TOR VERGATA

UNIVERSITÀ DEGLI STUDI DI ROMA

Univesità degli Studi di Roma

Tor Vergata

Parallel Sparse Matrix-Vector Product

Matteo Calzetta
kalzmatteo@gmail.com

Alessandro Lori
alessandrolori179@gmail.com

Abstract

Questo documento tratta le fasi seguite per sviluppare un nucleo di calcolo che implementi il prodotto tra matrice sparsa e vettore denso, nella forma $y = Ax$. Per il formato di memorizzazione della matrice sono stati usati i formati:

- CSR (Compressed sparse row)
- HLL (Hacked ellpack)

Il prodotto è stato eseguito in maniera seriale e poi parallelizzato, il parallelismo è stato implementato tramite OpenMP e CUDA. Analizzeremo come sono state memorizzate le matrici, implementazioni del prodotto in parallelo e tecniche di ottimizzazione di questo.

Contents

1 Introduction **1**

2 Data preprocessing **1**
2.1 CSR . . . 1
2.2 HLL . . . 2
2.3 HLL Column-Major . . . 3
2.4 HLL implementation . . . 3

3 Metrics **3**

4 Base products **4**

5 Result validation **5**

6 OpenMP **6**
6.1 OpenMP using CSR format . . . 7
6.2 OpenMP using HLL format . . . 8

7 CUDA **8**
7.1 CUDA with CSR . . . 10
7.1.1 Kernel 0 – Un thread per riga . . . 10
7.1.2 Kernel 1 – Un warp per riga . . . 11
7.1.3 Kernel 2 – Un warp per riga con uso di `shared memory` . . . 12
7.1.4 Kernel 3 – Un warp per riga con uso combinato di `shared memory` e `__shfl_sync()` 12
7.1.5 Kernel 4 – Un warp per riga con uso della Cache L2 . . . 13
7.1.6 Kernel 5 – Un warp per riga con uso della Texture Cache . . . 14
7.2 CUDA with HLL . . . 14
7.2.1 kernel 0 . . . 14
7.2.2 Kernel 1 . . . 15
7.2.3 Kernel 2 . . . 16
7.2.4 Kernel 3 . . . 16

8 Results **18**
8.1 Serial . . . 18
8.2 OpenMP Results . . . 18
8.2.1 OpenMP HLL . . . 18
8.2.2 OpenMP CSR . . . 20
8.2.3 CSR vs CSR-Guided . . . 21
8.3 CUDA Results . . . 22
8.3.1 CUDA CSR . . . 22
8.3.2 CUDA HLL . . . 23
8.4 Best HLL vs best CSR . . . 25

9 Load balance **25**

1 Introduction

Il prodotto tra matrice sparsa e vettore (SpMV - Sparse Matrix-Vector product) coinvolge matrici che contengono prevalentemente valori nulli, con pochi elementi diversi da zero distribuiti in modo irregolare. A seconda di come questi elementi non nulli sono posizionati, possiamo individuare diverse tipologie specifiche di matrici sparse.

La grande quantità di zeri presenti in queste matrici genererebbe uno spreco significativo di memoria se si utilizzassero metodi di memorizzazione tradizionali. Per risolvere questo problema, utilizziamo formati di memorizzazione specializzati, come il formato CSR e il formato HLL, che permettono di ridurre la memoria utilizzata e di ottimizzare le prestazioni di calcolo.

Per aumentare ulteriormente le performance di calcolo, sfruttiamo la parallelizzazione offerta dalle CPU (processore) e dalle GPU (scheda grafica).

Lo sviluppo del nostro progetto prevede quattro fasi principali:

1. Lettura delle matrici sparse e loro conversione nei formati ottimizzati CSR e HLL.
2. Implementazione del prodotto matrice-vettore per ciascun formato in forma seriale
3. Implementazione del prodotto matrice-vettore per ciascun formato parallelizzando tramite il framework su CPU (OpenMP)
4. Implementazione del prodotto matrice-vettore per ciascun formato parallelizzando tramite il framework su GPU (CUDA).

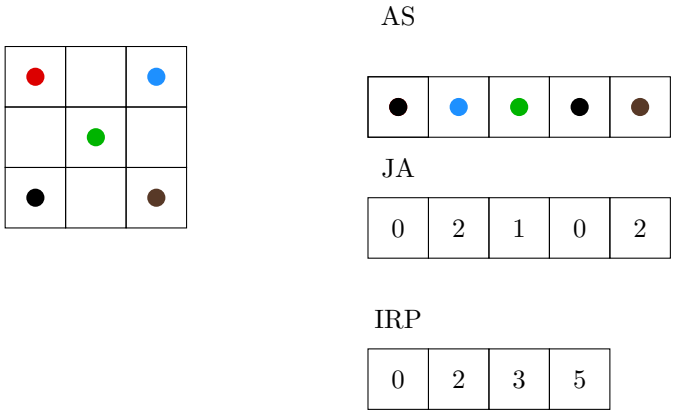
2 Data preprocessing

La conversione delle matrici utilizzate per testare le performance del nucleo di calcolo a partire dal formato originale Matrix Market (.mtx) sfrutta la libreria ANSI C library for Matrix Market I/O, che mette a disposizione le funzioni incluse in mmio.c e l'header mmio.h. Queste routine consentono di rilevare eventuali strutture particolari (ad esempio Pattern, Symmetric e Array) e ricostruire la matrice in maniera corretta. Una volta identificata la struttura della matrice, i valori vengono estratti e salvati nei formati di interesse (CSR e HLL), così da preparare i dati per le successive operazioni di calcolo. La rappresentazione nei due formati di interesse è fondamentale per migliorare il consumo di memoria e ottimizzare il prodotto, in quanto le matrici, essendo sparse, contengono molti zeri che non vogliamo né memorizzare né moltiplicare.

2.1 CSR

Nel formato di memorizzazione CSR (Compressed sparse row) l'idea di base è quella di comprimere le righe della matrice memorizzando soltanto gli elementi non nulli e creare degli array che ci aiutano a risalire alla posizione all'interno della matrice di partenza di questi elementi. I tre vettori principali sono:

- AS, vettore che contiene i valori non nulli della matrice;
- JA, vettore che memorizza gli indici colonna degli elementi non nulli;
- IRP, vettore di puntatori agli indici di AS in cui inizia ciascuna riga della matrice.



2.2 HLL

Il formato Hacked ELLPACK (HLL) nasce per risolvere il principale problema del formato ELLPACK: lo spreco di memoria causato dal padding. Infatti, in ELLPACK, tutte le righe devono avere lo stesso numero di colonne (pari alla riga più lunga della matrice), quindi se anche una sola riga ha molti elementi, si alloca memoria inutile anche per tutte le altre. Per ridurre questo overhead, HLL divide la matrice in gruppi di righe, chiamati Hack, e ogni gruppo viene memorizzato separatamente come una mini-matrice in formato ELLPACK. In questo modo, una riga molto lunga influenza solo il suo blocco, non l'intera matrice. Da notare che se scegliessimo uno come dimensione dei blocchi avremo una memorizzazione come quella CSR senza alcun padding, mentre se usassimo come dimensione dei blocchi la dimensione delle righe della matrice (avendo così un solo blocco) sarebbe come usare ELLPACK. Gli elementi che vanno memorizzati per ogni blocco ELLPACK sono:

- MAXNZ: Numero massimo di non zeri per riga (utilizzato per padding);
- JA: Array bidimensionale di indici colonna;
- AS: Array bidimensionale di coefficinti NZ della matrice.

L'array bidimensionale serve a localizzare i valori di AS all'interno del blocco, la riga in JA corrisponde alla riga all'interno del blocco ELLPACK.

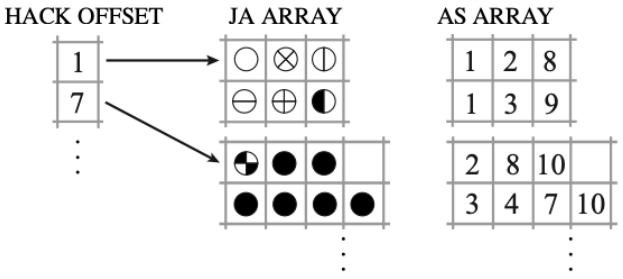


Figure 1: HLL Format

2.3 HLL Column-Major

Nel progetto, per ottimizzare l'accesso alla memoria durante l'esecuzione del prodotto matrice-vettore su GPU, è stata adottata una variante del formato HLL (Hacked ELLPACK) con disposizione column-major, ovvero per colonne logiche all'interno dei blocchi.

Come nel formato HLL tradizionale, la matrice viene suddivisa in blocchi (chiamati hack) di dimensione fissa HackSize, ciascuno contenente un sottoinsieme di righe consecutive. Ogni blocco è rappresentato in modo simile al formato ELLPACK, ma con la differenza fondamentale che gli elementi non nulli vengono memorizzati colonna per colonna, anziché riga per riga.

2.4 HLL implementation

A differenza della rappresentazione classica del formato ELLPACK, che prevede l'uso di array bidimensionali per memorizzare i valori e gli indici delle colonne, la nostra implementazione adotta array monodimensionali per rappresentare i campi `JA` e `AS` all'interno di ciascun blocco HLL.

Questa scelta offre molteplici vantaggi: semplifica la gestione della memoria tra host e device, migliora la compatibilità con CUDA, consente un controllo esplicito sul layout dei dati e permette di ottimizzare gli accessi alla memoria attraverso un'organizzazione più adatta ai modelli di accesso della GPU.

In particolare, utilizzando array monodimensionali, è possibile modellare direttamente un layout column-major all'interno di ogni blocco, garantendo accessi coalescenti tra thread adiacenti durante l'esecuzione dei kernel. Questo tipo di accesso risulta essenziale per sfruttare al massimo la larghezza di banda della memoria globale della GPU.

Inoltre, nel caso di array bidimensionali dinamici (tipicamente rappresentati come array di puntatori a righe), le righe non sono necessariamente memorizzate in posizioni contigue in memoria, il che introduce disallineamenti e accessi non efficienti. Evitare questa frammentazione migliora significativamente le performance del prodotto matrice-vettore.

