Università degli Studi di Milano - Bicocca
Scuola di Scienza
Dipartimento di Informatica, Sistemi e Comunicazione
Corso di Scienze Informatiche

Metodo del Calcolo Scientifico - Assignment 1 - Decomposizione con metodo di Cholesky

Relazione di:
Pellegrini Damiano 886261
Sanvito Marco 886493

Anno accademico 2024-2025

Sommario

Il presente progetto si propone di confrontare soluzioni proprietarie e open-source per la risoluzione di sistemi lineari sparsi mediante la fattorizzazione di Cholesky. Verrà analizzato come, attraverso un'implementazione personalizzata, siano stati ottenuti risultati più accurati e veloci rispetto alle soluzioni proprietarie prese in esame, approfondendo le sfide incontrate durante il processo di sviluppo.

Indice

Introduzione

La fattorizzazione di Cholesky rappresenta un metodo efficiente per risolvere sistemi lineari quando la matrice dei coefficienti è simmetrica e definita positiva. Data una matrice A, la fattorizzazione di Cholesky produce una matrice triangolare inferiore L tale che:

$$A = L \cdot L^T \tag{1}$$

Dove:

- A è una matrice simmetrica definita positiva (il che implica che tutti i suoi autovalori sono strettamente positivi)
- L è una matrice triangolare inferiore
- L^T è la trasposta di L

Questa decomposizione risulta particolarmente utile nella risoluzione di sistemi lineari, nell'ottimizzazione numerica e nelle applicazioni statistiche.

Esiste anche una variante della fattorizzazione di Cholesky che introduce una matrice diagonale D, portando alla seguente espressione:

$$A = L \cdot D \cdot L^T \tag{2}$$

Dove:

- A è una matrice simmetrica definita positiva
- L è una matrice triangolare inferiore con diagonale unitaria (tutti gli elementi diagonali sono 1)
- D è una matrice diagonale contenente i pivot della fattorizzazione
- L^T è la trasposta di L

Questa variante è particolarmente utile quando si desidera evitare di estrarre la radice quadrata degli elementi diagonali di L, semplificando così i calcoli e migliorando la stabilità numerica.

Nelle applicazioni pratiche, molti problemi ingegneristici e scientifici generano matrici di grandi dimensioni in cui la maggior parte degli elementi sono zero (matrici sparse). Questo porta a un problema durante la fattorizzazione di Cholesky, cioè la gestione del fill-in. Il fill-in è un fenomeno che si verifica durante la fattorizzazione di Cholesky, in cui gli zeri nella matrice originale diventano non zero nella matrice triangolare inferiore L. Questo può portare a un significativo aumento del numero di elementi non-zero e, di conseguenza, a maggiori requisiti di memoria e tempo di calcolo per la fattorizzazione. Il fill-in è fortemente influenzato dall'ordinamento delle righe e delle colonne della matrice originale e quindi un ordinamento appropriato può ridurre drasticamente il numero di elementi non-zero che appaiono durante la fattorizzazione. Sono stati quindi sviluppati diversi algoritmi e tecniche per ridurre il fill-in. Tra questi, uno dei più utilizzati è l'AMD (Approximate Minimum Degree), che opera riordinando le righe e colonne in base al grado dei nodi nel grafo associato alla matrice. Altri approcci includono l'ordinamento Nested Dissection e le tecniche di Minimum Fill.

In questa relazione, analizzeremo in dettaglio l'implementazione della fattorizzazione di Cholesky in MATLAB, esplorando le sue caratteristiche, i punti di forza e le eventuali limitazioni. Successivamente, applicheremo questi concetti per sviluppare un'implementazione open source in C++ (e se possibile un'implementazione comparabile a quella di MATLAB), confrontando approfonditamente le due soluzioni in termini di efficienza computazionale, gestione della

memoria e scalabilità su diverse tipologie di matrici sparse. Questo confronto ci permetterà di verificare se ha senso avventurarsi in librerie open source per la fattorizzazione di Cholesky, oppure se è più opportuno pagare per una libreria commerciale come MATLAB.

1. Matlab

1.1. Introduzione a MATLAB

MATLAB (acronimo di «MATrix LABoratory») è un ambiente di calcolo numerico avanzato e un linguaggio di programmazione di alto livello sviluppato da MathWorks. Concepito originariamente negli anni "70 dal matematico Cleve Moler come interfaccia user-friendly per le librerie numeriche LINPACK ed EISPACK, MATLAB si è evoluto in un ecosistema completo per il calcolo scientifico e l"analisi numerica, diventando uno strumento fondamentale in svariati campi scientifici e ingegneristici.

Le caratteristiche distintive che hanno contribuito al suo successo includono:

- Gestione nativa ed efficiente delle operazioni matriciali
- Vasta collezione di funzioni matematiche ottimizzate per diverse applicazioni computazionali
- Strumenti avanzati di visualizzazione scientifica e generazione di grafici interattivi
- Ambiente di sviluppo integrato (IDE) con funzionalità complete di debug e profiling
- Architettura estensibile attraverso toolbox specializzati per domini specifici (elaborazione segnali, ottimizzazione, machine learning, statistica, ecc.)
- Interfacce per l'integrazione con altri linguaggi di programmazione come C, C++ e Python

La sintassi di MATLAB è intuitiva e orientata alla risoluzione di problemi matematici, rendendo relativamente semplice l'implementazione di algoritmi complessi con poche righe di codice.

1.2. Fattorizzazione di Cholesky in MATLAB

MATLAB implementa la fattorizzazione di Cholesky attraverso la funzione built-in chol, specificamente progettata per determinare la decomposizione di Cholesky di una matrice simmetrica definita positiva. La funzione chol offre diverse varianti sintattiche per adattarsi a esigenze computazionali specifiche, di cui la principale è:

$$R = \text{chol}(A) \tag{3}$$

Dove:

- A è una matrice simmetrica definita positiva (ovvero una matrice per cui tutti gli autovalori sono positivi)
- R è una matrice triangolare superiore tale che $A = R^T R$

Questa funzione realizza la fattorizzazione di Cholesky di una matrice simmetrica definita positiva A. Nel caso in cui A non sia simmetrica, MATLAB la tratta come se fosse simmetrica utilizzando solo la parte triangolare superiore. Se A non è definita positiva, MATLAB restituisce un errore.

Poiché il nostro caso si concentra su matrici sparse di dimensioni variabili, utilizzeremo la seguente sintassi:

$$[R, \text{flag}, p] = \text{chol}(A, \text{'vector'}) \tag{4}$$

Dove:

- A è una matrice simmetrica definita positiva
- R è una matrice triangolare superiore risultante dalla fattorizzazione
- flag è un indicatore che assume valore 0 se la matrice è definita positiva, 1 altrimenti
- p è un vettore di permutazione che ottimizza l'ordinamento delle righe e colonne di A

Per le matrici sparse di grandi dimensioni, un aspetto cruciale è la gestione del *fill-in* — fenomeno per cui elementi inizialmente nulli diventano non-zero durante la fattorizzazione, aumentando significativamente la complessità computazionale e l'utilizzo di memoria.

MATLAB affronta questo problema utilizzando l'algoritmo AMD (Approximate Minimum Degree), una strategia di riordinamento che analizza la struttura di sparsità della matrice e approssima una permutazione ottimale delle righe e colonne. Questa permutazione minimizza il fill-in atteso durante la fattorizzazione, riducendo notevolmente sia i requisiti di memoria che il tempo di calcolo.

La relazione matematica che esprime questa permutazione è:

$$R^T R = A(p, p) \tag{5}$$

dove p rappresenta il vettore di permutazione e $A(p, p)$ indica la matrice A con righe e colonne riordinate secondo p. Questo approccio produce una fattorizzazione matematicamente equivalente ma computazionalmente molto più efficiente, con un fattore sparso R che preserva maggiormente la struttura di sparsità originale.

Questa implementazione consente di ridurre la complessità algoritmica da $O(n^3)$ nel caso denso a $O(nc \cdot \text{nnz}(A))$ nel caso sparso ben ordinato, dove $\text{nnz}(A)$ rappresenta il numero di elementi non-zero. Questa complessità ottimizzata è ottenibile in casi favorevoli e dipende fortemente dall'efficacia del riordinamento e dalla struttura specifica della matrice.

1.2.1. Funzione chol in MATLAB

Per rendere più completo il confronto ed avere una base di partenza, abbiamo deciso di analizzare la struttura interna dell'implementazione di chol in MATLAB. Dall'analisi è emerso che questa funzione utilizza internamente il pacchetto CHOLMOD (CHOLesky MODification), un componente della libreria SuiteSparse T. Davis [1]. SuiteSparse rappresenta una raccolta completa e altamente ottimizzata di algoritmi per l'algebra lineare sparsa, sviluppata principalmente sotto la guida di Timothy A. Davis e disponibile come software open source su GitHub T. Davis [1].

Per verificare empiricamente l'utilizzo di SuiteSparse in MATLAB, abbiamo applicato, con piccole modifiche, uno script diagnostico che identifica le librerie matematiche sottostanti e le relative versioni tramite funzioni di debug non documentate. Lo script di partenza è disponibile presso undocumentedmatlab.com.

I risultati dello script confermano che MATLAB si affida effettivamente a molteplici componenti della libreria SuiteSparse per le operazioni su matrici sparse.

L'output generato dallo script ha evidenziato le seguenti librerie:

- Found: AMD version 2.2.0, May 31, 2007: approximate minimum degree ordering
- Found: colamd version 2.5, May 5, 2006: OK.

- Found: CHOLMOD version 1.7.0, Sept 20, 2008: : status: OK
- UMFPACK V5.4.0 (May 20, 2009), Control:
- SuiteSparseQR, version 1.1.0 (Sept 20, 2008)

1.2.2. Analisi di CHOLMOD

Approfondendo l'architettura del pacchetto CHOLMOD, abbiamo scoperto che questo si basa essenzialmente sulle librerie BLAS (Basic Linear Algebra Subprograms) e LAPACK (Linear Algebra PACKage) per eseguire operazioni di algebra lineare ad alte prestazioni. Ulteriori dettagli sul funzionamento di queste librerie sono consultabili presso netlib.org/blas [2] e netlib.org/lapack [3].

Queste librerie esistono in diverse implementazioni, ciascuna ottimizzata per architetture hardware specifiche. Le implementazioni più diffuse includono OpenBLAS (ottimizzata per molteplici architetture), Apple Accelerate (Implementazione per sistemi operativi MacOS) e Intel MKL (oneAPI Math Kernel Library), quest'ultima particolarmente performante su processori Intel.

Nel caso specifico di MATLAB su Windows e Linux, l'implementazione utilizzata è Intel MKL, che garantisce prestazioni ottimali su architetture x86 e x86-64.

