Sistemi di Calcolo e Applicazioni

Progetto di fine corso

Massimo Buniy
massimo.buniy@students.uniroma2.eu

Lorenzo Grande
lorenzo.grande@students.uniroma2.eu

***Sommario*—Questo documento presenta i risultati ottenuti nello sviluppo e nell'analisi delle implementazioni del prodotto matrice-vettore, realizzate con OpenMP e CUDA.**
Le prestazioni sono state valutate sul server di dipartimento messo a disposizione, evidenziando i vantaggi e le limitazioni dei due approcci.

I. Introduzione

Il progetto verte sulla realizzazione di un nucleo di calcolo per il prodotto tra una matrice sparsa A e un vettore x, in grado di calcolare l'espressione $y \leftarrow Ax$. Questa operazione, nota come SpMV (*Sparse Matrix-Vector Multiplication*), è ampiamente utilizzata in ambiti quali il calcolo scientifico, l'analisi numerica e le simulazioni su larga scala.

In questo lavoro sono stati analizzati due approcci paralleli per accelerare lo SpMV:

- **OpenMP**, per sfruttare il parallelismo a livello di CPU multi-core;
- **CUDA**, per utilizzare la potenza di calcolo massicciamente parallelo delle GPU.

Per entrambe le tecnologie sono stati adottati due formati di rappresentazione delle matrici sparse:

- **CSR** (*Compressed Sparse Row*);
- **HLL** (*Hybrid Linear List*).

L'obiettivo principale è confrontare le prestazioni delle diverse combinazioni (OpenMP vs. CUDA e CSR vs. HLL), valutando:

- tempi di esecuzione;
- scalabilità al variare delle dimensioni del problema;

II. Formato delle matrici

Poiché l'obiettivo principale del progetto è la realizzazione di un prodotto matrice-vettore, è fondamentale comprendere innanzitutto le caratteristiche degli oggetti coinvolti.

Un elemento comune a tutte le matrici indicati per i test è la distribuzione dei valori non nulli: si tratta infatti di *matrici sparse*. Di conseguenza, ai fini dell'elaborazione numerica, soltanto una piccola parte degli elementi risulta effettivamente rilevante rispetto alla dimensione complessiva della matrice, soprattutto considerando i sistemi di grandi dimensioni che si incontrano nei casi reali.

Nel contesto delle matrici sparse si possono distinguere due principali categorie:

- le matrici sparse **strutturate**, in cui la posizione degli elementi non nulli segue schemi ben definiti (ad esempio matrici tridiagonali, a banda, ecc.);
- le matrici sparse **non strutturate**, in cui la distribuzione dei valori non nulli non segue alcun ordine prestabilito.

Per semplificare la progettazione, tutte le matrici — anche quelle strutturate — sono state trattate come se fossero non strutturate. Questo approccio comporta una maggiore occupazione di memoria rispetto all'uso di formati dedicati per le matrici strutturate, ma consente di applicare un unico algoritmo in modo uniforme a tutti i casi.

Nel trattamento delle matrici sparse, non è solo l'algoritmo di calcolo a determinare l'efficienza, ma anche — e soprattutto — la modalità di memorizzazione della matrice. La rappresentazione dei dati ha un impatto cruciale: una struttura di memorizzazione adatta permette di ridurre sia lo spazio occupato sia il numero di accessi in memoria, elemento spesso determinante per le prestazioni nei moderni sistemi computazionali.

A differenza delle matrici dense, in cui tutti gli elementi (compresi gli zeri) vengono memorizzati esplicitamente, una matrice sparsa può essere codificata memorizzando solo gli elementi non nulli e le informazioni necessarie a determinarne la posizione. Questo approccio consente di risparmiare memoria e di evitare operazioni inutili durante l'esecuzione del prodotto $y = Ax$.

Nel contesto di questo progetto, sono stati utilizzati principalmente due formati di rappresentazione: il formato *CSR* e il formato *HLL*.

A. Formato CSR

Il formato *CSR* (Compressed Sparse Row) è una rappresentazione compatta per matrici sparse che consente un accesso efficiente ai dati riga per riga. Una matrice di dimensioni $M \times N$, contenente NZ elementi non nulli, viene rappresentata attraverso le seguenti strutture dati:

- **M**: numero di righe della matrice;
- **N**: numero di colonne;
- **IRP(1:M+1)**: vettore dei puntatori all'inizio di ciascuna riga nel vettore dei dati. L'elemento `IRP(i)` indica l'indice di inizio degli elementi della riga i, mentre `IRP(M+1)` corrisponde a $NZ + 1$, segnando la fine dell'ultima riga;
- **JA(1:NZ)**: vettore contenente gli indici di colonna corrispondenti agli elementi non nulli;
- **AS(1:NZ)**: vettore dei valori dei coefficienti non nulli.

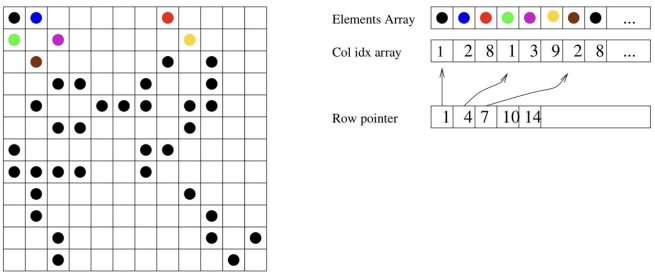

Figura 1: Rappresentazione CSR

B. ELLPACK

Il formato *ELLPACK* è un metodo di rappresentazione delle matrici sparse che si basa sull'assunzione di un numero massimo noto di elementi non nulli per riga. Data una matrice di dimensioni $M \times N$, contenente NZ elementi non nulli e con $MAXNZ$ che rappresenta il numero massimo di non zeri presenti in una singola riga, la matrice viene descritta tramite le seguenti strutture:

- **M**: numero di righe della matrice;
- **N**: numero di colonne;
- **MAXNZ**: massimo numero di elementi non nulli per riga;
- **JA(1:M, 1:MAXNZ)**: array bidimensionale che contiene, per ciascuna riga, gli indici di colonna degli elementi non nulli. Le posizioni inutilizzate vengono riempite con un valore di padding (tipicamente zero o -1);
- **AS(1:M, 1:MAXNZ)**: array bidimensionale dei coefficienti non nulli, organizzati riga per riga. Anche qui, eventuali posizioni inutilizzate sono riempite con zeri.

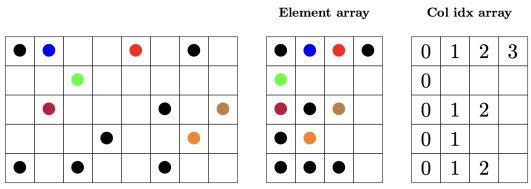

Figura 2: Rappresentazione ELLPACK

C. HLL

Il formato *HLL* (Hybrid Linear List) è una strategia ibrida per la rappresentazione di matrici sparse, pensata per combinare i vantaggi del formato ELLPACK con una maggiore flessibilità nella gestione di righe con densità variabile. La matrice viene rappresentata secondo i seguenti passaggi:

- Si definisce un parametro **HackSize**, che indica il numero di righe da includere in ciascun blocco;
- La matrice di input viene suddivisa in blocchi orizzontali composti da **HackSize** righe ciascuno;
- Ogni blocco viene quindi rappresentato utilizzando il formato *ELLPACK*, consentendo un accesso efficiente ai dati all'interno del blocco stesso.

Questo approccio consente di mitigare uno dei principali limiti del formato ELLPACK — ovvero la presenza di molte celle inutilizzate in caso di forti disomogeneità tra le righe — adattando dinamicamente la struttura della matrice alla sua effettiva distribuzione di non zeri.

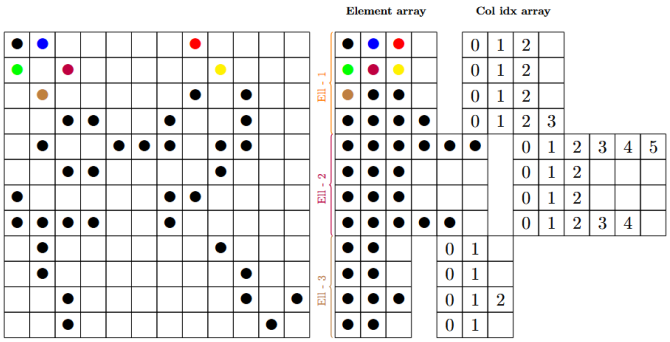

Figura 3: Rappresentazione HLL

III. Metriche

Per analizzare le performance del nostro progetto di calcolo, abbiamo adottato due metriche fondamentali: il numero di FLOPS (Floating Point Operations per Second) e lo SpeedUp. Questi indicatori permettono di valutare rispettivamente l'efficienza computazionale e i benefici ottenuti grazie alla parallelizzazione.

A. FLOPS

I FLOPS (Floating Point Operations Per Second) rappresentano il numero di operazioni in virgola mobile che la CPU è in grado di eseguire in un secondo. Per il nostro caso, il valore viene calcolato con la seguente formula:

$$\text{FLOPS} = \frac{2 \cdot NZ}{T}$$

dove:

- **NZ** è il numero di elementi non nulli nella matrice;
- **T** è il tempo medio di esecuzione dell'operazione.

