UNIVERSITÀ
DELLA CALABRIA

DIPARTIMENTO DI INGEGNERIA INFORMATICA, MODELLISTICA, ELETTRONICA E SISTEMISTICA (DIMES)

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Relazione del Progetto:

Product Quantization for Nearest Neighbor Search

a cura di:

Alessandro Barbieri (207015)

Antonio Commisso (207020)

Antonella Sola (207007)

A.A. 2018/2019

Descrizione del problema

La ricerca Nearest Neighbor (NN), nota anche come ricerca di prossimità o ricerca per similitudine è utilizzata per l'analisi e l'elaborazione di dati su larga scala, in particolare per l'analisi di contenuti multimediali che infatti, sono spesso ad alta dimensionalità. Questa tecnica ha molti campi applicativi, come ad esempio il recupero di immagini in base al contenuto (data mining multimediale), in cui i dati nel maggiore dei casi, sono rappresentati con vettori multidimensionali.
La ricerca NN è un problema di ottimizzazione che mira a trovare in un dato insieme il punto (o i punti) più vicino o il punto più simile ad un dato punto.

Formalmente: Dato un *dataset*, ovvero un insieme $Y \subset R^d$ di n vettori d-dimensionali (detti anche punti), ed un punto $x \in R^d$ (detto anche query o interrogazione), il nearest neighbor NN(x) di x in Y è il punto y ∈ Y che minimizza la distanza Euclidea dist(x, y) da x:

$$dist(x,y) = \sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$$

ovvero:

$$\mathrm{NN}(x) = arg \min_{y \epsilon Y} dist(x,y)$$

La ricerca NN su larga scala di dati ad alta dimensionalità è molto onerosa dal momento che gli archivi sono limitati, così come lo sono anche le risorse computazionali. Per questo motivo l'approccio brute force, che consiste nel calcolare la distanza tra x ed ogni punto y di Y per poi restituire il punto y^* avente distanza minima, rappresenta la tecnica più semplice per il calcolo del nearest neighbor ma non la più efficiente.
Invece di realizzare l'esatto algoritmo NN, da alcuni anni, la ricerca è orientata su un algoritmo di ricerca, il cosiddetto Approximate Nearest Neighbor (ANN) il quale risulta avere un costo temporale inferiore all'approccio brute force. Tale algoritmo non garantisce di restituire l'esatto nearest neighbor del punto query, ma bensì solo una sua approssimazione, che in genere si assume sufficientemente accurata per gli scopi applicativi.

Descrizione dell'algoritmo

Un metodo utilizzato nella ricerca ANN è la **Vector Quantization (VQ):**
Un *codebook* C è un insieme $C = \{c_1, c_2, ..., c_k\}$ di k vettori d-dimensionali $c_i \in \mathbb{R}^d$, detti anche *centroidi.*
Un quantizzatore vettoriale (VC), q è una funzione che mappa ogni punto $x \in \mathbb{R}^d$ in un centroide,

ovvero q(x) ∈ C. In particolare, q (oppure q_c se si vuole enfatizzare l'insieme di centroidi C su cui q è definito) restituisce il centroide di C che risulta essere più vicino ad x:

$$q(x) = arg \min_{c_i \in C} dist(x, c_i)$$

Un buon insieme di centroidi C per un *dataset* Y è tale da minimizzare la somma delle distanze al quadrato tra ogni punto y di Y ed il corrispondente centroide $q_c(y)$, ovvero

$$C^* = arg \min_{C \subseteq R^d : |C| = k} \sum_{y \in Y} dist(y, q_c(y))^2$$

Determinare l'ottimo globale C^* per la precedente funzione obiettivo è un problema intrattabile. Nella pratica si utilizza come codebook un ottimo locale di tale funzione obiettivo, che può essere efficientemente calcolato utilizzando l'algoritmo di clustering k-means.

L'algoritmo k-means inizializza i centrodi $c_1, c_2, \ldots, c_k$ di C selezionando k punti causali di Y e poi procede in maniera iterativa. Ad ogni iterazione ogni centroide viene sostituito dal centro geometrico dei punti ad esso più prossimi. L'algoritmo converge quando il valore della funzione obiettivo in due iterazioni successive non supera una determinata soglia.

Un secondo metodo utilizzato nella ricerca ANN al crescere della dimensionalità di d è il **Product Quantization (PQ):**

Dato un *dataset* Y, è di interesse riuscire a costruire un codebook C tale che, per ogni punto y di Y, y e $q_c(y)$ sono molto vicini. Purtroppo, all'aumentare della dimensionalità d per ottenere una tale proprietà la dimensione k del codebook dev'essere esponenziale in d.

