Architetture e Programmazione dei Sistemi di Elaborazione
Progetto a.a. 2020/21
"Polynomial Regression and Stochastic Gradient Descent"
in linguaggio assembly x86-32+SSE, x86-64+AVX e openMP

Relazione Gruppo 15: Davide Iacopino 224798, , Francesco Palumbo 189891, Giuseppe Tripodi 224785, Maria Rita Velardo 189686

Parte 1: Conversione

La conversione del dataset è stata realizzata con la funzione `void convert_data(params* input)`. Essa invoca una serie di funzioni accessorie, tra cui `int numeroComb(int deg, int d, int *numComb)`, la quale calcola il numero di colonne della matrice *x*,* output della conversione. Questo numero (indicato con *t*) corrisponde alla somma, per *h* che varia da *0* a *deg*, del numero di combinazioni con ripetizione di *d* oggetti (le *d* componenti di *x*) a gruppi di *h*.

$$comb_h = \frac{(d+h-1)!}{h!\,(d-1)!}$$

Per evitare il calcolo dei fattoriali, la funzione sfrutta la proprietà che

$$comb_{h+1} = comb_h \cdot \frac{d+h-1}{h}$$

Si noti inoltre che

$$comb_0 = \frac{(d-1)!}{(d-1)!} = 1, \qquad comb_1 = comb_0 \cdot \frac{d+1-1}{1} = d$$

Dopo aver invocato questa funzione, in `convert_data` si alloca la matrice *x**, alla quale si aggiungono alcune colonne fittizie (padding) al fine di ottenere un numero di componenti pari ad un multiplo di 4. Ciò consente di ottenere l'allineamento di ogni riga (la matrice è memorizzata per righe), utile per la parte in linguaggio assembly.

Si prosegue inizializzando alcuni elementi della matrice *x**, corrispondenti alla prima componente di ogni riga (sempre pari a 1) e alle componenti di padding (poste a 0). A tal proposito si usa una funzione `void init_padd(VECTOR ar, MATRIX x, int mt, int n)`, implementata in assembly, che sfrutta la loop vectorization andando a inizializzare parallelamente le componenti di padding di una riga ed il primo elemento della riga successiva, sfruttando la memorizzazione per righe e, quindi, il fatto che questi valori siano memorizzati in modo contiguo. Eventuali altre celle inizializzate (qualora il numero di colonne di padding sia inferiore a 3) saranno poi sovrascritte successivamente.

In questa funzione, così come in alcune delle successive, non è stato implementato il loop unrolling poiché, a seguito di alcuni test, non si è riscontrato un miglioramento nelle prestazioni con l'applicazione di questa tecnica agli algoritmi realizzati.

Successivamente si costruisce la matrice *mComb*, in cui ogni riga rappresenta una combinazione corrispondente ad una componente di una riga di *x**. Più precisamente, si costruisce solo la matrice delle combinazioni di grado massimo, sfruttando il fatto che le combinazioni di grado inferiore si possano ottenere scorrendo da destra a sinistra ogni riga di *mComb* e leggendo un numero di elementi pari a `h`.

Per costruire *mComb*, è invocato il metodo `void combinazioni2(int* ris, int h, int d)`, che calcola le varie combinazioni sfruttando il fatto che esse siano la traduzione in base *d* di una serie di interi crescenti, alcuni di essi consecutivi (per evitare ripetizioni a volte bisogna "saltare" alcuni numeri). Questa funzione ne invoca altre accessorie, come `int creaNum(int cifra, int h)`, che crea un numero di h

cifre uguali e pari alla cifra passata in input, e `int cambioBase(int i, int bp, int ba)`, che effettua il cambio di base del numero passato in argomento.

Costruita *mComb*, `convert_data` ha tutto il necessario per scrivere la matrice *x**, operazione effettuata con le invocazioni di `void inserisciComb(int n, int nCH, int degh, int indice, int* mComb, MATRIX xast, MATRIX x, int mt, int d, int deg)`, funzione implementata in linguaggio assembly. Anche qui si sfrutta la tecnica della loop vectorization per calcolare in parallelo 4 elementi della matrice *x**, leggendo 4 combinazioni elemento per elemento e moltiplicando in un registro accumulatore i corrispondenti elementi dell'osservazione corrente di *x*. Questo algoritmo è ripetuto per ognuna delle osservazioni; la scrittura della matrice, dunque, procede per righe.

La descrizione fatta finora dell'algoritmo in C si riferisce sia alla versione a 32 bit sia alla versione a 64 bit, in quanto, a livello di codice, tra le due versioni non è presente alcuna differenza.

