Progetto Architettura degli Elaboratori

Sessione estiva

Anno accademico 2021/2022

Adam El Idrissi

numero matricola 299083

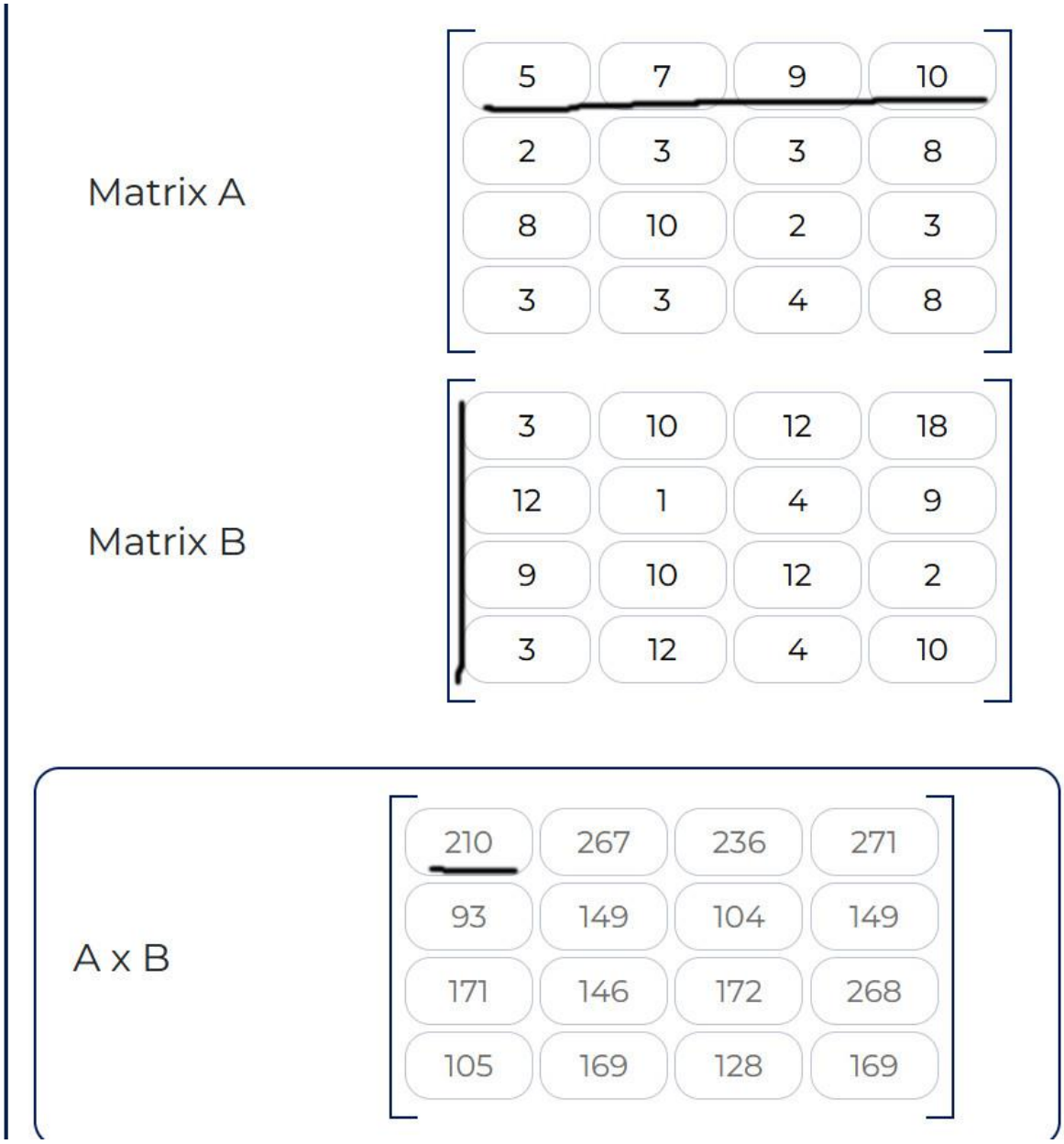
Indice contenuti:

Introduzione

Il progetto è composto dall'implementazione in linguaggio Assembly relativo al simulatore WinMips64 di un moltiplicatore di matrici 4x4. Di seguito verranno riportate le versioni del codice in linguaggio C e in Assembly, con le conseguenti prestazioni e migliorie effettuate. Le 2 specifiche principali tenute in considerazione per la miglioria delle prestazioni sono il minor numero di cicli di clock di esecuzione ed il minor numero possibile di stalli che possano rallentare l'esecuzione.

Descrizione algoritmo scelto

L'algoritmo implementato è una moltiplicazione di matrici 4x4, che moltiplica riga per colonna la prima per la seconda matrice e registra i risultati in una matrice apposita. Il calcolo dei valori si effettua moltiplicando i valori di una riga per i valori di una colonna e sommando i risultati parziali, come nelle seguenti figure di esempio:



(5 * 3) + (7 * 12) + (9 * 9) + (10 * 3) = 210

Matrix A

5	7	9	10
2	3	3	8
8	10	2	3
3	3	4	8

Matrix B

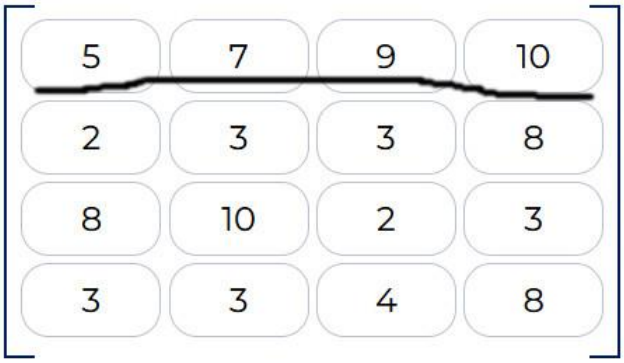
3	10	12	18
12	1	4	9
9	10	12	2
3	12	4	10

A x B

210	267	236	271
93	149	104	149
171	146	172	268
105	169	128	169

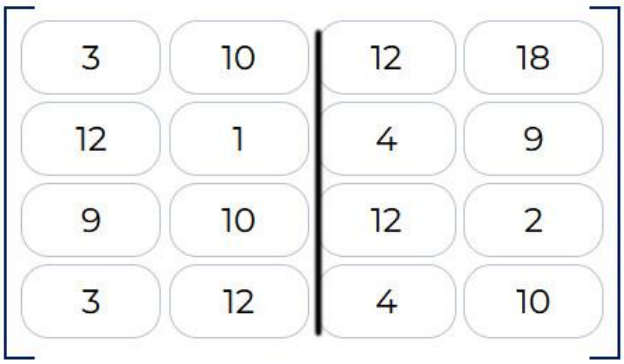
(5* 10) + (7 * 1) + (9 *10) + (10 * 12) = 267

Matrix A



5	7	9	10
2	3	3	8
8	10	2	3
3	3	4	8

Matrix B



3	10	12	18
12	1	4	9
9	10	12	2
3	12	4	10

A x B

210	267	236	271
93	149	104	149
171	146	172	268
105	169	128	169

(5 * 12) + (7 * 4) + (9 * 12) + (10 * 4) = 236

Matrix A

5	7	9	10
2	3	3	8
8	10	2	3
3	3	4	8

Matrix B

3	10	12	18
12	1	4	9
9	10	12	2
3	12	4	10

A x B

210	267	236	271
93	149	104	149
171	146	172	268
105	169	128	169

(2 * 3) + (3 * 12) + (3 * 9) + (8 * 3) = 93

Implementazione in linguaggio C

Viene sotto riportata l'implementazione in linguaggio C

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    // Definizione matrici
    int A[4][4] = {{5, 7, 9, 10}, {2, 3, 3, 8}, {8, 10, 2, 3}, {3, 3, 4, 8}};       // Matrice input A
    int B[4][4] = {{3, 10, 12, 18}, {12, 1, 4, 9}, {9, 10, 12, 2}, {3, 12, 4, 10}}; // Matrice input B
    int C[4][4] = {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}};        // Matrice risultato C
    int i, j, k;    // Contatori loop

