**Relazione del Progetto di Architetture e Programmazione dei Sistemi di Elaborazione**

**Gruppo : Arcuri Francesco, Mangione Fabrizio, Morrone Stefano, Perna Stefano (PM)**

Indice

[Sommario 1](#_Toc93574426)

[Premessa 1](#_Toc93574427)

[Codice in C 1](#_Toc93574428)

[Ottimizzazioni C 3](#_Toc93574429)

[Assembly 4](#_Toc93574430)

[Ottimizzazioni 4](#_Toc93574431)

[x86-32+SSE 5](#_Toc93574432)

[Baricentro 6](#_Toc93574433)

[Calcola valore F 11](#_Toc93574434)

[Movimento individuale 13](#_Toc93574435)

[Calcola y 13](#_Toc93574436)

[Calcola f\_y 15](#_Toc93574437)

[Altre funzioni assembly 15](#_Toc93574438)

[x86-64+AVX 16](#_Toc93574439)

# Premessa

Il presente elaborato si pone lo scopo di descrivere il progetto didattico assegnato durante il corso di “Architetture e Progettazione dei Sistemi di Elaborazione”, andando a commentare e descrivere le varie fasi di sviluppo e di evoluzione del codice.

Si prende come assunto la conoscenza della traccia assegnata, inoltre, si fa notare che l’elaborato di relazione è stato prodotto parallelamente allo sviluppo del codice, pertanto, riporta le fasi del progetto attraverso un una visione locale e non su una visione complessiva del risultato finale.

# Codice in C

Il primo passo del progetto, come suggerito dal docente stesso, è stato quello di decidere la modalità di rappresentazione delle matrici (per riga o per colonna), ovvero come destrutturare la logica bidimensionale di una matrice per rappresentarla direttamente su un unico vettore. La scelta effettuata di comune accordo, che appariva congeniale per le esigenze del progetto e per le operazioni da effettuare, è stata la rappresentazione per riga, in quanto consente di avere in sequenza tutte le coordinate ordinate per ogni singolo pesce su un unico vettore. Questa scelta è stata ritenuta la soluzione migliore in quanto le operazioni che vengono effettuate lavorano in sequenza sui pesci, andando a eseguire calcoli sulle coordinate di ogni singolo pesce.

Il progetto prosegue con lo sviluppo dell’algoritmo “fish school search” attraverso un codice in C, utilizzando i file già messi a disposizione dai docenti, è stata elaborata la struttura dell’algoritmo sul file “fss32c.c”. Per spezzare la complessità dell’algoritmo viene realizzato il codice che ne implementa la logica generale nella funzione “void fss(params\* input){…}”. La struttura principale all’interno della funzione fss è realizzata utilizzando funzioni che sono da sviluppare e che sono state assegnate ai vari membri del gruppo.

Immagine che contiene testo

Descrizione generata automaticamenteOltre alle variabili in input è nata la necessità di inserire altre variabili, per memorizzare dati globali che servono al di fuori della singola iterazione e per memorizzare i vari risultati parziali. Di seguito sono riportate le variabili aggiunte per l’esecuzione dell’algoritmo.

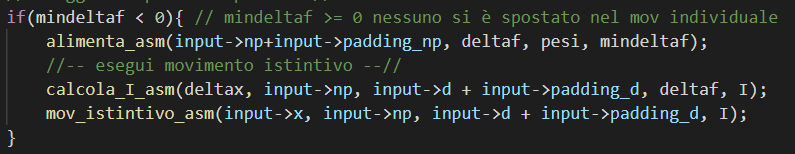
Si passa ora al cuore centrale dell’algoritmo che viene ripetuto fino ad arrivare al numero massimo di iterazioni indicate nell’input. Nelle varie iterazioni si ripetono i diversi passi dell’algoritmo, come viene mostrato nel codice C seguente.

Immagine che contiene testo

Descrizione generata automaticamente

La divisione in funzioni ha permesso sia di spezzare la complessità e rendere più leggibile l’algoritmo (e) che di dividere il codice da realizzare tra i componenti del gruppo.

La scrittura della prima versione delle varie funzioni è stata relativamente rapida, anche grazie all’aiuto dato dalla traccia dettagliata; infatti, è stato sufficiente seguire le varie operazione aritmetiche dell’algoritmo per realizzare le varie funzioni in C. Una volta combinato il tutto, il programma chiaramente non funzionava. Si è quindi resa necessaria una fase di debug, in cui sono stati identificati gli errori nel codice. Tra gli errori principali identificati c’è stato quello relativo all’utilizzo eccessivo dei numeri random, che andava oltre quelli messi a disposizione dai docenti e la difficoltà principale è stata identificare il problema piuttosto che la sua risoluzione. Qualche altro piccolo errore è stato rilevato nell’accesso alle strutture dati, non per quanto concerne la logica, ma nell’utilizzo del linguaggio C. Completata questa fase si è arrivati a una prima risoluzione corretta del problema.