B. SpeedUp

Lo SpeedUp misura il guadagno in termini di prestazioni ottenuto con la parallelizzazione, confrontando i tempi di esecuzione tra la versione sequenziale e quella parallela del programma. È definito dalla formula:

$$\text{SpeedUp} = \frac{T_s}{T_p}$$

dove:

- T_s è il tempo di esecuzione della versione sequenziale;
- T_p è il tempo di esecuzione della versione parallela.

Un valore di SpeedUp maggiore di 1 indica un miglioramento effettivo grazie alla parallelizzazione.

C. Nota sulla misurazione dei tempi

a) Tutti i tempi di esecuzione riportati, sia per il calcolo dei FLOPS sia per lo SpeedUp, sono stati ottenuti come media su dieci run consecutive dello stesso algoritmo. Questo approccio consente di ridurre l'influenza di variazioni occasionali nelle misurazioni, garantendo una valutazione più affidabile delle prestazioni.
b) Le misurazioni dei tempi di esecuzione utilizzate per il calcolo delle metriche si riferiscono esclusivamente alla fase computazionale dell'algoritmo, escludendo i tempi di caricamento dei dati in memoria e i controlli di correttezza. Ogni tempo riportato rappresenta la media su dieci esecuzioni consecutive dello stesso algoritmo, al fine di garantire maggiore affidabilità e ridurre l'impatto di eventuali fluttuazioni.

IV. OpenMP

A. Introduzione

OpenMP (Open Multi-Processing) è un'interfaccia di programmazione basata su direttive, progettata per facilitare lo sviluppo di programmi paralleli su architetture con memoria condivisa. È supportata da diversi compilatori C, C++ e Fortran, ed è ampiamente utilizzata per la sua semplicità di integrazione all'interno di codice seriale esistente. Nel nostro progetto, abbiamo adottato il linguaggio C per implementare la parallelizzazione del prodotto matrice-vettore, sfruttando un insieme di t thread per ripartire il carico computazionale. L'obiettivo era quello di ottenere una distribuzione equilibrata del lavoro tra i thread, riducendo il tempo di esecuzione complessivo e migliorando l'efficienza del calcolo.
Nel corso del progetto sono state sviluppate e confrontate diverse strategie di parallelizzazione del prodotto matrice-vettore. Le implementazioni differiscono per vari aspetti tecnici che saranno analizzati nel dettaglio in seguito. In particolare, sono state considerate:

- **Modalità di assegnazione del lavoro ai thread**: alcune soluzioni suddividono le righe della matrice equamente in blocchi statici per ogni thread, mentre altre impiegano la direttiva `#pragma omp for` per una suddivisione automatica gestita dal compilatore.
- **Uso di ottimizzazioni SIMD**: in alcune versioni è stato impiegato il pragma `simd` con riduzione per sfruttare il parallelismo a livello di istruzione all'interno dei thread.
- **Strategie di scheduling**: oltre alla suddivisione statica, sono state testate modalità di scheduling automatico, affidando a OpenMP la scelta della politica di bilanciamento.
- **Assegnazione esplicita di intervalli personalizzati**: una delle soluzioni prevede la suddivisione manuale dell'intervallo di righe per ciascun thread, utilizzando una struttura ausiliaria (`ThreadDataRange`) per un controllo fine del bilanciamento del carico.

Queste soluzioni hanno permesso di esplorare diverse configurazioni, valutando come le scelte implementative influenzino le prestazioni del calcolo parallelo, in termini di *speedup* e numero di *FLOPs* eseguiti. Infine, le performance dell'algoritmo sono state analizzate al variare del numero di thread, fino al massimo consentito dall'hardware utilizzato. Sul server del dipartimento, in particolare, è stato possibile utilizzare fino a 40 thread in parallelo.
Questa strategia sperimentale ci ha permesso di valutare in modo sistematico l'impatto del parallelismo sulle prestazioni del sistema, evidenziando i benefici e i limiti derivanti dall'uso di OpenMP in ambito scientifico.

B. Prodotto Matrice-Vettore con OpenMP nel formato CSR

Come detto in precedenza, nel seguente progetto sono state implementate diverse soluzioni per il calcolo del prodotto matrice-vettore nel caso di matrici sparse, rappresentate nel formato **CSR (Compressed Sparse Row) e HLL**.

L'obiettivo principale è stato quello di analizzare e confrontare diverse strategie di parallelizzazione con **OpenMP**, valutando l'efficienza e la scalabilità delle soluzioni proposte.

Nel seguito della sezione, ciascuna versione relativa alla rappresentazione CSR sarà descritta con pseudocodice e accompagnata da una breve analisi che ne spiega la logica e le motivazioni progettuali.

1) Soluzione 1

Algorithm 1: Prodotto Matrice-vettore in formato CSR: Divisione esplicita delle righe

Input: CSRMatrix `csr`, vettore `vector`, numero di thread `num_threads`
Output: Vettore risultato `result`

Controllo input:
if `csr = NULL` **then**
  Stampare errore e terminare
end
if `vector = NULL` **then**
  Stampare errore e terminare
end
Inizializzazione:
Creare `result` di dimensione M
if *creazione fallita* **then**
  Stampare errore e terminare
end
Impostare numero di thread OpenMP a `num_threads`
Inizio sezione `parallel`:
Calcolo del numero di righe da assegnare a ciascun thread
Ogni thread calcola il prodotto riga-colonna per le righe a lui associate
Fine sezione `parallel`
return `result`

Questa versione divide esplicitamente le righe tra i thread. È utile per comprendere come avviene la parallelizzazione a basso livello, ma non ha ottimizzazioni nella gestione di come le righe sono associate ai vari thread.

2) Soluzione 2

Algorithm 2: Prodotto Matrice-vettore in formato CSR: Uso di `omp parallel for`

Input: CSRMatrix `csr`, vettore `vector`, numero di thread `num_threads`
Output: Vettore risultato `result`

Controllo input:
if `csr = NULL` **then**
  Stampare errore e terminare
end
if `vector = NULL` **then**
  Stampare errore e terminare
end
Inizializzazione:
Creare `result` di dimensione M
if *creazione fallita* **then**
  Stampare errore e terminare
end
Impostare numero di thread OpenMP a `num_threads`
Inizio direttiva `omp parallel for`:
for *ogni riga* i *da* 0 *a* $M-1$ **do**
  Calcolare il prodotto riga-colonna relativo alla riga i
  Salvare il risultato in `result.val[i]`
end
Fine direttiva `omp parallel for`
return `result`

In questa versione si sfrutta `omp parallel for` per parallelizzare automaticamente il ciclo sulle righe. Rispetto alla divisione manuale, il vantaggio principale è una gestione automatica ed efficiente della suddivisione del lavoro tra i thread, senza bisogno di calcolare esplicitamente gli intervalli.

3) Soluzione 3

Algorithm 3: Prodotto Matrice-vettore in formato CSR: Uso combinato di `omp parallel for` e `omp simd`

Input: CSRMatrix `csr`, vettore `vector`, numero di thread `num_threads`
Output: Vettore risultato `result`

Controllo input:
if `csr = NULL` **then**
  Stampare errore e terminare
end
if `vector = NULL` **then**
  Stampare errore e terminare
end
Inizializzazione:
Creare `result` di dimensione M;
if *creazione fallita* **then**
  Stampare errore e terminare;
end
Impostare numero di thread OpenMP a `num_threads`;
Inizio direttiva `omp parallel for`:;
for *ogni riga* i *da* 0 *a* $M-1$ **do**
  Inizializzare `sum` a 0.0;
  Inizio direttiva `omp simd` con `reduction`:;
  for *ogni elemento della riga* i **do**
    Calcolare il prodotto e sommarlo a `sum`;
  end
  Fine direttiva `omp simd`;
  Salvare `sum` in `result.val[i]`;
end
Fine direttiva `omp parallel for`;
return `result`;

In questa versione si combina `omp parallel for` con `omp simd` all'interno del ciclo interno. La direttiva `omp simd` permette al compilatore di vettorializzare il ciclo interno, sfruttando le unità SIMD presenti nei moderni processori per sommare più elementi contemporaneamente. Questo approccio migliora ulteriormente le prestazioni rispetto alla sola parallelizzazione sulle righe, specialmente per righe molto dense.

4) Soluzione 4

In questa versione, rispetto alla Soluzione 3, si è aggiunta la clausola `schedule(auto)` alla direttiva `omp parallel for`.

- La clausola `schedule(auto)` delega al compilatore la scelta della strategia di distribuzione delle iterazioni tra i thread.
- In teoria, il compilatore può decidere una suddivisione più efficiente basandosi sulle caratteristiche della macchina e del carico di lavoro.

È stato ritenuto utile provare questa modifica per verificare se un bilanciamento dinamico automatico poteva portare a miglioramenti prestazionali rispetto a una suddivisione statica implicita.

5) Soluzione 5

Algorithm 4: Prodotto Matrice-vettore in formato CSR: Divisione manuale dei range

Input: CSRMatrix `csr`, vettore `vector`, numero di thread `num_threads`, struttura `tdr`
Output: Vettore risultato `result`

Controllo input:

```
if csr = NULL then
    Stampare errore e terminare;
end
if vector = NULL then
    Stampare errore e terminare;
end
if tdr = NULL then
    Stampare errore e terminare;
end
Inizializzazione:;
Creare result di dimensione M;
if creazione fallita then
    Stampare errore e terminare;
end
Impostare numero di thread OpenMP a num_threads;
Inizio sezione parallel:;
foreach thread t do
    Ottenere start e end da tdr[t];
    for i ← start to end − 1 do
        Calcolare il prodotto riga-colonna per la riga i;
        Salvare il risultato in result[i];
    end
end
Fine sezione parallel;
return result;
```

Questa versione divide esplicitamente il lavoro tra i thread, ma invece di assegnare un numero fisso di righe come nella Soluzione 1, utilizza una struttura `ThreadDataRange` che specifica esattamente l'intervallo di righe da elaborare per ogni thread.

