Laurea Magistrale in Ingegneria Informatica

Corso di
Architetture e Programmazione
di Sistemi di Elaborazione

Product Quantization for Nearest Neighbors Search

Simona Nisticò — Matricola: 204901
Fabio Mancuso — Matricola: 204930
Silvio Filice — Matricola: 207062

Anno Accademico 2018-19

Introduzione

L'obiettivo di questo progetto è stato quello di implementare l'algoritmo di PQNN (Product Quantization for Nearest Neighbors Search) con lo scopo di raggiungere i migliori risultati in termini di prestazioni. I linguaggi di programmazione utilizzati sono C ed Assembly, quest'ultimo sarà utilizzato per tradurre quante più funzioni C possibili in modo da migliorare le performance delle sezioni di codice più frequenti. A tale scopo sono state utilizzate le istruzioni Intel x86-32+SSE per l'architettura a 32 bit ed Intel x86-64+AVX per l'architettura a 64 bit.

Il progetto è stato sviluppato utilizzando il sistema operativo Ubuntu, il compilatore GCC e l'assemblatore NASM. L'intera attività è stata suddivisa nei seguenti passi:

1) Implementazione dell'algoritmo in C.

2) Prime ottimizzazioni dell'algoritmo in C.

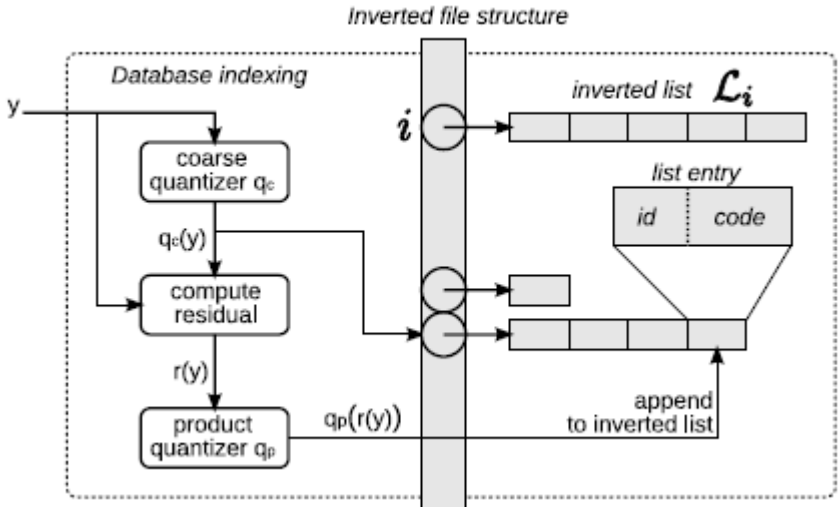
3) Individuazione delle funzioni più frequenti e ottimizzabili mediante le tecniche di loop unrolling e loop vectorizzation e loro implementazione in Assembly.

Presentazione Algoritmo

L'algoritmo sottoposto nel corso del progetto si articola in due fasi fondamentali: Indexing e Searching.

Nella fase di Indexing si utilizza, per la Ricerca Esaustiva e per quella Non-Esaustiva, la stessa funzione al fine di calcolare i centroidi: *calculate_centroids*. Questa scelta è stata attuata al fine di mantenere uniformità tra le due versioni di Indexing e per avere un'unica funzione da ottimizzare nei passi successivi. La possibilità di utilizzare questa funzione in entrambi i casi è data dai suoi parametri: tra di essi c'è il numero di gruppi (*m*) in cui deve essere suddiviso ogni punto per essere quantizzato ed esso corrisponde anche al numero di quantizzatori vettoriali che costituiscono il quantizzatore prodotto complessivo. Diventa evidente che se nell'indicizzazione non esaustiva viene passato un valore di m pari ad 1, la quantizzazione prodotto degenera in una quantizzazione vettoriale dell'intero dataset. Se per l'indicizzazione esaustiva la fase di indexing può considerarsi conclusa a seguito della costruzione del quantizzatore prodotto, nel caso dell'indicizzazione non esaustiva sono necessari altri passi. Dopo aver costruito il quantizzatore coarse (grossolano) sugli *nr* punti del dataset (altro punto che permette la riusabilità della funzione è il passaggio del numero di punti da considerare) si può procedere, dopo aver calcolato i centroidi coarse di appartenenza degli *n-nr* punti rimanenti del dataset, al calcolo dei residui del dataset. Una volta terminato il calcolo dei residui è possibile procedere alla costruzione del quantizzatore prodotto passando alla funzione *calculate_centroid* non più il dataset ma i residui ottenuti; la costruzione verrà effettuata sempre su *nr* punti.

Sempre relativamente alla ricerca non esaustiva, resta da realizzare la struttura che permette di risalire, a partire da un centroide coarse, alla lista dei punti che lo hanno come quantizzatore.