In seguito, è stata aggiunta un’ottimizzazione sul codice in C che è la seguente:

se nessun pesce si è spostato, tutti i deltaf e tutti i deltax sono pari a zero e quindi le operazioni inserite nel blocco dell’if perdono di significato, in quanto non porterebbero a nessuna variazione. Si decide di sostenere il costo di questo confronto per ogni iterazione in quanto le operazioni nell’*if* sono molto onerose è il poterle evitare anche solo poche volte si è visto porta un beneficio ai tempi di esecuzione.

## Ottimizzazioni C

Già nel codice C si sono trovate delle soluzioni allo scopo di ottimizzare l’esecuzione dell’algoritmo. Come prima cosa all’interno delle funzioni che definiscono le fasi principali dell’algoritmo sono state definite nuove funzioni, ad esempio, per il calcolo di numeratore e denominatore (il tutto ha come fine ultimo trovarsi una serie di funzioni facilmente trasformabili in assembly).

Immagine che contiene testo

Descrizione generata automaticamenteUn serie di soluzioni a nostro avviso interessante sono quelle utilizzate nella funzione di “Movimento Individuale” di seguito viene riportato parte del codice.

Si introduce una piccola leggenda per semplificare la lettura del codice

|  |  |
| --- | --- |
| X | Matrice coordinate correnti dei pesci |
| Y | Matrice coordinate dei pesci a seguito del movimento individuale |

Come prevede l’algoritmo stesso, i pesci ad essere spostati a seguito del movimento individuale sono solamente quelli che hanno acquisito una posizione migliore (in funzione del tipo di risultato che si sta cercando max/min). L’idea è quella di limitare il numero di scritture sull’array che contiene la posizione dei pesci. Quindi, se la maggioranza dei pesci si sono spostati, allora, la matrice Y diventerà la nuova matrice X attraverso uno scambio di puntatori. Ottenere questo risultato è possibile grazie al dato “deltaf” (delta F) di ogni pesce, infatti, il pesce i-esimo avrà “deltf” pari a zero se il pesce non si è spostato. Il numero di pesci spostato viene calcolato a ogni movimento individuale del singolo pesce, se il pesce viene spostato questo contatore è incrementato.

Un altro aspetto ad essere stato preso in considerazione e valutato è il seguente: in una fase preliminare si era pensato di aggiornare il valore “f\_min” (valore della funzione minimo) ad ogni spostamento, avendo a disposizione il valore di “f” corrente a seguito dello spostamento, andando confrontare il valore minimo corrente (memorizzato in una valribile) con quello appena calcolato, piuttosto che calcolarlo al termine delle iterazioni con un ulteriore ciclo. Questa operazione risultava molto onerosa a causa della presenza di numerosi salti condizionati ripetuti per ciascuna iterazione. Perciò si è deciso di calcolare il valore ottimo al termine delle iterazioni, così da avere i medesimi salti condizionati per una unica funzione che itera sui pesci, invece che per ciascuna delle funzioni presenti e per ciascuna iterazione, riuscendo ad ottenere in questo modo una ottimizzazione del codice.

Nelle altre funzioni non si è seguita nessuna ottimizzazione particolare, se non quella di cercare di limitare gli accessi in memoria e limitare le catene di puntatori. Su consiglio del docente si è sempre cercato di utilizzare funzioni *void* e di passare come parametro il valore di ritorno (dove necessario).

# Assembly

## Ottimizzazioni

Il **parallelismo a livello istruzione** (ILP) è una misura delle istruzioni in un programma, che possono essere eseguite in un calcolo parallelo.

La ricerca di codice parallelo a livello di istruzioni è una priorità nei moderni microprocessori che sono dotati di molte unità di calcolo e usualmente seguono una struttura a pipeline; quindi, l'individuazione e lo sfruttamento delle istruzioni eseguibili in parallelo permette di utilizzare le unità funzionali dei processori innalzandone le prestazioni.

Il **parallelismo MIMD** (Multiple Instruction stream Multiple Data stream) è un'architettura parallela in cui, unità di elaborazione distinte eseguono simultaneamente elaborazioni su flussi di dati diversi. Esempi di questa architettura sono i sistemi multiprocessore o i computer collegati in rete per le elaborazioni distribuite. L’idea è quella di individuare dei problemi che sono naturalmente risolvibili in maniera parallela e farli eseguire su core diversi in parallelo. Nel caso di studio, sì farà uso della direttiva **openMP**, ovvero un modello scalabile e portabile che fornisce al programmatore un'interfaccia semplice e flessibile per sviluppare applicazioni di calcolo parallele. Tale direttiva sarà apposta sull’operazione o sui blocchi di operazioni parallelizzabili nel codice C.

Per sopperire alla crescente disparità tra la velocità della CPU e quella della RAM, nel corso degli anni sono state introdotte una o più memorie cache. Per ottenere le massime prestazioni possibili della CPU, per un determinata operazione, i dati necessari per la corretta esecuzione devono essere caricati all’interno della memoria cache il più rapidamente possibile e il minor numero di volte possibile. Non gestire la cache in modo efficiente comporterà accessi alla memoria RAM non necessari, nonché tempi di esecuzione più lunghi.