Questo approccio può risultare utile in presenza di matrici sparse con distribuzione irregolare dei dati, perché permette una divisione del carico di lavoro più bilanciata tra i thread.

C. Prodotto Matrice-Vettore con OpenMP nel formato HLL

Oltre al formato **CSR**, il progetto prevede anche l'implementazione del prodotto matrice-vettore per matrici memorizzate nel formato **HLL**.

Il formato HLL organizza la matrice in diversi blocchi indipendenti, ciascuno rappresentato in formato **ELLPACK**. Di conseguenza, il calcolo è stato suddiviso in due fasi principali:

- Calcolo del prodotto tra ciascun blocco ELLPACK e il vettore;
- Aggregazione dei risultati parziali derivanti dai vari blocchi in un unico vettore risultato finale.

La parallelizzazione con **OpenMP** è stata effettuata a livello di blocchi: ogni thread elabora uno o più blocchi ELLPACK, in maniera indipendente dagli altri.

Non è stata invece introdotta parallelizzazione interna alla funzione che calcola il prodotto matrice-vettore del singolo blocco. Tale scelta è motivata dal fatto che i blocchi ELLPACK risultano generalmente di dimensione ridotta: parallelizzare anche all'interno avrebbe comportato un significativo overhead dovuto alla gestione dei thread e alla sincronizzazione, senza un reale guadagno in termini di prestazioni. Anzi, le misure sperimentali hanno mostrato che l'aggiunta di una seconda parallelizzazione interna peggiorava sensibilmente i tempi di esecuzione, a causa del costo di scheduling dei thread e della pressione sulla memoria condivisa.

Di seguito riportiamo le varie soluzioni adottate per la divisione dei vari blocchi tramite l'uso di openMP.

1) Soluzione 1 (HLL)

Algorithm 5: Prodotto Matrice-vettore in formato HLL: Parallelizzazione sui blocchi

Input: HLLMatrix `hll`, vettore `vector`, numero di thread `num_threads`
Output: Vettore risultato `result`

```
Controllo input:;
if hll = NULL then
    Stampare errore e terminare;
end
if vector = NULL then
    Stampare errore e terminare;
end
Inizializzazione:;
Creare result di dimensione M;
if creazione fallita then
    Stampare errore e terminare;
end
Impostare numero di thread OpenMP a num_threads;
Inizio sezione parallel for sui blocchi:;
Ogni thread elabora uno o più blocchi ELLPACK in parallelo;
Se si verifica un errore, impostare un flag di errore globale;
Fine sezione parallel for;
if errore rilevato then
    Stampare errore e terminare;
end
return result;
```

Questa versione sfrutta la naturale suddivisione della matrice HLL in blocchi, assegnando ad ogni thread il calcolo del prodotto matrice-vettore sui diversi blocchi in parallelo. L'uso di un flag globale permette di rilevare e gestire eventuali errori in modo sicuro durante l'esecuzione parallela.

2) Soluzione 2 (HLL)

Algorithm 6: Prodotto Matrice-vettore in formato HLL: Parallelizzazione dinamica sui blocchi

Input: HLLMatrix `hll`, vettore `vector`, numero di thread `num_threads`
Output: Vettore risultato `result`

Controllo input:;
if *`hll` = NULL* **then**
  Stampare errore e terminare;
end
if *`vector` = NULL* **then**
  Stampare errore e terminare;
end
Inizializzazione:;
Creare `result` di dimensione M;
if *creazione fallita* **then**
  Stampare errore e terminare;
end
Impostare numero di thread OpenMP a `num_threads`;
Inizio sezione `parallel for` sui blocchi con schedulazione dinamica:;
Ogni thread elabora blocchi ELLPACK in modo dinamico;
Se si verifica un errore, impostare un flag di errore globale;
Fine sezione `parallel for`;
if *errore rilevato* **then**
  Stampare errore e terminare;
end
return `result`;

In questa versione, la parallelizzazione sui blocchi è effettuata con una *schedulazione dinamica*, che consente una distribuzione più flessibile del lavoro tra i thread. Ogni thread può elaborare blocchi in modo dinamico, permettendo un bilanciamento migliore del carico quando i blocchi hanno dimensioni diverse. Il flag globale di errore permette una gestione sicura degli errori durante l'esecuzione parallela.

3) Soluzione 3 (HLL)

Algorithm 7: Prodotto Matrice-vettore in formato HLL: Parallelizzazione con gestione esplicita dei thread

Input: HLLMatrix `hll`, vettore `vector`, numero di thread `num_threads`, dati dei thread `tdr`
Output: Vettore risultato `result`

Controllo input:;
if *`hll` = NULL* **then**
  Stampare errore e terminare;
end
if *`vector` = NULL* **then**
  Stampare errore e terminare;
end
if *`tdr` = NULL* **then**
  Stampare errore e terminare;
end
Inizializzazione:;
Creare `result` di dimensione M;
if *creazione fallita* **then**
  Stampare errore e terminare;
end
Impostare numero di thread OpenMP a `num_threads`;
Inizio sezione `parallel for` con gestione dei thread tramite `tdr`:;
Ogni thread elabora una porzione di blocchi ELLPACK assegnata da `tdr`;
Se si verifica un errore, impostare un flag di errore globale;
Fine sezione `parallel for`;
if *errore rilevato* **then**
  Stampare errore e terminare;
end
return `result`;

In questa versione, la parallelizzazione è gestita esplicitamente utilizzando una struttura `ThreadDataRange` (`tdr`) che definisce la porzione di lavoro assegnata a ciascun thread. Ogni thread calcola il prodotto matrice-vettore per una specifica gamma di blocchi. La gestione dell'errore è stata inclusa tramite un flag globale, che permette di fermare l'elaborazione in caso di errore. L'assegnazione dei lavori ai thread può migliorare il bilanciamento del carico, in particolare quando il numero di blocchi da elaborare non è uniformemente distribuito.

V. CUDA

CUDA (Compute Unified Device Architecture) è un framework sviluppato da NVIDIA che consente di eseguire calcoli paralleli sulla GPU. La sua principale utilità risiede nella possibilità di accelerare operazioni computazionalmente intensive distribuendo il carico tra migliaia di thread eseguiti simultaneamente.

L'organizzazione del codice CUDA si basa su una gerarchia di thread. Ogni kernel lanciato sulla GPU viene eseguito da un numero molto elevato di *thread*, che sono raggruppati in *blocchi*, a loro volta organizzati in una *griglia*. Ogni thread all'interno del blocco e ogni blocco all'interno della griglia possiedono identificatori univoci, che possono essere usati per accedere a specifiche sezioni dei dati da elaborare.

Sia i blocchi sia le griglie possono avere una struttura mono-, bi- o tri-dimensionale. Questo permette di modellare

l'esecuzione del kernel sulla GPU in modo coerente con la struttura dei dati trattati:

- Se la griglia o il blocco sono monodimensionali, ogni thread/blocco è identificato da un solo indice (x).
- Se sono bidimensionali, si usano due indici (x, y).
- Se sono tridimensionali, gli indici sono tre (x, y, z).

Questa organizzazione rende CUDA estremamente flessibile, permettendo di modellare l'elaborazione in base alla struttura dei dati. Un aspetto importante è che la **dimensione del blocco non è fissa**: il programmatore può sceglierla in fase di progettazione del kernel, in funzione della natura del problema e delle risorse hardware disponibili. Tuttavia, questa scelta deve tener conto di alcune considerazioni pratiche:

- La dimensione deve essere un multiplo di 32 per allinearsi alla struttura dei warp ed evitare sprechi di risorse computazionali.
- Deve rispettare i limiti hardware della GPU, come la dimensione massima del blocco (Nel nostro caso 2014 threads).
- Il numero totale di blocchi dovrebbe essere almeno pari (idealmente un multiplo) al numero di SM presenti nella GPU, per garantire un utilizzo efficace di tutte le unità computazionali disponibili.

Un **SM** (Streaming Multiprocessor) rappresenta l'unità fondamentale di calcolo di una GPU NVIDIA. Ogni SM è in grado di gestire più **warp** contemporaneamente, ossia gruppi di 32 thread che eseguono la stessa istruzione su dati differenti. Affinché l'esecuzione sia efficiente, è importante che i dati assegnati ai thread di uno stesso warp si trovino in locazioni di memoria contigue, così da permettere accessi coalescenti alla memoria globale.

CUDA mette inoltre a disposizione due tipologie principali di memoria: la *memoria globale* (global memory), accessibile da tutti i thread ma con tempi di accesso più lenti, e la *memoria condivisa* (shared memory), più veloce ma condivisa solo tra i thread di uno stesso blocco. L'utilizzo della memoria condivisa può migliorare significativamente le prestazioni, ma comporta la necessità di sincronizzare esplicitamente i thread durante l'accesso concorrente in scrittura, per evitare condizioni di race.

