SCPA
Relazione di progetto

Christian Santapaola
0294464 *Università di Roma Tor Vergata*
Roma, Italia

***Abstract*—Questo progetto implementa i formati Compressed Sparse Row (CSR per brevità) e ELL per rappresentare matrici sparse: per questi formati fornisce più implementazioni del prodotto matrice-vettore parallele basate su CUDA e il framework OMP allo scopo di poi analizzare i risultati in FLOPS delle implementazioni fornite.**

I. Introduzione

Una matrice sparsa è una matrice in cui la maggior parte degli elementi sono zero; salvare e processare questi zeri rappresenta uno spreco di capacità e banda della memoria, tempo ed energia. Per risolvere questo problema sono stati proposti differenti formati di archiviazione con i loro corrispettivi metodi di computazione, la caratteristica principale di questi formati è l'utilizzo di tecniche di compressione per evitare la memorizzazione e la computazione sugli elementi zero al costo di aggiungere un certo grado di irregolarità nella rappresentazione dei dati della matrice. Sfortunatamente queste irregolarità possono causare problemi di sotto utilizzo della banda di memoria, problemi di divergenza tra flussi di controllo (*control flow divergence*) e uno sbilanciamento del carico nel calcolo parallelo. Raggiungere un giusto compromesso è importante: alcuni formati hanno un altro grado di compressione insieme ad un alto grado di irregolarità, mentre altri raggiungono un grado di compressione più basso in cambio di una rappresentazione dei dati più regolare. Va indicato che il livello di performance raggiungibile dal calcolo matrice-vettore di una matrice sparsa in un particolare formato dipende in maniera non trascurabile dalla distribuzione dei valori non nulli nella matrice. Per una matrice sparsa non è conveniente risolvere il calcolo di sistemi lineari del tipo $A \cdot \hat{x} + \hat{y} = 0$ tramite l'inversione della matrice $\hat{x} = A^{-1} \cdot (-\hat{y})$ perché la matrice A^{-1} non è una matrice sparsa, anzi molto spesso è una matrice densa di elementi non nulli, la cui memorizzazione e l'elaborazione risultano difficili. Per risolvere tali sistemi si adottano strategie iterative, nelle quali si predice un valore di $\hat{x}$ e si calcola il valore di $A \cdot \hat{x} + \hat{y}$ andando ad aggiornare ad ogni iterazione il valore di $\hat{x}$ tramite qualche criterio se il valore calcolato è diverso da 0; pertanto per questi algoritmi il prodotto matrice-vettore è solitamente il collo di bottiglia delle loro performance: risulta quindi che l'ottimizzazione e parallelizzazione di questa operazione è un problema dai molti risvolti pratici.

II. formato COO

```
typedef struct COOMatrix {
    float *data;
    u_int64_t *col_index;
    u_int64_t *row_index;
    u_int64_t row_size;
    u_int64_t col_size;
    u_int64_t num_non_zero_elements;
} COOMatrix;
```

Il formato COO rappresenta una matrice tramite:

- un array lineare contenente i valori non nulli;
- due array che per l'i-esimo valore dell'array dei valori non nulli contengono le sue coordinate riga e colonna nella matrice.

Il formato COO è un formato molto semplice da implementare e ha delle caratteristiche che lo rendono adatto a rappresentare le matrici sparse. Una delle sue caratteristiche più particolari è che, per come è stato definito il formato, l'ordine degli elementi non è importante. La struttura di questo formato lo rende inadatto alle computazioni parallele, in particolare per il calcolo di prodotti matrice-vettore, per cui è utilizzato per rappresentare matrici sparse in memoria. Per effettuare computazioni specializzate si converte il formato COO in uno più specifico come, ad esempio, i formati CSR e ELL.

III. Compressed Sparse Row

```
typedef struct CSRMatrix {
    float *data;
    u_int64_t *col_index;
    u_int64_t *row_pointer;
    u_int64_t num_non_zero_elements;
    u_int64_t row_size;
    u_int64_t col_size;
} CSRMatrix;
```

Il formato CSR salva gli elementi non nulli della matrice in un array lineare chiamato `data`. Per mantenerne la struttura si ha bisogno di due insiemi di segnaposto:

- `col_index`: un array lineare che salva per l'elemento i-esimo dell'array `data` il suo indice di colonna;
- `row_pointer`: un array lineare che tiene traccia della posizione iniziale di ogni riga nel suo formato compresso nell'array `data`; in generale, in `row_pointer[i]` è contenuto l'indice dell'array `data` per cui `data[row_pointer[i]]` è il primo elemento della i-esima riga.

Gli altri elementi inseriti nell'implementazione fornita non sono fondamentali per il funzionamento del formato, ma risultano molto convenienti poiché salvano i metadati della

matrice quali le sue dimensioni e il numero di elementi non nulli.

