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

**Gruppo 14 : Arcuri Francesco, Mangione Fabrizio, Morrone Stefano, Perna Stefano (PM)**

**Sommario**

[**Sommario** 1](#_Toc93784573)

[**Premessa** 1](#_Toc93784574)

[**Codice in C** 1](#_Toc93784575)

[**Ottimizzazioni C** 3](#_Toc93784576)

[**Assembly** 4](#_Toc93784577)

[**Ottimizzazioni** 4](#_Toc93784578)

[**x86-32+SSE** 5](#_Toc93784579)

[**Baricentro** 8](#_Toc93784580)

[**Calcola valore F** 15](#_Toc93784581)

[Utilizzo della libreria *OpenMP* 15](#_Toc93784582)

[**Movimento individuale** 17](#_Toc93784583)

[**Calcola y** 18](#_Toc93784584)

[**Calcola f\_y** 21](#_Toc93784585)

[**Altre funzioni assembly** 21](#_Toc93784586)

[**x86-64+AVX** 22](#_Toc93784587)

[1. Dimensione operandi 22](#_Toc93784588)

[2. Registri utilizzati 23](#_Toc93784589)

[3. Operazioni Particolari 23](#_Toc93784590)

[**Risultati e considerazioni finali** 28](#_Toc93784591)

# **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”, commentando la fase di sviluppo del codice.

Si assume 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 (row-order o column-order), ovvero come destrutturare la logica bidimensionale di una matrice per rappresentarla direttamente su un unico vettore. Si è deciso di rappresentare la matrice in forma row-order. Tale scelta è stata influenzata dalle particolari operazioni che hanno caratterizzato il progetto, nello specifico queste operano, ove possibile in parallelo, su sequenze di pesci e delle relative coordinate.

Immagine che contiene testo

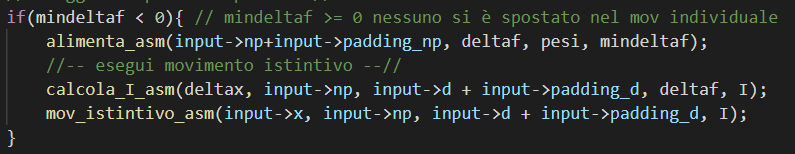
Descrizione generata automaticamenteL’implementazione dell’algoritmo ad alto livello è stata effettuata mediante l'impiego del linguaggio di programmazione C. Il contenuto di tale implementazione è stato caricato sul file "fss32c.c", presente all'interno della directory di progetto. Per ridurre la complessità la funzione “void fss(params\* input){…}”. implementa la logica generale. Essa prevede la chiamata a funzioni specifiche, le cui intestazioni sono esplicative del contributo apportato alla realizzazione del progetto in linguaggio di alto livello.

Immagine che contiene testo

Descrizione generata automaticamenteOltre alle variabili in input sono state definite altre variabili per memorizzare dati globali e risultati parziali. Di seguito sono riportate le variabili aggiunte:

Di seguito è mostrato il "cuore" dell'algoritmo. Tale frazione di codice è invocata tante volte quante il numero di iterazioni indicato nell'input. Nelle varie iterazioni si ripetono i diversi passi dell’algoritmo, come viene mostrato nel codice C seguente.

La divisione in funzioni ha permesso sia di ridurre la complessità al fine di rendere più leggibile l’algoritmo, che 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 operazioni aritmetiche dell’algoritmo per realizzare le funzioni in C. Una volta completata la fase di implementazione delle varie funzioni, si è resa necessaria una fase di debug al fine di scovare e risolvere gli errori emersi dalla esecuzione del suddetto codice. Tra gli errori identificati, quello più significativo faceva riferimento all'eccessivo utilizzo dei numeri random. La difficoltà di tale problema è stata la sua identificazione piuttosto che la 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 causa una riduzione dei tempi di esecuzione.

## **Ottimizzazioni C**

Immagine che contiene testo

Descrizione generata automaticamenteGià 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).Un 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.

Le altre funzioni non sono state caratterizzate da ottimizzazioni degne di nota, ad esclusione di quelle impiegate per limitare gli accessi in memoria e quelle relative alla riduzione delle catene dei 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.

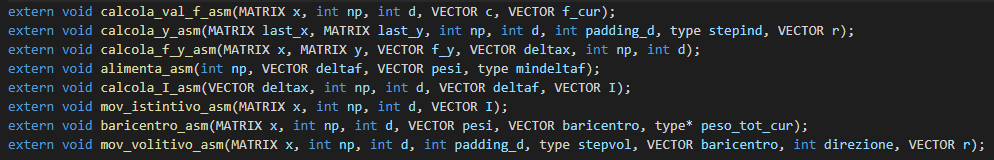
Nel caso di studio non si è resa necessaria un’ottimizzazione esplicita relativa alla memoria cache, poiché i calcoli effettuati sono intrinsecamente ottimizzati per questo particolare tipo di memoria. Le matrici sono memorizzate come array unidimensionali, i cui elementi sono memorizzati in ordine sequenziale 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, 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 poter lavorare 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.

Immagine che contiene testo

Descrizione generata automaticamenteLe funzioni rimanenti (che precedentemente non sono state citate) sono state riprodotte con una versione identica, ma con standard di nomenclatura differenti al fine di trovare coerenza con la procedura compilativa del docente.

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, ciò è garantito dalla neutralità degli zeri.

In considerazione del fatto che un generico elemento di tipo float occupa 4 celle di memoria, ciascuna da 8 bit per un totale di 32 bit, che le coordinate sono multiple di quattro, come detto precedentemente, è possibile applicare operazioni packed allineate in quanto gli indirizzi di memoria di tutte le variabili saranno sempre multipli di 16 (allineamento 16). Si noti che tali operazioni sono più veloci delle omologhe non allineate.

Il medesimo ragionamento è alla base dell’allineamento nella versione “x86-64+AVX”, ma a differenza di quando detto precedentemente verrà impiegato un allineamento a 32 poiché si lavorerà con elementi di tipo double ognuno dei quali ha peso 64 bit.

Si noti che alcune funzioni sono caratterizzate da due parametri, rispettivamente numero coordinate “d” e “padding\_d”, piuttosto che un unico parametro frutto della loro somma. Questa scelta è dipesa dalla impossibilità di applicare il padding ai numeri random a causa dell’eccessivo costo della “paddizzazione”.

