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.
- iFirstEl
  È un array contenente gli indici, relativi all'array values, corrispondenti ai primi elementi non nulli di ogni riga. Nella letteratura questo array è conosciuto anche con la denominazione di *rowPtr*.
- iNonZeroEl
  È un array contenente gli indici di colonna degli elementi non nulli. Nella letteratura questo array è conosciuto anche con la denominazione di *columnIndex*.

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.

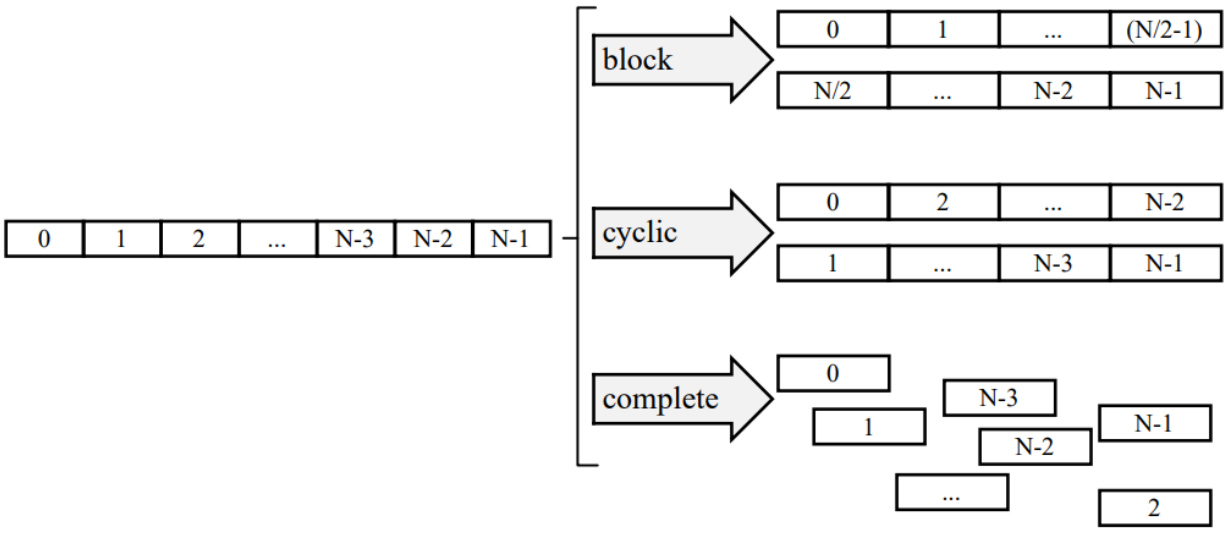

Figure 1: HLS Array Partitioning

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® Vivado®, analizzare le soluzioni proposte nella seguente tabella utilizzando le direttive proprietarie citate e caratterizzando in termini di latenza, dissipazione di potenza 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. Possible values for different implementations: 4, 8, 16.
*/
const static int size = 4;

/**
* Number of Non-Zero Elements. Possible values for different implementations: 9, 13, 16.
*/
const static int noZeroEl  = 9;

/**
* Number of Rows. Possible values for different implementations: 4, 8, 16.
*/
const static int rows = 16;

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

/**
* Standard Matrix Multiplication Design.
* @param matrix[size][size] Input matrix
* @param resMul Multiplication Result
* @param vector Input Vector
*/
void std_multiplication(int iFirstEl[rows+1], int iNonZeroEl[noZeroEl], DTYPE values[
    noZeroEl], DTYPE mulRes[size], DTYPE vector[size]);

/**
* Sparse Matrix Multiplication Design.
* @param iFirstEl[rows+1] Indexes First Elements
* @param iNonZeroEl[noZeroEl] Indexes Non Zero Elements
* @param values[noZeroEl] Input Values
* @param mulRes[size] Multiplication Result
* @param vector[size] Input Vector
*/
void smvm(DTYPE matrix[size][size], DTYPE *mulRes, DTYPE *vector);


#endif
```

Nello specifico, per le variabili *size*, *noZeroEl* e *rows* sono previsti diversi valori possibili in base alla configurazione scelta. Tale configurazione è data dalla diversa implementazione adatta per determinate soluzioni citate successivamente. Ad esempio, per valutare il partitioning di tipologia *cyclic* e di fattore pari a 8, è necessario impostare $size = 8$, $noZeroEl = 13$ e $rows = 8$.

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 $4 * 4$.

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

void std_multiplication(DTYPE matrix[size][size], DTYPE *mulRes, DTYPE *vector);
void smvm(int iFirstEl[rows+1], int iNonZeroEl[noZeroEl], DTYPE values[noZeroEl], DTYPE mulRes[size], DTYPE vector[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 vector[size] = {1, 1, 1, 1};
  DTYPE values[] = {3, 4, 5, 9, 2, 3, 1, 4, 6};
  int iNonZeroEl[] = {0, 1, 1, 2, 0, 2, 3, 1, 3};
  int iFirstEl[] = {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},
    {1, 0, 0, 0, 0, 0, 0, 0},
    {1, 0, 0, 0, 0, 0, 0, 0},
    {1, 0, 0, 0, 0, 0, 0, 0},
    {1, 0, 0, 0, 0, 0, 0, 0}
  };
  DTYPE vector[size] = {1, 1, 1, 1, 1, 1, 1, 1};
  DTYPE values[] = {3, 4, 5, 9, 2, 3, 1, 4, 6, 1, 1, 1, 1};
  int iNonZeroEl[] = {0, 1, 1, 2, 0, 2, 3, 1, 3, 0, 0, 0, 0};
  int iFirstEl[] = {0, 2, 4, 7, 9, 10, 11, 12, 13};
  */
  /*
  DTYPE matrix[size][size] = {
    {3, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 5, 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {2, 0, 3, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 4, 0, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {1, 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, 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, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
  };
  DTYPE vector[size] = {1, 1, 1, 1, 1, 1, 1, 1, 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 iNonZeroEl[] = {0, 1, 1, 2, 0, 2, 3, 1, 3, 0, 0, 0, 0, 0, 0, 0};
  int iFirstEl[] = {0, 2, 4, 7, 9, 10, 11, 12, 13, 14, 15, 16, 16, 16, 16, 16, 16};
  */
```

```
62
63   DTYPE mulResSW[size];
64   std_multiplication(matrix, mulResSW, vector);
65   DTYPE mulRes[size];
66   smvm(iFirstEl, iNonZeroEl, values, mulRes, vector);
67   cout << endl;
68   for(int i=0; i<size; ++i) {
69     cout << "mulResSW=" << mulResSW[i] << ", ";
70     cout << "mulRes=" << mulRes[i] << endl;
71     if(mulResSW[i] != mulRes[i] )
72       fail = 1;
73     if(fail == 1)
74       cout << "i=" << i << " failed." << endl;
75     else
76       cout << "i=" << i << " passed." << endl;
77   }
78   cout << endl;
79
80   return fail;
81 }
```

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

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

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

5 Solutions

Di seguito verranno illustrate e analizzate le soluzioni previste nella tabella sotto allegata. In particolare, per le soluzioni 7, 9 e 11, come verrà spiegato nelle successive sezioni, sono state previste ulteriori implementazioni così da poter risolvere determinati warning sollevati dal tool in questione.

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
	-	Pipeline, Unroll=4
	-	Pipeline, Unroll=4, Cyclic=4
	Pipeline	Pipeline, Unroll=4
8	-	Pipeline, Unroll=2, Cyclic=4
9	-	Pipeline, Unroll=8
	-	Pipeline, Unroll=8
	-	Pipeline, Cyclic=8
	Pipeline	Pipeline, Unroll=8
10	-	Pipeline, Unroll=2, Cyclic=8
11	-	Pipeline, Unroll=2, Block=8
	-	Pipeline, Unroll=2, Block=8
	-	Pipeline, Unroll=2, Block=16

Table 2: SMVM Solutions Summary

Nello specifico, nelle implementazioni dove è previsto l'utilizzo della direttiva di partitioning sono stati considerati tre array (iNonZeroEl, values, vector) 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 di uno dei tre array appena citati.

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 (iNonZeroEl, values, vector)
	-	• Pipeline, Unroll=2, Cyclic=2 (iNonZeroEl)
	-	• Pipeline, Unroll=2, Cyclic=2 (values)
	-	• Pipeline, Unroll=2, Cyclic=2 (vector)
7	**-**	**Pipeline, Unroll=4**
	-	Pipeline, Unroll=4
	-	Pipeline, Unroll=4, Cyclic=4
	-	• Pipeline, Unroll=4, Cyclic=4 (iNonZeroEl, values, vector)
	-	• Pipeline, Unroll=4, Cyclic=4 (iNonZeroEl)
	-	• Pipeline, Unroll=4, Cyclic=4 (values)
	-	• Pipeline, Unroll=4, Cyclic=4 (vector)
	Pipeline	Pipeline, Unroll=4
8	**-**	**Pipeline, Unroll=2, Cyclic=4**
	-	• Pipeline, Unroll=2, Cyclic=4 (iNonZeroEl, values, vector)
	-	• Pipeline, Unroll=2, Cyclic=4 (iNonZeroEl)
	-	• Pipeline, Unroll=2, Cyclic=4 (values)
	-	• Pipeline, Unroll=2, Cyclic=4 (vector)
9	**-**	**Pipeline, Unroll=8**
	-	Pipeline, Unroll=8
	-	Pipeline, Cyclic=8
	-	• Pipeline, Cyclic=8 (iNonZeroEl, values, vector)
	-	• Pipeline, Cyclic=8 (iNonZeroEl)
	-	• Pipeline, Cyclic=8 (values)
	-	• Pipeline, Cyclic=8 (vector)
	-	Pipeline, Cyclic=8 (vector)
10	**-**	**Pipeline, Unroll=2, Cyclic=8**
	-	• Pipeline, Unroll=2, Cyclic=8 (iNonZeroEl, values, vector)
	-	• Pipeline, Unroll=2, Cyclic=8 (iNonZeroEl)
	-	• Pipeline, Unroll=2, Cyclic=8 (values)
	-	• Pipeline, Unroll=2, Cyclic=8 (vector)
11	**-**	**Pipeline, Unroll=2, Block=8**
	-	Pipeline, Unroll=2, Block=8
	-	Pipeline, Unroll=2, Block=16
	-	• Pipeline, Unroll=2, Block=16 (iNonZeroEl, values, vector)
	-	• Pipeline, Unroll=2, Block=16 (iNonZeroEl)
	-	• Pipeline, Unroll=2, Block=16 (values)
	-	• Pipeline, Unroll=2, Block=16 (vector)

Table 3: SMVM Detailed Solutions

Qui, di seguito, viene riportata l'implementazione utilizzata per la realizzazione delle solutions sopra citate. In particolare, vengono commentate le direttive poiché utilizzate solo in corrispondenza delle solution 2-11 mentre bisogna precisare che la direttiva *trip count* (all'interno dell'architettura vengono specificate tre pragma *trip count* poiché dipende dalla configurazione scelta) dal momento che è presente in tutte le soluzioni hardware.

```
1 #include "definitions.h"
2
```

```
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++) {
    //#pragma HLS pipeline
    //#pragma HLS unroll factor=2
    DTYPE ytmp = 0;
    loop2: for (int k=iFirstEl[i]; k<iFirstEl[i+1]; k++) {
      //#pragma HLS pipeline
      //#pragma HLS unroll factor=2
      //#pragma HLS loop_tripcount min=0 max=4 avg=2
      //#pragma HLS loop_tripcount min=0 max=8 avg=4
      //#pragma HLS loop_tripcount min=0 max=16 avg=8
      //#pragma HLS array_partition variable=iNonZeroEl block factor=16
      //#pragma HLS array_partition variable=values block factor=16
      //#pragma HLS array_partition variable=vector block factor=16
      ytmp += values[k] * vector[iNonZeroEl[k]];
    }
    mulRes[i] = ytmp;
  }
}
```

Nello specifico, vengono riportate qui di seguito le possibili configurazioni e le corrispondenti direttive *trip count*.

- size=4, noZeroEl=9, rows=4
  #pragma HLS loop_tripcount min=0 max=4 avg=2
- size=8, noZeroEl=13, rows=8
  #pragma HLS loop_tripcount min=0 max=8 avg=4
- size=16, noZeroEl=16, rows=16
  #pragma HLS loop_tripcount min=0 max=16 avg=8

5.1 Solution 1

Qui, di seguito, viene riportata l'architettura relativa alla prima solution. Bisogna precisare che, solo per questa soluzione hardware, per semplicità è stata considerata la prima configurazione, cioè size=4, noZeroEl=9, rows=4.

```
#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++) {
    //#pragma HLS pipeline
    //#pragma HLS unroll factor=2
    DTYPE ytmp = 0;
    loop2: for (int k=iFirstEl[i]; k<iFirstEl[i+1]; k++) {
      //#pragma HLS pipeline
      //#pragma HLS unroll factor=2
      #pragma HLS loop_tripcount min=0 max=4 avg=2
      //#pragma HLS loop_tripcount min=0 max=8 avg=4
      //#pragma HLS loop_tripcount min=0 max=16 avg=8
      //#pragma HLS array_partition variable=iNonZeroEl block factor=16
      //#pragma HLS array_partition variable=values block factor=16
      //#pragma HLS array_partition variable=vector block factor=16
      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 4: HLS Solution 1 Timing Summary (ns)

Latency		Interval	
min	max	min	max
13	93	13	93

Table 5: HLS Unoptimized Solution 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 6: HLS Solution 1 Latency 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 7: HLS Solution 1 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 8: HLS Solution 1 C/RTL Cosimulation Summary

Resource	VHDL
SLICE	45
LUT	93
FF	161
DSP	3
BRAM	0
SRL	0

Table 9: HLS Solution 1 Export RTL Resource Usage

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

Table 10: HLS Solution 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