A. Prodotto matrice-vettore CUDA

```
1  __global__ void
2  SpMV_CSR_kernel(u_int64_t num_rows, const float *
       data, const u_int64_t *col_index, const
       u_int64_t *row_ptr, const float *x, float *y) {
3      const u_int64_t row = blockIdx.x * blockDim.x +
       threadIdx.x;
4      if (row < num_rows) {
5          float dot = 0.0f;
6          const u_int64_t row_start = row_ptr[row];
7          const u_int64_t row_end = row_ptr[row + 1];
8          for (u_int64_t elem = row_start; elem <
       row_end; elem++) {
9              dot += data[elem] * x[col_index[elem]];
10         }
11         y[row] += dot;
12     }
13 }
```

Per il calcolo parallelo del prodotto matrice-vettore possiamo fare le seguenti osservazioni: per ogni riga il calcolo del prodotto matrice-vettore è indipendente dal prodotto delle altre righe, quindi ad ogni thread CUDA assegniamo il compito di calcolare il prodotto matrice-vettore parziale per una singola riga della matrice. Dal codice possiamo notare come la riga 3 si occupa di identificare l'indice di riga da assegnare al thread. Questo, insieme al costrutto `if` della riga 4, permette di gestire un numero arbitrario di righe, anche nel caso in cui il numero di righe non sia un multiplo della dimensione del *thread block*. Nonostante la sua semplicità, questo kernel ha due grandi difetti:

- Il kernel non esegue accessi *coalesced* alla memoria, ovvero thread adiacenti tra loro accedono ad aree di memoria non adiacenti: questo causa uno spreco della banda di memoria.
- Il kernel può subire pesanti divergenze dei flussi di controllo (*control flow divergence*), ovvero quando i thread in esecuzione sull'ambiente CUDA eseguono quantità di lavoro differenti, andando a rallentare l'esecuzione generale: i thread, infatti, per finire e liberare le risorse di calcolo aspettano il thread più lento; il numero di iterazioni di ogni thread dipende dal numero di elementi non nulli della riga assegnata e quest'ultimo valore dipende dalla distribuzione della matrice.

Questi due difetti implicano che l'efficienza di esecuzione e di uso della banda di memoria dipendono dalla distribuzione della matrice in input.

B. Prodotto matrice vettore OMP

```
1  int CSRMatrix_SpMV_OMP(const CSRMatrix* matrix,
       const Vector* x, Vector* y, Benchmark *benchmark
       ) {
2      if (!matrix || !x || !y) {
3          return SPMV_FAIL;
4      }
5      if (x->size != matrix->col_size && y->size !=
       matrix->row_size) {
6          return SPMV_FAIL;
7      }
8      if (benchmark) {
9          benchmark->cpuTime = 0.0;
10         benchmark->gpuTime = 0.0;
11     }
12     double start = omp_get_wtime();
13 #pragma omp parallel for schedule(dynamic, 256)
       shared(y, x, matrix)
14     for (u_int64_t row = 0; row < matrix->row_size;
       row++) {
15         float dot = 0.0f;
16         int row_start = matrix->row_pointer[row];
17         int row_end = matrix->row_pointer[row + 1];
18         for (int elem = row_start; elem < row_end;
       elem++) {
19             dot += matrix->data[elem] * x->data[
       matrix->col_index[elem]];
20         }
21         y->data[row] += dot;
22     }
23     double end = omp_get_wtime();
24     if (benchmark) {
25         benchmark->cpuTime = (end - start) * 1000.0;
26     }
27     return SPMV_SUCCESS;
28 }
```

Seguendo l'algoritmo implementato per CUDA, si può notare come l'algoritmo su OMP sia molto simile nella sua struttura: si divide il lavoro in righe della matrice, sulle quali si calcola il parziale del prodotto matrice-vettore. Il fatto che la piattaforma di computazione e l'architettura siano differenti portano a scelte di ottimizzazione differenti. La scelta più importante è la suddivisione del lavoro per thread: nella versione CUDA è stata assegnata una singola riga per thread. Questa strategia, efficace nell'architettura CUDA, crea invece vari problemi di performance nell'architettura classica.

Un primo problema è il costo per thread molto più alto rispetto a CUDA: ogni thread dovrebbe effettuare un carico di lavoro almeno tale da giustificare il suo costo di creazione e gestione. Se, effettuando un benchmark dell'applicazione, si nota che si spende più tempo creando e organizzando i thread che eseguendo la computazione richiesta, la causa può essere una suddivisione del carico di lavoro errata.

Il secondo problema è noto come *false sharing*, un problema che nasce quando gli algoritmi che regolano la coerenza e l'invalidazione delle cache si incontrano con processi paralleli i quali accedono con pattern di lettura e scrittura a valori contenuti nello stesso blocco di cache: se il processo in scrittura cambia un singolo valore nel blocco di cache, l'intera cache viene invalidata; mentre per quanto riguarda i processi in lettura, questi potrebbero non dover accedere al valore modificato, ma agli altri valori che erano presenti nella cache e che non sono stati cambiati da nessun altro processo, seppure invalidati. Un esempio potrebbe essere assegnare ad ogni thread un valore su un array `A`: se un processo modifica il valore in `A[1]`, invalida la cache anche per i valori `A[0]` e `A[2]` e cosi via fino al riempimento della cache, rendendo l'accesso a quei valori più lenti per gli altri processi paralleli anche se i valori in `A[0]` e `A[2]` non hanno subito modifiche di alcun tipo. Per cercare di alleviare questi due problemi, si è scelto, tramite l'opzione `schedule(dynamic, 256)`, di separare la matrice in chunck da 256 righe consecutive. Questa scelta prova a dare a ogni thread un numero relativamente alto di righe, in modo che ognuno abbia in media abbastanza valori da processare, pur dipendendo dalla distribuzione dei valori

non nulli della matrice di input. Il maggiore numero di righe implica che un thread avrà più possibilità di lavorare su valori contigui in memoria, rendendoli più cache friendly e in teoria diminuendo il problema del false sharing: si ipotizza che con il maggior numero di righe assegnate per ogni thread sia più probabile che questi riescano a riempire la propria cache senza che altri possano invalidarla. La veridicità di questa ipotesi dipende dalla distribuzione della matrice.

