

University of Calabria, DIMES
High Level Synthesis of Digital Systems
2023-2024
Prof.ssa PERRI
Prof. FRUSTACI
Sparse Matrix Vector Multiplication Analysis

Giorgio Ubbriaco
247284
bbrgrg00h11d086x@studenti.unical.it

June 2024

Index

1	Introduction	3
1.1	Sparse Matrix	3
1.2	Compressed Row Storage (CRS)	3
1.3	HLS Partitioning	3
2	Tasks to be performed	5
3	Definitions	6
4	C Simulations	7
5	Solutions	9
5.1	Solution 1	9
5.2	Solution 2	12
5.3	Solution 3	14
5.4	Solution 4	15
5.5	Solution 5	16
5.6	Solution 6	17
5.7	Solution 7	18
5.8	Solution 8	19
5.9	Solution 9	20
5.10	Solution 10	21
5.11	Solution 11	22
6	Conclusions	23

Listings

definitions/definitions.h	6
c_simulations/smvmtb.cpp	7
c_simulations/smvmtb_output.cpp	8
c_simulations/stdm.cpp	8
solutions/s1/s1.cpp	9
solutions/s1/s1tripcount.cpp	10
solutions/s2/s2.cpp	12

List of Figures

1	HLS Array Partitioning	4
2	HLS Loop Pipelining	12

List of Tables

1	SMVM Solutions To Be Performed	5
2	SMVM Solutions To Be Performed	9
3	HLS Solution 1 without Trip Count Timing Summary (ns)	10
4	HLS Solution 1 without Trip Count Latency Summary (clock cycles)	10
5	HLS Solution 1 without Trip Count Latency Loops Summary	10
6	HLS Solution 1 with Trip Count Timing Summary (ns)	11
7	HLS Solution 1 with Trip Count Latency Summary (clock cycles)	11
8	HLS Solution 1 Latency with Trip Count Loops Summary	11
9	HLS Solution 1 with Trip Count Utilization Estimates Summary	11
10	HLS Solution 1 with Trip Count C/RTL Cosimulation Summary	11
11	HLS Solution 1 with Trip Count Export RTL Resource Usage	11
12	HLS Solution with Trip Count Export RTL Final Timing	11
13	HLS Solution 2 Timing Summary (ns)	12
14	HLS Solution 2 without Trip Count Latency Summary (clock cycles)	12
15	HLS Solution 2 without Trip Count Latency Loops Summary	13
16	HLS Solution 1 with Trip Count Utilization Estimates Summary	13
17	HLS Solution 1 with Trip Count C/RTL Cosimulation Summary	13
18	HLS Solution 1 with Trip Count Export RTL Resource Usage	13
19	HLS Solution with Trip Count Export RTL Final Timing	13

1 Introduction

1.1 Sparse Matrix

Nell'analisi numerica, una **matrice sparsa** è una matrice in cui la maggior parte degli elementi è pari a zero. Non esiste una definizione rigorosa della proporzione di elementi a valore nullo affinché una matrice possa essere considerata sparsa. Al contrario, se la maggior parte degli elementi è non nulla, allora la matrice è considerata densa.

Una matrice è tipicamente memorizzata come un array bidimensionale. Ogni voce della matrice rappresenta un elemento $a_{i,j}$ della matrice e vi si accede tramite i due indici i e j . Per una matrice $m \times n$, la quantità di memoria necessaria per memorizzare la matrice in questo formato è proporzionale a $m \times n$ (senza considerare che è necessario memorizzare anche le dimensioni relative alla matrice).

Nel caso di una matrice sparsa, è possibile ridurre notevolmente i requisiti di memoria memorizzando solo le voci non nulle. A seconda del numero e della distribuzione delle voci non nulle, è possibile utilizzare diverse strutture di dati che consentono di ottenere enormi risparmi di memoria rispetto all'approccio di base. Il compromesso è che l'accesso ai singoli elementi diventa più complesso e sono necessarie strutture aggiuntive per poter recuperare la matrice originale senza ambiguità.

I formati possono essere divisi in due gruppi:

- Quelli che supportano una modifica efficiente, come DOK (Dictionary of Keys), LIL (List of Lists) o COO (Coordinate List), utilizzati solitamente per la costruzione della matrice.
- Quelli che supportano l'accesso e le operazioni matriciali efficienti, come CRS (Compressed Row Storage) o CCS (Compressed Column Storage).

1.2 Compressed Row Storage (CRS)

Il formato **Compressed Row Storage (CRS)** permette la rappresentazione di una matrice tramite tre array unidimensionali consentendo un accesso veloce alle righe e una moltiplicazione matrice-vettore efficiente. In particolare, i tre array utilizzati sono i seguenti:

- **values**
È un array contenente tutti gli elementi della matrice non nulli.
- **rowPtr**
È un array contenente gli indici, relativi all'array values, corrispondenti ai primi elementi non nulli di ogni riga.
- **columnIndex**
È un array contenente gli indici di colonna degli elementi non nulli.

1.3 HLS Partitioning

Il partizionamento serve per risolvere un problema tipicamente causato dagli array. Gli array sono implementati come BRAM, solitamente progettate per un dual-port massimo. Questo può limitare il throughput di un algoritmo ad alta intensità di read/write. La larghezza di banda può essere migliorata dividendo l'array (una singola BRAM) in array più piccoli (più BRAM), aumentando di fatto il numero di porte. Gli array vengono partizionati utilizzando la direttiva `ARRAY_PARTITION`. Vivado HLS offre tre tipi di partizionamento degli array. I tre tipi di partizionamento sono:

- **block**
L'array originale viene suddiviso in blocchi di uguali dimensioni di elementi consecutivi dell'array originale.
- **cyclic**
L'array originale viene suddiviso in blocchi di uguali dimensioni che interlacciano gli elementi dell'array originale.

- **complete**

L'operazione predefinita consiste nel dividere l'array nei suoi singoli elementi. Ciò corrisponde alla risoluzione di una memoria in registri.

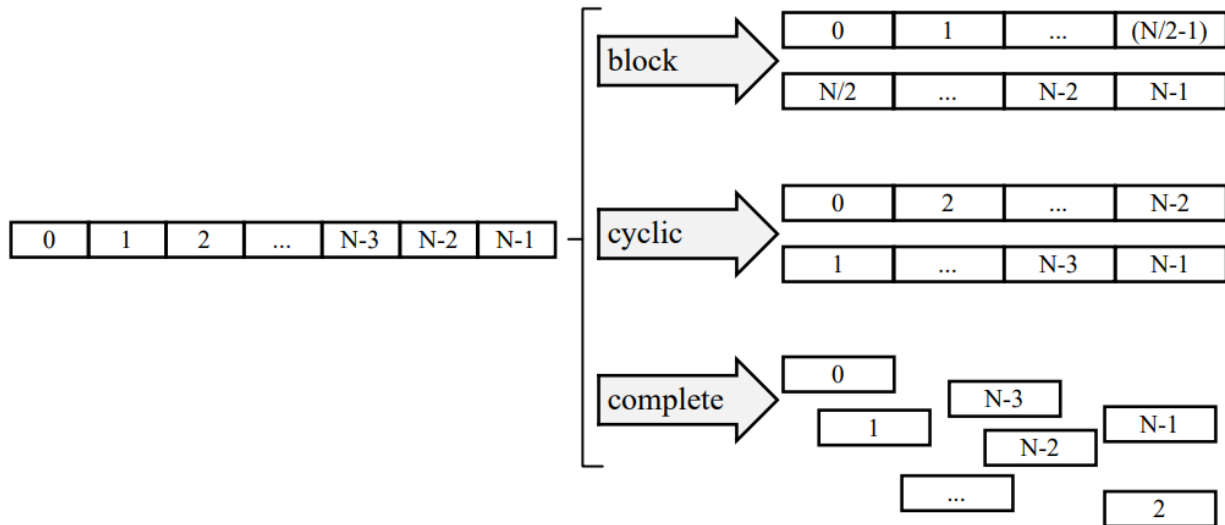


Figure 1: HLS Array Partitioning

2 Tasks to be performed

Prendendo come riferimento il formato CRS per il calcolo del prodotto tra una matrice sparsa ed un vettore e considerando il tool di sintesi ad alto livello per sistemi digitali, fornito da Xilinx[®], analizzare le soluzioni proposte nella seguente tabella utilizzando le direttive proprietarie citate e caratterizzando in termini di latenza e utilizzazione delle risorse.

Solution	Loop1	Loop2
1	-	-
2	-	Pipeline
3	Pipeline	-
4	Unroll=2	-
5	-	Pipeline, Unroll=2
6	-	Pipeline, Unroll=2, Cyclic=2
7	-	Pipeline, Unroll=4
8	-	Pipeline, Unroll=2, Cyclic=4
9	-	Pipeline, Unroll=8
10	-	Pipeline, Unroll=2, Cyclic=8
11	-	Pipeline, Unroll=2, Block=8

Table 1: SMVM Solutions To Be Performed

3 Definitions

Qui di seguito vengono riportate le definizioni e le intestazioni dei metodi corrispondenti alle soluzioni implementate per la moltiplicazione tra una matrice sparsa e un vettore. In particolare, ogni definizione presenta la documentazione associata. Inoltre, è stata prevista l'implementazione per la moltiplicazione standard così da poter verificare i risultati ottenuti tramite formato CRS.

```
1 #ifndef DEFINITIONS_H
2
3
4 /**
5  * Square Matrix Size.
6  */
7 const static int size = 4;
8
9 /**
10 * Number of Non-Zero Elements.
11 */
12 const static int nnz = 9;
13
14 /**
15 * Number of Rows.
16 */
17 const static int rows = 4;
18
19 /**
20 * Data Type.
21 */
22 typedef int DTYPE;
23
24 /**
25 * Matrix Vector Standard Multiplication Design.
26 * @param matrix[size][size] Input matrix
27 * @param y Multiplication Result
28 * @param x Input Vector
29 */
30 void std_multiplication(DTYPE matrix[size][size], DTYPE *y, DTYPE *x);
31
32 /**
33 * Sparse Matrix Vector Multiplication Design (CRS format).
34 * @param rowPtr[rows+1] Indexes First Elements
35 * @param columnIndex[nnz] Indexes Non Zero Elements
36 * @param values[nnz] Input Values
37 * @param y[size] Multiplication Result
38 * @param x[size] Input Vector
39 */
40 void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]);
41
42
43 #endif
```

4 C Simulations

Qui di seguito viene riportato il file testbench per la C Simulation in HLS e il corrispondente output ottenuto. In particolare, qui di seguito verrà riportato il caso in cui venga scelta la prima configurazione con matrice di dimensione 4×4 . Le altre configurazioni, relative ad alcune solution implementate, verranno presentate nei paragrafi successivi.

```
1 #include "definitions.h"
2 #include <iostream>
3 using namespace std;
4
5 void std_multiplication(DTYPE matrix[size][size], DTYPE *y, DTYPE *x);
6 void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]);
7
8
9 int main() {
10     int fail = 0;
11
12     DTYPE matrix[size][size] = {
13         {3,4,0,0},
14         {0,5,9,0},
15         {2,0,3,1},
16         {0,4,0,6}
17     };
18     DTYPE x[size] = {1, 1, 1, 1};
19     DTYPE values[] = {3, 4, 5, 9, 2, 3, 1, 4, 6};
20     int columnIndex[] = {0, 1, 1, 2, 0, 2, 3, 1, 3};
21     int rowPtr[] = {0,2,4,7,9};
22
23     /*
24     DTYPE matrix[size][size] = {
25         {3, 4, 0, 0, 0, 0, 0, 0, 0},
26         {0, 5, 9, 0, 0, 0, 0, 0, 0},
27         {2, 0, 3, 1, 0, 0, 0, 0, 0},
28         {0, 4, 0, 6, 0, 0, 0, 0, 0},
29         {0, 0, 0, 0, 0, 0, 0, 0, 0},
30         {0, 0, 0, 0, 0, 0, 0, 0, 0},
31         {0, 0, 0, 0, 0, 0, 0, 0, 0},
32         {0, 0, 0, 0, 0, 0, 0, 0, 0}
33     };
34     DTYPE x[size] = {1, 1, 1, 1, 1, 1, 1, 1, 1};
35     DTYPE values[] = {3, 4, 5, 9, 2, 3, 1, 4, 6};
36     int columnIndex[] = {0, 1, 1, 2, 0, 2, 3, 1, 3};
37     int rowPtr[] = {0, 2, 4, 7, 9, 9, 9, 9, 9};
38     */
39     /*
40     DTYPE matrix[size][size] = {
41         {3, 4, 0, 0, 0, 0, 0, 0, 0},
42         {0, 5, 9, 0, 0, 0, 0, 0, 0},
43         {2, 0, 3, 1, 0, 0, 0, 0, 0},
44         {0, 4, 0, 6, 0, 0, 0, 0, 0},
45         {1, 0, 0, 0, 0, 0, 0, 0, 0},
46         {1, 1, 0, 0, 0, 0, 0, 0, 0},
47         {1, 1, 0, 0, 0, 0, 0, 0, 0},
48         {1, 1, 0, 0, 0, 0, 0, 0, 0}
49     };
50     DTYPE x[size] = {1, 1, 1, 1, 1, 1, 1, 1, 1};
51     DTYPE values[] = {3, 4, 5, 9, 2, 3, 1, 4, 6, 1, 1, 1, 1, 1, 1, 1};
52     int columnIndex[] = {0, 1, 1, 2, 0, 2, 3, 1, 3, 0, 0, 1, 0, 1, 0, 1};
53     int rowPtr[] = {0, 2, 4, 7, 9, 10, 12, 14, 16};
54     */
55
56     DTYPE ystd[size];
57     std_multiplication(matrix, ystd, x);
58     DTYPE y[size];
59     smvm(rowPtr, columnIndex, values, y, x);
```

```

60     cout << endl;
61     for(int i=0; i<size; ++i) {
62         cout << "ystd=" << ystd[i] << ", ";
63         cout << "y=" << y[i] << endl;
64         if(ystd[i] != y[i] )
65             fail = 1;
66         if(fail == 1)
67             cout << "i=" << i << " failed." << endl;
68         else
69             cout << "i=" << i << " passed." << endl;
70     }
71     cout << endl;
72
73     return fail;
74 }

```

```

1  INFO: [SIM 211-2] ***** CSIM start *****
2  INFO: [SIM 211-4] CSIM will launch GCC as the compiler.
3  Compiling ../.././smvmTB.cpp in debug mode
4  Compiling ../.././smvm.cpp in debug mode
5  Compiling ../.././stdm.cpp in debug mode
6  Generating csim.exe
7
8  ystd=7, y=7
9  i=0 passed.
10 ystd=14, y=14
11 i=1 passed.
12 ystd=6, y=6
13 i=2 passed.
14 ystd=10, y=10
15 i=3 passed.
16
17 INFO: [SIM 211-1] CSim done with 0 errors.
18 INFO: [SIM 211-3] ***** CSIM finish *****
19 Finished C simulation.

```

Nello specifico, qui di seguito, viene riportata l'implementazione della moltiplicazione standard utilizzata per verificare i risultati sopra allegati.

```

1  #include "definitions.h"
2
3  void std_multiplication(DTYPE matrix[size][size], DTYPE *y, DTYPE *x) {
4      for (int i = 0; i < size; i++) {
5          DTYPE ytmp = 0;
6          for (int j = 0; j < size; j++)
7              ytmp += matrix[i][j] * x[j];
8          y[i] = ytmp;
9      }
10 }

```


5 Solutions

Di seguito verranno illustrate e analizzate le soluzioni previste nella tabella sotto allegata.

Nello specifico, nelle implementazioni dove è previsto l'utilizzo della direttiva di partitioning sono stati considerati tre array (columnIndex, values, x) a cui corrispondono quattro solution differenti. In particolare, è stata prevista una soluzione in cui viene effettuato il partitioning di tutte e tre gli array contemporaneamente e le rimanenti tre implementazioni in cui, per ognuna di essa, è stato previsto il partizionamento singolo di uno dei tre array appena citati. Tali implementazioni sono riportate qui di seguito.

Solution	Loop1	Loop2
1	-	-
2	-	Pipeline
3	Pipeline	-
4	Unroll=2	-
5	-	Pipeline, Unroll=2
6	- - - - -	Pipeline, Unroll=2, Cyclic=2 <ul style="list-style-type: none"> • Pipeline, Unroll=2, Cyclic=2 (columnIndex) • Pipeline, Unroll=2, Cyclic=2 (values) • Pipeline, Unroll=2, Cyclic=2 (x) • Pipeline, Unroll=2, Cyclic=2 (columnIndex, values, x)
7	-	Pipeline, Unroll=4
8	- - - -	Pipeline, Unroll=2, Cyclic=4 <ul style="list-style-type: none"> • Pipeline, Unroll=2, Cyclic=4 (columnIndex) • Pipeline, Unroll=2, Cyclic=4 (values) • Pipeline, Unroll=2, Cyclic=4 (x) • Pipeline, Unroll=2, Cyclic=4 (columnIndex, values, x)
9	-	Pipeline, Unroll=8
10	- - - -	Pipeline, Unroll=2, Cyclic=8 <ul style="list-style-type: none"> • Pipeline, Unroll=2, Cyclic=8 (columnIndex) • Pipeline, Unroll=2, Cyclic=8 (values) • Pipeline, Unroll=2, Cyclic=8 (x) • Pipeline, Unroll=2, Cyclic=8 (columnIndex, values, x)
11	- - - -	Pipeline, Unroll=2, Block=8 <ul style="list-style-type: none"> • Pipeline, Unroll=2, Block=8 (columnIndex) • Pipeline, Unroll=2, Block=8 (values) • Pipeline, Unroll=2, Block=8 (x) • Pipeline, Unroll=2, Block=8 (columnIndex, values, x)

Table 2: SMVM Solutions To Be Performed

5.1 Solution 1

Qui, di seguito, viene riportata l'architettura relativa alla prima solution.

```

1 #include "definitions.h"
2
3 void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
  x[size]) {
4   loop1: for (int i=0; i<rows; i++) {
5     DTYPE ytmp = 0;
6     loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
7       ytmp += values[k] * x[columnIndex[k]];
8     }
9     y[i] = ytmp;
10  }
11 }
```

Si può notare come venga utilizzata una variabile temporanea *ytmp* poiché essa viene utilizzata per calcolare l'uscita corrispondente. In particolare, il risultato calcolato ad ogni iterazione viene sommato a quello della precedente iterazione. Pertanto, essendo che l'uscita deve essere solo assegnata e non letta per ogni iterazione, si utilizza una variabile temporanea per calcolare il risultato. Solo alla fine delle iterazioni si potrà assegnare il risultato all'uscita corrispondente.

Effettuando la sintesi è possibile evidenziare il seguente report:

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.510	1.25

Table 3: HLS Solution 1 without Trip Count Timing Summary (ns)

Latency		Interval	
min	max	min	max
?	?	?	?

Table 4: HLS Solution 1 without Trip Count Latency Summary (clock cycles)

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count
	min	max		achieved	target	
- loop1	?	?	?	-	-	4
+ loop2	?	?	5	-	-	?

Table 5: HLS Solution 1 without Trip Count Latency Loops Summary

Si può notare come la latenza associata a questa architettura risulta essere "?", cioè non definita. In particolare tale non definizione è dovuta al loop2 del quale non è definito il trip count associato essendo il numero di iterazioni corrispondente non noto a priori. Nello specifico, il ciclo 2 dipende dai valori presenti all'interno dell'array *columnIndex* che non sono incogniti poiché dipendono dai valori in input all'architettura. Viceversa, la latenza per ogni iterazione (IL), essendo che dipende dalla tipologia di operazioni, risulta essere definita. Per quanto riguarda, invece, il loop1, esso presenta un'iteration latency non definita poiché dipendente direttamente dal loop2 di cui non si è a conoscenza della latenza totale come spiegato precedentemente. Pertanto, la latenza totale del loop1 e, di conseguenza, la latenza totale associata all'architettura risulta essere non nota a priori. Quindi, per poter risolvere si specifica all'interno dell'implementazione la direttiva *trip_count*. In particolare, si possono specificare tre valori all'interno di tale pragma: *min*, *max* e *avg*. Tali valori fanno riferimento rispettivamente al numero minimo, massimo e medio di iterazioni del loop di riferimento. Pertanto, tale direttiva permette al tool di analizzare come la latenza del loop contribuisce alla latenza totale dell'architettura così permettendo al progettista di effettuare ulteriori ottimizzazioni al design.

Pertanto, si allega l'architettura risultante.

```

1 #include "definitions.h"
2
3 void smvm(int iFirstEl[rows+1], int iNonZeroEl[noZeroEl], DTYPE values[noZeroEl], DTYPE
4   mulRes[size], DTYPE vector[size]) {
5   loop1: for (int i=0; i<rows; i++) {
6     DTYPE ytmp = 0;
7     loop2: for (int k=iFirstEl[i]; k<iFirstEl[i+1]; k++) {
8       #pragma HLS loop_tripcount min=0 max=4 avg=2
9       ytmp += values[k] * vector[iNonZeroEl[k]];
10    }
11    mulRes[i] = ytmp;
12  }

```

Effettuando la sintesi è possibile evidenziare il seguente report:

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.510	1.25

Table 6: HLS Solution 1 with Trip Count Timing Summary (ns)

Latency		Interval	
min	max	min	max
13	93	13	93

Table 7: HLS Solution 1 with Trip Count Latency Summary (clock cycles)

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count
	min	max		achieved	target	
- loop1	12	92	3~23	-	-	4
+ loop2	0	20	5	-	-	0~4

Table 8: HLS Solution 1 Latency with Trip Count Loops Summary

Si può notare come, dopo aver applicato la direttiva di trip_count, i valori di latenza risultano essere definiti numericamente. In particolare, il loop2 presenta un numero di iterazioni compresa tra 0 e 4, cioè rispettivamente il valore minimo e massimo specificati nel pragma di trip_count. Bisogna ricordare che tale direttiva non ha impatto sull'architettura ma ha solo impatto sui cicli di latenza.

Qui di seguito, viene allegato l'utilizzazione delle risorse stimata dal processo di sintesi.

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	137
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	-	-
Multiplexer	-	-	-	71
Register	-	-	241	-
Total	0	3	241	208
Available	280	220	106400	53200
Utilization (%)	0	1	~0	~0

Table 9: HLS Solution 1 with Trip Count Utilization Estimates Summary

Successivamente effettuando la C/RTL Cosimulation e l'Export RTL è possibile evidenziare i seguenti report.

RTL	Status	Latency			Interval		
		min	avg	max	min	avg	max
VHDL	Pass	58	58	58	NA	NA	NA

Table 10: HLS Solution 1 with Trip Count C/RTL Cosimulation Summary

Resource	VHDL
SLICE	48
LUT	93
FF	161
DSP	3
BRAM	0
SRL	0

Table 11: HLS Solution 1 with Trip Count Export RTL Resource Usage

Timing	VHDL
CP required	10.000
CP achieved post-synthesis	5.745
CP achieved post-implementation	5.692

Table 12: HLS Solution 1 with Trip Count Export RTL Final Timing

5.2 Solution 2

Qui, di seguito, viene riportata l'architettura relativa alla seconda solution.

```

1 #include "definitions.h"
2
3 void smvm(int iFirstEl[rows+1], int iNonZeroEl[noZeroEl], DTYPE values[noZeroEl], DTYPE
   mulRes[size], DTYPE vector[size]) {
4   loop1: for (int i=0; i<rows; i++) {
5     DTYPE ytmp = 0;
6     loop2: for (int k=iFirstEl[i]; k<iFirstEl[i+1]; k++) {
7       #pragma HLS loop_tripcount min=0 max=4 avg=2
8       #pragma HLS pipeline
9       ytmp += values[k] * vector[iNonZeroEl[k]];
10    }
11    mulRes[i] = ytmp;
12  }
13 }

```

In particolare, nella soluzione hardware in questione è stato aggiunto la direttiva di pipeline all'interno del loop2. Pertanto, ci si dovrebbe aspettare una minore latenza totale dal momento che il pipelining permette di scindere le operazioni complesse in più operazioni semplici. In questo modo si può far lavorare l'architettura con dati temporalmente differenti.

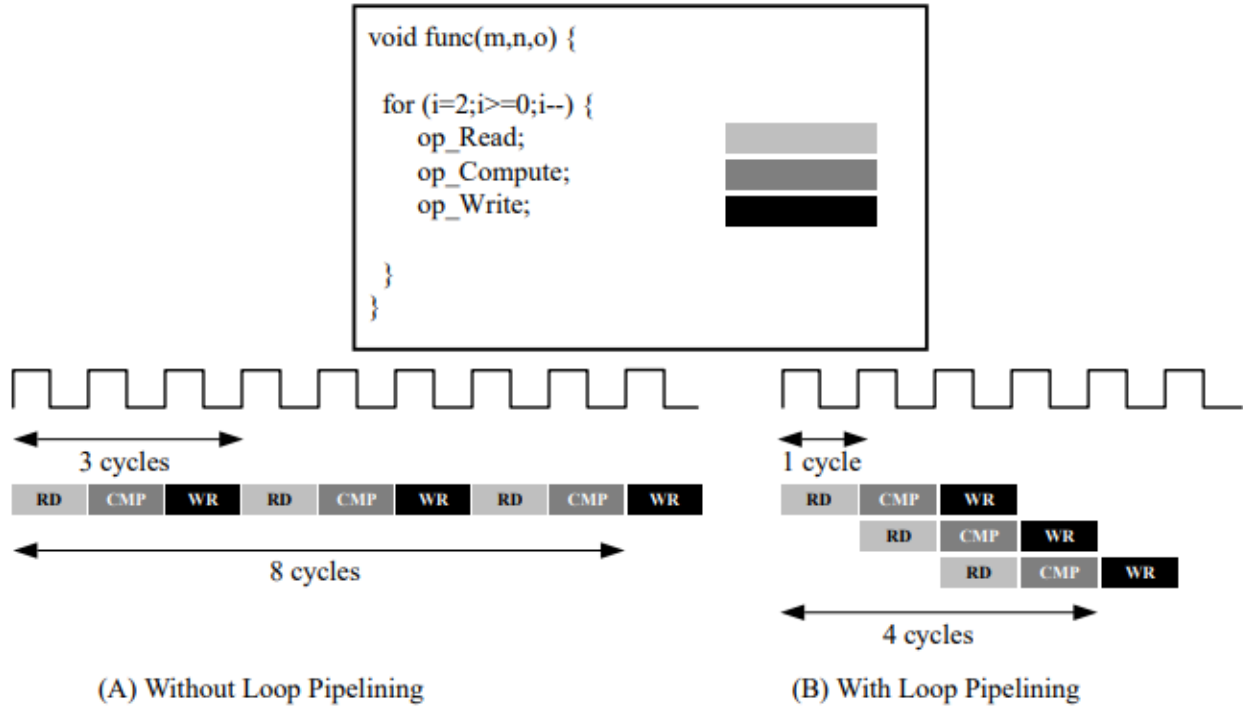


Figure 2: HLS Loop Pipelining

Effettuando la sintesi è possibile evidenziare il seguente report:

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.510	1.25

Table 13: HLS Solution 2 Timing Summary (ns)

Latency		Interval	
min	max	min	max
17	45	17	45

Table 14: HLS Solution 2 without Trip Count Latency Summary (clock cycles)

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count
	min	max		achieved	target	
- loop1	16	44	4~11	-	-	4
+ loop2	0	7	5	1	1	0~4

Table 15: HLS Solution 2 without Trip Count Latency Loops Summary

Si può notare, rispetto alla solution precedente, come in questo caso venga specificato un valore numerico di Initiation Interval (II). In particolare, l'II.achieved risulta essere il medesimo di quello target, cioè uguale a 1. Teoricamente, come in questo caso, si dovrebbe ottenere II.target=II.achieved. Se così non fosse allora il tool non è riuscito a raggiungere l'obiettivo prefissato e si dovrebbero attuare modifiche all'architetture o caso mai prevedere l'utilizzo di ulteriori direttive.

Qui di seguito, viene allegato l'utilizzazione delle risorse stimata dal processo di sintesi.

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	141
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	-	-
Multiplexer	-	-	-	78
Register	-	-	340	32
Total	0	3	340	251
Available	280	220	106400	53200
Utilization (%)	0	1	~0	~0

Table 16: HLS Solution 1 with Trip Count Utilization Estimates Summary

Successivamente effettuando la C/RTL Cosimulation e l'Export RTL è possibile evidenziare i seguenti report.

RTL	Status	Latency			Interval		
		min	avg	max	min	avg	max
VHDL	Pass	38	38	38	NA	NA	NA

Table 17: HLS Solution 1 with Trip Count C/RTL Cosimulation Summary

In particolare, si può notare come, in seguito all'introduzione della direttiva di pipeline, il numero di risorse risulta essere cambiato. Nello specifico, l'utilizzazione delle LUT è aumentata di circa il 24% mentre quella dei FF è diminuita di circa il 14%.

Resource	VHDL
SLICE	38
LUT	115
FF	139
DSP	3
BRAM	0
SRL	0

Table 18: HLS Solution 2t Export RTL Resource Usage

Timing	VHDL
CP required	10.000
CP achieved post-synthesis	5.745
CP achieved post-implementation	5.718

Table 19: HLS Solution 2 Export RTL Final Timing

5.3 Solution 3

5.4 Solution 4

5.5 Solution 5

5.6 Solution 6

5.7 Solution 7

5.8 Solution 8

5.9 Solution 9

5.10 Solution 10

5.11 Solution 11

6 Conclusions