Indian Institute of technology Palakkad
lab-10 REPORT

Professor : Dr.Satyajit Das
Subject : SoC Design Lab
Course Code : CS5102
Submitted by : Subham Ball
Roll No. - 152202017 (Mtech SoCD)

---

1 Abstract

This report discusses the use of Vivado HLS for optimizing floating-point algo- rithms in C/C++ code to achieve cost, performance, and power benefits while implementing matrix multiplication on 32x32 matrices. We explore the Xil- inx PL design flow aspects of compiling and optimizing the C/C++ design into a high-performance hardware accelerator, generating an AXI4-Stream us- ing Vivado IP Integrator, connecting the hardware accelerator to an AXI DMA peripheral in the AP SoC PL and to the ACP in the AP SoC PS, and writing software that manages the peripherals and measures system-level performance. The Xilinx Vivado HLS tool provides an easy-to-use interface for specifying floating-point algorithms, and the Xilinx PL design flow provides a powerful and flexible platform for implementing high-performance floating-point algorithms on traditional microprocessors. This report aims to provide a comprehensive understanding of the Xilinx PL design flow for matrix multiplication and its implementation using Vivado HLS.

2 Introduction

Matrix multiplication is a fundamental operation in applied mathematics and is widely used in various fields, including signal processing, data analysis, and machine learning. The operation involves multiplying two matrices to produce a third matrix. Due to its computational complexity, efficient implementations of matrix multiplication are essential for

achieving high performance in applica- tions that rely on this operation.One way to optimize matrix multiplication is to use hardware accelerators, which can perform the operation much faster than traditional microprocessors. Vivado HLS is a high-level synthesis tool that can

be used to specify and optimize floating-point algorithms in C/C++ code for im- plementation on hardware accelerators. This approach provides designers with the ability to achieve cost, performance, and power benefits while implementing their algorithms on traditional microprocessors.In this report, we explore the Xilinx PL design flow for implementing matrix multiplication on 32x32 matri- ces using Vivado HLS. We start by discussing the benefits of using Vivado HLS and the Xilinx PL design flow for developing high-performance floating-point algorithms. We then present the steps involved in compiling and optimizing the C/C++ code, generating an AXI4-Stream using Vivado IP Integrator, and connecting the hardware accelerator to peripherals in the AP SoC PL and ACP in the AP SoC PS.The report concludes by presenting the results of our ex- periments, which demonstrate the performance benefits of using the Xilinx PL design flow and Vivado HLS for implementing matrix multiplication. Overall, this report aims to provide designers with a comprehensive understanding of the Xilinx PL design flow for matrix multiplication and its implementation using Vivado HLS.

3 implementation

The code is given below

```
    # include  < stdio
.h  > # include <
stdlib .h >
# include  " mmult
. h " # define
MCR_SIZE 1024
void standalone_mmult ( float A [32][32] , float
B [32][32] , float C [32][32])
{
mmult_hw < float , 32 >( A , B , C );
}
// void HLS_accel ( AXI_VAL in_stream [2* MCR_SIZE ] ,
AXI_VAL out_stream [ MCR_SIZE ])
void HLS_accel ( AXI_VAL INPUT_STREAM [2*
MCR_SIZE ] , AXI_VAL OUTPUT_STREAM [
MCR_SIZE ])
{
# pragma HLS INTERFACE s_axilite port = return
```

```
bundle = CONTROL_BUS
# pragma HLS INTERFACE axis
port = OUTPUT_STREAM
```

```
# pragma HLS INTERFACE axis
port = INPUT_STREAM
// HLS DEPRECATED MODE
// // Map ports to Vivado HLS interfaces
// # pragma HLS INTERFACE ap_fifo port = in_stream
// # pragma HLS INTERFACE ap_fifo port = out_stream
// // Map HLS ports to AXI interfaces
/
# pragma HLS RESOURCE variable = in_stream core = AXIS
metadata = " - bu
/
# pragma HLS RESOURCE variable = out_stream core = AXIS
metadata = " - b
//
# pragma HLS RESOURCE variable = return core = AXI4LiteS
metadata =" - b
wrapped_mmult_hw < float , 32 , 32*32 , 4
, 5 , 5 >( INPUT_STREAM , OUTPUT_STREAM
);
return ;
}
```

Thes testbecnh is given below

```
    # include < stdio
.h > # include <
stdlib .h >
# include " mmult . h "
typedef float T
; int const DIM
= 32;
int const SIZE = DIM * DIM ;
void mmult_sw ( T a [ DIM ][ DIM ] , T b [ DIM ][ DIM ] , T
out [ DIM ][
{
// matrix multiplication of a A * B matrix
for ( int ia = 0; ia < DIM ; +
+ ia ) for ( int ib = 0; ib <
DIM ; ++ ib )
{
float sum = 0;
for ( int id = 0; id < DIM ;
++ id ) sum += a [ ia ][ id
] * b [ id ][ ib ]; out [ ia ][
```

```
ib ] = sum ;
}
}
# ifdef DB_DEBUG
int main ( void )
```

```
{
int ret_val = 0;
ret_val = test_matrix_mult <T , DIM , SIZE ,
4 ,5 ,5 >(); return ret_val ;
}
# else
int main ( void )
{
int ret_val
= 0; int i ,j
, err ;
T matOp1 [ DIM ][
DIM ]; T matOp2 [
DIM ][ DIM ];
T matMult_sw [ DIM ][
DIM ]; T matMult_hw [
DIM ][ DIM ];
/* * Matrix Initiation
*/ for ( i = 0; i <
DIM ; i ++) for ( j =
0; j < DIM ; j ++)
matOp1 [ i ][ j ] = ( float )( i
+ j ); for ( i = 0; i < DIM ; i
++)
for ( j = 0; j < DIM ; j ++)
matOp2 [ i ][ j ] = ( float )( i * j );
/* * End of Initiation */
printf ( " NORMAL  MODE \ r \ n " );
standalone_mmult ( matOp1 , matOp2 ,
matMult_hw );
/* reference Matrix Multiplication */
mmult_sw ( matOp1 , matOp2 , matMult_sw );
/* * Matrix comparison */
err = 0;
for ( i = 0; (i < DIM && ! err
); i ++) for ( j = 0; (j < DIM
&& ! err ); j ++)
if ( matMult_sw [ i ][ j ] != matMult_hw
[ i ][ j ]) err ++;
if ( err == 0)
printf ( " Matrixes  identical ...  Test
successful !\ r \ n " ); else
printf ( " Testfailed !\ r \ n
```

```
" ); return err ;
}
```

After creating an ip now we export this this ip to vivaod we integrate it with pro- cessing system the block diagram is given below. Now generate the bitstream

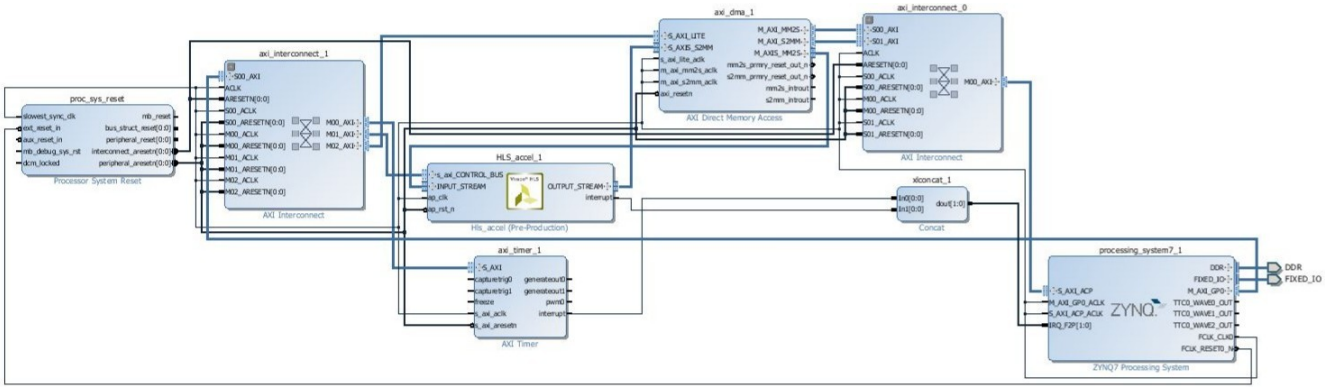

Figure 1: Block diagram

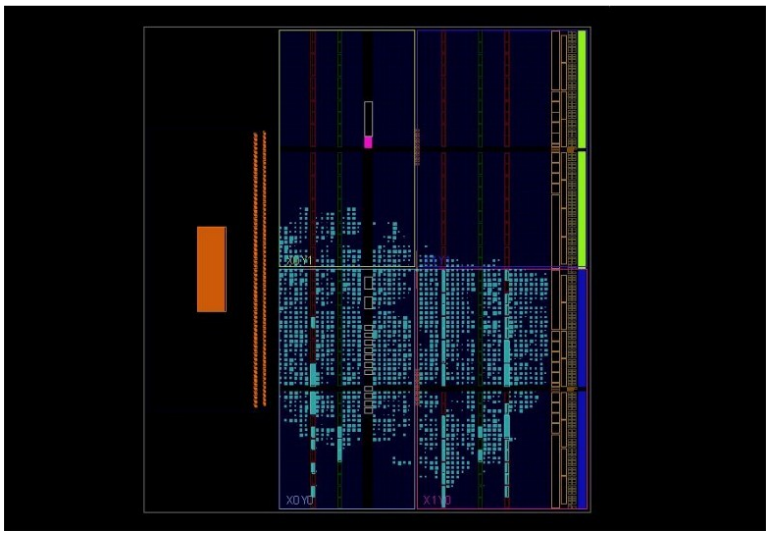

Figure 2: utilization

and export the hardware and launch the sdk too the code is given below

```
# include < stdio
.h > # include < stdlib
```

```
.h >
```

```
# include " platform . h "
# include " xparameters
. h " # include "
xtmrctr . h "
# include " xaxidma . h "
# include " lib_xmmult_hw
. h " # define NUM_TESTS
1024
# define DIM
32
# define SIZE
( DIM * DIM )
// # define X P A R _ A XI _ T I ME R _ DE V I
C E_ I D # define X P A R _ A X I _T I M E R_
D E V IC E _ I D
( X P A R _ A X I _ T I M E R _ 1 _ D E V I C E _ I D ) //
Vivado 2014.4
( X P A R _ A X I _ T I M E R _ 1 _ D E V I C E _ I D ) //
Vivado 2015.2.
// TIMER Instance
XTmrCtr timer_dev ;
// AXI DMA Instance
XAxiDma AxiDma ;
// void print ( char *
str ); int init_dma (){
XAxiDma_Config * CfgPtr
;
int status ;
CfgPtr = XAxiDma_LookupConfig ( ( X PA
R _ A XI _ D M A_ 1 _ D EV I C E_ I D ) );
if (! CfgPtr ){
print ( " Error looking for AXI DMA
config \ n \ r " ); return XST_FAILURE ;
}
status = XA xiDm a_Cfg Init ializ e (& AxiDma
, CfgPtr ); if ( status != XST_SUCCESS ){
print ( " Error initializing
DMA \ n \ r " ); return
XST_FAILURE ;
}
// check for scatter gather mode
if ( XAxiDma_HasSg (& AxiDma )){
print ( " Error DMA configured in SG
```

```
                mode \n \r "); return XST_FAILURE
;
}
/* Disable interrupts , we use polling mode */
```

```
X Ax i D ma _ IntrDisable (& AxiDma , XAXIDMA_IRQ_ALL_MASK
, X AXIDM A_DE X Ax i D ma _ IntrDisable (& AxiDma ,
XAXIDMA_IRQ_ALL_MASK , X AXIDM A_DM
// Reset DMA
XAxiDma_Reset (& AxiDma );
while (! XAxiDma_ResetIsDone (&
AxiDma )) {} return XST_SUCCESS
;
}
int main ( int argc , char ** argv )
{
int i , j ,
k ; int err
=0;
int status ;
int num_of_trials
= 1; float A [ DIM
][ DIM ]; float B [
DIM ][ DIM ];
float res_hw [ DIM ][
DIM ]; float res_sw [
DIM ][ DIM ];
unsigned int dma_size = SIZE * sizeof
( float ); float acc_factor ;
unsigned int init_time , curr_time ,
calibration ; unsigned int begin_time ;
unsigned int end_time ;
unsigned int
run_time_sw = 0;
unsigned int
run_time_hw = 0;
init_platform ();
if ( argc >= 2)
{
num_of_trials = atoi ( argv [1]);
}
xil_printf ( " \r ********************* * * * *
* * xil_printf ( " \r FP 32 x32 MATRIX MULT
->
AX DMA AR ACP
M r " );
n \
xil_printf ( " \r XAPP1170
1
```

```
        redesigned with Vivado      +
        HLS      +  IP   Integrator
        2015.2.1
\n\r " );
xil_printf ( " \_r * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
//
//
for ( i = 0; i <10; i ++)
print ("\ nHello World \ n \ r ");
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *
// Init DMA
status = init_dma ();
if ( status != XST_SUCCESS ){
print ( " \ rError : DMA init failed
\ n " ); return XST_FAILURE ;
}
print ( " \ r \ nDMA Init done \ n \ r " );
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * *
// Setup HW timer
status = XTmrCtr_Initialize (& timer_dev , X PA R _ A XI
_ T I ME R _ DE if ( status != XST_SUCCESS )
{
print ( " \ rError : timer setup failed \ n " );
// return XST_FAILURE ;
}
X Tmr Ct r_ SetOptions (& timer_dev ,
XPAR_AXI_TIMER_DEVICE_ID , XTC_ ENA
// Calibrate HW timer
XTmrCtr_Reset (& timer_dev , X P A R_ A X I _T I M E R_ D
E V IC E _ I D init_time = XTmrCtr_GetValue (& timer_dev ,
X P A R_ A X I _T I M E R_ D curr_time = XTmrCtr_GetValue
(& timer_dev , X P A R_ A X I _T I M E R_ D calibration =
curr_time - init_time ;
// Loop measurement
XTmrCtr_Reset (& timer_dev , X P A R_ A X I _T I M E R_ D
E V IC E _ I D begin_time = XTmrCtr_GetValue (& timer_dev ,
X P A R_ A X I _T I M E R_ D for ( i = 0; i < NUM_TESTS ;
i ++);
end_time = XTmrCtr_GetValue (& timer_dev , X P A R _A X I _
T I M E R _D E
run_time_sw = end_time - - calibration
begin_time ;
xil_printf ( " \ rLoop time for % d iterations is % d cycles
fo ( i = i < DIM i ++)
r 0; ;
```

```
fo ( j =  j < DIM  j ++)
r      O;          ;
{
A  i ][ j =( floa )( i + j
[      ]           t );
```

```
B [ i ][ j ] = ( float )( i * j );
}
/* * End of Initiation */
for ( k =0; k < num_of_trials ; k ++)
{
// call the software version of the function
xil_printf ( " \ rRunning _ Matrix Mult in SW \ n " );
XTmrCtr_Reset (& timer_dev , X P A R_ A X I _T I M E R_ D
E V IC E _ I D begin_time = XTmrCtr_GetValue (& timer_dev ,
X P A R_ A X I _T I M E R_ D for ( i = 0; i < NUM_TESTS ;
i ++)
{
matrix_multiply_ref (A , B , res_sw );
}
end_time = XTmrCtr_GetValue (& timer_dev , X P A R _A X I _
TI M E R_ D E run_time_sw = end_time - begin_time -
calibration ;
xil_printf ( " \ r \ nTotal _ run _ time for_ SW_ on _
Processor is run_time_sw /
NUM_TESTS , NUM_TESTS );
\
// call the HW accelerator
XTmrCtr_Reset (& timer_dev , X P A R_ A X I _T I M E R_ D
E V IC E _ I D begin_time = XTmrCtr_GetValue (& timer_dev ,
X P A R_ A X I _T I M E R_ D
// Setup the HW Accelerator
Setup_HW_Accelerator (A , B , res_hw ,
dma_size ); for ( i = 0; i < NUM_TESTS
; i ++) {
Start_HW_Accelerator ();
Run_HW_Accelerator (A , B , res_hw , dma_size );
}
end_time = XTmrCtr_GetValue (& timer_dev , X P A R _A X I _
TI M E R_ D E run_time_hw = end_time - begin_time -
calibration ;
xil_printf (
" \ rTotal run _ time_ for _ AXI _ DMA_ + _ HW_ accelerator_ is _ %
d run_time_hw / NUM_TESTS , NUM_TESTS );

// Compare the results from sw and hw
for ( i = 0; i < DIM
; i ++) for ( j = 0;
j < DIM ; j ++)
```

```
if ( res_sw [ i ][ j ] != res_hw [ i
][ j ]) { err = 1;
}
```

```
// HW vs . SW speedup factor
acc_factor = ( float ) run_time_sw / ( float ) run_time_hw ;
xil_printf ( " \ r \033[1 mAcceleration _ factor : % d .%
d                                    \033[0 m \ ( int )
acc_factor , ( int ) ( acc_factor * 1000) % 1000);
}
if ( err == 0)
print ( " \ rSWand _ HW_ results_ match !\ n \ r "
); else
print ( " \ rERROR :      _ results
                    mismatch \ n \ r " );
cleanup_platform ();
return 0;
}
```

4 Conclusion

The Xilinx FPGAs provide designers with a versatile and efficient platform for implementing floating-point designs in C/C++. The parallel performance, low power consumption, and embedded CPUs of Xilinx FPGAs make them an ideal solution for optimizing floating-point designs. The comprehensive toolchain available for C/C++ flows allows for performance trade-offs and in-depth anal- ysis throughout the design process.To demonstrate the power of Xilinx FPGAs, we optimized a 32x32 matrix multiplication core for 32-bit floating-point pre- cision using the Vivado HLS tool. The C/C++ code for floating-point matrix multiplication was efficiently transformed into an RTL design using Vivado HLS and exported as an IP core. The core was connected to the Zynq-7000 AP SoC PS through a DMA core in the PL subsystem of the Zynq-7000 device via an AXI4-Stream interface to the ACP.The hardware peripheral matrix multiplier running at a 100 MHz clock frequency computed almost five times faster than its software counterpart executed on the ARM CPU at a 666 MHz clock fre- quency. This demonstrates the superior performance of hardware acceleration over traditional microprocessors for complex computations such as matrix multiplication.hhjm The entire design process presented in this report can be fully automated using the advanced system design flow known as SDSoC. In con- clusion, Xilinx FPGAs offer designers a powerful, versatile, and cost-effective platform for implementing floating-point designs, and the Vivado

HLS tool pro- vides an efficient and streamlined process for transforming C/C++ code into hardware accelerators.