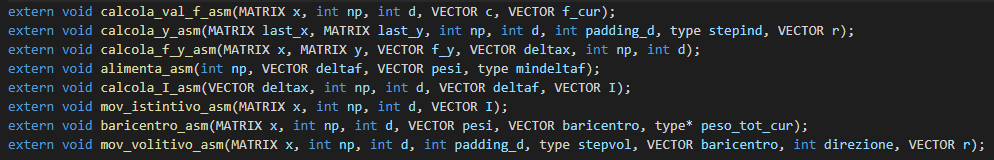
Nel caso di studio non si è resa necessaria un’ottimizzazione esplicita relativa alla memoria cache, poiché i calcoli effettuati sono intrinsecamente ottimizzati per questo particolare tipo di memoria. Le matrici sono memorizzate come array unidimensionali, i cui elementi sono memorizzati in ordine sequenziale nella RAM, inoltre l’accesso a tali elementi segue il criterio di memorizzazione. Le azioni predefinite dal gestore della memoria sono ottimali per questo calcolo e non richiedono alcuna modifica.

Nell’ottimizzare il codice sono state utilizzate diverse tecniche, tra queste troviamo:

* **Loop Unrolling**: Consiste nella riduzione del numero di salti condizionati, andando ad eseguire operazioni ripetute in un ciclo, in blocchi di operazioni su un gruppo di elementi del ciclo, così da ridurre le ripetizioni del ciclo stesso. Questa tecnica consente un’ottimizzazione del codice, ma comporta un utilizzo maggiore dei registri.
* **Loop Vectorization**: Consiste nell’effettuare operazioni su più operandi contemporaneamente, questo è possibile grazie alla dimensione dei registri (dipendenti dall’architettura) e a operazioni specifiche per operare con questi registri. Con questa architettura (registri a 128 bit) e il tipo float (32 bit) come tipo di dati trattati ci permette di avere sui registri XMM(i) quattro elementi la volta. Grazie a questi registri è possibile sia leggere e scrivere quattro elementi per volta ed effettuare operazioni in contemporanea su di essi.

# x86-32+SSE

Terminata fase di stesura dell’algoritmo in C, si è passati alla fase di ottimizzazione delle funzioni scritte in linguaggio ad alto livello mediante la traduzione di queste in codice Assembly. La prima versione implementata fa riferimento all’architettura “x86-32+SSE”.

L’impiego del linguaggio di programmazione a basso livello assembly ha richiesto la modifica di alcune unità di codice C. Un esempio è la definizione di funzioni “extern” come mostrato di seguito: 

Durante lo sviluppo delle varie funzioni in Assembly si è deciso di eseguire il *for-loop* sui “pesci” direttamente nel codice a basso livello al fine di raggiungere maggiore efficienza. Questa scelta non era però compatibile con l’utilizzo della direttiva openMP.

Si è quindi realizzato per le funzioni:

* calcola\_val\_f\_asm
* calcola\_y\_asm
* calcola\_f\_y\_asm
* mov\_volitivo\_asm
* mov\_istintivo\_asm

un’ulteriore funzione con la medesima intestazione seguita dall’aggiunta di “omp”. Una intestazione siffatta, indica che la funzione presa in esame supporta openMP, che il *for-loop* sui pesci è realizzato nel codice C e che, di conseguenza, la funzione è invocata una volta per ogni pesce.

Le altre funzioni, che precedentemente non sono state citate, sono comunque state riprodotte con una versione identica, ma con standard di nomenclatura differenti al fine di trovare coerenza con la procedura compilativa del docente.

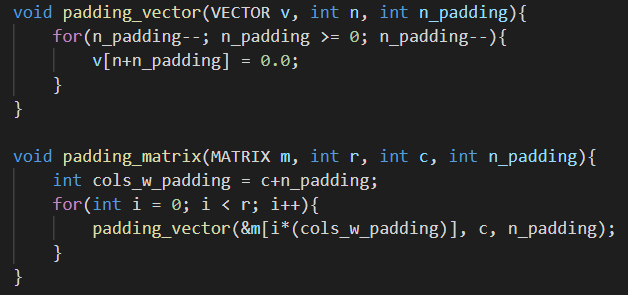
Immagine che contiene testo

Descrizione generata automaticamente

La fase successiva è caratterizzata dalla realizzazione di funzioni ottimizzate al fine di raggiungere la massima velocità di calcolo e limitare gli accessi in memoria, naturalmente con i limiti imposti dal numero di registri.

Come si può notare da queste schermate alcune funzioni prevedono un parametro *padding\_d* che permette l’impiego dell’operazione MOVAPS (allineate) in linguaggio di basso livello. Questo consente, nel caso in cui il resto della divisione per quattro fosse diverso da zero, comunque il corretto utilizzo della precedente operazione. Per colmare il buco dovuto all’assenza di elementi che rendano la dimensione del vettore multiplo di 4, sono stati inseriti degli zeri che non alterano la semantica delle operazioni in cui sono presenti, dovuta alla propria neutralità.

In altre funzioni, in cui non è necessario accedere ai numeri random, il padding è già compreso nel parametro d. La scelta di divederlo in alcune funzioni e per evitare di utilizzare numeri random del file del docente per elementi di padding.



