UNIVERSITÀ DELLA CALABRIA

DIPARTIMENTO DI INGEGNERIA INFORMATICA

Architetture avanzate degli elaboratori

Protein Simulated Annelling

Docenti:
Angiulli Fabrizio
Fassetti Fabio
Nisticò Simona

Candidati:
Ciccia Umberto
Domenico
Mat. 263844
Mirabelli Francesca
Mat. 264173

ANNO ACCADEMICO 2024/2025

Visita il progetto su GitHub

Indice

Introduzione

Il progetto si focalizza sul calcolo e sulla previsione degli angoli diedrici che definiscono la struttura terziaria di una proteina. La struttura terziaria di una proteina è cruciale per la sua funzione biologica, ed è spesso complessa da determinare sperimentalmente a causa della difficoltà nel misurare direttamente la configurazione spaziale degli atomi coinvolti. Una volta determinati gli angoli diedrici, è possibile ricostruire la configurazione spaziale complessiva della proteina e calcolare l'energia associata a tale conformazione. Attraverso questo processo, si può identificare la struttura più stabile, che corrisponde alla conformazione a energia minima.

Il principale obiettivo del progetto è affrontare la predizione del ripiegamento proteico. Questo viene realizzato attraverso l'uso di algoritmi avanzati che simulano il comportamento molecolare della proteina e la guidano verso una configurazione stabile. Per raggiungere questi obiettivi, il progetto è stato implementato in diverse versioni software, ottimizzando e adattando il codice per sfruttare al meglio diverse architetture hardware. Le versioni implementate sono le seguenti:

- **Architettura x86-32 con supporto SSE**: Questa versione è stata progettata per architetture a 32 bit, con l'introduzione di set di istruzioni SIMD (Single Instruction, Multiple Data) come SSE, che consentono di elaborare più dati simultaneamente, migliorando così le prestazioni rispetto alla versione sequenziale.

- **Architettura x86-64 con supporto AVX**: In questa versione, il codice è stato adattato per sfruttare l'architettura a 64 bit, con il sup-

porto per le istruzioni AVX (Advanced Vector Extensions), che offrono un'elaborazione più rapida e parallela per applicazioni scientifiche complesse.

- **Architettura x86-64 con supporto AVX e OpenMP**: Questa versione si concentra sull'ulteriore ottimizzazione delle prestazioni mediante l'utilizzo di OpenMP, una libreria che consente la parallelizzazione delle operazioni a livello di codice.

La fase iniziale del progetto ha visto la scrittura del codice in linguaggio C, per garantire un funzionamento corretto e facilmente verificabile. In questa fase, sono stati condotti test per verificare la correttezza dell'algoritmo implementato, confrontando i risultati ottenuti con valori teorici e dati sperimentali, quando disponibili.

Successivamente, il progetto si è concentrato sulla versione a 32 bit, in cui sono state effettuate analisi approfondite riguardanti le prestazioni. In particolare, si è studiato l'impatto delle ottimizzazioni effettuate mediante l'inserimento di codice assembly per accelerare operazioni critiche. Oltre a queste ottimizzazioni in assembly, sono stati apportati dei miglioramenti delle prestazioni modificando direttamente il codice in C, per evitare chiamate a funzioni non necessarie e ridurre l'overhead computazionale.

Per la versione a 64 bit, sono stati esaminati i benefici derivanti dall'integrazione delle procedu in assembly già presenti nella versione a 32 bit. Queste implementazioni hanno portato dei notevoli miglioramenti al tempo di esecuzione.

Infine, l'ultima fase del progetto si è concentrata sull'ottimizzazione del codice utilizzando OpenMP, una libreria che permette di eseguire operazioni in parallelo su più core di un processore. Inizialmente, abbiamo modificato il sistema di calcolo del tempo di esecuzione per includere i comandi relativi a OpenMP, in modo da misurare con maggiore precisione i miglioramenti delle prestazioni. Successivamente, ci siamo focalizzati sull'utilizzo del costrutto `parallel for`, che permette di parallelizzare facilmente i cicli di iterazione nel

codice, un aspetto fondamentale per accelerare il calcolo di grandi strutture proteiche.