Un *quantizzatore prodotto* (PQ) fornisce una soluzione efficiente al suddetto problema. Dato un parametro m (in genere con d multiplo di m), ogni vettore $x \in R^d$ viene spezzato in m sotto-vettori a $d^* = d/m$ dimensioni. Nel seguito $u_j(x)$ denota il j-esimo ($1 \leq j \leq m$) sotto-vettore di x, composto dal j-esimo gruppo di d^* elementi consecutivi di x. I sotto-vettori di ogni gruppo j vengono quantizzati separatamente ottenendo m distinti quantizzatori vettoriali $q_1, q_2, \ldots, q_m$, detti anche sotto-quantizzatori. La quantizzazione di x si ottiene quindi come segue:

$$q(x) = (q_1(u_1(x)), q_2(u_2(x)), \ldots, q_m(u_m(x))),$$

ovvero è data dalla concatenazione degli m centroidi d^*-dimesionali restituiti dagli m distinti quantizzatori q_j applicati ognuno al rispettivo sotto-vettore $u_j(x)$, j=1,2,...,m.

Si assume che i codebook $C_1,C_2,...,C_m$ associati agli m quantizzatori $q_1,q_2,...,q_m$ siano formati dallo stesso numero k^* di centroidi. Quindi, il numero totale k di centroidi che devono essere memorizzati da un PQ è dato da mk^*, mentre il numero totale di distinti centroidi che possono essere ottenuti mediante la loro concatenazione è di gran lunga più elevato, ovvero pari a $(k^*)^m$.

Ricerca ANN esaustiva

Utilizzando un PQ è possibile calcolare una distanza Euclidea approssimata tra il punto query *x* ed ogni punto *y* del *dataset* utilizzando due strategie: calcolo della ***distanza simmetrica (SDC)*** oppure calcolo della ***distanza asimmetrica (ADC).*** Dato un parametro K, tala ricerca restituisce i K punti del *dataset* che minimizzano la ADC oppure la SDC.

La *distanza simmetrica* si ottiene come segue

$$dist(x, y) \approx dist_S(x, y) = dist(q(x), q(y)) = \sqrt{\sum_{j=1}^{m} dist(q_j(u_j(x)), q_j(u_j(y)))^2}. \quad (1)$$

Poichè $q_j(u_j(x)),q_j(u_j(y)) \in C_j$, le distanze tra ogni coppia di centroidi $c',c'' \in C_j$ possono essere precalcolate (ad un costo $O(mk^*d)$) e riutilizzate, per ogni punto query *x*, per calcolare la distanza simmetrica ad un costo ridotto, ovvero O(m) anzichè O(d).

La *distanza asimmetrica* si ottiene come segue

$$dist(x,y) \approx dist_A(x,y) = dist(x,q(y)) = \sqrt{\sum_{j=1}^{m} dist(u_j(x), q_j(u_j(y)))^2}\ . \quad (2)$$

In questo caso, dato un punto query *x*, le distanze $dist(u_j(x),q_j(u_j(y)))$ tra $u_j(x)$ ed ogni centroide $c' \in C_j$ possono essere precalcolate e riutilizzate per calcolare la distanza asimmetrica ad un costo ridotto.

La differenza tra le due soluzioni è che nel caso della distanza simmetrica le distanze precalcolate sono indipendenti dalla query e quindi possono essere riutilizzate per ogni altra query, mentre la distanza simmetrica ha bisogno della query e quindi le distanze precalcolate non possono essere riutilizzate per altre query. Per contro, la distanza asimmetrica è più accurata di quella simmetrica.

Ricerca ANN non esaustiva

La tecnica precedente consente di ridurre il costo della singola distanza, ma richiede che la query sia confrontata con la versione quantizzata di ogni punto del *dataset*. Quando il numero n di punti del *dataset* è molto grande si rende necessario ridurre anche il numero di punti da confrontare con la query, adottando una tecnica di ricerca non esaustiva.

A questo scopo si utilizzano due quantizzatori: un quantizzatore vettoriale cosiddetto *coarse* (ovvero grossolano) q_c (VQ) ed un quantizzatore prodotto (più accurato) q_p (PQ). Il quantizzatore vettoriale q_c utilizza k_c centroidi d-dimensionali C_c e viene utilizzato per definire il vettore dei residui

$$r(y) = y - q_c(y),$$

corrispondente al vettore y nel caso di origine dello spazio coincidente con il suo centroide $q_c(y)$, mentre il quantizzatore prodotto q_p corrisponde alla quantizzazione dei residui $r(y)$. Utilizzando q_c e q_p il vettore y può essere approssimato come segue

$$y \approx q_c(y) + q_p(y - q_c(y))$$

e di conseguenza la distanza $dist(x, y)$ tra x ed y può essere approssimata come segue

$$dist(x, y) \approx dist(x - q_c(y), y - q_c(y)). \quad (3)$$

Il quantizzatore q_p è unico per tutti i punti del *dataset* ed i suoi centroidi vengono determinati facendo uso di un campione R_Y di $n_r \leq n$ residui del *dataset*, ovvero $R_Y \subseteq \{r(y) : y \in Y\}$ e $|R_Y| = n_r$.

La tecnica di indicizzazione non esaustiva opera come segue:

(i) si determinano i w centroidi grossolani $c_i \in C_c$ che risultano essere più vicini alla query x;

(ii) per ogni centroide c_i determinato al passo (i), si calcolano le distanze approssimate sfruttando il quantizzatore prodotto q_p — utilizzando l'Eq. (3) in congiunzione con l'Eq. (1) (SDC) oppure con l'Eq. (2) (ADC) — tra x ed ogni altro punto y tale che $q_c(y) = c_i$ e si collezionano i K punti associati alle distanze complessivamente più piccole;

(iii) i K punti determinati al passo (ii) vengono restituiti come ANN approssimati della query x.

Progettazione e implementazione dell'algoritmo in C

All'interno del progetto, l'algoritmo di ricerca dell'Approximate Nearest Neighbor è stato implementato suddividendolo in base ai due approcci di ricerca proposti in letteratura quali:

1. *Ricerca ANN esaustiva*
2. *Ricerca ANN non esaustiva.*

Entrambe le ricerche sono dotate dei metodi

- *Pqnn_index*
- *Pqnn_search*

Il primo è utilizzato per istanziare e popolare le strutture dati da sfruttare nella successiva fase di ricerca, in particolare, apprende i centroidi utilizzando il metodo k-means. Nel secondo invece, sulla base delle strutture dati implementate nel metodo di cui sopra, si procede con la ricerca vera e propria degli ANN per ogni query. In generale, si è optato di memorizzare le matrici come vettori di elementi contigui, al fine di sfruttare il principio di località. Nella maggior parte dei casi, in seguito ad alcune verifiche, è stata scelta la rappresentazione in *row-major-order* dimostratasi nella pratica la più conveniente.
Al fine di agevolare l'implementazione, si è pensato di sviluppare metodi generici quali:

- *Kmeans*
- *Distanza*
- *CalcolaPQ*
- *CalcolaFob*
- *CalcolaIndice*

utilizzabili in entrambe le tipologie di ricerca.
Il metodo *kmeans* riceve come parametri, oltre alla struct params anche una struct kmeans_data, contenente i puntatori ad alcune strutture utilizzate all'interno dell'algoritmo, al fine di poter essere usato in entrambe le ricerche.

La struct è così definita:

```
typedef struct{
    float* source;
    int dim_source;
    int* index;
    float* dest;
    int index_rows,index_columns;
    int n_centroidi;
    int d;
} kmeans_data;
```

- *source* indica l'insieme dei punti utilizzati che, per la ricerca esaustiva sarà il *dataset* mentre per la non esaustiva potrà essere o il *dataset* o il *resisualset*;
- *dim_source* indica il numero di punti presenti in source;
- *index* è il vettore che associa ad ogni punto della source il corrispondente centroide;
- *dest* invece è il *codebook* da restituire;
- *Index_row, index_columns, n_centroidi, d* rappresentano rispettivamente le dimensioni dell'indice, il numero di centroidi e il numero di dimensioni di ogni vettore.

Oltre alla struct sopra citata, kmeans riceve in input anche due interi quali start ed end che rappresentano l'uno l'indice di inizio, e l'altro la fine della partizione del *dataset* su cui si sta operando. Nel caso del calcolo dei centroidi grossolani, non essendo il *dataset* suddiviso in partizioni, start è impostato pari a 0 ed end pari al numero di dimensioni della sorgente.

Il metodo *kmeans* inizialmente richiama la funzione *calcolaPQ*, successivamente esegue un ciclo fino al convergere della funzione obiettivo o, fino al numero massimo di iterazioni specificato in input. In tale ciclo sono inizialmente azzerati i valori del codebook poi, sono sommate ad ogni centroide le coordinate dei punti appartenenti alla stessa cella di Voronoi. A questo punto, ogni coordinata è divisa per il numero di punti appartenenti alla cella. Infine, dopo aver richiamato nuovamente la funzione *calcolaPQ* in modo tale da ricavare i centroidi associati ad ogni punto del *dataset*, è richiamata la funzione *calcolaFob* al fine di determinare il nuovo valore della funzione obiettivo in modo e controllare sulla base di quest'ultimo se continuare ad iterare o meno. Il metodo *distanza* è stato scritto nel modo più generico possibile, riceve in input due puntatori a float ed un intero *dimensione*

i quali, rispettivamente indicano il numero di coordinate e la locazione iniziale dei due punti tra cui sarà calcolata la distanza. La funzione *calcolaPQ* riceve in input oltre alla struct kmeans_data descritta in precedenza, tre interi: partition, start ed end. Il primo contiene l'indice della partizione del *dataset* su cui operare, start ed end contengono invece, l'indice iniziale e finale della partizione del *dataset*. La funzione calcola quale centroide è più vicino ad ogni punto del *dataset* per determinare la cella di Voronoi di appartenenza.
La funzione *calcolaFob* riceve in input oltre alle struct params e kmeans_data tre interi: ipart, start ed end utilizzati secondo la stessa logica della funzione precedente. Quest'ultima non fa altro che calcolare per ogni punto del *dataset* la distanza euclidea con il corrispondente centroide e, restituire la somma di queste distanze al quadrato vale a dire cioè il valore della funzione obiettivo.
CalcolaIndice è utilizzato per calcolare l'indice per accedere alla posizione corretta della matrice *distanze_simmetriche* contenente le distanze tra due centroidi. La matrice è simmetrica rispetto alla diagonale principale (con valori pari a zero) per cui si è preferito evitare di calcolare e mantenere in memoria distanze doppie e quindi salvarla solo come matrice triangolare inferiore. Le righe della matrice avranno dimensioni crescenti e, per sfruttare al meglio il principio di località, esse sono salvate in memoria consecutivamente.

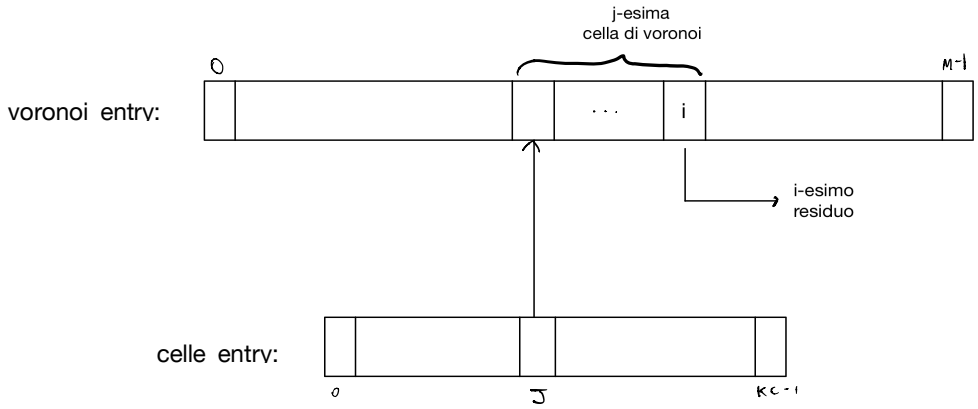
Ricerca ANN esaustiva

Per la ricerca esaustiva durante la fase di *indexing*, in prima istanza, si inizializza il codebook con l'utilizzo della funzione *memcpy* di C al fine di copiare i primi k punti del *dataset*, nell' ipotesi che questo sia il modo più efficiente per copiare parte di una matrice in un'altra. Successivamente viene eseguito il metodo kmeans sulle m partizioni del *dataset* e, in seguito, nel caso della distanza simmetrica, è richiamata la funzione *creaMatriceDistanze* al fine di precalcolare le distanze tra ogni coppia di centroidi per ogni partizione del *dataset*.
Nella funzione *search*, invece, se si utilizza la distanza simmetrica è richiamata la funzione *calcolaPQ* per ogni partizione del *querySet* affinché si possa associare ogni punto del *queryset* al corrispondente centroide. Successivamente, qualunque sia il tipo di distanza scelta, viene richiamata per ogni query la funzione *calcolaNN* che si occupa dell'effettivo calcolo dei Nearest Neighbor. Quest' ultima è suddivisa in due parti a seconda del valore di *knn*: se *knn* è uguale ad uno sarà usato il classico algoritmo per il calcolo del minimo, mentre se *knn* è maggiore di uno verrà utilizzata un'estensione dell'algoritmo precedente, adattata a trovare i *knn* minimi. In questa versione, invece di utilizzare una variabile per il minimo corrente si utilizza un vettore di dimensione *knn*. Questo algoritmo è ispirato

all'*insertion sort* che è in realtà poco efficiente, ma per *knn* molto minore di *n*, risulta essere invece molto valido.

Ricerca ANN non esaustiva

Per la ricerca non esaustiva nella fase di *indexing* rispetto alla ricerca esaustiva si utilizza in più una struttura dati ausiliaria a due livelli. Per l'implementazione di quest'ultima si è pensato di utilizzare un vettore in grado di codificare le celle di Voronoi (*voronoi_entry*) dei centroidi grossolani q_c, ordinati in modo crescente. In ogni posizione di tale vettore vi sono gli indici dei residui corrispondenti alla cella di Voronoi di appartenenza. I residui sono calcolati con il metodo *calcola_residui* che, partendo dal vettore *qc_indexes* assegna ad ogni punto del *dataset* un centroide grossolano e ne calcola la differenza rispetto a tutte le *d* dimensioni con il metodo *compute_residual*. Si è scelto di ottimizzare questa funzione, traducendola in linguaggio assembly in modo tale da poter sfruttare i vantaggi di srotolamento dei cicli e vettorizzazione del codice.
Per salvare alla posizione i-esima l'indice d' inizio dell'i-esima cella di Voronoi si è sfruttato l'utilizzo di un secondo vettore (*celle_entry*). In questo modo è stato possibile memorizzare una matrice sfaldata in un unico vettore.



Durante la fase di ricerca, si calcolano i w centroidi grossolani più prossimi ad ogni query mediante l'ausilio di un Maxheap a capacità limitata. Selezionati quest'ultimi per ogni centroide viene ricavato il residuo della query con il metodo compute_residual, e calcolate le distanze tra residuo e query. Tale distanza sarà inserita in un secondo Maxheap che si occuperà di scegliere le *knn* distanze più piccole, corrispondenti di fatto agli ANN. Ciò è garantito dal fatto che il MaxHeap riceve come chiave la distanza e come valore l'indice di un vettore del *dataset*.

Implementazione dell’algoritmo in linguaggio Assembly

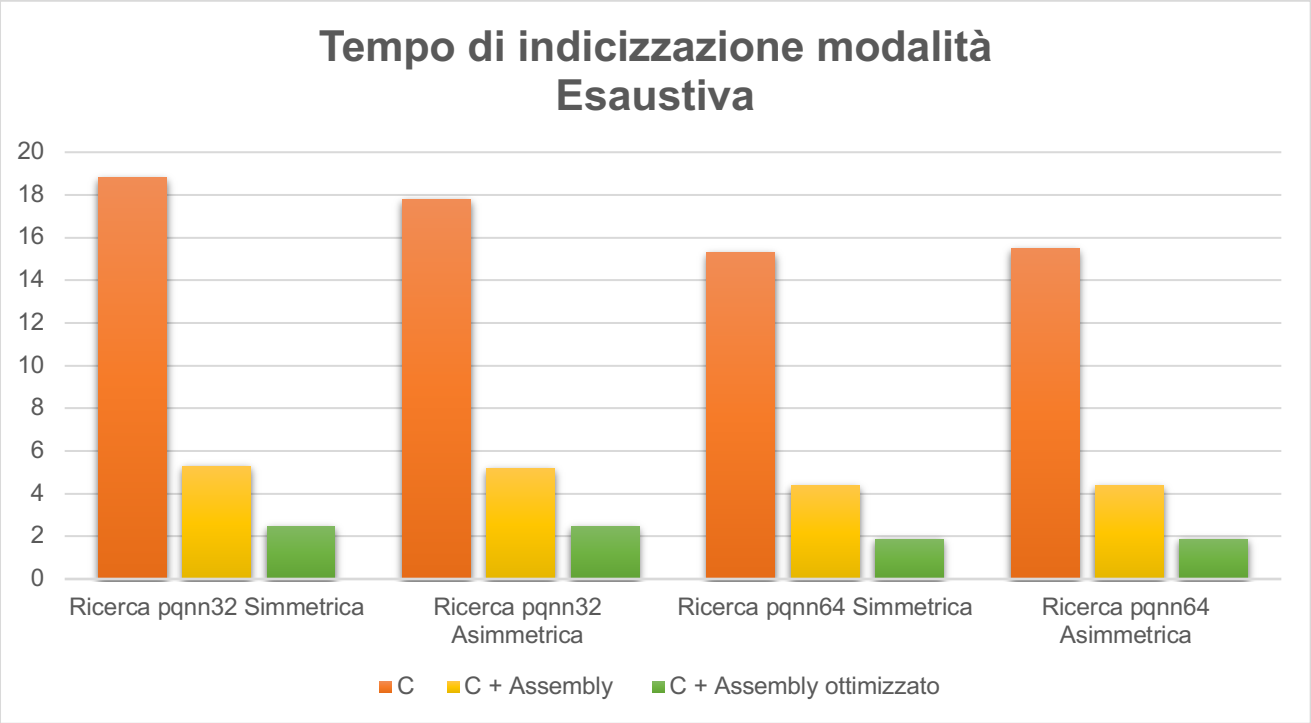
Alcuni metodi del codice fino ad ora illustrato sono stati tradotti in linguaggio Assembly fornendo due soluzioni software, una per l’architettura x86-32+SSE e l’altra per l’architettura x86-64+AVX. Nell’implementazione di tali soluzioni sono state adottate due tecniche di ottimizzazione:

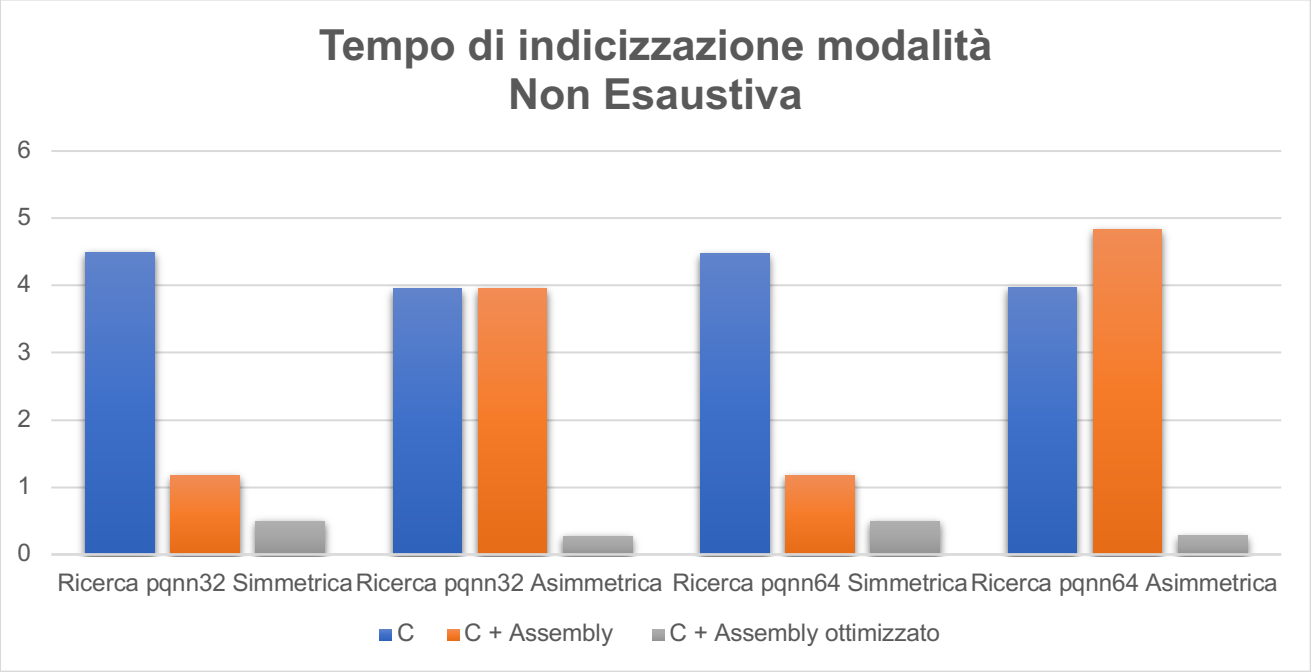
• loop unrolling

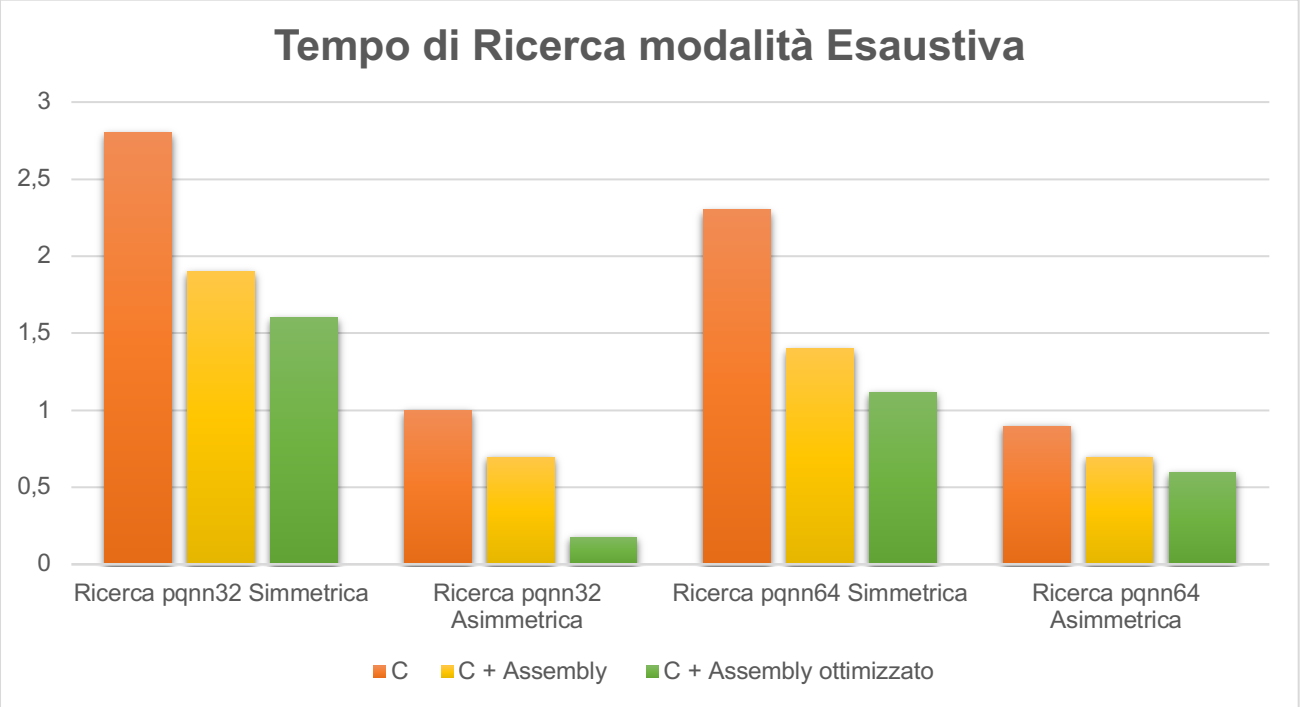
• code vectorization

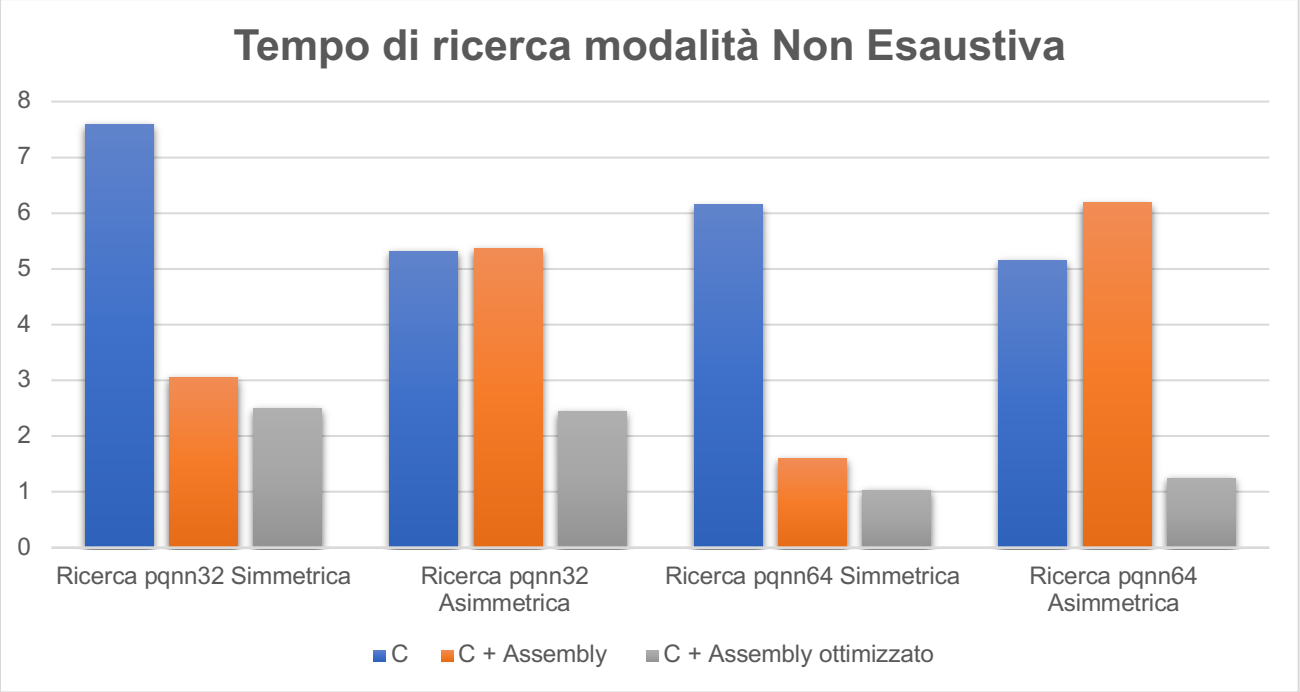
Si è scelto di combinare entrambe le tecniche, scegliendo un opportuno fattore di unrolling per ogni funzione e aumentando così il grado di parallelismo del codice, dando la possibilità di effettuare più operazioni possibili dello stesso tipo contemporaneamente. In particolar modo per implementare la tecnica del loop unrolling sono stati sfruttati i registri vettoriali a virgola mobile messi a disposizione da ciascuna architettura ovvero gli 8 registri XMM a 128 bit per quanto riguarda SSE e i 16 registri YMM a 256 bit per quanto riguarda AVX. L’implementazione della tecnica del code vectorization è stata invece resa possibile per l’esistenza di istruzioni tipo packed grazie alle quali si ha la possibilità di effettuare operazioni dello stesso tipo in parallelo sui gruppi di numeri contenuti nei registri vettoriali. In particolar modo un registro XMM, essendo a 128 bit, può contenere 4 valori di tipo float, mentre i registri YMM 8 di tipo float essendo a 256 bit aumentando ancora di più il grado di parallelismo. Con l’architettura x86-64+AVX, per