



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

Architetture Programmazione dei Sistemi di
Elaborazione

Relazione progetto a.a 2018/19

Algoritmo “Product Quantization for Nearest
Neighbor Search”

in linguaggio NASM x86-32+SSE e x86-64+AVX

Professore
Fabrizio Angiulli

Studenti
Gianmarco Magnone 204920
Lorenzo Morelli 207038
Antonio Piluso 204865

A.A. 2018-2019

1. Introduzione

Obiettivo del progetto è mettere a punto un'implementazione dell'algoritmo “*Product Quantization for Nearest Neighbor Search*” (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 NASM x86-32+SSE e dalla sua estensione x86-64+AVX (nasm) e dal sistema operativo Linux (ubuntu). In particolare, il codice deve consentire di effettuare sia la ricerca ANN esaustiva (parametro -exhaustive, default) che la ricerca ANN non esaustiva (parametro -noexhaustive). Inoltre, deve permettere di scegliere la distanza approssimata da utilizzare, ovvero la distanza simmetrica (parametro -sdc, default) oppure la distanza asimmetrica (parametro -adc). Come consigliato dalla traccia si è deciso di andare prima ad implementare una versione base e stabile del codice in linguaggio C, per poi andare ad analizzare i vari metodi computazionalmente meno efficienti e convertirli in NASM sfruttando le tecniche di ottimizzazione studiate durante il corso. Lo sviluppo del progetto è stato registrato e documentato grazie all'utilizzo del sistema di versioning Git e dalla piattaforma github. Le versioni riportate nel documento sono un sottoinsieme delle versioni che è possibile consultare sulla repository dal link: <https://github.com/Lory999555/project-C-nasm> dove è anche possibile analizzare in dettaglio tutte le modifiche apportate. I test in generale sono stati effettuati in maniera esaustiva su diversi Dataset e diverse dimensioni che variano dai $1.000 * 16$ a $40.000 * 2.000$ (punti*dim) testando un ampio range di valori dei parametri. Grazie ad una fase di testing esaustiva è stato possibile rilevare e fixare molti bug e comprendere al meglio i pregi ed i difetti delle nostre implementazioni. Nell'immagine di seguito sono rappresentate le versioni del progetto:

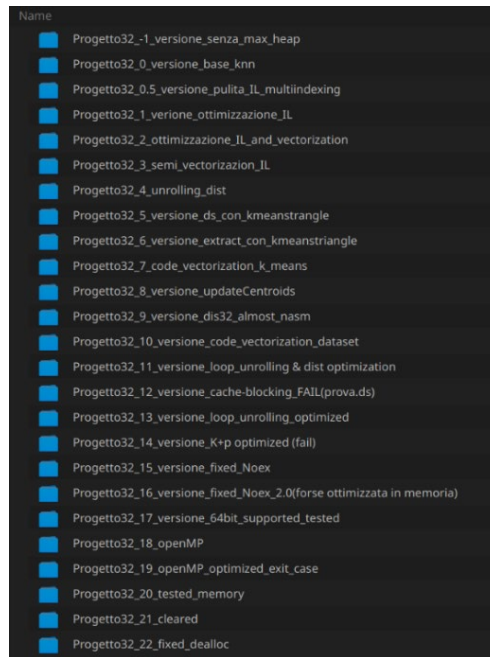


Figura 1: Versioni realizzate

Successivamente vedremo nel dettaglio le maggiori differenze tra le varie versioni motivandone il perché di alcune scelte e documentandone i miglioramenti sui tempi e sulla memoria.

2. Scelte implementative

L'implementazione dell'intero progetto si è basata sul paper¹ fornito dal professore, cercando di rimanere il più fedele possibile alle procedure e agli algoritmi descritti al suo interno, con l'aggiunta di alcune modifiche necessarie ai fini del miglioramento delle prestazioni. È stata quindi effettuata un'attenta analisi dei pro e contro dell'algoritmo, soffermandosi in modo particolare sui calcoli da effettuare sulle principali strutture dati quali Dataset, Queryset e Centroids (con il termine “Centroids” indicheremo d'ora in avanti la struttura dati contenente i centroidi). Da questa analisi è emerso che se le

¹ Product quantization for nearest neighbor search - Hervé Jégou, Matthijs Douze, Cordelia Schmid

dimensioni del Dataset fossero proporzionali al fattore di parallelismo “ p ”, allora sarebbe stato indifferente (rispetto alla località degli accessi) salvare il Dataset per righe (*row major order*) o per colonne (*column major order*), mentre se il fattore “ p ” non fosse stato proporzionale alle dimensioni, allora sarebbe stato conveniente effettuare un salvataggio per colonne. Le scelte implementative sono state fatte prima della scrittura del codice C in modo da improntare l’algoritmo verso una direzione ben precisa: per prima cosa si è studiato affondo come tale algoritmo doveva essere implementato e quindi quali erano le computazioni più dispendiose da fare sulle varie strutture descritte in precedenza. Da questa analisi è emerso che la struttura Dataset, che contiene al suo interno tutti i punti y , vettorizzata per colonna avrebbe garantito un minor numero di miss nella cache e il calcolo parallelo di più punti per un solo centroide. In più tale scelta si adatta meglio alla struttura dei Dataset in quanto, essendo solitamente le dimensioni più piccole rispetto al numero dei punti, un eventuale calcolo su Dataset non perfettamente parallelizzabili (es: 8001 punti con 129 dimensioni) agevola la computazione in quanto si evita di dover effettuare il ciclo resto (ipotizzando il fattore $p = 4$) $8001 * num_centroidi$ volte nella versione per riga rispetto a doverlo effettuare $129 * num_centroidi$ volte. In più dai calcoli effettuati è emerso che il numero di miss nelle due varianti, ipotizzando che la cache riesca a caricare “ p ” elementi, sia:

$$Colonna = \left(128 + \frac{128}{4}\right) * \frac{8000}{4} * 256 = 81.920.000 \text{ miss}$$

