Sparse Matrix - Multivector product

Daniele La Prova, Luca Fiscariello

February 2023

1 Introduzione

Il progetto ha come obiettivo quello di implementare un prodotto tra una matrice sparsa e un multivettore. In algebra lineare, una matrice sparsa è una matrice in cui la maggior parte degli elementi è uguale a zero. In altre parole, una matrice è considerata sparsa se ha un numero relativamente piccolo di elementi non nulli rispetto al numero totale di elementi. Le matrici sparse hanno alcune proprietà uniche rispetto alle matrici dense , in particolare nella loro rappresentazione e nella loro manipolazione. Infatti, poiché la maggior parte degli elementi è uguale a zero, rappresentare una matrice sparsa utilizzando un formato di memorizzazione tradizionale sarebbe inefficace in termini di utilizzo della memoria e di velocità di esecuzione delle operazioni. In questo progetto sono stati implementati diversi formati di memorizzazione di matrici sparse.

1. `ELLPACK` : Nel formato ELLPACK, una matrice viene rappresentata come una combinazione di due parti: una matrice di valori e una matrice di indici. La matrice di valori contiene i valori non nulli della matrice originale, mentre la matrice degli indici tiene traccia delle posizioni di questi valori nella matrice originale.

2. `CSR` : In questo formato, una matrice viene rappresentata da tre array: un array di valori non nulli, un array di indici di colonna e un array di puntatori di riga. L'array di valori non nulli contiene tutti i valori diversi da zero della matrice, ordinati per riga. L'array di indici di colonna contiene gli indici di colonna corrispondenti ai valori non nulli nell'array di valori. Infine, l'array di puntatori di riga indica l'inizio delle colonne che appartengono a ogni riga.

3. `COO` : In questo formato, una matrice viene rappresentata da tre array: un array di valori non nulli, un array di indici di riga e un array di indici di colonna. L'array di valori non nulli contiene tutti i valori diversi da zero della matrice, ordinati per ordine di apparizione nella matrice. L'array di indici di riga e di colonna contengono rispettivamente gli indici di riga e di colonna corrispondenti ai valori non nulli nell'array di valori;

4. `MatrixMarket`: Tale formato non è altro che un involucro a un puntatore a un `FILE *` che rappresenta la matrice MatrixMarket su file. Tale formato è in grado di leggere matrici reali, simmetriche, skew e pattern e di operare su di esse, oltre a leggerne il banner per scoprirne dimensioni e numero di non-zero. È possibile usare tale formato come intermedio per poter convertire una matrice su file in uno dei formati precedentemente illustrati;

Per i primi due formati sono stati realizzati i prodotti in Cuda e OpenMP cercando di ottimizzare per quanto possibile le prestazioni.
Sono stati implementati inoltre dei formati per la memorizzazione delle matrici dense:

1. `MultiVector`: un formato che memorizza un multivettore in un array denso bidimensionale, in cui ogni riga memorizza un puntatore a un array le cui posizioni indicano le colonne;

2. `ArrayDense`: un formato che memorizza una matrice densa in un array unidimensionale. Molto utile per memorizzare il risultato di implementazioni di prodotti tra matrici che coinvolgono l'uso di thread paralleli, ovvero OMP e CUDA, poiché thread safe.

2 Metodologia

2.1 Modello

È stato adottato un approccio Object-Oriented per poter modellare le astrazioni chiavi e i componenti del sistema. Poiché il linguaggio C non supporta nativamente i concetti di OOP, molti degli aspetti di tale approccio sono stati seguiti tramite convenzioni, e potrebbero pertanto non essere applicati in tutte le situazioni e alla lettera.

Sono stati applicati i seguenti pattern per modellare le astrazioni chiave (Figure 2):

- `STATE`: È definito un oggetto astratto `Matrix` che espone agli utenti alcuni attributi e metodi di istanza che tutti i formati devono inizializzare e implementare. In questa maniera, è disponibile un contratto comune a tutti i formati per un qualsiasi utente che non è interessato a come è effettivamente implementato il formato sottostante.;

- `BUILDER`: È disponibile un metodo `newMatrix()` che alloca un oggetto Matrix privo di un formato. Questo costruttore dovrebbe essere utilizzato solo dai costruttori offerti da ogni formato, in maniera tale da costruire un oggetto Matrix preliminare che poi viene finalizzato dai costruttori dei singoli formati, andando a compilare il campo `data` con un puntatore a un oggetto che rappresenta il formato in memoria e inizializzando opportunamente tutti gli attributi e i metodi di `Matrix` con i propri valori e implementazioni;

- `PROTOTYPE`: Ogni formato che implementa il metodo `cloneEmpty()` consente a un utente di poter creare un nuovo oggetto `Matrix` vuoto dello stesso formato su cui è stato invocato il metodo, utilizzando se necessari gli stessi parametri che sono stati usati dal costruttore del formato clonato. Ciò è stato utile ad esempio per poter generare un MultiVettore per ogni valore di K desiderato, senza dover conoscere esplicitamente l'implementazione del multivettore. Inoltre, assumendo che si implementi un sistema che smisti le matrici con il giusto formato con le giuste implementazioni di prodotto, sarebbe possibile in teoria generare per ogni matrice su file una corrispondente matrice in ognuno dei formati desiderati, senza conoscere esplicitamente i costruttori di quei formati e i parametri che necessitano.

- `MEDIATOR` (Figure 3): Il componente Mediator espone delle funzioni che permettono di ricopiare gli elementi contenuti in una matrice di un certo formato in un altra nello stesso o altro formato:

  - `convert`: Estrae tutti gli elementi non-zero da una matrice invocando il suo metodo `getNonZero()` per tutte le posizioni fino al suo `numNonZero`, e utilizza il metodo `put()` dell'altra matrice per inserirli al suo interno. È il metodo più efficiente da utilizzare se la matrice `from` è in un formato in memoria e implementa i metodi necessari, assumendo però che le implementazioni di `getNonZero()` e di `put()` siano efficienti;
  - `convertFromMM()`: Sfrutta alcune conoscenze sul formato MatrixMarket per leggere velocemente matrici su file e convertirle in un altro formato. È stata definita per cercare

di velocizzare le conversioni da file poiché l'uso del metodo `getNonZero()` del formato MatrixMarket imponeva tempi di attesa non umananamente sostenibili;

– `convert_dense_too()`: Implementa la conversione usando solo i metodi `get()` e `put()`. È più lento di `convert()`, ma può essere applicato anche a matrici che non implementano il metodo `getNonZero()`, ad esempio i formati che implementano matrici dense.

Grazie all'utilizzo di questi pattern, è stato possibile includere tra i formati anche matrici non-sparse (MultiVettore e ArrayDense) e matrici non in memoria (MatrixMarket). La lettura di una matrice MatrixMarket su file in memoria può essere implementata usando una funzione del mediator, che può essere applicata anche tra matrici in formati entrambi in memoria.

L'utilizzo di questi pattern implica anche una maggiore estensibilità del modello. Infatti, è molto facile aggiungere un formato al momento sconosciuto senza dover cambiare molto del resto del codice.

3 Implementazione prodotti

Per implementare i prodotti tra formati di matrici sparse e multivettori è stato deciso di scrivere un file di interfacce `product.h`, in cui è possibile aggiungere implementazioni di prodotti che devono rispettare tutte la stessa interfaccia. Per poter aggiungere un'implementazione di prodotto, è sufficiente scriverne l'implementazione in un file `.c` (oppure `.cu`) dedicato e aggiungerne l'interfaccia tra le altre.

3.1 `productMatrixMatrixSerial()`

È stato deciso di fornire un'implementazione "ingenua" del prodotto seriale utilizzando i metodi dell'interfaccia `Matrix` per poter ottenere una baseline facilmente senza preoccuparsi del formato delle matrici coinvolte, al costo di una ridotta efficienza. (1)

Per ogni elemento non-zero della matrice sparsa, si scorre tutte le colonne del multivettore e accumula nell'elemento corrispondente della matrice risultato il prodotto tra l'elemento non-zero e l'elemento del multivettore che si trova alla riga con indice pari alla colonna dell'elemento non zero e alla colonna corrente del multivettore.

3.2 `productMatrixMatrixParallelEllpack()`

La funzione *productMatrixMatrixParallelEllpack* implementa il prodotto parallelo tra matrice sparsa in formato Ellpack e un multivettore in CUDA. Il prodotto è stato implementato utilizzando le seguenti ottimizzazioni:

- `Parallelismo senza sincronizzazione`: Il modello di programmazione CUDA fornisce un'API per sincronizzare tutti i thread appartenenti allo stesso blocco Tuttavia, non esiste alcuna API per eseguire la sincronizzazione tra thread di differenti blocchi. Non gestire adeguatamente la sincornizzazione potrebbe determinare rallentamenti sulle operazioni di scrittura e lettura in memoria globale. La scelta più logica che può essere fatta per evitare questo problema è quello di associare ad ogni blocco di thread un insieme di righe.