Data la carenza di registri ad uso generale a 32 bit si è provveduto ad utilizzare alcuni registri vettoriali al posto di essi. Ad esempio, dal momento che la funzione ha un triplo ciclo innestato, sarebbero serviti ben sei registri per contenere le variabili interne dei cicli. Bensì sono stati utilizzati i due registri `XMM6`(che contiene le variabili, opportunamente convertite in float, che servono per iterare nel ciclo) e `XMM7`(che contiene le variabili per verificare le condizioni di uscita), al posto di una normale `CMP` bisogna utilizzare una `COMISS` che confronta gli scalari più bassi di due registri `XMM` e aggiorna le `EFLAGS` di conseguenza (è necessario dunque spostare tramite degli shift circolari le variabili interessate nel primo elemento del registro vettoriale).
Altri esempi di utilizzo dei registri vettoriali in luogo di quelli ad usi generale sono nella gestione dei puntatori alle matrici *x**, *x* ed *mComb*.

Con queste due ottimizzazioni in assembly si è ottenuto un tempo Conversion time = 0.000601 secs, quasi di un ordine di grandezza inferiore a quello ottenuto nella versione esclusivamente in C Conversion time = 0.003 secs.

Parte 2: Regressione

L'obbiettivo della regressione è quello di calcolare iterativamente la stima del vettore theta attraverso il meccanismo della discesa stocastica del gradiente

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta C(\theta_t, x_i, y_i)$$

con lo scopo di minimizzare l'errore quadratico medio

$$\frac{1}{n}\sum_{i=1}^{n}\left(y_i - f\left(x_{i_1}\vartheta\right)\right)^2$$

Abbiamo pensato di dividere l'algoritmo principale, che si occupa di calcolare la regressione, in due varianti distinte:

- SGD
- Adagrad

L'algoritmo principale seleziona la variante da eseguire verificando quale sia il valore della variabile in input "adagrad".

Procediamo con la descrizione delle due funzioni in C, riferendoci sia alla versione a 32 bit sia alla versione a 64 bit, in quanto, a livello di codice, tra le due versioni non è presente alcuna differenza.

La variante SGD ad ogni iterazione estrae dei batch dalla matrice uno per volta, ognuno formato da k righe ad eccezione dell'ultimo batch che potrebbe corrispondere ad una sottomatrice con un numero di righe minore di k (in particolare estraendo le righe attraverso un for i con valori che partono da zero e vengono incrementati di k l'ultimo batch avrà un numero di righe che è il valore intero minore tra k e n-i). Una volta

estratto un batch, questo viene passato alla funzione calcolaTheta_sgd che in primo luogo richiama la funzione sommatoriaSGD. La funzione sommatoria SGD esegue l'operazione

$$\sum_{j=i}^{i+v-1} \left(\langle \theta, x_j \rangle - y_j\right) \cdot x_j$$

attraverso un ciclo for che si muove sulle righe j del batch e per ogni riga calcola il prodotto scalare tra la stima di theta calcolata per i batch precedenti e la riga j-esima della matrice x^*, sottrae a questo la componente j-esima di y, lo scalare risultante viene moltiplicato alla riga j-esima della matrice x^* infine si aggiorna il vettore sommatoria sommando le componenti del vettore risultante dal calcolo precedente. Con il vettore restituito da sommatoriaSGD, nella funzione calcolaTheta_sgd, si prosegue effettuando il calcolo del vettore theta corrente

$$\theta_{t+1} = \theta_t - \frac{\eta}{v} \sum_{j=i}^{i+v-1} \left(\langle \theta, x_j \rangle - y_j\right) \cdot x_j$$

Il valore di theta finale è quello che viene restituito al termine delle iterazioni (al termine del while(it< iter)).

Passiamo adesso all'ottimizzazione Adagrad. Anche questa variante ad ogni iterazione estrae i batch della matrice uno per volta ma il calcolo che esegue è differente. La prima differenza la si trova nella funzione calcola_adagrad dove viene aggiunta, rispetto alla versione precedente, una matrice G di k righe e t colonne. La seconda differenza la si ha nel calcolo della sommatoria

$$\sum_{j=i}^{i+v-1} \frac{g_j}{\sqrt{G_j + \varepsilon}}$$

Dove

$$g_j \coloneqq \left(\langle \theta, x_j \rangle - y_j\right) \cdot x_j$$

$$G_j \coloneqq \sum_{t=1}^{it} g_j^2$$

calcolaThetaAdagrad chiama la funzione sommatoriaAdagrad che calcola per ogni j del batch il vettore $g_j$ed aggiorna la matrice G. Il calcolo della sommatoria è effettuato dal metodo sommatoriaInterna richiamato da sommatoriaAdagrad. Infine, con il vettore risultante dalla sommatoria calcolaTheta_adagrad può effettuare il calcolo di theta corrente come

$$\theta_{t+1} = \theta_t - \frac{\eta}{v} \sum_{j=i}^{i+v-1} \frac{g_j}{\sqrt{G_j + \varepsilon}}$$

Il valore di theta finale è quello che viene restituito al termine delle iterazioni (al termine del while(it< iter)).