$$Riga = \frac{128}{4} * 2 * 256 * 8000 = 131.072.000 \text{ miss}$$

Quindi da queste verifiche si è scelto di utilizzare il Dataset per colonna e ovviamente basare tutto il calcolo del *k-means* su questa scelta progettuale. Le altre due strutture hanno invece la caratteristica di effettuare molti accessi in memoria utilizzando il centroide più vicino ad un dato punto. Questo comportamento non è compatibile con la parallelizzazione per colonna in quanto i punti non risulterebbero contigui in memoria, mentre utilizzandoli per riga, le dimensioni di tali punti risultano contigue in memoria e quindi compatibili con varie ottimizzazioni. Alla luce di ciò si è quindi deciso di effettuare il salvataggio del Dataset per colonne, mentre quello di Queryset e Centroids per righe. Altra importante scelta implementativa è stata quella di utilizzare un Maxheap, come suggerito nel paper¹, che permette di mantenere i *knn* punti. In tale struttura viene utilizzata una gestione particolare degli elementi al suo interno per minimizzare gli accessi e soprattutto gli spostamenti interni. La logica implementata è la seguente:

```
void max_heap(int* index, float* result_dist, int y,
float new_max;
if (full==true || c_max_heap == dim) {
    bool trovato = false;
    new_max = tmp;
    for(int j = 0; j < dim; j++)
    {
        if(! float *result_dist ;t[j]==max){
            result_dist[j]=tmp;
            index[j]=y;
            trovato=true;
        }else if (result_dist[j] > new_max)
        {
            new_max = result_dist[j];
        }
    }
    result[0] = new_max;
}
else
{
    index[c_max_heap]=y;
    result_dist[c_max_heap]=tmp;
    if (tmp > pre_max_heap) {
        pre_max_heap=tmp;
    }
    c_max_heap++;
    if (c_max_heap==dim) {
        result[0] = pre_max_heap;
    }else
    {
        result[0] = max;
    }
}
}
```

Figura 2: Metodo maxHeap

-utilizzo di una variabile *bool* per decidere se la struttura deve essere popolata, e quindi utilizzare una gestione diversa per l’inserimento dei primi *knn* elementi.

-la funzione interagisce con la struttura andando ad esportare una variabile dislocata dalla struttura stessa che mantiene sempre il valore del massimo corrente in modo da poter interrogare tale variabile per decidere se entrare o meno al suo interno.

-se l’entrata è obbligatoria il metodo selezionerà il primo elemento che corrisponde alla distanza massima corrente e verrà sostituito con il nuovo valore entrante, in modo da non dover effettuare nessun riposizionamento. Nel mentre l’algoritmo calcola quale sarà il nuovo massimo corrente restituendolo, il tutto ciclando una sola volta la struttura.

-i risultati vengono caricati in ANN direttamente dalla struttura senza un ordine ben preciso e senza utilizzare il massimo corrente in modo da non doverlo ricalcolare nq volte.

Non essendo l’ordine esplicitamente richiesto dalla traccia, tale scelta è stata presa per una questione puramente di ottimizzazione in quanto la struttura assicura che i risultati siano effettivamente i *knn* più vicini ma non dà alcuna sicurezza sull’ordine e soprattutto non dà priorità ai primi elementi entranti piuttosto che agli ultimi.

Per quanto riguarda l’utilizzo del *k-means*, si è provato ad implementare una versione ottimizzata che evita il calcolo di distanze superflue applicando la disuguaglianza triangolare e tenendo traccia di lower bounds e upper bounds per le

distanze tra punti e centroidi (come suggerito nel paper²). L'algoritmo risultava però troppo complesso e ci si è resi conto che era possibile applicare poche ottimizzazioni in NASM rispetto a quelle che si potevano applicare sull'implementazione classica del *k-means*. Proprio per questo motivo si è scelto di utilizzare l'algoritmo base del *k-means*, andando poi ad applicare diverse ottimizzazioni in NASM. La scelta alla fine è ricaduta su un *k-means* standard perché si è ritenuto più opportuno far prevalere la velocità dettata dalle ottimizzazioni effettuate e studiate a lezione, piuttosto che ottenere un algoritmo più veloce ma meno "lavorato" a basso livello, in quanto avrebbe obbligato il team a non poter utilizzare molte delle tecniche implementate nella versione base. Di seguito sono riportati frammenti di codici del *triangle k-means* e del *k-means* standard.

```
do {
    iter++;
    /* save error from last step */
    old_error = error; error = 0;
    /* clear old counts and temp centroids */
    for (i = 0; i < k; i++) {
        counts[i] = 0;
        for (j = 0; j < d; j++){
            c[i*d+j] = 0;
        }
    }
    printf("Identify the closest cluster in %d iteration\n", iter);
    for (h = 0; h < n; h++) {
        /* identify the closest cluster */
        min_distance = FLT_MAX; // DBL_MAX;
        for (i = 0; i < k; i++) {
            distance = 0;
            for (j = 0; j < d; j++){
                distance += pos(data[h*d+j]) - c[i*d+j]; 2;
            }
            if (distance < min_distance) {
                labels[h] = i;
                min_distance = distance;
            }
        }
        /*printf("Update size and temp centroid of the destination cluster for %d point\n", h);
        /* update size and temp centroid of the destination cluster */
        for (j = 0; j < d; j++){
            c[labels[h]*d+j] += data[h*d+j]; // c'era un ++
        }
        counts[labels[h]]++;
        /* update standard error */
        error += min_distance;
    }
    printf("Update all centroids\n");
    for (i = 0; i < k; i++) { /* update all centroids */
        for (j = 0; j < d; j++) {
            if (counts[i] != 0) {
                c[i*d+j] = c[i*d+j] / counts[i];
                /*printf("SI ----> c[%d][%d] = %f \n", i, j, c[i*d+j]);
            }
            else {
                c[i*d+j] = c[i*d+j];
                /*printf("NO ----> c[%d][%d] = %f \n", i, j, c[i*d+j]);
            }
        }
    }
}
```

Figura 3: standard *k-means*

```
//initialization
for (int i = 0; i < n; i++)
{
    /* identify the closest cluster */
    int pos=0;
    distance = dist(&data[i*d], &centroids[pos*d], d);
    min_distance=distance;
    labels[i]=pos;
    int currentCentroid = pos;

    //lowerbound[point_id][closestCentroid]
    l[i*k+pos]=min_distance;

    for (int i_c = 0; i_c < k; i_c++) {
        if (i_c != currentCentroid) {
            int ind = mapping(labels[i], pos, k);
            if (0.5*ICD[ind] < min_distance) {
                distance = dist(&data[i*d], &centroids[i_c*d], d);
                l[i*k+pos]=distance;
                if (min_distance > distance) {
                    //labels[i] = pos;
                    labels[i]=i_c;
                    min_distance = distance;
                }
            }
        }
    }
    //pos++;
}
u[i]=min_distance;
counts[labels[i]]++;

for (j = 0; j < d; j++) {
    c[labels[i]*d+j] += data[i*d+j]; // c'era un ++
}

/* update standard error */
error += min_distance;
}
```

