Parallel Sparse Matrix-Vector Product

Andrea Andreoli
Facoltà di Ingegneria Informatica
Università degli Studi di Roma Tor Vergata
Roma, Italia
aandreo.2001@gmail.com

Pierfrancesco Lijoi
Facoltà di Ingegneria Informatica
Università degli Studi di Roma Tor Vergata
Roma, Italia
pierfrancesco.lijoi@libero.it

***Sommario*—In questo documento vengono descritte le fasi di sviluppo di un nucleo di calcolo parallelo per il prodotto tra una matrice sparsa e un vettore, formalmente definito come:**

$$y \leftarrow Ax$$

dove la matrice sparsa A è rappresentata utilizzando i seguenti formati di memorizzazione:

- **CSR (Compressed Sparse Row)**
- **HLL (Hybrid Linked List)**

Il parallelismo è stato implementato utilizzando i framework OpenMP e CUDA, sfruttando rispettivamente le capacità di calcolo parallelo delle CPU multicore e delle GPU. Vengono analizzate le tecniche adottate per affrontare le sfide principali, tra cui la gestione della memoria, la sincronizzazione e l'ottimizzazione degli accessi ai dati.

***Index Terms*—prodotto matrice-vettore, matrice sparsa, calcolo parallelo, vettore, CUDA, OpenMP**

I. Introduzione

Il prodotto tra matrice sparsa e vettore (*Sparse Matrix-Vector Multiplication*, SpMV) rappresenta un'operazione fondamentale in numerose applicazioni scientifiche e ingegneristiche, come:

- La risoluzione di sistemi lineari
- Metodi iterativi
- Simulazioni numeriche e analisi di grafi

L'importanza di questa operazione deriva dal fatto che molte delle matrici che emergono in problemi reali sono **sparse**, ovvero caratterizzate da un gran numero di elementi nulli, il che consente di ottimizzare lo spazio di memoria e il tempo computazionale sfruttando apposite strutture dati.

Nonostante la relativa semplicità concettuale dell'operazione SpMV, la sua implementazione efficiente pone diverse sfide. Le principali difficoltà risiedono nella gestione della memoria e nella natura irregolare degli accessi ai dati. In questo contesto, il parallelismo gioca un ruolo cruciale nel migliorare le prestazioni di SpMV. La suddivisione del lavoro computazionale su più processori, thread o unità computazionali consente di accelerare il calcolo sfruttando le capacità delle architetture parallele, come CPU multicore e GPU.

L'operazione SpMV può essere analizzata in termini di due fasi principali:

1) Il recupero degli elementi non nulli dalla matrice sparsa con conseguente memorizzazione in formato CSR e HLL
2) Il calcolo dei prodotti scalari tra questi elementi e i corrispondenti elementi del vettore

Nel seguito saranno analizzate diverse tecniche sviluppate per realizzare la parallelizzazione dell'operazione SpMV.

II. Preprocessamento dei dati

Il collaudo del nucleo di calcolo è stato effettuato utilizzando una serie di matrici sparse, ciascuna memorizzata in un file *.mtx* disponibile sul sito **Matrix Market**. Per gestire questi file, è stata implementata una funzione dedicata al preprocessamento delle matrici, con l'obiettivo di analizzarne le caratteristiche e prepararle per le successive operazioni computazionali.

La funzione effettua, in primo luogo, la lettura del file per determinare eventuali proprietà particolari della matrice, come la presenza di una struttura a pattern (matrice senza valori espliciti, rappresentata solo dalla posizione degli elementi non nulli) o la simmetria, sia superiore che inferiore. Una volta identificate queste caratteristiche, vengono estratti i dati in formato tripla (riga, colonna, valore), i quali rappresentano la posizione e il valore di ciascun elemento non nullo della matrice.

Questi dati vengono infine memorizzati in una struttura dati che conserva la matrice in una forma "grezza", direttamente derivata dalle informazioni lette dal file. Questa rappresentazione preliminare funge da punto di partenza per la successiva conversione nei formati di memorizzazione ottimizzati, come CSR o HLL.

III. Rappresentazione in memoria delle matrici sparse

La rappresentazione efficiente di matrici sparse è fondamentale per ridurre il consumo di memoria e ottimizzare le operazioni computazionali, come appunto il prodotto matrice-vettore. In particolare si pone l'attenzione su due formati specifici:

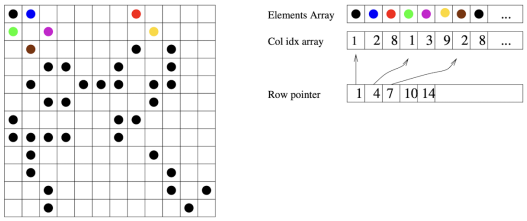
- CSR
- HLL

Entrambi i formati offrono vantaggi specifici che li rendono adatti a diverse architetture e applicazioni, e la scelta tra CSR e HLL dipende tipicamente dalle caratteristiche della matrice e dagli obiettivi di ottimizzazione dell'algoritmo.

A. CSR

Il formato di memorizzazione **CSR** (*Compressed Sparse Row*) è uno dei più utilizzati per rappresentare matrici sparse in modo efficiente, sia in termini di spazio che di operazioni computazionali. In questo formato, la matrice è rappresentata tramite tre vettori principali:

- **AS** che contiene i valori non nulli della matrice
- **JA** che registra le colonne corrispondenti a ciascun valore in AS
- **IRP** che indica gli indici in AS dove inizia ogni riga della matrice



L'operazione di SpMV è realizzata in questo modo:

Algoritmo 1 Prodotto matrice sparsa-vettore con formato CSR

for $i = 1 : m$ **do**
  $t = 0$;
  for $j = \text{irp}(i) : \text{irp}(i+1) - 1$ **do**
    $t = t + \text{as}(j) * \text{x}(\text{ja}(j))$;
  end for
  $\text{y}(i) = t$;
end for

B. HLL

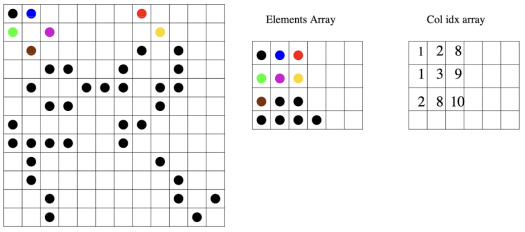
Il formato di memorizzazione **HLL** (Hybrid Link List) si può definire formalmente in questo modo:

- Si stabilisce un parametro HackSize (nel caso di studio vale 32)
- Si partiziona la matrice di input in blocchi di HackSize righe
- Ciascun blocco viene memorizzato in formato **ELLPACK**
- I blocchi vengono memorizzati in una struttura dati da determinare

1) ELLPACK: La matrice viene rappresentata tramite due vettori 2D:

- **JA** che registra le colonne corrispondenti a ciascun valore in AS
- **AS** che contiene i coefficienti della matrice

Notare che le righe che abbiano un numero di nonzeri inferiore a *MAXNZ* (numero massimo di nonzeri per riga all'interno di un blocco) devono essere riempite con coefficienti appropriati, ossia zeri in AS. L'operazione



di SpMV è realizzata in questo modo:

Algoritmo 2 Prodotto matrice sparsa-vettore con formato ELLPACK

for $i = 1 : m$ **do**
  $t = 0$;
  for $j = 1 : maxnzr$ **do**
    $t = t + \text{as}(i, j) * \text{x}(\text{ja}(i, j))$;
  end for
  $\text{y}(i) = t$;
end for

IV. OpenMP

A. Introduzione

OpenMP (Open Multi-Processing) è un'API multi-piattaforma per la programmazione parallela su sistemi a memoria condivisa. È supportata da diversi linguaggi di programmazione, tra cui C, C++ e Fortran. Nel nostro caso di studio, è stato scelto di utilizzare il linguaggio C per implementare la parallelizzazione del prodotto matrice-vettore, distribuendo il carico computazionale su un insieme di t thread.

L'obiettivo principale della parallelizzazione è stato quello di bilanciare equamente il carico di lavoro tra i thread, massimizzando l'efficienza del calcolo. A tal fine, la strategia di suddivisione del lavoro è stata adattata in base al formato di memorizzazione della matrice sparsa, ottimizzando l'assegnazione delle operazioni aritmetiche tra i thread disponibili.

Un aspetto fondamentale del nostro approccio è stato l'analisi delle prestazioni del nucleo di calcolo in funzione del numero di thread impiegati. Le iterazioni sono state eseguite con un numero variabile di thread, scelto secondo le potenze di due (2, 4, 8, 16, 32, ...), fino a raggiungere il numero massimo di thread supportati dal processore. Nel caso del server del dipartimento, questo valore è pari a 40 thread.

Questa metodologia ha permesso di ottenere una visione dettagliata dell'andamento delle performance, fornendo dati utili per comprendere come il parallelismo influenzi l'efficienza del calcolo.

B. Parallelizzazione con formato CSR

Per eseguire il calcolo parallelo dell'operazione SpMV sfruttando la memorizzazione CSR della matrice, è stato necessario effettuare una suddivisione del carico di lavoro tra i thread disponibili nell'esecuzione corrente. La strategia di distribuzione del carico si è basata su due principi fondamentali:

- Ogni thread deve avere lo stesso carico di lavoro, ovvero il numero di nonzeri su cui eseguire le operazioni in virgola mobile deve essere il più equilibrato possibile tra tutti i thread.
- I thread devono accedere a righe adiacenti della matrice, per sfruttare la località spaziale della memoria e ridurre il tempo di accesso alle righe non adiacenti.