A. Implementazioni CUDA

Come fatto anche per OpenMP, sono state sviluppate e confrontate più versioni dell'algoritmo di prodotto matrice-vettore anche in CUDA, partendo da una versione più basilare fino ad arrivare ad una implementazioni più ottimizzata, volte a sfruttare meglio il parallelismo offerto dall'architettura GPU.

L'obiettivo principale di queste versioni è massimizzare l'utilizzo delle risorse della GPU, considerando le principali problematiche tipiche di questo ambiente:

- **Occupazione e bilanciamento del carico**: suddividere il lavoro in modo equilibrato tra i thread per evitare tempi morti e massimizzare il throughput.
- **Efficienza della memoria**: minimizzare gli accessi non coalescenti alla memoria globale e sfruttare quando possibile la memoria condivisa.
- **Divergenza tra i thread**: ridurre la presenza di ramificazioni condizionali all'interno di un warp, che causano rallentamenti nell'esecuzione.

1) Inizializzazione dell'ambiente GPU

Prima dell'esecuzione dei kernel CUDA, è stato necessario preparare l'ambiente di esecuzione sulla GPU, sia per la rappresentazione CSR che per HLL. Tale fase di inizializzazione include:

- **Trasferimento della struttura dati della matrice dal lato host al device**, allocando in memoria un oggetto `CSRMatrix` (o equivalente per HLL) nella memoria unificata o globale della GPU.
- **Allocazione e trasferimento del vettore di input**, con successiva applicazione di `cudaMemAdvise` per indicare alla GPU che l'accesso sarà in sola lettura e per ottimizzare il posizionamento della memoria.
- **Allocazione del vettore risultato sul device**, con successivo caricamento della struttura dati e inizializzazione a zero prima di ogni chiamata kernel.
- **Configurazione della griglia e dei blocchi**, calcolando dinamicamente il numero di blocchi necessari in base al numero di righe della matrice e alla dimensione del blocco.

Questa fase è fondamentale per garantire il corretto funzionamento del kernel CUDA e per ottenere buone prestazioni durante il calcolo parallelo.

2) Implementazione CSR

Soluzione 1

Algorithm 8: Prodotto Matrice-vettore in formato CSR: Kernel CUDA - Soluzione 1

Input: Matrice CSR `csr`, vettore `v`, vettore risultato `result`
Output: Vettore risultato `result`

```
1  Inizializzazione:;
2  Creare una variabile sum inizializzata a 0;
3  Ogni thread elabora una riga della matrice:;
4  row ← indice della riga associata al thread;
5  if row < csr->M then
6  |  sum ← 0;
7  |  for j = csr->IRP[row] to csr->IRP[row
   |   + 1] - 1 do
8  |  |  sum += csr->AS[j] *
   |  |   v[csr->JA[j]];
9  |  end
10 |  result->val[row] = sum;
11 end
12 return result;
```

Questa soluzione sfrutta un kernel CUDA che calcola il prodotto matrice-vettore in formato CSR, parallelizzando il calcolo su ogni riga della matrice. Ogni thread è responsabile di una riga e somma i prodotti tra gli elementi non nulli della

matrice e gli elementi del vettore. L'approccio garantisce un'alta efficienza computazionale grazie alla parallelizzazione su GPU, che consente di gestire matrici sparse di grandi dimensioni.

Soluzione 2

Algorithm 9: Prodotto Matrice-vettore in formato CSR: Kernel CUDA con Warp Reduction - Soluzione 2

Input: Matrice CSR `csr`, vettore `v`, vettore risultato `result`
Output: Vettore risultato `result`

```
1  Inizializzazione:;
2  Calcolare l'ID del warp e la posizione del lane;
3  Ogni thread elabora una parte della riga:;
4  row_start, row_end ← indici di inizio e fine per
     la riga corrente;
5  sum ← 0;
6  for i = row_start + lane to row_end con
     passo WARP_SIZE do
7      if i < row_end then
8          sum += csr->AS[i] *
             v[csr->JA[i]];
9      end
10 end
11 mask ← __ballot_sync;
12 if row_start + lane >= row_end then
13     mask ← 0;
14 end
15 sum ← warpReduceSum(sum, mask);
16 if lane == 0 then
17     result->val[warp_id] = sum;
18 end
19 return result;
```

Questa soluzione introduce una riduzione a livello di warp per ottimizzare la somma degli elementi non nulli nella matrice CSR. Ogni warp elabora una riga della matrice, distribuendo il lavoro tra i thread all'interno del warp stesso. Utilizzando `__ballot_sync` e una riduzione in parallelo, l'approccio riduce il numero di operazioni globali necessarie e aumenta l'efficienza computazionale. La riduzione a livello di warp consente di ridurre il traffico di memoria e sfruttare appieno le capacità della GPU.

3) Implementazione HLL

Soluzione 1:

Algorithm 10: Prodotto Matrice-vettore in formato HLL: Parallelizzazione per blocchi

Input: Matrice HLL `hll`, vettore `vector`, vettore risultato `result`
Output: Vettore risultato `result`

```
1  Inizializzazione:;
2  block_id ← ID del blocco corrente;
3  thread_id ← ID del thread corrente;
4  if block_id ≥ hll->numBlocks then
5      return
6  end
7  block ← hll->blocks[block_id];
8  if thread_id ≥ block->M then
9      return
10 end
11 sum ← 0;
12 for j = 0 to block->MAXNZ - 1 do
13     col ← block->JA[thread_id][j];
14     sum += block->AS[thread_id][j] *
        vector[col];
15 end
16 result->val[block->startRow +
     thread_id] = sum;
17 return result;
```

In questa soluzione, ogni blocco CUDA è responsabile dell'elaborazione di un blocco ELLPACK della matrice HLL. All'interno di ciascun blocco CUDA, ogni thread è associato a una riga del blocco ELLPACK. Questo significa che ogni thread esegue il calcolo del prodotto matrice-vettore per una riga specifica del blocco ELLPACK, utilizzando i valori della riga e le rispettive colonne non nulle.

Il ciclo `#pragma_unroll` ottimizza l'elaborazione dei valori non nulli della riga, migliorando le prestazioni evitando il controllo su ciascun elemento della matrice. Grazie alla suddivisione in blocchi ELLPACK, la memoria è utilizzata in modo più efficiente, e la parallelizzazione è semplice, poiché ogni thread si occupa di una parte ben definita del calcolo.

Soluzione 2

Algorithm 11: Prodotto Matrice-Vettore HLL con ottimizzazione della coalescenza

Input: HLLMatrix `hll`, vettore `vector`, vettore risultato `result`
Output: Vettore risultato `result`

```
1  Inizializzazione:;
2  Impostare il numero di blocchi CUDA a
     hll.numBlocks;
3  foreach blocco CUDA block_id do
4      if block_id ≥ hll.numBlocks then
5          return
6      end
7      block ← hll.blocks[block_id];
```

Algorithm 11: (continua) Prodotto Matrice-Vettore HLL con ottimizzazione della coalescenza

```
8
9     foreach thread nel blocco CUDA thread_id do
10        if thread_id ≥ block.M then
11            return
12        end
13        sum ← 0;
14        for j = 0 to block.MAXNZ - 1 do
15            col ← block.JA[thread_id][j];
16            sum += block.AS[thread_id][j]
               * vector[col];
17        end
18        result.val[block.startRow +
           thread_id] ← sum;
19    end
20 end
```

In questa soluzione, ogni blocco CUDA elabora un blocco ELLPACK della matrice HLL, come nella soluzione precedente. Tuttavia, il risultato della somma parziale per ogni riga del blocco ELLPACK viene scritto nella memoria globale in modo che l'accesso sia coalescente. Questo significa che i thread scrivono i dati in posizioni di memoria contigue, ottimizzando le operazioni di lettura e scrittura.

La coalescenza della memoria si realizza con l'accesso contiguo a posizioni di memoria, migliorando l'efficienza dell'accesso alla memoria globale e riducendo la latenza dei trasferimenti dati. La struttura della matrice HLL consente di risparmiare memoria rispetto alla rappresentazione standard, migliorando l'efficienza complessiva.

VI. Benchmark

Per il collaudo delle implementazioni sono state utilizzate matrici sparse provenienti dalla *Suite Sparse Matrix Collection* disponibile all'indirizzo https://sparse.tamu.edu/. Le matrici sono state scaricate nel formato `MatrixMarket`.

Tra le matrici utilizzate per il collaudo figurano le seguenti:

adder_dcop_32	af23560	af_1_k101
amazon0302	bcsstk17	cage4
cant	cavity10	cop20k_A
Cube_Coup_dt0	dc1	FEM_3D_thermal1
lung2	mac_econ_fwd500	mhd4800a
mhda416	mcfe	ml_Laplace
nlpkkt80	olm1000	olafu
PR02R	raefsky2	rdist2
roadNet-PA	thermal1	thermal2
thermomech_TK	webbase-1M	west2021

Tabella I: Elenco delle matrici utilizzate per il collaudo

Le matrici coprono una vasta gamma di dimensioni, densità e strutture topologiche, risultando utili per valutare il comportamento degli algoritmi in scenari sia sintetici che reali.

