ARCHITETTURE AVANZATE PER I SISTEMI DI ELABORAZIONE E PROGRAMMAZIONE

Costanza Ferrari, Michele De Fusco, Luca Cuconato

Relazione del progetto

Indice

Capitolo 1

Introduzione

La traccia del progetto richiedeva agli studenti, sostanzialmente, di implementare l'algoritmo KNN (K Nearest Neighbors) utilizzato come modello di machine learning e impiegato per la classificazione degli oggetti.

L'obiettivo è quello di trovare, per ogni elemento della collezione Query i k punti più simili presenti nel dataset dei dati.

La ricerca dei k elementi più simili viene fatta sulla base di una misurazione, la distanza euclidea, che va a calcolare la distanza fra vettori prendendo in considerazione ogni componente.

Il problema principale risiede nelle risorse che il calcolo della distanza euclidea, calcolata per ogni vettore presente in Q per tutti gli elementi presenti nel dataset, richiederebbe. Il tempo di esecuzione, sviluppando la distanza euclidea sarebbe $O(n * D * q)$ in cui n sono gli elementi del dataset, q sono gli elementi presenti nella collezione Query e D sono le dimensioni degli elementi nel dataset, le colonne della matrice del dataset.

Valutando che si sta lavorando con collezioni molto estese di dati sarebbe buona norma provare a ridurre per quanto possibile i tempi di esecuzione.

1.0.1 Come ovviare al problema del costo computazionale

Per ridurre i costi la strategia consigliata dalla traccia è quella di fare una sorta di pruning del dataset, quindi di non calcolare la distanza euclidea fra ogni query ed ogni punto del dataset ma di lavorare solo su alcuni di questi punti definiti Pivot. Questa tecnica è utilizzata per ridurre lo spazio di ricerca dei vicini.

I pivot sono un sottoinsieme di punti h selezionati dal dataset che fungono da riferimenti spaziali fissi. L'idea centrale è che, conoscendo la distanza tra un punto del dataset e un pivot, e calcolando la distanza tra la query e lo stesso pivot, è possibile stimare la distanza tra query e punto senza calcolarla effettivamente.

Questa stima si basa sulla disuguaglianza triangolare applicata agli spazi metrici:

$$|d(q,p) - d(v,p)| \leq d(q,v) \tag{1.1}$$

Il termine a sinistra, $|d(q,p) - d(v,p)|$, rappresenta un limite inferiore (lower bound) della distanza reale $d(q,v)$.

Durante la fase di ricerca, se questo limite inferiore risulta superiore alla distanza del k-esimo vicino attualmente più lontano nella lista dei risultati (d_{max}), il punto v può essere scartato immediatamente.

Questa strategia permette di ridurre drasticamente il numero di calcoli esatti in virgola mobile, limitando l'uso delle istruzioni SIMD più onerose (SSE/AVX) solo ai candidati che hanno un'alta probabilità di far parte del set dei vicini più prossimi.

1.0.2 La distanza approssimata e la quantizzazione binaria

Nonostante l'efficacia del pruning tramite pivot, un numero considerevole di punti del dataset potrebbe comunque richiedere una valutazione più approfondita e quindi il calcolo della distanza euclidea.

Per evitare di ricorrere al calcolo della distanza euclidea esatta, la traccia introduce il concetto di distanza approssimata ($\tilde{d}$).

Questa tecnica si basa sulla quantizzazione binaria dei vettori: per ogni vettore $v \in DS$ (e per ogni query q), vengono estratti due vettori binari, v^+ e v^-, di dimensione D. Tali vettori identificano le componenti che presentano maggiore valore assoluto:

- v^+ contiene 1 in corrispondenza delle x componenti più grandi e positive, 0 altrove;
- v^- contiene 1 in corrispondenza delle x componenti più grandi e negative, 0 altrove.

Grazie a questa rappresentazione, la distanza tra due punti può essere stimata mediante operazioni di prodotto scalare tra vettori binari, estremamente efficienti da calcolare a livello hardware:

$$\tilde{d}(v, w) = (v^+ \cdot w^+) + (v^- \cdot w^-) - (v^+ \cdot w^-) - (v^- \cdot w^+) \tag{1.2}$$

.

In sintesi, il flusso decisionale per ogni punto v del dataset rispetto a una query q segue una gerarchia di tre livelli:

1. Filtro Pivot: Calcolo del lower bound tramite disuguaglianza triangolare. Se il punto è palesemente lontano, viene scartato.