    // Esecuzione della moltiplicazione
    for (i = 0; i < 4; i++)
    {
        for (j = 0; j < 4; j++)
        {
            for (k = 0; k < 4; k++)
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
```

Come si può notare, tutto ciò di cui si ha bisogno sono dei loop che permettano di scorrere le posizioni necessarie ed effettuare calcoli con i valori contenuti in esse, il risultato verrà poi inserito nella corrispondente posizione della matrice risultato.

Implementazione in linguaggio Assembly per WinMips64

Viene riportata sotto la prima versione dell'implementazione in Assembly

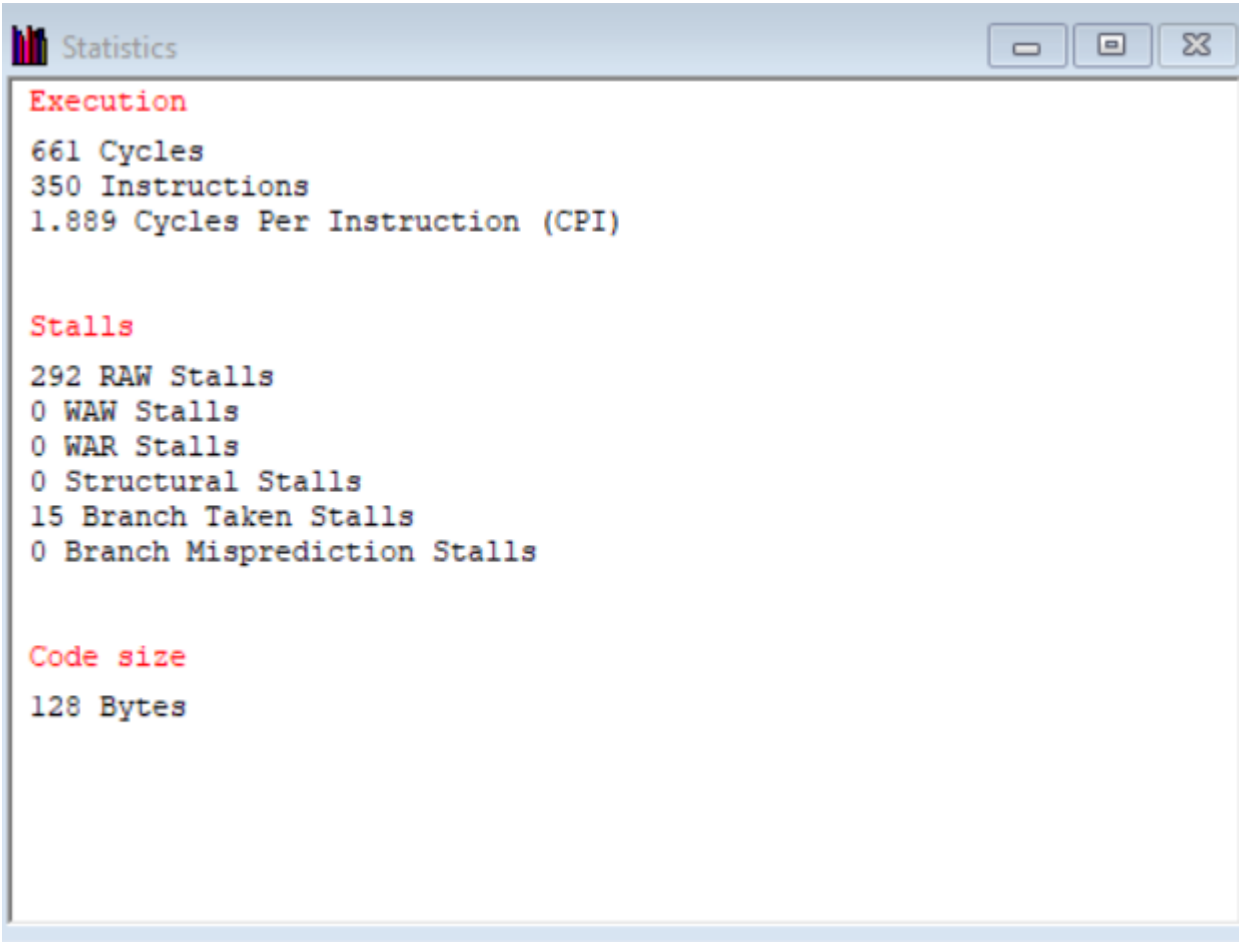
Nota: I valori delle matrici presenti nei prossimi esempi sono diversi dai precedenti in modo tale da non utilizzare solo valori interi.

```
    .data
a1:         .double 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0; elements first matrix
b1:         .double 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5, 9.5, 10.5, 11.5, 12.5, 13.5, 14.5, 15.5, 16.5; elements second matrix
c1:         .double 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0;        first row result matrix
n1:         .word   4                       ; counter for rows of the result matrix
n2:         .word   4                       ; counter for columns of the result matrix

    .text
start: LW   r1, n1(r0)          ; loading numbers of elements in matrices (loop counter)
       LW   r5, n2(r0)          ; loading numbers of elements in matrices (loop counter)
    DADDI   r2, r0, a1          ; pointer to the first element in the first matrix
    DADDI   r3, r0, b1          ; pointer to the first element in the second matrix
    DADDI   r4, r0, c1          ; pointer to the first element in the result matrix


loop1: L.D  f0, 0(r2)           ; read a1[i]
    L.D     f1, 8(r2)           ; read a1[i+1]
    L.D     f2, 16(r2)          ; read a1[i+2]
    L.D     f3, 24(r2)          ; read a1[i+3]
    L.D     f4, 0(r3)           ; read b1[i]
    L.D     f5, 32(r3)          ; read b2[i]
    L.D     f6, 64(r3)          ; read b3[i]
    L.D     f7, 96(r3)          ; read b4[i]

    MUL.D   f8, f0, f4          ; multiply the value of f0 to f4 (a1[i] * b1[i])
    MUL.D   f9, f1, f5          ; multiply the value of f1 to f5 (a1[i+1] * b2[i])
    MUL.D   f10, f2, f6         ; multiply the value of f2 to f6 (a1[i+2] * b3[i])
    MUL.D   f11, f3, f7         ; multiply the value of f3 to f7 (a1[i+3] * b4[i])
    ADD.D   f12, f8, f9         ; add elements for the first row of the result matrix
    ADD.D   f13, f10, f11       ; add elements for the first row of the result matrix
    ADD.D   f14, f12, f13       ; add elements for the first row of the result matrix
    S.D     f14, 0(r4)          ; write the result in c1[i]
    DADDI   r2, r2, 32          ; move to the next row (a2[i])
    DADDI   r1, r1, -1          ; decrement the loop counter
    DADDI   r4, r4, 32          ; move to the next the element of the result matrix
    BNEZ    r1, loop1           ; jump to loop if not equal 0

    DADDI   r3, r3, 8           ; move to the next row (b1[i+1])
    DADDI   r2, r0, a1          ; pointer to the first element in the first matrix
    DADDI   r4, r4, 8           ; move to the next column of the result matrix
    DADDI   r5, r5, -1          ; decrement the loop counter
    DADDI   r1, r1, 4           ; increment the first loop counter for another column of the result matrix
    BNEZ    r5, loop1           ; jump to loop if not equal 0
end:
    HALT
```

Descrizione e prestazioni codice

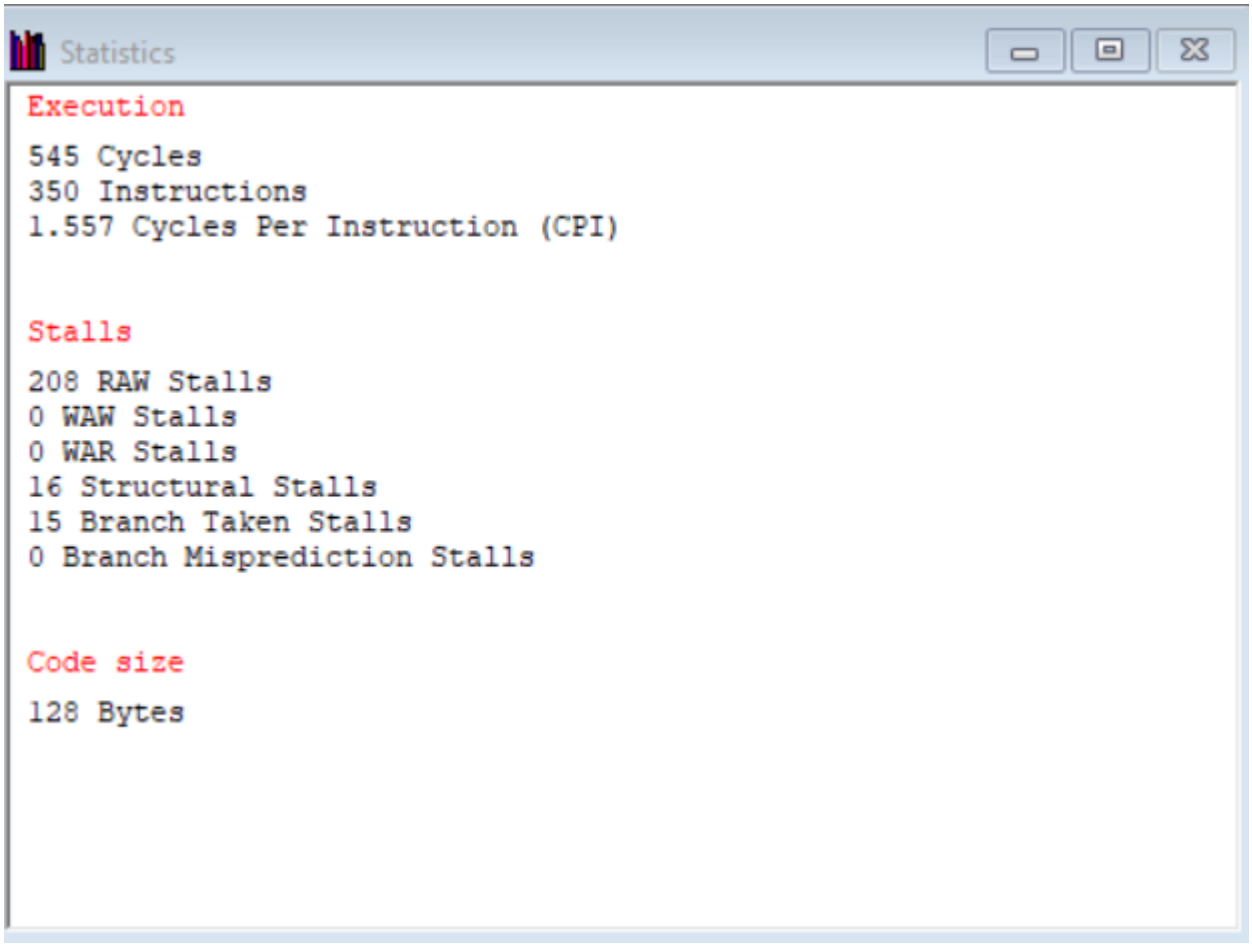
Questa versione si serve di 2 loop con il primo innestato nel secondo. Tramite il primo loop viene calcolato un risultato facente parte della prima colonna della matrice risultato per 4 volte (formando perciò la prima colonna risultato), per passare poi alla colonna successiva e ripetere il primo loop per in totale 4 volte (ovvero il numero di colonne che la compongono) tramite il secondo loop, producendo quindi i risultati sotto forma di colonne di valori, per un totale di 16 iterazioni.

Prestazioni con il data forwarding disattivato:



Il data forwarding permette di utilizzare un dato senza essere prima necessariamente salvato in memoria, ovvero di utilizzarlo non appena disponibile dalla fase di esecuzione EX o MA. Ciò permette di migliorare le prestazioni del codice sia per quanto riguarda i cicli di clock totali che per il CPI ed il numero di stalli, in particolare gli stalli RAW (read after write) creati dalla necessità di leggere un dato che non è ancora stato riscritto in memoria e quindi non è teoricamente pronto per essere riletto, bloccando l'esecuzione delle istruzioni successive.

Prestazioni con il data forwarding attivato:



Le prestazioni sono migliorate con ben 116 cicli di clock di esecuzione in meno, il CPI è notevolmente diminuito, il numero di stalli RAW è diminuito da 292 a 208, gli unici stalli che si sono aggiunti sono gli stalli strutturali, i quali sono però inevitabili con operazioni che richiedono più cicli per l'esecuzione, tutto ciò a parità di grandezza di codice e numero istruzioni eseguite.

Ottimizzazioni e prestazioni codice ottimizzato

Le ottimizzazioni sono delle modificazioni del codice volte al miglioramento di esso stesso, sotto forma di riduzione del numero di cicli di clock d'esecuzione o riduzione del numero di stalli totali. Le principali forme di ottimizzazioni statiche (rivolte strettamente al codice) sono *Loop Unrolling*, *Instruction Reordering* e *Register Renaming*.

Il *Loop Unrolling* consiste nello srotolare parzialmente o totalmente i loop presenti nel codice, diminuendo o eliminando i vari stalli creati dai salti stessi, diminuendo anche il numero di cicli d'esecuzione totali e velocizzando l'esecuzione. Il grado di srotolamento dipende anche dalle specifiche a cui dare più importanza, questo poiché srotolare uno o più cicli aumenta la grandezza del codice: con disponibilità di spazio limitato si può preferire un codice meno efficiente ma più compresso, se invece il parametro più importante è il tempo (come nel caso del presente progetto), si preferisce un codice più pesante ma allo stesso tempo più efficiente.

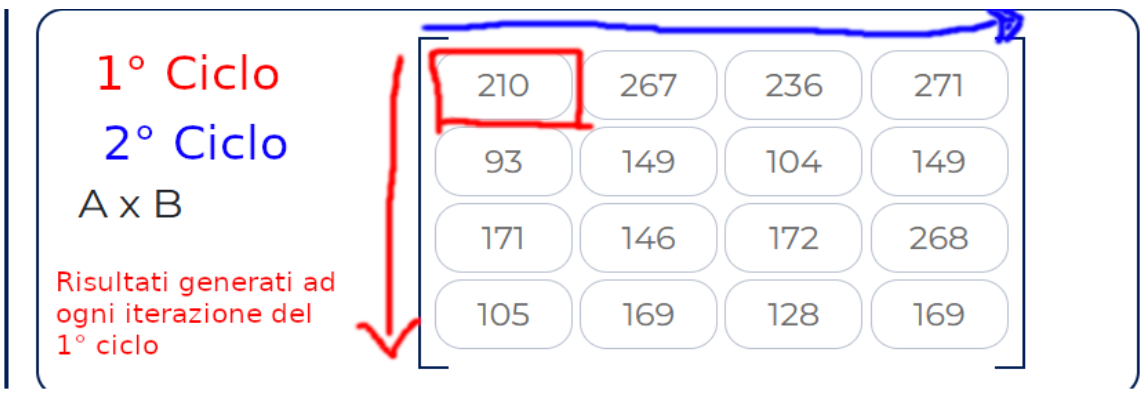
L' *Instruction Reordering* consiste nel riordinare la posizione delle istruzioni per fare in modo che gli stalli creati da alcune operazioni vengano colmati con l'esecuzione di altre operazioni, rendendo il codice più efficiente e diminuendo notevolmente il numero di stalli RAW, WAR e WAW.

Il *Register Renaming* consiste nel rinominare dei registri interni rendendoli riutilizzabili, sovrascrivendo i dati nei registri che non sono più necessari durante l'esecuzione, ciò permette di ottimizzare ed apparentemente incrementare lo spazio a disposizione.

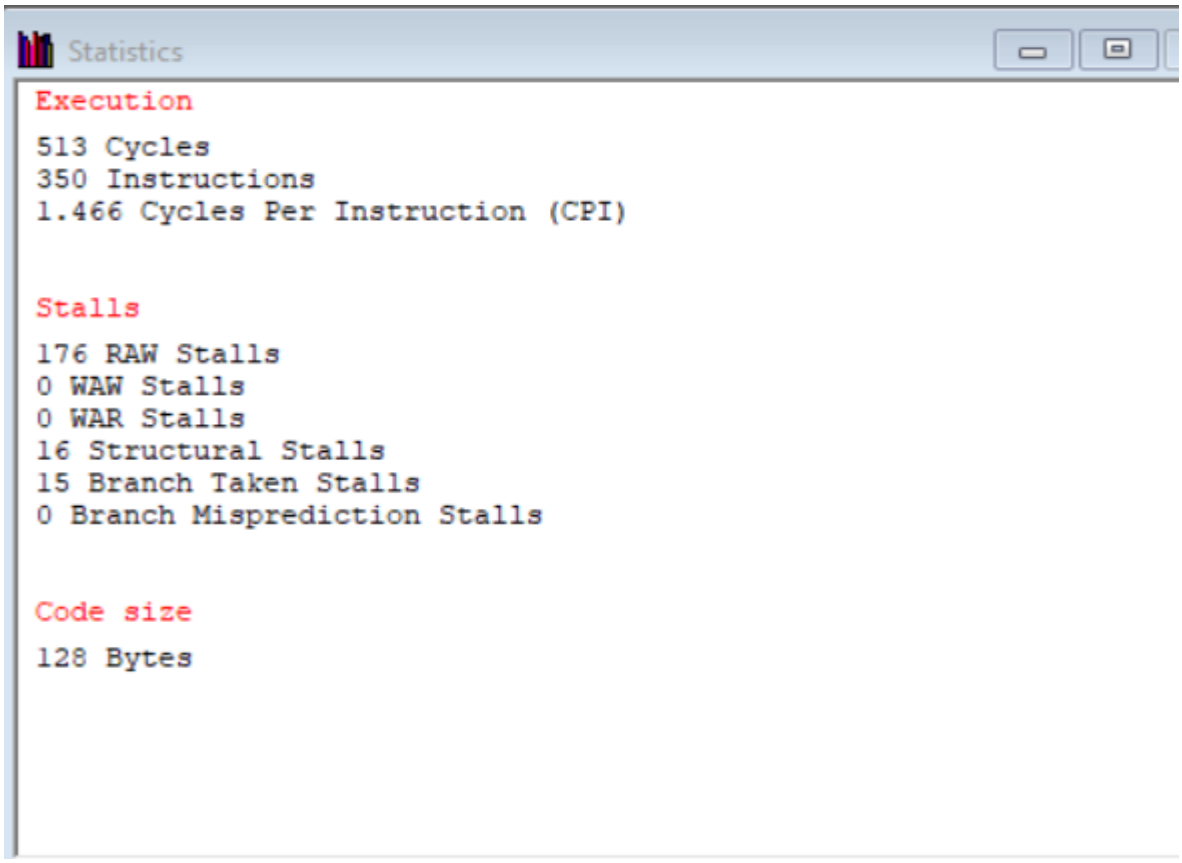
La prima ottimizzazione effettuata è un *Instruction Reordering*, permettendo di sfruttare i cicli di clock che corrisponderebbero ad uno stallo per altre istruzioni:

```
    .data
a1:         .double 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0; elements first matrix
b1:         .double 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5, 9.5, 10.5, 11.5, 12.5, 13.5, 14.5, 15.5, 16.5; elements second matrix
c1:         .double 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0;        first row result matrix
n1:         .word   4                       ; counter for rows of the result matrix
n2:         .word   4                       ; counter for columns of the result matrix

    .text
start: LW   r1, n1(r0)          ; loading numbers of elements in matrices (loop counter)
       LW   r5, n2(r0)          ; loading numbers of elements in matrices (loop counter)
    DADDI   r2, r0, a1          ; pointer to the first element in the first matrix
    DADDI   r3, r0, b1          ; pointer to the first element in the second matrix
    DADDI   r4, r0, c1          ; pointer to the first element in the result matrix


loop1: L.D  f0, 0(r2)           ; read a1[i]
    L.D     f1, 8(r2)           ; read a1[i+1]
    L.D     f2, 16(r2)          ; read a1[i+2]
    L.D     f3, 24(r2)          ; read a1[i+3]
    L.D     f4, 0(r3)           ; read b1[i]
    L.D     f5, 32(r3)          ; read b2[i]
    L.D     f6, 64(r3)          ; read b3[i]
    L.D     f7, 96(r3)          ; read b4[i]

    MUL.D   f8, f0, f4          ; multiply the value of f0 to f4 (a1[i] * b1[i])
    MUL.D   f9, f1, f5          ; multiply the value of f1 to f5 (a1[i+1] * b2[i])
    MUL.D   f10, f2, f6         ; multiply the value of f2 to f6 (a1[i+2] * b3[i])
    MUL.D   f11, f3, f7         ; multiply the value of f3 to f7 (a1[i+3] * b4[i])
    DADDI   r2, r2, 32          ; move to the next row (a2[i])
    DADDI   r1, r1, -1          ; decrement the loop counter
    ADD.D   f12, f8, f9         ; add elements for the first row of the result matrix
    ADD.D   f13, f10, f11       ; add elements for the first row of the result matrix
    ADD.D   f14, f12, f13       ; add elements for the first row of the result matrix
    S.D     f14, 0(r4)          ; write the result in c1[i]
    DADDI   r4, r4, 32          ; move to the next the element of the result matrix
    BNEZ    r1, loop1           ; jump to loop if not equal 0