Immagine che contiene tavolo

Descrizione generata automaticamenteImmagine che contiene tavolo

Descrizione generata automaticamente

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”.

Immagine che contiene testo

Descrizione generata automaticamente

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**

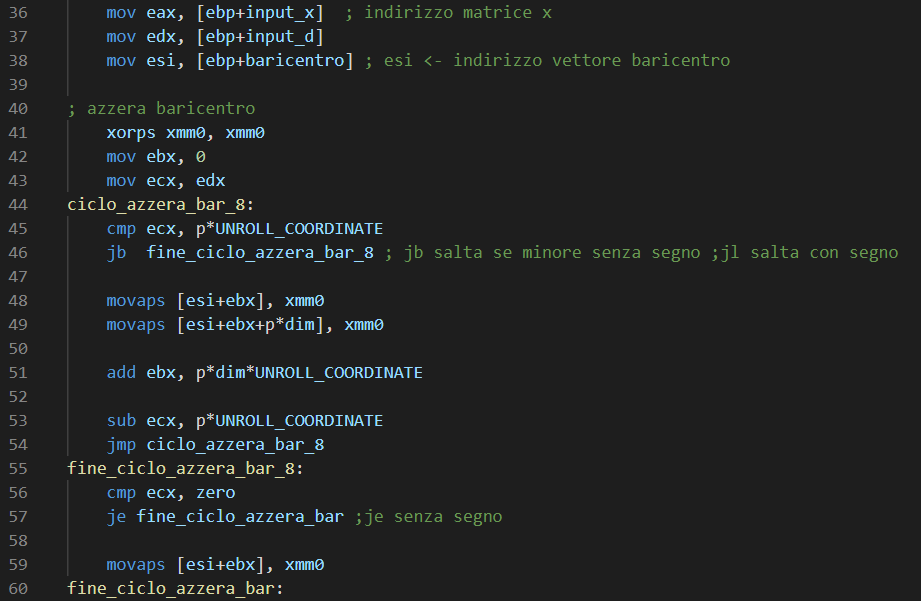
Durante la stesura del codice C si è riflettuto sull’eventuale divisione del calcolo del numeratore e del denominatore in due funzioni separate. La traduzione di tali operazione in Assembly ha mostrato che le due sub-funzioni avrebbero analizzato dati comuni annullando qualsiasi vantaggio di tipo prestazionale.

Immagine che contiene testo

Descrizione generata automaticamenteSono state definite delle costanti di offset nel “section .text”, 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 è caratterizzato da diversi commenti al fine di renderlo “user-friendly”, dunque di più facile comprensione. Inoltre, l’utilizzo di questi rende la ricerca e la correzione di eventuali errori la più semplice possibile.

Di seguito saranno riportate le osservazioni principali sul codice sovrastante (si seguirà lo stesso schema per le porzioni a seguire).

La riga 44 è caratterizzata da un ciclo, che esegue delle operazioni di lettura dei dati dalla memoria e di azzeramento. Tali operazioni sono obbligatorie, poiché il calcolo del numeratore della funzione baricentro necessita del supporto della memoria principale (RAM). Nello specifico, la locazione di memoria che viene utilizzata è quella destinata (e allocata) al baricentro. Tale locazione è utilizzata temporaneamente per salvare il peso totale (calcolato).

In considerazione di quanto detto precedentemente, nella fase finale dell’algoritmo basterà accedere in memoria e dividere il numeratore per il peso totale.

La riga 45 è caratterizzata dalla costante “p”, il cui significato è packed, ovvero indica quanti elementi per ciascuna iterazione vengono letti dalla memoria (in questo caso quattro float).

“UNROLL\_CORDINATE” indica il numero di 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.