2. Filtro Approssimato: Se il punto supera il primo filtro, si calcola $\tilde{d}(q, v)$. Se l'approssimazione è peggiore del candidato k-esimo attuale, il punto viene scartato.

3. Verifica Esatta: Solo per i punti superstiti si procede al calcolo della distanza euclidea esatta (Eq. 1) e all'eventuale aggiornamento della lista dei k vicini. Il guadagno in termini di tempi di esecuzione risiede proprio in quest'ultimo passaggio: la formula matematica della distanza euclidea viene effetutata solo su questi ultimi elementi che hanno passato le prime operazioni di filtraggio.

Capitolo 2

Ottimizzazioni richieste

2.1 Obiettivi del progetto

L'obiettivo primario dell'attività progettuale risiede nello sviluppo di un'implementazione ad alte prestazioni dell'algoritmo di ricerca dei K vicini (K-Nearest Neighbors). Data la complessità computazionale intrinseca del problema, pari a $O(n \cdot D)$, il progetto si focalizza sul superamento dei colli di bottiglia del codice seriale attraverso l'uso di tecniche di ottimizzazione hardware-aware, sfruttando il calcolo vettoriale (SIMD) e il parallelismo multi-thread.

2.2 Varianti architetturali e requisiti di precisione

La traccia impone la realizzazione di due soluzioni distinte, differenziate per architettura di riferimento e precisione numerica, al fine di confrontare l'impatto delle diverse estensioni del set di istruzioni x86.

2.2.1 Versione 32-bit SSE (Single Precision)

Questa versione è ottimizzata per l'esecuzione in ambienti a 32 bit, privilegiando la velocità di calcolo su dati a precisione singola.

- **Precisione:** Utilizzo del tipo di dato `float` (32 bit).

- **Set di istruzioni:** Sfruttamento del repertorio **SSE** (*Streaming SIMD Extensions*).
- **Parallelismo hardware:** Impiego dei registri `XMM` da 128 bit, che consentono di processare simultaneamente 4 componenti `float`.

2.2.2 Versione 64-bit AVX (Double Precision)

La versione a 64 bit è orientata a scenari che richiedono elevata accuratezza numerica, sfruttando le capacità delle estensioni vettoriali più moderne.

- **Precisione:** Utilizzo del tipo di dato `double` (64 bit).
- **Set di istruzioni:** Impiego del repertorio **AVX** (*Advanced Vector Extensions*).
- **Parallelismo hardware:** Utilizzo dei registri `YMM` da 256 bit. Nonostante la precisione doppia, l'ampiezza del registro permette l'elaborazione di 4 elementi `double` in parallelo.

2.3 Ottimizzazioni di basso livello e gestione dati

Per garantire l'efficienza massima nelle routine Assembly, sono state adottate strategie specifiche di gestione della memoria e delle istruzioni.

2.3.1 Gestione della dimensionalità e Cleanup Loop

Un aspetto critico affrontato nelle versioni ottimizzate riguarda la gestione della dimensionalità D. Poiché non è garantito che D sia un multiplo esatto della capacità dei registri (4 elementi per ciclo), è stata implementata una sezione di *cleanup* scalare. Al termine del loop principale vettorializzato, un ciclo sequenziale processa le componenti rimanenti, garantendo l'accuratezza del calcolo per qualsiasi valore di D ed evitando accessi a memoria non allocata.

2.3.2 Ottimizzazione degli accessi e allineamento

L'efficienza SIMD è strettamente legata alla modalità di caricamento dei dati. L'allineamento a 16 byte (SSE) e 32 byte (AVX) non è solo un vincolo, ma una scelta per abilitare le istruzioni di *fast loading* come `MOVAPS` e `VMOVAPD`. Questo previene le penalità dovute ai *misaligned accesses*, massimizzando il throughput dei dati verso la CPU.

2.3.3 Istruzioni hardware specializzate

Le implementazioni Assembly sfruttano istruzioni dedicate per ridurre i cicli di clock:

- **Riduzione Orizzontale:** L'uso di `HADDPS` (SSE) consente di sommare le componenti interne a un registro con un'unica operazione, velocizzando la finalizzazione della distanza.
- **Population Count:** Nella distanza approssimata, l'istruzione `POPCNT` sostituisce interi cicli di confronto, contando i bit impostati a 1 nelle maschere binarie in un singolo ciclo hardware.

