Parallel Sparse Matrix-Vector Product

Martorelli Luca - 0357263
Cortese Alessandro- 0350035

Laurea Magistrale
Ingegneria Informatica

14 Aprile 2025

TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Roadmap

Descrizione del Problema

L'obiettivo del progetto è quello di sviluppare più nuclei di calcolo parallelo per il prodotto tra una matrice sparsa ed un vettore (SpVN). Problema esprimibile nel seguente modo:

$$y \leftarrow Ax$$

Dove A è una matrice sparsa.
La matrice viene rappresentata in uno dei seguenti formati:

- `CSR` - Compressed Storage Rows;
- `HLL` - Hacked ELLPACK

I risultati ottenuti con i nuclei di calcoli paralleli sono stati confrontati con una implementazione seriale, in particolare i risultati sono stati confrontati con la soluzione ottenuta con il calcolo seriale utilizzando il formato `CSR`.

Raccolta dei dati

I dati sono stati raccolti sul server di dipartimento avente le seguenti specifiche:

- CPU: Intel Xeon Silver 4210;
- GPU: NVIDIA RTX5000

Il processore è un 10 core 20 thread. Il server dispone di un dual socket, il che porta il numero di core fisici a 20 e il numero di thread logici a 40.
Ogni confgurazione di nucleo di calcolo viene iterata per ottenere le prestazioni medie.
Ogni configurazione dei kernel CUDA è stata eseguita con un numero prefissato di thread.

Conversione (1)

Le matrici di collaudo utilizzate sono salvate su file nel formato Matrix Market. La loro lettura avviene tramite le apposite funzioni di libreria che consente la ricostruzione corretta della matrice nel formato utilizzato.
Le matrici salvate possono essere:

- **simmetriche**: su file viene memorizzato solo un triangolo, in memoria vengono ricostruiti gli array di indici di riga, colonna e i valori della matrice per poi rifare il ricalcolo del numero di valori diversi da zero;
- **general**: su file vengono memorizzati i valori non nulli indicando anche l'indice di riga e colonna;
- **pattern**: come nel caso di general, ma non viene indicato il coefficiente dell'elemento, pari ad 1.0. In questo caso, al momento della ricostruzione, viene impostato manualmente il valore dell'elemento.

Conversione (2)

Le matrici vengono prima salvate in un formato il più generale possibile in grado di catturare le caratteristiche più importanti della stessa, formato contenente:

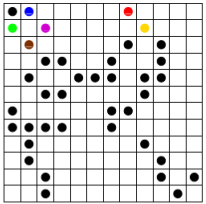
- array degli indici di riga;
- array degli di colonna;
- array dei coefficienti non nulli;
- numero di elementi non nulli.

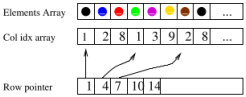
Una volta ottenuta questa rappresentazione la matrice viene convertita nel formato di memorizzazione `CSR` e `HLL` utilizzati per l'operazione.
Per quanto riguarda il vettore utilizzato x per il prodotto, esso viene generato per ogni matrice processata della dimensione opportuna.
I coefficienti del vettore vengono generati psuedorandomicamente grazie alla funzione `rand()`.

CSR

Il formato di memorizzazione CSR rappresenta una matrice MxN con NZ non-zeri nel seguente modo:

- **M**: numero di righe;
- **N**: numero di colonne;
- **IRP(1: M+1)**: vettore dei puntatori all'inizio di ciascuna riga;
- **JA(1:NZ)**: vettore degli indici di colonna;
- **AS(1:NZ)**: vettore dei coefficienti.





HLL (1)

Il formato HLL può essere definito in questo modo:

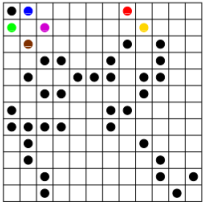
- si stabilisce un parametro **HackSize**, nel caso di studio vale 32;
- si partiziona la matrice di input in blocchi di HackSize righe;
- ciascun blocco viene memorizzato in formato **ELLPACK**, descritto nel seguito;
- i blocchi vengono memorizzati in una struttura dati da determinare.

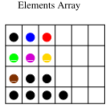
HLL (2)

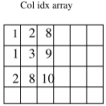
Una matrice $M \times N$ con NZ non-zeri e tale che $MAXNZ$ sia il massimo numero di nonzeri per riga, su tutte le righe, viene descritta come segue:

- **M**: numero di righe;
- **N**: numero di colonne;
- $\boldsymbol{MAXNZ}$: massimo di valori non-zeri per riga;
- **JA(1:M,1:MAXNZ)**: array bidimensionale di indici di colonna;
- **AS(1:M,1:MAXNZ)**: array bidimensionale di coefficienti.