- `Coalescenza`: La coalescenza è una tecnica di ottimizzazione della gestione della memoria in CUDA che mira a migliorare l'efficienza degli accessi alla memoria globale della GPU. Tale tecnica consiste nell'organizzare gli accessi alla memoria globale in modo che i thread della GPU accedano a blocchi di dati adiacenti. In questo modo, i blocchi di dati possono essere

letti o scritti in modo continuo e efficiente. In particolare, i thread vengono organizzati in blocchi di dimensioni multiple del warp size. I thread all'interno di un blocco accedono ai dati adiacenti e vengono eseguiti in modo simultaneo sulla GPU, riducendo il numero di accessi alla memoria globale. Per fare un esempio se una riga della matrice sparsa è formata da 50 elementi non nulli, il thread con id 0 nel blocco di dimensione 32 andrà a leggere l'elemento in posizione 0 e quello in posizione 32.

- `Memoria shared`: La memoria condivisa è una memoria a bassa latenza e alta velocità presente all'interno delle GPU, utilizzata per migliorare le prestazioni delle applicazioni CUDA. La memoria condivisa è particolarmente utile per le applicazioni che coinvolgono l'elaborazione parallela di grandi quantità di dati dove la latenza della memoria globale può rappresentare un collo di bottiglia per le prestazioni a causa degli accessi frequenti. Gli accessi alla memoria condivisa sono molto veloci perché la memoria condivisa è fisicamente più vicina ai thread della GPU rispetto alla memoria globale. Inoltre, la memoria condivisa è utilizzata anche per implementare una cache a livello di blocco, che consente di memorizzare i dati frequentemente utilizzati in modo molto efficiente.

- `Evitare conflitti sui banchi di memoria condivisa` : I conflitti sui banchi di memoria si verificano quando due o più thread all'interno di un blocco della GPU cercano di accedere allo stesso banco di memoria condiviso allo stesso tempo. Quando ciò accade, la GPU deve serializzare gli accessi al banco, causando una penalizzazione sulle prestazioni dell'applicazione. Per risolvere questo problema è stata utilizzata la funzione *cudaDeviceSetSharedMemConfig* offerta da CUDA che permette di incrementare la dimensione di un banco di memoria condivisa da 32 a 64 bit. Questa operazione si rende necessaria dal momento che i dati delle matrici sono memorizzati in memoria come valori double.

L'implementazione effettiva del prodotto utilizza tutte queste ottimizzazioni. La matrice Ellpack è memorizzata in memoria attraverso due matrici, *A_Values* che mantiene i valori double della matrice sparsa e *A_Cols* che mantiene invece gli indici di colonna dei non zeri. L'idea è quella di scomporre la matrice *A_Values* in blocchi bidimensionali e associare ciascuno di questi blocchi di dati un blocco di $32 * 32$ thread sempre bidimensionale. Successivamente il singolo blocco della matrice *A_Values* viene portato in memoria condivisa. In memoria condivisa vengono copiati anche i valori double del multivettore che devono essere moltiplicati per i non zero. Per individuare questi valori del multivettore si rende necessario l'utilizzo della matrice *A_Cols*.

L'immagine Figure 1 mostra l'idea che c'è dietro l'implementazione. L'immagine è tratta da un prodotto di due matrici dense, ovviamente sono state apportate le modifiche necessarie per adattarlo ad un prodotto matrice sparsa- multivettore. Ogni blocco di thread si prende un blocco di dati dalla matrice *A_Values* e lo moltiplica per il rispettivo blocco di dati del multivettore. Questo procedimento permette di calcolare un blocco della matrice risultato. Il fatto di utilizzare blocchi bidimensionali di dati, combinato con l'utilizzo della memoria condivisa permette di sfruttare in maniera più efficiente il meccanismo di caching e la l'organizzazione gerarchica della memoria. Il grande limite di questa implementazione è che dovrebbe ottenere prestazioni discrete con matrici e prodotti che permettono di fare largo uso del meccanismo di chaching. Per matrici di piccole dimensioni, o multivettori con poche colonne, ci si aspettano prestazioni inferiori dato che non si riesce a sfruttare in maniera massiva il riutilizzo dei dati.

3.3 `productEllpackMultivectorParallelCPU()`

(2) L'approccio adottato per l'implementazione di questo prodotto è stato di assegnare a ogni thread una coppia di elementi, prendendone uno dalla matrice sparsa e uno dal multivettore. Ogni thread moltiplica i due elementi tra loro e accumula il risultato nel corrispondente elemento nella matrice risultato. In questo modo, ogni thread dovrebbe ricevere un carico di lavoro per quanto più possibile uguale a quello degli altri, rispetto all'approccio che prevede di assegnare coppie di righe e colonne per ogni thread, poiché le righe potrebbero avere un numero di elementi non zero molto diversi tra loro.

Per parallelizzare l'algoritmo è stata utilizzata la direttiva `omp parallel for` che divide le iterazioni di un ciclo tra i threads. È stato deciso di optare per una suddivisione statica del carico di lavoro poiché, per quanto detto in precedenza, ci si aspetta che il carico di lavoro sia uniorme tra i threads sia per complessità che per quantità, dunque una suddivisone dinamica non avrebbe portato vantaggi a fronte di costi di sincronizzazione. La direttiva `collapse(3)` permette di trasformare il ciclo a tre livelli in uno solo, permettendo così di distribuire ogni tripla $r_1, r_2, cSub$ a un thread diverso, permettendo quanto descritto nel paragrafo precedente al costo di introdurre un punto di sincronizzazione tra i threads che lavorano su uno stesso elemento risultato mediante la direttiva `omp atomic`, che però non dovrebbe introdurre molto overhead.

3.4 `productCsrMultivectorParallelGPU()`

L'algoritmo utilizzato per implementare il prodotto csr-multivettore in CUDA è illustrato in 3. Ogni riga della matrice CSR è associata al thread che ha la sua coordinata x uguale all'indice di riga, e per ogni colonna registrata in quella riga si accumula il prodotto tra il valore corrispondente alla colonna e il valore del multivettore che ha come riga la colonna di CSR e come colonna la coordinata y del thread.

La macro `getPitched` permette un veloce accesso in aree di memoria allocate con `cudaMallocPitch`.

Tutte le aree di memoria passate come parametri del kernel presentano il type qualifier `__restrict__` che indica a `nvcc` che i puntatori forniti contengono indirizzi diversi tra loro (nessuno tra loro è un alias di un altro tra loro), permettendo al compilatore di apportare riordino dele istruzioni e caricamento di dati in registri per ottimizzare le prestazioni.

Le aree di memoria che devono ospitare i dati del formato CSR sono allocate utilizzando `cudaMalloc()`, che permette di allineare i dati in maniera tale che gli accessi in memoria siano coalizzati, massimizzando così le prestazioni del trasferimento della memoria da Host a Device. Un approccio simile è stato adottato per allocare la memoria che ospita i dati del multivettore e della matrice risultato, mediante l'uso della funzione `cudaMallocPitch()`. Questi approcci hanno reso necessaria l'adozione di diverse accortezze:

- Gli elementi del multivettore sono memorizzati in un array bidimensionale, che registra per ogni riga un puntatore a un array di colonne. Ciò rende necessaria una conversione preliminare da multivettore ad ArrayDense per poter ottenere tutti gli elementi in un array unidimensionale denso, quindi completo di valori non-zero e zero;

- Gli indici usati da ogni trhead per spiazzare le aree di memoria che ospitano i dati delle matrici potrebbero andare out of bounds se la dimensione del blocco eccede quella delle sottomatrice a lui assegnata. Ciò può accadere se la matrice sparsa non È perfettamente partizionabile in tante sottomatrici per quante sono i blocchi, ognuna di dimensione pari a quella di un blocco, ovvero `WARP_SIZE` * `WARP_SIZE`. L'uso di un `if` per sanitizzare gli indici prima di usarli per accedere alla memoria è stato escluso in quanto avrebbe potuto introdurre una divergenza nei thread di

un warp. Pertanto, è stato deciso di allocare della memoria in eccesso in maniera tale che sia perfettamente divisible tra i blocchi degli SMPs, aggiustando le sizes delle memorie da allocare mediante la funzione `adjustToWarpSize()` con dei valori pari a zero. Usando opportunamente puntatori e indici, è possibile isolare nel risultato solo i valori che effettivamente appartengono alle rispettive matrici, escludendo i valori di padding. (Figure 12)

3.5 `productCsrMultivectorParallelCPU()`

La funzione *productCsrMultivectorParallelCPU* implementa il prodotto parallelo tra matrice sparsa in formato Ellpack e un multivettore utilizzando OpenMP. In questa implementazione, è stata utilizzata le seguenti direttive:

- *pragma omp parallel for*: per parallelizzare il ciclo esterno, che scorre tutte le righe e le colonne della matrice in formato CSR;
- *collapse(2)*: per combinare i due cicli nidificati (quello che scorre le righe e quello che scorre le colonne) in un singolo ciclo parallelo, in modo da evitare l'overhead del parallelismo per ogni ciclo.
- *schedule(static)*: per distribuire il lavoro in modo equo tra i thread.
- *pragma omp simd*: per sfruttare le istruzioni SIMD (Single Instruction Multiple Data) per parallelizzare l'operazione di prodotto scalare tra le righe della matrice CSR e le colonne del multivettore.
- *reduction(+:sum)*: per combinare più velocemente i risultati parziali di ogni thread e ottenere il risultato finale.

L'implementazione del codice seriale su cui sono state utilizzate queste direttive è molto semplice: si scorrono una riga per volta la matrice in formato CSR e ogni elemento della riga viene moltiplicato per l'elemento corrispondente del multivettore.

3.6 Esperimenti

Per ogni nome di file di matrice su cui si vogliono effettuare gli esperimenti, si effettua la conversione con il Mediator nel formato desiderato. Per ogni matrice convertita, si accoppia un Multivettore generato per ogni valore di k (larghezza) desiderato, e si riportano i rispettivi puntatori in due arrays paralleli. La funzione `doExperiments()` userà lo stesso indice per accedere alle matrici in questi due arrays paralleli e applicherà ad ogni coppia tutte le implementazioni di prodotto desiderate, per il numero di `TRIALS` desiderati.

Per ogni trial di un esperimento così effettuato viene generato un oggetto `Sample` che contiene tutte le informazioni di profilazione relative a quel trial.

Tutti i Samples così generati sono raccolti in un array, che viene infine convertito in un file csv che può essere elaborato da software statistici.

I valori dei MultiVettori sono generati mediante l'uso di una variabile aleatoria uniforme(0,1) i cui valori sono prodotti da un generatore di Lehmer, implementato nella libreria presente nella cartella `random`.

Per poter raccogliere i tempi di esecuzione, è stata utilizzata la funzione `clock_gettime()` prima e dopo aver lanciato il nucleo di calcolo omp, e gli eventi CUDA per la misurazione delle prestazioni dei calcoli su GPU.

4 Discussione Risultati

Per valutare la bontà delle implementazioni in CUDA è stato preso in considerazione il modello Roofline. Il modello Roofline è basato sulla visualizzazione grafica delle prestazioni di un'applicazione in funzione della banda di memoria, della potenza di calcolo disponibili sull'architettura di calcolo in uso e dall'intensità aritmetica. In pratica, la banda di memoria e la potenza di calcolo sono rappresentate sull'asse y e mentre sull'asse x è rappresentata l'intensità aritmetica. Gli upper bound del grafico rappresentano il limite massimo teorico delle prestazioni dell'applicazione, ovvero il massimo utilizzo possibile della banda di memoria e della potenza di calcolo disponibili. L'obiettivo è quindi quello di far sì che l'applicazione si avvicini il più possibile agli upper bound, attraverso l'ottimizzazione dell'utilizzo della banda di memoria e della potenza di calcolo.

4.1 Intensità aritmetica

L'intensità aritmetica è definita come il rapporto tra il numero di operazioni effettuate da un nucleo di calcolo e il quantitativo di byte acceduti in memoria. Per ELLPACK questo calcolo vale :

$$\frac{2 * k * nz}{16 * nz + 4nz} = \frac{k}{10}$$

Dove :

- **nz** : numeri di non zero della matrice sparsa.
- **k** : numeri di colonne del multivettore.
- **n** : numeri di righe della matrice sparsa.

Al denominatore nz viene moltiplicato per 16 perchè andranno letti in memoria tutti i non zeri della matrice sparsa e tutti i valori corrispondenti del multivettore, entrambi salvati come double. Si somma anche *4*nz* perchè ci saranno accessi in memoria per leggere gli indici di colonna dei valori della matrice sparsa memorizzati come interi.
Per CSR l'intensità aritmetica vale invece:

$$\frac{2 * k * nz}{16 * nz + 4n + 4 * nz} \approx \frac{k}{11}$$

Al denominatore è presente *16*nz* per lo stesso motivo descritto per Ellpack. La differenza in questo caso è verrà mantenuto in memoria anche un array di dimensione n che indica gli indici di inizio di ogni riga della matrice sparsa. In più viene memorizzato anche un vettore di dimensione nz contenente tutti gli indici di colonna dei non zeri.
Si osservi come l'intensità aritmetica dipende dal valore di K. Maggiore è il suo valore maggiore sarà l'intensità aritmetica e maggiori saranno le prestazioni che ci si aspetta di ottenere dal prodotto. Una conseguenza di questa analisi è che il prodotto matrice sparsa - multivettore è sia un problema memory bound che un problema compute bound. In particolare all'aumentare del valore di k, il problema tende a spostarsi da un problema memory bound a uno compute bound.

4.2 Calcolo degli upper bound

Per calcolare gli upper bound sono necessarie due informazioni : il picco massimo di gigaflops che può raggiungere la scheda grafica e il picco massimo teorico della larghezza di banda. Entrambe queste informazioni sono reperibili dalle specifiche della GPU. Per una GPU *Quadro rtx 5000* il

limite teorico di GFLOPS vale 330 gigaflops per le operazioni con dati in doppia precisione, mentre il picco massimo teorico della larghezza di banda vale 488GB/s. Nell'immagine **Figure 4** è mostrato il modello roofline per il prodotto in formato ELLPACK. Sull'asse delle x vengono riportati i valori di k per facilitare la lettura. Si può notare come i valori delle prestazioni seguano l'andamento degli upper bound. Tuttavia per nessun valore di k si riesce a raggiungere una convergenza perfetta ai valori di picco. Sono presenti molte matrici che fanno registrare valori di prestazioni molto alti. Il picco massimo di prestazioni ottenuto è per *k=64* e sono stati toccati i 150 gigaflops. Mediamente le prestazioni di picco ottenute si avvicinano alla metà delle prestazioni di picco teoriche. Si noti anche come alcune matrici non registrano prestazioni ottimali. Il perchè di questa scarsità di prestazioni verrà discusso in seguito.
Nell'immagine **Figure 5** è invece mostrato il modello roofline per il prodotto in CUDA realizzato con il formato CSR. Valgono le stesso osservazioni fatte in precedenza. L'unica differenza in questo caso è che si registrano prestazioni leggermente peggiori rispetto al prodotto in Ellpack.

4.3 Analisi prodotti CUDA

Come punto di partenza dell'analisi dei risultati si è considerato una matrice delle correlazioni che permette di capire in che modo i gigaflops di un prodotto possano essere influenzati da determinati parametri. In particolare sono state considerate due metriche:

- **Padding indicator** : Questa metrica è definita dal rapporto tra il numero medio di non zeri per riga è il numero massimo di non zeri per riga. Si tratta quindi di un valore compreso tra 0 e 1. Se questo parametro tende a 1 vuol dire che è presente poco padding all'interno della matrice in formato Ellpack.
- **Varianze dei non zero per colonna** : Questa metrica è stata considerata per vedere in che modo la varianza dei non zeri per riga impattasse sulle prestazioni. L'idea è che se la varianza è un valore molto alto , e se si associa un thread ad ogni riga della matrice sparsa, ci saranno alcuni thread che eseguiranno molto lavoro, altri che invece eseguiranno pochi calcoli. In pratica questa metrica fornisce un indicazione sulla distribuzione del lavoro da svolgere.

Nell'immagine **Figure 8** sono mostrati tutti i valori di correlazione per ogni coppia di attributo. I valori interessanti sono le coppie *gigaflops-padding indicator* e *gigaflops-VarNotZeroRows*. Si può osservare come il padding indicator sia correlato positivamente ai gigaflops. E' un risultato atteso dal momento che il padding della matrice ellpack influenza negativamente le prestazioni: più è alto minore saranno i gigaflops. Per la varianza si osserva invece un risultato inatteso: le prestazioni sono quasi indipendenti dal valore della varianza dei non zeri per riga. Altre osservazioni interessanti che si possono trarre dall'immagine è che i gigaflops sono correlati positivamente anche alla dimensione della matrice sparsa(numElement_mat1) e soprattutto dal numero di colonne del multivettore (numElement_mat2). Questi risultati portano ad una conferma delle osservazioni fatte in precedenza: per ottenere prestazioni migliori è necessario considerare matrici di grandi dimensioni con valori di k almeno maggiori di 16.

In Figure 13 è possibile vedere il valore medio di GFLOPS in funzione del numero dei non-zero nelle matrici (sono state scartate quelle con meno di 20000 elementi), con un multivettore di 64 colonne e di 8. Si può notare come i GFLOPS aumentino sensibilmente all'aumentare del valore di K, il che significa che non si sta ancora utilizzando la banda a pieno regime, e dunque è possibile elaborare quantità maggiori di dati se si utilizzassero valori di k piu alti. Si può notare inoltre come l'uso del formato ELLPACK comporti quasi sempre prestazioni migliori rispetto alla controparte

CSR.

In Figure 14 sono illustrati i valori di GFLOPS in funzione della media della varianza della presenza di elementi non-zero per ogni riga. È possibile notare come, a parità di varianza, i valori di GFLOPS per ELLPACK siano quasi sempre per la maggior parte tutti sopra a quelli di CSR. In Figure 15 è possibile vedere come le prestazioni del formato ELLPACK siano notevolmente più sensibili al cambio di valore della varianza rispetto a CSR, guadagnando molto in prossimità di alcuni valori di varianza, senza scendere mai sotto i valori di CSR. Si può notare come inoltre le linee di ELLPACK e CSR seguano un andamento simile, ovvero entrambe salgono e scendono insieme, differendo per l'entità. Ciò sembra suggerire che la varianza di elementi non-zero per riga influisca sulle prestazioni di entrambi i formati allo stesso modo, ma con peso in funzione del formato stesso.

In Figure 16 si può notare come una riduzione della percentuale di zeri per ogni riga aumenti progressivamente le prestazioni per il formato ELLPACK, mentre il formato CSR non sembra risentirne molto, come è ragionevole aspettarsi poiché è il formato ELLPACK a poter contenere zeri a causa del padding, mentre CSR non ne prevede.