Figura 4: Triangle *k-means*

L'algoritmo è stato testato andando a selezionare il numero dei centroidi pari al numero dei punti, così da verificare se le distanze ottenute combaciavano con le distanze fornite dal professore nel file *matlab*. Di seguito vengono riportati i risultati effettuati sull'ultima versione per andare ad evidenziare la bontà dell'algoritmo che nel caso sopracitato calcola le distanze corrette. Gli altri due casi sono approssimazioni effettuate con un basso numero di centroidi (a sinistra) e con un numero leggermente più elevato (immagine centrale).

```
Query #0: 0 6361 115 4832 245 3827 4881 2723
Query #1: 3309 7008 6937 5365 5311 3931 5067 3568
Query #2: 2177 1731 7817 5340 1109 5265 1063 5673
Query #3: 1346 2691 623 377 3365 361 438 7080
Query #4: 6591 3811 627 7594 1181 7608 669 7211
Query #5: 1057 5281 4437 5626 2887 2232 5017 5080
Query #6: 4801 4590 6540 7489 2383 6059 1229 2208
Query #7: 1385 3924 7566 1864 6887 7127 2740 3920
Query #8: 7633 735 2943 5793 7583 3575 5575 1665
Query #9: 2128 3539 3091 2843 8 7718 2954 7175
Query #10: 6945 2819 4299 5643 4480 6587 1830 6455
```

Figura 5: -exhaustive -adc -k 256

```
Query #0: 6361 7016 4832 115 2723 4881 6727 3827
Query #1: 3309 1567 5311 7008 6937 7618 3467 6075
Query #2: 1063 7817 5673 6314 2572 5265 1109 1731
Query #3: 1346 3365 361 7080 623 438 7212 101
Query #4: 7594 2574 6480 6792 627 7211 1916 1181
Query #5: 7123 465 1328 3812 5017 6603 1057 5626
Query #6: 7489 6540 4486 1229 6059 5837 2498 4590
Query #7: 3920 1450 7566 791 1344 6887 7127 1385
Query #8: 7583 2943 5575 1665 11 6458 5793 2634
Query #9: 7718 3108 7175 3091 8 2128 2843 4170
Query #10: 6587 410 4480 2107 2819 5643 6455 1830
```

Figura 6: -exhaustive -adc -k 512

```
Query #0: 3486 7382 3852 4881 3647 11 2723 7016
Query #1: 2588 5907 1567 3309 6937 7008 5311 7192
Query #2: 7817 1063 2228 6768 6314 1109 1731 3323
Query #3: 7915 4952 377 5629 623 6226 7080 361
Query #4: 3947 7211 7594 2326 446 4266 6792 2244
Query #5: 7123 1057 5080 6730 1328 2487 354 3812
Query #6: 697 1734 4590 6237 6059 6540 1229 4968
Query #7: 467 2037 1385 7647 7127 3920 6887 1344
Query #8: 1665 7308 5456 6532 7583 5793 5794 2943
Query #9: 5210 5849 3108 386 7175 6147 6413 5244
Query #10: 1206 5643 410 2819 6455 4480 6587 1830
```

Figura 7: -exhaustive -adc -8000

	1	2	3	4	5	6	7	8
1	4883	7017	3853	3487	5648	2724	7383	12
2	6938	5312	3310	2969	1568	7009	5908	7193
3	1732	1064	7818	3324	1110	6315	6769	2229
4	624	6227	4953	378	5630	7916	362	7081
5	7595	3948	4267	7212	447	2245	6793	2327
6	1329	7124	6731	1058	5081	2488	3813	355
7	6541	1230	4591	1735	4969	6060	6238	698
8	6888	7128	1386	3921	468	7648	2038	1345
9	5794	1666	7584	7309	5457	2944	6533	5795
10	7176	307	6414	5211	5245	3109	5890	6148

Figura 8: risultati matlab

N.B. nelle immagini, i risultati in matlab partono da indice 1 anziché indice 0 come invece avviene nell'algoritmo

² Using the Triangle Inequality to Accelerate k-Means - Charles Elkan

3. Versioni

Si riportano di seguito tutti i miglioramenti e le modifiche apportate da una versione all'altra. La prima versione funzionante, **versione 1**, è stata scritta interamente in linguaggio C senza apportare ottimizzazioni in NASM. In questa versione è stato implementato il Maxheap (non presente nelle versioni precedenti), e sono state vettorizzate tutte le strutture dati. Nelle versioni successive, fino alla **versione 5**, ci sono varie prove di vettorizzazione delle strutture e varie prove sul *k-means* ed il *triangle k-means* fino ad arrivare ad una versione stabile e definitiva in C.

Significativa è stata la **versione 6**, ovvero la versione finale nel linguaggio C, in cui è stato modificato il metodo del Maxheap effettuando un riempimento automatico delle prime *knn* posizioni. Altra sostanziale modifica consiste nell'implementazione del mapping tramite il quale era possibile salvare solo $\frac{k(k-1)}{2}$ elementi nella matrice delle distanze, in quanto matrice simmetrica e con elementi della diagonale pari a zero.

```
int mapping(int i,int j,int n, int index){
    /*int indice = 0;
    mapping32(i,j,n,&indice, index);
    return indice;*/

    if(i == j){
        return -1;
    }
    else if ( i > j ){
        return index - (n-j)*((n-j)-1)/2 + i - j - 1;
    }else
    {
        return index - (n-i)*((n-i)-1)/2 + j - i - 1;
    }
}
```

Figura 9: mapping

Le prime ottimizzazioni in NASM sono presenti dalla **versione 7** in cui si ha avuto un miglioramento dei tempi di indexing. Questo è dovuto, in particolare, all'implementazione del metodo *colDistance32*, utilizzato nell'algoritmo del *k-means* tramite il quale si è andati a parallelizzare il calcolo delle distanze tra punti e centroidi sfruttando le istruzioni SSE.

```
for (i = 0; i < n; i+=p) { //per ogni punto del ds
    //identify the closest cluster
    min_distance[0] = FLT_MAX; //DBL_MAX;
    min_distance[1] = FLT_MAX;
    min_distance[2] = FLT_MAX;
    min_distance[3] = FLT_MAX;

    for (j = 0; j < k; j++) { // per ogni centroide

        coldistance32(data,centroids,distance,i,j,d,n);

        //printVectorfloat(distance,p);

        for(int k=0;k<p;k++){
            if (distance[k] < min_distance[k]) {
                labels[i+k] = j;
                min_distance[k] = distance[k];
            }
        }
    }
}
```