A. Ambiente di test

Le misurazioni delle prestazioni sono state effettuate su un server del dipartimento, dotato delle seguenti caratteristiche hardware:

- **CPU**: Doppio socket Intel Xeon Silver 4210 @ 2.20GHz, per un totale di 20 core fisici e 40 thread logici. Ogni socket è associato a un nodo NUMA, con la seguente distribuzione: `NUMA node 0: CPU 0-9,20-29`, `NUMA node 1: CPU 10-19,30-39`.
- **Memoria**: 128 GB RAM.
- **GPU**: NVIDIA Quadro RTX 5000 con 48 multiprocessori, 49152 byte di memoria condivisa per blocco, e una dimensione massima di 1024 thread per blocco.

Il codice è stato sviluppato in locale utilizzando CLion, con esecuzione remota tramite connessione SSH al server. Il sistema operativo in esecuzione sul server è Linux, con compilatore `g++` per l'implementazione OpenMP e `nvcc` per l'implementazione CUDA.

B. Configurazione dell'esecuzione

Per l'implementazione **OpenMP**, i thread sono stati lanciati senza vincoli specifici sul binding NUMA: il sistema operativo ha gestito liberamente l'assegnazione dei thread ai core fisici e logici disponibili. Questo approccio non garantisce la località dell'accesso alla memoria, ma riflette un comportamento più realistico di esecuzione non ottimizzata. Ogni algoritmo è stato testato variando il numero di thread da 1 fino a 40, corrispondenti al numero massimo di thread logici disponibili sulla macchina (20 core fisici con hyper-threading abilitato). Questa scelta ha permesso di osservare l'andamento delle prestazioni in funzione del parallelismo disponibile.

Per l'implementazione **CUDA**, si è andata a studiare l'influenza della dimensione del blocco ('blockSize') sulle prestazioni. I valori testati sono stati i seguenti:

$$\{32,\ 64,\ 96,\ 128,\ 160,\ 192,\ 256,\ 320,\ 384,\ 512,\ 768,\ 1024\}$$

Ogni riga della matrice è stata assegnata a un blocco, con i thread del blocco che si suddividevano i non-zeri della riga per effettuare il prodotto scalare. I valori scelti per 'blockSize' riflettono un campionamento rappresentativo di dimensioni tipiche, includendo sia multipli della dimensione di warp (32), sia il massimo supportato dalla GPU (1024).

C. Modalità di misurazione

I tempi riportati escludono la fase di caricamento dei dati (lettura della matrice e del vettore da file). Nel caso di CUDA, i tempi relativi all'esecuzione del kernel sono stati distinti da quelli di trasferimento host-device.

Come già anticipato, ogni algoritmo è stato eseguito sequenzialmente per 10 volte consecutive. Questa ripetizione consente di mitigare l'effetto di eventuali variazioni casuali nell'esecuzione (ad esempio dovute a interferenze del sistema operativo o a fluttuazioni hardware), garantendo così una stima più robusta e affidabile delle prestazioni.

Nei risultati riportati è stata utilizzata la media aritmetica dei tempi misurati nelle dieci esecuzioni.

D. Classificazione delle matrici per dimensione

Le matrici utilizzate sono state classificate in base al numero di elementi non-zeri in quattro categorie:

- **small**: fino a 10k non-zeri
- **medium**: da 10k a 1M
- **large**: da 1M a 8M
- **xlarge**: matrici con dimensione superiore a 8M

Le soglie sono state scelte in modo da coprire una scala logaritmica di dimensioni, garantendo una suddivisione bilanciata tra matrici piccole, medie e grandi.

La classificazione delle matrici ha prodotto la seguente distribuzione:

- **small**: 4 matrici
- **medium**: 13 matrici
- **large**: 7 matrici
- **xlarge**: 6 matrici

Questa distribuzione consente di valutare le prestazioni degli algoritmi su una varietà bilanciata di dimensioni, con una maggiore concentrazione su matrici di dimensioni medie.

VII. Analisi critica delle prestazioni

In questa sezione vengono analizzati in dettaglio i risultati prestazionali ottenuti durante l'esecuzione del prodotto matrice-vettore, confrontando l'efficienza delle implementazioni in OpenMP e CUDA.

In particolare, in OpenMP si sono andate a misurare le prestazioni ottenute al variare del numero di thread a disposizione della API, osservando come il parallelismo a livello di CPU incida sull'efficienza al crescere del grado della concorrenza. Nel caso di CUDA, invece, si sono andate ad indagare le prestazioni degli algoritmi proposti al variare della dimensione dei brocchi (blocksize) con cui si faceva partire il kernel.

Le valutazioni tengono conto anche delle differenze introdotte dal formato di rappresentazione della matrice (CSR e HLL), che possono influenzare in modo significativo l'accesso alla memoria e, di conseguenza, le prestazioni complessive.

A. OpenMP

Nella seguente sottosezione andremo ad analizzare i risultati degli esperimenti ottenuti tramite OpenMP sia per la rappresentazione in formato CSR che in formato HLL.

1) CSR

L'analisi delle prestazioni delle cinque soluzioni openMP nel caso di matrice CSR ha evidenziato comportamenti distinti in base al numero di thread utilizzati e alla densità della matrice (ossia il numero di elementi non nulli).

Impatto del numero di thread Il primo aspetto osservato riguarda la suddivisione dei thread in due fasce:

- 0–20 thread: in questo range il sistema utilizza i core fisici della CPU. In questa fascia, l'aumento del numero di thread porta a un incremento regolare delle prestazioni (in GFLOPS), riflettendo un buon parallelismo.
- 21–40 thread: in questa seconda fascia si attivano anche gli hyperthread, ovvero i core logici forniti dal supporto hardware al multithreading simultaneo (SMT). In questa fase si osserva un incremento molto più contenuto delle prestazioni, spesso accompagnato da fluttuazioni o fenomeni di saturazione. Questo comportamento è coerente con il fatto che l'hyperthreading non raddoppia la capacità computazionale, ma permette solo un utilizzo più efficiente di risorse inattive. Inoltre, l'attivazione degli hyperthread introduce anche un overhead di sincronizzazione e una maggiore competizione per la cache e per la memoria condivisa, che può limitare i benefici ottenibili. Questo overhead può diventare particolarmente penalizzante nelle soluzioni in cui la gestione dei dati condivisi (come l'accesso concorrente alla memoria o le riduzioni globali) è più intensa, vanificando in parte i vantaggi del parallelismo aggiuntivo.

Nodi NUMA Un altro aspetto rilevante per l'analisi delle prestazioni è legato all'architettura NUMA (Non-Uniform Memory Access), in cui ciascun gruppo di core è associato a una porzione distinta di memoria locale. In questo contesto, l'accesso alla memoria locale è più veloce rispetto all'accesso alla memoria di un altro nodo NUMA. Tuttavia, i risultati presentati non tengono conto di questa caratteristica: non è stato imposto un vincolo per limitare l'esecuzione dei thread a un singolo nodo NUMA. Di conseguenza, i thread possono essere schedulati su core appartenenti a nodi diversi, introducendo variabilità nelle latenze di accesso alla memoria e possibili inefficienze dovute alla maggiore distanza tra dati e unità di calcolo. Questo può influire negativamente sulle prestazioni osservate, in particolare nei casi a più alto parallelismo.

Densità della matrice Un'altra osservazione significativa riguarda l'effetto del numero di elementi non nulli nella matrice: all'aumentare di questi ultimi, si registra un incremento dei GFLOPS per tutte le soluzioni. Ciò è atteso: più operazioni significano maggiore parallelismo sfruttabile e maggiore utilizzo delle risorse computazionali disponibili.

Confronto fra le soluzioni

- Tra *sol1*, *sol2* e *sol3*, la soluzione sol3 si distingue nettamente come la più efficiente. Questo risultato è coerente con la struttura del codice: sol3 rappresenta un'evoluzione delle prime due versioni, in quanto combina il parallelismo a livello di ciclo esterno (`omp parallel for`) con la direttiva `omp simd` e un'efficace riduzione.
- Le soluzioni *sol4* e *sol5* non mostrano un comportamento sempre prevedibile: in alcuni casi superano *sol3*, in altri no. *sol4* introduce la direttiva `schedule(auto)`, demandando al compilatore la scelta della suddivisione del carico; questo può portare a miglioramenti o peggioramenti a seconda del bilanciamento effettivo ottenuto. *sol5*, invece, si basa su una suddivisione manuale delle righe mediante la struttura `ThreadDataRange`, che può essere più efficiente in presenza di squilibri noti ma richiede un tuning accurato. Tuttavia, non è sempre la più

performante, probabilmente a causa dell'overhead della gestione personalizzata o di una suddivisione non ottimale nei casi generici.

2) HLL

L'analisi delle prestazioni per le tre soluzioni OpenMP sviluppate con il formato *HLL* (Hybrid Linked List), che internamente gestisce i blocchi secondo il formato ELLPACK, mostra andamenti coerenti con quanto osservato nel formato CSR, ma presenta anche caratteristiche distintive dovute alla diversa rappresentazione dei dati.