3 Metrics

Le prestazioni del prodotto matrice-vettore sono state valutate ripetendo più volte l'invocazione del nucleo di calcolo su ciascuna matrice, così da ottenere un tempo medio di esecuzione. Le misure considerano solo il tempo di calcolo del prodotto, escludendo il preprocessing. Le tre metriche considerate sono:

1. FLOPS: unità di misura della potenza di un computer.

$$FLOPS = \frac{2 \cdot NZ}{T} \tag{1}$$

   dove NZ è il numero di elementi non nulli nella matrice (ricalcolato in caso di simmetria) e T il tempo medio per invocazione.

2. Speedup: metrica di confronto tra versione parallela e seriale, definita come il rapporto tra il tempo di esecuzione della versione seriale e quella parallela:

$$\text{Speedup} = \frac{T_{\text{seriale}}}{T_{\text{parallelo}}}$$

3. Efficienza: misura quanto efficacemente vengono sfruttate le risorse computazionali impiegate durante l'esecuzione parallela. È definita come:

$$\text{Efficienza} = \frac{\text{Speedup}}{\text{NumThreads}}$$

4 Base products

Di seguito pseudicodice per le implementazioni seriali del prodotto matrice vettore nei 3 formati:

Algorithm 1 Prodotto matrice sparsa-vettore con formato CSR

```
1: for i = 1 to m do
2:     t ← 0
3:     for j = irp(i) to irp(i + 1) − 1 do
4:         t ← t + as(j) · x(ja(j))
5:     y(i) ← t
```

Algorithm 2 Prodotto matrice-vettore seriale con formato HLL row-major

```
1: for blockID = 0 to num_blocks − 1 do
2:     start_row ← blockID · HackSize
3:     end_row ← min((blockID + 1) · HackSize, M)
4:     maxNZ ← hll_matrix.blocks[blockID].max_nz_per_row
5:     row_offset ← 0
6:     for i = start_row to end_row − 1 do
7:         y[i] ← 0
8:         for j = 0 to maxNZ − 1 do
9:             idx ← row_offset + j
10:            if JA[idx] ≠ −1 then
11:                y[i] ← y[i] + AS[idx] · x[JA[idx]]
12:        row_offset ← row_offset + maxNZ
```

Gli algoritmi di prodotto matrice-vettore descritti sopra, nelle versioni seriali per i formati HLL row-major e column-major, sono stati implementati inizialmente con l'obiettivo di prendere confidenza con la struttura di memorizzazione dei dati e comprendere in modo approfondito come accedere correttamente agli elementi della matrice nei diversi layout.

Queste versioni base sono servite anche come riferimento per verificare la correttezza delle rappresentazioni e dei risultati numerici: nelle prime fasi del progetto abbiamo testato il funzionamento con matrici piccole, confrontando manualmente i risultati ottenuti con quelli attesi. Successivamente, abbiamo esteso i test anche a matrici più grandi e con caratteristiche particolari, come quelle pattern (senza valori associati) e simmetriche, ottenendo risultati coerenti con le aspettative.

Una volta consolidata la correttezza di queste versioni sequenziali, le abbiamo utilizzate come riferimento affidabile per validare l'output delle versioni più complesse e parallele. In particolare, sono state usate come base per il calcolo della norma relativa dell'errore tra il risultato seriale e quello parallelo, garantendo così la validazione numerica del comportamento delle versioni CUDA.

Algorithm 3 Prodotto matrice-vettore seriale con formato HLL column-major

Inizializza $y(i) \leftarrow 0$ per $i = 0$ **to** $M - 1$
for $b = 0$ **to** $num\_blocks - 1$ **do**
    $start\_row \leftarrow b \cdot HackSize$
    $rows \leftarrow HackSize$
    if $b = num\_blocks - 1$ **and** $M \bmod HackSize \neq 0$ **then**
        $rows \leftarrow M \bmod HackSize$
    $maxNZ \leftarrow hll\_matrix.blocks[b].max\_nz\_per\_row$
    for $r = 0$ **to** $rows - 1$ **do**
        $sum \leftarrow 0$
        for $c = 0$ **to** $maxNZ - 1$ **do**
            $idx \leftarrow c \cdot rows + r$
            $val \leftarrow AS[idx]$
            $col \leftarrow JA[idx]$
            $sum \leftarrow sum + val \cdot x[col]$
        $y[start\_row + r] \leftarrow sum$

Questa fase è risultata essenziale per il debugging delle prime implementazioni GPU, in quanto ha permesso di isolare eventuali anomalie legate all'accesso in memoria, alla gestione dei blocchi o all'uso della memoria condivisa.

5 Result validation

Per verificare la correttezza delle implementazioni del prodotto matrice-vettore nei diversi formati e paradigmi di parallelismo, è stato utilizzato il calcolo della norma euclidea (L2) tra il vettore risultato y prodotto da ciascun algoritmo e il vettore store generato dalla versione seriale di riferimento.

$$\|\mathbf{a} - \mathbf{b}\|_2 = \sqrt{\sum_{i=0}^{M-1} (a_i - b_i)^2}$$

In particolare, nelle fasi iniziali, abbiamo confrontato i risultati delle tre versioni sequenziali (CSR, HLL row-major, HLL column-major) su matrici di dimensione ridotta. I valori della norma ottenuti erano prossimi allo zero, il che ha confermato che tutte le implementazioni erano numericamente equivalenti.

Algorithm 4 Calcolo della norma L_2 tra due vettori

$norm \leftarrow 0$
for $i = 0$ **to** $M - 1$ **do**
    $diff \leftarrow store[i] - y[i]$
    $norm \leftarrow norm + diff^2$
return $\sqrt{norm}$

Successivamente, abbiamo esteso questo approccio anche alle versioni parallele:

- Per ogni implementazione in OpenMP, abbiamo confrontato il risultato con quello ottenuto dalla versione sequenziale dello stesso formato.
- Allo stesso modo, per ogni implementazione in CUDA, il risultato è stato validato rispetto alla sua controparte seriale.

Dal momento che le operazioni coinvolgono numeri rappresentati come double, la norma L2 potrebbe non risultare esattamente zero anche in presenza di implementazioni corrette. Questo è dovuto al fatto che l'aritmetica floating-point non è associativa e il riordinamento delle operazioni (es. causato dalla parallelizzazione o da piccoli cambiamenti nell'ordine di accesso ai dati) può generare piccole differenze numeriche. Tuttavia, queste differenze sono fisiologiche e trascurabili quando la norma risulta molto vicina a zero.

6 OpenMP

OpenMP è un'API progettata per facilitare la programmazione parallela su architetture multi-core e multiprocessore, ed è compatibile con i linguaggi C, C++ e Fortran. In questo progetto, è stato utilizzato il linguaggio C, sfruttando le direttive OpenMP per parallelizzare l'esecuzione del prodotto matrice-vettore (SpMV) suddividendo il carico di lavoro complessivo su un insieme di threads.

Per analizzare le performance in funzione del grado di parallelismo, ogni implementazione è stata eseguita più volte variando il numero di thread utilizzati. In particolare, i test sono stati effettuati utilizzando potenze di due, ovvero con [2, 4, 8, 16, 32, 40] thread, raggiungendo il massimo numero di thread supportato dalla piattaforma hardware in uso.

Uno degli aspetti fondamentali in questo contesto è la distribuzione bilanciata del carico di lavoro tra i thread: un'allocazione squilibrata può infatti compromettere le prestazioni, lasciando alcuni thread inattivi mentre altri sono ancora occupati. Una buona strategia di bilanciamento consente di massimizzare l'uso delle risorse computazionali e migliorare la scalabilità dell'algoritmo. A tal proposito abbiamo a disposizione una distribuzione del carico gestita da OpenMP attraverso la direttiva #pragma omp parallel for schedule(SCHEDULE_TYPE). Questa direttiva permette di parallelizzare un ciclo for distribuendo le iterazioni tra i threads disponibili secondo la strategia specificata dal parametro SCHEDULE_TYPE, questo parametro determina come le iterazioni vengono assegnate ai threads quindi di fatto la distribuzione del carico. OpenMP mette a disposizione 5 possibili tipi per il parametor SCHEDULE_TYPE: Static, Dynamic, Guided, Auto e Runtime.

- **Static**: le iterazioni del ciclo vengono suddivise in blocchi di dimensione fissa e assegnate in modo deterministico ai thread. Ogni thread riceve il proprio blocco all'inizio dell'esecuzione e lo esegue senza ulteriori riassegnazioni.
- **Dynamic**: le iterazioni vengono suddivise in blocchi di dimensione fissa, ma l'assegnazione è dinamica. Ogni thread riceve un primo blocco e, una volta terminato, richiede il successivo fino a completamento del ciclo. Questo approccio è utile in presenza di carichi sbilanciati.
- **Guided**: simile al dynamic, ma i blocchi assegnati dinamicamente hanno dimensione variabile: inizialmente grandi e poi via via più piccoli. Questo consente di sfruttare al meglio le capacità di calcolo nei primi cicli e ridurre il rischio di inattività nei thread verso la fine.
- **Auto**: la strategia di scheduling non viene definita esplicitamente. La scelta è demandata al compilatore, che decide in base alle caratteristiche hardware e alla configurazione dell'ambiente di esecuzione.

- **Runtime**: la strategia di scheduling viene decisa a tempo di esecuzione sulla base del valore della variabile d'ambiente `OMP_SCHEDULE`. Questo approccio consente flessibilità, lasciando la configurazione all'utente senza modificare il codice sorgente.

6.1 OpenMP using CSR format

Per realizzare l'esecuzione parallela dell'operazione SpMV in formato CSR, è innanzitutto necessario distribuire correttamente il carico di lavoro tra i thread disponibili. L'idea di base è suddividere le righe della matrice in sottoinsiemi, assegnando a ogni thread un intervallo contiguo di righe. In questo modo si ottiene una maggiore località spaziale negli accessi in memoria (evitando il salto tra blocchi di righe troppo distanti).