    DADDI   r3, r3, 8           ; move to the next row (b1[i+1])
    DADDI   r2, r0, a1          ; pointer to the first element in the first matrix
    DADDI   r4, r4, 8           ; move to the next column of the result matrix
    DADDI   r5, r5, -1          ; decrement the loop counter
    DADDI   r1, r1, 4           ; increment the first loop counter for another column of the result matrix
    BNEZ    r5, loop1           ; jump to loop if not equal 0
end:
    HALT
```

Tra le istruzioni MUL.D e le ADD.D vengono inserite le operazioni DADDI, creando per quanto possibile distanza tra i 2 set di istruzioni che altrimenti genererebbero degli stalli RAW, poiché al momento di richiamo dei dati da parte delle istruzioni di addizione, essi non sono ancora usciti dalla fase di EX delle moltiplicazioni, ovvero non sono ancora pronti.



Prestazioni del codice ottimizzato:

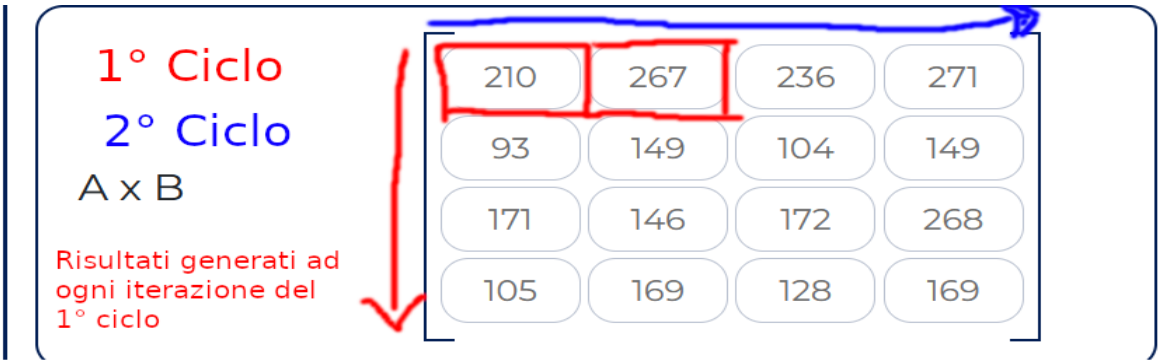


È possibile notare un non indifferente miglioramento delle prestazioni, in particolare 32 cicli di clock d'esecuzione in meno, un CPI migliore ed un minor numero di stalli RAW. Alcuni stalli RAW sono ancora presenti tra le istruzioni MUL.D e ADD.D per via dell'architettura stessa del simulatore, in cui sono necessari 7 cicli per eseguire una moltiplicazione e 4 per un'addizione, in questa versione non sono presenti altre istruzioni che possono fare da cuscinetto tra i 2 gruppi d'istruzioni, perciò l'attenzione si sposta sull'ottimizzazione successiva: *Loop Unrolling*.

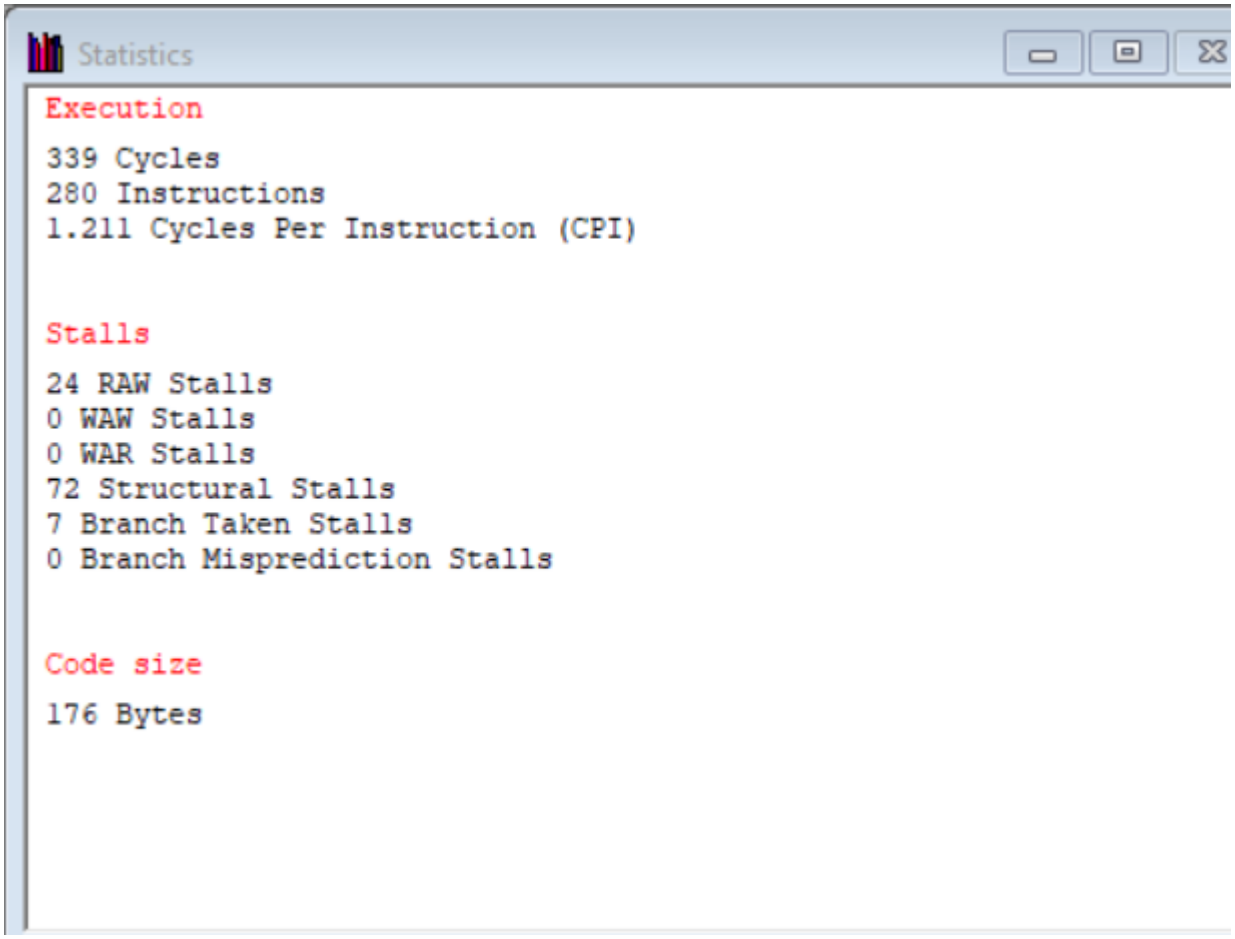
La seconda ottimizzazione è un *Loop Unrolling parziale* del primo dei 2 loop (il loop interno), permettendo di diminuire il numero di cicli d'esecuzione ed il numero di Branch Taken Stalls (creati appunto dalla presenza di salti):

```
    .data
a1:         .double 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0; elements first matrix
b1:         .double 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5, 9.5, 10.5, 11.5, 12.5, 13.5, 14.5, 15.5, 16.5; elements second matrix
c1:         .double 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0;        first row result matrix
n1:         .word   4                       ; counter for rows of the result matrix
n2:         .word   4                       ; counter for columns of the result matrix

    .text
start: LW   r1, n1(r0)          ; loading numbers of elements in matrices (loop counter)
       LW   r5, n2(r0)          ; loading numbers of elements in matrices (loop counter)
    DADDI   r2, r0, a1          ; pointer to the first element in the first matrix
    DADDI   r3, r0, b1          ; pointer to the first element in the second matrix
    DADDI   r4, r0, c1          ; pointer to the first element in the result matrix


loop1: L.D  f0, 0(r2)           ; read a1[i]
    L.D     f1, 8(r2)           ; read a1[i+1]
    L.D     f2, 16(r2)          ; read a1[i+2]
    L.D     f3, 24(r2)          ; read a1[i+3]
    L.D     f4, 0(r3)           ; read b1[i]
    L.D     f5, 32(r3)          ; read b2[i]
    L.D     f6, 64(r3)          ; read b3[i]
    L.D     f7, 96(r3)          ; read b4[i]
    L.D     f8, 8(r3)           ; read b1[i+1]
    L.D     f9, 40(r3)          ; read b2[i+1]
    L.D     f10, 72(r3)         ; read b3[i+1]
    L.D     f11, 104(r3)        ; read b4[i+1]
    MUL.D   f12, f0, f4         ; multiply the value of f0 to f4 (a1[i] * b1[i])
    MUL.D   f13, f1, f5         ; multiply the value of f1 to f5 (a1[i+1] * b2[i])
    MUL.D   f14, f2, f6         ; multiply the value of f2 to f6 (a1[i+2] * b3[i])
    MUL.D   f15, f3, f7         ; multiply the value of f3 to f7 (a1[i+3] * b4[i])
    MUL.D   f16, f0, f8         ; multiply the value of f0 to f8 (a1[i] * b1[i+1])
    MUL.D   f17, f1, f9         ; multiply the value of f1 to f9 (a1[i+1] * b2[i+1])
    MUL.D   f18, f2, f10        ; multiply the value of f2 to f10 (a1[i+2] * b3[i+1])
    MUL.D   f19, f3, f11        ; multiply the value of f3 to f11 (a1[i+3] * b4[i+1])
    ADD.D   f20, f12, f13       ; add elements for the first row/first column of the result matrix
    ADD.D   f21, f14, f15       ; add elements for the first row/first column of the result matrix
    ADD.D   f22, f16, f17       ; add elements for the first row/second column of the result matrix
    ADD.D   f23, f18, f19       ; add elements for the first row/second column of the result matrix
    DADDI   r1, r1, -1          ; decrement the loop counter
    ADD.D   f24, f20, f21       ; add elements for the first row/first column of the result matrix
    DADDI   r2, r2, 32          ; move to the next row (a2[i])
    ADD.D   f25, f22, f23       ; add elements for the first row/second column of the result matrix
    S.D     f24, 0(r4)          ; write the result in c1[i]
    DADDI   r4, r4, 8           ; move to the next the element of the same row of the second column of the result matrix
    S.D     f25, 0(r4)          ; write the result in c1[i+1]
    DADDI   r4, r4, 24          ; move to the next the element of the next row of the first column of the result matrix