Impatto del numero di thread Anche nel caso HLL, si evidenzia una distinzione tra l'utilizzo di core fisici e logici:

- 0–20 thread: si osserva un incremento regolare delle prestazioni in GFLOPS all'aumentare del numero di thread. In questa fase, tutte le soluzioni scalano bene, grazie al parallelismo intrinseco garantito dalla suddivisione in blocchi ELLPACK.
- 21–40 thread: l'utilizzo degli hyperthread porta a un miglioramento più contenuto, in linea con quanto osservato nel formato CSR. In particolare, si nota una maggiore variabilità nei risultati, con fenomeni di saturazione o regressione delle prestazioni, probabilmente causati dalla maggiore pressione sulla cache e sulla memoria condivisa, e dal costo aggiuntivo della sincronizzazione. Questa maggiore pressione sulla cache in strutture HLL rispetto a CSR è dovuta a diversi fattori:
  1) **Accessi meno contigui e più irregolari**
     Strutture come HLL, che utilizzano blocchi e matrici allineate, si basano su array bidimensionali o su array di puntatori a blocchi. Ciò comporta accessi meno sequenziali alla memoria, ridotta località spaziale (i dati usati vicini nel tempo non sono necessariamente vicini in memoria) e quindi cache miss più frequenti.
  2) **Aumento del volume totale di dati attivi**
     Strutture come ELLPACK usano array di dimensione $M \times \text{MAXNZ}$, anche se molte righe contengono meno elementi di `MAXNZ`, generando un uso inefficiente della memoria.

  La maggiore pressione sulla cache nel caso di HLL rispetto a CSR è quindi causata da accessi meno regolari e più frammentati alla memoria, un maggior ingombro di memoria dovuto a dati inutili, e, nei contesti paralleli, dalla condivisione di dati e dal costo della sincronizzazione.

Densità della matrice Le prestazioni crescono sensibilmente all'aumentare del numero di elementi non nulli, sia per via dell'aumento delle operazioni computazionali, sia perché la rappresentazione ELLPACK diventa più efficiente con righe più dense (minor padding e overhead). Di conseguenza, la struttura a blocchi HLL riesce a mantenere un buon bilanciamento anche per matrici molto dense.

Confronto fra le soluzioni

- **sol1**: è la versione più semplice, in cui ogni thread elabora un blocco in modo indipendente, senza alcuna direttiva specifica per la pianificazione. Le prestazioni sono buone, ma spesso inferiori a *sol2* e *sol3* nei casi con molte righe, in quanto il carico può risultare sbilanciato.
- **sol2**: introduce la direttiva `schedule(dynamic)`, che permette una pianificazione dinamica dei blocchi da assegnare ai thread. Questa soluzione migliora il bilanciamento del carico e si rivela particolarmente efficace in presenza di blocchi eterogenei per numero di righe o densità. Tuttavia, l'overhead della pianificazione dinamica può penalizzare leggermente le prestazioni nei casi molto regolari.
- **sol3**: implementa una divisione manuale del lavoro tramite la struttura `ThreadDataRange`, che assegna in modo esplicito i blocchi o le righe da elaborare. Questa soluzione è quella che garantisce il maggiore controllo ed è particolarmente efficace se il carico è ben distribuito, mostrando prestazioni spesso superiori alle altre due. Tuttavia, nei casi generici o in presenza di squilibri imprevisti, può risultare meno flessibile della pianificazione dinamica di *sol2*.

I risultati mostrano che le prestazioni ottenute, misurate in Gflops, sono inferiori rispetto a quelle raggiunte con l'utilizzo della struttura CSR (Compressed Sparse Row). Questo scostamento è attribuibile principalmente al diverso comportamento in termini di accessi alla memoria. La CSR, infatti, rappresenta i dati in modo compatto e contiguo, consentendo accessi più efficienti grazie a una migliore località spaziale. Al contrario, nella nostra struttura (come ad esempio HLL), l'uso di puntatori e di strutture meno regolari comporta un maggior numero di accessi non contigui alla memoria, con conseguente aumento dei cache miss. Questo impatto negativo sull'efficienza degli accessi alla memoria si riflette direttamente sulla riduzione dei Gflops ottenuti. A conferma di questa tesi, vi è il fatto che la struttura dati HLL_Aligned, progettata per migliorare l'allineamento dei dati in memoria e limitare gli accessi disordinati, ha mostrato un incremento significativo in termini di Gflops rispetto alla versione base. Le implementazioni, pur mantenendo una logica simile a quella precedente, beneficiano della maggiore regolarità nella disposizione dei dati, che consente un migliore sfruttamento della cache e una riduzione dei tempi di accesso. Ciò conferma ulteriormente quanto ipotizzato riguardo all'impatto critico degli accessi in memoria sul rendimento computazionale.

B. Cuda

In questa sottosezione andremo ad analizzare i risultati degli esperimenti ottenuti tramite CUDA sia per la rappresentazione in formato CSR che in formato HLL.

1) CSR

Per quanto riguarda le prestazioni delle soluzioni sviluppate per CUDA, si osserva come queste siano fortemente influenzate dal numero di elementi non nulli presenti nella matrice. A differenza di quanto avviene con OpenMP, non è possibile

individuare una soluzione universalmente migliore: a seconda della matrice utilizzata, la *sol1* può risultare nettamente più performante della *sol2*, o viceversa. Un esempio emblematico è rappresentato dalle matrici `amazon0302` e `cant`, entrambe classificate come "large", ma caratterizzate da comportamenti diametralmente opposti in termini di prestazioni tra le due soluzioni.

Nel confronto tra le due soluzioni CUDA implementate per il prodotto matrice-vettore, emergono differenze significative al variare della blockSize. La prima soluzione (*sol1*), in cui ogni thread elabora una singola riga della matrice, risulta semplice ed efficace nel caso di righe di dimensione simile, ma soffre in presenza di un carico di lavoro sbilanciato. La seconda (*sol2*), in cui ogni warp collabora al calcolo di una riga, si dimostra più efficiente per righe lunghe e dense, ma tende a sprecare risorse computazionali quando le righe sono corte o scarsamente popolate.

In entrambe le implementazioni, l'aumento eccessivo della blockSize comporta generalmente un peggioramento delle prestazioni, dovuto a una riduzione dell'occupancy causata dalla saturazione delle risorse disponibili per blocco (come registri e memoria condivisa), nonché a una minore efficienza nell'utilizzo dei thread attivi. Questo giustifica le perdite osservate, spesso superiori al 10%, rispetto alle configurazioni con blockSize più contenuta.

2) HLL

A differenza del formato CSR, nel caso del formato HLL le due soluzioni CUDA mostrano un comportamento praticamente identico, con prestazioni simili al variare della `blockSize`.

Questa sezione vuole sottolineare l'importanza della disposizione dei dati in memoria. Confrontando le prestazioni della stessa matrice nei formati HLL e HLL_Aligned, si osserva che quest'ultimo garantisce prestazioni almeno equivalenti a HLL anche nelle situazioni meno favorevoli, mentre nella maggior parte dei casi apporta un miglioramento significativo. Questo beneficio deriva da un accesso più efficiente alla memoria globale, favorito da un allineamento che consente operazioni più coalescenti (ossia più contigue) da parte dei thread, riducendo la latenza e aumentando il throughput. Tali vantaggi risultano evidenti anche nelle rappresentazioni grafiche dei risultati.

Le migliori prestazioni delle soluzioni basate su HLL si ottengono con una `blockSize` compresa tra 192 e 320 thread. In questo intervallo si raggiunge un buon equilibrio tra parallelismo e uso efficiente delle risorse hardware della GPU, garantendo elevata occupancy senza saturare troppo registri e memoria condivisa. Ciò massimizza l'efficienza dell'accesso alla memoria e l'esecuzione parallela del kernel, riducendo sensibilmente i tempi di esecuzione rispetto a configurazioni con `blockSize` più basse o più elevate.

VIII. Conclusioni

Questo progetto si è proposto di analizzare e confrontare le prestazioni di diverse implementazioni parallele del prodotto matrice-vettore sparso (SpMV), esplorando l'interazione tra architetture hardware (CPU multi-core e GPU), paradigmi di programmazione (OpenMP e CUDA) e formati di rappresentazione dei dati (CSR e HLL/HLL Aligned).

Dall'analisi critica delle prestazioni sono emersi alcuni punti chiave fondamentali:

a) L'Impatto Critico della Rappresentazione dei Dati

La scoperta più significativa di questo studio è il ruolo predominante giocato dal formato di memorizzazione della matrice. Sia su CPU che su GPU, le prestazioni sono state fortemente influenzate dalla località e dalla regolarità degli accessi alla memoria. Il formato CSR si è dimostrato quasi sempre superiore grazie alla sua natura compatta e contigua, che favorisce un utilizzo efficiente della cache su CPU. Al contrario, il formato HLL, pur offrendo un modello di parallelismo a blocchi, ha sofferto di accessi meno regolari e di un maggiore overhead di memoria, con conseguenti prestazioni inferiori. La netta superiorità della variante HLL Aligned rispetto a quella base, specialmente in ambiente CUDA, ha confermato in modo inequivocabile che l'allineamento dei dati e la promozione di accessi coalescenti sono prerequisiti indispensabili per massimizzare il throughput su architetture GPU.

b) Dinamiche di Parallelismo su CPU (OpenMP)

L'analisi su OpenMP ha confermato l'efficacia del parallelismo su core fisici, mostrando una scalabilità quasi lineare delle prestazioni. Tuttavia, l'attivazione dell'hyperthreading ha introdotto benefici marginali o nulli, a causa dell'overhead di sincronizzazione e della maggiore contesa per le risorse di memoria. Inoltre, è emerso che non esiste una singola strategia di parallelizzazione ottimale: soluzioni basate su direttive SIMD (sol3 per CSR) o su una gestione esplicita del carico (`schedule(dynamic)` o `ThreadDataRange` per HLL) si sono rivelate vincenti a seconda del formato e del bilanciamento del carico.

c) Sensibilità dell'Architettura GPU (CUDA)

A differenza di OpenMP, l'ambiente CUDA ha rivelato un comportamento più sensibile alla struttura intrinseca della matrice. Per il formato CSR, non è stato possibile identificare una soluzione universalmente migliore: l'approccio "un thread per riga" è risultato efficiente per matrici con righe bilanciate, mentre quello "un warp per riga" ha prevalso in presenza di righe lunghe, evidenziando un trade-off cruciale tra semplicità e adattabilità. La scelta della `blockSize` si è rivelata un fattore di tuning essenziale, con un intervallo ottimale (192–320) che bilancia *occupancy* e utilizzo delle risorse.

