Architetture e Programmazione dei Sistemi di Elaborazione Progetto

a.a. 2022/23

"Attention mechanism"in linguaggio assembly x86-32+SSE, x86-64+AVX e openMP

Cangeri Fabio 249495
Russo Matteo 242788
January 24, 2023

Abstract

Per lo svolgimento del progetto assegnatoci, chiamato Attention mechanism, abbiamo affrontato e suddiviso il lavoro nelle quattro parti richiesteci. Affronteremo, nei rispettivi paragrafi, i dettagli implementativi e l'analisi dell'efficienza delle versioni realizzate.

1 Sequenziale

Per la versione sequenziale abbiamo cercato, per quanto possibile, di costruire il miglior algoritmo, al fine di avere una solida base da ottimizzare ed avere un mezzo di paragone con le versioni "parallelizzate". Nell'algoritmo trattiamo una matrice j-esima alla volta dell'i-esimo tensore del dataset ds. Abbiamo cercato di innestare quante più operazioni possibili, tali da non alterare il risultato finale. L'algoritmo si compone di un totale di quattro funzioni:

- **void prodottoAllMatriciBias(MATRIX Q, MATRIX K, MATRIX V, MATRIX ds, MATRIX wq, MATRIX wk, MATRIX wv, VECTOR bq, VECTOR bk, VECTOR bv, int avanza, int n, int nn, int d)**
  In questa funzione, calcoliamo le nostre matrici Q, K e V come prodotto matriciale tra il dataset e le rispettive matrici dei pesi, sommando, nello stesso ciclo, gli opportuni vettori bias.
  Abbiamo utilizzato degli accorgimenti come il calcolo anticipato degli indici di accesso a posizioni delle matrici in modo tale da effettuare tale calcolo un'unica volta per le tre matrici generate; inoltre abbiamo utilizzato variabili locali piuttosto che accedere ad ogni iterazione alla porzione della matrice. Ciò ha velocizzato l'algoritmo sequenziale, riportanto un significativo miglioramento come si può vedere dal confronto delle tempistiche evidenziato in figura 1.

- **void prodottoMatriciInversa(MATRIX intermedio, MATRIX A, MATRIX B, float radice, int n, int nn)**
  Nella sopra citata funzione, eseguiamo la moltiplicazione tra una matrice ed una matrice inversa. Nello stesso "for" eseguiamo due calcoli:
  - Divisione di ogni elemento della matrice risultate per la radice del valore "d", noto da input;
  - Applicazione della funzione espressa nel punto 3 della traccia all'elemento della matrice appena calcolato.

  Così come descritto per la funzione precedente, anche in questo caso, l'utilizzo di variabili locali in cui salvare i risultati parziali del prodotto matriciale, senza dover accedere ad ogni iterazione alla porzione della matrice, ha riportato un sostanziale miglioramento del tempo di esecuzione.
- **void prodottoMatriciESalva(MATRIX output, MATRIX A, MATRIX B, int avanza, int n, int nn)**
  Questa funzione esegue il prodotto tra la matrice intermedia e la matrice V calcolata precedentemente.
  Si sottolineano con particolare attenzione le variabili *avanza_tensore, avanza_matrice, avanza_tensore_out, avanza_matrice_out*, le quali consentono di spostarci adeguatamente nelle porzioni di memoria rispettivamente del dataset ds e della matrice restituita in output.
- **void deallocaAllMatrici(MATRIX Q, MATRIX K, MATRIX V, MATRIX intermedio)**
  Infine quest'ultima funzione ha il compito di liberare lo spazio allocato dinamicamente al fine di non avere perdite di memoria.
  Tale controllo, mostrato in figura 2, è stato effettuato mediante il framework Valgrind, tool molto utile per verificare la corretta gestione delle memorie.

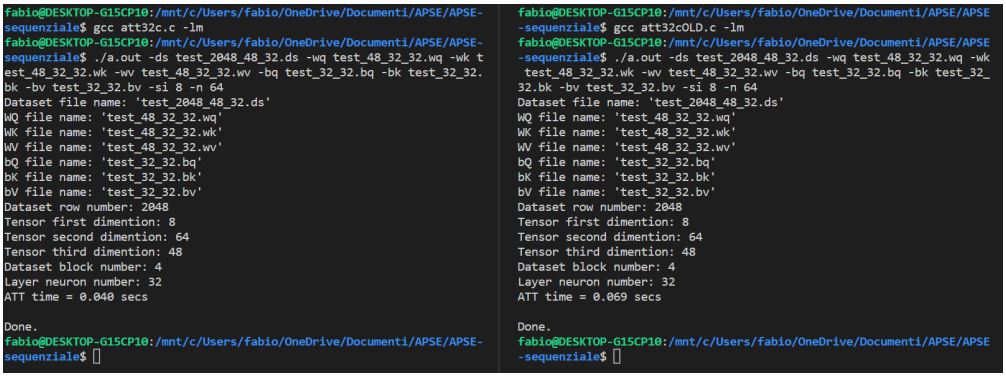

Figure 1: Confronto tempistiche con e senza introduzione di variabili locali, indici locali di accesso alle matrici e funzioni uniche

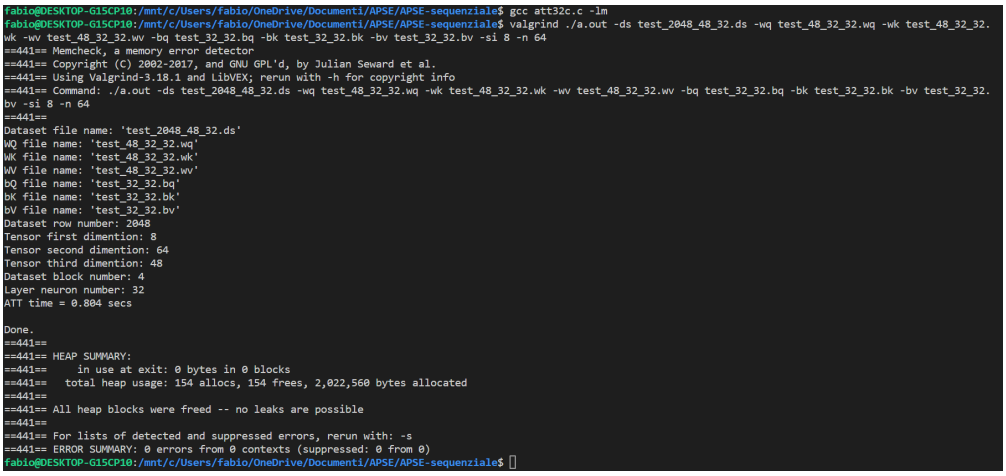

Figure 2: Analisi gestione della memoria

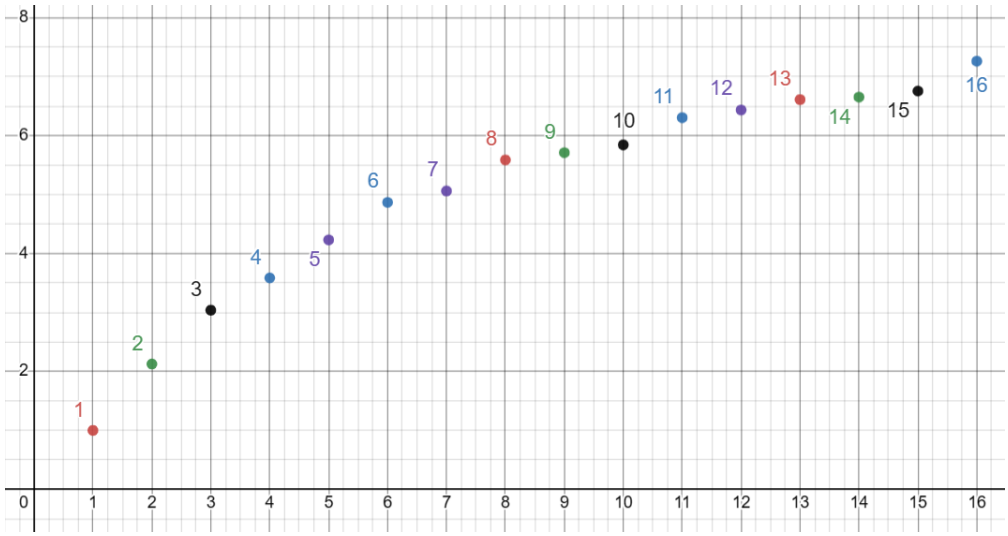

Figure 3: Speed Up OpenMP grafico

Numero Thread/ Threads	Tempo (s)	Speed UP	Efficiency
1	0,047368	/	/
2	0,025275	1,874104846686449	0,9370524233432245
3	0,016585	2,85607476635514	0,9520249221183801
4	0,013205	3,587126088602802	0,8967815221507005
5	0,011193	4,231930670955061	0,8463861341910122
6	0,009730	4,868242548818088	0,8113737581363481
7	0,009359	5,061224489795918	0,7230320699708455
8	0,008477	5,587825881797806	0,6984782352247257
9	0,008294	5,711116469737159	0,6345684966374622
10	0,008106	5,843572662225512	0,5843572662225512
11	0,007513	6,304805004658592	0,5731640913325993
12	0,007361	6,434995245211248	0,536249603767604
13	0,007165	6,61102581995813	0,5085404476890869
14	0,007119	6,653743503301025	0,4752673930929304
15	0,007011	6,756240193980887	0,4504160129320591
16	0,006522	7,262802821220485	0,4539251763262803

Figure 4: Tempistica dettagliata della versione OpenMP dell'algoritmo (Speed Up and Efficiency)

2 OpenMP

La versione OpenMP dell'algoritmo ha come obiettivo quello di ottimizzare il codice puntando al calcolo parallelo su matrici e tensori. A questo fine è stato distribuito il calcolo della j-esima matrice dell'i-esimo tensore su un differente processo. La direttiva utilizzata è:

```
#pragma omp parallel for collapse(2) num_threads(max_threads)
```

Analizzando le clausole utilizzate:

- **collapse:** consente di trattare dei for nested come un unico ciclo. In questo caso tale clausola è molto funzionale poiché non viene alterata la computazione dell'algoritmo, in quanto i due cicli for si riferiscono a porzioni differenti del dataset;
- **num_threads:** specifica il numero di thread utilizzati per il for in questione. Utile nel calcolo dello speedup mediante la specifica alla variabile max-threads del numero dei processi da utilizzare;

La direttiva consente di avere un miglioramento notevole, com'è possibile notare dall'analisi dello speed up in figura 3 e 4. Si noti che l'analisi è stata effetuata su una macchina con 8 CPU Cores e 16 Threads.
Per confrontare la versione OpenMP con la versione sequenziale, abbiamo utilizzato l'effettivo tempo trascorso mediante la funzione presente nella libreria <omp.h>

```
omp_get_wtime()
```

Figure 5: Tempo di esecuzione x86-32+SSE

---

3 x86-32+SSE

Per la versione x86-32+SSE sono state realizzate le seguenti funzioni:

- **extern void prodotto(MATRIX A, MATRIX B, int nn, int nn_x_i, int nn_x_j, type* a)**
  Tale codice Assembly viene richiamato all'interno della funzione sequenziale
  void prodottoMatriciInversa(MATRIX intermedio, MATRIX A, MATRIX B, float radice, int n, int nn)
  al fine di sostituire il ciclo for innestato più internamente, il quale contiene l'effettiva computazione. Sfruttando il principio dello *Streaming SIMD Extensions* per la versione a 32bit, ovvero lavorando con tipo di dato *float*, ciò consente di eseguire la computazione su quattro elementi per ogni iterazione in modo parallelo, migliorando così le prestazioni;
- **extern void prodottoMatrici(MATRIX output, MATRIX A, MATRIX B, int n, int nn)**
  Questa seconda funzione in linguaggio Assembly sostituisce totalmente la corrispettiva funzione C *void prodottoMatriciESalva(MATRIX output, MATRIX A, MATRIX B, int avanza, int n, int nn)* realizzando un prodotto tra matrici per intero. Basandoci su quanto appreso in una delle ultime esercitazioni svolte, in cui si trattava proprio tale argomento, abbiamo apportato delle modifiche nel codice C per favorire l'esecuzione della funzione; in particolare abbiamo invertito l'ordine delle matrici disponendole in column-major order.

La figura 5 mostra il miglioramento ottenuto in termini di tempo è visibile nella figura 5 se paragonato alla tempistica sequenziale della figura 1.

Probabilmente l'esecuzione dell'algoritmo su un dataset contenentente più dati consentirebbe di osservare un miglioramento ancora più marcato.

```
fabio@DESKTOP-G15CP10:/mnt/c/Users/fabio/OneDrive/Documenti/APS
E/APSE-sse$ nasm -f elf64 att64.nasm && gcc -m64 -msse -O0 -no-
pie sseutils64.o att64.o att64c.c -o att64c -lm && ./att64c -ds
 test_2048_48_64.ds -wq test_48_32_64.wq -wk test_48_32_64.wk -
wv test_48_32_64.wv -bq test_32_64.bq -bk test_32_64.bk -bv tes
t_32_64.bv -si 8 -n 64
Dataset file name: 'test_2048_48_64.ds'
WQ file name: 'test_48_32_64.wq'
WK file name: 'test_48_32_64.wk'
WV file name: 'test_48_32_64.wv'
bQ file name: 'test_32_64.bq'
bK file name: 'test_32_64.bk'
bV file name: 'test_32_64.bv'
Dataset row number: 2048
Tensor first dimention: 8
Tensor second dimention: 64
Tensor third dimention: 48
Dataset block number: 4
Layer neuron number: 32
ATT time = 0.039 secs

Done.
fabio@DESKTOP-G15CP10:/mnt/c/Users/fabio/OneDrive/Documenti/APS
E/APSE-sse$
```

Figure 6: Tempo di esecuzione x86-64+AVX

---

4 x86-64+AVX

Per la versione x86-64+AVX abbiamo realizzato la seguente funzione:

- **extern void prodottoAVX(MATRIX A, MATRIX B, int nn, int nn_x_i, int nn_x_j, type* a)**
  Si tratta sostanzialmente di una conversione della funzione *extern void prodotto(MATRIX A, MATRIX B, int nn, int nn_x_i, int nn_x_j, type* a)* realizzata nella versione x86-32+SSE, com'è possibile osservare dalla "signature", pressocché identica.
  La differenza sostanziale è che in questo caso il tipo di dato utilizzato è *double*, per cui i registri di cui si è fatto uso sono di 256 bit. Questo consente di mantenere le stesse ottimizzazioni viste per la versione a 32 bit, dove vengono utilizzati registri a 128 bit.

Al fine di mettere in luce le differenti tempistiche, per questa versione non è stata tradotta la funzione *extern void prodottoMatrici(MATRIX output, MATRIX A, MATRIX B, int n, int nn)* realizzata in x86-32+SSE; possiamo notare, infatti, che il tempo di esecuzione seppur leggermente migliorato è paragonabile alla versione sequenziale dell'algoritmo come mostrato in figura 6.