IV. ELL: ELLPACK Sparse Matrix

Il formato CSR risulta avere due problemi:

- Gli accessi in memoria *noncoalesced* causati da come i thread accedono i valori memorizzati della matrice.
- Il problema della divergenza dei flussi di controllo causato dal fatto che le righe della matrice hanno un numero variabile di valori non nulli al loro interno.

Il formato ELL è derivato dal progetto ELLPACK, nato per risolvere il problema noto come *elliptic limit value*.

```
typedef struct ELLMatrix {
    float *data;
    u_int64_t *col_index;
    size_t data_size;
    u_int64_t data_row_size;
    u_int64_t data_col_size;
    u_int64_t num_elem;
    u_int64_t row_size;
    u_int64_t col_size;
    u_int64_t num_non_zero_elements;
} ELLMatrix;
```

Il formato ELL può essere spiegato partendo dal formato CSR e dai suoi limiti. A partire dal formato CSR:

1) si determina la riga con il maggiore numero di elementi non nulli.
2) ad ogni altra riga si aggiungono dei valori nulli alla loro fine fino a che ogni riga non abbia la stessa dimensione della riga più lunga; si tratta di un'operazione di **padding** sulle righe.

Si nota che il vettore `col_index[]` necessità di padding per mantenere la corrispondenza valore-indice di colonna. Infine si può memorizzare la matrice con padding in *column major order*, che in C significa prendere la matrice memorizzata per righe contigue e **trasporla**. Per questo nuovo formato si possono notare i seguenti punti:

- I marker, per segnare l'inizio e la fine delle righe presenti in CSR, non sono più necessari poiché è stata regolarizzata la dimensione delle righe. A questo punto è possibile sapere esattamente, dato un indice, a quale riga appartiene quest'ultimo: per fare ciò è sufficiente sapere il numero di elementi presenti in ogni riga che, nel caso in esame, è implementato tramite il campo `num_elem`.
- La regolarizzazione della lunghezza delle righe, tramite la tecnica di padding, elimina il problema della divergenza dei flussi di controllo: ogni thread lavora sullo stesso numero di elementi quindi il carico di lavoro è distribuito equamente.
- La trasposizione permette di rendere gli accessi in memoria *coalesced*: quando il thread 1 accede al primo elemento della prima riga, il thread 2 accede al primo elemento della seconda riga, e cosi via; la trasposizione rende questi elementi vicini in memoria.

A. Prodotto matrice vettore per ELL

```
__global__ void SpMV_ELL_kernel(u_int64_t num_rows,
    const float *data, const u_int64_t *col_index,
    u_int64_t num_elem, const float *x, float *y) {
    int row = blockDim.x * blockIdx.x + threadIdx.x;
    if (row < num_rows) {
        float dot = 0.0f;
        for (int i = 0; i < num_elem; i++) {
            int index = row + num_rows * i;
            dot += data[index] * x[col_index[index
    ]];
        }
        y[row] += dot;
    }
}
```

L'implementazione di un kernel parallelo per il formato ELL è molto semplice: ogni thread CUDA prende in carico una riga del prodotto matrice-vettore da computare, esegue il prodotto e salva il risultato. Per i motivi illustrati precedentemente, questo kernel è più efficiente nel calcolo rispetto al kernel basato su CSR: il formato ELL sfrutta più efficacemente le risorse per il calcolo parallelo. Il formato ELL introduce un nuovo problema, non presente nel formato CSR, il quale si può spiegare tramite un esempio patologico: si immagini una matrice sparsa da 1000 righe e 1000 colonne, delle quali 999 hanno un singolo elemento e la riga rimanente ha 1000 elementi. Rappresentando questa matrice in formato ELL, in particolare aggiungendo il padding, possiamo notare come la matrice ottenuta non ottenga alcuna compressione nella sua dimensione. Si sarebbe potuto effettuare il calcolo sulla matrice non compressa ottenendo uno speed-up nella computazione, perché la matrice sparsa, compressa nel formato ELL, per questo caso non ottiene alcuna compressione. In casi meno estremi quello che succede è che poche righe molto dense di elementi non nulli possono causare un aumento significativo dell'utilizzo della la quantità di memoria richiesta per salvare la matrice: questo è un problema perché si potrebbe non avere abbastanza memoria per contenere l'intera matrice andando a ottenere fenomeni di swap, i quali degradano estremamente le prestazioni, fino al caso in extremis dell'impossibilità di eseguire il calcolo.

