Progetto SCPA

Fabiano Veglianti

Indice

1 Introduzione

Questo documento descrive come viene affrontato il problema della realizzazione di kernel SpMV, cioè nuclei di calcolo per la moltiplicazione tra una matrice sparsa e un vettore.

Si inizierà con una breve descrizione del problema, si vedrà come si è affrontato, inziando dall'acquisizione dei dati, proseguendo con la conversione nei formati usati e con nuclei di calcolo sviluppati per l'esecuzione su CPU e per l'esecuzione su GPU. Infine si discuteranno le prestazioni ottenute ed eventuali criticità del lavoro svolto.

1.1 SpMV

Il problema della moltiplicazione tra matrice sparsa e vettore è un problema che si presenta in molti ambiti, dunque è necessario saper effettuare questa operazione efficientemente.

Nel caso peggiore (matrice densa), il problema ha complessità $O(M * N^2)$; tuttavia, in caso di matrice sparsa si può e si deve sfruttare il fatto che la maggioranza degli elementi della matrice sono zero e che quindi possono essere non compresi nei calcoli effettuati. In particolare, nel caso in cui $NZ << M*N$, cioè il numero di non zero della matrice è molto minore della dimensione della matrice stessa, il problema ha una complessità $O(NZ * N)$ in quanto, come già detto, si considerano solo i non-zero e gli elementi del vettore.

Dal punto di vista della complessità spaziale, anche quì si sfrutta il fatto che solo gli elementi diversi da zero contribuiscono al calcolo del risultato per omettere (o cercare di omettere) la memorizzazione degli elementi nulli. Dunque, anziché memorizzare tutti gli $M*N$ elementi della matrice, si memorizzano solo gli NZ elementi, più alcune informazioni ausiliarie necessarie.

2 Acquisizione e memorizzazione dei dati

2.1 Acquisizione

Le matrici considerate per il test sono le matrici prese dal sito https://sparse.tamu.edu/ indicate nella consegna, cioè:

cage4	Cube_Coup_dt0	FEM_3D_thermal1
mhda416	ML_Laplace	thermal1
mcfe	bcsstk17	thermal2
olm1000	mac_econ_fwd500	thermomech_TK
adder_dcop_32	mhd4800a	nlpkkt80
west2021	cop20k_A	webbase-1M
cavity10	raefsky2	dc1
rdist2	af23560	amazon0302
cant	lung2	af_1_k101
olafu	PR02R	roadNet-PA

la prima operazione effettuata dopo il download è stata la lettura delle matrici.

Per la lettura delle matrici si sono utilizzate le funzioni fornite all'indirizzo http://math.nist.gov/MarketMatrix in quanto tutte sono memorizzate nel formato MarketMatrix. Per modellarle nel codice si è creata una struct apposita, **sparsematrix**.

Durante la lettura delle matrici si è tenuto conto del tipo di matrice, cioè, andando nel dettaglio:

- per le matrici "pattern", durante la lettura, si è assegnato il valore 1 agli elementi della matrice letti, in quanto tale è il valore omesso nella memorizzazione delle matrici di questo tipo.
- per le matrici "symmetric" si è ricostruita la parte di matrice omessa nella memorizzazione; tale ricostruzione è avvenuta contando gli elementi al di fuori della diagonale principale e raddoppiandoli scambiando contemporaneamente l'indice di riga con l'indice di colonna.

2.2 Pre-Conversione

Prima di convertire le matrici nei formati utilizzati nei nuclei di calcolo, cioè CSR e ELLPACK, si sono ordinati gli elementi della matrice in ordine crescente

di indice di riga, dunque in ordine crescente di indice di colonna. L'ordinamento è effettuato tramite un implementazione inplace dell'algoritmo QuickSort.

2.3 Formato CSR

Il formato CSR prevede la memorizzazione di 3 vettori:

- $IRP[M+1] : IRP[i]$ contiene l'indice del primo elemento dell'i-esima riga.
- $JA[NZ] : JA[k]$ contiene l'indice di colonna del k-esimo elemento NZ della matrice.
- $AS[NZ] : AS[k]$ contiene il valore del k-esimo elemento NZ della matrice.

oltre a ciò, nella struct che modella questo formato si mantengono M e N, numero di righe e di colonne della matrice. Avendo ordinato gli elementi della matrice, i vettori JA e AS sono esattamente i vettori di indici di colonna e di valori letti dal file; il vettore IRP, invece, è stato costruito utilizzando un vettore ausiliario *counters* che mantiene, per ogni riga, il numero di NZ in quella riga, dunque IRP è stato popolato come segue:

```
irp[0] = 0;
for(int i = 0; i < M; i++){
    irp[i + 1] = irp[i] + counters[i];
}
```

si noti che in virtù di ciò, supponendo che la riga di indice i sia vuota, si ha che $IRP[i+1] = IRP[i]$.

2.4 Formato ELLPACK

Il formato ELLPACK prevede la memorizzazione di 2 array 2D:

- $JA[M][MAXNZ] : JA[i][j]$ contiene l'indice di colonna del j-esimo NZ dell'i-esima riga.
- $AS[M][MAXNZ] : AS[i][j]$ contiene il valore del j-esimo NZ dell'i-esima riga.

oltre a ciò, nella struct che modella questo formato si mantengono M e N, numero di righe e di colonne della matrice e MAXNZ, massimo numero di NZ presenti su una riga della matrice.

In generale non tutte le righe della matrice hanno MAXNZ valori non zero, dunque le array JA e AS vanno riempite appositamente per gestire questi casi: per le righe che non hanno MAXNZ valori non zero viene effettuato 0-padding fino a raggiungere quella lunghezza, in JA, l'elemento corrispondente è riempito con l' indice di colonna dell'ultimo NZ incontrato lungo la riga.

Per memorizzare la matrice in questo formato si sono utilizzate due variabili temporanee: *sparse_matrix_element_index* che permette di scorrere i NZ della matrice e *last_col_index_in_row* che mantiene l'ultimo indice di colonna di un

NZ incontrato lungo la riga. Il codice prodotto funziona correttamente solo se prima si è proceduto con l'ordinamento crescente degli elementi della matrice per indice di riga, quindi indice di colonna.

3 Nuclei di calcolo

Per la risoluzione del problema sono stati realizzati diversi nuclei di calcolo pralleleo e due nuclei di calcolo seriale, uno per formato di memorizzazione, usati per assicurare la correttezza del risultato ottenuto e per valutare il miglioramento delle prestazioni ottenuto dalle implementazioni parallele.

3.1 Calcolo seriale

Come già detto, per il calcolo seriale sono stati realizzati due nuclei di calcolo, uno per ogni formato di memorizzazione usato. Questa differenziazione è stata fatta per poter valutare il miglioramento delle prestazioni a parità di formato di memorizzazione.