2.4 Parallelismo di alto livello con OpenMP

Per la versione AVX, è richiesta l'integrazione di **OpenMP**. Mentre la vettorializzazione accelera il calcolo della singola distanza (parallelismo a livello di dati), OpenMP introduce il parallelismo a livello di thread. La strategia prevede la distribuzione del loop delle query tra i vari core della CPU, permettendo di processare più punti contemporaneamente e sfruttando appieno l'architettura multi-core.

2.5 Sintesi tecnica delle configurazioni

La tabella seguente riassume le specifiche tecniche e le differenze hardware tra le implementazioni richieste.

Caratteristica	**Versione SSE**	**Versione AVX**
Precisione	Single Precision (32-bit)	Double Precision (64-bit)
Tipo di dato C	`float`	`double`
Registro SIMD	`XMM` (128 bit)	`YMM` (256 bit)
Allineamento	16 Byte	32 Byte
Istruzioni Chiave	`MOVAPS, HADDPS, POPCNT`	`VMOVAPD, VCMPPD, POPCNT`
Parallelismo	Data-level (SIMD)	Data-level + Thread-level (OpenMP)

Tabella 2.1: Confronto tecnico tra le architetture target del progetto.

Capitolo 3

Analisi dell'implementazione

In questo capitolo viene analizzato il codice sorgente del progetto, evidenziando le scelte tecniche adottate per garantire l'efficienza computazionale e il corretto interfacciamento tra i moduli in C e le routine in Assembly.

L'efficienza del sistema sviluppato non risiede esclusivamente nell'uso di istruzioni vettoriali, ma nasce da una progettazione precisa dell'interfaccia tra i diversi livelli del software.

Il punto di giunzione fondamentale tra l'astrazione del linguaggio C e la rigidità dell'Assembly è rappresentato dalla struttura input_data.

Questa non deve essere intesa come un semplice contenitore passivo di variabili, ma come il vero nucleo operativo che permette di superare i limiti intrinseci del passaggio dei parametri tra linguaggi di diverso livello.

In un'architettura a 32 bit, il passaggio di numerosi argomenti (come i puntatori alle matrici del dataset, delle query, dei pivot e i parametri di controllo k, h, x) graverebbe pesantemente sullo stack, introducendo un overhead che vanificherebbe i guadagni prestazionali della vettorizzazione.

L'adozione della struttura input_data permette invece di passare un unico indirizzo di memoria alla routine Assembly, la quale può poi navigare tra i dati con estrema precisione.

Questa "giunzione" è resa possibile dalla definizione di offset mnemonici che mappano la memoria fisica: nel codice Assembly, il riferimento a un dato non avviene tramite il nome della variabile, ma calcolando la distanza in byte dall'inizio della struttura. Ad esempio, l'accesso alla dimensionalità D o al numero di query nq viene gestito puntando agli indirizzi precisi (come [RDI+64] o [RDI+68]), garantendo che il codice a basso livello operi esatta-

mente sugli stessi dati manipolati dal C, senza alcuna ambiguità o spreco di cicli di clock per la gestione dei parametri.

3.1 Gestione della memoria e allineamento dati

L'efficienza delle istruzioni SIMD (SSE e AVX) è strettamente legata alla modalità con cui i dati vengono caricati dai banchi di memoria ai registri della CPU. Analizzando il modulo *matrix.c*, si osserva come l'allocazione del dataset e delle query non avvenga tramite una semplice funzione malloc, ma segua rigorosi vincoli di allineamento.

Per poter utilizzare istruzioni di caricamento rapido, come MOVAPS per SSE o VMOVAPD per AVX, i puntatori base delle matrici devono essere allineati a confini di 16 o 32 byte. Nel progetto, questa necessità è gestita tramite funzioni di allocazione specifica che garantiscono che l'indirizzo di memoria restituito sia un multiplo della dimensione del registro di destinazione. Tale accorgimento previene le penalità di performance dovute ai "misaligned accesses", che costringerebbero la CPU a eseguire cicli di lettura supplementari, riducendo drasticamente il throughput dei dati durante il calcolo delle distanze.

3.2 Modularità e setting dei target di compilazione

Ci siamo poi occupati di andare a definire diversi target di compilazione: Release_Scalar, Release_SSE2 e Release_AVX64. Questa separazione riflette una scelta implementativa precisa: mantenere un'unica logica di controllo in C capace di invocare diverse implementazioni del "back-end" computazionale.

I file distance.c e quantization.c fungono da strato di astrazione. Essi contengono le versioni scalari scritte in C, utilizzate come riferimento per la validazione dei risultati (baseline), e le dichiarazioni delle funzioni extern implementate in Assembly. Questo approccio modulare permette al compilatore di linkare le routine ottimizzate solo quando il target specifico lo richiede, garantendo che la versione a 32 bit float non venga inquinata da logiche a 64 bit double, e viceversa. Tale distinzione è fondamentale per testare accura-

tamente lo speedup ottenuto attraverso la vettorizzazione rispetto al codice sequenziale puro.

3.3 Vettorizzazione e calcolo parallelo nei registri SIMD

Analizzando i file distance_sse2.S e distance_avx2.S, emerge chiaramente come il calcolo della distanza euclidea sia stato trasformato da una sequenza di operazioni scalari a un flusso vettorializzato.

Invece di iterare singolarmente su ogni dimensione D dei vettori, il codice sfrutta i registri XMM (per la versione SSE a 32 bit) e YMM (per la versione AVX a 64 bit).

L'implementazione non si limita a caricare i dati, ma ottimizza il ciclo di calcolo $(a - b)^2$ riducendo al minimo gli accessi alla memoria attraverso l'utilizzo di istruzioni come SUBPS e MULPS.

Analizzando il loop principale, si nota l'utilizzo di istruzioni come SUBPS e MULPS. La strategia adottata prevede il caricamento di un blocco di componenti (4 float o 4 double) in un registro, la sottrazione simultanea con il corrispondente blocco del secondo vettore e l'elevamento al quadrato del risultato moltiplicando il registro per se stesso.

Un dettaglio tecnico fondamentale risiede nella gestione della riduzione orizzontale: dopo aver accumulato i quadrati delle differenze nei canali del registro vettoriale, il codice utilizza l'istruzione HADDPS (Horizontal Add Packed Single) nella versione SSE. Questa istruzione permette di sommare le componenti adiacenti all'interno dello stesso registro con un'unica operazione hardware, eliminando la necessità di complesse sequenze di shuffle e addizioni scalari, velocizzando così l'ottenimento della distanza parziale da confrontare con la soglia di pruning.

3.4 Ottimizzazione del Controllo di Flusso e Loop Unrolling

Un ulteriore livello di accuratezza è garantito dalla gestione della dimensionalità D.

Poiché non è garantito che D sia un multiplo esatto della dimensione dei registri SIMD (4 elementi per SSE o 8 per AVX), il codice implementa una gestione specifica dei resti).

Al termine del loop principale vettorializzato, è presente una sezione di "cleanup" scalare che processa le ultime componenti rimanenti. Questo assicura che il calcolo della distanza sia matematicamente esatto per qualsiasi dimensione del dataset, evitando al contempo accessi a zone di memoria non allocate che causerebbero errori di segmentazione.

Un'altra tecnica utilizzata in questa parte è la Loop Unrolling: invece di processare un singolo blocco vettoriale per ogni iterazione, il codice è strutturato per gestire più blocchi contemporaneamente.

Questa scelta riduce l'incidenza delle istruzioni di controllo del ciclo (incremento del contatore e salto condizionato) e permette alla CPU di sfruttare meglio il pipelining interno mantenendo le unità di esecuzione sempre alimentate.

Inoltre, l'interazione tra il codice Assembly e la logica di pruning descritta precedentemente è gestita in questo modo: le routine Assembly restituiscono il controllo al chiamante C non appena viene superata la soglia di distanza massima, oppure procedono al caricamento dei dati successivi se il punto è ancora un potenziale candidato.

Questo meccanismo è implementato tramite salti condizionali ottimizzati che tengono conto della predizione dei rami della CPU, evitando che un calcolo inutile rallenti l'intero processo di ricerca dei vicini.

3.5 Implementazione della quantizzazione e della distanza approssimata

Il processo di quantizzazione, implementato nelle routine Assembly, rappresenta il secondo livello di filtraggio del sistema.

L'analisi del codice rivela come il software trasformi i vettori ad alta dimensionalità in una coppia di maschere binarie, v^+ e v^-.

Questo passaggio non è una semplice riduzione di precisione, ma una vera e propria estrazione delle componenti con modulo maggiore.

Per identificare le x componenti più significative, il codice Assembly utilizza un approccio basato sull'analisi delle singole componenti del vettore.

La soglia di magnitudo utilizzata per generare le maschere binarie non è un valore statico, ma viene calcolata dinamicamente nel modulo C in base al parametro x. In Assembly, tale soglia viene "propagata" su tutti i canali del registro, permettendo all'istruzione CMPPS di confrontare l'intero blocco vettoriale in un unico ciclo di clock. Quindi attraverso l'istruzione di confronto vettoriale CMPPS (o VCMPPD in AVX), la CPU confronta simultaneamente 4 o 8 componenti con una soglia pre-calcolata.

Il risultato di questo confronto è una maschera di bit che viene poi salvata nei buffer temporanei.

Come spiegato precedentemente, ora è il momento del calcolo della distanza approssimata $\tilde{d}$, che serve per risparmiare calcoli e quindi potenza computazionale ed evitare di calcolare la distanza matematicamente corretta per ogni vettore.

Una volta ottenute le maschere binarie per la query e per il punto del dataset, il prodotto scalare tra vettori binari viene risolto tramite operazioni logiche AND seguite dall'istruzione PSADBW.

Per il calcolo della distanza approssimata $\tilde{d}(v, w)$, l'implementazione Assembly non si affida esclusivamente all'istruzione POPCNT, che potrebbe non essere presente su tutte le architetture target.

Si è scelto quindi di utilizzare l'istruzione PSADBW (Sum of Absolute Differences of Bytes).Il processo avviene in tre step vettoriali:

1. Si esegue un PAND bit-a-bit tra le maschere di quantizzazione.

2. Si utilizza una maschera costante di '1' (caricata nel registro xmm6 in SSE2 o ymm15 in AVX2) per isolare i bit attivi.

3. L'istruzione PSADBW somma orizzontalmente i byte, effettuando di fatto un population count vettoriale estremamente veloce senza la necessità di iterazioni scalari.

Questa scelta implementativa permette di sostituire centinaia di moltiplicazioni e somme in virgola mobile con poche operazioni bit-a-bit, accelerando la stima della distanza prima di decidere se procedere o meno con il calcolo euclideo esatto.

3.6 Gestione dei buffer temporanei

Un aspetto critico dell'analisi riguarda il rispetto della Windows x64 Call Convention (ABI). Come si osserva nei file distance_sse2.S e distance_avx2.S, il codice Assembly deve preservare i registri definiti non-volatili dal sistema operativo (da XMM6 a XMM15).

All'ingresso delle routine, il codice esegue un prologo rigoroso: alloca spazio sullo stack e salva i registri utilizzati (come XMM8-XMM11).

Prima del ritorno al chiamante, nell'epilogo, tali registri vengono ripristinati ai loro valori originali. Questo garantisce la stabilità dell'intero applicativo, evitando che la sovrascrittura accidentale di registri critici provochi crash, specialmente durante l'esecuzione parallela gestita da OpenMP."

All'ingresso della routine, il codice Assembly si occupa di preservare lo stato dei registri "callee-saved" (come ESI, EDI e EBX nella versione a 32 bit), ripristinandoli correttamente prima del ritorno al chiamante. Questo garantisce la stabilità dell'intero applicativo, evitando che la sovrascrittura accidentale di registri critici durante l'allocazione della memoria provochi crash al ritorno della funzione.

Questo controllo rigoroso previene i memory leak che, in un algoritmo che deve processare migliaia di query su dataset importanti, porterebbero rapidamente all'esaurimento delle risorse di sistema.

3.7 Interfaccia e integrazione Python

Al fine di garantire la massima usabilità l'intero sistema di ricerca K-NN è stato ingegnerizzato come un pacchetto Python nativo denominato Gruppo_Ferrari_DeFusco_Cuconato.

L'architettura del pacchetto segue gli standard di distribuzione definiti dai file pyproject.toml e setup.py, che permettono una compilazione 'on-the-fly' delle estensioni C direttamente sul sistema di destinazione.

L'integrazione core è stata realizzata tramite le Python C-API, utilizzando file di interfaccia (wrapper) che agiscono come 'ponte' tra la memoria gestita da Python e le routine a basso livello.

In particolare, i moduli quantpivot32, quantpivot64 e quantpivot64omp espongono una classe QuantPivot i cui metodi fit e predict incapsulano le chiamate alle funzioni ottimizzate in Assembly SSE2 e AVX2.