Lo schema adottato per la costruzione di tale struttura è il seguente:

1. Per ogni punto è noto il suo centroide coarse. Per questo motivo, si possono agevolmente costruire due strutture dati di supporto: una che contiene per ogni centroide coarse il numero di punti del dataset che essi quantizzano (*points_of_centroid*) ed un'altra (*celle_prima*) che contiene per ogni centroide la posizione della lista invertita a partire dalla quale si possono trovare i suoi punti. Questo permette di riservare virtualmente le porzioni della lista invertita ai diversi centroidi

2. Avviene quindi l'inserimento dei punti del dataset, che vengono scanditi in ordine. Qui è necessaria una seconda struttura ausiliaria *punti_caricati* che per ogni centroide coarse rappresenta il numero di punti del dataset ad esso appartenenti che sono stati effettivamente inseriti. In questo modo, dato il punto *i* che ha per centroide coarse *j*, *celle_prima[j]+punticaricati[j]*(m+1)* rappresenta il punto da cui iniziare ad inserire i valori che rappresentano il punto *i*.

Per quel che concerne la fase di ricerca, nel caso di ricerca esaustiva bisogna prima calcolare la versione quantizzata del punto di cui si stanno cercando i k nearest neighbour. Fatto ciò si procede a calcolare la distanza di ogni punto dalla versione quantizzata della query, utilizzando la versione quantizzata dei punti del dataset o meno a seconda che si voglia utilizzare la distanza simmetrica o asimmetrica.

Invece, nel caso di ricerca non esaustiva, bisogna determinare i *w* centroidi coarse più vicini alla query. Per ognuno di questi, viene calcolato il residuo della query rispetto ad esso e vengono considerati tutti i punti del dataset che esso quantizza. I k nearest neighbour non verranno quindi più cercati all'interno dell'intero dataset ma solo tra questi punti. Come nel caso precedente si potrà adottare una delle due diverse tipologie di distanza (nel caso di distanza simmetrica non si quantizzerà più la query ma il suo residuo).

Strutture Dati

Segue una panoramica sulle strutture dati utilizzate nel codice

- centroids: matrice di float di dimensione $m * k * \left(\frac{d}{m}\right)$, dove *m* è il numero di gruppi utilizzati nella quantizzazione prodotto, *k* il numero di centroidi presenti in ciascuno degli m codebook utilizzati per la quantizzazione e $\frac{d}{m}$ rappresenta la dimensione di ciascun sottogruppo considerato; la matrice è memorizzata come un vettore in row-major order e contiene al suo interno i centroidi ottenuti dalla fase di indexing. Logicamente questa struttura può essere pensata come la sequenza degli m codebook. Questa scelta è stata fatta per mantenere una correlazione logica tra aree diverse della matrice, in modo da poter sfruttare il principio di località della cache quando vengono fatte ricerche in questa matrice.
- centroid_of_point: matrice di interi di dimensione $n * m$, dove *n* è il numero di punti contenuti nel dataset ed *m* rappresenta il numero di gruppi in cui si dividerà ciascun punto nella quantizzazione prodotto. Memorizzata come un vettore in row-major order, contiene al suo interno la quantizzazione di ciascun punto, ottenuta attraverso il quantizzatore prodotto. Questa viene memorizzata inserendo per ciascun gruppo l'indice del centroide che approssima la porzione (*d_star* dimensioni) del punto relativa a quel gruppo. Nella ricerca non esaustiva questa avrà solo il ruolo di struttura di supporto per la fase di indexing ma non avrà alcuna utilità nella fase di search.
- distances_between_centroids: matrice di dimensioni $(m * k) * k$ memorizzata all'interno di un vettore in row-major order, allocata solo in caso di utilizzo di distanze simmetriche. Al suo interno contiene, dati due centroidi *i* e *j* appartenenti al gruppo *g*, la distanza tra il centroide *i* ed il centroide *j* del gruppo dato. Visto che le distanze sono simmetriche, la matrice risulterà simmetrica rispetto alla diagonale. Per questo motivo in un primo momento si era pensato di memorizzare, per ciascun gruppo, solamente le diagonali superiori. Ciò però comportava una indicizzazione più complessa che aveva serie ripercussioni sulle prestazioni dell'intera applicazione in caso di utilizzo di distanze simmetriche. Per questo motivo si è preferito rinunciare a prestazioni spaziali per beneficiare di migliori prestazioni dal punto di vista temporale.