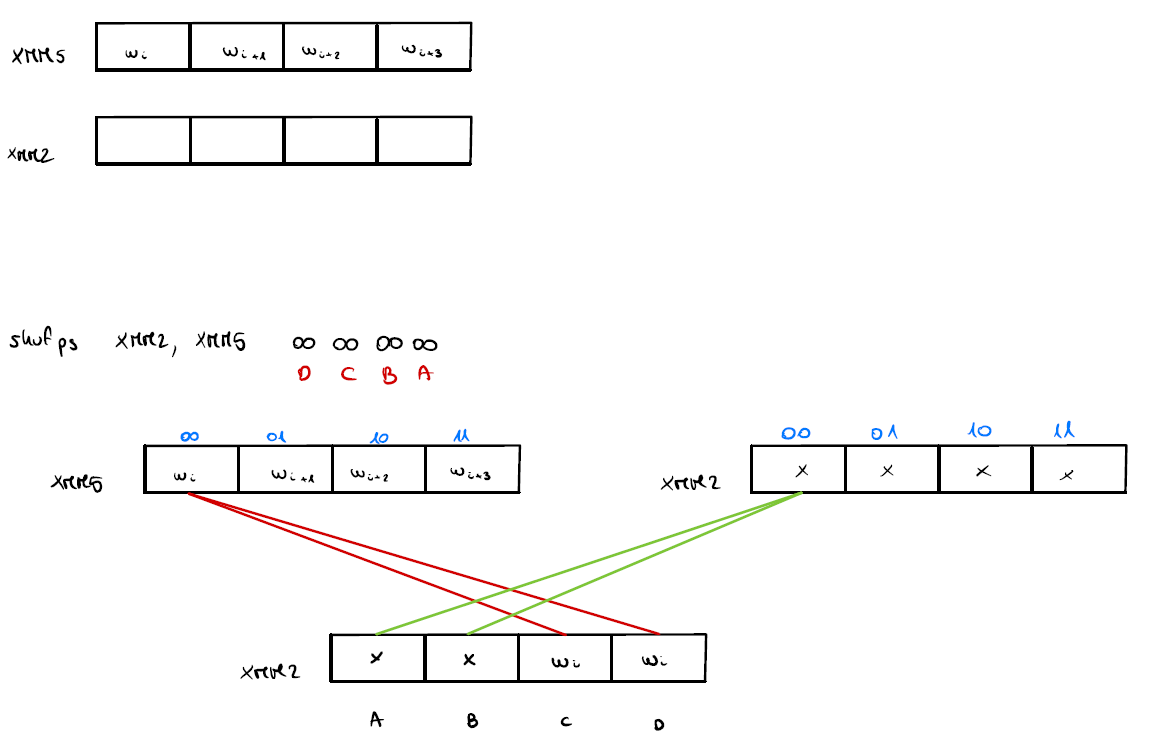
Immagine che contiene testo

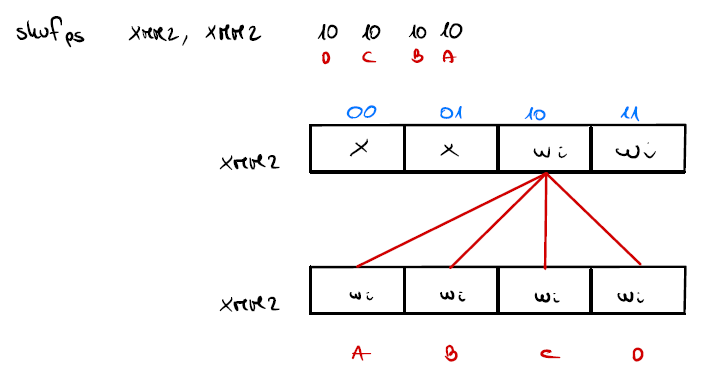
Descrizione generata automaticamente

Nel caso non venisse applicata la tecnica di unroll, non sarebbe possibile parallelizzare poiché mentre si calcola il quadrato sull'i-esimo elemento si verifica, in contemporanea, che l'indice successivo sia ≤ della lunghezza del vettore. Tale condizione è precalcolata per mezzo del circuito di predizione, che è di per sé un'operazione onerosa. Si noti che nel caso dell'unroll, per ogni esecuzione onerosa del branch prediction si eseguono comunque quattro quadrati. Ciò sta ad evidenziare un notevole beneficio prestazionale.

La riga 62 è caratterizzata dall’operazione di xor, necessaria per l’azzeramento del registro su cui è eseguita tale operazione. In tal caso l’azzeramento è impiegato al fine di effettuare la somma totale dei pesi, che viene calcolata mentre si itera sui “quattro pesci”.

La riga 82, come suggerisce il commento stesso, aggiorna la somma parziale dei pesi. Le due righe successive hanno l’obbiettivo di selezione il primo dei quattro elementi contenuti in xmm5 e replicarlo sul registro xmm2. Questa operazione è necessaria perché il blocco d’istruzioni preso in analisi, è ripetuto quattro volte. L’obbiettivo è dunque quello di leggere contemporaneamente quattro elementi dal vettore dei pesi dei pesci è quello di ridurre gli accessi in memoria.

Di seguito sono riportati delle figure che chiariscono le righe di codice 83 e 84.



La riga 86 aggiorna ecx che sarà utilizzato per il confronto presente 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.

Immagine che contiene testo

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

Immagine che contiene testo

Descrizione generata automaticamenteLa riga 120, come suggerisce il commento stesso, è caratterizzata dall’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.

Inizia una ripetizione del codice precedentemente esposto, sul secondo pesce del blocco da quattro. Questa operazione corrisponde al loop unrolling eseguito sui pesci.

Come conseguenza dell’unroll, resta da gestire il caso in cui i pesci non siano multipli di quattro, di conseguenza, i restanti saranno analizzati singolarmente. Segue un’immagine esplicativa:

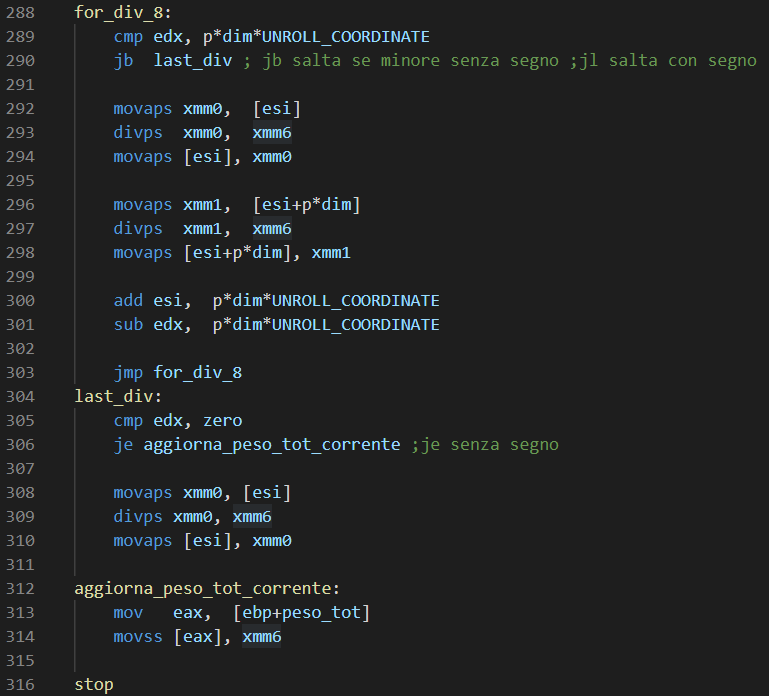
Immagine che contiene testo

Descrizione generata automaticamente

Immagine che contiene testo

Descrizione generata automaticamenteCompletata questa serie di operazioni, il numeratore sarà memorizzato nella locazione di memoria del baricentro. Al fine di calcolare il baricentro stesso, non ci resta che terminare il calcolo del denominatore.

Le operazioni sopra riportate calcolano il peso totale replicato quattro volte nel registro xmm6.

A questo punto resta da eseguire la divisione tra numeratore e denominatore, al fine di conseguire il calcolo del baricentro.

Quest’ultima parte di codice esegue l’operazione precedentemente introdotta. Si noti che l’operazione di padding effettuata sulle coordinate garantisce che queste siano sicuramente multiple di quattro. In considerazione di quanto detto è sempre possibile applicare operazioni packed al posto di iterare sulle singole coordinate rimanenti.

Un altro passaggio importante è il salvataggio in memoria del peso totale che servirà per operazioni successive e potrà essere utilizzato senza la necessità di ricalcolarlo.

## **Calcola valore F**

La 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 F in un punto, nello specifico, viene calcolato il valore della funzione per ogni pesce (che rappresenta un punto). Si noti che ad essere tradotto in linguaggio di basso livello non è l’intera operazione, ma solamente il calcolo de “x2” e “c\*x”. Il resto dell’operazione è effettuata in C utilizzando i seguenti dati:

Immagine che contiene testo

Descrizione generata automaticamente

Questo estratto di codice in C fa notare quanto detto precedentemente. 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 openMP che permette di parallelizzare, utilizzando i diversi core della CPU, le operazioni all’interno del ciclo for.