È possibile verificare questa configurazione attraverso i seguenti comandi MATLAB:

version("-blas") version("-lapack")

I risultati ottenuti sono i seguenti (la versione potrebbe variare a seconda della release di MATLAB):

ans = "Intel(R) oneAPI Math Kernel Library Version 2024.1-Product Build 20240215
for Intel(R) 64 architecture applications (CNR branch AVX2)" ans = "Intel(R) oneAPI Math Kernel Library Version 2024.1-Product Build 20240215
for Intel(R) 64 architecture applications (CNR branch AVX2) supporting Linear Algebra PACKage (LAPACK 3.11.0)"

Dato che Intel MKL e Apple Accelerate sono librerie commerciali, abbiamo deciso di fare non solo un confronto tra MATLAB e Open Source, ma anche di analizzare le differenze tra le varie implementazioni di BLAS e LAPACK.

1.3. Implementazione in MATLAB

1.3.1. Parametri analizzati

Tempo di esecuzione:

- **loadTime:** tempo necessario per caricare la matrice dal file in formato MATLAB (MAT) (ms)
- **decompTime:** tempo per eseguire la fattorizzazione di Cholesky (ms)
- **solveTime:** tempo per risolvere un sistema lineare usando i fattori (ms)

Per il calcolo del tempo, abbiamo utilizzato il profiler di MATLAB attraverso profile, che misurano il tempo di esecuzione assieme ad altre informazioni di un blocco di codice.

Utilizzo di memoria:

- **loadMem:** memoria utilizzata per caricare la matrice (Bytes)
- **decompMem:** memoria utilizzata per la fattorizzazione (Bytes)

- **solveMem:** memoria utilizzata per trovare la soluzione (Bytes)

Per il calcolo della memoria, abbiamo utilizzato una funzionalità del profiler di MATLAB non documentata, che permette di calcolare la memoria utilizzata in una porzione di codice. Ovviamente, essendo non documentata, non è garantita la sua stabilità e correttezza, ma l'abbiamo ritenuto il metodo migliore per il calcolo la memoria utilizzata.

Per utilizzare questa funzionalità, abbiamo usato il comando profile -memory on che avvisa il profiler di tenere traccia della memoria utilizzata.

Accuratezza:

- **Errore Relativo:** errore relativo della soluzione calcolata rispetto alla soluzione attesa

1.3.2. Metodologia

Per ogni matrice, abbiamo eseguito i seguenti passaggi:

- Caricamento della matrice in memoria da file in formato MATLAB
- Esecuzione della fattorizzazione di Cholesky sulla matrice
- Risoluzione del sistema lineare Ax = b
- Calcolo dell'errore relativo tra la soluzione calcolata e quella attesa

Per risolvere il sistema lineare $Ac \cdot x = b$ dove il termine b è noto ed è scelto in modo che la soluzione esatta sia il vettore $x_e = [1, 1, 1, 1, 1, 1, ...]$, cioè $b = Ac \cdot x_e$.

I risultati vengono poi esportati in un file CSV per successiva analisi e confronto con altre implementazioni.

1.4. Documentazione di MATLAB

MATLAB, in quanto software commerciale, presenta una documentazione eccellente: ben strutturata, dettagliata e arricchita da numerosi esempi applicativi. L'ecosistema integrato di funzioni predefinite e toolbox specializzati consente agli utenti di implementare rapidamente soluzioni a problemi complessi con un minimo di codice, riducendo significativamente i tempi di sviluppo rispetto a soluzioni che richiederebbero l'integrazione manuale di diverse librerie.

Questo rappresenta un vantaggio sostanziale rispetto a molte alternative open source, dove la documentazione può risultare frammentaria, incompleta o non aggiornata. In ambito scientifico e ingegneristico, la rapidità di prototipazione e sviluppo offerta da MATLAB giustifica spesso l'investimento economico, soprattutto considerando i costi indiretti legati al tempo di sviluppo.

Tuttavia, è importante evidenziare alcune limitazioni. La documentazione ufficiale, pur essendo esaustiva nell'illustrare l'utilizzo delle funzioni, raramente rivela i dettagli implementativi sottostanti. Nel caso della fattorizzazione di Cholesky, ad esempio, la documentazione specifica parametri e comportamenti attesi, ma non approfondisce gli algoritmi utilizzati o le ottimizzazioni applicate. Questa opacità può risultare problematica durante il debugging di casi particolari o quando si necessita di comprendere le ragioni di determinati comportamenti computazionali.

1.5. Commenti

Un aspetto particolarmente interessante emerso dalla nostra analisi riguarda l'architettura interna di MATLAB: alcune funzionalità core, inclusa la fattorizzazione di Cholesky, si basano su librerie open source come SuiteSparse, ottimizzate tramite implementazioni non open-source,

ma con licenza libera previa citazione, utilizzabili di BLAS/LAPACK come Intel MKL. Questo solleva interrogativi legittimi sul valore aggiunto del software commerciale rispetto all'utilizzo diretto delle librerie open source sottostanti.
Il valore di MATLAB risiede quindi non tanto nell'esclusività degli algoritmi implementati, quanto nell'integrazione di questi in un ambiente coerente, ben documentato e ottimizzato per la produttività scientifica. La questione se questo valore aggiunto giustifichi il costo di licenza dipende fortemente dal contesto applicativo, dalle esigenze specifiche dell'utente e dai vincoli di tempo e risorse del progetto. Nell'ambito della ricerca, non a scopo di lucro, sviluppare uno strumento che faccia utilizzo di librerie commerciali/proprietarie e open-source potrebbe rivelarsi ragionevole.

2. C++

2.1. Introduzione a C++

C++ rappresenta una scelta ottimale per l'implementazione di algoritmi di algebra lineare grazie alle sue caratteristiche di efficienza, controllo di basso livello e supporto per la programmazione orientata agli oggetti.
Nel contesto della fattorizzazione di Cholesky, C++ ci permette di:

- Ottimizzare le operazioni su matrici sparse di grandi dimensioni
- Integrare librerie specializzate per l'algebra lineare
- Controllare precisamente l'allocazione della memoria
- Sfruttare costrutti template per implementazioni generiche

2.2. Tool Chain C++

Per lo sviluppo del nostro progetto abbiamo utilizzato diversi strumenti in base all'ambiente operativo:

2.2.1. Ambiente Windows

In ambiente Windows, la nostra implementazione si è basata su:

- **Microsoft Visual C++ (MSVC)**: Il compilatore ufficiale di Microsoft che offre ottimizzazioni specifiche per architetture Intel/AMD.
- **Intel Fortran Compiler (IFX)**: Compilatore Intel per Fortran che abbiamo utilizzato per compilare alcune componenti di SuiteSparse.
- **CMake**: Sistema cross-platform per la gestione del processo di build, permettendo di generare progetti Visual Studio nativi mantenendo la portabilità del codice. La configurazione CMake ha facilitato l'integrazione delle diverse librerie utilizzate nel progetto.

2.2.2. Ambiente Linux

Per garantire la portabilità del codice e per effettuare test comparativi, abbiamo anche utilizzato:

- **GNU Compiler Collection (GCC)**: Compilatore C++ standard in ambienti Linux, utilizzato nella versione 11.3 con pieno supporto per C++17.

- **GNU Fortran (GFortran)**: Necessario per compilare alcune componenti delle librerie BLAS e LAPACK utilizzate dal progetto.
- **CMake**: Sistema cross-platform per la gestione del processo di build, permettendo di generare progetti Visual Studio nativi mantenendo la portabilità del codice. La configurazione CMake ha facilitato l'integrazione delle diverse librerie utilizzate nel progetto.

2.2.3. Ambiente MacOS

Per testare un ambiente più professionale ed utilizzare una libreria proprietaria differente, ci siamo forniti di:

- **Apple clang (clang)**: Compilatore C++ standard in ambienti Apple MacOS, utilizzato nella versione 17.0.0 con pieno supporto per C++17.
- **GNU Fortran (GFortran)**: Necessario per compilare alcune componenti delle librerie BLAS e LAPACK utilizzate dal progetto.
- **CMake**: Sistema cross-platform per la gestione del processo di build, permettendo di generare progetti Visual Studio nativi mantenendo la portabilità del codice. La configurazione CMake ha facilitato l'integrazione delle diverse librerie utilizzate nel progetto.

2.3. Librerie C++ per la fattorizzazione di Cholesky

2.3.1. SuiteSparse

SuiteSparse fornisce algoritmi altamente ottimizzati per matrici sparse. In particolare, abbiamo integrato CHOLMOD (con le sue dipendenze), la componente specializzata per la fattorizzazione di Cholesky di matrici sparse simmetriche definite positive, con licenza GNU LGPL.

CHOLMOD offre prestazioni superiori rispetto altre implementazioni per matrici di grandi dimensioni grazie a:

- Algoritmi di ordinamento avanzati (AMD, COLAMD, METIS) che riducono il fill-in durante la fattorizzazione
- Decomposizione supernodale che sfrutta operazioni BLAS di livello 3
- Supporto per calcoli multithreaded che sfruttano processori multi-core
- Gestione ottimizzata della memoria che riduce il sovraccarico per matrici molto sparse

2.3.2. Eigen

Eigen è una libreria C++ header-only di algebra lineare ad alte prestazioni, completamente sviluppata in template per massimizzare l'ottimizzazione in fase di compilazione, con licenza MPL2.

Una caratteristica distintiva di Eigen è la sua architettura estensibile che permette l'integrazione con diverse librerie esterne specializzate. Nel nostro progetto, abbiamo scelto di utilizzare l'interfaccia con CHOLMOD di SuiteSparse:

- **Interfaccia CHOLMOD**: Abbiamo sfruttato principalmente il modulo `CholmodSupport` di Eigen che permette di utilizzare gli algoritmi avanzati di CHOLMOD mantenendo la sintassi familiare di Eigen.

- **Alternative considerate**: Sarebbe stato possibile utilizzare l'implementazione nativa di Eigen (`SimplicialLLT`) con o senza supporto BLAS/LAPACK, che risulta adeguata per matrici di dimensioni moderate, ma dato che volevamo basarci sull'implementazione di MATLAB abbiamo optato per l'altra strada.
- **Alternative proprietarie**: Eigen supporta anche interfacce verso librerie proprietarie come Pardiso di Intel oneAPI e Accelerate di Apple, che offrono implementazioni altamente ottimizzate ma non open-source.

Per la fattorizzazione di Cholesky, il nostro approccio primario è stato:
`CholmodSupport::CholmodDecomposition` che delega il calcolo effettivo a CHOLMOD, beneficiando degli algoritmi di ordinamento avanzati e dell'ottimizzazione per sistemi multi-core e scegliendo in automatico che algoritmo di ordinamento usare e che tipo di fattorizzazione (supermodal vs simplicial).
La flessibilità di Eigen ci ha permesso di integrare efficacemente la potenza di CHOLMOD mantenendo un'interfaccia coerente e familiare nel codice principale, senza compromettere l'approccio open-source del progetto.

