A SYSTEM FOR SYNTHESIZING OPTIMIZED FPGA HARDWARE FROM MATLAB®

Malay Haldar and Anshuman Nayak
Mach Design Systems, Inc.
www.MachDesignSystems.com

Alok Choudhary and Prith Banerjee
Northwestern University
Evanston, IL 60208.

ABSTRACT

Efficient high level design tools that can map behavioral descriptions to FPGA architectures are one of the key requirements to fully leverage FPGA for high throughput computations and meet time-to-market pressures. We present a compiler that takes as input algorithms described in MATLAB and generates RTL VHDL. The RTL VHDL then can be mapped to FPGAs using existing commercial tools. The input application is mapped to multiple FPGAs by parallelizing the application and embedding communication and synchronization primitives automatically. Our compiler infers the minimum number of bits required to represent the variable through a precision analysis framework. The compiler can leverage optimized IP cores to enhance the hardware generated. The compiler also exploits parallelism in the input algorithm by pipelining in the presence of resource constraints. We demonstrate the utility of the compiler by synthesizing hardware for a couple of signal/image processing algorithms and comparing them with manually designed hardware.

1. INTRODUCTION

The concept of using FPGAs for custom computing evolved in the late 1980's. Wide adoption of the concept, however, has gained grounds only recently. One of the principal enabling factor was the availability of commercial synthesis and physical placement tools that raised the level of design abstraction to hardware description languages such as VHDL/Verilog. With gate counts for modern FPGAs reaching millions, we are poised for yet another revolution. The goal this time is to raise the level of abstraction to general purpose programming languages such as C/C++, Java and MATLAB[1]. Current design methodologies rely on the expertise of the hardware engineer to map the application onto a FPGA board. While this enables a lot of flexibility and fine grained control over the design, it also introduces a lot of logic design at a very low level. Not only is this process tedious and error-prone requiring costly debugging iterations, much of the work can be automated resulting in designs which are correct by construction. Another key aspect of mapping applications onto hardware is to exploit coarse and fine grained parallelism in the application. Again, concurrent simulations of multiple states is not the easiest thing to manage. Many mature and advanced compiler techniques exist that can discover and exploit parallelism and weigh different trade-offs automatically. All this will relieve the designer to focus on high level algorithmic aspects rather than learning about new board architectures or ways to boost performance by low level manipulations.

Synthesizing hardware from general purpose languages has received attention in both industry and academia. A broad classification of the different approached is possible from two perspectives :

1. *Target Language* : The approaches can be categorized according to the languages they target for synthesis. C/C++ has been the most popular choice [12, 15, 16, 17, 18, 20, 21, 22, 25, 19]. Java has been the focus of some recent works [24, 23]. Our focus is on MATLAB.
2. *Parallelism Specification* : The approaches can be classified depending on whether they attempt to automatically parallelize the input applications [22, 25, 20, 19] or they depend on the user to specify the parallelism [12, 18, 16, 17, 21, 24]. Depending on the user to specify the parallelism simplifies the compiler a lot, but it typically requires modifications/additions to the target language. It also burdens the user to extract the parallelism. User specified parallelism approaches does raise the design abstraction from VHDL/Verilog, but still require considerable manual iterations and interventions. Automatic parallelization makes the compiler complex but it doesn't require any modifications to the language and the user is not burdened with finding parallelism. This cuts down manual iterations to a minimum. Our approach is automatic parallelization, but experienced users can also direct the compiler through directives.

Optimized hardware synthesis from a general purpose language is a very complex task and the associated compiler framework has many components. The components include the front-end of the compiler dealing directly with the target language, the intermediate synthesis framework, the optimizations involved in synthesizing the hardware and the back-end which outputs the hardware and interfaces with lower level tools. All the components have their own specific issues, which were addressed in individual bits and pieces with many alternative solutions [4, 6, 5, 7, 10, 11, 15, 18, 19, 20, 22, 21, 24, 17, 16, 25]. Our attempt in this paper is to discuss the complete system of an optimizing hardware synthesis tool and put forward a working combination of the mass of solutions contributed for each aspect of the compiler.

2. WHY MATLAB ?

While most of the industry and academia has focused on C/C++ as the system description language, our main focus is on MATLAB® [1]. Whereas many synthesis issues seem independent of the input specification language, MATLAB does offer distinctive advantage due to the following two reasons :

1. MATLAB is extremely popular in the signal/image processing community with over 500,000 users. MATLAB is more intuitive than C/C++ and it enables simulation and visualization of algorithms with much less effort than C/C++. A direct synthesis path from MATLAB without first converting it into another language like C/C++ will be very useful and it will enable very rapid and easy evaluation of a lot of algorithms. Thus a designer will be able to directly see the tradeoffs resulting from high level algorithmic changes.
2. A key technique that enables multi-million gate designs is reuse of Intellectual Property (IP) cores. Such cores correspond to common functions such as FFT, Viterbi decoders and Matrix Multiplication. These functions are available in MATLAB as standard function calls and operators with standard

[1] MATLAB® is a registered trademark of Mathworks, Inc.

interfaces. This feature becomes particularly useful in recognizing that a particular IP block can be used for part of the input application and how to generate the interface signals corresponding to it. In languages without standard library calls for the algorithms, there may be innumerable ways to specify and invoke the algorithms. In such a situation it becomes very difficult to recognize that an IP block can be used for part of the algorithm and generating the interfaces for it.

However, MATLAB does have some disadvantages. The two main issues in that respect are :

1. MATLAB doesn't have any notion of type/shape for its variables. This becomes a nightmare from the compiler perspective and using existing techniques, in most cases inefficient code is generated that covers all or many of the possible types/shapes of the variables. We have developed a type/shape algebra framework that enables accurate inferencing, leading to efficient hardware generation [9]. In spite of the inferencing framework, if the compiler is unable to do a satisfactory job, the user can force the type/shape of a variable through directives.
2. Simulation of scalarized MATLAB code is slower than a compiled approach. This is because MATLAB is an interpreted language which incurs a lot of overhead if simple computations are done in a loop. However, our focus is on signal/image processing kind of applications where arbitrary loops and array manipulations is not the norm. Regular loops with extensive use of library functions is more common for such applications for which MATLAB is ideally suited.

3. COMPILATION OVERVIEW

We now present an overview of our compiler architecture. Figure 1 shows the different compiler phases. The front-end parses the input MATLAB program and builds a MATLAB AST(Abstract Syntax Tree). The input code may contain directives [3] regarding the types, shapes and precision of arrays that cannot be inferred, which are attached to the AST nodes as annotations. This is followed by a type-shape inference phase. MATLAB variables have no notion of type or shape. The type-shape phase analyzes the input program to infer the type and shape of the variables present for which type/shape is not provided by directives. This is followed by a scalarization phase where the operation on matrices are expanded out into loops. In case optimized library functions are available for a particular operation, it is not scalarized and the IP core corresponding to the library function is used instead. The scalarized code is then passed through the parallelization phase. The parallelization phase attempts to exploit coarse grain parallelism by either splitting a loop onto multiple FPGAs on the board (data-parallel approach) or by putting different tasks onto different FPGAs and pipelining the output of one to the input of another (systolic approach). The parallelization phase relies on communication libraries implemented for the target architecture board to communicate between the different FPGAs. A state machine description in VHDL is then synthesized from the parallelized scalarized MATLAB code for each of the FPGAs . Most of the hardware related optimizations are performed on the VHDL AST. A precision inference scheme finds the minimum number of bits required to represent each variable in the AST. The precision information is used in instantiating customized IP blocks corresponding to the functions and operators. Transformations are then performed on the AST to optimize it according to the memory accesses present in the program and characteristics of the external memory. This is followed by a phase to perform optimizations like pipelining under resource constraints that alter parts of the state machine that was constructed earlier. Finally a traversal of the optimized VHDL AST produces the output code.

4. EXPERIMENTAL SETUP AND BENCHMARKS

Our compiler is designed to produce code for most current FPGA architectures. The results presented in this paper are for hardware generated for the *WildChild*^{TM} FPGA board from Annapolis Micro Systems. It is a VME compatible board with eight *Xilinx 4010* FPGAs and one *Xilinx 4028* FPGA. The *Xilinx 4028* has an external memory that is 32-bit wide with 2^{18} addressable locations. The memories connected to the 4010s are 16-bit wide.

The benchmarks include Matrix Multiplication, FIR filter, IIR filter, Sobel edge detection algorithm, an Average filter and a Motion Estimation algorithm. These benchmarks represent typical signal/image processing applications that are of interest to us. On one hand, such applications are important as they are representative of a class of applications that are predicted to be ubiquitous in next generation computing platforms, in environments that demand high throughput. On the other hand, these applications have inherent parallelism suitable for exploitation by implementation in customized hardware.

5. MATLAB TO VHDL

One of the challenges in generating hardware from MATLAB is to figure out the type/shape of the variables. As shown in Figure 2, the semantics of an operator can depend on the assignments to the operands. To generate hardware, the compiler must figure

```
a = 1 ;                 a = rand( 256 , 234 );
b = 3 ;                 b = ones(234, 512 );

c = a * b ;             c = a * b ;

  ( i )                   ( ii )
```

Figure 2. The semantics of an operator depends on the type/shape of the operands in MATLAB. In (i) * is a scalar multiplication whereas in (ii) it is a matrix multiplication.

out the exact data type i.e, integer or floating point, or complex numbers etc. The compiler also needs to figure out the shape i.e, how many dimensions the matrix (array) has, what are the extents in each dimension, etc. Our type shape algebra framework automatically figures out the type-shape of the variables [9]. In pathological cases where the compiler is unable to infer the type/shape of the variables, the user can assist the compiler by specifying the type/shape of selected variables. Once the type/shape of the variables are determined, the matrix operations are scalarized, the operations are expanded out into loops. Scalarization of the MATLAB AST is necessary when the objective is to perform a source-to-source transformation to a target language that is statically typed and which only supports elemental operations. MATLAB is an array-based language with many built-in functions to support array operations. Hence, to generate a VHDL description, it is necessary that the corresponding MATLAB AST is scalarized. Figure 3 shows an example where VHDL code is generated corresponding to a matrix multiply operation. Extensive discussion of VHDL generation from MATLAB is reported in [4]. The framework is capable of handling multi-dimension matrices which are mapped to an external memory. In addition, the loop and function call constructs of MATLAB are also supported. Figure 4 shows the experimental results of execution times of the benchmarks on a *Xilinx 4028* using manual and compiler approaches. As can be seen, the manually designed hardware on the average is five times better than the compiler output, noting that it took several months to complete the manual designs while the compiler generated the hardware in a matter of minutes.**Reduction of design time is the key advantage of using the compiler**. In the next few sections, we elaborate how our compiler closes the performance gap between its output and the manually designed hardware.

6. PRECISION INFERENCING

One important factor in generating customized hardware for an application is to efficiently utilize the silicon budget available. A key observation in this regard is that most image/signal processing computations are confined to 8 $\backsim$ 16 bits. To fully leverage

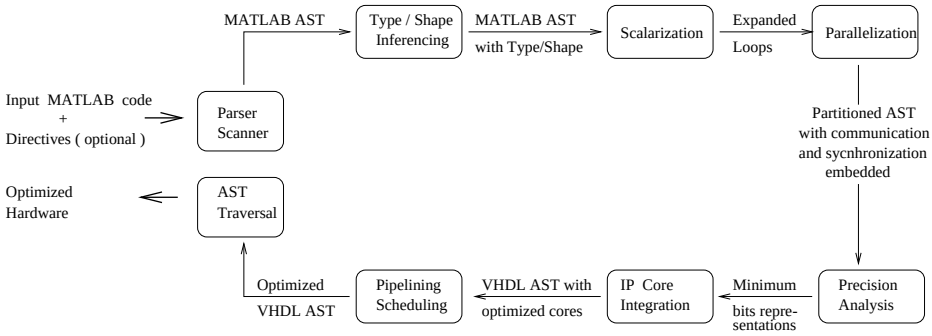

Figure 1. Overview of the synthesis framework.

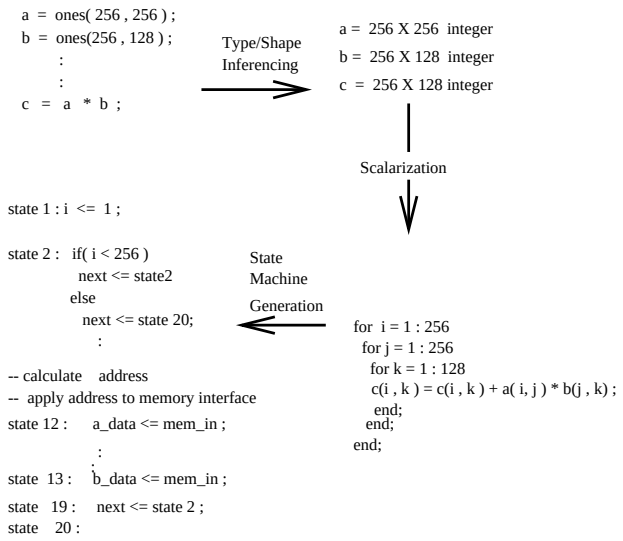

Figure 3. An example showing how a state machine is synthesized for matrix multiplication by first doing type/shape analysis, followed by scalarization.

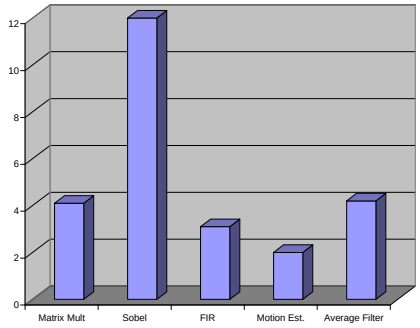

Figure 4. Ratio of execution times of compiler generated hardware compared to manually designed hardware is shown. For example, for the matrix multiplication benchmark, the compiler generated hardware is 4 times slower than the manually designed hardware.

this fact, the minimum number of bits required to represent each variable must be inferred and appropriate operators instantiated in place of generic 32-bit operators. However, figuring out the precision manually in a real life design can be very tiresome and error prone. Our precision inferencing algorithm propagates value range information back and forth the AST to figure out the minimum bits required to represent a variable, see Figure 5. In case where

```
a = 8 ;      % 4 bits required
b = 4 ;      % 3 bits required
d = a + b ;  % 4 bits required
e = b + input()  % unknown , give
                 % directive
```

Figure 5. Illustration of precision inferencing.

the precision of variables cannot be determined statically, the user can specify the precision by a directive; otherwise the most conservative estimate is taken. For floating point variables, in association with the precision inferencing algorithm an error analysis and propagation scheme is included. The error analysis determines the resolution of the floating point variables needed given a specified error that can be tolerated at the output. Details of the precision and error analysis algorithms can be found in [7]. Figure 6 shows the savings of resources in terms of CLBs when the precision inferencing algorithm is applied as opposed to instantiating generic 32 bit operators.

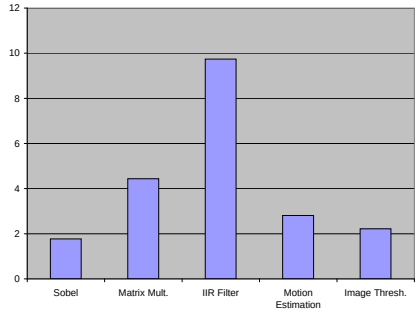

Figure 6. Ratio of the resource utilization in terms of CLBs while instantiating 32-bit operators as compared to determining the minimum number of bits required by precision inferencing.

7. IP CORE INTEGRATION

IP cores range from optimized implementations of FFT and Viterbi decoders to adders and multipliers [13, 14]. To produce a design that rivals manual designs, one must be able to integrate these IP cores automatically into the designs synthesized. To leverage IP cores provided by different vendors, our compiler provides a standard and open interface between the compiler and the IP core

database. The IP core database contains the EDIF/HDL implementations and interfaces to the IP cores. Along with each IP core, a code (in C/C++) is also present that generates the interface for using the particular IP core. This code is referred to as the interface generator of the IP core. The interface generator can access information about the AST through a uniform and open interface with the compiler. The compiler needs to provide information regarding the variables and operators, like whether they are signed/unsigned, integer/floating point, constant/variable and what precision. The IP database on the other hand must provide information to the compiler regarding the area/performance of the IP cores. The standard interface is a way of facilitating these information flows from different IP cores provided by different vendors. Details of the interface and issues surrounding the interface generator were described in [2]. Figure 7 summarizes the improvements obtained by using the optimized IP cores as opposed to generic operators and scalarizing all the functions.

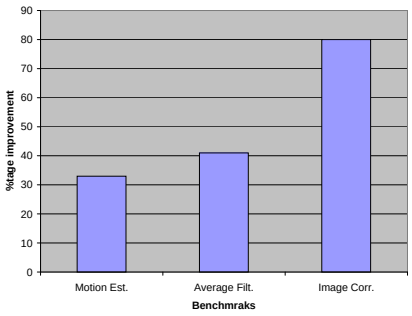

Figure 7. Percentage improvement in execution time by using optimized IP cores over generic operators.

8. PIPELINING

A close study of the manually designed hardware and the compiler generated hardware showed that the principal reason behind the better performance of the manually generated hardware was exploitation of fine grain parallelism and pipelining of the memory accesses. This prompted us to devise an automated way of pipelining the memory accesses and exploit fine grained parallelism. Our pipelining framework achieves this objective, an overview of which is given in Figure 8. The pipelining phase starts by performing dependency analysis of the loops and basic blocks. The GCD test is employed to figure out loop carried dependencies. In case there are no backward dependencies in a loop, the loop is deemed pipelinable. Next the number of memory ports are read as input to the pipelining algorithm. The pipelining algorithm then performs modulo scheduling [6] which overlaps different iterations of a loop such that number of memory access in any state does not exceed the number of memory ports specified. The modulo scheduling algorithm can be based on either ASAP (as soon as possible) or ALAP (as late as possible) algorithms. The reason the pipelining algorithm is based on memory ports is that many of the image/signal processing applications are memory bound. They tend to perform simple operations on relatively large data sets that reside in external memories. Hence, optimizing the memory accesses in general has a huge impact on performance. Conflicts created in variables due to overlapping of iterations is solved by renaming the variables as discussed in [6]. Figure 9 shows the impact of pipelining on performance. The compiler generated pipelined hardware matches the manual designs in most cases. In fact, the compiler generated pipelined hardware fares better than the manually designed hardware for the FIR and Motion Estimation benchmark. This is due to the fact that manual designers typically pipeline and exploit parallelism *within* a single iteration of the loop. The compiler can handle much more complexity and exploits parallelism *across* the different iterations of the loops. For example, the pipeline kernel synthesized for the motion estimation benchmark contained 200 concurrent statements spanning 5 iterations. Such complexity can only be handled in an automated fashion.

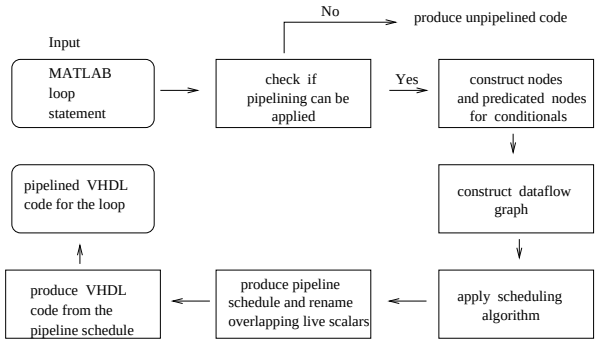

Figure 8. An overview of the pipelining framework.

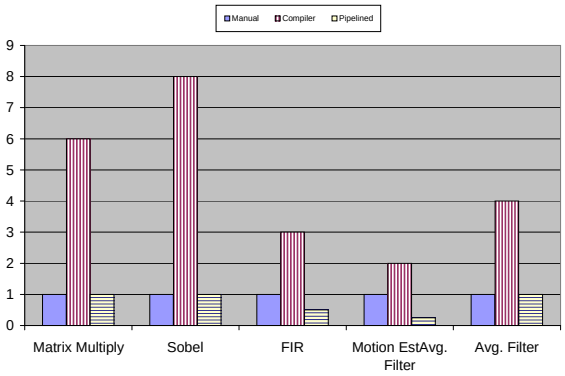

Figure 9. Ratio of the execution times of the compiler generated hardware with and without pipelining is shown, normalized to the execution time of manually designed hardware. For example, for the Sobel benchmark, the compiler generated hardware without pipelining is 8 times slower, whereas the pipelined hardware is as fast as the manual design.

9. RESOURCE CONSTRAINED PIPELINING

The pipelining algorithm presented in the previous section was only constrained by the number of memory ports and was based on an ASAP/ALAP scheduling algorithm variation. However, resource constraints must also be taken into account while pipelining. Typically, the pipelined versions consumed twice the amount of CLBs when compared with the non-pipelined versions for the benchmarks presented before. To introduce the capability of producing intermediate designs that were pipelined but used less aggressive parallelism, a resource constrained pipelining framework was introduced. The resource constraints can be specified to the pipelining algorithm in terms of number of high level operators. The pipelining algorithm uses a list scheduling algorithm modified for modulo scheduling [5], such that the number of operators used concurrently in any state of the pipeline does not exceed the specified number of operators. The list scheduling algorithm relies on a heuristic that estimates the priority of each operator while scheduling. We experimented with existing heuristics which gave precedence to operators with the maximum number of children or whose distance from the sink node of the control-data flow graph was longest. We also devised our own heuristics which was based on the aggregate resource requirements of the tree fanning out from the operator and the available resources. It has been shown that such an approach produces efficient pipeline schedules [5]. Figure 10 shows a summary of the results for the different heuristics

used while producing resource constrained pipelines.

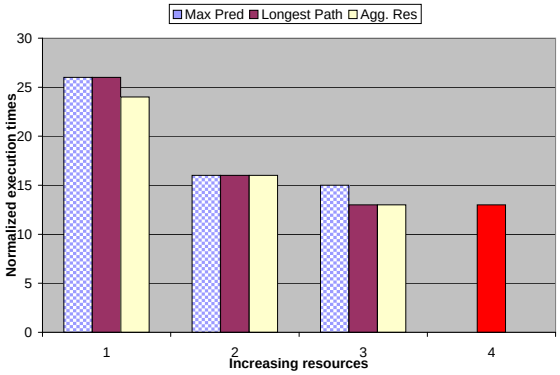

Figure 10. Normalized execution times corresponding to the different heuristics used in the list scheduling algorithm (maximum predecessor, longest path from sink, aggregate resources of fanning tree) for the Sobel benchmark is shown. The execution time with unconstrained resources is shown at the far right.

10. MULTI-FPGA PARALLELIZATION

The scheduling and pipelining framework described above is geared towards utilizing the fine grained parallelism present in the application. Most current FPGA boards contain multiple FPGAs. A mechanism to leverage coarse grained parallelism is required to make use of the multiple FPGAs. In this direction, our parallelization framework partitions the scalarized MATLAB AST to generate a partitioned AST for each of the individual FPGAs present on the board. An architecture description of the FPGA board is required for specifying the target architecture to the parallelization phase. The parallelization phase assumes a set of standard communication and synchronization primitives are implemented on the FPGA board. The parallelization phase embeds these primitives to synchronize between the different parts of the AST mapped to different FPGAs. In particular, two approaches to parallelize the MATLAB AST are adopted. The first approach is a data parallel approach wherein the execution of a loop is spread across the FPGAs. The data that is operated on by the loop is split across the memories associated with the FPGAs. This is similar to parallelizing loops for distributed memory machines. In the second approach, different loops are mapped to different FPGAs, and the output of one loop is piped to another loop. This is similar to the systolic parallelization schemes. While the data parallel approach is most effective for memory bound applications due to the increased memory bandwidth of multiple FPGAs, the systolic approach is particularly useful for large applications as the logic for the application can be spread across several FPGAs.

For the Sobel edge detection benchmark a speedup of 7.5 was obtained by mapping the application onto 8 FPGAs. The speedup was obtained in comparison to the hardware generated for a single FPGA by the compiler. Comparison with manually designed hardware in this case is somewhat involved as manually mapping the applications onto multiple FPGAs is a very time consuming proposition (the very reason we designed the compiler!). Moreover, as the speedup obtained is very near to optimal, we conclude that the parallelization was satisfactory. Extensive benchmarking and comparisons of the parallelization phase is part of our current work.

11. SUMMARY

We now present the result of performing all the optimizations in tandem and comparing against manually designed hardware. We would like to emphasize once more that the manually generated hardware took months of design effort whereas the compiler generated the hardware in a matter of minutes. While a massive reduction in design time is achieved, the quality of the hardware generated was not compromised. Indeed, the hardware generated by the compiler were very close to the manually generated hardware in performance, in fact better in some cases. Figure 11(i) shows an input image to the Sobel edge detection algorithm. Figure 11(ii) shows the output of the Sobel edge detection algorithm as simulated in the MATLAB interpretor.

The same MATLAB code was then used to synthesize a pipelined hardware. The output of the hardware is shown in Figure 11(iii). The output matches the simulation result pixel by pixel. The designs generated by the compiler are correct by construction and do not require debugging iterations. Figure 12 shows the comparison of the execution times of the compiler generated hardware with the optimizations against the manually designed hardware for the benchmarks. Figure 13 shows a comparison of the resource utilization for the same. The performance of the compiler output and manually optimized hardware are comparable. The resource utilization of the compiler generated hardware are within a factor of four of the manually designed hardware.

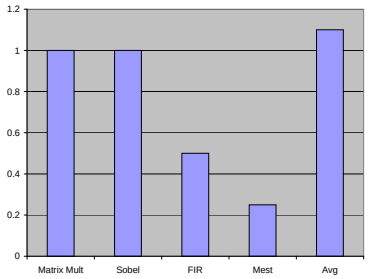

Figure 12. Ratio of the execution times of the compiler generated hardware with the optimizations as compared to the manually generated hardware.

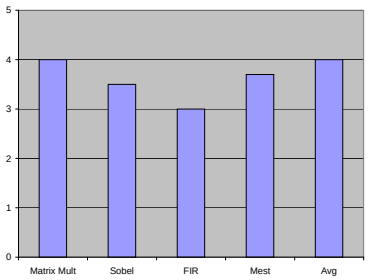

Figure 13. Ratio of the CLBs used by the compiler generated hardware with optimizations as compared to the manually generated hardware.

12. FUTURE WORK

The major focus of our current and future work is in the following two directions

1. We are investigating methods to identify and utilize opportunities to synthesize on-chip caches to reduce the memory traffic and boost the performance of the synthesized hardware.
2. We are concentrating on accurate prediction of the resource and routing resources needed for a particular design to achieve design closure in minimum iterations possible.

13. CONCLUSIONS

In conclusion we have presented a compiler capable of generating highly optimized hardware from applications described in MATLAB. A set of effective optimizations implemented in the compiler

(i)Input Image (ii)MATLAB Interpreter (iii)Annapolis *Wildchild*

Figure 11. A grayscale image is shown in (i). Output of the Sobel edge detection algorithm simulated in the MATLAB interpretor is shown in (ii). The MATLAB code is used to synthesize hardware for the Annapolis *Wildchild* board and its output is shown in (iii).

ensures that the quality of the output hardware is comparable to manually optimized hardware. The optimizations include parallelization, precision inferencing, IP core integration and pipelining. The effectiveness of the compiler was demonstrated by synthesizing hardware for a couple of signal/image processing applications. The outputs of the synthesized hardware were functionally verified against the outputs of the MATLAB interpretor. The execution times were almost equivalent to manually designed hardware, in fact superior in some cases were large amount of parallelism was available across loops. The resource utilization were within a factor of four of the manual designs. **All this was achieved while reducing the design time from months to minutes.**

REFERENCES

[1] The Mathworks Homepage, www.mathworks.com.

[2] Malay Haldar, *Optimized Hardware Synthesis for FPGAs*, PhD Thesis, Northwestern University, 2001.

[3] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden and D. Zaretsky, *A MATLAB Compiler for Distributed, Heterogeneous, Reconfigurable Computing Systems* , Proc. IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM'00, April 2000.

[4] M. Haldar, A. Nayak, A. Choudhary and P. Banerjee, *FPGA Hardware Synthesis from MATLAB,* 14th Intl. Conf. on VLSI Design, January 2001.

[5] M. Haldar, A. Nayak, A. Choudhary and P. Banerjee, *Scheduling Algorithms for Automated Synthesis of Pipelined Designs on FPGAs for Applications described in MATLAB*, Intl. Conf. on Compilers, Architectures and Synthesis for Embedded Systems, CASES'2000, November 2000.

[6] M. Haldar, A. Nayak, N. Shenoy, A. Choudhary and P. Banerjee, *Automated Synthesis of Pipelined Designs on FPGAs for Signal and Image Processing Applications Described in MATLAB* , Asia Pacific DAC, January 2001.

[7] A. Nayak, M. Haldar, A. Choudhary and P. Banerjee, *Precision And Error Analysis Of MATLAB Applications During Automated Hardware Synthesis for FPGAs* , Design Automation and Test in Europe, March 2001.

[8] A. Nayak, M. Haldar, A. Choudhary and P. Banerjee, *Parallelization of MATLAB Applications for a Multi-FPGA System,* Int. Symp. on FPGA Custom Computing Machines, FCCM, April 2001.

[9] P. Joisha and P. Banerjee, *Lattice-Based Type Determination in MATLAB, with an Emphasis on Handling Type Incorrect Programs*, Tech Report CPDC-TR-2001-03-001, Northwestern University, March 2001.

[10] G. D. Micheli,*Synthesis and Optimization of Digital Circuits,*pg. 185-265, ISBN-0-07-016333-2, McGraw-Hill, Inc.

[11] D. Gajski, N. Dutt, A. Wu and S. Lin,*High-Level Synthesis: Introduction to Chip and System Design,* ISBN-0-7923-9194-2, Kluwer Academic Publishers.

[12] The SystemC Initiative, www.systemc.org.

[13] *Xilinx Core Solutions Databook, Second Edition,* Xilinx Inc.

[14] *Altera 1999 Intellectual Property Catalog,* Altera Inc.

[15] G. De Micheli, *Hardware Synthesis from C/C++ Models,* Proc. Design, Automation and Test in Europe Conference and Exhibition, March 1999.

[16] A. Ghosh, J. Kunkel and S. Liao, *Hardware Synthesis from C/C++*, Proc. Design, Automation and Test in Europe Conference and Exhibition, 1999.

[17] G. Arnout, *C for System Level Design*, Proc. Design, Automation and Test in Europe Conference and Exhibition, March 1999.

[18] J. Hammes, B. Rinker, W. Bohm and W. Najjar, *Cameron: High Level Language Compilation for Reconfigurable Systems*, Proc. Parallel Architectures and Compilation Techniques (PACT'99), October 1999.

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

[20] M. Weinhardt and W. Luk, *Pipeline Vectorization for Reconfigurable Systems* , Proc. Field-Programmable Custom Computing Machines, April 2000.

[21] 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.

[22] Y. Li, T. Callahan, E. Darnel, R. Harr, U. Kurkure and J. Stockwood, *Hardware-Software Co-Design of Embedded Reconfigurable Architectures*, Proc. 37th DAC, June 2000.

[23] 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.

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

[25] C Level Design, Inc.,*System Compiler : Compiling ANSI C/C++ to Synthesis-ready HDL* , http://www.cleveldesign.com .