### Utilizzo della libreria *OpenMP*

La libreria *OpenMP* permette di generare codice, a tempo di compilazione, che può essere eseguito in parallelo da più thread in modo semplice e intuitivo grazie all’utilizzo di alcune direttive. In questo progetto il suo utilizzo si è limitato esclusivamente alla parallelizzazione di alcuni cicli for, ritenuti essere più dispendiosi in termini di complessità di calcolo rispetto ad altri. Per far ciò è stato introdotto, prima di ogni ciclo for interessato la seguente direttiva:

*#pragma omp parallel for.*

Quest’ultima indica al compilatore di parallelizzare il blocco contenuto all’interno del *for*. *OpenMP*, automaticamente avvierà tanti thread quanti sono i core disponibili sulla macchina. Inoltre, ogni variabile dichiarata all’interno del blocco di codice del *for* è ad uso esclusivo di ogni singolo thread, in modo che non ci sia *race condition*.

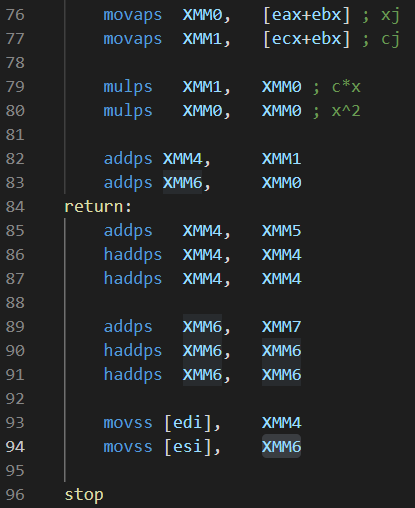
Inizialmente si era pensato di utilizzare il parametro *num\_threads(MAX\_NUM\_THREADS)* in modo da fissare un numero di thread da utilizzare nei blocchi di codice parallelizzati così da evitare rallentamenti nel caso in cui la macchina sia già occupata nell'eseguire altri processi.

|  |  |
| --- | --- |
| **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 prelevano 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. Una volta terminato il ciclo, qualora le coordinate non siano multiple di otto, è prevista una parte finale che gestisce le rimanenti, sicuramente un unico blocco di quattro (per effetto del padding), mediante operazioni packed allineate.



Al termine di queste operazioni, i risultati parziali sono sommati fra di loro, utilizzando le stesse operazioni descritte per la funzione precedente. Successivamente, tali dati, sono salvati in memoria e il controllo torna al programma in C.

## **Movimento individuale**

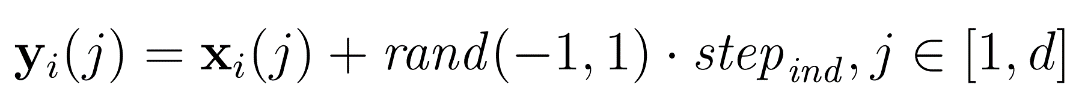
La funzione movimento individuale è una funzione discussa in precedenza, in relazione alle ottimizzazioni effettuate nel codice C. L’obbiettivo di questa funzione è quello di effettuare uno spostamento casuale per ogni pesce, secondo la seguente equazione:

Immagine che contiene testo

Descrizione generata automaticamenteCome si può notare questa funzione fa uso dei numeri casuali, la cui gestione sarà analizzata in “calcola\_y”. La funzione non è interamente tradotta in Assembly, ma solo alcune sue parti lo sono. In particolare, si prende in analisi la versione senza la direttiva openMP in quanto la discussione fatta in precedenza circa la versione parallelizzata con openMP ha la medesima valenza anche in questo caso, ad eccezione dell’utilizzo della direttiva *#pragma omp critical{…}* che racchiude il blocco di codice in riga 408 in modo che l’aggiornamento della variabile *mindeltaf* venga effettuato da un solo thread per volta e non ci siano problemi di inconsistenza.

La funzione mov\_individuale, del linguaggio C, richiama due funzioni Assembly, rispettivamente: “*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 per scopi successivi, come ad esempio: 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.

|  |  |
| --- | --- |
| **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) |

Si utilizza anche il registro xmm7 per salvare il parametro stepind che è un fattore moltiplicativo nella formula dello spostamento. Una volta letto dalla memoria, il valore è replicato sul registro.

Immagine che contiene testo

Descrizione generata automaticamentePer eseguire il calcolo delle y sono necessari numeri casuali compresi tra -1 e 1. A causa di ciò, è necessario moltiplicare, i numeri random prelevati, per due e successivamente sottrarre a questi il valore uno, affinché si possa ottenere il range da noi desiderato (-1,to 1). A livello operativo, questo si traduce nel caricare sul registro xmm6 un vettore costituito da elementi 2.0 ([2.0,2.0,2.0,2.0]) e nel registro xmm5 un vettore costituito da elementi 1.0 ([1.0,1.0,1.0,1.0]).

Le ottimizzazioni effettuate per questo particolare 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.

Immagine che contiene testo

Descrizione generata automaticamente

In questo blocco di codice si calcola lo spostamento di otto coordinate per volta (loop unrolling sulle coordinate) e si salvano tali dati nel vettore y.

Si evidenzia la necessità di inserire un blocco di codice che possa gestire le coordinate rimanenti, al più tre, nel caso in cui la dimensione di d non fosse divisibile per otto. Si noti che questa gestione è ripetuta per gli altri tre pesci (loop unrolling sui pesci).

Immagine che contiene testo

Descrizione generata automaticamente

### **Calcola f\_y**

Immagine che contiene testo

Descrizione generata automaticamenteTale funzione Assembly è molto simile alla precedente 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.

Si rammenta 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 |

Immagine che contiene testo

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

## **Altre funzioni assembly**

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

* Immagine che contiene testo, antenna

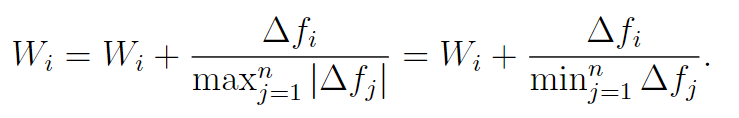
  Descrizione generata automaticamentecalcola\_I\_asm, è la funzione che si occupa di calcolare “*I*” che rappresenta lo spostamento nel movimento istintivo. La funzione è identica al baricentro, ma opera su dati differenti.
* Immagine che contiene testo

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

Immagine che contiene testo

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

Essendo che il movimento volitivo è caratterizzato dalla lettura dei numeri random, anche in tal caso saranno applicate le accortezze viste nella funzione “calcola\_y” per la lettura di tali valori.