Figura 10: estratto k-means

```
fork:
    ;printregs xmm7
    mov     esi,[ebp+dataset]    ;dataset
    mov     edi,[ebp+n]         ;n
    imul    edi,ecx             ;4*k*n
    add     esi,edi             ;dataset + 4*k*n

    movaps  xmm0,[eax+esi] ; DS[i..i+p-1][k] = DS[4*i+4*k*n..4*(i+k*n+p-1)]
    ;printregs  xmm0

    mov     esi,[ebp+c]         ;centroids
    mov     edi,[ebp+dimension] ;d
    imul    edi,ebx             ;4*j*d
    add     esi,edi             ; centroids + 4*j*d

    movss   xmm1,[ecx+esi] ; C[j][k] = C[4*k+4*j*d]

    shufps  xmm1,xmm1,0
    ;printregs  xmm1
    subps   xmm0,xmm1
    mulps   xmm0,xmm0          ;tmp[i..i+p-1] rispetto al j-esimo centroide
    addps   xmm2,xmm0          ; distance[i..i+p-1] += tmp
    ;printregs  xmm2
```

Figura 11: colDistance32

Più nello specifico, con questo metodo, si andavano a prendere in considerazione la prima dimensione di 4 punti differenti e ci si andava a calcolare la distanza con la prima dimensione del centroide. Tale processo veniva poi iterato per ogni dimensione riuscendo così a calcolare la distanza di 4 punti alla volta con un centroide.

Un sostanziale miglioramento della fase di searching si è ottenuto dalla **versione 9** nella quale è stato implementato un metodo, chiamato *dist32*, che è molto simile al *colDistance32* precedentemente analizzato ma con una sostanziale differenza: nel *colDistance32* il risultato sarà un vettore contenente "p" distanze e quindi sono state utilizzate solo le istruzioni *addps* mentre in *dist32* il risultato è la singola distanza tra 2 punti con l'utilizzo dell'istruzione *haddps* secondo la struttura vista a lezione. La traduzione in NASM di questo metodo ha portato una notevole diminuzione del tempo di searching in quanto viene utilizzato nella fase dei pre-calcoli, calcolando la distanza tra centroidi (nella ricerca simmetrica) o la distanza tra centroidi e punti del Queryset (nella ricerca asimmetrica).

```
ciclo:
    cmp     esi,edi             ;(>=d-4)?
    jge     fine
    movaps  xmm0,[eax+4*esi] ;x[l]
    subps   xmm0,[ebx+4*esi] ;x[l]-y[l]
    mulps   xmm0,xmm0           ;(..)^2
    ;printregs  xmm0
    addps   xmm1,xmm0           ;distance+=(..)^2
    add     esi,4               ;avanzo di indice

    movaps  xmm0,[eax+4*esi]
    subps   xmm0,[ebx+4*esi]
    mulps   xmm0,xmm0
    ;printregs  xmm0
    addps   xmm1,xmm0
    ;printregs  xmm1
    add     esi,4
    jmp     ciclo
fine:
    movaps  xmm0,[eax+4*esi] ;sommo gli ultimi elementi rimanenti
    subps   xmm0,[ebx+4*esi]
    mulps   xmm0,xmm0
    haddps  xmm1,xmm0           ;merge di tutte le somme
    ;printregs  xmm1
    haddps  xmm1,xmm1
    ;printregs  xmm1
```

Figura 12: dist32

```
float* pre_adc(MATRIX x, float* centroids,int d,int m, int k){
    //float* result=(float**)get_block(sizeof(float*),m);
    float* result= alloc_matrix(m,k);
    int sub=d/m;
    int i,j;
    float distance;
    MATRIX uj_x;
    for(j=0; j<m; j++){
        uj_x = Uj_x(x, j, m, 1, d);
        //result[j]=alloc_matrix(k,1);
        for(i = 0; i < k; i++){
            //result[j*k+i] = dist(uj_x, &centroids[j*k*sub+i*sub],sub);
            distance = 0;
        }
    }
}
```

Figura 13: pre-calcoli adc

Nella **versione 11** del progetto c'è stata un'analisi dei tempi di tutti i metodi, andando a contornare i vari metodi con l'istruzione per la gestione del clock, in modo da avere un quadro completo di cosa potesse essere ottimizzato e quanto la sua traduzione in NASM potesse apportare delle tangibili migliorie. Grazie a ciò ci si è infatti accorti che il calcolo degli indici per il salvataggio della diagonale superiore era troppo dispendioso nonostante la traduzione del metodo in NASM (la maggiore perdita di tempo era probabilmente dovuta ad operazioni di divisione necessarie ai fini del calcolo dell'indice, gestite in maniera ottimizzata dal gcc di C ed implementate con la divisione in NASM). Proprio per queste ragioni si è scelto di evitare il mapping e salvare banalmente l'intera matrice, con l'accortezza di inserire manualmente il valore anche nella posizione simmetrica e di porre ogni elemento della diagonale pari a 0. Questa semplice modifica ha portato una notevole diminuzione del tempo di completamento del searching, ottenendo un miglioramento di tale tempo del 67%.

```
float* pre_sdc(float* centroids, int d, int m, int k){
    int k_2 = k*k;
    float* result = alloc_matrix(m, k_2);
    int sub=d/m;
    int i, j, c, j_d;
    float distance;
    for(j=0; j<m; j++){
        for(i=0; i<k; i++){
            for(j_d=i+1; j_d<k; j_d++){
                distance = 0;
                rowDistance32Sdc(centroids, &distance, i, j, j_d, k, sub);
                result[j*k_2+i*k+j_d]=distance;
                result[j*k_2+j_d*k+i]=distance;
                result[j*k_2+i*k+i]=0;
            }
        }
    }
}
```

Figura 14: pre-calcolo sdc

```
tmp=0;
for(z=0; z < input->m; z++){
    /*t=mapping(c_x[z], l_i[ind*nodo+1+z], input->k, mapping_n);
    if (t!=-1) {
        tmp+= stored_distance[z*mapping_n+t];
    }*/
    tmp+= stored_distance[z*k_2+c_x[z]*input->k+l_i[ind*nodo+1+z]];
    //cont++;
}
```

Figura 15: eliminazione mapping

Per quanto riguarda l'indexing, è stato implementato un unroll direttamente in C che ha apportato piccoli miglioramenti consentendo all'algoritmo di calcolare 16 distanze contemporaneamente.

```
for (i = 0; i < n; i+=p*unroll){ //per ogni punto del ds
    //identify the closest cluster
    assignValue(min_distance, FLT_MAX, 0);
    assignValue(min_distance, FLT_MAX, p);
    assignValue(min_distance, FLT_MAX, p*2);
    assignValue(min_distance, FLT_MAX, p*3);

    //clock_t t11 = clock();

    for (j = 0; j < k; j++){ // per ogni centroide

        colDistance32(data, centroids, distance, i, j, d, n);
        colDistance32(data, centroids, &distance[p], i+p, j, d, n);
        colDistance32(data, centroids, &distance[p*2], i+p*2, j, d, n);
        colDistance32(data, centroids, &distance[p*3], i+p*3, j, d, n);
    }
}
```