Per quanto riguarda la ricerca non esaustiva, oltre a quelle già citate, si ricorre a strutture aggiuntive:

- coarse_centroids: matrice di dimensione $kc * d$, dove *kc* è il numero di centroidi coarse utilizzati e *d* la dimensione dei punti, contenente al suo interno i centroidi coarse ottenuti dalla fase di indexing; viene memorizzata su un vettore in row-major order.
- lista_invertita: struttura di interi di dimensioni $n * (m + 1)$, dove *n* è il numero di punti del dataset ed *m* il numero di sottogruppi in cui viene diviso ciascun punto. La struttura ha una forma a matrice memorizzata su un vettore in row-major order. Per ogni riga è presente in posizione 0 l'id del punto del dataset ed a seguire in ciascuna delle m celle successive sarà memorizzato il numero di riga del centroide del quantizzatore prodotto che approssima quella porzione di residuo.
- punti_caricati: vettore di interi di dimensione kc, con *kc* pari al numero di centroidi coarse, di supporto per la visita della lista invertita. In ciascuna cella sarà caricato il numero di punti del dataset quantizzati dal centroide coarse corrispondente.
- celle_prima: vettore di interi, anch'esso di dimensione *kc*, in cui ogni posizione indica il numero di celle che precedono la porzione della lista invertita relativa al centroide coarse corrispondente ad essa. Si è scelto di memorizzare questa struttura per evitare l'introduzione di un overhead computazionale nell'accesso alla lista invertita, senza di esse infatti si dovrebbe andare ad accumulare il numero di punti relativi ai centroidi coarse precedenti a quello che stiamo considerando per poter calcolare quale sia l'inizio della zona di nostro interesse.

Fasi del progetto

L'attività progettuale è stata sviluppata procedendo secondo i seguenti passi. In un primo momento è stato implementato l'algoritmo interamente in linguaggio C con qualche piccola ottimizzazione; successivamente, una volta individuate le funzioni ottimizzabili, per ogni versione del codice, è stata introdotta una funzione scritta in assembly ottimizzata mediante loop unrolling e loop vectorizzation. Ciascuna fase è stata intervallata dai test per verificare la correttezza e per commisurarne le prestazioni.

1. Prima implementazione in C

La prima versione del codice è stata stilata esclusivamente in linguaggio C, già in questa fase sono state adottate delle ottimizzazioni all'interno del codice, in particolare è stato applicato un loop unrolling di un fattore pari a 4 nelle funzioni residual, objective_function e distance che si occupano rispettivamente del calcolo dei residui, della funzione obbiettivo e del calcolo della distanza al quadrato tra due punti. Si riporta di seguito il codice delle tre funzioni sopra citate.

```
1.  void residual(VECTOR res,VECTOR x,VECTOR centroid,int d){
2.      int i=0;
3.
4.      for(i;i<=d-4;i+=4){
5.          res[i] = x[i]-centroid[i];
6.          res[i+1] = x[i+1]-centroid[i+1];
7.          res[i+2] = x[i+2]-centroid[i+2];
8.          res[i+3] = x[i+3]-centroid[i+3];
9.      }
10.
11.     // Ciclo per eventuali residui
12.     for(i;i<d;i++)
13.         res[i] = x[i]-centroid[i];
14.
15. }//residual
```

Figura 1 - residual con loop unrolling

```
1.  float objective_function(int n,int m, MATRIX distances_from_centroids){
2.        float sum = 0;
3.      int i=0;
4.
5.      for(i; i<n*m-4; i+=4){
6.        sum+=distances_from_centroids[i];
7.        sum+=distances_from_centroids[i+1];
8.        sum+=distances_from_centroids[i+2];
9.        sum+=distances_from_centroids[i+3];
10.     }
11.
12.     for(i;i<n*m;i++){
13.       sum += distances_from_centroids[i];
14.     }
15.
16.     return sqrt(sum);
17. }// objective_function
```

Figura 2 - objective_function con loop unrolling

```
1.  float distance(VECTOR x1,VECTOR x2,int d){
2.      float diff,sum = 0;
3.      int i=0;
4.
5.      // Si lavora su gruppi di 4
6.      for(i; i<=d-4; i+=4){
7.          diff = x1[i]-x2[i];
8.          sum += diff*diff;
9.          diff = x1[i+1]-x2[i+1];
10.         sum += diff*diff;
11.         diff = x1[i+2]-x2[i+2];
12.         sum += diff*diff;
13.         diff = x1[i+3]-x2[i+3];
14.         sum += diff*diff;
15.     }
16.
17.
18.     // Processiamo un eventuale residuo
19.     for(i;i<d;i++){
20.         diff = x1[i]-x2[i];
21.         sum += diff*diff;
22.     }
23.
24.     return sum;
25.
26. }//distance
```

Figura 3 - distance con loop unrolling

Un’altra particolarità del codice risiede nella politica per il riempimento della matrice ANN, ciascuna porzione contenente i k nearest neighbours per la query considerata viene riempita come un array ordinato in maniera decrescente (la misura di riferimento è ovviamente la distanza). Nella posizione 0 della porzione si troverà il knn-esimo nearest neighbur, nella posizione knn invece il primo nearest neighbour. Questa scelta progettuale è stata intrapresa al fine di evitare la scansione dell’intera porzione prima di poter stabilire se il punto del dataset debba essere inserito nei k nearest neighbours del punto considerato. Con questa strategia sarà sufficiente confrontare la distanza tra

la quantizzazione del punto del dataset e la query e la distanza tra la query e l'ultimo knn, se questa sarà maggiore sarà inutile scansionare la porzione.