Un ulteriore vincolo importante è cercare di garantire che il numero di elementi non nulli (non-zero) per ogni thread risulti equilibrato: ciascun thread dovrebbe ricevere un carico di lavoro (in termini di operazioni in virgola mobile) che sia il più vicino possibile a quello degli altri. Se uno dei sottoinsiemi di righe assegnato a un thread risulta avere pochi (o nessun) elementi non nulli, occorre rivedere la suddivisione e, se necessario, eliminare quel thread dall'assegnazione aggiornandone il conteggio totale. Allo stesso tempo, è essenziale garantire che nessun thread si trovi con un range di righe vuoto, così da evitare risorse sprecate.

In condizioni ideali, si suppone che i non-zero siano distribuiti in modo regolare tra le righe; tuttavia, quando ciò non accade, è necessario riaggregare alcune righe adiacenti per creare nuovi sottoinsiemi di righe che bilancino meglio il numero totale di non-zero per thread. Questo approccio "adattivo" permette di mantenere un livello accettabile di bilanciamento anche in presenza di una distribuzione irregolare degli elementi non nulli.

Implementando questa strategia, ogni thread viene associato a un intervallo contiguo di righe e procede a iterare sui vettori CSR (`AS`, `JA`, `IRP`) solo per gli indici di sua competenza, in maniera indipendente rispetto agli altri thread. Questo accorgimento sfrutta al meglio il parallelismo a livello di riga e riduce sia gli overhead di sincronizzazione, sia quelli di gestione della memoria.

- Si conta quanti elementi non nulli contiene ogni riga (array `row_nnz`).
- Si calcola il carico totale e lo si divide per `num_threads` (ottenendo `target_workload`).
- Si esplorano le righe accumulando il carico nella variabile `current_workload`. Al superamento di `target_workload`, si inizia una nuova partizione.
- Quando si raggiunge la fine della matrice, si salva l'ultima partizione e si rialloca il vettore delle partizioni per rispecchiare effettivamente il numero di thread utilizzati.

Algorithm 5 Algoritmo di bilanciamento del carico per CSR

procedure LOADBALANCE(*csr*, *num_threads*)
  Alloca un array *thr_partition* di dimensione $(num\_threads + 1)$.
  Inizializza l'indice iniziale della prima partizione a riga 0.
  Alloca un array *row_nnz* per il conteggio dei non-zero per ciascuna riga.
  Calcola il numero totale di non-zero, *total_nnz*, sommando quelli di ogni riga.
  Determina il carico di lavoro target per ogni thread: $target\_workload = total\_nnz / num\_threads$.
  Inizializza $current\_workload = 0$ e $thread\_id = 0$.
  for $i = 0$ **to** $csr.M - 1$ **do**
    Somma i non-zero della riga i a *current_workload*.
    if $current\_workload \geq target\_workload$ **e** $thread\_id < num\_threads - 1$ **then**
      Conclude la partizione corrente: $thread\_id \leftarrow thread\_id + 1$.
      Aggiorna $thr\_partition[thread\_id] \leftarrow i + 1$.
      Azzera *current_workload* per la nuova partizione.
  Conclude l'ultima partizione e imposta $thr\_partition[thread\_id + 1] \leftarrow csr.M$.
  $thread\_id \leftarrow thread\_id + 1$ ▷ Ora thread_id = numero totale di partizioni create
  $actual\_threads \leftarrow thread\_id$.
  if $actual\_threads < num\_threads$ **then**
    Ridimensiona l'array *thr_partition* a $(actual\_threads + 1)$ elementi tramite `realloc`.
  Ritorna *thr_partition*

In seguito, si è sfruttata la suddivisione uniforme del carico di lavoro per l'esecuzione parallela: ogni thread elabora il proprio sottoinsieme di righe facendo riferimento ai dati nei vettori CSR. In questo modo, il carico di lavoro risulta bilanciato e tutti i thread svolgono un numero simile di operazioni in virgola mobile.

6.2 OpenMP using HLL format

Per parallelizzare il calcolo del prodotto matrice-vettore utilizzando il formato HLL abbiamo usufruito della direttiva #pragma omp parallel for schedule(SCHEDULE_TYPE) messa a disposizione da openMP per distirbuire il carico di lavoro tra i threads. Il parametro SCHEDULE_TYPE è stato impostato a guided in quanto è risultato il miglior approccio alla gestione del carico di lavoro per threads. Attraverso il parametro guided ogni thread riceverà uno o più blocchi della matrice da computare, ogni thread identificherà le righe dei blocchi da processare per poi elaborarle tutte. Inoltre la direttiva di OpenMP non richiede sincronizzazione tra i thread perché ogni thread lavora su un insieme disgiunto di righe non creando conflitti di scrittura sul vettore risultante. Uno svantaggio dell'uso della direttiva di OpenMP con lo scheduling impostato a guided è l'introduzione di un overhead determinato dall'assegnazione dinamica dei blocchi, che obbliga il sistema a dover monitorare i threads per capire quando hanno finito l'esecuzione dei blocchi dati e di operazioni interne per determinare il futuro blocco da assegnare.

7 CUDA

CUDA è una piattaforma di calcolo parallelo sviluppata da NVIDIA che consente di sfruttare le GPU per eseguire operazioni altamente parallele e computazionalmente intensive. Rispetto alle CPU, le GPU dispongono di migliaia di core, permettendo così di eseguire contemporaneamente un numero

Algorithm 6 Moltiplicazione matrice-vettore con HLL e OpenMP

procedure HLLMatVecOpenMP($hll\_matrix, x, y, num\_threads$)
  parallel for con `schedule(guided)` e `num_threads` threads
  for $blockID \leftarrow 0$ extbfto $hll\_matrix.num\_blocks - 1$ **do**
    $start\_row \leftarrow blockID \times HackSize$
    $end\_row \leftarrow (blockID + 1) \times HackSize$
    if $end\_row > hll\_matrix.M$ **then**
      $end\_row \leftarrow hll\_matrix.M$
    $max\_nz\_per\_row \leftarrow hll\_matrix.blocks[blockID].max\_nz\_per\_row$
    $row\_offset \leftarrow 0$
    for $i \leftarrow start\_row$ extbfto $end\_row - 1$ **do**
      $sum \leftarrow 0.0$
      for $j \leftarrow 0$ extbfto $max\_nz\_per\_row - 1$ **do**
        $idx \leftarrow row\_offset + j$
        if $hll\_matrix.blocks[blockID].JA[idx] \neq -1$ **then**
          $sum \leftarrow sum + hll\_matrix.blocks[blockID].AS[idx] \times x[hll\_matrix.blocks[blockID].JA[idx]]$
      $y[i] \leftarrow sum$
      $row\_offset \leftarrow row\_offset + max\_nz\_per\_row$

molto elevato di operazioni. In particolare, CUDA consente di delegare calcoli specifici alla GPU trasferendo inizialmente i dati dall'host (CPU) al device (GPU). Tali operazioni parallele vengono eseguite attraverso speciali funzioni dette kernel, che vengono lanciate e completate interamente sul device.

L'architettura CUDA organizza i thread in una struttura gerarchica ben definita per massimizzare l'efficienza. L'unità base di esecuzione è il thread, che viene eseguito all'interno di un warp, cioè un gruppo di 32 thread che rappresenta l'unità minima di esecuzione simultanea sulla GPU. I warp sono raggruppati in blocchi di thread, ciascuno dei quali opera autonomamente su una porzione definita dei dati. L'insieme completo di blocchi costituisce la griglia (grid), che copre l'intero dataset da elaborare. I thread e i blocchi all'interno della griglia possono essere organizzati in una, due o tre dimensioni:

- Se la griglia e i blocchi sono monodimensionali, si userà un unico indice (x) per identificare thread e blocchi,
- Se sono bidimensionali, si useranno due indici (x,y),
- Nel caso tridimensionale, gli indici saranno tre (x,y,z)

Questa struttura gerarchica permette ai thread di accedere in maniera indipendente a specifiche parti del dataset complessivo, facilitando l'organizzazione e la gestione efficiente dei dati in memoria.

I thread sono eseguiti da più Streaming Multiprocessors (SM) della GPU, ognuno dei quali ha la responsabilità della gestione e dell'esecuzione di specifici warp. La distribuzione bilanciata dei thread sugli SM è fondamentale per evitare sovraccarichi o sottoutilizzo delle risorse GPU.

Alla luce di questo, è necessario trovare dimensione e numero di blocchi ottimale tenendo in considerazione le seguenti cose:

- Dimensione del blocco, che deve essere multiplo della grandezza degli SM, perché i thread degli

SM vengono eseguiti in un unico ciclo di clock. Scegliere una dimensione diversa da questa potrebbe portare ad una parallelizzazione inefficiente e quindi diminuire le prestazioni

- Limite fisico dell'hardware sottostante, nel caso specifico la GPU su cui sono stati eseguiti i test supporta blocchi di dimensione massima 1024

Le funzioni eseguite su kernel inoltre, hanno un'altra variabile: il tipo di memoria utilizzata.

- Shared Memory: È una memoria che viene allocata con un massimo di 48KB all'interno di un singolo SM (Streaming Multiprocessor). Non garantisce direttamente coalescenza ma la permette in caso di corretta organizzazione dei dati in essa evitando così accessi disordinati alla memoria globale e quindi non coalescenti. La memoria condivisa permette una latenza di accesso ai dati nettamente inferiore rispetto all'accesso ai dati in memoria globale, 20-30 cicli contro 400-800 cicli della memoria globale. Uno degli svantaggi della memoria condivisa è la necessità di sincronizzare i thread, questo potrebbe portare a dei momenti di overhead e race conditions ovvero accessi non coerenti ai dati risultando in un ordine non deterministico di accesso da parte dei thread. La funzione usata per la sincronizzazione è __syncthreads() che forza tutti i thread di un blocco a fermare l'esecuzione e aspettare che tutti i thread abbiano raggiunto quel determinato punto di codice.

- CacheL2: È una memoria interna di caching della GPU che non garantisce la coalscenza dei dati ma è estremamente più veloce della memoria globale in accesso, necessita meno cicli di clock, 50-100 contro i 400-800 della memoria globale. È gestita interamente dall'hardware, la GPU decide dinamicamente quali dati mantenere in CacheL2 e quali sostituire. Quando eseguiamo __ldg(DATO) viene consultata la presenza del dato nella CacheL2, se presente viene tornata altrimenti questo viene preso dalla memoria globale, ritornato e caricato in CacheL2 per eventuali accessi futuri. Ciò porta un netto vantaggio qual'ora il dato viene trovato nella cache a fronte altrimenti di una latenza maggiore se dopo il controllo in CacheL2 il dato non è presente bisogna accedere alla memoria globale per recuperarlo.