Questi principi sono mirati a minimizzare l'overhead introdotto dalla gestione dei thread e dagli accessi alla memoria. Una volta memorizzata la matrice nel formato CSR è stato necessario implementare una distrubuzione del carico efficiente. L'implementazione della suddivisione uniforme delle operazioni in virgola mobile, è stata basata sull'assunzione che gli elementi non nulli, fossero equamente distribuiti tra le righe della matrice. A partire da questa condizione ideale, la distribuzione delle righe è stata adattata in modo che ogni thread ricevesse un sottoinsieme di righe che rispettasse tre proprietà:

- Ogni thread deve essere assegnato solo a un sottoinsieme di righe non vuoto, ovvero che sia presente almeno una elemento. Quando un thread veniva assegnato a un sottoinsieme vuoto, questo veniva rimosso e il numero di thread utilizzati veniva aggiornato di conseguenza;
- Ogni thread deve gestire un sottoinsieme di righe tale che il numero di elementi non nulli nel sottoinsieme di righe non fosse vuoto. Se il sottoinsieme di righe di un thread non rispettava questa condizione, il carico di lavoro veniva redistribuito e il thread veniva eliminato aggiornando di conseguenza il numero di thread utilizzati;
- Ogni thread deve essere assegnato a un sottoinsieme di righe adiacenti, e il numero di nonzeri nel sottoinsieme assegnato deve essere il più vicino possibile a quello degli altri sottoinsiemi assegnati agli altri thread. Se necessario, veniva eseguita una nuova combinazione di righe adiacenti, mantenendo sempre le condizioni precedenti

In tal modo nel caso in cui la condizione ideale non si verifica, viene ottenuto una condizione di bilanciamento artificiale che permette l'uniforme carico di lavoro per ogni thread. L'implementazione della distribuzione uniforme del carico è descritto nell'Algoritmo 3.

Algoritmo 3 Partizionamento delle righe per il calcolo parallelo con formato CSR

Input: M, nz, $num\_threads$, IRP
Output: $start\_row$, end_row

Inizializzazione:
    Allocare memoria per $start\_row$, end_row e $nnz\_per\_thread\_count$

Distribuzione del carico:
for ogni riga i della matrice **do**
    Assegnare la riga al thread corrente, bilanciando il numero di nonzeri
    if la quota di nonzeri per il thread corrente è raggiunta **then**
        Passare al prossimo thread
    end if
end for

Gestione della fine delle partizioni:
if ultimo thread non ha righe assegnate **then**
    Assegnare le righe rimanenti al thread finale
end if

Rimozione dei thread non validi:
for ogni thread t **do**
    if il thread ha righe valide **then**
        Mantenere il thread
    end if
end for

Deallocazione memoria:
Liberare la memoria per $nnz\_per\_thread\_count$

Successivamente, è stata sfruttata questa suddivisione uniforme del carico di lavoro nel calcolo parallelo, facendo iterare ogni thread sul sottoinsieme di righe assegnatogli, utilizzando i dati contenuti nei vettori CSR. In questo modo, il carico di lavoro è stato distribuito in modo bilanciato tra i thread così da effettuare tutti lo stesso numero di operazioni in virgola mobile.

Il flusso di calcolo parallelo viene descritto dall'Algoritmo 4.

Algoritmo 4 Calcolo parallelo del prodotto matrice-vettore con formato CSR

1: **Input:** IRP, JA, AS, x, y, $start\_row$, end_row, $num\_threads$
2: **Output:** y
3:
4: **Inizializzazione:** Impostare il numero di thread
5:
6: **Calcolo parallelo:**
7: **for** ogni thread t in parallelo **do**
8: **for** $i = start\_row[t]$ to $end\_row[t]$ **do**
9: Inizializzare $y[i]$
10: **for** ogni elemento non nullo nella riga i **do**
11: Aggiungere il prodotto del valore non nullo e l'elemento del vettore x
12: **end for**
13: **end for**
14: **end for**

C. Parallelizzazione con formato HLL

Per eseguire il calcolo parallelo dell'operazione SpMV sfruttando la memorizzazione HLL della matrice, è stata utilizzata una strategia simile. La distribuzione del carico si è basata su due principi fondamentali:

- Ogni thread deve eseguire un carico di lavoro equilibrato, con un numero simile di nonzeri tra i vari thread
- I thread devono accedere a blocchi adiacenti della matrice, per ottimizzare la località spaziale della memoria e ridurre i tempi di accesso

Questi principi mirano a minimizzare l'overhead legato alla gestione dei thread e agli accessi alla memoria. Una volta memorizzata la matrice nel formato HLL è stato necessario implementare una distrubuzione del carico efficiente. L'implementazione della distribuzione del carico è stata basata sull'assunzione che gli elementi non nulli, fossero equamente distribuiti tra le righe della matrice. Quindi idealmente di riflesso equamente distribuiti anche nei blocchi ELLPACK in cui è stata memorizzata la matrice nel formato HLL. A partire da questa condizione ideale, la distribuzione del carico è stata adattata in modo che ogni thread ricevesse un sottoinsieme di blocchi, seguendo le seguenti proprietà:

- Ogni thread deve essere assegnato solo a un sottoinsieme di blocchi non vuoto, ovvero che sia presente almeno una elemento. Quando un thread viene assegnato a un sottoinsieme vuoto, questo veniva rimosso e il numero di thread utilizzati veniva aggiornato di conseguenza;
- Ogni thread deve gestire un sottoinsieme di blocchi tale che il numero di elementi non nulli contenuto nei blocchi del sottoinsieme non fosse nullo. Se il sottoinsieme di un thread non rispettava questa condizione, il carico di lavoro veniva redistribuito e il thread veniva eliminato aggiornando di conseguenza il numero di thread utilizzati;
- Ogni thread deve essere assegnato a un sottoinsieme di blocchi adiacenti, e il numero di elementi non nulli contenuto nei blocchi del sottoinsieme assegnato deve essere il più vicino possibile a quello assegnati agli altri thread. Se necessario, veniva eseguita una nuova combinazione di righe adiacenti, mantenendo sempre le condizioni precedenti

In tal modo nel caso in cui la condizione ideale non si verifica, viene ottenuto una condizione di bilanciamento artificiale che permette di avere un carico di lavoro uniforme per ogni thread. L'implementazione della distribuzione uniforme del carico è descritta nell'Algoritmo 5.

Algoritmo 5 Distribuzione dei blocchi ai thread con formato HLL

1: **Input:** Numero di thread, matrice sparsa
2: **Output:** Inizio e fine dei blocchi per ogni thread, numero di thread validi
3:
4: **Inizializzazione:** Preparare la struttura per i blocchi assegnati ai thread
5:
6: **Distribuzione dei blocchi:**
7: **for** ogni blocco **do**
8: **if** il blocco corrente può essere assegnato al thread corrente **then**
9: Assegnare il blocco al thread
10: **else**
11: Redistribuire il blocco a un thread precedente o successivo
12: **end if**
13: **end for**
14:
15: **Gestione dei thread:**
16: **for** ogni thread **do**
17: **if** il thread non ha blocchi validi **then**
18: Rimuovere il thread dal conteggio finale
19: **end if**
20: **end for**
21:
22: **Verifica finale:** Controllare che tutti i blocchi siano stati assegnati correttamente
23:
24: **Deallocazione:** Liberare la memoria temporanea

Successivamente, è stato sfruttata la distribuzione dei blocchi ai thread, nel calcolo parallelo. Ogni thread è stato incaricato di eseguire il calcolo del prodotto matrice-vettore sul sottoinsieme assegnato, utilizzando i dati contenuti nei blocchi Ellpack usati nella meoriz-

zazione nel formato HLL. In questo modo, il carico di lavoro è stato distribuito in modo bilanciato tra i thread così da effettuare tutti lo stesso numero di operazioni in virgola mobile.

Il flusso di calcolo parallelo, dove ogni thread opera sul proprio sottoinsieme di blocchi, è descritto dall'Algoritmo 6.

Algoritmo 6 Calcolo parallelo del prodotto matrice-vettore con formato HLL

Input: Matr. HLL, vettore x, vettore y, numero di thread, inizio e fine dei blocchi per ogni thread
Output: Vettore y

Inizializzazione: Impostare il numero di thread

Calcolo parallelo:
for ogni thread t in parallelo **do**
    for ogni blocco assegnato al thread t **do**
        Calcolare l'intervallo di righe per il blocco
        for ogni riga nel blocco **do**
            Inizializzare il risultato $y[i]$
            for ogni elemento non nullo nella riga **do**
                Aggiungere il prodotto del valore non nullo e l'elemento del vettore x
            end for
        end for
    end for
end for

V. CUDA

A. Introduzione

CUDA (*Compute Unified Device Architecture*) è un framework sviluppato da NVIDIA per l'elaborazione parallela su GPU. Consente agli sviluppatori di sfruttare la potenza di calcolo massiva delle GPU per accelerare applicazioni computazionalmente intensive, in particolare, saranno sfruttate le alte prestazioni per parallelizzare l'operazione SpMV.

L'architettura **CUDA** si basa su un'organizzazione **gerarchica** dei **thread**, progettata per massimizzare l'efficienza nell'elaborazione parallela. I **thread** sono raggruppati in **blocchi**, che a loro volta sono organizzati in una **griglia** . Ogni **thread** e ogni **blocco** sono identificati da indici univoci, che possono essere utilizzati per accedere in modo indipendente ai dati.

L'organizzazione dei thread in blocchi, e nella relativa griglia, può essere **multidimensionale**: i **thread** all'interno di un **blocco** e i **blocchi** all'interno della **griglia** possono essere organizzati in **una, due o tre dimensioni**.In particolare:

- Se la **griglia** o il **blocco** sono **monodimensionali**, si avrà un solo indice (**x**) per identificare un thread o un blocco.
- Se la **griglia** o il **blocco** sono **bidimensionali**, si avranno due indici (**x** e **y**) per l'indirizzamento.
- Se la **griglia** o il **blocco** sono **tridimensionali**, si utilizzeranno tre indici (**x**, **y** e **z**) per identificare un thread o un blocco.