2. Versione 1

Nella versione successiva del codice è stata implementata in linguaggio assembly la funzione *accumulate*, il cui compito è quello di accumulare i valori delle dimensioni di ciascun punto relativamente al centroide di appartenenza, per poter effettuare l'aggiornamento dei centroidi. Per quanto riguarda il codice a 32 bit (implementato con istruzioni SSE) si è scelto di applicare il loop unrolling a 64, 16 e 4 bit al fine di migliorare le prestazioni; per quanto riguarda invece l'implementazione a 64 bit (istruzioni AVX) il loop unrolling applicato è a 128 e 16 bit. Si mostra un esempio di loop unrolling per le versioni a 32 ed a 64-bit.

```
129
130 for_16:
131     cmp edx, 16     ; Confronto edx < 16 ?
132     jl for_4     ; Se edx è strettamente minore di 16, gestisco il residuo
133
134     ;Loop Unrolling 1: 4 valori
135     movaps xmm0, [eax]    ; xmm0 = tmp[centroide*d_star]
136     addps xmm0, [ecx]     ; xmm0 = xmm0+[src]   ;tmp[centroide*d_star] += ds[i*d+(g*d_star)]
137     movaps [eax], xmm0    ; tmp[centroide*d_star] = xmm0
138
139     ;Loop Unrolling 2: 4 valori
140     movaps xmm0, [eax+16]    ; xmm0 = tmp[centroide*d_star]
141     addps xmm0, [ecx+16]     ; xmm0 = xmm0+[src]   ;tmp[centroide*d_star] += ds[i*d+(g*d_star)]
142     movaps [eax+16], xmm0    ; tmp[centroide*d_star] = xmm0
143
144     ;Loop Unrolling 3: 4 valori
145     movaps xmm0, [eax+32]    ; xmm0 = tmp[centroide*d_star]
146     addps xmm0, [ecx+32]     ; xmm0 = xmm0+[src]   ;tmp[centroide*d_star] += ds[i*d+(g*d_star)]
147     movaps [eax+32], xmm0    ; tmp[centroide*d_star] = xmm0
148
149     ;Loop Unrolling 4: 4 valori
150     movaps xmm0, [eax+48]    ; xmm0 = tmp[centroide*d_star]
151     addps xmm0, [ecx+48]     ; xmm0 = xmm0+[src]   ;tmp[centroide*d_star] += ds[i*d+(g*d_star)]
152     movaps [eax+48], xmm0    ; tmp[centroide*d_star] = xmm0
153
154     sub edx, 16      ;sottraggo i 16 elementi già presi
155     add eax, 64     ;mi sposto di 16 elementi (64 posizioni)
156     add ecx, 64
157
158     jmp for_16      ; Salto incondizionato
159
```

Figura 4 - loop unrolling + loop vectorizzation nella versione a 32 bit

```
123
124  for_16:
125
126      cmp rdx, 16        ; Confronto n*m < 8 ?
127      jl for_remain              ; Se edx è strettamente minore di 8, gestisco il residuo
128
129                  ;Loop Unrolling 1: 8 valori
130      vmovaps ymm0,[rax]      ; copio i primi 8 valori di dest in ymm0
131      vaddps  ymm0,[rcx]      ; aggiungo a ymm0 i primi 8 valori di source
132      vmovaps [rax],ymm0      ; ricopio la somma dei primi 8 valori di dest e source in dest
133
134                  ;Loop Unrolling 2: 8 valori
135      vmovaps ymm0,[rax+32]      ; copio i secondi 8 valori di dest in ymm0
136      vaddps  ymm0,[rcx+32]      ; aggiungo a ymm0 i secondi 8 valori di source
137      vmovaps [rax+32],ymm0      ; ricopio la somma dei secondi 8 valori di dest e source in dest
138
139
140      sub rdx, 16            ;sottraggo i 32 elementi già presi
141      add rax, 64           ;mi sposto di 32 elementi (128 posizioni)
142      add rcx, 64           ;mi sposto di 32 elementi (128 posizioni)
143      jmp for_16
144
```

Figura 5 - loop unrolling + loop vectorizzation nella versione a 64 bit

3. Versione 2

In questa versione viene introdotta l'implementazione in Assembly della funzione *divide* che divide le cordinate di ciascun centroide per il contatore che tiene traccia del numero di punti appartenenti al centroide al fine di fare la media delle cordinate ed ottenere i nuovi centroidi. Il loop unrolling adottato nella versione a 32 bit è a 64, 16 e 4 bit, nella versione a 64 bit è a 128 ed a 16 bit.

4. Versione 3

Qui introduciamo l'implementazione in assembly ottimizzata della funzione *memset_float* che serve per settare porzioni di array di float con un valore che si riceve in input. Per il loop unrolling nella versione a 32 bit è stata utilizzata la stessa politica illustrata precedentemente, mentre nella versione a 64 si è adottato un loop unrolling a 128, 32, 8 e 4 bit, visto che gli array da inizializzare hanno dimensione variabile.

