Adaptive Computing in NASA Multi-Spectral Image Processing

Mark L. Chang
Northwestern University
Evanston, IL 60208 USA
mchang@ece.nwu.edu

Scott A. Hauck
University of Washington
Seattle, WA 98195 USA
hauck@ee.washington.edu

Abstract

This paper presents our work accelerating a NASA image processing application by utilizing adaptive computing techniques. In our hand mapping of a typical algorithm, we were able to obtain well over an order of magnitude speedup over conventional software-only techniques with commercial off-the-shelf hardware. This work complements [6, 7] by widening the comparison between hardware and software approaches and updating the effort to include more current technologies. Additionally, we discuss the applicability of NASA applications to adaptive computing and present this work in the context of Northwestern's MATCH Project—a MATLAB to adaptive computing systems compiler.

1   Introduction

In 1991 NASA launched the *Earth Science Enterprise* initiative to study, from space, the Earth as a scientific system. The centerpiece of this enterprise is the Earth Observing System (EOS), which plans to launch its first satellite, *Terra*, before the turn of the century. With the launch of Terra, ground processing systems will have to process more data than ever before. In just six months of operation, Terra is expected to produce more data than NASA has collected since its inception [7].

As a complement to the Terra satellite, NASA is establishing the EOS Data and Information System (EOSDIS). Utilizing Distributed Active Archive Centers (DAACs), EOSDIS aims to provide a means to process, archive, and distribute science and engineering data with conventional high-performance parallel-processing systems.

Our work, combined with the efforts of others, strives to augment the ground-based processing centers with adaptive computing technologies such as workstations equipped with FPGA co-processors. These co-processors allow engineers to implement their algorithms in a hardware substrate with hardware-like speeds while maintaining the flexibility of software.

A trio of factors motivates the use of adaptive computing over custom ASICs: speed, adaptability, and cost. The current trend in satellites is to have more than one instrument downlinking data, which leads to instrument-dependent processing.   These processing "streams" involve many different algorithms producing many different "data products". Due to factors such as instrument mis-calibration, decay, damage, and incorrect pre-flight assumptions, these algorithms often change during the lifetime of the instrument. Thus, while a custom ASIC would most likely give better performance than an FPGA solution, it would be far more costly in terms of time and money due to an ASIC's cost, relative inflexibility, and long development cycle.

This paper presents the results we have obtained accelerating a NASA multi-spectral image processing application with an FGPA co-processor. We discuss the application, our implementation and results, and conclude with the MATCH Project, our automated MATLAB-to-FPGA compiler.

2   Multi-Spectral Image Classification

In this work, we focus on accelerating a typical multi-spectral image classification algorithm.   This algorithm uses multiple spectrums of instrument observation data to classify each satellite image pixel into one of many classes. This technique transforms the multi-spectral image into a form that is more useful for analysis by humans.   It also represents a form of data compression or clustering analysis.   In our

implementation these classes consist of terrain types such as urban, agricultural, rangeland, and barren. In other implementations these classes could be any significant distinguishing attributes present in the dataset. One proposed scheme to perform this automatic classification is the Probabilistic Neural Network classifier as described in [5]. In this scheme, each multi-spectral image pixel vector is compared to a set of "training pixels" or "weights" that are known to be representative of a particular class. The probability that the pixel under test belongs to the class under consideration is given by the following formula.

$$f(\vec{X} \mid S_k) = \frac{1}{(2\pi)^{d/2}\sigma^d} \frac{1}{P_k} \sum_{i=1}^{P_k} \exp\left[-\frac{(\vec{X}-\overrightarrow{W_{ki}})^T(\vec{X}-\overrightarrow{W_{ki}})}{2\sigma^2} \right]$$

Equation 1

Here, $\vec{X}$ is the pixel vector under test, $\vec{W}_{ki}$ is the weight i of class k, d is the number of bands, k is the class under consideration, σ is a data-dependent "smoothing" parameter, and P_k is the number of weights in class k. This formula represents the probability that pixel vector $\vec{X}$ belongs to the class S_k. This comparison is then made for all classes and the class with the highest probability indicates the closest match.

3   Implementation

The PNN algorithm was implemented in multiple software languages as well as on a COTS adaptive computing engine. The software languages included Matlab, Java, and C, while the hardware version was written in VHDL.

3.1   Software approaches

3.1.1   MATLAB (iterative)

Because of the high-level nature of MATLAB, this is likely the simplest and slowest implementation of the PNN algorithm. The MATLAB source code snippet is shown in Figure 1. This code uses none of the optimized vector routines that MATLAB provides. Instead, it uses an iterative approach, which is known to be slow in interpreted MATLAB. The best use for this approach is to benchmark improvements made by other approaches.

MATLAB has many high-level language constructs and features that make it easier for scientists to prototype algorithms, especially when compared to more "traditional" languages such as C or Fortran. Features such as implicit vector and matrix operations, implicit variable type declarations, and automatic memory management make it a natural choice for scientists during the development stages of an algorithm. Unfortunately, MATLAB is also much slower than its C and Fortran counterparts, due to its overhead. In later sections we discuss how the MATCH compiler can compile MATLAB codes to reconfigurable systems (such as DSPs and FPGAs), thus maintaining the simplicity of MATLAB while achieving higher performance.

```
for p=1:rows*cols
   fprintf(1,'Pixel: %d\n',p);
   % load pixel to process
   pixel = data( (p-1)*bands+1:p*bands);
   class_total = zeros(classes,1);
   class_sum   = zeros(classes,1);
   % class loop
   for c=1:classes
      class_total(c) = 0;
      class_sum(c) = 0;
      % weight loop
      for w=1:bands:pattern_size(c)*bands-bands
         weight = class(c,w:w+bands-1);
         class_sum(c) = exp( -(k2(c)*sum( (pixel-weight').^2 ))) + class_sum(c);
      end
      class_total(c) = class_sum(c) * k1(c);
   end
   results(p) = find( class_total == max( class_total ) )-1;
end
```

Figure 1: Matlab (iterative) code

3.1.2 MATLAB (vectorized)

To gain more performance, the iterative MATLAB code in Figure 1 was rewritten to take advantage of the vectorized math functions in MATLAB. The input data was reshaped to take advantage of vector addition in the core loop of the routine. The resultant code is shown in Figure 2.

```
% reshape data
weights = reshape(class',bands,pattern_size(1),classes);
for p=1:rows*cols
   % load pixel to process
   pixel = data( (p-1)*bands+1:p*bands);
   % reshape pixel
   pixels = reshape(pixel(:,ones(1,patterns)),bands,pattern_size(1),classes);
   % do calculation
   vec_res = k1(1).*sum(exp( -(k2(1).*sum((weights-pixels).^2)) ));
   vec_ans = find(vec_res==max(vec_res))-1;
   results(p) = vec_ans;
end
```

Figure 2: Matlab (vectorized) code

3.1.3 Java and C

A Java version of the algorithm was adapted from the source code used in [6]. Java's easy to use GUI development API (the Abstract Windowing Toolkit) made it the obvious choice to test the software approaches by making it simple to visualize the output. A C version was also developed based upon the Java version.

3.2 Hardware approaches

3.2.1 FPGA Co-Processor

The hardware used was the Annapolis Microsystems WildChild FPGA co-processor [2]. This board contains an array of Xilinx FPGAs—one "master" Xilinx 4028EX and eight Xilinx 4010E's, referred to as Processing Elements (PEs)—interconnected via a 36-bit crossbar and a 36-bit systolic array. PE0 has 1MB of 32-bit SRAM while PEs 1-8 have 512K of 16-bit SRAM each. The layout is shown in Figure 3. The board is installed into a VME cage along with its host, a FORCE SPARC 5 VME card. For reference, the Xilinx 4028EX and 4010E have 1024 and 400 Configurable Logic Blocks (CLBs), respectively. This, according to [9] is roughly equivalent to 28,000 gates for the 4028EX and 950 gates for the 4010E. This system will be referred to as the "MATCH Testbed".
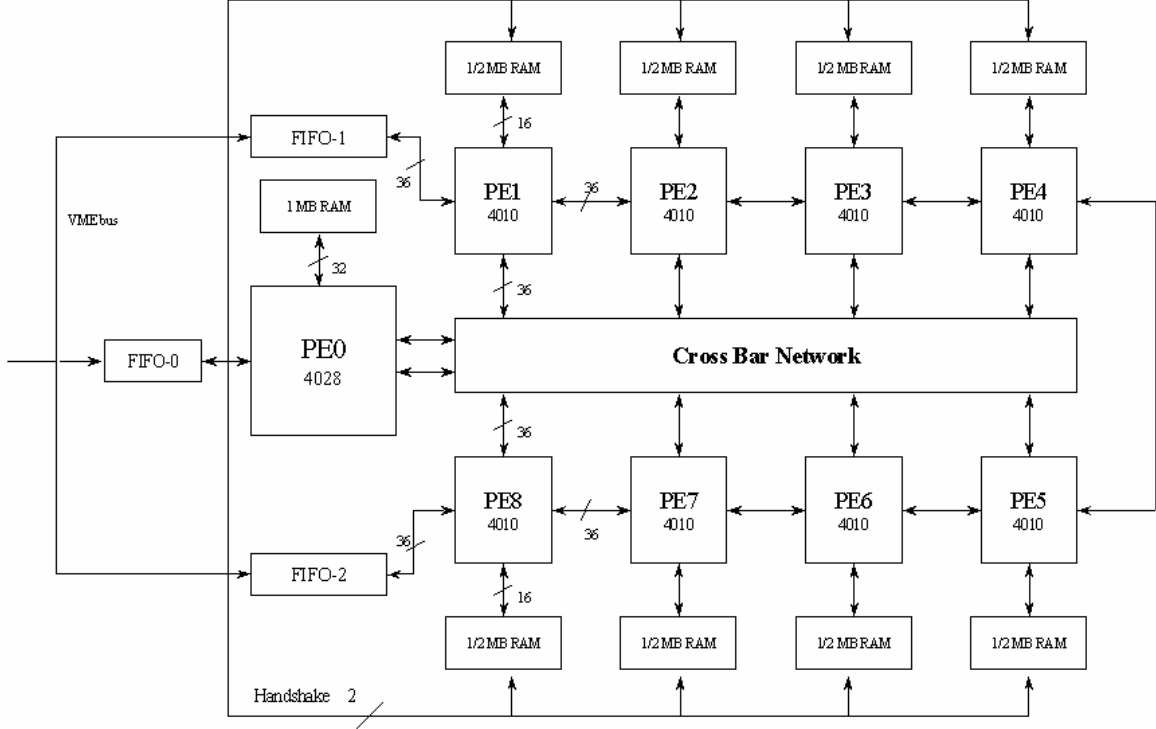
Figure 3: Annapolis Microsystems WildChild(tm) FPGA Co-processor

3.2.2 Initial mapping

The initial hand mapping of the algorithm is shown in Figure 4.

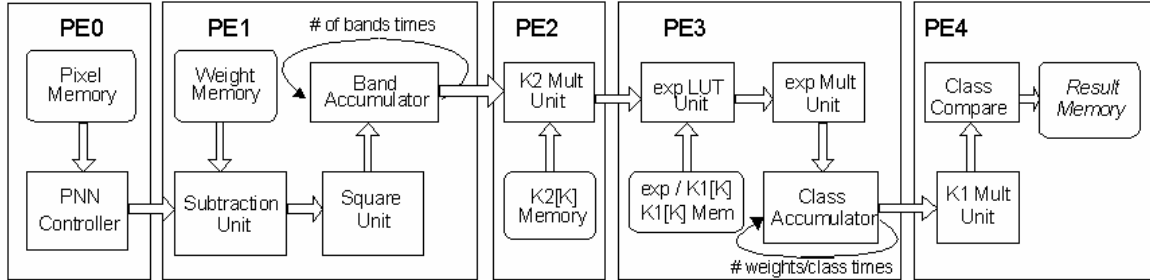(Note: $K_2 = \dfrac{1}{2\sigma^2}$ and $K_1 = \dfrac{1}{(2\pi)^{d/2}\sigma^d}$)

Figure 4: Initial FPGA mapping

PE	% Used
PE0	5%
PE1	67%
PE2	85%
PE3	82%
PE4	82%

Table 1: FPGA Utilization -- Initial mapping

The mappings were done completely by hand utilizing behavioral VHDL. The designs were simulated with Mentor Graphics' VHDL simulator using VHDL models supplied by Annapolis Microsystems. The

code was synthesized using Synplicity's Synplify synthesis tool. Placement and routing was accomplished with Xilinx Alliance tools. As shown in Figure 4, the computation for Equation 1 is spread across five FPGAs.

The architecture of the WildChild board suggests that PE0 be the master controller for any system mapped to the board. For example, PE0 is in control of the FIFOs, several global handshaking signals, and the crossbar. Thus, in our design, PE0 is utilized as the controller and "head" of the computation pipeline. PE0 is responsible for synchronizing the other processing elements, including beginning and ending the computation.

The pixels to be processed are loaded into the PE0 memory. When the system comes out of its Reset state, PE0 waits until it receives handshaking signals from all the slave processing elements. It then acknowledges the signal and sends a pixel to PE1. PE0 then waits until PE1 signals completion before it sends another pixel. This process repeats until all available pixels have been exhausted.

PE1 implements $(\vec{X} - \vec{W}_{ki})^T (\vec{X} - \vec{W}_{ki})$. As the input pixel data in our data set is 10 bits wide, we perform an 11-bit signed subtraction with each weight in each training class, and obtain a 20-bit result from the squaring operation. In the equation, $\vec{X}$ and $\vec{W}_{ki}$ represent vectors since the input data is multi-band. In our hardware, we accumulate across these bands, giving us a 22-bit result, which is then passed to PE2 along with the current class under test. When we have compared the input pixel with all weights from all classes, we signal PE0 to deliver a new pixel.

PE2 implements the K_2 multiplication, where $K_2 = \dfrac{1}{2\sigma^2}$, and σ is a class-derived smoothing parameter given with the input data set. The K_2 values are determined by the host computer and loaded into PE2 memory. The processing element uses the class number passed to it by PE1 and looks up the 16-bit K_2 value in memory. The 16-bit K_2 values are stored in typical fixed-point integer format, in this case shifted left by 18 bits. The result of the multiplication is a 38-bit value where bits 0...17 represent the fractional part of the result, and bits 18...37 represent the decimal part. To save work in follow-on calculations we examine the multiplication result. The follow-on computation is to compute $e^{(...)}$. For values larger than about 32, $e^{-32} = 12.6 * 10^{-15}$. Given the precision of later computations, we can and should consider this result to be zero. Conversely, if the multiplication result is zero, then the exponentiation should result in a 1. We test the multiplication result for these conditions and set flags appropriately before sending the multiplication result and the flags to the next PE. Only bits 1...22 of the multiplication result are sent to the next PE. Bit 0 is of such low order that it makes no significant impact on follow-on calculations.

PE3 implements the exponentiation in the equation. As our memories in the slave PEs are only 16-bits wide, we cannot obtain enough precision to cover the wide range of exponentiation required by the algorithm by using only one lookup table. Since we can break up e^{-a} into $e^{-b} * e^{-c}$, by using two table lookups and one multiplication, we can achieve a higher precision result. Given that the low-order bits are mostly fractional, we devote less memory space to them than the higher order, decimal, portions. Thus, out of the 22 bits, the lower 5 bits look up e^{-b} while the upper 17 bits look up e^{-c} in memory. If the input flags do not indicate that the result should be zero or one, these two 16-bit values are multiplied together and accumulated. When all weights for a given class have been accumulated, we send the highest 32 bits of the result (the most significant) to PE4. Additionally, in the next clock cycle we send the appropriate K_1 constant read from local memory.

PE4 performs the K_1 multiplication and class comparison. The accumulated result from PE3 is multiplied by the K_1 constant. This is the final value, $f(\vec{X} \mid S_k)$, for a given class, S_k. As discussed, this represents the probability that pixel vector $\vec{X}$ belongs to class S_k. PE4 compares each class result and keeps the class index of the highest value of $f(\vec{X} \mid S_k)$ for all classes. When we have compared all classes for the pixel under test, we write the maximum class index to memory. The final mapping utilization for each processing element is given in Table 1.

3.2.3   Host program

As with any co-processor, there must be a host processor that issues work to the co-processor. In this case it is the FORCE VME Board. The host program is written in C and is responsible for packaging the data for writing to the FPGA co-processor, configuring the processing elements, and collecting the results when the FPGA co-processor signals its completion. All communication between the host computer and the FPGA co-processor are via the VME bus.

3.2.4   Optimized mapping

To obtain higher performance we developed several optimizations to the initial mapping. The resultant optimized hand mapping is shown in Figure 5. Specifically, due to the accumulators in PE1 and PE3, there are different inter-arrival rates for each of the processing units. The processing elements following the accumulators thus have lower inter-arrival rates. In our data set there are four bands and five classes. Thus, PE2 and PE3 have inter-arrival rates of one-fourth that of PE1, while PE4 has an inter-arrival rate of one-twentieth that of PE1. Exploiting this we can use PE2 and PE4 to process more data in what would normally be their "stall", or "idle" cycles. This is accomplished by replicating PE1, modifying PE2 through PE4 to accept input every clock cycle, and modifying PE0 to send more pixels through the crossbar in a staggered fashion.

PE	% Used
PE0	5%
PE1-4	75%
PE5	85%
PE6	61%
PE7	54%
PE8	97%

Table 2: FPGA Utilization -- Optimized mapping

As shown in Figure 5, we now utilize all nine FPGAs on the WildChild board. FPGA utilization figures are given in Table 2. PE0 retains its role as master controller and pixel memory handler (Pixel Reader). Now, the Pixel Reader unit sends four pixels through the crossbar, one to each of four separate Subtract/Square units (PE1-PE4). The Pixel Reader then rotates the crossbar configuration to direct the output of each Subtract/Square unit to the K2 Multiplier unit in sequence.

The Subtract/Square units in PE1-PE4 remain virtually unchanged from the initial mapping, except for minor handshaking changes with the Pixel Reader unit. Each of the Subtract/Square units now must fetch one of the four different pixels that appear on the crossbar. This is accomplished by assigning each Subtract/Square unit a unique ID tag that corresponds to which pixel to fetch (0 through 3). The other change is to the output stage. Two cycles before output, each Subtract/Square unit signals the Pixel Reader unit to switch crossbar configurations to allow its output to be fed to the K2 Multiplier unit through the crossbar.

The K2 Multiplier unit, now in PE5, is pipelined in order to accept input every clock cycle. The previous design was not pipelined in this fashion. Now, instead of being idle for three out of four cycles, the K2 Multiplier unit operates every clock cycle.
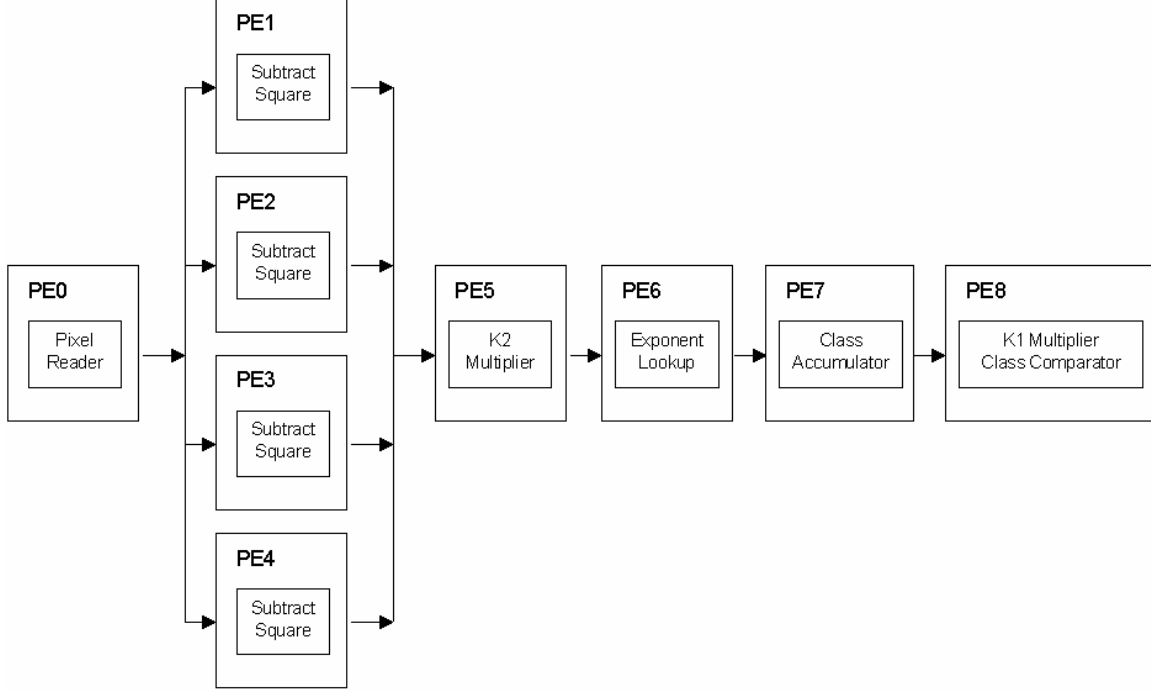
Figure 5: Optimized mapping

The Exponent Lookup unit changes significantly in this optimized mapping. Due to the need for a pipelined unit, we had to reduce the number of memory lookups within the PE. Having two lookups (one for e^{-b} and one for e^{-c}) would require stalling the pipeline and reducing overall performance. By wiring additional data and address lines through the crossbar to PE7's (the Class Accumulator unit) memory, we can perform two memory reads per clock cycle.

The class accumulator has been moved out of the exponentiation unit and into its own processing element. This is because the pipeline will be operating on four pixels at a time, thus requiring four separate accumulators. These accumulators are implemented in PE7, the Class Accumulator unit.

Finally, in PE8, we implement the K_1 multiplier and Class Comparison unit. Instead of receiving K_1 constants from the previous processing element, the constants are read from the local PE8 memory. This is possible because although all the PEs have the same number of memory locations, PE0 needs four times as many locations because each pixel consists of four different bands of data. In the result memory, we need only one location per pixel. PE8 accumulates class index maximums for each of the four pixels after the input data has been multiplied by the appropriate K_1 class constants. Like in the initial design, after all classes for all four pixels have been compared, the results are written to memory. For this design to fit within one Xilinx 4010E FPGA, we were forced to utilize a smaller 8x8 multiplier in multiple cycles to compute the final 38-bit multiplication result. This is possible because PE8 has a much slower inter-arrival rate than PE7, as it is after an accumulator.

4   Results

4.1   Test platform

The test platform for the software versions of the algorithm (MATLAB/Java/C) was an HP Visualize C-180 Workstation running HP-UX 10.20 with 128MB of RAM. The reference platform for the hardware version was the entire MATCH Testbed as a unit. This includes a 100MHz FORCE SPARC 5 VME card running

Solaris 2.5.1 (SunOS 5.5.1) with 64MB of RAM and the WildChild FPGA co-processor. Additional timings were performed on software versions running on the MATCH Testbed. Unfortunately, Matlab was not available for our MATCH Testbed at the time of this writing.

The Matlab programs were written as Matlab scripts (m-files) and executed using version 5.0.0.4064. The Java version was compiled to Java bytecode using HP-UX Java build B.01.13.04 and executed with the same version utilizing the Just-In-Time compiler. The C version was compiled to native HP-UX code using the GNU gcc compiler version 2.8.1.

4.2   Performance

Table 3 shows the results of our efforts in terms of number of pixels processed per second and lines of code required for the algorithm. The lines of code number includes input/output routines and excludes graphical display routines. Software version times are given for both the HP workstation and the MATCH Testbed where appropriate.

Platform	Method	Pixels/sec	Lines of code
HP	Matlab (iterative)	1.6	39
HP	Matlab (vectorized)	36.4	27
HP	Java	149.4	474
HP	C	364.1	371
MATCH	Java	14.8	474
MATCH	C	92.1	371
MATCH	Hardware+VHDL (initial)	1942.8	2205
MATCH	Hardware+VHDL (optimized)	5825.4	2480

Table 3: Performance results

When compared to more current workstations (1997) such as the HP, the optimized adaptive hardware implementation achieves a speedup of 3640 over the slowest implementation — the Matlab Iterative benchmark — 40 over the HP Java version, and 16 over the HP C version. An alternate comparison can be made to quantify the acceleration that the FPGA co-processor can provide over just the FORCE 5V Sparc system. This might be a better comparison for those seeking more performance from an existing reference platform by simply adding a co-processor such as the WildChild board. In this case, we compare similar technologies, dating from 1995. In this case, the hardware implementation is 390 times faster over the MATCH Testbed Java version, and 63 times faster than the MATCH Testbed executing the C version.

Of course, the cost in this approach is the volume of code and effort required. The number of lines metric is included in an attempt to gauge the relative difficulty of coding each method. This is a strong motivating factor for our work in the MATCH Project.

5   MATCH

The MATCH Project [3, 4] at Northwestern University encompasses much more than just the hand mapping of algorithms to an adaptive computing system. MATCH is a project whose goal is to develop a MATLAB compilation environment for distributed heterogeneous adaptive computing systems. The main objective of the MATCH project is to simplify tasks such as the one described here. Specifically, MATCH strives to make it easier for users to develop code that targets an adaptive computing system, such as our MATCH Testbed. The core effort is in the design of a MATLAB compiler that can generate code targeting COTS FPGA coprocessors, embedded microprocessors, and DSP coprocessors. We describe the MATCH compiler here briefly. For more detailed information, please refer to [3, 4].

As our work here shows, currently mappings to adaptive computing systems are time consuming, require personnel proficient in developing for the target hardware, and a low-level understanding of the algorithm architecture. To allow more users to take advantage of the performance potential of hardware acceleration through adaptive computing techniques, the MATCH compiler will compile MATLAB codes to a configurable computing system automatically. The most obvious benefit of this MATLAB compiler will

be in its translation of concise, easy to write, widely accepted MATLAB codes into fast hardware implementations with little or no user intervention.

More than just FPGAs, a complete configurable computing system would have other configurable components, such as DSPs and embedded processors, to share the workload. In most cases, purely FPGA-based systems are not ideal targets for complete algorithm implementation. Code that is infrequently used wastes valuable space on an FPGA, while operations such as floating point and complex arithmetic are better suited for a general purpose processor. In these cases, the MATCH compiler and an associated MATCH Testbed will be able to automatically optimize performance by partitioning work among a variety of processing elements. Here, the goal of the MATCH compiler is to optimize performance under resource constraints, and to obtain performance within a factor of 2-4 of the best manual approach.

The basic compiler approach begins with parsing MATLAB code into an intermediate representation, an Abstract Syntax Tree (AST). From this, we build a data and control dependence graph from which we can identify scopes for varying granularities of parallelism. This is accomplished by repeatedly partitioning the AST into one or more sub trees. Nodes in the resultant AST that map to predefined library functions map directly to their respective targets, and any remaining procedural code is considered user-defined procedures. A controlling thread is automatically generated for the system controller — in our case, the FORCE 5V processor. This controlling thread makes calls to the functions that have been mapped to any of the processing elements in the order defined by the data and control dependency graphs. Any remaining user defined procedures are automatically generated from the AST. The target for this generated code is determined by the scheduling system [8].

Currently, our compiler is capable of compiling MATLAB code to any of the three targets (DSP, FPGA, and embedded processor) as directed by user-level directives in the MATLAB code. The compiler currently exclusively uses library calls to cover the AST. We have developed library implementations of such common functions as matrix multiplication, FFT, and IIR/FIR filters for FPGA, DSP, and embedded targets. Work is ongoing in MATLAB type inferencing and scalarization to allow automatic generation of C+MPI codes and Register Transfer Level (RTL) VHDL which will be used to map user-defined procedures (non-library functions) to any of the resources in our MATCH Testbed. Work is also ongoing in integrating our scheduling system, Symphany [8], into the compiler framework to allow automatic scheduling and resource allocation without the need for user directives.

6 Conclusion

We have accelerated a typical NASA image-processing application by using an adaptive computing engine and a hand mapping of the algorithm. We have developed techniques and approaches for accelerating an existing hand mapping of an algorithm and have shown a more efficient implementation of our algorithm. Finally we have introduced the MATCH project which will enable scientists with little or no knowledge of VHDL and reconfigurable computing to obtain higher performance from their MATLAB code.

There are many motivating factors for developing a working relationship with NASA with respect to the MATCH project. The principle factor being that NASA applications are well suited to the MATCH project. Satellite ground-based applications, especially with the launch of Terra, will need to be accelerated in order to accomplish the scientific tasks that they were designed for in a timely and cost-effective manner. As we have shown, a typical image-processing application can be accelerated by several orders of magnitude over conventional software-only approaches by using adaptive computing techniques. But, to accomplish this requires someone that is knowledgeable in the use of FPGA co-processors and is comfortable in a hardware description language such as VHDL or Verilog. Very few scientists can be bothered to learn such languages and concepts. This too goes well with MATCH, as MATCH strives to enable users of a high-level language such as MATLAB to increase the performance of their MATLAB codes without intimate knowledge of the target hardware.

NASA has also demonstrated an interest in adaptive technologies, as can be witnessed by the existence of the Adaptive Scientific Data Processing (ASDP) group at NASA's Goddard Space Flight Center in Greenbelt, MD. The ASDP is a research and development project founded to investigate adaptive computing with respect to satellite telemetry data processing [1]. They have done work in accelerating

critical ground-station data processing tasks using reconfigurable hardware devices. Their work will only be complemented with the advancement of the MATCH compiler.

Finally, our work fits into the MATCH framework as a benchmark for the MATCH compiler. If the goal of the MATCH compiler is to get performance within a factor of 2-4 of the best hand-tuned mapping of an algorithm [4], then our work with multi-spectral image processing is not only a good benchmark for compiler performance, but it will also help identify functions and procedures necessary for real-world applications. Work is ongoing in the testing of the compiler and these codes.

7 Acknowledgments

8 Bibliography

[1]   Adaptive Scientific Data Processing. "The ASDP Home Page". http://fpga.gsfc.nasa.gov (22 Jul. 1999)

[2]   Annapolis Microsystems. *Wildfire Reference Manual*. Maryland: Annapolis Microsystems, 1998.

[3]   P. Banerjee, A. Choudhary, S. Hauck, N. Shenoy. "The MATCH Project Homepage". http://www.ece.nwu.edu/cpdc/Match/Match.html (1 Sept. 1999)

[4]   P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Chang, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden. *MATCH: A MATLAB Compiler for Configurable Computing Systems*. Technical report CPDC-TR-9908-013, submitted to IEEE Computer Magazine, August 1999.

[5]   S. R. Chettri, R. F. Cromp, M. Birhmingham. *Design of neural networks for classification of remotely sensed imagery*. Telematics and Informatics, Vol. 9, No. 3, pp. 145-156, 1992.

[6]   M. Figueiredo, C. Gloster. *Implementation of a Probabilistic Neural Network for Multi-Spectral Image Classification on a FPGA Custom Computing Machine*. 5th Brazilian Symposium on Neural Networks—Belo Horizonte, Brazil, December 9-11, 1998.

[7]   T. Graessle, M. Figueiredo. *Application of Adaptive Computing in Satellite Telemetry Processing*. 34th International Telemetering Conference—San Diego, California, October 26-29, 1998.

[8]   U.N. Shenoy, A. Choudhary, P. Banerjee. *Symphany: A Tool for Automatic Synthesis of Parallel Heterogeneous Adaptive Systems*. Technical report CPDC-TR-9903-002, 1999.

[9]   Xilinx, Inc.. *The Programmable Logic Data Book-1998*. San Jose, California: Xilinx, Inc.