Sintesi Finale

In conclusione, questo progetto ha dimostrato che la massimizzazione delle prestazioni nel calcolo sparso non dipende da un singolo fattore, ma da una profonda sinergia tra algoritmo, formato dati e architettura hardware.

adder_dcop_32

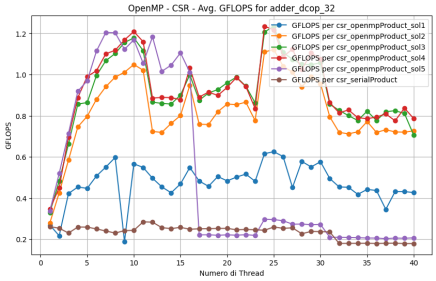

Figura 4: GFLOPS OpenMP_CSR

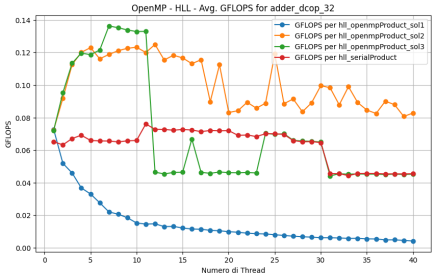

Figura 6: GFLOPS OpenMP_HLL

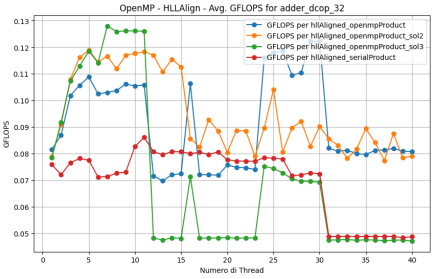

Figura 8: GFLOPS OpenMP_HLLAlign

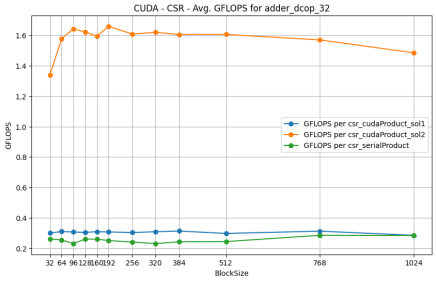

Figura 5: GFLOPS CUDA_CSR

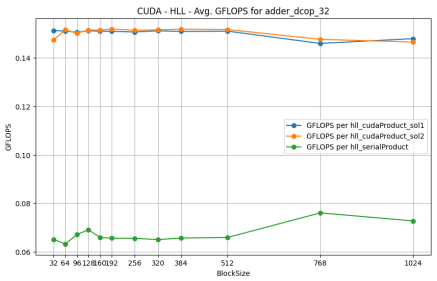

Figura 7: GFLOPS CUDA_HLL

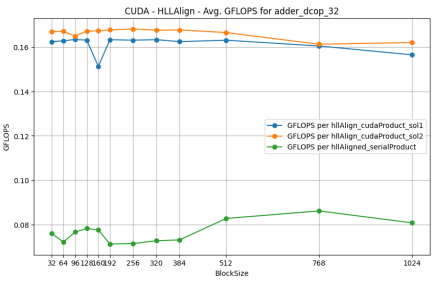

Figura 9: GFLOPS CUDA_HLLAlign

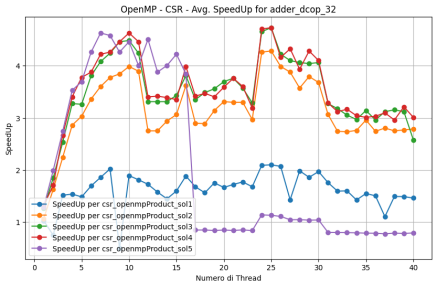

Figura 10: SPEEDUP OpenMP_CSR

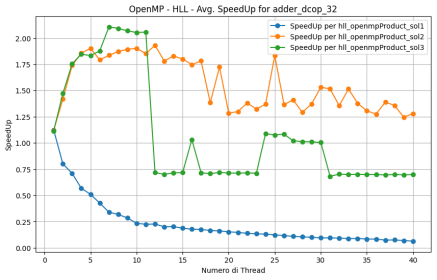

Figura 12: SPEEDUP OpenMP_HLL

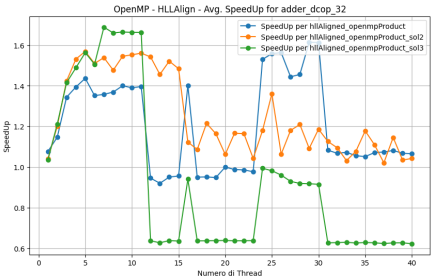

Figura 14: SPEEDUP OpenMP_HLLAlign

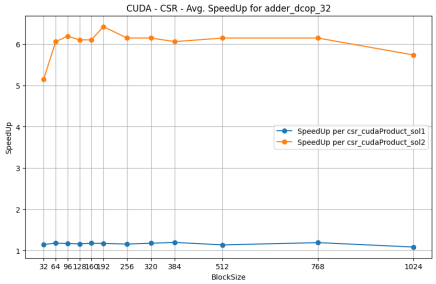

Figura 11: SPEEDUP CUDA_CSR

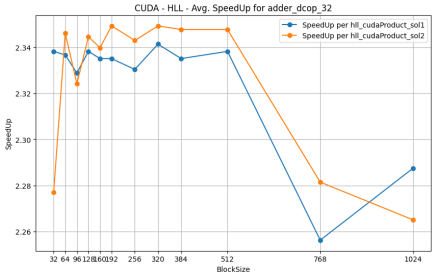

Figura 13: SPEEDUP CUDA_HLL

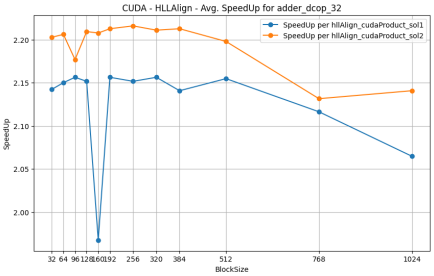

Figura 15: SPEEDUP CUDA_HLLAlign

af23560

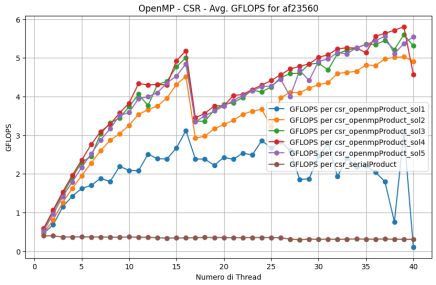

Figura 16: GFLOPS OpenMP_CSR

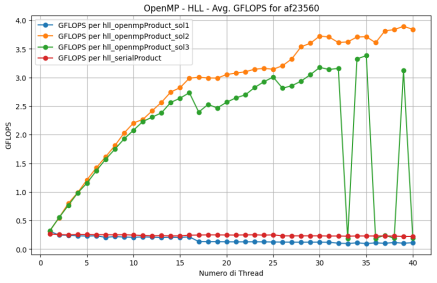

Figura 18: GFLOPS OpenMP_HLL

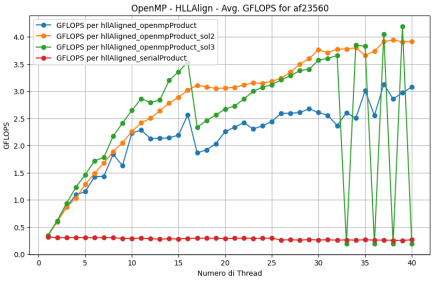

Figura 20: GFLOPS OpenMP_HLLAlign

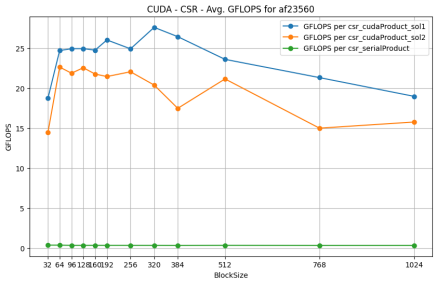

Figura 17: GFLOPS CUDA_CSR

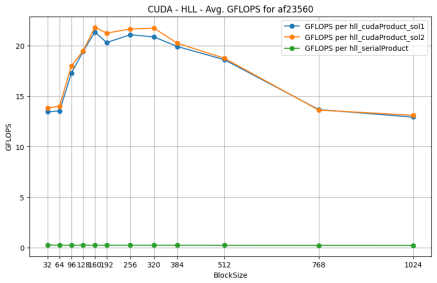