5. Versione 4

La funzione che si è andata ad ottimizzare in assembly è *residual*, sia per la versione a 32 bit del codice che per quella a 64 sono stare utilizzate per il loop unrolling le stesse strategie utilizzate per le corrispondenti versioni di accumulate e distance.

6. Versione 5

In questa versione la funzione che è stata ottimizzata è *objective_function* il cui loop unrolling è stato implementato secondo la stessa strategia precedentemente illustrata.

7. Versione 6 – Assembly

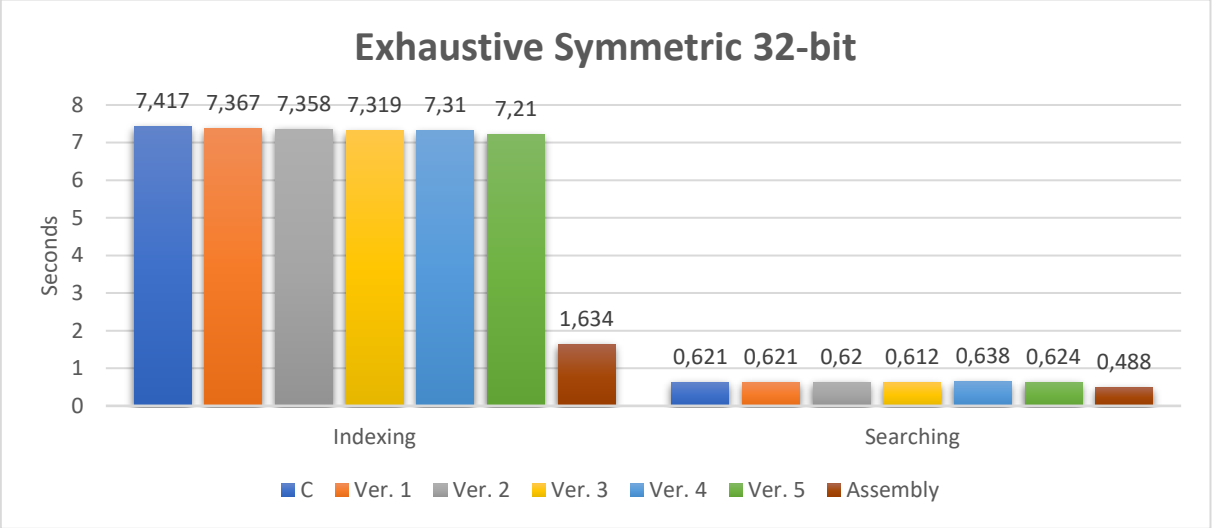
Viene introdotta l'ultima funzione ottimizzata in assembly, ovvero *distance*. Questa si è dimostrata nel corso del progetto la funzione più critica per quanto concerne le prestazioni dell'intera applicazione siccome è la più richiamata all'interno del codice. Si mostreranno più dettagliatamente, nella sezione a seguire, i guadagni ottenuti in termini di prestazione nelle varie versioni del codice ed altri dettagli riguardanti le prestazioni. Nell'ultima versione si è inoltre introdotto il padding all'interno del dataset e del queryset in maniera tale da evitare accessi non allineati in memoria dovuti al fatto che il valore della dimensione di ciascun sottogruppo possa non essere divisibile per 4 nella versione a 32 bit e per 8 nella versione a 64 bit. L'alternativa a questa soluzione sarebbe stata l'utilizzo delle versioni non allineate delle istruzioni SSE ed AVX e ciò avrebbe ridotto le performance a causa della minor efficienza di tale versione delle istruzioni.