Figura 16: unroll in C

Le **versioni 12-13-14** sono in qualche modo collegate tra di loro in quanto c'è stato un tentativo di implementazione del cache blocking che, purtroppo, non ha portato ai risultati sperati. Nella prima delle 3 versioni, l'idea era quella di andare a dividere la struttura del Dataset e quella dei centroidi in blocchi non eccedenti la cache. Tali blocchi riuscivano a catturare al loro interno un insieme di punti e un sottoinsieme delle loro reali dimensioni e l'ottimizzazione consisteva nel calcolare le distanze tra i punti e tutti i centroidi appartenenti ad un blocco, per poi andare a sommare tali distanze con le distanze calcolate nelle computazioni dei blocchi successivi.

```
for (i = 0; i < n; i+=BLOCKSIZE){ //per ogni punto del ds
    //identify the closest cluster
    assignValue(min_distance, FLT_MAX, BLOCKSIZE);

    //printVectorfloat(min_distance, BLOCKSIZE);
    //printVectorfloat(distance, BLOCKSIZE*BLOCKSIZE);

    for (j = 0; j < k; j+=BLOCKSIZE){
        assignValue(distance, 0, 0, BLOCKSIZE*BLOCKSIZE);

        for(b=0; b<d; b+=BLOCKSIZE){
            //clock_t t11 = clock();

            for (z = 0; z < BLOCKSIZE; z++){ // per ogni centroide

                colDistance32Block(data, centroids, &distance[z*BLOCKSIZE], i, j+z, d, n, b);
                colDistance32Block(data, centroids, &distance[z*BLOCKSIZE + p], i+p, j+z, d, n, b);
                colDistance32Block(data, centroids, &distance[z*BLOCKSIZE + p*2], i+p*2, j+z, d, n, b);
                colDistance32Block(data, centroids, &distance[z*BLOCKSIZE + p*3], i+p*3, j+z, d, n, b);
            }
        }
    }
}
```

Figura 17: estratto k-means con cache blocking

```
fork:
    ;printregs xmm7
    mov     esi, [ebp+dataset]      ;dataset
    mov     edi, [ebp+n]           ;n
    imul    edi, ecx               ; 4*k*n
    add     esi, edi               ;dataset + 4*k*n

    movaps  xmm0, [eax+esi] ; DS[...i+p-1][k] = DS[4*i+4*k*n..4*(i+k*n+p-1)]
    ;printregs  xmm0

    mov     esi, [ebp+c]           ;centroids
    mov     edi, [ebp+dimension]   ;d
    imul    edi, ebx               ; 4*j*d
    add     esi, edi               ; centroids + 4*j*d

    movss   xmm1, [ecx+esi] ; C[j][k] = C[4*k+4*j*d]

    shufps  xmm1, xmm1, 0
    ;printregs  xmm1
    subps   xmm0, xmm1
    mulps   xmm0, xmm0            ;tmp[i..i+p-1] rispetto al j-esimo centroide
    ;sqrtps  xmm0, xmm0
    addps   xmm2, xmm0            ; distance[i..i+p-1] += tmp
    ;printregs  xmm2

    add     ecx, dim              ; k++
    ;addps   xmm7, [inizio]
    cmp     ecx, edx              ; (k < (b+blocksize)*4) ?
    jb      fork
```

Figura 18: colDistance32Block

Questo idealmente permetteva di effettuare meno miss all'interno della struttura Dataset, perché prima del cache blocking, essendo il Dataset salvato per colonne, una chiamata del metodo `colDistance32` generava 128 miss (esempio su prova.ds)

in quanto lo spostamento della dimensione comportava un salto nella struttura Dataset di “n” posizioni, perdendo dunque il vantaggio dettato dalla località degli accessi. Con l’ottimizzazione del cache blocking la speranza era quella di abbattere questi costi in quanto si riutilizzavano delle dimensioni che entravano in cache e quindi il calcolo di una porzione di Dataset con una porzione di centroidi non comportava ulteriori miss oltre quelle necessarie. Tale strategia è stata implementata secondo la figura 17 ma, effettuando dei test, la fase di indexing al contrario delle aspettative andava a peggiorare i tempi e si è arrivati alla conclusione che l’accesso alla struttura Dataset, in numero di miss, migliorava ma andava ad aumentare il numero di miss nella struttura dei centroidi (cosa che nella versione precedente veniva invece gestita in maniera ottimale). Questo tentativo di ottimizzazione ha fatto emergere una cattiva gestione dell’unrolling in quanto, effettuato in C, si andava a richiamare 4 volte la funzione `colDistance32` perdendo quella che era la località degli accessi sul Dataset. Per questo motivo, nella **versione 13**, l’unrolling è stato gestito completamente in NASM, andando così a caricare 16 elementi in vari registri XMM, come si può notare dalle immagini del nuovo metodo `colDistance32Optimized`:

```
for (i = 0; i <= n-size; i+=size){ //per ogni punto del ds
//identify the closest cluster
assignValue(min_distance,FLT_MAX,size);
//printf("\n-----PRIMO-----%d-----",i+size);

//clock_t t11 = clock();
clock_t t11 = clock();
for (j = 0; j < k; j++){ // per ogni centroide

//t11 = clock();
colDistance32Optimized(data,centroids,distance,i,j,d,n);
//t11 = clock() - t11;
//tot+=t11;
```

Figura 19: ottimizzazione unrolling in C

```
fork:
;printregs xmm7
mov     esi,[ebp+c]      ;centroids
mov     edi,[ebp+dimension] ;d
imul    edi,ebx         ; 4*j*d
add     esi,edi          ; centroids + 4*j*d
movss   xmm3,[ecx+esi]  ; C[j][k] = C[4*k+4*j*d]

mov     esi,[ebp+dataset] ;dataset
mov     edi,[ebp+n]      ;n
imul    ecx             ; 4*k*n
add     esi,edi          ;dataset + 4*k*n

movaps  xmm0,[eax+esi]  ; DS[i..i+p-1][k] = DS[4*i+4*k*n..4*(i+k*n+p-1)]
movaps  xmm1,[eax+esi+16] ;
movaps  xmm2,[eax+esi+32] ;
movaps  xmm3,[eax+esi+48] ;
;printregs xmm0

shufps  xmm3,xmm3,0
;printregs xmm1
subps   xmm0,xmm3
subps   xmm1,xmm3
subps   xmm2,xmm3
subps   xmm3,xmm3

mulps   xmm0,xmm0      ;tmp[i..i+p-1] rispetto al j-esimo centroide
mulps   xmm1,xmm1
mulps   xmm2,xmm2
mulps   xmm3,xmm3
sqrtps  xmm0,xmm0

addps   xmm4,xmm0      ; distance[i..i+p-1] += tmp
addps   xmm5,xmm1
addps   xmm6,xmm2
addps   xmm7,xmm3
```