3.1.1 Calcolo seriale - CSR

Il nucleo di calcolo seriale per il formato CSR è il seguente:

```
double result;
int row_start_index, row_end_index;
for(int row = 0; row < M; row++){
    row_start_index = csr->IRP[row];
    row_end_index = csr->IRP[row+1];
    result = 0.0;
    for(int j = row_start_index; j < row_end_index; j++){
        int col_index = csr->JA[j];
        double mval = csr->AS[j];
        double vval = vec->val[col_index];
        result += mval * vval;
    }
    val[row] = result;
}
```

si è utilizzata la variabile `result` per non dover aggiornare il valore di `val[row]` in cache ad ogni iterazione del ciclo interno, infatti la scrittura di `val[row]` avviene una sola volta, alla fine del ciclo interno.

3.1.2 Calcolo seriale - ELLPACK

Il nucleo di calcolo seriale per il formato ELLPACK è il seguente:

```
double result;
int row;
for(int i = 0; i < M; i++){
    row = i * maxnz;
    result = 0.0;
    for(int j = 0; j < maxnz; j++){
        int col_index = ellpack->JA[row+j];
        double mval = ellpack->AS[row+j];
        double vval = vec->val[col_index];
        result += mval * vval;
    }
    val[i] = result;
}
```

anche per il formato ELLPACK si è fatta la stessa considerazione sulla variabile `result`.

3.2 Calcolo parallelo - CPU

Sono stati realizzati due nuclei di calcolo parallelo, uno per ciascun formato di memorizzazione scelto. In entrambi i casi si sono utilizzate le direttive di OpenMP per distribuire le righe della matrice tra i vari threads.

3.2.1 Calcolo parallelo - CPU - CSR

Il nucleo di calcolo parallelo su CPU per il formato CSR è:

```
int M = csr->M;
int nonzerosPerRow = nz/M;
int chunksize = NZ_PER_CHUNK/nonzerosPerRow;
...
int i,j,tmp;
#pragma omp parallel
{
    #pragma omp for private(i, j, tmp) schedule(dynamic, chunksize)
    for (i=0; i<M; i++)
    {
        double result = 0.0;
        #pragma omp simd reduction(+ : result)
        for (j = csr->IRP[i]; j < csr->IRP[i+1]; j++)
        {
            tmp = csr->JA[j];
            result += csr->AS[j] * vec->val[tmp];
        }
        val[i] = result;
    }
}
```

il ciclo più esterno, quello sulle righe, è stato suddiviso tra i vari threads utilizzando uno schedule dinamico con una chunksize scelta ad hoc per ogni matrice in modo tale che ogni chunk includa mediamente NZ_PER_CHUNK non-zeri.

La scelta tra uno schedule dinamico e statico non vede vincitore sempre lo schedule dinamico, infatti questo risulta migliore per le matrici che hanno un numero molto variabile di non-zeri per riga perché permette di bilanciare maggiormente il carico tra i vari threads; tuttavia, introduce un overhead maggiore rispetto allo schedule statico, il quale quindi risulta vincente se il bilanciamento del carico tra i threads avviene naturalmente grazie al fatto che il numero di non-zeri per riga è poco variabile.

Per quanto riguarda il ciclo interno si è voluta applicare la vettorizzazione per velocizzarne l'esecuzione. La vettorizzazione del ciclo interno dovrebbe, in teoria, avvenire automaticamente specificando l'opzione $-O3$ al compilatore *gcc*, tuttavia per essere più sicuri che ciò accada si è specificato esplicitamente di volerla avere tramite la direttiva `#simd`.

Per verificare che il processore supporti la vettorizzazione è necessario verificare che il processore sia provvisto del set di istruzioni *Advanced Vector Extension* e ciò si può fare digitando il comando `grep avx /proc/cpuinfo`.

La vettorizzazione del ciclo interno della spmv con formato CSR ha comunque 2 punti deboli:[1]

1. **Loop Remainder**: vettorizzando le iterazioni sulla riga, se il numero di non zero sulla riga non è un multiplo della *SIMD vector lenght* si hanno delle iterazioni rimanenti che non possono essere eseguite alla stessa velocità delle altre. Questo aspetto non è caratterizzante se si riescono a vettorizzare tante iterazioni, cioè, in altri termini, se il numero di non zeri per riga è abbastanza grande.

2. **Data Locality**: l'accesso agli elementi della matrice avviene riga per riga e il formato di memorizzazione garantisce gli elementi sulla stessa riga siano vicini in memoria, dunque il pattern di accesso agli elementi della matrice è sequenziale . Tuttavia, l'accesso agli elementi del vettore è per indirizzamento indiretto, e in particolare questo comporta che se i non-zeri sulle righe sono distribuiti in modo random, anche l'accesso agli elementi del vettore seguirà un pattern randomico, dunque non si può sfruttare la località dei dati.

Si noti che, come nel caso sequenziale, si utilizza la variabile `result` evitare di aggiornare il valore in cache ad ogni iterazione del ciclo interno.

3.2.2 Calcolo parallelo - CPU - ELLPACK

Il nucleo di calcolo parallelo su CPU per il formato ELLPACK è:

```
int i,j,tmp, row;
#pragma omp parallel
{
    #pragma omp for private(i, j, tmp, row) schedule(static)
    for (i=0; i<M; i++)
    {
        double result = 0.0;
        row = i * maxnz;
        #pragma omp simd reduction(+ : result)
        for (j = 0; j < maxnz; j++)
        {
            tmp = ellpack->JA[row + j];
            result += ellpack->AS[row + j] * vec->val[tmp];
        }
        val[i] = result;
    }
}
```

il ciclo più esterno, quello sulle righe, è stato suddiviso tra i vari threads utilizzando uno schedule statico, in questo caso, infatti, il bilanciamento del carico è naturalmente indotto dal formato di memorizzazione: tutte le righe, dopo il padding, hanno la stessa lunghezza, quindi non è necessario introdurre un overhead per cercare di bilanciare artificialmente il carico tra i threads.

Come nel caso precedente, per il ciclo interno si è tentato di abilitare esplicitamente la vettorizzazione tramite l'apposita direttiva. Rimangono le stesse criticità di loop remainder e data locality che si avevano nel caso CSR.

Anche in questo caso si utilizza la variabile `result` evitare di aggiornare il valore in cache ad ogni iterazione del ciclo interno.

3.3 Calcolo parallelo - GPU

Per il calcolo parallelo su GPU sono stati realizzati più nuclei di calcolo: per quanto riguarda CSR sono stati realizzati 3 nuclei di calcolo, mentre per quanto riguarda ELLPACK 1.

3.3.1 Calcolo parallelo - GPU - CSR

Per il calcolo parallelo su GPU usando il formato CSR, si sono implementati 3 nuclei di calcolo:

1. **CSR-Scalar**: un'implementazione del spmv basato su CSR che assegna ogni riga della matrice sparsa ad un thread.
2. **CSR-Vector**: un'implementazione del spmv basato su CSR che assegna ogni riga della matrice sparsa ad un warp.
3. **CSR-Adaptive**: un'implementazione del spmv basato su CSR che stabilisce a runtime se eseguire CSR-Vector o un altro algoritmo noto come CSR-Stream, il quale assegna più righe ad un singolo blocco.

la presentazione e lo pseudocodice dell'algoritmo CSR-Adaptive è stato preso da Efficient Sparse Matrix-Vector Multiplication[2].

3.3.1.1 Calcolo parallelo - GPU - CSR - CSR-Scalar

Iniziamo con la descrizione di CSR-Scalar:

```
const int row = blockIdx.x * blockDim.x + threadIdx.x;
//row corrisponde a tid
if(row < M){
    int row_start = IRP[row];
    int row_end = IRP[row+1];

    double sum = 0;
    for(int i = row_start; i < row_end; i++)
    {
        sum += AS[i] * val_vec[JA[i]];
    }
    val_res[row] = sum;
}
```

CSR-Scalar assegna ogni riga della matrice ad un thread. Questo approccio, che è quello usato nel nucleo di calcolo per CPU, presenta due problemi principali quando applicato sulla GPU:

- un numero variante di NZ per riga comporta uno sbilanciamento del carico tra i vari threads con conseguente sottoutilizzazione delle risorse hardware e peggioramento delle performance.

- l'accesso agli elementi della matrice in memoria non è coalizzato.

di questi due il problema dell'accesso alla memoria non coalizzato si ha per quasiasi tipo di matrice sparsa, mentre lo sbilanciamento del carico tra i vari thread è mitigato se la matrice con cui stiamo lavorando è una matrice diagonale a blocchi, infatti in questo caso threads consecutivi lavoreranno su righe consecutive della matrice che, nella maggior parte dei casi, apparterranno allo stesso blocco e avranno quindi lo stesso numero di non-zeri; tuttavia, questo caso particolare non rende CSR-Scalar un algoritmo buono in generale.

Per l'esecuzione di CSR-Scalar si sono usati blocchi 1D posti in una griglia 1D. In particolare si sono usati blocchi di 256 threads. La dimensione della griglia è stata calcolata appropriatamente per coprire tutte le righe della matrice.

```
const dim3 block_size = dim3(BLOCK_SIZE_CSR_SCALAR);
const dim3 grid_size = dim3((M + BLOCK_SIZE_CSR_SCALAR - 1)/
    BLOCK_SIZE_CSR_SCALAR);
```

3.3.1.2 Calcolo parallelo - GPU - CSR - CSR-Vector

Proseguiamo con la discussione di CSR-Vector:

```
const int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
const int row = thread_id / 32; //uguale a warp_id
const int lane = thread_id % 32;

double sum = 0;
if(row < M)
{
    int row_start = IRP[row];
    int row_end = IRP[row+1];

    for(int i = row_start + lane; i < row_end; i+= 32)
    {
        sum += AS[i] * val_vec[JA[i]];
    }



    sum = warp_reduce(sum);

    if(lane == 0)
        val_res[row] = sum;
}
```

CSR-Vector assegna un warp ad ogni riga della matrice sparsa; dunque, detto `lane` l'indice di un thread nel warp, ciascun thread calcola un risultato parziale relativo ai NZ nella riga il cui indice `i` soddisfa `i%32==lane`, al termine di questa operazione, per la quale non è necessaria sincronizzazione esplicita dal momento che i threads in un warp sono sincronizzati, si effettua una riduzione per sommare insieme i risultati parziali di ciascun thread. La riduzione è effettuata come segue:

```
__device__ double warp_reduce(double value){
  for(int offset = 16; offset > 0; offset /=2)
    value += __shfl_down_sync(0xffffffff, value, offset);

  return value;
}
```

l'utilizzo della primitiva a livello di warp `__shfl_down_sync` permette di effettuare lo scambio dei dati tra registri e ciò risulta più efficiente di una riduzione effettuata in shared memory[3].

CSR-Vector effettua accessi alla memoria in maniera coalizzata, dunque ci si aspetta che sia più veloce di CSR-Scalar; inoltre la divergenza all'interno di un warp è limitata all'ultima iterazione, nel caso in cui il numero di NZ all'interno della riga non fosse un multiplo della dimensione del warp.

CSR-Vector presenta, però, un problema nel caso in cui il numero di NZ per riga fosse basso, infatti, in questo caso solo un sottoinsieme di threads di ogni warp effettua lavoro utile e quindi le performance si riducono notevolmente.

Per l'esecuzione di CSR-Vector si sono usati blocchi 1D posti in una griglia 1D. In particolare si sono usati blocchi di 256 threads. La dimensione della griglia è stata calcolata appropriatamente per coprire tutte le righe della matrice.

```
const dim3 block_size = dim3 (BLOCK_SIZE_CSR_VECTOR);
const dim3 grid_size = dim3 ((32*M + BLOCK_SIZE_CSR_VECTOR - 1)/
    BLOCK_SIZE_CSR_VECTOR);
```

3.3.1.3 Calcolo parallelo - GPU - CSR - CSR-Adaptive

Terminiamo con la descrizione di CSR-Adaptive il quale è un algoritmo che a runtime stabilisce se usare CSR-Vector oppure CSR-Stream.

L'idea dietro CSR-Adaptive è la seguente: se una riga della matrice è "lunga", su questa viene eseguito CSR-Vector, cioè si assegna ad un warp; se incontriamo più righe della matrice "corte", allora queste righe vengono raggruppate insieme in un **blocco di righe** e si assegna questo blocco di righe ad un blocco di threads, questo secondo caso prende il nome di CSR-Stream. Il caso CSR-Stream prevede accune accortezze: anzitutto i risultati parziali calcolati da ciascun thread non possono essere ridotti tramite una primitiva a livello di warp, come in CSR-Vector, perché in questo caso i risultati parziali appartengono a righe diverse della matrice e perché i risultati parziali di una stessa riga potrebbero essere calcolati da threads che appartengono a warp diversi dello stesso blocco. Per fare ciò si è memorizzato il risultato parziale in shared_memory e poi si è proseguito con la riduzione per ciascuna riga che compone il blocco di righe (dunque il numero di riduzioni da effettuare, così come il numero di elementi da ridurre, non è fissato, ma varia tra i diversi blocchi di righe).

Il nome CSR-Stream deriva dal fatto viene effettuato lo "stream" dei risultati parziali in shared memory.

Andiamo a vedere nel dettaglio l'implementazione. Iniziamo come determinare i **blocchi di righe**, l'algoritmo presentato è l'implementazione dell'Algoritmo2 dell'articolo Efficient Sparse Matrix-Vector Multiplication presente nei riferimenti:

```
{
  if(!onlyCount){
    row_blocks[0] = 0;
  }
  int last_i = 0; //ultima riga inserita in un blocco di righe
  int current_block = 1;
  int nz = 0; //nz che sto cercando di inserire nel blocco di righe
  for(int i = 1; i <= M; i++)
  {
    nz += IRP[i] - IRP[i - 1];

    if(nz == NZ_PER_BLOCK)
    {
      //quesa riga riempie esattamente fino a NZ_PER_BLOCK
      last_i = i;

      if (!onlyCount){
        row_blocks[current_block] = i;
      }
      current_block++;
      nz = 0;
    }
    else if (nz > NZ_PER_BLOCK)
    {
      //la riga considerata non entra nel blocco corrente di righe
      if (i - last_i > 1)
      {
        //se il blocco di righe che sto considerando vorrebbe
    contenere piu' di una riga chiudo il blocco alla riga
    precedente e riconsidero la riga
        if (!onlyCount){
          row_blocks[current_block] = i - 1;
        }
        current_block++;
        i--;
      }
      else
      {
        //il blocco di righe che sto considerando vorrebbe
    contenere un'unica riga -> non posso far altro che avere un
    blocco di righe che contiene piu' di 32 elementi
        // in csr_adaptive verra' gestito da csr_vector
        if (!onlyCount){
          row_blocks[current_block] = i;
        }
        current_block++;
      }

      last_i = i;
      nz = 0;
    }
    //se ci sono NZ_PER_BLOCK righe vuote, devo creare un blocco di
     righe senza tener conto del numero di nz
    else if (i - last_i > NZ_PER_BLOCK)
    {
      last_i = i;
      if (!onlyCount){
```

```
53          row_blocks[current_block] = i;
54        }
55        current_block++;
56        nz = 0;
57      }
58    }
59
60    if (!onlyCount)
61      row_blocks[current_block] = M;
62
63    return current_block;
64  }
```

iniziamo col dire che l'algoritmo deve essere eseguito 2 volte: la prima volta con `onlyCount = true` serve per conoscere quanti blocchi di righe si verranno a creare, la seconda volta con `onlyCount = false` per memorizzare a quale riga inizia ciascun blocco. Il primo blocco inizia, evidentemente, alla riga zero, poi si scorrono le righe e si contano i non zero per riga; arrivati alla riga i-esima abbiamo 3 possibilità:

1. se il numero di non-zeri accumulati fino alla riga i-esima è esattamente pari al massimo numero di non-zeri per blocco di righe -> chiudiamo il blocco di righe e passiamo a considerare la riga successiva.

2. se il numero di non-zeri accumulati fino alla riga i-esima supera il numero massimo di non-zeri per blocco di righe si hanno due casi:

   (a) se il numero di non-zeri accumulati fino alla riga i-esima comprende i non-zeri di più righe -> chiudiamo il blocco di righe alla riga precedente e riconsideriamo la riga i-esima.

   (b) se il numero di non-zeri accumulati fino alla riga i-esima comprende solo i non-zeri della riga i-esima -> creo un blocco di righe con solo la riga i-esima.

3. se il numero di righe considerate supera il massimo numero di non-zeri per blocco di righe allora chiudo il blocco di righe.

si noti che il caso 3 non è presente nello pseudocodice dell'Algoritmo2 nell'articolo Efficient Sparse Matrix-Vector Multiplication, tuttavia è necessario per gestire il caso in cui la matrice presenta righe nulle, se questo caso viene omesso, ciò che succede è che si crea un blocco di righe contenente righe non consecutive e dunque si ha un errore di calcolo nella fase di riduzione. Il risultato dell'algoritmo è un' array che ha in posizione i-esima l'indice della riga a cui inizia l'i-esimo blocco di righe.

Per quanto riguarda lo schema generale di CSR-Adaptive si ha che se un blocco di righe contiene 2 o più righe si esegue CSR-Stream, altrimenti si esegue CSR-Vector:

```
__global__ void csr_adaptive(const int M,
                                const int *JA,
                                const int *IRP,
                                const double *AS,
                                const double *vec_val,
                                double *res_val,
                                const int *row_blocks)
{
    const int block_row_start = row_blocks[blockIdx.x];
    const int block_row_end = row_blocks[blockIdx.x + 1];

    if (block_row_end - block_row_start > 1)
    {
        // CSR-Stream
    }
    else
    {
        // CSR-Vector
    }
}
```

Vediamo, infine, l'implementazione di CSR-Stream, il cui pseudocodice è presentato nell'Algoritmo3 dell'articolo Efficient Sparse Matrix-Vector Multiplication:

```
const int block_row_start = row_blocks[blockIdx.x];
const int block_row_end = row_blocks[blockIdx.x + 1];
...
// CSR-Stream
const int nz = IRP[block_row_end] - IRP[block_row_start];

__shared__ double sharedmem[NZ_PER_BLOCK];
const int i = threadIdx.x;
const int block_data_begin = IRP[block_row_start];
const int thread_data_begin = block_data_begin + i;

if (i < nz)
    sharedmem[i] = AS[thread_data_begin] * vec_val[JA[
    thread_data_begin]];
__syncthreads ();

const int threads_for_reduction = prev_power_of_two(blockDim.x / (
    block_row_end - block_row_start));

if (threads_for_reduction > 1)
    {
    //Riduzione dei nz ad opera di piu' threads
    const int thread_in_tfr = i % threads_for_reduction; //indice
    del thread in threads_for_reduction
    const int local_row = block_row_start + i /
    threads_for_reduction; //indice della riga in IRP che il thread
     i deve contribuire a ridurre

    double sum = 0.0;

    //local_row potrebbe non appartenere a questo blocco
    if (local_row < block_row_end)
        {
        //indice del primo elemento della riga local_row nel blocco
     di righe
        const int local_first_element = IRP[local_row] - IRP[
    block_row_start];
        //indice dell'ultimo elemento della riga local_row nel
    blocco di righe + 1
        const int local_last_element = IRP[local_row + 1] - IRP[
    block_row_start];

    //ciascuno dei threads_for_reduction thread somma in sum i
    risultati parziali della riga in sharedmem a salti di
    threads_for_reduction
        for (int local_element = local_first_element +
    thread_in_tfr; local_element < local_last_element;
    local_element += threads_for_reduction)
            sum += sharedmem[local_element];
        }
    __syncthreads ();
    //dunque ciascun thread salva il valore di sum in sharedmem,
    cosi' ci sono threads_for_reduction (potenza di 2) valori da
    ridurre.
    sharedmem[i] = sum;
```

```
41
42      //ogni riga ha threads_for_reduction elementi in sharedmem da
        ridurre
43      for (int j = threads_for_reduction / 2; j > 0; j /= 2)
44      {
45          //riduzione
46          __syncthreads ();
47
48          //non tutti i threads devono scrivere
49          //la prima proposizione prende solo la prima meta' dei
        threads
50          //la seconda proposizione evita di uscire dalla sharedmem
51          const bool use_result = thread_in_tfr < j && i + j <
        NZ_PER_BLOCK;
52
53          if (use_result)
54              sum += sharedmem[i + j];
55          __syncthreads ();
56
57          if (use_result)
58            sharedmem[i] = sum;
59      }
60      //il thread 0 salva il valore trovato
61      if(thread_in_tfr == 0 && local_row < block_row_end)
62          res_val[local_row] = sum;
63      }
64  else
65      {
66      //riduzione di tutti i nz ad opera di un singolo threads
67          int local_row = block_row_start + i;
68          while (local_row < block_row_end)
69          {
70              double sum = 0.0;
71
72              for ( int j = IRP[local_row] - block_data_begin; j <
        IRP[local_row + 1] - block_data_begin; j++)
73                  sum += sharedmem[j];
74
75              res_val[local_row] = sum;
76              local_row += NZ_PER_BLOCK;
77          }
78      }
```

percorriamo passo per passo l'algoritmo:

- anzitutto si effettua lo stream in shared memory dei risultati parziali, si noti che si effettuano accessi coalizzati agli elementi della matrice in memoria globale.
- poi si prosegue definendo il numero di thread per la riduzione in base alla dimensione del blocco e al numero di righe nel blocco di righe.
- si effettua la riduzione facendo attenzione al fatto che si stanno effettuando più riduzioni, una per ogni riga, di indice `local_row`, presente nel blocco di righe.
- si salvano i risultati.

Si omette di ripetere la parte di CSR-Vector per la quale l'unica accortezza da avere è il fatto che adesso un blocco di righe composto da una sola riga non viene assegnato ad un warp, ma ad blocco di threads, dunque occorre scrivere il codice in modo che solo il warp 0 lavori.

Per l'esecuzione di CSR-Adaptive si sono usati blocchi 1D posti in una griglia 1D. In particolare si sono usati blocchi di dimensione pari al massimo numero di non-zeri per blocco di righe. La dimensione della griglia è stata settata pari al numero di blocchi di righe individuati e dipende quindi dal massimo numero di non-zeri per blocco di righe.

```
const dim3 block_size = dim3 (NZ_PER_BLOCK);
const dim3 grid_size = dim3 (block_count);
```

3.3.2 Calcolo parallelo - GPU - ELLPACK

Per il calcolo parallelo su GPU usando il formato ELLPACK si è realizzato il seguente nucleo di calcolo:

```
__global__ void ellpack_kernel(int M,
                  int max_nz,
                  const int *JA,
                  const double *AS,
                  const double *val_vec,
                  double *val_res)
int row = blockIdx.x * blockDim.x + threadIdx.x; //uguale a tid

if(row < M){
    double sum = 0;
    for(int i = 0; i < max_nz; i++){
        const int index = M * i + row; //tiene conto della
    trasposizione della matrice
        sum += AS[index] * val_vec[JA[index]];
    }
    val_res[row] = sum;
}
```

tale nucleo di calcolo vede l'assegnazione di ogni riga della matrice a ciascun thread. Per fare in modo che gli accessi ai dati siano coalizzati, prima di effettuare i calcoli si è effettuata la trasposizione della matrice, cioè degli array 2D JA e AS che definiscono la matrice nel formato ELLPACK, in modo da avere gli elementi di ciascuna colonna adiacenti in memoria.

Per l'esecuzione del nucleo di calcolo basato sul formato ELLPACK, si è usata una griglia 1D di blocchi 1D. In particolare, si è scelto per i blocchi una dimensione di 256 threads, dunque la dimensione della griglia è stata calcolata in modo da coprire tutte le righe della matrice.

```
const dim3 block_size = dim3 (BLOCKSIZEELLPACK);
const dim3 grid_size = dim3 ((M + BLOCKSIZEELLPACK - 1)/
    BLOCKSIZEELLPACK);
```

4 Testing

Per verificare la correttezza dei risultati ottenuti si confronta il vettore risultante dal prodotto parallelo con quello risultante dal prodotto seriale: si valuta la differenza rispetto al valore assoluto di ciascuna componente e si verifica che questo valore sia più piccolo di un milionesimo.

5 Analisi delle prestazioni

Le prestazioni sono state misurate cronometrando solo il tempo necessario per il calcolo del prodotto matrice vettore. Per ogni matrice il calcolo è stato ripetuto 32 volte, per ognuna di queste volte si è cronometrato il tempo necessario per il completamento, poi ne è stata fatta la media.

Le metriche raccolte sono:

- il tempo (medio) di calcolo;
- i flops;
- lo speedup rispetto all'implementazione seriale.

5.1 Analisi delle prestazioni sulla CPU

Per quanto riguarda l'analisi delle performance sulla CPU, si è deciso di variare il numero di threads da un minimo di 1 ad un massimo pari al doppio del numero di cores disponibili sulla piattaforma di calcolo utilizzata. Dal momento che si sono effettuati i test su un processore *Intel Xeon Silver 4210 da 2.20GHz*, il quale possiede 20 cores, si sono effettuati i test fino 40 threads.

La figura 1 e la figura 2 mostrano i Giga Flops ottenuti dal calcolo parallelo in CPU avendo usato rispettivamente il formato CSR e il formato ELLPACK, al variare del numero di threads. Dalle due figure precedenti possiamo notare per CSR un'aumento regolare dei GF fino a quando arriviamo al massimo numero di core fisici (20), poi le prestazioni peggiorano per quasi tutte le matrici e continuano con un andamento estremamente irregolare. Per ELLPACK, la situazione è molto simile, tranne per il fatto che in questo caso si registra, per quasi tutte le matrici, un peggioramento delle prestazioni quando si utilizzano tra gli 11 e i 14 threads.

Si noti nella tabella 2 l'assenza di risultati per le matrici *webbase-1M* e *dc1*. Questo è a causa del fatto che la quantità di memoria necessaria per memorizzare le array JA e AS per queste matrici supera la RAM libera. Questo controllo è abbastanza approssimativo, tuttavia è sufficiente, per queste matrici, a garantire il corretto funzionamento del programma; infatti, per queste matrici succede che il numero di righe e il numero massimo di non-zeri su una riga sono tali da causare un buffer overflow nelle variabili nel momento in cui si calcolano gli indici (ad esempio in `row = i * maxnz;` nel nucleo di calcolo parallelo su CPU), con conseguente comportamento imprevedibile del programma - nel migliore dei casi segmentation fault.

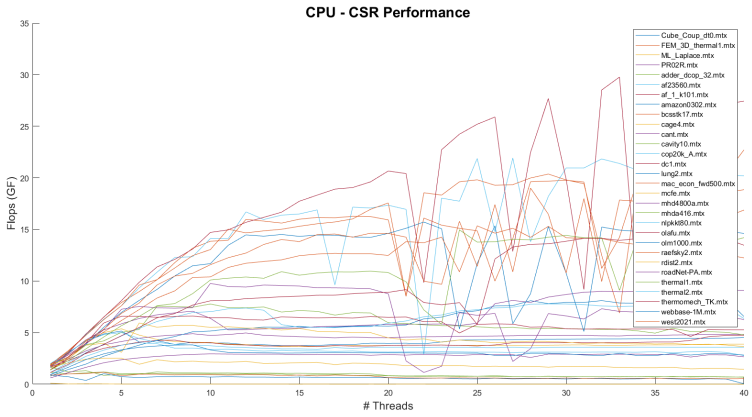

Figure 1: CPU CSR Perfomance

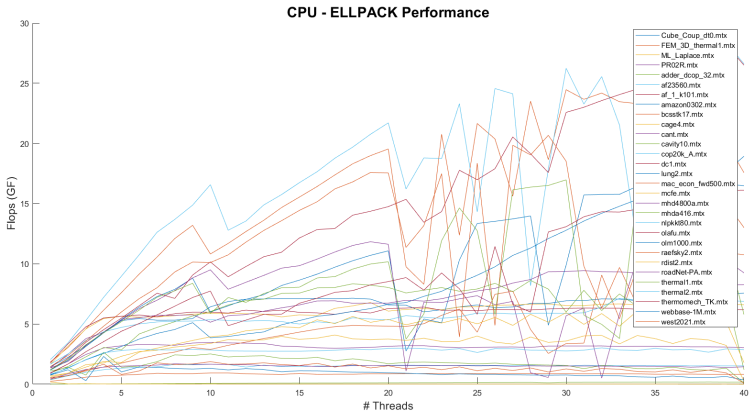

Figure 2: CPU ELLPACK Perfomance

Matrice	**Tempo (ms)**	**Flops (GF)**	**SpeedUp (%)**
cage4.mtx	0,012219	0,00802	0
mhda416.mtx	0,020625	0,830255	48,4848
mcfe.mtx	0,025156	1,93844	107,329
olm1000.mtx	0,012875	0,620738	46,6019
adder_dcop_32.mtx	0,027594	0,815112	76,1042
west2021.mtx	0,022938	0,641134	65,3951
cavity10.mtx	0,025625	5,96035	331,707
rdist2.mtx	0,025344	4,49294	236,745
cant.mtx	1,38525	5,78579	343,909
olafu.mtx	0,098281	20,6582	1232,18
Cube_Coup_dt0.mtx	64,1682	3,96477	235,776
ML_Laplace.mtx	14,7862	3,74539	222,965
bcsstk17.mtx	0,048813	17,5631	942,382
mac_econ_fwd500.mtx	0,204063	12,4804	914,426
mhd4800a.mtx	0,023344	8,76055	428,38
cop20k_A.mtx	0,888781	5,90546	573,144
raefsky2.mtx	0,040281	14,6111	883,786
af23560.mtx	0,055844	17,3432	888,193
lung2.mtx	0,067406	14,6148	857,487
PR02R.mtx	3,63547	4,50293	262,222
FEM_3D_thermal1.mtx	0,054094	15,9257	787,522
thermal1.mtx	0,106188	10,8197	970,924
thermal2.mtx	5,51091	3,11394	362,39
thermomech_TK.mtx	0,159938	8,89795	1016,02
nlpkkt80.mtx	16,0638	3,57384	201,416
webbase-1M.mtx	2,04231	3,0412	352,493
dc1.mtx	0,236687	6,47602	408,978
amazon0302.mtx	0,439	5,62586	903,872
af_1_k101.mtx	9,391	3,73776	216,495
roadNet-PA.mtx	2,15013	2,86848	476,391

Table 1: Prestazioni CPU - Formato CSR - 20 Threads

Matrice	**Tempo (ms)**	**Flops (GF)**	**SpeedUp (%)**
cage4.mtx	0,010781	0,00909	0
mhda416.mtx	0,010063	1,70176	149,068
mcfe.mtx	0,013	3,75108	576,923
olm1000.mtx	0,008406	0,950721	95,1673
adder_dcop_32.mtx	0,197281	0,11401	1685,41
west2021.mtx	0,009625	1,5279	249,351
cavity10.mtx	0,019	8,03863	978,947
rdist2.mtx	0,021156	5,38224	1039,88
cant.mtx	1,18972	6,73669	509,112
olafu.mtx	0,137656	14,7491	1245,13
Cube_Coup_dt0.mtx	38,644	6,58349	464,988
ML_Laplace.mtx	9,12528	6,06885	367,901
bcsstk17.mtx	0,177937	4,81798	1175,13
mac_econ_fwd500.mtx	2,92928	0,869421	373,368
mhd4800a.mtx	0,017594	11,6237	966,252
cop20k_A.mtx	3,33041	1,57598	464,808
raefsky2.mtx	0,033531	17,5523	1264,49
af23560.mtx	0,044625	21,7034	1245,94
lung2.mtx	0,088938	11,0766	961,349
PR02R.mtx	5,20009	3,14807	345,648
FEM_3D_thermal1.mtx	0,044094	19,5375	1149,82
thermal1.mtx	0,112937	10,173	981,959
thermal2.mtx	5,78116	2,96837	393,606
thermomech_TK.mtx	0,167	8,52165	1192,22
nlpkkt80.mtx	10,7373	5,3467	313,951
webbase-1M.mtx			
dc1.mtx			
amazon0302.mtx	0,371812	6,64247	1164,83
af_1_k101.mtx	5,57778	6,29307	368,426
roadNet-PA.mtx	3,98444	1,54792	358,394

Table 2: Prestazioni CPU - Formato ELLPACK - 20 Threads

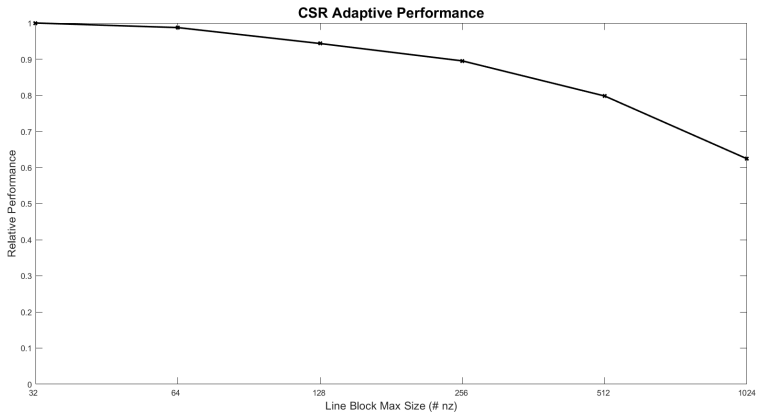

Figure 3: CPU Adaptive Perfomance al variare di NZ_PER_BLOCK

5.2 Analisi delle prestazioni sulla GPU

5.2.1 CSR-Adaptive Tuning

Prima di discutere le prestazioni sulla GPU, è necessario fare un piccolo tuning all'algoritmo CSR-Adaptive, cioè quello della scelta del massimo numero di non-zeri per blocco di righe. Seguendo lo stesso criterio adottato nell'articolo in cui è presentato l'algoritmo, si valuta la media armonica delle prestazioni mostrate dall'algoritmo sulle matrici di test, al variare del massimo numero di non-zeri per blocco di righe. Il risultato, in figura 3 mostra che la scelta convienente è al più 32 non-zeri per blocco di righe.

5.2.2 Confronto algoritmi su GPU

Le figure 4, 5 e 6 confrontano in un barplot le prestazioni dei vari algoritmi su GPU. Possiamo vedere che non c'è un algoritmo che risulta migliore sempre, ma che di caso in caso un algoritmo può risultare il migliore.

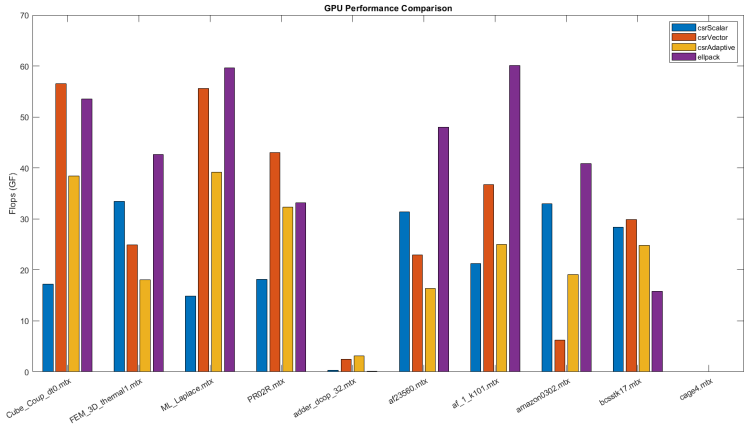

Figure 4: GPU Comparison 1

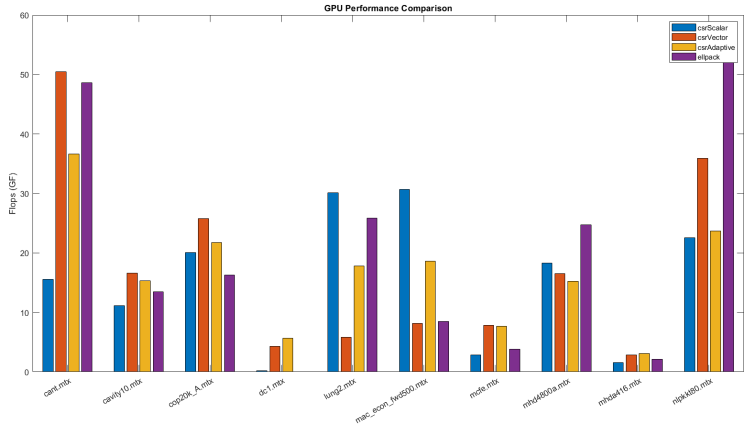

Figure 5: GPU Comparison 2

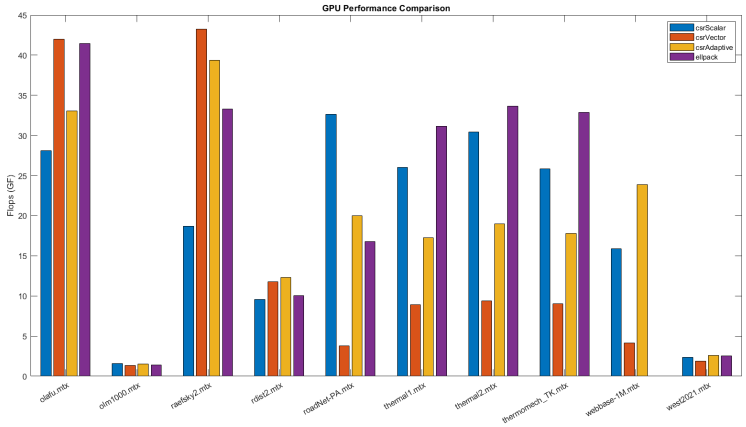

Figure 6: GPU Comparison 3

Nella tabella 3 vengono confrontate le prestazioni dei vari algoritmi (in termini di giga flops) per le varie matrici. In giallo è evidenziato l'algoritmo migliore, in verde il peggiore. Nella stessa tabella sono riportati anche il numero medio di NZ per riga e la deviazione standard dello stesso valore. Dalle matrici testate possiamo stabilire le seguenti correlazioni:

- se il numero medio di NZ per riga e alto e la deviazione standard è bassa, allora il formato ELLPACK risulta vincente;
- se il numero medio di NZ per riga è alto e la deviazione standard è alta, allora CSR-Vector risulta essere il migliore. In questo caso ELLPACK "soffre" il padding dovuto all'alta varianza del numero di NZ per riga.
- se il numero medio di NZ per riga è alto e la deviazione standard è alta, allora CSR-Scalar risulta essere il peggiore. Se la dev. std. è alta allora alcuni threads saranno molto più impegnati di altri.
- se il numero medio di NZ per riga è basso CSR-Vector è il peggiore, infatti molti thread nel warp saranno inattivi.

matrix	M	nz	nz_mean	nz_std	csr-scalar	csr-vector	csr-adaptive	ellpack
cage4	9	49	5,44444	0,496904	0,018656	0,019452	0,018192	0,019754
mhda416	416	8562	20,5817	6,322128	1,58204	2,89257	3,11289	2,12114
mcfe	765	24382	31,8719	16,91127	2,88374	7,80474	7,70485	3,81565
olm1000	1000	3996	3,996	1,997994	1,56675	1,32012	1,52898	1,42384
adder_dcop_32	1813	11246	6,20298	30,77715	0,321908	2,45707	3,14661	0,147633
west2021	2021	7353	3,6383	2,389425	2,36812	1,88975	2,57548	2,5087
cavity10	2597	76367	29,4059	14,94717	11,133	16,6341	15,2902	13,4639
rdist2	3198	56934	17,803	13,75616	9,54948	11,7851	12,3247	10,0192
cant	62451	4007383	64,1684	14,05624	15,5308	50,4877	36,6309	48,5905
olafu	16146	1015156	62,8735	12,40939	28,0783	42,0111	33,0621	41,4832
Cube_Coup_dt0	2164760	127206144	58,7622	4,471622	17,1894	56,5386	38,3796	53,504
ML_Laplace	377002	27689972	73,4478	3,524812	14,8579	55,6141	39,1553	59,6766
bcsstk17	10974	428650	39,0605	15,41357	28,4241	29,8451	24,8579	15,8287
mac_econ_fwd500	206500	1273389	6,16653	4,435865	30,6652	8,12784	18,6178	8,45785
mhd4800a	4800	102252	21,3025	5,799509	18,2609	16,5496	15,2025	24,7614
cop20k_A	121192	2624331	21,6543	13,79268	20,0269	25,7627	21,7764	16,2621
raefsky2	3242	294276	90,7699	21,19257	18,6895	43,2123	39,3654	33,3136
af23560	23560	484256	20,5542	1,270752	31,3475	22,9712	16,4071	48,0151
lung2	109460	492564	4,49995	1,939095	30,0922	5,85702	17,8242	25,8218
PR02R	161070	8185136	50,8173	19,69827	18,1132	43,0578	32,3013	33,1998
FEM_3D_thermal1	17880	430740	24,0906	4,308782	33,4387	24,9531	18,0181	42,6602
thermal1	82654	574458	6,95015	0,876772	26,0319	8,91054	17,258	31,125
thermal2	1228045	8580313	6,98697	0,811404	30,4125	9,39343	19,0061	33,674
thermomech_TK	102158	711558	6,96527	0,71524	25,8129	9,04013	17,7794	32,8482
nlpkkt80	1062400	28704672	27,0187	3,735117	22,5702	35,8925	23,6748	57,3271
webbase-1M	1000005	3105536	3,10552	25,34522	15,8662	4,125	23,8968	
dc1	116835	766396	6,55964	361,4983	0,220853	4,33715	5,65589	
amazon0302	262111	1234877	4,71127	0,951538	32,952	6,24172	19,0613	40,8812
af_1_k101	503625	17550675	34,8487	1,256579	21,2346	36,7681	25,0009	60,1254
roadNet-PA	1090920	3083796	2,82678	1,025783	32,6155	3,80184	20,0077	16,7908

Table 3: Performance GPU - Comparison

5.2.3 Speedup su GPU

La tabella 4 contiene gli speedup registrati rispetto alla versione seriale del codice. Si ricordi che per i due formati si sono implementate due versioni seriali distinte.

matrix	speedup (%)			
	csr-scalar	**csr-vector**	**csr-adaptive**	**ellpack**
cage4	19,0367	0	0	0
mhda416	83,1486	152,027	163,607	161,031
mcfe	159,669	432,138	616,211	594,679
olm1000	78,416	66,072	76,5257	71,2631
adder_dcop_32	27,193	229,408	279,799	2093,85
west2021	161,031	115,652	175,131	341,18
cavity10	539,398	903,943	760,837	1560,3
rdist2	444,482	558,89	746,834	1742,19
cant	941,954	2931,08	2091,43	3755,19
olafu	1589,01	2400,26	1859,66	3346,75
Cube_Coup_dt0	997,228	3336,21	2224,33	3687,03
ML_Laplace	849,19	3200,17	2230,54	3526,72
bcsstk17	1455,52	1552,65	1267,11	3869,94
mac_econ_fwd500	2263,67	656,795	1342,9	3559,45
mhd4800a	892,937	752,61	698,781	2046,25
cop20k_A	1930,71	2427,71	2123,02	4721,23
raefsky2	1038,39	2422,91	2180,46	2422,6
af23560	1579,49	1110	808,064	2349,91
lung2	1680,06	320,459	971,612	2275,17
PR02R	1040,08	2488,21	1849,25	3696,94
FEM_3D_thermal1	1700,11	1181,79	842,885	2604,73
thermal1	2247,65	763,927	1430,01	3175,03
thermal2	3492,35	1064,33	2153,17	4587,81
thermomech_TK	2820,5	942,054	1905,23	4609,45
nlpkkt80	1269,58	2017,41	1309,9	3385,25
webbase-1M	1800,92	460,246	2662,82	
dc1	13,6737	268,244	335,045	
amazon0302	5794,53	1045,78	3152,76	7385,83
af_1_k101	1212,26	2101,04	1412,18	3642,33
roadNet-PA	5213,64	607,668	3197,29	3761,3

Table 4: Performance GPU - Speedup

5.2.4 Prestazioni sulla Nvidia Quadro RTX 5000

Le performance sono state ottenute misurando le prestazioni su una *Nvidia Quadro RTX 5000*, la quale ha una larghezza di banda di picco pari a:
$Bandwidth = Mem_{clockrate} * \frac{Mem_{buswidth}}{8} * 2$
$=> Bandwidth = 7001MHz * \frac{256bit}{8} * 2 = 448\frac{GB}{s}$

È necessario conoscere la larghezza di banda per il problema spmv perché questo problema è un problema *memory bound*; in particolare, dal momento che l'intensità aritmetica del problema è di $\frac{1}{6}$, ci aspettiamo che la velocità massima raggiungibile non sia maggiore di $\frac{1}{6}$ della larghezza di banda di picco, cioè quindi di circa 75GF. Si noti che il calcolo è stato effettuato partendo dalla larghezza di banda di picco e non dalla larghezza di banda sostenuta, dunque il valore ottenuto è un upperbound teorico della velocità massima ottenibile.

Le prestazioni ottenute dal codice prodotto oscillano, per le matrici di grandi dimensioni, tra i 25GF e i 60GF, quindi si ritiene ragionevole concludere che c'è ancora un po' di margine di miglioramento, pur essendo le prestazioni ottenute soddisfacenti.

6 Conclusioni e Criticità

Alcune conclusioni di carattere generale sono:

- effettuare il prodotto matrice sparsa vettore in parallelo esclusivamente su GPU è più conveniente rispetto a effettuarla su CPU.
- non esiste un algoritmo (o formato di memorizzazione), tra quelli visti, che risulta essere migliore in tutti i casi; sarebbe buona pratica, quindi, effettuare un' analisi preliminare della matrice prima di decidere quale strada intraprendere: usare (tra i formati visti) il formato CSR o ELLPACK e, tra gli algoritmi visti su CSR, quale.

Alcune criticità del lavoro svolto sono:

- Ci si è resi conto tardivamente del fatto che nel formato MarketMatrix gli elementi della matrice sono memorizzati in ordine di colonna e poi di riga, dunque non si è sfruttato questo ordinamento per la conversione della matrice nei formati CSR e ELLPACK. Questa mancanza rende la conversione nei formati CSR e ELLPACK più lenta di quanto sarebbe potuta essere.
- I formati considerati non affrontano il problema dell'accesso per indirizzamento indiretto agli elementi del vettore, quindi questa criticità dello spmv rimane irrosolta.
- I nuclei di calcolo prodotti, funzionano per qualsiasi matrice sparsa, tuttavia non tengono conto dell'eventuale struttura della matrice.
- Efficient Sparse Matrix-Vector Multiplication viene detto che il criterio migliore per scegliere se eseguire CSR-Stream piuttosto che CSR-Vector è "se il numero di righe nel blocco di righe è maggiore di 2 allora si usa CSR-Stream, altrimenti si usa CSR-Vector"; nell'implementazione realizzata, invece, si è scelto questo valore pari a 1 per poter riutlizzare la stessa implementazione di CSR-Vector testata da sola.

References

[1] H. Zhang, R. T. Mills, K. Rupp, B. F. Smith (2018) Vectorized Parallel Sparse Matrix-Vector Multiplication in PETSc Using AVX-512. https://caidao22.github.io/publication/zhang-2018/ICPP_KNL_final.pdf

[2] J. L. Greathouse, M. Daga (2014) Efficient Sparse Matrix-Vector Multiplication on GPUs using CSR Storage Format. http://www.computermachines.org/joe/pdfs/sc2014_csr-adaptive.pdf

[3] Using Cuda Warp-Level Primitives https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/