L'approccio di parallelizzazione ha portato a miglioramenti delle prestazioni, consentendo al programma di gestire simulazioni più complesse e di operare su dataset di dimensioni maggiori.

In conclusione, questo progetto ha contribuito a sviluppare un sistema di predizione del ripiegamento proteico altamente ottimizzato, capace di sfruttare al meglio le risorse hardware disponibili, riducendo i tempi di calcolo e migliorando la precisione nella previsione della struttura terziaria delle proteine. Tali sviluppi potrebbero avere applicazioni significative in bioinformatica, farmacologia e medicina, dove la conoscenza della struttura delle proteine è essenziale per la progettazione di nuove terapie.

Metodologia di sviluppo

Per lo sviluppo del progetto, abbiamo utilizzato Git come strumento di controllo versione, sfruttando al massimo le sue funzionalità di branching e merging per facilitare la collaborazione tra i membri del team. Il codice è stato sviluppato passo dopo passo, con ogni partecipante che ha contribuito in modo sinergico e parallelo, garantendo la coerenza e l'integrazione fluida delle varie componenti del sistema. Ogni fase dello sviluppo è stata documentata in modo preciso, creando commit dettagliati che descrivono le modifiche apportate. Questo approccio ha permesso di mantenere una cronologia chiara e facilmente tracciabile, facilitando la gestione dei conflitti e ottimizzando il processo di testing e revisione del codice.

Architettura x86-32 con supporto SSE

Il progetto ha subito diverse modifiche e ottimizzazioni che hanno portato a un miglioramento significativo dei tempi di esecuzione. Questa sezione analizza ogni implementazione effettuata, evidenziando i vantaggi ottenuti in termini di prestazioni per l'architettura a 32 bit. Le implementazioni verrano descritte nell'ordine con la quale sono state effettuate.

Versione in C pura

La prima parte del progetto si è focalizzata sull'implementazione dell'algoritmo di predizione in C.

- Tempo di Esecuzione: **0,650 secondi.**

La prima versione esclusivamente in C, senza l'uso di assembly, ha evidenziato limiti in termini di ottimizzazione dei calcoli a basso livello, dimostrando la necessità di interventi in assembly per migliorare ulteriormente. Ovviamente questa prima versione ottenuta è stata sviluppata avendo come obiettivo l'ottenimento dei risultati attesi.

Versione C con implementazione in assembly della distanza Euclidea

In questa versione è stata sviluppata la distanza euclidea in assembly

Listing 1: euclidean_dist_sse

```
1  euclidean_dist_sse:
2          push            ebp
3          mov             ebp, esp
4          push            ebx
5          push            esi
6          push            edi
7
8          MOV EAX, [EBP+A]
9          MOV EBX, [EBP+B]
10         MOV ECX, [EBP+C]
11
12         MOVAPS   XMM0, [EAX]
13         MOVAPS   XMM1, [EBX]
14
15         SUBPS XMM1, XMM0
16
17         MULPS XMM1, XMM1
18         MOVSS [e], XMM1
19
20         HADDPS XMM1, XMM1
21         HADDPS XMM1, XMM1
22
23         MOVSS [e], XMM1
24
25         SQRTPS XMM1,XMM1
26         MOVSS [e], XMM1
27
28         MOVSS [ECX], XMM1
29         MOVSS [e], XMM1
30
31         pop     edi
32         pop     esi
33         pop     ebx
34         mov     esp, ebp
35         pop     ebp
36         ret
```

L'implementazione non ha dato miglioramenti in termini di tempo a causa dell'overhead dato dalla continua chiamata alla funzione.

- Tempo di Esecuzione: **0,750 secondi.**

In questa versione iniziale, le componenti critiche come il calcolo della distanza euclidea sono state implementate parzialmente in assembly, non ottenendo un primo miglioramento rispetto alla versione solo in C, causa overhead delle chiamate.

