

Adaptive Computing in NASA Multi-Spectral Image Processing

Master's Thesis
December, 1999

Mark L. Chang
Northwestern University

As submitted to
Northwestern University
Department of Electrical and
Computer Engineering
Evanston, IL USA

Adviser
Scott A. Hauck
University of Washington
Seattle, WA USA

Abstract

Typical NASA applications have image processing tasks that require high performance implementations. Interest in targeting FPGAs for high performance hardware-based implementations is growing, fueled by research that shows orders of magnitude speedups are possible within this domain. This thesis presents our work in utilizing adaptive computing techniques to accelerate a NASA image processing application. 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. We discuss this work in the context of the MATCH Project—an automatic compilation system that will allow users to compile MATLAB codes directly to FPGAs. Our work represents research involving real-world algorithms taken from NASA applications. By researching and implementing these algorithms by hand, we can develop optimization knowledge to significantly improve the results of the MATCH compiler.

1 Introduction

In 1991 NASA created the Earth Science Enterprise initiative to study the Earth as a scientific system. The centerpiece of this enterprise is the Earth Observing System (EOS), which successfully launched its first satellite, Terra, on December 18, 1999. 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]. It is therefore clear that the Terra satellite will pose an interesting processing dilemma.

As a complement to the Terra satellite, NASA has established 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 using adaptive computing technologies such as workstations equipped with FPGA co-processors. Increasingly, Field-Programmable Gate Arrays (FPGAs) are being used as an implementation medium spanning the development space between general-purpose processors and custom ASICs. FPGAs represent a relative middle ground, where development time is high compared to software, costs are low compared to ASICs (in low volume), and performance is higher than software, and sometimes approaches custom ASIC speeds.

A trio of factors motivates the use of adaptive computing in this domain 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 calibration error, 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 thesis describes our work in hand mapping a typical NASA image processing algorithm to an adaptive computing engine and demonstrates a significant speedup versus standard software-only execution. This work drives the development of our MATCH compiler, described in detail in later sections, by serving as both a benchmark for performance and a learning tool for incorporation of optimization techniques.

2 Field-Programmable Gate Arrays (FPGAs)

The typical computer sold today—be it a personal computer, workstation, or supercomputer—is based upon a general-purpose processor model. At the core of this model is a general-purpose processor (GPP) that responds to a fixed, well-defined instruction set. This instruction set directs the GPP to perform arithmetic, logic, branching, and data transfer functions. The nature of a GPP is that it implements all basic arithmetic and logic functions in order to support any basic computation. While this architecture provides great flexibility, it comes at a price: performance.

The performance loss can be attributed to the GPP’s lack of hardware-optimized structures for specific tasks. Without these specialized structures there will invariably be computations that will be sub-optimal in performance. For example, if a GPP did not implement a multiply in hardware, it would be forced to utilize an adder to perform the multiplication. For a word-sized multiply (ie. 32-bits), this method would require at least 32 additions and possibly 32 accumulations. In a single-cycle GPP, this would require somewhere between 32 and 64 cycles. On the other hand, if the GPP supported multiplication directly in hardware, it would complete the multiplication in a single cycle with perhaps a 4 cycle latency due to pipeline issues. Obviously, this is a great improvement. At this level of complexity, GPPs usually incorporate multipliers and dividers, as these functions are used by most applications.

Unfortunately, it is impossible for designers of GPPs to include hardware structures for every possible type of computation. While the current trend of computing power follows Moore's Law of growth, for some applications there will never be enough computing power within a GPP. For these applications, designers have oftentimes turned to custom ASICs, in the form of co-processors, to offload work from the GPP. Common examples are floating-point, audio, and graphics co-processors. While co-processors undoubtedly offer greater performance in most cases, they do so by limiting their optimizations to specific operations. Thus, a typical floating-point co-processor can only perform floating-point arithmetic and trigonometric math functions; an audio co-processor performs signal decoding and 3D spatialization; a graphics co-processor typically performs coordinate transformations, buffering, scaling, and texture/lighting mapping. All of these co-processors add functionality and performance to augment a GPP, but none are generalized to perform arbitrary tasks.

This inherent inflexibility makes the custom ASIC co-processor viable only for markets that have a high enough volume to defray the long development cycle and high costs of custom ASIC design. Arithmetic, audio, and graphics co-processors fall into this high-volume category. Unfortunately, for one-off and low-volume application accelerators, ASICs are not economically viable.

A solution that is growing in popularity is the Field-Programmable Gate Array, or FPGA. The FPGA is, as its name implies, a programmable gate array. Descendants of PAL/PLA structures, the typical FPGA is an array of logic blocks in a sea of interconnections, commonly referred to as an island-style FPGA. These logic blocks, dependent on the specific implementation, can typically implement functions as fine-grained as two-input Boolean functions and 3-input multiplexers, or as coarse-grained as full-adders and small ROM/RAM arrays. Not only are these logic blocks user-programmable, the routing mesh that connects these logic blocks together is also user-programmable.

With this architecture a designer can implement—given enough FPGA resources—*any arbitrary computation*. Additionally, most types of FPGAs are *re*-configurable, allowing the designer to utilize a single FPGA to perform a variety of tasks at different times. This is the essence of adaptive/reconfigurable computing.

3 MATCH

The one aspect of FPGAs impeding more widespread, innovative use, is the lack of quality, easy-to-use development tools. As attractive as the FPGA is as an implementation technology, the complexity of actually mapping to these devices often restricts their use to niche applications. These niche applications tend to be speed-critical applications that either do not have the volume to justify the ASIC design costs, or require some other unique characteristic of FPGAs, such as reconfigurability, or fault tolerance. Using current tools, developers must create by hand implementations in a hardware description language such as VHDL or Verilog, and map them to the target devices. Issues such as inter-chip communication and algorithm partitioning are also largely a manual process.

Many signal processing application developers, especially scientists, are unwilling to master the complexities of hardware description languages such as VHDL and Verilog. Unfortunately, these are the same scientists that would benefit the most from FPGA technologies. Within the domains of signal and image processing, though, high-level languages such as MATLAB have become popular in algorithm prototyping. Therefore a MATLAB compiler that can target hardware automatically would be invaluable as a development tool. This is the basis for the development of our MATCH compiler.

The objective of the MATCH (MATlab Compiler for Heterogeneous computing systems) compiler project is to make it easier to develop efficient codes for heterogeneous computing systems. Towards this end we are implementing and evaluating an experimental prototype of a software system that will take MATLAB descriptions of various applications and automatically map them onto a heterogeneous computing system. This system, dubbed the MATCH testbed, consists of embedded processors, digital signal processors and field-programmable gate arrays built from commercial off-the-shelf components. More detailed information can be found in sections 6.2 and 6.4.1.

The basic compiler approach begins with parsing MATLAB code into an intermediate representation known as 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, typically a

general-purpose processor. This controlling thread makes calls to the functions that have been mapped to 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 and implemented on an appropriate processing medium. The appropriate 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 instructed by user-level directives in the MATLAB code. The compiler uses exclusively library calls to cover the AST. The MATCH group has developed library implementations of such common functions as matrix multiplication, FFT, and IIR/FIR filters for FPGA, DSP, and embedded targets. Currently in development are MATLAB type inferencing and scalarization techniques 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. Integration of our scheduling system, Symphony [8], into the compiler framework to allow automatic scheduling and resource allocation without the need for user directives is also ongoing.

This MATLAB-to-FPGA/DSP compiler will allow developers to achieve high performance without requiring extensive hardware expertise. Unsophisticated users can quickly develop prototype algorithms and test them in an easy-to-master interpreted environment, fostering fast exploration of different algorithms.

4 Motivation

The idea of using hardware to accelerate an application is not a new one, nor is it the path of least resistance as compared to more conventional methods, such as increasing the computing power dedicated to a specific task. This is typically accomplished by increasing the clock speed of a single processing element or by increasing the number of processing elements and implementing a parallel version of the algorithm. Unfortunately, neither of these methods is particularly cost-effective and can require either a partial or a complete re-write of the code to take advantage of the hardware changes, especially in the case of implementing a parallel-processing version of an algorithm. But the true difficulty resulting from a hardware change is in the learning curve the new hardware poses and the subsequent time required to re-target the code, efficiently and effectively, to the new hardware. As mentioned earlier, it would be

necessary for the developer to have an intimate knowledge of both the target hardware and the algorithm specifics to implement the algorithm effectively.

The MATCH compiler is a more cost-effective method, in terms of both real money and time, to improve the performance of algorithms. By allowing users to write high level code (MATLAB), we can abstract the hardware details and hardware changes through the use of the compiler. Additionally, users do not lose their investment in current hardware, as much of the hardware targeted by the MATCH compiler is available as add-on hardware, functioning as co-processors that complement the host computer. Finally, these add-on components typically cost far less than general-purpose computers, yet deliver equivalent, if not better, performance.