Il padding è stato effettuato su:

1. Vettori, il cui numero di elementi non risulta essere un multiplo di 4.
2. Matrici, sul numero di colonne (numero di coordinate) seguendo la stessa logica. Si ottiene così un insieme di elementi che supporta la “mov allineata”.

Le vari funzioni assembly sono realizzate con parametri (costanti) contenuti nel “*section .data*” che identificano diversi fattori. L’inserimento di tali costanti rende il codice di più facile lettura.

## Immagine che contiene testo Descrizione generata automaticamenteBaricentro

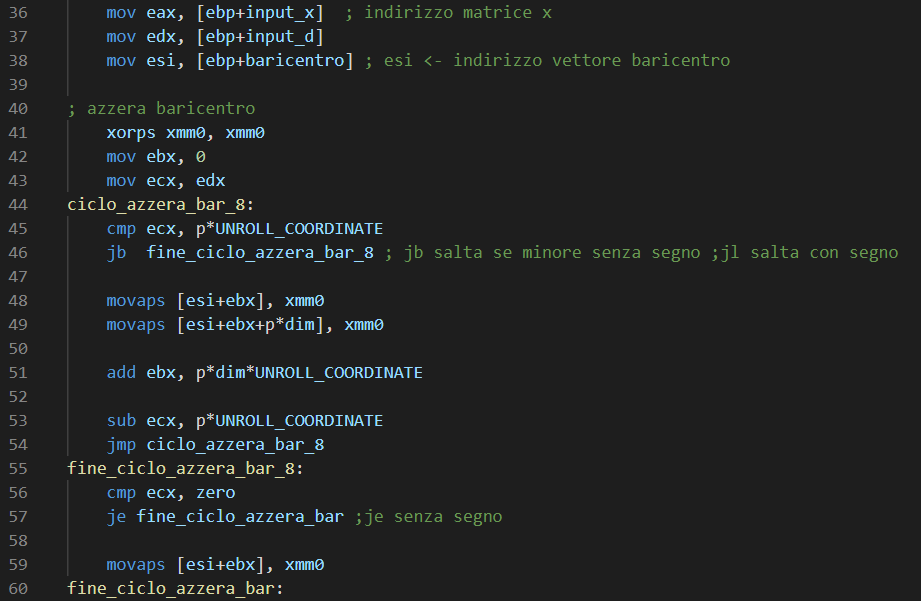
Durante la stesura del codice C si è riflettuto sull’eventuale divisione del calcolo del numeratore e del denominatore in due funzioni separate. La traduzione di tali operazione in Assembly ha mostrato che le due sub-funzioni avrebbero analizzato dati comuni annullando qualsiasi vantaggio di tipo prestazionale.

Immagine che contiene testo

Descrizione generata automaticamenteSono state definite delle costanti di offset nel “section .tex”, che rispecchiano l’ordine dei parametri della funzione chiamante in C. Tali offset rappresentano la posizione progressiva rispetto l’ebp, della chiamata di funzione nello stack.

Questa tecnica è stata riproposta per tutte le funzioni in Assembly, così da garantire maggiore leggibilità del codice; (la trattazione di questo aspetto non è ripetuta per le funzioni successive).

|  |  |
| --- | --- |
| **eax** | Indirizzo della matrice X |
| **ebx** | contatore pesci (indica il corrente) |
| **ecx** | contatore delle coordinate del pesce corrente |
| **edx** | d – numero di coordinate (\* dim – dimensione degli operandi) |
| **esi** | Indirizzo al vettore baricentro |
| **edi** | np – numero di pesci |



Come si può notare il codice contiene diversi commenti utili, sia per la lettura dello stesso, che per la correzione in caso di errori. Di seguito saranno riportate le osservazioni principali sul codice sovrastante (si seguirà lo stesso schema per le porzioni a seguire).

Alla riga 44 inizia un ciclo, per compiere delle operazioni di lettura dei dati della memoria e di azzeramento, poiché per calcolare il numeratore abbiamo bisogno del supporto della memoria principale (RAM). Nello specifico, la locazione di memoria che andiamo ad utilizzare è quella allocata per il baricentro. Si utilizza temporaneamente questa locazione di memoria per salvare il peso totale (calcolato), al termine non ci resta che accedere in memoria e dividere il numeratore per il peso totale.