Figura 20: `colDistance32Optimized`

Tale codice ha apportato un grosso miglioramento nella fase di indexing e ha spinto a riprovare l’ottimizzazione del cache blocking andando ad implementare un nuovo metodo `colDistance32OptimizedBlock` che ha la stessa logica del metodo descritto precedentemente. Nonostante ciò, anche in questo caso, non c’è stato un riscontro positivo in termini di prestazioni per le motivazioni precedentemente discusse.

```
for (i = 0; i < n; i+=BLOCKSIZE){ //per ogni punto del ds
//identify the closest cluster
assignValue(min_distance,FLT_MAX,BLOCKSIZE);
//printfVectorFloat(min_distance,BLOCKSIZE);
//printfVectorFloat(distance,BLOCKSIZE*BLOCKSIZE);

for (j = 0; j < k; j+=BLOCKSIZE){
assignValue(distance,0,0,BLOCKSIZE*BLOCKSIZE);

for(b=0; b < d; b+=BLOCKSIZE){
//clock_t t11 = clock();

for (z = 0; z < BLOCKSIZE; z++){ // per ogni centroide


colDistance32OptimizedBlock(data,centroids,&distance[z*BLOCKSIZE],i,j,z,d,n,b);
//colDistance32Block(data,centroids,&distance[z*BLOCKSIZE + p],i*p,j*z,d,n,b);
//colDistance32Block(data,centroids,&distance[z*BLOCKSIZE + p*2],i*p*2,j*z,d,n,b);
//colDistance32Block(data,centroids,&distance[z*BLOCKSIZE + p*3],i*p*3,j*z,d,n,b);
//printfVectorFloat(distance,BLOCKSIZE*BLOCKSIZE);


}

}
```

Figura 21: 2° tentativo cache blocking

Le ultime due versioni che andremo ad analizzare sono la **versione 16** e la **18**. La prima è caratterizzata da un notevole miglioramento nell’utilizzo della memoria. A causa di alcune strutture dati, non opportunamente deallocate, è stato infatti notato un eccessivo utilizzo della memoria da parte dell’algoritmo che in alcuni casi arrivava ad occupare ben 4 Gb di memoria RAM (prova effettuata sul Dataset di dimensione 40.000 per 1.000), causando il blocco dell’algoritmo. Si è quindi risolto il problema modificando leggermente il codice e passando così ad un dispendio di memoria di poche centinaia di Mb.

Name	CPU	Memory	Download	Upload	PID
 lollo@lollo-pc:~/Desktop/Versioning/Progetto32_1...	12.9%	2.3 GB			4917

Name	CPU	Memory	Download	Upload	PID
 lollo@lollo-pc:~/Desktop/Versioning/Progetto32_1...	12.5%	249.3 MB			5817

Altra importante modifica effettuata in questa versione riguarda invece la ricerca non esaustiva. Fino a quel momento veniva infatti effettuata una fase di indexing utilizzando “nr” punti del Dataset e ciò comportava l'utilizzo solo di un suo sottoinsieme andando di conseguenza ad abbassarne drasticamente i tempi. L'algoritmo è stato modificato in modo da implementare il corretto funzionamento della ricerca non esaustiva, andandolo a dividere in due blocchi dove il primo lavora solo su gli “nr” punti calcolandone i centroidi, mentre il secondo viene eseguito alla fine delle iterazioni del *k-means*, quindi quando i centroidi sono già calcolati sugli “nr” punti, e si occupa di andare ad associare ai restanti punti i centroidi più vicini. Tale implementazione ovviamente porta con sé più computazioni, e quindi nel grafico riportato nelle conclusioni emerge un aumento del tempo dovuto appunto alla correzione di questa gestione. Un'altra modifica apportata è stata la divisione dei metodi che interagiscono con le strutture in Aligned (metodoA) ed Unaligned (metodoU), in modo da poter richiamare le rispettive funzioni NASM riscritte a loro volta per supportare tale modifica.

Nella seconda delle 2 versioni (versione 18) è stato modificato il caso di uscita come descritto dal professore. Questa semplice modifica ha apportato una sostanziale diminuzione dei tempi di indexing in quanto permette all'algoritmo di effettuare quasi sempre al più 10 iterazioni, rispetto alla media di 50 iterazioni effettuate con il vecchio caso di uscita.

```
if(error == old_error){
    calc=0;
}else{
    calc = (fabs(old_error-error)/old_error);
}
}while (!(t_min <= iter && ((t_max < iter) || calc <= t)));;
```

Figura 22: exit case

Nella versione consegnata (versione 22) sono stati corretti alcuni bug che portavano a dei problemi di deallocazione, andando a modificare dei metodi per l'aggiornamento dei centroidi in alcuni file NASM che presentavano delle inesattezze sul calcolo parallelizzato. Dalla versione 16 è stata implementata anche la variante a 64 bit che è stata sottoposta anch'essa a tutte le ottimizzazioni precedentemente descritte. Tale variante non ha apportato quasi nessuna modifica in C, se non l'accortezza di allocare le strutture allineate a 32 bit. Tutti i metodi NASM sono stati convertiti utilizzando le nuove istruzioni fornite dall'estensione del repertorio AVX, che quindi ha permesso di portare il fattore di parallelismo da 4 a 8 e di conseguenza il calcolo delle varie distanze da 16 a 32. Grazie alla presenza del doppio dei registri, si sono anche riusciti ad implementare in maniera più agevole alcuni metodi come il *colDistance64Optimized*, che risultava abbastanza articolato nella versione SSE.

```
fork1:
;printregs xmm7
mov     esi,[ebp+c]      ;centroids
mov     edi,[ebp+dimension] ;d
imul    edi,ebx         ; 4*j*d
add     esi,edi          ; centroids + 4*j*d
movss   xmm3,[ecx+esi] ; C[j][k] = C[4*k+4*j*d]

mov     esi,[ebp+dataset] ;dataset
mov     edi,[ebp+n]      ;n
imul    edi,ecx         ; 4*k*n
add     esi,edi          ;dataset + 4*k*n

movaps  xmm0,[eax+esi] ; DS[i..i+p-1][k] = DS[4*i+4*k*n..4*(i+k*n+p-1)]
movaps  xmm1,[eax+esi+16] ;
movaps  xmm2,[eax+esi+32] ;

shufps  xmm3,xmm3,0
;printregs xmm1
subps   xmm0,xmm3
subps   xmm1,xmm3
subps   xmm2,xmm3
subps   xmm3,xmm3

mulps   xmm0,xmm0 ;tmp[i..i+p-1] rispetto al j-esimo centroide
mulps   xmm1,xmm1
mulps   xmm2,xmm2
mulps   xmm3,xmm3
sqrtps  xmm0,xmm0

addps   xmm4,xmm0 ; distance[i..i+p-1] += tmp
addps   xmm5,xmm1
addps   xmm6,xmm2
addps   xmm7,xmm3

movaps  xmm0,[eax+esi+48] ;
subps   xmm0,xmm3
mulps   xmm0,xmm0
addps   xmm7,xmm0
```