evitare un calo di prestazioni per il passaggio da una modalità all’altra sono state utilizzate unicamente le istruzioni che lavorano a 256 bit, ossia le istruzioni con il prefisso v-, facendo attenzione a non alternarle mai con quelle che lavorano a 128 bit anche laddove queste ultime risultavano essere più convenienti. Si è riscontrato che nel caso in cui d/m, ossia il numero di dimensioni fratto quello delle partizioni, non risulti essere multiplo di 4 per la versione x86-32+SSE e 8 per la versione x86-64+AVX, non è possibile utilizzare l’istruzione movaps. Una possibile soluzione a questo problema è l’utilizzo del padding ma, in presenza di funzioni con un fattore di unrolling molto elevato, si rischia di dover aggiungere molti zeri e aumentare molto e inutilmente l’occupazione di memoria. Si è preferita dunque, una soluzione alternativa che vede l’utilizzo dell’istruzione movups. Si è verificato che il tempo di esecuzione del programma con l’utilizzo di questa istruzione è paragonabile alla versione allineata. È stato quindi necessario l’utilizzo del ciclo resto, in particolare, per funzioni con un elevato fattore di unrolling si è scelto di introdurre un ciclo intermedio al fine di aumentare il livello di parallelismo tra le operazioni riducendo così al minimo le iterazioni del ciclo scalare. In questo ciclo intermedio, si è utilizzata soltanto la tecnica del code vectorization piuttosto che quella del loop unrolling.

Di seguito, si riportano dei grafici che mostrano la variazione del tempo di esecuzione tra le varie versioni del codice sviluppato. Da quest'ultimi si evince il miglioramento ottenuto scegliendo di tradurre alcune funzioni in linguaggio Assembly. In particolare, le performance più elevate sono state raggiunte introducendo nei codici Assembly le ottimizzazioni precedentemente elencate. Nell' esecuzione della ricerca esaustiva sono stati usati i parametri di default, mentre nella ricerca non esaustiva sono stati modificati i valori di kc e nr come segue: kc=350, nr=400.






Tempo di Ricerca modalità Esaustiva
3
2,5
2
1,5
1
0,5
0
Ricerca pqnn32 Simmetrica
Ricerca pqnn32 Asimmetrica
Ricerca pqnn64 Simmetrica
Ricerca pqnn64 Asimmetrica
C
C + Assembly
C + Assembly ottimizzato


Tempo di ricerca modalità Non Esaustiva
8
7
6
5
4
3
2
1
0
Ricerca pqnn32 Simmetrica
Ricerca pqnn32 Asimmetrica
Ricerca pqnn64 Simmetrica
Ricerca pqnn64 Asimmetrica
C
C + Assembly
C + Assembly ottimizzato