Versione C con implementazione in assembly di rama_energy

In questa versione è stata implementato il calcolo di rama_energy in assembly e ciò ha portato ad un miglioramento delle prestazioni.

- Tempo di Esecuzione: **0,410 secondi.**

Concentrandosi esclusivamente sull'ottimizzazione della rama_energy in assembly, si è ridotto quasi della metà il tempo rispetto alla prima versione, grazie a un focus più mirato sulle operazioni intensive. Il codice della procedura è il seguente:

Listing 2: rama_energy_assembly

```
1  rama_energy_assembly:
2      push ebp
3      mov ebp, esp
4      push ebx
5      push esi
6      push edi
7
8      mov eax, [ebp + 8]
9      mov ebx, [ebp + 12]
10     mov ecx, [ebp + 16]
11
12     xorps xmm0, xmm0
```

```
13
14      movss xmm1, [alpha_phi]
15      movss xmm2, [alpha_psi]
16      movss xmm3, [beta_phi]
17      movss xmm4, [beta_psi]
18      movss xmm5, [half]
19
20      mov esi, 0
21      .loop:
22
23          movss xmm6, [eax + esi * 4]
24          movss xmm7, [ebx + esi * 4]
25
26          movaps xmm0, xmm6
27          subss xmm0, xmm1
28          mulss xmm0, xmm0
29          movaps xmm2, xmm7
30          subss xmm2, xmm2
31          mulss xmm2, xmm2
32          addss xmm0, xmm2
33          sqrtss xmm0, xmm0
34
35          movaps xmm2, xmm6
36          subss xmm2, xmm3
37          mulss xmm2, xmm2
38          movaps xmm3, xmm7
39          subss xmm3, xmm4
40          mulss xmm3, xmm3
41          addss xmm2, xmm3
42          sqrtss xmm2, xmm2
43
44          minss xmm0, xmm2
45
46          mulss xmm0, xmm5
47          addss xmm0, xmm0
48
49          add esi, 1
```

```
50          cmp esi, 256
51          jl .loop
52
53      movss [ecx], xmm0
54
55      pop edi
56      pop esi
57      pop ebx
58      mov esp, ebp
59      pop ebp
60      ret
```

Ulteriori ottimizzazioni del codice C

In questa versione, prima di analizzare eventuali nuove procedure da scrivere in assembly ci siamo concentrati sull'ottimizzazione del codice C, eliminato tutti i for evitabili, come ad esempio quelli che compievano solo quattro iterazioni. Tramite questa ottimizzazione il processore ha potuto effettuare i calcolare in parallelo i risultati attesi e migliorare le prestazioni.

- Tempo di Esecuzione: **0,370 secondi.**

Nella versione iniziale in C, venivano effettuate numerose chiamate a funzioni, il che comportava un notevole incremento dell'overhead e, di conseguenza, un aumento del tempo di esecuzione. Per ovviare a questo problema, abbiamo eseguito direttamente all'interno delle funzioni operazioni come divisioni, moltiplicazioni tra vettori e calcoli della distanza euclidea, evitando così di fare chiamate a funzioni esterne.

Versione C senza chiamate a funzioni per operazioni specifiche

In questa versione, come detto precedentemente ,abbiamo eliminato le eccessive chiamate a funzioni, evitando di allocare i record di attivazione i

risultati attesi sono migliorati.

- Tempo di Esecuzione: **0,333 secondi.**

Rimuovendo chiamate a funzioni esterne per operazioni come distanza euclidea, la normalizzazione, prodotti scalari e divisioni scalari, si è ridotto ulteriormente il sovraccarico computazionale, migliorando la velocità di esecuzione.

Versione C senza chiamate a funzioni per prodotto matrice-vettore

Anche in questa versione abbiamo evitato ulteriori chiamante a funzioni inutili. Grazie a questo abbiamo evitato di allocare altri record di attivazione i risultati attesi sono migliorati.

Tempo di Esecuzione: **0,320 secondi.**

Una semplificazione ulteriore è stata apportata eliminando le chiamate per il prodotto matrice-vettore, consolidando i calcoli direttamente nel codice principale.