Dopo la compilazione del codice C, abbiamo effettuato i test e siamo arrivati ad avere i seguenti risultati:

- 32 SGD →Regression time = 2.652 secs
- 32 ADAGRAD →Regression time = 18.028 secs
- 64 SGD →Regression time = 2.464 secs
- 64 ADAGRAD →Regression time = 14.649 secs

Abbiamo deciso di collaborare nella scelta delle parti di codice da scrivere in assembly per ottimizzare il codice. Le parti che abbiamo deciso di ottimizzare, sia per la versione a 32 bit sia per la versione a 64 sono:

- Funzione che calcola il prodotto scalare (sia per la versione sgd sia per la versione adagrad)
- Funzione che calcola la sommatoria in sommatoriaSGD (solo per la versione sgd)
- Funzione che calcola il valore aggiornato di theta (sia per la versione sgd sia per la versione adagrad)
- Funzione che calcola la sommatoria interna (solo per la versione adagrad)
- Funzione che inizializza un vettore con il valore che viene passato come parametro (presente sia nella versione sgd sia nella versione adagrad)

Abbiamo scelto di riportare in assembly queste funzioni in quanto corrispondono a punti critici del programma la cui scrittura in assembly permette di ottimizzare il codice inserendo ottimizzazione ti tipo ILP-based e SIMD-based.

Nella stesura in linguaggio assembly della funzione che calcola la sommatoria in sommatoriaSGD e della funzione che calcola il valore aggiornato di theta ci siamo accorti che entrambe le funzioni effettuano il seguente calcolo:

Vettore_destinazione = vettore_destinazione + fattore_moltiplicativo * vettore_sorgente (1)

Nel calcolo della sommatoria sgd abbiamo:

$$Sommatoria = sommatoria + delta * x^*_j$$

con:

- sommatoria = vettore_destinazione;
- x^*_j = vettore_sorgente;
- delta = prodottoScalare(theta, x^*_j) – y_j = fattore_moltiplicatico;

mentre nel calcolo aggiornato di thetala formula è:

$$stimaTheta = stimaTheta + (– eta/v) *sommatoria$$

con:

- stimaTheta = vettore_destinazione
- sommatoria = vettore_sorgente
- -eta/v = fattore_moltiplicativo

La scelta è stata, quindi, quella di scrivere un'unica funzione assembly, chiamata calcolaVettore, che effettua il calcolo (1).

FUNZIONE calcolaVettore

L'algoritmo creato che implementa questa funzione è lo stesso in entrambe le versioni (32 e 64). Le uniche differenze presenti oltre ad i diversi registri usati consistono nella scelta di un diverso fattore di unrolling dettato da limiti costruttivi.

Il calcolo da effettuare si presta bene ad un'ottimizzazione SIMD-based, ogni operazione svolta quindi lavora con più elementi per volta.

Abbiamo notato, inoltre, che è possibile srotolare il ciclo per eseguire più operazioni alla volta effettuando quindi un loop unrolling.

Per la scelta del fattore di unrolling abbiamo preso in esame diversi possibili valori, valutando con quale fattore si riuscisse ad ottimizzare maggiormente l'esecuzione.

Procedo con la descrizione delle diverse tempistiche ottenute nell'implementazione a 32 bit.

Con un fattore di unrolling pari a 4 i risultati che otteniamo sono i seguenti.

LoopUnrolling = 4:

- 32 SGD →Regression time = 1.093075 secs
- 32 ADAGRAD →Regression time = 1.231962 secs

Tuttavia, avendo altri registri disponibili, abbiamo provato ad incrementare l'unrolling.

LoopUnrolling = 6:

- 32 SGD →Regression time = 0.856493 secs
- 32 ADAGRAD →Regression time = 0.965118 secs

Avendo riscontrato un effettivo miglioramento, abbiamo optato per questo specifico fattore di unrolling in questo metodo.

Analogo discorso con la versione a 64 bit di questo algoritmo, dove i test hanno mostrato un miglioramento più netto usando un fattore di unrolling pari ad 8.