Come rappresentato anche dalla seguente immagine:

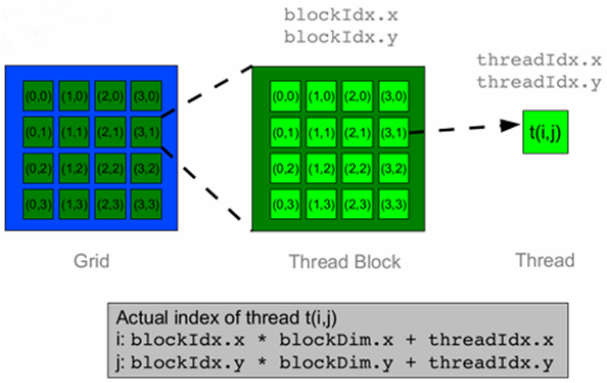


Figura 1: Identificazione univoca nella gerarchia dei thread

La scelta della dimensione della griglia in CUDA influisce direttamente sulle prestazioni dell'algoritmo parallelo, poiché determina come i blocchi di thread vengono assegnati ai multiprocessori della GPU. Una configurazione non ottimale può portare a spreco di risorse, bassa occupazione e sotto-utilizzo della GPU.

Inoltre è importante sottolineare che nell'architettura NVIDIA un gruppo di 32 thread consecutivi appartenenti a un blocco, che vengono eseguiti simultaneamente da uno Streaming Multiprocessor (unità computazionale principale all'interno della GPU) viene detto **warp**.

Sulla base di queste considerazioni, è necessario determinare la dimensione ed il numero dei blocco di thread tenendo conto di tre fattori principali:

- **Grandezza del blocco**: la dimensione del blocco deve essere un multiplo di 32, poiché i thread all'interno di un warp vengono eseguiti simultaneamente. Un blocco non allineato alla dimensione del warp può causare un uso inefficiente delle risorse hardware, riducendo il parallelismo effettivo e le prestazioni
- **Capacità limite della GPU**: la dimensione massima del blocco è limitata dalla capacità della GPU in uso. Nel caso della GPU del server del dipartimento su cui sono stati effettuati i test, la dimensione massima del blocco è pari a 1024 thread
- **Numero dei blocchi**:Per ottenere un bilanciamento del carico in CUDA, non basta semplicemente assegnare lo stesso numero di operazioni a ciascun blocco di thread. È altrettanto importante che il numero totale di blocchi sia superiore al numero

di Streaming Multiprocessors (SM) e, nel caso ideale, un multiplo esatto di esso.

Per gli esperimenti effettuati sul nucleo di calcolo parallelo è stato considerato anche un altro fattore: le funzioni eseguite sulla GPU, chiamate **kernel**, utilizzano due tipi di memoria principali. La **memoria globale**, caratterizzata da tempi di latenza di accesso molto elevati, e la **memoria condivisa**, che presenta tempi di latenza inferiori ed è condivisa tra i thread dello stesso blocco. Tuttavia, l'utilizzo della memoria condivisa richiede una sincronizzazione esplicita tra i thread durante le operazioni di scrittura, il che può introdurre ulteriori rallentamenti nel calcolo parallelo. Di conseguenza, non è possibile stabilire a priori quale delle due tecnologie sia più efficiente. Questo aspetto verrà approfondito attraverso sperimentazioni mirate, che saranno oggetto di discussione nei prossimi paragrafi.

B. Parallelizzazione con formato CSR

Per la parallelizzazione dell'operazione SpMV, con la matrice sparsa memorizzata in formato CSR, sono state sviluppate tre versioni:

1) Prima versione senza shared memory: Questa prima versione, pur essendo semplice, consente di ottenere buone prestazioni per l'esecuzione dell'operazione SpMV. L'implementazione utilizza una griglia unidimensionale di blocchi, la cui dimensione è calcolata secondo la seguente formula:

$$grid\_dim = \frac{M + BD - 1}{BD}$$

dove BD rappresenta la dimensione del blocco monodimensionale. Questa formula è stata scelta per tenere conto degli eventuali arrotondamenti necessari in caso di divisioni non esatte tra il numero totale di righe della matrice M e la dimensione del blocco BD.

Anche i blocchi sono organizzati in una sola dimensione, a sottolineare la semplicità dell'approccio. Per questa implementazione, si è scelto un valore di:

$$BD = 1024$$

In questa configurazione, ogni thread è responsabile del calcolo relativo a una riga della matrice. Per assegnare a ciascun thread la riga su cui operare, viene calcolato un indice globale che identifica in modo univoco la riga su cui il thread lavorerà. Questo indice è ottenuto tramite la seguente formula:

$$\text{row} = \text{blockIdx.x} \cdot \text{blockDim.x} + \text{threadIdx.x}$$

Dopo aver identificato la riga, il thread esegue il prodotto scalare tra i valori non nulli della riga e il corrispondente vettore $\boldsymbol{x}$, memorizzando il risultato nella posizione corretta del vettore $\boldsymbol{y}$.

2) Seconda versione con shared memory: L'obiettivo di questa seconda versione è suddividere il carico computazionale tra i thread in modo più dettagliato, assegnando ciascuna riga della matrice a un intero blocco bidimensionale di thread. La griglia è unidimensionale e ha dimensione:

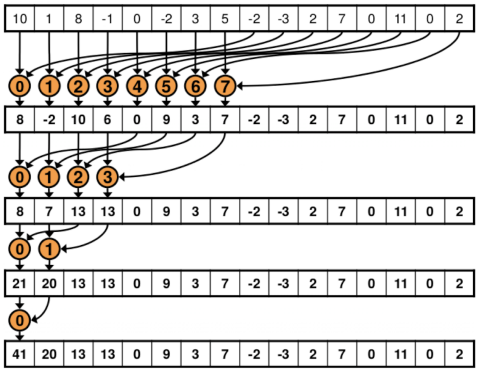


Figura 2: Sequential addressing

$$grid\_dim = M$$

ossia pari al numero di righe della matrice. In questo modo, ogni blocco elabora una singola riga della matrice.

Per quanto riguarda la dimensione del blocco, essa deve rispettare i limiti imposti dalla GPU utilizzata, che nel caso del server di dipartimento è pari a 1024 thread per blocco. Pertanto, le dimensioni del blocco devono rispettare il seguente vincolo:

$$BDX \cdot BDY \leq 1024$$

L'idea alla base di questa implementazione è sfruttare la shared memory per migliorare l'efficienza computazionale. In particolare, ogni blocco di thread utilizza la memoria condivisa per calcolare il prodotto scalare tra una riga della matrice e il vettore $\boldsymbol{x}$, riducendo il numero di accessi alla memoria globale. Nella shared memory, ciascun thread memorizza il risultato parziale derivante dal prodotto scalare della componente di interesse della riga associata al blocco. Nel caso in cui il numero di elementi non nulli per riga sia maggiore di 1024 (cioè superiore al numero massimo di thread supportati per blocco), una singola locazione della shared memory sarà utilizzata per accumulare i risultati di più componenti, elaborate dallo stesso thread mediante uno schema a stride.

Dopo che tutti i thread avranno completato il calcolo del prodotto per le rispettive componenti della riga, viene eseguita una **riduzione** per sommare i risultati parziali e calcolare il valore totale della componente corrispondente alla riga, identificata dall'indice del blocco (blockIdx.x). In questa implementazione, la riduzione viene effettuata in modo sequenziale, come illustrato in Figura 2.

3) Terza versione con shared memory: Questa terza versione segue lo stesso principio della seconda, ma

per migliorarne ulteriormente le prestazioni, si è passati da una riduzione sequenziale a una riduzione parallela all'interno della memoria condivisa.

C. Parallelizzazione con il formato HLL

Nell'ambito dell'operazione di moltiplicazione matrice-vettore (SpMV) su matrici sparse memorizzate in formato HLL, sono state sviluppate due implementazioni: una che non fa uso della memoria condivisa e una che, invece, la impiega.

Per ciascuna di queste versioni è stata progettata una specifica struttura della griglia di blocchi, configurata dinamicamente in base alla dimensione della matrice considerata.

In tal modo si è mirato a distribuire in modo ottimale il carico di calcolo e a massimizzare le prestazioni del calcolo parallelo.

Si evidenzia la rilevanza fondamentale della dimensione della griglia di blocchi di thread nel calcolo parallelo su GPU, in quanto determina la corretta distribuzione del carico di lavoro e incide in modo significativo sull'efficienza complessiva delle prestazioni.

Inoltre, entrambi le versione hanno in comune la conversione della matrice nel formato HLL prima di eseguire le operazioni per la configurazione dinamica della griglia di thread e le operazioni parallele.

1) Prima versione senza shared memory: In questa versione, la griglia è monodimensionale e ciascun blocco contiene un numero di thread pari alla dimensione di un warp (32 thread), allineandosi così al modello di esecuzione nativo delle GPU NVIDIA. Sebbene un blocco non debba necessariamente avere un numero di thread pari a un multiplo di 32, è consigliabile per massimizzare l'efficienza ed evitare sprechi di risorse.

Inoltre, il numero di blocchi viene impostato in modo da essere multiplo del numero di *Streaming Multiprocessors*, garantendo una distribuzione omogenea del carico di lavoro e un utilizzo ottimale delle risorse. L'implementazione della configurazione della griglia monodimenisonale è descritta dal seguente algoritmo:

Algoritmo 7 Configurazione della griglia monodimensionale

```
1: procedure CONFIGUREGRID(M, SM_count)
2:     Calcola il numero totale di thread, allineato alla dimensione di un warp
3:     Imposta il numero di thread per blocco uguale alla dimensione di un warp
4:     Calcola il numero di blocchi necessari per coprire tutti i thread
5:     if il numero di blocchi non è multiplo di SM_count then
6:         Aggiusta il numero di blocchi affinché sia multiplo di SM_count
7:     end if
8: end procedure
```

Basandosi su una **griglia monodimensionale**, ogni thread è responsabile del calcolo di una singola riga della matrice sparsa. Il calcolo avviene eseguendo il **prodotto scalare** tra gli elementi non zero della riga e i corrispondenti valori del vettore di input d_x, memorizzando il risultato nel vettore di output d_y.