    BNEZ    r1, loop1           ; jump to loop if not equal 0

    DADDI   r3, r3, 16          ; move twice the next row (b1[i+2])
    DADDI   r2, r0, a1          ; pointer to the first element in the first matrix
    DADDI   r5, r5, -2          ; decrement the loop counter
    DADDI   r1, r1, 4           ; increment the first loop counter for another column of the result matrix
    BNEZ    r5, loop1           ; jump to loop if not equal 0
end:
    HALT
```

Questa versione del codice presenta il secondo loop parzialmente srotolato, generando quindi in una sola iterazione i risultati di 2 colonne e non solo 1 come nelle precedenti versioni, ciò rende possibile ridurre il numero totale di iterazioni del secondo loop da 4 a 2 poiché sono necessarie solo 2 ripetizioni del primo loop (che genera un valore di 2 colonne ad ognuna delle sue 4 iterazioni) per generare l'intera matrice risultato.



Prestazioni del codice ottimizzato:



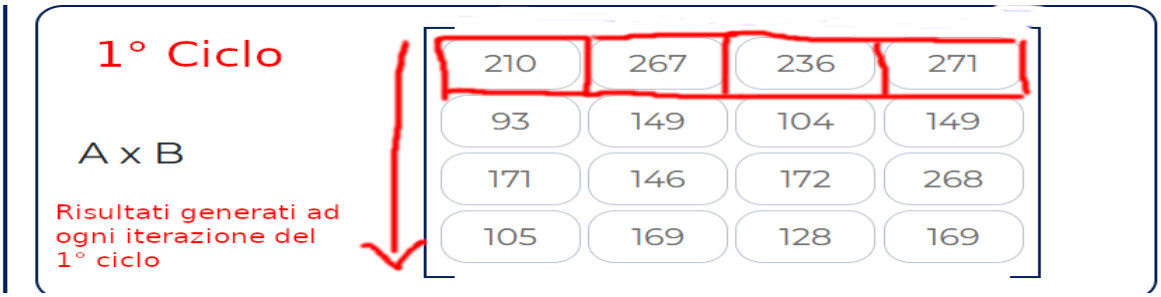
È possibile notare una netta diminuzione del numero totale dei cicli di clock d'esecuzione, un miglioramento del CPI che man mano si avvicina ad un ideale 1, gli stalli RAW sono passati da 176 a 24 ed i Branch Taken Stalls sono passati a 7.

Gli unici parametri che hanno subito un peggioramento sono il numero di stalli strutturali ed ovviamente la grandezza del codice. Ci sono ulteriori possibilità di miglioramento, per cui si passa ad un ulteriore *Loop Unrolling*.

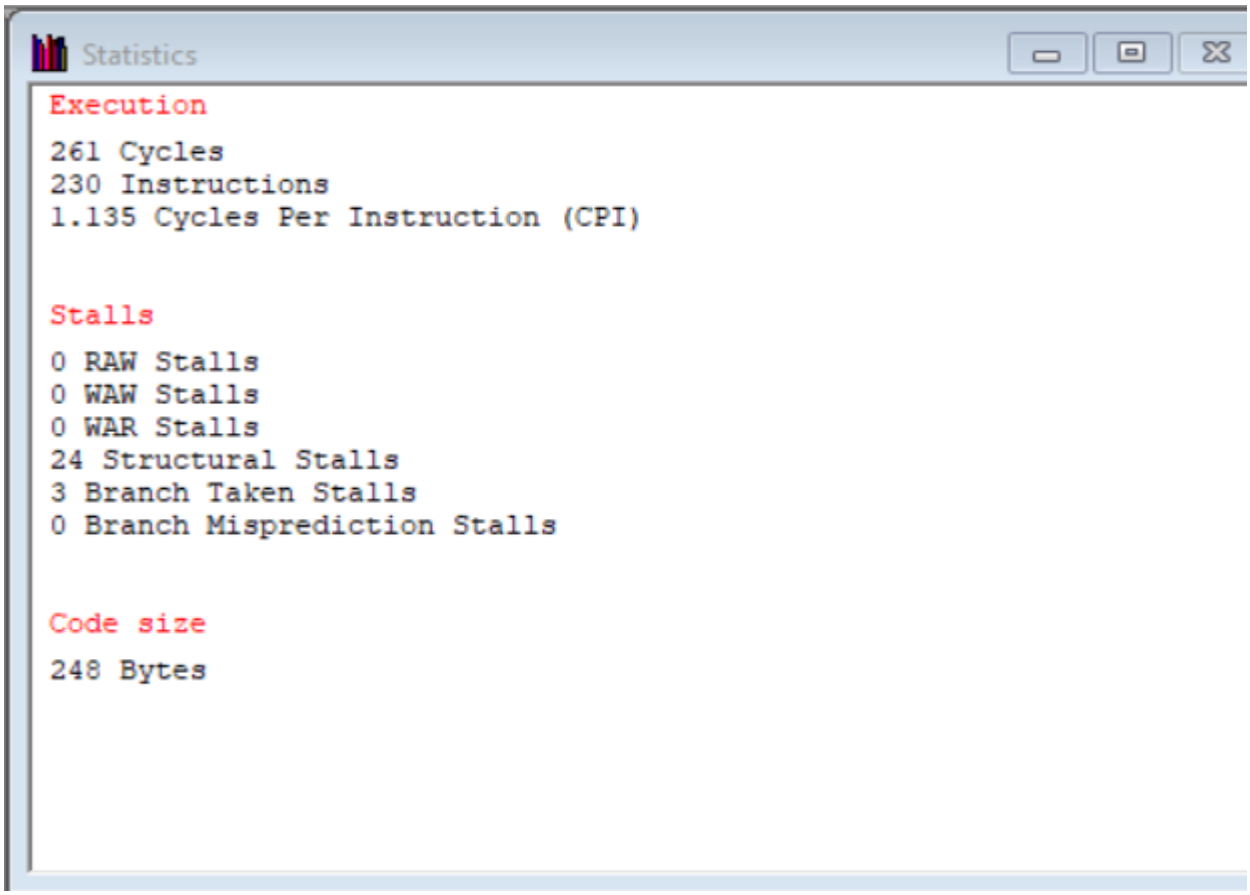
La terza ottimizzazione effettuata è un *Loop Unrolling totale* del ciclo esterno, diminuendo ulteriormente il numero di cicli di clock di esecuzione ed il numero di Branch Taken Stalls, inoltre viene effettuato Instruction Reordering:

```
    .data
a1:         .double 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0; elements first matrix
b1:         .double 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5, 9.5, 10.5, 11.5, 12.5, 13.5, 14.5, 15.5, 16.5; elements second matrix
c1:         .double 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0;        first row result matrix
n1:         .word   4                       ; counter for rows of the result matrix
n2:         .word   4                       ; counter for columns of the result matrix

    .text
start: LW   r1, n1(r0)          ; loading numbers of elements in matrices (loop counter)
       LW   r5, n2(r0)          ; loading numbers of elements in matrices (loop counter)
    DADDI   r2, r0, a1          ; pointer to the first element in the first matrix
    DADDI   r3, r0, b1          ; pointer to the first element in the second matrix
    DADDI   r4, r0, c1          ; pointer to the first element in the result matrix


loop1: L.D  f0, 0(r2)           ; read a1[i]
    L.D     f1, 8(r2)           ; read a1[i+1]
    L.D     f2, 16(r2)          ; read a1[i+2]
    L.D     f3, 24(r2)          ; read a1[i+3]
    L.D     f4, 0(r3)           ; read b1[i]
    L.D     f5, 32(r3)          ; read b2[i]
    L.D     f6, 64(r3)          ; read b3[i]
    L.D     f7, 96(r3)          ; read b4[i]
    L.D     f8, 8(r3)           ; read b1[i+1]
    L.D     f9, 40(r3)          ; read b2[i+1]
    L.D     f10, 72(r3)         ; read b3[i+1]
    L.D     f11, 104(r3)        ; read b4[i+1]
    L.D     f12, 16(r3)         ; read b1[i+2]
    L.D     f13, 48(r3)         ; read b2[i+2]
    L.D     f14, 80(r3)         ; read b3[i+2]
    L.D     f15, 112(r3)        ; read b4[i+2]
    L.D     f16, 24(r3)         ; read b1[i+3]
    L.D     f17, 56(r3)         ; read b2[i+3]
    L.D     f18, 88(r3)         ; read b3[i+3]
    L.D     f19, 120(r3)        ; read b4[i+3]

    MUL.D   f20, f0, f4         ; multiply the value of f0 to f4 (a1[i] * b1[i])
    MUL.D   f21, f1, f5         ; multiply the value of f1 to f5 (a1[i+1] * b2[i])
    MUL.D   f22, f2, f6         ; multiply the value of f2 to f6 (a1[i+2] * b3[i])
    MUL.D   f23, f3, f7         ; multiply the value of f3 to f7 (a1[i+3] * b4[i])

    MUL.D   f24, f0, f8         ; multiply the value of f0 to f8 (a1[i] * b1[i+1])
    MUL.D   f25, f1, f9         ; multiply the value of f1 to f9 (a1[i+1] * b2[i+1])
    MUL.D   f26, f2, f10        ; multiply the value of f2 to f10 (a1[i+2] * b3[i+1])
    MUL.D   f27, f3, f11        ; multiply the value of f3 to f11 (a1[i+3] * b4[i+1])

    MUL.D   f28, f0, f12        ; multiply the value of f0 to f12 (a1[i] * b1[i+2])
    MUL.D   f29, f1, f13        ; multiply the value of f1 to f13 (a1[i+1] * b2[i+2])
    MUL.D   f30, f2, f14        ; multiply the value of f2 to f14 (a1[i+2] * b3[i+2])
    MUL.D   f31, f3, f15        ; multiply the value of f3 to f15 (a1[i+3] * b4[i+2])

    MUL.D   f4, f0, f16         ; multiply the value of f0 to f16 (a1[i] * b1[i+3])
    MUL.D   f5, f1, f17         ; multiply the value of f1 to f17 (a1[i+1] * b2[i+3])
    MUL.D   f6, f2, f18         ; multiply the value of f2 to f18 (a1[i+2] * b3[i+3])
    MUL.D   f7, f3, f19         ; multiply the value of f3 to f19 (a1[i+3] * b4[i+3])

    ADD.D   f20, f20, f21       ; add elements for the first row/first column of the result matrix
    ADD.D   f21, f22, f23       ; add elements for the first row/first column of the result matrix

    ADD.D   f22, f24, f25       ; add elements for the first row/second column of the result matrix
    ADD.D   f23, f26, f27       ; add elements for the first row/second column of the result matrix