A key factor in determining the success of an automated compilation environment such as the MATCH compiler is the quality of the resultant compiled code. To this end, we believe that applications will drive the compiler development. This is the aspect of the MATCH compiler we are focused upon in this work. Good driver applications benefit compiler development in many ways, such as:

- Defining critical functions that need to be supported by the compiler
- Providing sample codes that can be used to test the compiler
- Establishing a performance baseline
- Discovering performance bottlenecks and possible optimizations

As NASA scientists are a major potential target for using our MATCH compiler, we have sought out typical NASA applications and implemented one on our MATCH test platform. One of the first circuits we have investigated is a multi-spectral image processing task.

5 Multi-Spectral Image Classification

In this work, we focus on accelerating a typical multi-spectral image classification application. The core algorithm uses multiple spectrums of instrument observation data to classify each satellite image pixel into one of many classes. 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 underlying dataset. This technique, generally speaking, transforms

the multi-spectral image into a form that is more useful for analysis by humans, similar to data compression or clustering analysis.

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.

$$f(\bar{X} | S_k) = \frac{1}{(2\mathbf{p})^{d/2} \mathbf{s}^d} \frac{1}{P_k} \sum_{i=1}^{P_k} \exp \left[-\frac{(\bar{X} - \bar{W}_{ki})^T (\bar{X} - \bar{W}_{ki})}{2\mathbf{s}^2} \right]$$

Equation 1

Here, \bar{X} is the pixel vector under test, k is the class under consideration, \bar{W}_{ki} is the weight i of class k , d is the number of bands, \mathbf{s} 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 \bar{X} belongs to the class S_k . This comparison is then made for all classes. The class with the highest resultant probability indicates the closest match.

Applying the PNN algorithm to a sample of multi-band input data achieves the results shown in Figure 1.

6 Implementation

The complete PNN algorithm was implemented in a wide array of languages and platforms in order to accurately gauge the performance of adaptive computing techniques. The software languages used include MATLAB, Java, C, and C+MPI. Target platforms include single and parallel-processor workstations utilizing UltraSPARC, MicroSPARC, MIPS R10000, and HP PA-RISC processors, as well as embedded and adaptive computing resources, described in detail below.

6.1 Software platforms

The *reference platform* for the single-processor software versions of the algorithm was a 180MHz HP Visualize C-180 workstation running HP-UX 10.20 with 128MB of RAM. MATLAB, Java, and C versions were implemented and timed on this platform.

There were several platforms to test parallel versions of the algorithm. These included an eight-processor SGI Origin, a 16-node IBM SP2 running at 66MHz, and a network of four Texas Instruments DSPs. The SGI Origin utilizes 195MHz MIPS R10000 processors and has 1GB of main memory. Each node of the IBM SP2 includes 128MB of memory and an IBM Power2 processor. Finally, a Transtech TDM-428 VME board carries four 60MHz Texas Instruments TMS320C40 with 8MB of local memory per DSP.

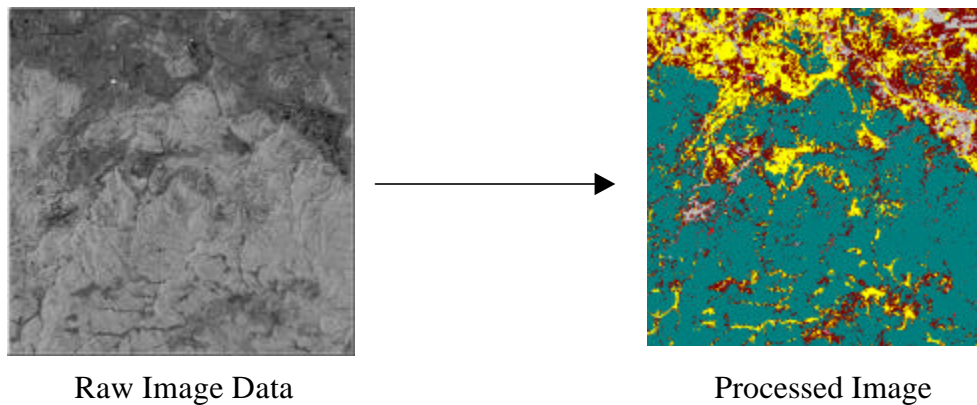


Figure 1: Results of processing. Each color in the right-hand image represents a different type of terrain detected.

6.2 Hardware platform

The reference platform for the hardware implementations was the entire MATCH testbed as a unit. This includes a 100MHz FORCE SPARC 5 VME card running Solaris with 64MB of RAM and the Annapolis Microsystems WildChild FPGA co-processor. A more detailed architecture overview is given in section 6.4.1.

6.3 Software approaches

6.3.1 MATLAB (iterative)

Because of the high-level interpreted nature of MATLAB, this is the simplest and slowest implementation of the PNN algorithm. The MATLAB source code snippet is shown in Figure 2. 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 MATLAB. On the other hand, the code is short and intuitive to write, making this the obvious first-cut at the algorithm definition. 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.

As can be seen from the code snippet of Figure 1, the core of the PNN algorithm is very simple and consists primarily of three nested loops. Starting with the outermost loop, which loops through all the pixels in the input image, we select a pixel and prepare to compare it to all the classes. The middle loop selects which class of weights we are comparing the pixel under test to. Finally, the calculations are carried out in the innermost loop, which iterates over all the weights in the particular class under consideration. There is one implicit vector computation in line 14 (`pixel-weight'`) where an element-by-element subtraction is performed between the pixel under test and the weight under consideration. These vector elements represent the different spectral bands within both the input data and the weight data. In our case four different bands are represented by each pixel and weight vector.

Finally, the last line of computation selects the maximum weight generated by the weighting function in the innermost loop and assigns it to the result vector. This represents the class with the closest match to the pixel under test.

6.3.2 MATLAB (vectorized)

To gain more performance, the iterative MATLAB code in Figure 2 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 3.

```

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 2: Matlab (iterative) code

```

% 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 3: Matlab (vectorized) code

MATLAB has optimized functions that allow us to trade memory for speed in many cases. In this instance, we reshape `pixel`—the pixel vector under test (four elements per pixel in our dataset)—to match the size of the `weights` vector (800 elements per class in our dataset). Then, we can eliminate the innermost loop from Figure 2 and use vectorized subtraction as in line 9 above. As the results show (see section 7.2), this vectorization greatly increases the performance of the MATLAB implementation.

6.3.3 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 full listing of the core Java routine can be found in Section 11.

A C version was easily implemented from the base Java code, as the syntax was virtually identical. The listing for the C version appears in Section 12.

6.3.4 Parallel Implementations

In order to fully evaluate the performance of hardware, it is important to benchmark our results against more traditional forms of application speedup. A typical method for obtaining acceleration in general purpose machines is to utilize multiple processors to perform computations in parallel. The only limits to the amount of speedup one can obtain using this method are the available hardware and the inherent parallelism within the algorithm.

Fortunately, the PNN algorithm has a high degree of parallelism. Specifically, the evaluation of a pixel in the PNN algorithm is not dependent on neighboring pixels. Thus, we can process, in parallel, as many pixels from the multi-band input data as our resources will allow. Furthermore, the parameters to which we compare these pixels under test do not change over the course of the computation, requiring little other than initial setup communications between processing elements, reducing overhead and giving better performance.

Parallel versions of the PNN algorithm were implemented on several different parallel-computing platforms, including the SGI Origin, IBM SP2, and a network of Texas Instruments DSPs. For each of these implementations a standardized library of parallel functions, known as the Message Passing Interface (MPI), was used. MPI defines a set of functions that can be used for communication and synchronization between processors that exist and operate in separate memory spaces. Additionally, the SGI Origin supports another mode of operation called shared memory parallel processing. In this instance, all processors “see” the same memory space, and therefore do not require explicit messages to share data or synchronize.

6.3.4.1 Shared memory

The shared memory implementation was the most similar to the single processor version of the code. The core of the code is available in section 13. The approach taken was to use static block scheduling to break up the input image data into equal-sized pieces for each of the processors. Since none of the processors would ever try to write to the same location in the result memory, no explicit locking of variables needed to occur. The inherent parallelism of the algorithm made this implementation straightforward. The only machine that supports this form of parallel processing was the SGI Origin.

6.3.4.2 C+MPI: IBM and SGI

The MPI version on general-purpose processors was more complex than the shared memory version. As mentioned earlier, MPI allows only explicit sharing of data through message passing between processors. Thus, one processor, the master, is in charge of all I/O and must distribute all relevant data to the other processors.

The distribution of work was the same as in the shared memory version: static block scheduling by rows. As can be seen in the code listing in section 14, the master processor distributes the weights and auxiliary data to the other processors through the function `MPI_Bcast(...)`, a single-source broadcast to all processors. The master processor then sends each slave processor its portion of the input image through a pair of functions, `MPI_Send(...)` and `MPI_Recv(...)`.