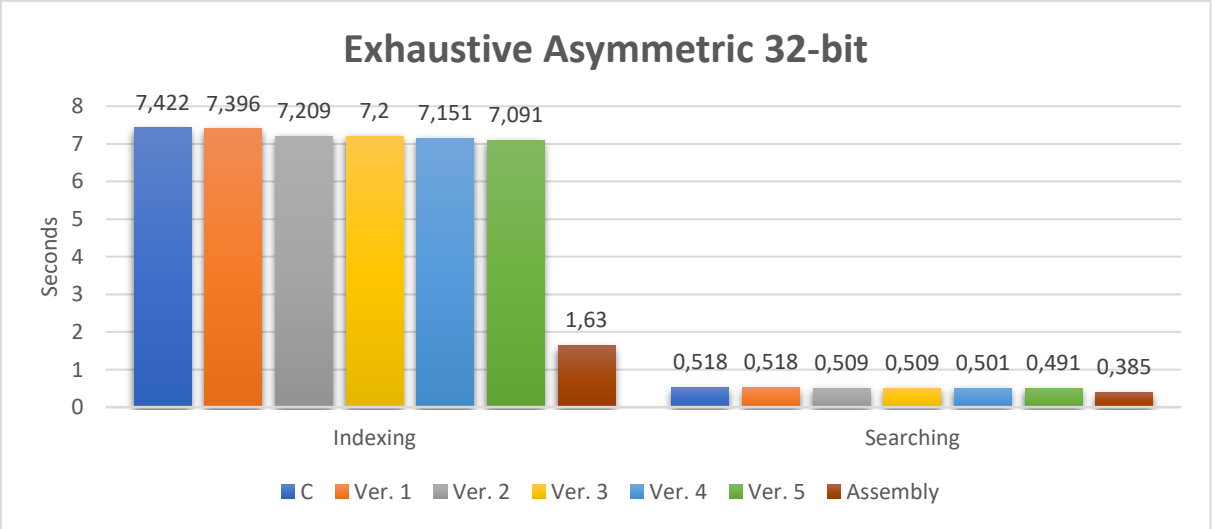
Prestazioni

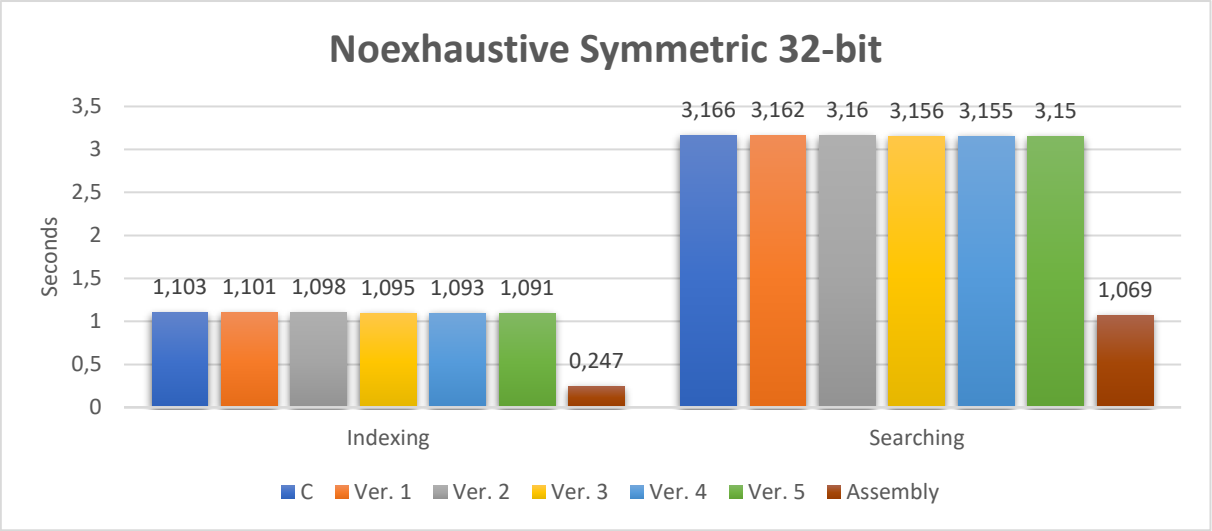
In questa sezione confrontiamo le prestazioni ottenute nelle varie versioni del codice. Il calcolatore utilizzato in fase di test presenta le seguenti specifiche:

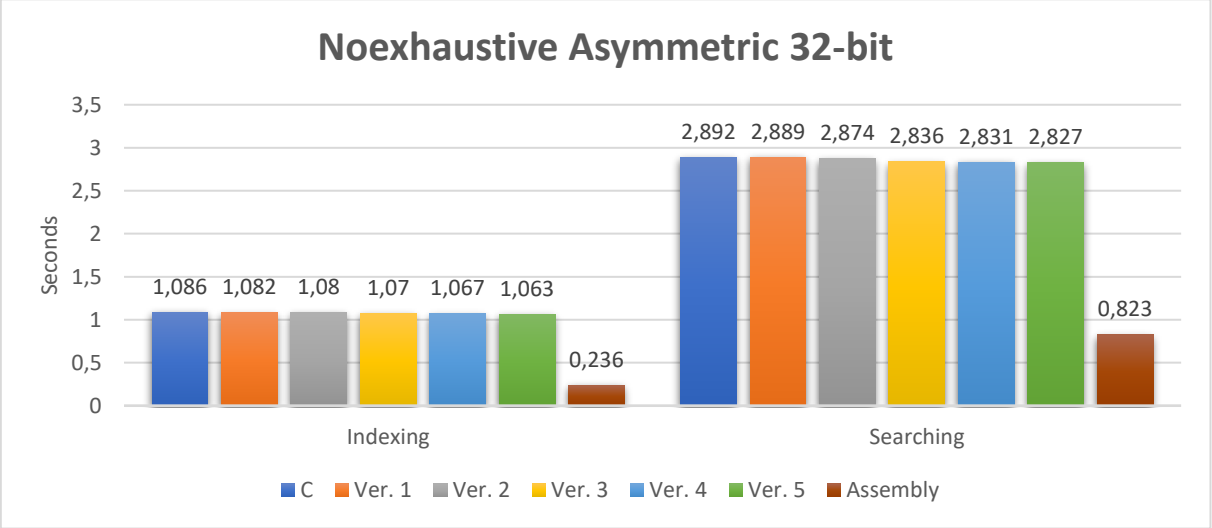
- Intel® Core™ i7-7700HQ @ 2.80 GHz;
  RAM: 16 GB.