Versione C con implementazioni in assembly di rama_energy e hydrophobic_energy

In questa versione è stata implementata anche hydrophobic_energy in assembly.

Tempo di Esecuzione: **0,240 secondi.**

Integrando questa ottimizzazione con quella già implementata per rama_energy, si è registrato uno dei miglioramenti più significativi. La procedura ha il seguente codice:

Listing 3: hydrophobic_energy

```
hydrophobic_energy_assembly:
    push ebp
    mov ebp, esp
    push ebx
    push esi
    push edi

    mov eax, [ebp + 8]
    mov ebx, [ebp + 12]
    mov ecx, [ebp + 16]
    mov edx, [ebp + 20]

    xorps xmm0, xmm0

    mov esi, 0
    .outer_loop:
        cmp esi, ebx
        jge .end_outer_loop

        mov edi, esi
        add edi, 1

        mov eax, ecx
        add eax, esi
        shl eax, 2
        add eax, ecx
        movaps xmm1, [eax + 12]

        .inner_loop:
            cmp edi, ebx
            jge .end_inner_loop

            mov eax, ecx
            add eax, edi
            shl eax, 2
            add eax, ecx
```

```
37              movaps xmm2, [eax + 12]
38
39              movaps xmm3, xmm2
40              subps xmm3, xmm1
41              mulps xmm3, xmm3
42              movhlps xmm4, xmm3
43              addps xmm3, xmm4
44              movaps xmm4, xmm3
45              shufps xmm4, xmm4, 1
46              addss xmm3, xmm4
47              sqrtss xmm3, xmm3
48
49              movzx edx, byte [eax + esi]
50              sub edx, 65
51              movss xmm4, [hydrophobicity + edx * 4]
52
53              movzx edx, byte [eax + edi]
54              sub edx, 65
55              movss xmm5, [hydrophobicity + edx * 4]
56
57              mulss xmm4, xmm5
58              divss xmm4, xmm3
59
60              addss xmm0, xmm4
61
62              add edi, 1
63              jmp .inner_loop
64
65          .end_inner_loop:
66          add esi, 1
67          jmp .outer_loop
68
69      .end_outer_loop:
70      movss [edx], xmm0
71
72      pop edi
73      pop esi
```

```
74      pop ebx
75      mov esp, ebp
76      pop ebp
77      ret
```

Versione C con implementazioni in assembly di rama_energy, hydrophic_energy e electrostatic_energy

In questa versione è stata implementata anche electrostatic_energy

- Tempo di Esecuzione: **0,230 secondi.**

Questa ulteriore ottimizzazione ha portato a un bilanciamento tra accuratezza e prestazioni, pur mantenendo un tempo molto competitivo se pur con un leggero miglioramento rispetto alla versione precedente.

Listing 4: electrostatic_energy

```
1  electrostatic_energy_assembly:
2      push ebp
3      mov ebp, esp
4      push ebx
5      push esi
6      push edi
7
8      mov eax, [ebp + 8]
9      mov ebx, [ebp + 12]
10     mov ecx, [ebp + 16]
11     mov edx, [ebp + 20]
12
13     xorps xmm0, xmm0
14
15     mov esi, 0
16     .outer_loop:
17         cmp esi, ebx
```

```
18          jge .end_outer_loop
19
20          mov edi, esi
21          add edi, 1
22
23          mov eax, ecx
24          add eax, esi
25          shl eax, 4
26          add eax, ecx
27          movaps xmm1, [eax + 12]
28
29          .inner_loop:
30              cmp edi, ebx
31              jge .end_inner_loop
32
33              mov eax, ecx
34              add eax, edi
35              shl eax, 4
36              add eax, ecx
37              movaps xmm2, [eax + 12]
38
39              movaps xmm3, xmm2
40              subps xmm3, xmm1
41              mulps xmm3, xmm3
42              movhlps xmm4, xmm3
43              addps xmm3, xmm4
44              movaps xmm4, xmm3
45              shufps xmm4, xmm4, 1
46              addss xmm3, xmm4
47              sqrtss xmm3, xmm3
48
49              movzx edx, byte [eax + esi]
50              sub edx, 65
51              movss xmm4, [charge + edx * 4]
52
53              movzx edx, byte [eax + edi]
54              sub edx, 65
```

```
55              movss xmm5, [charge + edx * 4]
56
57              mulss xmm4, xmm5
58              divss xmm4, xmm3
59              divss xmm4, dword [four]
60
61              addss xmm0, xmm4
62
63              add edi, 1
64              jmp .inner_loop
65
66          .end_inner_loop:
67          add esi, 1
68          jmp .outer_loop
69
70      .end_outer_loop:
71      movss [edx], xmm0
72
73      pop edi
74      pop esi
75      pop ebx
76      mov esp, ebp
77      pop ebp
78      ret
```