La matrice è memorizzata nel formato **HLL (Hybrid Linked List)**, che combina **ELLPACK** per una gestione efficiente degli elementi non zero e una struttura a liste collegate per gestire la variabilità della sparsità. La matrice è suddivisa in **blocchi ELLPACK**, e ciascun **blocco CUDA** si occupa di un sottoinsieme di righe della matrice.

All'interno di ogni blocco, i thread lavorano in parallelo su righe differenti: ogni thread elabora una riga distinta del blocco, evitando dipendenze tra i thread e garantendo un utilizzo efficiente delle risorse hardware della GPU. Questa suddivisione del lavoro consente di **massimizzare il parallelismo** e migliorare le prestazioni computazionali, riducendo la latenza di accesso alla memoria globale.

Identificazione della riga da parte di ciascun thread

Nel kernel CUDA, ogni thread identifica la riga su cui deve lavorare in base al proprio indice globale, calcolato come:

$$\text{global\_row} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$$

Questo valore rappresenta l'indice della riga assegnata al thread all'interno della matrice. Successivamente, per gestire la suddivisione in blocchi ELLPACK, si determina:

- Il **blocco** della matrice a cui appartiene la riga:

$$\text{block\_id} = \frac{\text{global\_row}}{\text{HackSize}}$$

- La **riga locale** all'interno del blocco:

$$\text{local\_row} = \text{global\_row} \% \text{HackSize}$$

Infine, per accedere agli elementi della riga, viene calcolato l'**offset** all'interno del blocco ELLPACK:

$$\text{row\_offset} = \text{local\_row} \times \texttt{block->max\_nz\_per\_row}$$

Questa strategia garantisce che ogni thread lavori in maniera indipendente su una riga specifica, evitando conflitti e ottimizzando l'uso delle risorse della GPU.

2) Seconda versione con shared memory: In questa versione la griglia e i blocchi sono bi-dimensionali e la configurazione viene ottimizzata per garantire un'efficiente distribuzione del carico di lavoro, tenendo conto dei limiti hardware e delle caratteristiche della matrice sparsa. Il processo di configurazione dinamica

della griglia dei blocchi di thread, segue i seguenti passaggi:

1) **Determinazione del numero massimo di elementi non zero per riga**: Si esamina ogni blocco della matrice per identificare il numero massimo di elementi non zero per riga. Questo valore viene successivamente allineato a un multiplo di *HackSize* per garantire una gestione ottimale dei dati all'interno della GPU.
2) **Allineamento del numero di non zeri per riga**: Se il numero massimo di elementi non zero per riga non è già un multiplo di HackSize, viene arrotondato al più vicino multiplo superiore. Inoltre, si verifica che il numero massimo di non zeri per riga non superi il numero massimo di thread per blocco diviso per HackSize, per evitare di superare i limiti imposti dalla GPU.
3) **Allineamento del numero di blocchi**: Il numero totale di blocchi (numBlock) viene allineato anch'esso a un multiplo di HackSize per una distribuzione più equilibrata del lavoro. Se il numero di blocchi supera la dimensione massima della griglia supportata dalla GPU (maxGridDimX), viene limitato a questo valore massimo.
4) **Distribuzione della griglia su due dimensioni**: La griglia viene poi distribuita su due dimensioni per ottimizzare la parallelizzazione e migliorare il bilanciamento del carico di lavoro. Il numero di blocchi lungo l'asse X (grid_x) è calcolato come la radice quadrata del numero totale di blocchi, mentre il numero di blocchi lungo l'asse Y (grid_y) è determinato arrotondando il risultato della divisione del numero totale di blocchi per grid_x.
5) **Configurazione finale della griglia e dei blocchi**: Infine, le dimensioni del blocco (*BLOCK_DIM1*) e della griglia (*GRID_DIM1*) vengono impostate. Ogni blocco avrà una dimensione di HackSize $\times$ HackSize, e la griglia sarà composta da grid_x blocchi lungo l'asse X e grid_y lungo l'asse Y, per ottimizzare la distribuzione del carico di lavoro e l'uso delle risorse hardware della GPU.

La griglia configurata in questo modo consente di assegnare a ciascun blocco di thread la responsabilità di calcolare il prodotto matrice-vettore per un sottoinsieme di righe della matrice, mentre ogni thread all'interno di un blocco si occupa di calcolare una parte del prodotto scalare per una riga specifica. Il processo del calcolo parallelo avviene nei seguenti passi principali:

1) **Distribuzione delle righe sui blocchi**:Ogni **thread** calcola il proprio **indice globale di riga** global_row utilizzando la formula:

$$\text{global\_row} = (\text{blockIdx.x} + \text{blockIdx.y} \times \text{gridDim.x}) \times \text{blockDim.x} + \text{threadIdx.x}$$

Questo calcolo assegna in modo univoco ogni **riga della matrice** a un **thread** all'interno della griglia.

- Se il valore di global_row supera il numero totale di righe M, il thread termina immediatamente;
- La matrice è divisa in **blocchi ELLPACK**, e ogni thread identifica a quale **blocco** (block_id) appartiene la riga su cui lavorerà.

2) **Allocazione della memoria condivisa**: Per **migliorare l'efficienza** e **ridurre l'accesso alla memoria globale**, viene utilizzata una **memoria condivisa bidimensionale**:

$$\text{shared\_sum}[32][32]$$

Dove è importante definire i seguenti aspetti:

- Questa memoria è **inizializzata a zero** da tutti i thread prima di iniziare i calcoli;
- Ogni thread usa questa memoria per **accumulare i risultati parziali del prodotto scalare**.
- La **sincronizzazione** con `syncthreads()` è necessaria prima di procedere al suo utilizzo.

3) **Calcolo del prodotto scalare**: Ogni thread moltiplica un **elemento non nullo** della propria riga con il **valore corrispondente del vettore di input**:

$$\text{shared\_sum}[\text{threadIdx.x}][\text{thread\_col}] = \texttt{block->AS}[\text{row\_offset} + \text{thread\_col}] \times d_x[\texttt{block->JA}[\text{row\_offset} + \text{thread\_col}]]$$

Dopo il calcolo, viene eseguita una **sincronizzazione** per garantire che tutti i thread abbiano completato il loro calcolo prima della fase successiva.

4) **Riduzione parallela**: Per ottenere la somma finale del prodotto scalare, viene eseguita una **riduzione parallela nella memoria condivisa** con la formula:

$$\text{shared\_sum}[\text{threadIdx.x}][\text{thread\_col}] += \text{shared\_sum}[\text{threadIdx.x}][\text{thread\_col} + \text{stride}]$$

dove il valore di **stride** viene dimezzato a ogni iterazione fino a ottenere il valore finale.

5) **Scrittura del risultato**: Alla fine della riduzione, **solo il primo thread della colonna** (thread_col $= 0$) scrive il risultato finale nel vettore di output:

$$d_y[\text{global\_row}] = \text{shared\_sum}[\text{threadIdx.x}][0]$$

In questo modo, il calcolo parallelo sfrutta la potenza di calcolo della GPU, riducendo il tempo di esecuzione e migliorando l'efficienza del prodotto matrice-vettore.

VI. Misurazione delle prestazioni

Tutte le misure sono state ottenute considerando il tempo medio, ottenuto dalla media dei tempi di esecuzione misurati per ogni invocazione del nucleo di calcolo. Le metriche considerate per misurare le performance nel nostro caso di studio sono:

- **Flops**: È un'unità di misura della potenza di calcolo di un computer, in particolare delle sue capacità di elaborare calcoli matematici complessi. Questa metrica viene misurata tramite la seguente formula:
$$FLOPS = \frac{2 \cdot NZ}{T}$$
Dove NZ è il numero di elementi non nulli nella matrice e T è il tempo medio di esecuzione, ottenuto dalla media di tutte le invocazioni effetuate dell'unità di calcolo.
- **Speed Up**: Questa metrica consente di confrontare l'efficienza dell'esecuzione parallela rispetto a quella seriale. Questa metrica viene misurata tramite la seguente formula:
$$S(W,p) = \frac{T_s(W)}{T_p(W,p)}$$
Dove le componenti per il calcolo di questa metrica sono:
  - $T_s(W)$: che rappresenta il tempo di esecuzione seriale;
  - $T_p(W,p)$: che rappresenta il tempo di esecuzione parallela su p processori;
  - W: rappresenta la dimensione del problema.
- **Efficienza** : Questa metrica misura quanto bene vengono sfruttati i processori (o le risorse di calcolo) durante l'esecuzione di un algoritmo parallelo, rispetto a un'ideale esecuzione in cui ogni processore contribuisce in modo perfettamente lineare.

VII. Misurazione dell'errore relativo

Per valutare la precisione dell'esecuzione parallela di *SpMV* rispetto alla controparte sequenziale, si calcola un *errore relativo element-wise* per ciascun elemento dei vettori y_{seq} (sequenziale) e y_{par} (parallelo). In particolare, per ogni elemento i:

$$\varepsilon_i = \begin{cases} 0 & \text{se } |y_{\text{seq}}[i] - y_{\text{par}}[i]| \leq \text{absTolerance} \\ \dfrac{|y_{\text{seq}}[i] - y_{\text{par}}[i]|}{\max\left(|y_{\text{seq}}[i]|,\ |y_{\text{par}}[i]|\ \text{toleranceRel}\right)} & \text{altrimenti} \end{cases}$$

Le costanti **absTolerance** e **toleranceRel** (dell'ordine di 10^{-7}–10^{-6}) servono a ignorare differenze inferiori alla precisione numerica in doppia precisione, evitando di classificare come *errore* scostamenti molto piccoli. Si sommano quindi i soli ε_i diversi da zero e se ne fa la media:

$$\bar{\varepsilon} = \frac{1}{|\mathcal{I}|} \sum_{i \in \mathcal{I}} \varepsilon_i,$$

dove $\mathcal{I}$ è l'insieme degli indici in cui $\varepsilon_i \neq 0$. Questa stima quantifica quanto il risultato parallelo differisca in media dal riferimento sequenziale, principalmente a causa della **non associatività** delle operazioni in virgola mobile. Infatti, l'esecuzione multi-thread e i diversi partizionamenti della matrice possono modificare l'ordine delle somme, introducendo variazioni rispetto al risultato calcolato dalla versione sequenziale. Questa misurazione dell'errore commesso, viene eseguita per ognuna delle invocazioni del nucleo di calcolo. In tal modo, sarà possibile considerare l'errore medio effettuato su tutte le iterazioni.