Per concludere l'immagine Figure 10 mostra un confronto di prestazioni medie tra il prodotto implementato con il formato CSR e il prodotto implementato con Ellpack. Si può osservare come mediamente Ellpack abbia prestazioni migliori anche se ci sono determinate matrici in cui CSR registra gigaflops maggiori.

4.4 Analisi prodotti OpenMP

Per valutare la bontà dei prodotti implementati in OpenMP si calcola lo speedup rispetto al prodotto seriale. Lo speedup in questo contesto è definito come il rapporto tra gigaflops del prodotto parallelo OpenMp e il prodotto seriale.
Si può notare nelle immagini Figure 11 e Figure 9 come ci sia un effettivo miglioramento in termini di gigaflops eseguendo il prodotto con OpenMP. L'incremento dei gigaflops per Ellpack è di circa 10-12 volte. Con CSR si registra invece un incremento anche di 1400 volte.

Infine viene mostrato il boxplot delle prestazioni raggiunte dall'implementazione dei due prodotti (Figure 11) (Figure 9). Per il prodotto in formato CSR si può notare come venga raggiunto il picco massimo di prestazioni di 9 gigaflops. I gigaflops ottenuti sono tuttavia molto aleatori, si osservano anche matrici in cui le prestazioni non sono eccellenti. Questo è dovuto probabilmente ad una distribuzione non omogenea del lavoro dei thread. Per come è implementato il prodotto infatti ogni thread prende una riga della matrice sparsa e una colonna del multivettore ciclicamente. Potrebbero esserci matrici particolarmente squilibrate in cui alcuni thread eseguono molto lavoro e altri invece molto meno. Questo andamento altalenante si registra anche per per il prodotto OpenMP implementato per il formato Ellpack. Si noti però come entrambi i prodotti hanno prestazioni migliori in corrispondenza delle stesse matrici.

I dati che vengono mostrati nei boxplot sono stati ottenuti lanciando un numero di thread di default impostati da OpenMP. Tuttavia sono stati condotti anche altri esperimenti per verificare l'andamento delle prestazioni al variare del numero di thread in esecuzione. Le immagini Figure 17 e Figure 18 mostrano infatti come variano i gigaflops all'aumentare dei thread allocati per i due formati di matrice ELLPACK e CSR. L'immagine Figure 17 fa riferimento in particolare alle prestazioni delle matrici implementate con il formato CSR. Si può osservare come inizialmente per quasi tutte

le matrici ci sia un incremento lineare delle prestazioni all'aumentare dei thread allocati. E' ovviamente un risultato atteso, maggiore è il numero di thread in esecuzione più si riesce a sfruttare il parallelismo. A questo incremento di prestazione c'è tuttavia un limite: non ha senso far aumentare all'infinito i numeri di thread in esecuzione in quanto essi dovranno essere eseguiti su core fisici. Se la concorrenza per l'accesso ad un core fisico è eccessiva, l'overhead di gestione e schedulazione dei thread diventa non trascurabile. Questo influisce negativamente sulle prestazioni. Tale osservazione trova riscontro nel fatto che allocando un numero maggiore di 40 thread i gigaflops non aumentano più linearmente. Ma anzi tendono leggermente a diminuire e stabilizzarsi.

Nell'immagine Figure 18 sono mostrate invece le prestazioni delle matrici in formato Ellpack. Dal grafico si evince un risultato inaspettato: si ha un picco prestazionale nel momento in cui si usa un solo thread, e poi si cresce monto lentamente all'aumentare dei threads fino al numero di threads disponibili, per poi ripeggiorare se si supera quel valore. Ciò sembra suggerire che l'implementazione omp del prodotto con ellpack non sfrutti appieno i benefici dati dal parallelismo dei thread.

5 Conclusioni

Per quanto riguarda il prodotto in CUDA il formato Ellpack ha registrato delle prestazioni leggermente migliori. Questo è stato dovuto all'applicazione di un maggior numero di ottimizzazioni. Come è stato possibile vedere dal modello di Roofline sono presenti ancora margini di miglioramento per i prodotti in quanto non è stato raggiunto il picco massimo delle prestazioni. Per il prodotto in OpenMP invece è il formato CSR che ha registrato delle prestazioni migliori. Dall'analisi statistica e teorica che è stata condotta è emerso che le prestazioni in termini di gigaflops dipendono sostanzialmente da 2 fattori:

1. Dimensione della matrice sparsa
2. Dimensione del multivettore

Per la matrice in formato ellpack le prestazioni dipendono anche dal padding, invece la varianza dei non zero nelle righe incide molto poco. I prodotti che hanno fatto registrare prestazioni migliori sono quelli in cui le matrici sparse avevano pochissimo padding, erano di dimensioni notevoli e il valore k del multivettore assumeva il valore 64.

6 Suddivisione lavoro

Tutte le attività di progettazione e organizzazione del codice, le analisi delle possibili ottimizzazioni , la generazione dei grafici e l'analisi dei dati sono state effettuate in coppia. Di seguito sono riportate le attività specifiche di implementazione.

Luca Fiscariello	**Daniele La Prova**
Implementazione formato Ellpack	Implementazione formato CSR
Implementazione prodotto seriale "generico"	Implementazione formato COO
Implementazione mediator	Implementazione formato MatrixMM
Implementazione prodotto cuda Ellpack	Implementazione prodotto cuda CSR
Implementazione prodotto OpenMP CSR	Implementazione prodotto OpenMP Ellpack

7 Allegati

Algorithm 1 Prodotto seriale

```
//Scorro tutti gli elementi non zeri della prima matrice
for(int i =0; i< matrix1->numNonZero; i++){

    nze = matrix1->getNonZero(matrix1,i);

    //scorro tutti gli elementi della riga "nze->row" della seconda matrice
    for(int j =0; j< matrix2->cols; j++){

        /**
         * Moltiplico un elemento non nullo della prima matrice per tutti gli elementi della riga della seconda matrice.
         * Sommo questo risultato parziale nella matrice risultato. La posizione in cui sommare questo risultato parziale
         * è definita dalla riga dell'elemento della prima matrice e dalla colonna del valore della seconda matrice.
         */
        result[nze->row][j] += nze->value * matrix2->get(matrix2,nze->col,j);
    }
}
```

Algorithm 2 Prodotto ELLPACK * MultiVettore (OMP)

```
/**
 * All variables used in the parallel region are shared unless specified otherwise.
 *
 * schedule(static):
 *  Each thread will be assigned a fixed number of iterations.
 *  Since each iteration should do the same amount of work, this should be a faster option
 *  compared to a dynamic schedule, since the latter implies synchronization overhead.
 *
 * collapse(3):
 *  All for cycles are collapsed into a single loop, so its iterations
 *  are distributed among the threads. This is possibile because each loop has a
 *  fixed number of iterations indipendent from other loops.
 */
#pragma omp parallel for default(shared) schedule(static) collapse(3)
for (int r1 = 0; r1 < ellpackData ->rowsSubMat; r1++){
    for (int c2 = 0; c2 < matrix2 -> cols; c2 ++){
        for (int cSub = 0; cSub < ellpackData ->colsSubMat; cSub++){

            /**
             * Ogni elemento della matrice risultato è aggiornato atomicamente, implicando
             * una sincronizzazione tra i thread dedicati a uno stesso elemento, ma non tra quelli
             * dedicati a elementi diversi.
             * https://www.openmp.org/spec-html/5.0/openmpsu95.html#x126-4840002.17.7
            */
            #pragma omp atomic
            resultData[r1 * matrix2 ->cols + c2] += ellpackData ->matValues[r1][cSub] *
            multivectorData[ellpackData ->matCols[r1][cSub]][c2];
        }
    }
}
```

Algorithm 3 Prodotto CSR * Multivettore (CUDA)

```
/**
 * Associamo ogni thread di un blocco a una riga della matrice.
 * Dunque dobbiamo dividere le righe per ogni blocco a gruppi di BD righe.
 * Ogni thread prende le colonne della riga che gli è stata assegnata e accumula il
 * prodotto sommando il risultato parziale  al prodotto tra il valore corrispondente alla
 * colonna e il valore del multivettore corrispondente.
 */
row = threadIdx.x + blockIdx.x * blockDim.x;
mvCol = threadIdx.y + blockIdx.y * blockDim.y;
startCol = firstColOfRowIndexes[row];
endCol = firstColOfRowIndexes[row + 1];
for (int c = startCol; c < endCol; c++){
    partial += values[c] * getPitched(mv, columns[c], mvCol, mv_pitch, double);
}
getPitched(result, row, mvCol, result_pitch, double) = partial;
```

