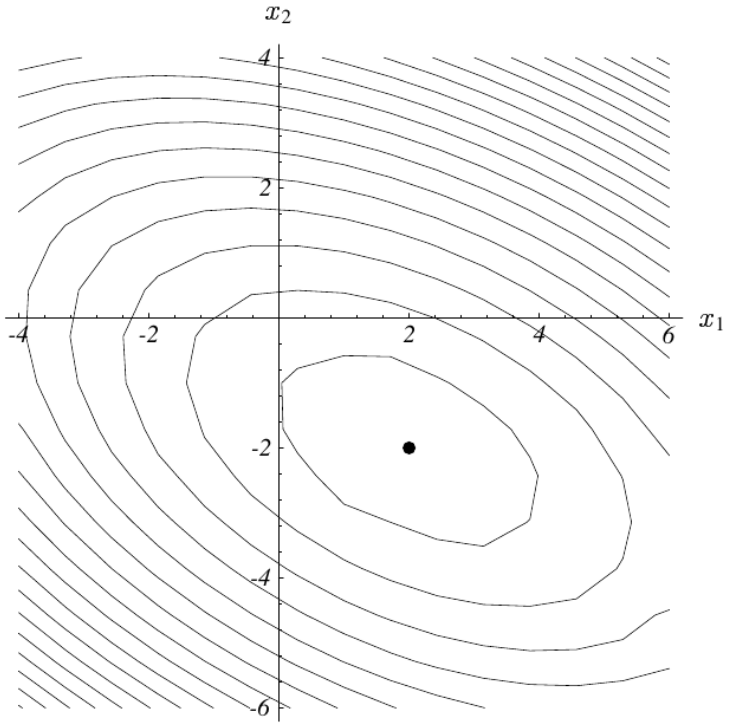


Figura 3.2: Contorno della forma quadratica. Ogni curca corrisponde ad un f(x) fisso

Il gradiente ci indica la direzione in cui il paraboloide cresce, quindi il punto in cui il gradiente della funzione è pari a zero $f^{'}(x) = 0$ sarà il minimo del paraboloide. Con qualche passaggio dall'equazione 3.1.1 si ottiene:

$$f^{'}(x) = \frac{1}{2}A^{T}x + \frac{1}{2}Ax - b \tag{3.1.3}$$

se A + simmetrica si ottiene

$$f^{'}(x) = Ax - b \tag{3.1.4}$$

e quindi valutare il gradiente in 0 non significa altro che risolvere il sistema lineare Ax = b. Inoltre, dato che la matrice A è definita positiva, siamo certi che $f^{'}(x)$ è il minimo assoluto della funzione.

Definizione 3.1.1. In matematica, e più precisamente in algebra lineare, una matrice definita positiva è una matrice quadrata A tale che, detto $\mathbf{x}^*$ il trasposto complesso coniugato di $\mathbf{x}$, si verifica che la parte reale di $\mathbf{x}^*A\mathbf{x}^*$ è positiva per ogni vettore complesso $\mathbf{x} \neq \mathbf{0}$

Nel nostro caso stiamo trattando un vettore reale, quindi ne deriva che una matrice è definita positiva se e solo se:

$$x^{T}Ax > 0 \tag{3.1.5}$$

ma dato che preso un qualsiasi punto p abbaimo che

$$f(p) = f(x) + \frac{1}{2}(p - x)^{T}A(p - x) \tag{3.1.6}$$

dato che la disequazione 3.1.5 è vera, allora il secondo termine della 3.1.6 sarà positivo $\forall p \neq x$, garantendoci cosi che risolvere $Ax = b$ dia come soluzione il minimo globale del paraboloide.

3.1.2 Algoritmo della discesa del gradiente

Arrivati a questo punto abbiamo capito quindi che l'algoritmo utilizzerà in input un vettore iniziale $\mathbf{x}_{(0)}$ (solitamente inizializzato a $\mathbf{0}$) e andrà ad effettuare una raffinazione della soluzione nelle iterazioni successive seguendo la direzione opposta del gradiente. Graficamente, quello che stiamo cercando di fare è di trovare il passo che ci faccia scendere più velocemente nel paraboloide, seguendo quindi la direzione $-f^{'}(x) = b - Ax_{(i)}$. Ora consideriamo le seguenti definizioni

Definizione 3.1.2. (Errore) Definiamo con $e_{(k)}$ l'errore della soluzione al passo k, ovvero $e_{(k)} := |x - x_k|$

Definizione 3.1.3. (Residuo) Definiamo con $r_{(k)}$ il residuo del sistema al passo k, ovvero $r_{(k)} := b - Ax_{(k)}$

e notiamo che $\lim_{k\to\infty} e_{(k)} = 0$. Si può facilmente ricavare che

$$r_{(k)} = -Ae_{(k)} \tag{3.1.7}$$

rendendo cosi possibile quantificare l'errore della soluzione, dato che non conoscendo il vero valore di x risulterebbe impossibile calcolare $e_{(k)}$ altrimenti. Inoltre, è anche vero che

$$r_{(k)} = -f^1(x_{(k)}) \tag{3.1.8}$$

per definizione. Quindi possiamo capire la direzione in cui effettuare il nostro passo ad ogni iterazione in base al residuo.

$$x_{(k+1)} = x_{(k)} + \alpha r_{(k)} \tag{3.1.9}$$

L'ultimo passaggio da capire è come scegliere α affinché l'incremento ad ogni iterazione sia corretto. Per trovare il valore di α che minimizzi la nostra funzione f, quindi $\frac{\delta f(x_{(k)})}{\delta \alpha} = 0$

$$\frac{\delta f(x_{(k)})}{\delta \alpha} = f^{'}(x_{(k)})^T \frac{\delta x_{(k)}}{\delta \alpha} = f^{'}(x_{(k)})^T r_{(k)}$$

da cui ne ricaviamo che $f^{'}(x_{(k)})$ deve essere ortogonale a $r_{(k)}$. Sappiamo però che $f^{'}(x_{(k)})$ = - $r_{(k+1)}$, otteniamo quindi che per garantire l'ortogonalità:

$$r_{(k+1)}^T r_{(k)} = 0$$

$$(b - Ax_{(k+1)})^T r_{(k)} = 0$$
$$(b - A(x_{(k)} + \alpha r_{(k)})^T r_{(k)} = 0$$
$$(b - A(x_{(k)})^T r_{(k)} - \alpha(Ar_{(k)}))^T r_{(k)} = 0$$
$$((b - A(x_{(k)})^T r_{(k)} = \alpha(Ar_{(k)}))^T r_{(k)}$$
$$r_{(k)}^T r_{(k)} = \alpha(Ar_{(k)}))^T r_{(k)}$$

$$\alpha = \frac{r_{(k)}^T r_{(k)}}{r_{(k)}^T A r_{(k)}} \tag{3.1.10}$$

possiamo finalmente formulare il nostro algoritmo della discesa del gradiente, andando ad avere un idea visiva di come questo aggiorni la soluzione ad ogni passo riferendoci alla figura 3.3

Algoritmo 2 Steepest Descent

Input: $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$, $x_0 \in \mathbb{R}^n$, ϵ, $k_{\max}$
Output: x
$k \leftarrow 0$
$r_0 \leftarrow b - Ax_0$ ▷ Calcolo del residuo iniziale
while $\|r_k\| > \epsilon$ e $k < k_{\max}$ **do**
    $\alpha_k \leftarrow \frac{r_k^T r_k}{r_k^T A r_k}$ ▷ Calcolo del passo ottimale
    $x_{k+1} \leftarrow x_k + \alpha_k r_k$ ▷ Aggiornamento della soluzione
    $r_{k+1} \leftarrow r_k - \alpha_k A r_k$ ▷ Aggiornamento del residuo
    $k \leftarrow k + 1$
return x_k

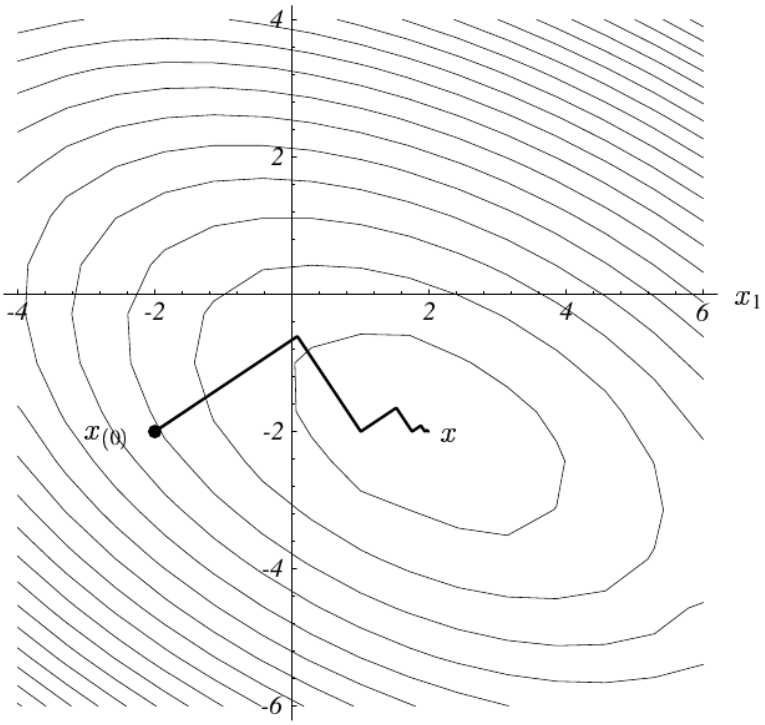


Figura 3.3: Discesa del Gradiente

3.2 Gradiente Coniugato

Il metodo della discesa del gradiente spesso si trova ad effettuare più incrementi nella stessa direzione, rendendo il numero di iterazioni maggiore rispetto al minimo teorico. Lo scopo del gradiente coniugato è proprio quello di minimizzare i passi di discesa del paraboloide, selezionando delle direzioni di ricerca $\{d_{(0)}, d_{(1)}, \ldots, d_{(n)}\}$ ortogonali tra loro. L'incremento diventerà quindi

$$x_{(k+1)} = x_k + \alpha_{(k)} d_{(k)} \tag{3.2.1}$$

dove $e_{(k+1)}$ è perpendicolare a $d_{(k)}$ al fine di minimizzare i passi da effettuare. Questa proprietà può quindi essere sfruttata per trovare il valore di $\alpha_{(k)}$ ottimale utilizzando un procedimento analogo al precedente, ovvero

$$\alpha_{(k)} = -\frac{d_{(k)}^T e_{(k)}}{d_{(k)}^T d_{(k)}} \tag{3.2.2}$$

Il problema ancora persiste, dato che non possiamo calcolare $\alpha_{(k)}$ senza conoscere $e_{(k)}$. La soluzione quindi è quella di scegliere $d_{(k)}$ in modo che la direzione di ricerca sia A-ortogonale.

Definizione 3.2.1. Due vettori $d_{(i)}$ e $d_{(j)}$ si dicono A-ortogonali o coniugati se

$$d_{(i)}^T A d_{(j)} = 0 \tag{3.2.3}$$

ottenendo cosi dei vettori ortogonali tra loro, una volta proiettati sul piano A. Abbiamo quindi una nuova condizione da rispettare per trovare il valore di $\alpha_{(k)}$

$$\frac{\delta f(x_{(k+1)})}{\delta \alpha} = 0$$

$$f'(x_{(k+1)} \frac{\delta x_{(k+1)})}{\delta \alpha} = 0$$

$$-r_{(k+1)}^T d_{(k)} = 0$$

$$d_{(k)}^T A e_{(k+1)} = 0$$

Riprendendo l'equazione 3.2.2 otteniamo quindi

$$\alpha_{(k)} = -\frac{d_{(k)}^T A e_{(k)}}{d_{(k)}^T A d_{(k)}} = \frac{d_{(k)}^T r_{(k)}}{d_{(k)}^T A d_{(k)}} \tag{3.2.4}$$

Notare come se la direzione di ricerca fosse il residuo, la formula diventerebbe identica alla 3.1.10 utilizzata nel metodo di discesa del gradiente. L'ultimo elemento da capire per poter implementare l'algoritmo è come trovare i vettori $\{d_{(0)}, d_{(1)}, \ldots, d_{(n)}\}$. Il

processo più comunemente usato è il processo coniugato di Gram-Schmidt. Consideriamo un insieme di n vettori linearmente indipendenti $\{u_0, u_1, \ldots, u_n\}$ (una possibile scelta può essere considerare gli assi delle coordinate, anche se sono possibili scelte diverse), allora per costruire $d_{(k)}$ consideriamo u_k e sottraiamo le componenti che non sono A-ortogonali ai vettori d precedenti

$$d_{(k)} = u_k + \sum_{i=1}^{k-1} \beta_{ki} d_{(k)}$$

e quindi il valore di β sarebbe:

$$\beta_{ij} = -\frac{u_i^T A d_{(j)}}{d_{(j)}^T A d_{(j)}} \tag{3.2.5}$$

La difficoltà nell'usare il processo coniugato di Gram-Schmidt riguarda il dover mantenere in memoria tutte le vecchi vettori d che rappresentano le vecchie direzioni di ricerca, richiedendo inoltre $O(n^3)$ operazioni, risultando praticamente in un eliminazione di Gauss. Proprio per questo, il GC ha avuto poco utilizzo fin quando non si è trovato un modo per ovviare a queste problematiche. Infatti, si può dimostrare, come la scelta dei vettori $r_{(k)}$ come u risulta una scelta eccellente dal punto di vista computazionale. La prima grande trasformazione è che l'incremento del residuo diventa

$$r_{(k+1)} = r_{(k)} - \alpha(k) A d_{(k)} \tag{3.2.6}$$

La scelta di r garantisce che l'insieme dei vettori costruito sia linearmente indipendente, dato che ogni $r_{(k+1)}$ è ortogonale alla direzione di ricerca precedente. Quindi il nostro insieme di vettori $D_n = span\{r_{(0)}, r_{(1)}, \ldots, r_{(n-1)}\}$ ci fa capire che ogni residuo è anche ortogonale al precedente. Inoltre, l'equazione 3.2.6 ci mostra come $r_{(k+1)}$ sia una combinazione lineare dei precedenti residui e di $Ad_{(k)}$. Sapendo che $d_{(k)} \in D_n$, ogni nuovo sottospazio D_{n+1} sarà formato dall'unione del sottospazio D_n e da AD_n, quindi:

$$D_n = span\{d_{(0)}, Ad_{(0)}, A^2 d_{(0)}, \ldots, A^{n-1} d_{(0)}\} = span\{r_{(0)}, Ar_{(0)}, A^2 r_{(0)}, \ldots, A^{n-1} r_{(0)}\} \tag{3.2.7}$$

Questo è un sottospazio di Krylov, costruito applicando una matrice ripetutamente su un vettore. La proprietà interessante di questo sottospazio è che $r_{(k+1)}$ è ortogonale a D_{k+1}, implicando quindi che $r_{(k+1)}$ è A-ortogonale a D_k.
Con questa nuova prospettiva il processo coniugato di Gram-Schmidt diventa applicabile, trasformando la 3.2.5 in:

$$\beta_k = \frac{r_{(k)}^T r_{(k)}}{d_{(k-1)}^T r_{(k-1)}} == \frac{r_{(k)}^T r_{(k)}}{r_{(k-1)}^T r_{(k-1)}} \tag{3.2.8}$$

dove per $\beta_{(i)}$ si intende $\beta_{i,i-1}$, facendo cosi diventare questo algoritmo $O(m)$ dove m è il numero di elementi non nulli della matrice. Ne risulta il seguente algoritmo:

Algoritmo 3 Conjugate Gradient

Input: $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, $x_0 \in \mathbb{R}^n$, ϵ, $k_{\max}$
Output: x
$r_0 \leftarrow b - Ax_0$ ▷ Calcolo del residuo iniziale
$d_0 \leftarrow r_0$ ▷ Inizializzazione della direzione di discesa
$k \leftarrow 0$
while $\|r_k\| > \epsilon$ e $k < k_{\max}$ **do**
    $\alpha_k \leftarrow \frac{r_k^T r_k}{d_k^T A d_k}$ ▷ Calcolo del passo ottimale
    $x_{k+1} \leftarrow x_k + \alpha_k d_k$ ▷ Aggiornamento della soluzione
    $r_{k+1} \leftarrow r_k - \alpha_k A d_k$ ▷ Aggiornamento del residuo
    if $\|r_{k+1}\| \leq \epsilon$ **then**
        break ▷ Interrompe se la soluzione è sufficientemente accurata
    $\beta_k \leftarrow \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$ ▷ Calcolo del fattore di coniugazione
    $d_{k+1} \leftarrow r_{k+1} + \beta_k d_k$ ▷ Aggiornamento della direzione di discesa
    $k \leftarrow k + 1$
return x_k