Figura 23: versione SSE

```
fork1:
mov     r12,rsi         ;centroids
mov     r10,r9          ;d
imul    r10,r8          ; 4*j*d
add     r12,r10          ; centroids + 4*j*d
vbroadcastss ymm8,[r13+r12] ; C[j][k] = C[4*k+4*j*d]

mov     r12,rdi         ;dataset
mov     r10,r15         ;n
imul    r10,r13         ; 4*k*n
add     r12,r10          ;dataset + 4*k*n

vmovaps ymm0,[rcx+r12] ; DS[i..i+p-1][k] = DS[4*i+4*k*n..4*(i+k*n+p-1)]
vmovaps ymm1,[rcx+r12+32] ;
vmovaps ymm2,[rcx+r12+64] ;
vmovaps ymm3,[rcx+r12+96] ;

vsubps  ymm0,ymm8
vsubps  ymm1,ymm8
vsubps  ymm2,ymm8
vsubps  ymm3,ymm8

vmulps  ymm0,ymm0 ;tmp[i..i+p-1] rispetto al j-r12mo centroide
vmulps  ymm1,ymm1
vmulps  ymm2,ymm2
vmulps  ymm3,ymm3
sqrtps  xmm0,xmm0

vaddps  ymm4,ymm0 ; distance[i..i+p-1] += tmp
vaddps  ymm5,ymm1
vaddps  ymm6,ymm2
vaddps  ymm7,ymm3
```

Figura 24: versione AVX

Anche le distanze per riga sono state convertite a 64 bit andando a migliorare i tempi soprattutto con dimensioni molto grandi in quanto, con dimensioni ridotte, si va ad aumentare il numero di esecuzione del ciclo resto. Da svariate prove, la versione 64 bit risulta essere molto più prestante nella fase di indexing su tutti i tipi di Dataset, e risulta essere molto più prestante nella fase di searching soprattutto con Dataset che presentano un elevato numero di dimensioni. L'ultima variante implementata è quella che sfrutta l'ottimizzazione *OpenMP* andando ad esportare delle direttive per il pre-processore, che rendono parallelo (distribuito su più thread), ad esempio, il calcolo dei for. Tale ottimizzazione è stata applicata sul ciclo presente nel metodo *productQuantization* in quanto vengono richiamati in maniera indipendente i *k-means* sui vari sottogruppi dettati dal fattore “m”. Grazie a questa implementazione c'è stato un ulteriore abbassamento

dei tempi, che per quanto riguarda l'indexing, ha evidenziato un miglioramento di quasi il 50% rispetto alla versione a 64 bit, che a sua volta va a migliorare di un 40% la versione già ottimizzata a 32bit, come si può vedere nei grafici in conclusione.

Conclusioni

In conclusione, vengono riportati i test documentati e gli andamenti grafici relativi a tali test per apprezzarne le migliorie apportate nelle varie versioni. In seguito, vengono riportate le tabelle contenenti tutti i tempi in secondi delle varie versioni testate su prova.ds e Dataset.ds (Dataset home-made con 20.000 punti di dimensione 1.000 e utilizzato anche come Queryset). Per rendere omogeneo il tutto, i test sono stati effettuati solo sulla versione a 32 bit, ad eccezione degli ultimi 3 test rappresentati nella seconda tabella, dove viene specificata la versione utilizzata, e della “(19) versione OpenMP/x64” in cui è stata utilizzata la variante a 64 bit.