    ADD.D   f24, f28, f29       ; add elements for the first row/third  column of the result matrix
    ADD.D   f25, f30, f31       ; add elements for the first row/third column of the result matrix

    ADD.D   f26, f4, f5         ; add elements for the first row/fourth column of the result matrix
    ADD.D   f27, f6, f7         ; add elements for the first row/fourth column of the result matrix

    ADD.D   f28, f20, f21       ; add elements for the first row/first column of the result matrix
    ADD.D   f29, f22, f23       ; add elements for the first row/second column of the result matrix
    ADD.D   f30, f24, f25       ; add elements for the first row/third column of the result matrix
    ADD.D   f31, f26, f27       ; add elements for the first row/fourth column of the result matrix

    DADDI   r2, r2, 32          ; move to the next row (a2[i])
    S.D     f28, 0(r4)          ; write the result in c1[i]
    S.D     f29, 8(r4)          ; write the result in c1[i+1]
    S.D     f30, 16(r4)         ; write the result in c1[i+2]
    S.D     f31, 24(r4)         ; write the result in c1[i+3]

    DADDI   r5, r5, -1          ; decrement the loop counter
    DADDI   r4, r4, 32          ; move to the next the row of the result matrix
    BNEZ    r5, loop1           ; jump to loop if not equal 0
end:
    HALT
```

Questa versione presenta un loop completamente srotolato, generando con ogni iterazione un valore per ognuna delle 4 colonne e non solo 2 come nella precedente versione. Ora è necessario un solo loop che faccia in modo che l'iterazione termini dopo che tutti i valori delle 4 colonne siano stati generati, rendendo necessarie solo 4 iterazioni totali.



Prestazioni del codice ottimizzato:



Rispetto alla precedente versione, non ci sono stalli RAW grazie alla presenza di un numero sufficiente di istruzioni, infatti è possibile ordinarle in modo che nessuna causi mai uno stallo RAW. Inoltre il numero di cicli totali è diminuito a 261, il numero di stalli strutturali è diminuito a 24 e quello di Branch Taken a 3 poiché in questa versione vengono effettuati solo 3 salti, il CPI è nuovamente migliorato, l'unico aspetto negativo anche in questo caso è la grandezza del codice. Un ulteriore *Loop Unrolling* può conferire ulteriore ottimizzazione.

La quarta ottimizzazione effettuata è un *Loop Unrolling* totale del codice, con necessariamente *Register Renaming* e seguendo lo stesso principio di *Instruction Reordering* della versione precedente:

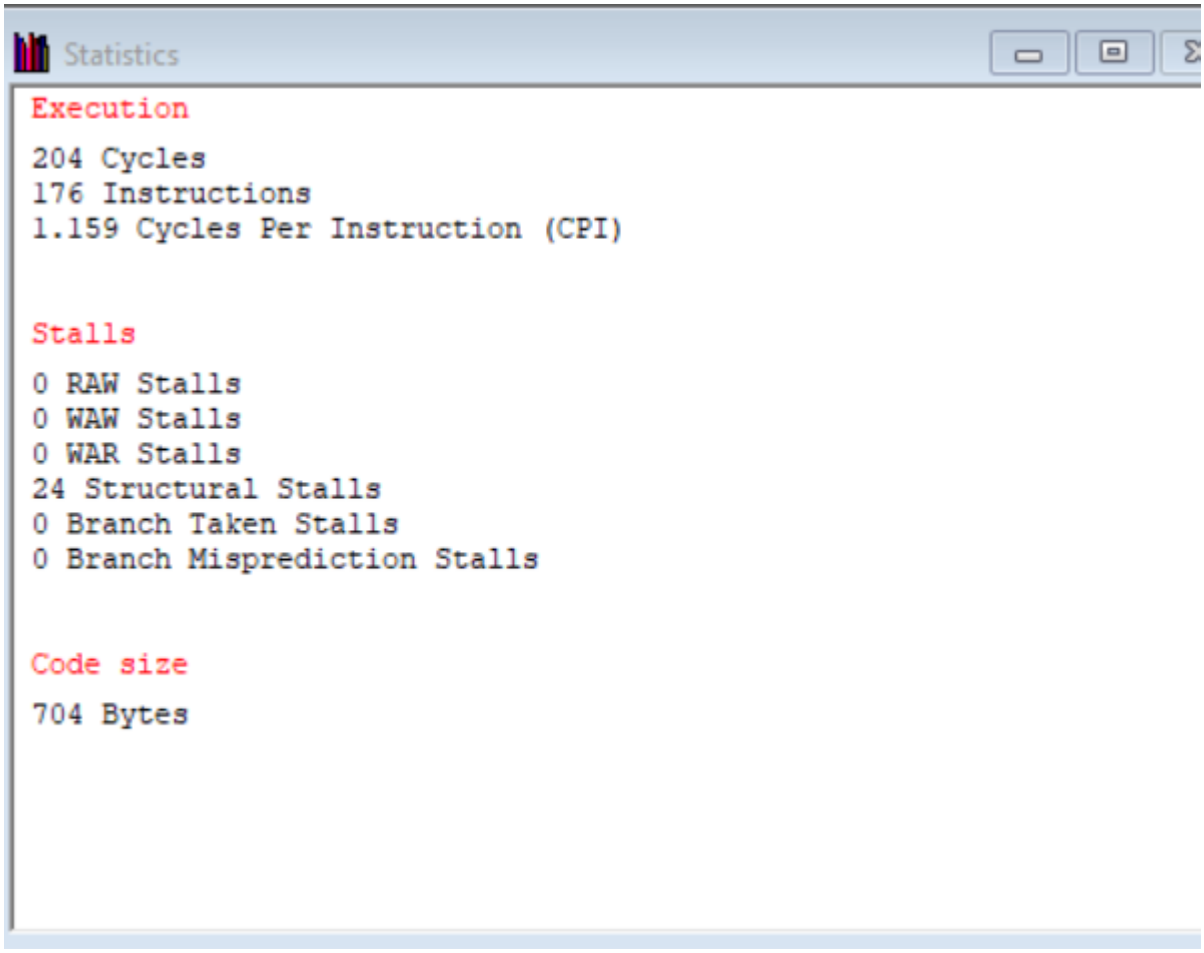
```
    .data
a1:         .double 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0; elements first matrix
b1:         .double 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5, 9.5, 10.5, 11.5, 12.5, 13.5, 14.5, 15.5, 16.5; elements second matrix
c1:         .double 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0;        first row result matrix

    .text
start:
    DADDI   r2, r0, a1          ; pointer to the first element in the first matrix
    DADDI   r3, r0, b1          ; pointer to the first element in the second matrix
    DADDI   r4, r0, c1          ; pointer to the first element in the result matrix


loop1: L.D  f0, 0(r2)           ; read a1[i]
    L.D     f1, 8(r2)           ; read a1[i+1]
    L.D     f2, 16(r2)          ; read a1[i+2]
    L.D     f3, 24(r2)          ; read a1[i+3]
    L.D     f4, 0(r3)           ; read b1[i]
    L.D     f5, 32(r3)          ; read b2[i]
    L.D     f6, 64(r3)          ; read b3[i]
    L.D     f7, 96(r3)          ; read b4[i]
    L.D     f8, 8(r3)           ; read b1[i+1]
    L.D     f9, 40(r3)          ; read b2[i+1]
    L.D     f10, 72(r3)         ; read b3[i+1]
    L.D     f11, 104(r3)        ; read b4[i+1]
    L.D     f12, 16(r3)         ; read b1[i+2]
    L.D     f13, 48(r3)         ; read b2[i+2]
    L.D     f14, 80(r3)         ; read b3[i+2]
    L.D     f15, 112(r3)        ; read b4[i+2]
    L.D     f16, 24(r3)         ; read b1[i+3]
    L.D     f17, 56(r3)         ; read b2[i+3]
    L.D     f18, 88(r3)         ; read b3[i+3]
    L.D     f19, 120(r3)        ; read b4[i+3]

    MUL.D   f20, f0, f4         ; multiply the value of f0 to f4 (a1[i] * b1[i])
    MUL.D   f21, f1, f5         ; multiply the value of f1 to f5 (a1[i+1] * b2[i])
    MUL.D   f22, f2, f6         ; multiply the value of f2 to f6 (a1[i+2] * b3[i])
    MUL.D   f23, f3, f7         ; multiply the value of f3 to f7 (a1[i+3] * b4[i])

    MUL.D   f24, f0, f8         ; multiply the value of f0 to f8 (a1[i] * b1[i+1])
    MUL.D   f25, f1, f9         ; multiply the value of f1 to f9 (a1[i+1] * b2[i+1])
    MUL.D   f26, f2, f10        ; multiply the value of f2 to f10 (a1[i+2] * b3[i+1])
    MUL.D   f27, f3, f11        ; multiply the value of f3 to f11 (a1[i+3] * b4[i+1])

    MUL.D   f28, f0, f12        ; multiply the value of f0 to f12 (a1[i] * b1[i+2])
    MUL.D   f29, f1, f13        ; multiply the value of f1 to f13 (a1[i+1] * b2[i+2])
    MUL.D   f30, f2, f14        ; multiply the value of f2 to f14 (a1[i+2] * b3[i+2])
    MUL.D   f31, f3, f15        ; multiply the value of f3 to f15 (a1[i+3] * b4[i+2])

    MUL.D   f4, f0, f16         ; multiply the value of f0 to f16 (a1[i] * b1[i+3])
    MUL.D   f5, f1, f17         ; multiply the value of f1 to f17 (a1[i+1] * b2[i+3])
    MUL.D   f6, f2, f18         ; multiply the value of f2 to f18 (a1[i+2] * b3[i+3])
    MUL.D   f7, f3, f19         ; multiply the value of f3 to f19 (a1[i+3] * b4[i+3])

    ADD.D   f20, f20, f21       ; add elements for the first row/first column of the result matrix
    ADD.D   f21, f22, f23       ; add elements for the first row/first column of the result matrix

    ADD.D   f22, f24, f25       ; add elements for the first row/second column of the result matrix
    ADD.D   f23, f26, f27       ; add elements for the first row/second column of the result matrix

    ADD.D   f24, f28, f29       ; add elements for the first row/third  column of the result matrix
    ADD.D   f25, f30, f31       ; add elements for the first row/third column of the result matrix

    ADD.D   f26, f4, f5         ; add elements for the first row/fourth column of the result matrix
    ADD.D   f27, f6, f7         ; add elements for the first row/fourth column of the result matrix

    ADD.D   f28, f20, f21       ; add elements for the first row/first column of the result matrix
    ADD.D   f29, f22, f23       ; add elements for the first row/second column of the result matrix
    ADD.D   f30, f24, f25       ; add elements for the first row/third column of the result matrix
    ADD.D   f31, f26, f27       ; add elements for the first row/fourth column of the result matrix

    S.D     f28, 0(r4)          ; write the result in c1[i]
    S.D     f29, 8(r4)          ; write the result in c1[i+1]
    S.D     f30, 16(r4)         ; write the result in c1[i+2]
    S.D     f31, 24(r4)         ; write the result in c1[i+3]

loop2: L.D  f0, 32(r2)          ; read a2[i]
    L.D     f1, 40(r2)          ; read a2[i+1]
    L.D     f2, 48(r2)          ; read a2[i+2]
    L.D     f3, 56(r2)          ; read a2[i+3]
    L.D     f4, 0(r3)           ; read b1[i]
    L.D     f5, 32(r3)          ; read b2[i]
    L.D     f6, 64(r3)          ; read b3[i]
    L.D     f7, 96(r3)          ; read b4[i]

    MUL.D   f20, f0, f4         ; multiply the value of f0 to f4 (a2[i] * b1[i])
    MUL.D   f21, f1, f5         ; multiply the value of f1 to f5 (a2[i+1] * b2[i])
```

```
87      MUL.D   f22, f2, f6         ; multiply the value of f2 to f6 (a2[i+2] * b3[i])
88      MUL.D   f23, f3, f7         ; multiply the value of f3 to f7 (a2[i+3] * b4[i])
89
90      MUL.D   f24, f0, f8         ; multiply the value of f0 to f8 (a2[i] * b1[i+1])
91      MUL.D   f25, f1, f9         ; multiply the value of f1 to f9 (a2[i+1] * b2[i+1])
92      MUL.D   f26, f2, f10        ; multiply the value of f2 to f10 (a2[i+2] * b3[i+1])
93      MUL.D   f27, f3, f11        ; multiply the value of f3 to f11 (a2[i+3] * b4[i+1])
94
95      MUL.D   f28, f0, f12        ; multiply the value of f0 to f12 (a2[i] * b1[i+2])
96      MUL.D   f29, f1, f13        ; multiply the value of f1 to f13 (a2[i+1] * b2[i+2])
97      MUL.D   f30, f2, f14        ; multiply the value of f2 to f14 (a2[i+2] * b3[i+2])
98      MUL.D   f31, f3, f15        ; multiply the value of f3 to f15 (a2[i+3] * b4[i+2])
99
100     MUL.D   f4, f0, f16         ; multiply the value of f0 to f16 (a2[i] * b1[i+3])
101     MUL.D   f5, f1, f17         ; multiply the value of f1 to f17 (a2[i+1] * b2[i+3])
102     MUL.D   f6, f2, f18         ; multiply the value of f2 to f18 (a2[i+2] * b3[i+3])
103     MUL.D   f7, f3, f19         ; multiply the value of f3 to f19 (a2[i+3] * b4[i+3])
104
105     ADD.D   f20, f20, f21       ; add elements for the second row/first column of the result matrix
106     ADD.D   f21, f22, f23       ; add elements for the second row/first column of the result matrix
107
108     ADD.D   f22, f24, f25       ; add elements for the second row/second column of the result matrix
109     ADD.D   f23, f26, f27       ; add elements for the second row/second column of the result matrix
110
111     ADD.D   f24, f28, f29       ; add elements for the second row/third column of the result matrix
112     ADD.D   f25, f30, f31       ; add elements for the second row/third column of the result matrix
113
114     ADD.D   f26, f4, f5         ; add elements for the second row/fourth column of the result matrix
115     ADD.D   f27, f6, f7         ; add elements for the second row/fourth column of the result matrix
116
117     ADD.D   f28, f20, f21       ; add elements for the second row/first column of the result matrix
118     ADD.D   f29, f22, f23       ; add elements for the second row/second column of the result matrix
119     ADD.D   f30, f24, f25       ; add elements for the second row/third column of the result matrix
120     ADD.D   f31, f26, f27       ; add elements for the second row/fourth column of the result matrix
121
122     S.D     f28, 32(r4)         ; write the result in c2[i]
123     S.D     f29, 40(r4)         ; write the result in c2[i+1]
124     S.D     f30, 48(r4)         ; write the result in c2[i+2]
125     S.D     f31, 56(r4)         ; write the result in c2[i+3]
126
127 loop3: L.D  f0, 64(r2)          ; read a3[i]
128     L.D     f1, 72(r2)          ; read a3[i+1]
129     L.D     f2, 80(r2)          ; read a3[i+2]
130     L.D     f3, 88(r2)          ; read a3[i+3]
131     L.D     f4, 0(r3)           ; read b1[i]
132     L.D     f5, 32(r3)          ; read b2[i]
133     L.D     f6, 64(r3)          ; read b3[i]
134     L.D     f7, 96(r3)          ; read b4[i]
135
136     MUL.D   f20, f0, f4         ; multiply the value of f0 to f4 (a3[i] * b1[i])
137     MUL.D   f21, f1, f5         ; multiply the value of f1 to f5 (a3[i+1] * b2[i])
138     MUL.D   f22, f2, f6         ; multiply the value of f2 to f6 (a3[i+2] * b3[i])
139     MUL.D   f23, f3, f7         ; multiply the value of f3 to f7 (a3[i+3] * b4[i])
140
141     MUL.D   f24, f0, f8         ; multiply the value of f0 to f8 (a3[i] * b1[i+1])
142     MUL.D   f25, f1, f9         ; multiply the value of f1 to f9 (a3[i+1] * b2[i+1])
143     MUL.D   f26, f2, f10        ; multiply the value of f2 to f10 (a3[i+2] * b3[i+1])
144     MUL.D   f27, f3, f11        ; multiply the value of f3 to f11 (a3[i+3] * b4[i+1])
145
146     MUL.D   f28, f0, f12        ; multiply the value of f0 to f12 (a3[i] * b1[i+2])
147     MUL.D   f29, f1, f13        ; multiply the value of f1 to f13 (a3[i+1] * b2[i+2])
148     MUL.D   f30, f2, f14        ; multiply the value of f2 to f14 (a3[i+2] * b3[i+2])
149     MUL.D   f31, f3, f15        ; multiply the value of f3 to f15 (a3[i+3] * b4[i+2])
150
151     MUL.D   f4, f0, f16         ; multiply the value of f0 to f16 (a3[i] * b1[i+3])
152     MUL.D   f5, f1, f17         ; multiply the value of f1 to f17 (a3[i+1] * b2[i+3])
153     MUL.D   f6, f2, f18         ; multiply the value of f2 to f18 (a3[i+2] * b3[i+3])
154     MUL.D   f7, f3, f19         ; multiply the value of f3 to f19 (a3[i+3] * b4[i+3])
155
156     ADD.D   f20, f20, f21       ; add elements for the third row/first column of the result matrix
157     ADD.D   f21, f22, f23       ; add elements for the third row/first column of the result matrix
158
159     ADD.D   f22, f24, f25       ; add elements for the third row/second column of the result matrix
160     ADD.D   f23, f26, f27       ; add elements for the third row/second column of the result matrix
161
162     ADD.D   f24, f28, f29       ; add elements for the third row/third  column of the result matrix
163     ADD.D   f25, f30, f31       ; add elements for the third row/third column of the result matrix
164
165     ADD.D   f26, f4, f5         ; add elements for the third row/fourth column of the result matrix
166     ADD.D   f27, f6, f7         ; add elements for the third row/fourth column of the result matrix
167
168     ADD.D   f28, f20, f21       ; add elements for the third row/first column of the result matrix
169     ADD.D   f29, f22, f23       ; add elements for the third row/second column of the result matrix
170     ADD.D   f30, f24, f25       ; add elements for the third row/third column of the result matrix
```