After each processor is finished working on its portion of the input image, it must send its results back to the master processor for output to the result file. This is accomplished again through the send/receive function pairs.

6.3.4.3 C+MPI: DSP

The final parallel implementation was on a network of four TI DSPs. As with most DSPs they were designed for embedded applications, thus requiring a host computer to download data and target code to them to perform any computation. This host computer was a SPARC5 FORCE board, the identical host computer used for the FPGA co-processor in following sections. Thus, there are two listings, one for the DSP code and one for the host computer (sections 15 and 16 respectively).

The host is in charge of I/O and distribution of data to the DSPs, while the DSPs perform the computation.

A few considerations in this version:

- the MPI implementation had a bug that did not allow transfers between DSPs to exceed 1024 bytes – the solution was to iteratively send smaller portions of large datasets
- RAM available to user programs on the DSPs was limited to about 4MB out of 8MB total
- memory allocation did not correctly report failure of the `malloc()` routine which resulted in board lock-up

- only two communication channels were available: host to DSP0 and DSP0 to the three slave DSPs
- only three MPI functions currently implemented: send, receive, barrier

These problems notwithstanding, the implementation was similar to the previous C+MPI versions, again utilizing static block scheduling by rows of input pixel data. The host controller loads all input data and sends everything except the pixel data to DSP0, the master DSP. The host then enters a loop, delivering a small enough portion of the input image data to the master DSP per iteration.

The master DSP then sends everything but the pixel data it received to each of the slave DSPs, and divides the pixel data by rows and delivers the appropriate portions to each DSP. After computation is completed, each of the slave DSPs delivers the result to the master DSP, which in turn sends the aggregate result to the host controller. This process iterates until all pixels have been processed.

6.4 Hardware approaches

6.4.1 FPGA Co-Processor

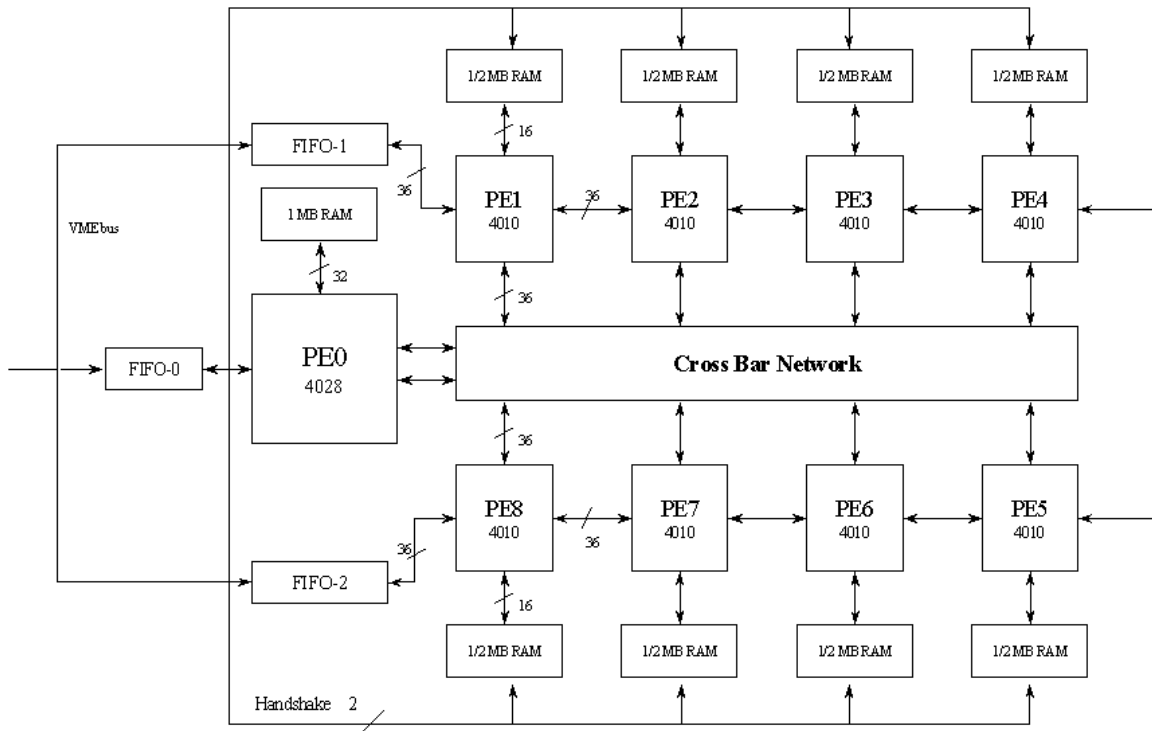


Figure 4: Annapolis Microsystems WildChild(tm) FPGA Co-processor

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 “slave” 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 4. 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”.

6.4.2 Initial mapping

The initial hand mapping of the algorithm is shown in Figure 5. (Note the coefficients from Equation 1

have been collapsed into two variables for easier reference: $K_2 = \frac{1}{2s^2}$ and $K_1 = \frac{1}{(2p)^{d/2} s^d}$)

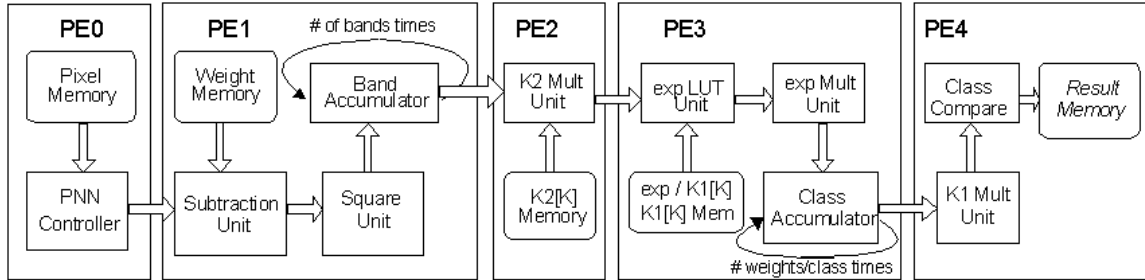


Figure 5: Initial FPGA mapping

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

Table 1: FPGA Utilization -- Initial mapping

The mappings were written in behavioral VHDL and simulated with the Mentor Graphics VHDL simulator to verify correctness before actual hardware implementation. The code was synthesized using Synplicity’s Synplify synthesis tool. Placement and routing was accomplished with Xilinx Alliance (M1) tools. As shown in Figure 5, the computation for Equation 1 is spread across five FPGAs.

The architecture of the WildChild board strongly suggests that PE0 be the master controller for any system mapped to the board as 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 and beginning and ending the computation.

Pixels to be processed are loaded into the PE0 memory. When the system comes out of its Reset state (such as upon bootup), PE0 waits until it receives handshaking signals from the slave processing elements. It then acknowledges the signal and sends a pixel to PE1 via the 36-bit crossbar. A “pixel” in this sense, consists of four 10-bit values read from memory, each value representing a different spectral band. PE0 then waits until PE1 signals completion before it sends another pixel. This process repeats until all available pixels have been exhausted. At the end of the entire computation, PE0 raises the host interrupt, which signals completion to the host program.

PE1 implements $(\bar{X} - \bar{W}_{ki})^T (\bar{X} - \bar{W}_{ki})$. The multi-band input pixel arrives from PE0 via the crossbar and is stored in local registers, while the multi-band weight values are read from local memory. The input pixel in our data set is 10 bits wide by 4 elements (bands) deep. We perform an 11-bit element-wise signed subtraction with each weight in each training class, and obtain a 20-bit result from the squaring operation. We accumulate across the spectral bands, giving us a 22-bit scalar result. This result is then passed through the 36-bit systolic connection to PE2 along with a value representing the current class under test. We continue to process in the same fashion for each class in our data set, outputting a 22-bit scalar result for each class comparison. 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 \mathbf{S} 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. PE2 uses the class number passed to it by PE1 to index into local memory and retrieve a 16-bit K_2 value. Examination of the test data set reveals the range of K_2 to be between 0.125 and 0.003472. Limited to 16-bit precision because of the memory architecture of the WildChild board, the 16-bit K_2 values are stored in typical fixed-point integer format, in this case shifted left by 18 bits to gain precision. 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 and area in follow-on calculations, we examine the multiplication

result. The follow-on computation is to compute $e^{(\dots)}$ in PE3. For values larger than about 32 the exponentiation results in: $e^{-32} = 12.6 * 10^{-15}$, a very insignificant value. 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 1. We test the multiplication result for these conditions and set “zero” and “one” flags. Therefore, we need not send any values larger than 32, resulting in any bits higher than the 23rd bit to be extraneous. Additionally, the lowest-order bit, bit 0, represents such a small value, it will not impact follow-on calculations. Thus, only bits 1...22 of the multiplication result are sent to PE3, along with the flags indicating “zero” and “one” conditions.

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 in the algorithm by using only one lookup table. Since we can break up e^{-a} into $e^{-b} * e^{-c}$ by using two consecutive 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 suggest the result to be either zero or one, these two 16-bit values obtained from memory 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. This limitation is due to the width of the systolic connection between processing elements. In the next clock cycle we also send the K_1 constant associated with the class under consideration. Like K_2 in PE2, K_1 is class dependent, determined by the host, and loaded into the memory prior to the computation. This constant is sent to the next PE because there was insufficient space in PE3 to implement another multiplier.

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, $f(\vec{X} | S_k)$, 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 $f(\vec{X} | S_k)$ for all classes. When we have compared all

classes for the pixel under test, we assign the class with the highest probability to the next consecutive memory location.

The final mapping utilization for each processing element is given in Table 1.

6.4.3 Host program

As with any co-processor, there must be a host processor of some sort to issue work. In this case, it is the FORCE VME Board. The host program for the FORCE VME board 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.

The host program first clears all the memories before loading them. It then computes both constants and loads them into the appropriate memory locations of PE2 and PE3. The exponential look-up values have been precomputed and written to data files that are subsequently loaded into in PE3's memory. Likewise, the weights are read from the input data file and loaded into PE1's memory. Finally, the multi-band pixel data is too large to completely fit within the memory of PE0. Therefore, the host program simply iteratively loads portions of the pixel data into PE0's memory and restarts the computation. At the end of each iteration, the host program reads the result memory from PE4, clears it, and begins again.

6.4.4 Optimized mapping

To obtain higher performance, we applied a number of optimizations to the initial mapping. The resultant optimized hand mapping is shown in Figure 6. The accumulators in PE1 and PE3 slow the inter-arrival rates of follow-on calculations, giving rise to one optimization opportunity. 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. This mapping takes advantage of optimization opportunities present in the algorithm.

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 6, we now use 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). The biggest modification is that the Pixel Reader unit now sends four pixels through the crossbar instead of one—one destined for each of four separate Subtract/Square units (PE1-PE4). To facilitate the pixel reaching the right Subtract/Square unit, each pixel is tagged with an ID that corresponds to which unit should fetch the current pixel.

Another new feature has to do with multiplexing the output of each of the Subtract/Square units into the K2 Multiplier unit. This is again accomplished through handshaking signals. When one of the Subtract/Square units has completed a class, it signals PE0. PE0 in turn reconfigures the crossbar to direct the output from the signaling Subtract/Square unit to the K2 Multiplier unit.

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 fetches 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 Subtract/Square unit only fetches the pixel on the crossbar if its internal ID matches the ID of the pixel on the crossbar. 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 in order to allow the Subtract/Square 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.

The Exponent Lookup unit has not changed in function, but has changed considerably in implementation. Since results are now coming every clock cycle from the K2 Multiplier unit, exponentiation must be

pipelined in order to accept input every clock cycle. In the previous implementation, local memory was accessed twice for each multiply—once for e^{-a} , and once for e^{-b} . This would not work in the current design as it would not be possible to accept input every clock cycle. The change made was to utilize 5 bits of the crossbar to send the lookup to PE7, the Class Accumulator unit. Since the Class Accumulator unit does not utilize memory, we can use that memory and map the address for the e^{-b} lookup through the crossbar and into the RAM address in PE7. Additionally, we map the resultant 16-bit value read from memory through the crossbar again and multiply the two 16-bit values together in another pipeline stage. The result is passed to the class accumulator through the systolic array connection.

The Class Accumulator unit is slightly modified to work in a higher-throughput system. Instead of accumulating a single pixel at a time, the accumulator must work on four pixels at once, one for each of the Subtract Square units. This is accomplished by creating a vector of accumulator registers and working every clock cycle. At the end of the accumulation, the results are sent via the systolic array connection to the K1 Multiplier/Class Comparator unit.

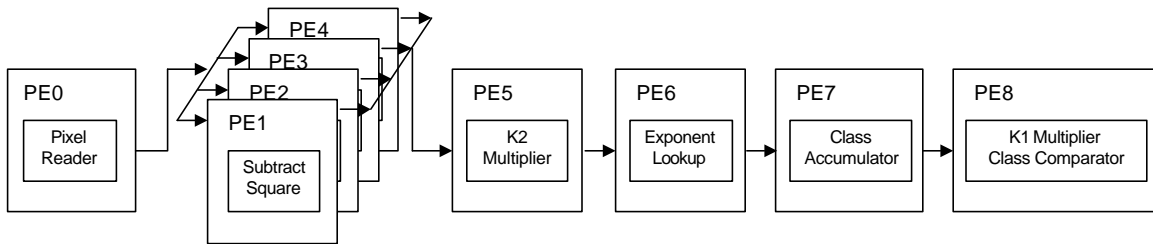


Figure 6: Optimized mapping

The K1 Multiplier/Class Comparator unit, unlike the rest of the system, does not need to function every clock cycle. The Class Accumulator, by its nature, does not produce an output every clock cycle. Therefore, we are free to use a slow but area-efficient multi-cycle multiplier to perform the K1 multiplication. Instead of receiving K_1 constants from the previous processing element, the constants are read from local memory. This is possible because we do not utilize all memory locations in the local memory, allowing us to store constants. This unit 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, the results are written to memory after all classes for all four pixels have been compared.

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.

7 Results

7.1 Test environment

The software and hardware test platforms have been described in detail in sections 6.1 and 6.2. Timing was only measured for computational sections of the algorithm, thus ignoring file I/O time, which varies widely for each platform. Timing was included in each implementation of the algorithm, which utilized a simple wall-clock method to measure elapsed time. In addition to the implementations described above, additional timings were performed on software versions (Java and C) running on the MATCH testbed. Unfortunately, Matlab was not available for our MATCH testbed nor the parallel machines (SGI Origin, IBM SP2, DSPs) at the time of this writing.

For the reference platform (HP) 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 on the HP platform

For the parallel platforms, the native SGI, IBM, and DSP parallel C compilers were used. For the additional software implementation timings on the MATCH testbed, gcc version 2.7.2.3 and Java version 1.1.8 compilers were used.

7.2 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” value 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.

When compared to more current technologies (1997), such as the HP workstation, the optimized hardware implementation achieves the following speedups: 3640 versus the Matlab Iterative benchmark, 40 versus the HP Java version, and 16 versus the HP C version. This comparison is somewhat of a direct CPU to CPU comparison.

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 (single processor)

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. Comparing in this fashion, 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.

Platform	Processors	Pixels/sec	Lines of Code
SGI Origin (shared)	1	962	509
SGI Origin (shared)	2	1922	509
SGI Origin (shared)	4	3837	509
SGI Origin (shared)	8	7441	509
SGI Origin (MPI)	1	965	496
SGI Origin (MPI)	2	1928	496
SGI Origin (MPI)	4	3862	496
SGI Origin (MPI)	8	7592	496
IBM SP2 (MPI)	1	322	496
IBM SP2 (MPI)	2	629	496
IBM SP2 (MPI)	4	1259	496
IBM SP2 (MPI)	8	2506	496
DSP (MPI)	4	245	699

Table 4: Performance results (multiple processor)

Finally, one can compare traditional methods of application acceleration, namely parallel computing methods, with our hardware implementation. From Table 4 we can see that while the SGI outperforms the optimized hardware version, it is only by 30%. One must also take into account the fact that the SGI Origin cost Northwestern University approximately \$200,000 in 1997. Comparatively, the MATCH testbed was about \$50,000, which includes the cost of the VME chassis, the embedded PowerPC processors, the FPGA co-processor, and the DSP board. In terms of performance, the hardware implementation attained approximately the same performance as the algorithm running on six processors of the SGI Origin. Compared to the IBM SP2, which was \$500,000 in 1996, the FPGA version handily

outperformed even the eight-processor implementation. And finally, versus the DSP board in the MATCH testbed, the FPGA implementation was more than 20 times faster.

One conclusion we can draw from these results is that the hardware implementation is a much more cost-effective means, from a hardware investment standpoint, to achieve higher performance. It also important to note that no MATLAB implementations would have been able to take advantage of multiple processors automatically, thus reducing the performance to that of a single processor. In that case, the hardware implementation is six times faster than a single SGI Origin processor, and nearly 20 times faster than a single node of the SP2.

Of course, a significant cost in the hardware 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. With a speedup potential of over three orders of magnitude over what NASA scientists might typically use (MATLAB iterative) to prototype an algorithm, it is clear that reconfigurable computing is an approach worth considering.

8 Conclusion

There is little doubt that FPGAs will play a role in high-performance computing in NASA's future. Be it in ground-based data processing centers, or on-board processing for spacecraft, the need for higher performance computation is growing. But reality is that the level of support for these technologies is decidedly low when compared to their software counterparts. This is the deficiency we are trying to address with the development of the MATCH compiler and its associated framework.

Satellite ground-based processing, 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 have the time to learn such languages and concepts. Fortunately, MATCH will enable users of a high-level language such as MATLAB to

increase the performance of their codes without intimate knowledge of the target hardware, thus enabling scientists to harness the power of reconfigurable computing.

NASA has 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 be complemented with the advancement of the MATCH compiler, ultimately exposing more NASA scientists to the benefits of reconfigurable computing.

A greater understanding of how developers will use the MATCH compiler will yield a better development tool that in turn yields better results. Endeavoring to make the compiler more efficient in targeting FPGA resources for NASA algorithms, we will continue to research driver applications. Using NASA applications as drivers will allow us to investigate optimization techniques that are immediately relevant to NASA scientists, who typically work with image and signal processing algorithms. This research will result in a compiler that is capable of producing higher-quality results for NASA applications by applying learned optimization techniques automatically.

In this work we have shown that reconfigurable computing can offer several orders of magnitude in speedups over software-only techniques for a typical NASA image processing application. While these speedups are not trivial, nearly another order of magnitude is possible through clever optimizations of the initial mapping. These optimization techniques are invaluable in the development of a smart, usable compiler.

9 Acknowledgments

The author would like to thank Marco A. Figueiredo and Terry Graessle from the ASDP at NASA Goddard for their invaluable cooperation. We would also like to thank Prof. Clay Gloster of North Carolina State University for his initial development work on the PNN algorithm. This research was funded in part by DARPA contracts DABT63-97-C-0035 and F30602-98-0144, and NSF grants CDA-9703228 and MIP-9616572.

10 References

- [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. Crompton, M. Birmingham. *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 Telemetry Conference—San Diego, California, October 26-29, 1998.
- [8] U.N. Shenoy, A. Choudhary, P. Banerjee. *Symphony: 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*. California: Xilinx, Inc.

11 Appendix A – Java code listing for PNN routine

```
/* Java classification routine core */
public void JAVA_FP_Classify()
{
    String time_string, row_string;
    Long initial_time = (new Date()).getTime(),
        row_initial_time=0,
        row_elapsed_time=0,
        row_final_time=0,
        elapsed_time;

    double tmp,
        tmpfl[] = new double[raw_data.num_classes],
        sigma_x,
        maxval_x,
        curval_x,
        term_x,
        exp_m_x,
        class_sum_x;

    int Class,
        weight,
        diff,
        pix=0,
        start;

    long psum;

    this.mythread = Thread.currentThread();
    this.mythread.setPriority(Thread.MIN_PRIORITY);

    ((Component)parent).setCursor( new Cursor(Cursor.WAIT_CURSOR) );

    // pop up our timing dialog
    PnnDialog dialog = new PnnDialog(parent);
    dialog.setVisible(true);
    dialog.setTitle("Local JAVA Classification");
    System.out.println("Local JAVA Classification");

    // precompute some sigma-based values
    tmp = Math.pow(2.0*(22.0/7.0) , (raw_data.bands/2.0) );
    for (Class = 0; Class < raw_data.num_classes; Class++)
        tmpfl[Class] = tmp * Math.pow(raw_data.Class[Class].sigma, raw_data.bands);
    sigma_x = Math.pow(raw_data.Class[--Class].sigma, 2);

    // classify entire image, row by row
    for (int y = 0; y < raw_data.rows; y++)
    {
        // classify a row
        start = (y*raw_data.cols);
        row_elapsed_time = ( row_final_time - row_initial_time )/1000;
        row_initial_time = new Date().getTime();
        row_string = "Last: "+Long.toString( row_elapsed_time )+"s";

        for(int x=start; x<(start+raw_data.cols); x++)
        {
            // get elapsed time samples and display times in dialog
            // but don't do it too often, or things get too slow!
            if ( (x%50) == 0 )
            {
                elapsed_time = ((new Date()).getTime()) - initial_time)/1000;
                time_string = "Elapsed: " + Long.toString( elapsed_time ) + "s " +
                    row_string;
                dialog.TextLabel.setText(time_string);
            }

            maxval_x = -1;
            weight=0;

            for(Class=0; Class<raw_data.num_classes; Class++)
            {
                class_sum_x = 0;
            }
        }
    }
}
```

```

for(int pattern=0; pattern<raw_data.Class[Class].patterns; pattern++)
{
    psum = 0;

    for(int bands=0; bands<raw_data.bands; bands++)
    {
        diff = raw_data.pixels[pix+bands] - raw_data.weights[weight++];
        psum += diff * diff;
    } // end of bands loop

    term_x = ((double)psum)/(2.0*sigma_x);
    exp_m_x = Math.exp(-term_x);
    class_sum_x += exp_m_x;

} // end of pattern loop

if (raw_data.Class[Class].patterns == 0)
{
    curval_x = -1;
}
else
{
    curval_x = class_sum_x / (tmpfl[Class] *
        raw_data.Class[Class].patterns);
}

// assign color to pixel based on the largest class score.
if (maxval_x < curval_x)
{
    maxval_x = curval_x;
    pixels[x] = color_set.ClassColors[raw_data.Class[Class].ClassNumber];
}

} // end of Class loop

    pix += raw_data.bands; // increment the raw data pixel "pointer"
}

// a row is finished; send the newly classified row to the screen
source.newPixels(0, y, raw_data.cols, 1);
parent.repaint(0);
mythread.yield();
row_final_time = new Date().getTime();

elapsed_time = ((new Date().getTime()) - initial_time )/1000;
time_string = "Finished in " + Long.toString( elapsed_time ) + "s";

dialog.TextLabel.setText(time_string);
dialog.setDismissButton(true);
Toolkit.getDefaultToolkit().beep();

// how do we get this to appear without mouse motion?
((Component)parent).setCursor( new Cursor(Cursor.DEFAULT_CURSOR) );
}

```

12 Appendix B – C code listing for PNN routine

```

/* classifies using local floating-point algorithm... */
void pnn(void)
{
    double tmp, sigma_x, maxval_x, curval_x, term_x, exp_m_x, class_sum_x;
    double *tmpfl;
    int class, weight, diff, pix=0, start;
    int x, y, pattern, bands;
    long psum;

    printf("Beginning classification process\n");
    tmpfl = (double *)malloc(data.num_classes*sizeof(double));

    /* precompute some sigma-values */
    tmp = pow(2.0*M_PI, (data.bands/2.0) );
    for( class=0; class< data.num_classes; class++ ) {

```

```

    tmpfl[class] = tmp * pow(data.class[class].sigma, data.bands);
}
sigma_x = pow(data.class[--class].sigma,2);

/* classify entire image, row by row */
printf("Percent done...\n");
for (y = 0; y<data.rows; y++) {
    fprintf(stderr, "\r%.2f%", (float)100*y/data.rows);
    /* classify a row */
    start = (y*data.cols);
    for(x=start; x<(start+data.cols); x++) {
        maxval_x = -1;
        weight=0;

        for(class=0; class<data.num_classes; class++) {
            class_sum_x = 0;

            for(pattern=0; pattern<data.class[class].patterns; pattern++) {
                psum = 0;

                for(bands=0; bands<data.bands; bands++) {
                    diff = data.pixels[pix+bands] - data.weights[weight++];
                    psum += diff * diff;
                } /* end of bands loop */

                term_x = ((double)psum)/(2.0*sigma_x);
                exp_m_x = exp(-term_x);
                class_sum_x += exp_m_x;
            } /* end of pattern loop */

            if (data.class[class].patterns == 0) {
                curval_x = -1;
            }
            else {
                curval_x = class_sum_x / (tmpfl[class] * data.class[class].patterns);
            }

            /* assign color to pixel based on the largest class score. */
            if (maxval_x < curval_x) {
                maxval_x = curval_x;
                image[x] = local_color[data.class[class].num];
            }
        } /* end of Class loop */
        pix += data.bands; /* increment the raw data pixel "pointer" */
    } /* end of x (row) loop */

    /* a row is finished; send the newly classified row to the screen
       source.newPixels(0, y, raw_data.cols, 1);
    */
} /* end of y (entire image) block */
printf("\rDone\n");
}

```

13 Appendix C – Shared memory listing

```

/* classifies using local floating-point algorithm... */
void pnn_parallel(void)
{
    double tmp, sigma_x, maxval_x, curval_x, term_x, exp_m_x, class_sum_x;
    double *tmpfl;
    int class, weight, diff, pix, start;
    int x, y, pattern, bands;
    int rows_per_proc, myid, numprocs, start_row, end_row;
    long psum;

    /* static block partitioning */
    myid = m_get_myid();
    numprocs = m_get_numprocs();
    rows_per_proc = data.rows/numprocs;

```

```

start_row = myid*rows_per_proc;
end_row   = (myid+1)*rows_per_proc;

/* advance pixel input to correct offset */
pix = start_row*data.bands*data.cols;

printf("Processor %d: rows %d to %d / pixel %d\n",myid,start_row,end_row-1,pix);
tmpfl = (double *)malloc(data.num_classes*sizeof(double));

/* precompute some sigma-values */
tmp = pow(2.0*M_PI, (data.bands/2.0) );
for( class=0; class< data.num_classes; class++ ) {
    tmpfl[class] = tmp * pow(data.class[class].sigma, data.bands);
}
sigma_x = pow(data.class[--class].sigma,2);

/* classify our block of the image, row by row */
for( y = start_row; y<end_row; y++ ) {
    /* classify a row */
    start = (y*data.cols);

    for(x=start; x<(start+data.cols); x++) {
        maxval_x = -1;
        weight=0;

        for(class=0; class<data.num_classes; class++) {
            class_sum_x = 0;

            for(pattern=0; pattern<data.class[class].patterns; pattern++) {
                psum = 0;

                for(bands=0; bands<data.bands; bands++) {
                    diff = data.pixels[pix+bands] - data.weights[weight++];
                    psum += diff * diff;
                } /* end of bands loop */

                term_x = ((double)psum)/(2.0*sigma_x);
                exp_m_x = exp(-term_x);
                class_sum_x += exp_m_x;
            } /* end of pattern loop */

            if (data.class[class].patterns == 0) {
                curval_x = -1;
            }
            else {
                curval_x = class_sum_x / (tmpfl[class] * data.class[class].patterns);
            }

            /* assign color to pixel based on the largest class score. */
            if (maxval_x < curval_x) {
                maxval_x = curval_x;
                image[x] = data.class[class].num;
            }
        } /* end of Class loop */
        pix += data.bands; /* increment the raw data pixel "pointer" */
    } /* end of x (row) loop */
} /* end of y (entire image) block */
m_sync();
}

```

14 Appendix D – MPI listing

```

/* classifies using local floating-point algorithm... */
void pnn(void)
{
    double tmp, sigma_x, maxval_x, curval_x, term_x, exp_m_x, class_sum_x;
    double *tmpfl;
    int class, weight, diff, pix=0, start;
    int x, y, pattern, bands;
    long psum;

```

```

/* values used by everyone that need to be broadcast
   Just recreate the data sets with partial pixels and weights */

/* parallel variables */
int myid, nprocs, i, p;
MPI_Status status;

/* setup parallel parameters */
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
MPI_Comm_size(MPI_COMM_WORLD,&nprocs);

MPI_Bcast( &data.bands, 1, MPI_INT, 0, MPI_COMM_WORLD );
MPI_Bcast( &data.patterns, 1, MPI_INT, 0, MPI_COMM_WORLD );
MPI_Bcast( &data.num_classes, 1, MPI_INT, 0, MPI_COMM_WORLD );
MPI_Bcast( &data.rows, 1, MPI_INT, 0, MPI_COMM_WORLD );
MPI_Bcast( &data.cols, 1, MPI_INT, 0, MPI_COMM_WORLD );
MPI_Bcast( &data.imagefile_bands, 1, MPI_INT, 0, MPI_COMM_WORLD );
MPI_Bcast( &data.bytes_per_entry, 1, MPI_INT, 0, MPI_COMM_WORLD );
MPI_Bcast( &data.max_weight, 1, MPI_INT, 0, MPI_COMM_WORLD );
MPI_Bcast( &data.plength, 1, MPI_INT, 0, MPI_COMM_WORLD );

/* update values for parallel processing */
data.plength /= nprocs;
data.rows /= nprocs;

/* everyone else prepare the buffers */
if( myid != 0 ) {
    data.weights = (int *)malloc(data.max_weight*sizeof(int));
    data.pixels = (int *)malloc(data.plength*sizeof(int));
    data.class = (Class *)malloc(data.num_classes*sizeof(Class));
    image = (int *)malloc(data.rows*data.cols*sizeof(int));
}

/* send data to all processors */
if( myid == 0 ) {
    for( p = 1; p<nprocs; p++ ) {
        /* class array */
        for( i=0; i<data.num_classes; i++ ) {
            MPI_Send(&( data.class[i].num ), 1, MPI_INT, p, 0, MPI_COMM_WORLD);
            MPI_Send(&( data.class[i].patterns ), 1, MPI_INT, p, 0, MPI_COMM_WORLD);
            MPI_Send(&( data.class[i].sigma ), 1, MPI_INT, p, 0, MPI_COMM_WORLD);
        }
        /* weights */
        MPI_Send(data.weights, data.max_weight, MPI_INT, p, 0, MPI_COMM_WORLD);
        /* pixels */
        MPI_Send(&( data.pixels[p*data.plength] ), data.plength, MPI_INT, p, 0,
MPI_COMM_WORLD);
    }
}
/* receive */
else {
    /* the classes */
    for( i=0; i<data.num_classes; i++ ) {
        MPI_Recv(&( data.class[i].num ), 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        MPI_Recv(&( data.class[i].patterns ), 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        MPI_Recv(&( data.class[i].sigma ), 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    }
    /* weights */
    MPI_Recv(data.weights, data.max_weight, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    /* pixels */
    MPI_Recv(data.pixels, data.plength, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
}

/* precompute some sigma-values */
tmpfl = (double *)malloc(data.num_classes*sizeof(double));
tmp = pow(2.0*M_PI, (data.bands/2.0) );
for( class=0; class< data.num_classes; class++ ) {
    tmpfl[class] = tmp * pow(data.class[class].sigma, data.bands);
}
sigma_x = pow(data.class[--class].sigma,2);

```

```

printf("%d working on %d rows\n",myid, data.rows);
/* classify entire image, row by row */
for (y = 0; y<data.rows; y++) {
  /* classify a row */
  start = (y*data.cols);
  for(x=start; x<(start+data.cols); x++) {
    maxval_x = -1;
    weight=0;

    for(class=0; class<data.num_classes; class++) {
      class_sum_x = 0;
      for(pattern=0; pattern<data.class[class].patterns; pattern++) {
        psum = 0;
        for(bands=0; bands<data.bands; bands++) {
          diff = data.pixels[pix+bands] - data.weights[weight++];
          psum += diff * diff;
        } /* end of bands loop */
        term_x = ((double)psum)/(2.0*sigma_x);
        exp_m_x = exp(-term_x);
        class_sum_x += exp_m_x;
      } /* end of pattern loop */

      if (data.class[class].patterns == 0) {
        curval_x = -1;
      }
      else {
        curval_x = class_sum_x / (tmpfl[class] * data.class[class].patterns);
      }

      /* assign color to pixel based on the largest class score. */
      if (maxval_x < curval_x) {
        maxval_x = curval_x;
        image[x] = data.class[class].num;
      }
    } /* end of Class loop */
    pix += data.bands; /* increment the raw data pixel "pointer" */
  } /* end of x (row) loop */
  /* a row is finished */
} /* end of y (entire image) block */

/* gather up the data */
if( myid == 0 ) {
  /* receive from all processors */
  for( p = 1; p<nprocs; p++ ) {
    MPI_Recv( &(amp; image[p*data.rows*data.cols] ), data.rows*data.cols, MPI_INT, p, 0,
MPI_COMM_WORLD, &status );
    printf("Received data from processor %d\n",p);
  }
}
else {
  /* send from each of the slaves */
  MPI_Send( image, data.rows*data.cols, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
}

```

15 Appendix E – DSP listing

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "/files1/match/mpi_match/mpi_match.h"

/* inline routine */
#define MCKECK(m) if (!m) { fprintf(stderr, "malloc failed\n"); exit(0); }

/* constants */
#define M_PI 3.14159265358979323846

main(int argc, char **argv)
{

```

```

/* types */
typedef unsigned char byte;
int *image;

/* *****
   Variables
   ***** */

/* structures */
struct Class {
    int num;
    int patterns;
    int sigma;
};

struct Data {
    int bands, patterns, num_classes;
    int rows, cols, imagefile_bands, bytes_per_entry;
    int max_weight;
    int *weights;
    int *pixels;
    int plength;
    struct Class *class;
};

typedef struct Class Class;
typedef struct Data Data;

Data data;

/* mpi vars */
int myid, i, proc, *buf, nprocs=4, the_id, chunksize, chunkmax;
int pixel_chunk, slave_plength, slave_rows;
MATCH_MPI_Status status;
/* pnn vars */
double tmp, sigma_x, maxval_x, curval_x, term_x, exp_m_x, class_sum_x;
double *tmpfl;
int class, weight, diff, pix=0, start;
int x, y, pattern, bands;
long psum;

/* Init MPI */
MATCH_MPI_Init(&argc,&argv);
MATCH_MPI_Comm_rank(MATCH_MPI_COMM_WORLD,&myid);

if( myid == dsp0 ) {
    /* get data from force */
    printf("Waiting for force...\n");

    MATCH_MPI_Recv( &data.bands, 1, MATCH_MPI_INT, force0, 0,
                    MATCH_MPI_COMM_WORLD, &status );
    MATCH_MPI_Recv( &data.patterns, 1, MATCH_MPI_INT, force0, 0,
                    MATCH_MPI_COMM_WORLD, &status );
    MATCH_MPI_Recv( &data.num_classes, 1, MATCH_MPI_INT, force0, 0,
                    MATCH_MPI_COMM_WORLD, &status );
    MATCH_MPI_Recv( &data.rows, 1, MATCH_MPI_INT, force0, 0,
                    MATCH_MPI_COMM_WORLD, &status );
    MATCH_MPI_Recv( &data.cols, 1, MATCH_MPI_INT, force0, 0,
                    MATCH_MPI_COMM_WORLD, &status );
    MATCH_MPI_Recv( &data.imagefile_bands, 1, MATCH_MPI_INT, force0, 0,
                    MATCH_MPI_COMM_WORLD, &status );
    MATCH_MPI_Recv( &data.bytes_per_entry, 1, MATCH_MPI_INT, force0, 0,
                    MATCH_MPI_COMM_WORLD, &status );
    MATCH_MPI_Recv( &data.max_weight, 1, MATCH_MPI_INT, force0, 0,
                    MATCH_MPI_COMM_WORLD, &status );
    MATCH_MPI_Recv( &data.plength, 1, MATCH_MPI_INT, force0, 0,
                    MATCH_MPI_COMM_WORLD, &status );

    /* prep buffers */
    data.weights = (int *)malloc(data.max_weight*sizeof(int));
    if( !data.weights ) { printf("weight malloc failed!\n"); exit(1); }
}

```

```

data.class = (Class *)malloc(data.num_classes*sizeof(Class));
if( !data.class ) { printf("class malloc failed!\n"); exit(1); }
data.pixels = (int *)malloc(data.plength*sizeof(int));
if( !data.pixels ) { printf("class malloc failed!\n"); exit(1); }
image = (int *)malloc(data.rows*data.cols*sizeof(int));
if( !image ) { printf("image malloc failed!\n"); exit(1); }

/* class array */
for( i=0; i<data.num_classes; i++ ) {
    MATCH_MPI_Recv(&( data.class[i].num ), 1, MATCH_MPI_INT, force0, 0,
        MATCH_MPI_COMM_WORLD, &status);
    MATCH_MPI_Recv(&( data.class[i].patterns ), 1, MATCH_MPI_INT, force0, 0,
        MATCH_MPI_COMM_WORLD, &status);
    MATCH_MPI_Recv(&( data.class[i].sigma ), 1, MATCH_MPI_INT, force0, 0,
        MATCH_MPI_COMM_WORLD, &status);
}

/* weights */
MATCH_MPI_Recv(data.weights, data.max_weight, MATCH_MPI_INT, force0, 0,
    MATCH_MPI_COMM_WORLD, &status);

/* distribute data to other dsps */
data.rows /= 4;
slave_plength = data.plength / nprocs;
slave_rows = data.rows / nprocs;

for( proc=1; proc<4; proc++ ) {
    MATCH_MPI_Send( &data.bands, 1, MATCH_MPI_INT, dsp0+proc, 0,
        MATCH_MPI_COMM_WORLD );
    MATCH_MPI_Send( &data.patterns, 1, MATCH_MPI_INT, dsp0+proc, 0,
        MATCH_MPI_COMM_WORLD );
    MATCH_MPI_Send( &data.num_classes, 1, MATCH_MPI_INT, dsp0+proc, 0,
        MATCH_MPI_COMM_WORLD );
    MATCH_MPI_Send( &slave_rows, 1, MATCH_MPI_INT, dsp0+proc, 0,
        MATCH_MPI_COMM_WORLD );
    MATCH_MPI_Send( &data.cols, 1, MATCH_MPI_INT, dsp0+proc, 0,
        MATCH_MPI_COMM_WORLD );
    MATCH_MPI_Send( &data.imagefile_bands, 1, MATCH_MPI_INT, dsp0+proc, 0,
        MATCH_MPI_COMM_WORLD );
    MATCH_MPI_Send( &data.bytes_per_entry, 1, MATCH_MPI_INT, dsp0+proc, 0,
        MATCH_MPI_COMM_WORLD );
    MATCH_MPI_Send( &data.max_weight, 1, MATCH_MPI_INT, dsp0+proc, 0,
        MATCH_MPI_COMM_WORLD );
    MATCH_MPI_Send( &slave_plength, 1, MATCH_MPI_INT, dsp0+proc, 0,
        MATCH_MPI_COMM_WORLD );

    for( i=0; i<data.num_classes; i++ ) {
        MATCH_MPI_Send(&( data.class[i].num ), 1, MATCH_MPI_INT, dsp0+proc, 0,
            MATCH_MPI_COMM_WORLD);
        MATCH_MPI_Send(&( data.class[i].patterns ), 1, MATCH_MPI_INT, dsp0+proc,
            0, MATCH_MPI_COMM_WORLD);
        MATCH_MPI_Send(&( data.class[i].sigma ), 1, MATCH_MPI_INT, dsp0+proc, 0,
            MATCH_MPI_COMM_WORLD);
    }

    /* send the weights in chunks */
    chunkmax = 16;
    chunksize = 512;
    for( i=0; i<chunkmax; i++ ) {
        MATCH_MPI_Send(&data.weights[i*chunksize], chunksize, MATCH_MPI_INT,
            dsp0+proc, 0, MATCH_MPI_COMM_WORLD);
    }
} /* proc loop */

/* big loop for pixels and processing */
for( pixel_chunk = 0; pixel_chunk < 4; pixel_chunk++ ) {
    /* pixels */
    MATCH_MPI_Recv(data.pixels, data.plength, MATCH_MPI_INT, force0, 0,
        MATCH_MPI_COMM_WORLD, &status );
    /* send the pixels */
    for( proc=1; proc<4; proc++ ) {

```

```

    chunkmax = 128;
    chunksize = 512;
    for( i=0; i<chunkmax; i++ ) {
        MATCH_MPI_Send(&data.pixels[(proc*slave_plength)+(i*chunksize)],
            chunksize, MATCH_MPI_INT, dsp0+proc, 0, MATCH_MPI_COMM_WORLD);
    }
}

/* PNN COMPUTATION */
tmpfl = (double *)malloc(data.num_classes*sizeof(double));

/* precompute some sigma-values */
tmp = pow(2.0*M_PI, (data.bands/2.0) );
for( class=0; class< data.num_classes; class++ ) {
    tmpfl[class] = tmp * pow(data.class[class].sigma, data.bands);
}
sigma_x = pow(data.class[--class].sigma,2);

pix = 0;
for( y = 0; y<slave_rows; y++ ) {
    /* classify a row */
    start = (y*data.cols);
    for(x=start; x<(start+data.cols); x++) {
        maxval_x = -1;
        weight=0;

        for(class=0; class<data.num_classes; class++) {
            class_sum_x = 0;
            for(pattern=0; pattern<data.class[class].patterns; pattern++) {
                psum = 0;
                for(bands=0; bands<data.bands; bands++) {
                    diff = data.pixels[pix+bands] - data.weights[weight++];
                    psum += diff * diff;
                } /* end of bands loop */
                term_x = ((double)psum)/(2.0*sigma_x);
                exp_m_x = exp(-term_x);
                class_sum_x += exp_m_x;
            } /* end of pattern loop */

            if (data.class[class].patterns == 0) {
                curval_x = -1;
            }
            else {
                curval_x = class_sum_x / (tmpfl[class] *
                    data.class[class].patterns);
            }

            /* assign color to pixel based on the largest class score. */
            if (maxval_x < curval_x) {
                maxval_x = curval_x;
                image[x] = data.class[class].num;
            }
        } /* end of Class loop */
        pix += data.bands; /* increment the raw data pixel "pointer" */
    } /* end of x (row) loop */
    /* a row is finished */
    printf("Row %d done\n",y);
} /* end of y (entire image) block */
printf("Done\n");

/* get data back from dsps */
chunkmax = 32;
chunksize = 512;
for( proc=1; proc<4; proc++ ) {
    for( i=0; i<chunkmax; i++ ) {
        MATCH_MPI_Recv( &(image[proc*slave_rows*data.cols+i*chunksize]),
            chunksize, MATCH_MPI_INT, dsp0+proc, 0, MATCH_MPI_COMM_WORLD, &status );
    }
}

/* send data to the force */

```

```

MATCH_MPI_Send( image, data.rows*data.cols*nprocs, MATCH_MPI_INT, force0,
                0, MATCH_MPI_COMM_WORLD);
} /* pixel chunk loop */
}
/* every other DSP */
else {
/* get data from dsp0 */
MATCH_MPI_Recv( &data.bands, 1, MATCH_MPI_INT, dsp0, 0,
                MATCH_MPI_COMM_WORLD, &status );
MATCH_MPI_Recv( &data.patterns, 1, MATCH_MPI_INT, dsp0, 0,
                MATCH_MPI_COMM_WORLD, &status );
MATCH_MPI_Recv( &data.num_classes, 1, MATCH_MPI_INT, dsp0, 0,
                MATCH_MPI_COMM_WORLD, &status );
MATCH_MPI_Recv( &data.rows, 1, MATCH_MPI_INT, dsp0, 0, MATCH_MPI_COMM_WORLD,
                &status );
MATCH_MPI_Recv( &data.cols, 1, MATCH_MPI_INT, dsp0, 0, MATCH_MPI_COMM_WORLD,
                &status );
MATCH_MPI_Recv( &data.imagefile_bands, 1, MATCH_MPI_INT, dsp0, 0,
                MATCH_MPI_COMM_WORLD, &status );
MATCH_MPI_Recv( &data.bytes_per_entry, 1, MATCH_MPI_INT, dsp0, 0,
                MATCH_MPI_COMM_WORLD, &status );
MATCH_MPI_Recv( &data.max_weight, 1, MATCH_MPI_INT, dsp0, 0,
                MATCH_MPI_COMM_WORLD, &status );
MATCH_MPI_Recv( &data.plength, 1, MATCH_MPI_INT, dsp0, 0,
                MATCH_MPI_COMM_WORLD, &status );

data.weights = (int *)malloc(data.max_weight*sizeof(int));
data.pixels = (int *)malloc(data.plength*sizeof(int));
data.class = (Class *)malloc(data.num_classes*sizeof(Class));
image = (int *)malloc(data.rows*data.cols*sizeof(int));

for( i=0; i<data.num_classes; i++ ) {
    MATCH_MPI_Recv(&( data.class[i].num ), 1, MATCH_MPI_INT, dsp0, 0,
                  MATCH_MPI_COMM_WORLD, &status);
    MATCH_MPI_Recv(&( data.class[i].patterns ), 1, MATCH_MPI_INT, dsp0, 0,
                  MATCH_MPI_COMM_WORLD, &status);
    MATCH_MPI_Recv(&( data.class[i].sigma ), 1, MATCH_MPI_INT, dsp0, 0,
                  MATCH_MPI_COMM_WORLD, &status);
}

/* receive weights */
chunkmax = 16;
chunksize = 512;
for( i=0; i<chunkmax; i++ ) {
    MATCH_MPI_Recv(&data.weights[i*chunksize], chunksize, MATCH_MPI_INT, dsp0,
                  0, MATCH_MPI_COMM_WORLD, &status);
}

for( pixel_chunk = 0; pixel_chunk < 4; pixel_chunk++ ) {
/* recv the pixels in the same way */
chunkmax = 128;
chunksize = 512;
for( i=0; i<chunkmax; i++ ) {
    MATCH_MPI_Recv(&data.pixels[i*chunksize], chunksize, MATCH_MPI_INT,
                  dsp0, 0, MATCH_MPI_COMM_WORLD, &status);
}

/* PNN COMPUTATION */
tmpfl = (double *)malloc(data.num_classes*sizeof(double));

/* precompute some sigma-values */
tmp = pow(2.0*M_PI, (data.bands/2.0) );
for( class=0; class< data.num_classes; class++ ) {
    tmpfl[class] = tmp * pow(data.class[class].sigma, data.bands);
}
sigma_x = pow(data.class[--class].sigma,2);

pix = 0;
for (y = 0; y<data.rows; y++) {
/* classify a row */
start = (y*data.cols);

```

```

for(x=start; x<(start+data.cols); x++) {
    maxval_x = -1;
    weight=0;

    for(class=0; class<data.num_classes; class++) {
        class_sum_x = 0;
        for(pattern=0; pattern<data.class[class].patterns; pattern++) {
            psum = 0;
            for(bands=0; bands<data.bands; bands++) {
                diff = data.pixels[pix+bands] - data.weights[weight++];
                psum += diff * diff;
            } /* end of bands loop */

            term_x = ((double)psum)/(2.0*sigma_x);
            exp_m_x = exp(-term_x);
            class_sum_x += exp_m_x;

        } /* end of pattern loop */

        if (data.class[class].patterns == 0) {
            curval_x = -1;
        }
        else {
            curval_x = class_sum_x / (tmpfl[class] *
                data.class[class].patterns);
        }

        /* assign color to pixel based on the largest class score. */
        if (maxval_x < curval_x) {
            maxval_x = curval_x;
            image[x] = data.class[class].num;
        }
    } /* end of Class loop */
    pix += data.bands; /* increment the raw data pixel "pointer" */
} /* end of x (row) loop */
/* a row is finished */
} /* end of y (entire image) block */

/* send data back to dsp0 */
chunkmax = 32;
chunksize = 512;
for( i=0; i<chunkmax; i++ ) {
    MATCH_MPI_Send(&image[i*chunksize], chunksize, MATCH_MPI_INT, dsp0, 0,
MATCH_MPI_COMM_WORLD);
}
}
}
MATCH_MPI_Finalize();
}

```

16 Appendix F – DSP host controller listing

```

/* distribute the data and read it back from dsp0 */
void distribute_data(void)
{
    /* parallel variables */
    int myid, i, chunkmax, chunksize, pixel_chunk;
    MATCH_MPI_Status status;

    /* setup parallel parameters */
    MATCH_MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    /* pixel chunk modification */
    data.plength /= 4;

    /* send data to dsp0 */
    printf("Sending scalar data to dsp0\n");
    MATCH_MPI_Send( &data.bands, 1, MATCH_MPI_INT, dsp0, 0, MATCH_MPI_COMM_WORLD );
    MATCH_MPI_Send( &data.patterns, 1, MATCH_MPI_INT, dsp0, 0, MATCH_MPI_COMM_WORLD );
    MATCH_MPI_Send( &data.num_classes, 1, MATCH_MPI_INT, dsp0, 0, MATCH_MPI_COMM_WORLD );
    MATCH_MPI_Send( &data.rows, 1, MATCH_MPI_INT, dsp0, 0, MATCH_MPI_COMM_WORLD );
}

```

```

MATCH_MPI_Send( &data.cols, 1, MATCH_MPI_INT, dsp0, 0, MATCH_MPI_COMM_WORLD );
MATCH_MPI_Send( &data.imagefile_bands, 1, MATCH_MPI_INT, dsp0, 0, MATCH_MPI_COMM_WORLD
);
MATCH_MPI_Send( &data.bytes_per_entry, 1, MATCH_MPI_INT, dsp0, 0, MATCH_MPI_COMM_WORLD
);
MATCH_MPI_Send( &data.max_weight, 1, MATCH_MPI_INT, dsp0, 0, MATCH_MPI_COMM_WORLD );
MATCH_MPI_Send( &data.length, 1, MATCH_MPI_INT, dsp0, 0, MATCH_MPI_COMM_WORLD );

printf("Sending class data\n");
for( i=0; i<data.num_classes; i++ ) {
    MATCH_MPI_Send(&( data.class[i].num ), 1, MATCH_MPI_INT, dsp0, 0,
        MATCH_MPI_COMM_WORLD);
    MATCH_MPI_Send(&( data.class[i].patterns ), 1, MATCH_MPI_INT, dsp0, 0,
        MATCH_MPI_COMM_WORLD);
    MATCH_MPI_Send(&( data.class[i].sigma ), 1, MATCH_MPI_INT, dsp0, 0,
        MATCH_MPI_COMM_WORLD);
}
/* weights */
printf("Sending weights\n");
MATCH_MPI_Send(data.weights, data.max_weight, MATCH_MPI_INT, dsp0, 0,
    MATCH_MPI_COMM_WORLD);

/* pixels */
for( pixel_chunk =0; pixel_chunk< 4; pixel_chunk++ ) {
    MATCH_MPI_Send( &data.pixels[data.length*pixel_chunk], data.length, MATCH_MPI_INT,
dsp0, 0, MATCH_MPI_COMM_WORLD);
    /* gather up the data */
    MATCH_MPI_Recv( &image[pixel_chunk*data.rows*data.cols/4], data.rows*data.cols/4,
        MATCH_MPI_INT, dsp0, 0, MATCH_MPI_COMM_WORLD, &status );
}
}

```