Architettura x86-64 con supporto AVX

Durante lo sviluppo del progetto, sono state apportate diverse ottimizzazioni mirate a migliorare i tempi di esecuzione sulle architetture a 64 bit. Tuttavia, l'analisi relativa ai tempi di esecuzione a 64 bit evidenzia un aspetto cruciale: in molte implementazioni non si sono riscontrati miglioramenti significativi.

Anche in questo capitolo le implementazioni verranno descritte nell'ordine nella quale sono state affrontate.

N.B. : i test effettuati per la versione a 32bit rispetto a questa a 64 bit sono stati effettuati su due macchine con un architettura hardware diversa.

Versione in C pura

La prima parte dell'implementazione a 64 bit si è concentrata sull'implementazione lato C, che ovviamente era già pronta vista la precedente implementazione 32 bit, con le stesse ottimizzazioni già presenti nel codice C usato per la versione a 32.

- Tempo di Esecuzione: **0,650 secondi.**

La versione in C puro mostra un tempo di esecuzione riportato, ma non vi sono miglioramenti rispetto alla controparte a 32 bit. Questo risultato indica che il codice C non sfrutta appieno le potenzialità offerte dai registri e dalle ottimizzazioni native dei processori a 64 bit.

Versione C con l'implementazione in assembly di euclidean-dist e rama-energy-assembly

Come nella versione a 32 bit, la scelta di implementare la distanza euclidea non è stata ottimale, per cui l'implementazione è stata scartata.

Listing 5: euclideandistavx

```
euclidean_dist_avx:
        push    rbp
        mov     rbp, rsp
        pushaq

        MOV RAX, [RBP+A]
        MOV RBX, [RBP+B]
        MOV RCX, [RBP+C]

        VMOVUPD   YMM0, [RDI]
        VMOVUPD   YMM1, [RSI]
        VSUBPD YMM1,YMM0, YMM1
        VMULPD YMM1,YMM1, YMM1
        VMOVSD [e], XMM1

        VHADDPD YMM1, YMM1
        VHADDPD YMM1, YMM1
        VPERM2F128 YMM0,YMM1,YMM1,00000001b
        VADDSD XMM1,XMM0
        VMOVSD [e], XMM1

        VSQRTPD YMM1,YMM1
          VMOVSD [e], XMM1

        VMOVSD [RDX], XMM1
        VMOVSD [e], XMM1

        popaq
        mov             rsp, rbp
```

```
30          pop             rbp
31          ret
```

- Tempo di Esecuzione: **0,730 secondi.**

Questo suggerisce che l'ottimizzazione mirata come nell'architettura a 32 bit non ha prodotto vantaggi misurabili o significativi quando eseguita su sistemi a 64 bit.

Versione C con l'implementazione in assembly di rama_energy

La sola implementazione della funzione rama-energy, invece si è rivelata più efficiente della precedente. Tutto ciò era dovuto all'eccessivo overhead creato dalle eccessive chiamate alla funzione euclidean-distance. Per evitare questo abbiamo inserito il calcolo della distanza direttamente all'interno delle funzioni che la necessitano.