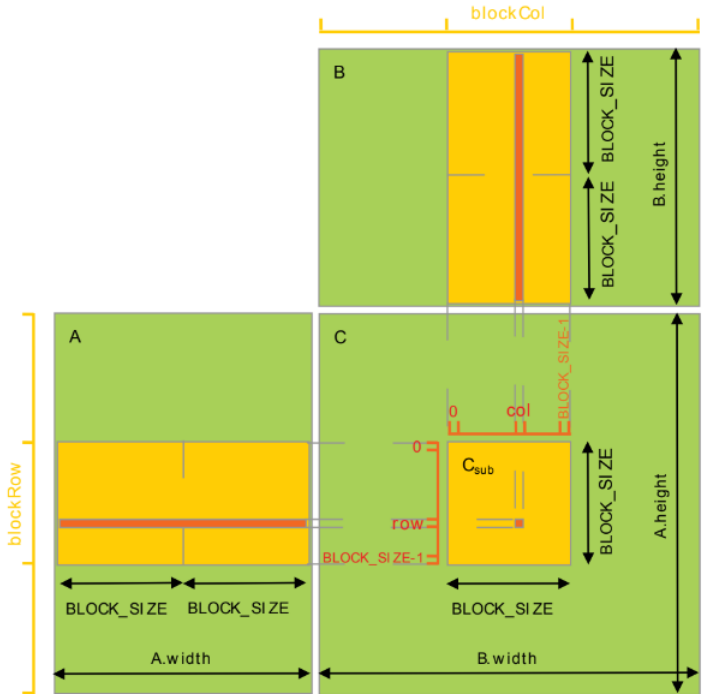

Figure 1: Esempio grafico dell'implementazione del prodotto matrice Ellpack- multivettore

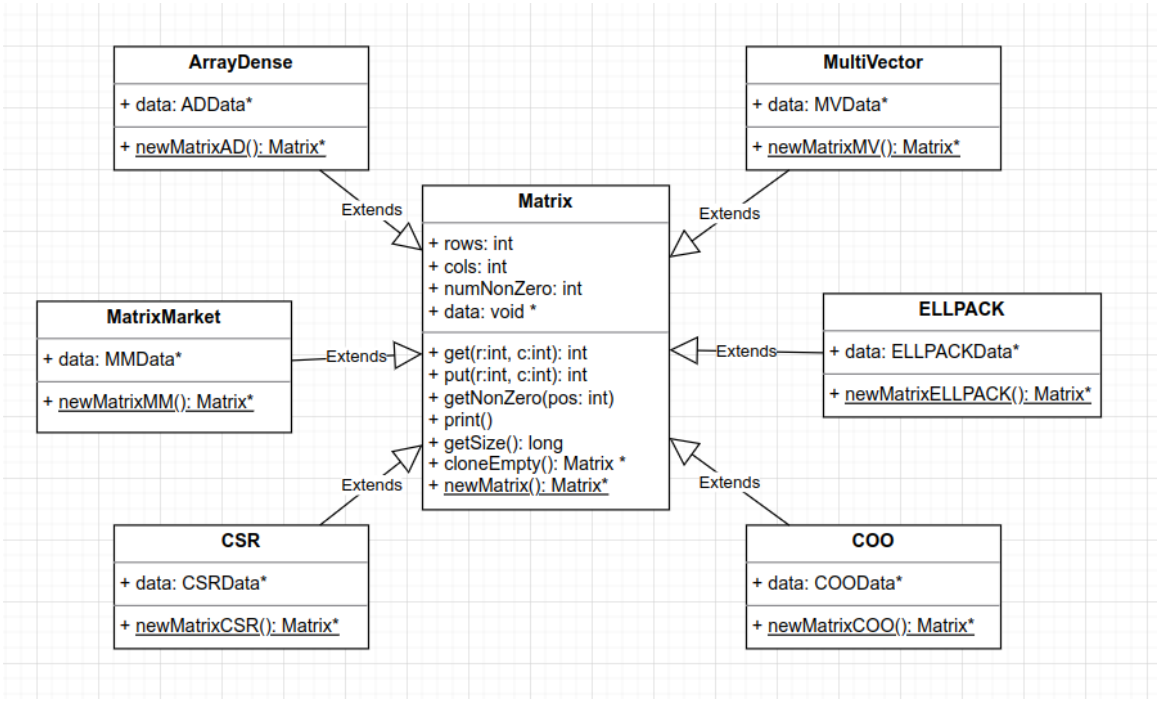

Figure 2: Modellazione di una matrice con i suoi formati

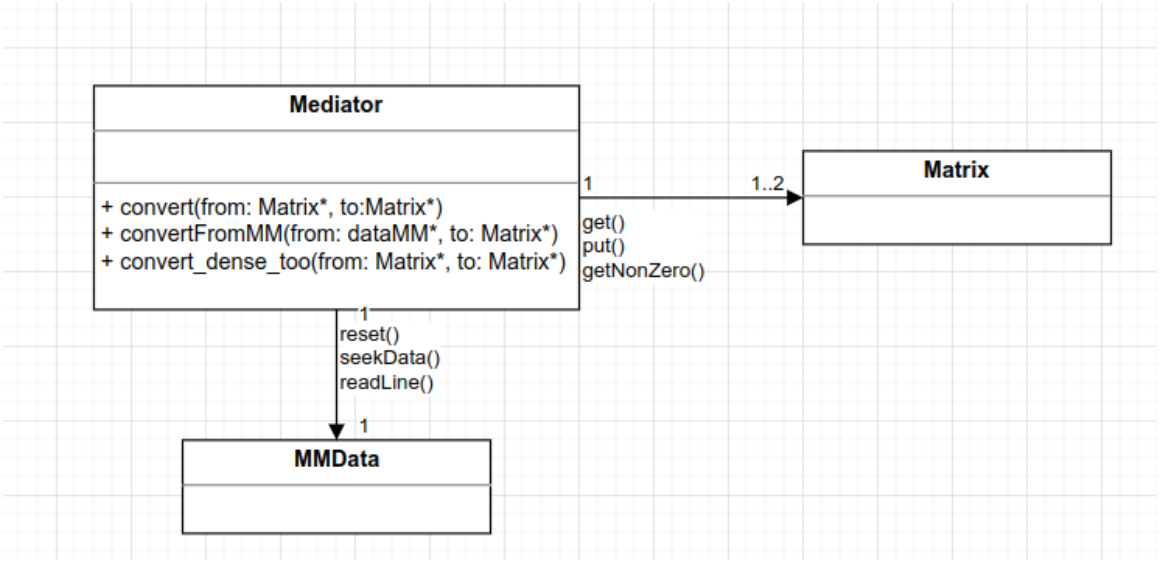

Figure 3: Modellazione del mediator

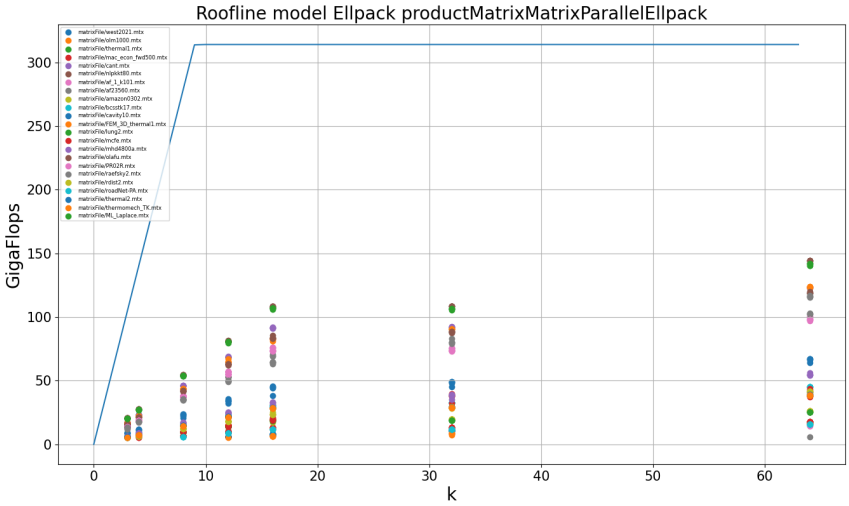

Figure 4:

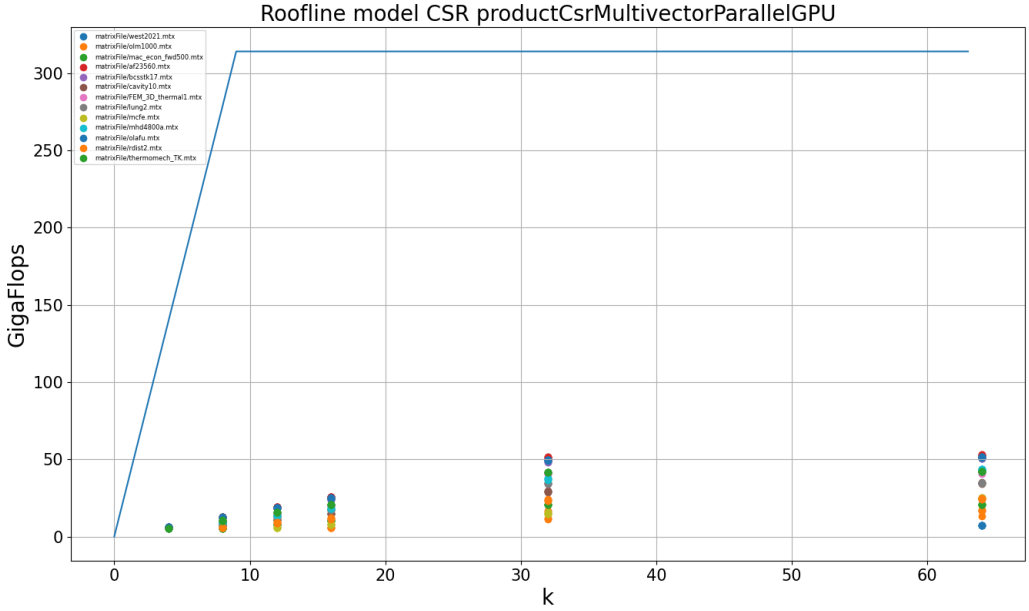

Figure 5:

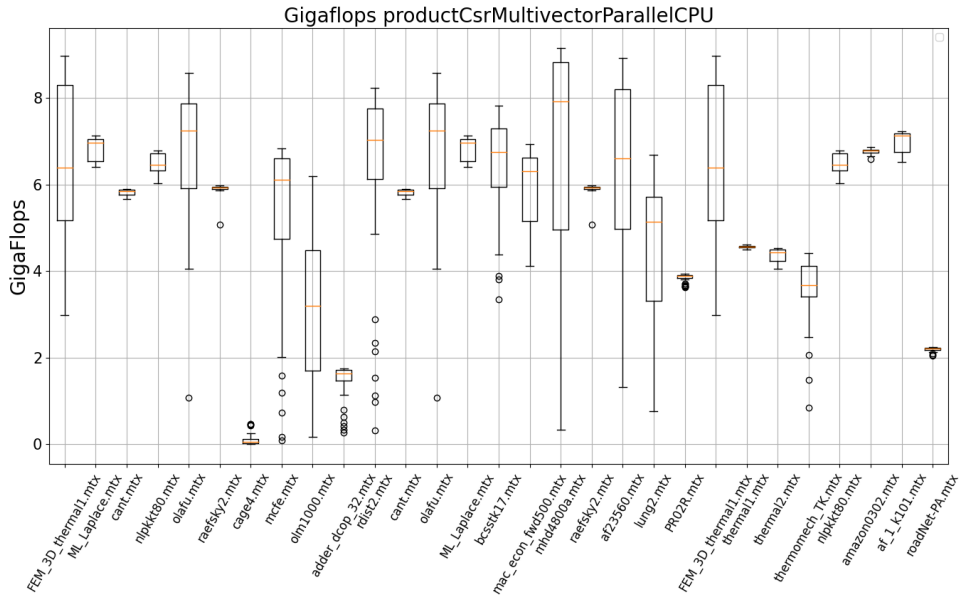

Figure 6:

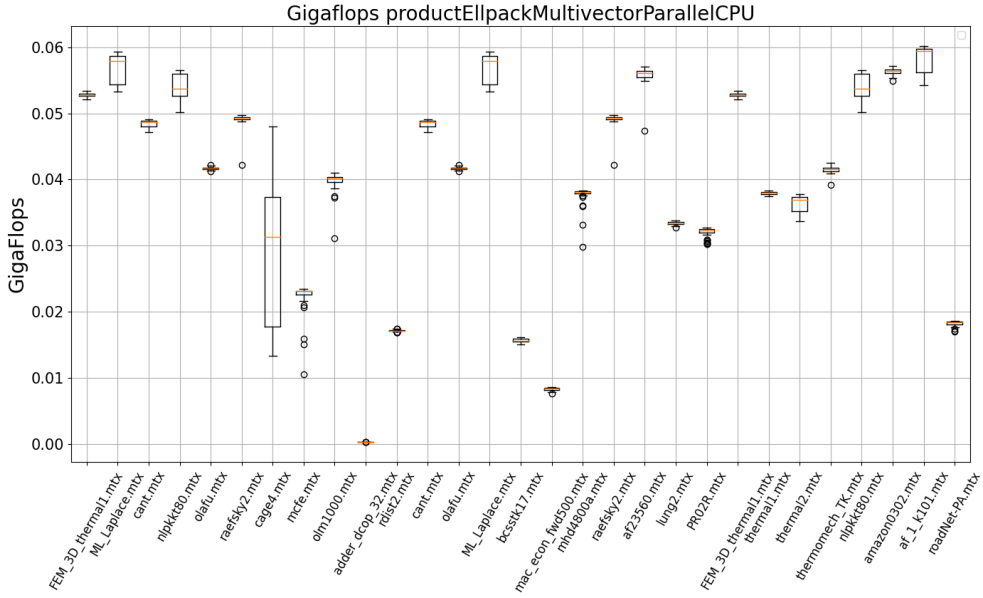

Figure 7:

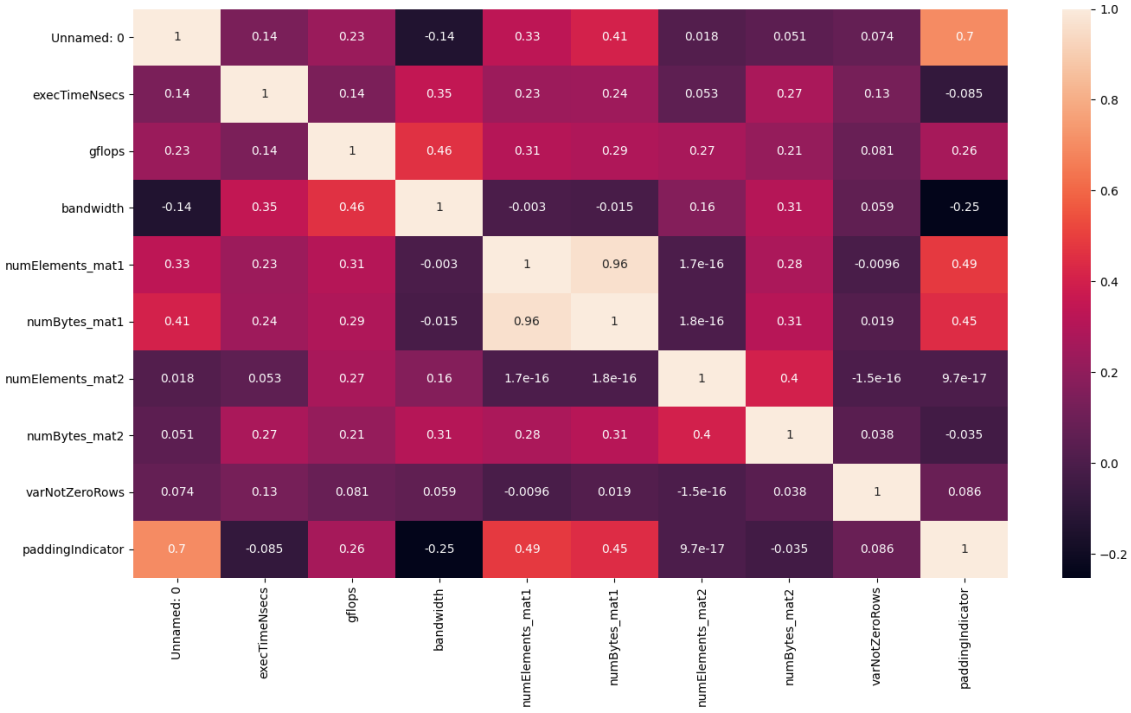
Unnamed: 0
execTimeNsecs
gflops
bandwidth
numElements_mat1
numBytes_mat1
numElements_mat2
numBytes_mat2
varNotZeroRows
paddingIndicator
1 0.14 0.23 -0.14 0.33 0.41 0.018 0.051 0.074 0.7
0.14 1 0.14 0.35 0.23 0.24 0.053 0.27 0.13 -0.085
0.23 0.14 1 0.46 0.31 0.29 0.27 0.21 0.081 0.26
-0.14 0.35 0.46 1 -0.003 -0.015 0.16 0.31 0.059 -0.25
0.33 0.23 0.31 -0.003 1 0.96 1.7e-16 0.28 -0.0096 0.49
0.41 0.24 0.29 -0.015 0.96 1 1.8e-16 0.31 0.019 0.45
0.018 0.053 0.27 0.16 1.7e-16 1.8e-16 1 0.4 -1.5e-16 9.7e-17
0.051 0.27 0.21 0.31 0.28 0.31 0.4 1 0.038 -0.035
0.074 0.13 0.081 0.059 -0.0096 0.019 -1.5e-16 0.038 1 0.086
0.7 -0.085 0.26 -0.25 0.49 0.45 9.7e-17 -0.035 0.086 1
1.0
0.8
0.6
0.4
0.2
0.0
−0.2

Figure 8:

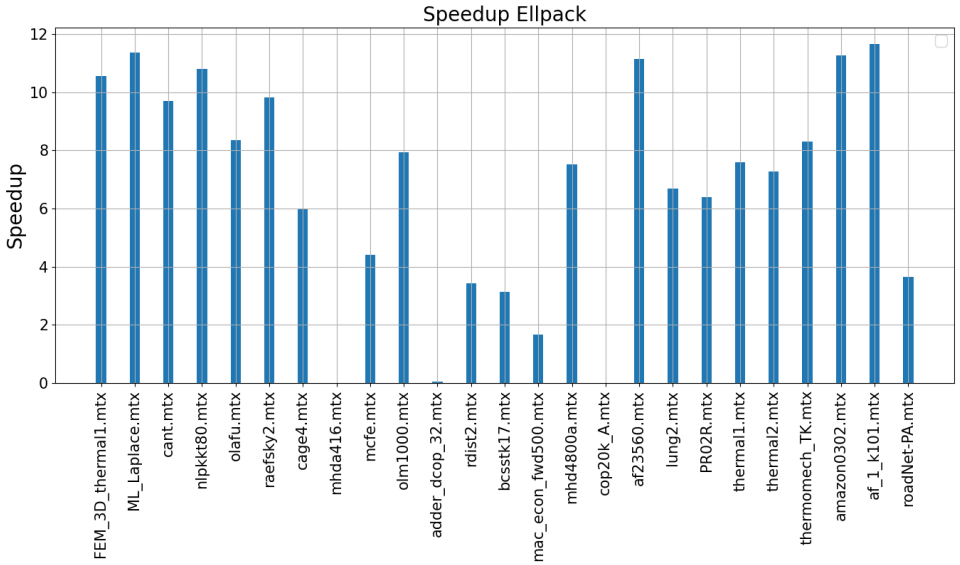

Figure 9:

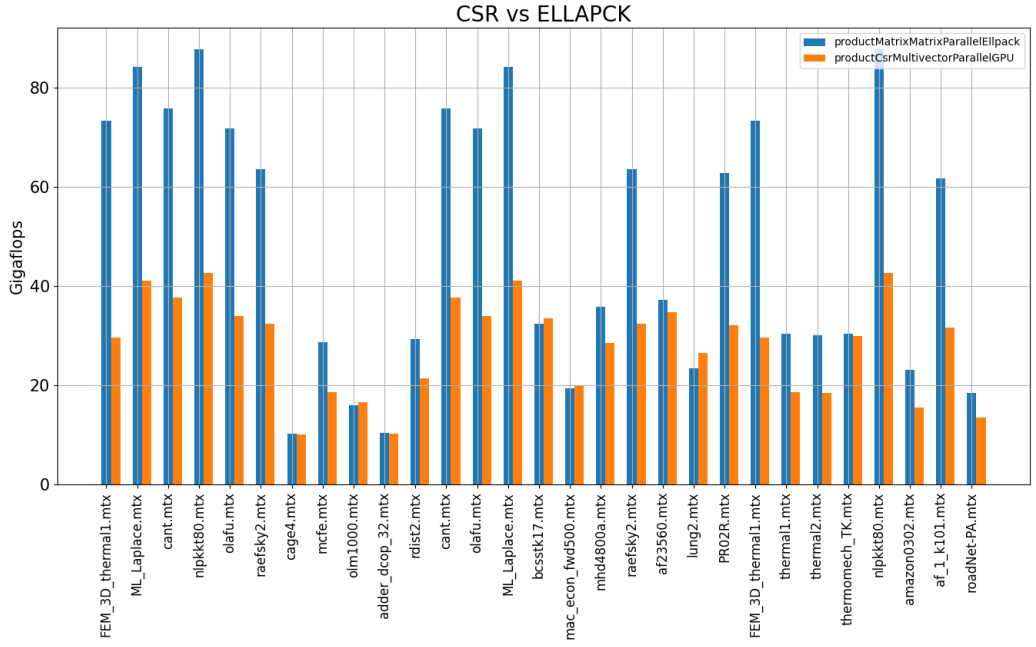

Figure 10:

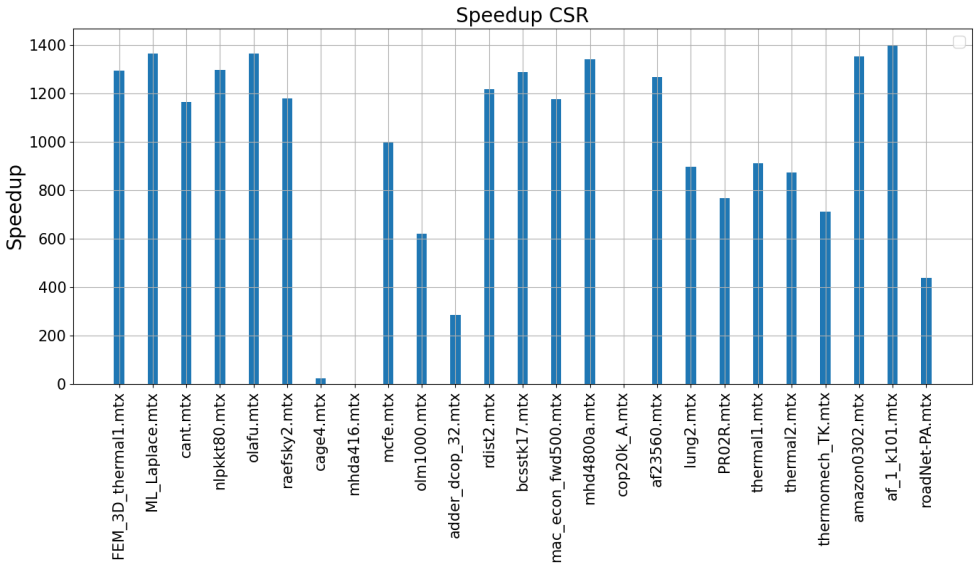
Speedup CSR
Speedup
0
200
400
600
800
1000
1200
1400
FEM_3D_thermal1.mtx
ML_Laplace.mtx
cant.mtx
nlpkkt80.mtx
olafu.mtx
raefsky2.mtx
cage4.mtx
mhda416.mtx
mcfe.mtx
olm1000.mtx
adder_dcop_32.mtx
rdist2.mtx
bcsstk17.mtx
mac_econ_fwd500.mtx
mhd4800a.mtx
cop20k_A.mtx
af23560.mtx
lung2.mtx
PR02R.mtx
thermal1.mtx
thermal2.mtx
thermomech_TK.mtx
amazon0302.mtx
af_1_k101.mtx
roadNet-PA.mtx

Figure 11:

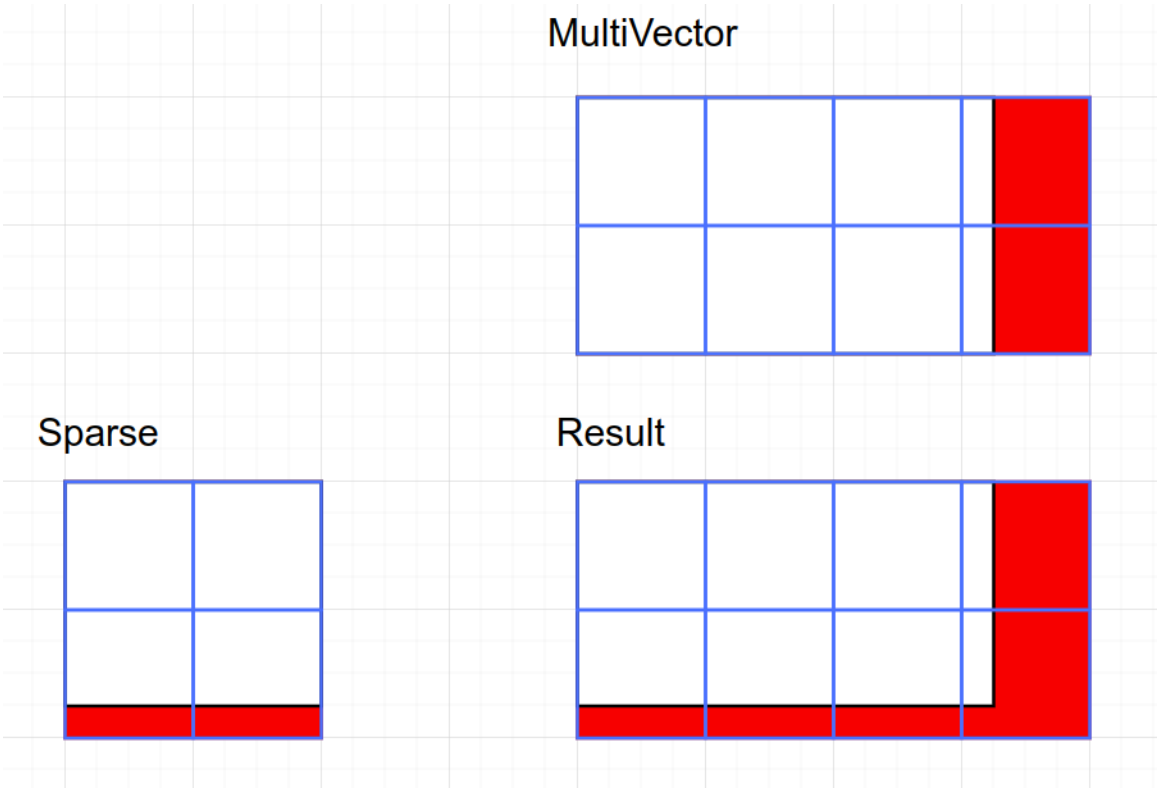

Figure 12: In Bianco: memoria che appartiene effettivamente alle rispettive matrici; In blu: memoria accessibile da un determinato blocco; In rosso: memoria di padding aggiunta da `adjustToWarpSize()`.

