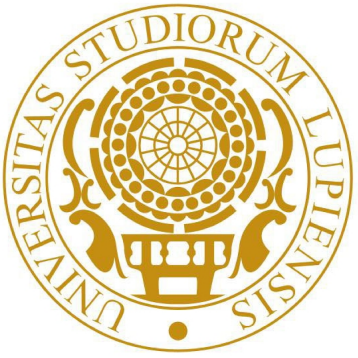
UNIVERSITÀ DEL SALENTO



Facoltà di Ingegneria

Corso di Laurea Magistrale in Computer Engineering

High Performance Computing

Moltiplicazione Matrice sparsa-Vettore

Professore: **Giovanni Aloisio**

Studenti: **Capoccia Leonardo,**
Basile Davide,
Corvaglia Salvatore

Anno Accademico: 2018/2019

Sommario

Introduzione

Molto spesso, nel campo dell'High Performance Computing, ci si imbatte nel problema di dover modificare un determinato programma allo scopo di migliorarne le performance. In genere è necessario suddividere il lavoro di ottimizzazione in tre sotto-problemi:

- Profiling del codice: attraverso questa fase è possibile determinare il numero di volte in cui ogni funzione viene chiamata e il tempo (con relativa percentuale) che si passa in ognuna di esse. Attraverso questo procedimento è possibile determinare la funzione o il kernel più oneroso in termini di risorse e di conseguenza, quello su cui ci si concentrerà ai fini dell'ottimizzazione.
- Realizzazione del modello Roofline: attraverso questo modello è possibile definire se il kernel o la funzione considerata sono di tipo performance-bounded e quindi è la velocità del processore a limitare le performance del programma, oppure se è di tipo memory-bounded, e di conseguenza è l'accesso in memoria che dovrebbe essere ottimizzato.
- Ottimizzazione del kernel: in funzione dell'analisi fatta del modello roofline si fa un ottimizzazione mirata.

La metodologia di ottimizzazione standard prevede un miglioramento in base ai risultati del modello roofline:

1. Nel caso dal roofline model il kernel risulti a sinistra del ridge point, quello che si fa è aumentare la data locality, ossia si aumentano i calcoli fatti sui dati in possesso in un determinato istante: questa tecnica è chiamata cache blocking. In questo modo si ottiene una diminuzione del traffico dati, il che mi permette di aumentare la arithmetic intensity che mi permette, a parità di performance, di spostare le coordinate del mio kernel verso destra. Questo procedimento viene reiterato fino a ottenere idealmente un kernel non più memory-bounded, ma performance-bounded.
2. Una volta che il kernel si trova a destra del ridge point nel grafico del roofline model, quello che è possibile fare è ottimizzare e modificare le operazioni fatte all'interno del kernel in modo da aumentarne le performance. Una tecnica classica che permette di fare ciò è l'utilizzo di istruzioni vettorizzate, il che permette di fare le stesse operazioni nel kernel in meno cicli di clock e ottenere di conseguenza FLOPS maggiori. Ovviamente questa è una trattazione un pò semplicistica perché l'utilizzo delle istruzioni vettorizzate modifica anche il traffico dati, il che spinge la posizione del kernel nel roofline model verso sinistra, fino ad arrivare, in situazioni critiche, in zone memory-bounded.
   Come è possibile evincere da questa trattazione, quindi, l'ottimizzazione delle performance è un processo iterativo che richiede diversi tentativi e che richiede la conoscenza approfondita della macchina su cui dovrà girare un determinato kernel per poter ottenere delle performance ideali.

Descrizione codice

Il codice, preso da GitHub[1], è l'implementazione di un algoritmo per la moltiplicazione di una matrice sparsa per un vettore. L'algoritmo prevede l'utilizzo di diverse funzioni, tra cui:

- `genSparseMatrix`: funzione per generare una matrice sparsa, usando il parametro `SPARSITY` come percentuale di valori nulli da mettere nella matrice da moltiplicare;
- `genDenseVector`: funzione usata per generare un vettore da usare nella moltiplicazione;
- `checkNNZ`: funzione usata per trovare il numero di valori diversi da zero all'interno della matrice generata da `genSparseMatrix`;
- `compression`: usata per comprimere i dati della matrice prima della moltiplicazione. Questo è utile in quanto la matrice è una matrice sparsa, avente cioè un numero di zeri più o meno alto. Questa funzione crea un vettore, a partire dalla matrice, avente una dimensione pari al numero di elementi nella matrice diversi da 0.
- `solutionSpMV`: usata per calcolare il vettore risultato della moltiplicazione. La complessità asintotica della funzione è $O(m)$ invece che $O(n^2)$, in cui n rappresenta il numero di colonne della matrice (assunto essere uguale al numero di righe) mentre m rappresenta il numero di elementi della matrice diversi da 0. La complessità asintotica è potenzialmente diminuita grazie alla funzione `compression`, che permette di trarre beneficio dalle informazioni di sparsità della matrice.

Funzione principale

La funzione `main` si occupa di coordinare tutte le operazione al fine di risolvere il problema della moltiplicazione matrice-vettore.
Le prime operazioni svolte in tale funzione sono il parsing degli argomenti passati da riga di comando. I primi due rappresentano la dimensione della matrice, mentre il terzo rappresenta una variabile booleana che abilita o disabilita il logging fatto dalle perf regions.

```
const int m = argc > 1 ? atoi(argv[1]) : MATRIXZISE;
const int n = argc > 2 ? atoi(argv[2]) : MATRIXZISE;
const bool perf_region = argc > 3 ? (strcmp(argv[3], "true") ? false : true) :
       false;
```

[1] "CSnakeEyes (Cristian Ayub Chavira) · GitHub." https://github.com/CSnakeEyes/SpMV.

```
if (perf_region) {
      perf_regions_init();
      puts("\n*** Matrix Vector multiply ***\n");
      perf_region_start(0, (PERF_FLAG_TIMINGS | PERF_FLAG_COUNTERS));
}
```

Nel caso le **perf_regions** siano abilitate, si abilitano i contatori chiamando le API delle **perf_regions** stesse ed inizia il conteggio.
A questo punto si passa a generare una matrice e un vettore con le dimensioni impostate da riga di comando:

```
float **A = genSparseMatrix(m, n);
float *x = genDenseVector(m);
```

Ora si hanno tutti gli elementi su cui operare per fare le moltiplicazioni. Si passa perciò al calcolo dei numeri di elementi differenti da 0 che ci sono all'interno della matrice e si salva tale valore all'interno della variabile **realNNZ**:

```
float realNNZ = checkNNZ(A,m,n);
double sparsePercentage = (realNNZ * 100) / (double)(m*n);
```

Quello che viene fatto ora è l'allocazione in memoria di alcuni vettori per poter immagazzinare tutti gli elementi diversi da 0 della matrice:

```
float *values = (float  *)malloc(nnz * sizeof(float));
int *colIndex = (int *)malloc(nnz * sizeof(int));
int *rowIndex = (int *)malloc(nnz * sizeof(int));
compression(A,values,colIndex,rowIndex,m,n);
```

Una volta presi tutti questi elementi della matrice, ciò che viene fatto è predisporre alcune variabili per poter ricavare il tempo di esecuzione della funzione principale:

```
struct timeval tp;
gettimeofday(&tp, NULL);
elaps = - (double)(tp.tv_sec + tp.tv_usec/1000000.0);
```

Una volta fatto ciò si può chiamare la funzione che effettua i calcoli veri e propri:

```
float *y = solutionSpMV(values, colIndex, rowIndex, x, m);
```

Eseguite le moltiplicazioni si calcola il tempo impiegato per effettuare tutte le moltiplicazioni:

```
gettimeofday(&tp, NULL);
elaps = elaps + ((double)(tp.tv_sec + tp.tv_usec/1000000.0));
```

Nel caso le **perf_regions** siano abilitate, si calcolano le performance ottenute dal nostro algoritmo, si bloccano i contatori e si ripulisce la memoria occupata da tali contatori:

```
if (perf_region) {
	MFLOPS = ((nnz)*3)/(elaps*1000000);
	printf("Iteration = %d, Matrix dim = %d\n", nnz, m*n);
	printf("Elapsed time: %lf\n", elaps);
	printf("MFLOPS: %lf\n", MFLOPS);

	perf_region_stop(0);

	perf_regions_finalize();
}
```

Il calcolo dei **MFLOPS**, così come mostrato nel codice, è dovuto al fatto che nella funzione **solutionSpMV** si effettuano 3 operazioni floating point, di cui 2 moltiplicazioni e una addizione per ogni iterazione del ciclo **for**.

Subroutine di calcolo

La subroutine di calcolo esegue il prodotto tra la matrice sparsa creata in precedenza e il vettore ottenuto con le tecniche sopra descritte. La funzione che richiama la subroutine è **solutionSpMV**, che prima alloca e poi inizializza un vettore di supporto per salvare i risultati del prodotto.

```
float *solution = (float *)malloc(m * sizeof(float));
initializeArray(solution, m);
```

Successivamente, all'interno di un ciclo **for**, vengono moltiplicati gli elementi della matrice e del vettore.

```
for (int i = 0; i < nnz; i++) {
	float val = (values[i] * x[colIndex[i]]);
	solution[rowIndex[i]] += (values[i] * x[colIndex[i]]);
}
```

Infine viene restituito il risultato della funzione.

Profiling

Nello studio del profiling dell'applicazione abbiamo posto la nostra attenzione su quale funzione rappresentasse il collo di bottiglia dell'intera applicazione. Per fare ciò, ci si è avvalsi del supporto della libreria `gprof`, che permette di ottenere il tempo di esecuzione totale del programma, le frazioni di tempo occupate da ogni funzione chiamata ed il numero di volte che una funzione viene chiamata all'interno di un'altra.
Nel nostro caso è risultato che la funzione `main` impiega:

- Il 52.94% del tempo totale nella funzione `genSparseMatrix` per la generazione della matrice, chiamandola solo 1 volta;
- Il 20.59% del tempo nella funzione `compression` chiamandola solo una volta;
- Il 14.71% del tempo per la computazione del prodotto attraverso la funzione `solutionSpMV`;
- L'11.76% è invece speso per il calcolo dei valori diversi da 0 attraverso la funzione `checkNNZ`.

Inoltre, come si evince dalla Fig. 1, il tempo totale di esecuzione è quello impiegato dalla funzione `main`, ma solo lo 0% del tempo è impiegato nella funzione stessa.
Da questo ragionamento si evince, quindi, che la funzione `solutionSpMV` di nostro interesse occupa una piccolissima percentuale di tempo della computazione totale e, di conseguenza, un miglioramento sarà poco percettibile. Diversamente la generazione della matrice sparsa attraverso `genSparseMatrix` non sarà migliorabile in quanto il tempo impiegato dalla funzione è solo il tempo necessario al sistema per allocare la memoria richiesta dalla matrice.

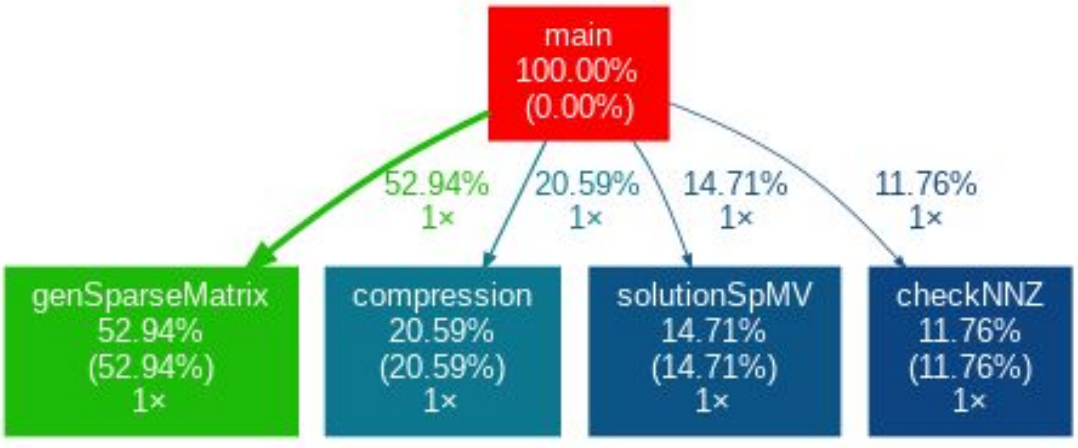


Fig. 1 - Profiling del codice.

Modello Roofline

Ad oggi, il calcolo di funzioni complesse richiede una grande quantità di risorse di calcolo e altrettante risorse di memoria. Con il passare degli anni, la diversificazione delle tecnologie relative ai processori ha fatto sì che essi superassero la velocità di accesso ai dati nelle memorie. Il modello del roofline ci serve per capire quale tra il processore o la memoria fungono da collo di bottiglia per il completamento di un dato kernel. Nel nostro caso, avendo a disposizione un Intel Xeon E5 Sandy Bridge 2670 con tre livelli di memoria cache, con 64KB per L1, 256KB per L2 e 20MB per la cache L3 condivisa da tutti i processori.

Per poter predire i risultati del modello abbiamo prima calcolato la dimensione della matrice in modo da essere grande abbastanza da riempire completamente le cache fino al livello 3. Per fare ciò abbiamo impostato la variabile `SPARSITY` a 1, così da forzare la matrice a essere una matrice densa, e forzare di conseguenza il vettore restituito dalla funzione `compression` ad avere un numero totale di elementi pari a quello della matrice. Fatto questo, è stato fatto un semplice calcolo, infatti una matrice di numeri reali avente n righe e n colonne occupa:

$$SIZE = n^2 \times 4\ bytes$$

Considerando di voler saturare completamente la cache L3 avente dimensione pari a 20MB, il calcolo per trovare il numero di colonne (e quindi di righe) è:

$$n = \sqrt{\frac{SIZE}{4\ bytes}} = \sqrt{\frac{20\ MB}{4\ bytes}} \simeq 2290$$

Forzare le dimensioni della matrice ad essere di dimensioni leggermente maggiori a 2290×2290 non ci dà i risultati sperati in quanto nei calcoli sono stati ignorati i processi di saturazione delle cache L1 ed L2. Quello che si è fatto, quindi, è stato utilizzare una dimensione maggiore per ovviare a questo problema, utilizzando prima un valore pari a 2500×2500. Usando questo valore si è potuto predire quindi che il kernel si trovasse a sinistra del ridge point, questo perché la matrice aveva dimensioni maggiori delle cache e quindi risultava essere in RAM. Questo di conseguenza genera una moltitudine di cache miss, portando così la memoria a essere il collo di bottiglia nei calcoli. Questo comportamento può essere visto in Fig. 2. Tale andamento è dato dal fatto che la memoria RAM non risulta sufficientemente performante e quindi abbastanza veloce da raggiungere la velocità di calcolo del processore, che si ritroverà a essere rallentato dall'attesa dei nuovi dati su cui operare.

Fatto questo è stato effettuato un altro esperimento forzando le dimensioni della matrice a essere tali da permettere alle matrici di entrare in cache. Così facendo si può predire che il kernel sarà a destra del ridge point nel modello roofline, infatti le memorie cache sono molto più veloci della RAM, il che permette di dare i dati al processore non appena ne ha bisogno, senza che la memoria sia il collo di bottiglia. Questo comportamento è possibile visualizzarlo in Fig. 3.

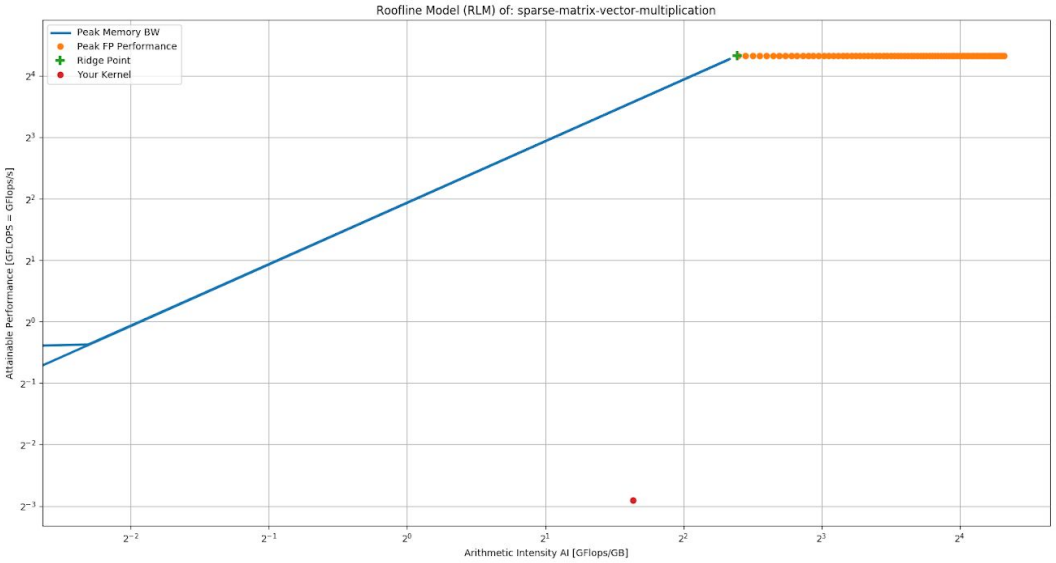


Fig. 2 - Roofline model della RAM con dimensioni della matrice pari a 2500×2500

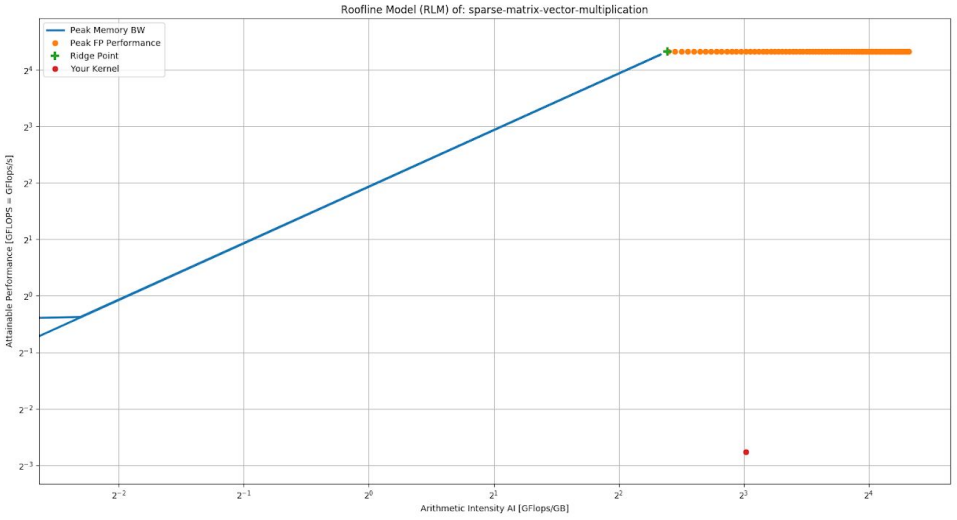


Fig. 3 - Roofline model con dimensioni della matrice pari a 2200×2200