LoopUnrolling = 4:

- 64 SGD →Regression time = 0.484732 secs
- 64 ADAGRAD →Regression time = 1.082251 secs

LoopUnrolling = 8:

- 64 SGD →Regression time = 0.311021 secs
- 64 ADAGRAD →Regression time = 0.649683 secs

FUNZIONE prodottoScalare

Per quanto riguarda la funzione che calcola il prodotto scalare, usata nelle funzioni "sommatoriaSGD" e "sommatoriaAdagrad", si presta molto bene ad essere parallelizzata in quanto può fare prodotti multipli su elementi diversi del vettore.

Anche in questo caso abbiamo svolto numerosi test per determinare il fattore di unrolling che meglio si adatta a questo algoritmo, ottenedo come risultato che le prestazioni migliori si ottengono con un fattore di unrolling uguale a sei nella versione a 32 bit e pari ad 8 in quella a 64 bit.

FUNZIONEsommatoriaInterna

La funzione calcola il valore della sommatoria interna presenta nella variante adagrad dell'algoritmo. Abbiamo implementato in assembly anche questa funzione ottenendo un notevole miglioramento dei tempi di esecuzione, miglioramento che si è poi accentuato notevolmente a seguito dell'inserimento delle ottimizzazioni di tipo ILP e SIMD.

L'algoritmo implementato si occupa di tradurre in assembly il seguente frammento di codice:

```
for(int v = 0; v <tC; v++ ) {
```

```
gjv = app * Dbatch[j*tC + v];
G[j*tC + v] += powf(gjv,2);
ret[v] += gjv / sqrtf(G[j*tC + v]);
}
```

Anche questo particolare algoritmo consente una parallelizzazione parziale delle operazioni svolte all'interno del ciclo.

FUNZIONEinizializza

Avendo notato che un'operazione molto frequente nel nostro algoritmo era quella di azzerare o comunque di inizializzare matrici o vettori ad uno specifico valore, abbiamo pensato di creare una funzione che si occupa di fare ciò.

Per ottenere anche un vantaggio prestazionale abbiamo implementato in assembly questa funzione, lavorando quindi con più elementi per volta.

La funzione modifica direttamente il valore delle aree di memoria interessate inizializzando il valore ad un particolare parametro ricevuto dalla funzione. Abbiamo deciso di implementare questo algoritmo usando esclusivamente le ottimizzazioni derivanti dall'uso dei registri SSE o AVX, in questo modo possiamo modificare il valore di più celle della matrice o del vettore alla volta.

Effettuando le modifiche direttamente in memoria non abbiamo ritenuto necessario l'uso di ottimizzazioni SIMD-based, in quanto avremmo rischiato non solo di non avere effettivi miglioramenti delle prestazioni ma anche di generare diversi tipi di errori nell'esecuzione.

Questa scelta è stata avvalorata da alcuni testi che hanno mostrato come l'uso del loop Unrolling non ha un effettivo riscontro sui tempi di esecuzione che rimangono fondamentalmente invariati se non in alcuni casi anche peggiori.

Senza LoopUnrolling:

- 32 SGD → Regression time = 0.875652 secs
- 32 ADAGRAD →Regression time = 0.974689 secs

LoopUnrolling = 4:

- 32 SGD → Regression time = 0.916493 secs
- 32 ADAGRAD → Regression time = 0.965118 secs

Abbiamo fatto le stesse osservazioni anche nella versione a 64 bit ottenendo gli stessi risultati.

Parte 3: OpenMP

Per implementare la versione OpenMP abbiamo cercato un punto particolare del codice tale da avere prestazioni migliori attraverso l'inserimento della direttiva openMP.
Abbiamo riscontrato che fosse opportuno inserire la direttiva all'interno della funzione convert_data(params* input), prima del for che si occupa di calcolare la matrice x*.
Per effettuare ciò, rispetto alla versione che non fa uso di openMP, abbiamo considerato che fosse necessario estrarre dalla funzione inserisciComb implementata in codice assembly il ciclo for che scorre le osservazioni, che abbiamo portato nel codice C, applicando su esso la parallelizzazione openMP.
La scelta è stata quella di inserire la direttiva solo nella parte in cui viene effettuata la conversione (convert_data(params* input)), ritenendo non possibile farlo per la parte in cui facciamo la regressione.

Le modifiche apportate con l'introduzione della direttiva sono uguali sia per la versione a 32 bit che per la versione a 64 bit.
L'unica differenza tra la versione a 32 bit e la versione a 64 bit la si ha nella scelta del numero di threads e dopo una serie di test abbiamo constatato che il numero di threads utilizzare per avere le migliori prestazioni sono 4 per la versione a 32 bit e 3 per la versione a 64 bit.