B. Approccio ibrido al prodotto matrice-vettore ELL

Per risolvere il problema fin qui descritto si utilizza un approccio ibrido:

1) si partiziona la matrice in due parti, tali che nella prima siano presenti tutte le righe il cui numero massimo di elementi non nulli non superi una certa soglia **N** e le rimanenti nell'altra.
2) si rappresentano le due partizioni con formati di matrici sparse differenti tra loro: la partizione con pochi elementi per riga sarà rappresentata nel formato ELL mentre la partizione con righe molto dense di elementi sarà rappresentata con il formato per coordinate COO.
3) si calcola il prodotto per le due partizione separatamente, visto che per il prodotto matrice-vettore la computazione del parziale per riga non necessità di informazioni prese dalla computazione sulle altre righe, e infine si riuniscono i risultati tramite una somma dei vettori parziali.

Questa soluzione risolve il problema causato dal padding di ELL poiché è possibile deciderne la soglia massima. Se si computa il prodotto matrice-vettore per il formato COO sulla CPU si ottiene un ambiente di calcolo parallelo ibrido in cui questo è eseguito in parallelo su GPU e CPU, andando a utilizzare tutte le risorse di calcolo disponibili. Questo approccio porta dei problemi di non banale risoluzione, quali la scelta del parametro di partizione **N**. In generale si desidera avere il più alto numero possibile di righe nel formato ELL, poiché il calcolo sulla GPU è più perfomante di quello sulla CPU, ma se si sceglie un parametro troppo alto si rientra nel problema del padding spiegato precedentemente, rendendo questo approccio inefficiente; inoltre questa scelta viene anche influenzata dalla distribuzione dei valori non nulli della matrice. Nel progetto si è scelto di impostare un numero massimo di valori non nulli per ogni riga poiché è stata preferita la semplicità di implementazione. Il numero massimo di valori ammessi è 64, determinato tramite una serie di trial and error cercando il miglior rapporto tra prestazioni e occupazione di memoria.

Per effettuare un singolo prodotto matrice-vettore, la matrice ELL e il vettore devono essere trasferiti sulla GPU per eseguire il calcolo, e da quest'ultima, il vettore risultato deve tornare sulla memoria host per essere unito al risultato della computazione su COO: questa operazione risulta quindi in un overhead significativo, ma nel caso in cui venga eseguita molte volte sulla stessa matrice l'overhead si ammortizza con il numero di iterazioni effettuate.

C. ELL su OMP

```
int ELLMatrix_SpMV_OMP(const ELLMatrix* matrix,
    const Vector* x, Vector* y, Benchmark *benchmark
    ) {
    if (!matrix || !x || !y) {
        return SPMV_FAIL;
    }

    if (x->size != matrix->col_size && y->size !=
    matrix->row_size) {
        return SPMV_FAIL;
    }

    if (benchmark) {
        benchmark->cpuTime = 0.0;
        benchmark->gpuTime = 0.0;
    }
    double start = omp_get_wtime();

#pragma omp parallel for schedule(dynamic, 256)
    shared(y, x, matrix)
    for (u_int64_t row = 0; row < matrix->row_size;
    row++) {
        float dot = 0.0f;
        for (u_int64_t i = 0; i < matrix->num_elem;
    i++) {
            u_int64_t index = row * matrix->num_elem
     + i;
            dot += matrix->data[index] * x->data[
    matrix->col_index[index]];
        }
       y->data[row] += dot;
    }
    double end = omp_get_wtime();
    if (benchmark) {
        benchmark->cpuTime = (end - start) * 1000.0;
    }

    return SPMV_SUCCESS;
}
```

Per il prodotto riga-vettore di ELL su OMP si possono fare considerazioni simili a quelle fatte per il kernel CUDA, infatti si può notare come la struttura del calcolo sia identica, tuttavia l'architettura di computazione differente implica scelte di ottimizzazione differenti; in CUDA la matrice in formato ELL è memorizzata in *column major order* per renderla più friendly agli accessi dei thread cuda, i quali condividevano le loro cache. Per i thread classici invece questo tipo di condivisione non solo non è efficiente perché le cache L1 non sono condivise, ma può risultare dannoso per il fenomeno del false sharing: i thread classici preferiscono lavorare su zone contigue in memoria che non vengono accedute da altri thread in scrittura, quindi la matrice viene memorizzata in *row major order* per fare in modo che ogni thread lavori su linee della matrice contigue. La seconda differenza con cuda è il bilanciamento del workload: ogni thread CUDA lavorava con una sola riga, mentre su OMP un singolo thread lavora su 256 righe, ciò avviene poiché ogni thread ha più overhead, quindi risulta migliore che effettui più lavoro per giustificarne l'esistenza. Inoltre le righe consecutive sono disposte in maniera contigua in memoria, quindi è meglio che se ne occupi un singolo thread piuttosto che avere thread multipli lavorare su linee vicine e poi dover saltare su righe distanti conclusa la computazione, non sfruttando la località delle cache per la computazione successiva.

V. Conclusioni

A. Benchmark

Il benchmark è stato strutturato nel seguente modo: data una serie di matrici sparse, queste sono convertite nei formati CSR e ELL ed è eseguito il prodotto matrice-vettore 512 volte per ognuna di esse sullo stesso vettore. Quindi il benchmark calcola:

$$M^{512} \cdot x = M \cdots (M \cdot (M \cdot X)) \tag{1}$$

Per ogni prodotto è salvato il tempo di esecuzione della computazione e alla fine è calcolato il tempo medio di esecuzione T_m per ogni matrice. Quest'ultimo è usato per calcolare i FLOPS utilizzando la seguente formula:

$$FLOPS = 2 \cdot \frac{nz}{T_m} \tag{2}$$

con nz il numero di elementi non nulli per la matrice specifica.

