1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

DISPEA
DIPARTIMENTO DI
SCIENZE PURE E
APPLICATE

Relazione Progetto d'esame
Architettura degli Elaboratori

Sessione Estiva 2023/2024

Ottimizzazione prodotto tra matrici

AUTORI

Mattia Gasperoni
matricola: 329235

Andrea Marchionni
matricola: 326970

Claudio Valentin Gaspar
matricola: 328833

Indice

1 Specifica

1.1 Scopo del progetto

Ci siamo posti l'obiettivo di ottimizzare il prodotto tra due matrici 2×2. Inizialmente, abbiamo scritto il codice in un linguaggio di alto livello (C) e poi successivamente in un linguaggio di basso livello (*Assembly*). Dopodiché, abbiamo ottimizzato il programma con l'aiuto del software WinMIPS64, cercando di renderlo il più efficiente possibile sulla base di diversi parametri, quali il CPI (Clock Cycles Per Instruction), la dimensione del codice e il numero di cicli di clock.

1.2 Specifica Funzionale

Il prodotto tra matrici quadrate è un'operazione fondamentale in algebra lineare e trova applicazione in numerosi campi, questa operazione prende come input due matrici quadrate e produce un'altra matrice quadrata come output. Una matrice è considerata quadrata quando il numero delle righe è uguale al numero delle colonne.

Se A e B sono due matrici quadrate di dimensione $n \times n$, il loro prodotto, indicato come $C = A \times B$, è una matrice quadrata della stessa dimensione. L'elemento c_{ij} della matrice risultante C è calcolato come la somma dei prodotti degli elementi corrispondenti delle righe di A e delle colonne di B:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Graficamente, il prodotto tra due matrici 2x2 può essere rappresentato come segue:

$$\begin{pmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{pmatrix} \times \begin{pmatrix} B_{0,0} & B_{0,1} \\ B_{1,0} & B_{1,1} \end{pmatrix} = \begin{pmatrix} C_{0,0} & C_{0,1} \\ C_{1,0} & C_{1,1} \end{pmatrix}$$

I calcoli specifici per ottenere gli elementi della matrice C sono:

$$\begin{aligned} C_{0,0} &= A_{0,0} \cdot B_{0,0} + A_{0,1} \cdot B_{1,0} \\ C_{0,1} &= A_{0,0} \cdot B_{0,1} + A_{0,1} \cdot B_{1,1} \\ C_{1,0} &= A_{1,0} \cdot B_{0,0} + A_{1,1} \cdot B_{1,0} \\ C_{1,1} &= A_{1,0} \cdot B_{0,1} + A_{1,1} \cdot B_{1,1} \end{aligned}$$

2 Progettazione

2.1 Dati di Input e di Output

L'algoritmo implementato per il prodotto tra matrici quadrate viene testato utilizzando dati specifici. Gli input sono mantenuti costanti per consentire un confronto preciso dell'efficienza delle diverse ottimizzazioni. I dati di input e output utilizzati sono i seguenti:

$$\begin{pmatrix} 4.3 & 2.1 \\ 3.3 & 5.2 \end{pmatrix} \times \begin{pmatrix} 2.3 & 1.1 \\ 4.3 & 6.2 \end{pmatrix} = \begin{pmatrix} 18.92 & 17.75 \\ 29.95 & 35.87 \end{pmatrix}$$

Di seguito sono dettagliati i calcoli per ogni elemento della matrice risultante:

$$\begin{aligned} C_{0,0} &= 4.3 \cdot 2.3 + 2.1 \cdot 4.3 = 9.89 + 9.03 = 18.92 \\ C_{0,1} &= 4.3 \cdot 1.1 + 2.1 \cdot 6.2 = 4.73 + 13.02 = 17.75 \\ C_{1,0} &= 3.3 \cdot 2.3 + 5.2 \cdot 4.3 = 7.59 + 22.36 = 29.95 \\ C_{1,1} &= 3.3 \cdot 1.1 + 5.2 \cdot 6.2 = 3.63 + 32.24 = 35.87 \end{aligned}$$

Gli input sono quindi costituiti da due matrici 2×2 di numeri reali, e l'output è una matrice 2×2 risultante dal loro prodotto. Non sono previste limitazioni specifiche sui tipi di dati oltre a quelli indicati, ma è importante notare che i risultati sono determinati esclusivamente da questi valori fissi.

2.2 Tipo di Dati

2.2.1 C

In ANSI C, la matrice viene rappresentata come un array bidimensionale. Nel nostro caso, avendo utilizzato operandi reali, l'array sarà di tipo *double*. La definizione di una matrice $m \times n$ in C è:

```
double matrice[m][n];
```

Dove m e n sono rispettivamente il numero di righe e colonne della matrice.

2.2.2 Assembly

Nel contesto Assembly, la gestione delle matrici e dei dati è più complessa rispetto ai linguaggi di alto livello, a causa dell'assenza di strutture dati integrate. In particolare, si utilizzano segmenti di dati per memorizzare le matrici e le informazioni associate, come le dimensioni e i risultati parziali. Nel nostro caso, gestiamo tre array principali: le due matrici operande e la matrice risultato. Inoltre, vengono memorizzate le dimensioni delle matrici e variabili per i risultati parziali.

```
.data
; MATRICE A (2x2)
a:      .double 4.3, 2.1, 3.3, 5.2
; MATRICE B (2x2)
b:      .double 2.3, 1.1, 4.3, 6.2
; MATRICE C (2x2) (risultato)
c:      .double 0, 0, 0, 0
;COLONNE A
n:      .word   2
;RIGHE A
o:      .word   2
;COLONNE B
m:      .word   2
;PARZIALE
p:      .double 0
```

2.3 Ottimizzazione e Conflitti

Per effettuare l'ottimizzazione del codice abbiamo utilizzato il software **WinMIPS64** grazie ad esso abbiamo potuto confrontare le diverse versioni del codice Assembly visionando le seguenti statistiche:

Esecuzione	
Cicli	
Istruzioni	
CPI	
Stalli	
Stalli RAW	
Stalli WAW	
Stalli WAR	
Stalli Strutturali	
Branch Taken Stalls	
Branch Misprediction	
Code Size	
Bytes	

In Assembly per migliorare l'ottimizzazione del nostro codice abbiamo implementato le seguenti tecniche di ottimizzazione:

- ***Loop Unrolling***: Il ciclo viene "srotolato" per ridurre il numero di salti e incrementi di contatore, aumentando il numero di istruzioni eseguite in una singola iterazione.
- ***Instruction Reordering***: Questa tecnica cerca di ottimizzare l'uso delle risorse del processore, riorganizzando l'ordine delle istruzioni in un programma senza alterare il risultato finale.
- ***Register Renaming***: Consiste nel rinominare i registri del processore per gestire i conflitti tra le istruzioni che utilizzano gli stessi registri fisici.

Dal punto di vista prestazionale la cosa più importante da ottimizzare è sicuramente il corpo del loop poiché gli stalli presenti al suo interno saranno moltiplicati per il numero di iterazioni effettuate.

2.3.1 Stalli

I tipi di stalli che possiamo trovare tra le varie istruzioni possono essere di tipo:

- **RAW** (Read After Write): avviene in caso un'istruzione richieda un dato non ancora calcolato.
- **WAR** (Write After Read): si verifica nel momento in cui un'istruzione legge un dato che si trova in una locazione in cui un'istruzione successiva sta per salvare un altro dato.
- **WAW** (Write After Write): Entrambe le istruzioni scrivono nello stesso registro o locazione di memoria. È cruciale mantenere l'ordine delle scritture per evitare conflitti.
- **Strutturale** : è il tentativo di usare la stessa risorsa hardware da parte di diverse istruzioni in modi diversi nello stesso ciclo di clock.
- **Branch Taken** : avviene quando si esegue un branch (vai a un'istruzione diversa dalla successiva), l'istruzione nell'indirizzo di destinazione non è ancora stata decodificata.

3 Implementazione Algoritmo

3.1 Codice in C

Il prodotto di due matrici in C viene calcolato utilizzando tre cicli *for* annidati. Questo approccio è adatto per la moltiplicazione di matrici di dimensioni fisse, poiché gli array sono dichiarati staticamente. L'algoritmo segue questi passaggi principali:

1. **Iterazione sulle righe della matrice risultante:** Il primo ciclo *for* scorre attraverso ogni riga della matrice risultante. Si occupa di selezionare la riga corrente della matrice di output, nella quale verranno accumulati i risultati parziali del prodotto.

2. **Iterazione sulle colonne della matrice risultante:** Il secondo ciclo *for* scorre attraverso ogni colonna della matrice risultante, lavora in parallelo con il primo e seleziona la colonna corrente della matrice di output nella quale verrà inserito il valore finale calcolato.

3. **Calcolo del prodotto scalare:** Il terzo ciclo *for* esegue il calcolo del prodotto scalare tra la riga corrente della prima matrice e la colonna corrente della seconda matrice. Questo ciclo è essenziale per la somma dei prodotti che compongono ogni singolo elemento della matrice risultante.

Di seguito è riportato il codice che implementa questa logica:

```c
#include <stdio.h>

int main()
{
    /* dichiarazione delle variabili locali alla funzione*/

    double a[2][2] =
                    {{4.3, 2.1},
                    {3.3, 5.2}};    /*input: Matrice A*/

    double b[2][2] =
                    {{2.3, 1.1},
                    {4.3, 6.2}};    /*input: Matrice B*/

    int i, j, k,                    /* lavoro: variabili scorrimento matrici*/
        n = 2;                      /* lavoro: dimensione delle matrici*/

    double c[2][2] =
                    {{0.0, 0.0},
                    {0.0, 0.0}};    /*output: Matrice C risultato*/

    /*calcolo del prodotto delle matrici A e B*/
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i][j] += a[i][k] * b[k][j];

    /*stampa del prodotto delle matrici*/
    printf("Matrice Prodotto:\n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
            printf("%.2f ", c[i][j]);
        printf("\n");
    }

    return 0;
}

```

3.2 Codice in Assembly

3.2.1 Versione 0

Abbiamo denominato questa versione come "0" poiché il codice è identico alla versione successiva, con la sola differenza che qui è assente il *data forwarding*.

```
1   .data
2   ; MATRICE A (2x2)
3   a:      .double 4.3, 2.1, 3.3, 5.2
4   ; MATRICE B (2x2)
5   b:      .double 2.3, 1.1, 4.3, 6.2
6   ; MATRICE C (2x2) (risultato)
7   c:      .double 0, 0, 0, 0
8   ;COLONNE A
9   n:      .word   2
10  ;RIGHE A
11  o:      .word   2
12  ;COLONNE B
13  m:      .word   2
14  ;PARZIALE
15  p:      .double 0
16
17  .text
18  start:
19          DADDI   r1, r0, a       ; punta al primo elemento della matrice A
20          DADDI   r2, r0, b       ; punta al primo elemento della matrice B
21          DADDI   r3, r0, c       ; punta al primo elemento della matrice C
22          LW      r4, n(r0)       ; carica il numero di colonne di A
23          LW      r5, o(r0)       ; carica il numero di righe di A
24          LW      r6, m(r0)       ; carica il numero di colonne di B
25          L.D     f0, p(r0)       ; inizializzazione parziale
26
27  loop:
28          L.D     f1, 0(r1)       ; leggi a[i]
29          L.D     f2, 0(r2)       ; leggi b[i]
30          MUL.D   f1, f1, f2      ; a[i] * b[i]
31          ADD.D   f0, f0, f1      ; p = p + a[i] * b[i] (risultato parziale)
32          DADDI   r1, r1, 8       ; scorri elemento A
33          DADDI   r2, r2, 16      ; scorri elemento B
34          DADDI   r4, r4, -1      ; decrementa contatore colonne A
35          BNEZ    r4, loop        ; ripeti se contatore colonne A diverso da 0
36          S.D     f0, 0(r3)       ; salva il risultato
37          MUL.D   f0, f0, f3      ; resetta il risultato parziale
38
39  puntatori:
40          DADDI   r3, r3, 8       ; scorri registro risultati di C
41          DADDI   r2, r2, -24     ; riporta puntatore B all inizio + 1
42          DADDI   r1, r1, -16     ; riporta puntatore A all inizio
43          DADDI   r4, r4, 2       ; resetta contatore colonne A
44          DADDI   r6, r6, -1      ; decrementa contatore colonne B
45          BNEZ    r6, loop        ; ripeti se contatore colonne B diverso da 0
46
47  righe:
48          DADDI r1, r1, 16        ; riporta puntatore A alla seconda riga
49          DADDI r2, r2, -16       ; riporta puntatore B all inizio
50          DADDI r6, r6, 2         ; resetta numero di colonne B
51          DADDI r5, r5, -1        ; decrementa contatore righe A
52          BNEZ  r5, loop          ; ripeti se contatore righe di A diverso da 0
53
54  end:    HALT
55
```

Le statistiche di questa versione sono:

Esecuzione	
Cicli	241
Istruzioni	114
CPI	2.114
Stalli	
Stalli RAW	108
Stalli WAW	0
Stalli WAR	0
Stalli Strutturali	8
Branch Taken Stalls	7
Branch Misprediction	0
Code Size	
Bytes	116

Table 1: Versione 0

Possiamo notare che la maggior parte degli stalli sono di tipo RAW (Read After Write). Questi stalli si verificano tra le istruzioni *MUL.D* e *ADD.D*, poiché l'istruzione *ADD.D* necessita del risultato dell'istruzione *MUL.D* come operando. In altre parole, l'istruzione *ADD.D* deve attendere che l'istruzione *MUL.D* completi la sua esecuzione e produca il risultato necessario.

Inoltre, si verificano stalli strutturali tra le istruzioni *BNEZ* e *DADDI*, che utilizzano lo stesso registro, `r4`, per operazioni diverse. Questi stalli si verificano quando due istruzioni competono per l'accesso alla stessa risorsa hardware, causando conflitti e ritardi nell'esecuzione.

Infine, gli stalli di tipo *Branch Taken* si verificano dopo l'istruzione *BNEZ*, quando il processore deve aspettare che l'istruzione di destinazione del branch venga recuperata e decodificata prima di proseguire l'esecuzione.

3.2.2 Versione 1

Come specificato nella Versione 0, il codice di questa versione rimane invariato, ma viene introdotto il *data forwarding*. Il *data forwarding* è una tecnica che riduce i conflitti dovuti alle dipendenze dei dati, utilizzando multiplexer a 3 vie per bypassare i risultati intermedi tra le fasi del pipeline. Questo meccanismo consente all'ALU di ricevere e utilizzare i risultati di calcoli precedenti già al ciclo di clock successivo, migliorando le prestazioni e riducendo i ritardi nel processore.

Le statistiche di questa versione sono:

Esecuzione	
Cicli	163
Istruzioni	114
CPI	1.430
Stalli	
Stalli RAW	86
Stalli WAW	0
Stalli WAR	12
Stalli Strutturali	12
Branch Taken Stalls	7
Branch Misprediction	0
Code Size	
Bytes	116

Table 2: Versione 1

Esecuzione	
Cicli	241
Istruzioni	114
CPI	2.114
Stalli	
Stalli RAW	108
Stalli WAW	0
Stalli WAR	0
Stalli Strutturali	8
Branch Taken Stalls	7
Branch Misprediction	0
Code Size	
Bytes	116

Table 3: Confronto Precedente (Versione 0)

Grazie all'implementazione del *data forwarding*, osserviamo una diminuzione significativa degli stalli *RAW* (Read After Write), e una riduzione del CPI e dei cicli di clock.

Tuttavia, si verifica un aumento degli stalli strutturali e la comparsa di stalli *WAR* (Write After Read). Questi ultimi si manifestano perché l'istruzione *S.D* deve prima scrivere il valore di `f0` in memoria prima che *MUL.D* possa leggere il nuovo valore di `f0` per la moltiplicazione.

3.2.3 Versione 2

```
1   .data
2   ; MATRICE A (2x2)
3   a:      .double 4.3, 2.1, 3.3, 5.2
4   ; MATRICE B (2x2)
5   b:      .double 2.3, 1.1, 4.3, 6.2
6   ; MATRICE C (2x2) (risultato)
7   c:      .double 0, 0, 0, 0
8   ;RIGHE A
9   o:      .word   2
10  ;COLONNE B
11  m:      .word   2
12  ;PARZIALE
13  p:      .double 0
14
15  .text  ;Loop Unrolling, Rimozione colonne A
16  start:
17          DADDI   r1, r0, a       ; punta al primo elemento della matrice A
18          DADDI   r2, r0, b       ; punta al primo elemento della matrice B
19          DADDI   r3, r0, c       ; punta al primo elemento della matrice C
20          LW      r5, o(r0)       ; carica il numero di righe di A
21          LW      r6, m(r0)       ; carica il numero di colonne di B
22          L.D     f0, p(r0)       ; inizializzazione parziale
23
24  loop:
25          L.D     f1, 0(r1)       ; leggi a[i]
26          L.D     f2, 0(r2)       ; leggi b[i]
27          MUL.D   f1, f1, f2      ; a[i] * b[i]
28          ADD.D   f0, f0, f1      ; p = p + a[i] * b[i] (risultato parziale)
29
30          L.D     f1, 8(r1)       ; leggi a[i]
31          L.D     f2, 16(r2)      ; leggi b[i]
32          MUL.D   f1, f1, f2      ; a[i] * b[i]
33          ADD.D   f0, f0, f1      ; p = p + a[i] * b[i] (risultato parziale)
34
35          S.D     f0, 0(r3)       ; salva il risultato
36          MUL.D   f0, f0, f3      ; resetta il risultato parziale
37
38  puntatori:
39          DADDI   r3, r3, 8       ; scorri registro risultati di C
40          DADDI   r2, r2, 8       ; punta alla nuova collonna di B
41          DADDI   r6, r6, -1      ; decrementa contatore colonne B
42          BNEZ    r6, loop        ; ripeti se contatore colonne B diverso da 0
43
44  righe:
45          DADDI r1, r1, 16        ; riporta puntatore A alla seconda riga
46          DADDI r2, r2, -16       ; riporta puntatore B all inizio
47          DADDI r6, r6, 2         ; resetta numero di colonne B
48          DADDI r5, r5, -1        ; decrementa contatore righe A
49          BNEZ  r5, loop          ; ripeti se contatore righe di A diverso da 0
50
51  end:    HALT
```

Le statistiche di questa versione sono:

Esecuzione	
Cicli	156
Istruzioni	73
CPI	2.137
Stalli	
Stalli RAW	102
Stalli WAW	0
Stalli WAR	36
Stalli Strutturali	6
Branch Taken Stalls	3
Branch Misprediction	0
Code Size	
Bytes	104

Table 4: Versione 2

Esecuzione	
Cicli	163
Istruzioni	114
CPI	1.430
Stalli	
Stalli RAW	86
Stalli WAW	0
Stalli WAR	12
Stalli Strutturali	12
Branch Taken Stalls	7
Branch Misprediction	0
Code Size	
Bytes	116

Table 5: Confronto Precedente (Versione 1)

Nella Versione 2 del codice, è stato implementato un **Loop Unrolling** con l'obiettivo di eliminare alcune delle dipendenze e permettere l'adozione di ulteriori tecniche di ottimizzazione.

Grazie a questa tecnica, si è potuto eliminare il contatore delle colonne della matrice A, riducendo così gli *stalli strutturali* e i *Branch Taken Stalls*. In particolare, i cicli sono diminuiti da 163 a 156, mentre le istruzioni sono state ridotte da 114 a 73.

Tuttavia, il Loop Unrolling ha portato a un aumento degli stalli di tipo *RAW* (Read After Write) e *WAR* (Write After Read), rispettivamente da 86 a 102 e da 12 a 36, a causa della maggiore complessità del codice e dei conflitti tra le istruzioni. Inoltre, il CPI è aumentato da 1.430 a 2.137, indicando un uso meno efficiente delle risorse della CPU rispetto alla versione precedente. Questi dati sottolineano i compromessi tra riduzione delle dipendenze e gestione efficiente delle risorse della CPU.

3.2.4 Versione 3

```
1   .data
2   ; MATRICE A (2x2)
3   a:      .double 4.3, 2.1, 3.3, 5.2
4   ; MATRICE B (2x2)
5   b:      .double 2.3, 1.1, 4.3, 6.2
6   ; MATRICE C (2x2) (risultato)
7   c:      .double 0, 0, 0, 0
8   ;RIGHE A
9   o:      .word   2
10  ;COLONNE B
11  m:      .word   2
12  ;PARZIALE
13  p:      .double 0
14
15  .text  ;Instruction Reordering e Register Renaming
16  start:
17          DADDI   r1, r0, a       ; punta al primo elemento della matrice A
18          DADDI   r2, r0, b       ; punta al primo elemento della matrice B
19          DADDI   r3, r0, c       ; punta al primo elemento della matrice C
20          LW      r5, o(r0)       ; carica il numero di righe di A
21          LW      r6, m(r0)       ; carica il numero di colonne di B
22          L.D     f0, p(r0)       ; inizializzazione parziale
23
24  loop:
25          L.D     f1, 0(r1)       ; leggi a[i]
26          L.D     f2, 0(r2)       ; leggi b[i]
27          L.D     f3, 8(r1)       ; leggi a[i]
28          L.D     f4, 16(r2)      ; leggi b[i]
29          MUL.D   f1, f1, f2      ; a[i] * b[i]
30          MUL.D   f3, f3, f4      ; a[i] * b[i]
31          ADD.D   f0, f1, f3      ; p = p + a[i] * b[i] (risultato parziale)
32          S.D     f0, 0(r3)       ; salva il risultato
33          MUL.D   f0, f0, f5      ; resetta il risultato parziale
34
35  puntatori:
36          DADDI   r3, r3, 8       ; scorri registro risultati di C
37          DADDI   r2, r2, 8       ; punta alla nuova collonna di B
38          DADDI   r6, r6, -1      ; decrementa contatore colonne B
39          BNEZ    r6, loop        ; ripeti se contatore colonne B diverso da 0
40
41  righe:
42          DADDI r1, r1, 16        ; riporta puntatore A alla seconda riga
43          DADDI r2, r2, -16       ; riporta puntatore B all inizio
44          DADDI r6, r6, 2         ; resetta numero di colonne B
45          DADDI r5, r5, -1        ; decrementa contatore righe A
46          BNEZ  r5, loop          ; ripeti se contatore righe di A diverso da 0
47
48  end:    HALT
```

Le statistiche di questa versione sono:

Esecuzione	
Cicli	120
Istruzioni	69
CPI	1.739
Stalli	
Stalli RAW	62
Stalli WAW	0
Stalli WAR	12
Stalli Strutturali	6
Branch Taken Stalls	3
Branch Misprediction	0
Code Size	
Bytes	100

Table 6: Versione 3

Esecuzione	
Cicli	156
Istruzioni	73
CPI	2.137
Stalli	
Stalli RAW	102
Stalli WAW	0
Stalli WAR	36
Stalli Strutturali	6
Branch Taken Stalls	3
Branch Misprediction	0
Code Size	
Bytes	104

Table 7: Confronto Precedente (Versione 2)

L'introduzione di **Instruction Reordering** e **Register Renaming** nella Versione 3 ha portato a miglioramenti significativi rispetto alla Versione 2. Queste tecniche hanno permesso una riduzione dei cicli di esecuzione, passando da 156 a 120, e una diminuzione degli stalli *RAW* (Read After Write) da 102 a 62. Anche gli stalli *WAR* (Write After Read) sono stati ridotti da 36 a 12.

Questi miglioramenti sono dovuti alla capacità di riordinare le istruzioni in modo da minimizzare le dipendenze e di utilizzare registri diversi per evitare conflitti.

Il CPI (Clock Cycles Per Instruction) è migliorato significativamente, scendendo da 2.137 a 1.739, indicando un uso più efficiente delle risorse della CPU.

3.2.5 Versione 4

```
1   .data
2   ; MATRICE A (2x2)
3   a:      .double 4.3, 2.1, 3.3, 5.2
4   ; MATRICE B (2x2)
5   b:      .double 2.3, 1.1, 4.3, 6.2
6   ; MATRICE C (2x2) (risultato)
7   c:      .double 0, 0, 0, 0
8   ;RIGHE A
9   o:      .word   2
10  ;PARZIALE
11  p:      .double 0
12
13  .text  ;Loop Unrolling, Rimozione colonne B
14  start:
15          DADDI   r1, r0, a       ; punta al primo elemento della matrice A
16          DADDI   r2, r0, b       ; punta al primo elemento della matrice B
17          DADDI   r3, r0, c       ; punta al primo elemento della matrice C
18          LW      r5, o(r0)       ; carica il numero di righe di A
19          L.D     f0, p(r0)       ; inizializzazione parziale
20
21  loop:
22          L.D     f1, 0(r1)       ; leggi a[i]
23          L.D     f2, 0(r2)       ; leggi b[i]
24          L.D     f3, 8(r1)       ; leggi a[i]
25          L.D     f4, 16(r2)      ; leggi b[i]
26          MUL.D   f5, f1, f2      ; a[i] * b[i]
27          MUL.D   f6, f3, f4      ; a[i] * b[i]
28          ADD.D   f0, f5, f6      ; p = p + a[i] * b[i] (risultato parziale)
29          S.D     f0, 0(r3)       ; salva il risultato
30          MUL.D   f0, f0, f7      ; resetta il risultato parziale
31          L.D     f2, 8(r2)       ; leggi b[i]
32          L.D     f4, 24(r2)      ; leggi b[i]
33          MUL.D   f5, f1, f2      ; a[i] * b[i]
34          MUL.D   f6, f3, f4      ; a[i] * b[i]
35          ADD.D   f0, f5, f6      ; p = p + a[i] * b[i] (risultato parziale)
36          S.D     f0, 8(r3)       ; salva il risultato
37          MUL.D   f0, f0, f7      ; resetta il risultato parziale
38          DADDI   r3, r3, 16      ; scorri registro risultati di C
39
40  righe:
41          DADDI r1, r1, 16        ; riporta puntatore A alla seconda riga
42          DADDI r5, r5, -1        ; decrementa contatore righe A
43          BNEZ  r5, loop          ; ripeti se contatore righe di A diverso da 0
44
45  end:    HALT
```

Le statistiche di questa versione sono:

Esecuzione	
Cicli	90
Istruzioni	46
CPI	1.957
Stalli	
Stalli RAW	58
Stalli WAW	0
Stalli WAR	12
Stalli Strutturali	5
Branch Taken Stalls	1
Branch Misprediction	0
Code Size	
Bytes	104

Table 8: Versione 4

Esecuzione	
Cicli	120
Istruzioni	69
CPI	1.739
Stalli	
Stalli RAW	62
Stalli WAW	0
Stalli WAR	12
Stalli Strutturali	6
Branch Taken Stalls	3
Branch Misprediction	0
Code Size	
Bytes	100

Table 9: Confronto Precedente (Versione 3)

La Versione 4 introduce ulteriori ottimizzazioni grazie al **Loop Unrolling**, riducendo il numero di cicli da 120 a 90 e quello delle istruzioni da 69 a 46 rispetto alla Versione 3. Questo processo ha eliminato alcune dipendenze, permettendo l'applicazione di tecniche aggiuntive per diminuire ulteriormente il numero di cicli.

L'applicazione del **Loop Unrolling** ha consentito di rimuovere il contatore delle colonne della matrice B, precedentemente usato come indice del ciclo. Nonostante un leggero aumento del CPI da 1.739 a 1.957, la riduzione complessiva di cicli e istruzioni dimostra un progresso significativo nell'efficienza del codice. La Versione 4 si presenta quindi come un miglioramento rispetto alla precedente, mostrando un'ottimizzazione efficace del processo di calcolo.

3.2.6 Versione 5

```
1   .data
2   ; MATRICE A (2x2)
3   a:      .double 4.3, 2.1, 3.3, 5.2
4   ; MATRICE B (2x2)
5   b:      .double 2.3, 1.1, 4.3, 6.2
6   ; MATRICE C (2x2) (risultato)
7   c:      .double 0, 0, 0, 0
8   ;RIGHE A
9   o:      .word   2
10  ;PARZIALE
11  p:      .double 0
12
13  .text  ;Instruction Reordering e Register Renaming
14  start:
15          DADDI   r1, r0, a       ; punta al primo elemento della matrice A
16          DADDI   r2, r0, b       ; punta al primo elemento della matrice B
17          DADDI   r3, r0, c       ; punta al primo elemento della matrice C
18          LW      r5, o(r0)       ; carica il numero di righe di A
19          L.D     f0, p(r0)       ; inizializzazione parziale
20
21  loop:
22          L.D     f1, 0(r1)       ; leggi a[i]
23          L.D     f2, 0(r2)       ; leggi b[i]
24          L.D     f3, 8(r1)       ; leggi a[i]
25          L.D     f4, 16(r2)      ; leggi b[i]
26          L.D     f5, 8(r2)       ; leggi b[i]
27          L.D     f6, 24(r2)      ; leggi b[i]
28          MUL.D   f7, f1, f2      ; a[i] * b[i]
29          MUL.D   f8, f3, f4      ; a[i] * b[i]
30          MUL.D   f9, f1, f5      ; a[i] * b[i]
31          MUL.D   f10, f3, f6     ; a[i] * b[i]
32          ADD.D   f11, f7, f8     ; p = p + a[i] * b[i] (risultato parziale)
33          ADD.D   f12, f9, f10    ; p = p + a[i] * b[i] (risultato parziale)
34          DADDI r3, r3, 16        ; scorri registro risultati di C
35          DADDI r1, r1, 16        ; riporta puntatore A alla seconda riga
36          DADDI r5, r5, -1        ; decrementa contatore righe A
37          S.D   f11, 0(r3)        ; salva il risultato
38          S.D   f12, 8(r3)        ; salva il risultato
39          BNEZ  r5, loop          ; ripeti se contatore righe di A diverso da 0
40
41  end:    HALT
```

Le statistiche di questa versione sono:

Esecuzione	
Cicli	59
Istruzioni	42
CPI	1.405
Stalli	
Stalli RAW	10
Stalli WAW	0
Stalli WAR	0
Stalli Strutturali	4
Branch Taken Stalls	1
Branch Misprediction	0
Code Size	
Bytes	96

Table 10: Versione 5

Esecuzione	
Cicli	90
Istruzioni	46
CPI	1.957
Stalli	
Stalli RAW	58
Stalli WAW	0
Stalli WAR	12
Stalli Strutturali	5
Branch Taken Stalls	1
Branch Misprediction	0
Code Size	
Bytes	104

Table 11: Confronto Precedente (Versione 4)

La Versione 5 presenta un miglioramento significativo rispetto alla Versione 4 grazie all'utilizzo combinato di **Instruction Reordering** e **Register Renaming**, che hanno ottimizzato ulteriormente l'efficienza del codice.

In particolare, il numero di cicli è sceso da 90 a 59, mentre le istruzioni sono state ridotte da 46 a 42. Il CPI è diminuito da 1.957 a 1.405, indicando un uso più efficiente delle risorse della CPU. Queste ottimizzazioni hanno permesso di eliminare il contatore delle colonne delle matrici A e B, riducendo i conflitti e migliorando la pipeline.

Una delle principali differenze rispetto alla Versione 4 è l'eliminazione completa degli stalli di tipo *WAR (Write After Read)*, migliorando ulteriormente l'efficienza del programma.

3.2.7 Versione 6

```
1   .data
2   ; MATRICE A (2x2)
3   a:      .double 4.3, 2.1, 3.3, 5.2
4   ; MATRICE B (2x2)
5   b:      .double 2.3, 1.1, 4.3, 6.2
6   ; MATRICE C (2x2) (risultato)
7   c:      .double 0, 0, 0, 0
8   ;PARZIALE
9   p:      .double 0
10
11  .text  ;Loop Unrolling Totale + Instruction Reordering e Register Renaming
12  start:
13          DADDI   r1, r0, a       ; punta al primo elemento della matrice A
14          DADDI   r2, r0, b       ; punta al primo elemento della matrice B
15          DADDI   r3, r0, c       ; punta al primo elemento della matrice C
16          L.D     f0, p(r0)       ; inizializzazione parziale
17
18          ; Caricamento prima riga di A e B
19          L.D     f1, 0(r1)       ; leggi a[0][0]
20          L.D     f2, 0(r2)       ; leggi b[0][0]
21          L.D     f3, 8(r1)       ; leggi a[0][1]
22          L.D     f4, 8(r2)       ; leggi b[0][1]
23          ; Caricamento seconda riga di A e B
24          L.D     f5, 16(r1)      ; leggi a[1][0]
25          L.D     f6, 16(r2)      ; leggi b[1][0]
26          L.D     f7, 24(r1)      ; leggi a[1][1]
27          L.D     f8, 24(r2)      ; leggi b[1][1]
28
29          ; Moltiplicazioni per la prima riga di C
30          MUL.D   f9, f1, f2      ; a[0][0] * b[0][0]
31          MUL.D   f10, f3, f4     ; a[0][1] * b[0][1]
32          MUL.D   f11, f1, f6     ; a[0][0] * b[1][0]
33          MUL.D   f12, f3, f8     ; a[0][1] * b[1][1]
34          ; Moltiplicazioni per la seconda riga di C
35          MUL.D   f15, f5, f2     ; a[1][0] * b[0][0]
36          MUL.D   f16, f7, f4     ; a[1][1] * b[0][1]
37          MUL.D   f17, f5, f6     ; a[1][0] * b[1][0]
38          MUL.D   f18, f7, f8     ; a[1][1] * b[1][1]
39
40          ; Somme per la prima riga di C
41          ADD.D   f13, f9, f10    ; p = p + a[0][0] * b[0][0] + a[0][1] * b[0][1]
42          ADD.D   f14, f11, f12   ; p = p + a[0][0] * b[1][0] + a[0][1] * b[1][1]
43          ; Somme per la seconda riga di C
44          ADD.D   f19, f15, f16   ; p = p + a[1][0] * b[0][0] + a[1][1] * b[0][1]
45          ADD.D   f20, f17, f18   ; p = p + a[1][0] * b[1][0] + a[1][1] * b[1][1]
46
47          ; Salvataggio della prima riga di C
48          S.D     f13, 0(r3)      ; salva il risultato in c[0][0]
49          S.D     f14, 8(r3)      ; salva il risultato in c[0][1]
50          ; Salvataggio della seconda riga di C
51          S.D     f19, 16(r3)     ; salva il risultato in c[1][0]
52          S.D     f20, 24(r3)     ; salva il risultato in c[1][1]
53
54  end:    HALT
55
```

Le statistiche di questa versione sono:

Esecuzione	
Cicli	39
Istruzioni	29
CPI	1.345
Stalli	
Stalli RAW	2
Stalli WAW	0
Stalli WAR	0
Stalli Strutturali	7
Branch Taken Stalls	0
Branch Misprediction	0
Code Size	
Bytes	116

Table 12: Versione 6

Esecuzione	
Cicli	59
Istruzioni	42
CPI	1.405
Stalli	
Stalli RAW	10
Stalli WAW	0
Stalli WAR	0
Stalli Strutturali	4
Branch Taken Stalls	1
Branch Misprediction	0
Code Size	
Bytes	96

Table 13: Confronto Precedente (Versione 5)

Nella Versione 6 abbiamo effettuato un **Loop Unrolling Totale** e grazie all'applicazione combinata di: **Instruction Reordering** e **Register Renaming** ottenendo la versione migliore per l'ottimizzazione del nostro codice.

Queste ottimizzazioni hanno portato a un miglioramento complessivo delle prestazioni, con una significativa riduzione dei cicli, da 59 a 39, e delle istruzioni, da 42 a 29. Il CPI è diminuito ulteriormente da 1.405 a 1.345, dimostrando un uso più efficiente del processore.

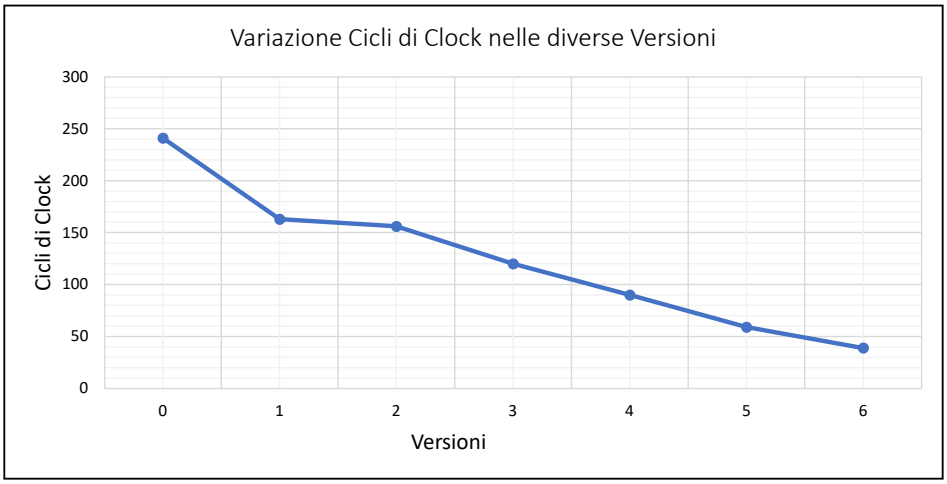
L'impiego del Loop Unrolling ha eliminato i conflitti tra le istruzioni, mentre l'Instruction Reordering e il Register Renaming hanno migliorato la gestione delle dipendenze tra i dati. Nonostante un leggero aumento degli stalli strutturali, dovuto a un maggiore utilizzo delle risorse hardware, ciò non compromette le prestazioni generali.

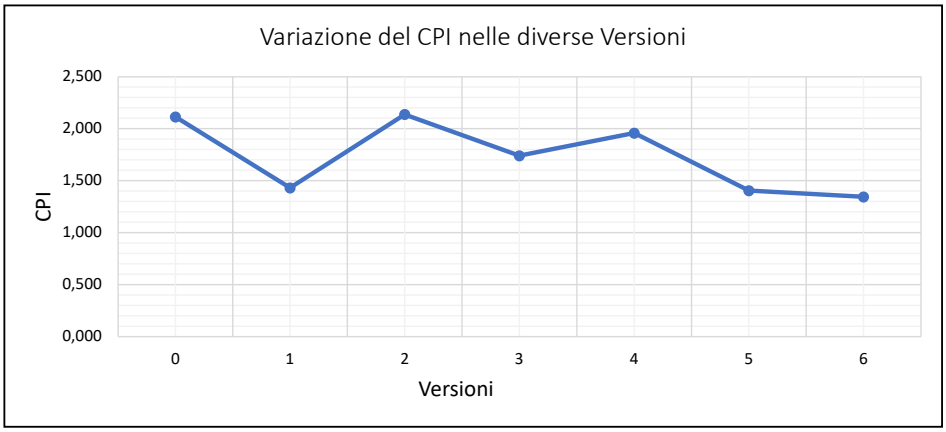
4 Conclusioni Finali

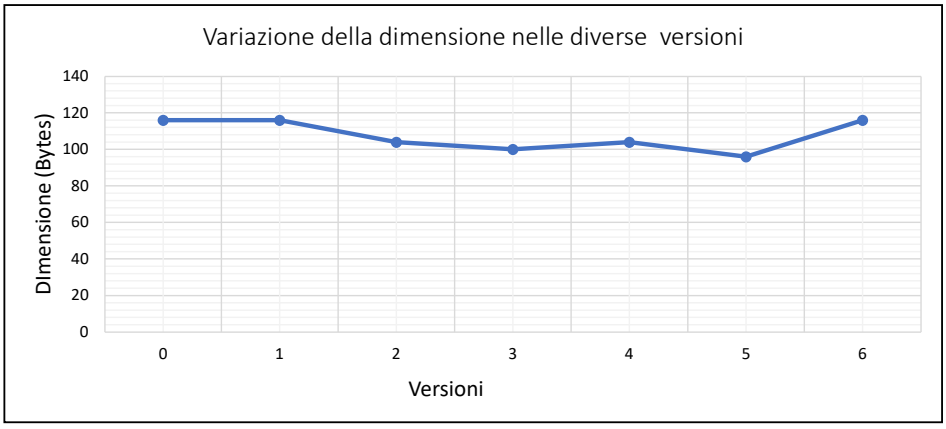
L'analisi delle diverse versioni ha dimostrato chiaramente l'importanza delle tecniche di ottimizzazione nel codice assembly. L'applicazione sistematica di tecniche come il **Data Forwarding**, **Loop Unrolling**, **Instruction Reordering**, e il **Register Renaming** ha portato a miglioramenti significativi nelle prestazioni complessive.

La Versione 6 rappresenta il culmine delle ottimizzazioni del nostro codice, raggiungendo le migliori prestazioni in termini di CPI, cicli totali e minimizzazione degli stalli.

Questi grafici illustrano visivamente l'evoluzione delle prestazioni attraverso le varie versioni del codice:







5 Confronto con altre matrici Quadrate

Per comprendere come la nostra ottimizzazione si comporterebbe con un aumento dei dati di input, abbiamo deciso di testarla su una matrice quadrata di dimensioni maggiori rispetto alla configurazione iniziale 2x2.In particolare, abbiamo scelto di ottimizzare il prodotto tra due matrici 4x4. Tuttavia, non è stato possibile utilizzare la nostra versione migliore, la versione 6, come base per questa ottimizzazione. Questo perché lo sdrotolamento completo del ciclo per matrici di tali dimensioni avrebbe richiesto un numero eccessivo di registri, superiore ai 32 disponibili per gestire sia le operazioni che i dati. Di conseguenza, abbiamo scelto di utilizzare la versione 5 come punto di partenza per l'ottimizzazione.

5.1 Versione 1

Di seguito il codice per calcolare il prodotto tra due matrici 4×4 senza alcuna ottimizzazione:

```
1   .data
2   ; MATRICE A (4x4)
3   a:      .double 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9, 1.2, 2.3, 3.4, 4.5, 5.6,
    6.7, 7.8
4   ; MATRICE B (4x4)
5   b:      .double 1.2, 2.3, 3.4, 4.5, 5.6, 6.7, 7.8, 8.9, 9.1, 1.2, 2.3, 3.4, 4.5, 5.6,
    6.7, 7.8
6   ; MATRICE C (4x4) (risultato)
7   c:      .double 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
8   ;COLONNE A
9   n:      .word   4
10  ;RIGHE A
11  o:      .word   4
12  ;COLONNE B
13  m:      .word   4
14  ;PARZIALE
15  p:      .double 0
16
17  .text
18  start:
19          DADDI   r1, r0, a       ; punta al primo elemento della matrice A
20          DADDI   r2, r0, b       ; punta al primo elemento della matrice B
21          DADDI   r3, r0, c       ; punta al primo elemento della matrice C
22          LW      r4, n(r0)       ; carica il numero di colonne di A
23          LW      r5, o(r0)       ; carica il numero di righe di A
24          LW      r6, m(r0)       ; carica il numero di colonne di B
25          L.D     f0, p(r0)       ; inizializzazione parziale
26
27  loop:
28          L.D     f1, 0(r1)       ; leggi a[i]
29          L.D     f2, 0(r2)       ; leggi b[i]
30          MUL.D   f1, f1, f2      ; a[i] * b[i]
31          ADD.D   f0, f0, f1      ; p = p + a[i] * b[i] (risultato parziale)
32          DADDI   r1, r1, 8       ; scorri elemento A
33          DADDI   r2, r2, 32      ; scorri elemento B
34          DADDI   r4, r4, -1      ; decrementa contatore colonne A
35          BNEZ    r4, loop        ; ripeti se contatore colonne A diverso da 0
36          S.D     f0, 0(r3)       ; salva il risultato
37          MUL.D   f0, f0, f3      ; resetta il risultato parziale
38
39  puntatori:
40          DADDI   r3, r3, 8       ; scorri registro risultati di C
41          DADDI   r2, r2, -120        ; riporta puntatore B all inizio + 1
42          DADDI   r1, r1, -32     ; riporta puntatore A all inizio
43          DADDI   r4, r4, 4       ; resetta contatore colonne A
44          DADDI   r6, r6, -1      ; decrementa contatore colonne B
45          BNEZ    r6, loop        ; ripeti se contatore colonne B diverso da 0
46
47  righe:
48          DADDI r1, r1, 32        ; riporta puntatore A alla seconda riga
49          DADDI r2, r2, -32       ; riporta puntatore B all inizio
50          DADDI r6, r6, 4         ; resetta numero di colonne B
51          DADDI r5, r5, -1        ; decrementa contatore righe A
52          BNEZ  r5, loop          ; ripeti se contatore righe di A diverso da 0
53
54  end:    HALT
```

Le statistiche di questa versione sono:

Esecuzione	
Cicli	947
Istruzioni	668
CPI	1.418
Stalli	
Stalli RAW	628
Stalli WAW	0
Stalli WAR	48
Stalli Strutturali	80
Branch Taken Stalls	63
Branch Misprediction	0
Code Size	
Bytes	116

Table 14: Versione 1

Nella Versione 1, la maggior parte degli stalli sono di tipo *RAW (Read After Write)*. Questi stalli si verificano tra le istruzioni *MUL.D* e *ADD.D*, poiché l'istruzione *ADD.D* dipende dal risultato dell'istruzione *MUL.D* per completare la sua esecuzione. In altre parole, *ADD.D* deve attendere che *MUL.D* produca il risultato necessario.

Si verificano anche stalli strutturali tra le istruzioni *BNEZ* e *DADDI*, che utilizzano lo stesso registro, *r4*, per operazioni diverse. Questi stalli si manifestano quando due istruzioni competono per l'accesso alla stessa risorsa hardware, causando conflitti e ritardi nell'esecuzione.

Infine, gli stalli di tipo Branch Taken si verificano dopo l'istruzione *BNEZ*, quando il processore deve attendere che l'istruzione di destinazione del branch venga recuperata e decodificata prima di proseguire l'esecuzione.

5.2 Versione 2

```
1   .data
2   ; MATRICE A (4x4)
3   a:      .double 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9, 1.2, 2.3, 3.4, 4.5, 5.6,
    6.7, 7.8
4   ; MATRICE B (4x4)
5   b:      .double 1.2, 2.3, 3.4, 4.5, 5.6, 6.7, 7.8, 8.9, 9.1, 1.2, 2.3, 3.4, 4.5, 5.6,
    6.7, 7.8
6   ; MATRICE C (4x4) (risultato)
7   c:      .double 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
8   ;RIGHE A
9   o:      .word   4
10  ;PARZIALE
11  p:      .double 0
12
13  .text   ;Loop Unrolling + Instruction Reordering e Register Renaming, Rimozione
    colonne A e B
14  start:
15          DADDI   r2, r0, a       ; punta al primo elemento della matrice A
16          DADDI   r3, r0, b       ; punta al primo elemento della matrice B
17          DADDI   r4, r0, c       ; punta al primo elemento della matrice C
18          LW      r6, o(r0)       ; carica il numero di righe di A
19          L.D     f0, p(r0)       ; inizializzazione parziale
20
21
22  loop:   L.D     f1, 0(r2)       ; leggi a[i]
23          L.D     f2, 0(r3)       ; leggi b[i]
24          L.D     f3, 8(r2)       ; leggi a[i]
25          L.D     f4, 32(r3)      ; leggi b[i]
26          L.D     f5, 16(r2)      ; leggi a[i]
27          L.D     f6, 64(r3)      ; leggi b[i]
28          L.D     f7, 24(r2)      ; leggi a[i]
29          L.D     f8, 96(r3)      ; leggi b[i]
30
31          L.D     f9, 8(r3)       ; leggi b[i]
32          L.D     f10, 40(r3)     ; leggi b[i]
33          L.D     f11, 72(r3)     ; leggi b[i]
34          L.D     f12, 104(r3)    ; leggi b[i]
35
36          L.D     f13, 16(r3)     ; leggi b[i]
37          L.D     f14, 48(r3)     ; leggi b[i]
38          L.D     f15, 80(r3)     ; leggi b[i]
39          L.D     f16, 112(r3)    ; leggi b[i]
40
41          L.D     f17, 24(r3)     ; leggi b[i]
42          L.D     f18, 56(r3)     ; leggi b[i]
43          L.D     f19, 88(r3)     ; leggi b[i]
44          L.D     f20, 120(r3)    ; leggi b[i]
45
46          MUL.D   f21, f1, f2     ; a[i]*b[i]
47          MUL.D   f22, f3, f4     ; a[i]*b[i]
48          MUL.D   f23, f5, f6     ; a[i]*b[i]
49          MUL.D   f24, f7, f8     ; a[i]*b[i]
50          MUL.D   f25, f1, f9     ; a[i]*b[i]
51          MUL.D   f26, f3, f10    ; a[i]*b[i]
52          MUL.D   f27, f5, f11    ; a[i]*b[i]
53          MUL.D   f28, f7, f12    ; a[i]*b[i]
54          MUL.D   f2, f1, f13     ; a[i]*b[i]
55          MUL.D   f4, f3, f14     ; a[i]*b[i]
56          MUL.D   f6, f5, f15     ; a[i]*b[i]
57          MUL.D   f8, f7, f16     ; a[i]*b[i]
58          MUL.D   f9, f1, f17     ; a[i]*b[i]
59          MUL.D   f10, f3, f18    ; a[i]*b[i]
60          MUL.D   f11, f5, f19    ; a[i]*b[i]
61          MUL.D   f12, f7, f20    ; a[i]*b[i]
62
63          ADD.D   f25, f25, f26   ; p = p + a[i]*b[i] (risultato parziale)
64          ADD.D   f27, f27, f28   ; p = p + a[i]*b[i] (risultato parziale)
65          ADD.D   f21, f21, f22   ; p = p + a[i]*b[i] (risultato parziale)
66          ADD.D   f23, f23, f24   ; p = p + a[i]*b[i] (risultato parziale)
67          ADD.D   f9, f9, f10     ; p = p + a[i]*b[i] (risultato parziale)
68          ADD.D   f2, f2, f4      ; p = p + a[i]*b[i] (risultato parziale)
69          ADD.D   f6, f6, f8      ; p = p + a[i]*b[i] (risultato parziale)
70          ADD.D   f11, f11, f12   ; p = p + a[i]*b[i] (risultato parziale)
```

```
72          ADD.D   f21, f21, f23   ; risulato finale 1
73          ADD.D   f25, f25, f27   ; risultato finale 2
74          ADD.D   f2, f2, f6      ; risultato finale 3
75          ADD.D   f9, f9, f11     ; risultato finale 4
76
77          DADDI   r6, r6, -1      ; decrementa contatore righe a
78          DADDI   r4, r4, 32      ; scorri registro risultati
79          DADDI   r2, r2, 32      ; riporta puntatore a alla seconda riga
80
81          S.D     f21, 0(r4)      ; salva il risultato
82          S.D     f25, 8(r4)      ; salva il risultato
83          S.D     f2, 16(r4)      ; salva il risultato
84          S.D     f9, 24(r4)      ; salva il risultato
85          BNEZ    r6, loop        ; ripeti se contatore right a diverso da 0
86
87  end:    HALT
```

Confronto tra le due versioni:

Esecuzione	
Cicli	261
Istruzioni	230
CPI	1.135
Stalli	
Stalli RAW	0
Stalli WAW	0
Stalli WAR	0
Stalli Strutturali	24
Branch Taken Stalls	3
Branch Misprediction	0
Code Size	
Bytes	248

Table 15: Versione 2

Esecuzione	
Cicli	947
Istruzioni	668
CPI	1.418
Stalli	
Stalli RAW	628
Stalli WAW	0
Stalli WAR	48
Stalli Strutturali	80
Branch Taken Stalls	63
Branch Misprediction	0
Code Size	
Bytes	116

Table 16: Confronto Precedente (Versione 1)

In questa Versione sono state applicate tecniche di ottimizzazione come il **Loop Unrolling**, il **Register Renaming** e l'**Instruction Reordering**, migliorando notevolmente l'efficienza del codice. Queste tecniche hanno portato a una drastica riduzione del numero di cicli, da 947 a 261, e delle istruzioni, da 668 a 230.

Una delle principali differenze rispetto alla Versione precedente è l'eliminazione completa degli stalli di tipo *RAW (Read After Write)* e *WAR (Write After Read)*. Questi stalli sono stati eliminati ottimizzando l'uso dei registri e riordinando le istruzioni, permettendo una gestione più efficiente delle dipendenze dei dati.

Gli *stalli strutturali* sono diminuiti significativamente, passando da 80 a 24, grazie a una migliore gestione delle risorse hardware.

Anche gli stalli di tipo *Branch Taken* sono stati ridotti da 63 a 3, ottimizzando il controllo di flusso e migliorando le prestazioni complessive.

Nonostante l'aumento della dimensione del codice da 116 a 248 byte, dovuto all'applicazione del **Loop Unrolling**, i miglioramenti in termini di riduzione dei cicli e degli stalli rendono questa Versione significativamente più efficiente rispetto alla Versione base.

5.3 Confronto matrice 2x2 e 4x4

In questa sezione specificheremo le differenze sostanziali tra il prodotto delle matrici 2×2 e 4×4. Di seguito vi sono le tabelle contenenti i valori relativi al prodotto delle matrici ottimizzate:

Esecuzione	
Cicli	59
Istruzioni	42
CPI	1.405
Stalli	
Stalli RAW	10
Stalli WAW	0
Stalli WAR	0
Stalli Strutturali	4
Branch Taken Stalls	1
Branch Misprediction	0
Code Size	
Bytes	96

Table 17: Prodotto 2x2

Esecuzione	
Cicli	261
Istruzioni	230
CPI	1.135
Stalli	
Stalli RAW	0
Stalli WAW	0
Stalli WAR	0
Stalli Strutturali	24
Branch Taken Stalls	3
Branch Misprediction	0
Code Size	
Bytes	248

Table 18: Prodotto 4x4

Nella sezione ***Esecuzione*** con l'aumento dei dati di input, si nota un aumento dei cicli di clock e delle istruzioni, mentre il CPI si riduce in modo significativo. Questo comportamento può essere spiegato dal fatto che l'incremento del carico di lavoro consente al processore di ottimizzare meglio l'uso delle risorse, riducendo il numero medio di cicli necessari per completare ogni istruzione. In altre parole, il processore sfrutta al meglio la parallelizzazione e il pipelining, migliorando l'efficienza complessiva dell'esecuzione.

Nella sezione ***Stalli***, si osserva la completa eliminazione degli stalli di tipo *RAW (Read After Write)* che nella versione 2x2 erano causati dalla bassa quantita di istruzioni presenti nel codice e venivano generato tra le istruzioni MUL.D e ADD.D.

D'altra parte, si osserva un aumento degli stalli strutturali e di tipo Branch Taken. Gli Strutturali sono dovuti dalle istruzioni ADD.D e DADDI che tentatano di usare la stessa risorsa hardware da parte di diverse istruzioni in modi diversi nello stesso ciclo di clock Gli stalli di tipo Branch Taken sono legati alla predizione errata dei salti condizionali, in questo caso sono associati all'istruzione BNEZ (Branch if Not Equal to Zero).

Nella sezione ***Code Size*** notiamo un aumento della dimensione del codice in Bytes che è direttamente correlato al numero di istruzioni generate. Quando vengono aggiunti nuovi dati di input, spesso è necessario aggiungere nuove istruzioni per gestire questi dati, il che porta a un aumento complessivo della dimensione del codice eseguibile.