VIII. Risultati

In questa sezione viene condotta un'analisi dettagliata dei dati sulle performance raccolte. I risultati ottenuti saranno commentati e discussi, mettendo in luce le principali tendenze, le criticità riscontrate e i successi conseguiti. L'obiettivo è fornire una visione chiara e approfondita degli elementi che hanno influenzato l'andamento delle performance, creando così una solida base per le successive discussioni e conclusioni.

A. OpenMP

Nelle seguenti sottosezioni saranno discussi e analizzati i risultati ottenuti in OpenMP, considerando i due diversi formati di memorizzazione utilizzati.

1) CSR: Iniziamo l'analisi dei risultati esaminando la misurazione dei megaFlops al variare del numero di thread utilizzati, come illustrato nella Figura 3.

Inizialmente, con un numero ridotto di thread, si osserva un incremento delle prestazioni al crescere del numero di thread. Tuttavia, una volta superato un certo numero di thread (tra i 10 e i 15), le prestazioni raggiungono un picco e tendono poi a stabilizzarsi o addirittura a diminuire. Questo comportamento è dovuto all'overhead introdotto dalla gestione dei thread e dalla concorrenza nelle risorse di sistema.

Un altro aspetto interessante riguarda l'influenza del numero di nonzeri nella matrice. Le categorie con una maggiore densità di nonzeri, come 100.000-500.000 e >=10.000.000, mostrano prestazioni significativamente superiori rispetto a quelle con una densità inferiore. Questo risultato è atteso, in quanto un maggiore densità di nonzeri consente ai thread di eseguire un carico di

lavoro più consistente, riducendo il numero di operazioni di sincronizzazione tra i thread e migliorando l'efficienza complessiva.

Al contrario, le matrici con una densità inferiore di nonzeri, come $<$10.000 e 10.000-100.000 non riescono a sfruttare appieno il parallelismo, poiché il carico di lavoro per ciascun thread è troppo basso. In questi casi, l'overhead dovuto alla gestione del parallelismo supera i guadagni ottenuti dall'aumento del numero di thread, portando ad un calo delle prestazioni.

In sintesi, il comportamento osservato conferma che è cruciale bilanciare adeguatamente il numero di thread con il carico di lavoro per thread. In particolare, le matrici con una mggiore densità di nonzeri offrono un potenziale maggiore per sfruttare efficacemente il parallelismo, consentendo di ottenere prestazioni più elevate. Questo fenomeno evidenzia l'importanza della scelta del formato di memorizzazione e del tipo di matrice per ottimizzare le prestazioni nei calcoli paralleli.

Inoltre, è particolarmente interessante analizzare l'andamento della metrica che misura lo speedup, come mostrato nella Figura 4.

Il grafico presentato mostra l'andamento dello speedup medio in relazione al numero di thread utilizzati, considerando diverse categorie di matrici raggruppate in base al numero di nonzeri.

In generale, si osserva che all'inizio, con un numero ridotto di thread, lo speedup cresce rapidamente, in particolare per le matrici con una densità maggiore di nonzeri. Le curve relative alle categorie 100.000-500.000 e 1.000.000-2.500.000 mostrano un aumento significativo dello speedup, indicando che l'algoritmo parallelo riesce a sfruttare efficacemente l'aumento dei thread in queste condizioni. Tuttavia, superata una certa soglia di thread, lo speedup tende a stabilizzarsi e, in alcuni casi, a diminuire come per le categorie con 500.000-1.000.000 prima di raggiungere la stabilità ha un piccolo declino. Questo fenomeno rappresenta un chiaro segnale di overhead causato dalla gestione dei thread e dalla sincronizzazione, il cui impatto diventa sempre più evidente all'aumentare del numero di thread oltre un certo puntoQuesto viene evidenziato in maniera evidente per la categorie di matrici con un numero di nonzeri $<$10.000 e 10.000-100.000 .

Un altro aspetto fondamentale riguarda l'influenza del numero di nonzeri. Le matrici con un numero maggiore di nonzeri, come quelle delle categorie 1.000.000-2.500.000 e 2.500.000-10.000.000 mostrano uno speedup molto più elevato. Questo è dovuto al fatto che un maggior numero di nonzeri comporta un carico di lavoro maggiore per ogni thread, consentendo una distribuzione più efficace del lavoro tra i thread. Al contrario, le matrici con meno nonzeri, come $<$10.000 e 10.000-100.000, mostrano uno speedup più basso. In questi casi, il carico di lavoro per thread è inferiore e l'overhead di parallelizzazione diventa più rilevante, limitando il miglioramento delle prestazioni.

In sintesi, il grafico evidenzia che, per ottenere i migliori valori di speedup in un algoritmo parallelo, è essenziale bilanciare il numero di thread con il carico di lavoro per thread.

Per ottenere un quadro completo delle prestazioni del nucleo di calcolo parallelo, è fondamentale analizzare anche l'efficienza, una metrica che indica quanto efficacemente il parallelismo viene sfruttato, confrontando il miglioramento delle prestazioni con il numero di thread impiegati.

Questa metrica è rappresentata nel grafico dell'efficienza media (Figura 5), che illustra l'andamento della variabile al variare del numero di thread, tenendo conto delle diverse categorie di matrici classificate in base al numero di nonzeri.

Dal grafico si evince che l'efficienza media diminuisce progressivamente con l'aumentare del numero di thread per tutte le categorie di matrici. Questo fenomeno è dovuto principalmente all'overhead di gestione dei thread. Con l'aumento dei thread, soprattutto oltre una certa soglia, la gestione della concorrenza diventa meno efficiente, riducendo così l'efficienza complessiva.

Le matrici con un numero maggiore di nonzeri per riga, come quelle nelle categorie 1.000.000-2.500.000 e 2.500.000-10.000.000, mostrano una diminuzione dell'efficienza meno significativa. Questo suggerisce che, con un carico di lavoro maggiore per ogni thread, l'algoritmo riesce a sfruttare meglio i thread disponibili, anche quando il numero di thread aumenta. In altre parole, ciò conferma che le matrici più con un gran numero di nonzeri traggono maggior vantaggio dall'aumento del numero di thread, riuscendo a mantenere un'efficienza relativamente più elevata.

D'altra parte, le matrici con un densità inferiore di nonzeri, come quelle nelle categorie $<$10.000 e $10.000-100.000$, mostrano un calo dell'efficienza più rapido. In questi casi, l'overhead dovuto alla gestione del parallelismo tra i thread supera rapidamente i benefici derivanti dall'aumento del numero di thread, riducendo drasticamente l'efficienza.

Per identificare il numero ottimale di thread per eseguire l'unità di calcolo, è necessario confrontare il grafico dell'efficienza media con quello dello speedup. In generale, il numero ottimale di thread può essere individuato osservando il punto in cui l'efficienza media inizia a stabilizzarsi e dove lo speedup non migliora più significativamente o inizia a decrescere. Il numero ottimale di thread è quindi il punto in cui l'incremento di prestazioni si stabilizza, senza una perdita sostanziale di efficienza.

Questo comportamento è valido per tutte le fasce di nonzeri analizzate, ma risulta particolarmente evidente nelle matrici con un numero maggiore di nonzeri,

come quelle nelle categorie 1.000.000-2.500.000 e 2.500.000-10.000.000. E' possibile concludere che, in generale, il numero *ottimale di thread* si trova sempre nell'intervallo discreto $[15, 20]$, dove l'algoritmo sfrutta al meglio il parallelismo senza incorrere in un eccessivo overhead.

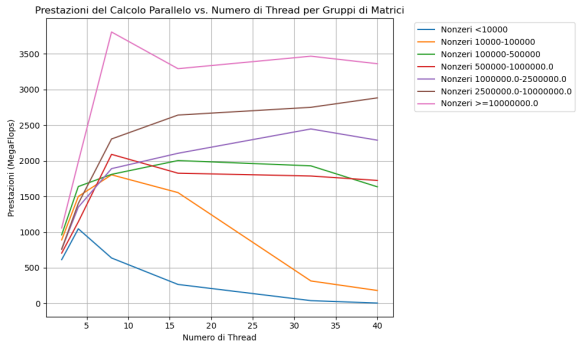


Figura 3: MegaFlops al variare del numero di thread nel formato CSR

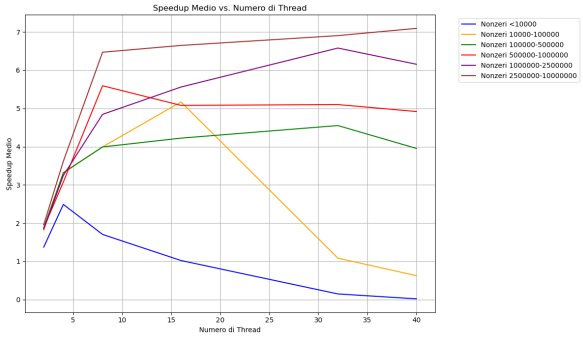


Figura 4: Speedup al variare del numero di thread nel formato CSR

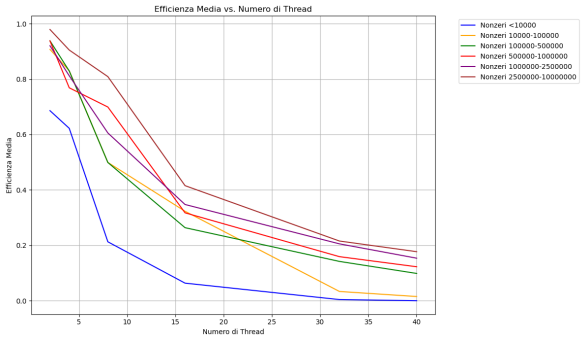


Figura 5: Efficienza al variare del numero di thread nel formato CSR

2) HLL: Iniziamo l'analisi dei risultati esaminando la misurazione dei megaFlops al variare del numero di thread utilizzati, come illustrato nella Figura 6.

Il grafico presentato mostra le prestazioni in megaFlops per il calcolo parallelo in HLL, al variare del numero di thread, per diverse categorie di matrici in base al numero di nonzeri.

Dal grafico si evince che, con un numero ridotto di thread, le prestazioni aumentano in modo significativo, con un incremento più marcato per le matrici con un numero maggiore di nonzeri. Le curve che descrivono le categorie di matrici raggrupate in base al loro numero di nonzeri, hanno tutte una rapida crescità delle loro performance partendo da un numero basso di thread. La rapidità della loro crescità è direttamente proporzionale al numero di nonzeri della categoria che rappresentano. In particolare questa rapidità più significativa della crescita è evidenziata dalle matrici con un numero elevato di non zeri, indicando che l'algoritmo parallelo riesce a sfruttare efficacemente l'aumento dei thread, specialmente per matrici con un gran numero di nonzeri.

Al contrario, le matrici con un numero di nonzeri inferiore nella matrice sparsa, come quelle nelle categorie 10.000-100.000 e $<$10.000, mostrano un aumento più lento delle prestazioni. In particolare, la curva blu ($<$10.000) e la curva arancione (10.000-100.000) mostrano un miglioramento più contenuto, che tende a stabilizzarsi. Questo indica che, per matrici con un ridotto numero di nonzeri, l'algoritmo non riesce a sfruttare appieno l'aumento dei thread, probabilmente a causa di un carico di lavoro insufficiente per ogni thread, con l'overhead che inizia ad incidere negativamente.

In conclusione, il grafico mostra che le prestazioni delle matrici con una gran quantità di elementi non nulli continuano a migliorare fino a un certo numero di thread, per poi stabilizzarsi o subire una lieve flessione oltre i 30-35 thread. Ciò indica che, superata una certa soglia, l'overhead legato alla gestione dei thread annulla i benefici del parallelismo. Per le matrici con pochi nonzeri, invece, il punto di stabilizzazione si raggiunge molto prima, intorno ai 10-15 thread.

Inoltre, è interessante analizzare lo speedup, poiché indica quanto l'esecuzione parallela risulti più veloce rispetto alla versione seriale. L'andamento di questa metrica in relazione al numero di thread è illustrato nella Figura 7.

Il grafico dello speedup medio mostra come le prestazioni migliorino all'aumentare del numero di thread, considerando diverse categorie di matrici classificate in base al numero di nonzeri.

Dal grafico si nota che, per tutte le categorie di matrici, lo speedup aumenta rapidamente con l'aumentare del numero di thread, con un picco significativo nelle matrici con un numero maggiore di nonzeri. In particolare, le curve per le categorie $>$=10.000.000 e 1.000.000-2.500.000 mostrano un aumento dello speedup molto evidente con il numero di thread, indicando che l'algoritmo parallelo riesce a sfruttare bene l'aumento dei thread in queste condizioni. Questo com-

portamento era atteso, poiché un numero più elevato di nonzeri garantisce un carico di lavoro maggiore e una distribuzione più equilibrata tra i thread.

Al contrario, le matrici con un numero inferiore di nonzeri, come quelle nelle categorie $<$10.000 e 10.000-100.000, mostrano un aumento dello speedup meno significativo. In particolare, la curva blu ($<$10.000) e la curva arancione (10.000-100.000) evidenziano un miglioramento che tende a stabilizzarsi molto prima rispetto alle categorie con numeri di nonzeri elevati. Questo comportamento suggerisce che, per matrici con pochi nonzeri, l'overhead di sincronizzazione e gestione dei thread diventa troppo significativo, riducendo i guadagni in termini di prestazioni.

Dopo il picco iniziale, tutte le curve tendono a stabilizzarsi o a decrescere leggermente con l'aumento dei thread oltre i 30-35.

Infine, per determinare il numero ottimale di thread e ottenere una visione complessiva dell'andamento delle prestazioni, è fondamentale affiancare allo studio delle metriche precedenti l'analisi dell'efficienza al variare dei thread, come mostrato in Figura 8.

L'efficienza è un indicatore cruciale, in quanto mostra quanto efficacemente le prestazioni migliorano con l'aumento del numero di thread. Inoltre, aiuta a identificare il numero ottimale di thread per una data matrice.

Dal grafico si osserva che, per tutte le categorie di matrici, l'efficienza media diminuisce progressivamente al crescere del numero di thread. Le curve relative alle matrici con un numero maggiore di nonzeri, come 1.000.000-2.500.000 e 2.500.000-10.000.000, mostrano una diminuzione dell'efficienza più lenta rispetto alle matrici con meno nonzeri. Questo suggerisce che l'algoritmo parallelo riesce a sfruttare meglio i thread disponibili quando il carico di lavoro per thread è maggiore.

Al contrario, le matrici con un numero inferiore di nonzeri, come $<$10000 e 10000-100000, mostrano una rapida diminuzione dell'efficienza. In questi casi, il carico di lavoro per thread è ridotto e l'overhead introdotto dalla gestione dei thread inizia a dominare, riducendo rapidamente l'efficienza. Le curve per queste matrici si avvicinano velocemente a valori molto bassi di efficienza.

Le curve tendono a stabilizzarsi quando il numero di thread supera i 20-25 per le matrici con più nonzeri, mentre per quelle con meno nonzeri, l'efficienza si stabilizza prima. In particolare, si osserva che dopo il picco iniziale, l'efficienza media non migliora significativamente e si avvicina a valori più bassi con l'aumentare del numero di thread, indicando che oltre un certo punto l'algoritmo non riesce più a sfruttare efficacemente l'aumento dei thread.

Il numero ottimale di thread si trova generalmente al punto in cui l'efficienza media inizia a stabilizzarsi e non migliora ulteriormente. Come indicato dai grafici, per le matrici con un numero maggiore di nonzeri, il numero ottimale di thread si aggira tra i 20-30 thread. Per matrici con un numero di nonzeri inferiore, invece, l'efficienza diminuisce rapidamente e il numero ottimale di thread risulta essere inferiore, intorno ai 10-15 thread, per evitare sovraccarichi da gestione dei thread.

Per concludere lo studio riguardante le prestazioni, è fondamentale, dopo aver valutato le metriche dello speedup, dell'efficienza e dei megaFlops, osservare che il nucleo di calcolo può introdurre un errore durante il calcolo. Questo errore è dovuto alla **non associatività delle operazioni in virgola mobile**. Le operazioni in virgola mobile, infatti, non sono sempre associative, il che significa che la stessa sequenza di operazioni può produrre risultati leggermente differenti a seconda dell'ordine in cui vengono eseguite.

Questa ultima metrica viene descritta dalla Figura 9, la quale evidenzia come, in base alla progettazione del calcolo parallelo, l'errore introdotto sia nullo o trascurabile. Questo risultato indica che il calcolo parallelo, progettato correttamente, non introduce errori significativi a causa della non associatività, garantendo così una buona accuratezza nel risultato finale.

L'errore minimo, quindi, è accettabile e non influisce sostanzialmente sulla qualità dei risultati prestazionali, come evidenziato dalla comparazione con i valori di riferimento nelle metriche precedentemente analizzate.

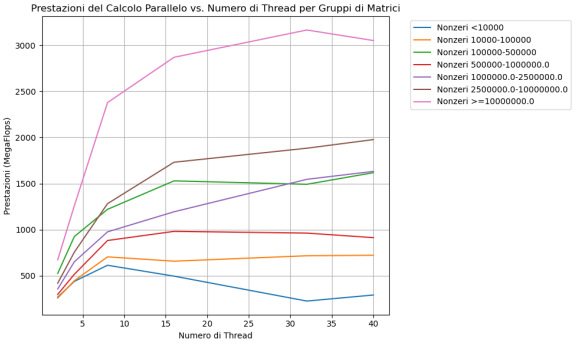


Figura 6: MegaFlops al variare del numero di thread nel formato HLL

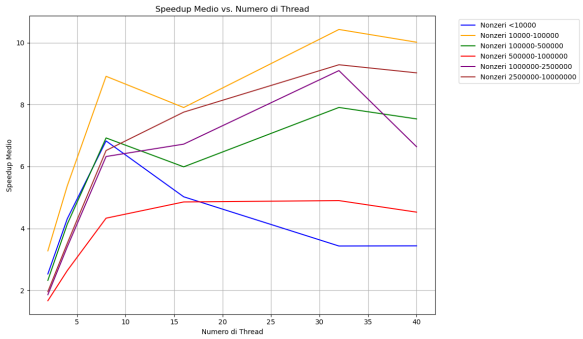


Figura 7: Speedup al variare del numero di thread nel formato HLL

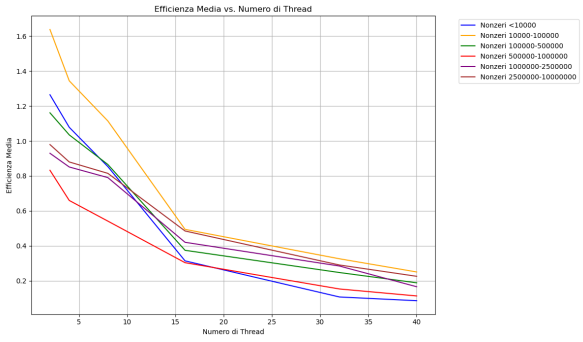


Figura 8: Efficienza al variare del numero di thread nel formato HLL

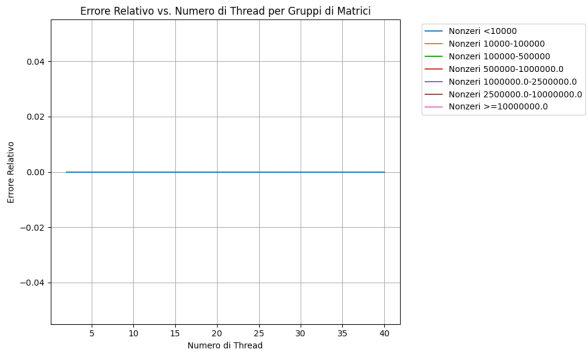


Figura 9: Errore Relativo al variare del numero di thread in OpenMP

B. CUDA

Nella seguente sezione, si andrà ad affrontare un analisi dettagliata delle prestazioni, effettuando un confronto dettagliato tra i vari approcci adottati.

