

**Corso di Laurea Magistrale in Ingegneria Informatica**

**Corso di Architetture e Programmazione dei Sistemi di Elaborazione**

**Progetto A.A. 2018/2019**

**Algoritmo “*Product Quantization* for Nearest Neighbor Search” in linguaggio assembly x86-32+SSE e x86-64+AVX**

**Docente: Studenti:**

**Prof. Fabrizio Angiulli Cascardo Gianpaolo matr. 1111**

**Galati Davide matr.2222**

**Nicoletti Maria Chiara matr. 3333**

***Indice***

[1. Abstract 3](#_Toc11686371)

[2. Specifiche di Progetto 4](#_Toc11686372)

[2.1 Descrizione del Problema 4](#_Toc11686373)

[2.2 Descrizione dell’Attività Progettuale 8](#_Toc11686374)

[3. Scelte d’Implementazione e Fasi Progettuali 10](#_Toc11686375)

[3.1 Implementazione nel Linguaggio C 10](#_Toc11686376)

[3.2 Implementazione a basso livello 14](#_Toc11686377)

[3.2.1 Implementazione in x86-32 + SSE 15](#_Toc11686378)

[3.2.2 Ottimizzazione SSE 15](#_Toc11686379)

[3.2.3 Implementazione in X86-64+AVX 16](#_Toc11686380)

[3.3.4 Ottimizzazione AVX 16](#_Toc11686381)

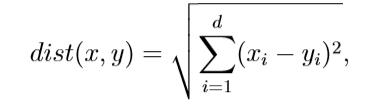
# Abstract

Lo scopo del progetto didattico assegnato è quello di realizzare una versione ottimizzata, attraverso l’utilizzo delle estensioni *SSE* e *AVX,* di un algoritmo di “Product Quantization for Nearest Neighbor Search”. L’idea alla base del funzionamento di tale algoritmo prevede di decifrare lo spazio in un piano cartesiano di sottospazi di bassa dimensione e di quantizzare ogni sottospazio separatamente. La realizzazione del progetto è stata suddivisa in più fasi, partendo dall’analisi del problema e concludendo con l’ottimizzazione ed il miglioramento del codice scritto. L’idea è stata, infatti, quella di separare la parte di implementazione da quella di ottimizzazione, scrivendo quindi inizialmente del codice funzionante senza badare troppo alle prestazioni. Dopodiché si è cercato di ridurre il numero di istruzioni utilizzate attraverso alcuni accorgimenti mirati al miglioramento delle prestazioni.

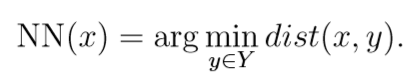
# Specifiche di Progetto

## 1.1 Descrizione del Problema

Dato un *dataset*, ovvero un insieme Y ⊂ di *n* vettori *d*-dimensionali (detti anche *punti*), ed un punto x ∈ (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:

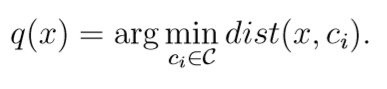


ovvero:

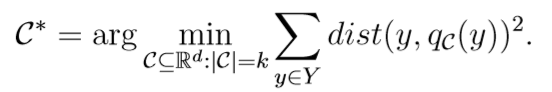


L’approccio brute force, che consiste nel calcolare la distanza tra xed ogni punto y di Y per poi restituire il punto y∗ avente distanza minima, rappresenta la tecnica più semplice per il calcolo del nearest neigbhor. Tale approccio ha complessità O(nd), dove n è il numero di punti in Y. Nonostante siano state proposte in letteratura diverse tecniche volte e ridurre tale complessità, all’aumentare della dimensionalità d dei dati tali tecniche degenerano in una scansione sequenziale di tutti i punti di Y, risultando quindi non più eﬃcienti dell’approccio brute force. Poiché nelle applicazioni reali il problema della ricerca del nearest neighbor richiede di lavorare con dataset enormi, composti da un numero n di punti dell’ordine delle centinaia di migliaia o dei milioni e da un numero di dimensioni d dell’ordine delle centinaia, nella pratica anche un algoritmo di costo lineare in n ed in d risulta insoddisfacente. Al ﬁne di alleviare tale problematica, si può far ricorso alla ricerca dell’approximate nearest neighbor (ANN). Una tecnica di ricerca dell’ANN è un algoritmo che 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. Naturalmente, per avere senso una tale tecnica deve presentare un costo temporale inferiore all’approccio brute force.

**Vector Quantization (VQ)**. Un codebook C è un insieme C = {} di k vettori d-dimensionali ∈, detti anche centroidi. Un quantizzatore vettoriale (VC), o solo quantizzatore per semplicità, q è una funzione che mappa ogni punto x ∈ in un centroide, ovvero q(x) ∈ C. In particolare, q (oppure 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:



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 (y), ovvero



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 centroidi {} 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.

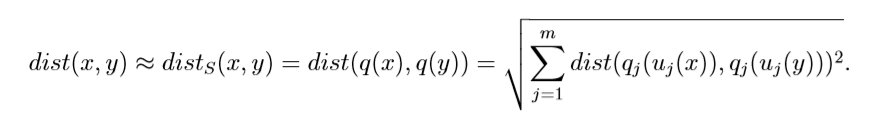
**Product Quantization (PQ**). Dato un dataset *Y* è di interesse riuscire a costruire un codebook C tale che, per ogni punto y di *Y*, y e (y) sono molto vicini. Purtroppo all’aumentare della dimensionalità d per ottenere una tale proprietà la dimensione k del codebook dev’essere esponenziale in d. 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 ∈ viene spezzato in m sottovettori a d∗ = d/m dimensioni. Nel seguito (x) denota il j-esimo (1 ≤ j ≤ 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, detti anche sotto-quantizzatori. La quantizzazione di x si ottiene quindi come segue:



Ovvero è data dalla concatenazione degli m centroidi d\*-dimensionali restituiti dagli m distinti quantizzatori applicati ognuno al rispettivo sotto-vettore (x), j = 1, 2, ..., m. Si assume che i codebook associati agli m quantizzatori , 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.

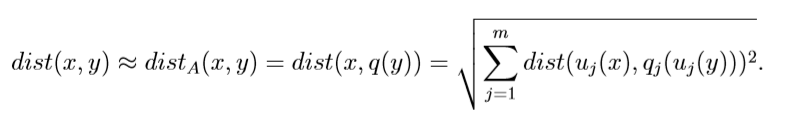
**Ricerca ANN esaustiva**. Utilizzando un PQ è possibile calcolare una distanza Euclidea approssimata tra il punto query x ed ogni punto y del dataset utilizzando due strategie: calcolo della distanza simmetrica (SDC) oppure calcolo della distanza asimmetrica (ADC).

La distanza simmetrica si ottiene come segue:



Poiché (x)), (y)) ∈, le distanze tra ogni coppia di centroidi c′, c′′∈, 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) anziché O(d).

La distanza asimmetrica si ottiene come segue:

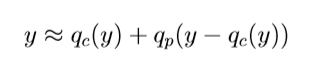


In questo caso, dato un punto query x, le distanze dist(*(*x)),(y))) tra x) ed ogni centroide c′ ∈ possono essere precalcolate e riutilizzate per calcolare la distanza asimmetrica ad un costo ridotto. La diﬀerenza tra le due soluzioni è che nel caso della distanza simmetrica le distanze precalcolate 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. Dato un parametro *K*, la ricerca restituisce i *K* punti del dataset che minimizzano la ADC oppure la SDC.

**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) (VQ) ed un quantizzatore prodotto (più accurato) (PQ). Il quantizzatore vettoriale utilizza centroidi d-dimensionali e viene utilizzato per deﬁnire il vettore dei residui:



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



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



Il quantizzatore è unico per tutti i punti del dataset ed i suoi centroidi vengono determinati facendo uso di un campione di ≤ n residui del dataset, ovvero ⊆ {r(y): y ∈ Y} e | | =. La tecnica di indicizzazione non esaustiva opera come segue: (i) si determinano i w centroidi grossolani ∈ che risultano essere più vicini alla query x; (ii) per ogni centroide determinato al passo (i) si calcolano le distanze approssimate sfruttando il quantizzatore prodotto— 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 (y) = 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. Per ulteriori dettagli si rimanda all’articolo [1] che descrive la tecnica di Product Quantization.

## Descrizione dell’Attività Progettuale

Obiettivo del progetto è mettere a punto un’implementazione dell’algoritmo PQNN in linguaggio C e di migliorarne le prestazioni utilizzando le tecniche di ottimizzazione basate sull’organizzazione dell’hardware. L’ambiente sw/hw di riferimento è costituito dal linguaggio di programmazione C (gcc), dal linguaggio assembly x86-32+SSE e dalla sua estensione x86-32+AVX (nasm) e dal sistema operativo Linux (ubuntu). In particolare il codice deve consentire di eﬀettuare sia la ricerca ANN esaustiva (parametro -exaustive, default) che la ricerca ANN non esaustiva (parametro -noexaustive). Inoltre, deve permettere di scegliere la distanza approssimata da utilizzare, ovvero la distanza simmetrica (parametro -sdc, default) oppure la distanza asimmetrica (parametro -adc). Inoltre il codice deve supportare i seguenti parametri:

-knn <K-value>: numero *K* di ANN approssimati restituiti per ogni query x (default *K* = 1);

-m <m-value>: numero m di gruppi utilizzati dal quantizzatore prodotto (PQ) (default m = 8);

-k <k\*-value>: numero di k\* centroidi utilizzati da ogni sotto-quantizzatore del quantizzatore prodotto (default k\* = 256);

-kc <kc-value>: numero kc di centroidi utilizzati dal quantizzatore coarse qc nel caso di ricerca non esaustiva (default kc = 8192);

-w <w-value>: numero w di centroidi del quantizzatore coarse qc da selezionare per eﬀettuare la ricerca non esaustiva (default w = 16);

-nr <nr-value>: dimensione nr (default nr = ⌊n/20⌋) del campione di residui utilizzati per costruire il quantizzatore prodotto qp utilizzato nella ricerca non esaustiva;

-kmeans <eps-value> [<tmin-value> [<tmax-value>]]: soglia ε(default ε= 0.01) per la terminazione dell’algoritmo *k*-means e numero minimo tmin (default tmin = 10) e massimo tmax (default tmax = 100) di iterazioni eseguite dall’algoritmo. Sia t il numero corrente di iterazioni dell’algoritmo *k*-means e sia ∆ la diﬀerenza relativa tra la sua funzione obiettivo valutata in due iterazioni successive. La condizione di terminazione da utilizzare è la seguente: (tmin ≤ t) AND (tmax < t) OR (∆ ≤ ε) .

Qualora un valore di un parametro (sia esso di default o speciﬁcato dall’utente) non sia applicabile, il codice deve segnalarlo con un messaggio e terminare.

Di seguito si riportano ulteriori linee guida per lo svolgimento del progetto:

• Si consiglia di aﬀrontare il progetto nel seguente modo:

1. Codiﬁcare l’algoritmo interamente in linguaggio C, possibilmente come sequenza di chiamate a funzioni;

2. Sostituire le funzioni scritte in linguaggio ad alto livello che necessitano di essere ottimizzate con corrispondenti funzioni scritte in linguaggio Assembly.

3. Ciò consentirà di veriﬁcare che l’algoritmo che si intende ottimizzare è corretto e di gestire più facilmente la complessità del progetto.

• Al ﬁne di migliorare la valutazione dell’attività progettuale è preferibile presentare nella relazione un confronto tra le prestazioni delle versioni intermedie, ognuna delle quali introduce una particolare ottimizzazione, e ﬁnale del codice. Obiettivo del confronto è sostanziare la bontà delle ottimizzazioni eﬀettuate.

• Occorre lavorare in autonomia e non collaborare con gli altri gruppi. Soluzioni troppo simili riceveranno una valutazione negativa. I progetti verranno messi in competizione.

• Sono richieste due soluzioni software, la prima per l’architettura x86-32+SSE e la seconda per l’architettura x86-64+AVX.

Per i dettagli riguardanti la redazione del codice fare riferimento ai ﬁles pqnn32c.c, pqnn32.nasm, runpqnn32 (versione x86-32+SSE) e pqnn64c.c, pqnn64.nasm, runpqnn64 (versione x86-64+AVX) disponibili sulla piattaforma didattica.dimes.unical.it.

Per l’interfacciamento tra programmi in linguaggio C e programmi in linguaggio assembly fare riferimento al documento allegato alla descrizione del progetto.

• Le soluzioni base non devono far uso delle istruzioni OpenMP. Opzionalmente, è possibile consegnare delle soluzioni aggiuntive che facciano uso anche di istruzioni OpenMP. I nomi dei relativi ﬁle dovranno contenere il suﬃsso “\_omp” (es. pqnn32c omp.c).

• Il software dovrà essere corredato da una relazione. Per la presentazione del progetto è possibile avvalersi di slide.

• Prima della data di consegna del progetto verranno pubblicate le convenzioni da rispettare riguardanti i nomi e la collocazione dei ﬁle/directory al ﬁne della compilazione e l’esecuzione dei codici di programma mediante script appositamente predisposti. **Dato l’elevato numero di progetti, sarà cura del candidato accertarsi di aver rispettato pienamente le convenzioni di consegna.**

# Scelte d’Implementazione e Fasi Progettuali

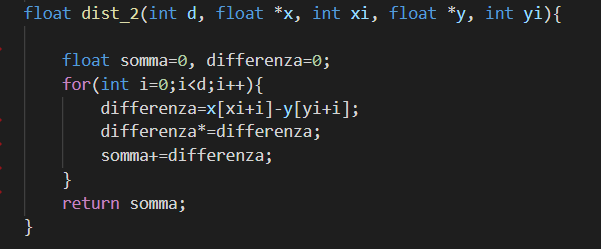
## 2.1 Implementazione nel Linguaggio C

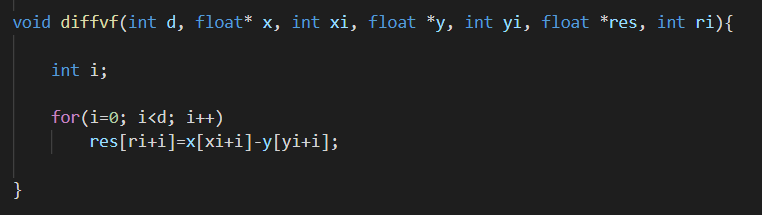
Al fine di agevolare lo svolgimento del lavoro progettuale, si è pensato di organizzarlo in quattro macro-fasi:

* **STUDIO DELLE TECNOLOGIE E DEL PROBLEMA:** questa fase è stata utilissima per prendere dimestichezza con gli strumenti di programmazione utilizzati e per comprendere al meglio il problema dell’algoritmo PQNN.
* **IMPLEMENTAZIONE AD ALTO LIVELLO:** durante la seconda fase del progetto si è pensato a come codificare il software in maniera modulare (ovvero come sequenza di chiamate a funzioni interamente in linguaggio C).
* **IMPLEMENTAZIONE AD BASSO LIVELLO:** sono stati implementati i vari moduli del software ad alto livello come funzioni Assembly x86-32 SSE ed x86-64 AVX.
* **VALUTAZIONE DELLE PRESTAZIONI E POSSIBILI MIGLIORAMENTI:** è l’ultima fase progettuale durante la quale sono state analizzate le prestazioni dei vari metodi utilizzati, e sono stati apportati degli ulteriori piccoli miglioramenti in termini di efficienza dell’algoritmo.

A livello pratico, quindi, per la realizzazione del progetto didattico assegnato, il primo passo da seguire è stata la configurazione dell’ambiente di sviluppo VISUAL STUDIO CODE con il relativo DEBUG. Come è stato anche indicato dalle specifiche progettuali, si è proceduto alla scrittura dell’Algoritmo Product Quantization for Nearest Neighbor Search in un linguaggio ad alto livello. La scelta è ricaduta sul linguaggio C, un linguaggio ad “alto livello sì, ma non troppo”, poiché consente, in maniera abbastanza naturale e semplice, l’interfacciamento con le funzioni Assembly. Il lavoro progettuale è iniziato con l’allocazione in memoria del dataset – ovvero un insieme di n vettori d- dimensionali (detti anche punti), ed un punto x (detto anche query o interrogazione)**.** Soltanto dopo aver pianificato ciò, in seguito si è potuto passare alla realizzazione dei seguenti metodi, o meglio, all’implementazione delle seguenti funzioni C:

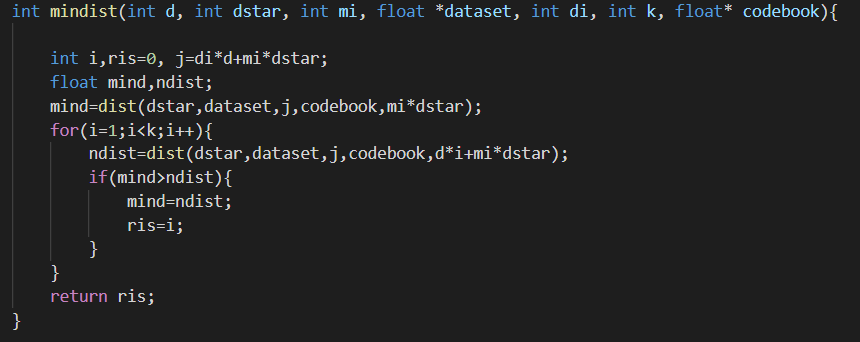
* float **dist\_2** (int d, float \*x, int xi, float \*y, int yi):

calcola la distanza Euclidea al quadrato tra un punto di un primo vettore x (segnato a partire da un indice xi) ed un punto di un secondo vettore y (segnato a partire da un indice yi). A partire dai 2 indici “xi”, “yi” vengono analizzati “d” elementi.

* void **diffvf** (int d, float \*x, int xi, float \*y, int yi, float \*res, int ri):

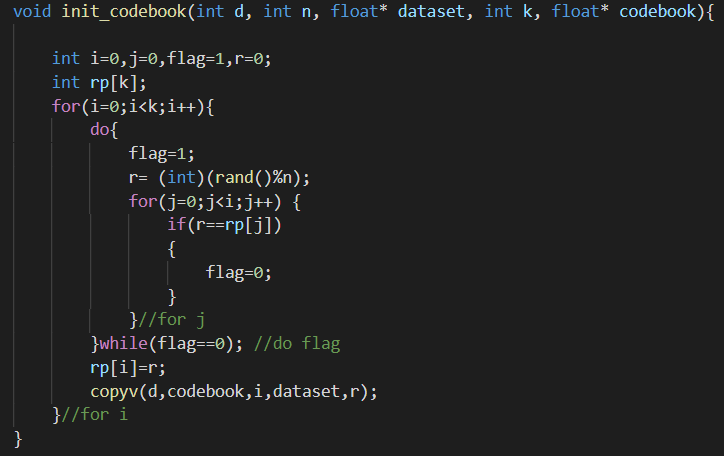
è il metodo che sottrae gli elementi del vettore x dagli elementi del vettore y. I risultati verranno memorizzati nel vettore “res”.

* int **mindist** (int d, int dstar, int mi, float \*dataset, int di, int k, float \*codebook,):

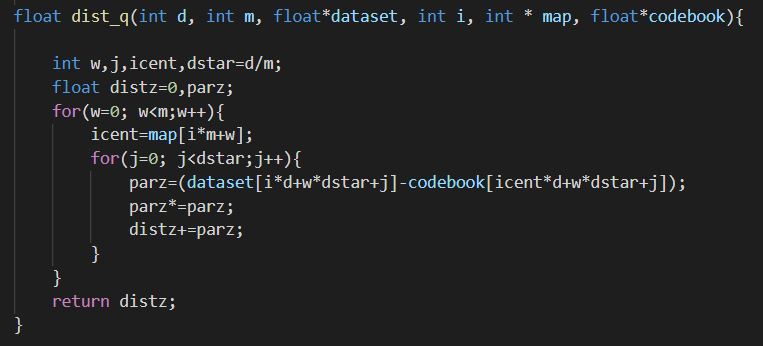
considerato il punto indicizzato nel dataset a partire dall’indice di (in R^d) si cerca un centroide nel codebook (tra k centroidi) tale da minimizzare la distanza. Ogni distanza viene calcolata con la funzione dist, e funziona sia con i sottovettori che non.

* void **init\_codebook** (int d, int n, float \*dataset, int k, float \*codebook,):

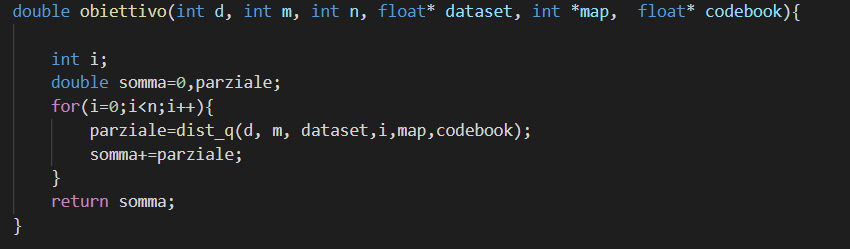
si riempie il codebook con valori random facenti parte del dataset di partenza.

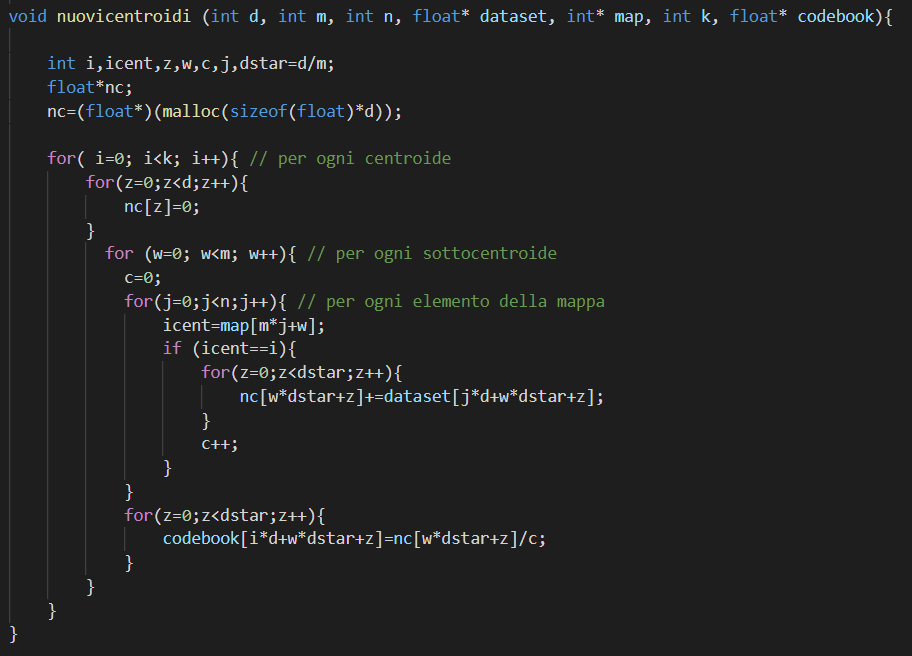


* float **dist\_q** (int d, int m, float \*dataset, int i, int \*map, float \*codebook,):

metodo che calcola la distanza al quadrato tra un punto ed i suoi centroidi.

* void **obiettivo** (int d, int m, int n, float \*dataset, int \*map, float \*codebook,):

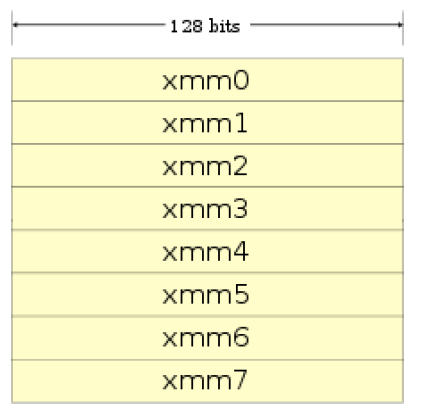
calcola il valore della funzione obiettivo attuale come la somma delle distanze al quadrato di ogni punto dal suo centroide.

* void **nuovicentroidi** (int d, int m, int n, float \*dataset, int \*map, int k, float \*codebook,):

è il metodo che genera un set di nuovi centroidi calcolando la media degli elementi che gli sono associati.

## Implementazione a basso livello

Le architetture odierne e l’organizzazione dell’hardware consentono di ottimizzare i calcoli permettendo l’elaborazione in contemporanea di più elementi mediante la cosìddetta “*vettorizzazione*”. Più in particolare, i microprocessori Intel permettono la vettorizzazione attraverso due set di istruzioni **SIMD** (Single Instruction Multiple Data):

* le **SSE (**Single SIMD Extension): sono un insieme di istruzioni progettate ed introdotte da Intel per la sua architettura x86 nel 1999 ed utilizzate nel suo processore Pentium 3.Il primo tentativo di SIMD da parte della Intel, la tecnologia MMX, fu una delusione. MMX presenta, infatti, due seri problemi: riutilizza i registri a virgola mobile rendendo impossibile per la CPU lavorare sui dati in virgola mobile e i dati SIMD contemporaneamente, ma può operare solo sugli interi. SSE aggiunge otto nuovi registri a 128 bit con nomi che vanno da XMM0 a XMM7. Ogni registro raggruppa quattro numeri a virgola mobile a 32 bit (precisione singola). Ogni singolo registro è in grado di contenere insieme:
* quattro numeri in virgola mobile in singola precisione a 32 bit;
* quattro interi a 32 bit;
* due numeri in virgola mobile in doppia precisione a 64 bit;
* due interi a 64 bit;
* otto interi da 16 bit;
* sedici byte da 8 bit.

SSE inserisce nel set di istruzioni sia operazioni su scalari singoli, sia su gruppi di numeri in virgola mobile (packed). Infatti, tra le varie istruzioni troviamo, ad esempio, quelle per i movimenti memoria-registro/ registro-memoria/ registro-registro (MOVAPS, MOVUPS, MOVLPS, MOVHPS…) oppure nel caso di scalari (MOVSS); per il calcolo aritmetico su questi numeri (come ad esempio: ADDPS, MULPS, SUBPS, DIVPS, SQRTPS…) o nel caso di singoli elementi (ADDSS, MULSS, SUBSS…).

* le **AVX** (Advanced Vector Extension): come lascia intendere l’acronimo, rappresentano un’estensione delle SSE e sono più recenti. Sono state infatti introdotte nel 2011. Il banco dei registri è stato ampliato con l’aggiunta di ulteriori 8 registri a 256 bit, mentre i precedenti registri xmm sono stati estesi a 256 bit. In totale, quindi, le AVX possono contare su 16 registri a 256 bit (ymm0 - ymm15) in grado di processare in contemporanea 8 numeri floating point a 32 bit oppure 4 numeri floating point a precisione doppia. Grazie a questa novità si riesce ad ottenere un raddoppio dei calcoli in virgola mobile ed a migliorare l’efficienza dell’organizzazione dei dati, rendendola più efficiente. Va, comunque, menzionato il fatto che i vecchi registri xmm possono essere utilizzati nelle istruzioni AVX poiché sono in aliasing con i 128 bit meno significativi dei nuovi registri ymm. Un’altra caratteristica peculiare del nuovo set di istruzioni AVX riguarda il fatto che le istruzioni possono prendere fino a 3 operandi, cioè una destinazione e due sorgenti; questo consente di aumentare di molto le possibilità e gli ambiti di utilizzo. Le applicazioni che dovrebbero trarre i maggiori benefici dovrebbero essere quelle di tipo multimediale, in particolare quelle di modellazioni 3D e di calcolo scientifico. La maggior parte delle istruzioni SSE possono essere riutilizzate anche in ambito AVX semplicemente anteponendo una V allo mnemonico utilizzato. Per esempio, VMOVUPS è l’istruzione corrispondente in AVX dell’istruzione SSE MOVUPS. La traccia del progetto assegnato prevedeva di sviluppare due versioni dello stesso algoritmo, una per architettura x86-32 avvalendosi delle istruzioni SSE, ed una variante per x86-64 con l’ausilio delle istruzioni AVX.

### Implementazione in x86-32 + SSE

Una volta conclusa, questa prima parte di progettazione ad alto livello dell’algoritmo, il progetto assegnato prevedeva una codifica sempre dello stesso algoritmo in un linguaggio a più basso livello di astrazione. A tal fine, si è ritenuto opportuno progettare alcune delle chiamate a funzione del linguaggio C in Assembly, rendendo così l’algoritmo facile da verificare, da correggere nonché da ottimizzare, gestendo al meglio la complessità. Nella versione a 32 bit, si è proceduto implementando alcuni dei metodi presenti in pqnn32c.c in nasm; nello specifico le funzioni sulle quali si è lavorato sono le seguenti:

* dist\_2

### Ottimizzazione SSE

Per ottimizzare il codice Assembly, sono state utilizzate due tecniche di ottimizzazione:

* *Code Vectorization:* questa tecnica consente di effettuare in parallelo la stessa operazione su diversi elementi di un vettore. Nello specifico, la vettorizzazione nel nostro codice prevede di effettuare la stessa operazione su 4 elementi contemporaneamente; qualora non fossero presenti 4 elementi, l’operazione sarà effettuata elemento per elemento.
* *Loop Unrolling:* anche detto *“srotolamento”,* consiste nel combinare più iterazioni consecutive di un ciclo in una sola iterazione riducendone il numero e, quindi, anche le volte che le istruzioni di controllo del ciclo vengono eseguite.

Per la versione a 32 bit il fattore di Unroll è stato posto pari a…, poiché provando ad aumentarlo le prestazioni del codice….

### 3.2.3 Implementazione in X86-64+AVX

La fase progettuale successiva ha previsto la progettazione dell’algoritmo PQNN nella versione estesa di SSE, ovvero in x86-64+AVX. Secondo una prima occhiata, abbastanza superficiale, potrebbe sembrare una semplice traduzione della versione a32 bit ed, infatti, concettualmente lo è. La sostanziale differenza risiede nella numerosità dei registri, aspetto questo da non sottovalutare visto che potendo utilizzare registri a 64 bit, permette di poter lavorare con più dati rispetto alla versione a 32 bit. L’implementazione in Assembly 64 bit segue, quindi, la stessa logica con i registri generali “*eax, ebx, ecx, edx, esi, edi*” che vengono sostituiti con i corrispondenti registri “*rax, rbx, rcx, rdx, rsi, rdi”* a cui si aggiungono altri 4 registri generali (*r10, r11, r12, r13*).

### 3.3.4 Ottimizzazione AVX

Arrivati a questo punto, dopo aver scritto le funzioni C in Assembly, dopo aver ottimizzato la sua versione a 32 bit, tradotte a 64 bit, si è cercato di ottimizzare anche il codice AVX. Le tecniche che sono state utilizzate sono sempre 2 e sono:

* *Code Vectorization:*