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

Le funzioni sopra citate non richiedono un’analisi dettagliata come avvenuto per le precedenti. Pertanto, si è ritenuto superfluo descrivere nuovamente la stessa logica di ottimizzazione già discussa nelle prime funzioni assembly.

# **x86-64+AVX**

Per quanto concerne tale architettura, gli algoritmi utilizzati sono i medesimi adoperati in x86-32+SSE, ciò che cambia è la dimensione degli operandi e la traduzione delle istruzioni da SSE in AVX. Per tale motivo nel seguente paragrafo non verranno nuovamente spiegati i ragionamenti seguiti da ogni algoritmo, ma verranno semplicemente evidenziate le modifiche principali effettuate, al fine di tradurre le istruzioni del repertorio SSE nelle istruzioni del repertorio AVX.

## Dimensione operandi

La principale differenza rispetto all’architettura sviluppata precedentemente è la dimensione degli operandi, questi non saranno più dei float (dimensione 32 bit), ma double (dimensione 64 bit). Ciò comporta, che tutte le istruzioni Assembly saranno caratterizzate dalla lettera D al termine del loro nome (Ad esempio MOVAP**S** diventerà VMOVAP**D**). Nell’algoritmo principale C, che richiama tutte le funzioni Assembly, sarà sufficiente ridefinire la direttiva presente nella riga 50 del file fss64c.c.

|  |  |
| --- | --- |
|  |  |

A seguito di tale modifica tutti gli operandi saranno identificati come double.

Un ulteriore accortezza sarà quella di modificare i parametri utilizzati nella funzione per l’allocazione della memoria in quanto, come visto precedentemente, per il padding sulle coordinate sarà necessario allocare la memoria partendo da un indirizzo multiplo di 32.

|  |  |
| --- | --- |
| Immagine che contiene testo  Descrizione generata automaticamente | Immagine che contiene testo  Descrizione generata automaticamente |

## Registri utilizzati

Il repertorio AVX comprende 16 nuovi registri:

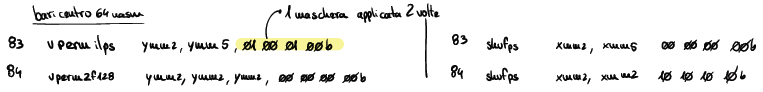
* Da YMM0 a YMM7, i quali non sono altro che una estensione dei corrispondenti registri XMM;
* Da YMM8 a YMM15 che sono registri a 64 bit (totale spazio singolo registro 256 bit).

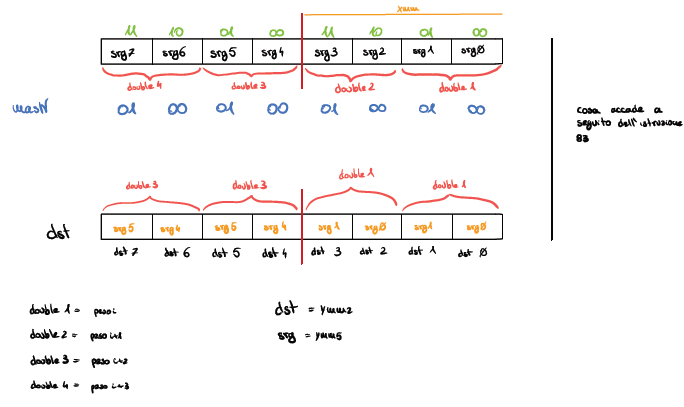
Nelle funzioni in Assembly del nuovo repertorio tutti i registri XMM sono stati trasformati negli omologhi YMM. Particolare attenzione è stata posta ai casi in cui i valori letti siano singoli double, tali casi hanno richiesto l’impiego dei registri XMM a causa della mancanza di operazioni “singole” sul repertorio AVX.

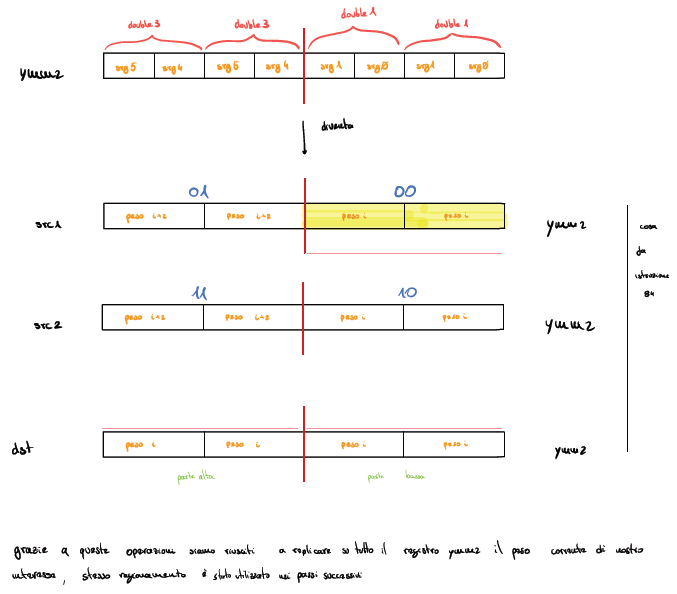
## Operazioni Particolari

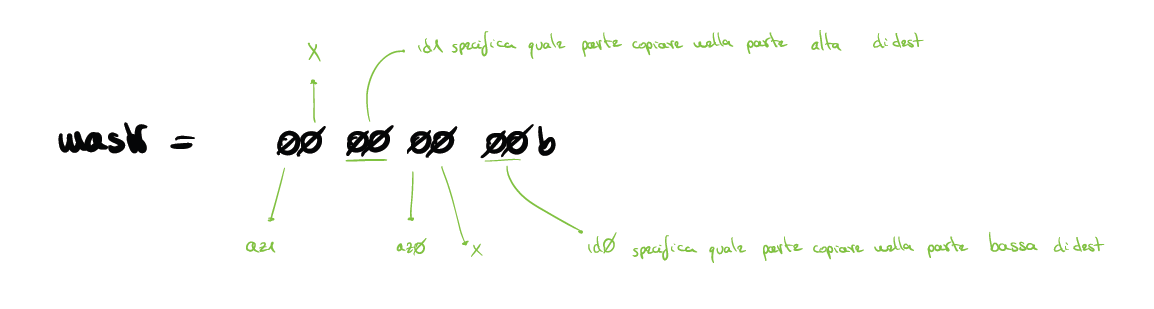
In questo sotto paragrafo verranno presentate le soluzioni progettuali adoperate al fine di tradurre il codice dell’architettura x86-32+SSE in x86-64+AVX.