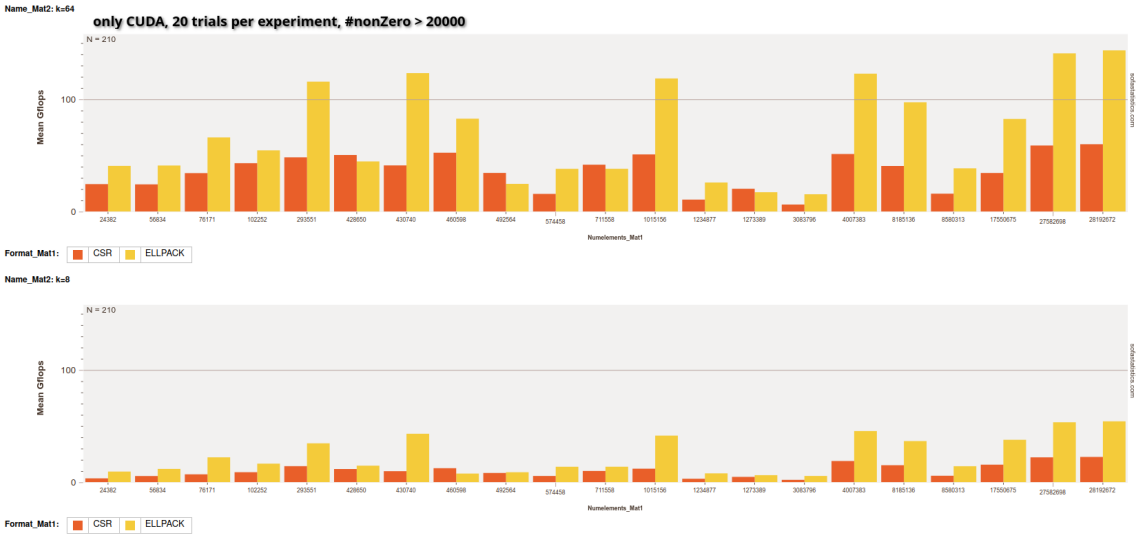

Figure 13:

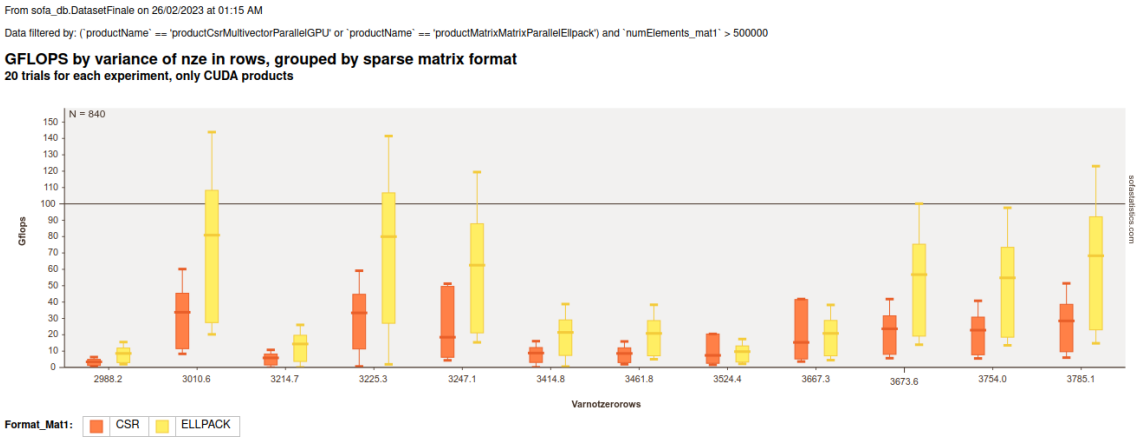

Figure 14:

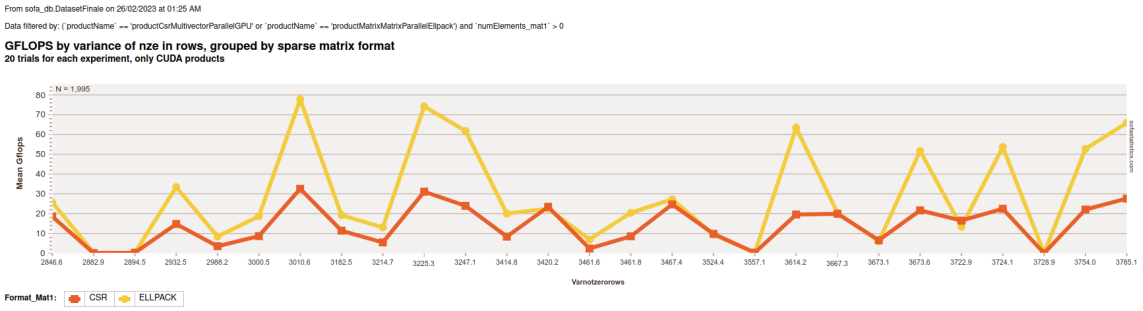

Figure 15:

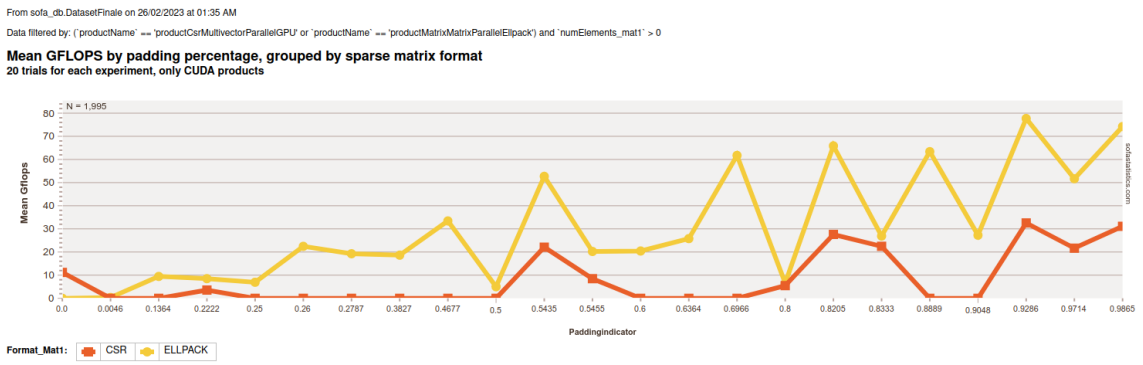

Figure 16:

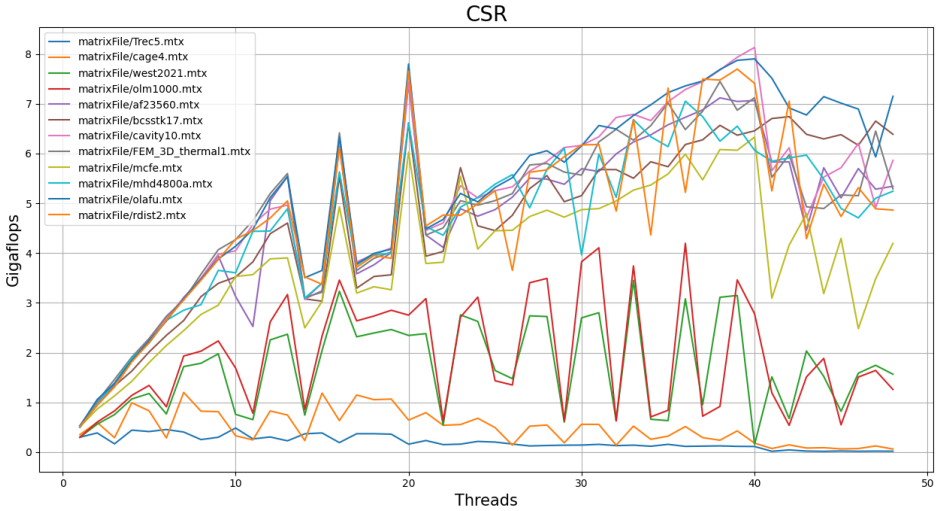

Figure 17:

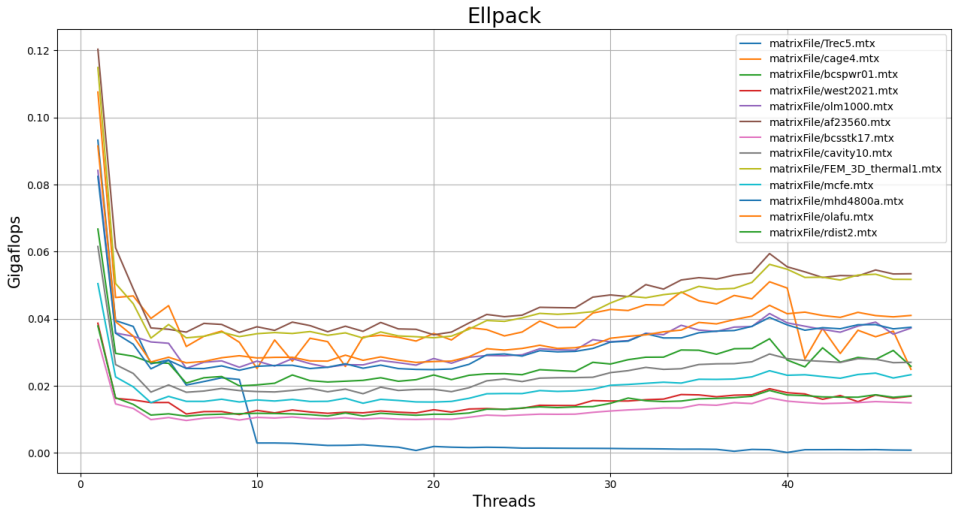
Ellpack
Gigaflops
Threads
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0
10
20
30
40
matrixFile/Trec5.mtx
matrixFile/cage4.mtx
matrixFile/bcspwr01.mtx
matrixFile/west2021.mtx
matrixFile/olm1000.mtx
matrixFile/af23560.mtx
matrixFile/bcsstk17.mtx
matrixFile/cavity10.mtx
matrixFile/FEM_3D_thermal1.mtx
matrixFile/mcfe.mtx
matrixFile/mhd4800a.mtx
matrixFile/olafu.mtx
matrixFile/rdist2.mtx

Figure 18: