UNIVERSITÀ DELLA CALABRIA
DIPARTIMENTO DI
INGEGNERIA INFORMATICA,
MODELLISTICA, ELETTRONICA
E SISTEMISTICA
DIMES

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:

Fabrizio Angiulli

Studenti:

Gianpaolo Cascardo 189088
Davide Galati 189222
Maria Chiara Nicoletti 166773

Indice generale

Scopo e specifiche del progetto

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 la proiezione dello spazio iniziale del dataset in sottospazi a dimensione ridotta e processarne ognuno 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 dapprima una versione funzionante del software, cercando in seguito di ridurne la complessità utilizzando alcuni accorgimenti mirati al miglioramento delle prestazioni.

Descrizione del problema

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

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

ovvero:

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

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 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 $C = \{c_1, c_2, \ldots, c_k\}$ di *k* vettori *d*-dimensionali $c_i \in \mathbb{R}^d$, detti anche *centroidi*. Un *quantizzatore vettoriale* (VC), o solo *quantizzatore* per semplicità, *q* è una funzione che mappa ogni punto $x \in \mathbb{R}^d$ in un *centroide*, ovvero $q(x) \in C$. 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

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

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 centroidi $\{c_1, c_2, \ldots, c_k\}$ 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.

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 $x \in \mathbb{R}^d$ viene spezzato in *m* sottovettori a $d^* = d/m$ dimensioni. Nel seguito $u_j(x)$ denota il j-esimo ($1 \le j \le m$) 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**-dimensionali restituiti dagli *m* distinti quantizzatori q_j applicati ognuno al rispettivo sotto-vettore $u_j(x)$, $j = 1,2,\ldots,m$.

Si assume che i codebook $C_1, C_2, \ldots, C_m$, associati agli *m* quantizzatori $q_1, q_2, \ldots q_m$, 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).

La *distanza simmetrica* si ottiene come segue:

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

Poiché $q_j(u_j(x)), q_j(u_j(y)) \in C_j$, 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 $O(m)$ anziché $O(d)$.

La *distanza asimmetrica* si ottiene come segue:

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

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 asimmetrica 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) 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*

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

corrispondente al vettore y nel caso di origine dello spazio coincidente con il suo centroide q_c, 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 e y può essere approssimata come segue:

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

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.

1 Scelte d'implementazione e fasi progettuali

1.1 Progettazione ed implementazione

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 funzionamento 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 A 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:** 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, per la realizzazione del progetto didattico assegnato, il primo passo da seguire è stata la configurazione dell'ambiente di sviluppo **Visual Studio Code** utilizzando **cgdb** come debugger.

Il lavoro progettuale è iniziato con la rappresentazione in memoria del dataset. Si è scelto infatti di memorizzare il singolo punto in un array d-dimensionale e l'intero set come una concatenazione di n punti consecutivi. Similmente sono stati rappresentati i punti del codebook (centroidi) e quelli del queryset (le interrogazioni)**.**

Seguirà, in maniera più dettagliata, una serie di scelte implementative in relazione anche a quelle che sono le specifiche progettuali.

- Le coordinate dei punti all'interno dei file sono memorizzate come numeri float. Per continuità si è deciso di utilizzare matrici di float per rappresentarle. Gli indici degli

ANN sono memorizzati con vettori di interi, mentre le distanze come double per garantire una maggiore precisione dei risultati.

- Al fine di facilitare la trasposizione del codice in assembly, tutte le matrici (quadrate e cubiche) sono state rappresentate come array. Si è ritenuto opportuno concatenare le righe linearmente (*row-major order*) data la natura del problema.
- Si è deciso di trasporre piccole parti di codice in assembly per evitare l'accrescere di interazioni con la memoria
- Eccetto che in un caso, si è deciso di non far dipendere nessun metodo dalle strutture del codice di avvio (in modo da rendere più modulare possibile il codice)

L'algoritmo si divide in due macro-fasi: in un primo passo bisogna proiettare lo spazio dei punti in sottospazi e raggrupparli intorno a dei particolari punti detti *centroidi*. Questa operazione è detta **indexing** e più formalmente crea le così dette celle di Voronoi attraverso le quali il calcolo viene poi suddiviso.

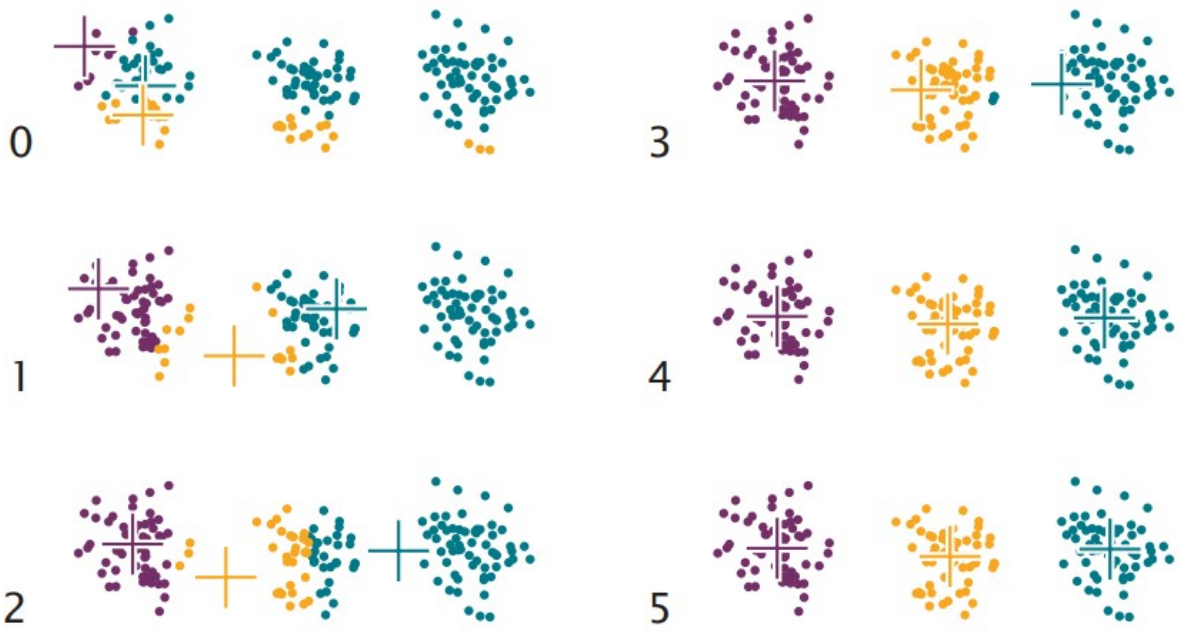


Figura 1: Rappresentazione grafica dell'algoritmo di accentramento (clustering)

Successivamente, con l'ausilio della suddivisione ottenuta, si ricercano i punti del dataset più vicini a quelli del queryset scegliendo una tra quattro possibili strategie: **ricerca simmetrica esaustiva** (SDC exaustive), **ricerca asimmetrica esaustiva** (ADC exaustive), **ricerca simmetrica non esaustiva** (SDC no exaustive) e **ricerca asimmetrica non esaustiva** (ADC no exaustive); questa fase è detta di **searching**.

1.2 Indexing

Inizialmente si selezionano i primi *k* punti del dataset come centroidi del codebook. Tra le possibilità vi era quella di scegliere casualmente questi punti; tuttavia non influenzando il risultato, nei test svolti, si è deciso di optare per un metodo che non pesasse sui tempi di esecuzione dell'algoritmo. Questa fase è implementata nella funzione:

```
void init_codebook(int d, int n, float* dataset, int k, float* codebook)
```

Il passo successivo prevede il perfezionamento dei punti del codebook, calcolati come media aritmetica dei punti associati come previsto dall'algoritmo

```
void k_means( int d, int m, float eps, int tmin, int tmax, int k, float*
              codebook, int n, float* dataset, int* map)
```

all'interno del quale ogni punto viene visto come insieme di *m* sotto punti, ai quali viene ripetutamente riassegnato il centroide più vicino attraverso la funzione

```
void pq(int d, int m, int k, float *codebook, int n, float *dataset, int*map)
```

che genera una corrisponenza punto-centroide memorizzato nella matrice di indici *map* (con un numero di righe pari al dataset).

È possibile variare *m* al fine di simulare il comportamento dell'algoritmo **vector quantization** (vq).

All'interno delle funzioni sopracitate vengono utilizzate anche:

```
-la macro KMEANS_STEP(d,m,n,dataset,map,k,codebook) che chiama nuovicentroidi
-void nuovicentroidi (int d,int m,int n, loat* dataset,int* map,int k,float* codebook)
-double obiettivo(int d, int m, int n, float* dataset, int *map,  float* codebook)
-int mindist(int d, int dstar, int mi, float *dataset, int di, int k, float* codebook)
```

```
-la macro DIST_E_2(D,X,XI,Y,YI,RIS)
```

In mindist lo scopo è quello di determinare l'indice del centroide più vicino ad un punto specifico del dataset calcolandone la distanza minima. Il calcolo viene rimandato alla macro DIST_E_2, che in assenza dell'implementazione in assembly viene estesa in:

```
float somma=0
    differenza=0;
    for(register int i=0;i<D;i++){
            differenza=X[(XI)+i]-Y[(YI)+i];
            differenza*=differenza;
            somma+=differenza;
    }
RIS=somma;
```

l'utilizzo della macro permette di intercambiare velocemente le varie versioni del codice. Questo approccio è supportato anche dalla definizione a tempo di compilazione, come parametro di **GCC**, della variabile **ASM**.

In caso di ricerca *non esaustiva* l'indicizzazione fa riferimento ad un dataset, generalmente ridotto, composto da *nr* punti. Inoltre, viene aggiunto un ulteriore codebook composto da centroidi *coarse*. Vengono create ulteriori mappe sui residui, associandoli ai codebook prima sopracitati. In questa fase viene richiamata la funzione:

```
-void centroidi_associati(int d, int w, float* qs,int ix, int k, float*codebook, int*mapxw)
```

che associa ad ogni punto query un insieme di *w* centroidi su cui si baserà poi la fase di searching

1.3 Searching

La ricerca avviene minimizzando la somma delle distanze al quadrato. In caso di quella **simmetrica** questo confronto si basa sulla distanza tra i centroidi associati al queryset e

quelli associati al dataset. Quella **asimmetrica,** invece, la calcola tra i punti del queryset e i centroidi associati al dataset.

Di seguito sono riportate le funzioni utilizzate per implementare le due ricerche nel caso **esaustivo**:

```
-void ANNSDC(int d, int m, int k, float* codebook, int K, int*ANN, double* ANN_values,
          int n, int*map, int nq, float*qs)
-void ANNADC(int d, int m, int k, float* codebook, int K, int*ANN, double* ANN_values,
          int n, int*map, int nq, float*qs)
```

Internamente sono state richiamate ulteriori funzioni che hanno permesso di suddividere il lavoro svolto:

`-void ADC (int d, int k, int m, double* distanze, int K, int*ANN, double* ANN_values,int n, int* map, int ix, float *qs)` richiamata da `ANNADC` che, fissato un punto query, cerca i suoi vicini nel dataset.

`-void SDC (int d, int k, int m, int nrd, double* distanze, int K, int*ANN, double* ANN_values, int n, int* map, int ix, int *qx)` richiamata da `ANNSDC` che, fissato un punto query, cerca i suoi vicini nel dataset.

`-void mergeSort(double* values, int* indices, int start, int end, int offset)` richiamata in entrambi i casi al termine della ricerca operando in ordine crescente.

Per quanto riguarda le strutture dati di supporto, è stato necessario precalcolare le distanze in due modi diversi in base al tipo di ricerca utilizzata:

- nel caso della ricerca **SDC** si è utilizzata una matrice cubica $k \times k \times m$. All'indice `(i,j,w)` corrisponde la distanza tra il centroide *i* e il centroide *j* nel sottospazio *w.*
- nel caso della ricerca **ADC** si è utilizzata una matrice quadrata $k \times m$ precalcolata per ogni punto del queryset. All'indice `(i,w)` corrisponde la distanza tra il punto e il centroide *i* nel sottospazio *w.*

In caso di ricerca **non esaustiva** si utilizza, una delle due tecniche sopracitate basandosi sul confronto tra i residui dei punti. La ricerca si limita ad un sottoinsieme di centroidi considerati più vicini al punto del queryset.

Non sono stati citati, ma sono comunque utilizzati, funzioni di minore rilevanza che svolgono copia, differenza ed altre operazioni tra vettori o matrici.

2 Ottimizzazione a basso livello, prestazioni e tempi

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.

2.1 Progettazione in assembly

Una volta conclusa, questa prima parte di progettazione ad alto livello dell'algoritmo, il progetto assegnato prevedeva una codifica sempre dello stesso algoritmo in assembly. Nella traduzione del codice si è ritenuto opportuno considerare le fuzioni più volte richiamate nel flusso dell'algoritmo. La scelta è pertanto ricaduta su:

-`DIST_E_2(D,X,XI,Y,YI,RIS)` tradotto nella procedura `float dist_2_asm(int d,float *x,int xi,float *y,int yi)`

-`DIFFVF(D,X,XI,Y,YI,RES,RI)` tradotto nella procedura `void diffvf_asm(int d,float *x,int xi,float *y,int yi,float * res,int ri)`

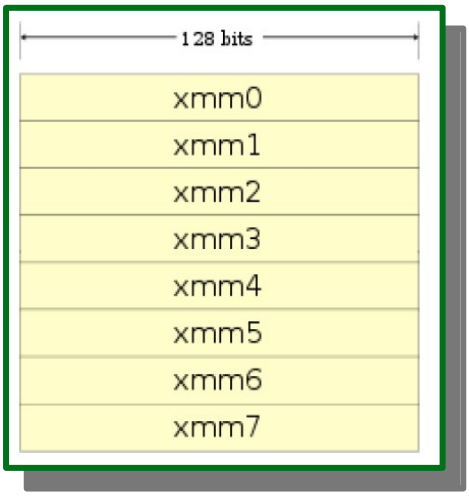
-`INIT_ARRAY(D,A,AI,IV)` tradotto nella procedura `void azzera_array(int d,float*v)`

-`COPYV(D,DEST,DESTI,SRC,SRCI)` tradotto nella procedura `void copyv_asm(int d,float* dest,int desti,float *src,int srci)`

Si noti che grazie all'impostazione modulare progettata, è stato possibile scrivere un solo file C senza cambiare l'intestazione delle procedure assembly al variare delle due implementazioni a 32 o 64 bit. Per sfruttare al massimo questo meccanismo si è fatto uso di macro e diversi parametri passati a **GCC**.

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

- *Code Vectorization*: tecnica SIMD che 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 questa fase si è deciso di strutturare più livelli di *unrolling* fissando per ognuno un fattore diverso. Nel caso generico avremo che una prima fase replica le operazioni 32 volte, seguita da una replica di 8 ed infine 2. Successivamente viene utilizzato un approccio *scalare*, terminando quindi anche la vectorization.

2.2 Implementazione in assembly

La fase di implementazione è stata una traduzione uno ad uno del codice C a cui sono stati applicati i principi di cui sopra. Per semplificare i processi e suddividere il codice in maniera più chiara ed efficiente, si è pensato di scrivere delle macro che racchiudessero il cuore di ogni procedura.

Segue un esempio estratto dai file asm a 32 bit della traduzione della funzione che calcola la distanza euclidea al quadrato:

```
dist_2_asm:
    ; ... inizializzazioni ...
    mov   esi, ebx ;esi=d
    sub   esi, 4*R ;esi=d-p*r
    inc   esi      ;esi=d-p*r+1

LOOPRDIST:
    cmp edi, esi
    jge LOOPSDIST

    dist_2_step
    ; ...
    ; copia r volte
    ; ...
    dist_2_step

    jmp     LOOPRDIST

LOOPSDIST:
    cmp edi, ebx
    jge ENDDIST
    mov      edx, [ebp+16]
    add      edx, edi
    movss   xmm1, [eax+edx*4]
    mov     edx, [ebp+24]
    add     edx, edi
    subss  xmm1, [ecx+edx*4]
    mulss   xmm1,xmm1
    addss   xmm0,xmm1
    inc edi
    jmp LOOPSDIST

ENDDIST:
    haddps xmm0, xmm0
    haddps xmm0, xmm0
    movd [ebp-4], xmm0
    fld dword [ebp-4]
```

```
%macro dist_2_step 0
    mov      edx, [ebp+16]
    add      edx, edi
    movups   xmm1, [eax+edx*4]
    mov      edx, [ebp+24]
    add      edx, edi
    subps    xmm1, [ecx+edx*4]
    mulps    xmm1,xmm1
    addps    xmm0,xmm1
    add      edi, 4
%endmacro
```

Quanto descritto sopra è stato applicato anche all'implementazione a 64 bit.

2.3 Tempi e prestazioni

Si è ritenuto interessante mettere a confronto i tempi di esecuzione del software eseguito con la sola implementazione in c, con assembly a 32 bit e con assembly a 64 bit. Verranno riportati anche risultati ottenuti attraverso varie sperimentazioni (variando il fattore di unrolling o abilitando e disabilitando l'implementazione in assembly di alcune funzioni anziché altre). Si noti che i risultati ottenuti in termini di output (indici ANN e relative distanze) non cambiano al variare dell'architettura utilizzata ma solo della tecnica applicata per calcolarli.

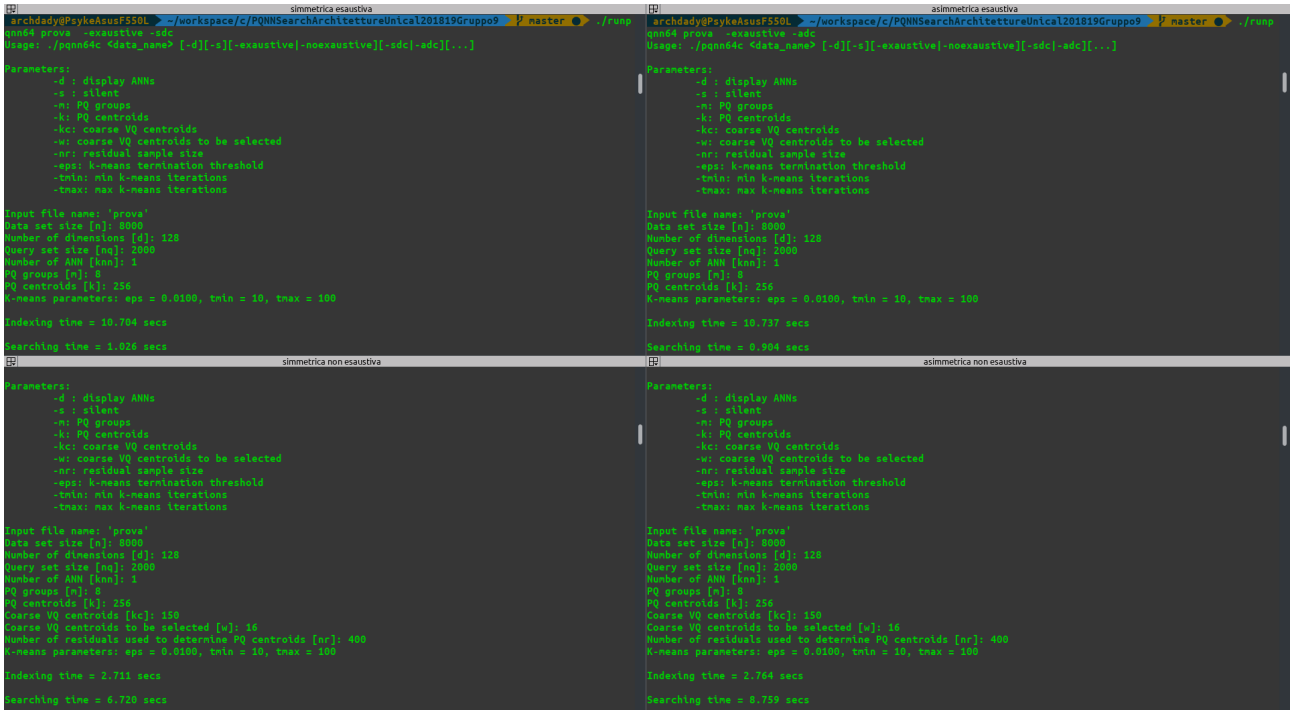


Figura 2: tempi esecuzione file C

I tempi sono suddivisi in 4 quadranti, ognuno riporta i risultati ottenuti con una metodologia tra quelle precedentemente descritte. In particolare:

- in alto a sinistra si è fatto uso della tecnica **esaustiva SDC**
- in alto a destra si è fatto uso della tecnica **esaustiva ADC**
- in basso a sinistra si è fatto uso della tecnica **non esaustiva SDC**
- in basso a destra si è fatto uso della tecnica **non esaustiva ADC**

Quanto detto sopra è valido anche nelle immagini che seguono:

simmetrica esaustiva

```
Usage: ./pqnn32c <data_name> [-d][-s][-exaustive|-noexaustive][-sdc|-adc][...]

Parameters:
        -d : display ANNs
        -s : silent
        -m: PQ groups
        -k: PQ centroids
        -kc: coarse VQ centroids
        -w: coarse VQ centroids to be selected
        -nr: residual sample size
        -eps: k-means termination threshold
        -tmin: min k-means iterations
        -tmax: max k-means iterations

Input file name: 'prova'
Data set size [n]: 8000
Number of dimensions [d]: 128
Query set size [nq]: 2000
Number of ANN [knn]: 1
PQ groups [m]: 8
PQ centroids [k]: 256
K-means parameters: eps = 0.0100, tmin = 10, tmax = 100

Indexing time = 3.087 secs

Searching time = 1.083 secs
```

asimmetrica esaustiva

```
Usage: ./pqnn32c <data_name> [-d][-s][-exaustive|-noexaustive][-sdc|-adc][...]

Parameters:
        -d : display ANNs
        -s : silent
        -m: PQ groups
        -k: PQ centroids
        -kc: coarse VQ centroids
        -w: coarse VQ centroids to be selected
        -nr: residual sample size
        -eps: k-means termination threshold
        -tmin: min k-means iterations
        -tmax: max k-means iterations

Input file name: 'prova'
Data set size [n]: 8000
Number of dimensions [d]: 128
Query set size [nq]: 2000
Number of ANN [knn]: 1
PQ groups [m]: 8
PQ centroids [k]: 256
K-means parameters: eps = 0.0100, tmin = 10, tmax = 100

Indexing time = 3.146 secs

Searching time = 0.949 secs
```

simmetrica non esaustiva

```
Usage: ./pqnn32c <data_name> [-d][-s][-exaustive|-noexaustive][-sdc|-adc][...]

Parameters:
        -d : display ANNs
        -s : silent
        -m: PQ groups
        -k: PQ centroids
        -kc: coarse VQ centroids
        -w: coarse VQ centroids to be selected
        -nr: residual sample size
        -eps: k-means termination threshold
        -tmin: min k-means iterations
        -tmax: max k-means iterations

Input file name: 'prova'
Data set size [n]: 8000
Number of dimensions [d]: 128
Query set size [nq]: 2000
Number of ANN [knn]: 1
PQ groups [m]: 8
PQ centroids [k]: 256
Coarse VQ centroids [kc]: 150
Coarse VQ centroids to be selected [w]: 16
Number of residuals used to determine PQ centroids [nr]: 400
K-means parameters: eps = 0.0100, tmin = 10, tmax = 100

Indexing time = 0.507 secs

Searching time = 3.575 secs
```

asimmetrica non esaustiva

```
Usage: ./pqnn32c <data_name> [-d][-s][-exaustive|-noexaustive][-sdc|-adc][...]

Parameters:
        -d : display ANNs
        -s : silent
        -m: PQ groups
        -k: PQ centroids
        -kc: coarse VQ centroids
        -w: coarse VQ centroids to be selected
        -nr: residual sample size
        -eps: k-means termination threshold
        -tmin: min k-means iterations
        -tmax: max k-means iterations

Input file name: 'prova'
Data set size [n]: 8000
Number of dimensions [d]: 128
Query set size [nq]: 2000
Number of ANN [knn]: 1
PQ groups [m]: 8
PQ centroids [k]: 256
Coarse VQ centroids [kc]: 150
Coarse VQ centroids to be selected [w]: 16
Number of residuals used to determine PQ centroids [nr]: 400
K-means parameters: eps = 0.0100, tmin = 10, tmax = 100

Indexing time = 0.515 secs

Searching time = 3.242 secs
```

Figura 3: tempi esecuzione file asm 32bit

simmetrica esaustiva

```
Usage: ./pqnn64c <data_name> [-d][-s][-exaustive|-noexaustive][-sdc|-adc][...]

Parameters:
        -d : display ANNs
        -s : silent
        -m: PQ groups
        -k: PQ centroids
        -kc: coarse VQ centroids
        -w: coarse VQ centroids to be selected
        -nr: residual sample size
        -eps: k-means termination threshold
        -tmin: min k-means iterations
        -tmax: max k-means iterations

Input file name: 'prova'
Data set size [n]: 8000
Number of dimensions [d]: 128
Query set size [nq]: 2000
Number of ANN [knn]: 1
PQ groups [m]: 8
PQ centroids [k]: 256
K-means parameters: eps = 0.0100, tmin = 10, tmax = 100

Indexing time = 2.503 secs

Searching time = 1.922 secs
```

asimmetrica esaustiva

```
Usage: ./pqnn64c <data_name> [-d][-s][-exaustive|-noexaustive][-sdc|-adc][...]

Parameters:
        -d : display ANNs
        -s : silent
        -m: PQ groups
        -k: PQ centroids
        -kc: coarse VQ centroids
        -w: coarse VQ centroids to be selected
        -nr: residual sample size
        -eps: k-means termination threshold
        -tmin: min k-means iterations
        -tmax: max k-means iterations

Input file name: 'prova'
Data set size [n]: 8000
Number of dimensions [d]: 128
Query set size [nq]: 2000
Number of ANN [knn]: 1
PQ groups [m]: 8
PQ centroids [k]: 256
K-means parameters: eps = 0.0100, tmin = 10, tmax = 100

Indexing time = 2.556 secs

Searching time = 1.754 secs
```

simmetrica non esaustiva

```
npqnn64 prova  -noexaustive -sdc
Usage: ./pqnn64c <data_name> [-d][-s][-exaustive|-noexaustive][-sdc|-adc][...]

Parameters:
        -d : display ANNs
        -s : silent
        -m: PQ groups
        -k: PQ centroids
        -kc: coarse VQ centroids
        -w: coarse VQ centroids to be selected
        -nr: residual sample size
        -eps: k-means termination threshold
        -tmin: min k-means iterations
        -tmax: max k-means iterations

Input file name: 'prova'
Data set size [n]: 8000
Number of dimensions [d]: 128
Query set size [nq]: 2000
Number of ANN [knn]: 1
PQ groups [m]: 8
PQ centroids [k]: 256
Coarse VQ centroids [kc]: 150
Coarse VQ centroids to be selected [w]: 16
Number of residuals used to determine PQ centroids [nr]: 400
K-means parameters: eps = 0.0100, tmin = 10, tmax = 100

Indexing time = 0.415 secs

Searching time = 4.570 secs
```