Figura 19: GFLOPS CUDA_HLL

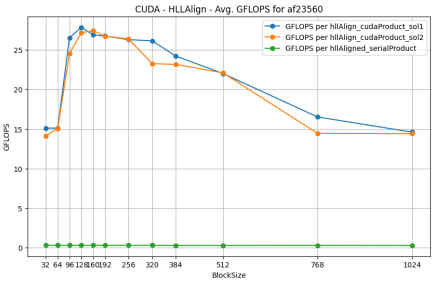

Figura 21: GFLOPS CUDA_HLLAlign

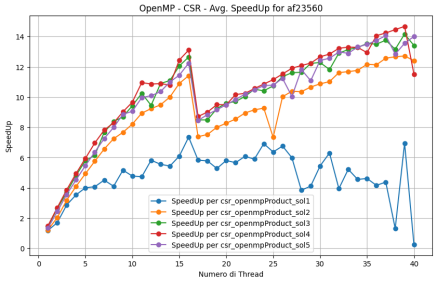

Figura 22: SPEEDUP OpenMP_CSR

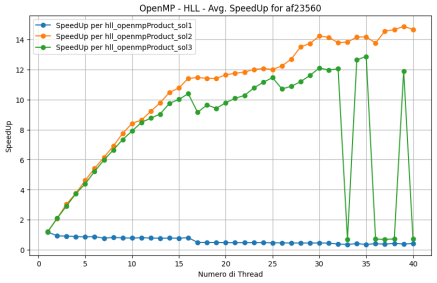

Figura 24: SPEEDUP OpenMP_HLL

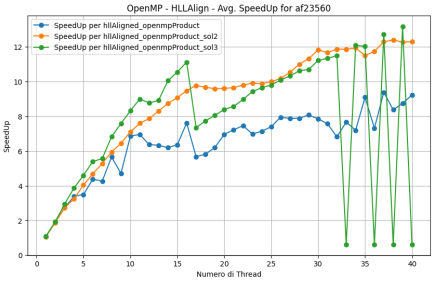

Figura 26: SPEEDUP OpenMP_HLLAlign

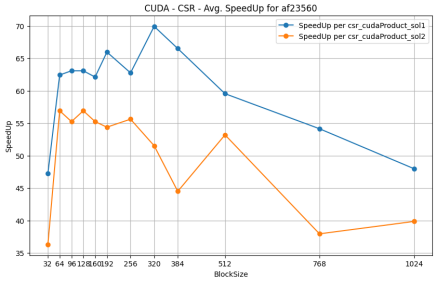

Figura 23: SPEEDUP CUDA_CSR

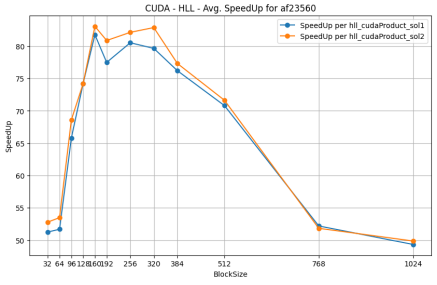

Figura 25: SPEEDUP CUDA_HLL

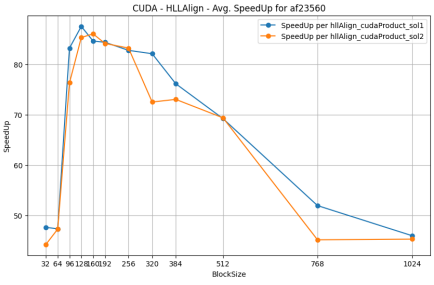

Figura 27: SPEEDUP CUDA_HLLAlign

west2021

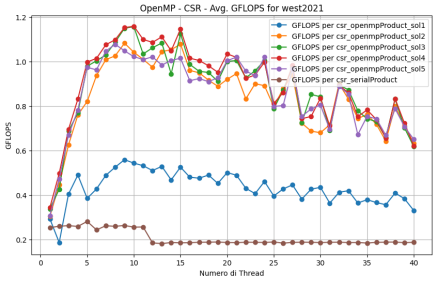

Figura 28: GFLOPS OpenMP_CSR

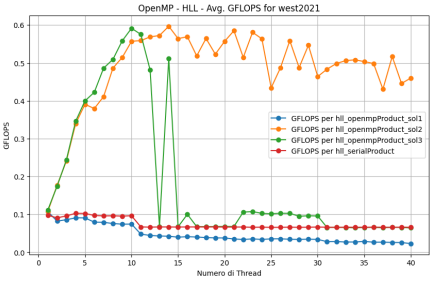

Figura 30: GFLOPS OpenMP_HLL

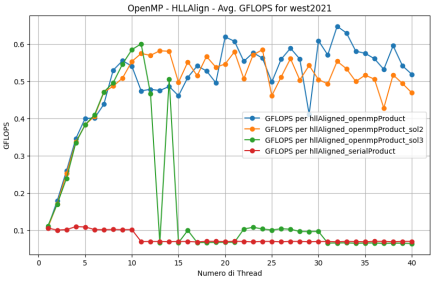

Figura 32: GFLOPS OpenMP_HLLAlign

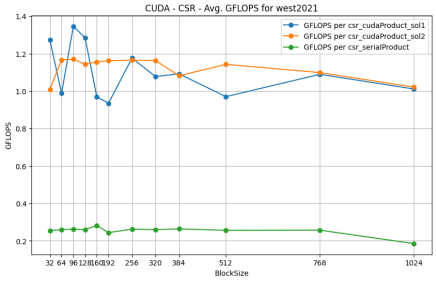

Figura 29: GFLOPS CUDA_CSR

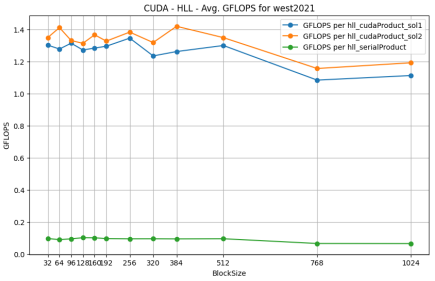

Figura 31: GFLOPS CUDA_HLL

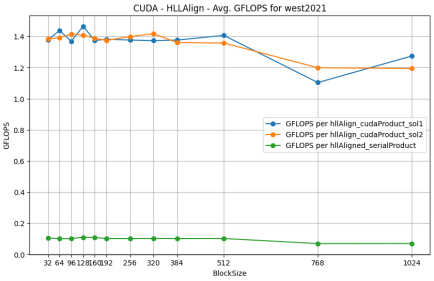

Figura 33: GFLOPS CUDA_HLLAlign

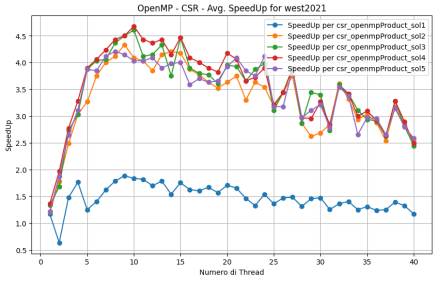

Figura 34: SPEEDUP OpenMP_CSR

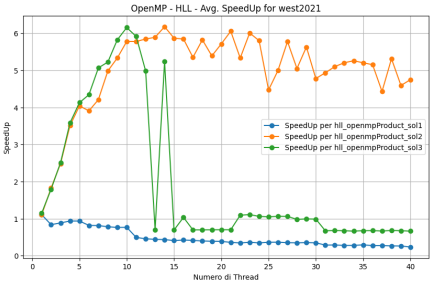

Figura 36: SPEEDUP OpenMP_HLL

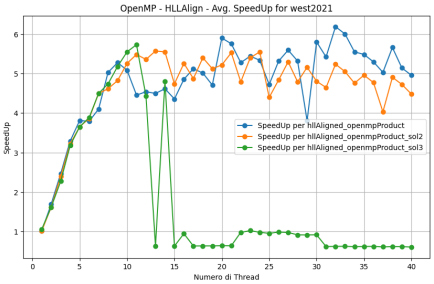

Figura 38: SPEEDUP OpenMP_HLLAlign

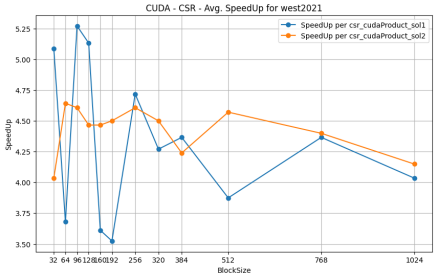

Figura 35: SPEEDUP CUDA_CSR

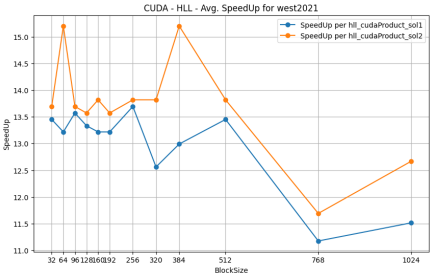

Figura 37: SPEEDUP CUDA_HLL

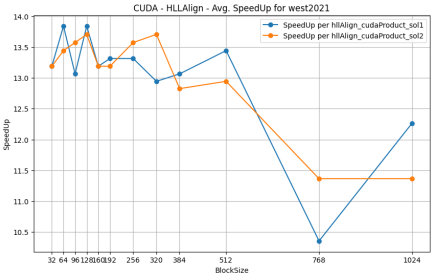

Figura 39: SPEEDUP CUDA_HLLAlign