- Tempo di esecuzione: **0,550 secondi**

L'implementazione effettuata in assembly è la seguente:

Listing 6: rama_energy_assembly

```
1  rama_energy_assembly:
2      push    rbp
3      mov     rbp, rsp
4      pushaq
5
6      mov     rdi, [rbp + 16]
7      mov     rsi, [rbp + 24]
8
9      vmovsd  xmm0, qword [alpha_phi]
10     vbroadcastsd ymm0, xmm0
11     vmovsd  xmm1, qword [alpha_psi]
12     vbroadcastsd ymm1, xmm1
```

```
13      vmovsd  xmm2, qword [beta_phi]
14      vbroadcastsd ymm2, xmm2
15      vmovsd  xmm3, qword [beta_psi]
16      vbroadcastsd ymm3, xmm3
17
18      vxorpd  ymm4, ymm4, ymm4
19
20      mov     rcx, 256
21      xor     r8, r8
22
23  .loop:
24
25      vmovsd  xmm5, qword [rdi + r8 * 8]
26      vbroadcastsd ymm5, xmm5
27      vmovsd  xmm6, qword [rsi + r8 * 8]
28      vbroadcastsd ymm6, xmm6
29
30      vsubpd  ymm7, ymm5, ymm0
31      vmulpd  ymm7, ymm7, ymm7
32
33      vsubpd  ymm8, ymm6, ymm1
34      vmulpd  ymm8, ymm8, ymm8
35
36      vaddpd  ymm7, ymm7, ymm8
37
38      vsqrtpd ymm7, ymm7
39
40      vsubpd  ymm8, ymm5, ymm2
41      vmulpd  ymm8, ymm8, ymm8
42
43      vsubpd  ymm9, ymm6, ymm3
44      vmulpd  ymm9, ymm9, ymm9
45
46      vaddpd  ymm8, ymm8, ymm9
47
48      vsqrtpd ymm8, ymm8
49
```

```
50      vminpd   ymm7, ymm7, ymm8
51
52      vmulpd   ymm7, ymm7, [half]
53      vaddpd   ymm4, ymm4, ymm7
54
55      inc      r8
56      loop     .loop
57      vmovsd   qword [rdx], xmm4
58
59      popaq
60      mov      rsp, rbp
61      pop      rbp
62      ret
```

Versione C con implementazioni in assembly di rama_energy e hydrophobic_energy_assembly

L'implementazione in assembly di hydrophobic_energy e l'utilizzo della procedura rama_energy, precedentemente implementata, ha portato ad un ulteriore miglioramento

- Tempo di esecuzione: **0,490 secondi**

La funzione è stata così implementata:

Listing 7: hydrophobic_energy_assembly

```
1
2       hydrophobic_energy_assembly:
3       push     rbp
4       mov      rbp, rsp
5       pushaq
6
7       vxorpd xmm0, xmm0, xmm0
8
9       xor r8, r8
10  outer_loop_hydro:
```

```
    cmp r8, rsi
    jge done_hydro

    mov r9, r8
    imul r9, 9
    add r9, 3

    vmovupd ymm1, [rdx + r9*8]

    mov r10, r8
    inc r10

inner_loop_hydro:
    cmp r10, rsi
    jge outer_loop_end_hydro

    mov r11, r10
    imul r11, 9
    add r11, 3

    vmovupd ymm2, [rdx + r11*8]

    vsubpd ymm3, ymm2, ymm1
    vmulpd ymm3, ymm3, ymm3
    vhaddpd ymm3, ymm3, ymm3
    vhaddpd ymm3, ymm3, ymm3

    vsqrtpd ymm3, ymm3

    vmovsd xmm4, [ten]
    vucomisd xmm3, xmm4
    jae skip_update_hydro

    movzx r12, byte [rdi + r8]
    sub r12, 65
    movzx r13, byte [rdi + r10]
    sub r13, 65
```

```
48
49      vmovsd xmm4, [hydrophobicity + r12*8]
50      vmovsd xmm5, [hydrophobicity + r13*8]
51
52      vmulsd xmm4, xmm4, xmm5
53      vdivsd xmm4, xmm4, xmm3
54      vaddsd xmm0, xmm0, xmm4
55
56  skip_update_hydro:
57      inc r10
58      jmp inner_loop_hydro
59
60  outer_loop_end_hydro:
61      inc r8
62      jmp outer_loop_hydro
63
64  done_hydro:
65      vmovsd [rcx], xmm0
66
67      popaq
68      mov rsp, rbp
69      pop rbp
70      ret
```

Versione C con implementazioni in assembly di rama_energy, hydrophobic_energy e electrostatic_energy)

In questa versione è stata aggiunta l'implementazione in assembly di electrostatic_energy.

