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

Listings

List of Figures

List of Tables

1 Introduction

1.1 Sparse Matrix

Nell'analisi numerica, una **matrice sparsa** è una matrice in cui la maggior parte degli elementi è pari a zero. Non esiste una definizione rigorosa della proporzione di elementi a valore nullo affinché una matrice possa essere considerata sparsa. Al contrario, se la maggior parte degli elementi è non nulla, allora la matrice è considerata densa.
Una matrice è tipicamente memorizzata come un array bidimensionale. Ogni voce della matrice rappresenta un elemento $a_{i,j}$ della matrice e vi si accede tramite i due indici i e j. Per una matrice m $\times$ n, la quantità di memoria necessaria per memorizzare la matrice in questo formato è proporzionale a m $\times$ n (senza considerare che è necessario memorizzare anche le dimensioni relative alla matrice).
Nel caso di una matrice sparsa, è possibile ridurre notevolmente i requisiti di memoria memorizzando solo le voci non nulle. A seconda del numero e della distribuzione delle voci non nulle, è possibile utilizzare diverse strutture di dati che consentono di ottenere enormi risparmi di memoria rispetto all'approccio di base. Il compromesso è che l'accesso ai singoli elementi diventa più complesso e sono necessarie strutture aggiuntive per poter recuperare la matrice originale senza ambiguità.
I formati possono essere divisi in due gruppi:

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

1.2 Compressed Row Storage (CRS)

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

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

1.3 HLS Partitioning

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

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

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

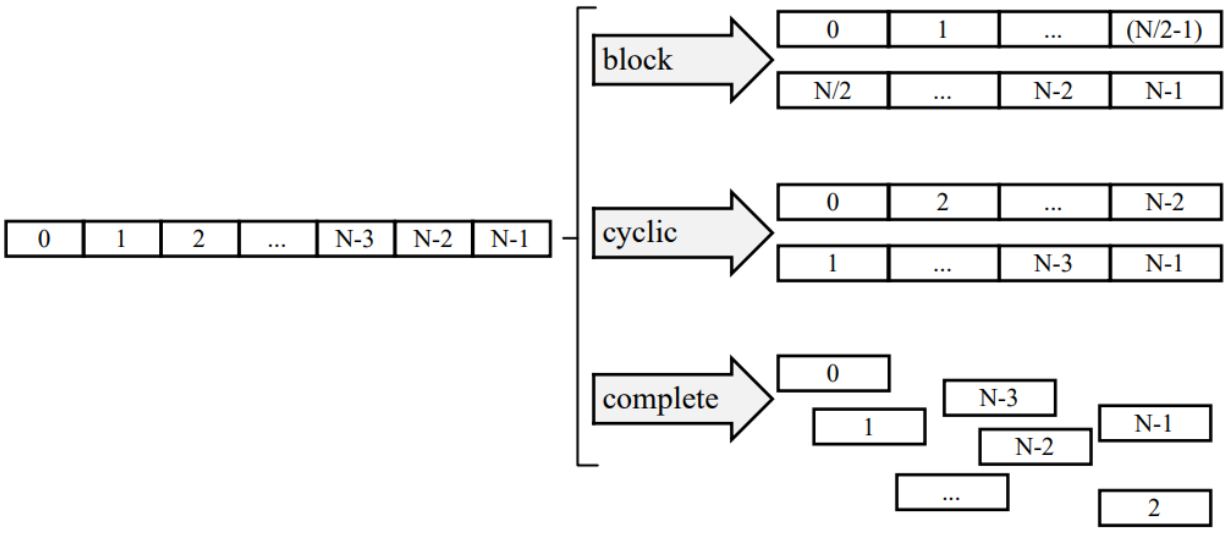

Figure 1: HLS Array Partitioning

2 Tasks to be performed

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

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

Table 1: SMVM Solutions To Be Performed

3 Definitions

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

```
#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.

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

Solution	Loop1	Loop2
1	-	-
2	-	Pipeline
3	Pipeline	-
4	Unroll=2	-
5	-	Pipeline, Unroll=2
6	-	Pipeline, Unroll=2, Cyclic=2
	-	• Pipeline, Unroll=2, Cyclic=2 (columnIndex)
	-	• Pipeline, Unroll=2, Cyclic=2 (values)
	-	• Pipeline, Unroll=2, Cyclic=2 (x)
	-	• Pipeline, Unroll=2, Cyclic=2 (columnIndex, values, x)
7	-	Pipeline, Unroll=4
8	-	Pipeline, Unroll=2, Cyclic=4
	-	• Pipeline, Unroll=2, Cyclic=4 (columnIndex)
	-	• Pipeline, Unroll=2, Cyclic=4 (values)
	-	• Pipeline, Unroll=2, Cyclic=4 (x)
	-	• Pipeline, Unroll=2, Cyclic=4 (columnIndex, values, x)
9	-	Pipeline, Unroll=8
10	-	Pipeline, Unroll=2, Cyclic=8
	-	• Pipeline, Unroll=2, Cyclic=8 (columnIndex)
	-	• Pipeline, Unroll=2, Cyclic=8 (values)
	-	• Pipeline, Unroll=2, Cyclic=8 (x)
	-	• Pipeline, Unroll=2, Cyclic=8 (columnIndex, values, x)
11	-	Pipeline, Unroll=2, Block=8
	-	• Pipeline, Unroll=2, Block=8 (columnIndex)
	-	• Pipeline, Unroll=2, Block=8 (values)
	-	• Pipeline, Unroll=2, Block=8 (x)
	-	• Pipeline, Unroll=2, Block=8 (columnIndex, values, x)

Table 2: SMVM Solutions To Be Performed

5.1 Solution 1

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

```
#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 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** min	max	**Iteration Latency**	**Initiation Interval** achieved	target	**Trip Count**
- loop1	?	?	?	-	-	4
+ loop2	?	?	5	-	-	?

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

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

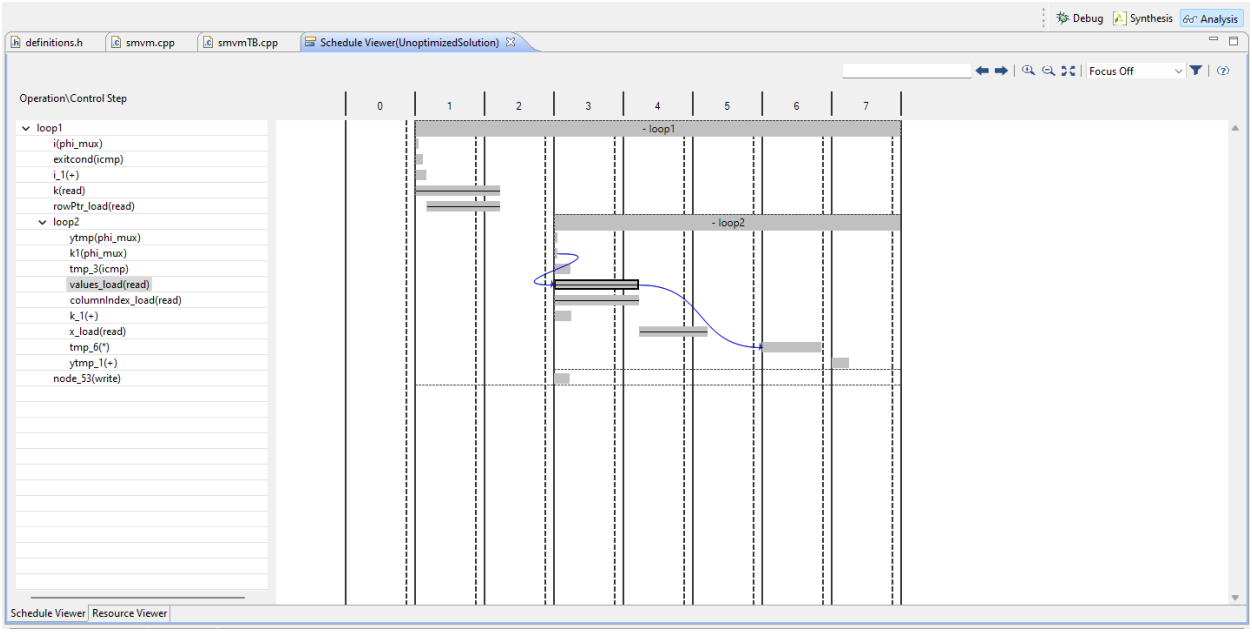

Figure 2: HLS Solution 1 Analysis

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

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

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.

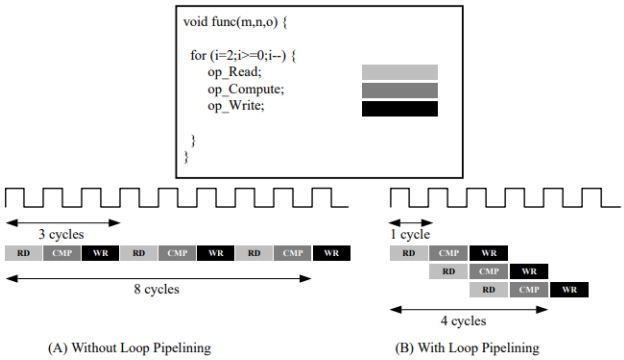

Figure 3: HLS Loop Pipelining

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 caso mai prevedere l'utilizzo di ulteriori direttive.
Qui di seguito, viene allegato l'utilizzazione delle risorse stimata dal processo di sintesi.

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

Table 16: HLS Solution 2 Utilization Estimates Summary

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

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

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

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

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

Table 18: HLS Solution 2t Export RTL Resource Usage

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

Table 19: HLS Solution 2 Export RTL Final Timing

5.3 Solution 3

Qui, di seguito, viene riportata l'architettura relativa alla seconda 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 definita all'interno del loop1, in corrispondenza dell'Initiation Interval di tale ciclo non è definito alcun valore numerico. Tanto è vero che, analizzando i log della sintesi presenti nella console è possibile identificare il seguente warning.

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

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

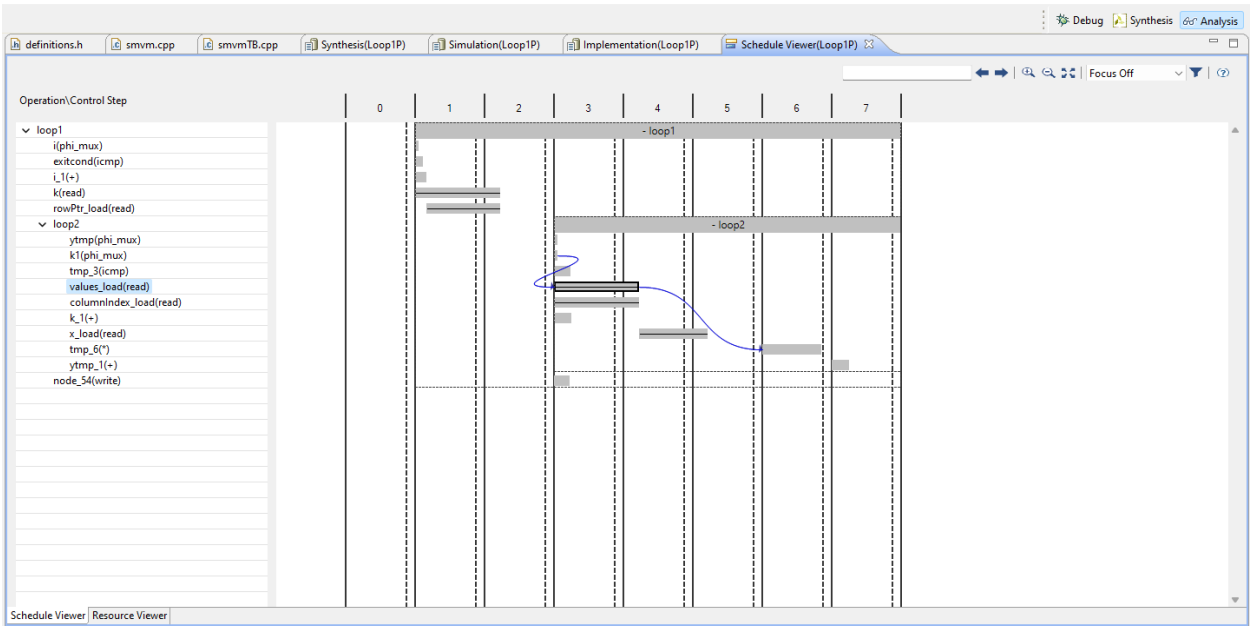

Figure 4: HLS Solution 3 Analysis

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

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

Table 23: HLS Solution 3 Utilization Estimates Summary

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

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

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

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

Table 25: HLS Solution 2t Export RTL Resource Usage

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

Table 26: HLS Solution 2 Export RTL Final Timing

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

```
#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
      #pragma HLS pipeline
      ytmp += values[k] * x[columnIndex[k]];
    }
    y[i] = ytmp;
  }
}
```

Adottando questo approccio, però, la nuova soluzione hardware in questione si ricondurrebbe alla solution 2 precedentemente analizzata dal momento che la direttiva di pipeline nel loop1 verrebbe comunque ignorata e quello che verrebbe effettivamente attuato sarebbe il pragma di pipeline all'interno del loop2. Infatti, effettuando la sintesi si può notare come sia presente il medeesimo warning, alleegato precedentemente, e come sia i valori di latenza sia quelli dell'utilizzazione delle risorse siano i medesimi.
WARNING: [SCHED 204-65] Unable to satisfy pipeline directive: Loop contains subloop(s) not being unrolled or flattened.

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.510	1.25

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

Latency		Interval	
min	max	min	max
17	45	17	45

Table 28: HLS Solution 3 Modified Latency Summary (clock cycles)

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

Table 29: HLS Solution 3 Modified Latency Loops Summary

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

Table 30: HLS Solution 3 Modified Utilization Estimates Summary

5.4 Solution 4

Qui, di seguito, viene riportata l'architettura relativa alla seconda 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 aggiunto la direttiva di unrolling con fattore pari a 2 all'interno del loop1.

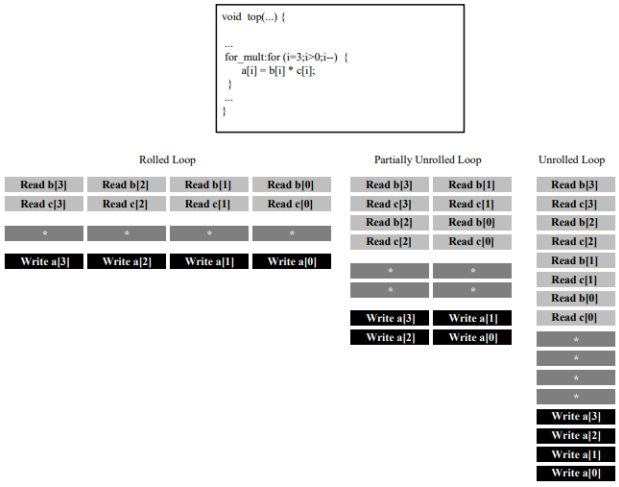

Figure 5: HLS Loop Unrolling

Effettuando la sintesi è possibile evidenziare il seguente report:

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.510	1.25

Table 31: HLS Solution 4 Timing Summary (ns)

Latency		Interval	
min	max	min	max
11	91	11	91

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

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

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

Table 33: HLS Solution 4 Latency Loops Summary

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

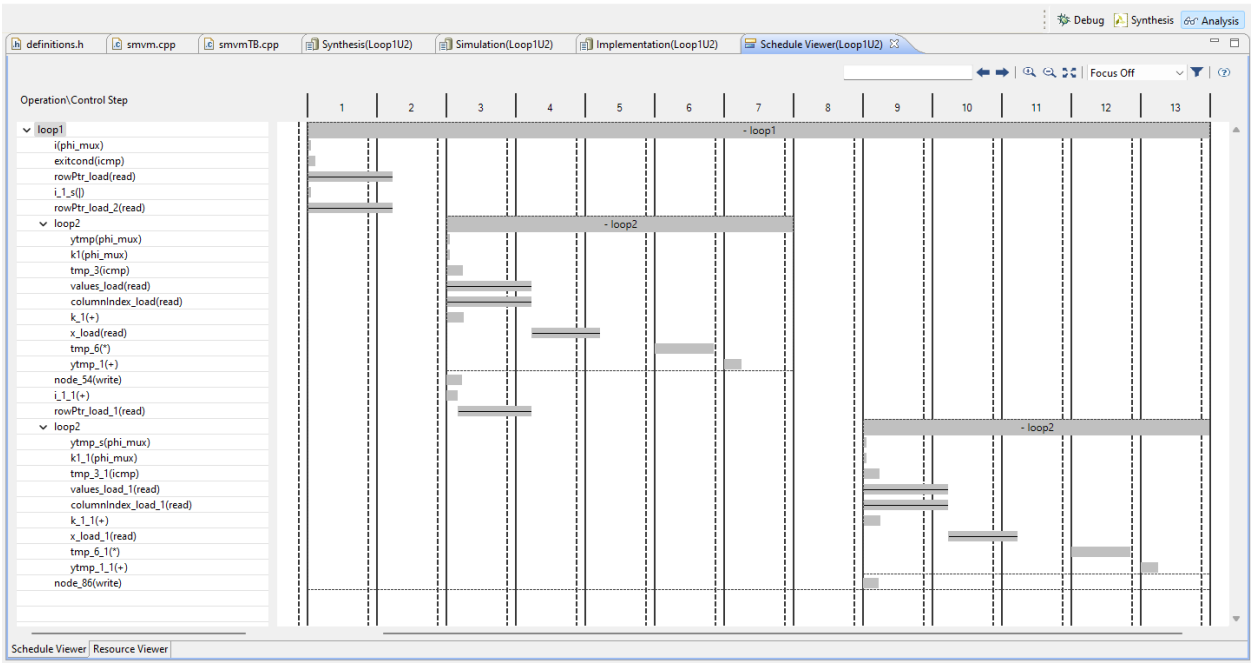

Figure 6: HLS Solution 4 Analysis

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

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

Table 34: HLS Solution 4 Utilization Estimates Summary

RTL	Status	Latency			Interval		
		min	avg	max	min	avg	max
VHDL	Pass	56	56	56	NA	NA	NA

Table 35: HLS Solution 1 with Trip Count C/RTL Cosimulation Summary

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

Table 36: HLS Solution 2t Export RTL Resource Usage

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

Table 37: HLS Solution 2 Export RTL Final Timing

5.5 Solution 5

5.6 Solution 6

5.7 Solution 7

5.8 Solution 8

5.9 Solution 9

5.10 Solution 10

5.11 Solution 11

6 Conclusions