Ogni risultato riportato è frutto della media dei tempi di più esecuzioni, in maniera tale da ottenere risultati che non siano influenzati dallo stato di carico della CPU.
Faremo riferimento all'esecuzione del codice a 32 bit analizzando le varie tipologie di indicizzazione e di ricerca. Di seguito vengono riportati i tempi registrati nelle varie versioni del codice.





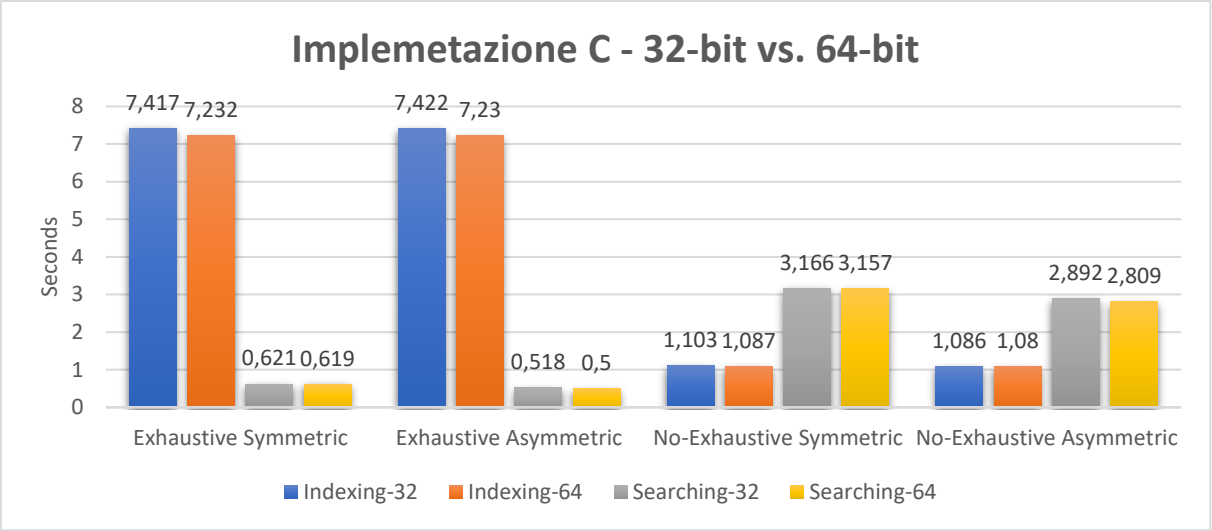




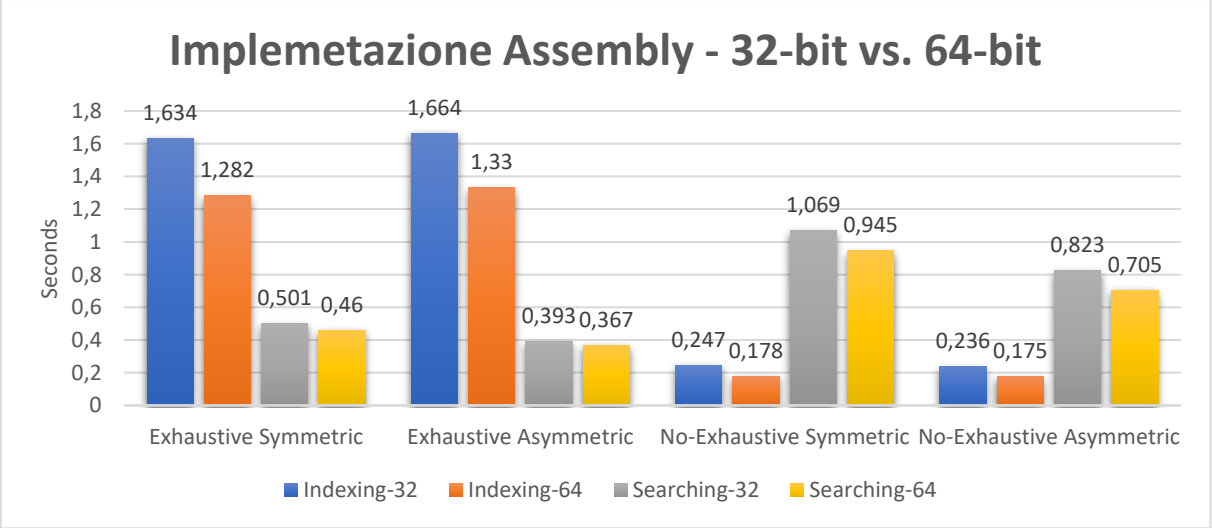
Possiamo notare come in tutte le versioni l'aumento più significativo di prestazioni si abbia con l'introduzione della funzione distance ottimizzata in assembly. Tale risultato è dovuto al fatto che questa è la funzione richiamata il maggior numero di volte.