- Tempo di esecuzione: **0,450 secondi**

La funzione è stata così implementata:

Listing 8: electrostatic_energy_assembly

```
electrostatic_energy_assembly:

    push    rbp
    mov     rbp, rsp
    pushaq


    vxorpd xmm0, xmm0, xmm0

    xor r8, r8
outer_loop:
    cmp r8, rsi
    jge done


    mov r9, r8
    imul r9, 9
    add r9, 3


    vmovupd ymm1, [rdx + r9*8]


    mov r10, r8
    inc r10

inner_loop:
    cmp r10, rsi
    jge outer_loop_end


    mov r11, r10
    imul r11, 9
    add r11, 3


```

```
37      vmovupd ymm2, [rdx + r11*8]
38
39
40      vsubpd ymm3, ymm2, ymm1
41      vmulpd ymm3, ymm3, ymm3
42      vhaddpd ymm3, ymm3, ymm3
43      vhaddpd ymm3, ymm3, ymm3
44
45      vsqrtpd ymm3, ymm3
46
47
48      vmovsd xmm4, [ten]
49      vucomisd xmm3, xmm4
50      jae skip_update
51
52
53      movzx r12, byte [rdi + r8]
54      sub r12, 65
55      movzx r13, byte [rdi + r10]
56      sub r13, 65
57
58      vmovsd xmm4, [charge + r12*8]
59      vmovsd xmm5, [charge + r13*8]
60
61
62      vxorpd xmm6, xmm6, xmm6
63      vucomisd xmm4, xmm6
64      je skip_update
65      vucomisd xmm5, xmm6
66      je skip_update
67
68
69      vmulsd xmm4, xmm4, xmm5
70      vmulsd xmm5, xmm3, [four]
71      vdivsd xmm4, xmm4, xmm5
72      vaddsd xmm0, xmm0, xmm4
73
```

```
74 skip_update:
75     inc r10
76     jmp inner_loop
77
78 outer_loop_end:
79     inc r8
80     jmp outer_loop
81
82 done:
83     vmovsd [rcx], xmm0
84
85     popaq
86     mov rsp, rbp
87     pop rbp
88     ret
```

Architettura x86-64 con supporto AVX e OpenMP

Nell'ultima versione del progetto da implementare, cioè quella che sfruttava OpenMP abbiamo inizialmente modificato le modalità di calcolo del tempo di esecuzione, introducendo i seguenti comandi:

```
type startTime = omp_get_wtime();
pst(input);
type endTime = omp_get_wtime();
time = endTime - startTime;
```

Successivamente ci siamo concentrati sull'individuare le probabilità funzioni parallelizzabile e poi sulla loro implementazione. Abbiamo sfruttato le procedure assembly, precedentemente sviluppate,come rama_energy e hydrophobic_energy e l'analisi del codice C di partenza ci ha permesso di individuare una funzione parallelizzabile che è electrostatic_energy.