Nel file baricentro64.nasm è stato necessario tradurre accuratamente l’operazione di shuff non essendo presente nel repertorio AVX, attraverso l’utilizzo delle operazioni VPERILPS e VPERM2F128. Si noti che tale ragionamento verrà applicato a tutte le funzioni Assembly, le quali utilizzano operazioni di shuff nel repertorio SSE.



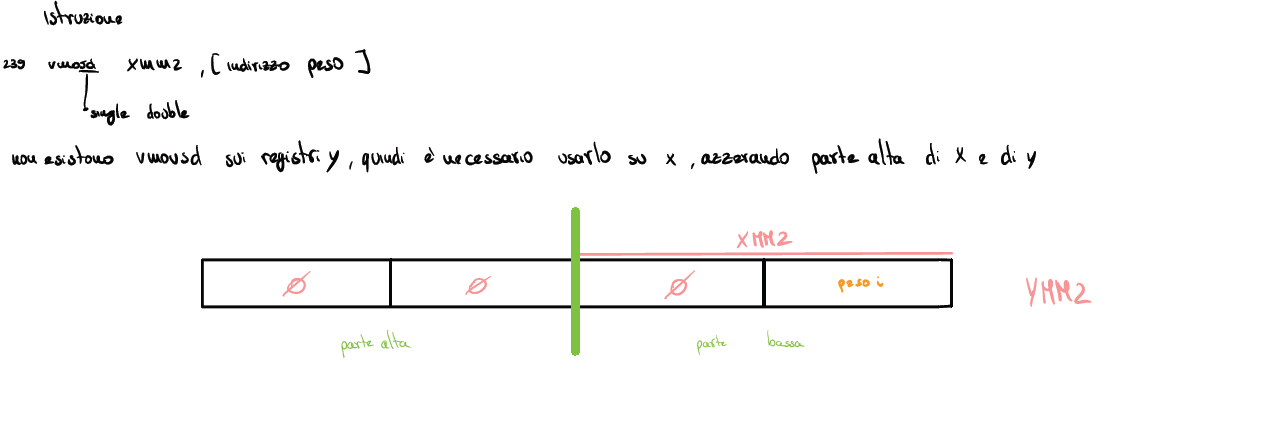




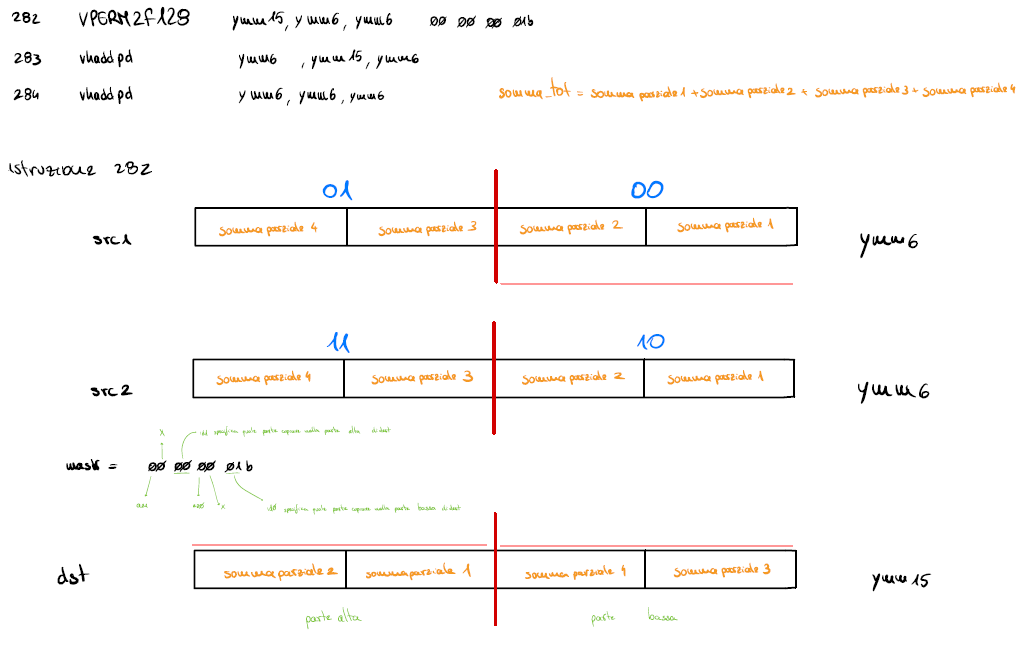


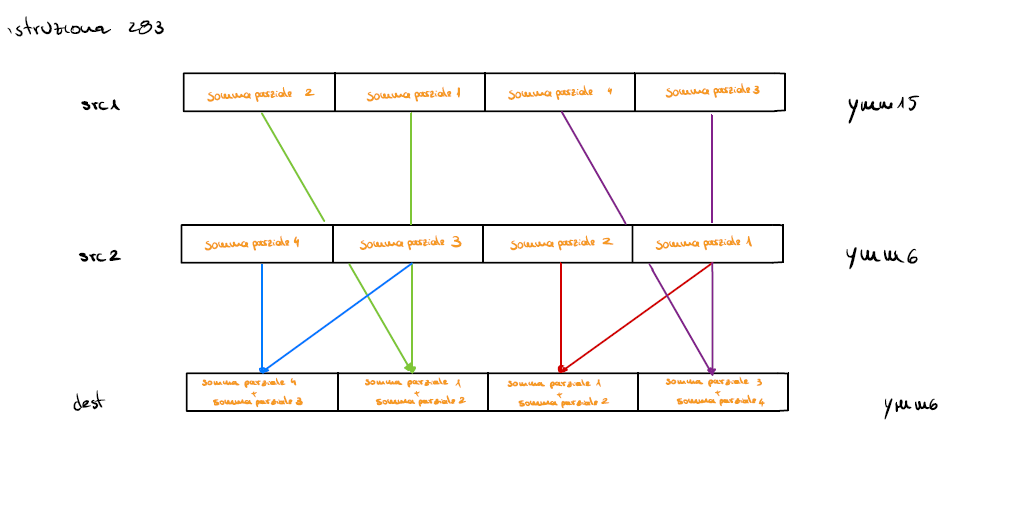
A causa delle problematiche relative all’impiego delle istruzioni vmovsd, il loro utilizzo è stato limitato solo ai casi strettamente necessari.

