PRECISION AND ERROR ANALYSIS OF MATLAB APPLICATIONS DURING AUTOMATED HARDWARE SYNTHESIS FOR FPGAS

Anshuman Nayak, Malay Haldar, Alok Choudhary and Prith Banerjee
{nayak, malay, choudhar, banerjee }@ece.nwu.edu
Center for Parallel and Distributed Computing
Northwestern University
Evanston, IL 60208-3118

ABSTRACT

We present a compiler that takes high level signal and image processing algorithms described in MATLAB and generates an optimized hardware for an FPGA with external memory. We propose a precision analysis algorithm to determine the minimum number of bits required by an integer variable and a combined precision and error analysis algorithm to infer the minimum number of bits required by a floating point variable. Our results show that on an average, our algorithms generate hardware requiring a factor of 5 less FPGA resources in terms of the Configurable Logic Blocks (CLBs) consumed as compared to the hardware generated without these optimizations. We show that our analysis results in the reduction in the size of lookup tables for functions like sin, cos, sqrt, exp etc. Our precision analysis also enables us to pack various array elements into a single memory location to reduce the number of external memory accesses. We show that such a technique improves the performance of the generated hardware by an average of 35%.

1. INTRODUCTION

Field Programmable Gate Arrays (FPGAs) have been recently used as an effective platform for implementing many image/signal processing applications. Though the concept of using FPGAs for custom computing evolved in the late 1980's, certain recent advancements in FPGA technology has made reconfigurable computing more feasible. Current trends indicate that FPGAs have a faster growth of transistor density than even general processors. The implication of this is that there will be sufficient transistor budget for larger and more complex applications to be implemented on FPGAs. Most hardware designers today use hardware description languages like VHDL/Verilog or low level CAD tools to implement designs on FPGAs. This involves directly dealing with the complexities of the hardware and understanding the cycle-by-cycle behavior of millions of gates, which can be very tedious and time consuming. Clearly, there is a need for system level design tools that would provide designers a higher level of abstraction enabling the next generation of complex applications of FPGAs with reduced time-to-market.

Many researchers have focused on the use of general purpose languages as a target for hardware synthesis. C/C++ is the most popular target language [19-24]. Some other researchers have attempted to use Java as the target language too [25-27]. Our choice of the MATLAB language is guided by the following facts - **(1)** MATLAB is extremely popular with the signal/image processing community and is easier and more intuitive to use than C/C++ **(2)** MATLAB has a rich set of libraries for signal/image processing functions which can be directly mapped to efficient libraries, thus making MATLAB very conducive to design reuse. **(3)** Large amounts of parallelism can be extracted from MATLAB programs with little or no dependency analysis, as opposed to complex dependency analysis required by languages like C/C++.

Some of the major issues in compilation from a high level language for FPGAs is in generating hardware that will not only fit within the FPGAs, but which will also provide high performance. Since controlling the bitwidth of the variables will result in instantiation of lesser precision operators in hardware leading to FPGA resource savings, there is a need to assign bits in an optimal manner. The need to conserve bits has been investigated for architectures like Intel's MMX [2], HP MAX-2 [3] and SUN VIS [4], which allow data paths to operate on subwords. Also, with the current interest in generating low power systems, researchers have proposed turning off bit slices [1]. Stephenson et. al. [5] have proposed a precision analysis scheme to determine the required bit level precision for various target architectures. All of the above work in subword control have focussed entirely in trying to optimize the bits for integer programs. Since most real applications have floating point operations, our work presents a unified scheme to determine the minimum number of bits for both integer and floating point applications. We also present a scheme to use the precision information to pack several array elements to a memory location, so that the total number of memory accesses is reduced, thereby improving performance.

The contribution of the paper can be summarized as follows:

- We present a value range propagation algorithm to determine the minimum number of bits required for the integral part of floating point representation and for integers
- We present an error analysis algorithm to determine the minimum number of bits required to represent the fractional part of our floating point representation
- We present a memory packing algorithm to pack more than one array elements into a single memory location to improve the performance of the hardware generated

This work presents an automated way of improving the hardware generated by the MATCH compiler [15]. The rest of the paper is organized as follows. Section 2 presents an overview of the MATCH compiler. Section 3 motivates the need for a bitwidth analysis phase and presents our representation of integer and real variables in the VHDL description of the hardware. Section 4 and Section 5 describe our algorithm for precision and error analysis to optimize the FPGA resources consumed. Section 6 describes our memory packing algorithm to improve the performance of the generated hardware. We present some experimental results in Section 7 and conclude in Section 8.

2. OVERVIEW OF THE MATCH PROJECT

The work presented in this paper is part of the MATCH compiler [13]. The MATCH compiler takes in the description of an appli-

cation in MATLAB and partitions it into software to be executed on general purpose and embedded processors and hardware to be mapped to FPGAs. The hardware generated are targeted for the Xilinx FPGAs on the $Wildchild^{TM}$ board from Annapolis Micro Systems. In this paper, we address the issues involved in generating an efficient hardware once the frontend of the compiler has partitioned the system into hardware and software [27]. In particular, we focus on optimizing the FPGA resources in terms of number of Configurable Logic Blocks (CLBs) used up by the instantiation of various operators and registers and in improving the performance of the system by reducing the total number of clock cycles required for external memory accesses. Figure 1 shows an overview of the MATCH compiler. The input MATLAB code is parsed in to develop a MATLAB AST based on a grammer developed by us [14]. Since MATLAB is a dynamically typed language, the type and shape of the variables are unknown at compile time. Hence, a compiler phase infers the type of the variables and dimensions of the matrices and uses this information to scalarize the MATLAB AST. The AST is then levelized wherein complex expressions are broken down into simple expressions with at most three operands. A dependancy analysis phase infers the control and data dependancies present in the AST. A precision and error analysis phase infers the optimum number of bits required for representing the variables in the MATLAB AST and generates a resource optimized VHDL AST. Finally, a memory packing phase packs more than one array element into a single memory location depending on the array precision and optimizes on the number of memory accesses. The output VHDL code is then passed through commercial synthesis and place and route tools to generate a netlist and bit-stream for the FPGAs.

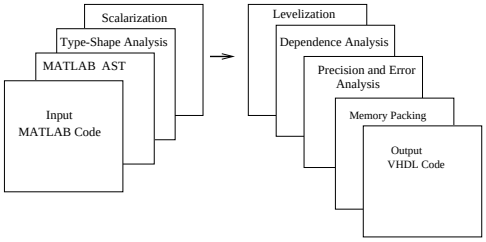

Figure 1. Overview of the Synthesis Framework

3. BITWIDTH ANALYSIS FOR HARDWARE GENERATION

It is well known that when a hardware designer manually converts a high level specification of an application into hardware, he can take advantage of detailed knowledge of the bit level precision of various variables in the application in order to implement an optimum hardware. In order to motivate the need for sophisticated algorithms to synthesize hardware, we performed an experiment on five MATLAB applications namely, Sobel transform, Motion Estimation, Homogeneous Region Testing, IIR Filter and Matrix Multiplication. We took each application and tried to hand map it to FPGAs by writing an RTL VHDL code with the exact precision needed for each variable. We next had our MATCH compiler without the bitwidth analysis take the application and generate RTL VHDL code automatically. This resulted in all variables in the VHDL code being defined as 32 bit wide operands. This is because high level languages used for system level design like MATLAB, C, C++ and Java do not have the notion of bit level precision. In the rest of the paper, we define such a hardware wherein all variables are mapped to 32 bit vectors as unoptimized. The VHDL code generated is then input to the logic synthesis tool from Synplicity and the place and route tool from Xilinx to generate the bit stream for the *WildChild* board. Figure 2 shows the ratio of FPGA

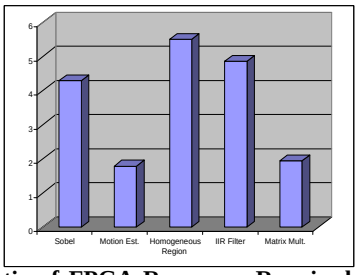

Figure 2. Ratio of FPGA Resources Required for Compiler generated hardware without our optimizations to hand optimized hardware

resources in terms of Configurable Logic Blocks (CLBs) for compiler generated hardware [15] to hand generated optimized hardware. As can be seen, the unoptimized hardware generated by the compiler takes about a factor of 4 more resources than the hand generated hardware. Table 1 shows the total number of CLB's required by the benchmark designs on the Xilinx 4028 FPGA. For the integer matrix multiplication, the unoptimized design required 591 CLB's while a hand generated optimized hardware required only 133 CLB's. This is because the hand generated designs instantiated exact precision operators as compared to the compiled design. Hence, we require a bitwidth analysis phase to determine the minimum number of bits required to represent a integer or a real variable. Further, since MATLAB does not have a representation for variables of different bitwidths, we require a representation for integer and real variables of different precision in the output VHDL code.

Table 1. Total number of CLB's required by Hand optimized and Compiler generated designs

	Sobel	Motion Est.	Homogeneous Region Test	IIR Filter	Matrix Mult.
Hand	199	205	39	220	304
Compiled	856	368	215	1071	591

3.1. Representation of Integer and Real Variables

All variables in the output RTL VHDL code are mapped to bit vectors of type *std_logic_vector* of width as decided by the bitwidth analysis phase. The most significant bit of the bit vector is reserved for the sign bit for both integer and real variables. For integer variables, the value of the variable is converted to binary and directly mapped to the bit vector. For real variables, the most widely used representation is the IEEE floating point representation. Fig 3(a) shows the MATLAB code for the multiplication of two real numbers. Figure 3(b) shows the variables represented in the IEEE format where all variables are represented by 32 bits. Such a representation is often avoided for reconfigurable computing platforms because the floating point operators typically require too much area to be practical. One of the accepted methods of performing fractional operations is to compute the three components of floating point result, sign, exponent and mantissa independently [9, 10]. But such systems do not have high clock rates and are also limited by area [9]. One alternative representation is to use fixed point representations. The main advantage of such a representation is that all operations can be performed using integer operators resulting in much less usage of FPGA resources.

We use a fixed point scheme in which real numbers are represented by both a integer part and a fractional part. The advantage of such an approach can be seen from Figure 3(b) and (c) where we require 32 bits to represent both 5.5 and 1399.75 with a floating point representation while it takes only 4 bits to represent 5.5 and 13 bits to represent 1399.75 with a fixed point representation. This will result in the instantiation of a 17 bit multiplier for the multiplication in Figure 3 (c) as compared to a 64 bit multiplier if floating point representation is used. Hence, the FPGA resources are opti-

mally used. The main disadvantage of such a representation is that the number of bits required for the integral part would be high if the value range of the variable is high. This is acceptable for our study since the dynamic range of all variables in almost all image and signal processing applications is small. Another disadvantage of such a representation is that since both the number of bits required for the integer part and the number of bits required for the fractional part would vary for different variables, integer operators would not give correct results as they require the decimal bits to be perfectly alligned. One solution to this is to remember the number of bits for the fractional part for each real variable and generate conditional code so that integer operators could be used. We have made an assumption that the number of bits required to represent the fractional part is constant for all real variables while the number of bits required to represent the integer part can vary. Figure 4 shows that the multiplication of the two numbers using this assumption requires integer operators. The middle column shows that the multiplication involves a *Xor* operation on the sign bits and a normal integer operation on the other bits. The last column of Figure 4 shows the actual integer multiplication. Our algorithms in the next section can accurately determine the number of bits required for the integer part of the real variable. Hence, the output of the multiplication in the last column of Figure 4 is sampled so that the first 5 bits is for the integer part of the result *c* (since decimal 13 requires 5 bits) and the next 4 bits is sampled for the fractional part (since both the variables *a* and *b* have 4 bits for the fractional part). The main advantage of this representation is that different variables will have different number of bits as required for their representation unlike the floating point representation so that integer operators of the optimal precision would be instantiated leading to resource savings.

We next present a precision analysis algorithm to determine the minimum number of bits required to represent integer variables and the integer part of real variables.

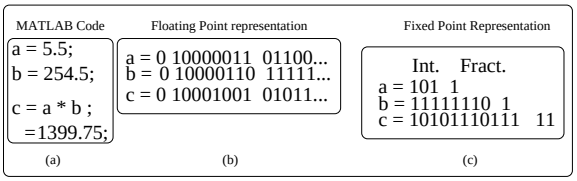

Figure 3. Representation of Real Variables

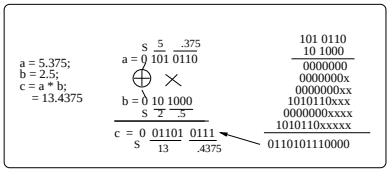

Figure 4. Example showing that Multiplication using our representation uses integer operators

4. PRECISION ANALYSIS

In our representation of variables, the minimum number of bits required to represent integer variables and the integer part of real variables is directly related to the maximum value that the variable attains throughout the program run. Hence, precision analysis or the minimum number of bits required to represent the integer part of the variables can be inferred by value range propagation [28]. We next discuss the value range propagation algorithm to accurately determine the minimum number of bits required for the integer part of the variables.

Algorithm 1 *Precision Analysis for Integer Benchmarks*

Input: MATLAB AST with all variables defined as either Integer or Real
Output: VHDL AST with all variables defined as a bit vector with optimum number of bits
Algorithm :

1. Levelize the MATLAB AST
2. Introduce temporaries so that *integer* $\times$ *float* are converted into *float* $\times$ *float*.
i.e. float f = i $\times$ 3.147
becomes float i' = (float) i; float f = i' $\times$ 3.147;
3. Create a data flow graph with a single static assignment property.
4. Associate 3 structures with each variable in the AST :

- Up: represents the data range during backward propagation
- Down: represents the data range during forward propagation
- Actual: represents the actual data range = Up $\sqcap$ Down.

5.Initialize each of these structures for each variable to $< -INT_{max}, INT_{max} >$
6. Read in target architecture features from a file so that memory width and address width can be used to optimize on the precision of the array elements.
7. do $\{$
8.
Traverse the SSA data flow graph in the forward direction and infer the value range of variables in the lhs of an assignment expression
9.
Calculate the data ranges for the variable being calculated
Note: Data ranges is to be calculated even for real variables

- Find the data range of the result if the transformation is known, else, if the transformation is unknown, as in library and function calls, leave the data values at the maximum
- For loop constructs, the data ranges for variables in the body of the loop can be calculated by actually traversing the loop. If loop bounds are unknown at compile time, then all variables modified inside the loop are assigned the maximum data range to ensure program correctness
- A simple optimization which can be applied when the loop body computations are linear is to find a closed form expression in terms of the loop trip count and the growth factor [7].

10.
Perform error Analysis by finding out the error of the lhs of an assignment expression according to the transformations given in Figure 5
11.
Traverse the SSA data flow graph in the backward direction and infer the value range of variables in the rhs of an assignment expression from similar transformations as in Step **9**.
12.
$\}$while (none of the data ranges change or for a fixed number of iterations);
13. Change the symbol Table to reflect precision information.
14. Perform other Optimizations like *Constant Propagation* and *Dead Code Elimination.*
15. Reflect changes in the VHDL AST so that all variables are represented as bit vectors.
16. Make modifications to the VHDL AST to account for commercial High Level Synthesis Tool peculiarities

4.1. Algorithm for Precision Analysis

Algorithm 1 determines the minimum number of bits required to represent a variable. Step **1** of the algorithm levelizes the MATLAB AST so that all assignment operations are converted to a three operand format. This helps in formulating a series of transformations as shown in [5] which can now be applied on these statements to infer the value range. To avoid converting induction variables used inside loops to be type promoted to real numbers, it is necessary to use temporaries as shown in Step **2**. Value range propagation is simplified by the assumption that every use of a variable has only one reaching definition. Hence, a dataflow graph with a static single assignment (SSA) property is generated. Step **3** uses a Array based SSA representation [8] wherein each array element is renamed so that precision inferencing becomes more accurate.We have implemented a forward and backward propagation algorithm to determine the maximum value of each variable. The precision analysis phase ends once the value range of all variables stabilizes. Certain precision information can be derived from the target architecture for which VHDL is generated. For example, the memory of the slave FPGA's on the *WildChild* board is 16 bits wide and the external memory has 2^{18} locations. Step **6** reads in this information from an architecture file and uses it for inferring the precision of address variables and array elements. An added benefit of Value Range Propagation is in optimizations like *Constant Propagation* [16] and *Dead Code Elimination.*

5. ERROR ANALYSIS

Though the value range propagation algorithm in the previous section can determine the minimum number of bits required for the integer part of the real variable, this is not true for the fractional part of the real variable. This is because a floating point variable can attain innumerable values between two integers. If we use less number of bits to represent the fractional part, then we will be decreasing the resolution of the variable, thereby introducing an error in computations. Hence, we require an error analysis phase to determine the tolerable error.

5.1. Algorithm for Error Analysis

Step **10** of Algorithm 1 finds the error in the fixed point representation of each variable based on transformations outlined in Figure 5. Most image processing applications take as input an image and output another modified image. The actual algorithm performs some floating point operations on these input images to give us the final output image. Hence, the error tolerance in such applications is very high. We can infer the number of resolution bits for real numbers when :

- The compiler assumes that since the intermediate value of 254.99 and 254.01 would result in the same value of 254 for the output data (since output image is an integer), we can have a tolerable error of 1 in the intermediate values
- The user specifies the tolerable error in the pixels of the output image
- The user uses *printf* statements in the MATLAB code and defines the output resolution
- The compiler assumes that since the code was to be executed as a sequential MATLAB code which has a default resolution of 4, all output variables have a resolution of 4 and back propagate this information in the error analysis phase to determine the resolution of intermediate real variables

Hence, the tolerable error for the intermediate real variable used in calculating the output pixel is determined. The forward propagation algorithm 1 uses the transformations outlined in Figure 5 to find out the error in the calculation of the intermediate real variables, both because of its representation using lesser number of bits and also because of its computation from other real variables which have errors in their representation. This error is in terms of the number of bits *t* used in representing the fractional part of the variable. Both the information, namely the tolerable error and the error due to computation using less number of bits is used to determine the minimum number of bits required to represent the variable. Hence, an error analysis will give us the minimum number of bits required to represent the fractional part of the real numbers while the precision analysis algorithm in the previous section will give us the minimum number of bits required to represent the integer part of the real number.

Further, since most image processing applications have calls to *sin, cos, exp* and *sqrt* functions, it is necessary to instantiate lookup tables for these functions in the FPGA core, else, the time taken to transfer data out of the FPGA, execute these functions on the general purpose processor and bring back the data would result in a performance bottleneck. To instantiate a lookup table for the function $y = cos(x)$, it can be seen that if the precision of variable *x* is *p* bits, then, the maximum number of points in the X axis that variable *x* can take is 2^p. Hence, the lookup table would have 2^p rows. Also, if the resolution of all floating point variables is found out by the error analysis stage to be *r* bits, then, since *cos(x)* attains values between -1 and +1, all rows in the lookup table would be *r* bits wide. Hence, for the unoptimized hardware without error analysis, the lookup table would have 2^{32} rows of 32 bits width. On the other hand, after our error analysis phase has decided the optimum resolution to be *r* bits, the size of the table would be reduced to 2^r rows of *r* bits wide resulting in savings in FPGA resources.

6. MEMORY PACKING

It is well known that most of the computations in image processing applications involve memory accesses. When such applications are compiled for a system with an external memory as is true for most commercially available FPGA boards, memory access becomes a performance bottleneck. Hence, reducing the number of external memory accesses could lead to performance gains. An example image processing code for *Region Splitting* in MATLAB is given in Figure 6. An important observation in this code is that each iteration of the loop makes a memory access which is independent of other loop iterations. Also, the memory access patterns are uniform. Most image processing applications that we considered have characteristics which are similar to Figure 6, namely no loop carried dependence and uniform memory access. If these applications are targeted for execution on commercial FPGA boards with an external memory as in the *WildChild* and the *WildStar* board from Annapolis Micro Systems, then each memory access could take as long as 3-4 clock cycles on any of these boards. One way of improving the performance is pipelining the memory accesses [12]. Yet another method which can be implemented over pipelining is by packing more than one array element into the same memory location. For example, the *WildChild* architecture has an external memory which is 32 bits wide for *PE0*. In Figure 6, if we assume that the image *a* and *b* are in a gray scale format and have a value range of $< 0, 255 >$, then the precision of the images is 8 bits and we can pack upto 4 array elements in one memory location. In the *Region Splitting* code shown, since the loop iterations are independent, we can unroll the loop by a factor of 4 so that in each loop iteration, there are 4 different array element accesses which have the same physical memory locations. Hence, the total number of memory access is decreased by a factor of 4 reducing the total number of clock cycles.

6.1. Algorithm for Memory Packing

Algorithm 2 finds the optimum *Packing Order* (PO) for each array, where PO is defined by the maximum number of array elements

If we use only *t* bits for decimal representation instead of *INF* number of bits, then the error would be :

$\epsilon = 2^{-(t+1)} + 2^{-(t+2)} + 2^{-(t+3)} + \cdots + \infty$
$= 2^{-(t+1)}(1 + \frac{1}{1} + \frac{1}{2^2} + \cdots + \infty$
$= 2^{-(t+1)}(\frac{1}{1-\frac{1}{2}})$
$= 2^{-t}$

rep(a) is the representation of variable *a* in our representation

- a = (float) b /* b is an integer */
  $\Rightarrow \epsilon_a = 0$
- float a = 7.3245658
  $\Rightarrow$ rep(a) = a + ϵ_a
  where $\epsilon_a \leq 2^{-t}$
- a = b + c
  $\Rightarrow$ rep(a) = rep(b) + rep(c)
  = b + ϵ_b + c + ϵ_c
  = (b + c) + (ϵ_b + ϵ_c)
  = a + ϵ_a
  Hence, $\epsilon_a = \epsilon_b + \epsilon_c$
- a = b - c
  $\Rightarrow$ rep(a) = rep(b) - rep(c)
  = b + ϵ_b - c + ϵ_c
  = (b - c) + (ϵ_b + ϵ_c)
  = a + ϵ_a
  Hence, $\epsilon_a = \epsilon_b + \epsilon_c$
- a = b × c
  $\Rightarrow$ rep(a) = rep(b) × rep(c) + ϵ
  This ϵ arises due to rounding of/truncation of the *2t* bits generated on multiplication to *t* bits.
  Hence, rep(a) = (b + ϵ_b) × (c + ϵ_c) + ϵ
  = bc + (b ϵ_c + c ϵ_b + ϵ)
  = a + ϵ_a
  Hence, $\epsilon_a \leq$ rep(b) × ϵ_c + rep(c) × ϵ_b + 2^{-t}
- a = $\frac{b}{c}$
  = $(\frac{b+\epsilon_b}{c+\epsilon_c})$
  = $(\frac{b+\epsilon_b}{c})(1 + \frac{\epsilon_c}{c})^{-1}$
  = $(\frac{b+\epsilon_b}{c})(1 - \frac{\epsilon_c}{c})$
  = $\frac{b}{c} - \frac{b}{c^2}\epsilon_c + \frac{\epsilon_b}{c}$
  = a + ϵ_a
  Hence, $\epsilon_a = \frac{1}{c}(\epsilon_b - \frac{b}{c}\epsilon_c)$

Figure 5. Subset of Transformations for Error Analysis

```
for  j = M1:1:M2
    for i = N1:1:N2
        sum = sum + a[j][i] ;
sum  = sum /  ((N2 - N1) * (M2 - M1)) ;
for j = M1:1:M2
  for i = N1: 1: N2
    b[j][i] = (unsigned char) sum ;
```

Figure 6. Example MATLAB code showing application of Region Splitting

```
for i = 1:1:20                    for i = 1:4:20
   a[i+2] = b[i] + c[i+1];  ⟶        a[i+2] = b[i] + c[i+1];
end                                  a[i+3] = b{i+1] + c[i+2];
                                     a[i+4] = b[i+2] + c[i+3];
                                     a[i+5] = b[i+3] + c[i+4];
                                  end
```

Figure 7. Example showing loop unrolled for Memory Packing

that can be packed in each memory location. The minimum number of bits required by the array elements can be determined either by : **1** parsing the input image files provided, **2** provided by the user via directives and **3** computed by the precision analysis phase. Since most of the images read in from MATLAB are stored in a 2-dimensional array, the precision of the input images is inferred by parsing the input matrices to get the maximum value of the array elements. Figure 7 shows a typical loop described in MATLAB and its unrolled version. As memory packing requires consecutive array access across loops, step **8** of algorithm 2, finds out the array access patterns accross loop iterations. Since the maximum unroll factor of the loop can be equal to the array PO, we need to find the array access pattern of the first PO iterations of the loop. The unroll factor of each memory access in a loop is defined by the number of array element accesses across loops which lie in the same physical memory location. To minimize the number of memory accesses, step **12** unrolls the loop by the maximum unroll factor. For the unrolled loop in Figure 7, both the arrays *a* and *c* require two memory accesses while array *b* requires one memory access in a single iteration of the loop. Thus, the total number of memory accesses is reduced by 55 % due to memory packing.

Algorithm 2 *Memory Packing Algorithm*

Input: MATLAB AST with all array elements mapped to a different memory address
Output: VHDL AST with certain array element packed into the same memory location
Algorithm :

1. Parse input data files or parse in user directives or use algorithm 1 to get precision of input arrays
2. Use algorithm 1 to get precision of intermediate arrays
3. Decide *Array Packing Order* (APO) of each array where :
array APO = floor(Memory width / array precision)
4. actual *Packing Order* (PO) = max(APO)
5. **for** all innermost loops in the application
6. Perform a simple Dependance Analysis to check for loop carried dependancies
7. **for** all array accesses in a loop
8. Calculate the set $(X_0, X_1, \cdots, X_{PO-1})$ where X_i is the array element access in the *ith* iteration of the loop
9. Calculate the set $X_i \% PO$ which is the array element access pattern in the loop
10. Calculate maximum unroll factor of the loops so that in each loop iteration, this particular array access when unrolled leads to only 1 packed memory access
11. **end for**
12. Final Unroll Factor = gcd(maximum of all the individual array access unroll factors, loop stop value)
13. Unroll the loop in the MATLAB AST
14. Mark all array accesses in the loop which are redundant
15. **end for**
16. Perform all other Optimizations in the MATLAB AST
17. In creation of the VHDL AST, all memory addresses are changed to (memory address) % PO
18. Dead Code elimination removes all redundant address accesses in VHDL AST

7. EXPERIMENTAL RESULTS

The experimental setup consists of the precision and error analysis algorithm followed by the memory packing algorithm implemented in the framework of the MATCH compiler [13]. Experiments were carried out on a set of widely used image processing applications like *Transitive Closure*, *Sobel Edge Detector*, *Motion Estimation*, *Image Thresholding*, *Homogeneous Region Testing*, *Matrix multiplication*, *Vector sum*, *Inverse Hough Transform*, *Hough Transform*, *IIR Filter*, *Gaussian Noise Generator* and *Laplacian Noise Generator*. Of these, the first seven are

Table 2. Experimental Results showing efficient FPGA resource usage and improvement in performance for benchmark algorithms, * : The design could not be placed and routed on the Xilinx 4028, - : Design not available

	unoptimized hardware			with precision analysis			with precision analysis and memory packing			manually generated hardware		
Benchmarks	CLBs	Freq.	Time	CLBs	Freq.	Time	CLBs	Freq.	Time	CLBs	Freq.	Time
Sobel	856	20.7	*	483	23.6	0.41	561	21.8	0.38	199	18.6	0.06
Image Thresholding	162	28.4	0.09	73	29.7	0.07	144	20.0	0.05	41	27.3	0.04
Homogeneous	215	25.6	0.11	93	31.7	0.08	149	28.2	0.06	39	26.4	0.02
Matrix Mult.	591	20.1	*	133	25.1	12.61	192	21.3	5.7	304	19.9	4.6
Closure	1177	19.3	*	164	24.1	12.71	431	21.7	0.88	-	-	-
Vector Sum	116	32.7	0.06	86	38.4	0.05	132	35.1	0.02	41	28.3	0.01

integer benchmarks and the last five are floating point benchmarks. A detailed description of these algorithms can be found in [26]. For each benchmark, first a description of the algorithm in MATLAB was passed through our compiler without any optimizations to get the unoptimized hardware. Secondly, the algorithm in MATLAB was passed through our compiler with the precision, error and memory packing phases to get the optimized hardware. The output of our compiler was the description of a hardware in VHDL. We used the *Synplify* tools from *Synplicity* to get the netlist and the *Alliance* tools from *Xilinx* to get the FPGA bit stream for the Xilinx *XC4028* FPGA with an external memory on the $WildChild^{TM}$ board from Annapolis Micro Systems.

Figure 8 shows that on an average, the designs consume about a factor of 5 less FPGA resources after our precision analysis phase as compared to the unoptimized hardware. It can be seen that for some benchmarks like *IIR Filter*, the optimized hardware uses a factor of 9.5 less resources than the unoptimized hardware. Further, Figure 2 shows that our manually designed hardware for *IIR Filter* consumes resources which are a factor of 4.7 less than the unoptimized hardware, which implies that our automated tool generates a more resource efficient hardware, by almost a factor of 2, as compared to even a manually designed hardware. The reason for this is that though it is easy to determine the minimum number of bits manually for the input and the output variables even for complex designs, computing the precision for intermediate variables for hardware spanning over a 1000 lines of VHDL code is very tedious and error prone. This is because, the user has to mimick the precision analysis phase in propagating value ranges throughout the code. Hence, it can be inferred from Figure 8 that for large designs, our compiler would generate efficient hardware which would be as good as or better than a manually designed one. Table 2 shows the actual CLBs required and the execution time for some designs after the optimization phases. It can be seen that the execution time of the designs decreases by about 20 % after the precision analysis phase. This is because the number of CLBs required for the logic decreases after this phase so that the commercial high level synthesis tools can route designs in a more efficient manner leading to increased frequency of execution.

Figure 9 shows the average reduction in FPGA resources after our combined precision and error analysis algorithm to be a factor of 3.5 as compared to the unoptimized hardware for applications with floating point operations. The reason for these savings is because we were able to use a unified approach of precision and error analysis to determine the minimum number of bits required for real variables. The final savings in FPGA resources would be far more than the number shown. This is because our error analysis phase would also determine the minimum size of the *sin, cos, sqrt, log* lookup tables so that the error is minimized. For example, without any error analysis, the user would have instantiated a *cos* lookup table with 2^{32} rows of 32 bit width for the *Inverse Hough Transform* MATLAB code for execution on an FPGA board. Our error analysis phase infers the minimum resolution of real variables to be 14 for this application. Hence, our compiler would instantiate a *cos* lookup table of 2^{14} rows with width of 14 bits which would lead to a huge savings of FPGA resources.

Figure 10 shows that on an average, our optimized hardware after memory packing is faster by 35 %. This is because our optimization tries to reduce the total number of accesses to the external memory in the program. For most applications which are easily parallelizable like *Vector Sum*, we can get almost 60 % reduction in the execution times. Column 4 of Table 2 shows that the resources consumed after the memory packing phase goes up by almost 50 %. This is because our memory packing algorithm unrolls the MATLAB *for* loops to extract more parallelism. Hence, there is clearly a resource versus performance tradeoff. For applications like *Sobel Transform*, the major part of the algorithm is computed inside a loop with a huge list of statements, which would have been quadrupled if it were unrolled for memory packing. Most high level syntheis tools like *Synplify* are not able to perform resource sharing optimally in such conditions. Hence, though unrolling would improve the hardware performance, the packing algorithm is selectively applied in the *Sobel* application resulting in an improvement of only 8 %.

Table 2 shows the details of our experimental results including the CLB count, clock frequency of the synthesized design and the execution time of the design on the *WildChild* board for various benchmark applications for the hardware without the optimizations, the optimized hardware afyer precision and error analysis, for the optimized hardware generated after precision analysis and memory packing and for the manually generated optimized hardware. It can be seen from Table 2 that the manually generated hardware is better than the hardware generated by the optimizing compiler by almost a factor of 2.7. This is because the manually generated hardware makes use of the fact that the external memory accesses on the *WildChild* board can be pipelined. Hence, pipelined memory reads and writes take one clock cycle as compared to three clock cycles for our compiler generated designs. Since a pipelining algorithm [11] can be implemented after memory packing, we expect the designs generated by the compiler after the pipelining phase has been integrated to the current MATCH framework to be as good as the manually generated hardware.

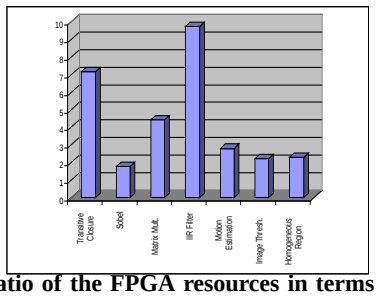

Figure 8. Ratio of the FPGA resources in terms of CLBs required by the unoptimized hardware to that required by the optimized hardware after precision analysis for integer applications

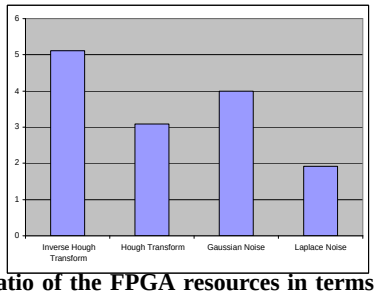

Figure 9. Ratio of the FPGA resources in terms of CLBs required by the unoptimized hardware to that required by the optimized hardware after precision and error analysis for floating point applications

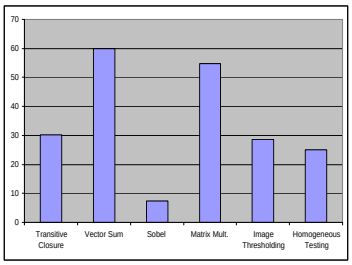

Figure 10. Percentage reduction in the execution time after memory packing

8. CONCLUSION

We have presented a framework for generating an efficient hardware for image/signal processing applications described in MATLAB. We have proposed a representation of floating point variables which would lead to optimal usage of FPGA resources. Also, we have proposed a precision and error analysis algorithm to generate hardware with an average resource requirement reduced by a factor of 5 as compared to an unoptimized hardware before our analysis. We have also proposed a memory packing algorithm to generate faster hardware requiring an average of 35 % less execution time. We have proven the strength of the optimizing compiler by synthesizing hardware for certaing image processing algorithms that are as good as the manually designed hardware in performance and resource needs.

REFERENCES

[1] D. Brroks and M. Martonosi, *Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance*, Proc. of High Performance Computer Architecture, January 1999

[2] A. Peleg and U. Weiser, *MMX Technology Extension to the Intel Architecture*, IEEE Micro, vol. 16, no. 4, p. 42-50

[3] R. Lee, *Subword Parallelism with MAX-2*, IEEE Micro, vol. 16, no. 4

[4] M. Tremblay et. al., *The Visual Instruction Set (VIS) in UltraSPARC*, Proc. COMPCON 1995, p. 462-469

[5] M. Stephenson, J. Babb and S. Amarasinghe, *Bitwidth Analysis with Application to Silicon Compilation*, Proc. of the SIGPLAN Conference on Programming Language Design and Implementation, June 2000.

[6] J. Patterson, *Accurate Static Branch Prediction by Value Range Propagation*, Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1995, p. 67-78

[7] M. P. Gerlek, E. Stoltz, and M. Wolfe, *Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-Driven SSA Form*, Transactions on Programming Languages (TOPLAS), volume 17, issue 1, January, 1995

[8] K. Knobe and V. Sarkar, *Array SSA form and its use in Parallelization*, Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, January 1998, p. 107-120

[9] W.B. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers and K.D. Underwood, *A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs*, Proc. of the IEEE Symposium on FPGAs for Custom Computing Machine, April 1998

[10] N. Shirazi, A. Walters and P. Athanas, *Quantitative analysis of floating point arithmetic on FPGA based custom computing machines*, Proc. of the IEEE Symposium on FPGAs for Custom Computing Machine, 1995, pp. 155-162

[11] D E. Maydan, John L. Hennessy and M. S. Lam, *Efficient and exact data dependence analysis*,Proceedings of the conference on Programming language design and Implementation, June 1991, pp. 1-14

[12] M. Haldar, A. Nayak, A. Choudhury and P. Banerjee, *Automated Synthesis of Pipelined Designs on FPGAs for Signal and Image Processing Applications described in MATLAB*, submitted to Asia South Pacific Design Automation Conference, 2001

[13] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak and S. Periyacheri, *A MATLAB Compiler for Distributed, Heterogeneous, reconfigurable Computing Systems*, Proc. IEEE Symposium on FPGA as Custom Computing Machines, FCCM 2000

[14] P. Joisha, A. Kanhare, P. Banerjee, N. Shenoy, A. Choudhary, *The Design and Implementation of a Parser and Scanner for the MATLAB Language in the MATLAB Compiler*, Technical Report, center for Parallel and Distributed Computing, Northwestern University, CPDC-TR-9909-017, Sep. 1999

[15] M. Haldar, A. Nayak, N. Shenoy, A. Choudhary and P. Banerjee A. Choudhary, *FPGA Hardware Synthesis from MATLAB*, The 14th International Conference on VLSI Design, India, Jan 2001

[16] E. Stoltz, M. Wolfe and M.P. Gerlek, *Constant Propagation: A Fresh Demand-Driven Look*, Proc. of the 1994 ACM Symposium on Applied Computing, March 1994, pp. 400-404

[17] I. Page *Constructing hardware-software systems from a single description*, Journal of VLSI Signal Processing, pp. 87-107, 1996

[18] J. Babb, M. Rinard, C.A. Moritz, W. Lee, M. Frank, R. Barua, S. Amarasinghe *Parallelizing Applications into Silicon*, FCCM 1999

[19] D. Galloway *The Transmogrifier C Hardware Description Language and Compiler for FPGAs*, FCCM'95

[20] M. Gokhale, J. Stone, J. Arnold and M. Kalinowski, *Stream-Oriented FPGA Computing in the Streams-C High Level Language*, Proc. Field-Programmable Custom Computing Machines, April 2000.

[21] G. Doncev, M. Leeser and S. Tarafdar, *High-Level Synthesis for Designing Cu stom Computing Hardware*, Proc. Field-Programmable Custom Computing Machines, April 1998.

[22] A. Duncan, D. Hendry, and P. Gray, *An Overview of the COBRA-ABS High Level Synthesis System for Multi-FPGA Systems*, Proc. Field-Programmable Custom Computing Machines, April 1998.

[23] J.M.P.Cardoso and H.C.Neto *Towards an Automatic Path from Java(tm) Bytecodes to Hardware Through High-Level Synthesis*, Proceedings of the 5th IEEE International Conference on Electronics, Circuits and Systems (ICECS-98), Lisbon, Portugal, September 7-10, 1998, pp. 85-88

[24] B. L. Hutchings and B. E. Nelson,*Using General-Purpose Programming Languages for FPGA Design*, Proc. 37th Design Automation Conference, June 2000.

[25] R. Helaihel and K. Olukotun, *Java as a Specification Language for Hardware-Software Systems*, Proc. International Conference on Computer-Aided Design, pp. 690-697. November 1997.

[26] I. Pitas, *Digital Image Processing Algorithms and Applications*, John Wiley & Sons Inc., 2000

[27] N. Shenoy, A. Choudhary and P. Banerjee, *A System-Level Synthesis Algorithm with Guaranteed Solution Quality*, Proc. Design Automation and Test in Europe, March 2000

[28] S. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kauffmann Publishers, 1997