```
171         ADD.D   f31, f26, f27       ; add elements for the third row/fourth column of the result matrix
172
173         S.D     f28, 64(r4)         ; write the result in c3[i]
174         S.D     f29, 72(r4)         ; write the result in c3[i+1]
175         S.D     f30, 80(r4)         ; write the result in c3[i+2]
176         S.D     f31, 88(r4)         ; write the result in c3[i+3]
177
178     loop4: L.D  f0, 96(r2)          ; read a4[i]
179         L.D     f1, 104(r2)         ; read a4[i+1]
180         L.D     f2, 112(r2)         ; read a4[i+2]
181         L.D     f3, 120(r2)         ; read a4[i+3]
182         L.D     f4, 0(r3)           ; read b1[i]
183         L.D     f5, 32(r3)          ; read b2[i]
184         L.D     f6, 64(r3)          ; read b3[i]
185         L.D     f7, 96(r3)          ; read b4[i]
186
187         MUL.D   f20, f0, f4         ; multiply the value of f0 to f4 (a4[i] * b1[i])
188         MUL.D   f21, f1, f5         ; multiply the value of f1 to f5 (a4[i+1] * b2[i])
189         MUL.D   f22, f2, f6         ; multiply the value of f2 to f6 (a4[i+2] * b3[i])
190         MUL.D   f23, f3, f7         ; multiply the value of f3 to f7 (a4[i+3] * b4[i])
191
192         MUL.D   f24, f0, f8         ; multiply the value of f0 to f8 (a4[i] * b1[i+1])
193         MUL.D   f25, f1, f9         ; multiply the value of f1 to f9 (a4[i+1] * b2[i+1])
194         MUL.D   f26, f2, f10        ; multiply the value of f2 to f10 (a4[i+2] * b3[i+1])
195         MUL.D   f27, f3, f11        ; multiply the value of f3 to f11 (a4[i+3] * b4[i+1])
196
197         MUL.D   f28, f0, f12        ; multiply the value of f0 to f12 (a4[i] * b1[i+2])
198         MUL.D   f29, f1, f13        ; multiply the value of f1 to f13 (a4[i+1] * b2[i+2])
199         MUL.D   f30, f2, f14        ; multiply the value of f2 to f14 (a4[i+2] * b3[i+2])
200         MUL.D   f31, f3, f15        ; multiply the value of f3 to f15 (a4[i+3] * b4[i+2])
201
202         MUL.D   f4, f0, f16         ; multiply the value of f0 to f16 (a4[i] * b1[i+3])
203         MUL.D   f5, f1, f17         ; multiply the value of f1 to f17 (a4[i+1] * b2[i+3])
204         MUL.D   f6, f2, f18         ; multiply the value of f2 to f18 (a4[i+2] * b3[i+3])
205         MUL.D   f7, f3, f19         ; multiply the value of f3 to f19_(a4[i+3] * b4[i+3])
206
207         ADD.D   f20, f20, f21       ; add elements for the fourth row/first column of the result matrix
208         ADD.D   f21, f22, f23       ; add elements for the fourth row/first column of the result matrix
209
210         ADD.D   f22, f24, f25       ; add elements for the fourth row/second column of the result matrix
211         ADD.D   f23, f26, f27       ; add elements for the fourth row/second column of the result matrix
212
213         ADD.D   f24, f28, f29       ; add elements for the fourth row/third column of the result matrix
214         ADD.D   f25, f30, f31       ; add elements for the fourth row/third column of the result matrix
215
216         ADD.D   f26, f4, f5         ; add elements for the fourth row/fourth column of the result matrix
217         ADD.D   f27, f6, f7         ; add elements for the fourth row/fourth column of the result matrix
218
219         ADD.D   f28, f20, f21       ; add elements for the fourth row/first column of the result matrix
220         ADD.D   f29, f22, f23       ; add elements for the fourth row/second column of the result matrix
221         ADD.D   f30, f24, f25       ; add elements for the fourth row/third column of the result matrix
222         ADD.D   f31, f26, f27       ; add elements for the fourth row/fourth column of the result matrix
223
224         S.D     f28, 96(r4)         ; write the result in c4[i]
225         S.D     f29, 104(r4)        ; write the result in c4[i+1]
226         S.D     f30, 112(r4)        ; write the result in c4[i+2]
227         S.D     f31, 120(r4)        ; write the result in c4[i+3]
228     end:
229         HALT
```

Questa versione non presenta loop poiché sono stati entrambi totalmente srotolati, grazie al continuo Register Renaming è possibile eseguire tutte le operazioni con i 32 registri interni disponibili, vengono calcolati tutti i risultati della prima riga della matrice risultato, per poi passare alle righe successive fino al completamento.

Prestazioni del codice ottimizzato:

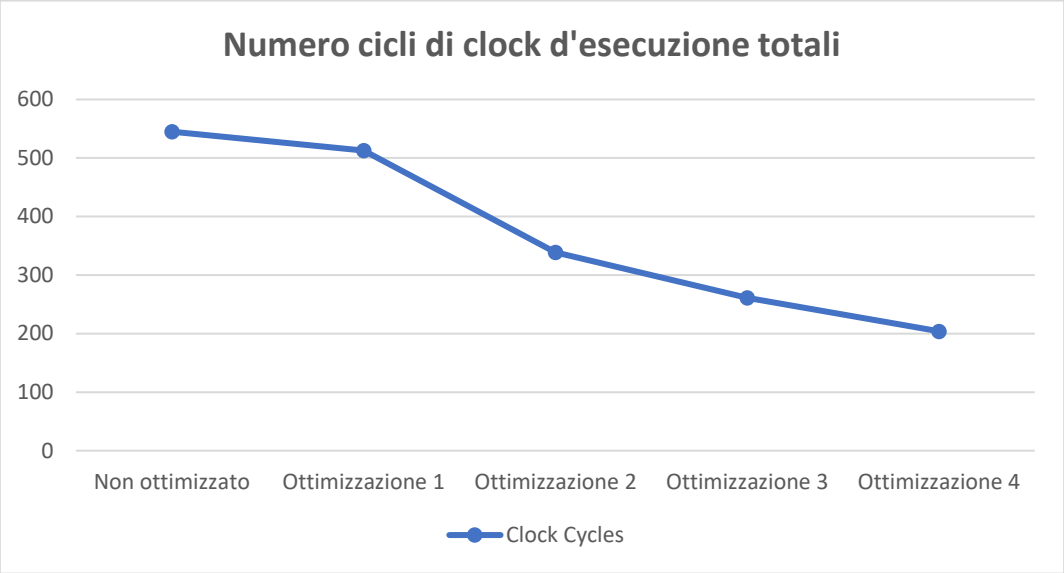


Come previsto il numero di cicli totali è notevolmente diminuito, il CPI è leggermente aumentato, non sono presenti stalli al di fuori di stalli strutturali, il punto dolente è la grandezza del codice, ma considerando le specifiche di progetto decise e tenute in considerazione, l'obiettivo è stato raggiunto ottenendo le migliori prestazioni possibili.

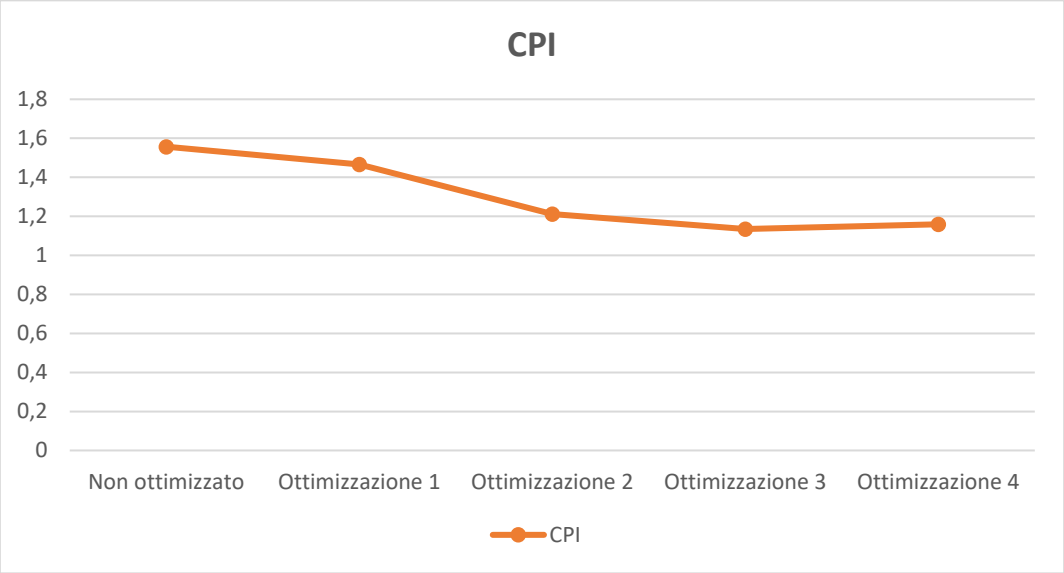
Note finali

Come si può notare quindi, i vari tipi di ottimizzazione hanno portato a prestazioni notevolmente migliori in tutti i campi:

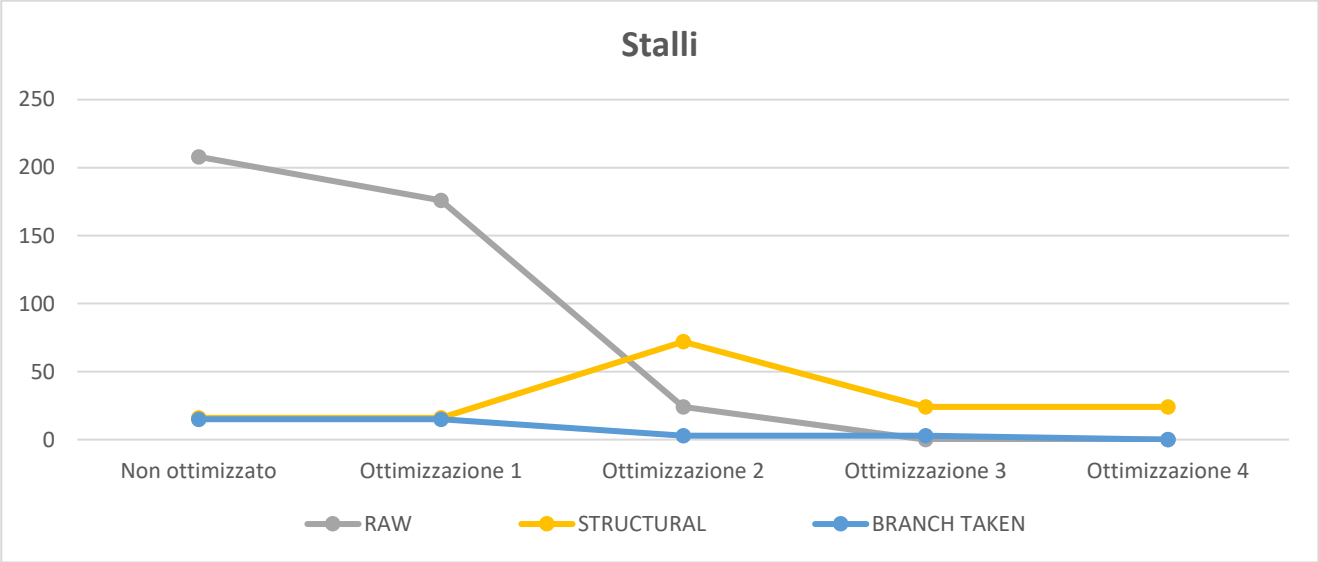
N.O.	545
OT.1	513
OT.2	339
OT.3	261
OT.4	204



N.O.	1,557
OT.1	1,466
OT.2	1,211
OT.3	1,135
OT.4	1,159



	RAW	STRUCT	BRANCH TAKEN
N.O.	208	16	15
OT.1	176	16	15
OT.2	24	72	3
OT.3	0	24	3
OT.4	0	24	0



N.O.	128
OT.1	128
OT.2	176
OT.3	248
OT.4	704