B. Risultati

Il grafico in figura 1 rappresenta il risultato degli algoritmi implementati, distinti per formato e device di computazione con le matrici ordinate per il numero di valori non nulli. Dal grafico possiamo affermare che:

- Le implementazioni su CUDA hanno performance migliori rispetto alle implementazioni su OMP. Quest'ultimo era un risultato atteso perché CUDA offre un ambiente parallelo molto più efficiente rispetto all'ambiente classico fornito da OMP.
- tra le implementazioni OMP l'implementazione ELL/-COO è quella che restituisce le prestazioni peggiori. La

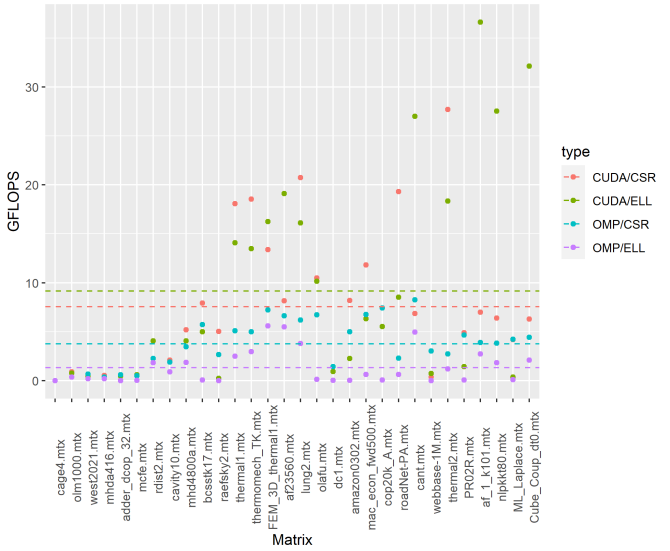

Fig. 1. Grafico dei GFLOPS risultanti per metodologia con le matrici ordinate per numero di valori non nulli

separazione in due formati differenti non è una scelta utile per le performance perché a differenza di CUDA il calcolo per le righe lunghe in COO va a togliere risorse al calcolo per ELL, tuttavia è necessaria per poter eseguire il calcolo delle matrici prese in esame le quali con la compressione data solo dal formato ELL non riescono ad essere contenute in memoria.

- tra le implementaziomi CUDA possiamo osservare che non c'è un formato che è sempre più performante dell'altro ma che le perforamance dipendono dalla distribuzione dei valori non nulli della matrice in esame. Si può osservare come per le matrici sparse con un più alto numero di valori non nulli il formato ELL/COO tenda ad essere il migliore, mentre per valori più bassi la linea non è definita chiaramente. Se si guardano i GFLOPS medi per i formati, possiamo vedere che il formato ELL/COO da risultati migliori.

Nella sezione precedente sono state approfondite le scelte fatte per migliorare le performance specifiche del prodotto matrice-vettore parallelo, in questa sezione si discute su ottimizzazioni effettuate non specifiche del prodotto, ma comunque importanti per le performance generali dell'applicazione.

Appendix A
Trasferimento in memoria CUDA

Uno dei problemi incontrati è stato il trasferimento di dati tra l'host e il device CUDA. CUDA offre delle funzioni per allocare memoria sul device nella forma della coppia `cudaMalloc()` e `cudaFree()` e per copiare dati tra l'host e il device CUDA tramite la funzione `cudaMemcpy()`. La funzione `cudaMemcpy()` offre un trasferimento sincrono, il quale non permette di sfruttare a pieno la capacità di banda offerta dal dispositivo nel caso in cui si debbano effettuare più copie in sequenza.

Per migliorare i tempi di trasferimento, CUDA offre la funzione `cudaMemcpyAsync()`: questa permette un trasferimento di dati asincrono che per il caso d'uso richiesto, ovvero trasferire da host a device CUDA una serie di dati in sequenza, risulta ottenere un aumento delle performance non trascurabili. Tuttavia questa funzione richiede una particolare attenzione riguardo l'allocazione della memoria sull'host, perché se ne è allocata una del tipo errato si possono creare bug molto subdoli da rintracciare.

Per utilizzare le API async è necessario utilizzare un tipo di memoria definito **pinned memory**. Il problema che la pinned memory si propone di risolvere è il seguente: se si vuole copiare una pagina di memoria dall'host al device CUDA, può succedere che l'host decida di spostare la pagina dalla memoria RAM allo swap; questo causa un page fault nella lettura della pagina e la conseguente richiesta, tramite la DMA, di ricaricare la pagina in memoria causando un perdita di prestazioni significativa. CUDA per risolvere questo problema fornisce queste pagine di memoria bloccate (locked, pinned), le quali non possono essere spostate dalla RAM, eliminando il problema appena descritto.

La funzione sincrona `cudaMemcpy()` opera nel seguente modo:

1) copia una parte dei dati di input in un buffer intermedio allocato sulla memoria pinned.
2) dal buffer intermedio copia sul device CUDA.
3) torna al punto 1) finché tutti i dati non sono stati copiati.

Questo metodo è molto costoso in termini di prestazioni sia perché la copia è indiretta sia perché la dimensione del buffer influenza la banda massima che possiamo utilizzare.