- Texture Memory o CacheTexture: È una memoria interna di caching della GPU separata e più lenta della CacheL2, 100-400 cicli di clock contro i 50-100 della CacheL2, ma ottimizzata per gli accessi sparsi o non coalescenti ai dati, adatta per operazioni su lookup di tabelle, rendering grafico interpolazione dati scientifici. È gestita interamente dall'hardware, la GPU decide dinamicamente quali dati mantenere in Texture Cache e quali sostituire. Quando eseguiamo tex1Dfetch< $TIPODATO$ >(NOME_TEXTURE, DATO); viene consulata la Cache Texture, se il dato è resente viene tornato altrimenti questo viene preso dalla memoria globale, ritornato e caricato nella Cache Texture per eventualia accessi futuri, ciò porta un netto vantaggio nella gestione di accesso a dati non coalescenti e alla latenza rispetto ad un accesso unico alla memoria globale, 100-400 cicli contro 400-800 cicli della memoria globale.

7.1 CUDA with CSR

Per l'esecuzione dell'operazione SpMV attraverso CUDA nel formato di memorizzazione CSR sono stati sviluppati 6 kernel di esecuzione:

7.1.1 Kernel 0 – Un thread per riga

Il kernel `spmv_csr_threads` esegue il prodotto matrice-vettore (`SpMV`) assegnando a ciascuna riga della matrice un thread responsabile della sua computazione. Questa strategia semplifica la logica del kernel,

in quanto ogni thread opera in modo indipendente, senza necessità di sincronizzazione.

Il numero di thread per blocco è fissato a 256, mentre il numero di blocchi viene calcolato dinamicamente in base al numero di righe M della matrice, secondo la formula:

$$\texttt{num\_blocks} = \frac{M + \texttt{threads\_per\_block} - 1}{\texttt{threads\_per\_block}}$$

Questa configurazione consente di allocare un numero sufficiente di blocchi per saturare i multiprocessori della GPU, migliorando l'occupazione e massimizzando il parallelismo. In questo modo si evita il sottoutilizzo delle risorse hardware disponibili.

Algorithm 7 Moltiplicazione matrice-vettore in formato CSR con CUDA

```
1: procedure SpMV_CSR_Threads(M, IRP, JA, AS, x, y)
2:     row ← blockIdx.x × blockDim.x + threadIdx.x
3:     if row < M then
4:         sum ← 0.0
5:         row_start ← IRP[row]
6:         row_end ← IRP[row + 1]
7:         for j ← row_start to row_end − 1 do
8:             sum ← sum + AS[j] × x[JA[j]]
9:         y[row] ← sum
```

7.1.2 Kernel 1 – Un warp per riga

Il kernel `spmv_csr_warps` implementa l'operazione `SpMV` assegnando a ciascuna riga della matrice un intero *warp* (32 thread) responsabile della sua computazione. Ogni blocco CUDA è composto da 4 warp, e la dimensione della griglia viene calcolata dinamicamente in base al numero di righe M della matrice, secondo la seguente formula:

$$\texttt{gridDim.x} = \frac{M + \texttt{blockDim.y} - 1}{\texttt{blockDim.y}}$$

dove `blockDim.y` rappresenta il numero di warp per blocco (nel nostro caso, 4).

Questa configurazione consente di assegnare più warp attivi alla computazione in parallelo, migliorando l'occupazione dei multiprocessori rispetto all'approccio "un thread per riga". Tuttavia, l'uso cooperativo dei thread all'interno del warp introduce la necessità di **ridurre i risultati parziali** calcolati da ciascuna lane.

Per risolvere questo problema di riduzione, viene utilizzata la primitiva `__shfl_sync()` (o una delle sue varianti: `__shfl_down_sync()`, `__shfl_up_sync()`, `__shfl_xor_sync()`). Queste funzioni permettono la **condivisione diretta dei valori tra i thread di uno stesso warp**, trasferendo una variabile da un thread sorgente agli altri, secondo parametri specifici della funzione.

Nel kernel proposto, viene adottata una strategia in cui **solo il thread con `lane_id == 0`** si occupa di ricevere i valori parziali calcolati dalle altre lane tramite la `__shfl_down_sync()`, completando così la riduzione della riga e scrivendo il risultato finale nel vettore `y`.

Algorithm 8 Moltiplicazione matrice-vettore in formato CSR con CUDA (Warp-based)

procedure SpMV_CSR_Warps(M, IRP, JA, AS, x, y)
  $row \leftarrow blockIdx.x \times blockDim.y + threadIdx.y$
  $lane \leftarrow threadIdx.x$
  if $row < M$ **then**
    $sum \leftarrow 0.0$
    $row\_start \leftarrow IRP[row]$
    $row\_end \leftarrow IRP[row + 1]$
    for $j \leftarrow row\_start + lane$ **to** $row\_end - 1$ **step** $WARP\_SIZE$ **do**
      $sum \leftarrow sum + AS[j] \times x[JA[j]]$
    for $offset \leftarrow WARP\_SIZE/2$ **to** 1 **step** /2 **do**
      $sum \leftarrow sum +$ `__shfl_sync`$(0xFFFFFFFF, sum, lane + offset)$
    if $lane == 0$ **then**
      $y[row] \leftarrow sum$

7.1.3 Kernel 2 – Un warp per riga con uso di `shared memory`

Il kernel `spmv_csr_warps_shmem` mantiene la stessa struttura del kernel precedente, basata sull'assegnazione di un warp per riga, ma differisce nel metodo di riduzione dei risultati parziali: al posto delle primitive `__shfl_sync()`, viene utilizzata la **shared memory** condivisa tra i thread del blocco.

In particolare, ogni thread del warp calcola un valore parziale e lo scrive in una cella dedicata della memoria condivisa. Successivamente, i thread dello stesso warp collaborano eseguendo una *riduzione logaritmica* direttamente sulla shared memory, sincronizzandosi ad ogni passo tramite `__syncthreads()`.

Questo approccio presenta due vantaggi principali:

- **Maggiore controllo sulla riduzione**: rispetto all'uso delle shuffle instructions, la riduzione in shared memory è completamente esplicita e può essere facilmente modificata o estesa.
- **Accessi maggiormente coalescenti e localizzati**: tutti i valori parziali sono memorizzati in un'area contigua della shared memory, evitando l'accesso ripetuto alla memoria globale e migliorando l'efficienza complessiva del kernel.

Infine, come nei kernel precedenti, è il thread con `lane_id == 0` a scrivere il risultato finale della riga nel vettore `y`. L'uso della shared memory rappresenta una strategia alternativa che, pur introducendo una leggera latenza dovuta alle sincronizzazioni, risulta utile in scenari in cui si desidera evitare le primitive warp-level o quando si lavora su architetture meno recenti.

7.1.4 Kernel 3 – Un warp per riga con uso combinato di `shared memory` e `__shfl_sync()`

Il kernel `spmv_csr_warps_shmem_shfl` adotta la stessa struttura dei precedenti kernel basati sui warp, assegnando a ciascuna riga della matrice un intero warp per la computazione del prodotto scalare. La differenza principale risiede nel metodo utilizzato per la riduzione dei risultati parziali: viene infatti impiegata una combinazione tra **shared memory** e la funzione `__shfl_sync()`.

Ogni thread del warp calcola il proprio contributo parziale e lo memorizza in una **shared memory bidimensionale**, in cui ogni riga è riservata ai thread di un singolo warp. Questa struttura permette un'organizzazione ordinata e un accesso coalescente ai dati, riducendo i conflitti e migliorando l'efficienza dell'uso della memoria condivisa.

Algorithm 9 Riduzione con memoria condivisa in CUDA

1: **procedure** SPMV_CSR_WARPS_SHMEM($shared\_sum, sum, lane, row, y$)
2: $shared\_sum[lane + threadIdx.y \times WARP\_SIZE] \leftarrow sum$
3: `__syncthreads()`
4: **for** $offset \leftarrow WARP\_SIZE/2$ **to** 1 **step** /2 **do**
5: **if** $lane < offset$ **then**
6: $shared\_sum[lane + threadIdx.y \times WARP\_SIZE] \leftarrow shared\_sum[lane + threadIdx.y \times WARP\_SIZE] + shared\_sum[lane + threadIdx.y \times WARP\_SIZE + offset]$
7: `__syncthreads()`
8: **if** $lane == 0$ **then**
9: $y[row] \leftarrow shared\_sum[threadIdx.y \times WARP\_SIZE]$

La riduzione vera e propria viene effettuata tramite la funzione `__shfl_sync()`, che consente la condivisione dei risultati parziali tra le lane del warp. In particolare, i valori salvati in shared memory vengono letti e trasferiti alla `lane 0` del warp, la quale esegue l'accumulo finale e scrive il risultato nel vettore di output `y`.

Questa combinazione di tecniche consente di sfruttare i vantaggi della shared memory (accessi locali, strutturati e veloci) e della riduzione warp-level (efficienza computazionale), risultando in una soluzione mista altamente flessibile ed efficace.

Algorithm 10 Riduzione su warp con memoria condivisa e __shfl_sync

1: **procedure** SPMV_CSR_WARPS_SHMEM_SHFL($shared\_sum, sum, warp\_id, lane, row, y$)
2: $shared\_sum[warp\_id][lane] \leftarrow sum$
3: `__syncthreads()`
4: $sum \leftarrow shared\_sum[warp\_id][lane]$
5: **for** $offset \leftarrow 16$ **to** 1 **step** /2 **do**
6: $sum \leftarrow sum +$ `__shfl_sync`$(0xFFFFFFFF, sum, lane + offset)$
7: **if** $lane == 0$ **then**
8: $y[row] \leftarrow sum$

7.1.5 Kernel 4 – Un warp per riga con uso della Cache L2

Il kernel `spmv_csr_warps_cachel2` mantiene la stessa configurazione di griglia e blocchi dei kernel precedenti, assegnando un warp a ciascuna riga della matrice. La particolarità di questa versione è l'utilizzo della **Cache L2** tramite l'istruzione `__ldg()` per accedere in modo ottimizzato ai valori numerici e agli indici di colonna della matrice HLL.