2.3.3. Fast Matrix Market

Per la lettura delle matrici sparse dal formato Matrix Market (MTX), abbiamo integrato la libreria Fast Matrix Market, con licenza BSD-2. Questa libreria ha consentito di importare efficientemente dataset di test di grandi dimensioni.
Fast Matrix Market si distingue per:

- Lettura parallelizzata che sfrutta tutti i core disponibili
- Parsing efficiente che riduce significativamente i tempi di caricamento
- Integrazione diretta con Eigen senza necessità di conversioni intermedie
- Supporto per diverse precisioni numeriche (float, double, complex)

2.4. Librerie BLAS e LAPACK

Le librerie BLAS (Basic Linear Algebra Subprograms) e LAPACK (Linear Algebra PACKage) rappresentano fondamenti essenziali per l'algebra lineare computazionale. Queste librerie standardizzate forniscono implementazioni ottimizzate di operazioni matriciali e vettoriali di base che costituiscono i blocchi fondamentali per algoritmi più complessi, inclusa la fattorizzazione di Cholesky.

2.4.1. Intel MKL

Intel Math Kernel Library (MKL) rappresenta l'implementazione commerciale di riferimento per BLAS e LAPACK, sviluppata e ottimizzata specificamente per processori Intel. Questa libreria offre prestazioni eccezionali su architetture x86 e x86-64 grazie a:

- Ottimizzazioni a livello di microarchitettura che sfruttano set di istruzioni specifici (AVX, AVX2, AVX-512)
- Parallelizzazione automatica che utilizza efficacemente processori multi-core
- Gestione intelligente della cache e della memoria per massimizzare il throughput
- Routine specializzate per matrici sparse che riducono significativamente il tempo di calcolo

2.4.2. OpenBLAS

OpenBLAS rappresenta l'alternativa open source più matura a Intel MKL, offrendo prestazioni competitive su diverse architetture hardware. Questa libreria deriva dal progetto GotoBLAS2 e si distingue per:

- Ottimizzazioni specifiche per diverse architetture (Intel, AMD, ARM, POWER)
- Supporto per parallelismo multi-thread attraverso implementazione OpenMP
- Compatibilità con l'interfaccia CBLAS standard
- Prestazioni scalabili fino a 256 core

2.4.3. Apple Accelerate

Accelerate è il framework di calcolo numerico sviluppato da Apple e integrato nativamente nei sistemi operativi macOS e iOS. Include implementazioni ottimizzate di BLAS e LAPACK specificamente progettate per l'hardware Apple, inclusi i processori M-Series basati su architettura Apple Silicon (ARM).

Caratteristiche distintive di Accelerate includono:

- Ottimizzazioni specifiche per chip Apple Silicon
- Integrazione profonda con l'ecosistema di librerie Apple e supporto per tecnologie come Grand Central Dispatch
- Supporto per calcoli vettoriali SIMD attraverso il framework vDSP
- Bilanciamento automatico tra prestazioni ed efficienza energetica

2.5. Implementazione in C++

2.5.1. Considerazioni generali

Avendo accesso al codice sorgente volendo è possibile adattare il codice alla risoluzione di un problema specifico, nel nostro caso abbiamo mantenuto un implementazione piuttosto generica, date le diverse matrici da trattare.

2.5.2. Parametri analizzati

A differenza di MATLAB, andiamo a ottenere anche il tipo di BLAS e il numero di thread utilizzati per l'esecuzione, nello specifico abbiamo misurato:

Tempo di esecuzione:

- **loadTime:** tempo necessario per caricare la matrice dal file in formato Matrix Market (MTX) (ms)
- **decompTime:** tempo per eseguire la fattorizzazione di Cholesky (ms)
- **solveTime:** tempo per risolvere un sistema lineare usando i fattori (ms)

Per misurare il tempo con precisione, abbiamo utilizzato le funzionalità della libreria standard C++:

```
auto start = std::chrono::high_resolution_clock::now();
// Operazione da misurare
auto end = std::chrono::high_resolution_clock::now();
auto duration =
    std::chrono::duration_cast<std::chrono::milliseconds>(
        end - start
    ).count();
```

Utilizzo di memoria:

- loadMem: memoria utilizzata per caricare la matrice (Bytes)
- decompMem: memoria utilizzata per la fattorizzazione (Bytes)
- solveMem: memoria utilizzata per trovare la soluzione (Bytes)

Il calcolo della memoria per le operazioni di caricamento della matrice è stato implementato manualmente, considerando:

```
size_t valuesSize = A.nonZeros() * sizeof(double);
size_t innerIndicesSize = A.nonZeros() * sizeof(int64_t);
size_t outerIndicesSize = (A.outerSize() + 1) * sizeof(int64_t);
auto loadMem = valuesSize + innerIndicesSize + outerIndicesSize;
```

Invece per il calcolo della memoria per le operazioni di fattorizzazione e risoluzione, abbiamo modificato parte del codice di CHOLMOD, aggiungendo un contatore per la memoria allocata. Questo contatore viene resettato prima di ogni operazione e aggiornato durante l'allocazione della memoria.

```
solver.cholmod().memory_allocated = 0;  // Reset contatore
// Operazione da misurare
auto operationMem = solver.cholmod().memory_allocated;
```

Accuratezza:

- Errore Relativo: errore relativo della soluzione calcolata rispetto alla soluzione attesa

Per ridurre l'errore nel calcolo dell'errore evitando il calcolo una delle due radici, abbiamo ricavato la seguente formula:

$$\sqrt{\frac{\|x - x_e\|^2}{\|x\|^2}} = \frac{\|x - x_e\|_2}{\|x\|_2} \tag{6}$$

Dove dato

$$\frac{\|x - x_e\|_2}{\|x\|_2} = \frac{\sqrt{(x - x_e) \cdot (x - x_e)}}{\sqrt{x \cdot x}} \tag{7}$$

con $\cdot$ prodotto scalare tra vettori, ho che

$$\frac{\|x - x_e\|^2}{\|x\|^2} = \frac{(x - x_e) \cdot (x - x_e)}{x \cdot x} \tag{8}$$

ovvero le somme delle componenti del vettore al quadrato.

2.5.3. Implementazione della fattorizzazione di Cholesky

In linea con l'approccio MATLAB, abbiamo implementato la fattorizzazione di Cholesky utilizzando la libreria CHOLMOD attraverso l'interfaccia fornita da Eigen:

```
Eigen::CholmodDecomposition<SparseMatrix> solver;
solver.compute(A); // Fattorizzazione della matrice A
xe = solver.solve(b); // Risoluzione del sistema Ax = b
```

Questa interfaccia consente di sfruttare le ottimizzazioni avanzate di CHOLMOD, inclusi gli algoritmi di ordinamento e la decomposizione supernodale, mantenendo al contempo la sintassi familiare di Eigen. La libreria CHOLMOD gestisce automaticamente la scelta dell'algoritmo di

ordinamento e del tipo di composizione più adatto in base alla struttura della matrice, ottimizzando così le prestazioni della fattorizzazione.

2.6. Documentazione e Integrazione Librerie C++

Per integrare efficacemente le librerie C++ nel nostro progetto, abbiamo dovuto affrontare diverse sfide legate alla documentazione e alla configurazione.

2.6.1. Eigen

La documentazione di Eigen rappresenta un eccellente esempio di riferimento tecnico per progetti open-source:

Completezza: Tutorial dettagliati, guida per le classi e documentazione delle API generata con Doxygen. Esempi: Numerosi esempi di codice che coprono tutti i moduli principali. Integrazione: Essendo header-only, l'integrazione richiede solo l'inclusione dei file header senza necessità di linking. Moduli esterni: La documentazione sul modulo CholmodSupport è più limitata rispetto ai moduli principali, richiedendo talvolta la consultazione del codice sorgente. L'integrazione di Eigen nel progetto è stata generalmente agevole grazie alla semplicità del modello header-only e ai chiari esempi disponibili nella documentazione ufficiale.

2.6.2. SuiteSparse

La documentazione di SuiteSparse, e in particolare di CHOLMOD, presenta caratteristiche distintive:

Documentazione scientifica: Articoli accademici dettagliati che descrivono gli algoritmi implementati. Documentazione tecnica: File README e documentazione interna al codice che descrivono l'API C. Limitazioni: Minore enfasi sugli esempi di integrazione in progetti C++ moderni. Build system: Documentazione limitata sull'integrazione con sistemi di build. Nonostante l'eccellente documentazione degli algoritmi sottostanti, l'integrazione di SuiteSparse ha richiesto maggiore impegno, specialmente per configurare correttamente le dipendenze tra i vari componenti. T. Davis [1]

2.6.3. Fast Matrix Market

La libreria Fast Matrix Market offre una documentazione concisa ma efficace:

GitHub README: Documenta chiaramente l'API principale e i casi d'uso comuni. Esempi: Include esempi di integrazione con Eigen che hanno facilitato significativamente l'adozione. Integrazione CMake: Fornisce configurazioni CMake moderne con supporto per find_package. L'integrazione di Fast Matrix Market è stata notevolmente semplice grazie alla documentazione mirata e agli esempi pratici, permettendo una rapida implementazione della lettura di matrici sparse in formato MTX.

2.6.4. Librerie BLAS e LAPACK

Le sfide più significative nel progetto sono emerse dall'integrazione delle implementazioni BLAS e LAPACK:

Documentazione frammentata: Ogni implementazione (Intel MKL, OpenBLAS, Accelerate) presenta una propria documentazione con convenzioni e approcci di configurazione diversi, ma questo non ha rappresentato il problema principale.

Difficoltà CMake: Abbiamo riscontrato notevoli difficoltà nell'integrazione attraverso CMake:
Mancanza di moduli CMake standardizzati per il rilevamento delle diverse implementazioni, rendendo inefficaci i moduli standard come FindBLAS e FindLAPACK. Necessità di linkare manualmente le librerie specificando esattamente i percorsi e i componenti richiesti, invece di poter utilizzare i meccanismi automatizzati di CMake. Configurazioni diverse richieste per Windows (MKL/MSVC) e Linux (OpenBLAS/GCC). Conflitti di simboli: In alcuni casi, quali l'utilizzo dell'interfaccia standard ILP64 LAPACK 3.11.0 dell'implementazione di Apple Accelerate ha causato conflitti di simboli difficili da risolvere.

2.6.5. Conclusioni

Dall'esperienza di integrazione delle diverse librerie, abbiamo tratto importanti conclusioni:
Le librerie con documentazione orientata agli esempi (Eigen, Fast Matrix Market) hanno richiesto tempi di integrazione significativamente minori.
Le dipendenze transitive non documentate tra librerie C/C++ rappresentano una sfida significativa per l'integrazione tramite CMake.
L'approccio più efficace è risultato essere lo sviluppo di configurazioni CMake modulari che isolano le complessità di ogni libreria.
La documentazione delle librerie di algebra lineare spesso privilegia la descrizione matematica degli algoritmi a scapito dei dettagli di integrazione tecnica.
Queste sfide di integrazione, sebbene impegnative, hanno permesso di sviluppare un sistema robusto e flessibile che può adattarsi a diverse implementazioni BLAS/LAPACK mantenendo un'interfaccia coerente attraverso Eigen.

2.7. Commenti

Il nostro progetto ha dimostrato con successo l'integrazione di librerie specializzate per l'algebra lineare in un ecosistema C++ moderno. Utilizzando Eigen come interfaccia ad alto livello e SuiteSparse (in particolare CHOLMOD) come motore di calcolo per la fattorizzazione di Cholesky, siamo riusciti a costruire un sistema flessibile e performante, capace di gestire matrici sparse di grandi dimensioni.
Un aspetto distintivo della nostra implementazione è stata la capacità di sfruttare diverse implementazioni di BLAS e LAPACK (Intel MKL, OpenBLAS, Accelerate), permettendoci di confrontare direttamente le prestazioni di soluzioni commerciali e open-source. Questa flessibilità ci ha consentito di simulare efficacemente il comportamento di MATLAB, che utilizza internamente CHOLMOD con implementazioni BLAS ottimizzate.
L'obiettivo principale dell'esperimento era verificare se un'alternativa completamente open-source potesse offrire prestazioni paragonabili alla soluzione commerciale di MATLAB.
Per quanto l'ideazione di una soluzione artigianale possa sembrare complicato, lo sviluppo del codice in se è stata forse la parte meno impegnativa. Maggiori difficoltà invece, le abbiamo incontrate nell'integrazione delle diverse librerie, specialmente quelle legate a BLAS e LAPACK, hanno rivelato la necessità di migliorare gli strumenti di build e la documentazione per questi componenti fondamentali dell'ecosistema di calcolo scientifico.
Nonostante queste sfide, il nostro progetto dimostra che è possibile costruire una piattaforma di calcolo numerico avanzata basata interamente su tecnologie open-source, offrendo un'alternativa

valida a soluzioni commerciali come MATLAB per applicazioni che richiedono la fattorizzazione di Cholesky su matrici sparse di grandi dimensioni.

3. Risultati

3.1. Specifiche del sistema

Specifiche del sistema per Windows:

- **Processore**: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz 2.81 GHz
- **Architettura**: Sistema operativo a 64 bit, processore basato su x64
- **RAM installata**: 16 GB
- **Archiviazione**: 238 GB SSD e 1 TB HDD Esterno
- **Scheda grafica**: NVIDIA GeForce GTX 1060 with Max-Q Design (6 GB)
- **Memoria Virtuale**: 7680 MB su SSD e 32768 MB su HDD Esterno
- **Sistema operativo Windows**: Windows 10 Home

Specifiche del sistema per Linux:

Processore, Architettura, RAM, Archiviazione e Scheda Grafica sono gli stessi del sistema Windows.

- **Memoria Virtuale**: 40448 MB su HDD Esterno
- **Sistema operativo Linux**: WSL 2 con Ubuntu 25.04

La memoria virtuale era possibile averla solo su un solo disco, quindi è stata scelta (per mancanza di spazio) quella dell'HDD esterno.

Specifiche del sistema per MacOS:

- **Processore**: Apple M1 Pro
- **Architettura**: ARM64 (10 core)
- **RAM installata**: 16 GB
- **Archiviazione**: 1 TB SSD
- **Scheda grafica**: Apple M1 Pro GPU (16 core)
- **Memoria Virtuale**: 56 GB su SSD
- **Sistema operativo**: MacOS Sonoma 15.4

3.2. Matrici analizzate

Ordinate in base al numero di elementi non zero, le matrici analizzate sono:

Nome Matrice	Righe & Colonne	Valori non zero
ex15	6867	98671
shallow_water1	81920	327680
cfd1	70656	1828364
cfd2	123440	3087898
parabolic_fem	525825	3674625
apache2	715176	4817870
G3_circuit	1585478	7660826
StocF-1465	1465137	21005389
Flan_1565	1564794	117406044

Tabella 1: Matrici analizzate

Tutte queste matrici sono sparse, simmetriche e positive definite. Sono state scaricate dal repository https://sparse.tamu.edu/.

3.3. Considerazioni Iniziali

Dato che sia MATLAB che C++ utilizzano la stessa libreria CHOLMOD, ci si aspetta che i risultati siano simili. Tuttavia, potrebbero emergere delle differenze, soprattutto a causa della versione obsoleta di MATLAB, meno aggiornata rispetto a quella di C++. Inoltre, l'uso differente delle librerie BLAS potrebbe influenzare ulteriormente i risultati.

3.4. Risultati MATLAB

Durante l'esecuzione, due matrici hanno causato un errore interno della libreria CHOLMOD in Windows, mentre in Linux hanno provocato la terminazione forzata del processo. Di conseguenza, non sono state incluse nei risultati. Le matrici problematiche sono:

- Flan_1565
- StocF-1465

3.4.1. Memoria

Matrix	Uso Memoria Windows (MB)			Uso Memoria Linux (MB)		
	load	decomp	solve	load	decomp	solve
ex15	0	3.48	0	2.34	3.85	0.34
shallow_water1	5.02	36.05	0	6.16	37.51	0.91
cfd1	27.92	552.88	0	30.57	555.84	0.83
cfd2	47.18	1129.79	0	47.48	1128.11	1.23
parabolic_fem	64.24	560.48	4.02	65.15	560.48	4.3
apache2	79.13	2722.26	5.47	80.08	175921860.43	5.66
G3_circuit	129.25	2915.38	12.13	130.49	175921860.43	15.27

Tabella 2: Confronto utilizzo memoria tra sistemi operativi su MATLAB

Dalla tabella emerge chiaramente che il profiler di memoria di MATLAB potrebbe non essere completamente affidabile, poiché riporta valori pari a zero per il caricamento della matrice. Questo fenomeno è probabilmente dovuto al coinvolgimento della libreria esterna CHOLMOD, che MATLAB non traccia direttamente attraverso il suo profiler.
Di conseguenza, la memoria riportata sembra riflettere solo quella utilizzata da MATLAB durante la chiamata alla libreria esterna, senza considerare la memoria effettivamente impiegata per l'operazione in sé.

3.4.2. Tempi

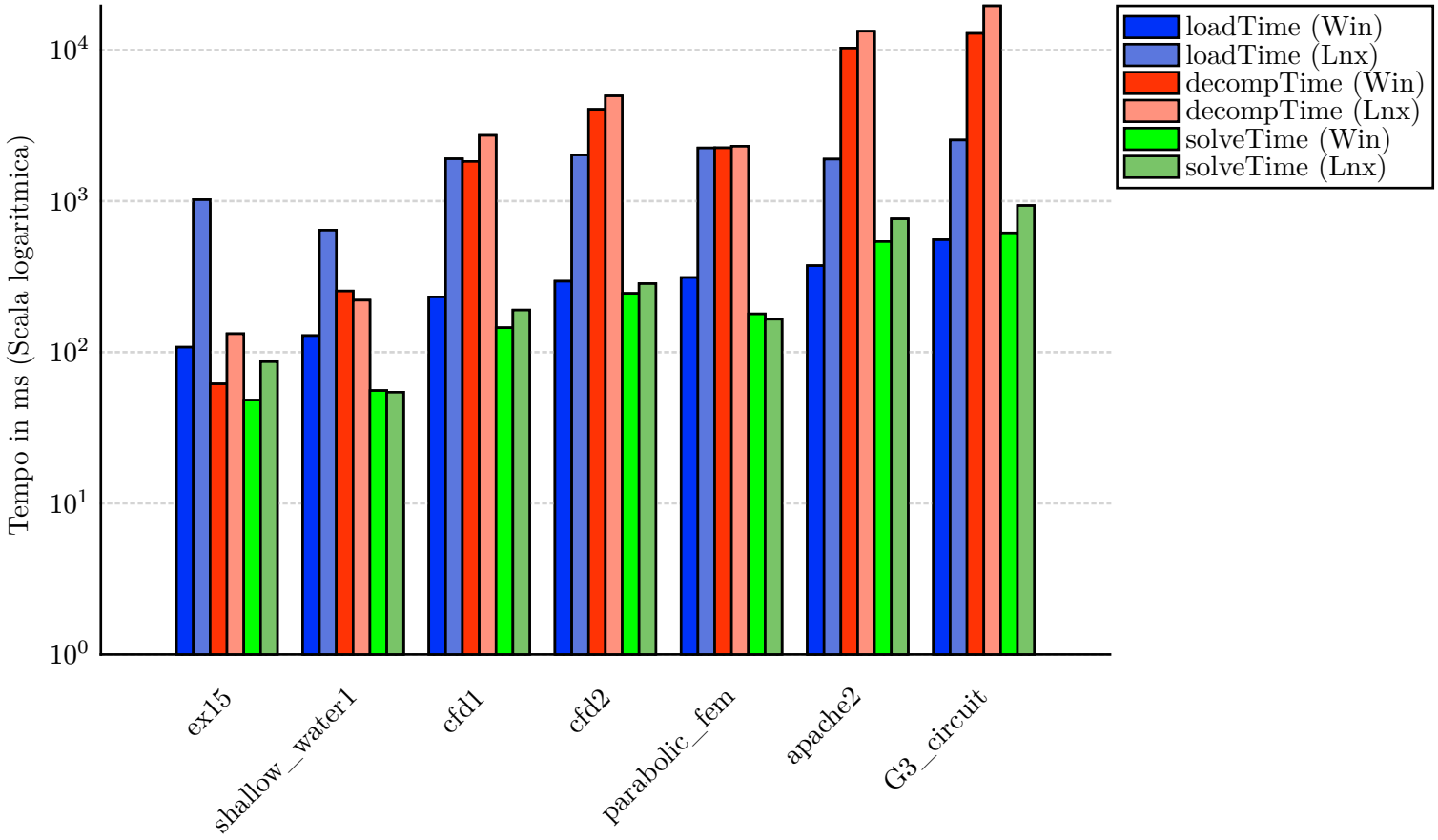

Figura 1: Confronto tempi tra sistemi operativi su MATLAB

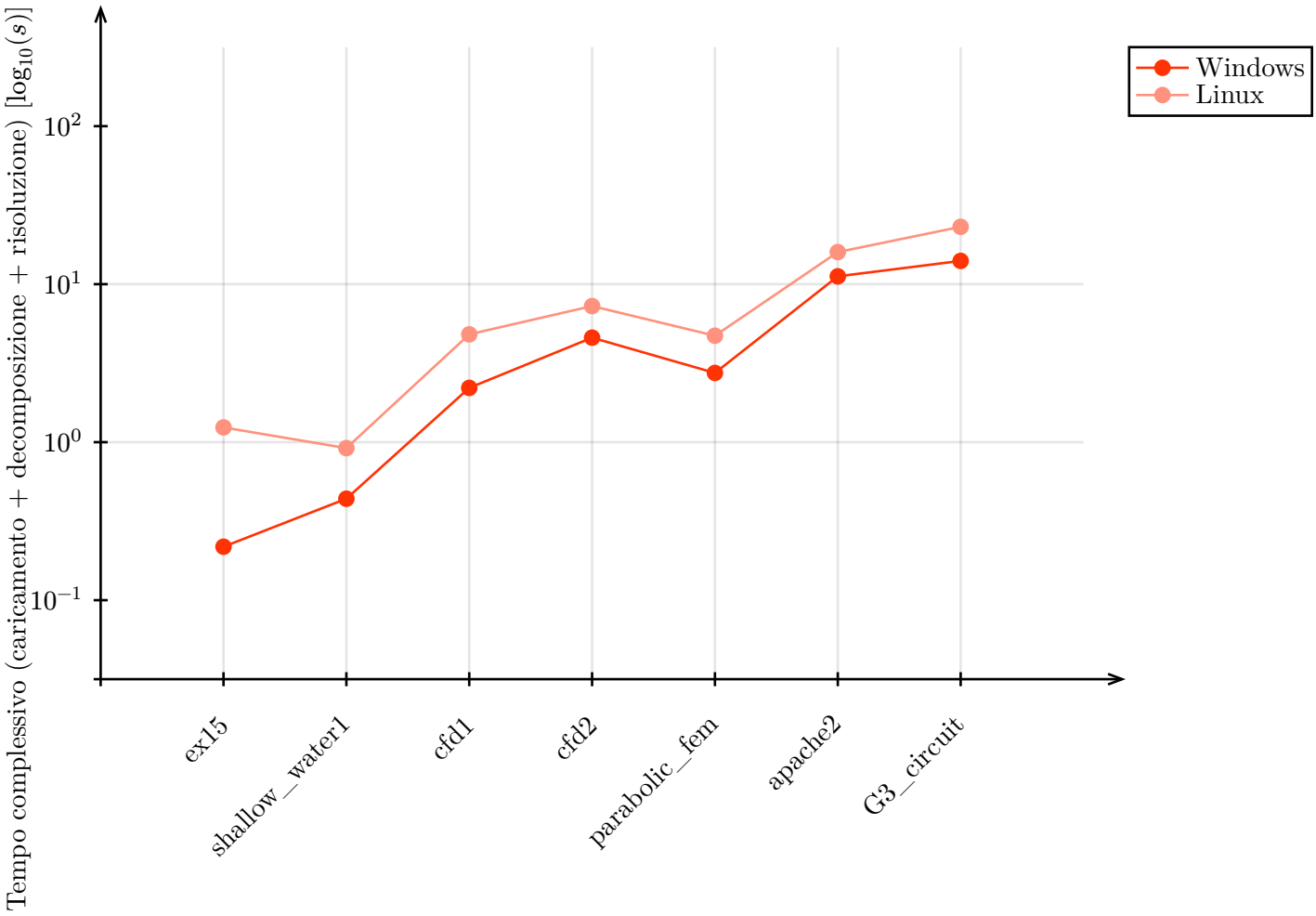

Figura 2: Confronto tempo complessivo tra sistemi operativi su MATLAB

Osservando prima il column chart e poi il line chart, notiamo che i tempi di decomposizione e risoluzione non sembrano presentare grandi differenze. Tuttavia, il tempo di caricamento risulta significativamente più alto in Linux rispetto a Windows. Questo è probabilmente dovuto al fatto che, per Linux, utilizziamo WSL2, il quale non ha accesso diretto all'hardware e deve quindi operare attraverso un layer di virtualizzazione.

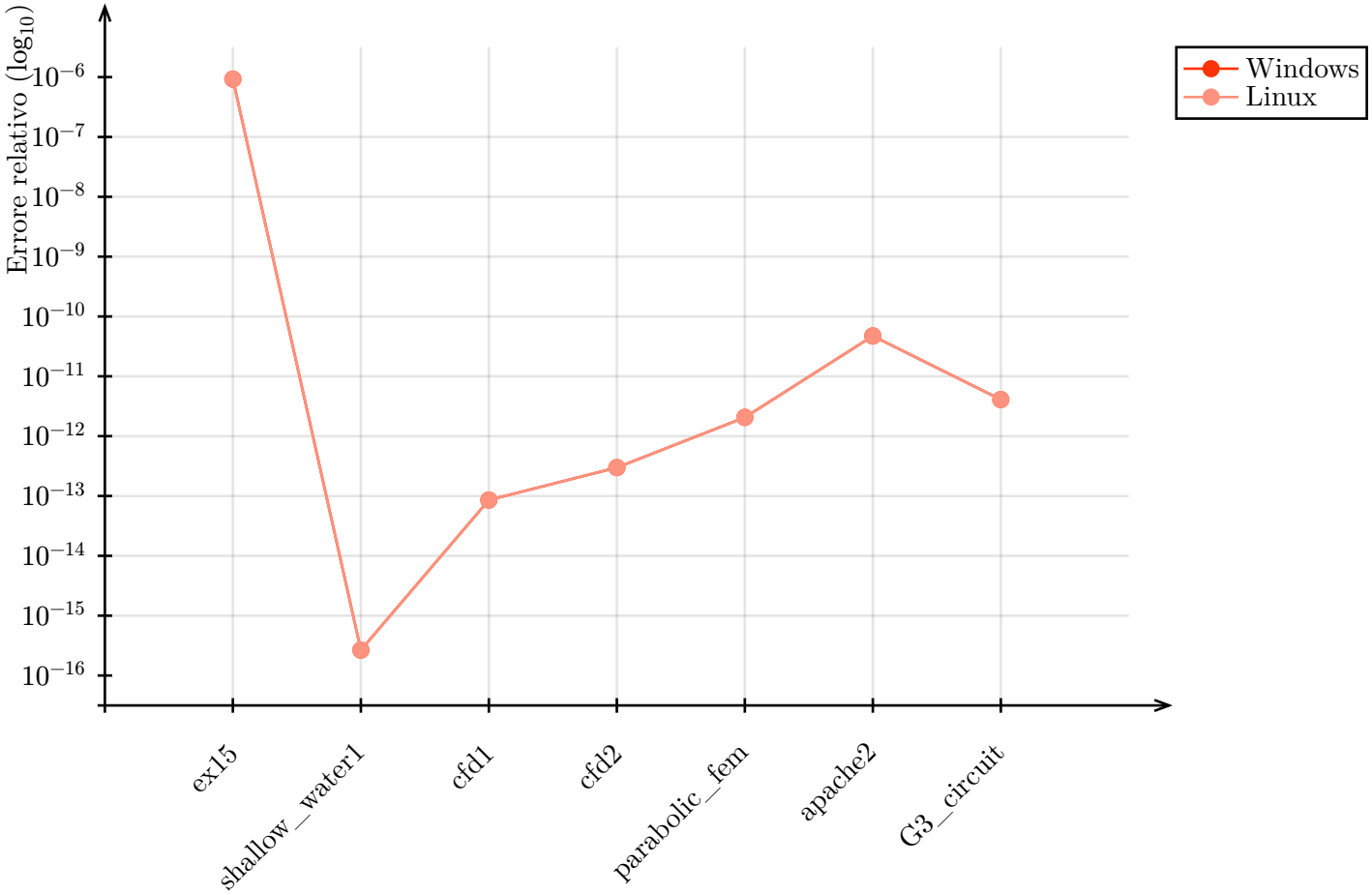

Figura 3: Confronto errore relativo tra sistemi operativi su MATLAB

Osservando l'errore, notiamo che è identico su entrambe le piattaforme, quindi in MATLAB non si riscontra alcuna differenza tra i due sistemi operativi.

3.5. Risultati C++

A differenza di MATLAB, C++ è riuscito a completare tutte le matrici.

3.5.1. Memoria

La memoria è identica tra i tre sistemi operativi, quindi non è necessario ripeterla per ciascuno. Questo dipende dal modo in cui viene ottenuta, ovvero dalle allocazioni interne di CHOLMOD, e ha senso, dato che la memoria è determinata principalmente dall'architettura del sistema piuttosto che dal sistema operativo.

Nome Matrice	**Mem Load (MB)**	**Mem Decomp (MB)**	**Picco Mem Decomp (MB)**	**Mem Risoluzione (MB)**	**Picco Mem Risoluzione (MB)**
ex15	1.56	9.6	4.21	0.11	0.11
shallow_water1	5.63	68.41	46.23	1.25	1.25
cfd1	28.44	331.16	221.42	1.09	1.09
cfd2	48.06	581.7	396.06	1.89	1.89
parabolic_fem	60.08	691.59	471.25	8.03	8.03
apache2	78.97	1734.88	1407.36	10.93	10.93
G3_circuit	128.99	1777.83	1224.56	24.2	24.2
StocF-1465	331.69	11411.76	10103.63	22.41	22.41
Flan_1565	1803.41	21283.12	14451.67	23.92	23.92

Tabella 3: Confronto utilizzo memoria tra sistemi operativi su C++

Analizzando brevemente i dati si vede una grande differenza rispetto a MATLAB, questo è perchè in C++ siamo molto piu precisi dato che andiamo a ottenere la memoria direttamente da CHOLMOD, mentre in MATLAB non possiamo sapere esattamente quanta memoria viene utilizzata da CHOLMOD.

3.5.2. Tempi

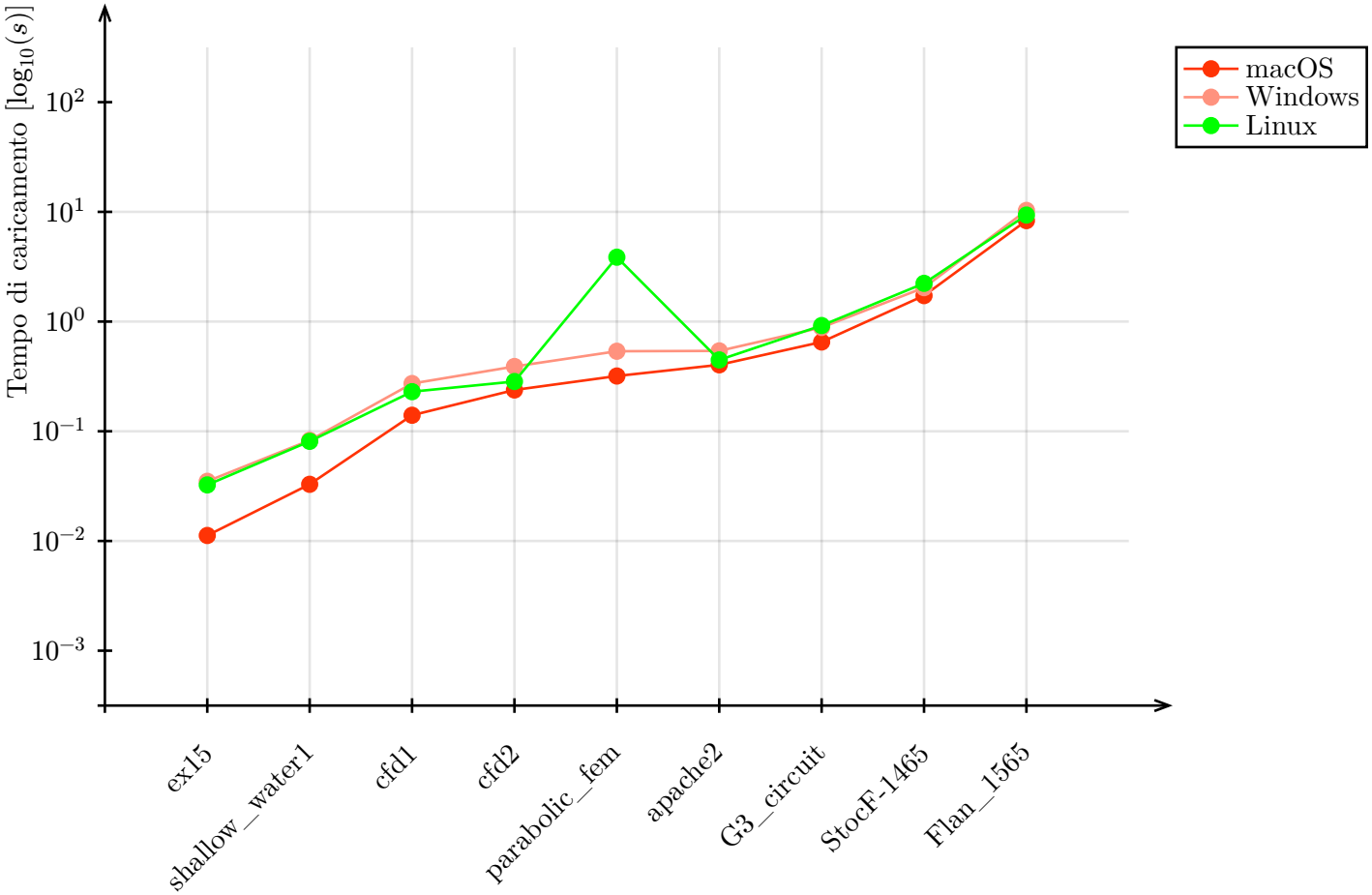

Figura 4: Confronto tempo di caricamento tra sistemi operativi su C++

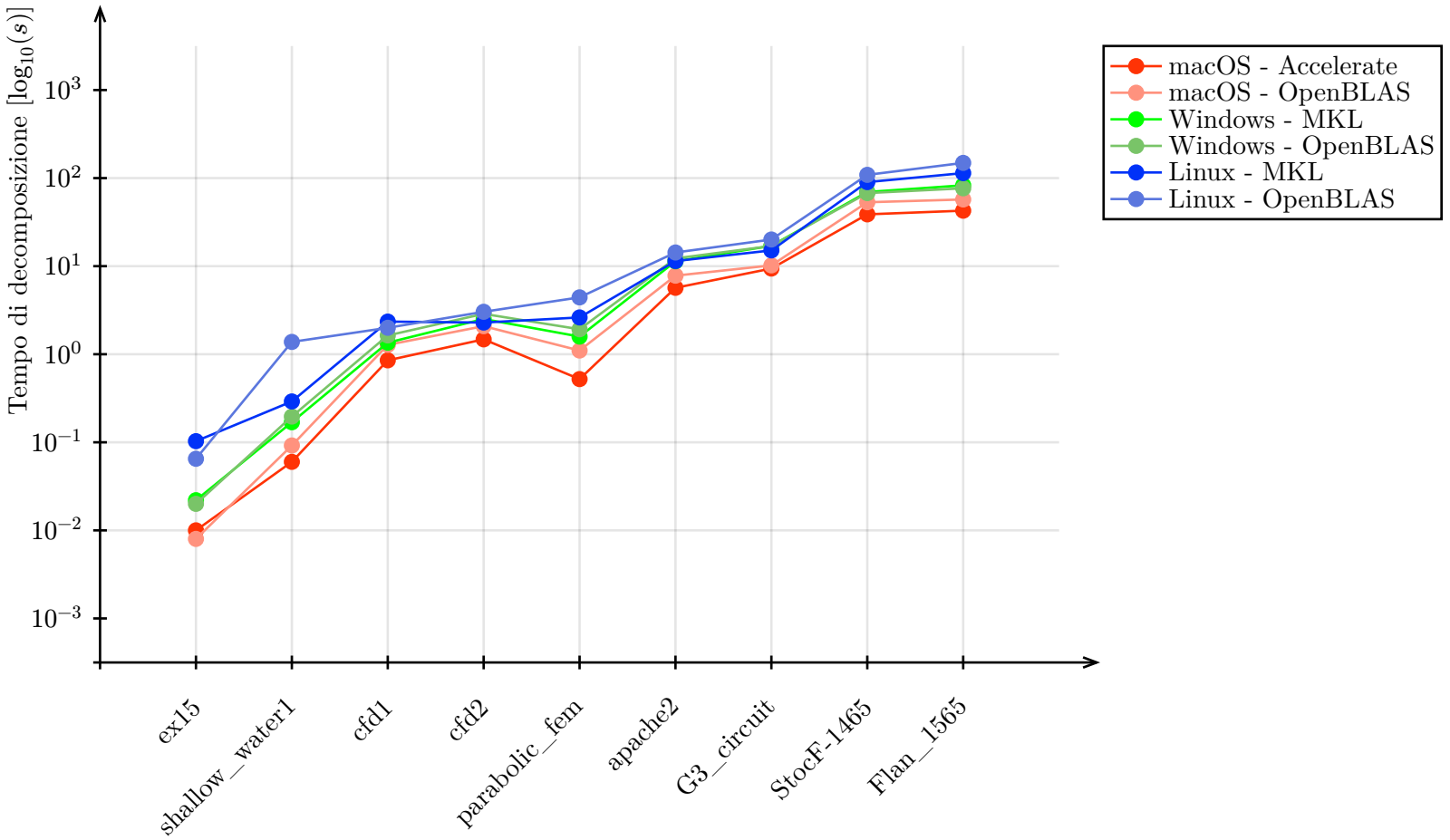

Figura 5: Confronto tempo di decomposizione tra sistemi operativi su C++

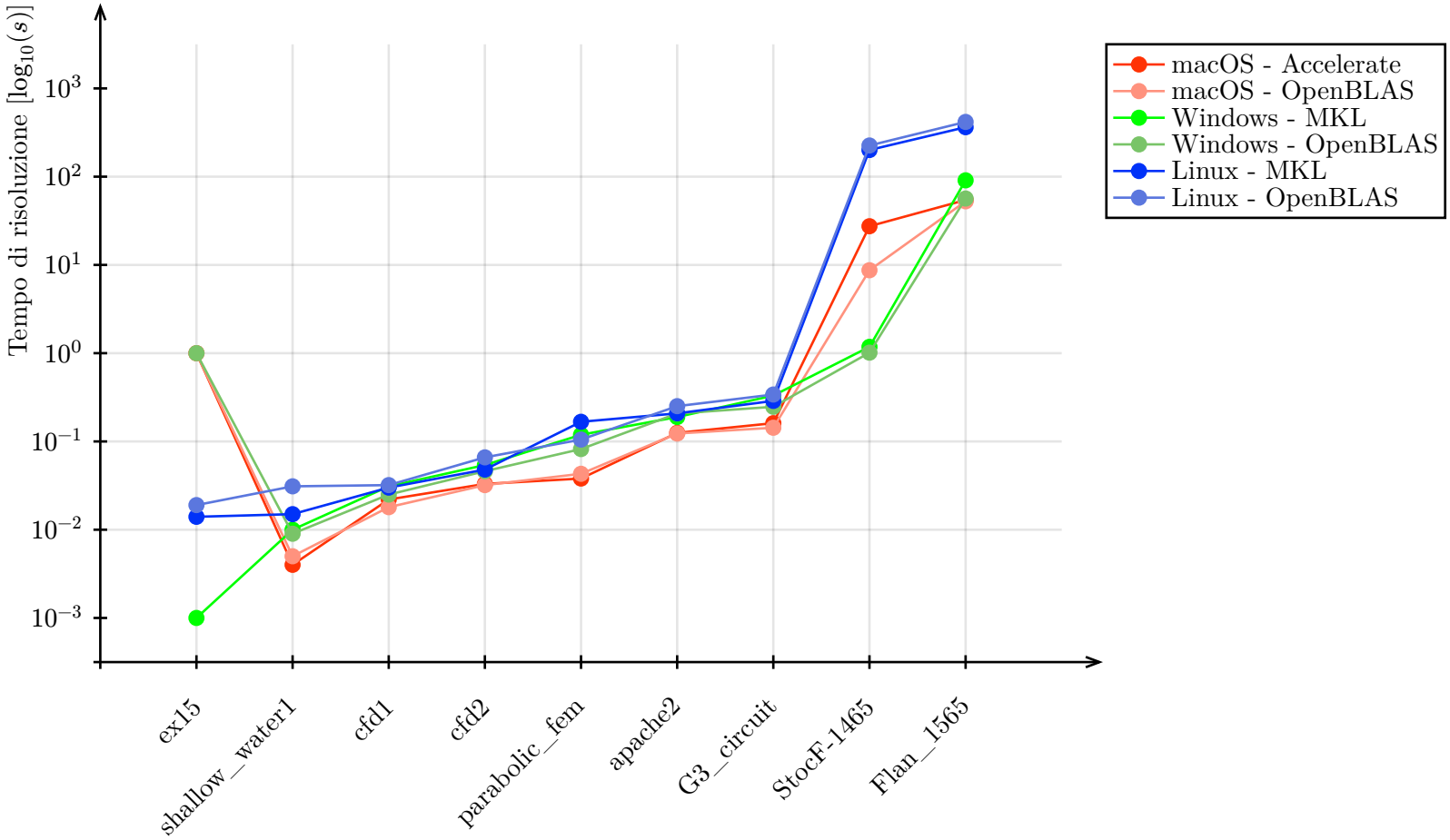

Figura 6: Confronto tempo di risoluzione tra sistemi operativi su C++

Riepilogo dei Tempi Compessivi.

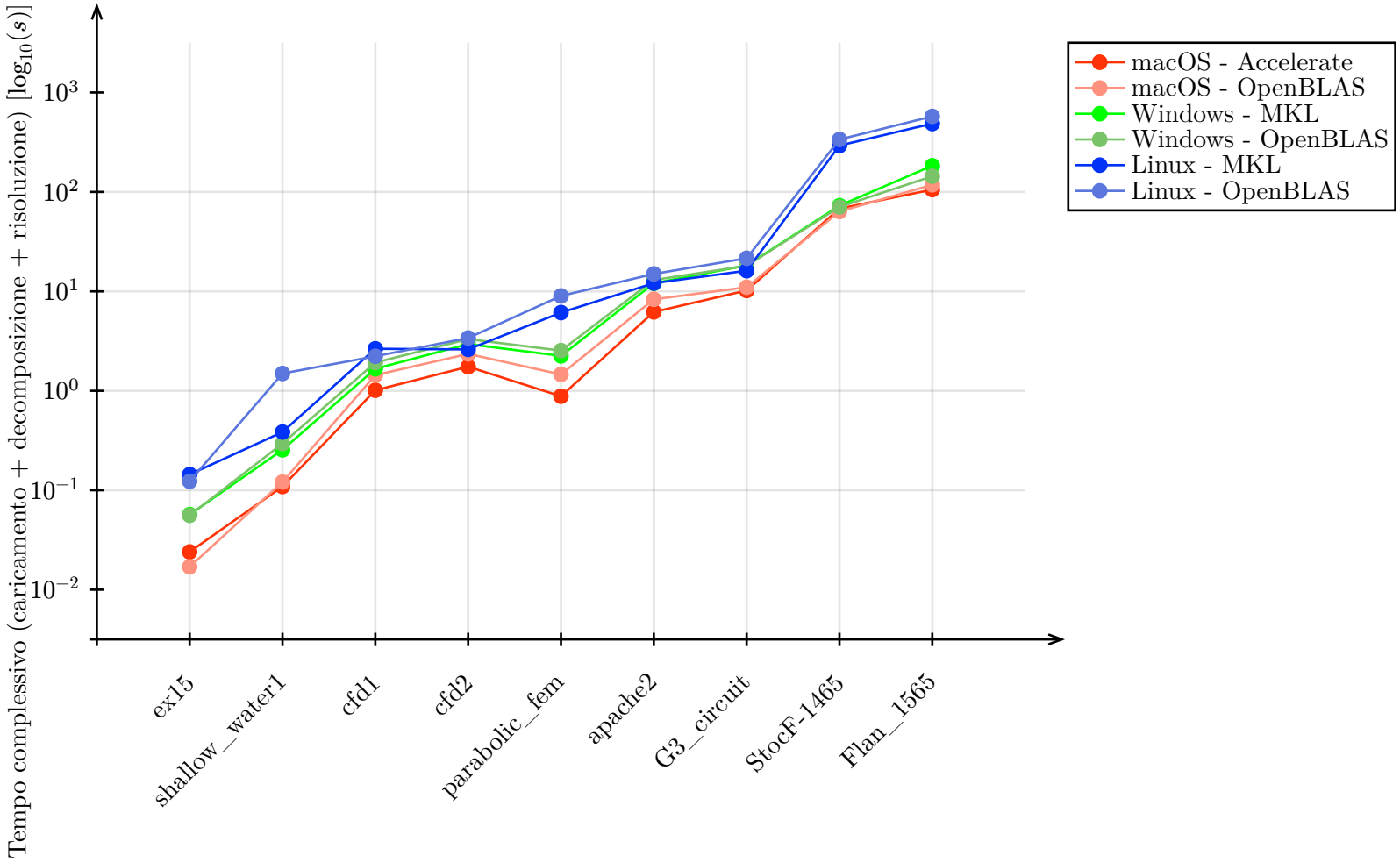

Figura 7: Confronto tempo complessivo tra sistemi operativi su C++

Guardando i tempi notiamo che in generale linux sembra essere piu lento di tutti gli altri sistemi operativi, questo è dovuto al fatto che per linux usiamo WSL2. Mentre macOS sembra essere il piu veloce di tutti, ma non troppo rispetto a Windows, questo è dovuto all'hardware migliore rispetto a quello per Windows e Linux.

3.5.3. Errore Relativo

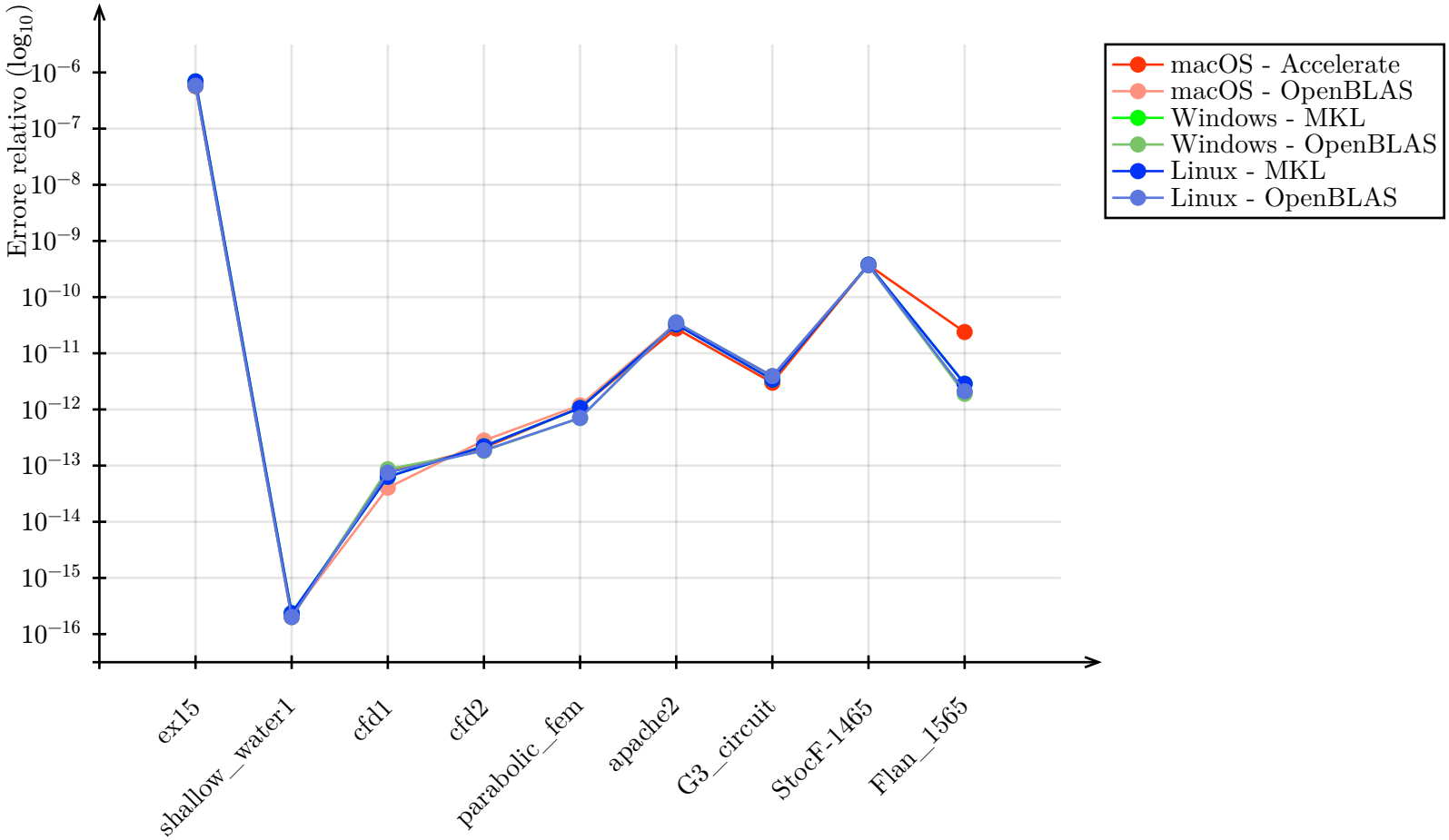

Figura 8: Confronto errore relativo tra sistemi operativi su C++

Invece guardando l'errore notiamo che ci sono delle piccole differenze tra i sistemi operativi e i blas, ma non sono significative. Questo è un buon risultato, poiché significa che la libreria CHOLMOD funziona in modo simile su entrambe le piattaforme e con diversi BLAS. E probabilemnte la differenza è dovuta ai diversi compilatori e alle loro ottimizzazioni.

3.6. Confronto MATLAB e C++

Inanzitutto è importante notare come in MATLAB le due matrici più grandi non siano state completate, questo dovuto probabilmente alla versione obsoleta di CHOLMOD in MATLAB, che non ha dei miglioramenti negli algoritmi di riordinamento e anche correzzione di errori.

3.6.1. Memoria

Dato che la memoria in MATLAB non è affidabile, invece di fare un confronto abbiamo deciso di analizzare l'uso in memoria di C++, aspettandoci che l'uso di memoria sia equiparabile a quella di MATLAB, dato l'utilizzo di CHOLMOD.

L'analisi dell'uso della memoria basandoci sulla Tabella 3 mostra i seguenti punti salienti:

1. **Carico di memoria iniziale:** Osserviamo che le matrici più piccole, come ex15 e shallow_water1, hanno un consumo ridotto di memoria, inferiore ai 10 MB. Questo è prevedibile, poiché la loro complessità computazionale è limitata. Tuttavia, quando esaminiamo matrici molto più grandi, come Flan_1565, notiamo un incremento drastico del carico di memoria, che supera i 1.8 GB. Questo indica che l'allocazione della memoria iniziale cresce proporzionalmente alla dimensione e alla complessità della matrice.
2. **Memoria richiesta per la decomposizione:** Il processo di decomposizione delle matrici rappresenta il momento più intensivo in termini di memoria. Ad esempio, la decomposizione della matrice Flan_1565 richiede oltre 21 GB di memoria. Questo suggerisce che, per strutture di grande dimensione, l'algoritmo utilizzato deve gestire un enorme quantitativo di dati e operazioni, generando un picco di utilizzo. Matrici di media grandezza come apache2 e G3_circuit richiedono invece circa 1.7-1.8 GB, evidenziando una crescita meno drastica ma comunque consistente. Questo è dovuto al fenomeno del fill-in che anche se ridotto dato l'utilizzo di algoritmi di riordinamento, è comunque presente e richiede una certa quantità di memoria.
3. **Picco di memoria di decomposizione:** In diversi casi, il picco di memoria durante la fase di decomposizione è inferiore al valore totale della memoria richiesta. Questo può significare che l'allocazione della memoria varia nel tempo e viene gestita dinamicamente, evitando sprechi di risorse. In pratica, la memoria viene allocata progressivamente secondo necessità, ottimizzando l'uso delle risorse disponibili.
4. **Memoria richiesta per la risoluzione:** Un aspetto interessante è che, rispetto alla decomposizione, la fase di risoluzione della matrice ha un impatto molto più contenuto sull'utilizzo della memoria. Questo accade perché la risoluzione si basa sui risultati ottenuti in fase di decomposizione e non richiede un'elaborazione intensiva sugli stessi dati. Di conseguenza, il consumo di memoria rimane relativamente basso.

3.6.2. Tempi

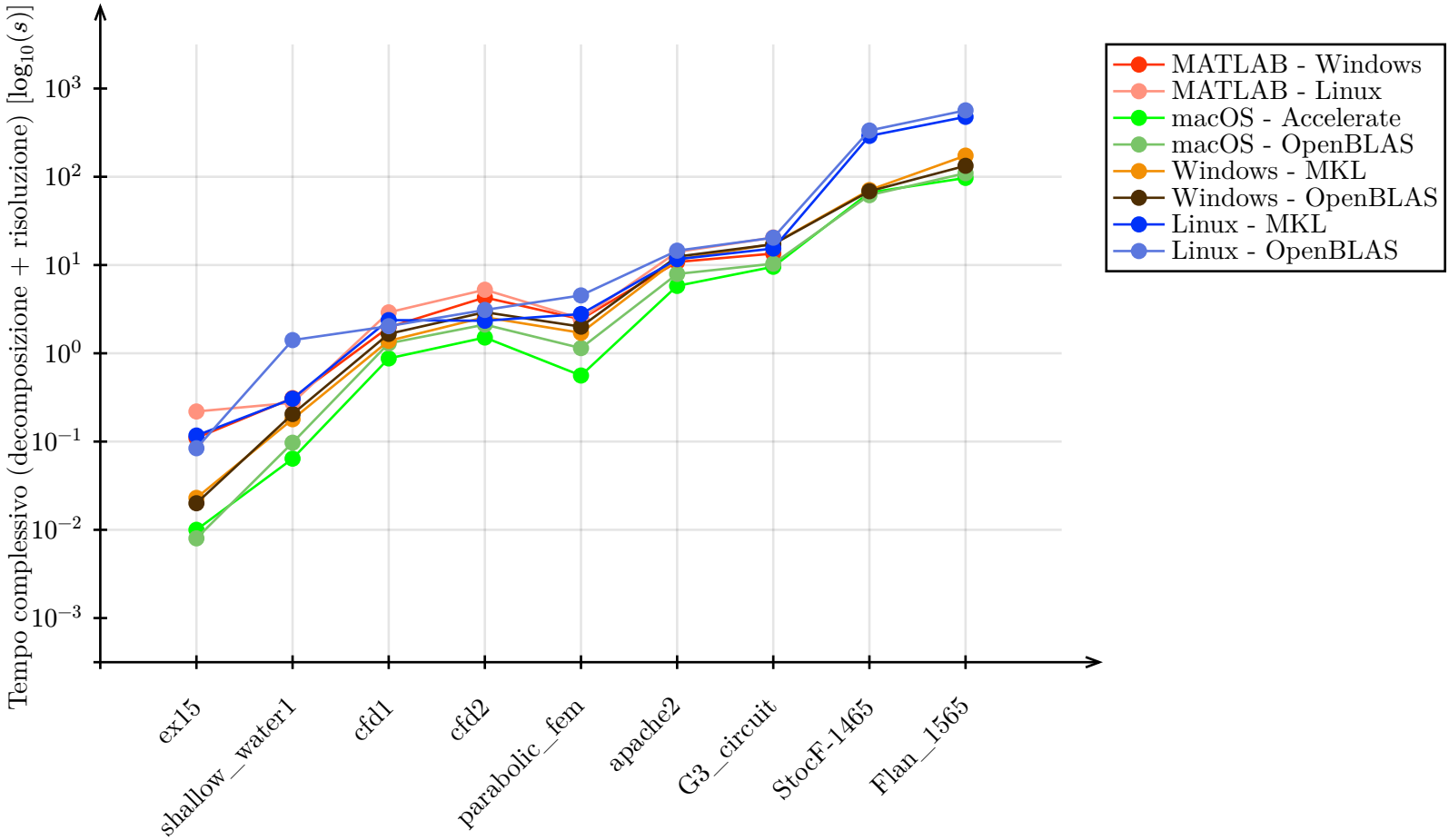

Figura 9: Confronto tempo complessivo tra MATLAB e C++

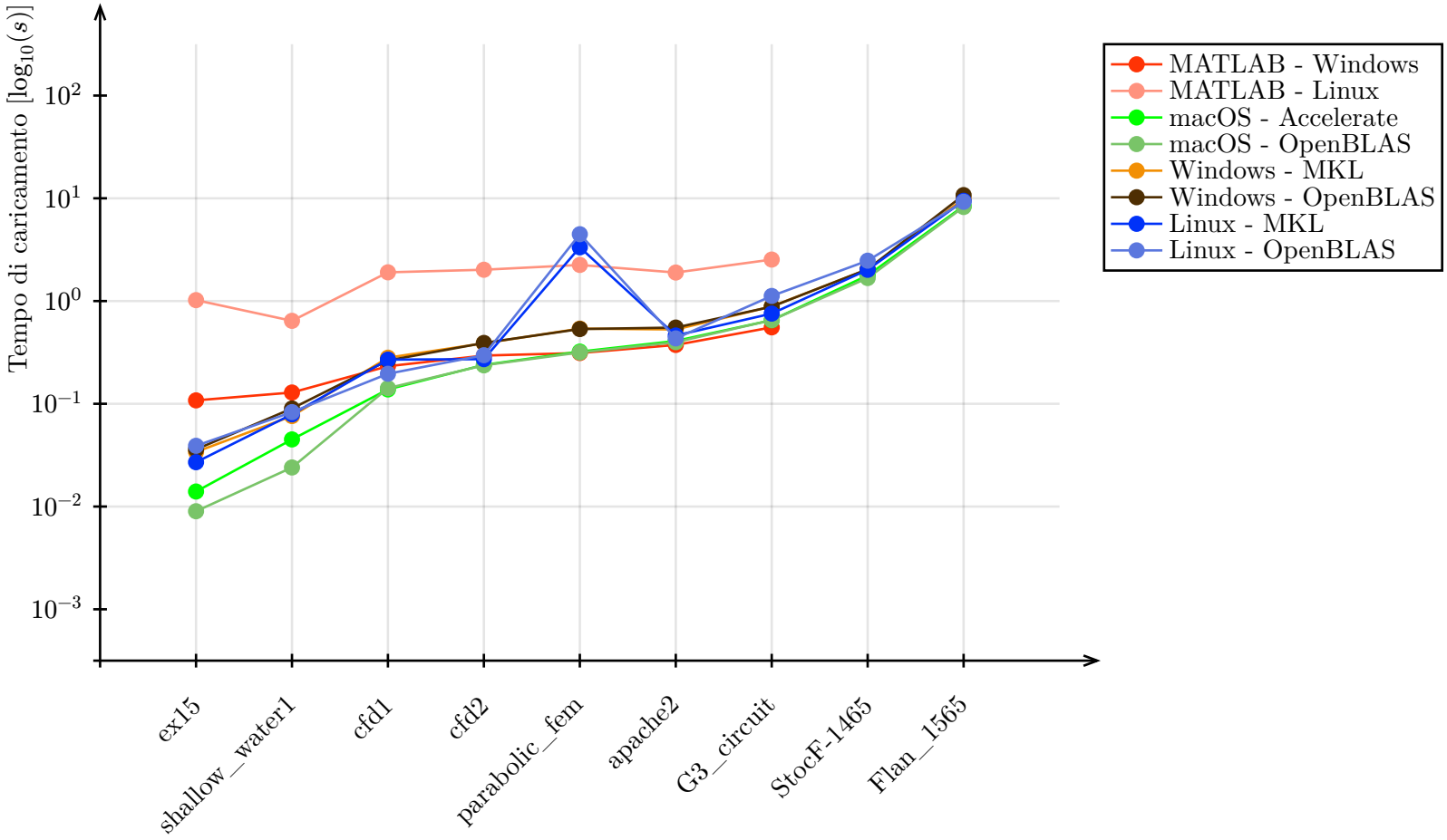

Figura 10: Confronto tempo complessivo tra MATLAB e C++

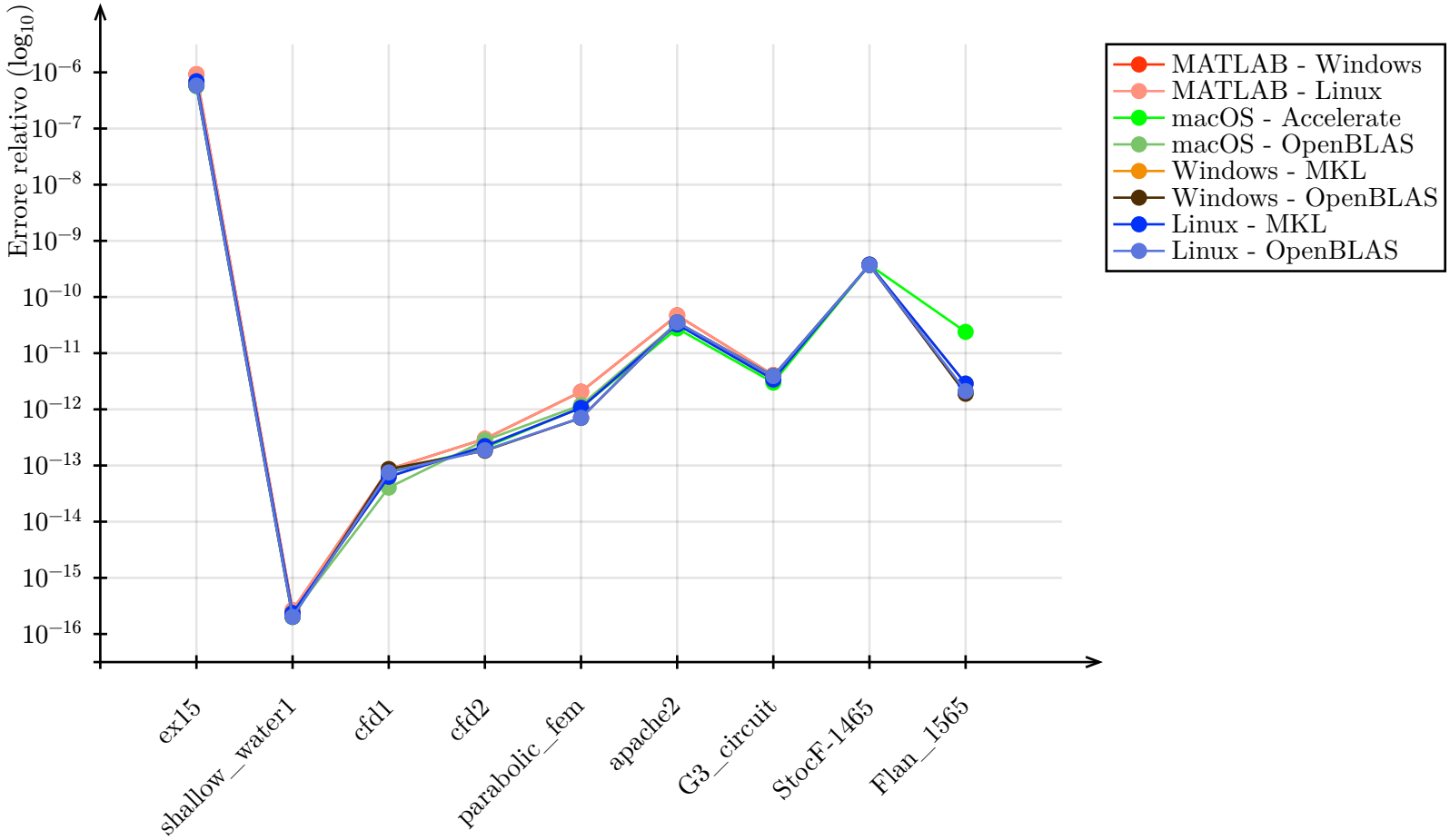

Figura 11: Confronto tempo complessivo tra MATLAB e C++

Bibliografia

[1] T. Davis, «DrTimothyAldenDavis/SuiteSparse». Consultato: 9 aprile 2025. [Online]. Disponibile su: https://github.com/DrTimothyAldenDavis/SuiteSparse

[2] «BLAS (Basic Linear Algebra Subprograms)». Consultato: 8 maggio 2025. [Online]. Disponibile su: https://netlib.org/blas/

[3] «LAPACK — Linear Algebra PACKage». Consultato: 8 maggio 2025. [Online]. Disponibile su: https://netlib.org/lapack/