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 **5**
1.1 Sparse Matrix 5
1.2 Compressed Row Storage (CRS) 5

2 Tasks to be performed **6**

3 Definitions **7**

4 C Simulations **8**

5 Solutions **10**
5.1 Solution 1 11
5.2 Solution 2 14
5.3 Solution 3 16
5.4 Solution 4 18
5.5 Solution 5 21
5.6 Solution 6 24
5.7 Solution 7 28
5.8 Solution 8 32
5.9 Solution 9 35
5.10 Solution 10 41
5.11 Solution 11 47

6 Conclusions **51**

Listings

definitions/definitions.h . . . 7
c_simulations/smvmTB.cpp . . . 8
c_simulations/smvmTB_output.cpp . . . 9
c_simulations/stdm.cpp . . . 9
solutions/s1/s1.cpp . . . 11
solutions/s1/s1tripcount.cpp . . . 12
solutions/s2/s2.cpp . . . 14
solutions/s3/s3.cpp . . . 16
solutions/s4/s4.cpp . . . 18
solutions/s5/s5.cpp . . . 21
solutions/s6/s6all3.cpp . . . 24
solutions/s6/s6columnIndex.cpp . . . 25
solutions/s6/s6values.cpp . . . 25
solutions/s6/s6x.cpp . . . 25
solutions/s7/s7.cpp . . . 28
solutions/s7/s7columnIndex.cpp . . . 28
solutions/s7/s7values.cpp . . . 29
solutions/s7/s7x.cpp . . . 30
solutions/s8/s8all3.cpp . . . 32
solutions/s8/s8columnIndex.cpp . . . 32
solutions/s8/s8values.cpp . . . 32
solutions/s8/s8x.cpp . . . 33
solutions/s9/s9.cpp . . . 35
solutions/s9/s9columnIndex.cpp . . . 35
solutions/s9/s9values.cpp . . . 36
solutions/s9/s9x.cpp . . . 37
solutions/s9/s9rowPtr4.cpp . . . 37
solutions/s9/s9y4.cpp . . . 38
solutions/s9/s9pipelineloop1.cpp . . . 39
solutions/s10/s10all3.cpp . . . 41
solutions/s10/s10columnIndex.cpp . . . 41
solutions/s10/s10values.cpp . . . 41
solutions/s10/s10x.cpp . . . 42
solutions/s10/headermodified.h . . . 42
solutions/s10/s10all3modified.cpp . . . 42
solutions/s10/s10columnIndexmodified.cpp . . . 43
solutions/s10/s10valuesmodified.cpp . . . 43
solutions/s10/s10xmodified.cpp . . . 43
solutions/s11/definitions.h . . . 47
solutions/s11/s11all3.cpp . . . 47
solutions/s11/s11columnIndex.cpp . . . 47
solutions/s11/s11values.cpp . . . 48
solutions/s11/s11x.cpp . . . 48
solutions/s11/headermodified.h . . . 48

List of Figures

1 HLS Solution 1 Analysis . . . 12
2 HLS Loop Pipelining . . . 14
3 HLS Loop Unrolling . . . 18
4 HLS Solution 4 Analysis . . . 19
5 HLS Solution 5 Analysis . . . 22

6 HLS Array Partitioning . . . 24
7 HLS Solution 10 Analysis . . . 45
8 Utilization Export RTL Plot . . . 52

List of Tables

1 SMVM Solutions To Be Performed . . . 6
2 Detailed SMVM Solutions To Be Performed . . . 10
3 HLS Solution 1 without Trip Count Timing Summary (ns) . . . 11
4 HLS Solution 1 without Trip Count Latency Summary (clock cycles) . . . 11
5 HLS Solution 1 without Trip Count Latency Loops Summary . . . 11
6 HLS Solution 1 with Trip Count Timing Summary (ns) . . . 12
7 HLS Solution 1 with Trip Count Latency Summary (clock cycles) . . . 12
8 HLS Solution 1 Latency with Trip Count Loops Summary . . . 12
9 HLS Solution 1 with Trip Count Utilization Estimates Summary . . . 13
10 HLS Solution 1 with Trip Count C/RTL Cosimulation Summary . . . 13
11 HLS Solution 1 with Trip Count Export RTL Resource Usage . . . 13
12 HLS Solution 1 with Trip Count Export RTL Final Timing . . . 13
13 HLS Solution 2 Timing Summary (ns) . . . 14
14 HLS Solution 2 Latency Summary (clock cycles) . . . 14
15 HLS Solution 2 Latency Loops Summary . . . 14
16 HLS Solution 2 Utilization Estimates Summary . . . 15
17 HLS Solution 1 with Trip Count C/RTL Cosimulation Summary . . . 15
18 HLS Solution 2t Export RTL Resource Usage . . . 15
19 HLS Solution 2 Export RTL Final Timing . . . 15
20 HLS Solution 3 Timing Summary (ns) . . . 16
21 HLS Solution 3 Latency Summary (clock cycles) . . . 16
22 HLS Solution 3 Latency Loops Summary . . . 16
23 HLS Solution 3 Utilization Estimates Summary . . . 17
24 HLS Solution 3 C/RTL Cosimulation Summary . . . 17
25 HLS Solution 3 Export RTL Resource Usage . . . 17
26 HLS Solution 3 Export RTL Final Timing . . . 17
27 HLS Solution 4 Timing Summary (ns) . . . 18
28 HLS Solution 4 Latency Summary (clock cycles) . . . 18
29 HLS Solution 4 Latency Loops Summary . . . 19
30 HLS Solution 4 Utilization Estimates Summary . . . 19
31 HLS Solution 4 C/RTL Cosimulation Summary . . . 20
32 HLS Solution 4 Export RTL Resource Usage . . . 20
33 HLS Solution 4 Export RTL Final Timing . . . 20
34 HLS Solution 5 Timing Summary (ns) . . . 21
35 HLS Solution 5 Latency Summary (clock cycles) . . . 21
36 HLS Solution 5 Latency Loops Summary . . . 21
37 HLS Solution 5 Utilization Estimates Summary . . . 22
38 HLS Solution 5 C/RTL Cosimulation Summary . . . 22
39 HLS Solution 5 Export RTL Resource Usage . . . 23
40 HLS Solution 5 Export RTL Final Timing . . . 23
41 HLS Solution 6 Timing Summary (ns) . . . 26
42 HLS Solution 6 Latency Summary (clock cycles) . . . 26
43 HLS Solution 6 Latency Loops Summary . . . 26
44 HLS Solution 6 Utilization Estimates [#] . . . 26
45 HLS Solution 6 C/RTL Cosimulation Report . . . 27
46 HLS Solution 6 Export RTL Report . . . 27
47 HLS Solution 7 Timing Summary (ns) . . . 28
48 HLS Solution 7 Latency Summary (clock cycles) . . . 28

49 HLS Solution 7 Latency Loops Summary . . . 28
50 HLS Solution 7 with columnIndex partitioning Latency Loops Summary . . . 29
51 HLS Solution 7 with columnIndex and values partitioning Latency Loops Summary . . . 29
52 HLS Solution 7 with columnIndex, values and x partitioning Timing Summary (ns) . . . 30
53 HLS Solution 7 with columnIndex, values and x partitioning Latency Summary (clock cycles) 30
54 HLS Solution 7 with columnIndex, values and x partitioning Latency Loops Summary . . . 30
55 HLS Solution 7 with columnIndex, values and x partitioning Utilization Estimates Summary 31
56 HLS Solution 7 with columnIndex, values and x partitioning C/RTL Cosimulation Summary 31
57 HLS Solution 7 with columnIndex, values and x partitioning Export RTL Resource Usage . . 31
58 HLS Solution 7 with columnIndex, values and x partitioning Export RTL Final Timing . . . 31
59 HLS Solution 8 Timing Summary (ns) . . . 33
60 HLS Solution 8 Latency Summary (clock cycles) . . . 33
61 HLS Solution 8 Latency Loops Summary . . . 33
62 HLS Solution 8 Utilization Estimates [#] . . . 34
63 HLS Solution 8 C/RTL Cosimulation Report . . . 34
64 HLS Solution 8 Export RTL Report . . . 34
65 HLS Solution 9 Timing Summary (ns) . . . 35
66 HLS Solution 9 Latency Summary (clock cycles) . . . 35
67 HLS Solution 9 Latency Loops Summary . . . 35
68 HLS Solution 9 with columnIndex partitioning Latency Loops Summary . . . 36
69 HLS Solution 9 with columnIndex and values partitioning Latency Loops Summary . . . 36
70 HLS Solution 9 with columnIndex, values and x partitioning Latency Loops Summary . . . 37
71 HLS Solution 9 with columnIndex, values, x and rowPtr partitioning Latency Loops Summary 38
72 HLS Solution 9 with columnIndex, values, x, rowPtr and y partitioning Latency Loops Summary 39
73 HLS Solution 9 with columnIndex, values and x partitioning and loop1 pipelined Latency Loops Summary . . . 39
74 HLS Solution 9 with columnIndex, values and x partitioning and loop1 pipelined Utilization Estimates Summary . . . 40
75 HLS Solution 9 with columnIndex, values and x partitioning and loop1 pipelined C/RTL Cosimulation Summary . . . 40
76 HLS Solution 9 with columnIndex, values and x partitioning and loop1 pipelined Export RTL Resource Usage . . . 40
77 HLS Solution 9 with columnIndex, values and x partitioning and loop1 pipelined Export RTL Final Timing . . . 40
78 HLS Solution 10 Timing Summary (ns) . . . 43
79 HLS Solution 10 Latency Summary (clock cycles) . . . 44
80 HLS Solution 10 Latency Loops Summary . . . 44
81 HLS Solution 10 Utilization Estimates [#] . . . 44
82 HLS Solution 10 C/RTL Cosimulation Report . . . 45
83 HLS Solution 10 Export RTL Report . . . 46
84 HLS Solution 11 Timing Summary (ns) . . . 49
85 HLS Solution 11 Latency Summary (clock cycles) . . . 49
86 HLS Solution 11 Latency Loops Summary . . . 49
87 HLS Solution 11 Utilization Estimates [#] . . . 49
88 HLS Solution 11 C/RTL Cosimulation Report . . . 50
89 HLS Solution 11 Export RTL Report . . . 50
90 HLS Conclusions Export RTL Report . . . 51

1 Introduction

1.1 Sparse Matrix

Nell'analisi numerica, una **matrice sparsa** è una matrice in cui la maggior parte degli elementi presenta un valore nullo. 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 $a_{i,j}$ e vi si accede tramite l'indice di riga i e l'indice di colonna 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.

```
#ifndef DEFINITIONS_H


/**
 * Square Matrix Size.
 */
const static int size = 4;

/**
 * Number of Non-Zero Elements.
 */
const static int nnz = 9;

/**
 * Number of Rows.
 */
const static int rows = 4;

/**
 * Data Type.
 */
typedef int DTYPE;

/**
 * Matrix Vector Standard Multiplication Design.
 * @param matrix[size][size] Input matrix
 * @param y Multiplication Result
 * @param x Input Vector
 */
void std_multiplication(DTYPE matrix[size][size], DTYPE *y, DTYPE *x);

/**
 * Sparse Matrix Vector Multiplication Design (CRS format).
 * @param rowPtr[rows+1] Indexes First Elements
 * @param columnIndex[nnz] Indexes Non Zero Elements
 * @param values[nnz] Input Values
 * @param y[size] Multiplication Result
 * @param x[size] Input Vector
 */
void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]);


#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.

```
#include "definitions.h"
#include <iostream>
using namespace std;

void std_multiplication(DTYPE matrix[size][size], DTYPE *y, DTYPE *x);
void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]);


int main() {
  int fail = 0;

  DTYPE matrix[size][size] = {
    {3,4,0,0},
    {0,5,9,0},
    {2,0,3,1},
    {0,4,0,6}
  };
  DTYPE x[size] = {1, 1, 1, 1};
  DTYPE values[] = {3, 4, 5, 9, 2, 3, 1, 4, 6};
  int columnIndex[] = {0, 1, 1, 2, 0, 2, 3, 1, 3};
  int rowPtr[] = {0,2,4,7,9};

  /*
  DTYPE matrix[size][size] = {
    {3, 4, 0, 0, 0, 0, 0, 0},
    {0, 5, 9, 0, 0, 0, 0, 0},
    {2, 0, 3, 1, 0, 0, 0, 0},
    {0, 4, 0, 6, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0}
  };
  DTYPE x[size] = {1, 1, 1, 1, 1, 1, 1, 1};
  DTYPE values[] = {3, 4, 5, 9, 2, 3, 1, 4, 6};
  int columnIndex[] = {0, 1, 1, 2, 0, 2, 3, 1, 3};
  int rowPtr[] = {0, 2, 4, 7, 9, 9, 9, 9, 9};
  */
  /*
  DTYPE matrix[size][size] = {
    {3, 4, 0, 0, 0, 0, 0, 0},
    {0, 5, 9, 0, 0, 0, 0, 0},
    {2, 0, 3, 1, 0, 0, 0, 0},
    {0, 4, 0, 6, 0, 0, 0, 0},
    {1, 0, 0, 0, 0, 0, 0, 0},
    {1, 1, 0, 0, 0, 0, 0, 0},
    {1, 1, 0, 0, 0, 0, 0, 0},
    {1, 1, 0, 0, 0, 0, 0, 0}
  };
  DTYPE x[size] = {1, 1, 1, 1, 1, 1, 1, 1};
  DTYPE values[] = {3, 4, 5, 9, 2, 3, 1, 4, 6, 1, 1, 1, 1, 1, 1, 1};
  int columnIndex[] = {0, 1, 1, 2, 0, 2, 3, 1, 3, 0, 0, 1, 0, 1, 0, 1};
  int rowPtr[] = {0, 2, 4, 7, 9, 10, 12, 14, 16};
  */

  DTYPE ystd[size];
  std_multiplication(matrix, ystd, x);
  DTYPE y[size];
  smvm(rowPtr, columnIndex, values, y, x);
```

```
  cout << endl;
  for(int i=0; i<size; ++i) {
    cout << "ystd=" << ystd[i] << ", ";
    cout << "y=" << y[i] << endl;
    if(ystd[i] != y[i] )
      fail = 1;
    if(fail == 1)
      cout << "i=" << i << " failed." << endl;
    else
      cout << "i=" << i << " passed." << endl;
  }
  cout << endl;

  return fail;
}
```

```
INFO: [SIM 211-2] *************** CSIM start ***************
INFO: [SIM 211-4] CSIM will launch GCC as the compiler.
Compiling ../../../smvmTB.cpp in debug mode
Compiling ../../../smvm.cpp in debug mode
Compiling ../../../stdm.cpp in debug mode
Generating csim.exe

ystd=7, y=7
i=0 passed.
ystd=14, y=14
i=1 passed.
ystd=6, y=6
i=2 passed.
ystd=10, y=10
i=3 passed.

INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
Finished C simulation.
```

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

```
#include "definitions.h"

void std_multiplication(DTYPE matrix[size][size], DTYPE *y, DTYPE *x) {
  for (int i = 0; i < size; i++) {
    DTYPE ytmp = 0;
    for (int j = 0; j < size; j++)
      ytmp += matrix[i][j] * x[j];
    y[i] = ytmp;
  }
}
```

5 Solutions

Di seguito verranno illustrate e analizzate le soluzioni previste nella tabella sotto allegata. In particolare, 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. Nello specifico, 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: Detailed SMVM Solutions To Be Performed

5.1 Solution 1

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

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

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 k 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 *rowPtr* che 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. In particolare, tenendo conto che in generale rowPtr[i] indica il numero di elementi non nulli al di sopra della i-esima riga all'interno della matrice, il loop2 è come se scandisse riga per riga la matrice. All'iterazione i del loop1 considera le iterazioni possibili all'interno del loop2 comprese tra il valore k=rowPtr[i] (corrispondente al numero di elementi non nulli presenti al di sopra della riga i) e il valore k=rowPtr[i+1] (corrispondente al numero di elementi non nulli presenti al di sopra della riga i+1). Quindi, la differenza tra rowPtr[i+1] e rowPtr[i] rappresenta il numero di elementi presenti all'interno della riga i. Pertanto, tale direttiva permette al tool di analizare 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.

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

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.

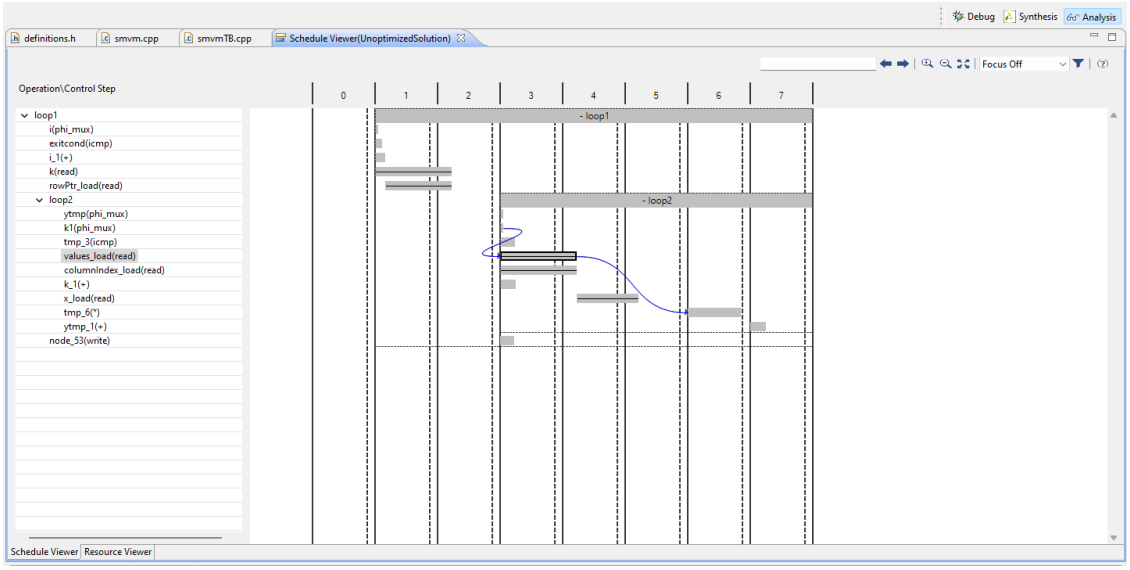

Figure 1: HLS Solution 1 Analysis

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.

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

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.

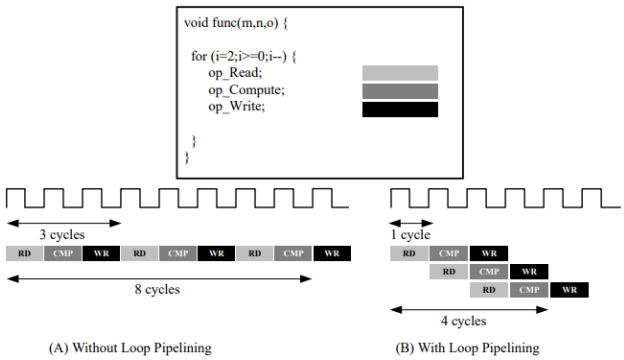

Figure 2: HLS Loop Pipelining

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'architetture o eventualmente 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.

```
#include "definitions.h"

#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    #pragma HLS pipeline
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

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 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 i seguenti warning.

WARNING: [XFORM 203-503] Cannot unroll loop 'loop2' (smvmProject/smvm.cpp:21) in function 'smvm' completely: variable loop bound.

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

In particolare, il tool non è riuscito a soddisfare la richiesta di pipeline a causa dei bound non noti e, pertanto, non riuscendo ad effettuare l'automatic unrolling del loop2. Infatti, si può notare come i valori di latenza siano i medesimi di quelli della solution 1.

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.

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.

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

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

Table 24: HLS Solution 3 C/RTL Cosimulation Summary

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

Table 25: HLS Solution 3 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 3 Export RTL Final Timing

5.4 Solution 4

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

```
#include "definitions.h"

#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    #pragma HLS unroll factor=2
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

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.

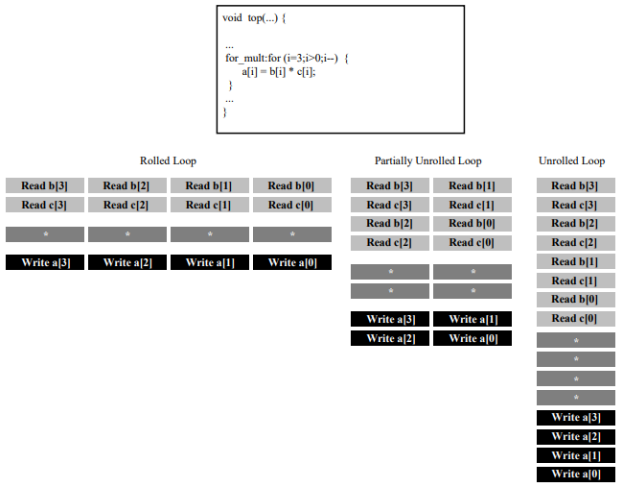

Figure 3: HLS Loop Unrolling

Effettuando la sintesi è possibile evidenziare il seguente report:

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.510	1.25

Table 27: HLS Solution 4 Timing Summary (ns)

Latency		Interval	
min	max	min	max
11	91	11	91

Table 28: 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 min	Latency max	Iteration Latency	Initiation Interval achieved	Initiation Interval target	Trip Count
- loop1	10	90	5∼45	-	-	2
+ loop2	0	20	5	-	-	0∼4
+ loop2	0	20	5	-	-	0∼4

Table 29: 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 aumenta dal momento che vengono gestiti due loop2.

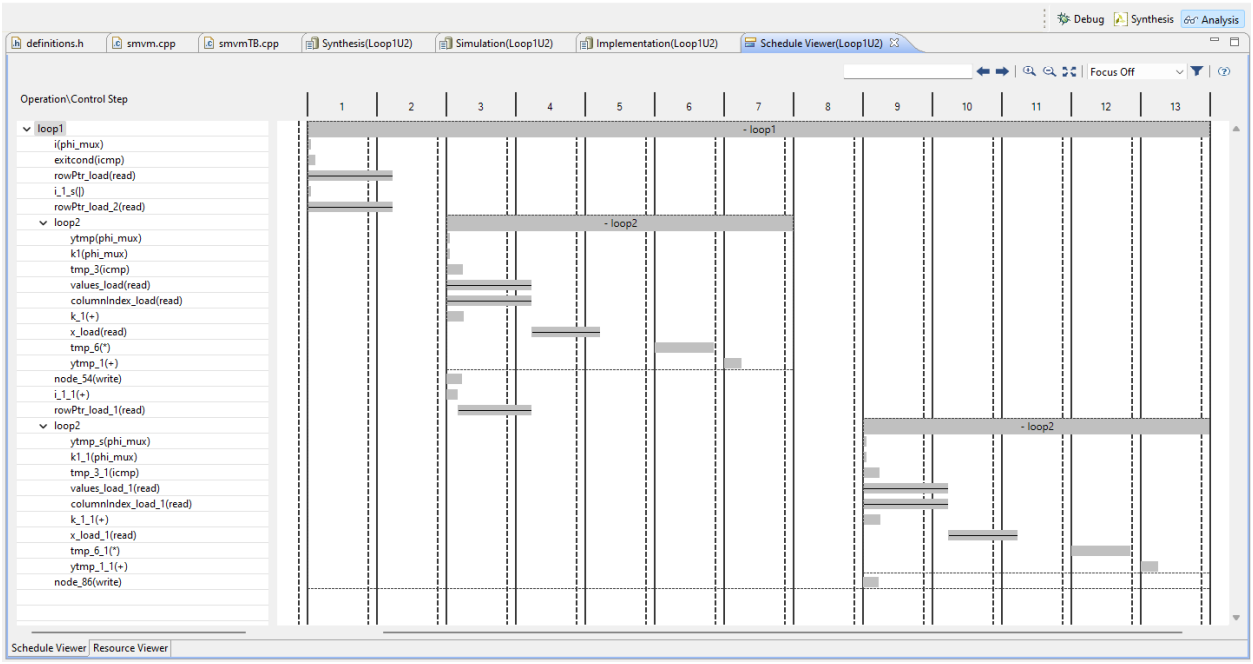

Figure 4: HLS Solution 4 Analysis

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 30: 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 31: HLS Solution 4 C/RTL Cosimulation Summary

In particolare, rispetto alla solution 1 si ha un aumento di circa il 73% delle slice, del 100% delle LUT e di circa l'81% 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 32: 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 33: HLS Solution 4 Export RTL Final Timing

5.5 Solution 5

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

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

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

Effettuando la sintesi è possibile evidenziare il seguente report:

Clock	**Target**	**Estimated**	**Uncertainty**
ap_clk	10.00	8.510	1.25

Table 34: HLS Solution 5 Timing Summary (ns)

Latency		**Interval**	
min	max	min	max
33	41	33	41

Table 35: 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	**Latency**		**Iteration Latency**	**Initiation Interval**		**Trip Count**
Name	min	max		achieved	target	
- loop1	32	40	8∼10	-	-	4
+ loop2	4	6	5	1	1	0∼2

Table 36: 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.

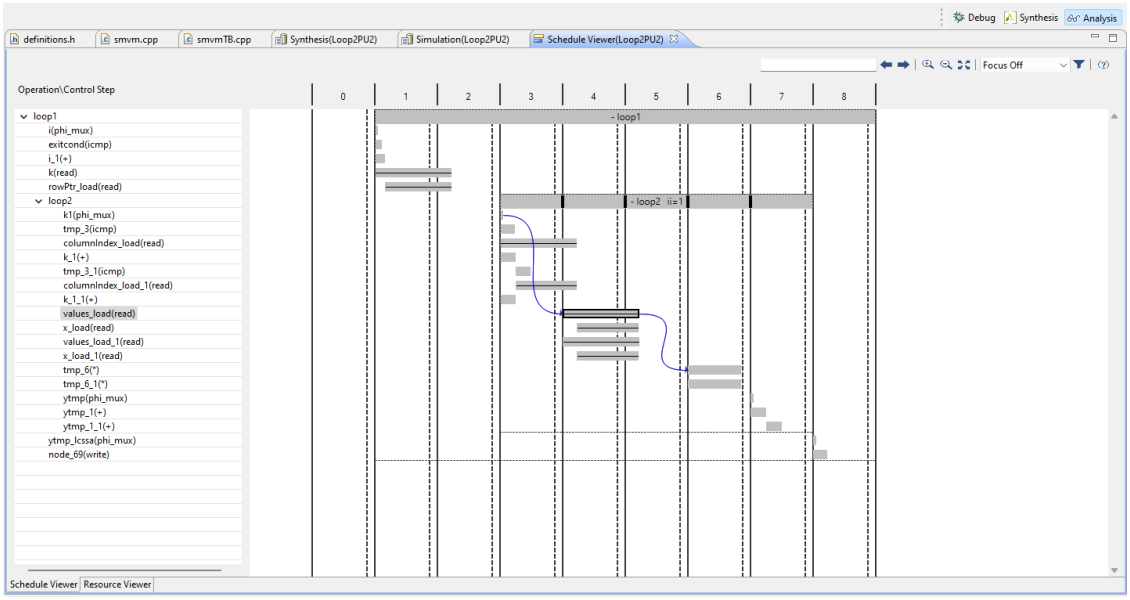

Figure 5: HLS Solution 5 Analysis

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 37: 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 38: HLS Solution 5 C/RTL Cosimulation Summary

Si può notare, rispetto alla soluzione hardware 2, un aumento dell'utilizzazione delle risorse di circa l'82% per quanto riguarda le slice, 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 proprio in seguito all'introduzione della direttiva di unrolling di fattore pari a 2 all'interno del loop2.

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

Table 39: 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 40: 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.

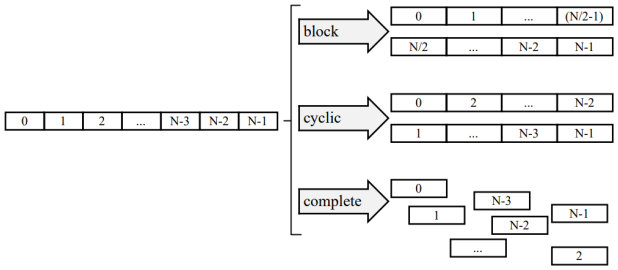

Figure 6: HLS Array Partitioning

Nella soluzione hardware in questione verrà utilizzata la direttiva di 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.

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
      #pragma HLS array_partition variable=columnIndex cyclic factor=2
      #pragma HLS array_partition variable=values cyclic factor=2
      #pragma HLS array_partition variable=x cyclic factor=2
```

```
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
      #pragma HLS array_partition variable=columnIndex cyclic factor=2
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
      #pragma HLS array_partition variable=values cyclic factor=2
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
      #pragma HLS array_partition variable=x cyclic factor=2
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

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 41: 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 42: 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 43: 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 44: HLS Solution 6 Utilization Estimates [#]

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 45: 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 contemporaneamente 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 46: HLS Solution 6 Export RTL Report

5.7 Solution 7

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

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=4
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

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: [SCHED 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 47: HLS Solution 7 Timing Summary (ns)

Latency		Interval	
min	max	min	max
37	45	37	45

Table 48: 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 49: HLS Solution 7 Latency Loops Summary

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

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
```

```
      #pragma HLS unroll factor=4
      #pragma HLS array_partition variable=columnIndex complete
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

Effettuando nuovamente la sintesi, si ottiene il seguente log nella console e i seguenti valori di latenza.
WARNING: [SCHED 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 min	Latency max	Iteration Latency	Initiation Interval achieved	Initiation Interval target	Trip Count
- loop1	32	40	8∼10	-	-	4
+ loop2	4	6	5	2	1	0∼1

Table 50: 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 parallello all'array values. Infatti, anche in questo caso il valore di Initiation Interval 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.

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=4
      #pragma HLS array_partition variable=columnIndex complete
      #pragma HLS array_partition variable=values complete
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

Effettuando nuovamente la sintesi, si ottiene il seguente log nella console e i seguenti valori di latenza.
WARNING: [SCHED 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 min	Latency max	Iteration Latency	Initiation Interval achieved	Initiation Interval target	Trip Count
- loop1	32	40	8∼10	-	-	4
+ loop2	4	6	5	2	1	0∼1

Table 51: 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 Initiation Interval 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.

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=4
      #pragma HLS array_partition variable=columnIndex complete
      #pragma HLS array_partition variable=values complete
      #pragma HLS array_partition variable=x complete
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

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

WARNING: [SCHED 204-21] Estimated clock period (10.208ns) exceeds the target (target clock period: 10ns, clock uncertainty: 1.25ns, effective delay budget: 8.75ns). WARNING: [SCHED 204-21] The critical path in module 'smvm' consists of the following:
'add' operation ('ytmp_1_3', smvmProject/smvm.cpp:34) [139] (2.55 ns)
'phi' operation ('ytmp', smvmProject/smvm.cpp:34) with incoming values : ('ytmp_1_3', smvmProject/smvm.cpp:34) [68] (0 ns)
'add' operation ('ytmp_1', smvmProject/smvm.cpp:34) [105] (2.55 ns)
'add' operation ('ytmp_1_1', smvmProject/smvm.cpp:34) [117] (2.55 ns)
'add' operation ('ytmp_1_2', smvmProject/smvm.cpp:34) [128] (2.55 ns)

Clock	**Target**	**Estimated**	**Uncertainty**
ap_clk	10.00	10.208	1.25

Table 52: HLS Solution 7 with columnIndex, values and x partitioning Timing Summary (ns)

Latency		**Interval**	
min	max	min	max
29	33	29	33

Table 53: HLS Solution 7 with columnIndex, values and x partitioning Latency Summary (clock cycles)

Loop Name	**Latency** min	max	**Iteration Latency**	**Initiation Interval** achieved	target	**Trip Count**
- loop1	28	32	7∼8	-	-	4
+ loop2	3	4	4	1	1	0∼1

Table 54: HLS Solution 7 with columnIndex, values and x partitioning Latency Loops Summary

Si può evidenziare come, in questo caso, il valore di Initiation Interval raggiunto sia il medesimo di quello target, cioè pari a 1. Inoltre, 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 e al 2% per le LUT.

Name	**BRAM_18K**	**DSP48E**	**FF**	**LUT**
DSP	-	-	-	-
Expression	-	12	0	534
FIFO	-	-	-	-
Instance	-	-	-	252
Memory	0	-	-	-
Multiplexer	-	-	-	99
Register	-	-	854	128
Total	0	12	854	1013
Available	280	220	106400	53200
Utilization (%)	0	5	∼0	1

Table 55: 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	29	29	29	NA	NA	NA

Table 56: 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 $8.110ns$. Bisogna notare, però, che l'utilizzazione delle risorse risulta essere notevolmente alta dal momento che il tool è riuscito ad attuare i partizionamenti di fattore pari a 4 su tutti e tre gli array precedentemente citati.

Resource	**VHDL**
SLICE	314
LUT	915
FF	297
DSP	12
BRAM	0
SRL	0

Table 57: 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	8.110

Table 58: HLS Solution 7 with columnIndex, values and x partitioning Export RTL Final Timing

5.8 Solution 8

Nella soluzione hardware in questione verrà utilizzata la direttiva di pipeline, la direttiva di unrolling di fattore pari a 2 e la direttiva di partizionamento di tipologia cyclic. Nello specifico, verranno analizzate le seguenti implementazioni relative al loop2:

- Pipeline, Unroll=2, Cyclic=4 (columnIndex, values, x)
- Pipeline, Unroll=2, Cyclic=4 (columnIndex)
- Pipeline, Unroll=2, Cyclic=4 (values)
- Pipeline, Unroll=2, Cyclic=4 (x)

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

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
      #pragma HLS array_partition variable=columnIndex cyclic factor=4
      #pragma HLS array_partition variable=values cyclic factor=4
      #pragma HLS array_partition variable=x cyclic factor=4
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
      #pragma HLS array_partition variable=columnIndex cyclic factor=4
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
      #pragma HLS array_partition variable=values cyclic factor=4
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
      #pragma HLS array_partition variable=x cyclic factor=4
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

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 59: HLS Solution 8 Timing Summary (ns)

Solution	Latency		Interval	
	min	max	min	max
columnIndex, values, x	29	37	29	37
columnIndex	33	41	33	41
values	33	41	33	41
x	29	37	29	37

Table 60: HLS Solution 8 Latency Summary (clock cycles)

Si può notare come i valori di latenza totale, delle latenze totali dei loop corrispondenti e di Iteration Latency risultano essere differenti tra le varia solution. In particolare, la soluzione che implementa il partizionamento dei tre array (columnIndex, values e x) presenta valori di latenza uguali a quelli della soluzione che implementa il partizionamento dell'array x. Bisogna notare, però, che il valore del trip count risulta essere il medesimo per ogni solution. In particolare, essendo previsto un unrolling di fattore pari a 2 all'interno del loop2, il corrispondente valore risulta essere dimezzato rispetto alla solution 1 in cui non è presente alcun pragma di parallelismo.

Solution	Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count
		min	max		achieved	target	
columnIndex, values, x	- loop1	28	36	7∼9	-	-	4
	+ loop2	3	5	4	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	28	36	7∼9	-	-	4
	+ loop2	3	5	4	1	1	0∼2

Table 61: HLS Solution 8 Latency Loops Summary

Solution	BRAM_18K	DSP48E	FF	LUT
columnIndex, values, x	0	6	630	628
columnIndex	0	6	535	560
values	0	6	634	581
x	0	6	596	441

Table 62: HLS Solution 8 Utilization Estimates [#]

Solution	RTL	Status	Latency			Interval		
			min	avg	max	min	avg	max
columnIndex, values, x	VHDL	Pass	33	33	33	NA	NA	NA
columnIndex	VHDL	Pass	37	37	37	NA	NA	NA
values	VHDL	Pass	37	37	37	NA	NA	NA
x	VHDL	Pass	33	33	33	NA	NA	NA

Table 63: HLS Solution 8 C/RTL Cosimulation Report

Si può notare come, in corrrispondenza della soluzione hardware basata sul partizionamento dei tre array, si ha la maggiore utilizzazione di slice, LUT e FF. Inoltre, si può evidenziare come le soluzioni basate rispettivamente sul partitioning di columnIndex e values risultano avere pressoché la medesima utilizzazione dal momento che le dimensioni di tali strutture dati è la medesima. In particolare, rispetto alla soluzione hardware similare, dove era stato utilizzato un fattore di partitioning pari a 2 di tipologia cyclic (solution 6), si può notare, in corrispondenza del partizionamento dei tre array, un incremento dei FF di circa il 59%, un incremento delle LUT e delle slice di circa il 9%.

Solution	SLICE	LUT	FF	DSP	BRAM	CP required	CP achieved post-synthesis	CP achieved post-implementation
columnIndex, values, x	123	343	315	6	0	10	6.540	6.571
columnIndex	85	261	226	6	0	10	7.496	7.654
values	92	274	196	6	0	10	7.927	7.780
x	104	248	313	6	0	10	6.540	6.844

Table 64: HLS Solution 8 Export RTL Report

5.9 Solution 9

Qui, di seguito, viene riportata l'architettura relativa alla solution 9.

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=8
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

In particolare, rispetto alla soluzione hardware 5 e 7, dove rispettivamente era stato considerato un parallelismo di fattore pari a 2 e un parallelismo di fattore pari a 4, in questa solution è stato considerato un unrolling di fattore pari a 8. In particolare, ciò che ci si aspetta è un aumento delle risorse ed eventuali problematiche relative al timing, similari a quelle riscontrate nella solution 7, 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: [SCHED 204-69] Unable to schedule 'load' operation ('columnIndex_load_6', smvmProject/smvm.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 65: HLS Solution 9 Timing Summary (ns)

Latency		Interval	
min	max	min	max
45	61	45	61

Table 66: HLS Solution 9 Latency Summary (clock cycles)

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count
	min	max		achieved	target	
- loop1	44	60	11∼15	-	-	4
+ loop2	7	11	8	4	1	0∼1

Table 67: HLS Solution 9 Latency Loops Summary

Pertanto, si potrebbe aggiungere una direttiva di partizionamento con fattore pari a 8 relativa all'array menzionato all'interno del log, cioè columnIndex.

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
```

```
      #pragma HLS pipeline
      #pragma HLS unroll factor=8
      #pragma HLS array_partition variable=columnIndex complete
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

Effettuando nuovamente la sintesi, si ottiene il seguente log nella console e i seguenti valori di latenza.
WARNING: [SCHED 204-69] Unable to schedule 'load' operation ('values_load_6', 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	Latency		Iteration Latency	Initiation Interval		Trip Count
Name	min	max		achieved	target	
- loop1	40	56	10∼14	-	-	4
+ loop2	6	10	7	4	1	0∼1

Table 68: HLS Solution 9 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'architetture, gli accessi in parallello all'array values. Infatti, il valore di Initiation Interval 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.

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=8
      #pragma HLS array_partition variable=columnIndex complete
      #pragma HLS array_partition variable=values complete
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

Effettuando nuovamente la sintesi, si ottiene il seguente log nella console e i seguenti valori di latenza.
WARNING: [SCHED 204-69] Unable to schedule 'load' operation ('x_load_6', 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	Latency		Iteration Latency	Initiation Interval		Trip Count
Name	min	max		achieved	target	
- loop1	40	56	10∼14	-	-	4
+ loop2	6	10	7	4	1	0∼1

Table 69: HLS Solution 9 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 Initiation Interval 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.

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=8
      #pragma HLS array_partition variable=columnIndex complete
      #pragma HLS array_partition variable=values complete
      #pragma HLS array_partition variable=x complete
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

Effettuando nuovamente la sintesi, si ottiene il seguente log nella console e il seguente report.
WARNING: [SCHED 204-68] The II Violation in module 'smvm': Unable to enforce a carried dependence constraint (II = 1, distance = 1, offset = 0) between 'add' operation ('ytmp_1_7', smvmProject/smvm.cpp:34) and 'add' operation ('ytmp_1', smvmProject/smvm.cpp:34).
WARNING: [SCHED 204-68] The II Violation in module 'smvm': Unable to enforce a carried dependence constraint (II = 1, distance = 1, offset = 1) between 'add' operation ('ytmp_1_5', smvmProject/smvm.cpp:34) and 'add' operation ('ytmp_1_2', smvmProject/smvm.cpp:34).
WARNING: [SCHED 204-68] The II Violation in module 'smvm': Unable to enforce a carried dependence constraint (II = 2, distance = 1, offset = 1) between 'add' operation ('ytmp_1_6', smvmProject/smvm.cpp:34) and 'add' operation ('ytmp_1_2', smvmProject/smvm.cpp:34).

In particolare, tali warning suggeriscono problematiche relative alla variabile ytmp. Nello specifico, ymtp è una variabile temporanea (di appoggio) di tipo DTYPE, cioè int, e pertanto il partizionamento su tale variabile non avrebbe senso poiché quest'ultimo funziona solo con gli array. Bisogna notare però che il problema non è relativo a ytmp ma è associato a y, cioè l'output, che è un vettore. Inoltre, y fa riferimento al loop1 e questo suggerisce che gli eventuali problemi sono relativi al ciclo1 ed, essendo che il loop1 presenta al suo interno il loop2, allora tali problemi si ripercuotono anche sul loop2. In aggiunta, si evidenzia come in questo caso l'Initiation Interval raggiunto risulta essere decrementato da 4 a 3 ma comunque ancora maggiore di quello target.

Loop	**Latency**		**Iteration Latency**	**Initiation Interval**		**Trip Count**
Name	min	max		achieved	target	
- loop1	32	44	8∼11	-	-	4
+ loop2	4	7	5	3	1	0∼1

Table 70: HLS Solution 9 with columnIndex, values and x partitioning Latency Loops Summary

Un'idea potrebbe quella di effettuare un partizionamento anche sull'array rowPtr.

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
```

```
      #pragma HLS unroll factor=8
      #pragma HLS array_partition variable=columnIndex complete
      #pragma HLS array_partition variable=values complete
      #pragma HLS array_partition variable=x complete
      #pragma HLS array_partition variable=rowPtr complete
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

Effettuando nuovamente la sintesi, si ottiene il seguente log nella console e il seguente report.
WARNING: [SCHED 204-68] The II Violation in module 'smvm': Unable to enforce a carried dependence constraint (II = 1, distance = 1, offset = 0) between 'add' operation ('ytmp_1_7', smvmProject/smvm.cpp:34) and 'add' operation ('ytmp_1', smvmProject/smvm.cpp:34).
WARNING: [SCHED 204-68] The II Violation in module 'smvm': Unable to enforce a carried dependence constraint (II = 1, distance = 1, offset = 1) between 'add' operation ('ytmp_1_5', smvmProject/smvm.cpp:34) and 'add' operation ('ytmp_1_2', smvmProject/smvm.cpp:34).
WARNING: [SCHED 204-68] The II Violation in module 'smvm': Unable to enforce a carried dependence constraint (II = 2, distance = 1, offset = 1) between 'add' operation ('ytmp_1_6', smvmProject/smvm.cpp:34) and 'add' operation ('ytmp_1_2', smvmProject/smvm.cpp:34).
In particolare, si riscontrano i medesimi warning allegati precedentemente.

Loop	**Latency**		**Iteration Latency**	**Initiation Interval**		**Trip Count**
Name	min	max		achieved	target	
- loop1	28	40	7∼10	-	-	4
+ loop2	4	7	5	3	1	0∼1

Table 71: HLS Solution 9 with columnIndex, values, x and rowPtr partitioning Latency Loops Summary

Pertanto, a tale proposito si potrebbe effettuare un partizionamento anche su y.

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=8
      #pragma HLS array_partition variable=columnIndex complete
      #pragma HLS array_partition variable=values complete
      #pragma HLS array_partition variable=x complete
      #pragma HLS array_partition variable=rowPtr complete
      #pragma HLS array_partition variable=y complete
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

Effettuando nuovamente la sintesi, si ottiene il seguente log nella console e il seguente report.
WARNING: [SCHED 204-68] The II Violation in module 'smvm': Unable to enforce a carried dependence constraint (II = 1, distance = 1, offset = 0) between 'add' operation ('ytmp_1_7', smvmProject/smvm.cpp:35) and 'add' operation ('ytmp_1', smvmProject/smvm.cpp:35).
WARNING: [SCHED 204-68] The II Violation in module 'smvm': Unable to enforce a carried dependence constraint (II = 1, distance = 1, offset = 1) between 'add' operation ('ytmp_1_5', smvmProject/smvm.cpp:35) and 'add' operation ('ytmp_1_2', smvmProject/smvm.cpp:35).
WARNING: [SCHED 204-68] The II Violation in module 'smvm': Unable to enforce a carried dependence constraint (II = 2, distance = 1, offset = 1) between 'add' operation ('ytmp_1_6', smvmProject/smvm.cpp:35)

and 'add' operation ('ytmp_1_2', smvmProject/smvm.cpp:35).
Nello specifico, si riscontrano nuovamente i medesimi warning allegati precedentemente.

Loop Name	Latency min	Latency max	Iteration Latency	Initiation Interval achieved	Initiation Interval target	Trip Count
- loop1	28	40	7∼10	-	-	4
+ loop2	4	7	5	3	1	0∼1

Table 72: HLS Solution 9 with columnIndex, values, x, rowPtr and y partitioning Latency Loops Summary

Pertanto, dal momento che i warning sono ancora presenti e che il valore di Initiation Interval non tende a diminuire, si potrebbe provare effettuando anche il pipelining del loop1.

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    #pragma HLS pipeline
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=8
      #pragma HLS array_partition variable=columnIndex complete
      #pragma HLS array_partition variable=values complete
      #pragma HLS array_partition variable=x complete
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

Effettuando nuovamente la sintesi, si ottiene il seguente log nella console e il seguente report.
WARNING: [XFORM 203-503] Ignored partial unroll directive for loop 'loop2' (smvmProject/smvm.cpp:21) because its parent loop or function is pipelined.
WARNING: [XFORM 203-503] Cannot unroll loop 'loop2' (smvmProject/smvm.cpp:21) in function 'smvm' completely: variable loop bound.
WARNING: [SCHED 204-65] Unable to satisfy pipeline directive: Loop contains subloop(s) not being unrolled or flattened.

Loop Name	Latency min	Latency max	Iteration Latency	Initiation Interval achieved	Initiation Interval target	Trip Count
- loop1	16	36	4∼9	-	-	4
+ loop2	0	5	3	1	1	0∼4

Table 73: HLS Solution 9 with columnIndex, values and x partitioning and loop1 pipelined Latency Loops Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	141
FIFO	-	-	-	-
Instance	-	-	-	63
Memory	0	-	-	-
Multiplexer	-	-	-	78
Register	-	-	211	-
Total	0	3	211	282
Available	280	220	106400	53200
Utilization (%)	0	1	∼0	∼0

Table 74: HLS Solution 9 with columnIndex, values and x partitioning and loop1 pipelined Utilization Estimates Summary

Si può notare come i warning relativi a ytmp non sono più presenti all'interno della console e che, inoltre, l'Initiation Interval, relativo al loop2, presenta un valore raggiunto uguale a quello target, cioè pari a 1. In aggiunta, si possono evidenziare ulteriori warning all'interno della console. In particolare, il tool non è riuscito a soddisfare la richiesta di pipeline a causa dei bound non noti e, pertanto, non riuscendo nemmeno a soddisfare la direttiva di unrolling all'interno del loop2. Infatti, si può notare come i valori di Initiation Interval (achieved e target) del loop1 risultano essere non definiti, mentre il trip count relativo relativo al loop2 risulta essere il medesimo di quello della solution2 dove non era presente alcun pragma di parallelismo nel ciclo 2.
Pertanto, considerando l'implementazione finale ottenuta in corrispondenza della solution in questione, si effettua la C/RTL Cosimulation e l'Export RTL e si analizzano, di conseguenza, i report corrispondenti.

RTL	Status	**Latency**			**Interval**		
		min	avg	max	min	avg	max
VHDL	Pass	30	30	30	NA	NA	NA

Table 75: HLS Solution 9 with columnIndex, values and x partitioning and loop1 pipelined C/RTL Cosimulation Summary

Resource	VHDL
SLICE	75
LUT	248
FF	131
DSP	3
BRAM	0
SRL	0

Table 76: HLS Solution 9 with columnIndex, values and x partitioning and loop1 pipelined Export RTL Resource Usage

Timing	VHDL
CP required	10.000
CP achieved post-synthesis	5.745
CP achieved post-implementation	6.120

Table 77: HLS Solution 9 with columnIndex, values and x partitioning and loop1 pipelined Export RTL Final Timing

5.10 Solution 10

Nella soluzione hardware in questione verrà utilizzata la direttiva di pipeline, di unrolling di fattore pari a 2 e la direttiva di partizionamento di tipologia cyclic. Nello specifico, verranno analizzate le seguenti implementazioni relative al loop2:

- Pipeline, Unroll=2, Cyclic=8 (columnIndex, values, x)
- Pipeline, Unroll=2, Cyclic=8 (columnIndex)
- Pipeline, Unroll=2, Cyclic=8 (values)
- Pipeline, Unroll=2, Cyclic=8 (x)

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

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
      #pragma HLS array_partition variable=columnIndex cyclic factor=8
      #pragma HLS array_partition variable=values cyclic factor=8
      #pragma HLS array_partition variable=x cyclic factor=8
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
      #pragma HLS array_partition variable=columnIndex cyclic factor=8
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
      #pragma HLS array_partition variable=values cyclic factor=8
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
      #pragma HLS array_partition variable=x cyclic factor=8
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

Effettuando la sintesi si ottiene il seguente log.

ERROR: [XFORM 203-103] Cannot partition array 'x' (smvmProject/smvm.cpp:11): incorrect partition factor 8.

In particolare, la console sta segnalando che effettivamente non riesce a partizionare l'array x dal momento che la dimensione dell'array risulta essere non compatibile con il fattore di partitioning dichiarato. Infatti, la dimensione di x inizialmente dichiarata in definitions.h è pari a 4 mentre il fattore di partizionamento che si sta utilizzando è pari a 8. A questo proposito si potrebbe modificare il valore di dimensionamento relativo a x all'interno dell'header. In particolare è necessario modificare sia il valore di size sia il valore di rows. Inoltre, bisogna anche modificare i parametri dichiarati all'interno della direttiva trip count dal momento che il numero di righe e, quindi, di iterazioni risulta essere differente.

```
#ifndef DEFINITIONS_H


/**
* Square Matrix Size.
*/
const static int size = 8;

/**
* Number of Non-Zero Elements.
*/
const static int nnz = 9;

/**
* Number of Rows.
*/
const static int rows = 8;

...

#endif
```

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=8 avg=4
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
      #pragma HLS array_partition variable=columnIndex cyclic factor=8
      #pragma HLS array_partition variable=values cyclic factor=8
      #pragma HLS array_partition variable=x cyclic factor=8
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
```

```
  }
}
```

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=8 avg=4
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
      #pragma HLS array_partition variable=columnIndex cyclic factor=8
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=8 avg=4
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
      #pragma HLS array_partition variable=values cyclic factor=8
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=8 avg=4
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
      #pragma HLS array_partition variable=x cyclic factor=8
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

Pertanto, effettuando la sintesi si ottiene 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 78: HLS Solution 10 Timing Summary (ns)

Solution	Latency		Interval	
	min	max	min	max
columnIndex, values, x	57	89	57	89
columnIndex	65	97	65	97
values	65	97	65	97
x	57	89	57	89

Table 79: HLS Solution 10 Latency Summary (clock cycles)

Solution	Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count
		min	max		achieved	target	
columnIndex, values, x	- loop1	56	88	7∼11	-	-	8
	+ loop2	3	7	4	1	1	0∼4
columnIndex	- loop1	64	96	8∼12	-	-	8
	+ loop2	4	8	5	1	1	0∼4
values	- loop1	64	96	8∼12	-	-	8
	+ loop2	4	8	5	1	1	0∼4
x	- loop1	56	88	7∼11	-	-	8
	+ loop2	3	7	4	1	1	0∼4

Table 80: HLS Solution 10 Latency Loops Summary

Solution	BRAM_18K	DSP48E	FF	LUT
columnIndex, values, x	0	6	762	900
columnIndex	0	6	539	752
values	0	6	640	773
x	0	6	727	492

Table 81: HLS Solution 10 Utilization Estimates [#]

In particolare, dall'interfaccia Analysis sotto allegata (corrispondente al partitioning dell'array x), si può notare come l'unrolling di fattore 2 sia stato applicato correttamente dal momento che vengono effettuate 2 operazioni di moltiplicazione e 2 operazioni di somma in un'unica iterazione del loop2. Nello specifico, si può evidenziare come i prodotti richiedano l'utilizzo di 3 DSP ognuno tale da giustificare l'utilizzazione di tali risorse pari a 6 come riportato nel report di sintesi. Inoltre, si può notare come il partitioning di fattore pari a 8 abbia effettuato il partizionamento dell'array x in 8 sub-array. In particolare, ogni sub-array prevede un'utilizzazione di bit pari a 32 e una corrispondente utilizzazione dei FF pari a 32 ognuno. Questo aspetto è di fondamentale importanza poiché dal momento che l'array x presenta dimensione pari a 8 e dal momento che il fattore di partizionamento è pari a 8, questo vuol dire che il tool ha effettuato un partitioning di tipo cyclic corrispondente ad un partitioning di tipo complete, cioè il risultato sono dei sub-array di dimensione pari a 1 (poiché la dimensione di x coincide con il fattore di partizionamento). Tanto è vero che effettuando l'array partitioning di tipologia complete sul solo array x, si avrebbe la stessa utilizzazione di risorse e, soprattutto, l'interfaccia Analysis e Resource Profile corrisponderebbero a quelle sotto allegate. Infatti, bisogna ricordare che l'array partitioning di tipologia complete comporta che l'array venga suddiviso in singoli registri mentre quello di tipologia cyclic comporta che vengano creati blocchi ciclici di uguali dimensioni interlacciando gli elementi dell'array iniziale. Pertanto, ciò che cambia tra le due tipologie è che il secondo vada ad assegnare in questi sub-array in maniera ciclica gli elementi dell'array iniziale in base al fattore scelto. Però, nel caso in cui tale fattore corrisponde alla dimensione dell'array iniziale, nel momento in cui il tool assegna l'ultimo elemento ad un sub-array (dopodiché dovrebbe assegnare il prossimo elemento se presente nell'array, ma non in questo caso, al primo sub-array e così via) terminando la procedura di partitioning e lasciando in ogni sub-array un solo elemento. Questo sostanzialmente corrisponderebbe ad avere 8 registri a 32 bit ognuno come succederebbe nella tipologia complete. Invece, per quanto riguarda gli array columnIndex e values, dal

momento che presentano dimensione differente rispetto al fattore di partizionamento, questo si traduce in differenti scheduling e performance.

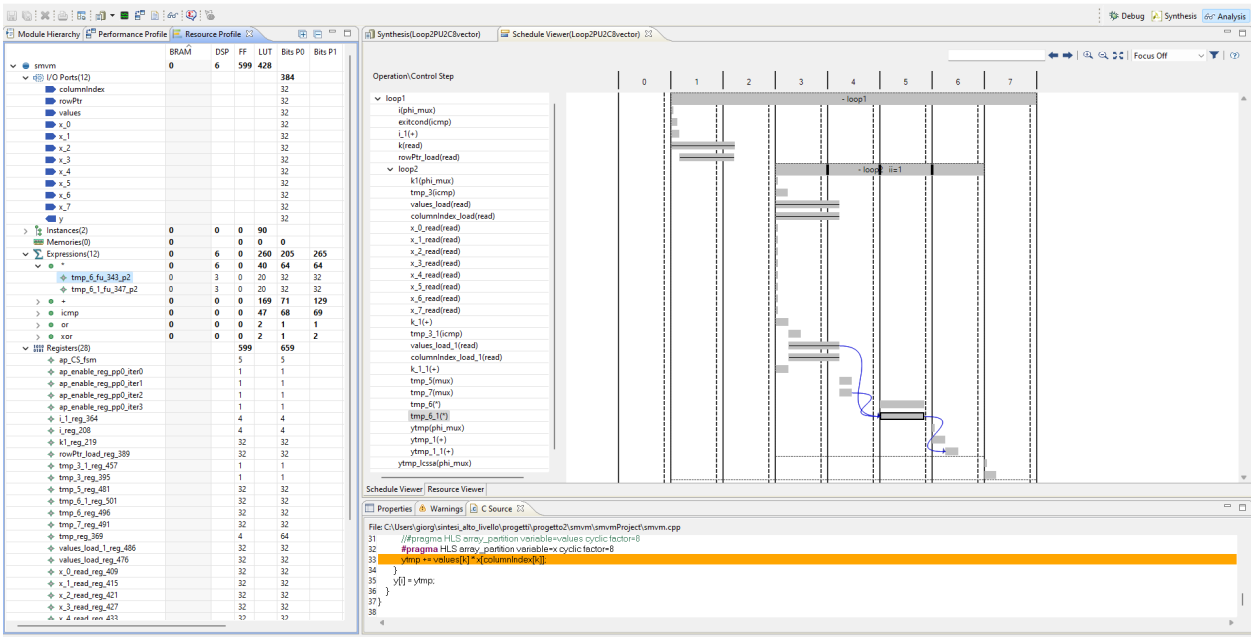

Figure 7: HLS Solution 10 Analysis

Solution	RTL	Status	Latency			Interval		
			min	avg	max	min	avg	max
columnIndex, values, x	VHDL	Pass	61	61	61	NA	NA	NA
columnIndex	VHDL	Pass	69	69	69	NA	NA	NA
values	VHDL	Pass	69	69	69	NA	NA	NA
x	VHDL	Pass	61	61	61	NA	NA	NA

Table 82: HLS Solution 10 C/RTL Cosimulation Report

Si può notare come l'utilizzazione delle risorse sia aumentata rispetto alle soluzioni 6 (loop2 Pipeline, Unroll=2, Cyclic=2) e 8 (loop2 Pipeline, Unroll=2, Cyclic=4). In particolare, considerando le soluzioni hardware corrispondenti al partizionamento di tutti e tre gli array, si evidenzia rispettivamente un aumento di circa l'81% e di circa il 66% in corrispondenza delle slice rispetto alla soluzione 6 e 8, un aumento di circa il 77% e di circa il 63% in corrispondenza delle LUT, un aumento di circa il 127% e di circa il 43%.
Per quanto riguarda le soluzioni hardware corrispondenti al partizionamento di columnIndex, rispetto alla soluzione 6 e 8, si registra rispettivamente un aumento di un'unità e una diminuzione di 2 slice, un aumento di nove unità e di 7 unità delle LUT, un aumento di 6 unità e di 4 unità dei FF.
Per quanto riguarda le solution corrispondenti al partitioning di values, rispetto alla soluzione 6 e 8, si registra rispettivamente un aumento di circa il 18% e di circa il 27% delle slice, un aumento di 15 unità e di circa il 25% delle LUT, una diminuzione di 25 unità e un aumento di 3 unità dei FF.
Per quanto riguarda le soluzioni hardware corrispondenti al partizionamento di x, rispetto alla soluzione 6 e 8, si registra rispettivamente un aumento di circa il 44% e di circa il 38% delle slice, un aumento di circa il 24% e di circa il 25% delle LUT, un aumento di circa il 124% e di circa il 42% dei FF.

Solution	SLICE	LUT	FF	DSP	BRAM	CP required	CP achieved post-synthesis	CP achieved post-implementation
columnIndex, values, x	204	558	449	6	0	10	6.540	6.840
columnIndex	83	268	230	6	0	10	7.496	8.115
values	117	342	199	6	0	10	7.927	7.603
x	143	311	444	6	0	10	6.540	6.790

Table 83: HLS Solution 10 Export RTL Report

Ovviamente questo aumento di utilizzazione delle risorse è dovuto all'incremento del fattore di partizionamento adottato nelle soluzioni hardware. Infatti, si evidenzia che il maggiore incremento delle risorse si ha rispetto alla soluzione 6 dove era previsto un fattore pari a 2 rispetto alla soluzione in questione dove il fattore di partitioning previsto è stato di 8.

5.11 Solution 11

Considerando il codice utilizzato nella precedente solution, nella soluzione hardware in questione verrà utilizzata la direttiva di pipeline, di unrolling di fattore pari a 2 e la direttiva di partizionamento di tipologia block. Nello specifico, verranno analizzate le seguenti implementazioni relative al loop2:

- Pipeline, Unroll=2, Block=8 (columnIndex, values, x)
- Pipeline, Unroll=2, Block=8 (columnIndex)
- Pipeline, Unroll=2, Block=8 (values)
- Pipeline, Unroll=2, Block=8 (x)

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

```
#ifndef DEFINITIONS_H


/**
* Square Matrix Size.
*/
const static int size = 8;

/**
* Number of Non-Zero Elements.
*/
const static int nnz = 9;

/**
* Number of Rows.
*/
const static int rows = 8;

...

#endif
```

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=8 avg=4
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
      #pragma HLS array_partition variable=columnIndex block factor=8
      #pragma HLS array_partition variable=values block factor=8
      #pragma HLS array_partition variable=x block factor=8
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=8 avg=4
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
```

```
      #pragma HLS array_partition variable=columnIndex block factor=8
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=8 avg=4
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
      #pragma HLS array_partition variable=values block factor=8
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

```
#include "definitions.h"

void smvm(int rowPtr[rows+1], int columnIndex[nnz], DTYPE values[nnz], DTYPE y[size], DTYPE
    x[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=rowPtr[i]; k<rowPtr[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=8 avg=4
      #pragma HLS pipeline
      #pragma HLS unroll factor=2
      #pragma HLS array_partition variable=x block factor=8
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

Effettuando la sintesi si ottiene il seguente log.

WARNING: [XFORM 203-105] Cannot partition array 'columnIndex' (smvmProject/smvm.cpp:11): indivisible factor 8 on dimension 1, which has 9 elements.

WARNING: [XFORM 203-105] Cannot partition array 'values' (smvmProject/smvm.cpp:11): indivisible factor 8 on dimension 1, which has 9 elements.

In particolare, la console sta segnalando che il tool non riesce a partizionare correttamente, tramite la tipologia block e il fattore 8 dichiarato, gli array columnIndex e values dal momento che presentano un numero di elementi che risulta essere non multiplo del fattore di array partitioning dichiarato. Pertanto, a tale proposito si modifica, di conseguenza, la variabile nnz, cioè il numero di elementi non nulli all'interno della matrice, nell'header. Nello specifico è stato scelto un valore pari a 16. Pertanto, ciò che ci si aspetta è che il tool divida tali array, columnIndex e values, in 8 sub-array aventi ognuno 2 elementi. Invece, per quanto riguarda l'array x, avente dimensione pari a 8, ci si aspetta che il tool lo divida in 8 sub-array da un elemento ognuno.

```
#ifndef DEFINITIONS_H


/**
* Square Matrix Size.
*/
const static int size = 8;

/**
* Number of Non-Zero Elements.
*/
```

```
12 const static int nnz = 16;
13
14 /**
15 * Number of Rows.
16 */
17 const static int rows = 8;
18
19 ...
20
21 #endif
```

Pertanto, effettuando la sintesi si ottiene 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 84: HLS Solution 11 Timing Summary (ns)

Solution	Latency		Interval	
	min	max	min	max
columnIndex, values, x	57	89	57	89
columnIndex	65	97	65	97
values	65	97	65	97
x	57	89	57	89

Table 85: HLS Solution 11 Latency Summary (clock cycles)

Solution	Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count
		min	max		achieved	target	
columnIndex, values, x	- loop1	56	88	7∼11	-	-	8
	+ loop2	3	7	4	1	1	0∼4
columnIndex	- loop1	64	96	8∼12	-	-	8
	+ loop2	4	8	5	1	1	0∼4
values	- loop1	64	96	8∼12	-	-	8
	+ loop2	4	8	5	1	1	0∼4
x	- loop1	56	88	7∼11	-	-	8
	+ loop2	3	7	4	1	1	0∼4

Table 86: HLS Solution 11 Latency Loops Summary

Solution	BRAM_18K	DSP48E	FF	LUT
columnIndex, values, x	0	6	789	674
columnIndex	0	6	536	503
values	0	6	597	503
x	0	6	727	492

Table 87: HLS Solution 11 Utilization Estimates [#]

Solution	RTL	Status	**Latency**			**Interval**		
			min	avg	max	min	avg	max
columnIndex, values, x	VHDL	Pass	64	64	64	NA	NA	NA
columnIndex	VHDL	Pass	72	72	72	NA	NA	NA
values	VHDL	Pass	72	72	72	NA	NA	NA
x	VHDL	Pass	64	64	64	NA	NA	NA

Table 88: HLS Solution 11 C/RTL Cosimulation Report

Si può notare come, rispetto alla solution 10 dove era stato applicato un partitioning dello stesso fattore ma di tipologia cyclic, l'utilizzazione delle risorse in corrispondenza del solo partizionamento dell'array x non cambia dal momento che è stata solo modificata il valore della variabile nnz, cioè la dimensione degli array columnIndex e values.

Solution	**SLICE**	**LUT**	**FF**	**DSP**	**BRAM**	**CP required**	**CP achieved post-synthesis**	**CP achieved post-implementation**
columnIndex, values, x	179	454	450	6	0	10	6.541	6.677
columnIndex	90	267	227	6	0	10	7.489	7.664
values	122	391	232	6	0	10	7.506	7.768
x	143	311	444	6	0	10	6.540	6.790

Table 89: HLS Solution 11 Export RTL Report

6 Conclusions

Analizzando l'insieme delle soluzioni hardware proposte, è possibile effettuare alcune considerazioni finali riguardo l'utilizzazione delle risorse corrispondente ad ogni solution. Bisogna specificare che nel plot sotto allegato, le solution che fanno riferimento al partizionamento degli array columnIndex, values e x saranno definite, per semplicità, nella seguente maniera:

- (columnIndex, values, x) → a
- columnIndex → b
- values → c
- x → d

Ad esempio, la solution 10 basata sul partitioning di columnIndex, values e x verrà identificata all'interno del plot come "10a" mentre la solution 8 basata sul partitioning di values verrà identificata all'interno del plot come "8c".

Solution	**SLICE**	**LUT**	**FF**	**DSP**	**BRAM**
1	48	93	161	3	0
2	38	115	139	3	0
3	48	94	161	3	0
4	83	186	292	3	0
5	69	184	196	6	0
6					
• columnIndex, values, x	113	316	198	6	0
• columnIndex	84	259	224	6	0
• values	99	327	224	6	0
• x	99	250	198	6	0
7	314	915	297	12	0
8					
• columnIndex, values, x	123	343	315	6	0
• columnIndex	85	261	226	6	0
• values	92	274	196	6	0
• x	104	248	313	6	0
9	75	248	131	3	0
10					
• columnIndex, values, x	204	558	449	6	0
• columnIndex	83	268	230	6	0
• values	117	342	199	6	0
• x	143	311	444	6	0
11					
• columnIndex, values, x	179	454	450	6	0
• columnIndex	90	267	227	6	0
• values	122	391	232	6	0
• x	143	311	444	6	0

Table 90: HLS Conclusions Export RTL Report

In particolare, è possibile notare come la solution 3 prevede la medesima utilizzazione della solution 1. Infatti, come già precedentemente esposto nella sezione corrispondente, tale solution corrisponde alla solution 1 dal momento che la direttiva di pipelining prevista nel loop1 non viene interpretata dal tool a causa dei bound non noti. Pertanto, il tool ignora tale pragma sintetizzando il circuito come se tale direttiva non fosse presente all'interno del loop1.
Inoltre, è possibile evidenziare come nella soluzione hardware 5 è prevista l'utilizzazione di 6 DSP, rispetto

alle solution precedenti ad essa in cui ne erano previste 3. Questo incremento è dovuto al fatto che nella solution 5 è previsto un unrolling di fattore pari a 2. Pertanto, se nelle soluzioni base, cioè dove non era presente alcun parallelismo all'interno del loop2, era prevista un'utilizzazione di 3 DSP, allora nella soluzione 5 ne saranno previste il doppio. Analogamente, nella soluzione hardware 7 è previsto un unrolling di fattore pari a 4. Pertanto, come si può evidenziare dalla tabella sopra allegata, si ha un notevole incremento in corrispondenza delle LUT e, inoltre, l'utilizzazione delle DSP risulta essere pari a 12. Invece, per quanto riguarda la soluzione hardware 9, dove è previsto un unrolling di fattore pari a 8, l'utilizzazione di tali risorse risulta essere minore delle solution precedentemente citate. Questo è dovuto al fatto che non è stato possibile implementare la solution in questione dal momento che il tool non riusciva a garantire gli accessi in parallelo alla memoria. In particolare, dopo aver previsto ulteriori direttive di partizionamento all'interno del loop2, il tool ha evidenziato un'ulteriore problematica dovuta a ytmp che si è cercato di risolvere introducendo un pragma di pipeline all'interno del loop1. Tale ulteriore direttiva ha evidenziato i problemi anche citati nella soluzione 3 dove non è stato possibile attuare il pipelining del loop1 a causa dei bound non noti. Per tale motivo, anche nella solution 9 sono stati riscontrati i medesimi warning che hanno portato il tool a sintetizzare l'architettura senza considerare l'unrolling di fattore pari a 8 nel loop2 e il pipelining del loop1. Nel caso in cui, il tool fosse riuscito a sintetizzare tale solution prevedendo un parallelismo di fattore pari a 8, l'utilizzazione delle DSP, secondo i risultati delle solution basate su unrolling precedentemente citate, sarebbe stata pari a 24.
Inoltre, si può notare come, in corrispondenza della soluzione hardware 10, la quale prevede quattro solution in base al partizionamento effettuato, si ha un incremento dell'utilizzazione delle risorse rispetto alla solution 8 dove l'approccio risulta essere il medesimo tranne che per il fattore di partitioning utilizzato che nel caso delle soluzioni 10 risulta essere il doppio. In particolare, per le soluzioni 10 è stato previsto un valore di rows e size pari al doppio di quelli previsti per le soluzioni 8. Pertanto, il numero di risorse previste risulta essere maggiore. Tanto è vero che, per la solution 10 (columnIndex, values, x) si ha un incremento di FF di circa il 43%, un incremento di LUT di circa il 63% e un incremento di slice di circa il 66%. Per quanto riguarda, invece, il numero di DSP utilizzati, questo risulta essere il medesimo tra le soluzioni 10 e le soluzioni 8 dal momento che in entrambe è previsto un fattore di unrolling pari a 2. Riguardo, invece, le soluzioni 11 nelle quali è stato previsto un partizionamento di tipologia block e di fattore pari a 8, l'utilizzazione delle risorse risulta essere quasi la medesima delle soluzioni 10. In particolare, per l'attuazione di tale direttiva è stata prevista, come precedentemente citato nell'apposita sezione, la variazione del valore della variabile nnz da 8 a 16. Infatti, tale valore nelle soluzioni 10 risulta essere pari a 8 mentre nelle soluzioni 11 risulta essere pari a 16. Tanto è vero che, le variazioni di utilizzazione di risorse tra le due categorie di solution si ha principalmente in corrispondenza dei partizionamenti degli array che prevedono un dimensionamento basato sulla variabile nnz (columnIndex e values). Infatti, in corrispondenza del partizionamento dell'array x, sia nella solution 10 che nella solution 11, l'utilizzazione delle risorse risulta essere la medesima.

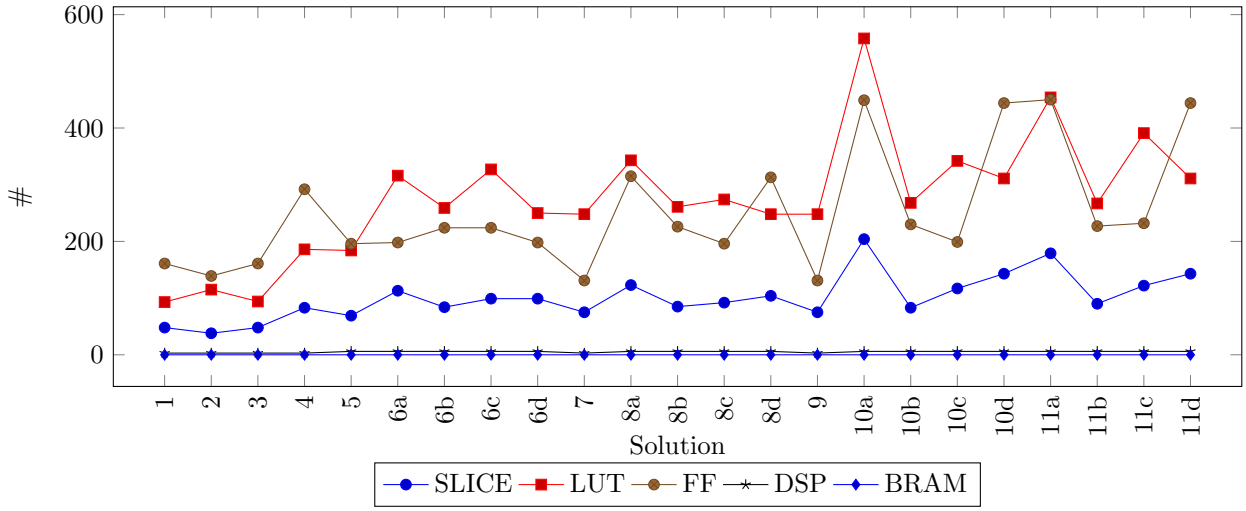

Figure 8: Utilization Export RTL Plot