32 bit vs 64 bit

Confrontiamo adesso la differenza di prestazioni tra le versioni a 32 ed a 64 bit facendo riferimento in prima istanza al codice implementato esclusivamente in C, relativo alla prima versione, e successivamente a quello finale con tutte le ottimizzazioni effettuate durante il corso del progetto. Per quanto riguarda la prima versione notiamo come la differenza di prestazioni tra la versione a 32 e quella a 64 bit è poco significativa; prestazioni lievemente migliori si notano nella versione a 64 bit.



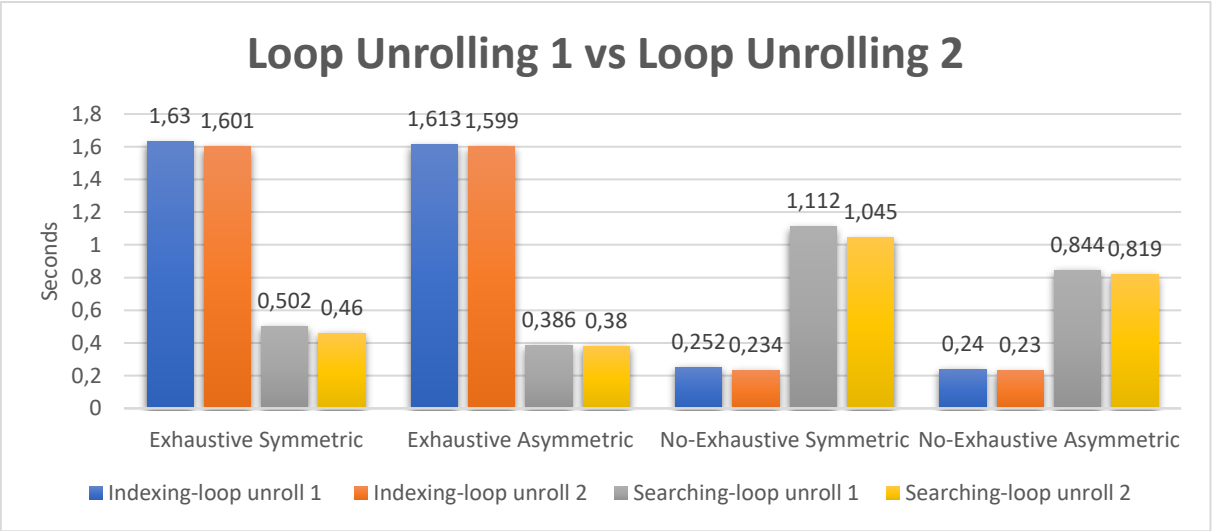
Per quanto riguarda il confronto tra le prestazioni ottenute nel codice a 32 bit e quello a 64 bit nella versione finale con le ottimizzazioni delle funzioni cruciali in Assembly, notiamo come la differenza tra le prestazioni del codice 32 e quello 64 bit sia più marcata, questa differenza è dovuta alla loop vectorizzation che nella versione a 32 bit elaborerà 4 operandi alla volta, mentre in quella a 64 ne elaborerà 8 simultaneamente. I tempi di esecuzione non si dimezzano in quanto ci sono anche altre porzioni di codice che danno il loro contributo alle prestazioni e che non presentano tali differenze.



Unrolling delle funzioni in Assembly

Scrivere tutte le funzioni in Assembly ha consentito di ottenere un boost delle prestazioni dovuto all'utilizzo delle istruzioni/registri vettoriali, ma al fine di ottenere un ulteriore miglioramento dei tempi è stata applicato la strategia dell'unrolling a tutti i loop presenti in tali funzioni. Nello scegliere il modo in cui effettuare l'unrolling sono state effettuate diverse prove. Inizialmente è stato introdotto un unrolling "esponenziale" (ovvero a 128, 64, 32, 16, 8 e 4 istruzioni), ma ciò si è in realtà rivelato

controproducente. Questa "saturazione" dell'efficacia dell'unrolling al crescere del numero di step in cui si articola si giustifica considerando l'esecuzione speculativa delle istruzioni da parte di una macchina superscalare, che penalizza l'esecuzione ogni volta che la predizione di un salto condizionale si rivela errata. In effetti, nella seconda strategia di loop unrolling, vista la minore presenza di sati condizionati, si avrà una minore possibilità di svuotare la pipeline. Alla fine si è scelto un trade-off che porta la maggior parte delle istruzioni in AVX ad avere un unrolling a 128 istruzioni, a cui segue direttamente un secondo step di unrolling direttamente a 16.
Nel grafico seguente, si fa riferimento alla funzione che calcola la distanza per evidenziare quanto descritto sopra.



Loop vectorizzation e loop unrolling

In ultima analisi si mostrano i benefici ottenuti dall'introduzione nel codice assembly della loop vectorizzation, e successivamente del loop unrolling seguendo la seconda strategia illustrata sopra. Nel grafico di seguito si può notare come il miglioramento più consistente si ottenga nell'indexing, più precisamente nell'introduzione della loop vectorizzation.