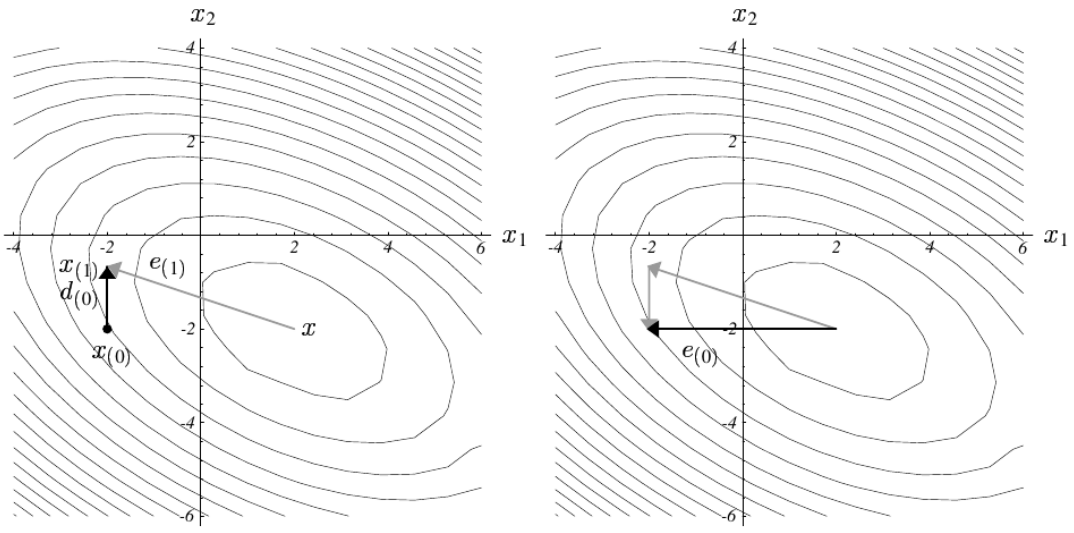


Figura 3.4: passi del Gradiente Coniugatosu un sistema bidimensionale

Capitolo 4

Aritmetica in virgola mobile

4.1 Rappresentazione in virgola mobile

La rappresentazione in virgola mobile è un sistema utilizzato per memorizzare numeri reali nei calcolatori moderni con una precisione variabile. Questo metodo permette di rappresentare numeri estremamente grandi o estremamente piccoli, cosa che non sarebbe possibile con la rappresentazione intera.
La virgola mobile si differenzia dalla virgola fissa, in cui la posizione della virgola decimale è fissa e definita a priori. Nella virgola mobile, invece, la posizione della virgola decimale è flessibile e viene determinata da un parametro chiamato esponente. In generale, un numero in virgola mobile può essere espresso nella forma:

$$(-1)^{\text{segno}} \times \text{mantissa} \times 2^{\text{esponente}-\text{bias}}$$

dove:

- Il **segno** determina se il numero è positivo o negativo.
- La **mantissa**, o significand, contiene i bit significativi del numero.
- L'**esponente** determina la scala o l'ordine di grandezza del numero.
- Il **bias** è un valore fisso utilizzato per rappresentare esponenti negativi.

L'introduzione di questo sistema di rappresentazione è stato necessario per non essere limitati dalla finitezza dei bit con cui un numero reale viene rappresentato all'interno di un calcolatore elettronico, con lo scopo di rappresentare numeri con ordini di grandezza molto diversi tra di loro. Questa codifica si ispira alla classica notazione scientifica, utilizzando però la base 2 in contrapposizione alla base 10. Questo permette di definire un intervallo dinamico determinato dal valore dell'esponente, cosi da poter rappresentare numeri maggiormente distanti tra loro, al contrario di una rappresentazione in virgola fissa.

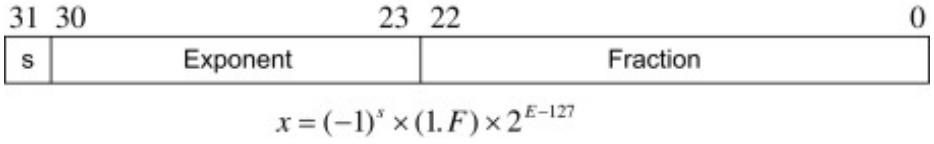


Figura 4.1: Virgola mobile a 32 bit

4.2 Lo Standard IEEE 754

Lo standard IEEE 754 [8] è stato introdotto per garantire una rappresentazione uniforme e precisa dei numeri in virgola mobile in tutti i computer moderni. Questo standard specifica formati per la rappresentazione dei numeri a virgola mobile sia in **singola precisione** (32 bit) che in **doppia precisione** (64 bit).
La struttura di un numero in virgola mobile secondo lo standard IEEE 754 è così suddivisa:

- **Segno**: 1 bit che indica se il numero è positivo (0) o negativo (1).
- **Esponente**: 8 bit (per singola precisione) o 11 bit (per doppia precisione). L'esponente è memorizzato come un numero *biased*, ovvero con un valore fisso (*bias*) sottratto per permettere la rappresentazione di esponenti negativi.
- **Mantissa**: 23 bit (per singola precisione) o 52 bit (per doppia precisione). La mantissa è normalizzata, il che significa che il primo bit è implicito e non memorizzato.

L'introduzione di questo standard nel 1985 è stata fondamentale per permettere la produzione di software portabile e replicabile, non dovendo creare interfacce per poter operare le FPU (Floating Point Unit) a livello hardware. Questo ha permesso un'ampia diffusione delle FPU, rendendo unificato il formato indipendentemente dal modello del processore o dal linguaggio di programmazione. Esistono anche versioni modificate simulate a livello software come ad esempio il bfloat16, ideato da Google cercando di replicare lo standard IEEE 754 con soli 16 bit, con l'obiettivo di accelerare le TPU (Tensor Processor Unit) per l'addestramento di reti neurali [9]. Nei classici formati IEEE 754 possiamo analizzare come la finestra dei numeri rappresentabili va da 10^{-38} a 10^{38} per il formato a singola precisione. I numeri sono rappresentati utilizzando 32 bit totali.

- 1 bit per il segno
- 8 bit per l'esponente (con bias di 127)
- 23 bit per la mantissa (con il bit implicito)

Nel formato a doppia precisione, invece i numeri sono rappresentati utilizzando 64 bit totali, avendo una finestra di rappresentazione che va da 10^{-308} a 10^{308}.La loro struttura è la seguente:

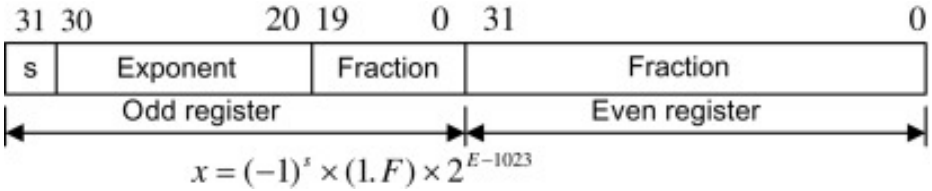


Figura 4.2: Virgola mobile a 64 bit

- 1 bit per il segno
- 11 bit per l'esponente (con bias di 1023)
- 52 bit per la mantissa (con il bit implicito)

Notare come la finestra di rappresentazione non è un intervallo, bensì un insieme discreto e proprio per questo sono inevitabili errori di approssimazione quando si rappresenta un numero reale in virgola mobile. Per dare un'idea concreta, di seguito un esempio di rappresentazione in binario del numero 5.75 nel formato IEEE 754 a singola precisione.

0 10000001 01100000000000000000000

4.2.1 Gestione degli Errori e delle Eccezioni

Lo standard IEEE 754 include anche specifiche per la gestione di situazioni speciali, come i numeri troppo piccoli o troppo grandi per essere rappresentati. Tra questi, i numeri **subnormali**, il valore speciale **NaN** (Not a Number) e l'infinito ($\pm\infty$).

- **Numeri Subnormali** I numeri subnormali (o denormalizzati) sono numeri estremamente piccoli che non possono essere rappresentati utilizzando la normale struttura di virgola mobile. In questi casi, l'esponente è impostato a 0 e la mantissa contiene il valore completo del numero, senza bit implicito.
- **Not a Number (NaN)** Il valore NaN è utilizzato per rappresentare risultati che non sono definiti, come divisioni per zero o radici quadrate di numeri negativi. Ci sono due tipi di NaN: **silenziosi** (quiet NaN) e **segnalanti** (signaling NaN).
- **Infinito** Lo standard prevede due valori per l'infinito: $+\infty$ e $-\infty$, che si ottengono quando il risultato di un'operazione supera la gamma dei numeri rappresentabili.

Precision	Unit Roundoff
fp16 (half)	$4.88 \cdot 10^{-4}$
fp32 (single)	$5.96 \cdot 10^{-8}$
fp64 (double)	$1.11 \cdot 10^{-16}$
fp128 (quad)	$9.63 \cdot 10^{-35}$

Figura 4.3: Comparazione di Unit Roundoff di varie precisioni

4.3 Modello per l'analisi dell'errore in virgola mobile IEEE 754

Grazie al formato in virgola mobile siamo in grado di rappresentare una grande quantità di numeri reali, dobbiamo sempre ricordare però che l'insieme dei numeri in virgola mobile è un insieme finito e proprio per questo rappresentare un numero reale in questo formato introduce inevitabilmente un errore di approssimazione. In particolare, considerando la rappresentazione in virgola mobile come un operatore $\dot{fl}() : \mathbb{R} \to F$ abbiamo:

$$fl(x) = x(1 + \delta) \tag{4.3.1}$$

dove $|\delta| < u$ [10], ovvero dello unit roundoff della precisione in cui stiamo effettuando il calcolo.

Definizione 4.3.1. (Unit Roundoff) In informatica lo unit roundoff, o epsilon di macchina (MACHEPS), è il più piccolo numero ϵ, appartenente a un dato insieme F di numeri in virgola mobile, diverso in valore assoluto da zero, che sommato algebricamente ad un qualsiasi numero n dà un numero diverso da n. Se prendiamo due numeri in F, per esempio a e b, per cui nell'insieme dei corrispondenti reali $a - b < \epsilon$ mentre nel dato insieme F si ottiene $a - b = 0$, ovvero si verifica un fenomeno di cancellazione dei dati.

Più in generale, per una qualsiasi operazione aritmetica $op = +, -, \cdot, /$

$$fl(x \quad op \quad y) = (x \quad op \quad y)(1 + \delta), \quad |d| \leq u \tag{4.3.2}$$

Questo modello è soddisfatto dall'aritmetica IEEE (in assenza di overflow o di underflow).Inoltre, risulta di notevole importanza la costante [11]

$$\gamma_n = \frac{nu}{1 - nu} \quad (nu < 1) \tag{4.3.3}$$

Da ora in poi, si indicherà con $u_[32]$ e $u_[64]$ lo unit roundoff rispettivamente del formato di rappresentazione a 32 e a 64 bit, di cui

$$u_{32} = 2^{-24} \approx 5,96 \times 10^{-8}$$

$$u_{64} = 2^{-53} \approx 1,11 \times 10^{-16}$$

Questo modello si pone come obiettivo quello di avere un limite dell'errore introdotto molto pessimistico, considerando come $nu < 1$ sia un vincolo molto stringente anche per dimensioni modeste di n. In generale, esistono limiti molto più realistici che si possono trovare per implementazioni a blocchi di alcuni algoritmi iterativi oppure esistono studi probabilistici sull'errore, utilizzando la tecnica dello stochastic rounding [12]. Spesso questa tecnica viene utilizzato per migliorare computazioni che utilizzano formati virgola mobile a 16 o 32 bit, al fine di migliorare il risultato finale [2], ma se questo tipo di arrotondamento non è presente a livello hardware, la simulazione a livello software può richiedere diversi cicli di macchina in più. Un analisi probabilistica quindi prenderebbe in considerazione anche la possibilità degli errori di approssimazione di cancellarsi a vicenda, ma per questo primo modello di analisi possiamo considerare i limiti trovati nel caso peggiore. Seguendo i risultati ottenuti da N. Higman in [10] da questo modello otteniamo

$$\begin{cases} |x^T \cdot y - fl(x^T \cdot y)| \leq \gamma_n |x||y| \\ |AB - fl(AB)| \leq \gamma_n |A||B| \end{cases} \tag{4.3.4}$$

Quindi per ogni operazione in virgola mobile effettuata si ottiene una costante di proporzionalità di errore pari a γ_n, dipendente dal numero di operazioni effettuate n. In un algoritmo eseguito da molteplici processi o da molteplici thread, l'errore introdotto sarà inferitore rispetto al singolo calcolo seriale.

L'utilizzo di questo modello sarà fondamentale in seguito per stabilire quali garanzie ci saranno sull'introduzione di una conversione del formato dei dati durante la computazione. Sempre ricordano che i limiti teorici imposti da questa analisi sono da ricondurre ad un caso peggiore, sono quindi utili per l'intento di quantificare l'aumento dell'errore introdotto dalla conversione e non dare una stima vera e proprio dell'errore stesso.

Capitolo 5

Implementazione dell'algortimo

Per l'implementazione del gradiente coniugato in precisione mista sono stati presi in considerazione diversi aspetti importanti, partendo dalla teoria che ha portato alla formulazione dell'algoritmo, fino ad andare a capire quale è il modo migliore per effettuare la conversione da un formato di rappresentazione dei dati all'altro. Analizzando gli aspetti principali che si possono avere come obiettivo quello dell'ottenere uno speedup rispetto alla versione in precisione doppia, ma con accuratezza migliore della precisione singola, mantenendo sempre la correttezza dell'algoritmo. Si possono individuare tre aspetti fondamentali da prendere in considerazione per il raggiungimento di questi obiettivi:

- **Accuratezza**: Come varia l'errore in relazione all'introduzione di conversioni durante l'algoritmo?
- **Speedup**: Quanto pesa la conversione sulle prestazioni?
- **Correttezza**: Come modificare l'implementazione affinché si ottengano i risultati sperati?

Nei seguenti paragrafi si analizzerà come è stata data risposta a queste domande, ottenute attraverso un processo di tentativi ed errori che hanno suggerito quale fosse la strada migliore da seguire.

5.1 Analisi di accuratezza del calcolo

Utilizzando il modello di calcolo precedentemente descritto in 4.3, possiamo sviluppare un confronto tra l'errore di approssimazione introdotta dalla rappresentazione in virgola mobile per un prodotto scalare con e senza conversione. Iniziamo con l'analizzare l'errore introdotto in un prodotto scalare:

$$fl(\mathbf{x}^T \cdot \mathbf{y}) = \sum_{i=1}^{n} fl(x_i y_i) = \sum_{i=1}^{n} (x_i y_i)(1+\delta) =$$

$$= (x_1y_1)(1+\delta) + fl(x_2y_2) + \cdots + fl(x_ny_n) =$$
$$= fl((x_1y_1)(1+\delta) + (x_2y_2)(1+\delta)) + \cdots + fl(x_ny_n) =$$
$$= ((x_1y_1)(1+\delta) + x_1y_1(1+\delta))(1+\delta) + \cdots + fl(x_ny_n) =$$
$$= ((x_1y_1) + x_1y_1)(1+\delta)^2 + \cdots + fl(x_ny_n)$$

svolgendo tutta la ricorrenza si ottiene

$$fl(\mathbf{x}^T \cdot \mathbf{y}) = (x_1y_1)(1+\delta)^n + (x_2y_2)(1+\delta)^n + (x_3y_3)(1+\delta)^{n-1} + \cdots + (x_ny_n)(1+\delta)^2$$

che coerentemente ai risultati ottenuti in [10] risulta

$$\prod_{i=0}^{n}(1+\delta_i)^{\rho_i} = 1 + \theta_n \tag{5.1.1}$$

dove

$$|\theta_n| \leq \frac{nu}{1-nu} =: \gamma_n \tag{5.1.2}$$

ottenendo come risultato finale

$$|x^T \cdot y - fl(x^T \cdot y)| \leq \gamma_n |x||y|$$

Analizzando un operazione di conversione su un vettore abbiamo due possibilità se consideriamo formati di rappresentazione a 32 e 64 bit, o si ha una conversione up da 32 a 64 bit o una conversione down da 64 a 32. In entrambi i casi l'errore introdotto, nel peggiore dei casi, può essere quello di un $\delta : |\delta| < u$, dove u è lo unit roundoff della precisione in cui stiamo effettuando la conversione. Quindi considerando la conversione come un operatore $cast(x) : F_{32} \to F_{64}$ abbiamo che:

$$cast(x) = x(1+\delta), \quad |\delta| < u_{64}$$

andando cosi a modificare la sommatoria del prodotto scalare nel seguente modo:

$$fl(cast(\mathbf{x})^T \cdot \mathbf{y}) = \sum_{i=1}^{n} fl(cast(x_i)y_i) = \sum_{i=1}^{n}(cast(x_i)y_i)(1+\delta) = \sum_{i=1}^{n}(x_iy_i)(1+\delta)^2$$

quindi con un ragionamento analogo a quello precedente otteniamo che:

$$fl(cast(\mathbf{x})^T \cdot \mathbf{y}) = (x_1y_1)(1+\delta)^{2n} + (x_2y_2)(1+\delta)^{2n} + (x_3y_3)(1+\delta)^{2n-1} + \cdots + (x_ny_n)(1+\delta)^4$$

da cui

$$|x^T \cdot y - fl(cast(x)^T \cdot y)| \leq \gamma_{2n} |x||y| \tag{5.1.3}$$

Analogamente possiamo analizzare un operazione in virgola mobile in cui entrambi gli operandi sono sottoposti ad un operazione di conversione partendo da

$$fl(cast(x)cast(y)) = fl(x(1+\delta)y(1+\delta)) = xy(1+\delta)^2 \tag{5.1.4}$$

otteniamo

$$|x^T \cdot y - fl(cast(x)^T \cdot cast(y))| \leq \gamma_{2n}|x||y| \tag{5.1.5}$$