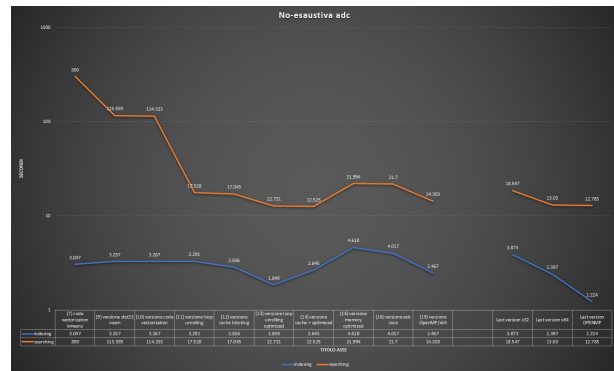
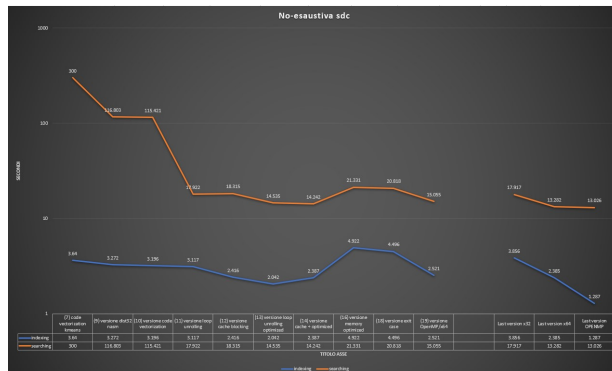
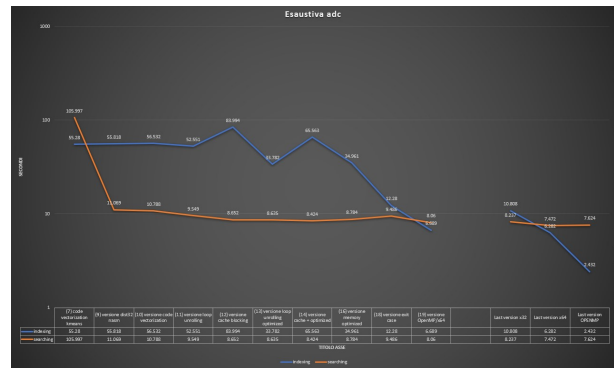
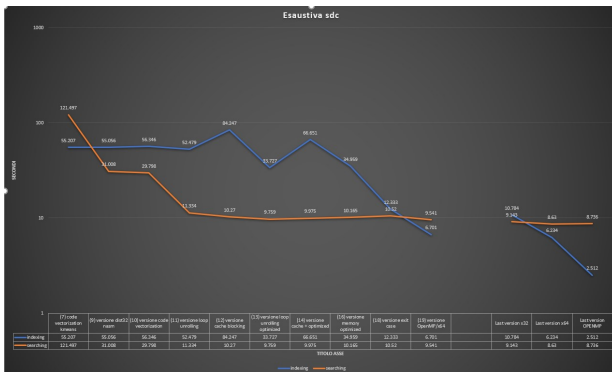
Versioni	Esaustiva sdc		Esaustiva adc		Non-esaustiva sdc		Non-esaustiva adc				
	indexing	searching	indexing	searching	indexing	searching	indexing	searching			
(0.1) basic senza max heap	152.478	8.903	158.438	3.185	305.953	13.315	306.74	13.164	32 bit	8000*128	2000*128
(0.2) versione base	161.879	3.237	162.276	1.837	305.01	13.485	305.948	13.638	32 bit	prova.ds	prova.qs
(0.3) versione pulita IL multindexi	159.4	3.397	159.262	1.703	299.015	10.884	298.386	11.043	32 bit	Knn=4	Knn=4
(1) versione ottimizzata IL	158.055	4.162	157.97	2.037	299.937	13.455	299.956	13.371	32 bit	M=8	M=8
(2) ottimizzazione IL and vectoriz	160.104	4.683	160.151	2.162	301.299	13.166	301.426	12.834	32 bit	K=256	K=256
(3) semi vectorization IL	157.925	3.161	157.804	1.723	298.725	10.962	299.921	10.894	32 bit	Kc=256	Kc=256
(4) unrolling dist	158.923	3.261	157.828	1.98	298.536	11.083	297.623	10.982	32 bit	W=8	W=8
(5) versione con kmeans triangle	158.652	3.168	158.012	1.842	298.236	11.125	297.236	10.562	32 bit	nr=n/20	nr=n/20
(6) versione extract_col	155.183	3.004	154.723	1.625	9.722	11.487	9.4	11.764	32 bit		
(7) code vectorization kmeans	4.378	3.326	4.493	1.743	0.208	10.859	0.135	10.987	32 bit		
(8) versione update centroids	4.221	3.296	4.314	1.804	0.204	11.136	0.137	11.454	32 bit		
(9) versione dist32 nasm	4.203	2.082	4.429	0.559	0.131	1.944	0.128	1.732	32 bit		
(10) versione code vectorization	4.322	1.797	4.506	0.532	0.153	1.547	0.144	1.744	32 bit		
(11) versione loop unrolling	3.553	0.78	3.729	0.472	0.127	0.526	0.124	0.551	32 bit		
(12) versione cache blocking	4.391	0.591	4.4	0.48	0.15	0.534	0.154	0.433	32 bit		
(13) versione loop unrolling optim	2.573	0.601	2.761	0.467	0.085	0.398	0.084	0.446	32 bit		
(14) versione cache + optimized	4.543	0.7	4.533	0.466	0.155	0.572	0.152	0.444	32 bit		
(15) versione noexhaustive fixed	2.546	0.696	2.535	0.542	0.28	0.398	0.269	0.486	32 bit		
(16) versione memory optimized	2.356	0.598	2.345	0.48	0.463	0.574	0.442	0.519	32 bit		
(17) versione introduzione x64	2.301	0.542	2.378	0.51	0.226	0.316	0.226	0.575	32 bit		
(18) versione exit case	0.727	0.553	0.715	0.536	0.212	0.321	0.21	0.59	32 bit		
(19) versione OpenMP/x64	0.419	0.556	0.452	0.446	0.127	0.284	0.139	0.379	64 bit		

Figura 25: tabella tempi di esecuzione su prova.ds

Versioni	Esaustiva sdc		Esaustiva adc		Non-esaustiva sdc		Non-esaustiva adc				
	indexing	searching	indexing	searching	indexing	searching	indexing	searching			
(7) code vectorization kmeans	55.207	121.497	55.28	105.997	3.64	300	3.047	300	32 bit		
(9) versione dist32 nasm	55.056	31.008	55.818	11.069	3.272	116.803	3.257	115.559	32 bit		
(10) versione code vectorization	56.346	29.798	56.532	10.788	3.196	115.421	3.267	114.333	32 bit	20000*1000	20000*1000
(11) versione loop unrolling	52.479	11.334	52.551	9.549	3.117	17.922	3.291	17.528	32 bit	dataset.ds	dataset.qs
(12) versione cache blocking	84.247	10.27	83.994	8.652	2.416	18.315	2.836	17.045	32 bit		
(13) versione loop unrolling optim	33.727	9.759	33.782	8.635	2.042	14.535	1.849	12.731	32 bit	Knn=4	Knn=4
(14) versione cache + optimized	66.651	9.975	65.563	8.424	2.387	14.242	2.645	12.525	32 bit	M=5	M=5
(16) versione memory optimized	34.959	10.165	34.961	8.784	4.922	21.331	4.618	21.994	32 bit	K=256	K=256
(18) versione exit case	12.333	10.52	12.28	9.486	4.496	20.818	4.017	21.7	32 bit	Kc=256	Kc=256
(19) versione OpenMP/x64	6.701	9.541	6.689	8.06	2.521	15.055	2.467	14.303	32 bit	W=8	W=8
										nr=n/20	nr=n/20
Last version x32	10.784	9.143	10.808	8.237	3.856	17.917	3.873	18.547	32 bit		
Last version x64	6.234	8.63	6.282	7.472	2.385	13.282	2.397	13.03	64 bit		
Last version OPENMP	2.512	8.736	2.432	7.624	1.287	13.026	1.224	12.785	64 OMP		

Figura 26: tabella tempi di esecuzione su dataset.ds

Di seguito vengono riportati i grafici relativi alla tabella "test" su dataset.ds per evidenziarne l'andamento:



Di seguito vengono riportati i grafici relativi alla tabella “test” su prova.ds per evidenziarne l'andamento:

