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

Le ottimizzazioni richieste

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.

3.0.1 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.

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

3.0.3 Vettorizzazione e calcolo parallelo nei registri SIMD

Analizzando i file quantpivot32.nasm e quantpivot64.nasm, 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.

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 implementativo di rilievo è la gestione della riduzione orizzontale: dopo aver accumulato i quadrati delle differenze nei vari canali del registro vettoriale, il codice deve sommare questi valori parziali per ottenere lo scalare della distanza finale. Questo passaggio, critico per le prestazioni, è risolto tramite istruzioni di somma orizzontale o tramite una sequenza di "shuffle" e addizioni, minimizzando i tempi morti della CPU e preparando il dato per il confronto con la soglia di pruning.

3.0.4 Ottimizzazione del Controllo di Flusso e Loop Unrolling

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.

Inoltre, l'interazione tra il codice Assembly e la logica di pruning descritta precedentemente è gestita in modo estremamente rigido: le routine Assembly restituiscono il controllo al chiamante C non appena viene superata la soglia di distanza massima, o 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.

3.0.5 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.

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 POPCNT (Population Count). Quest'ultima istruzione hardware conta il numero di bit impostati a 1 in un registro in un singolo ciclo di clock.

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.

3.0.6 Gestione dei buffer temporali

All'interno del file quantpivot32.nasm si può notare che la gestione di queste maschere binarie richiede memoria temporanea supplementare. Per evitare

la frammentazione della memoria o rallentamenti dovuti a continue chiamate di sistema, il codice utilizza le macro getmem e fremem.

Queste macro agiscono come un'interfaccia sicura verso le funzioni di gestione dei blocchi di memoria definite in C.

La loro importanza è duplice: da un lato garantiscono che lo spazio allocato per le maschere sia sufficiente a contenere i dati vettoriali allineati, dall'altro assicurano che ogni blocco di memoria temporanea venga restituito al sistema non appena la funzione Assembly termina il suo compito.

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.