asimmetrica non esaustiva

```
npqnn64 prova  -noexaustive -adc
Usage: ./pqnn64c <data_name> [-d][-s][-exaustive|-noexaustive][-sdc|-adc][...]

Parameters:
        -d : display ANNs
        -s : silent
        -m: PQ groups
        -k: PQ centroids
        -kc: coarse VQ centroids
        -w: coarse VQ centroids to be selected
        -nr: residual sample size
        -eps: k-means termination threshold
        -tmin: min k-means iterations
        -tmax: max k-means iterations

Input file name: 'prova'
Data set size [n]: 8000
Number of dimensions [d]: 128
Query set size [nq]: 2000
Number of ANN [knn]: 1
PQ groups [m]: 8
PQ centroids [k]: 256
Coarse VQ centroids [kc]: 150
Coarse VQ centroids to be selected [w]: 16
Number of residuals used to determine PQ centroids [nr]: 400
K-means parameters: eps = 0.0100, tmin = 10, tmax = 100

Indexing time = 0.416 secs

Searching time = 5.032 secs
```

Figura 4: tempi esecuzione file asm 64 bit

3 Considerazioni e conclusioni

Tra i test effettuati si riportano anche le seguenti considerazioni:

- Oltre un certo fattore di unrolling le prestazioni non sembrano subire variazioni significative. Questo certamente è una considerazione che andrebbe però rivista al variare della dimensione dell'input.
- Non tutte le traduzioni in assembly hanno influenzato le prestazioni in fase di esecuzione. In particolare si è notato che la funzione con maggiore peso risulta essere quella che calcola la distanza euclidea tra due punti.

Si vuole concludere sottolineando l'importanza di valutare delle implementazioni a basso livello per ridurre i tempi di esecuzione di algoritmi strutturalmente più complessi.

3.1 Crediti, contatti e riferimenti

Il progetto è stato interamente sviluppato da:

- Davide Galati, in arte "PsykeDady" [psdady@msn.com, psykedady@gmail.com]
- Gianpaolo Cascardo [gcascardo@hotmail.it]
- Maria Chiara Nicoletti [nicoletti_mariachiara@virgilio.it]

Il lavoro è stato reso pubblico nella pagina di Github:

https://github.com/PsykeDady/PQNNSearchArchitettureUnical201819Gruppo9/

Si vuole ringraziare il professore Fassetti per averci aiutato a districarci in alcuni strani fenomeni e meccanismi del "*fantastico*" mondo di programmazione a basso livello.

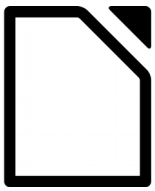
Per lo sviluppo del progetto, per stilare la relazione e tutto quello che concerne il lavoro svolto sono stati usati i seguenti strumenti:

- **Visual Studio Code**: code editor avanzato e programmabile sviluppato e fornito da Microsoft per i sistemi Linux, Windows e macOS tramite il framework di sviluppo Electron. Link.

- **GDB e CGDB**: gdb è un debugger con supporto a diversi linguaggi, tra cui *c* e *assembly*, nel progetto abbiamo abusato della sua versione scritta tramite il framework *ncurses* che ne consente un interazione grafica minimale tramite terminale Link.

- **Git e Github**: git è software di controllo versione distribuito, uno dei più usati in ambito development il cui sviluppo naque da Linus Torvalds per supportare il progetto Linux. Di per sé è uno strumento molto efficace per il lavoro in gruppo, ma accoppiato ad un buon servizio di hosting dedicato (come per l'appunto *github*, oggi proprietà di *Microsoft*) semplifica le meccaniche di uno sviluppo di un progetto software Link.

- **Libreoffice**: suite di lavoro per l'ufficio free e opensource, originariamente fork di OpenOffice è oggi una delle migliori alternative alla suite di Microsoft. La suite comprende vari moduli e strumenti che semplificano la produttività: scrittura dei documenti (Writer), fogli di calcolo (Calc), presentazioni e slides (Impress), Grafici e diagrammi (Draw), scrittura di formule matematiche (Math) e database (Base). Link.



~il team di sviluppo del Gruppo 9: