Floating-Point Bitwidth Analysis via Automatic Differentiation

Altaf Abdul Gaffar[1], Oskar Mencer[2], Wayne Luk[1], Peter Y.K. Cheung[3] and Nabeel Shirazi[4]

[1] Department of Computing, Imperial College, London SW7 2BZ, UK.

[2] Lucent, Bell Labs, Murray Hill, NJ 07974, USA.

[3] Department of EEE, Imperial College, London SW7 2BT, UK.

[4] Xilinx Inc., 2100 Logic Drive, San Jose, USA.

Abstract

Automatic bitwidth analysis is a key ingredient for high-level programming of FPGAs and high-level synthesis of VLSI circuits. The objective is to find the minimal number of bits to represent a value in order to minimize the circuit area and to improve efficiency of the respective arithmetic operations, while satisfying user-defined numerical constraints. We present a novel approach to bitwidth – or precision – analysis for floating-point designs. The approach involves analysing the dataflow graph representation of a design to see how sensitive the output of a node is to changes in the outputs of other nodes: higher sensitivity requires higher precision and hence more output bits. We automate such sensitivity analysis by a mathematical method called automatic differentiation, which involves differentiating variables in a design with respect to other variables. We illustrate our approach by optimising the bitwidth for two examples, a Discrete Fourier Transform implementation and a Finite Impulse Response filter implementation.

1. Introduction

FPGAs are starting to provide sufficient area to implement floating-point computations. The large size of floating-point arithmetic units, still remains the main limitation on floating-point computations on FPGAs. One way to deal with this difficulty is to minimize the number of bits in the operands, which in turn minimizes the area and possibly latency of the arithmetic operation.

Floating-point numbers consist of a fixed point mantissa (m) and an integer exponent (e) representing a number $m \cdot 2^e$. As a consequence, the number of bits for the exponent represents the range of possible values, while the number of bits in the mantissa determines the available precision for a particular variable.

We can split the problem of minimizing the bits in the operands into two parts: (1) range analysis, and (2) precision analysis. Range analysis has received much attention within recent integer bitwidth analysis work [2], [9], [11]. Precision analysis is a separate problem. In precision analysis, we are interested in the "sensitivity" of the output of a computation to a slight change to the inputs, or more specifically, the sensitivity of an output to the precision within an arithmetic unit. So far research into precision analysis has mainly focused on fixed point implementations [3], [4], [5], [8], [10].

The most straight-forward method for minimizing the number of bits is to try out various bitwidths and observe the output for each configuration space [6]. This technique, however involves an enormous search space. In this work we focus on a more scalable method that dynamically computes the derivatives of the computed function based on a method known as automatic differentiation. Automatic differentiation is well-known within the optimization community; it enables the computation of all derivatives of the functions in a program. Since our initial evaluation reveals that available automatic differentiation packages are very powerful but are too slow for our purposes, we implement our own version of automatic differentiation which is fully specialised to the task of precision analysis for floating-point computations.

The remainder of the paper is organized as follows. In Section 2 we explain the mathematical foundation of sensitivity analysis via differentiation, and the connection of sensitivity analysis to the minimal bitwidth of the mantissa of a floating-point number. Section 3 details our implementation of automatic differentiation within a C++ library with user defined types and overloaded operators. Section 4 describes the application examples and Section 5 gives detailed results of our bitwidth analysis, including estimated area savings.

2. Approach

In this section, we provide a description of our approach to bitwidth analysis together with a presentation of the mathematical reasoning behind it.

2.1. Design Flow

Our bitwidth analysis method consists of three major phases as shown in Figure 1. The first phase modifies a software floating-point program in C/C++ by changing the variable types. The second phase involves compiling the program and linking it against our automatic differentiation library. The third phase involves executing the compiled code to perform the analysis.

The analysis phase consists of two main passes. The *forward-pass* constructs the data flow graph for the design and performs the differentiation of the variables. The resulting differentials are stored in the nodes of the data flow graph. In the *backward-pass* we use the differentials calculated in the forward pass together with the user supplied cost function to calculate the bitwidths. The user supplied cost function contains the parameters, such as the acceptable error in the output measured against a reference output and the maximum area available for the design. The backward pass is applied iteratively on the design until the number of user supplied cost function parameters is maximised.

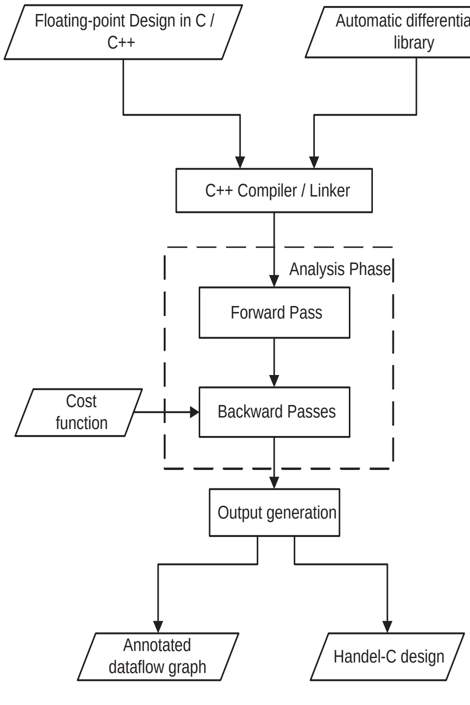

Figure 1: The design flow.

The backward passes generate an annotated data flow graph which contains the calculated bitwidths. This annotated data flow graph is then used by the output phase to either generate a hardware design description or to report the results back to the user.

2.2 Calculation of sensitivity

Consider a function $f : X \rightarrow Y$, where X and Y are the input and output to function f. The sensitivity of Y to X can be approximated by dY/dX or the gradient of Y with respect to X. Automatic differentiation is a method for calculating this gradient function if there is a functional or an algorithmic relationship between X and Y.

The sensitivity of Y to X can then be used to relate the error that occurs in X when its bitwidth is altered, to the error that occurs in Y as a result. Differentiation can approximate this change in Y without evaluating the function f for the new value of X.

2.3 Calculation of bitwidth

Consider a function $f : X \rightarrow Y$ again, where X and Y are the input nodes and output nodes of the function f. A change in Y, denoted by ΔY, is related to a change in X denoted by ΔX as follows:

$$\Delta Y = \Delta X \times \frac{df(X)}{dX} \tag{1}$$

If the output error is expressed as a *Relative Output Error* value, denoted by ϵ_y, then:

$$\epsilon_y = \frac{\Delta Y}{Y} = \left(\frac{\Delta X}{f(X)} \times \frac{df(X)}{dX} \right) \tag{2}$$

Consider a floating-point value represented by X with m_1 mantissa bits and e_1 exponent bits. It is converted to another floating-point number $\hat{X}$ by reducing the bit width of the mantissa to m_2 bits and the exponent to e_2 bits.

The mantissa bit truncation error ΔX can then be expressed as follows:

$$\Delta X = |\hat{X} - X| \tag{3}$$

$$0 \leq \Delta X \leq (|2^{-m_1} - 2^{-m_2}| \times 2^{E_{max}}) \tag{4}$$

The exponent E_{max} is the maximum value of the exponent of X. Here we assume that the mantissas are normalised.

From equation (1) we can determine how much error is tolerable at a given input node when we can tolerate a given error at the output node. Using equation (4) we can convert the error at the input X to a bitwidth specification for that input:

$$m_2 = \log_2 \left(\frac{1}{2^{-m_1} - \frac{|\Delta X/2|}{2\exp(E_{max})}} \right) \tag{5}$$

Consider a data flow graph G with N internal nodes. The sensitivity relationship between the internal nodes $node_i$ and an output of the data flow graph, $output$ and the change in the output, denoted by $\Delta output_i$ can be derived from equation (1):

$$\Delta output_i = \left(\Delta node_i \times \frac{doutput}{dnode_i} \right) \tag{6}$$

When dealing with a data flow graph with multiple internal nodes, we need to split the total error acceptable at the $output$, denoted by $\Delta output$ among the nodes, such that:

$$\Delta output = \sum_{i=1}^{N} \Delta output_i \tag{7}$$

$$\Delta output = \sum_{i=1}^{N} \left(\frac{df(node_i)}{dnode_i} \times \frac{node_i}{f(node_i)} \right) \tag{8}$$

Where f is the transfer function from $node_i$ to $output$.

The partitioning of the total error acceptable at the output ($\Delta output$) into errors resulting from internal nodes ($\Delta output_i$) can be achieved by the following two methods.

In method A, we consider that all the $\Delta output_i$ values are equal. This is shown in equation (9).

$$\Delta output_i = \frac{\Delta output}{N} \tag{9}$$

This is the simplest way of calculating the values of $\Delta output_i$, where all the nodes are treated equally.

In method B, the $\Delta output_i$ values are calculated according to weights W_i allocated to each node, such that $\sum_{i=1}^{N} W_i = 1$.

$$\Delta output_i = \Delta output \times W_i \tag{10}$$

We choose the weights according to the relative area estimates for the operations at each node. This method gives the designer the ability to specifically target area or speed optimisations provided by different node types.

3. Class library for Automatic Differentiation

Automatic Differentiation (AD) deals with differentiating a program at run time, given a specific set of input vectors. Initially we consider conventional AD packages FADBAD [1] and ADOLC [7]. Both these AD packages are based on an operator overloaded C++ library front end and are capable of higher order differentiation. When utilising these packages for our analysis, we find that the analysis execution time is extremely long. The reason is that these AD packages support many differentiation related features which contribute an execution overhead, since these extra features are not required by our analysis.

Another motivation for developing our own automatic differentiation packages is to incorporate the bit width analysis phase. This would improve the execution time of the analysis, by performing both the automatic differentiation and bitwidth analysis in the same step. Thus we implement minimal AD functionality, incorporating the bit width calculation phase as required for precision analysis, as a C++ operator overloaded class library. Since this is based on an operator overloaded library, minimal changes are required to convert ordinary C/C++ code for analysis.

Our C++ class library provides custom data types and operators. As a consequence, at run time each invocation of an operator adds a node to an internal data structure which keeps track not only of the computation, but also keeps track of the differentiated function. The result is that we can take a general C++ program and simply by changing the types of the variables to our user defined types, we implicitly insert an AD code that computes the value of the differentiated function.

For example, the C++ code below shows the differentiation of the function $c = (a + b) \times (a - b)$, at $a = 10.0$ and $b = 5.0$. The main changes required to the original C code is the changing of the type of variables a, b and c to type `autodiff`. In addition we mark out the independent variables with respect to which the differentiation would be done by using overloaded operator `<<=`.

```
autodiff a,b,c;
double gradient_a, gradient_b;
double value;

a <<= 10.0; // Set a as independent
               Variable
b <<= 5.0;  // Set b as independent
               Variable

c = ( a + b ) * ( a - b );

value = c; // Value of c
gradient_a = ( c >>= a ) // Value of dc/da
gradient_b = ( c >>= b ) // Value of dc/db
```

The calculated value of c is available when c is assigned to standard C data types such as `float` or `double`, simplifying integration of non-analysed C code. The gradients calculated can be obtained using the overloaded operator `>>=`.

Compiling and executing the program above generates not only the gradients which we illustrate in this example, but also the sensitivity information. This information is stored inside each variable for use by the precision analysis phase, which is not illustrated in the above code.

The main advantage of our method is that the designer can analyse sections of a large program without having to rewrite the entire program. Thus we are able to build a program analysis tool by using the full power of the C++ language and compiler. We use the GNU C++ compiler to compile the code for our analysis.

4. Examples

In order to illustrate the analysis, we consider two examples.

4.1 Discrete Fourier Transform (DFT)

Out first example is a Discrete Fourier Transform (DFT) implementation. The mathematical formula involved in the calculation is given as follows.

$$X(k) = \sum_{n=0}^{N-1} x(n).W_N^{n.k} \tag{11}$$

$$X(k) = \sum_{n=0}^{N-1} x(n).W_N[(n.k)\ mod\ N] \tag{12}$$

$$W_N = e^{-j2\pi/N} \tag{13}$$

A direct implementation of equations (9)-(11) results in the following C program:

```
/* Generation of coefficients W */
for ( n = 0 ; n < N ; i++ ) {
    W_real[i] =  cos( arg * i );
    W_imag[i] = -sin( arg * i );
}
/* The main computation kernel */
for( k = 0 ; k < N ; k++ ){ // Outer loop
   out_real = in_real;
   out_imag = in_imag;
   for ( n = 0 ; n < N ; n++ ){ // Inner loop
       p = ( n * k ) % N;
       out_real = out_real +
       in_real * W_real[p] - in_imag * W_imag[p];
       out_imag = out_imag +
       in_real * W_imag[p] + in_imag * W_real[p];
   }
}
```

First we analyse the data flow graph of the inner most loop of the above code segment. The resulting data flow graph is shown in Figure 2.

Consider the evaluation of the sensitivity (or gradient) given by $doutput/dinput$. Equation (14) and equation (15) from the inner loop of the DFT code relate the $output$ to the $input$:

$$\begin{aligned} out\_real = out\_real_{-1} + in\_real \times W\_real[p] \\ -in\_imag \times W\_imag[p] \end{aligned} \tag{14}$$

$$\begin{aligned} out\_imag = out\_imag_{-1} + in\_real \times W\_imag[p] \\ +in\_imag \times W\_real[p] \end{aligned} \tag{15}$$

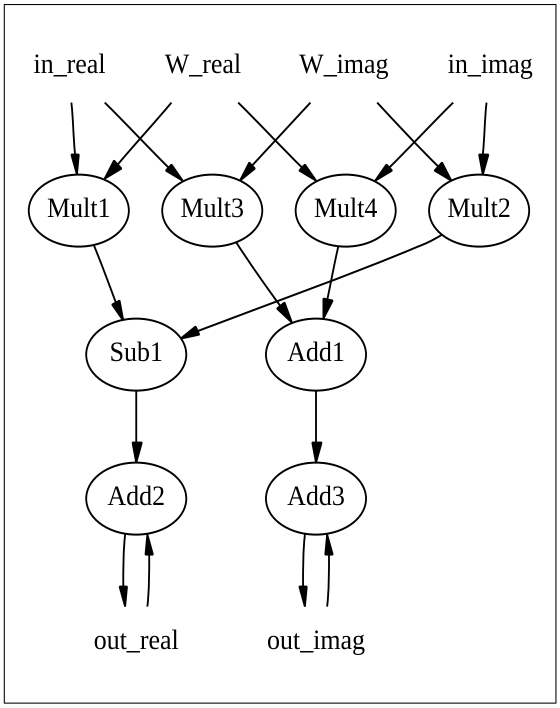

Figure 2: Data flow graph of the inner most loop of the DFT code.

where out_real_{-1} and out_imag_{-1} are the previous values of out_real and out_imag respectively.

Next, we differentiate the two outputs out_real and out_imag with respect to the two inputs in_real and in_imag. This gives rise to the four equations (16) - (19) expressing the gradients:

$$\frac{d\ out\_real}{d\ in\_real} = \frac{d\ out\_real_{-1}}{d\ in\_real} + W\_real[p] \tag{16}$$

$$\frac{d\ out\_real}{d\ in\_imag} = \frac{d\ out\_real_{-1}}{d\ in\_imag} - W\_imag[p] \tag{17}$$

$$\frac{d\ out\_imag}{d\ in\_real} = \frac{d\ out\_imag_{-1}}{d\ in\_real} + W\_imag[p] \tag{18}$$

$$\frac{d\ out\_imag}{d\ in\_imag} = \frac{d\ out\_imag_{-1}}{d\ in\_imag} + W\_real[p] \tag{19}$$

This evaluation of the differentials in equation (16) to equation (19) is performed with the use of automatic differentiation.

We deduce that there is a loop carried dependency, from equation (16) to equation (19) by observing the presence of the $d\ output_{-1}/d\ input$ terms. Since all the loops are unrolled by the automatic differentiation algorithm, this loop carried dependency is taken into account automatically.

We evaluate the gradients of the output with respect to all the other intermediate nodes from Figure 2 in a similar fashion.

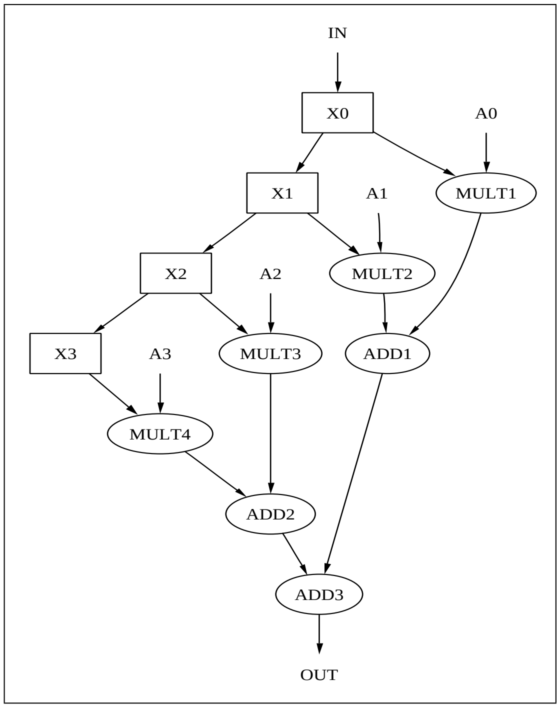

Figure 3: Data flow graph of the FIR filter.

We determine the mantissa bitwidth for the DFT implementation as follows. In the case of nodes which feed into a single output, equation (6) can be used directly by substituting the appropriate values. In the case of nodes in_real and in_imag which feed into two output nodes, the following equations (20) and (21), derived from equation (6), are used instead.

$$\begin{aligned}\Delta in\_real = \Bigg(&\Delta out\_real_{in\_real} \times \frac{d\ out\_real}{d\ in\_real}\\ &+\Delta out\_imag_{in\_real} \times \frac{d\ out\_imag}{d\ in\_real}\Bigg) \qquad (20)\end{aligned}$$

$$\begin{aligned}\Delta in\_imag = \Bigg(&\Delta out\_real_{in\_imag} \times \frac{d\ out\_real}{d\ in\_imag}\\ &+\Delta out\_imag_{in\_imag} \times \frac{d\ out\_imag}{d\ in\_imag}\Bigg) \qquad (21)\end{aligned}$$

The error tolerance values calculated in equation (5) and equation (6) lead to the mantissa bitwidths at each of the nodes in the data flow graph.

4.2 Finite Impulse Response (FIR) Filter

As the second example we show the bitwidth analysis for an implementation of an FIR filter. The filter is implemented according to the data flow graph shown in Figure 3.

DFT	Error Percentage					
Node Name	0%	1%	5%	10%	25%	50%
in_real	23	13	11	10	9	8
mult1	23	12	10	9	8	7
mult2	23	12	10	9	8	7
sub1	23	12	10	9	8	7
add2	23	15	12	11	10	9
in_imag	23	14	12	11	9	8
mult3	23	11	9	8	6	5
mult4	23	11	9	8	6	5
add1	23	11	9	8	6	5
add3	23	13	11	10	8	7

Table 1: Minimum mantissa bitwidth versus specified output error for the DFT implementation. The nodes refer to locations in the data flow graph in figure 2. $\Delta output_i$ is calculated using method A.

The main equation which describes the relationship between the nodes $X0$ to $X3$ and the output node OUT is:

$$OUT = X0 \times A0 + X1 \times A1 + X2 \times A2 + X3 \times A3 \quad (22)$$

Next we differentiate equation (22) with respect to the variables $X0$ to $X3$. This provides us with the gradients $dOUT/dX0$, $dOUT/dX1$, $dOUT/dX2$ and $dOUT/dX3$. In addition to these gradients we also evaluate the gradients for all the intermediary nodes in the data flow graph in Figure 3.

The mantissa bitwidth for the FIR filter is determined as follows. Using the gradients we calculate the error tolerance at each of the nodes $X0$ to $X3$ with the use of equation (6).

$$\Delta X0 = \Delta OUT_{X0} \times \frac{d\ OUT}{d\ X0} \qquad (23)$$

$$\Delta X1 = \Delta OUT_{X1} \times \frac{d\ OUT}{d\ X1} \qquad (24)$$

$$\Delta X2 = \Delta OUT_{X2} \times \frac{d\ OUT}{d\ X2} \qquad (25)$$

$$\Delta X3 = \Delta OUT_{X3} \times \frac{d\ OUT}{d\ X3} \qquad (26)$$

The values of $\Delta X0$ to $\Delta X1$ can then be used with equation (5) to calculate the mantissa bitwidths at each of these nodes. This process is repeated for the rest of the nodes in the data flow graph in Figure 3.

5 Custom Floating-Point Hardware Library

In order to implement the bit optimised floating-point designs produced by our method we develop a customisable

Implementation	Error Percentage				
	0%	1%	5%	10%	25%
FIR	2567	909	580	517	444
DFT	2903	1472	1287	1079	933

Table 3: The area usage in Xilinx Virtex2 slices for different amounts of error that can be tolerated for the FIR and DFT implementations. The results were obtained for a Xilinx XC2V2000 device.

Implementation	Error Percentage				
	0%	1%	5%	10%	25%
FIR	28	30	40	42	44
DFT	59	77	82	84	86

Table 4: The maximum operating speed in MHz for different tolerated amount of error at the outputs for the FIR and DFT implementations. The results were obtained for a Xilinx XC2V2000 device.

FIR	Error Percentage					
Node Name	0%	1%	5%	10%	25%	50%
X0	23	10	7	6	5	4
X1	23	9	7	6	4	3
X2	23	9	6	5	4	3
X3	23	8	6	5	3	2
MULT1	23	8	5	4	3	2
MULT2	23	8	5	4	3	2
MULT3	23	8	5	4	3	2
MULT4	23	8	5	4	3	2
ADD1	23	11	9	8	6	5
ADD2	23	11	9	8	6	5
ADD3	23	11	9	8	6	5

Table 2: Minimum mantissa bitwidth versus specified output error for the FIR filter implementation. The nodes refer to locations in the data flow graph in Figure 3. $\Delta output_i$ is calculated using method A.

hardware floating-point library. This library provides the user the ability to customise the bitwidths of the mantissas and exponents.

The implementation results illustrated in this paper make use of this customisable floating-point hardware library. Currently the library is still under development and does not have any FPGA placement information in it. Our hope is to develop it further by incorporating placement information which is also parameterisable with respect to the mantissa and exponent bit widths. We develop our library and the implementations in VHDL and use Synplicity to compile these into netlists. These are then placed and routed using Xilinx software.

6. Results

We use IEEE single precision floating-point as the starting representation from which we perform bitwidth reduction. Therefore the mantissa of the internal floating-point representations is initially set to 23 bits.

The minimum bitwidth results shown in Table 1 and Table 2 are obtained respectively using method A for the DFT and FFT examples. By varying the amount of error that we tolerate at the outputs (output error specification), we observe the sensitivity of the outputs to changes in internal bitwidths.

Table 3 presents the FPGA area for the DFT implementation and the FFT implementation on a Xilinx Virtex XC2V2000 FPGA. It is remarkable that tolerating 1% error at the outputs of the FIR filter reduces the area of the FIR implementation by 65%. The same 1% error for the DFT reduces the FPGA area of the DFT implementation by 45%.

Table 4 shows the variation in speed when tolerating different amounts of error at the outputs of the DFT and FFT. Again, it is remarkable that tolerating 5% error at the outputs results in a 30% speed improvement for the FIR implementation and a 40% speed improvement for the DFT implementation. Tolerating more than 5% error yields diminishing returns in FPGA area and speed.

More detail on FPGA area and speed results are provided in Figure 4. In our two examples, a tolerable error of less than 5% provides almost all the area and speed improvements possible.

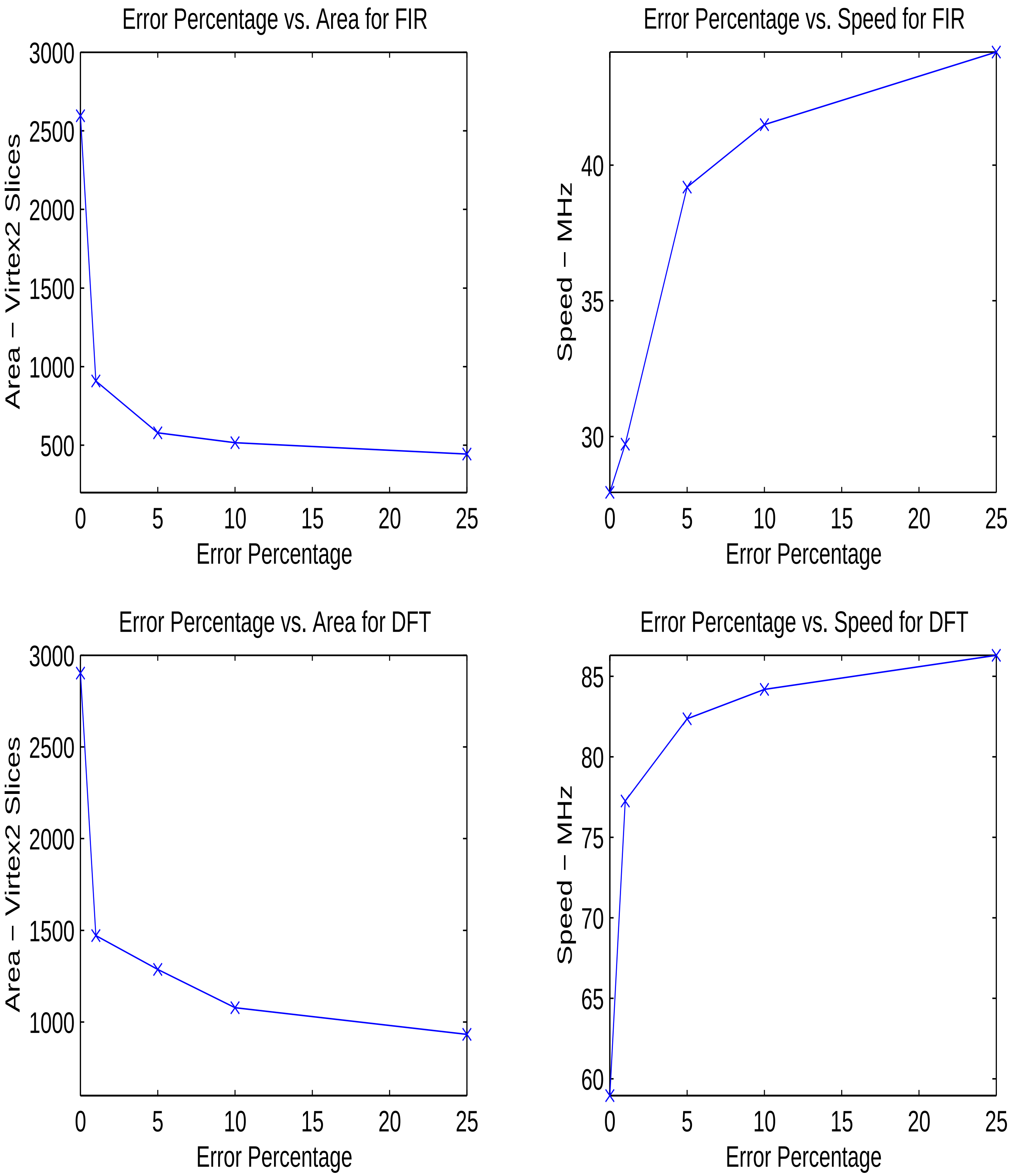

Figure 4: This figure shows the variation in area and speed against the output error percentage for the DFT and the FIR implementations. The results are obtained for a Xilinx XC2V2000 device.

7. Conclusions

This paper shows that it is possible to use automatic differentiation for bitwidth analysis of the mantissa part of floating-point numbers. As to our knowledge this paper is the first to suggest this approach.

The paper shows how to achieve a reduction in the width of the floating point mantissa, minimise the FPGA area and maximise the speed of an FPGA implementation, while keeping an eye on the amount of error at the outputs. On the implementation side, C++ turns out to provide an excellent environment for quick prototyping of the ideas presented in this paper. Current and future work includes extending our analysis to use automatic differentiation for piece-wise differentiable functions.

Acknowledgements

We would like to thank D. Gay for initial discussions on automatic differentiation. Many thanks to Lorenz Huelsbergen and Edward Ang for their comments and assistance. The support of Xilinx, Inc., the ORS Award Scheme and UK Engineering and Physical Sciences Research Council (Grant number GR/N 66599) is gratefully acknowledged.

References

[1] C. Bendtsen and O. Stauning. *FADBAD, A flexible C++ package for automatic differentiation.* Technical report, Department of Mathematical Meodelling, Technical University of Denmark, Lyngby, Denmark, August 1996.

[2] M. Budiu, S. Goldstein, K. Walker and M. Sakr. “BitValue inference:detecting and exploiting narrow bitwidth compilations”. In *Proc. EuroPar Conf.*, June 2000.

[3] R. Cmar, L. Rijnders, P. Schaumont, S. Vernalde and I. Bolsens. “A methodology and design environment for DSP ASIC fixed point refinement”. In *Proc. Design Automation and Test Europe Conf.*, 1999.

[4] G. Constantinides, P.Y.K. Cheung and W. Luk. “The multiple wordlength paradigm”. In *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*. IEEE Computer Society Press, 2001.

[5] P. D. Fiore. *A Custom Computing Framework for Orientation and Photography*. PhD thesis, Massachusetts Institute of Technology, 2000.

[6] A. A. Gaffar, W. Luk, P.Y.K Cheung, N. Shirazi and J. Hwang. “Automating customisation of floating-point designs”. In *Field-Programmable Logic and Applications*, volume 2438 of *LNCS*. Springer, 2002.

[7] A. Griewank et al. *A package for the automatic differentiation of alogorithms written in C/C++*. Technical report, Technical University, Dresden Germany, Ohio University, Athens USA, March 1999.

[8] K. Kum and W. Sung. “Combined word-length optimization and high-level synthesis of digital signal processing systems”. *IEEE Trans. on Computer-Aided Design*, Aug 2001.

[9] R. Razdan. *PRISC: Programmable Reduced Instruction Set Computers*. PhD thesis, Harvard University, 1994.

[10] S.A. Wadekar and A.C. Parker. “Accuracy sensitive word-length selection for algorithm optimization”. In *Computer Design: VLSI in Computers and Processors*, pp. 54–61, 1998.

[11] M. Stephenson, J. Babb and S. Amarasinghe. “Bitwidth analysis with application to silicon compilation”. In *SIGPLAN conference on Programming Language Design and Implementation*. ACM, June 2000.