Ora possiamo capire più approfonditamente quanto una computazione in precisione mista differisce da una a precisione singola comparando γ_n e γ_{2n}. Andiamo quindi ad analizzare i valori di n per cui la nuova costante di proporzionalità risulti $\gamma_{2n} < \gamma_n$, ricordando come queste siano costanti di proporzionalità che rappresentano il caso peggiore. Grazie a questo confronto, si cerca di capire se il limite trovato è più o meno stringente nel caso di computazione a precisione mista, piuttosto che a precisione fissa.

$$\gamma_{2n} - \gamma_n = \frac{2nu}{1-2nu} - \frac{nu}{1-nu} =$$

$$= nu(\frac{2}{1-2nu} - \frac{1}{1-nu}) =$$

$$= nu(\frac{2}{1-2nu} - \frac{1}{1-nu}) =$$

$$= nu(\frac{2-2nu-1+2nu}{(1-2nu)(1-nu)}) =$$

$$= nu(\frac{1}{(1-2nu)(1-nu)})$$

perciò cerco i valori di n per cui

$$\gamma_{2n} - \gamma_n < 0$$

allora

$$nu(\frac{1}{(1-2nu)(1-nu)}) < 0$$

e quindi

$$1 - 2nu < 0$$

dato che $nu > 0$ per costruzionee $1 - nu > 0$ perché $nu < 1$ possiamo enunciare il seguente lemma:

Lemma 5.1.1. *Dati due vettori x,y* $\in \mathbb{R}^n$ *di numeri reali, la cui rappresentazione in virgola mobile è rispettivamente* $fl(x)$ *e* $fl(y)$*. Dette* γ_n *e* γ_{2n} *le costanti di proporzionalità che rappresentano l'errore introdotto nel calcolo di un prodotto scalare in precisione singola e precisione mista, allora il limite di errore imposto dalla costante* $\gamma_2 n$ *sarà meno stringente del limite imposto da* $\gamma_n \iff n > \frac{1}{2u}$ *, ovvero*

$$\gamma_{2n} < \gamma_n \iff n > \frac{1}{2u} \tag{5.1.6}$$

Effettuando un rapido calcolo utilizzando i valori dello unit roundoff per 32 e 64 bit si ha:

$$\begin{cases} n_{32} > \frac{1}{2*5,96*10^{-8}} = 8389262 \\ n_{64} > \frac{1}{2*1.11*10^{-16}} = 4,5 * 10^{15} \end{cases}$$

Nel caso a 64 bit il numero ottenuto è ragionevolmente alto da permettere di affermare con sicurezza che la computazione in precisione mista introduce un errore che influenza certamente la computazione.

Nel caso a 32 bit invece potrebbe effettivamente capitare di avere matrici o vettori di un ordine di grandezza assimilabile al valore di n ottenuto. Questo fenomeno avviene in pochissimi casi nella pratica, dato che l'esecuzione di questo algoritmo avviene in parallelo, andando a ridimensionare notevolmente il numero di dati per processo, andando a modificare la diseguaglianza ottenuta nel seguente modo:

$$\frac{n_{32}}{np} < 8389262$$

$$n_{32} < np \cdot 8389262 \tag{5.1.7}$$

dove np indica il numero di processi e/o thread utilizzati nella computazione.

Concludendo questa analisi, si può apprezzare come l'introduzione di operazioni di conversione all'interno dei nuclei di calcolo che si andranno ad utilizzare per l'implementazione dell'algoritmo sono rilevanti nell'errore di approssimazione finale.

Per considerare tutto l'algoritmo ci sarebbe bisogno di uno studio probabilistico, al fine di stimare la cancellazione dell'errore che avviene durante la computazione, ma non era questo l'obiettivo inizialmente posto. Infatti, l'attenzione di questa analisi va posta sulla consapevolezza della misura dell'aumento del limite di accuratezza introdotto in un algoritmo di precisione mista.

Il suggerimento che questa analisi fornisce è quindi quello di utilizzare una conversione up da 32 a 64 bit, in modo da mitigare l'aumento dell'errore introdotto con la conversione, dal termine dello unit roundoff meno elevato rispetto al caso a 32 bit.

5.2 Benchmark sulla conversione nella computazione

Per capire quanto influisca effettivamente la conversione a livello di computazione è necessario effettuare un benchmark per avere delle misurazioni reali riguardanti il tempo impiegato per effettuare una conversione di un vettore.

Il benchmark è stato quindi effettuato su un processore Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz 2200.000 MHz dotato di 40 CPU core fisici. Si guardino i risultati delle figure 5.1 , 5.2, 5.3 e 5.4.

I dati presenti nei grafici sono stati ottenuti rimando costanti nel numero di elementi del vettore gestiti da ogni processo, utilizzando la libreria PSBLAS [13] come base per l'ambiente di parallelizzazione per la gestione della divisione del calcolo tra più processi. Lo standard utilizzato da PSBLAS per lo scambio di messaggi tra processo è stata MPI(Message Passing Interface), utilizzando in particolare la configurazione di tool-chain di compilazione con gnu 12.2.1 e mpich 4.2.2 (come versione di MPI). Inoltre, la compilazione è avvenuta utilizzando il flag -O3 e quindi sfruttando i registri vettoriali del processore. L'utilizzo di PSBLAS è risultato naturale data la natura del problema posto, sapendo che questa libreria è specializzata nella risoluzione di sistemi lineari sparsi [14].

Per effettuare un benchmark significativo quindi, si è fatto crescere il numero di elementi dei vettori presi in considerazione mantenendo 1000000 di elementi per processo, effettuando molteplici run dello stesso nucleo di calcolo per ottenere media e varianza della computazione stessa, seguendo il seguente schema:

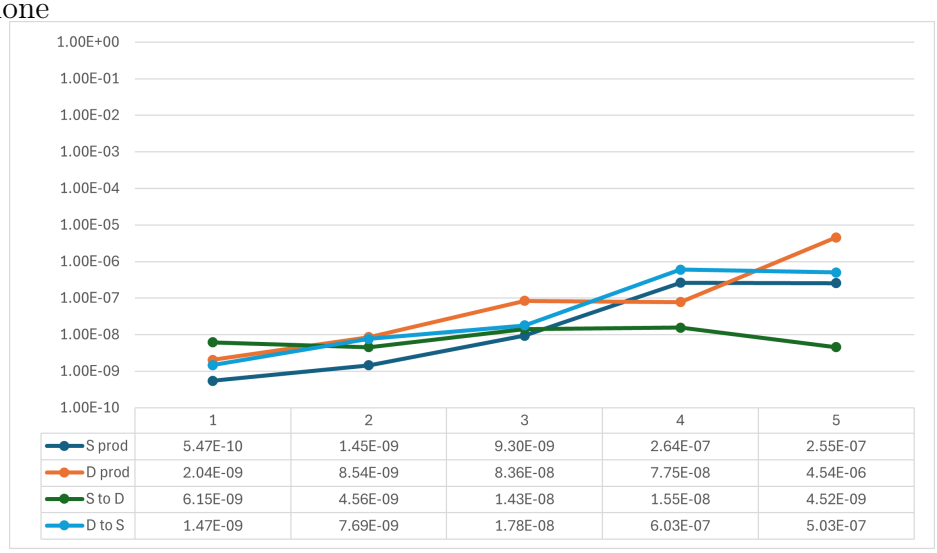
- **Run 1**: 2 processi, 2000000 come grandezza del vettore, 10 run
- **Run 2**: 4 processi, 4000000 come grandezza del vettore, 10 run
- **Run 3**: 8 processi, 8000000 come grandezza del vettore, 10 run
- **Run 4**: 16 processi, 16000000 come grandezza del vettore, 10 run
- **Run 4**: 32 processi, 32000000 come grandezza del vettore, 10 run

I risultati di questo benchmark sono da intendersi per comprendere meglio l'impatto dell'introduzione della conversione all'interno dell'algoritmo, anche se i nuclei di calcolo utilizzati nel Gradiente Coniugato introducono ulteriori aspetti da considerare.

Ad esempio l'introduzione di un overhead da parte del processo di conversione potrebbe essere mitigato da un vantaggio ottenuto successivamente sul trasferimento dei dati, dato che avendo un formato di rappresentazione che utilizza meno bit la larghezza di banda potrebbe non rappresentare più un bottleneck per matrici di elevate dimensioni.

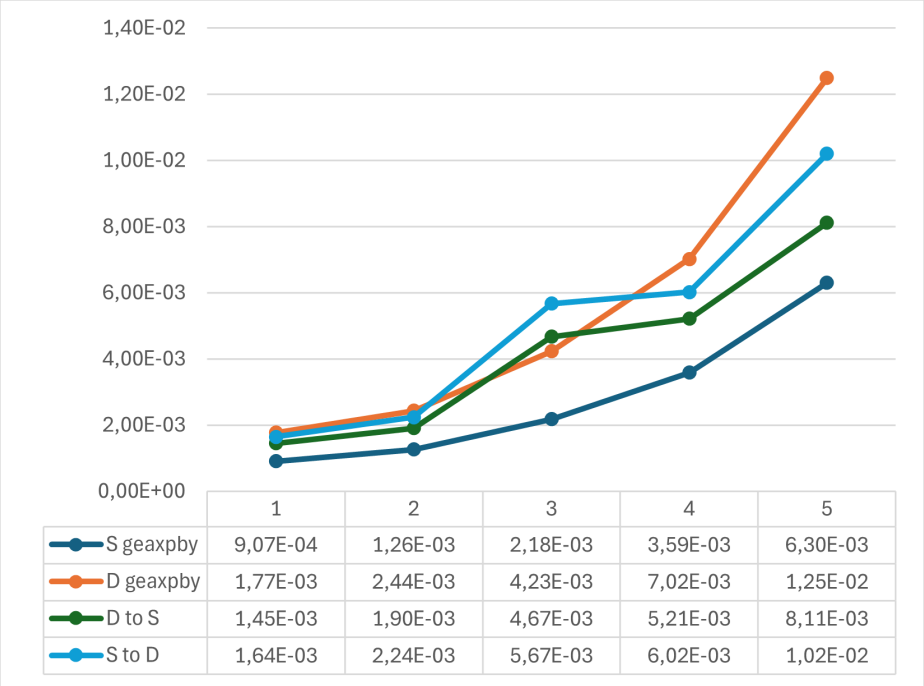
	1	2	3	4	5
S prod	9,12E-04	1,81E-03	3,84E-03	4,87E-03	7,08E-03
D prod	1,63E-03	1,93E-03	5,17E-03	5,40E-03	8,76E-03
S to D	1,64E-03	2,24E-03	5,67E-03	6,02E-03	1,02E-02
D to S	1,45E-03	1,90E-03	4,67E-03	5,21E-03	8,11E-03

Figura 5.1: Comparazione tra la media delle computazioni tra il prodotto scalare e la conversione



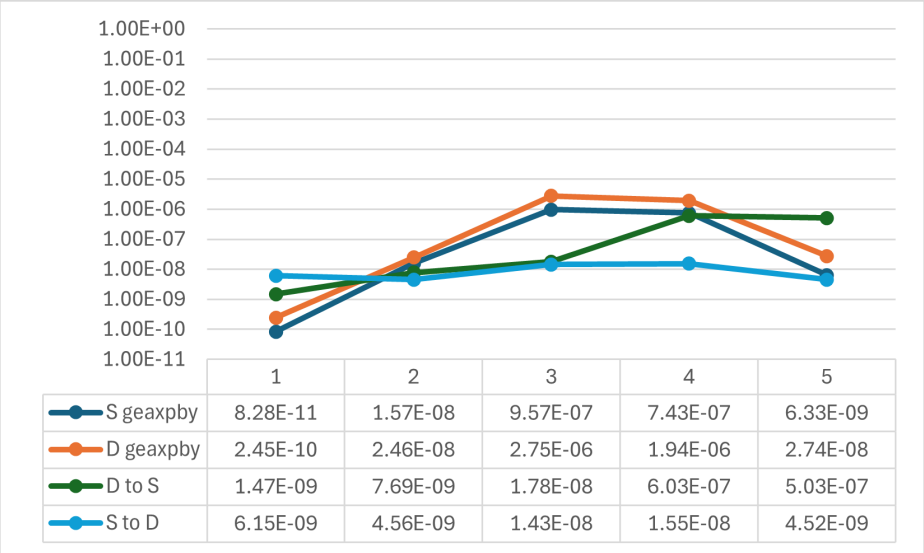
	1	2	3	4	5
S prod	5.47E-10	1.45E-09	9.30E-09	2.64E-07	2.55E-07
D prod	2.04E-09	8.54E-09	8.36E-08	7.75E-08	4.54E-06
S to D	6.15E-09	4.56E-09	1.43E-08	1.55E-08	4.52E-09
D to S	1.47E-09	7.69E-09	1.78E-08	6.03E-07	5.03E-07

Figura 5.2: Comparazione tra la varianza delle computazioni tra il prodotto scalare e la conversione in scala logaritmica



	1	2	3	4	5
S geaxpby	9,07E-04	1,26E-03	2,18E-03	3,59E-03	6,30E-03
D geaxpby	1,77E-03	2,44E-03	4,23E-03	7,02E-03	1,25E-02
D to S	1,45E-03	1,90E-03	4,67E-03	5,21E-03	8,11E-03
S to D	1,64E-03	2,24E-03	5,67E-03	6,02E-03	1,02E-02

Figura 5.3: Comparazione tra la media delle computazioni tra geaxpby e la conversione



	1	2	3	4	5
S geaxpby	8.28E-11	1.57E-08	9.57E-07	7.43E-07	6.33E-09
D geaxpby	2.45E-10	2.46E-08	2.75E-06	1.94E-06	2.74E-08
D to S	1.47E-09	7.69E-09	1.78E-08	6.03E-07	5.03E-07
S to D	6.15E-09	4.56E-09	1.43E-08	1.55E-08	4.52E-09

Figura 5.4: Comparazione tra la varianza delle computazioni tra geaxpby e la conversione in scala logaritmica

Come si può notare i tempi di esecuzione sembrerebbero comparabili, ma se effettuiamo la conversione all'interno dello stesso ciclo in cui effettuiamo il prodotto scalare, notiamo come i tempi non varino praticamente di nulla.

Infatti, effettuando una conversione su una singola locazione di memoria, si può osservare come il tempo di esecuzione scenda drasticamente fino all'ordine di grandezza di 10^{-8}, confermando che l'overhead maggiore consiste nell'accesso alla memoria più che alla conversione vero e proprio.

5.3 Implementazione in precisione mista

Avendo compreso diversi aspetti dell'algoritmo del gradiente coniugato, sono state prese in considerazione diverse possibilità basandosi su lavori di ricerca già svolti [15]. Molti sforzi sono stati fatti nell'analizzare l'utilizzo di precondizionatori in precisione inferiore rispetto all'algoritmo stesso [16].

L'obiettivo che questa versione dell'algoritmo in precisione mista si pone risulta ottenere accuratezza migliore della versione che utilizza la precisione singola, mantenendo delle prestazioni migliori della precisione doppia.

Basandosi sui risultati del benchmark, l'algoritmo deve mantenere i propri nuclei di calcolo classici, andando però ad effettuare la conversione solamente durante la computazione stessa e non in momenti separati, al fine di non introdurre un overhead troppo grande nella computazione.

5.3.1 Progettazione dell'algoritmo

Nell'algoritmo del gradiente coniugato si hanno due principali aspetti da considerare nel prendere un passo verso una soluzione approssimata migliore della precedente:

- La direzione del passo (il vettore d)
- La lunghezza del passo (il coefficiente α)

Considerando un aspetto implementativo l'obiettivo sta nell'utilizzare meno nuclei di calcolo in precisione doppia possibili, solo dove l'introduzione di una precisione maggiore comporti un beneficio dimostrabile.