Quindi per utilizzare le API asincrone dobbiamo allocare i dati sul sistema host direttamente sulla memoria `pinned`.

Se alla funzione asincrona si passano dei dati allocati su pagine di memoria non bloccate, la funzione non restituisce un errore ma opera con un comportamento errato e silente: infatti, aspettandosi che la pagina di memoria in input sia bloccata, non eseguirà alcun controllo nel caso in cui la pagina venga spostata durante il trasferimento; in tal caso la funzione continuerà a copiare dalla pagina di memoria anche se non contiene più i dati originali. CUDA offre due funzioni per gestire la memoria pinned:

- `cudaHostAlloc()`
- `cudaHostFree()`

Appendix B
Scelta della dimensione di blocco su CUDA

In Cuda tutti i thread in una griglia eseguono la stessa funzione kernel; questi ultimi utilizzano le loro coordinate nella griglia per distinguersi tra loro e assegnarsi il lavoro da eseguire. Questi thread sono suddivisi in una gerarchia a due livelli: una griglia è formata da blocchi e un blocco è formato da thread. I thread dello stesso blocco condividono lo stesso indice di blocco, mentre sono distinti dall'indice di thread. Al lancio del kernel devono essere specificati i parametri `gridDim` e `blockDim` i quali indicano:

- `gridDim`: quanti blocchi deve avere una griglia;
- `blockDim`: quanti thread deve avere un blocco.

In generale, una griglia è un array a 3 dimensioni di blocchi e un blocco è un array a 3 dimensioni di thread. Il secondo problema nell'ottimizzare il valore di questi parametri è che questa rappresentazione ideale dei thread dovrà essere eseguita su un dispostivo reale con un un architettura definita: un device CUDA è diviso in SM (streaming multiprocessor) in cui ognuno di questi viene a sua volta diviso in SP (streaming processor), i quali hanno un numero stabilito di CUDA cores; infine ogni core può eseguire un warp, ovvero un gruppo di 32 thread. Ogni blocco viene eseguito sullo stesso SM e, nel caso in cui tutte le capacità di calcolo del SM siano occupate, il blocco dovrà attendere il proprio turno. Ogni SM ha un numero massimo di blocchi del quale può occuparsi: questo implica che avere un singolo blocco che contiene tutti i thread da eseguire è una strategia fallimentare, perché si utilizzerebbe un solo SM, andando a sotto utilizzare le risorse del dispositivo. In generale alcuni thread dovranno forzatamente aspettare il proprio turno di esecuzione se le risorse di calcolo sono inferiori alla quantità di lavoro richiesta. Scegliere una dimensione di blocco molto alta risulta essere controproducente poiché nonostante il numero di thread per blocco virtualmente non abbia un limite, il numero di risorse per la computazione presenti su un SM ha un limite dovuto dal tipo di hardware in utilizzo; quindi avere più thread che risorse da utilizzare comporta un deterioramento delle prestazioni. Un'altra nota da esplicitare è che la dimensione dei blocchi dovrebbe essere sempre un multiplo della dimensione dei warp, altrimenti si avranno sempre dei thread inutilizzati. Lo scopo di questa introduzione è di affermare che la scelta dei parametri di griglia e di blocco influiscono sulle performance del codice e che una scelta errata può essere disastrosa. Nel progetto le risorsa da suddividere in griglie e blocchi è l'indice di riga della matrice, il quale è un indirizzo lineare. Per fare ciò è stata utilizzata un utility fornita da CUDA chiamata `cudaOccupancyMaxPotentialBlockSize()`, la quale cerca di dare la miglior dimensione per il block size dato il kernel da eseguire.

Appendix C
Parallelizzazione SpMV/COO

Per il prodotto matrice-vettore per il formato ELL, è stato utilizzato il formato COO per rappresentare le righe più dense della matrice e calcolare il prodotto su di esse, tuttavia l'implementazione fornita è sequenziale. Se si potesse parallelizzare la procedura si potrebbe ottenere un notevole aumento di performance.

```
int COOMatrix_SpMV(const COOMatrix* matrix, const
    Vector* x, Vector* y, Benchmark* benchmark) {
    clock_t start, end;
    if (!matrix || !x || !y) {
        return SPMV_FAIL;
    }

    if (x->size != matrix->col_size && y->size !=
    matrix->row_size) {
        return SPMV_FAIL;
    }
    start = clock();
    for (u_int64_t i = 0; i < matrix->
    num_non_zero_elements; i++) {
        y->data[matrix->row_index[i]] += matrix->
    data[i] * x->data[matrix->col_index[i]];
    }
    end = clock();
    if (benchmark) {
        benchmark->gpuTime = 0.0;
        benchmark->cpuTime = ((double)(end - start))
     / (double)CLOCKS_PER_SEC;
    }
    return SPMV_SUCCESS;
}
```

Questa è l'implementazione sequenziale fornita la quale, se fosse parallelizzabile, potrebbe aumentare le prestazione del prodotto matrice-vettore per il formato ELL/COO. Dopo una ricerca tramite Google è stata trovata la seguente soluzione:

```
__global__ void COOMatrix_SpMV(float *data, int *
    rows, int *cols, float *x, float *y, int
    num_not_zeroes_elem) {
    const int index = blockIdx.x * blockDim.x +
    threadIdx.x;
    if (index < num_not_zeroes_elem) {
        float dot = data[index] * x[cols[index]]
        atomicAdd(&y[rows[index], dot)
    }
}
```

Questo kernel è una trasposizione dell'algoritmo sequenziale in cui ad ogni thread viene assegnato un singolo elemento della matrice calcolando un parziale del prodotto riga-vettore per quella particolare riga, ed infine aggiunge il suo risultato con un addizione atomica per evitare race condition sulla somma. Questo kernel per quanto sembri corretto, in realta è completamente errato: contiene al suo interno un bug molto subdolo il quale avviene quando si mettono insieme il calcolo parallelo e `l'aritmetica float`. L'aritmetica float ha una caratteristica che la differenzia dall'aritmetica matematica, **le somme non sono associative** ovvero $x + y \neq y + x$. Questo fenomeno unito alla parallelizzazione implica che se questo kernel fosse eseguito sulla stessa matrice e lo stesso vettore n volte si otterrebbero n risultati differenti. Questo fenomeno è dovuto all'ordine casuale con il quale i thread ottengono l'accesso all'operazione atomica. Nel progetto non è stato implementato un algoritmo parallelo per il prodotto matrice-vettore su COO. Esistono implementazioni parallele che evitano il problema fin qui preso in analisi, ma usano algoritmi più sofisticati e complicati per evitarlo. Per le prospettive del progetto questi algoritmi più sofisticati non sono stati presi in considerazione perché ritenuti fuori dal suo scopo.

Appendix D
Tabelle dei risultati

In seguito sono riportati i risultati divisi in 4 tabelle separate per il framework parallelo (CUDA, OMP) e il formato utilizzato per rappresentare la matrice sparsa.

Matrix	Format	Device	Time	FLOPS
cavity10.mtx	CSR	CUDA	0.000073	2.097544
roadNet-PA.mtx	CSR	CUDA	0.000160	19.281770
nlpkkt80.mtx	CSR	CUDA	0.004649	6.403314
mac_econ_fwd500.mtx	CSR	CUDA	0.000215	11.818794
thermomech_TK.mtx	CSR	CUDA	0.000044	18.554057
PR02R.mtx	CSR	CUDA	0.003352	4.883847
thermal2.mtx	CSR	CUDA	0.000355	27.602134
FEM_3D_thermal1.mtx	CSR	CUDA	0.000064	13.391653
olafu.mtx	CSR	CUDA	0.000098	10.503180
amazon0302.mtx	CSR	CUDA	0.000302	8.180687
adder_dcop_32.mtx	CSR	CUDA	0.000715	0.031472
mhda416.mtx	CSR	CUDA	0.000033	0.525449
rdist2.mtx	CSR	CUDA	0.000028	4.039224
olm1000.mtx	CSR	CUDA	0.000009	0.918891
webbase-1M.mtx	CSR	CUDA	0.017869	0.347583
cant.mtx	CSR	CUDA	0.000593	6.859780
af_1_k101.mtx	CSR	CUDA	0.002585	6.984064
mhd4800a.mtx	CSR	CUDA	0.000040	5.171540
thermal1.mtx	CSR	CUDA	0.000037	17.978932
cop20k_A.mtx	CSR	CUDA	0.000367	7.430059
bcsstk17.mtx	CSR	CUDA	0.000056	7.913471
raefsky2.mtx	CSR	CUDA	0.000117	5.029762
cage4.mtx	CSR	CUDA	0.000009	0.010473
Cube_Coup_dt0.mtx	CSR	CUDA	0.020583	6.285412
mcfe.mtx	CSR	CUDA	0.000080	0.611824
west2021.mtx	CSR	CUDA	0.000022	0.657768
ML_Laplace.mtx	CSR	CUDA	0.013180	4.201842
lung2.mtx	CSR	CUDA	0.000048	20.641436
af23560.mtx	CSR	CUDA	0.000119	8.146565
dc1.mtx	CSR	CUDA	0.065268	0.023485

Matrix	Format	Device	Time	FLOPS
cavity10.mtx	CSR	OMP	0.000081	1.891112
roadNet-PA.mtx	CSR	OMP	0.001277	2.414736
nlpkkt80.mtx	CSR	OMP	0.007904	3.766030
mac_econ_fwd500.mtx	CSR	OMP	0.000387	6.575443
thermomech_TK.mtx	CSR	OMP	0.000170	4.787538
PR02R.mtx	CSR	OMP	0.003551	4.609598
thermal2.mtx	CSR	OMP	0.003663	2.677541
FEM_3D_thermal1.mtx	CSR	OMP	0.000122	7.061810
olafu.mtx	CSR	OMP	0.000151	6.851007
amazon0302.mtx	CSR	OMP	0.000528	4.675462
adder_dcop_32.mtx	CSR	OMP	0.000033	0.690395
mhda416.mtx	CSR	OMP	0.000048	0.358461
rdist2.mtx	CSR	OMP	0.000051	2.238723
olm1000.mtx	CSR	OMP	0.000022	0.358244
webbase-1M.mtx	CSR	OMP	0.002052	3.026583
cant.mtx	CSR	OMP	0.000501	8.120996
af_1_k101.mtx	CSR	OMP	0.004372	4.129246
mhd4800a.mtx	CSR	OMP	0.000059	3.453059
thermal1.mtx	CSR	OMP	0.000129	5.107630
cop20k_A.mtx	CSR	OMP	0.000373	7.304630
bcsstk17.mtx	CSR	OMP	0.000076	5.777312
raefsky2.mtx	CSR	OMP	0.000222	2.647903
cage4.mtx	CSR	OMP	0.000014	0.007073
Cube_Coup_dt0.mtx	CSR	OMP	0.036808	3.514731
mcfe.mtx	CSR	OMP	0.000078	0.624332
west2021.mtx	CSR	OMP	0.000022	0.667778
ML_Laplace.mtx	CSR	OMP	0.012915	4.287989
lung2.mtx	CSR	OMP	0.000160	6.173618
af23560.mtx	CSR	OMP	0.000145	6.660971
dc1.mtx	CSR	OMP	0.001104	1.387967

Matrix	Format	Device	Time	FLOPS
cavity10.mtx	ELL	CUDA	0.000081	1.893509
roadNet-PA.mtx	ELL	CUDA	0.000363	8.495816
nlpkkt80.mtx	ELL	CUDA	0.001084	27.466086
mac_econ_fwd500.mtx	ELL	CUDA	0.000404	6.299408
thermomech_TK.mtx	ELL	CUDA	0.000060	13.639122
PR02R.mtx	ELL	CUDA	0.011511	1.422103
thermal2.mtx	ELL	CUDA	0.000537	18.254400
FEM_3D_thermal1.mtx	ELL	CUDA	0.000053	16.159005
olafu.mtx	ELL	CUDA	0.000102	10.093906
amazon0302.mtx	ELL	CUDA	0.001072	2.303563
adder_dcop_32.mtx	ELL	CUDA	0.000049	0.454738
mhda416.mtx	ELL	CUDA	0.000050	0.344349
rdist2.mtx	ELL	CUDA	0.000028	4.059013
olm1000.mtx	ELL	CUDA	0.000010	0.770332
webbase-1M.mtx	ELL	CUDA	0.008458	0.734359
cant.mtx	ELL	CUDA	0.000151	27.007867
af_1_k101.mtx	ELL	CUDA	0.000496	36.373155
mhd4800a.mtx	ELL	CUDA	0.000051	4.036859
thermal1.mtx	ELL	CUDA	0.000047	13.993854
cop20k_A.mtx	ELL	CUDA	0.000499	5.459159
bcsstk17.mtx	ELL	CUDA	0.000089	4.963128
raefsky2.mtx	ELL	CUDA	0.002719	0.216499
cage4.mtx	ELL	CUDA	0.000014	0.006938
Cube_Coup_dt0.mtx	ELL	CUDA	0.004028	32.116746
mcfe.mtx	ELL	CUDA	0.000084	0.581727
west2021.mtx	ELL	CUDA	0.000037	0.396989
ML_Laplace.mtx	ELL	CUDA	0.148302	0.373426
lung2.mtx	ELL	CUDA	0.000061	16.192610
af23560.mtx	ELL	CUDA	0.000051	19.112980
dc1.mtx	ELL	CUDA	0.001668	0.918682

Matrix	Format	Device	Time	FLOPS
cavity10.mtx	ELL	OMP	0.000171	0.891922
roadNet-PA.mtx	ELL	OMP	0.004856	0.635059
nlpkkt80.mtx	ELL	OMP	0.016245	1.832413
mac_econ_fwd500.mtx	ELL	OMP	0.003940	0.646403
thermomech_TK.mtx	ELL	OMP	0.000279	2.919140
PR02R.mtx	ELL	OMP	0.312710	0.052350
thermal2.mtx	ELL	OMP	0.008217	1.193726
FEM_3D_thermal1.mtx	ELL	OMP	0.000153	5.612392
olafu.mtx	ELL	OMP	0.007668	0.134493
amazon0302.mtx	ELL	OMP	0.091619	0.026957
adder_dcop_32.mtx	ELL	OMP	0.001814	0.012397
mhda416.mtx	ELL	OMP	0.000110	0.156294
rdist2.mtx	ELL	OMP	0.000068	1.682997
olm1000.mtx	ELL	OMP	0.000026	0.309621
webbase-1M.mtx	ELL	OMP	0.541377	0.011473
cant.mtx	ELL	OMP	0.000839	4.851062
af_1_k101.mtx	ELL	OMP	0.006329	2.852476
mhd4800a.mtx	ELL	OMP	0.000113	1.805390
thermal1.mtx	ELL	OMP	0.000255	2.579029
cop20k_A.mtx	ELL	OMP	0.038275	0.071174
bcsstk17.mtx	ELL	OMP	0.005386	0.081630
raefsky2.mtx	ELL	OMP	0.110582	0.005322
cage4.mtx	ELL	OMP	0.000014	0.007113
Cube_Coup_dt0.mtx	ELL	OMP	0.061462	2.104887
mcfe.mtx	ELL	OMP	0.002093	0.023298
west2021.mtx	ELL	OMP	0.000078	0.187623
ML_Laplace.mtx	ELL	OMP	0.649208	0.085304
lung2.mtx	ELL	OMP	0.000262	3.759858
af23560.mtx	ELL	OMP	0.000176	5.502582
dc1.mtx	ELL	OMP	0.097119	0.015783