

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	4
1.1	Sparse Matrix	4
1.2	Compressed Row Storage (CRS)	4
2	Tasks to be performed	5
3	Definitions	6
4	C Simulations	7
5	Solutions	9
5.1	Solution 1	10
5.2	Solution 2	13
5.3	Solution 3	15
5.4	Solution 4	19
5.5	Solution 5	22
5.6	Solution 6	25
5.7	Solution 7	29
5.8	Solution 8	34
5.9	Solution 9	35
5.10	Solution 10	36
5.11	Solution 11	37
6	Conclusions	38

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	10
solutions/s1/s1tripcount.cpp	10
solutions/s2/s2.cpp	13
solutions/s3/s3.cpp	15
solutions/s3/s3modificata.cpp	17
solutions/s4/s4.cpp	19
solutions/s5/s5.cpp	22
solutions/s6/s6all3.cpp	25
solutions/s6/s6columnIndex.cpp	26
solutions/s6/s6values.cpp	26
solutions/s6/s6x.cpp	26
solutions/s7/s7.cpp	29
solutions/s7/s7columnIndex.cpp	29
solutions/s7/s7values.cpp	30
solutions/s7/s7x.cpp	31
solutions/s7/s7loop1pipeline.cpp	32

List of Figures

1	HLS Solution 1 Analysis	11
2	HLS Loop Pipelining	13
3	HLS Solution 3 Analysis	16
4	HLS Loop Unrolling	19
5	HLS Solution 4 Analysis	20
6	HLS Solution 5 Analysis	23
7	HLS Array Partitioning	25

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	12
10	HLS Solution 1 with Trip Count C/RTL Cosimulation Summary	12
11	HLS Solution 1 with Trip Count Export RTL Resource Usage	12
12	HLS Solution 1 with Trip Count Export RTL Final Timing	12
13	HLS Solution 2 Timing Summary (ns)	13
14	HLS Solution 2 Latency Summary (clock cycles)	13
15	HLS Solution 2 Latency Loops Summary	13
16	HLS Solution 2 Utilization Estimates Summary	14
17	HLS Solution 1 with Trip Count C/RTL Cosimulation Summary	14
18	HLS Solution 2t Export RTL Resource Usage	14
19	HLS Solution 2 Export RTL Final Timing	14

20	HLS Solution 3 Timing Summary (ns)	15
21	HLS Solution 3 Latency Summary (clock cycles)	15
22	HLS Solution 3 Latency Loops Summary	15
23	HLS Solution 3 Utilization Estimates Summary	16
24	HLS Solution 1 with Trip Count C/RTL Cosimulation Summary	16
25	HLS Solution 2t Export RTL Resource Usage	17
26	HLS Solution 2 Export RTL Final Timing	17
27	HLS Solution 3 Modified Timing Summary (ns)	17
28	HLS Solution 3 Modified Latency Summary (clock cycles)	17
29	HLS Solution 3 Modified Latency Loops Summary	17
30	HLS Solution 3 Modified Utilization Estimates Summary	18
31	HLS Solution 4 Timing Summary (ns)	19
32	HLS Solution 4 Latency Summary (clock cycles)	19
33	HLS Solution 4 Latency Loops Summary	20
34	HLS Solution 4 Utilization Estimates Summary	20
35	HLS Solution 4 C/RTL Cosimulation Summary	21
36	HLS Solution 4 Export RTL Resource Usage	21
37	HLS Solution 4 Export RTL Final Timing	21
38	HLS Solution 5 Timing Summary (ns)	22
39	HLS Solution 5 Latency Summary (clock cycles)	22
40	HLS Solution 5 Latency Loops Summary	22
41	HLS Solution 5 Utilization Estimates Summary	23
42	HLS Solution 5 C/RTL Cosimulation Summary	23
43	HLS Solution 5 Export RTL Resource Usage	24
44	HLS Solution 5 Export RTL Final Timing	24
45	HLS Solution 6 Timing Summary (ns)	27
46	HLS Solution 6 Latency Summary (clock cycles)	27
47	HLS Solution 6 Latency Loops Summary	27
48	HLS Solution 6 Utilization Estimates [#]	27
49	HLS Solution 6 C/RTL Cosimulation Report	28
50	HLS Loop Unrolling Factor=2 Solution Export RTL Report	28
51	HLS Solution 7 Timing Summary (ns)	29
52	HLS Solution 7 Latency Summary (clock cycles)	29
53	HLS Solution 7 Latency Loops Summary	29
54	HLS Solution 7 with columnIndex partitioning Latency Loops Summary	30
55	HLS Solution 7 with columnIndex and values partitioning Latency Loops Summary	30
56	HLS Solution 7 with columnIndex, values and x partitioning Timing Summary (ns)	31
57	HLS Solution 7 with columnIndex, values and x partitioning Latency Summary (clock cycles)	31
58	HLS Solution 7 with columnIndex, values and x partitioning Latency Loops Summary	31
59	HLS Solution 7 with columnIndex, values and x partitioning Utilization Estimates Summary	32
60	HLS Solution 7 with columnIndex, values and x partitioning C/RTL Cosimulation Summary	32
61	HLS Solution 7 with columnIndex, values and x partitioning Export RTL Resource Usage	32
62	HLS Solution 7 with columnIndex, values and x partitioning Export RTL Final Timing	32
63	HLS Solution 7 with loop1 pipelined Timing Summary (ns)	33
64	HLS Solution 7 with loop1 pipelined Latency Summary (clock cycles)	33
65	HLS Solution 7 with loop1 pipelined Latency Loops Summary	33
66	HLS Solution 7 with loop1 pipelined Utilization Estimates Summary	33
67	HLS Solution 7 with loop1 pipelined C/RTL Cosimulation Summary	33
68	HLS Solution 7 with loop1 pipelined Export RTL Resource Usage	33
69	HLS Solution 7 with loop1 pipelined Export RTL Final Timing	33

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.

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
	-	• 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
	-	• 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
	-	• 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
	-	• 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 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          #pragma HLS loop_tripcount min=0 max=4 avg=2
8          ytmp += values[k] * x[columnIndex[k]];
9      }
10     y[i] = ytmp;
11 }
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.

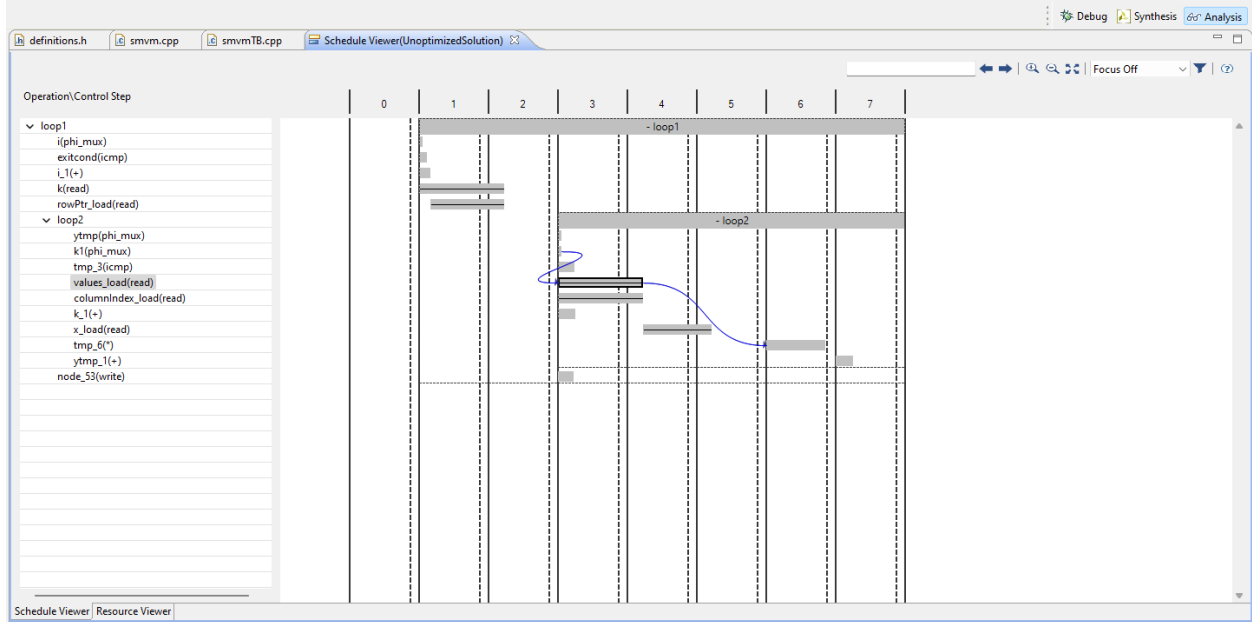


Figure 1: HLS Solution 1 Analysis

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 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       #pragma HLS loop_tripcount min=0 max=4 avg=2
8       #pragma HLS pipeline
9       ytmp += values[k] * x[columnIndex[k]];
10    }
11    y[i] = ytmp;
12  }
13 }

```

In particolare, nella soluzione hardware in questione, rispetto alla solution 1, 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.

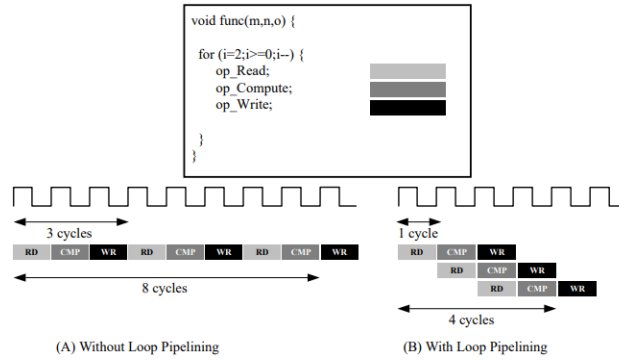


Figure 2: HLS Loop Pipelining

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 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 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'architettura 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 2 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

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

```

1 #include "definitions.h"
2
3 #include "definitions.h"
4
5 void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
   x[size]) {
6     loop1: for (int i=0; i<rows; i++) {
7         #pragma HLS pipeline
8         DTYPE ytmp = 0;
9         loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
10             #pragma HLS loop_tripcount min=0 max=4 avg=2
11             ytmp += values[k] * x[columnIndex[k]];
12         }
13         y[i] = ytmp;
14     }
15 }

```

In particolare, nella soluzione hardware in questione, rispetto alla solution 1, è stato aggiunta la direttiva di pipeline all'interno del loop1.

Effettuando la sintesi è possibile evidenziare il seguente report:

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.510	1.25

Table 20: HLS Solution 3 Timing Summary (ns)

Latency		Interval	
min	max	min	max
13	93	13	93

Table 21: HLS Solution 3 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 22: HLS Solution 3 Latency Loops Summary

Si può notare come, in questo caso l'Initiation Interval sia non specificato nel loop2 dal momento che la direttiva introdotta nella solution 2 è stata eliminata per la soluzione hardware in questione. Molto più importante è che, considerando la direttiva di pipeline definita all'interno del loop1, in corrispondenza dell'Initiation Interval di tale ciclo non è definito alcun valore numerico. Tanto è vero che, analizzando i log della sintesi presenti nella console è possibile identificare il seguente warning.

WARNING: [SCHD 204-65] Unable to satisfy pipeline directive: Loop contains subloop(s) not being unrolled or flattened.

In particolare, è come se il tool non riuscisse a soddisfare la richiesta di pipeline per il loop1 effettuata tramite la direttiva proprietaria. Questo potrebbe essere giustificato dal fatto che effettivamente la scissione dell'operazione "complessa" in micro-operazioni, all'interno del ciclo in questione, non è possibile effettuarla. Infatti, si può notare come i valori di latenza siano i medesimi di quelli della solution 1. Effettivamente, si potrebbe aggiungere la direttiva di pipeline all'interno del loop2, come fatto per la solution 2, dove sono presenti la maggior parte delle operazioni. In quel caso, infatti, il tool è riuscito a scomporre in micro-operazioni e così da permettere una minore latenza dal momento che i moduli potevano essere utilizzati da dati temporalmente differenti. Quello che si può notare è che nel loop1 le operazioni risultano essere l'inizializzazione della variabile temporanea *ytmp*, le operazioni interne al loop2 e la scrittura del valore di *ytmp* in *y*. Pertanto, le uniche operazioni complesse che potrebbero essere gestite tramite una direttiva di pipeline si trovano all'interno del loop2.

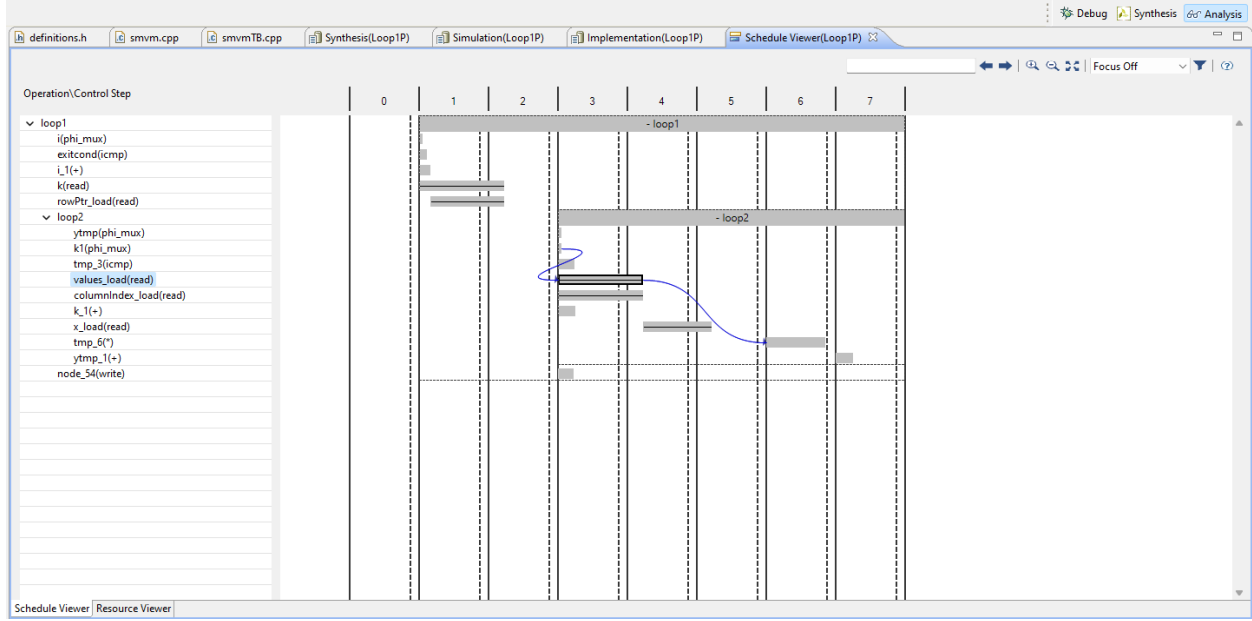


Figure 3: HLS Solution 3 Analysis

Qui di seguito, viene allegato l'utilizzazione delle risorse stimata dal processo di sintesi. Anche in questo caso il numero di risorse è il medesimo di quello ottenuto in corrispondenza della solution 1.

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 23: HLS Solution 3 Utilization Estimates Summary

Successivamente effettuando la C/RTL Cosimulation e l'Export RTL è possibile evidenziare i seguenti report. Anche in questo caso, sia il report del C/RTL Cosimulation sia quello dell'Export RTL risultano essere i medesimi di quelli della solution 1.

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

Table 24: HLS Solution 1 with Trip Count C/RTL Cosimulation Summary

Resource	VHDL
SLICE	48
LUT	94
FF	161
DSP	3
BRAM	0
SRL	0

Table 25: HLS Solution 2t Export RTL Resource Usage

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

Table 26: HLS Solution 2 Export RTL Final Timing

Pertanto, considerando la solution in questione, la si potrebbe modificare aggiungendo la direttiva di pipeline anche nel loop2 per capire se le ipotesi, effettuate precedentemente, possono essere confermate o meno.

```

1 #include "definitions.h"
2
3 #include "definitions.h"
4
5 void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
  x[size]) {
6   loop1: for (int i=0; i<rows; i++) {
7     #pragma HLS pipeline
8     DTYPE ytmp = 0;
9     loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
10      #pragma HLS loop_tripcount min=0 max=4 avg=2
11      #pragma HLS pipeline
12      ytmp += values[k] * x[columnIndex[k]];
13    }
14    y[i] = ytmp;
15  }
16 }

```

Adottando questo approccio, però, la nuova soluzione hardware in questione si ricondurrebbe alla solution 2 precedentemente analizzata dal momento che la direttiva di pipeline nel loop1 verrebbe comunque ignorata e quello che verrebbe effettivamente attuato sarebbe il pragma di pipeline all'interno del loop2. Infatti, effettuando la sintesi si può notare come sia presente il medesimo warning, alleegato precedentemente, e come sia i valori di latenza sia quelli dell'utilizzazione delle risorse siano i medesimi.

WARNING: [SCHD 204-65] Unable to satisfy pipeline directive: Loop contains subloop(s) not being unrolled or flattened.

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.510	1.25

Table 27: HLS Solution 3 Modified Timing Summary (ns)

Latency		Interval	
min	max	min	max
17	45	17	45

Table 28: HLS Solution 3 Modified 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 29: HLS Solution 3 Modified Latency Loops Summary

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 30: HLS Solution 3 Modified Utilization Estimates Summary

5.4 Solution 4

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

```

1 #include "definitions.h"
2
3 #include "definitions.h"
4
5 void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
  x[size]) {
6     loop1: for (int i=0; i<rows; i++) {
7         #pragma HLS unroll factor=2
8         DTYPE ytmp = 0;
9         loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
10             #pragma HLS loop_tripcount min=0 max=4 avg=2
11             ytmp += values[k] * x[columnIndex[k]];
12         }
13         y[i] = ytmp;
14     }
15 }

```

In particolare, nella soluzione hardware in questione, rispetto alla solution 1, è stata aggiunta la direttiva di unrolling con fattore pari a 2 all'interno del loop1.

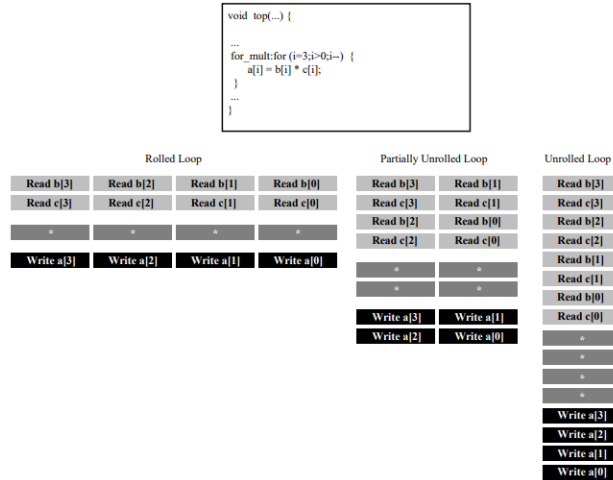


Figure 4: HLS Loop Unrolling

Effettuando la sintesi è possibile evidenziare il seguente report:

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.510	1.25

Table 31: HLS Solution 4 Timing Summary (ns)

Latency		Interval	
min	max	min	max
11	91	11	91

Table 32: HLS Solution 4 Latency Summary (clock cycles)

In particolare, si può notare come il valore di trip count del loop1 risulta essere dimezzato rispetto alla solution 1. Questo è dovuto all'attuazione della direttiva di unrolling di fattore 2 sul loop1 e così permettendo l'esecuzione in parallelo di due iterazioni.

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count
	min	max		achieved	target	
- loop1	10	90	5~45	-	-	2
+ loop2	0	20	5	-	-	0~4
+ loop2	0	20	5	-	-	0~4

Table 33: HLS Solution 4 Latency Loops Summary

Infatti, lo si può meglio notare tramite l'interfaccia analysis. In particolare, si può evidenziare come il loop1 venga parallelizzato. Nello specifico, vengono previsti due loop2 uno dopo l'altro facendo così aumentare la latenza per ogni iterazione del loop1. Quindi, il valore del trip count associato al ciclo 1 viene dimezzato mentre l'Iteration Latency associata allo stesso loop viene sostanzialmente raddoppiata.

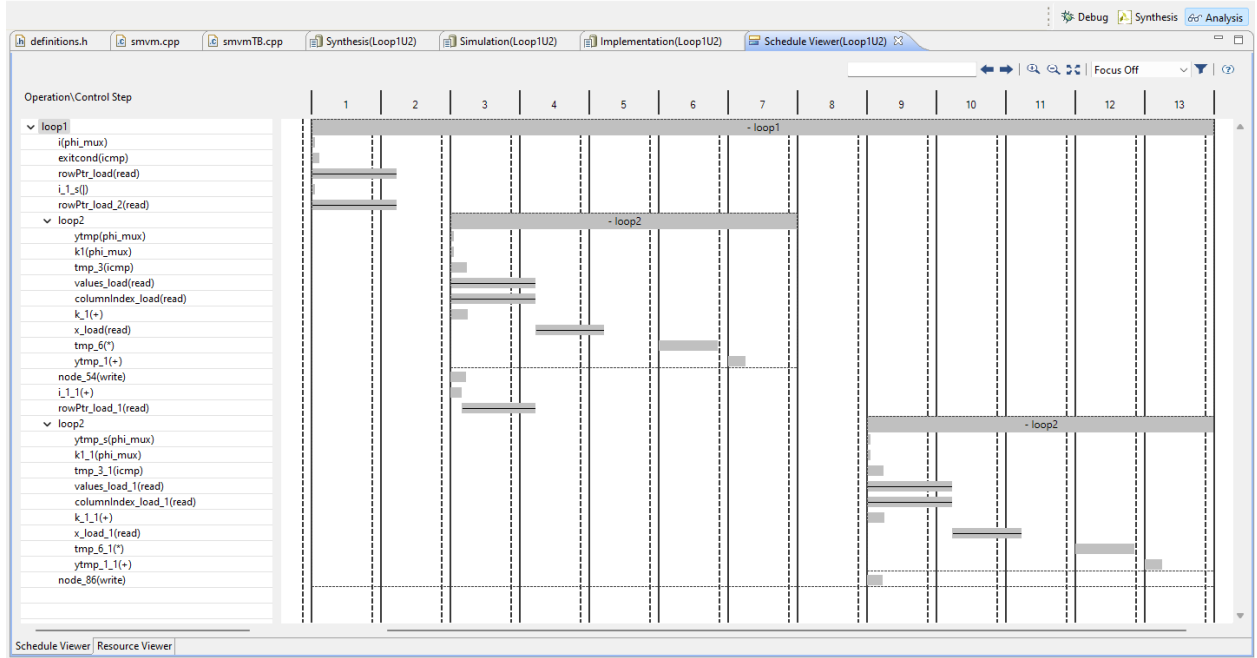


Figure 5: HLS Solution 4 Analysis

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	235
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	-	-
Multiplexer	-	-	-	197
Register	-	-	376	-
Total	0	3	376	432
Available	280	220	106400	53200
Utilization (%)	0	1	~0	~0

Table 34: HLS Solution 4 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	56	56	56	NA	NA	NA

Table 35: HLS Solution 4 C/RTL Cosimulation Summary

In particolare, rispetto alla solution 1 si ha un aumento del 100% e di circa l'81% rispettivamente dell'utilizzazione delle LUT e dei FF dal momento che è stato introdotto un parallelismo all'interno dell'architettura.

Resource	VHDL
SLICE	83
LUT	186
FF	292
DSP	3
BRAM	0
SRL	0

Table 36: HLS Solution 4 Export RTL Resource Usage

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

Table 37: HLS Solution 4 Export RTL Final Timing

5.5 Solution 5

Qui, di seguito, viene riportata l'architettura relativa alla quinta 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       #pragma HLS loop_tripcount min=0 max=4 avg=2
8       #pragma HLS pipeline
9       #pragma HLS unroll factor=2
10      ytmp += values[k] * x[columnIndex[k]];
11    }
12    y[i] = ytmp;
13  }
14 }

```

In particolare, nella soluzione hardware in questione, rispetto alla solution 2 dove era presenta solo la direttiva di unrolling nel loop2,, è stata aggiunta la direttiva di unrolling con fattore pari a 2 all'interno del loop2.

Effettuando la sintesi è possibile evidenziare il seguente report:

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.510	1.25

Table 38: HLS Solution 5 Timing Summary (ns)

Latency		Interval	
min	max	min	max
33	41	33	41

Table 39: HLS Solution 5 Latency Summary (clock cycles)

In particolare, si può notare come, rispetto alla solution 2 dove il trip count relativo al loop2 era pari a $0 \sim 4$, in questo caso il trip count associato al loop2 risulta essere dimezzato dal momento che è stato previsto un unrolling di fattore pari a 2 all'interno del ciclo in questione. Inoltre, dal momento che è stato introdotto una direttiva di pipeline all'interno del loop2, si può evidenziare come l'Initiation Interval raggiunto risulta essere il medesimo di quello target.

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count
	min	max		achieved	target	
- loop1	32	40	8~10	-	-	4
+ loop2	4	6	5	1	1	0~2

Table 40: HLS Solution 5 Latency Loops Summary

Si può notare come, all'interno del loop2, vengono effettuate in parallelo due letture relative alle variabili *columnIndex*, due relative alle variabili *values* e due relative alle variabili *x*.

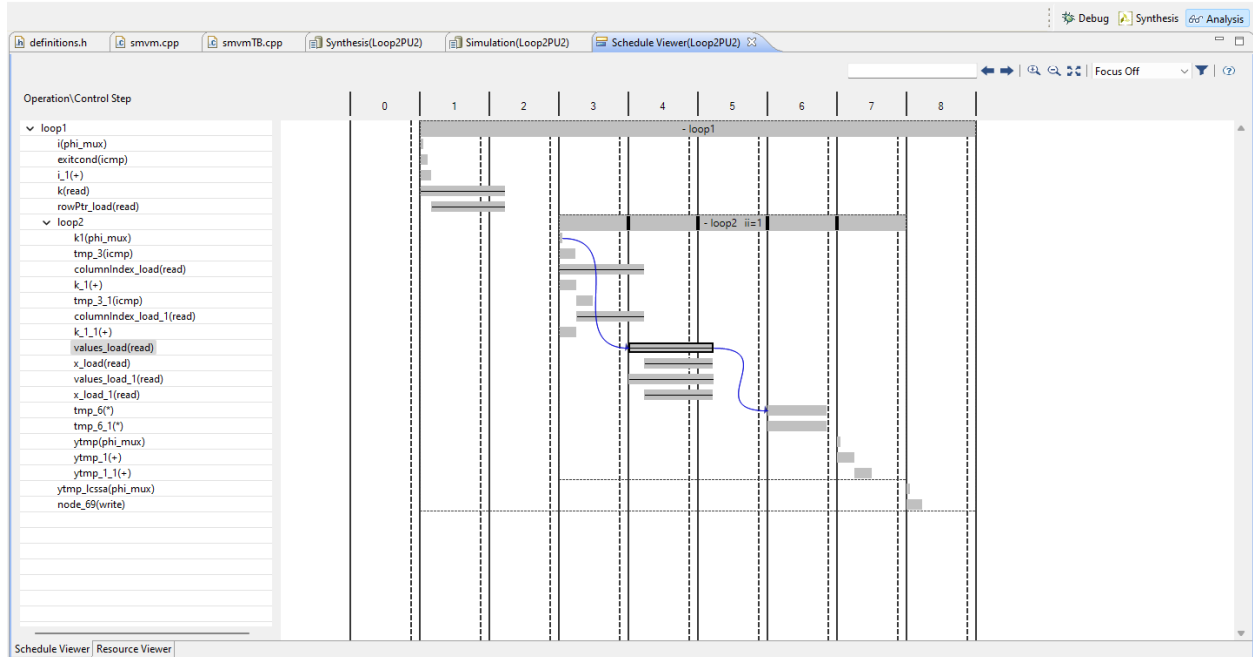


Figure 6: HLS Solution 5 Analysis

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	6	0	257
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	-	-
Multiplexer	-	-	-	78
Register	-	-	597	64
Total	0	6	597	399
Available	280	220	106400	53200
Utilization (%)	0	2	~0	~0

Table 41: HLS Solution 5 Utilization Estimates Summary

RTL	Status	Latency			Interval		
		min	avg	max	min	avg	max
VHDL	Pass	37	37	37	NA	NA	NA

Table 42: HLS Solution 5 C/RTL Cosimulation Summary

Si può notare, rispetto alla soluzione hardware 2, un aumento dell'utilizzazione delle risorse del 60% per quanto riguarda le LUT e di circa il 41% per quanto riguarda i FF. Inoltre, si può evidenziare come il numero dei DSP sia raddoppiato.

Resource	VHDL
SLICE	69
LUT	184
FF	196
DSP	6
BRAM	0
SRL	0

Table 43: HLS Solution 5 Export RTL Resource Usage

Timing	VHDL
CP required	10.000
CP achieved post-synthesis	7.927
CP achieved post-implementation	7.465

Table 44: HLS Solution 5 Export RTL Final Timing

5.6 Solution 6

Qui, di seguito, vengono riportate le architetture relative alla sesta solution. In particolare, come già precedentemente citato, tale solution prevede l'utilizzo della direttiva di 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.

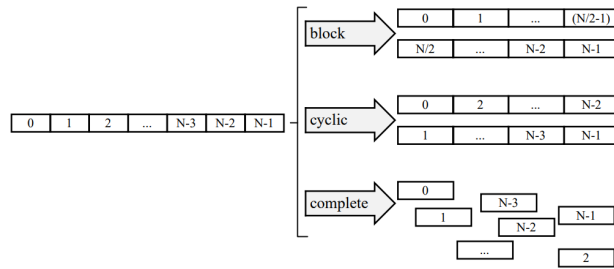


Figure 7: HLS Array Partitioning

Nella soluzione hardware in questione verrà utilizzata il partizionamento di tipologia cyclic e nello specifico, verranno analizzate le seguenti implementazioni relative al loop2:

- Pipeline, Unroll=2, Cyclic=2 (columnIndex, values, x)
- Pipeline, Unroll=2, Cyclic=2 (columnIndex)
- Pipeline, Unroll=2, Cyclic=2 (values)
- Pipeline, Unroll=2, Cyclic=2 (x)

In particolare, è possibile evidenziare nel dettaglio le differenti soluzioni hardware nei seguenti allegati.

```

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       #pragma HLS loop_tripcount min=0 max=4 avg=2
8       #pragma HLS pipeline
9       #pragma HLS unroll factor=2
10      #pragma HLS array_partition variable=columnIndex cyclic factor=2
11      #pragma HLS array_partition variable=values cyclic factor=2

```

```

12     #pragma HLS array_partition variable=x cyclic factor=2
13     ytmp += values[k] * x[columnIndex[k]];
14 }
15 y[i] = ytmp;
16 }
17 }

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             #pragma HLS loop_tripcount min=0 max=4 avg=2
8             #pragma HLS pipeline
9             #pragma HLS unroll factor=2
10            #pragma HLS array_partition variable=columnIndex cyclic factor=2
11            ytmp += values[k] * x[columnIndex[k]];
12        }
13        y[i] = ytmp;
14    }
15 }

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             #pragma HLS loop_tripcount min=0 max=4 avg=2
8             #pragma HLS pipeline
9             #pragma HLS unroll factor=2
10            #pragma HLS array_partition variable=values cyclic factor=2
11            ytmp += values[k] * x[columnIndex[k]];
12        }
13        y[i] = ytmp;
14    }
15 }

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             #pragma HLS loop_tripcount min=0 max=4 avg=2
8             #pragma HLS pipeline
9             #pragma HLS unroll factor=2
10            #pragma HLS array_partition variable=x cyclic factor=2
11            ytmp += values[k] * x[columnIndex[k]];
12        }
13        y[i] = ytmp;
14    }
15 }

```

Effettuando la sintesi è possibile evidenziare il seguente report:

Solution	Clock	Target	Estimated	Uncertainty
columnIndex, values, x	ap_clk	10.00	8.510	1.25
columnIndex	ap_clk	10.00	8.510	1.25
values	ap_clk	10.00	8.510	1.25
x	ap_clk	10.00	8.510	1.25

Table 45: HLS Solution 6 Timing Summary (ns)

Solution	Latency		Interval	
	min	max	min	max
columnIndex, values, x	33	41	33	41
columnIndex	33	41	33	41
values	33	41	33	41
x	33	41	33	41

Table 46: HLS Solution 6 Latency Summary (clock cycles)

Si può notare come, in corrispondenza di tutte e quattro le soluzioni hardware proposte in questa sezione, i valori di Iteration Latency, trip count, Initiation Interval relativa al loop2 e latenza totale del loop1 e loop2 risultano essere i medesimi. In particolare, il valore di trip count del loop2 risulta essere dimezzato, come quello della solution5, rispetto, ad esempio, alla solution2 dal momento che è presente la direttiva di unrolling di fattore pari a 2.

Solution	Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count
		min	max		achieved	target	
columnIndex, values, x	- loop1	32	40	8~10	-	-	4
	+ loop2	4	6	5	1	1	0~2
columnIndex	- loop1	32	40	8~10	-	-	4
	+ loop2	4	6	5	1	1	0~2
values	- loop1	32	40	8~10	-	-	4
	+ loop2	4	6	5	1	1	0~2
x	- loop1	32	40	8~10	-	-	4
	+ loop2	4	6	5	1	1	0~2

Table 47: HLS Solution 6 Latency Loops Summary

Solution	BRAM_18K	DSP48E	FF	LUT
columnIndex, values, x	0	6	535	623
columnIndex	0	6	533	472
values	0	6	533	472
x	0	6	599	463

Table 48: HLS Solution 6 Utilization Estimates [#]

Si può evidenziare come i valori di latenza associati alle quattro solution proposte risultano essere i medesimi di quelli ottenuti in corrispondenza della soluzione hardware 5.

Solution	RTL	Status	Latency			Interval		
			min	avg	max	min	avg	max
columnIndex, values, x	VHDL	Pass	37	37	37	NA	NA	NA
columnIndex	VHDL	Pass	37	37	37	NA	NA	NA
values	VHDL	Pass	37	37	37	NA	NA	NA
x	VHDL	Pass	37	37	37	NA	NA	NA

Table 49: HLS Solution 6 C/RTL Cosimulation Report

Si può notare come, in corrispondenza della soluzione basata su partitioning dell'array *columnIndex* e dell'array *values* si ha la medesima utilizzazione dei FF. Molto probabilmente questo risultato è legato al fatto che entrambi gli array presentano medesima dimensione, cioè pari a *nnz*. Inoltre, la minore utilizzazione delle risorse si ha in corrispondenza dell'array *x* a cui corrisponde, infatti, la dimensione minore tra i tre array considerati. In particolare, la soluzione in cui viene considerato il partizionamento dei tre array potrebbe essere considerata come la solution che richiede più risorse dal momento che presenta il maggior numero di slice utilizzate.

Solution	SLICE	LUT	FF	DSP	BRAM	CP required	CP achieved post- synthesis	CP achieved post- implementation
columnIndex, values, x	113	316	198	6	0	10	7.927	7.799
columnIndex	84	259	224	6	0	10	7.472	7.843
values	99	327	224	6	0	10	7.502	8.184
x	99	250	198	6	0	10	6.541	6.931

Table 50: HLS Loop Unrolling Factor=2 Solution Export RTL Report

5.7 Solution 7

Qui, di seguito, viene riportata l'architettura relativa alla settima 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       #pragma HLS loop_tripcount min=0 max=4 avg=2
8       #pragma HLS pipeline
9       #pragma HLS unroll factor=4
10      ytmp += values[k] * x[columnIndex[k]];
11    }
12    y[i] = ytmp;
13  }
14 }

```

In particolare, rispetto alla soluzione hardware 5 dove era stato considerato un parallelismo di fattore pari a 2, in questa solution è stato considerato un unrolling di fattore pari a 4. In particolare, ciò che ci si aspetta è un aumento delle risorse ed eventuali problematiche relative al timing dal momento che il tool deve gestire all'interno del loop2 più accessi in memoria paralleli.

Effettuando la sintesi è possibile evidenziare il seguente log nella console:

WARNING: [SCHD 204-69] Unable to schedule 'load' operation ('columnIndex_load_2', smvmProject/s-mvm.cpp:34) on array 'columnIndex' due to limited memory ports. Please consider using a memory core with more ports or partitioning the array 'columnIndex'.

Tale log sta a significare che non riesce a schedulare correttamente, dal punto di vista degli accessi in memoria, la load operation relativa all'array *columnIndex* dato dal numero limitato di porte relative alla memoria. In particolare, analizzando il report relativo alla sintesi, si può notare come l'Initiation Interval, associato al loop2, raggiunto risulta essere maggiore di quello target.

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.510	1.25

Table 51: HLS Solution 7 Timing Summary (ns)

Latency		Interval	
min	max	min	max
37	45	37	45

Table 52: HLS Solution 7 Latency Summary (clock cycles)

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count
	min	max		achieved	target	
- loop1	36	44	9~11	-	-	4
+ loop2	5	7	6	2	1	0~1

Table 53: HLS Solution 7 Latency Loops Summary

Pertanto, si potrebbe aggiungere una direttiva di partizionamento relativa all'array menzionato all'interno del log, cioè *columnIndex*.

```

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       #pragma HLS loop_tripcount min=0 max=4 avg=2
8       #pragma HLS pipeline
9       #pragma HLS unroll factor=4

```

```

10     #pragma HLS array_partition variable=columnIndex cyclic factor=4
11     ytmp += values[k] * x[columnIndex[k]];
12 }
13 y[i] = ytmp;
14 }
15 }

```

Effettuando nuovamente la sintesi, si ottiene il seguente log nella console e i seguenti valori di latenza.
WARNING: [SCHD 204-69] Unable to schedule 'load' operation ('values_load_2', smvmProject/smvm.cpp:34) on array 'values' due to limited memory ports. Please consider using a memory core with more ports or partitioning the array 'values'.

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count
	min	max		achieved	target	
- loop1	36	44	9~11	-	-	4
+ loop2	5	7	6	2	1	0~1

Table 54: HLS Solution 7 with columnIndex partitioning Latency Loops Summary

Si può notare come in questo caso il warning sia relativo all'array *values*. In particolare, la tipologia di warning è la medesima facendo presupporre che il tool non riesca a schedulare correttamente, secondo le direttive imposte dall'architettura, gli accessi in parallelo all'array *values*. Infatti, il valore di Iteration Latency raggiunto risulta essere ancora maggiore di quello di quello target. Pertanto, si potrebbe aggiungere una direttiva di partizionamento relativa all'array menzionato all'interno del log, cioè *values*.

```

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             #pragma HLS loop_tripcount min=0 max=4 avg=2
8             #pragma HLS pipeline
9             #pragma HLS unroll factor=4
10            #pragma HLS array_partition variable=columnIndex cyclic factor=4
11            #pragma HLS array_partition variable=values cyclic factor=4
12            ytmp += values[k] * x[columnIndex[k]];
13        }
14        y[i] = ytmp;
15    }
16 }

```

Effettuando nuovamente la sintesi, si ottiene il seguente log nella console e i seguenti valori di latenza.
WARNING: [SCHD 204-69] Unable to schedule 'load' operation ('x_load_2', smvmProject/smvm.cpp:34) on array 'x' due to limited memory ports. Please consider using a memory core with more ports or partitioning the array 'x'.

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count
	min	max		achieved	target	
- loop1	36	44	9~11	-	-	4
+ loop2	5	7	6	2	1	0~1

Table 55: HLS Solution 7 with columnIndex and values partitioning Latency Loops Summary

Si può notare come in questo caso il warning sia relativo all'array *x*. In particolare, la tipologia di warning è la medesima della precedente. Anche in questo caso il valore di Iteration Latency raggiunto risulta essere ancora maggiore di quello di quello target. Pertanto, si potrebbe aggiungere una direttiva di partizionamento relativa all'array menzionato all'interno del log, cioè *x*.

```

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       #pragma HLS loop_tripcount min=0 max=4 avg=2
8       #pragma HLS pipeline
9       #pragma HLS unroll factor=4
10      #pragma HLS array_partition variable=columnIndex cyclic factor=4
11      #pragma HLS array_partition variable=values cyclic factor=4
12      #pragma HLS array_partition variable=x cyclic factor=4
13      ytmp += values[k] * x[columnIndex[k]];
14    }
15    y[i] = ytmp;
16  }
17 }

```

Effettuando nuovamente la sintesi, si ottiene il seguente log nella console e il seguente report.

WARNING: [SCHD 204-21] Estimated clock period (10.208ns) exceeds the target (target clock period: 10ns, clock uncertainty: 1.25ns, effective delay budget: 8.75ns). WARNING: [SCHD 204-21] The critical path in module 'smvm' consists of the following:

'add' operation ('ytmp_1_3', smvmProject/smvm.cpp:34) [215] (2.55 ns)
 'phi' operation ('ytmp', smvmProject/smvm.cpp:34) with incoming values : ('ytmp_1_3', smvmProject/smvm.cpp:34) [48] (0 ns)
 'add' operation ('ytmp_1', smvmProject/smvm.cpp:34) [105] (2.55 ns)
 'add' operation ('ytmp_1_1', smvmProject/smvm.cpp:34) [145] (2.55 ns)
 'add' operation ('ytmp_1_2', smvmProject/smvm.cpp:34) [180] (2.55 ns)

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	10.208	1.25

Table 56: HLS Solution 7 with columnIndex, values and x partitioning Timing Summary (ns)

Latency		Interval	
min	max	min	max
37	41	37	41

Table 57: HLS Solution 7 with columnIndex, values and x partitioning Latency Summary (clock cycles)

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count
	min	max		achieved	target	
- loop1	36	40	9~10	-	-	4
+ loop2	5	6	6	1	1	0~1

Table 58: HLS Solution 7 with columnIndex, values and x partitioning Latency Loops Summary

Quello che si può notare è che all'interno della console viene visualizzato un warning indicante un periodo di clock stimato maggiore di quello target. In particolare, viene stimato un timing per ogni operazione in maniera dettagliata: 2.55ns per *add operation ytmp_1_3*, 0ns per *phi operation ytmp*, 2.55ns per *add operation ytmp_1*, 2.55ns per *add operation ytmp_1_1* e 2.55ns per *add operation ytmp_1_2*. Pertanto, calcolando la somma di tutti questi timing stimati si ottiene un periodo di clock stimato pari a 10.2ns. Nello specifico, il periodo di clock rimanente, cioè 0.008ns, evidentemente corrisponde al valore di timing relativo a *phi operation ytmp* che viene approssimato all'interno del report a 0ns.

Inoltre, si può notare come l'utilizzazione delle risorse sia notevolmente aumentata. In particolare, l'utilizzazione delle risorse, rispetto alle risorse disponibili della scheda, risultano essere pari al 5% per i DSP, all'1% per i FF e al 2% per le LUT.

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	12	0	848
FIFO	-	-	-	-
Instance	-	-	-	84
Memory	0	-	-	-
Multiplexer	-	-	-	321
Register	-	-	1338	128
Total	0	12	1338	1381
Available	280	220	106400	53200
Utilization (%)	0	5	1	2

Table 59: HLS Solution 7 with columnIndex, values and x partitioning Utilization Estimates Summary

Si procede con successivi passi così da verificare se tale problematica, riguardo il periodo di clock stimato superiore a quello target, possa essere risolta dal tool, tramite ulteriori ottimizzazioni, durante la fase di Export RTL.

RTL	Status	Latency			Interval		
		min	avg	max	min	avg	max
VHDL	Pass	37	37	37	NA	NA	NA

Table 60: HLS Solution 7 with columnIndex, values and x partitioning C/RTL Cosimulation Summary

Si può notare come il tool sia riuscito a risolvere la problematica riguardante il periodo di clock stimato superiore a quello target. Infatti, si evidenzia come quello raggiunto post-implementation risulta essere pari a $7.974ns$. Bisogna notare, però, che l'utilizzazione delle risorse risulta essere notevolmente alta dal momento che il tool ha attuato i partizionamenti di fattore pari a 4 su tutti e tre gli array precedentemente citati.

Resource	VHDL
SLICE	264
LUT	723
FF	571
DSP	12
BRAM	0
SRL	0

Table 61: HLS Solution 7 with columnIndex, values and x partitioning Export RTL Resource Usage

Timing	VHDL
CP required	10.000
CP achieved post-synthesis	7.449
CP achieved post-implementation	7.974

Table 62: HLS Solution 7 with columnIndex, values and x partitioning Export RTL Final Timing

Anche se le problematiche precedentemente citate sono state risolte, si potrebbe pensare di adottare un approccio differente così da cercare di ottenere una diminuzione delle risorse. In particolare, si potrebbero non considerare i tre partizionamenti all'interno del loop2 e aggiungere, invece, la direttiva di pipelining all'interno del loop1.

```

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         #pragma HLS pipeline
6         DTYPE ytmp = 0;
7         loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
8             #pragma HLS loop_tripcount min=0 max=4 avg=2
9             #pragma HLS pipeline
10            #pragma HLS unroll factor=4

```



```

11     ytmp += values[k] * x[columnIndex[k]];
12 }
13 y[i] = ytmp;
14 }
15 }

```

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	10.208	1.25

Table 63: HLS Solution 7 with loop1 pipelined Timing Summary (ns)

Latency		Interval	
min	max	min	max
17	45	17	45

Table 64: HLS Solution 7 with loop1 pipelined Latency Summary (clock cycles)

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count
	min	max		achieved	target	
- loop1	16	44	4~14	-	-	4
+ loop2	0	7	5	1	1	0~4

Table 65: HLS Solution 7 with loop1 pipelined Latency Loops Summary

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 66: HLS Solution 7 with loop1 pipelined Utilization Estimates Summary

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

Table 67: HLS Solution 7 with loop1 pipelined C/RTL Cosimulation Summary

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

Table 68: HLS Solution 7 with loop1 pipelined Export RTL Resource Usage

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

Table 69: HLS Solution 7 with loop1 pipelined Export RTL Final Timing

5.8 Solution 8

5.9 Solution 9

5.10 Solution 10

5.11 Solution 11

6 Conclusions