Questo approccio permette di gestire i dati tramite NumPy array (garantendo l'allineamento della memoria a 16 o 32 byte necessario per le istruzioni SIMD) senza rinunciare alle performance estreme del codice sorgente in linguaggio macchina. La struttura modulare del pacchetto, organizzata in sottocartelle con relativi file __init__.py, assicura infine una chiara separazione tra le diverse varianti architetturali, rendendo il software facilmente installabile tramite il gestore di pacchetti pip e pronto per essere utilizzato in ambienti di produzione ad alte prestazioni.

Il file knn_interface.py funge da punto di ingresso (entry-point) e ambiente di test per l'intero progetto. Lo script implementa un'interfaccia a riga di comando (CLI) basata sulla libreria argparse, permettendo all'utente di configurare dinamicamente tutti i parametri del problema: file di dataset e query (in formato .ds2 o .csv), numero di pivot (h), numero di vicini (k), bit di quantizzazione (x) e, soprattutto, la variante architetturale da utilizzare (32, 64, 64omp). A livello logico lo script si occupa di caricare i dati, orchestrare le diverse versioni, misurare e stampare i tempi di esecuzione.

Capitolo 4

Confronto dei tempi rispetto alle ottimizzazioni analizzate

In questo capitolo vengono analizzati i risultati ottenuti dai test prestazionali, confrontando le diverse implementazioni dell'algoritmo k-NN.

I dati, riassunti nella tabella seguente, permettono di valutare l'impatto reale della vettorizzazione Assembly e del parallelismo multi-thread rispetto alle implementazioni scalari in C.

Configurazione	Build (ms)	Query (ms)	Totale (ms)
32-bit Scalar (C)	157.00	6746.00	6903.00
32-bit SSE2 (Intrinsic)	114.00	3487.00	3601.00
32-bit SSE2 (Assembly)	**77.00**	400.00	477.00
32-bit SSE2 + OpenMP	**77.00**	**46.00**	**123.00**
64-bit Scalar (C)	158.00	6723.00	6881.00
64-bit AVX2 (Intrinsic)	109.00	2900.00	3009.00
64-bit AVX2 (Assembly)	**82.00**	631.00	713.00
64-bit AVX2 + OpenMP	**82.00**	**49.00**	**131.00**

** Valori più bassi indicano prestazioni migliori.*
Le celle evidenziate rappresentano la miglior prestazione all'interno del gruppo 32-bit o 64-bit.

I tempi di esecuzione riportati si riferiscono alle fasi di Build (costruzione delle strutture dati utili alle ottimizzazioni e la quantizzazione) e Query (ricerca dei k vicini, quindi l'algoritmo richiesto dal progetto).

L'analisi dei dati evidenzia miglioramenti radicali grazie all'uso del linguaggio Assembly e delle istruzioni SIMD.

Il passaggio dall'implementazione scalare in C a quella ottimizzata in Assembly (SSE2/AVX2) produce un salto prestazionale enorme:

1. Nella versione a 32-bit, la query passa da 6746 ms a soli 400 ms, con uno speedup di circa 17x;

2. La fase di Build vede il tempo dimezzato (da 157 ms a 77 ms), a conferma della maggiore efficienza nella gestione dei cicli di quantizzazione e memoria;

3. È interessante notare come l'implementazione Assembly sia significativamente più veloce della versione a "Intrinsic", dimostrando che la scrittura manuale del codice ha permesso una gestione dei registri e delle istruzioni (come HADDPS) più efficace di quella offerta dall'autovettorizzazione del compilatore.

L'introduzione di OpenMP porta le prestazioni a un livello superiore:

1. La configurazione 32-bit SSE2 + OpenMP raggiunge il tempo record di 46 ms nella fase di query;

2. Rispetto alla versione scalare iniziale (6746 ms), lo speedup totale per la query è di circa 146x. Questo risultato dimostra come l'algoritmo sia altamente parallelizzabile: mentre il SIMD accelera il calcolo della singola distanza, OpenMP permette di processare più query contemporaneamente.

4.1 Conclusioni

I test confermano che l'architettura a tre livelli (Filtro Pivot, Filtro Approssimato e Verifica Esatta) descritta nel Capitolo 1 è estremamente efficace. Lo scarto tra i tempi "Scalar" e "Assembly" indica che la stragrande maggioranza della potenza computazionale viene risparmiata grazie ai filtri binari e alla velocità delle istruzioni POPCNT e CMPPS implementate a basso livello.