La prima considerazione ci porta a dire che l'unico prodotto matrice vettore da effettuare per ogni iterazione $\rho_{32}^{(k)} \leftarrow A_{32} \cdot d_{32}^{(k)}$ deve essere eseguito in singola precisione, dato che questo nucleo di calcolo è quello che impatta maggiormente sulla computazione. Invece è interessante vedere come gli algoritmi in precisione mista e doppia, per dimensioni di matrici molto elevate, abbiano un numero diverso di iterazioni. Questo ci suggerisce che nella versione dell'algoritmo in doppia precisione, l'accuratezza del

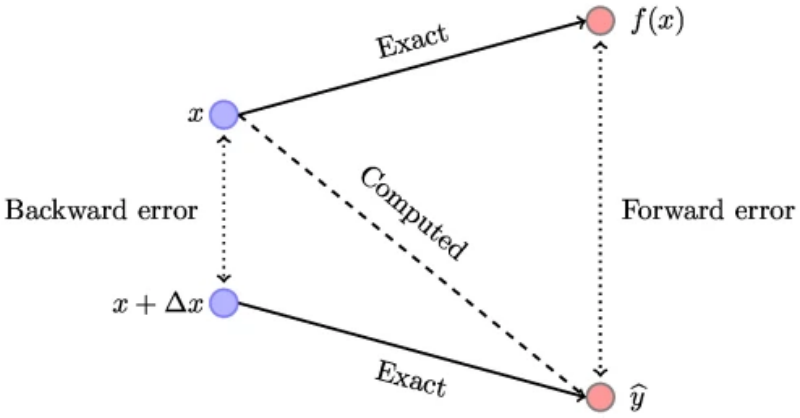


Figura 5.5: Differenza tra forward e backward error

vettore d che mantiene informazioni riguardante la direzione da prendere nell'ottimizzazione della soluzione ha un ruolo importante. Infatti, l'algoritmo del gradiente coniugato si muove ad ogni iterazione in una direzione diversa, proprietà garantita dalla (3.2.3), dato che i vettori d sono A-ortogonali tra loro.

Il primo nucleo di calcolo da convertire in doppia precisione è inevitabilmente quello dell'aggiornamento del vettore d e quindi quello della costruzione del sottospazio di Krylov generato dalla base dei vettori direzione, ovvero il nucleo di calcolo $d_{64}^{(k+1)} \leftarrow r_{64}^{(k+1)} + cast(\beta_{32}^{(k)})_{64} d_{64}^{(k)}$.

Come criterio di stop dell'algoritmo le scelte sono molteplici. Il nostro obiettivo è quello di fermare la computazione una volta ottenuto un risultato che sia apprezzabilmente simile al risultato esatto. Non è possibile misurare la distanza tra la soluzione trovata e la soluzione esatta $\|\mathbf{x} - \mathbf{x}^{(k)}\|$ dato che non è possibile conoscere $\mathbf{x}$. Per avere informazioni riguardanti l'accuratezza di $\mathbf{x}^{(k)}$ si utilizza il concetto di bacward-error [17].

Definizione 5.3.1. (Backward-error) Il Bacward-error è una misura dell'errore associato a una soluzione approssimativa di un problema. Mentre il farwaed-error è la distanza tra la soluzione approssimata e quella vera, il bacward-error rappresenta quanto i dati devono essere perturbati per produrre la soluzione approssimata.

Data una funzione $f() : \mathbb{R}^n \to \mathbb{R}^n$ e una sua approssimazione y, il bacward-error in y è il più piccolo $\Delta x : y = f(x + \Delta x)$, per una misura di dimensione adeguata. Possono esistere molteplici Δx che soddisfano questa equazione, quindi il bacward-error è identificato nella soluzione al problema di minimizzazione di questa funzione. Usando una norma vettoriale e misurando le perturbazioni in senso relativo, possiamo definire il bacward-error in y come:

$$\eta(y) = \min\{\, \epsilon : y = f(x + \Delta x),\ \|\Delta x\| \leq \epsilon \|x\| \,\}. \tag{5.3.1}$$

Nella figura 5.5 si può avere un'intuizione grafica della differenza tra i due errori. Le linee continue indicano le mappature esatte e la linea tratteggiata mostra la mappatura effettivamente calcolata, mostrando come il nostro scopo sia quello di usare Δx come misura dell'accuratezza della soluzione trovata.

L'analisi degli errori utilizzando il backward-error è stata sviluppata e resa popolare da James Wilkinson negli anni '50 e '60 [18]. Lo usò per la prima volta nel contesto del calcolo degli zeri dei polinomi, ma i maggiori successi del metodo arrivarono quando lo applicò ai calcoli di algebra lineare. In letteratura è ormai consolidato considerare il residuo $r = b - Ax$ come bacwar-error, soprattutto nel caso di sistemi lineari sparsi [19]. Essendo proporzionale al farward-error (3.1.7), il residuo indica quanto accurata è la soluzione trovata.

Il nostro criterio di stop dovrà quindi essere una qualche misura del residuo, il quale viene aggiornato ad ogni iterazione all'interno dell'algoritmo del GC. In particolare, il criterio di stop utilizzato in questa versione dell'algoritmo è $\frac{\|r\|}{\|b\|}$ dato che il calcolo della norma di b può essere effettuato una singola volta ad inizio della computazione non dovendo appesantire il resto dell'algoritmo. Inoltre, utilizzare $\frac{\|r\|}{\|b\|}$ introduce una normalizzazione che mitiga la varianza introdotta utilizzando matrici sparse con valori diversi, ovvero sia con valori molto grandi che con valori molto piccoli.

Un ulteriore garanzia di miglioramento riguarda il calcolo del residuo, perché questo determina il numero di iterazioni effettuate. Per questo il nucleo di aggiornamento del residuo ad ogni iterazione diventa $r_{64}^{(k+1)} \leftarrow r_{64}^{(k)} - cast(\alpha_{32}^{(k)})_{64} cast(\rho_{32}^{(k)})_{64}$.

Osservando i risultati dell'introduzione di queste modifiche all'implementazione dell'algoritmo in precisione mista, ci si può accorgere di come la precisione del ricalcolo del residuo abbia un impatto sull'accuratezza della soluzione ottenuta.

Calcolando il residuo esplicitamente al termine dell'algoritmo può mostrare valori diversi rispetto al residuo calcolato all'interno dell'algoritmo stesso. Le motivazioni di questo fenomeno sono riconducibili ad errori di approssimazione introdotti ad ogni iterazione, i quali vengono anche amplificati dall'introduzione della conversione nei nuclei di calcolo (5.1.6), (5.1.7), Per qesto è stato ideato un meccanismo di restart esterno al ciclo standard dell'algoritmo.

L'idea è quella di sfruttare la soluzione ottenuta al termine della computazione come "nuovo input", avendo però un ricalcolo esplicito del residuo in precisione doppia. Il costo da pagare è quello di un numero di iterazioni maggiori, che però vengono in parte compensate dall'utilizzo della doppia precisione durante il calcolo dei vettori di base del sottospazio di Krylov.

Arrivati a questo punto si aprono due possibilità su come calcolare il criterio di stop una volta entrati nella fase di restart:

- Precisione mista $r_{64}^{(k)} \leftarrow cast(b_{32}) - cast(A_{32})_{64} \cdot cast(x_{32}^{(k)})_{64}$
- Precisione doppia $r_{64}^{(k)} \leftarrow cast(b_{32}) - A_{64} \cdot cast(x_{32}^{(k)})_{64}$

Questa può sembrare una differenza impercettibile, ma può avere un impatto sia sul criterio di stop e quindi sul numero di iterazioni, sia sulla quantità di memoria utilizzata, dato che nel secondo caso bisognerebbe avere due versioni distinte della stessa matrice in memoria, dettaglio che può essere non banale nel momento in cui le matrici diventano molto grandi e la memoria RAM potrebbe non bastare per contenere tutti i dati del sistema.

Ne conseguono quindi le seguenti due proposte di algoritmi: che sono stati implementati e messi a confronto attraverso dei risultati ottenuti da una fase di testing (Algoritmo 4), (Algoritmo 5).

I due algoritmi sono una composizione dei nuclei di calcolo in precisione mista precedentemente discussi, con la principale differenza che sta nella conversione degli elementi della matrice A che avviene nella prima versione, a fronte della presenza di una versione della matrice in precisione doppia nella seconda versione.

Altro dettaglio implementativo importante da considerare rispetto alla versione precedentemente discussa in pseudocodice dell'algoritmo sta nell'introduzione del vettore ρ. Questo vettore aumenta la memoria utilizzata, ma riduce il numero di prodotti matrice vettore ad ogni iterazione, salvando il risultato e permettendo quindi una singola esecuzione di questa operazione ad ogni iterazione. Importante inoltre notare come la norma $\|b\|$ viene calcolata un unica volta all'inizio dell'algoritmo. Infatti il calcolo della norma di un vettore è un'operazione che richiede inevitabilmente comunicazione tra i processi, avendo una versione parziale degli elementi del vettore per ogni processo.

Il vettore che viene ritornato alla fine dell'algoritmo è una versione in singola precisione del vettore x. La motivazione sta nell'intenzione di migliorare l'accuratezza della versione in singola precisione del vettore x grazie all'utilizzo di una precisione mista e non andando ad alterare i valori assunti da x utilizzando una precisione maggiore per rappresentarlo.

Algoritmo 4 Conjugate Gradient in mixed-precision v1

Input: $A_{32} \in \mathbb{R}^{n \times n}$, $b_{32} \in \mathbb{R}^n$, $x_{32}^{(0)} \in \mathbb{R}^n$, ϵ_{64}, $k_{\max}$
Output: x_{32}
$k \leftarrow 0$
$b\_norm_{32} \leftarrow ||b_{32}||$
while (true) **do**
    $r_{64}^{(k)} \leftarrow cast(b_{32})_{64} - cast(A_{32})_{64} cast(x_{32}^{(k)})_{64}$
    if $\frac{\|r_{64}^{(k)}\|}{b\_norm_{32}} \leq \epsilon_{64}$ **then**
        break
    $d_{64}^{(k)} \leftarrow r_{64}^{(k)}$
    while (true) **do**
        if $k > k_{\max}$ **then**
            break
        $\rho_{32}^{(k)} \leftarrow A_{32} \cdot d_{32}^{(k)}$
        $\alpha_{32}^{(k)} \leftarrow \frac{r_{32}^{(k)T} r_{32}^{(k)}}{d_{32}^{(k)T} \rho_{(32)}^{(k)}}$
        $x_{32}^{(k+1)} \leftarrow x_{32}^{(k)} + \alpha_{32}^{(k)} d_{32}^{(k)}$
        $r_{64}^{(k+1)} \leftarrow r_{64}^{(k)} - cast(\alpha_{32}^{(k)})_{64} cast(\rho_{32}^{(k)})_{64}$
        if $\frac{\|r_{64}^{(k+1)}\|}{b\_norm_{32}} \leq \epsilon_{64}$ **then**
            break
        $\beta_{32}^{(k)} \leftarrow \frac{r_{32}^{(k+1)T} r_{32}^{(k+1)}}{r_{32}^{(k)T} r_{32}^{(k)}}$
        $d_{64}^{(k+1)} \leftarrow r_{64}^{(k+1)} + cast(\beta_{32}^{(k)})_{64} d_{64}^{(k)}$
        $k \leftarrow k + 1$
return $x_{32}^{(k)}$

Algoritmo 5 Conjugate Gradient in mixed-precision v2

Input: $A_{32} \in \mathbb{R}^{n \times n}$, $A_{64} \in \mathbb{R}^{n \times n}$, $b_{32} \in \mathbb{R}^n$, $x_{32}^{(0)} \in \mathbb{R}^n$, ϵ_{64}, $k_{\max}$
Output: x_{32}
$k \leftarrow 0$
$b\_norm_{32} \leftarrow \|b_{32}\|$
while (true) **do**
    $r_{64}^{(k)} \leftarrow cast(b_{32})_{64} - A_{64} cast(x_{32}^{(k)})_{64}$
    if $\frac{\|r_{64}^{(k)}\|}{b\_norm_{32}} \leq \epsilon_{64}$ **then**
        break
    $d_{64}^{(k)} \leftarrow r_{64}^{(k)}$
    while (true) **do**
        if $k < k_{\max}$ **then**
            break
        $\rho_{32}^{(k)} \leftarrow A_{32} \cdot d_{32}^{(k)}$
        $\alpha_{32}^{(k)} \leftarrow \frac{r_{32}^{(k)T} r_{32}^{(k)}}{d_{32}^{(k)T} \rho_{(32)}^{(k)}}$
        $x_{32}^{(k+1)} \leftarrow x_{32}^{(k)} + \alpha_{32}^{(k)} d_{32}^{(k)}$
        $r_{64}^{(k+1)} \leftarrow r_{64}^{(k)} - cast(\alpha_{32}^{(k)})_{64} cast(\rho_{32}^{(k)})_{64}$
        if $\frac{\|r_{64}^{(k+1)}\|}{b\_norm_{32}} \leq \epsilon_{64}$ **then**
            break
        $\beta_{32}^{(k)} \leftarrow \frac{r_{32}^{(k+1)T} r_{32}^{(k+1)}}{r_{32}^{(k)T} r_{32}^{(k)}}$
        $d_{64}^{(k+1)} \leftarrow r_{64}^{(k+1)} + cast(\beta_{32}^{(k)})_{64} d_{64}^{(k)}$
        $k \leftarrow k + 1$
return $x_{32}^{(k)}$

5.3.2 Problematiche implementative

L'introduzione di un calcolo esplicito all'interno dell'algoritmo ha introdotta una problematica presentatasi dopo i primi test. Infatti, calcolare il residuo in precisione doppia potrebbe risultare un un nuovo inizio del ciclo interno dell'algoritmo. Questo comporta un'ottimizzazione del risultato $x^{(k)}$ che avviene in precisione singola, che però potrebbe non essere in grado di migliorare la soluzione $x^{(k)}$ a tal punto da risultare in un calcolo esplicito un precisione doppia che passi il controllo per il criterio di stop.

Questo genera un loop infinito da cui l'algoritmo non esce mai, andando in stagnazione e non convergendo anche avendo trovato la migliore soluzione approssimata possibile. Per avere uno schema su quello che succede considerare l'algoritmo 4:

- Passo 6: Il residuo viene computato in doppia precisione
- Passo 7: Il criterio di stop non è soddisfatto ...
- Passo 15: Il valore di $x^{(k)}$ viene aggiornato
- Passo 16: Il valore di $r^{(k)}$ viene aggiornato
- Passo 17: Il criterio di stop viene rispettato
- Passo 6: Calcolo esplicito del residuo in doppia precisione
- Passo 7: Il criterio di stop non è soddisfatto
- ...
- Passo 15: Il valore di $x^{(k+1)}$ viene aggiornato
- Passo 16: Il valore di $r^{(k+1)}$ viene aggiornato
- Passo 17: Il criterio di stop viene rispettato
- Passo 6: Calcolo esplicito del residuo in doppia precisione
- Passo 7: Il criterio di stop non è soddisfatto
- ...

Per ovviare a questo problema è stato introdotto un contatore che blocca l'algoritmo quando un certo numero di restart viene eseguito. Questo nuovo parametro ci permette di limitare il numero di moltiplicazioni matrice vettore eseguite in doppia precisione e elimina il problema della stagnazione.

Per le statistiche ottenute nella fase sperimentale quindi è stato fissato il valore di $\frac{r}{b}$ sempre a 10^{-6}. Questo valore rendendo possibile una stagnazione, dato che la singola precisione ha uno unit roundoff molto vicino al limite imposto, Inoltre, l'errore introdotto in ogni computazione è pari a $n \cdot u_{32}$, rendendo cosi possibile che il più piccolo incremento possibile in precisione doppia venga mappato nello stesso elemento in precisione singola durante la conversione. Proprio per questo è stato impostato il nuovo contatore introdotto ad un valore massimo di 2, effettuando cosi al massimo 2 restart, rendendo di fatto nullo il problema della stagnazione.

Rimane però un problema aperto, quello di trovare un meccanismo che elimini questo fenomeno senza dover possibilmente eliminare dei restart leciti all'interno dell'esecuzione dell'algoritmo.

Capitolo 6

Risultati

6.1 Descrizione della computazione

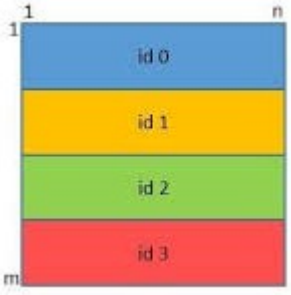
Per l'acquisizione dei dati è stato implementato l'algoritmo in Fortran e CUDA, utilizzando i nuclei di calcolo e l'ambiente parallelo fornito dalla libreria PSBLAS. L'algoritmo è stato realizzato in precisione singola, doppia e nelle due versioni miste, validando il calcolo utilizzando l'esecuzione delle routine del GC presenti in PSBLAS.

I parametri da definire per l'esecuzione dell'algoritmo sono molteplici e sono rimasti fissi per ogni computazione al fine di misurare unicamente differenze date dall'utilizzo di precisione mista e non di variazioni tra un esecuzione e un'altra. Data l'assenza di un precondizionatore il numero massimo di iterazioni considerato è stato 15000 iterazioni, cosi da non far fermare l'algoritmo prematuramente.

Il valore scelto per il backward-error è stato di $10^{-}6$ in modo da avere un criterio di stop calibrato su una precisione effettivamente limitante per il calcolo in precisione singola, in modo da poter apprezzare maggiormente i miglioramenti ottenuti dalla precisione doppia. Dal punto di vista del formato di memorizzazione della matrice dei coefficienti si è utilizzato un formato COO (Coordinate) [20] dal lato CPU e un formato HLL (Hacked ELLPACK) per GPU [21] andnando cosi a rendere la computazione matrice vettore $O(m)$ dove m è il numero degli elementi non nulli della matrice.

Parlando invece di distribuzione dei dati, la computazione è stata effettuata seguendo uno dei classici schemi presenti in BLAS ovvero utilizzando una distribuzione BLOCK ROWS. Ad ogni processo vengono assegnate un numero di righe pari a n/np dove n è la dimensione del problema, mentre np è il numero dei processi utilizzati. L'utilizzo di questo pattern di distribuzione dei dati garantisce un accesso in memoria uniforme, al contrario di altri pattern come BLOCK CYCLYC, rendendo cosi meno rilevante quest'aspetto nell'acquisizione dei tempi di computazione tra le diverse versioni. Questo accesso in memoria potrebbe risultare sfavorevole per processi che contengono righe aventi un numero di elementi diversi da zero molto elevato, proprio per questo nella

computazione su GPU è stato utilizzato il formato HLL. Per comprendere meglio la distribuzione dei dati, la seguente figura mostra una possibile divisione della matrice A nel caso di una distribuzione BLOCK ROWS [22] dove ad ogni processo viene assegnato un colore ed un id differente.



La matrice utilizzata nella computazione è stata generata utilizzando una feature di PSBLAS, avendo a disposizione una matrice derivante dalla discretizzazione di un sistema di PDE (Partial Differential Equation) bidimensionale che risulti in una matrice simmetria e definita positiva, cosi da poter avere la garanzia di convergenza del GC. Inoltre, la generazione della matrice utilizzando PSBLAS garantisce anche un controllo sul numero di elementi diversi da zero presenti, che risuttano essere all'incirca $\frac{n*7}{np}$, dove n è la dimensione della matrice stessa.

Avendo descritto le condizioni al contorno della computazione per renderla replicabile, si può procedere analizzando i risultati ottenuti su CPU e GPU. Si noti che l'algoritmo del GC è stato realizzato modificando alcuni kernel di computazione presenti nella libreria PSBLAS (Fortran) e PSBLAS-EXT (CUDA), anche se l'implementazione vera e propria è stata realizzata esternamente alla libreria. Proprio per questo motivo le routine in precisione mista saranno presenti in PSBLAS dalla versione 4.1 o successive della libreria stessa.

6.2 Risultati CPU

I risultati ottenuti su CPU sono stati registrati a partire da un processore Intel(R) Xeon(R) Silver 4210 da 2.20GHz avente un totale di 40 core. Al fine di capire qualche proprietà sulla scalabilità debole dell'algoritmo sono state effettuate run variando il numero di processi fino alla saturazione dei core presenti nell'hardware di riferimento, mantenendo 1000000 di righe per processo come bilanciamento del carico. Di seguito uno schema delle computazioni effettuate:

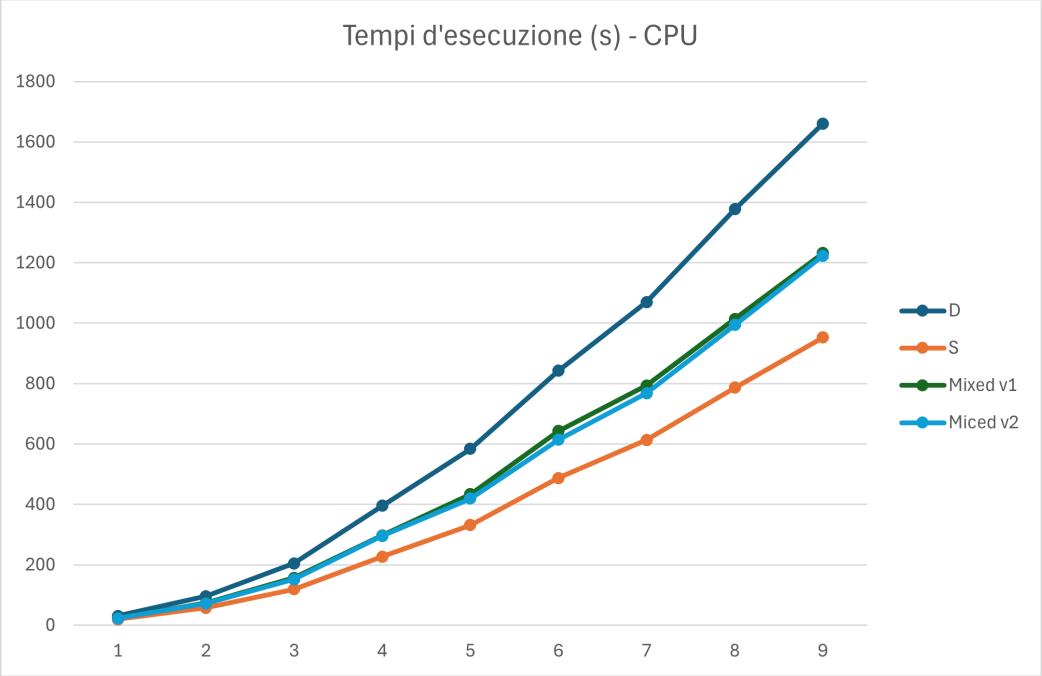


Figura 6.1: Tempi di esecuzione su CPU

- **Run 1**: 1 processo, matrice 1000000 x 1000000
- **Run 2**: 5 processi, matrice 5000000 x 5000000
- **Run 3**: 10 processi, matrice 10000000 x 10000000
- **Run 4**: 15 processi, matrice 15000000 x 15000000
- **Run 5**: 20 processi, matrice 20000000 x 20000000
- **Run 6**: 25 processi, matrice 25000000 x 25000000
- **Run 7**: 30 processi, matrice 30000000 x 30000000
- **Run 8**: 35 processi, matrice 35000000 x 35000000
- **Run 9**: 40 processi, matrice 40000000 x 40000000

Osservando i risultati ottenuti nei grafici 6.1, 6.2 e 6,3 possiamo vedere come la versione del Gradiente Coniugato in precisione mista abbia ottenuto i risultati desiderati. Infatti i tempi registrati rispecchiano pienamente l'utilizzo della precisione singola con l'aggiunta dei tempi di conversione durante le computazioni. Lo speedup ottenuto cosi facendo si aggira intorno al valore di 1,35 al contrario del valore di 1,7 circa

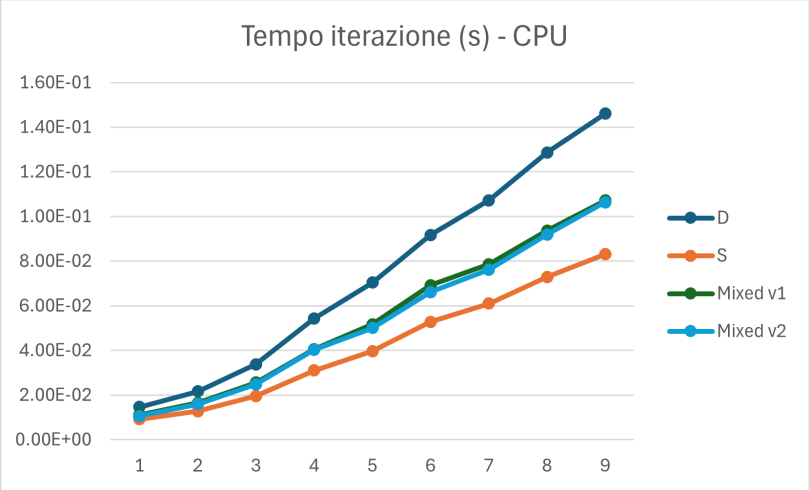


Figura 6.2: Tempi di esecuzione ad ogni iterazione su CPU

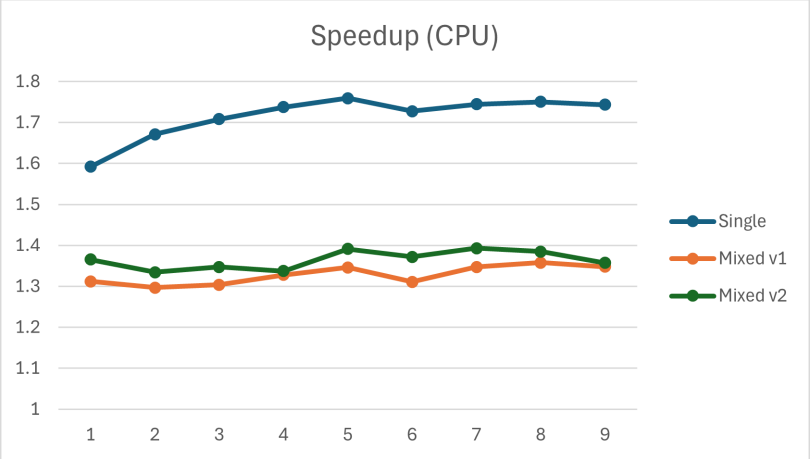


Figura 6.3: Speedup misurato su CPU

ottenuto grazie all'implementazione in precisione singola. Questa metrica è importante per capire quanto è migliorata la velocità d'esecuzione del nostro nuovo algoritmo in relazione all'implementazione dello stesso algoritmo in doppia precisione.

$$speedup_{single} = \frac{time_{double}}{time_{single}}$$

$$speedup_{mixed} = \frac{time_{double}}{time_{mixed}}$$

Inoltre è interessante osservare come le due versioni in precisione mista abbiano un tempo di esecuzione assimilabile, rendendo quello del tempo di esecuzione un fattore che non giustifica l'overhead di memoria introdotto nella seconda versione della precisione mista avendo due copie della matrice A.

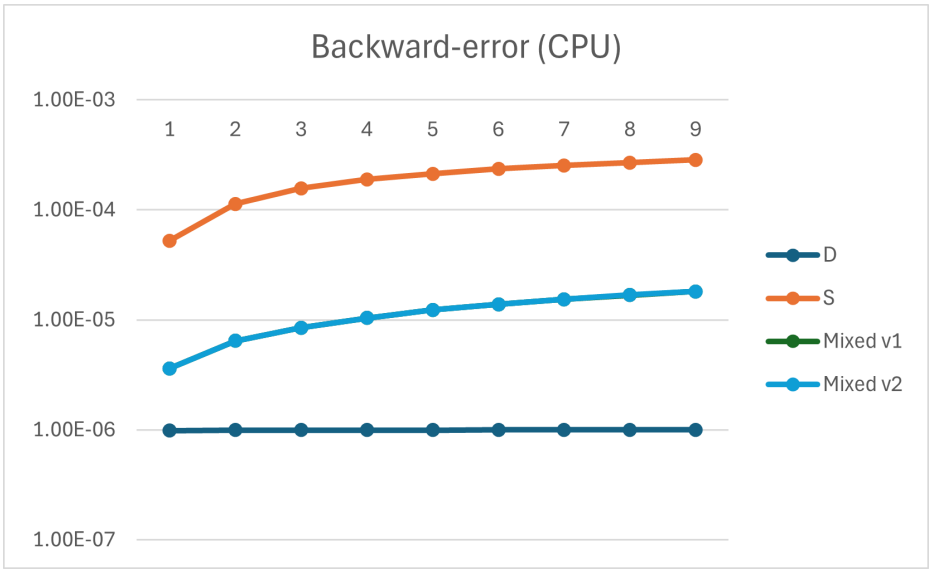


Figura 6.4: Backward-error in scala logaritmica su CPU

Un ulteriore conferma dell'utilità di un implementazione in precisione mista arriva analizzando il backward-error. Infatti, in termini di accuratezza si migliora di uno o due ordini di grandezza, passando da 10^-4 nel caso della precisione singola a un $10^{-6}/10^{-5}$ in precisione mista, avendo sempre come riferimento il 10^{-7} della precisione doppia.

Un riscontro positivo è dato anche dal valore che gli elementi del vettore **r** assumono. Come si può notare dalle figure 6.5 e 6.6, il massimo valore assunto da un elemento di **r** diminuisce drasticamente rispetto alla versione in precisione singola. Per fare un confronto, nella run con 40 processi i valori assunti dal massimo in **r** sono:

- Precisione doppia : 0.344494968
- Precisione singole : 43.78125000
- Precisione mista v1 : 1.281680074
- Precisione mista v2 : 1.251873560

Questo risultato è fondamentale per le applicazioni pratiche per cui questi algoritmi vengono utilizzati. Infatti avere una soluzione che si discosta al massimo di 1.3 per ogni elemento invece di 44 potrebbe essere un buon motivo per rinunciare alla velocità offerta dalla precisione singola per passare alla versione in precisione mista. L'ultimo confronto utili quindi è quello tra la memoria utilizzata in una e nell'altra versione dell'algoritmo. Dato che sembra evidente come in entrambe le versioni in precisione mista tutti i risultati siano assimilabili, avendo però il costo di una maggiore occupazione di memoria nella seconda versione, considereremo unicamente la prima in

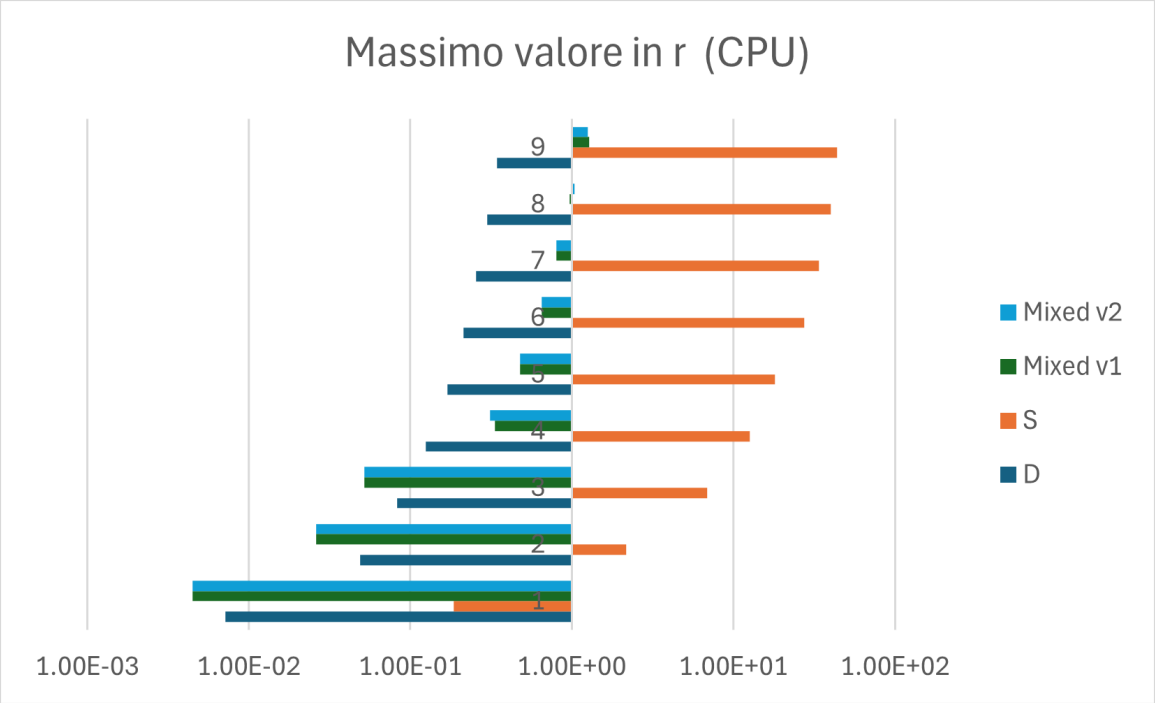


Figura 6.5: Valore del massimo di r in scala logaritmica su CPU

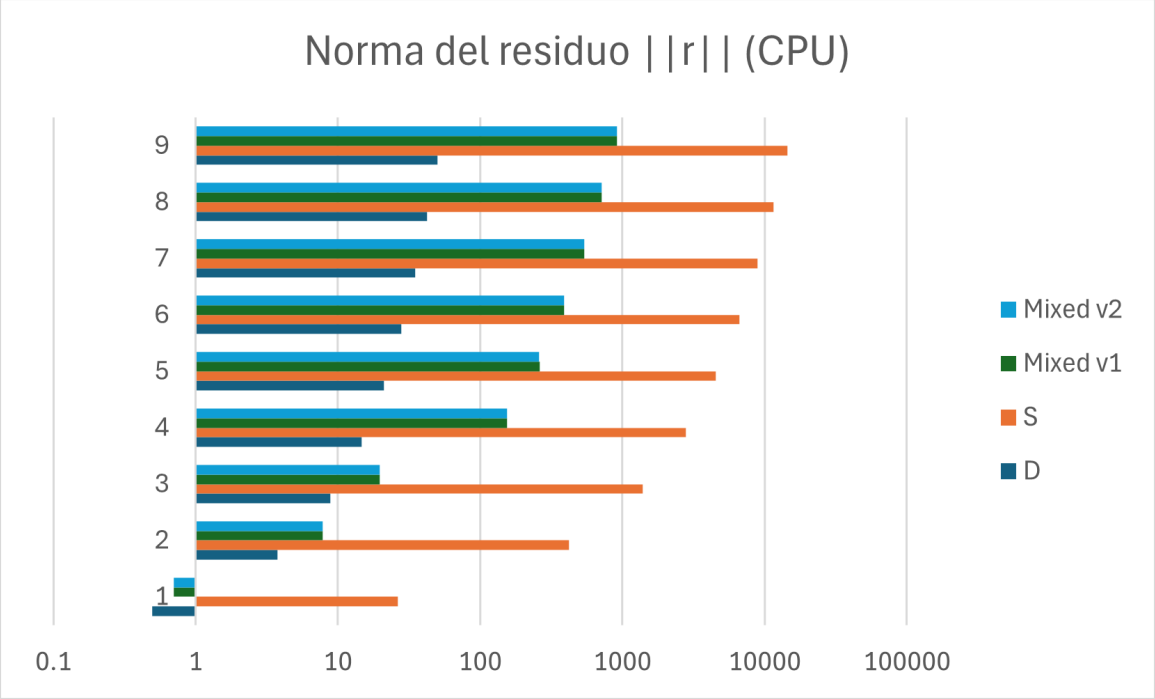


Figura 6.6: Valore della norma di r su CPU

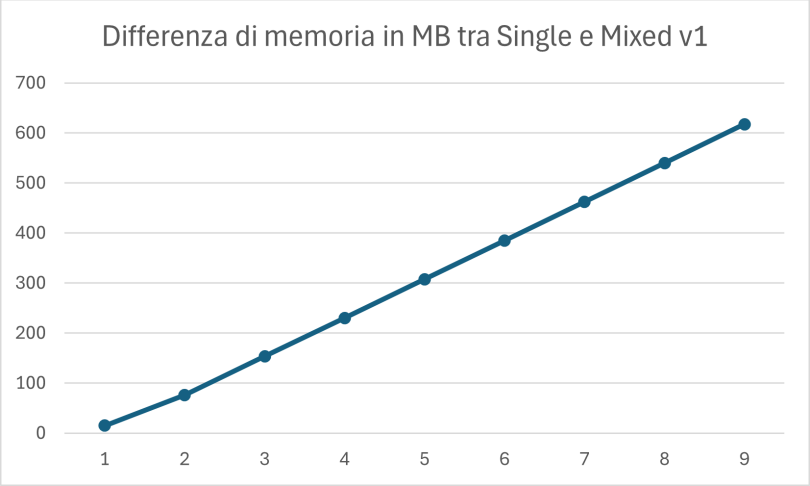


Figura 6.7: Differenza di memoria utilizzata tra precisione singola e mista espressa in MB

relazione alla versione in precisione singola. Come si può vedere in figura l'aumento di memoria utilizzata è costante ed è una minima parte rispetto alla memoria totale utilizzata. Prendendo come esempio una run con 40 processi abbiamo che la memoria utilizzata è:

- Precisione Singola - 3.00885 GB
- Precisione Doppia - 4.50779 GB
- Precisione mista v1 - 3.61214 GB

registriamo quindi un aumento del di 650 MB circa su un totale di memoria occupato di 3 GB, ovvero ci aggiriamo intorno al 20%.

6.3 Risultati GPU

L'acquisizione dei dati su GPU è stata effettuata utilizzando un Quadro RTX 5000 da 1.81 GHz di clock rate avente 3072 CUDA Cores, 16 GB di memoria, una cache L2 da 4 MB, 32 blocchi per warp, 1024 thread pre blocco e 48 KB di memoria condivisa per blocco.

Inoltre, data la presenza di un singolo nodo NUMA all'interno dell'architettura lato host, è stato scelto di utilizzare un unico processo per le computazione effettuate su device. Questo perché l'obiettivo posto era quello di valutare le prestazioni sul device sfruttando al meglio il parallelismo tra thread offerto dai CUDA core, senza introdurre overhead di comunicazione tra processi.

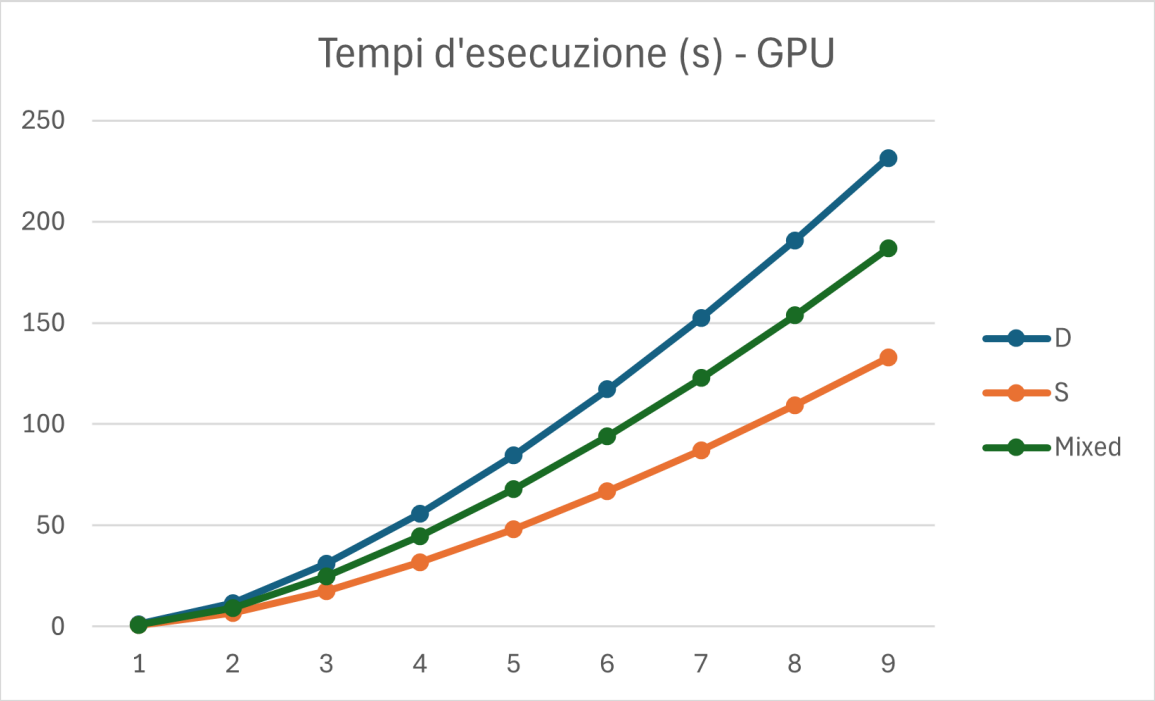


Figura 6.8: Tempi di esecuzione su GPU

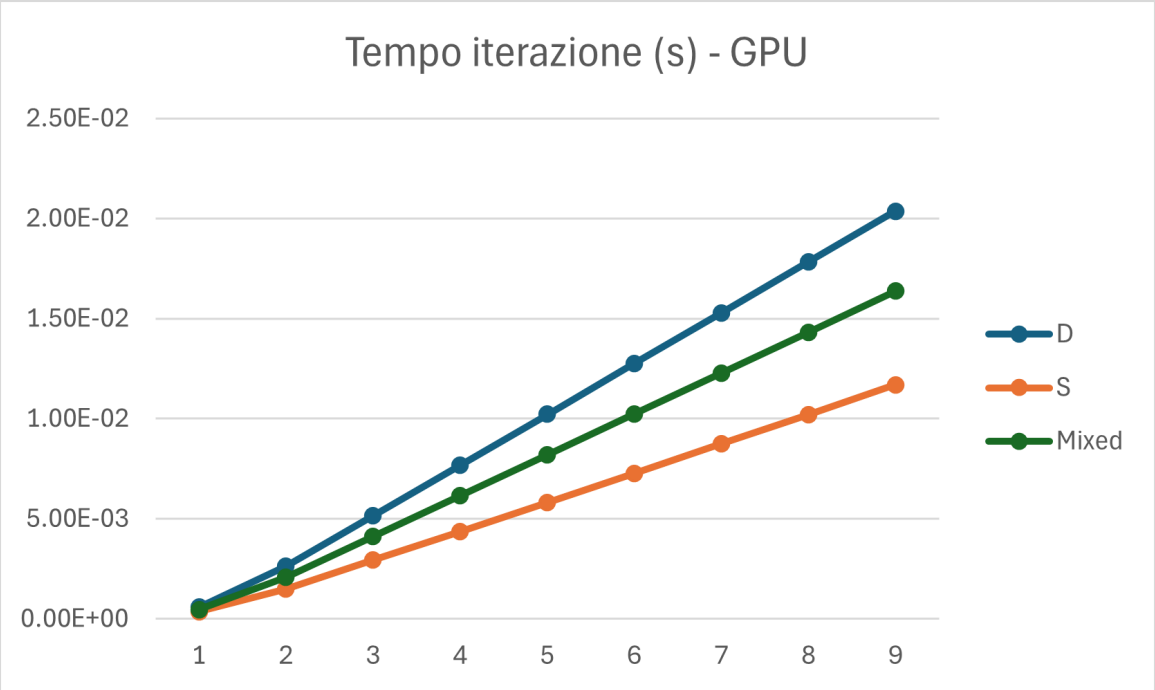


Figura 6.9: Tempi di esecuzione ad ogni iterazione su GPU

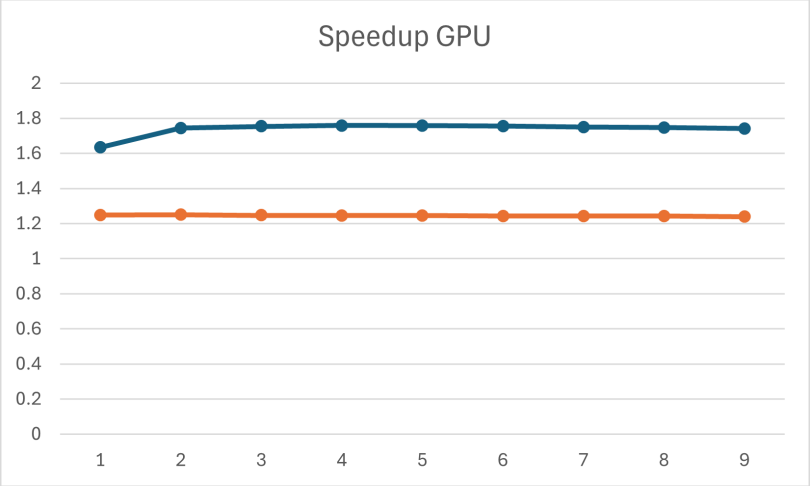


Figura 6.10: Speedup misurato su GPU

Alla luce dei risultati ottenuti su CPU è stata implementata un unica versione dell'algoritmo su GPU. In principio sono state fatte alcune prove con la seconda versione dell'implementazione in precisione mista, ottenendo però dei risultati inutilizzabili. Infatti, nella seconda versione era necessario implementare nuove routine di calcolo che non erano presenti in PSBLAS per far si che non ci fosse scambio di dati tra host e device. Questa è una problematica ricorrente quando si parla di nuclei di calcolo eseguiti su GPU, cercando di evitare qualsiasi scambio di dati non necessario che appesantisce notevolmente la computazione risultando in un vero e proprio bottlneck.

La strada percorsa ha previsto la modifica di alcuni nuclei di calcolo presenti in PSBLAS-EXT, andando ad interfacciarli con nuove routine sviluppate in precisione mista.

Analizzando i risultati su GPU osserviamo un comportamento analogo al al lato CPU, avendo ovviamente una riduzione nei tempi di esecuzione data l'elevata potenza di calcolo offerta del nuovo hardware. Proprio per questo la statistica da considerare per confrontare i dati ottenuti nelle due fasi di sperimentazione è lo speedup. Notiamo una lieve diminuzione da una media su CPU di circa 1,3 ad una media su GPU di circa 1,25 nel caso di precisione mista, mentre riamne stabile sul valore di 1,7 lo speedup nel caso di precisione singola. Questa riduzione dello speedup avviene a causa della struttura delle routine presenti in PSBLAS-EXT, dato che c'è bisogno di chiamare kernel di calcolo aggiuntivi rispetto alla versione CPU per effettuare parte delle conversioni da precisione singola a doppia e viceversa. Questo avviene mantenendo sempre i dati su GPU, ma in cicli differenti, pagando l'overhead introdotto dalla conversione come mostrato nel benchmark.

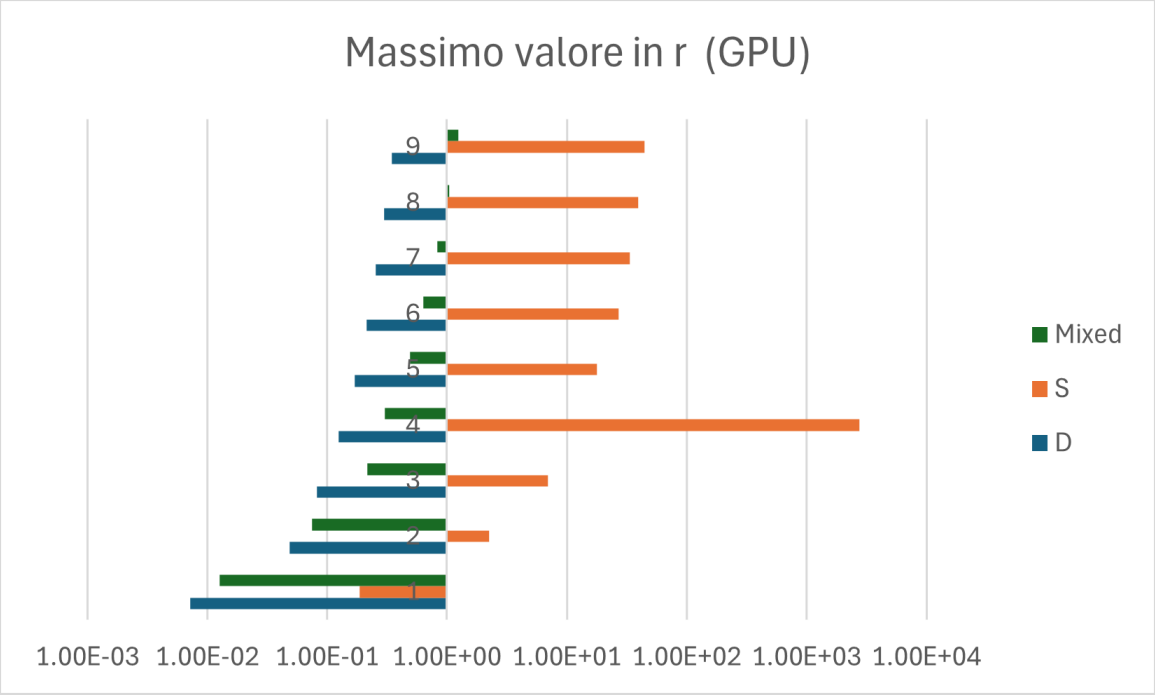


Figura 6.11: Valore del massimo di r in scala logaritmica su GPU

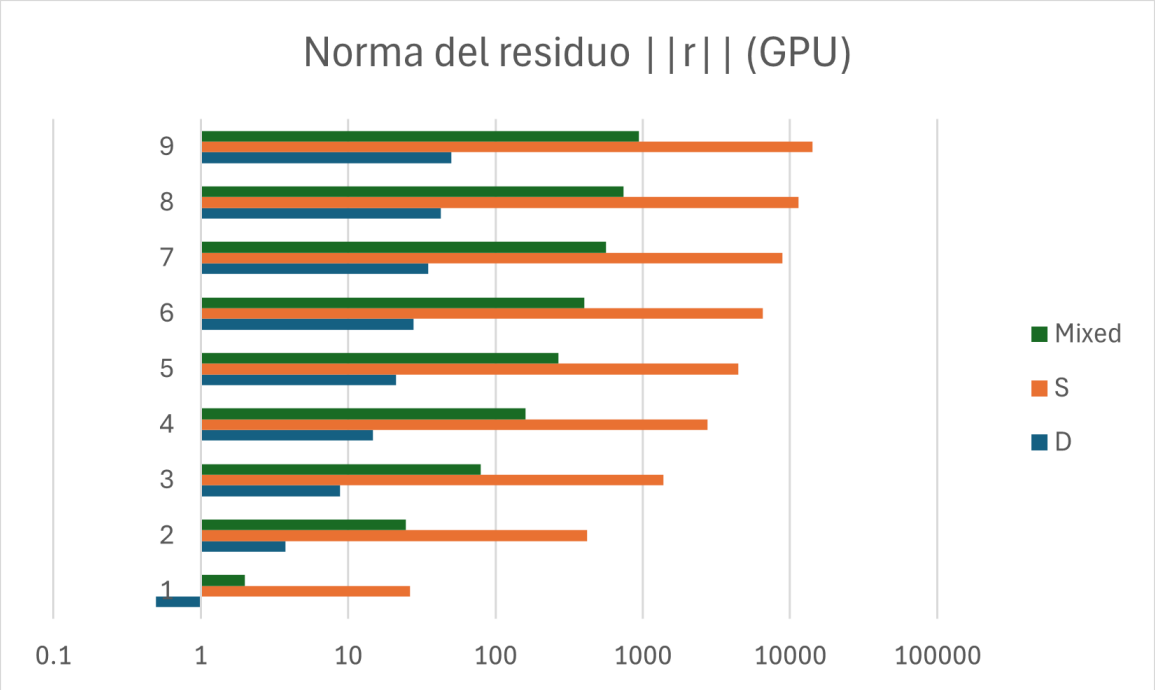


Figura 6.12: Valore della norma di r su GPU

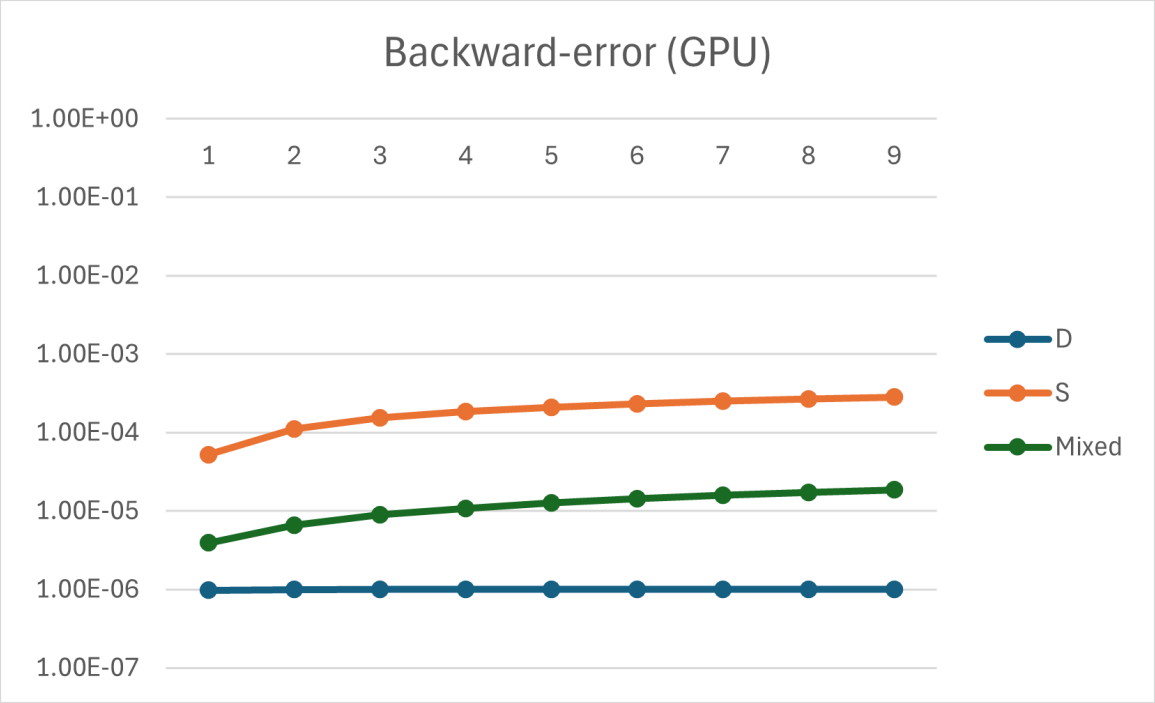


Figura 6.13: Backward-error in scala logaritmica su GPU

Anche nel caso delle statistiche riguardanti l'accuratezza e la memoria utilizzata manteniamo un comportamento analogo tra CPU e GPU, A livello di backward-error e valore massimo di **r** abbiamo risultati assimilabili, dal punto di vista dell'utilizzo della memoria abbiamo una diminuzione della memoria necessaria a rappresentare i coefficienti della matrice A. Questo è dovuto all'utilizzo del formato HLL specifico per GPU. La differenza di memoria utilizzata però sta nella presenza di copie doppie dello stesso vettore all'interno dell'algoritmo in precisione singola e doppia, proprio per questo il grafico relativo alla differenza di memoria utilizzata su GPU è il medesimo che su CPU (6.7). Pertanto valgono le considerazioni fatte poco prima parlaando di CPU.

6.4 Tabelle dei risultati

Per completezza vengono riportate di seguito le tabelle contenenti tutti i dati ottenuti durante la fase sperimentale da cui sono stati realizzati i grafici di cui sopra. Tutti i dati sono stati raccolti effettuando misurazioni sfruttando il timer temporale fornito da PSBLAS. In particolare, sono state utilizzate le routine psb_wtime() per ottenere una misurazione del tempo istantanea e psb_amx per ottenere la misurazione del processo più lento durante la computazione.

Double precision Conjugate Gradient - I (CPU)					
Processi	1	5	10	15	20
Iterazioni	2117	4426	6036	7267	8292
Tempo (s)	31.1393457	95.89911005	204.1636902	395.3297189	583.7599334
Tempo per iterazione	1.47E-02	2.17E-02	3.38E-02	5.44E-02	7.04E-02
Dimensioni matrice (n)	1000000	5000000	9998244	14992384	19998784
$\|\|\mathbf{r}\|\|/\|\|\mathbf{b}\|\|$	9.92E-07	9.99E-07	9.95E-07	9.99E-07	9.96E-07
Max r_i	7.19E-03	4.89E-02	8.27E-02	0.124827691	0.169044287
$\|r\|$	0.497148174	3.739232497	8.851776497	14.74553721	21.07041109
A mem	76.23292 MB	381.31012 MB	762.6126 MB	1.11679 GB	1.48976 GB
Desc memm	64 B	769.59375 KB	2.39061 MB	4.55339 MB	7.13701 MB
Vect mem	38.14697 MB	191.40564 MB	383.57391 MB	576.0498 MB	769.37561 MB

Tabella 6.1: Double precision Conjugate Gradient - I (CPU)

Double precision Conjugate Gradient - II (CPU)				
Processi	25	30	35	40
Iterazioni	9184	9979	10704	11371
Tempo (s)	842.6976777	1070.216634	1376.975278	1660.820467
Tempo per iterazione	9.18E-02	0.107246882	1.29E-01	0.146057556
Dimensioni matrice (n)	25000000	29997529	34999056	39992976
$\|\|\mathbf{r}\|\|/\|\|\mathbf{b}\|\|$	9.98E-07	9.98E-07	9.97E-07	9.99E-07
Max r_i	0.213074233	0.255883631	0.300035083	0.344494968
$\|r\|$	27.88644535	35.04718598	42.44355681	50.23179015
A mem	1.86235 GB	2.23466 GB	2.60728 GB	2.97933 GB
Desc memm	10.07953 MB	13.34142 MB	16.89558 MB	20.71709 MB
Vect mem	962.82959 MB	1.12933 GB	1.3188 GB	1.50823 GB

Tabella 6.2: Double precision Conjugate Gradient - II (CPU)

Single precision Conjugate Gradient - I (CPU)					
Processi	1	5	10	15	20
Iterazioni	2120	4474	6084	7318	8349
Tempo (s)	19.55688235	57.38979192	119.5338353	227.4966585	331.849368
Tempo per iterazione	9.22E-03	1.28E-02	1.96E-02	3.11E-02	3.97E-02
Dimensioni matrice (n)	1000000	5000000	9998244	14992384	19998784
$\|\|\mathbf{r}\|\|/\|\|\mathbf{b}\|\|$	5.26E-05	1.13E-04	1.57E-04	1.89E-04	2.13E-04
Max r_i	0.186035156	2.171875	6.859375	12.5625	18.078125
$\|r\|$	26.363657	422.6459351	1394.837891	2784.218994	4512.915039
A mem	57.17469 MB	285.9826 MB	571.95948 MB	857.69402 MB	1.11732 GB
Desc memm	64 B	769.59375 KB	2.39061 MB	4.55339 MB	7.13701 MB
Vect mem	19.07349 MB	95.70282 MB	191.78696 MB	288.0249 MB	384.68781 MB

Tabella 6.3: Single precision Conjugate Gradient - I (CPU)

Single precision Conjugate Gradient - II (CPU)				
Processi	25	30	35	40
Iterazioni	9245	10051	10780	11455
Tempo (s)	487.7531144	613.4805667	786.591807	952.5157959
Tempo per iterazione	5.28E-02	6.10E-02	7.30E-02	8.32E-02
Dimensioni matrice (n)	25000000	29997529	34999056	39992976
$\|\|\mathbf{r}\|\|/\|\|\mathbf{b}\|\|$	2.37E-04	2.54E-04	2.70E-04	2.86E-04
Max r_i	27.53125	33.875	40.0625	43.78125
$\|r\|$	6615.149414	8898.696289	11479.97266	14383.65332
A mem	1.39676 GB	1.676 GB	1.95546 GB	2.2345 GB
Desc memm	10.07953 MB	13.34142 MB	16.89558 MB	20.71709 MB
Vect mem	481.41479 MB	578.21646 MB	675.22705 MB	772.2139 MB

Tabella 6.4: Single precision Conjugate Gradient - II (CPU)

Mixed precision Conjugate Gradient v1 - I (CPU)					
Processi	1	5	10	15	20
Iterazioni	2130	4488	6105	7338	8375
Tempo (s)	23.73404421	73.9606829	156.5627147	297.6814464	433.6319539
Tempo per iterazione	1.11E-02	1.65E-02	2.56E-02	4.06E-02	5.18E-02
Dimensioni matrice (n)	1000000	5000000	9998244	14992384	19998784
$\|\|\mathbf{r}\|\|/\|\|\mathbf{b}\|\|$	1.40E-06	2.08E-06	2.21E-06	1.04E-05	1.23E-05
Max r_i	4.48E-03	2.61E-02	5.22E-02	0.33527901	0.477021432
$\|r\|$	0.702245831	7.773822952	19.67454736	153.6676064	260.8879717
A mem	57.17469 MB	285.9826 MB	571.95948 MB	857.69402 MB	1.11732 GB
Desc memm	64 B	769.59375 KB	2.39061 MB	4.55339 MB	7.13701 MB
Vect mem	34.33228 MB	172.26508 MB	345.21652 MB	518.44482 MB	692.43805 MB

Tabella 6.5: Mixed precision Conjugate Gradient v1 - I (CPU)

Mixed precision Conjugate Gradient v1 - II (CPU)				
Processi	25	30	35	40
Iterazioni	9278	10085	10818	11497
Tempo (s)	642.8235673	794.1742715	1014.247445	1232.263617
Tempo per iterazione	6.93E-02	7.87E-02	9.38E-02	0.107181318
Dimensioni matrice (n)	25000000	29997529	34999056	39992976
$\|\|\mathbf{r}\|\|/\|\|\mathbf{b}\|\|$	1.39E-05	1.54E-05	1.68E-05	1.81E-05
Max r_i	0.652186599	0.804890263	0.965150366	1.281680074
$\|r\|$	388.9309507	541.584269	715.7976874	912.2105994
A mem	1.39676 GB	1.676 GB	1.95546 GB	2.2345 GB
Desc memm	10.07953 MB	13.34142 MB	16.89558 MB	20.71709 MB
Vect mem	866.54663 MB	1.0164 GB	1.18692 GB	1.35741 GB

Tabella 6.6: Mixed precision Conjugate Gradient v1 - II (CPU)

Mixed precision Conjugate Gradient v2 - I (CPU)					
Processi	1	5	10	15	20
Iterazioni	2130	4488	6105	7338	8375
Tempo (s)	22.80105722	71.84887637	151.5688137	295.6565852	419.6171898
Tempo per iterazione	1.07E-02	1.60E-02	2.48E-02	4.03E-02	5.01E-02
Dimensioni matrice (n)	1000000	5000000	9998244	14992384	19998784
$\|\|\mathbf{r}\|\|/\|\|\mathbf{b}\|\|$	1.40E-06	2.08E-06	2.21E-06	1.04E-05	1.23E-05
Max r_i	4.48E-03	2.61E-0	2 5.22E-02	0.312927076	0.477021432
$\|r\|$	0.702004721	7.785931411	19.66482252	153.6935043	260.6248315
A mem	133.40762 MB	667.29272 MB	1.30329 GB	1.95438 GB	2.60708 GB
Desc memm	64 B	769.59375 KB	2.39061 MB	4.55339 MB	7.13701 MB
Vect mem	34.33228 MB	172.26508 MB	345.21652 MB	518.44482 MB	692.43805 MB

Tabella 6.7: Mixed precision Conjugate Gradient v2 - I (CPU)

Mixed precision Conjugate Gradient v2 - II (CPU)				
Processi	25	30	35	40
Iterazioni	9278	10085	10818	11497
Tempo (s)	614.1856809	768.1721877	994.2590229	1223.61136
Tempo per iterazione	6.62E-02	7.62E-02	9.19E-02	0.106428752
Dimensioni matrice (n)	25000000	29997529	34999056	39992976
$\|\|\mathbf{r}\|\|/\|\|\mathbf{b}\|\|$	1.39E-05	1.54E-05	1.68E-05	1.81E-05
Max r_i	0.652186599	0.804890263	1.043405801	1.25187356
$\|r\|$	388.5101091	541.4716074	716.1322122	912.6398968
A mem	3.25911 GB	3.91066 GB	4.56274 GB	5.21383 GB
Desc memm	10.07953 MB	13.34142 MB	16.89558 MB	20.71709 MB
Vect mem	866.54663 MB	1.0164 GB	1.18692 GB	1.35741 GB

Tabella 6.8: Mixed precision Conjugate Gradient v2 - II (CPU)

Double precision Conjugate Gradient - I (GPU)					
Processi	1	1	1	1	1
Iterazioni	2117	4426	6036	7267	8292
Tempo (s)	1.236609372	11.61741248	31.09668653	55.82989144	84.71863787
Tempo per iterazione	5.84E-04	2.62E-03	5.15E-03	7.68E-03	1.02E-02
Dimensioni matrice (n)	1000000	5000000	9998244	14992384	19998784
$\|\|\mathbf{r}\|\|/\|\|\mathbf{b}\|\|$	9.92E-07	9.99E-07	9.95E-07	9.99E-07	9.96E-07
Max r_i	7.19E-03	4.89E-02	8.27E-02	0.124827694	0.169044285
$\|r\|$	0.497148174	3.739232497	8.851776501	14.74553721	21.07041109
A mem	64.94637 MB	324.77559 MB	649.50602 MB	973.95253 MB	1.26875 GB
Desc memm	68 B	68 B	68 B	68 B	68 B
Vect mem	38.14697 MB	191.40564 MB	383.57391 MB	576.0498 MB	769.37561 MB

Tabella 6.9: Double precision Conjugate Gradient - I (GPU)

Double precision Conjugate Gradient - II (GPU)				
Processi	1	1	1	1
Iterazioni	9184	9979	10704	11371
Tempo (s)	117.1772493	152.6099871	190.8889711	231.5389824
Tempo per iterazione	1.28E-02	1.53E-02	1.78E-02	2.04E-02
Dimensioni matrice (n)	25000000	29997529	34999056	39992976
$\|\|\mathbf{r}\|\|/\|\|\mathbf{b}\|\|$	9.98E-07	9.98E-07	9.97E-07	9.99E-07
Max r_i	0.21307423	0.255883627	0.300035083	0.344494966
$\|r\|$	27.88644535	35.04718598	42.44355682	50.23179015
A mem	1.58605 GB	1.90311 GB	2.22043 GB	2.53727 GB
Desc memm	68 B	68 B	68 B	68 B
Vect mem	962.82959 MB	1.12933 GB	1.3188 GB	1.50823 GB

Tabella 6.10: Double precision Conjugate Gradient - II (GPU)

Single precision Conjugate Gradient - I (GPU)					
Processi	1	1	1	1	1
Iterazioni	2117	4426	6036	7267	8292
Tempo (s)	0.756776101	6.663434529	17.72591129	31.72641734	48.1887912
Tempo per iterazione	3.57E-04	1.51E-03	2.94E-03	4.37E-03	5.81E-03
Dimensioni matrice (n)	1000000	5000000	9998244	14992384	19998784
$\|\|\mathbf{r}\|\|/\|\|\mathbf{b}\|\|$	5.25E-05	1.12E-04	1.55E-04	1.87E-04	2.11E-04
Max r_i	1.87E-01	2.25390625	6.921875	2763.652832	17.8359375
$\|r\|$	26.30088043	418.1888733	1377.905518	2763.652832	4464.197266
A mem	45.88045 MB	229.43062 MB	458.82816 MB	688.02504 MB	917.78786 MB
Desc memm	68 B	68 B	68 B	68 B	68 B
Vect mem	19.07349 MB	95.70282 MB	191.78696 MB	288.0249 MB	384.68781 MB

Tabella 6.11: Single precision Conjugate Gradient - I (GPU)

Single precision Conjugate Gradient - II (GPU)				
Processi	1	1	1	1
Iterazioni	9184	9979	10704	11371
Tempo (s)	66.78419376	87.15177769	109.2351247	132.9572015
Tempo per iterazione	7.27E-03	8.73E-03	1.02E-02	1.17E-02
Dimensioni matrice (n)	25000000	29997529	34999056	39992976
$\|\|\mathbf{r}\|\|/\|\|\mathbf{b}\|\|$	2.33E-04	2.51E-04	2.68E-04	2.83E-04
Max r_i	27	33.65625	39.625	44.6875
$\|r\|$	6503.936523	8826.452148	11400.42871	14240.73828
A mem	1.12042 GB	1.3444 GB	1.56857 GB	1.79239 GB
Desc memm	68 B	68 B	68 B	68 B
Vect mem	481.41479 MB	578.21646 MB	675.22705 MB	772.2139 MB