Iniziamo dall'analisi del grafico che descrive la variazione della metrica Megaflops nelle tre versioni realizzate per il formato CSR. Si ricorda che la differenza tra le tre versioni consiste in (si indica ciascuna versione con il nome del file .json prodotto dall'esecuzione del nucleo di calcolo):

- CUDA_CSR_v1: implementa il calcolo parallelo senza l'utilizzo di shared memory;
- CUDA_CSR_v2_32x32: implementa il calcolo parallelo con l'utilizzo di shared memory e la riduzione sequenziale del calcolo parallelo. La size 32x32 indica la dimensione dei blocchi di thread;
- CUDA_CSR_v3_32x32: implementa il calcolo parallelo con l'utilizzo di shared memory e la riduzione sequenziale del calcolo parallelo. La size 32x32 indica la dimensione dei blocchi di thread;

Il confronto delle varie prestazioni viene illustrato in Figura 10.

Il grafico presentato mostra il confronto delle prestazioni in megaFlops per diverse matrici, utilizzando quattro diverse versioni dell'algoritmo CUDA. Le matrici sono organizzate da sinistra verso destra, con una quantità di nonzeri che cresce progressivamente.

Nelle matrici con densità di nonzeri inferiore, come nel caso di `cage4`, `west1021` e `adder_dcop_32`, l'implementazione CUDA_CSR_v1 (in blu) mostra generalmente le migliori prestazioni risetto alle altre versioni parallele. Questo comportamento è dovuto alla semplicità dell'algoritmo, che non utilizza la memoria condivisa né riduzioni parallele. Si sottolinea però, come dal grafico vinene sottolineata il forte divario di prestazione tra il calcolo seriale e quello parallelo (aldilà della versione), questo perchè l'overgead introdotto per la gestione dei thread è maggiore dei benifici ottenuti, dovuto dal basso numero di nonzeri e dal relativo scarso sfruttamento delle risorse parallele.

Inoltre risulta subito eviednete come la versione CUDA_CSR_v1 offra le migliori prestazioni appena si ha un numero di non zeri sufficienti per poter utilizzare il parallelismo. Si evidenzia come le prestazioni migliorino in maniera sempre più significante all'aumentare della quantità di non zeri contenuti nella matrice. Questo è riscontrabile osservando come nella matrice `bcsstk17` la versione CUDA_CSR_v1 raddopi il numero di megaFLops rispetto alle altre versioni parallele , fino a renderle praticamente trascurabili per matrici come `thermal2.mtx` e `roadNet-PA`. Questo è dovuto dall'efficenza della progettazione di una griglia di blocchi di thread studiata appositamente per avere la massima efficienza per le caratteristiche della matrice considerata. Questo aspetto si combina perfettamente con un ottimo utilizzo delle risorse parallele sfruttando la massimo i thread presenti nei blocchi della griglia.

Inoltre, gli algoritmi CUDA_CSR_v2_32x32 (in arancione) e CUDA_CSR_v3_32x32 (in verde), che sfruttano la memoria condivisa, non riescono a ottenere le stesse prestazioni. Sebbene la memoria condivisa offra un miglioramento nelle prestazioni, l'overhead di sincronizzazione e la gestione parallela dei thread non

sono sufficientemente vantaggiosi per matrici con una bassa densità di nonzeri.

Con matrici che presentano una densità di nonzeri maggiore, come `amazon302`, `mac_econ_fwd500` e `ML_Laplace`, le differenze tra gli algoritmi diventano più evidenti. CUDA_CSR_v3_32x32 (verde), che utilizza la memoria condivisa combinata con la riduzione parallela, mostra un miglioramento significativo delle prestazioni rispetto ad CUDA_CSR_v2_32x32 (arancione), che impiega la riduzione sequenziale. L'utilizzo della riduzione parallela consente di ridurre notevolmente i tempi di calcolo, soprattutto su matrici con grandi quantità di elementi non nulli, dove il carico di lavoro per thread è più consistente.

In confronto, CUDA_CSR_v2_32x32 mostra buone prestazioni, ma non è in grado di ottenere gli stessi guadagni di CUDA_CSR_v3_32x32, poiché la riduzione sequenziale limita l'efficienza su matrici con un gran numero di nonzeri. La capacità di CUDA_CSR_v3_32x32 di eseguire la riduzione in parallelo permette di ottenere prestazioni superiori grazie all'ottimizzazione delle operazioni di riduzione che sfruttano meglio il parallelismo.

Per matrici particolarmente grandi e complesse, come `cant`, `thermal.mtx` e `nlpkkt80.mtx`, CUDA_CSR_v3_32x32 continua a dominare rispetto a CUDA_CSR_v2_32x32, raggiungendo valori di megaFlops molto più elevati rispetto agli altri algoritmi. La riduzione parallela consente di ottenere prestazioni eccellenti, riducendo significativamente i tempi di calcolo, specialmente su matrici grandi, dove il numero di operazioni parallele è elevato.

Inoltre, estremamente interessante è l'analisi della differenza delle prestazioni nella versioni CUD_CSR_v2 al variare della grandezza dei blocchi di thread, come descritto in Figura 12.

Il grafico mostra le prestazioni in megaFlops per diverse matrici e quattro diverse dimensioni di blocco di thread (1024x1, 128x8, 32x32, e 512x2). Le matrici sono organizzate da sinistra verso destra, in base alla quantità di nonzeri.

Per matrici sparse con un basso numero di non zeri, come `cage4` e `west1021`, le differenze di prestazioni tra le diverse dimensioni di blocco sono meno significative, indicando che l'overhead della gestione parallela dei thread non è influente su queste matrici.

Con matrici con maggiore ad alto numero di nonzeri, come `amazon302`, `mac_econ_fwd500` e `ML_Laplace`, le differenze tra le dimensioni di blocco diventano più evidenti. CUDA_CSR_v2 con blocchi di dimensione 32x32 (verde) ottiene le migliori prestazioni per matrici con grandi quantità di nonzeri. Questo potrebbe essere dovuto aalla parallelizzazione più equilibrata grazie alla dimensione del blocco, che permette di ridurre i tempi di calcolo nelle operazioni sui dati più consistenti.

Le stesse osservazioni possono essere estese anche all'analisi della Figura 13, che mostra l'andamento delle prestazioni in relazione alla dimensione dei blocchi di thread.

È importante effettuare lo studio delle performance, osservando il diverso andamento in base alla versione del calcolo parallelo con il formato HLL in CUDA realizzato. Questo aspetto può essere osservato nella Figura 14.

Il grafico presenta un confronto delle prestazioni in megaFlops tra tre versioni dell'algoritmo HLL in CUDA: CUDA_serial_HLL (verde), CUDA_HLL_v1 (blu) e CUDA_HLL_v2 (arancione). Le matrici sono organizzate da sinistra a destra, dalla densità di nonzeri inferiore a quella superiore, consentendo di osservare il comportamento degli algoritmi su matrici sparse e dense.

Per le matrici con un numero molto contenuto di nonzeri, come `cage4` e `west1021`, la versione CUDA_HLL_v1 (blu) ottiene prestazioni superiori rispetto a CUDA_HLL_v2 (arancione). La versione v1 non utilizza la memoria condivisa e sfrutta una configurazione più semplice, ma riesce comunque a ottenere buone prestazioni su matrici con una quantità bassa di elementi non nulli. In queste matrici, l'overhead introdotto dalla gestione della memoria condivisa e dalla griglia bidimensionale di CUDA_HLL_v2 potrebbe risultare meno efficiente.

Per le matrici con un gran numero di nonzeri, come `amazon302` e `ML_Laplace`, le prestazioni di CUDA_HLL_v1 (blu) sono spesso superiori a quelle di CUDA_HLL_v2 (arancione), nonostante CUDA_HLL_v2 sfrutti la memoria condivisa e la griglia bidimensionale. Questo comportamento può essere attribuito al fatto che la griglia bidimensionale non riesce a sfruttare pienamente tutte le risorse parallele disponibili sulla GPU. Inoltre, l'overhead legato alla gestione della memoria condivisa e dei thread nella versione v2 si rivela un fattore limitante, specialmente su matrici più grandi. In questi casi, CUDA_HLL_v1 mostra un calcolo più semplice e diretto, che sembra ottimizzare meglio le prestazioni in presenza di un elevato numero di elementi non nulli.

Il grafico suggerisce che, mentre CUDA_HLL_v1 offre prestazioni eccellenti su matrici con un qualsiasi numero di nonzeri, CUDA_HLL_v2 non riesce a sfruttare pienamente la parallelizzazione e la memoria condivisa a causa dell'overhead introdotto dalla griglia bidimensionale e dalla gestione dei thread. Questo comporta una penalizzazione nelle prestazioni, soprattutto per le matrici di grande dimensione. La versione seriale (verde) mostra prestazioni notevolmente inferiori, come era prevedibile, poiché non sfrutta la parallelizzazione.

In conclusione, le prestazioni delle versioni CUDA sono influenzate dalla quantità di nonzeri nelle matrici

e dalla configurazione della griglia di thread. La versione CUDA_HLL_v1 ottiene le migliori prestazioni, in particolare per matrici con molti nonzeri. Al contrario, CUDA_HLL_v2, pur sfruttando la memoria condivisa e una griglia bidimensionale, introduce un overhead che limita l'efficienza, specialmente su matrici più grandi. La versione seriale risulta essere la meno performante in tutte le condizioni di test.

Per concludere l'analisi delle performance nel nostro studio, è utile confrontare tutte le versioni realizzate nei rispettivi formati di memorizzazione. Le differenze di prestazioni tra le varie implementazioni sono illustrate nella Figura 11.

Il grafico mostra il confronto delle prestazioni in megaFlops tra quattro varianti dell'algoritmo CUDA CSR e HLL. Le matrici sono ordinate da sinistra verso destra, con una quantità di nonzeri che aumenta progressivamente.

Per matrici con un bassa numero di nonzeri, come `cage4`, `west1021` e `adder_dcop_32`, CUDA_CSR_v1 (blu) e CUDA_CSR_v2_32x32 (verde) ottengono generalmente le migliori prestazioni.

Con matrici con un numero maggiore di nonzeri, come `amazon302`, `mac_econ_fwd500`, e `ML_Laplace`, le differenze di prestazioni diventano più evidenti. In questo caso, CUDA_HLL_v1 (rosso) ottiene le prestazioni migliori, probabilmente grazie alla rappresentazione in memoria favorevole nel caso di HLL.

Il grafico conferma quindi che per matrici con quantità molto ridotte di nonzeri, le prestazioni tra i vari algoritmi non mostrano grandi differenze. Tuttavia, per matrici con molti elementi non nulli, le implementazioni realizzate per il formato HLL risultano essere le più efficaci.

Per concludere l'analisi di questa metrica, le performance sono influenzate principalmente dalla quantità di nonzeri. Le matrici con pochi nonzeri sono più efficientemente gestite da CUDA_CSR_v1 e CUDA_CSR_v2_32x32, mentre per le matrici con un gran numero di nonzeri, CUDA_CSR_v1 e CUDA_HLL_v1 offrono prestazioni superiori.

Si sottolinea che lo studio delle metriche deve essere valutato considerando anche l'errore relativo introdotto nei risultati a causa delle caratteristiche intrinseche delle operazioni in virgola mobile. L'errore relativo risulta essere nullo o trascurabile, se compreso tra 10^{-7} e 10^{-6}, per tutte le versioni che utilizzano il formato di memorizzazione CSR. Inoltre, per il formato HLL, la versione CUDA_CSR_v1 mostra lo stesso comportamento, con errore relativo minimo. Tuttavia, per la versione CUDA_CSR_v2, che utilizza una griglia bidimensionale e la memoria condivisa, non si verifica lo stesso scenario, e l'errore relativo risulta più significativo.

Si ricorda che la versione CUDA_CSR_v2 utilizzava una griglia bidimensionale con blocchi bidimensionali e la memoria condivisa. L'errore relativo introdotto dalla non associatività delle operazioni in virgola mobile viene descritto nella Figura 15, dove le matrici sono ordinate da sinistra verso destra in ordine crescente di nonzeri.

Dall'analisi di tale grafico, si conferma che la versione migliore in termini di accuratezza è CUDA_CSR_v1, grazie alla sua efficienza nello sfruttare le risorse parallele, che hanno prodotto un risultato corretto. Inoltre, la progettazione della griglia dinamica di CUDA_CSR_v1, idonea alle caratteristiche della matrice considerata, ha contribuito a mantenere l'errore relativo entro limiti trascurabili.

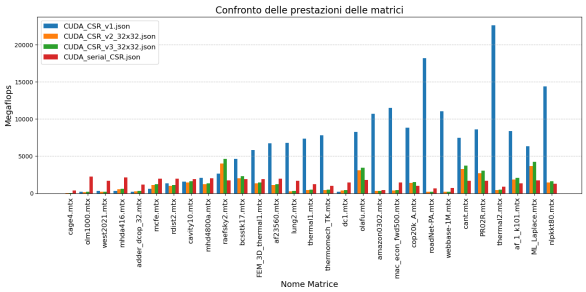


Figura 10: Confronto tra le implementazioni CUDA per il formato di memorizzazione CSR

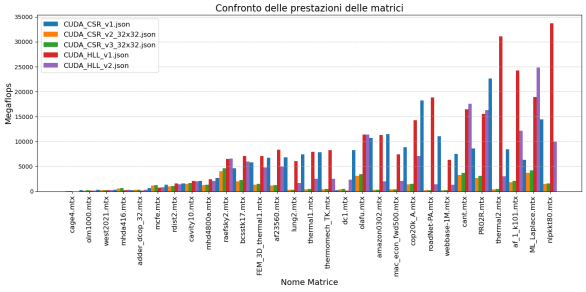


Figura 11: Confronto tra le implementazioni CUDA confrontando CSR e HLL

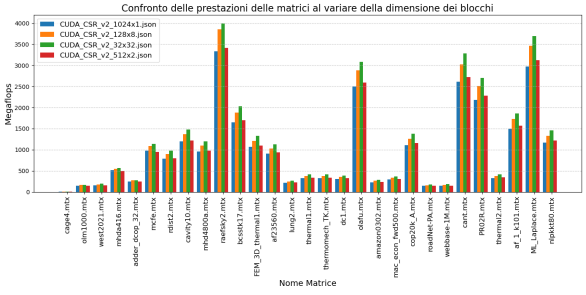


Figura 12: Confronto delle prestazioni ottenute per la versione 2 (CUDA) con formato CSR al variare della dimensione dei blocchi

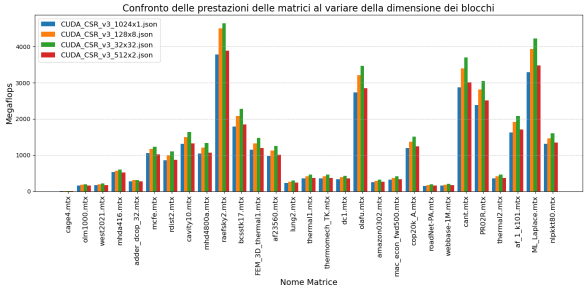


Figura 13: Confronto delle prestazioni ottenute per la versione 3 (CUDA) con formato CSR al variare della dimensione dei blocchi

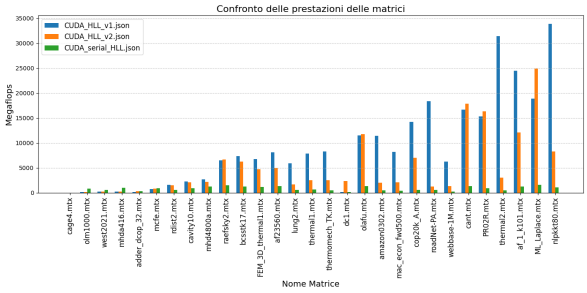


Figura 14: Confronto delle prestazioni ottenute per la versione seriale e parallela con e senza memoria condivisa in CUDA nel formato HLL

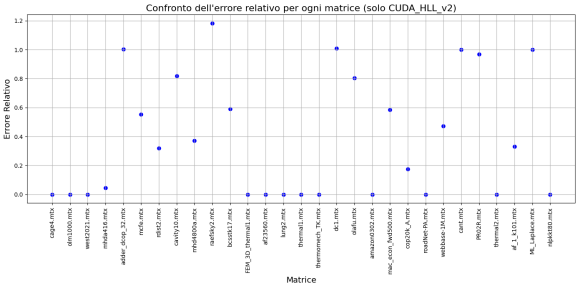


Figura 15: Errore relativo introdotto nel calcolo nella versione CUDA_CSR_v1 nel formato HLL

IX. Suddivisione del lavoro

L'intero progetto è stato suddiviso in modo uniforme, con una stretta collaborazione in ogni fase, garantendo una sinergia tra i partecipanti. Seppur in perfetta collaborazione, si può descrivere una suddivisone del lavoro in maniera uniforme:

- Implementazione del preprocessamento dati: Andrea Andreoli
- Implementazione della funzione di conversione in CSR: Pierfrancesco Lijoi
- Implementazione della funzione di conversione in HLL: Andrea Andreoli
- Implementazione della logica di suddivisione delle righe tra thread in CSR (OpenMP): Pierfrancesco Lijoi
- Implementazione del prodotto matrice-vettore in CSR (OpenMP): Andrea Andreoli
- Implementazione della logica di suddivisione dei blocchi tra thread in HLL (OpenMP): Andrea Andreoli e Pierfrancesco Lijoi
- Implementazione del prodotto matrice-vettore in HLL (OpenMP): Pierfrancesco Lijoi
- Implementazione del prodotto matrice-vettore in CSR (CUDA) v1: Piefrancesco Lijoi
- Progettazione della dimensione della griglia in CSR (CUDA) v2-v3: Pierfrancesco Lijoi
- Implementazione del prodotto matrice-vettore in CSR (CUDA) v2-v3: Andrea Andreoli
- Progettazione della dimensione della griglia in HLL (CUDA) v1: Pierfrancesco Lijoi
- Implementazione del prodotto matrice-vettore in HLL (CUDA) v1: Pierfrancesco Lijoi
- Progettazione della dimensione della griglia in HLL (CUDA) v2: Pierfrancesco Lijoi e Andrea Andreoli
- Implementazione del prodotto matrice-vettore in HLL (CUDA) v2: Pierfrancesco Lijoi e Andrea Andreoli
- Analisi e raccolta delle statistiche di performance e dei grafici: Andrea Andreoli e Pierfrancesco Lijoi
- Stesura del report: Andrea Andreoli e Pierfrancesco Lijoi