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.

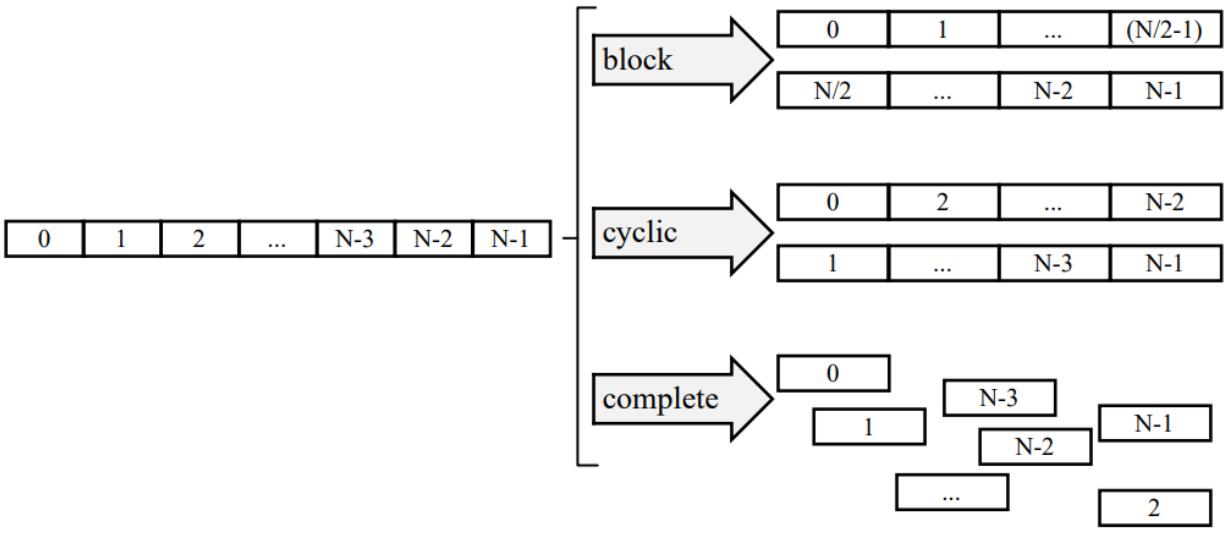

Figure 1: HLS Array Partitioning

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 Unoptimized Solution without Trip Count Latency Summary (clock cycles)

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

```
#include "definitions.h"

void smvm(int iFirstEl[rows+1], int iNonZeroEl[noZeroEl], DTYPE values[noZeroEl], DTYPE
    mulRes[size], DTYPE vector[size]) {
  loop1: for (int i=0; i<rows; i++) {
    DTYPE ytmp = 0;
    loop2: for (int k=iFirstEl[i]; k<iFirstEl[i+1]; k++) {
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      ytmp += values[k] * vector[iNonZeroEl[k]];
    }
    mulRes[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 Unoptimized Solution with Trip Count Latency Summary (clock cycles)

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

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

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	58	58	58

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

Resource	**VHDL**
SLICE	45
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 with Trip Count Export RTL Final Timing

5.2 Solution 2

5.3 Solution 3

5.4 Solution 4

5.5 Solution 5

5.6 Solution 6

5.7 Solution 7

5.8 Solution 8

5.9 Solution 9

5.10 Solution 10

5.11 Solution 11

6 Conclusions