Tabella 6.12: Single precision Conjugate Gradient - II (GPU)

Mixed precision Conjugate Gradient - I (GPU)					
Processi	1	1	1	1	1
Iterazioni	2125	4437	6052	7288	8318
Tempo (s)	0.989450014	9.289650819	24.92846682	44.79670675	68.05019307
Tempo per iterazione	4.66E-04	2.09E-03	4.12E-03	6.15E-03	8.18E-03
Dimensioni matrice (n)	1000000	5000000	9998244	14992384	19998784
$\|\|\mathbf{r}\|\|/\|\|\mathbf{b}\|\|$	3.95E-06	6.60E-06	8.89E-06	1.09E-05 1.27E-05	
Max r_i	1.27E-02	7.46E-02	2.16E-01	0.301751109	0.491928352
$\|r\|$	1.981556036	24.69472365	79.09112501	160.4690115	268.3068952
A mem	45.88045 MB	229.43062 MB	458.82816 MB	688.02504 MB	917.78786 MB
Desc memm	68 B	68 B	68 B	68 B	68 B
Vect mem	34.33228 MB	172.26508 MB	345.21652 MB	518.44482 MB	692.43805 MB

Tabella 6.13: Mixed precision Conjugate Gradient - I (GPU)

Mixed precision Conjugate Gradient - II (GPU)				
Processi	1	1	1	1
Iterazioni	9215	10015	10744	11414
Tempo (s)	94.17183515	122.7482299	153.6934516	186.8546253
Tempo per iterazione	1.02E-02	1.23E-02	1.43E-02	1.64E-02
Dimensioni matrice (n)	25000000	29997529	34999056	39992976
$\|\|\mathbf{r}\|\|/\|\|\mathbf{b}\|\|$	1.44E-05	1.60E-05	1.74E-05	1.87E-05
Max r_i	0.633552696	0.827248326	1.043405801	1.25187356
$\|r\|$	403.0184746	563.1934701	741.7706946	938.973075
A mem	1.12042 GB	1.3444 GB	1.56857 GB	1.79239 GB
Desc memm	68 B	68 B	68 B	68 B
Vect mem	866.54663 MB	1.0164 GB	1.18692 GB	1.35741 GB

Tabella 6.14: Mixed precision Conjugate Gradient - II (GPU)

6.5 Conclusioni

Per concludere è importante fare un punto della situazione, accentuando il lavoro svolto e i possibili miglioramenti, puntando a definire degli sviluppi futuri. Prima di partire però, bisogna sempre ricordare l'obiettivo di partenza, ovvero quello di sviluppare una nuova implementazione di un metodo iterativo conosciuto, sfruttando al meglio le risorse hardware presenti nei principali calcolatori elettronici presenti nei centri di calcolo,

6.5.1 Riepilogo

La strada per ottenere i risultati appena discussi è stata molto lunga, ponendosi come obiettivo sempre chiaro in testa quello di capire al meglio gli aspetti teorici dell'argomento al fine di estrapolarne un uso pratico, cosi come principio dell'analisi numerica. Perciò, sarebbe importante fissare chiaramente i punti principali con i quali questa implementazione è stata progettata e realizzata:

Per prima cosa è stato effettuato uno studio alla ricerca dello stato dell'arte riguardante gli algoritmi in precisione mista. Avendo compreso i metodi attualmente implementati in letteratura si è pensato di dover realizzare una versione in precisione mista di un algoritmo specifico, per esplorare un campo di ricerca che non fosse stato ancora visionato. Quindi, si è proceduti effettuando uno studio sui principali metodi iterativi per la risoluzione di matrici sparse, al fine di comprenderli al meglio e di individuare le proprietà utili per il nostro scopo. La scelta è ricaduta sull'algoritmo del Gradiente Coniugato, sia per la sua composizione, sia per la sua semplicità rispetto ad altri algoritmi come BiCGSTAB o GMRES.

Successivamente si è passati allo studio di un modello di teoria dell'errore per comprendere al meglio gli effetti della conversione dei dati. Per questo si è partiti da un analisi dello standard IEEE 754 di rappresentazione dei dati in virgola mobile, prartendo dalle basi poste da N. Higham [10], al fine di sviluppare un analisi del nostro caso di studio specifico.

Una volta ottenuta la costante di proporzionalità γ_{2n} all'errore introdotto durante la conversione dei dati, è stato possibile procedere a pensare ad aspetti implementativi. Infatti, aver determinato un aumento controllato dell'errore di approssimazione introdotto dalla rappresentazione in virgola mobile, rappresenta una certezza di avere un errore quantificabile, anche se per eccesso. Si è poi proceduti effettuando un benchmark per comparare le principali routine presenti in PSBLAS con le quali si è realizzata l'implementazione dell'algoritmo (psb_gdot e psb_geaxpby). Il benchmark ha mostrato come la conversione avesse peso nella computazione solo se effettuata separatamente dai nuclei di calcolo, dato che il costo predominante riguardava gli accessi in memoria piuttosto che la conversione in se.

Infine si è passati alla progettazione e all'implementazione vera e propria dei nuclei di calcolo necessari, andando a modificare la libreria PSBLAS per i calcoli su CPU e la sua estensione PSBLAS-EXT per i calcoli su GPU, in particolare la sotto-libreria SPGPU. Per concludere si sono effettuati gli esperimenti del caso, andando a verificare i miglioramenti apportati dall'utilizzo della precisione mista direttamente all'interno dell'algoritmo. I risultati ottenuti confermano quindi le ipotesi iniziali basate sullo studio teorico dello stato dell'arte e dell'algoritmo.

6.5.2 Sviluppi futuri

Gli sviluppi che possono estendersi da questo lavoro sono molteplici. La ricerca nell'ambito dell'utilizzo di formati di rappresentazione dati misti sta diventando sempre più centrale nel campo nel calcolo scientifico, rendendo sempre più importante la presenza di figure legate al mondo dell'informatica che però abbiano una buona conoscenza del problema posto in esame. Proprio per questo sarebbe importante continuare su diversi fronti:

- **Analisi dell'errore**: L'analisi dell'errore effettuata in questo lavoro ha come finalità quella di dimostrare l'esistenza di un limite teorica di approssimazione nel caso di conversione dei dati, sarebbe opportuno effettuare un'analisi probabilistica al fine di considerare anche la cancellazione dell'errore dovuto ad operazioni consecutive in virgola mobile.

- **Generalizzazione dell'implementazione**: L'implementazione proposta ha mostrato come l'idea di effettuare un calcolo esplicito del residuo in precisione maggiore porti ad ottenere risultati ampiamente migliori per quanto riguarda il backward-error, sarebbe interessante studiare come applicare lo stesso approccio ad altri metodi iterativi.

- **Studio dell'influenza dei precondizionatori**: Il numero di iterazioni registrate ad ogni esecuzione è molto elevato, data l'assenza dell'uso di pecondizionatori. Sarebbe importante capire l'andamento dell'algoritmo in precisione mista su un sistema precondizionato.

- **Studio del problema della stagnazione**: I primi test hanno mostrato come l'algoritmo in precisione mista potesse essere soggetto al fenomeno della stagnazione. Lo studio di questo fenomeno potrebbe portare a capire come evitarlo o come capire quando avviene con certezza, avendo cosi un criterio migliore per evitarlo.

- **Ampliamento di PSBLAS**; Le routine e i kernel di calcolo realizzati non rappresentano la totalità delle funzionalità che questa libreria potrebbe offrire in precisione mista. Sicuramente sarebbe opportuno ampliare la gamma di formati di memorizzazione dei dati supportati per le computazioni in precisione mista.

Capitolo 7

Ringraziamenti

I ringraziamenti sono una di quelle tradizioni che personalmente non mi è mai piaciuta più di tanto. Non perché non abbia nessuno da ringraziare per il traguardo raggiunto, ma perché trovo un po' sminuente racchiudere 5 anni della mia vita in poche parole. Spero fortemente che le persone che mi hanno accompagnato in questo lungo periodo della mia vita capiscano la mia gratitudine per i gesti quotidiani e il tempo passato insieme piuttosto che nelle righe seguenti. Però dopo questa premessa è doveroso passare a dei ringraziamenti veri e propri.

Parto dalle banalità ringraziando i miei genitori che mi hanno permesso di studiare e di fare tutto quello che desiderassi in questi anni cercando sempre di evitarmi preoccupazioni esterne ai miei obiettivi. Per quanto possa sembrare scontato, avere l'opportunità di capire quello che si vuole fare nella vita avendo la sicurezza di poter sempre cambiare strada rende il percorso sicuramente più sereno. Intraprendere la strada universitaria mi è sembrato inizialmente doveroso, ma andando avanti la paura di aver sbagliato la mia scelta era sempre dietro l'angolo. Sapere di avere la possibilità di avere sempre una seconda scelta mi ha permesso di chiarire i miei dubbi senza alcuna costrizione esterna.

Subito dopo vorrei ringraziare il mio gruppo Scout, dove sono cresciuto e che mi ha reso la persona che sono oggi. Sono tanti i nomi che dovrei fare per ringraziare tutte le persone con cui ho percorso un tratto della mia strada in questo ambito, spero di non dimenticare nessuno. Per primo devo ringraziare Alessandrone, il mio migliore amico che mi ha accompagnato sia all'università che fuori, porto sicuro con cui scherzare e passare tempo in serenità, confermandomi sempre di più che può esistere un rapporto di amicizia genuino basato sulla voglia di divertirsi e di aiutarsi reciprocamente senza aspettarsi nulla in cambio. Francesco, con il quale ho condiviso bellissime esperienze e interessi, avendo sempre un punto di riferimento per essere spronato a continuare ad impegnarsi in quello che si fa. Davide e Pigi, che sono stati dei grandi esempi nella mia crescita e che mi hanno spronato a fare sempre del mio meglio in tutto quello che avessi fatto, facendomi anche capire come vivere gli anni universitari godendomi

appieno le esperienze vissute e non soltanto focalizzandomi sullo studio. Paolo, Matteo, Lorenzo, Mario e Simone, sempre disponibili ogni qual volta avessi bisogno di un aiuto o di un confronto. Luca, aver percorso un pezzo di strada importante insieme a me, sopportando i miei sfoghi e i miei consigli. Alessandra, che in ogni momento è pronto con un abbraccio e con un sorriso per superare i momenti difficili. Maddalena, compagna di innumerevoli chiacchierate e di confronti che raramente è possibile fare tra ragazzi. Roberto, che mi ha aiutato innumerevoli volte in questi anni, rimanendo un grande esempio per la sua instancabile voglia di fare. Tutti i miei capi e coloro con cui ho condiviso un'esperienza di servizio, perché i ricordi costruiti insieme in questi anni sono unici e mi rimarranno cari sempre.

Mi preme molto ringraziare Giulietta, una delle persone più forti che io abbia mai conosciuto, con cui ho condiviso tantissimo durante questi anni e che mi ha fatto capire molti aspetti della persona che sono oggi. Anche senza sentirsi o vedersi ogni giorno è possibile avere un rapporto di amicizia sincero che è raro da trovare al giorno d'oggi.

Poi vorrei ringraziare tutte le conoscenze fatte durante questi anni universitari, anche non essendo un tipo molto socievole ho trovato qualcuno con cui condividere il mio percorso. Giada, compagnia di mille serate e amica amica dal primo giorno di università. Simone, compagno di tutti i progetti e di tutto lo studio degli ultimi anni, è stato bello trovare qualcuno che fosse sempre presente per un chiarimento e che fosse una sicurezza su cui contare nonostante la mia vita frenetica e piena di impegni di tutti i giorni. Valerio e Ludovico, persone fantastiche e spero compagni di molti altri concerti.

Tutti i miei amici d'infanzia, Valerio, Luis, Francesco, Vincenzo, Roberto, Andrea e Marco, per aver condiviso momenti di spensieratezza insieme essendo in grado di ritornare sempre bambini nei momenti insieme.

In ambito lavorativo,a tutti i miei colleghi, in particolare a Mr. Garden (aka Croccopollo), per essere stato un supporto nella mia vita lavorativa e extra. Grazie per essere un esempio da seguire per avere una vita soddisfacente ma equilibrata, avendomi dimostrato che si può effettivamente fare la differenza nel nostro mondo essendo gentili e disponibili con il prossimo. Massimiliano, il mio datore di lavoro, per avermi dato la possibilità di studiare e lavorare contemporaneamente senza avermi mai fatto pesare la situazione, anzi mostrandomi come esistano realtà lavorative dove valori come amicizia, soddisfazione personale e passione per il proprio lavoro esistano ancora.

Concludo ringraziando tutte le persone importanti nella mia vita quotidiana per questo traguardo raggiunto, non tanto per averlo raggiunto, ma per come averlo raggiunto. Avendo degli amici e delle persone intorno a me con cui poter crescere, vivendo gli anni universitari come una formazione per la mia vita e non solo per una formazione solamente nozionistica. Grazie di tutto! È stata una cosa ben fatta!

Elenco delle figure

Elenco degli algoritmi

Elenco delle tabelle

Bibliografia

[1] R. W. Hamming, *Numerical Methods for Scientists and Engineers.* USA: McGraw-Hill, Inc., 1973.

[2] H. H. Goldstine, *A History of Numerical Analysis from the 16th through the 19th Century.* Springer Science & Business Media, 2012, vol. 2.

[3] Y. Saad, *Iterative methods for sparse linear systems.* SIAM, 2003.

[4] Y. Idomura, T. Ina, Y. Ali, and T. Imamura, "Acceleration of fusion plasma turbulence simulations using the mixed-precision communication-avoiding krylov method," *International Conference for High Performance Computing*, 2020.

[5] N. Palmer, "More reliable forecasts with less precise computations: a fast-track route to cloud-resolved weather and climate simulators?" *International Conference for High Performance Computing*, 2014.

[6] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," *arXiv preprint arXiv:1412.7024*, 2014.

[7] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," August 1994.

[8] D. Hough, "Applications of the proposed ieee 754 standard for floating-point arithetic," *Computer*, vol. 14, no. 03, pp. 70–74, 1981.

[9] J. Osorio, A. Armejach, E. Petit, G. Henry, and M. Casas, "A bf16 fma is all you need for dnn training," *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 3, pp. 1302–1314, 2022.

[10] N. J. Higham, *Accuracy and Stability of Numerical Algorithms.* USA: Society for Industrial and Applied Mathematics, 2002.

[11] N. J. Higham and T. Mary, "Mixed precision algorithms in numerical linear algebra," *Acta Numerica*, vol. 31, p. 347–414, 2022.

[12] M. Croci, M. Fasi, N. J. Higham, T. Mary, and M. Mikaitis, "Stochastic rounding: implementation, error analysis and applications," *Royal Society Open Science*, vol. 9, no. 3, p. 211631, 2022.

[13] S. Filippone and M. Colajanni, "Psblas: a library for parallel linear algebra computation on sparse matrices," *ACM Trans. Math. Softw.*, vol. 26, no. 4, p. 527–550, Dec. 2000. [Online]. Available: https://doi.org/10.1145/365723.365732

[14] S. Filippone, P. D'Ambra, and M. Colajanni, *Using a Parallel Library of Sparse Linear Algebra in a Fluid Dynamics Application Code on Linux Clusters*, pp. 441–448.

[15] E. Carson, T. Gergelits, and I. Yamazaki, "Mixed precision s-step lanczos and conjugate gradient algorithms," *Numerical Linear Algebra with Applications*, vol. 29, no. 3, p. e2425, 2022.

[16] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov, "Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy," *ACM Trans. Math. Softw.*, vol. 34, no. 4, Jul. 2008. [Online]. Available: https://doi.org/10.1145/1377596.1377597

[17] R. M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*. Springer Publishing Company, Incorporated, 2013.

[18] J. H. Wilkinson, "Error analysis of direct methods of matrix inversion," *Journal of the ACM (JACM)*, vol. 8, no. 3, pp. 281–330, 1961.

[19] M. Arioli, J. W. Demmel, and I. S. Duff, "Solving sparse linear systems with sparse backward error," *SIAM Journal on Matrix Analysis and Applications*, vol. 10, no. 2, pp. 165–190, 1989.

[20] M. Martone, S. Filippone, S. Tucci, P. Gepner, and M. Paprzycki, "Use of hybrid recursive csr/coo data structures in sparse matrix-vector multiplication," *Proceedings of the International Multiconference on Computer Science and Information Technology, IMCSIT 2010*, vol. 5, 01 2010.

[21] D. Barbieri, S. Filippone, V. Cardellini, and A. Fanfarillo, "Three storage formats for sparse matrices on gpgpus," 02 2015.

[22] P. Alonso, J. M. Badía, and A. M. Vidal, "Parallel algorithms for the solution of toeplitz systems of linear equations," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Waśniewski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 969–976.