L'uso esplicito della funzione `__ldg()` permette al compilatore di caricare i dati dalla memoria globale sfruttando la cache L2, migliorando la *locality* e riducendo la latenza degli accessi rispetto ai tradizionali accessi diretti alla global memory. Questo è particolarmente vantaggioso nel contesto del prodotto matrice-vettore, dove molti thread accedono ripetutamente a porzioni simili del vettore di input o a indici di colonna adiacenti.

Ciascun thread all'interno del warp esegue quindi il calcolo del proprio contributo parziale utilizzando i dati caricati con `__ldg()`, e la riduzione finale viene gestita come nei kernel precedenti, tipicamente tramite una *warp-level reduction* (es. `__shfl_down_sync()` o shared memory), in base

Algorithm 11 Accesso ai dati tramite __ldg

```
1: procedure SPMV_CSR_WARPS_CACHEL2_(row_start, row_end, lane, JA, AS, x, sum)
2:     for j ← row_start + lane to row_end − 1 step 32 do
3:         col ← __ldg(JA[j])
4:         sum ← sum + __ldg(AS[j]) × __ldg(x[col])
```

all'implementazione specifica.

Questo approccio è utile soprattutto per matrici di grandi dimensioni o con pattern di accesso ripetuti, dove l'efficienza della cache può portare a un vantaggio sensibile in termini di performance.la funzone __shfl_sync() come negli altri kernel.

7.1.6 Kernel 5 – Un warp per riga con uso della Texture Cache

Il kernel `spmv_csr_warp_texture` adotta la stessa configurazione di griglia e blocchi dei kernel precedenti, assegnando un warp a ciascuna riga della matrice. La particolarità di questa versione è l'utilizzo della **Texture Cache** per accedere ai valori e agli indici di colonna della matrice HLL, tramite la funzione:

`tex1Dfetch<`*`tipo_dato`*`>(nome_texture, indice)`

La Texture Cache è progettata specificamente per ottimizzare accessi **non coalescenti**, come quelli tipici delle sparse matrix, e fornisce una latenza inferiore rispetto alla global memory in scenari in cui i pattern di accesso sono irregolari o poco prevedibili.

In questo kernel, i thread del warp recuperano i dati tramite la texture memory per eseguire il calcolo dei contributi parziali al prodotto scalare. Una volta calcolati, questi vengono ridotti utilizzando la primitiva `__shfl_sync()`, in modo del tutto analogo ai kernel precedenti basati su riduzione warp-level.

L'utilizzo della texture memory può portare benefici significativi su GPU che supportano bene questa forma di caching, soprattutto in applicazioni sparse dove la locality spaziale è ridotta ma l'accesso a determinati pattern di dati si ripete frequentemente.

7.2 CUDA with HLL

7.2.1 kernel 0

Il kernel matvec Hll cuda SH esegue il prodotto matrice-vettore tra una matrice sparsa in formato HLL e un vettore d'ingresso. In particolare, sfrutta il parallelismo offerto da CUDA assegnando a ciascun thread la computazione di una singola riga della matrice. Il formato HLL suddivide la matrice in blocchi regolari di 32 righe, ottimizzando così gli accessi in memoria e facilitando la parallelizzazione.

La funzione configure grid warp si occupa di calcolare la configurazione della griglia CUDA in modo da bilanciare il carico di lavoro tra i thread e i multiprocessori della GPU. In particolare, stabilisce due parametri fondamentali: il numero di thread per blocco (threads) e il numero di blocchi (blocks).

Algorithm 12 Configurazione della griglia CUDA con `configure_grid_warp`

Require: Numero di righe della matrice M, numero di SM `sm_count`
Ensure: Numero di thread per blocco `threads`, numero di blocchi `blocks`
1: `threads` $\leftarrow$ valore minimo tra `THREADS_PER_BLOCK` e M
2: `blocks` $\leftarrow$ valore arrotondato per eccesso di $M/$`threads`
3: **if** (`blocks` $\times$ `threads`) $> M$ **then**
4: Ricalcola `blocks` come $M/$`threads` arrotondato per eccesso
5: **if** `blocks` non è multiplo di `sm_count` **then**
6: Arrotonda `blocks` al multiplo successivo di `sm_count`

All'interno del kernel CUDA, ogni thread identifica la sua riga associata in base al suo indice globale, che è calcolato come

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

ed è univoco. Successivamente per gestire la suddivisione in blocchi, si calcola

$$\texttt{block\_id} = \left\lfloor \frac{\texttt{global\_row}}{\texttt{HACK\_SIZE}} \right\rfloor$$

$$\texttt{local\_row} = \texttt{global\_row} \bmod \texttt{HACK\_SIZE}$$

dove block id invidua il blocco HLL in cui si trova la riga, e local row la riga all'interno del blocco. Grazie al fatto che la mappatura tra thread e riga è 1 on 1, ogni thread accede a

$$d_y[\texttt{global\_row}] = \texttt{sum}$$

e questo evita race condition su d_y. Questa è una versione iniziale, scelta per la sua semplicità logica che si adatta bene alla struttura di memorizzazione HLL, visto che i dati memorizzati sono righe contigue in memoria grazie all'organizzazione per blocchi ELLPACK.

7.2.2 Kernel 1

Il kernel matvec Hll cuda warp è l'evoluzione del kernel precedente, in questa variante assegnamo un warp a ciascuna riga della matrice, i singoli thread del warp cooperano per calcolare il prodotto scalare associato. La configurazione in questo caso prevede blocchi bidimensionali, la configurazione è scelta come

$$\texttt{blockDim.x} = 32 \qquad \texttt{blockDim.y} = \text{numero di warp per blocco}$$

. Il processo per effettuare il calcolo in questo caso segue i seguenti passi:

- Il calcolo dell'indice di riga ovviamente cambia rispetto al caso monodimensionale, avviene nel seguente modo:

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

- Ogni thread del warp esegue una porzione del prodotto scalare tra la riga della matrice e il vettore d'ingresso, seguendo uno stride pari alla dimensione del warp: for (int j = lane; j < max nz per row; j += WARP SIZE) do ...

- I risultati parziali in questo caso vengono combinati grazie alla shuffle down sync. La funzione shfl down sync consente la comunicazione tra i thread di uno stesso warp tramite scambio diretto di registri. Nel kernel, viene utilizzata per effettuare una riduzione parallela dei valori parziali calcolati da ciascun thread, sommando i contributi con offset successivamente dimezzati. Questo approccio evita l'utilizzo di memoria condivisa e garantisce una riduzione efficiente all'interno del warp, lasciando alla lane 0 la responsabilità di scrivere il risultato finale.

7.2.3 Kernel 2

Nel kernel matvec_Hll_cuda_warp_shared, l'operazione di riduzione dei risultati parziali non viene più effettuata tramite le primitive warp-level (__shfl_down_sync), ma sfruttando esplicitamente la shared memory. Questo approccio consente di accumulare i valori calcolati da ciascun thread del warp in un'area condivisa di memoria veloce, accessibile da tutti i thread del blocco. La shared memory viene allocata dinamicamente al lancio del kernel, riservando per ogni warp un blocco contiguo di 32 elementi:

$$\texttt{sharedSize} = \texttt{WARP\_SIZE} \times \texttt{warps\_per\_block} \times \texttt{sizeof(double)}$$

e ogni thread calcola il proprio indice all'interno di questa area condivisa come:

$$\texttt{s\_index} = \texttt{warpIdInBlock} \times \texttt{WARP\_SIZE} + \texttt{lane}$$

e scrive il proprio valore parziale in sdata[s_index]. I thread all'interno del warp eseguono poi una riduzione logaritmica classica, combinando i valori a coppie con offset decrescenti. Ad ogni iterazione, è necessaria una barriera di sincronizzazione (__syncthreads()) per garantire la correttezza della somma.

Algorithm 13 Riduzione logaritmica in shared memory

Require: `sdata` vettore in shared memory, `s_index` indice corrente del thread, `lane` ID della lane nel warp

```
for offset ← WARP_SIZE ÷ 2 to 1 step offset ÷ 2 do
    if lane < offset then
        sdata[s_index] ← sdata[s_index] + sdata[s_index + offset]
    __syncthreads()
```

Il comportamento di questo kernel è quello di emulare il funzionamento della shuffle del kernel precedente, ma esplicitandone il funzionamento.

7.2.4 Kernel 3

Il kernel matvec_hll_column_kernel implementa una versione del prodotto matrice-vettore in cui i dati della matrice HLL sono organizzati in column-major order all'interno dei blocchi, con l'obiettivo di massimizzare la coalescenza degli accessi alla memoria globale durante l'esecuzione su GPU. La coalescenza è una proprietà che consente a thread consecutivi di accedere a indirizzi contigui in memoria globale in un'unica operazione. Assegnando un thread per riga, i thread di un warp accedano a elementi consecutivi nella memoria della matrice, ottenendo accessi perfettamente coalescenti. Questo migliora drasticamente l'efficienza della memoria e, di conseguenza, le performance del kernel. Se invece i thread accedono a indirizzi sparsi o disallineati, ogni accesso richiede una transazione separata: la memoria viene usata in modo inefficiente e le performance crollano. Supponiamo di avere un blocco con 3 righe e 3 colonne non nulle per riga. I valori vengono disposti in memoria nel seguente ordine:

JA = [R0_C0, R1_C0, R2_C0, R3_C0, R0_C1, R1_C1, R2_C1, R3_C1, R0_C2, R1_C2, R2_C2, R3_C2]. Con questo schema, se ogni thread è assegnato a una riga della matrice, gli accessi ai valori per il prodotto matrice-vettore avverranno come segue:

- Thread 0 accede a R0_C0, R0_C1, R0_C2,
- Thread 1 accede a R1_C0, R1_C1, R1_C2,
- Thread 2 accede a R2_C0, R2_C1, R2_C2
- Thread 3 accede a R3_C0, R3_C1, R3_C2

Grazie alla disposizione column-major, i primi accessi (R0_C0, R1_C0, R2_C0, R3_C0) sono consecutivi in memoria e avvengono simultaneamente per i thread adiacenti, risultando in accessi coalescenti alla memoria globale. Lo stesso vale per le colonne successive.

Algorithm 14 Prodotto matrice-vettore con HLL column-major (`matvec_hll_column_kernel`)

Require: Matrice HLL `d_hll`, vettore input `d_x`, vettore output `d_y`, numero di righe `M`

```
1: global_row ← blockIdx.x × blockDim.x + threadIdx.x
2: if global_row ≥ M then return
3: block_id ← global_row ÷ HACK_SIZE
4: local_row ← global_row mod HACK_SIZE
5: block ← d_hll.blocks[block_id]
6: rows ← block.rows_in_block
7: max_nz ← block.max_nz_per_row
8: sum ← 0
9: for c ← 0 to max_nz −1 do
10:    idx ← c× rows + local_row                    ▷ Accesso column-major
11:    col ← block.JA[idx]
12:    val ← block.AS[idx]
13:    if col ≥ 0 then
14:       sum ← sum + val × d_x[col]
15: d_y[global_row] ← sum
```

L'indirizzo dell'elemento da leggere è calcolato come:

$$\texttt{idx} = c \times \texttt{rows} + \texttt{local\_row}$$

I valori della colonna c per tutte le righe del blocco sono memorizzati consecutivamente. Quando thread adiacenti (che processano righe consecutive) accedono alla stessa colonna c, i loro accessi saranno indirizzati a celle consecutive in memoria. Dopo l'accumulo del prodotto scalare, ciascun thread scrive il proprio risultato nel vettore d_y:

$$\texttt{d\_y[global\_row]} \leftarrow \texttt{sum}$$

Poiché ogni thread è responsabile di una riga diversa, non si verificano conflitti in scrittura: l'accesso è esclusivo e indipendente.

8 Results

8.1 Serial

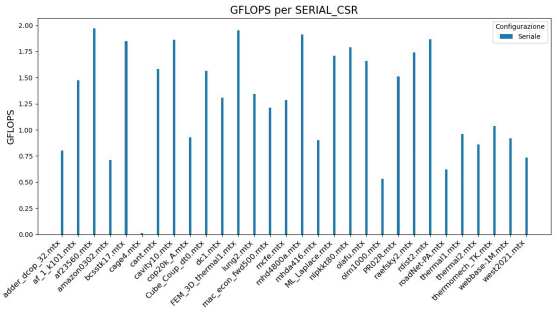


Figure 2: GFLOPS per SERIAL_CSR

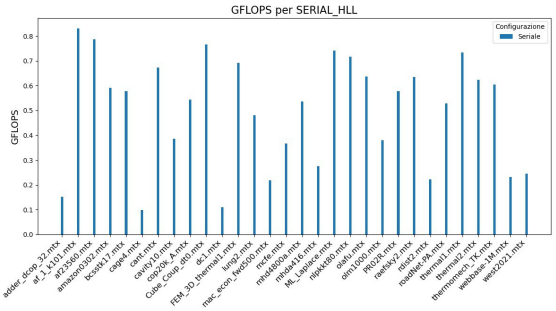


Figure 3: GFLOPS per SERIAL_HLL

L'analisi delle prestazioni in modalità seriale mette in evidenza una chiara superiorità del formato CSR rispetto a HLL. Come mostrano i risultati, le performance ottenute con CSR raggiungono picchi superiori a 2 GFLOPS, mentre HLL si attesta mediamente su valori inferiori a 1 GFLOPS.

Questa differenza è attribuibile al fatto che CSR, grazie alla sua struttura compatta e lineare, consente un accesso più diretto e localizzato alla memoria, risultando particolarmente efficiente in ambienti sequenziali. Al contrario, il formato HLL introduce un'organizzazione a blocchi con righe di lunghezza fissa, che in assenza di parallelismo comporta overhead dovuto al padding e alla gestione della struttura.

Va sottolineato che questo comportamento era atteso, in quanto HLL è progettato specificamente per sfruttare l'architettura delle GPU. In ottica CUDA, la regolarità del layout e la possibilità di accessi coalescenti compensano ampiamente le penalizzazioni introdotte in ambiente seriale.

In sintesi, i risultati confermano che CSR è la scelta ottimale per un'esecuzione seriale su CPU.

8.2 OpenMP Results

8.2.1 OpenMP HLL

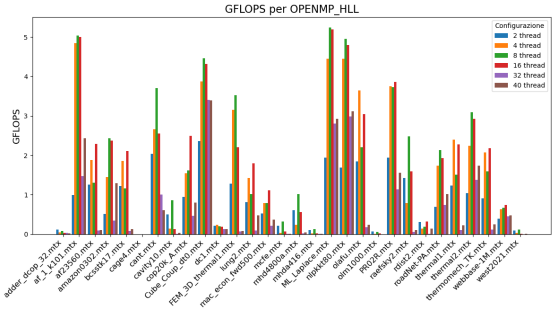


Figure 4: OpenMP HLL

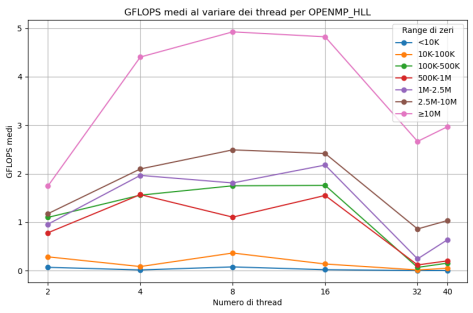


Figure 5: OpenMP AVG FLOP

Il grafico in Figura 4 mostra le prestazioni in GFLOPS dell'esecuzione del prodotto matrice-vettore (SpMV) con matrici nel formato HLL, utilizzando la parallelizzazione OpenMP su CPU. Dall'analisi si nota che, in media, le migliori prestazioni vengono raggiunte utilizzando 8 threads. Incrementare ulteriormente il numero di threads non porta necessariamente a migliori prestazioni: anzi, questo dipende molto dalla specifica matrice utilizzata.

La ragione principale per cui l'incremento dei threads non produce un aumento lineare delle prestazioni è legata all'aumento della complessità gestionale che ne deriva. All'aumentare del numero di threads, infatti, aumenta anche l'overhead dovuto sia alla gestione hardware della memoria condivisa, sia allo scheduling delle attività da parte di OpenMP (in particolare quando si utilizza la strategia "guided"). Con più threads attivi, il sistema deve infatti eseguire più frequentemente controlli per determinare quando ogni thread ha terminato il proprio compito e assegnargli nuove porzioni di lavoro. Questo overhead gestionale, accumulato nel tempo, finisce per annullare i potenziali vantaggi della parallelizzazione estrema.

Il grafico in Figura 5 mostra la media delle prestazioni ottenute dalle matrici, suddivise in gruppi sulla base del numero di elementi non nulli presenti. Dall'analisi si osserva che le migliori prestazioni vengono raggiunte con matrici che contengono oltre 10 milioni di elementi non-zero. Questo risultato è reso possibile dalla suddivisione della matrice in blocchi (hacksize) e dall'utilizzo dello scheduling di OpenMP, che garantisce un'elevata parallelizzazione del calcolo. Tale strategia risulta particolarmente efficace per matrici di grandi dimensioni, dove il vantaggio della parallelizzazione supera significativamente gli overhead di gestione del sistema.

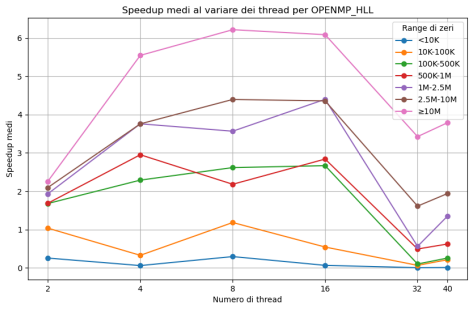


Figure 6: OpenMP AVG SpeedUp

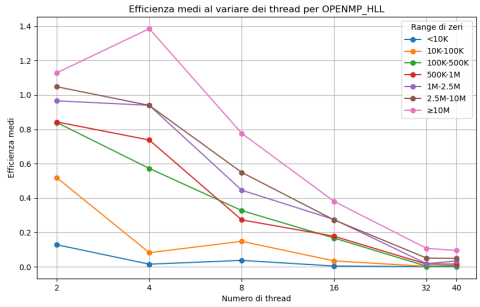


Figure 7: OpenMP AVG Efficiency prestazioni

Il grafico in Figura 6 mostra la media dello SpeedUp ottenuta sulle matrici, raggruppate in base al numero di elementi non nulli presenti. È evidente che lo SpeedUp cresce al crescere delle dimensioni delle matrici, raggiungendo generalmente il massimo valore medio con 8 threads. Oltre questo numero di threads, la curva dello SpeedUp tende a diminuire, indicando un peggioramento delle prestazioni dovuto probabilmente all'overhead della gestione parallela, fenomeno già evidenziato nei grafici precedenti.

Il grafico in Figura 7 presenta invece la media dell'Efficiency, anch'essa suddivisa per gruppi di matrici sulla base del numero di elementi non nulli. Contrariamente a quanto osservato per lo SpeedUp, l'Efficiency raggiunge in media il valore massimo con 2 threads. Questo indica che il guadagno prestazionale passando da un'esecuzione seriale a 2 threads è proporzionalmente più significativo rispetto a quello ottenuto aumentando ulteriormente il numero di threads. È inoltre rilevabile una

notevole diminuzione dell'Efficiency avvicinandosi ai valori di 32 e 40 threads, con valori che in alcuni casi risultano prossimi allo zero.

8.2.2 OpenMP CSR

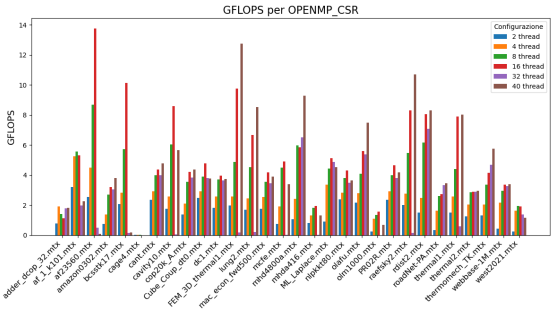


Figure 8: OpenMP CSR

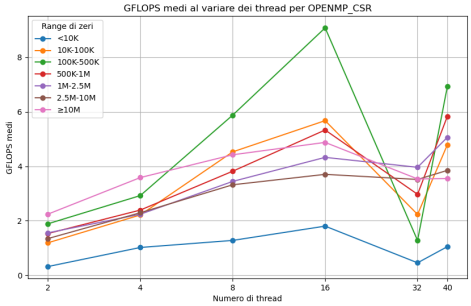


Figure 9: OpenMP AVG FLOP

Il grafico in Figura 8 mostra le prestazioni in GFLOPS dell'esecuzione del prodotto matrice-vettore (SpMV) con matrici nel formato CSR, utilizzando la parallelizzazione OpenMP su CPU e un bilanciamento del carico per threads implementato manualmente. Dall'analisi si nota che, in media, le migliori prestazioni vengono raggiunte utilizzando 16 threads. Anche la computazione con 40 threads sembra ottenere ottimi risultati soprattutto su matrici di medie dimensioni come `FEM_3D_thermal1.mtx` e `raefsky2.mtx`. Possiamo notare invece come per alcune matrici di medie o piccole dimensioni la computazione con 32 threads ottenga risultati mediocri rispetto a 16 o 40 threads. Questo può essere dovuto ad uno sbilanciamento nella gestione dei threads rispetto ai core fisici sottostanti provocando un uso non ottimale o eccessivamente concorrenziale della cache e della memory bandwidth oltre a dei possibili NUMA Effects ovvero l'accesso da parte di un thread alla memoria non unificata presente nell'altro socket. Con la computazione con 40 threads come già visto le performance tornano a salire rispetto ai 32threads, questo può essere dovuto ad una corretta gestione di carico di lavoro da parte del sistema sui 40 core fisici sottostanti schedulando i threads in modo più uniforme tra i due socket massimizzando così l'uso della cache e della memory bandwidth.

Il grafico in Figura 9 mostra la media delle prestazioni ottenute dalle matrici, suddivise in gruppi sulla base del numero di elementi non nulli presenti. Dall'analisi si osserva che le migliori prestazioni vengono raggiunte con matrici che contengono un numero di valori non nulli compresi tra 100 mila e 500 mila. Questo risultato è reso possibile probabilmente grazie alla dimensione non eccessiva della matrice e quindi al potenziale ottimo uso da parte dei threads alla memoria cache del processore evitando così continui accessi in RAM.

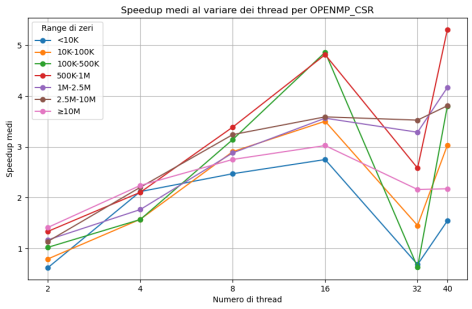


Figure 10: OpenMP AVG SpeedUp

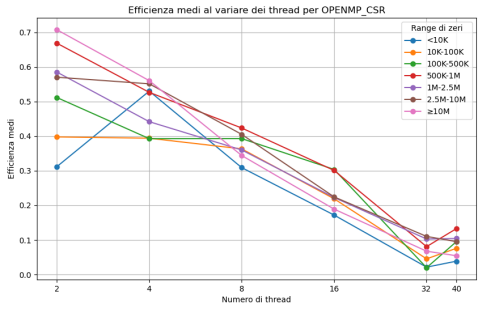


Figure 11: OpenMP AVG Efficiency

Il grafico in Figura 10 mostra la media dello SpeedUp ottenuta sulle matrici, raggruppate in base al numero di elementi non nulli presenti. Dall'analisi si nota come, in media, le matrici facenti parti del range di elementi non nulli compresi tra i 500 mila e 1 milione ottengano una crescita migliore all'aumentare dei threads rispetto agli altri range, raggiungendo il picco sui 40 threads. Ciò è indice del fatto che il rapporto tra la loro esecuzione seriale e quella parallela subisce una crescita maggiore rispetto alla crescita delle esecuzioni svolte sulle matrici facenti parti degli altri range. Il grafico in Figura 11 presenta invece la media dell'Efficiency, anch'essa suddivisa per gruppi di matrici sulla base del numero di elementi non nulli. Contrariamente a quanto osservato per lo SpeedUp, l'Efficiency raggiunge in media il valore massimo con 2 threads. Questo indica che il guadagno prestazionale passando da un'esecuzione seriale a 2 threads è proporzionalmente più significativo rispetto a quello ottenuto aumentando ulteriormente il numero di threads. È inoltre rilevabile una notevole diminuzione dell'Efficiency avvicinandosi ai valori di 32 e 40 threads, con valori risultanti prossimi allo zero.

8.2.3 CSR vs CSR-Guided

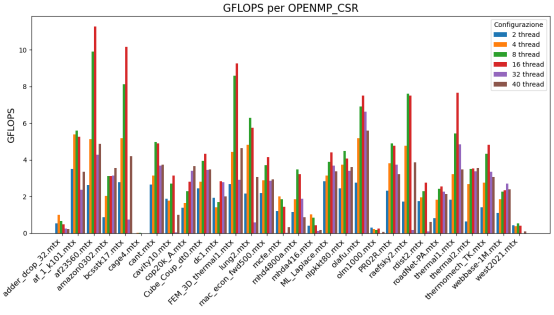


Figure 12: OpenMP CSR-Guided

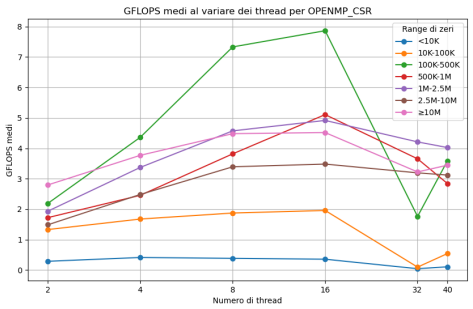


Figure 13: OpenMP AVG FLOP-Guided

I grafici in Figura 12 e in Figura 13 mostrano rispettivamente le prestazioni in GFLOPS dell'esecuzione del prodotto matrice-vettore (SpMV) e la media delle prestazioni ottenute dalle matrici suddivise in gruppi. Questa esecuzione nel formato CSR a differenza della precedente ha un bilanciamento del carico per threads svolto usando la primitiva messa a disposizione da OpenMP attraverso lo schedul-

ing "guided". Rispetto ai grafici in Figura 8 inerente all'esecuzione CSR con bilanciamento manuale, possiamo vedere come il bilanciamento messo a disposizione da OpenMP risulti essere meno performante in generale ma che rispetto al grafico in Figura 9 per le esecuzioni 2 threads, 4 threads e 8 threads la media delle prestazioni ottenute dalle matrici suddivise per gruppi raggiunge valori di picco maggiori.

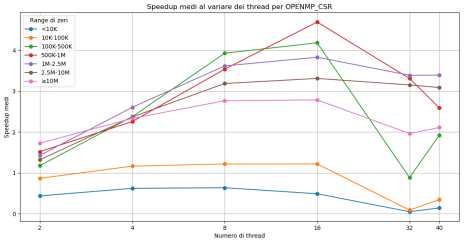


Figure 14: OpenMP AVG SpeedUp-Guided

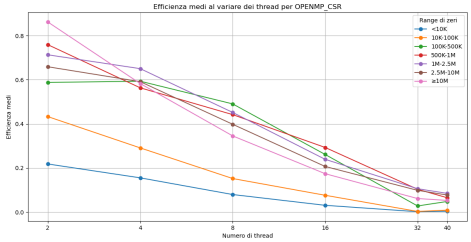


Figure 15: OpenMP AVG Efficiency-Guided

I grafici in Figura 14 e in Figura 15 mostrano rispettivamente la media dello SpeedUp e la media dell'Efficiency entrambe ottenute sulle matrici, raggruppate in base al numero di elementi non nulli presenti per l'esecuzione CSR con scheduling "guided". Possiamo osservare come rispetto al grafico in Figura 10 lo SpeedUp ottiene valori di picco maggiori per esecuzioni con 2 threads, 4 threads e 8 threads. Possiamo di conseguenza vedere come rispetto al grafico in Figura 11 il valore di picco raggiunto per le esecuzioni menzionate raggiunge valori superiori. In conclusione possiamo quindi dire che per le esecuzioni che vanno da 2 a 8 threads per la versione con schedule "guided" ottengono un guadagno prestazionale proporzionalmente maggiore rispetto alle stesse con bilanciamento del carico implementato manualmente.

8.3 CUDA Results

8.3.1 CUDA CSR

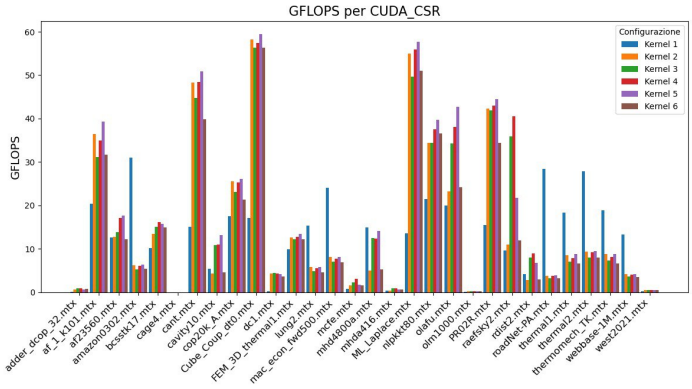


Figure 16: CUDA CSR

Il grafico mostra le prestazioni in GFLOPS di sei versioni del kernel CUDA applicate al formato CSR (Compressed Sparse Row), tutte basate su un layout row-major. L'obiettivo dell'esperimento è valutare come differenti strategie di parallelizzazione e ottimizzazione dell'accesso alla memoria influenzino le performance del prodotto matrice-vettore su GPU.

Kernel 1 rappresenta l'implementazione più semplice: un thread per riga, con accessi diretti alla memoria globale. Pur essendo facile da implementare, questa versione soffre di scarsa coalescenza degli accessi, specialmente in presenza di righe con pochi elementi, e mostra prestazioni inferiori rispetto alle versioni successive.

Kernel 2 adotta un modello a *warp per riga*, migliorando l'efficienza nella gestione delle operazioni di somma grazie alla parallelizzazione intra-warp attraverso l'uso della primitiva `__shfl_sync()`. Risultando in un modesto miglioramento.

Kernel 3 introduce la **shared memory** per accumulare i risultati parziali del prodotto, riducendo l'accesso alla memoria globale per il vettore risultato `y`. Questo kernel apporta benefici limitati in quanto per la maggior parte delle matrici i costi di gestione della memoria condivisa e la necessità costante di sincronizzare tra loro i threads può risultare in un overhead eccessivo rispetto al guadagno di cicli di accesso alla memoria.

Kernel 4 combina shared memory e riduzione tramite le primitive `__shfl_sync()`, ottenendo un buon equilibrio tra parallelizzazione e riduzione del traffico in memoria. Questa soluzione mostra in media ottime performance rispetto ai kernel precedenti grazie alla necessita di sincronizzare meno volte i threads e all'accesso non più alla memoria globale bensì a quella condivisa per realizzare la riduzione tramite la primitiva precedentemente menzionata.

Kernel 5 sfrutta la **cache L2** tramite l'uso della funzione `__ldg()` per leggere i dati della matrice e del vettore `x`, migliorando la latenza, qual'ora i dati siano presenti in cache, rispetto alla memoria globale. Questo approccio risulta essere in media il migliore specialmente per matrici di grandi dimensioni come `Cube_Coup`, `ML_Laplace`, e altre simili, grazie anche alla mancanza di latenza nel gestire la sincronizzazione tra i threads risultando meno invasivo rispetto alla memoria condivisa.

Kernel 6 carica il vettore `x` in **texture memory**, una cache specializzata progettata per pattern di accesso non coalescente. Questo approccio è pensato per migliorare i casi in cui gli accessi al vettore non seguono un ordine regolare. Le performance risultano competitive, anche se non sempre superiori alla cache L2, suggerendo che la texture memory è vantaggiosa solo in presenza di accessi altamente irregolari.

In generale, si osserva una tendenza chiara: passando da kernel 1 a kernel 6, le ottimizzazioni che riducono l'accesso ripetuto alla memoria globale e migliorano la località dei dati portano a incrementi significativi delle performance. Tuttavia, la scelta della tecnica migliore dipende dalla struttura specifica della matrice. Ad esempio, il kernel 4 è spesso il più efficiente, ma in alcuni casi il kernel 5 (cache L2) o il kernel 6 (texture) ottengono risultati superiori, confermando che non esiste un'unica soluzione ottimale per tutte le tipologie di input.

8.3.2 CUDA HLL

Il grafico mostra un confronto diretto tra quattro diverse versioni del kernel CUDA sviluppate per il prodotto matrice-vettore sul formato HLL. L'obiettivo dell'analisi è valutare l'impatto che diverse scelte progettuali, sia in termini di granularità computazionale che di layout in memoria, hanno sulle prestazioni misurate in GFLOPS.

Kernel 1 rappresenta la configurazione di base: un thread per riga e accesso ai dati in row-major. In questa versione, gli accessi alla memoria globale non sono coalescenti, portando a una bassa efficienza nella gestione del bandwidth. Come prevedibile, le prestazioni risultano modeste nella maggior parte

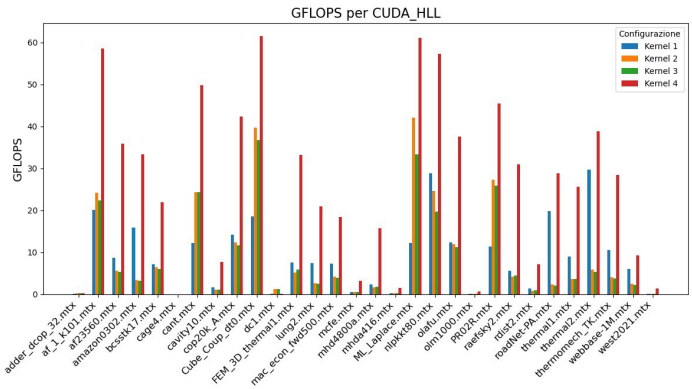


Figure 17: HLL

dei casi.

Kernel 2 introduce un primo miglioramento, assegnando un intero warp a ciascuna riga e utilizzando le primitive `__shfl_sync()` per realizzare la riduzione interna al warp. Questa configurazione migliora la parallelizzazione intra-warp e riduce il costo della somma dei prodotti.

Kernel 3 estende il modello precedente introducendo la **shared memory** come spazio di accumulo temporaneo per la riduzione dei risultati. Questa scelta offre maggiore flessibilità rispetto all'uso di primitive come `__shfl_sync()`, in quanto consente di gestire esplicitamente il flusso dei dati tra i thread. Tuttavia, i risultati mostrano che questa gestione manuale, pur essendo più controllabile, può risultare in alcuni casi meno efficiente rispetto alle ottimizzazioni automatiche offerte dalle primitive warp-level fornite da CUDA.

Kernel 4, infine, rappresenta la versione più ottimizzata. Si torna all'assegnazione thread-per-riga, ma con un cambio fondamentale: la matrice viene organizzata in formato column-major. Questo accorgimento consente agli accessi dei thread consecutivi (in una warp) di risultare coalescenti, permettendo alla GPU di sfruttare appieno la larghezza di banda della memoria globale. Il risultato è un incremento sostanziale delle performance, con picchi che superano i **60 GFLOPS**, più del doppio rispetto ai migliori risultati delle versioni precedenti.

L'evidenza sperimentale conferma che la *coalescenza degli accessi a memoria globale* rappresenta il fattore determinante nelle performance dei kernel CUDA applicati a strutture sparse. Più che la sola struttura del kernel (thread vs warp, riduzione con shared memory), è l'organizzazione fisica dei dati a determinare il salto di qualità.

In sintesi, questo confronto dimostra che:

- le ottimizzazioni architetturali (come warp-level) migliorano l'efficienza computazionale,
- il layout dati (column-major) porta al miglior utilizzo della bandwidth,

È opportuno sottolineare che i risultati più significativi si sono osservati nel caso di matrici di grandi dimensioni, come `Cube_Coup`, `ML_Laplace`, e altre simili. In questi scenari, il numero elevato di righe permette un utilizzo più efficiente delle risorse GPU, in quanto il costo di configurazione del kernel (allocazione dei thread, gestione della griglia, sincronizzazioni) viene distribuito su un carico di lavoro sufficientemente ampio.

Al contrario, per matrici di dimensioni ridotte, l'overhead associato al setup iniziale dei thread e dei blocchi rappresenta una porzione significativa del tempo totale di esecuzione. In questi casi, tale

costo non viene ammortizzato in maniera efficace durante il calcolo vero e proprio, penalizzando le prestazioni globali del kernel. Questo effetto è particolarmente evidente nei kernel con configurazioni più complesse (warp-level, shared memory), che introducono ulteriore logica di gestione interna.

8.4 Best HLL vs best CSR

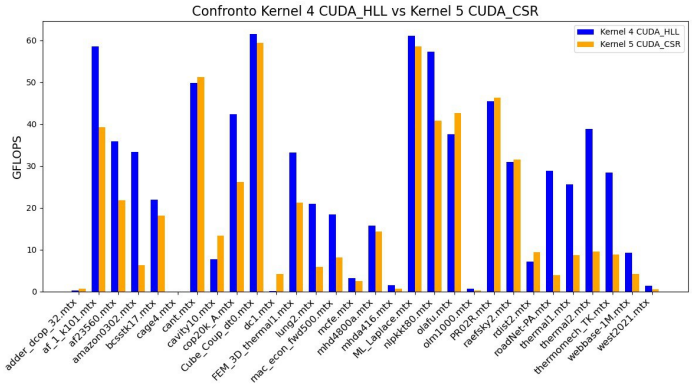


Figure 18: CSR vs HLL

Nel complesso, il kernel HLL risulta vincente, con prestazioni simili a CSR su matrici di grandi dimensioni e struttura regolare come `cant.mtx`, `Cube_Coup_dt0.mtx` e `ML_Laplace.mtx`, dove raggiunge picchi oltre i 60 GFLOPS. Questo risultato conferma la validità della scelta di un layout column-major combinato con accessi coalescenti, che permette alla GPU di sfruttare appieno la larghezza di banda della memoria globale.

Il kernel HLL mostra prestazioni più stabili e in alcuni casi molto superiori a CSR, come su `af_23560.mtx`, `cop20k_A.mtx`), `thermal2.mtx`).

È interessante osservare che, nonostante la maggiore flessibilità e compattezza del CSR, il formato HLL, spesso considerato subottimale in ambiente seriale, mostra risultati eccellenti in ambito CUDA, a patto che venga sfruttato con un layout adatto alla coalescenza.

In conclusione, il confronto dimostra che:

- Il formato HLL, se usato in column-major con accesso coalescente, può superare CSR su GPU grazie alla sua regolarità strutturale;
- Le ottimizzazioni mirate all'accesso alla memoria (coalescenza vs cache L2) sono determinanti per ottenere il massimo throughput.

9 Load balance

Il carico di lavoro è stato suddiviso in maniera omogenea, dopo una fase iniziale in cui c'è stata una stretta collaborazione per comprendere formati e punti di forza di questi. Di seguito la suddivisione:

- Preprocessamento dati, conversione in formato CSR ed HLL: Matteo Calzetta e Alessandro Lori
- Prodotti seriali nel formato CSR ed HLL: Matteo Calzetta ed Alessandro Lori

- OpenMP with CSR: Matteo Calzetta
- OpenMP with HLL: Alessandro Lori
- CUDA with CSR: Alessandro Lori
- CUDA with HLL: Matteo Calzetta
- JSON format for data storage: Matteo Calzetta e Alessandro Lori
- Graph plotting: Alessandro Lori
- Reporting: Matteo Calzetta e Alessandro Lori