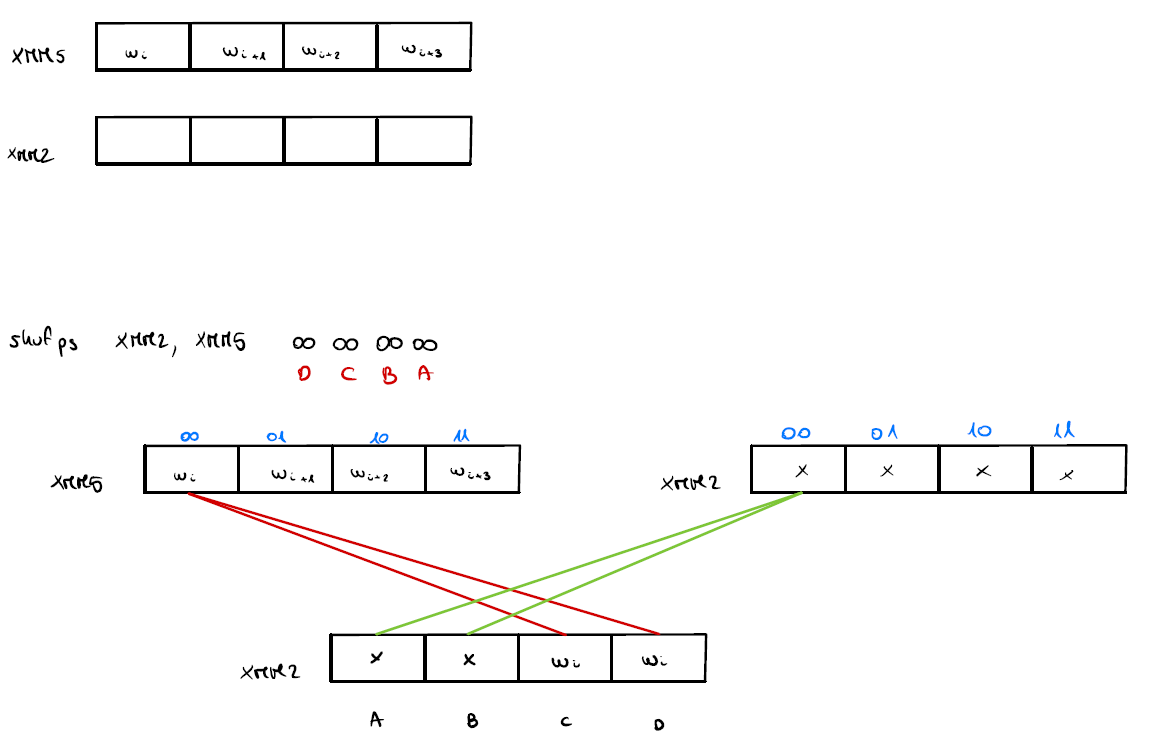
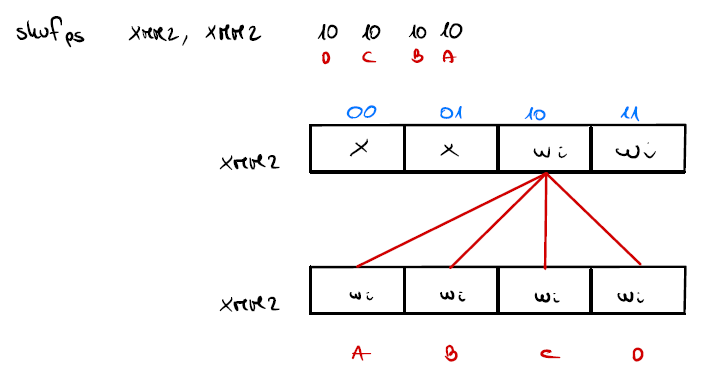
Nella riga 45 troviamo la costante “p” che sta per packed, ovvero ci indica quanti elementi per volta vengono letti dalla memoria, in questo caso quattro float. “UNROLL\_CORDINATE” indica le volte che viene effettuata l’operazione di unroll. Applicando questa tecnica si riduce il numero di salti condizionati del loop che verificano che l'indice sia all'interno della lunghezza del vettore. Il problema in questo caso è che bisogna individuare un corretto valore di unroll, solitamente si stima. In generale dipende dalla macchina dove si esegue il codice, inoltre il suo valore dovrebbe essere un sottomultiplo della dimensione del vettore evitando così eventuali cicli di resto.

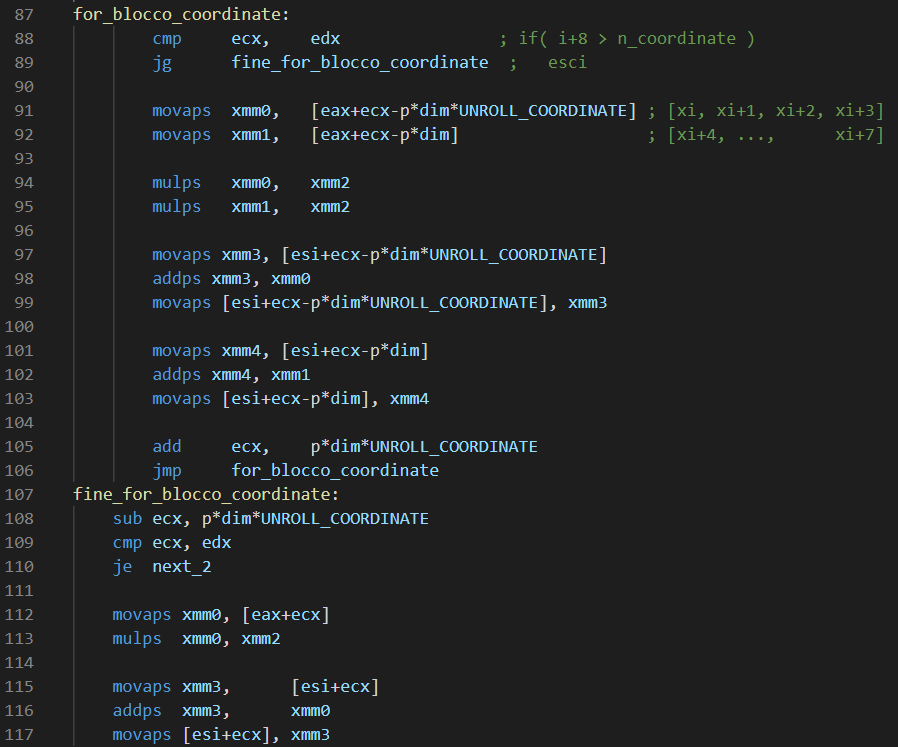
Nel caso in cui non applicassi l'unroll, non riuscirei a parallelizzare poiché mentre eseguo il quadrato sull'i-esimo elemento sto verificando che l'indice successivo sia ≤ della lunghezza del vettore. Tale condizione è precalcolata tramite il circuito di predizione, che è di per sé un'operazione onerosa, nel caso dell'unroll per ogni esecuzione onerosa del branch prediction si eseguono comunque quattro quadrati.

Immagine che contiene testo

Descrizione generata automaticamente

Alla riga 62 l’operazione di xor di un registro su sé stesso rappresenta l’azzeramento dello stesso. In questo caso si azzera il registro per effettuare la somma totale dei pesi, che viene calcolata mentre si itera sui quattro pesci.

Nella riga 82 come suggerisce il commento stesso si aggiorna la somma parziale dei pesi. Nelle due righe successive l’obbiettivo è quello di andare a selezione il primo dei quattro elementi contenuti in xmm5 e andarlo a replicare sul registro xmm2. Questa operazione è necessaria perché il blocco d’istruzioni che stiamo analizzando è ripetuto quattro volte. L’obbiettivo, infatti, di leggere contemporaneamente cinque elementi dal vettore dei pesi dei pesci è quello di ridurre gli accessi in memoria. Di seguito sono riportati dei disegni che chiariscono le righe di codice 83 e 84.

Nella riga 86 si aggiorna ecx che sarà utilizzato per il confronto che troviamo a riga 88, questo serve per valutare che il numero di coordinate ancora da analizzare sia effettivamente multiplo di 8, e quindi si possa proseguire con questo codice.

Nelle righe 91 e 92 due si leggono dalla memoria otto coordinate del pesce i-esimo, mentre nelle righe 94 e 95 si moltiplicano per il peso del pesce. Nei due blocchi di istruzione successivi che vanno da riga 97 a riga 103 si aggiorna il numeratore andando a sommare i valori appena calcolati.

Immagine che contiene testo

Descrizione generata automaticamente

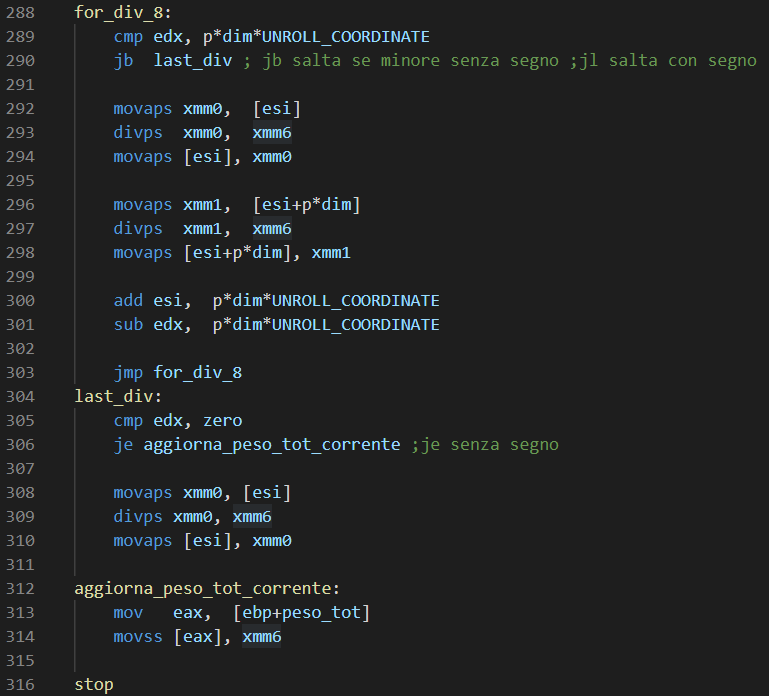
Nella riga 120, come suggerisce il commento stesso, si passa all’analisi del secondo pesce e, di conseguenza, come si può vedere nelle due righe successive si utilizza il secondo peso presente in xmm5 e lo si replica su xmm2.

Come anticipato, ora inizia una ripetizione del codice precedente sul secondo pesce del blocco da quattro, che è viene analizzato, questa operazione non è altro che l’unroll sui pesci.

Immagine che contiene testo

Descrizione generata automaticamenteCome conseguenza di questa operazione di unroll di quattro pesci per volta, resta da gestire il caso in cui i pesci non siano multipli di quattro e quindi i rimanenti sono analizzati singolarmente. Una volta completata questa serie di operazioni troveremo il numeratore memorizzato nella locazione di memoria del baricentro e ci si può calcolare la somma dei pesi.

Con questa coppia di operazioni calcoliamo il peso totale che si troverà replicato quattro volte nel registro. A questo punto rimane solamente l’operazione di calcolo del baricentro con la divisione tra numeratore e denominatore.



In questa ultima parte di codice si effettua l’operazione sopra annunciata, infatti, come per le parti precedenti prima si ha l’operazione sulle coordinate a gruppi di otto elementi per ogni salto condizionato e poi le rimanenti non multiple di 8. Un altro passaggio importante è il salvataggio in memoria del peso totale che servirà per operazioni successive e potrà essere utilizzato senza necessità di ricalcolarlo.

## Calcola valore F

Immagine che contiene testo

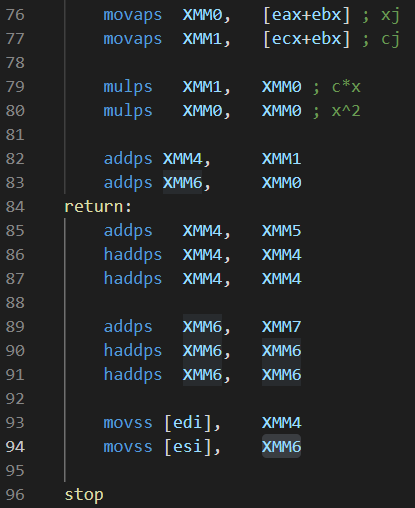
Descrizione generata automaticamenteLa funzione che calcola il valore di F viene mostrata nella sua versione “omp” così da dare un’idea dell’approccio utilizzato in questo caso. La funzione come suggerisce il nome stesso “calcola\_val\_f” calcola il valore della funzione F in un punto, in particolare viene calcolato il valore della funzione per ogni pesce (che rappresenta un punto). In questo caso però ad essere tradotto in assembly non è l’intera operazione, ma solamente il calcolo de “x2” e “c\*x”, il resto dell’operazione è effettuata in C utilizzando questi dati.

Questo estratto di codice in C fa notare appunto quanto sopra descritto, con la chiamata alla funzione esterna in assembly (*calcola\_val\_f\_asm\_omp*) vengono calcolati i valori *x\_2* (l’operazione di elevazione al quadrato di x) e i valori *c\_x* (il risultato del calcolo di c per x). Il valore della funzione calcolato nel punto identificato da ogni singolo pesce viene salvato in un vettore (*f\_cur*). Inoltre, in questo codice è inserita la direttiva (omp) che permette di parallelizzare (utilizzando le diverse CPU) le operazioni all’interno del ciclo for.

|  |  |
| --- | --- |
| **eax** | Indirizzo della matrice X |
| **ebx** | Contatore delle coordinate |
| **ecx** | Matrice dei coefficienti |
| **edx** | Numero coordinate totali per dimensione delle coordinate |
| **esi** | Indirizzo di x2 |
| **edi** | Indirizzo di c\*x |

Immagine che contiene testo

Descrizione generata automaticamente

I registri xmm4, xmm5, xmm6 e xmm7 sono azzerati per salvare progressivamente la somma delle operazioni. Nella porzione di codice sopra riportata si vanno a prelevare i dati necessari per effettuare i calcoli su otto coordinate per ciclo. Le ottimizzazioni effettuate in questa porzione di codice sono le stesse descritte in precedenza. Le operazioni sono semplici moltiplicazioni e addizioni. Una volta finito il ciclo, qual ora le coordinate non siano in modulo otto, è pervista una parte finale che gestisce le rimanenti, sicuramente in modulo quattro per il padding.

Al termine di queste operazioni, i risultati parziali sono sommati fra di loro, andando a utilizzare le stesse operazioni descritte per la funzione precedente, poi sono salvati in memoria e il controllo può tornare al programma in C.

## Movimento individuale

La funzione movimento individuale è una funzione su cui si è discusso in precedenza in relazione alle ottimizzazioni effettuate nel codice C. L’obbiettivo di questa funzione è quello di effettuare uno spostamento casuale ad ogni pesce, secondo la seguente equazione:

Come si può notare questa funzione fa uso dei numeri casuali e quindi si vedrà la gestione di questi ultimi nel calcolo di y. La funzione non è interamente tradotta in Assembly, ma solo alcune sue parti lo sono. In particolare, si prende in analisi la sua versione senza la direttiva openMP (la discussione fatta in precedenza circa la versione parallelizzata con openMP ha la medesima valenza anche in questo caso).

Immagine che contiene testo

Descrizione generata automaticamente

Le funzioni introdotte in Assembly sono due “*calcola\_y\_asm*” e “*calcola\_f\_y\_asm*”, la restante parte del codice sfrutta i dati prodotti da queste due funzioni per calcolare altri parametri utili in seguito, come il valore della funzione nei punti individuati dalla matrice y e il delta F, per valutare gli spostamenti da mantenere e quelli da ignorare.

### Calcola y

La funzione “*calcola\_y\_asm*” si occupa di calcolare lo spostamento di ogni singolo pesce e in contemporanea calcola il valore del delta x.

Queste due operazioni sono eseguite contemporaneamente in quanto prevedono l’utilizzo degli stessi dati per essere calcolate.

|  |  |
| --- | --- |
| **eax** | Indirizzo della matrice X |
| **ebx** | Indirizzo della matrice y |
| **ecx** | Coordinata j-esima (contatore coordinate) |
| **edx** | Numero coordinate totali per dimensione delle coordinate |
| **esi** | Indirizzo vettore numeri casuali |
| **edi** | Pesce i-esimo (contatore pesce) |

Immagine che contiene testo

Descrizione generata automaticamenteSi utilizza anche il registro xmm7 per salvare il parametro stepind che, come vediamo, è un fattore moltiplicativo nella formula dello spostamento. Una volta letto dalla memoria il valore è replicato sul registro. Nel registro xmm6 viene caricato un vettore di “2” che servirà come fattore moltiplicativo dei numeri random che sono compresi tra 0 e 1, mentre noi abbiamo bisogno di numeri casuali tra -1 e 1. Per ottenere questi numeri moltiplichiamo per il registro xmm6 (2.0) e sottraiamo il registro xmm5 in cui è caricato il valore “1”. Con queste operazioni otteniamo numeri casuali nell’intervallo d’interesse.

Immagine che contiene testo

Descrizione generata automaticamenteLe ottimizzazioni effettuate in questo codice assembly sono le stesse discusse in precedenza, ovvero analizzare quattro coordinate per volta ed effettuare un salto condizionato sul *for* dei pesci ogni quattro pesci analizzati.

In questo blocco di codice si va a calcolare lo spostamento di otto coordinate la volta e si salvano questi dati nel vettore y. È presente anche un blocco di codice per la gestione delle coordinate rimanenti che non sono multiple di otto. Questo codice è ripetuto per altri tre pesci, per un totale di quattro pesci per ciclo.

Nella parte seguente del codice a partire da riga 98 viene gestita (come in altre occasioni) l’eventualità in cui il numero di coordinate non sia divisibile per otto, e quindi, abbiamo un ciclo che effettua le stesse operazioni, ma su una singola coordinata per volta.

### Calcola f\_y

Questa funzione assembly è molto simile alla funzione vista in precedenza che calcolava il valore della funzione di *f*, infatti, il principio di base è il medesimo, ovvero, calcolare i due dati principali per poter valutare la funzione in un punto, ma in aggiunta questa funzione calcola il delta x.

Ricordiamo quindi che la funzione assembly non calcola direttamente il valore di *f* ma i valori “*y\*c*”e “*y2*”.

|  |  |
| --- | --- |
| **eax** | Indirizzo della matrice X |
| **ebx** | Indirizzo della matrice Y |
| **ecx** | Numero coordinate totali per dimensione delle coordinate |
| **edx** | Numero pesci |
| **esi** | Indirizzo vettore costanti della funzione |
| **edi** | Indirizzo del vettore dei delta x |

Le ottimizzazioni effettuate su questa funzione sono le medesime descritte per le funzioni precedenti, in particolare, presenta le medesime operazioni della funzione calcola\_val\_f.

Immagine che contiene testo

Descrizione generata automaticamente

Il cambiamento rispetto alla funzione precedente sta in queste poche righe di codice che hanno il compito di calcolare la differenza tra la coordinata y e la x così da ottenere il deltaX.

## Altre funzioni assembly

Oltre alle funzioni sopra descritte le altre funzioni rimanenti realizzate in assembly sono:

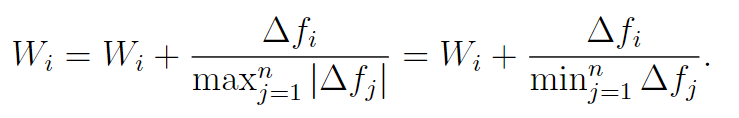
* Immagine che contiene testo, antenna

  Descrizione generata automaticamentecalcola\_I\_asm, è la funzione che si occupa di calcolare “*I*” che rappresenta lo spostamento nel movimento istintivo. La funzione è identica al baricentro, ma opera su dati differenti.
* Immagine che contiene orologio

  Descrizione generata automaticamentemov\_istintivo, completa il movimento istintivo andando a realizzare l’effettivo spostamento delle coordinate
* mov\_volitivo, funzione che sposta il braco verso il suo baricentro o lontano da esso in funzione all’aumento o diminuzione del peso totale del braco.

Immagine che contiene testo

Descrizione generata automaticamenteIn particolare, si avvicina al baricentro se il peso totale del branco è aumentato, altrimenti si allontana.

* alimenta\_asm, l’operazione di alimentazione va a modificare il peso del pesce in funzione del miglioramento che questo ha ottenuto.

Le funzioni sopra citate non vantano un’analisi di dettaglio come le precedenti, per tanti si è ritenuto superfluo descrivere nuovamente la stessa logica di ottimizzazione già discussa con le prime funzioni assembly.

# x86-64+AVX