Listing 9: Electrostatic Energy avx

```
type electrostatic_energy(char *s, int n, MATRIX coords)
{
    type energy = 0;
    int i, j, k;
    VECTOR coords_c_alpha_i = alloc_matrix(1, 4);
    VECTOR coords_c_alpha_j = alloc_matrix(1, 4);

#pragma omp parallel for num_threads(4)
    for (i = 0; i < n; i++)
    {
```

```
11          int idx_i = i * 3 * 3 + 3;
12
13          coords_c_alpha_i[0] = coords[idx_i];
14          coords_c_alpha_i[1] = coords[idx_i + 1];
15          coords_c_alpha_i[2] = coords[idx_i + 2];
16          coords_c_alpha_i[3] = 0;
17
18          inner_for_elec(i, n, coords, coords_c_alpha_i,
                coords_c_alpha_j, energy, s);
19      }
20      return energy;
21  }
22  void inner_for_elec(int i, int n, MATRIX coords, VECTOR
        coords_c_alpha_i, VECTOR coords_c_alpha_j, type energy, char
        *s)
23  {
24      for (int j = i + 1; j < n; j++)
25      {
26
27          int idx_j = j * 3 * 3 + 3;
28
29          coords_c_alpha_j[0] = coords[idx_j];
30          coords_c_alpha_j[1] = coords[idx_j + 1];
31          coords_c_alpha_j[2] = coords[idx_j + 2];
32          coords_c_alpha_j[3] = 0;
33          type dist;
34
35          dist = sqrtf((coords_c_alpha_j[0] - coords_c_alpha_i
                [0]) * (coords_c_alpha_j[0] - coords_c_alpha_i[0]) +
                 (coords_c_alpha_j[1] - coords_c_alpha_i[1]) * (
                coords_c_alpha_j[1] - coords_c_alpha_i[1]) + (
                coords_c_alpha_j[2] - coords_c_alpha_i[2]) * (
                coords_c_alpha_j[2] - coords_c_alpha_i[2]));
36
37          int pos_i = s[i] - 65;
38          int pos_j = s[j] - 65;
39
```

```
40          if (i != j && dist < 10.0 && charge[pos_i] != 0 &&
                charge[pos_j] != 0)
41              energy = energy + (charge[pos_i] * charge[pos_j]) /
                    (dist * 4.0);
42      }
43  }
```

L'utilizzo del costrutto parallel for all'interno di questa funzione è stato possibile inserendo il for interno come funzione così facendo abbiamo parallelizzato i calcoli, evitando risultati sbagliati dovuti alla dipendenza dei due indici i e j.

- Tempo di esecuzione: **0,550 secondi**

Conclusioni

La stesura di questo progetto si è rivelata particolarmente utile non solo per acquisire una comprensione più profonda delle tecnologie implementate, ma anche per esplorare il loro potenziale impiego in contesti reali.

L'integrazione di SSE, AVX e OpenMP ha rappresentato un'opportunità preziosa per imparare e applicare diverse strategie di ottimizzazione, migliorando notevolmente le prestazioni del programma. Queste tecnologie, infatti, offrono approcci avanzati per sfruttare al massimo le risorse hardware disponibili, consentendo di eseguire operazioni in parallelo e di ridurre i tempi di calcolo.

Un ulteriore aspetto che abbiamo appreso durante il processo di sviluppo è che le ottimizzazioni non dipendono esclusivamente dall'utilizzo di queste tecnologie, ma possono anche essere conseguite modificando direttamente il codice C. In effetti, l'analisi e la revisione del codice a livello di alto livello sono state cruciali per eliminare inefficienze, ridurre il carico computazionale e migliorare la gestione delle risorse. Ottimizzare il flusso logico e le strutture dati del programma è stato altrettanto importante quanto l'applicazione di tecniche di parallelizzazione avanzate, poiché una base solida di codice C ottimizzato può portare a miglioramenti significativi senza la necessità di complessi interventi hardware.

Inoltre, abbiamo compreso che il processo di ottimizzazione è un lavoro continuo, in cui il bilanciamento tra il miglioramento delle performance e la chiarezza del codice è fondamentale. L'approfondimento delle caratteristiche di SSE, AVX e OpenMP ci ha fornito gli strumenti per migliorare le prestazioni, ma la gestione accurata e strategica del codice sorgente è risultata

ugualmente essenziale per garantire un risultato finale efficiente e sostenibile. Questo progetto ci ha così dato una visione complessa e multidimensionale di come le ottimizzazioni possano essere integrate in modo sinergico per ottenere soluzioni altamente performanti in ambito scientifico e tecnologico.