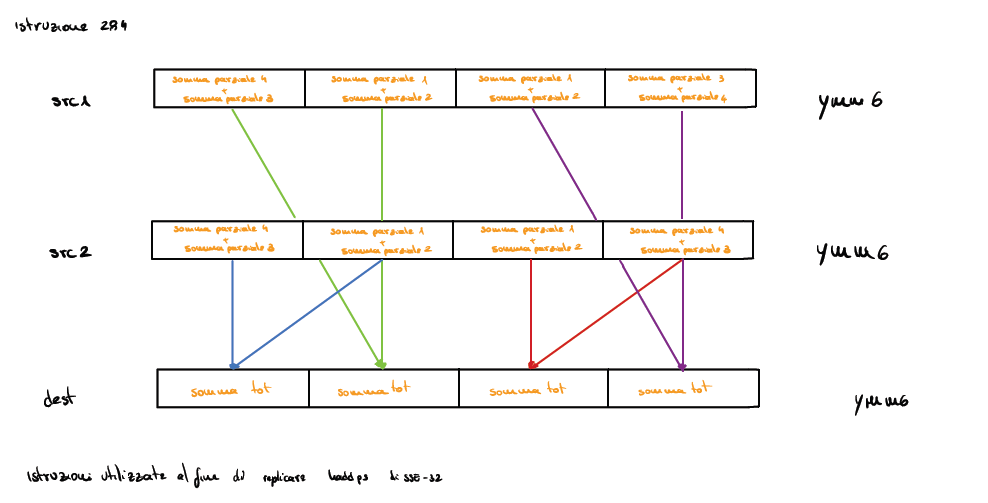
A titolo di esempio, si riporta la gestione dell’istruzione vmovsd presente nel file baricentro64.nasm:



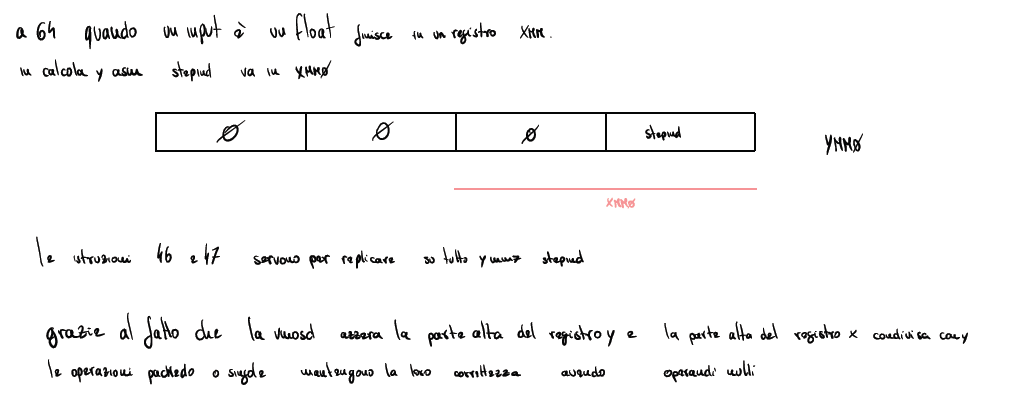
Non essendo presenti nel repertorio AVX le istruzioni di haddp, di seguito si presenta la traduzione di queste istruzioni, con riferimento alla funzione assembly calcola baricentro.







Si noti che il nuovo repertorio istruzioni tiene conto di nuove convenzioni per il passaggio dei parametri dal linguaggio di alto livello C ad Assembly, come riportato nel pdf “C\_calling\_conventions”. L’architettura AVX offre un vantaggio, in quanto non è necessario prelevare i parametri della funzione (dallo stack) per caricarli sui registri, ma sono automaticamente caricati su dei registri specifici nel passaggio dal linguaggio C ad assembly basato su architettura AVX. Proprio a seguito delle nuove convenzioni si evidenzia che gli argomenti di tipo floating-point vengono passati nei registri da XMM0 a XMM7, quindi nel caso della funzione calcola\_y\_64.nasm, sono state effettuate le seguenti operazioni sotto riportate.



# **Risultati e considerazioni finali**

Di seguito si riportano delle tabelle riassuntive che mostrano l’andamento delle prestazioni del programma al variare del:

* Numero di pesci
* Coordinate
* Iterazioni
* Dimensione operandi

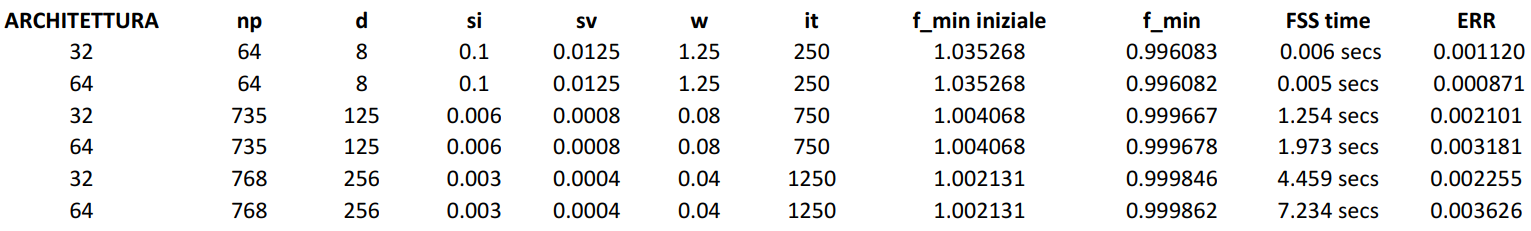
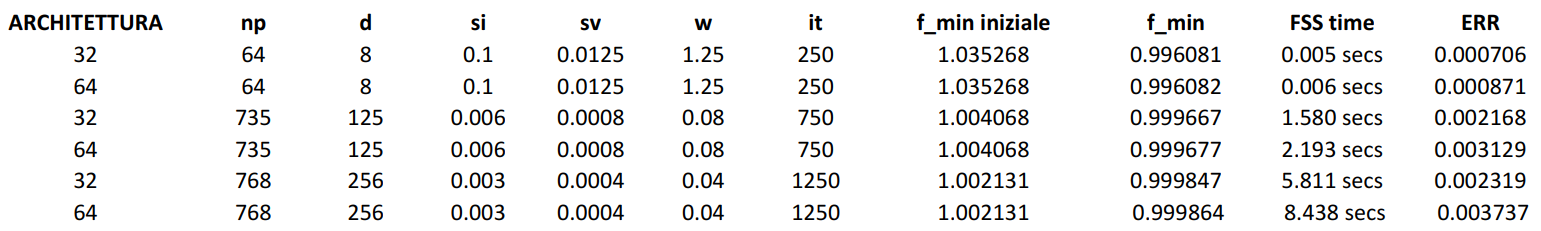
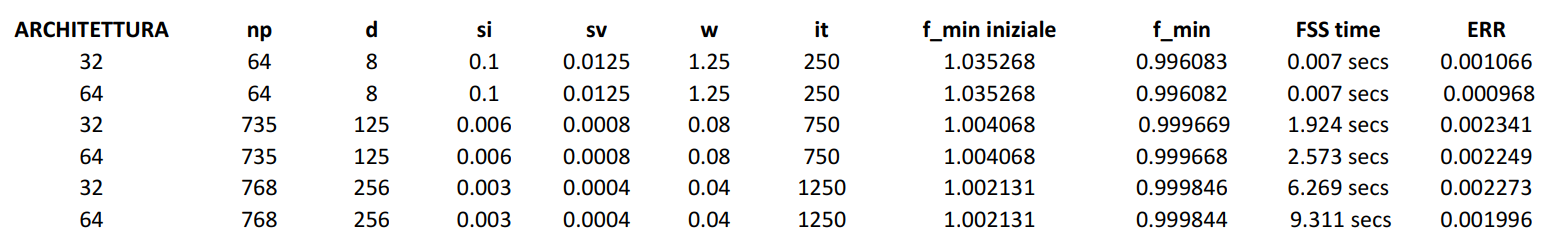
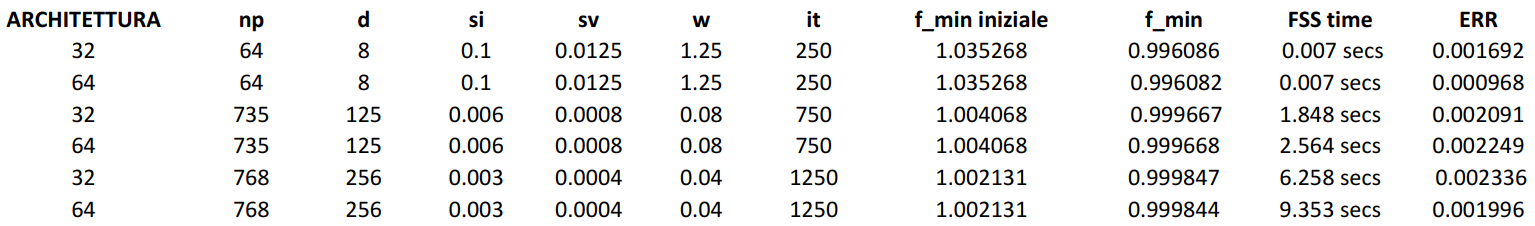
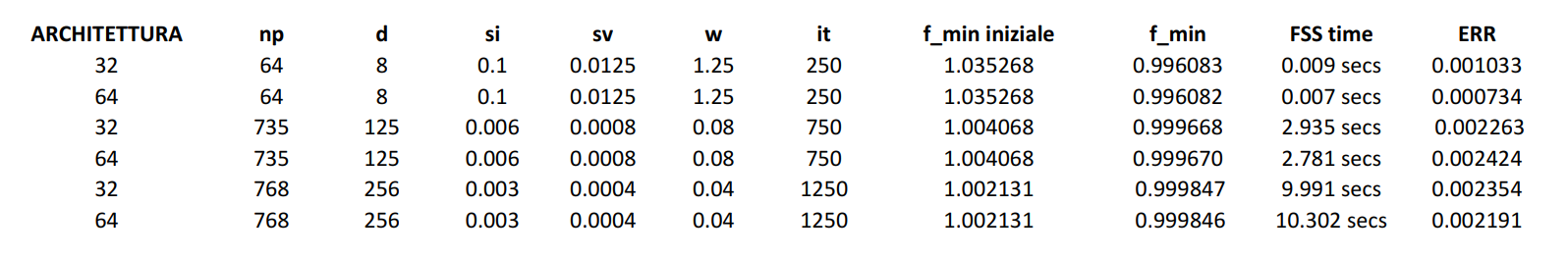
La prima e la seconda tabella è caratterizzata dal solo codice C privo di alcuna ottimizzazione legata al linguaggio di basso livello Assembly. L’ unica differenza è che nel secondo caso è stato inserito il padding in modo da predisporre il codice per l’introduzione delle procedure ASM.   
\*\*\* MANGIONE\*\* (Ogni tabella è stata ottenuta aggiungendo una procedura in ASM per volta rispetto al codice precedente).  
L’ ultima tabella evidenzia i miglioramenti ottenuti a seguito dell’applicazione delle procedure di ottimizzazione utilizzate e studiate nel corso di Architetture e Programmazione di Sistemi di Elaborazione a.a. 2021/2022.

Immagine che contiene tavolo

Descrizione generata automaticamente

**Risultati esecuzione di codice solo in C (senza padding)**

**Risultati esecuzione di codice solo in C (con padding)**



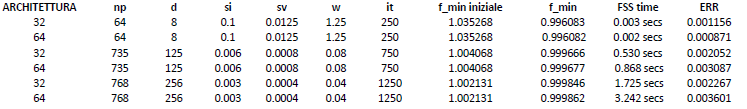
**Risultati a seguito dell’introduzione del movimento individuale in ASM**

**Risultati a seguito dell’introduzione di alimenta in ASM**

**Risultati a seguito dell’introduzione del movimento istintivo in ASM**

**Risultati a seguito dell’introduzione del baricentro in ASM**

**Risultati a seguito dell’introduzione del movimento volitivo in ASM**



**Risultati codice definitivo anche con OMP (introdotto per ultimo calcola\_val\_f in ASM)**

Immagine che contiene tavolo

Descrizione generata automaticamente