Bisogna considerare che se le righe dovessero avere un numero di non-zeri inferiore a $MAXNZ$, ovvero il numero massimo di nonzeri per riga all'interno di un blocco, devono essere riempite con coefficienti appropriati, ossia zeri in AS.







HLL (3)

Per ottimizzare gli accessi in memoria, sia per quanto l'esecuzione con la CPU che con la GPU, si è deciso di utilizzare array monodimensionali piuttosto che delle matrici per memorizzare gli indici di colonna e i coefficienti della matrice.
In questo modo si è cercato di mantenere i coefficienti in maniera contigua in memoria in grado di avere una contiguità degli accessi per un accesso lineare per ottenere un vantaggio prestazionale.
Un altro vantaggio che si ottiene utilizzando questo formato è quello di evitare problemi legati alla discontinuità nella memoria relativa ad array bidimensionali dinamici.

```
1  typedef struct
2  {
3      char name[256];
4      int M;
5      int N;
6      int *offsets;
7      int offsets_num;
8      int *col_index;
9      double *data;
10     int data_num;
11     int hacks_num;
12     int hack_size;
13     int *max_nzr;
14 } HLL_matrix;
```

Metriche Prestazionali (1)

Vengono utilizzate le seguenti metriche per i risultati sperimentali:

- **Flops**: viene misurata tramite la seguente formula:

Performance:

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

Dove:

- NZ: rappresenta il numero di non-zeri della matrice;
- T: tempo di esecuzione dell'implementazione in secondi.

Metriche Prestazionali (2)

- **SpeedUp**: confronta l'efficienza dell'esecuzione parallela rispetto a quella seriale:

Speedup:

$$S(W,p) = \frac{T_s(W)}{T_p(W,p)}$$

Dove:

- W: rappresenta la dimensione del problema;
- p: indica il numero di processori utilizzati;
- $T_s(W)$: indica il tempo utilizzato per l'esecuzione seriale;
- $T_p(W,p)$: indica il tempo di esecuzione parallela utilizzando p processori.

- **Efficienza**:

$$\frac{Speedup}{NumThreads}$$

Misurazione dell'Errore

Per valutare la precisione dell'esecuzione parallele rispetto alla versione sequenziale si utilizza una funzione che misura l'errore del risultato appena ottenuto con un riferimento noto.
L'errore viene calcolato utilizzando la seguente formula:

Norma:

$$s = \frac{1}{n} \sum_{i=0}^{n-1} \delta_i, \quad \text{con} \quad \delta_i = \begin{cases} |z_i - y_i|, & \text{se } |z_i - y_i| \geq \varepsilon \\ 0, & \text{altrimenti} \end{cases}$$

Dove:

- z è il vettore ottenuto dal calcolo parallelo del prodotto matrice-vettore;
- y è il vettore di riferimento;
- n è la dimensione del vettore;
- ϵ è una soglia di tolleranza al di sotto della quale le differenze vengono ignorate.

Calcolo Seriale - CSR

Il ciclo esterno ha il compito di iterare sistematicamente su tutte le righe della matrice M, mentre all'interno di ciascuna iterazione, il ciclo interno si occupa di attraversare gli elementi non nulli presenti nella riga corrente.
Per ogni elemento, viene calcolato il prodotto tra il valore memorizzato nella matrice e il corrispondente valore del vettore denso x, così da contribuire progressivamente alla costruzione del risultato finale.

Algorithm 1 SpMV(y, irp, ja, as, x)

#y: vettore risultante
#irp: vettore dei puntatori all'inizio di ciascuna riga
#ja: vettore degli indici di colonna
#as: vettore dei coefficienti
#x: vettore colonna

for $i = 1$ **to** m **do**
  $t \leftarrow 0$
  for $j = irp(i)$ **to** $irp(i+1) - 1$ **do**
    $t \leftarrow t + as(j) \times x(ja(j))$
  end for
  $y(i) \leftarrow t$
end for

Calcolo Seriale - HLL (1)

L'algoritmo esegue il prodotto matrice-vettore per una matrice sparsa in formato ELLPACK.
Il ciclo esterno attraversa ogni riga della matrice, inizializzando un accumulatore t a zero. Il ciclo interno, fino al massimo numero di non-zeri per riga (`maxnzr`), moltiplica ciascun coefficiente `as(i, j)` per il valore corrispondente del vettore denso x, individuato tramite `ja(i, j)`.
Il risultato di ogni moltiplicazione viene sommato nell'accumulatore t.
Una volta completato il ciclo interno, il valore di t, che rappresenta la somma dei contributi della riga corrente, viene assegnato alla corrispondente posizione nel vettore risultato y. Il processo si ripete per tutte le righe della matrice.

Algorithm 2 SpMV(y, maxnzr, ja, as, x)

#y: vettore risultante
#maxnzr: massimo di valori non-zeri per riga
#ja: vettore degli indici di colonna
#as: vettore dei coefficienti
#x: vettore colonna

for $i = 1$ **to** m **do**
  $t \leftarrow 0$
  for $j = 1$ **to** $maxnzr$ **do**
    $t \leftarrow t + as(i, j) \times x(ja(i, j))$
  end for
  $y(i) \leftarrow t$
end for

Calcolo Seriale - HLL (2)

Utilizzando la struttura indicata precedentemente, il prodotto seriale con il formato HLL può essere eseguito nel seguente modo:

```
1  void matvec_serial_hll(HLL_matrix *hll_matrix, double *x, double *y)
2  {
3      int rows = num_of_rows(hll_matrix, 0);
4      for (int h = 0; h < hll_matrix->hacks_num; ++h)
5      {
6          for (int r = 0; r < num_of_rows(hll_matrix, h); ++r)
7          {
8              double sum = 0.0;
9              for (int j = 0; j < hll_matrix->max_nzr[h]; ++j)
10             {
11                 int k = hll_matrix->offsets[h]+ r * hll_matrix->max_nzr[h] + j;
12                 sum += hll_matrix->data[k] * x[hll_matrix->col_index[k]];
13             }
14             y[rows * h + r] = sum;
15         }
16     }
17 }
```

Calcolo Parallelo OMP - CSR (1)

- Per il prodotto matrice-vettore, in OMP, si adotta un approccio a più fasi per sfruttare il parallelismo in modo ottimale.
- **Obiettivo:** bilanciare il carico di lavoro tra i thread in base alla distribuzione dei non-zeri.
- La funzione `matrix_partition` calcola una partizione dinamica delle righe della matrice.

Passaggi principali:

- Conteggio dei non-zeri per ogni riga: `row_non_zero_count`.
- Calcolo del carico target per thread:

$$\texttt{target\_workload} = \frac{\texttt{total\_non\_zero\_elements}}{\texttt{num\_threads}}$$

- Assegnazione dinamica delle righe ai thread, mantenendo il carico vicino al target.
- Eventuale ridimensionamento del vettore `temp_start_row_indices` in base ai thread usati.

Calcolo Parallelo OMP - CSR (2)

- Il prodotto matrice-vettore viene eseguito nella funzione `product`.
- Il parallelismo è abilitato con:
  - `#pragma omp parallel`
- Il numero di thread effettivamente utilizzati è determinato da `actual_num_threads`.

Misurazione delle performance:

- Utilizzo di `omp_get_wtime()` per misurare il tempo di esecuzione.
- Analisi dell'efficienza del bilanciamento carico tramite il tempo rilevato.

Algorithm 4 product(csr_matrix, x, y, num_threads, execution_time, first_row)

#*csr_matrix*: Matrice CSR
#*x*: Vettore che effettua il prodotto
#*y*: Vettore risultato del prodotto
#*num_threads*: Numero di thread
#*execution_time*: Tempo di esecuzione del prodotto
#*first_row*: Tabella di assegnazione delle righe ai thread

Inizializzazione $start\_time \leftarrow Tempo\ iniziale$
Creazione di un insieme di num_threads thread per il parallelismo

$thread\_id \leftarrow ID\ del\ thread$

for $row = 1$ **to** $first\_row[thread\_id + 1] - 1$ **do**
  $sum \leftarrow 0$
  for $idx = csr\_matrix- > IRP[row]$ **to** $csr\_matrix- > IRP[row + 1]$ **do**
    $sum \leftarrow sum + csr\_matrix- > AS[idx] * input\_vector[csr\_matrix- > JA[idx]]$
  end for
  $output\_vector[row] \leftarrow sum$
end for

$end\_time \leftarrow Tempo\ finale$
$*execution\_time \leftarrow end\_time\ -\ start\_time$

Calcolo Parallelo OMP - HLL (1)

- Rispetto al formato `CSR`, con il formato `HLL` si è deciso di non utilizzare un approccio sulla divisione del carico manuale, ma di utilizzare le direttive fornite da OpenMP.
- Viene utilizzata la direttiva `pragma omp parallel for schedule(SCHEDULE_TYPE)` assegnando al parametro `SCHEDULE_TYPE` il valore `guided`.
  In questo modo le iterazioni vengono assegnate ai thread in modo dinamico. I blocchi assegnati ai thread sono composti da righe disgiunte in modo tale che i thread non abbiano bisogno di sincronizzazione.
- Viene utilizzata la direttiva `shared(hll_matrix, x, y, rows)` per indicare quali variabili sono condivise tra thread utilizzati per il calcolo in modo da non fare delle copie superflue.

Calcolo Parallelo OMP - HLL (2)

- Il parallelismo è abilitato con `#pragma omp parallel`.
- Il numero di thread effettivamente utilizzati è determinato da `num_threads`.
- Per l'effettivo calcolo dei coefficienti è stato utilizzata la direttiva `#pragma omp simd reduction(+ : sum)`.
- Viene utilizzata la funzione `__builtin_prefetch()` in modo da ridurre il tempo di latenza di accesso ai dati.
- Come nel caso di CSR, le tempistiche vengono raccolte con l'API `omp_get_wtime()`.

Algorithm 5 hll_parallel_product(hll_matrix, x, y, num_threads, time_used)

#hll_matrix: matrice nel formato HLL
#x: Vettore di input
#y: Vettore di output
#num_threads: Numero di thread da utilizzare
#time_used: Tempo di esecuzione misurato

$rows \leftarrow num\_of\_rows(hll\_matrix, 0)$
$start \leftarrow$ tempo corrente

Parallelizzazione: Parallel for con **num_threads**, strategia **guided**
for $h = 0$ **to** $hll\_matrix.hacks\_num - 1$ **do**
  $rows\_in\_h \leftarrow num\_of\_rows(hll\_matrix, h)$
  $max\_nzr\_h \leftarrow hll\_matrix.max\_nzr[h]$
  $offset\_h \leftarrow hll\_matrix.offsets[h]$
  for $r = 0$ **to** $rows\_in\_h - 1$ **do**
    $sum \leftarrow 0.0$
    $data\_ptr \leftarrow hll\_matrix.data[offset\_h + r * max\_nzr\_h]$
    $col\_ptr \leftarrow hll\_matrix.col\_index[offset\_h + r * max\_nzr\_h]$
SIMD per il prodotto scalare con prefetching
    for $j = 0$ **to** $max\_nzr\_h - 1$ **do**
      Prefetch di $data\_ptr[j + 4]$ e $col\_ptr[j + 4]$
      $sum \leftarrow sum + data\_ptr[j] * x[col\_ptr[j]]$
    end for
    $y[rows * h + r] \leftarrow sum$
  end for
end for
$end \leftarrow$ tempo corrente $*time\_used \leftarrow end - start$

csr_matvec_kernel

- **Obiettivo:** Ogni thread è responsabile dell'elaborazione di una singola riga della matrice, recuperando tutti gli elementi dalla memoria globale e calcolando il prodotto scalare tra la riga e il vettore. Il risultato viene poi salvato direttamente nella memoria globale.

Il kernel ha la seguente configurazione:

Dimensione Griglia:

$$\texttt{grid\_dim1} = \frac{M + \texttt{num\_threads\_per\_block} - 1}{\texttt{num\_threads\_per\_block}}$$

Calcolo Indice:

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

csr_matvec_shfl_reduction

- **Obiettivo:** ogni warp calcola una riga della matrice CSR.

$$\texttt{blockDim} = (\texttt{WARP\_SIZE}, \texttt{num\_threads\_per\_block})$$

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

- **Configurazione thread:**

```
row = blockIdx.x * blockDim.y + threadIdx.y
lane = threadIdx.x (posizione nel warp)
```

- **Calcolo locale:** Ogni thread somma prodotti parziali sugli elementi della riga.
- **Riduzione nel warp:** `__shfl_sync` somma i risultati parziali. Solo il `primo` thread scrive il risultato finale nella memoria globale.

Calcolo Parallelo CUDA - Kernel 3 CSR

csr_matvec_shared_memory

- **Obiettivo:** sfruttare la **shared memory** per ridurre la pressione sulla memoria globale e velocizzare la riduzione dei risultati parziali.
  - **1. Identificazione riga:** Ogni thread determina la riga con:

```
row = blockIdx.x * blockDim.y + threadIdx.y
```

  - **2. Calcolo locale:** Ogni thread calcola prodotti parziali su intervalli regolari:

```
j += WARP_SIZE
```

  - **3. Uso della shared memory:** Scrittura dei parziali in shared memory e riduzione gerarchica, sincronizzando i thread (`__syncthreads()`).
  - **4. Scrittura risultato:** Solo il primo thread scrive il risultato finale in memoria globale.
- **Nota:** la shared memory migliora la riduzione, ma gli accessi alla memoria globale restano solo parzialmente coalescenti.

Calcolo Parallelo CUDA - Kernel 4 CSR

csr_matvec_warp_cacheL2

- **Obiettivo:** Moltiplicazione matrice-vettore in formato CSR sfruttando parallelismo a livello di warp e cache L2.
- **Accessi alla memoria:**
  - Uso di puntatori locali per velocizzare l'accesso alle strutture `AS` e `JA`.
  - Accesso ai dati con `__ldg()` per sfruttare la cache L2.
- **Riduzione parziale nel warp:**
  - Somma dei prodotti parziali tramite `__shfl_sync()`.
- **Scrittura del risultato:**
  - Solo il primo thread del warp scrive il risultato finale nel vettore `y`.

Calcolo Parallelo CUDA - Kernel 1 HLL

cuda_hll_kernel_v1

- Ogni thread utilizzato è responsabile del processamento di un hack della matrice.
- Si utilizza la memoria globale per accedere alle informazioni della matrice.
- La dimensione della griglia viene calcolata nel seguente modo:

Dimensione Griglia

$$\texttt{grid\_dim} = \frac{M + \texttt{num\_threads\_per\_block} - 1}{\texttt{num\_threads\_per\_block}}$$

- Ogni thread calcola il proprio indice nel seguente modo:

Calcolo Indice

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

cuda_hll_kernel_v2

- Ogni thread è responsabile del processamento di una riga della matrice.
- Viene utilizzato la memoria globale.
- La dimensione della griglia viene calcolata nel seguente modo:

Dimensione Griglia

$$\texttt{grid\_dim} = \frac{M + \texttt{block\_dim.y} - 1}{\texttt{block\_dim.y}}$$

Dove `block_dim.y` rappresenta il numero di warp per blocco.

- Ogni thread calcola il proprio indice nel seguente modo:

Calcolo Indice

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

Calcolo Parallelo CUDA - Kernel 3 HLL (1)

cuda_hll_kernel_v3

- Ogni thread è responsabile del processamento di una riga della matrice.
- Viene utilizzato la memoria condivisa per ridurre le latenze di accesso alla memoria.
- La dimensione della griglia viene calcolata nel seguente modo:

Dimensione Griglia

$$\texttt{grid\_dim} = \frac{M + \texttt{block\_dim.y} - 1}{\texttt{block\_dim.y}}$$

Dove `block_dim.y` rappresenta il numero di warp per blocco.

- Ogni thread calcola il proprio indice nel seguente modo:

Calcolo Indice

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

Calcolo Parallelo CUDA - Kernel 3 HLL (2)

cuda_hll_kernel_v3

- Con `__syncthreads()` si garantisce che tutti i thread abbiano caricato i dati nella memoria condivisa.
- Ogni thread effettua la riduzione in modo locale per un elemento del vettore risultante, non ci sono riduzioni parallele o gloabali tra thread in modo che ogni thread lavori in modo indipendente.

Calcolo Parallelo CUDA - Kernel 4 HLL (1)

cuda_hll_kernel_v4

- Ogni warp è responsabile del processamento di una riga della matrice.
- Si utilizza una riduzione intra-warp
- Viene utilizzato la memoria globale.
- La dimensione della griglia viene calcolata nel seguente modo:

Dimensione Griglia

$$\texttt{grid\_dim} = \frac{M \times 32}{\texttt{thread\_used}}$$

Dove `M` è il numero di righe della matrice.

Calcolo Parallelo CUDA - Kernel 4 HLL (2)

cuda_hll_kernel_v4

- Ogni thread calcola il proprio indice nel seguente modo:

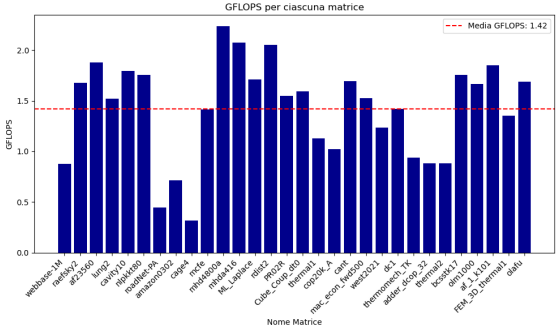
Calcolo Indice

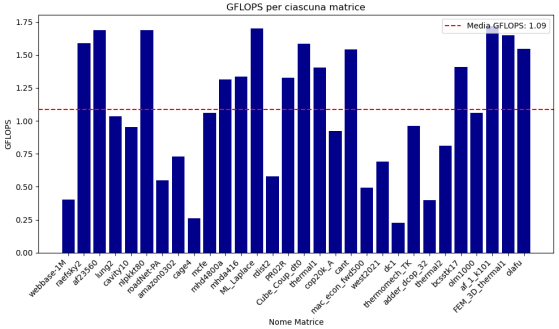
$$\texttt{index} = \texttt{blockIdx.x} \times \texttt{blockDim.x} + \texttt{threadIdx.x}$$

L'indice viene utilizzato per calcolare l'indice del warp e la riga su cui andare ad operare.

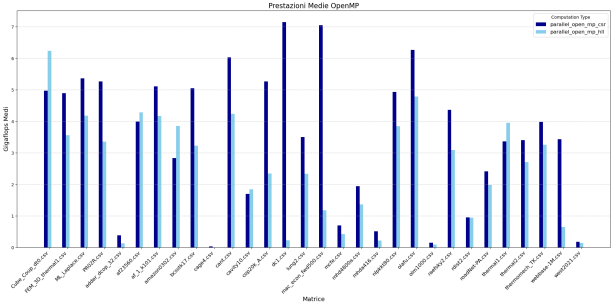
- Viene utilizzata la funzione `__shufl_down_sync()` per la riduzione a livello di warp per combinare i risultati parziali dei thread senza utilizzare la memoria condivisa.
- Solo il thread con indice `0` all'interno di uno warp accede in scrittura nel vettore risultante.
- Per il calcolo degli indici vengono utilizzati gli operatori bitwise del `C`

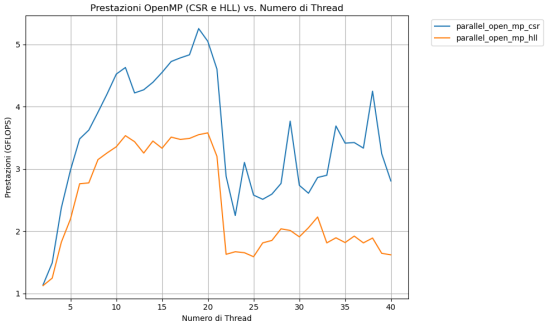
Risultati - Calcolo Seriale



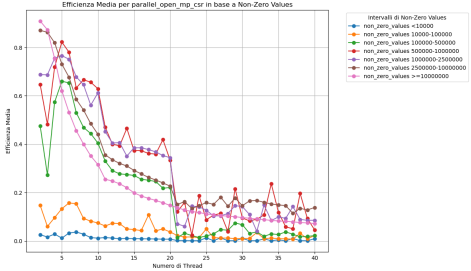


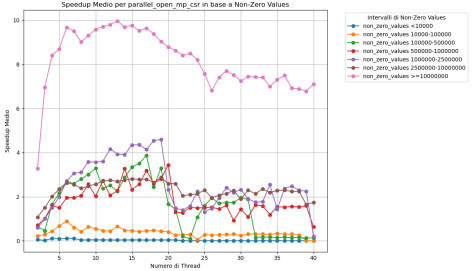
Risultati - Calcolo Parallelo OMP (1)

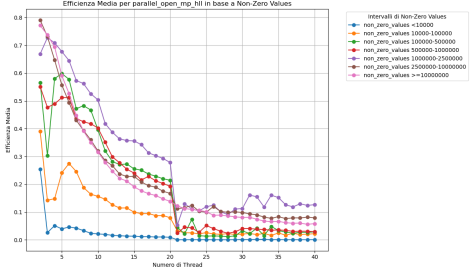


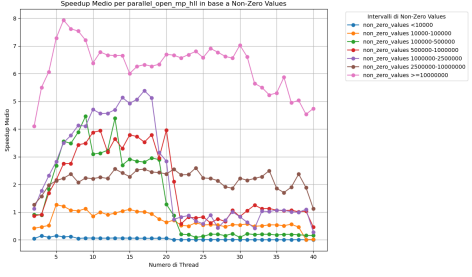


Risultati - Calcolo Parallelo OMP (2)

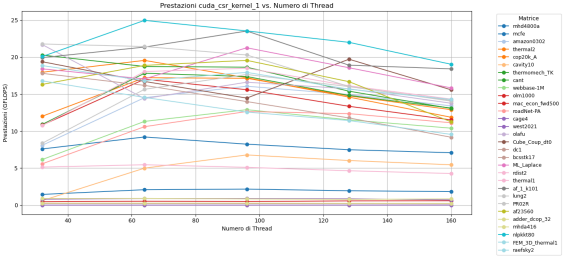


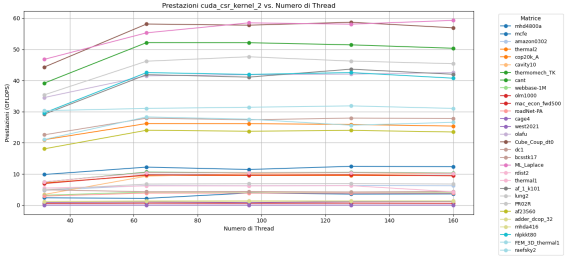


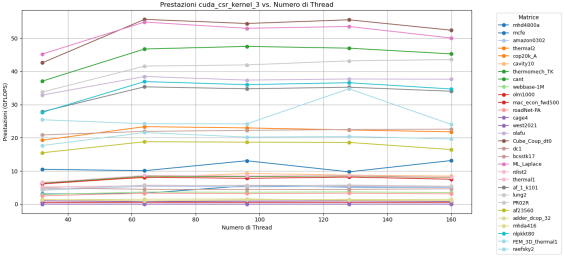


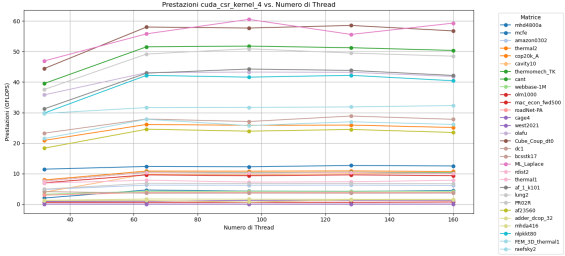


Risultati - Calcolo Parallelo CUDA CSR

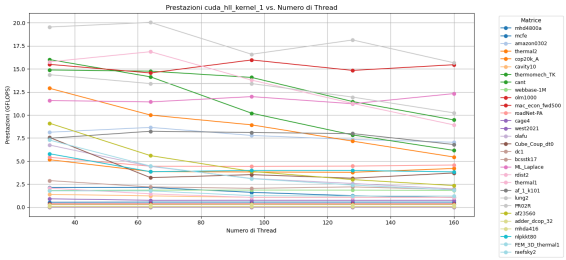


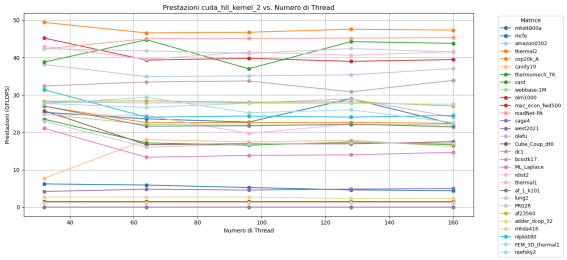


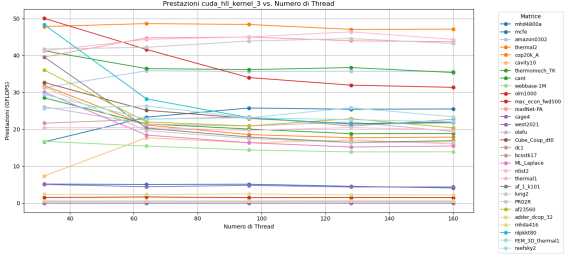


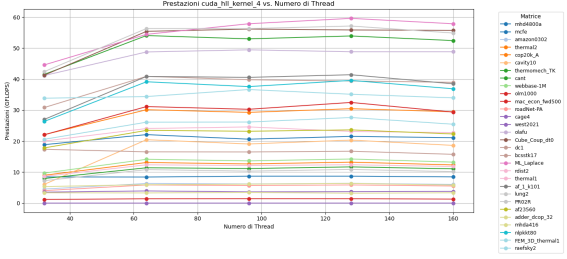


Risultati - Calcolo Parallelo CUDA HLL

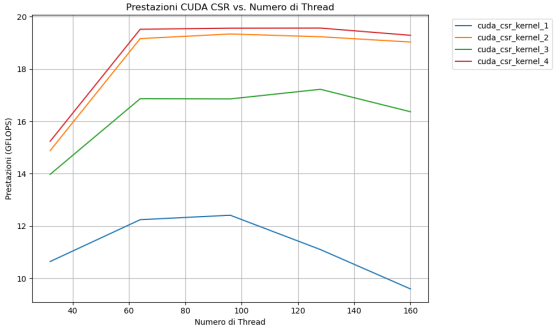


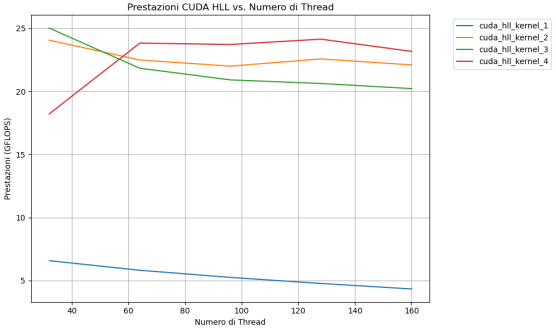




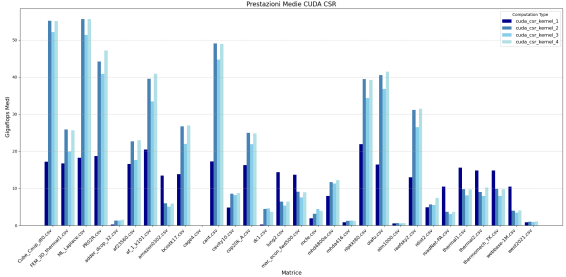


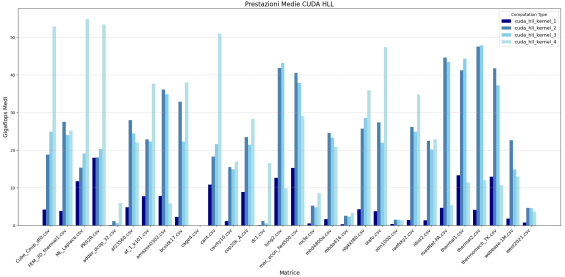
Risultati - Calcolo Parallelo CUDA Overall (1)



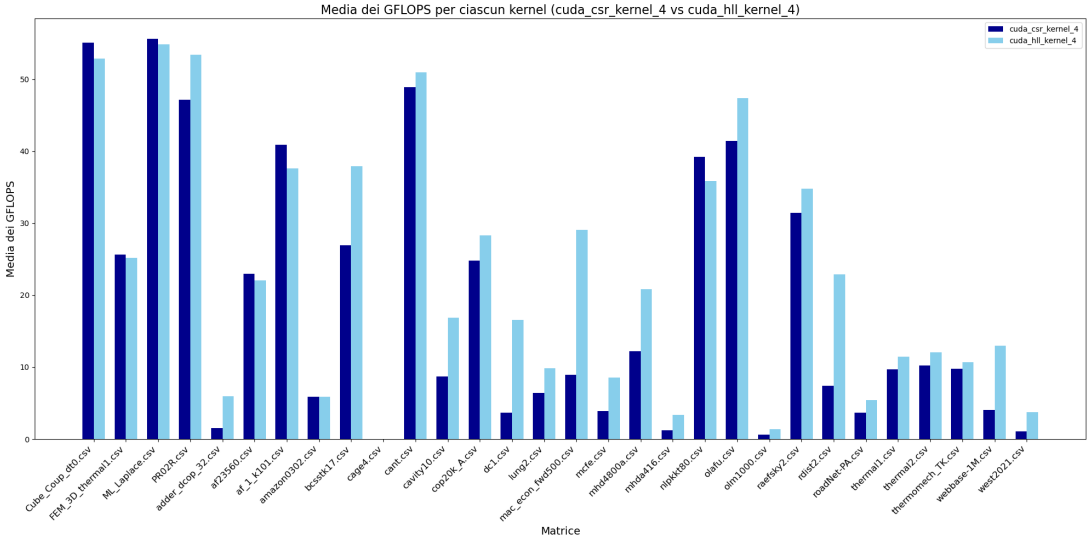


Risultati - Calcolo Parallelo CUDA Overall (2)





Risultati - Calcolo Parallelo CUDA Best Configurations



Grazie per l'attenzione

Luca Martorelli, Alessandro Cortese