Library Functions in Reconfigurable Hardware for Matrix and Signal Processing Operations in MATLAB*

Suresh Periyacheri Anshuman Nayak Alex Jones
Nagaraj Shenoy Alok Choudhary Prithviraj Banerjee

Center for Parallel and Distributed Computing
Department of Electrical and Computer Engineering
Technological Institute, Northwestern University
2145 Sheridan Road, Evanston IL 60208-3118.
Phone: (847) 491-3641 Fax: (847) 491-4455.
E-mail: {suresh, nayak, akjones, nagaraj, choudhar, banerjee}@ece.nwu.edu

Abstract

Reconfigurable or adaptive computing is an emerging area of computer science, which promises super-computing performance from small, relatively low cost systems composed of re-programmable hardware such as field-programmable gate arrays (FPGA). But large scale reconfigurable computing is yet to become a reality, mainly due to the limitations of available reconfigurable hardware and the lack of the tools to aid in the fast development of applications to run on them. The MATCH project at Northwestern University is an attempt to make this application development process easier by providing tools that can quickly map programs written in a high-level language (MATLAB) to code that can be executed simultaneously on a collection of commercial-off-the-shelf (COTS) components that include DSPs and embedded processors, apart from FPGAs. The test bed for such a system has been built and one of its components is the WILDCHILD+ multi-FPGA system from Annapolis Microsystems Inc. To aid in the development of MATLAB-based applications, a library of matrix and signal processing functions is being developed for reconfigurable hardware that can be used as building blocks for these applications. In this paper we describe four such functions that were developed and their interfaces to the MATLAB compiler which forms the core of our efforts in this project.

1 Introduction

Reconfigurable or adaptive computing is an emerging area of computer science, which promises super-computing performance from small, relatively low cost systems composed of re-programmable hardware such as field-programmable gate arrays (FPGA). Reconfigurable computing

* This research was supported by DARPA under Contract F30602-98-2-0144.

+WILDCHILD is a registered trademark of Annapolis Microsystems Inc.

blurs the traditional boundaries between software and hardware by introducing the concept of hardware/software co-design where the underlying hardware to execute an application changes with the needs of the application. In effect, it provides the programmer with a custom processor for each of his/her applications.

But large scale reconfigurable computing is yet to become a reality, mainly due to the limitations of available reconfigurable hardware and the lack of the tools to aid in the fast development of applications to run on them. The MATCH project [1] at Northwestern University is an attempt to make this application development process easier by providing tools that can quickly map programs written in a high-level language (MATLAB) to code that can be executed simultaneously on a collection of commercial-off-the-shelf (COTS) components that include DSPs and embedded processors, apart from FPGAs.

The test bed for such a system has been built and one of its components is the WILDCHILD multi-FPGA system [2] from Annapolis Microsystems Inc. As part of the MATCH project, a compiler is being developed that would read in a high-level application written in MATLAB and generate code running on a host processor that would spawn of parallel tasks on the FPGAs, DSPs and the embedded processors. The current effort on the FPGA side is to develop library functions for matrix and signal processing operations in hardware on the FPGAs and to provide simple interfaces to them for the MATLAB compiler. This paper outlines these efforts and describes the functions and their implementation on the WILDCHILD system. More details of these libraries are discussed in [13].

1.1 The MATLAB Compiler

The initial version of a MATLAB compiler for the MATCH system relies on a library-based approach to execute sections of the program in FGPA-based hardware. The libraries have a C language interface that consists of a single function call with arguments of a common pre-defined data type used by the compiler to ship data and results between different functions. In the initial version of these library interfaces, the FPGA configuration loading and evaluating functions occur in sequence within a single interfacing function. This is not necessary, though, because the FPGA configuration required to implement a function is independent of the data and hence can be loaded much in advance of the actual computation. Potentially, it is useful in hiding the long loading times by executing the loads in parallel with other operations. This will be included in the next version of the compiler and library interfaces.

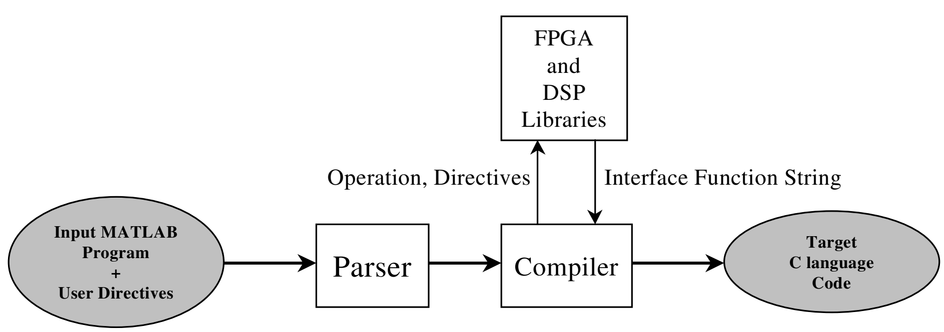


Figure 1 MATLAB Compiler

1.2 Compiler Interface to Libraries

The initial stages of the compiler identify tasks in the input MATLAB program (assisted by directives entered by the user) that are suitable for execution in FPGAs. This information is passed through several phases of the compiler while extracting dependencies among the various tasks to be executed in the FPGAs and other components of the system. The final phase of the compiler generates target C code embedded with the function calls to the FPGA and DSP libraries. Sample compiler generated C code showing this functional interface to the FPGA and DSP libraries is shown in Figure 2.

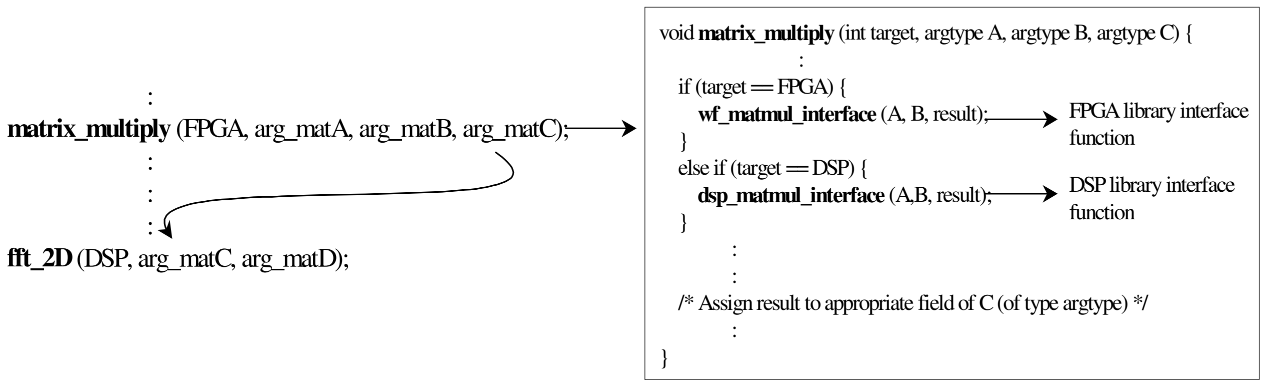


Figure 2 Compiler-Library Interface

2 Related Work

In the past few years there have been significant developments in the general area of adaptive and reconfigurable computing architectures, systems and software platforms. Some of these developments have been in the software engineering side in the form of tools which enable easy mapping of applications on reconfigurable systems and the development of such applications themselves. Some of these developments are explained below.

The Cameron project [3] at the Colorado State University is an attempt to develop an automatic tool for image processing applications in *Khoros,* an advanced and widely used software development environment for signal processing. It is specifically focused on Image Processing (IP) applications. The Cameron compiler targets an abstract architecture that consists of a set of parameterized hardware modules implemented in VHDL and mapped to a variety of FPGA-based reconfigurable platforms. Towards this end, they have implemented the IP components of a standard signal-processing library called VSIPL (Vector Signal Image Processing Library) in hardware using FPGAs.

Annapolis Micro Systems Inc., the manufacturer of the custom computer board used in the MATCH project, intends to develop a hybrid computing engine called WASPP (WILDFORCE(TM)-based Adaptive Digital Signal Processing Project). WASPP [4] combines the performance advantages of a Field Programmable Gate Array (FPGA)-based computer to the floating-point arithmetic and complex algorithm execution capabilities of a Digital Signal Processor (DSP). The goal of the research is to define a methodology for developing applications and to prove that a FPGA/DSP hybrid architecture is better suited for solving certain classes of problems than FPGAs or DSPs alone can solve. They plan to integrate the COTS FPGA and DSP development tools into a seamless design environment and a library of design components will be provided to make FPGA/DSP algorithm co-design easy to use and understand.

The CHAMPION project [5] at University of Tennessee, Knoxville, also focuses on providing tools to automate the process of mapping image processing applications in *Khoros* onto reconfigurable systems. The approach taken is to build a library of pre-compiled primitives that can be used as building blocks of image processing applications.

3 The WILDCHILD System: An Overview

The WILDCHILD custom computing engine is an integrated system consisting of the WILDCHILD multi-FPGA reconfigurable hardware unit and a host that controls it and provides a conventional software interface to the complete system. The WILDCHILD FPGA unit is a VME-compatible board that is installed in a standard chassis along with a VME-compatible host computer. The host, *Force 5V,* is a SPARC processor-based system running a version of the Solaris OS. The host and the WILDCHILD unit communicate through the VME bus, which is also shared by other components of the MATCH adaptive computing system.

3.1 The System Hardware

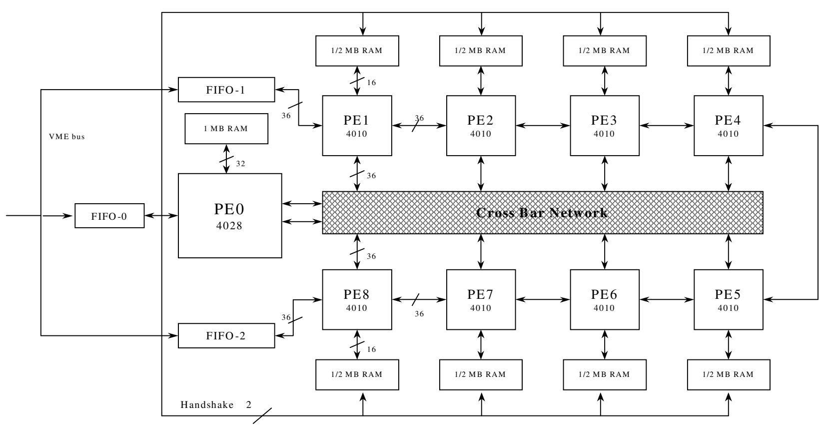


Figure 3 WILDCHILD System Architecture

The WILDCHILD board consists of 9 FPGAs of the popular Xilinx 4000 family. Eight of these are 4010 FPGAs, referred to as Processing Elements 1 through 8 (PE1-PE8), with 400 CLBs (about 10,000 gates) each and are identical. The ninth is a 4028 FPGA referred to as Processing Element 0 (PE0) with 1024 CLBs (about 30,000 gates). The nine FPGAs are arranged in a master-slave configuration as shown in Figure 3. Each FPGA on the board is connected to a pseudo dual-ported memory, which can be accessed by both the FPGA and the host. The memories connected to the 4010 FPGAs are 16 bits wide and contain 2^{18} addressable locations. The memory connected to PE0 is 32 bits wide and contains 2^{18} addressable locations. The architecture of the system enables high-throughput systolic computation using the 36-bit bus that connects each 4010 FPGA to its neighbors and also to the on-board FIFOs. There is also a crossbar network that can be used to realize any arbitrary interconnection of the FPGAs thus enabling irregular computations. As shown in Figure 3, two of the 3 on-board input/output FIFOs are connected to the systolic bus at each end of the PE1-PE8 chain. The FIFOs can be read or written directly by the host program. In addition to the crossbar, PE0 can communicate control and status information to each FPGA using a set of global handshake signals. The system clock rate is completely configurable to operate at any frequency though the board can only support operations up to a maximum frequency of 40 MHz.

3.2 The Host Interface Software

The host interface to the WILDCHILD on the *FORCE 5V* consists of a set of drivers that communicate with the WILDCHILD board using other drivers that talk to the VME back-plane. A set of custom headers and library functions for the WILDCHILD system are installed for application development in C using the *gcc* compiler.

The software interface to the WILDCHILD system consists of a set of application program interfaces (APIs) provided by the manufacturer. These are in the form of a library of function calls in C that are included in the program running on the host. These functions perform various tasks like loading a circuit on an FPGA to implement a function, configuring the crossbar, loading memories with data etc. Some of the important functions that are used to perform the basic tasks involved in running any application are listed below.

- *WFResetBoard* Resets the system board.
- *WfireOpen* Opens an interface to the WILDCHILD board.
- *WFireMaskPEInt* Masks the interrupt from a specified FPGA.
- *WfireProgramPE* Programs FPGA with the bit stream for a circuit.
- *WfireXBarConfig* Loads in a set of crossbar configurations.
- *WFireClearPEInts* Clears any pending interrupts on the host from the FPGAs.
- *WFireUnMaskPEInts* Unmasks FPGA interrupts.
- *WfireSetMemoryMode* Sets the mode of operation of the dual-port memory controller.
- *WfireClearMem* Clears the specified FPGA’s memory.
- *WfireWrite* Loads the specified FPGA’s local memory with data.
- *WfireRead* Reads the specified PE's local memory.
- *WFireClkSetFreq* Configures the system clock frequency.
- *WfireResetPE* Resets the specified FPGA.
- *WFireGetPEIntEvents* Reads the interrupt status register on the board.
- *WFireCloseBoard* Closes the WILDCHILD board.

4 Library Development on the WILDCHILD System

Four functions have been implemented in hardware using the WILDCHILD system which forms part of an ongoing library development effort for the MATCH project. Two of these – Filter and Fast Fourier Transform - are signal processing functions that occur frequently in kernels of signal processing applications. Two other functions implement the addition and multiplication operators for vector and matrix data types. In the following sections, we describe these functions and compare their execution times in FPGA hardware against execution times of the C language version of the functions compiled and run on the host processor. The performance of each library function is characterized for the configuration and runtimes and this information is used by the MATCH compiler for automated mapping of application tasks.

4.1 Filter Function

Using this function, FIR or IIR filtering of up to order 64 can be performed on vectors of maximum size 250,000 elements at hardware speeds. The data precision used is 8 bits fixed-point in fractional 2's complement format. Filtering is one of the most common operations performed in signal processing. MATLAB provides the ***filter*** function that can be used to perform different filtering operations on vectors of signal data. Most filters belong to one of two classes - FIR for Finite Impulse Response and IIR for Infinite Impulse Response filter. While the actual implementation of these two classes of filters is different, mathematically they are quite similar. The general filter equation is shown in Figure 4.

$$y(n) = \sum_{i=0}^{N} a_i x(n-i) + \sum_{i=1}^{N} b_i y(n-i)$$

Figure 4 General Filter Equation

An IIR filter can be used to perform FIR filtering by setting all of the constants '*b*' to 0. MATLAB and C codes for a 4-tap IIR filter operation is shown in Figure 5.

$$X = [x_0\ x_1 \cdots x_n]$$
$$a = [a_0\ a_1\ a_2\ a_3\ a_4]$$
$$b = [b_0\ b_1\ b_2\ b_3\ b_4]$$
$$Y = filter(a, b, X)$$

a and b are coefficient vectors
X is the input vector and Y is the output vector

(a)

```
x4=x3=x2=x1=0;
y4=y3=y2=y1=0;
for (i=0; i<n; i++) {
   y(i) = a(0)*x(i)+a(1)*x1+a(2)*x2+a(3)*x3+a(4)*x4
          - b(1)*y1 - b(2)*y2 - b(3)*y3 - b(4)*y4;
   x4=x3;  x3=x2;  x2=x1;  x1=x(i);
   y4=y3;  y3=y2;  y2=y1;  y1=y(i);
}
```

(b)

Figure 5 (a) MATLAB Code for Filter (b) Corresponding C Code

4.1.1 Design Alternatives for Implementing Filtering Operation

There are several schemes for implementing the general filter equation in Figure 4. The different schemes have different implications for the hardware complexity and performance of the filter. Also, the best hardware scheme may not be suitable for implementation on the WILDCHILD FPGA system due to resource constraints. Some of the schemes are described below.

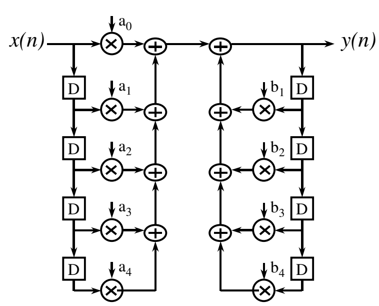


Figure 6 Direct Form I of 4th-order IIR Filter

Figure 6 shows the direct implementation of the basic delay-tap IIR filter equation. As can be seen from the figure, this would require as many delay blocks (registers) as there are delay elements in the equation.

The above form of the IIR filter, though efficient, is difficult to parallelize using multiple processors. Also, they are not scalable for higher-order filters that involve more delay taps. What is needed is a multi-processor design consisting of repeated units that is scalable. This makes the following form of the filter attractive for our implementation.

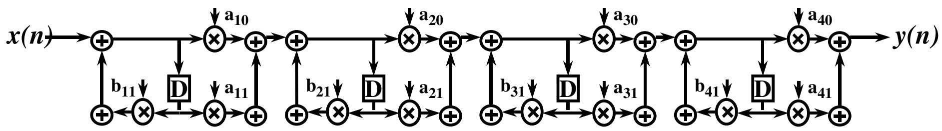


Figure 7 Cascade Form of 4th-order IIR Filter

The basic filter equation, through some transformations, can be factorized into a product of many smaller sized terms. This corresponds to a cascade of several lower-order filters to simulate the effect of the initial higher-order filter. For example, a 16-tap (16th order) filter can be constructed using a cascade of four 4-tap filters or eight 2-tap (second order) filters. Note that the filter coefficients for the new cascaded units are different from the original filter.

4.1.2 Implementation of the Filter Function on WILDCHILD

The cascaded form of the IIR filter lends well to implementation on the multi-FPGA architecture of the WILDCHILD system due to the presence of near-neighbor communication capability. Several FPGAs could be strung together in series to implement the required filter operation. The primary constraint in our implementation has been the resource requirements of the filter unit on each FPGA. Since the FPGAs on the WILDCHILD board are very modest, we were able to fit only a 1-tap filter (first order filter or 'integrator') on an FPGA.

Architecture of Filter on WILDCHILD

Figure 8 shows the architecture of the Filter function on the WILDCHILD multi-FPGA system.

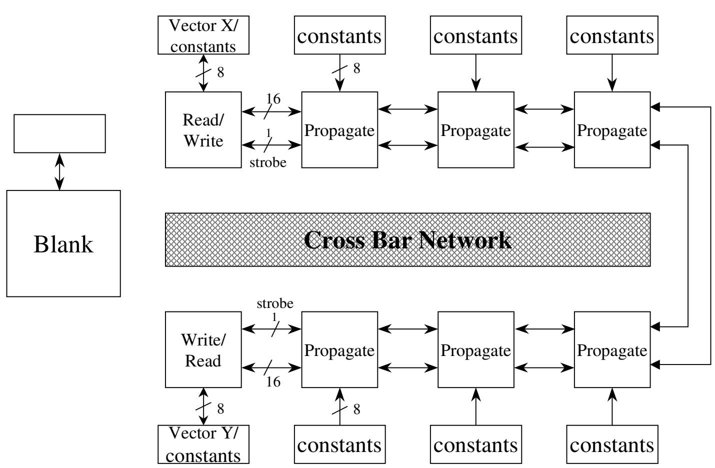


Figure 8 Architecture of Filter Function

Each 1-tap filter unit on the FPGAs is reversible. It can accept input from the left or right systolic bus and output the result on the right or left bus respectively. The input vector is initially stored in the local memory of PE1 and fed into the 1-tap filter on PE1. The partially filtered data then moves on to PE2, then to PE3 and so on in a pipelined manner until it reaches PE8 which writes its result to its local memory.

Thus an 8-tap filtering has been performed on the input vector. For higher-order filters, all PEs are reversed and the partially filtered data is read from PE8's local memory, passed through PE7, PE6 and so on until it reaches PE1 where it gets written to PE1's local memory. Thus a 16-tap filtering has been performed. This process of passing data back and forth can be repeated as many times as required to implement filters of any size. PE0 is not used in this design and is configured as a 'blank' FPGA

First Order Filter Module

The 1-tap filter is a reversible structure that can process data left-to-right or right-to-left. It consists of a delay unit, three 8-bit multipliers, a 16-bit adder, a 16-bit subtractor and registers to hold the filter coefficients. The filter coefficients for each iteration of the filter are loaded from memory into the registers between iterations. The filter also contains logic to detect and assert strobe signals and multiplexers to route source and result data. A schematic of the 1-tap filter unit is shown below in Figure 9.

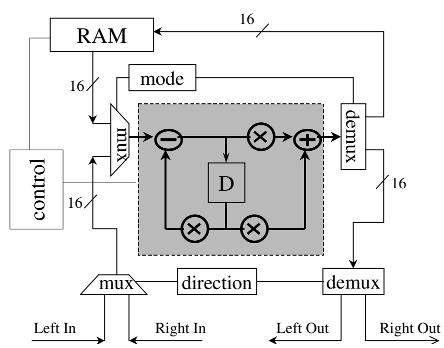


Figure 9 Schematic of First Order Filter

Each 1-tap filter could be in one of three different states - Read, Write or Propagate. In addition there is another state bit which determines the direction of flow of data i.e. left-to-right or right-to-left. The initial state of each processor is determined by a 'configuration' byte in the beginning of memory. For the usual case, the direction bit is initially set to left-to-right. PE1 is initially configured as Read, PE2 to PE7 are configured as Propagate and PE8 is configured as Write. After the first iteration, all PEs change their direction bit to right-to-left, PE8 goes into Read mode, PE1 goes to Write mode and all the other PEs remain in Propagate mode. This process of alternating between modes and directions is repeated until all iterations are complete. The result is read back from the local memory of the PE that was last in Write mode.

Resource Requirements and Clock Performance

The synthesis of the RTL description of the filter using the Synplify[α] synthesis tool resulted in a Xilinx netlist that required at least 329 CLBs (83%). The resultant circuit synthesized could

[α] Synplify is a registered trademark of Synplicity Inc.

potentially be clocked at a maximum frequency of 10 MHz. The netlist was a placed and routed using Xilinx's XACT[β] tools and the final version used 388 CLBs (97%) and could be clocked at a maximum frequency of 8 MHz.

4.1.3 Host Interface to Filter Function

The interface to the host is the function *wf_filter_interface()* which performs the mapping between the host data structures and the structures local to the FPGA system. This interface function contains within it two separate functions - the function loader and function evaluator. A part of the interface code is shown in Figure 10.

```
void wf_filter_interface(argtype order, argtype source, argtype
                         coeffs_A, argtype coeffs_B, void **dest)
{
      :
      :
      /* Set clock frequency */
      clk_freq = FLOAT(8.0);
      WFireClkSetFreq( hBoard, &clk_freq );

      /* Load configuration */
      wf_filter_load();

      /* Determine new filter coefficients from the original
        ones through factorization */
      factorize_coeffs(order.scalar,coeffs_A.data,coeffs_B.data,
                        &transformed_coeffs);
      in_vect.length = source.diminfo[0].bound;
      in_vect.byte = (unsigned char*)source.data;

      /* Call function */
      wf_filter(&in_vect,&transformed_coeffs,&out_vect);

      /* Assign destination */
      *dest = (void*)out_vect.byte;
}
```

Figure 10 Filter Host Interface

The loader configures all FPGAs (except PE0) on the WILDCHILD system with the 1-tap filter configuration and also loads in the default configuration of the crossbar. PE0 is loaded with the 'blank' configuration. The evaluator function is used to download data to the FPGA memories and to actually perform the filter operation. The first eight locations in memory are special purpose locations that contain some configuration information and the sets of filter coefficients. The evaluator function packs the configuration data, filter coefficients and the input data into a single buffer and performs one write operation to the target FPGA memory. Similarly, after receiving the function completion interrupt, it reads back the FPGA memory, unpacks result data from the data buffer and returns the result in the appropriate data structure. This data structure is then mapped into the compiler's data structure by the interface function. A part of the evaluator routine is shown below in Figure 11.

```
void wf_filter (WF_ARRAY *in_vect, WF_ARRAY *consts, WF_ARRAY *out_vect)
{
         :
  /* Set read mode, num of iterations and last data address */
         :
  /* Copy Constants Data */
         :
```

[β] XACT is a registered trademark of Xilinx Corporation

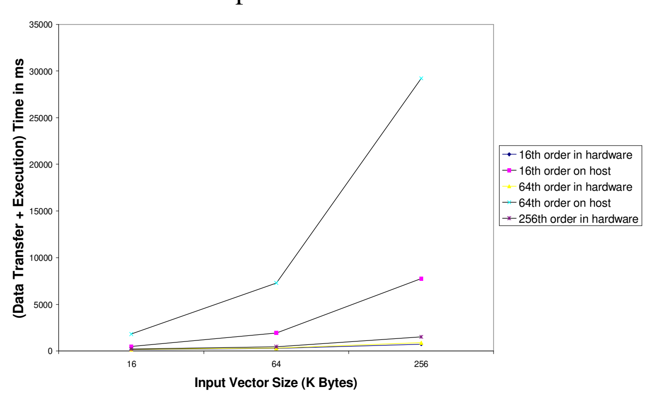
```
    /* Copy source data */
            :
    /* Write PE8 memory (reader)*/
    WFireWrite ( hBoard, 8, 0, 130+in_vect->length/4, buf);

    for (i=2; i<=7; i++) {
         :
         /* Write PEX (propagate) memory */
         WFireWrite ( hBoard, i, 0, 5+consts->length/32, buf);
    }
            :
    /* Write PE1 memory (writer)*/
    WFireWrite ( hBoard, 1, 0, 5+consts->length/32, buf);

    /* Give memory access to PEs */
    WFireSetMemoryMode( hBoard, WF_ALL_PE1_16, WF_HW_ARBITRATE );

    /* Start processors */
    WFireResetPE(hBoard, WF_ALL_PEs, WF_DISABLE);

    /* Determine PE to wait on */
            :
    /* Wait for PE interrupt */
    do {
         interrupt_vect = WFireGetPEIntEvents(hBoard);
    } while(!(interrupt_vect & pe_to_wait_on));

    /* Read back PE memory */
    WFireSetMemoryMode(hBoard, pe_to_wait_on, WF_BLOCKED);

    WFireRead (hBoard, pe_to_wait_on, 128, in_vect->length/2, buf);

    /* Assign result */
            :
}
```

Figure 11 Evaluator Function for Filter

4.1.4 Results

The filter function was tested on various data set sizes and for different filter taps (order). The filter was configured to run at 8 MHz, which is the maximum speed of the first-order filter module. The configuration and compute times are shown in Figure 12(a) and the results of the hardware version are compared against the C language version in the graph in Figure 12(b). It is clear that the hardware execution times are insignificant compared to the data transfer times. Also, the sum of the two times is much less compared to the execution times on the host processor.

Filter Taps	Vector size	Configuration Time (ms)	Data Transfer Time (ms)	Compute Time (ms)	Compute Time of C code on host (ms)
16	16K	2600	132+15=147	3	483
	64K	2600	188+58=246	13	1936
	256K	2600	440+230=670	52	7764
64	16K	2600	132+15=147	13	1813
	64K	2600	188+58=246	52	7290
	256K	2600	440+230=670	210	29219
256	16K	2600	132+15=147	52	7130
	64K	2600	188+58=246	210	28770
	256K	2600	440+230=670	840	115618



(a) (b)

Figure 12 FPGA/Host Execution Times

4.2 Matrix Multiplier Function

This function can be used to multiply two matrices of size up to 500X500 (for square matrices) elements in 16-bit fixed-point fractional 2's complement format. Matrix multiplication is a computationally intensive operation and involves O(n^3) complexity. In MATLAB, multiplication of matrices is simply specified by the use of the multiply operator '*'. The MATLAB and C codes required for multiplication of two matrices is shown below in Figure 13.

$$A = [a_{11}\ a_{12}\ a_{13}; a_{21}\ a_{22}\ a_{23}];$$
$$B = [b_{11}\ b_{12}; b_{21}\ b_{22}; b_{31}\ b_{32}];$$
$$C = A * B;$$

(a)

```
for (i=0; i<2; i++) {
  for (j=0; j<2; j++) {
    for (k=0; k<3; k++) {
      C[i][j] = C[i][j] + A[i][k]*B[k][j];
    }
  }
}
```

(b)

Figure 13 (a) MATLAB code for Matrix Multiplication (b) Corresponding C code

The matrix multiplication operation exhibits a high level of parallelism and this can be exploited using several methods of data partitioning. Some of these methods are attractive for multi-FPGA implementation and others are not. The factors that influence the decision to adopt a particular strategy in a shared or distributed memory parallel processing computer are different from that of a multi-FPGA system. Since the WILDCHILD system is a collection of FPGAs with separate local memories and there is no mechanism to share these memories across the FPGAs, the system is closer to a distributed memory parallel machine than to a shared memory machine.

4.2.1 Design Alternatives for Implementing Matrix Multiplication

There are several ways one could implement the multiplication function on a parallel platform. These implementations generally differ in the manner in which data and computation tasks are distributed among the multiple processors, which in turn depend on the architecture of the underlying system and the communication mechanisms. For the case of multi-FPGA systems, there are additional constraints like resource usage and communication bandwidths that affect the decision.

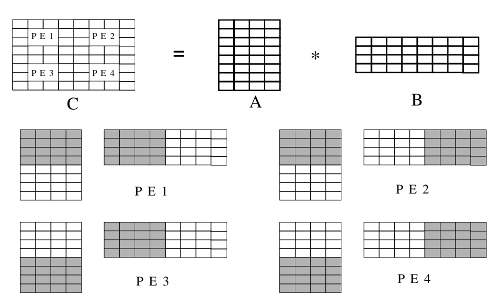


Figure 14 Block Partitioning of Result Matrix

Figure 14 shows a block partitioning of the output matrix C into N chunks, where N is the number of processors, and each processor performs the operations required to compute its portion of the output matrix. The set of rows of matrix A and columns of matrix B required for the purpose are stored locally in the processor's memory. This means row and column information is duplicated in the processors to minimize communication and increase performance. For example, for a matrix multiplication involving matrix A with 64 rows and matrix B with 32 columns, the output matrix C will have 64 rows and 32 columns. If there are 4 processors, each processor computes the following sections of the result matrix C: Processor 1 will compute rows 1:32 and columns 1:16, processor 2 will compute rows 1:32 and columns 17:32, processor 3 will compute rows 33:64 and columns 1:16 and processor 4 will compute rows 33:64 and columns 17:32.

In an alternative method of partitioning, the result matrix C is partitioned row-wise (or column-wise) and each processor is responsible for computing its set of rows (or columns). This requires the processor to store only the required rows (or columns) of the input matrix A in its local memory but it will need to store all the columns (or rows) of matrix B unless it can receive it from others through some sharing mechanism. One way to implement this sharing is to have a 'master' processor containing only data for matrix B to broadcast each of its columns on a broadcast network to all the 'slave' processors. The slaves then multiply the received column with their row data to generate the corresponding entries of the result matrix. Another way to implement sharing is to use a 'ring' topology where the master sends out a column to the first processor in the ring. This processor uses the column data to compute the corresponding result matrix elements and passes the column on to the next processor in the ring and so on.

For our implementation we chose the star topology for the following reasons: A broadcast in the WILDCHILD system is not an expensive process. All processors can be connected physically to each other using the network of wires that constitute the crossbar. A broadcast involves simply placing data on this mesh of wires and each processor can read it simultaneously during the same clock cycle. Since the interconnection network is only 36 bits wide and the processors can process only one multiply-accumulate operation per clock cycle, the columns of matrix B are broadcast one element every clock cycle. The broadcast based design does not involve any latency but the ring based design involves a latency to 'fill' the pipeline. Also, the logic that is required for the broadcast based design is simpler than that required for a ring-based design.

4.2.2 Implementation of the Matrix Multiplier Function on WILDCHILD

Architecture of Matrix Multiplier on WILDCHILD

Figure 15 shows the architecture of the matrix multiplier on the WILDCHILD FPGA system.

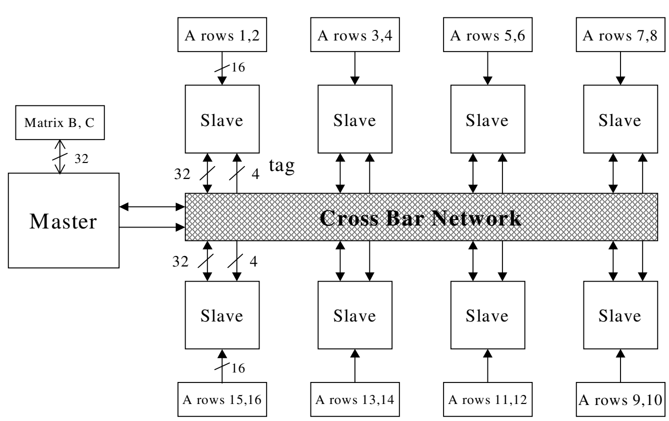


Figure 15 Architecture of Matrix Multiplier Function

Each PEX FPGA acts as a slave processor and the PE0 FPGA acts as the master processor. The input matrix A is row partitioned into as many partitions as there are slaves (max 8). Each slave processor is responsible for computing the corresponding row elements of the output matrix C. Thus, if matrix A has 64 rows and there are 8 slaves, rows 1 through 8 go to slave 1, rows 9 through 16 go to slave 2 and so on. The master processor (PE0) stores the entire input matrix B. The master broadcasts each column of matrix B, element by element during every clock cycle. The slave processor reads the column element from the crossbar, multiplies it with the corresponding row element and accumulates the result in a local register. Thus at the end of n clock cycles, where n is the number of elements in a column of B, each slave processor has accumulated the result for the respective element of the output matrix C. The accumulated result could be written to the local memories of the slave or they could be passed back to the master to be stored in a contiguous location. The memory utilization efficiency of either method would depend on the actual sizes of the input matrices. But since PE0 memory is twice as big as PEX memory, we predicted that the second method would lead to a better utilization of memory most of the time. The second method involves a more complex design compared to the first one. Since PE0 FPGA can only read from one PEX FPGA at a time, the slaves have to be scheduled to send their accumulated result to the master one after another on the crossbar. This involves each slave having to know its 'id' so that it knows when it is its turn to send its data on the bus to the master. This is achieved by having the configuration information in the beginning of the PEX memories to include the particular slave's id. So at the end of accumulation of a result matrix element, the slave waits on the crossbar tag bits until its id appears on it and then sends out the result.

Master Module

The master holds the entire matrix B in its memory. During initialization, the slaves read information about the source matrix sizes etc from the master, which broadcasts them on the crossbar. The tag consists of 4 bits with 3 bits used for the slave id and bit 4 used to signal a broadcast. The master broadcasts the column elements of matrix B one every clock cycle. When all the elements in a column have been sent, it sends out the id of the first slave (000) on the tag bits to signal to slave 0 to send out its accumulated result matrix element on the crossbar. It also sets the crossbar configuration so that PE0 (master) is connected to PE1 (slave). The data on the crossbar is written to memory in the next clock cycle. This process is repeated for all the slaves.

Resource Requirements and Clock Performance

The synthesis of the RTL description of the matrix mutliplier using the Synplify synthesis tool resulted in a Xilinx netlist that required at least 163 CLBs (16%). The resultant circuit synthesized could potentially be clocked at a maximum frequency of 28 MHz. The netlist was a placed and routed using Xilinx’s XACT tools and the final version used 192 CLBs (48%) and could be clocked at a maximum frequency of 25 MHz.

Slave Module

Each slave is responsible for computing a section of the output matrix. During initialization, the slave reads its id from the first location in its memory. It then reads information about the size of the source matrices etc from the master, which broadcasts them on the crossbar. The slave then waits for a broadcast signal from the master to begin reading column data from the crossbar. In every clock cycle, the slave reads a row element from its local memory, multiplies it with the column element on the crossbar and accumulates the result in a register. At the end of n clock cycles, where n is the number of elements in a column of matrix B, an element of the result matrix C has been computed. The slave then continuously reads the crossbar tag waiting for its id to appear on it. When it detects its id on the tag, it sends out the accumulated result on the crossbar. Figure 16 shows the slave circuit on PEX FPGA.

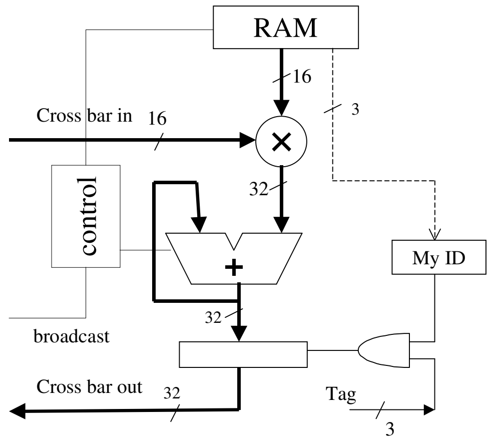


Figure 16 Schematic of Slave

Resource Requirements and Clock Performance

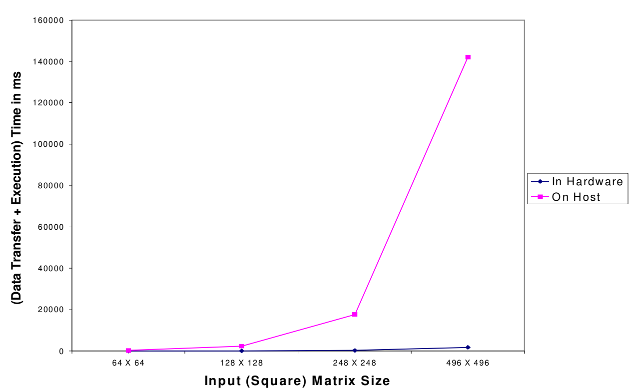
The synthesis of the RTL description of the matrix mutliplier using the Synplify synthesis tool resulted in a Xilinx netlist that required at least 192 CLBs (48%). The resultant circuit synthesized could potentially be clocked at a maximum frequency of 26 MHz. The netlist was a placed and routed using Xilinx's XACT tools and the final version used 215 CLBs (53%) and could be clocked at a maximum frequency of 20 MHz.

4.2.3 Results

The Matrix Multiplier function was tested on various input matrix sizes. The configuration and compute times are shown in Figure 17(a) and the results of the hardware version are compared against the C language version in the graph in Figure 17(b). The WILDCHILD system clock was set at 20 MHz, which is the maximum clock rate of the slave processor. It is evident from the data that the configuration and data transfer times dominate the evaluation time. The execution time of the C language version is about two orders of magnitude longer that the total of the download and execution times for the hardware version.

Matrix Size (Multiplication of square matrices)	Configuration Time (ms)	Data Transfer Time (Download+Readback) (ms)	Compute Time (ms)	Compute Time of C code on host (ms)
64 X 64	2620	31+7=38	1.95	300
128 X 128	2620	58+24=82	15	2380
248 X 248	2620	155+91=246	103	17702
496 X 496	2620	603+358=961	795	142034

(a)



(b)

Figure 17 FPGA/Host Execution Times

4.3 Scaled Matrix Adder Function

Using this function up to 7 matrices, each with a maximum of 250000 elements, can be added in parallel. The data precision used is 16-bit fixed-point in fractional 2's complement format. Several situations arise in the computation of matrix equations where there is a need to find the sum or difference of a set of matrices, often their scaled versions i.e. each element of every matrix is pre-multiplied by a simple constant before the addition or subtraction is performed. An example of such a situation is the solution of linear equations where the set of equations is represented as a single scaled matrix (vector) equation as shown in Figure 18. Once the equation is solved, the variables are substituted in the equations on the right-hand-side to compute the left-hand-side values.

$$
\begin{aligned}
z_1 &= x \cdot a_{11} + y \cdot a_{12} + z \cdot a_{13} \\
z_2 &= x \cdot a_{21} + y \cdot a_{22} + z \cdot a_{23} \\
&\vdots \\
z_n &= x \cdot a_{n1} + y \cdot a_{n2} + z \cdot a_{n3}
\end{aligned}
\qquad
\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix} = x \cdot \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{n1} \end{bmatrix} + y \cdot \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{n2} \end{bmatrix} + z \cdot \begin{bmatrix} a_{13} \\ a_{23} \\ \vdots \\ a_{n3} \end{bmatrix}
$$

Figure 18 Example of Scaled Vector Addition

Figure 19 shows the MATLAB and C versions of this operation.

$$
\begin{aligned}
A^1 &= [a^{11}\ a^{21}\ a^{31} \cdots a^{n1}]; \\
A^2 &= [a^{12}\ a^{22}\ a^{32} \cdots a^{n2}]; \\
A^3 &= [a^{13}\ a^{23}\ a^{33} \cdots a^{n3}]; \\
Z &= x * A^1 + y * A^2 + z * A^3;
\end{aligned}
$$

(a)

```
for (i=0; i<n; i++) {
    z[i] = x*a[i][1] + y*a[i][2] + z*a[i][3];
}
```

(b)

Figure 19 (a) MATLAB code for Matrix Multiplication (b) Corresponding C code

The scaling of the corresponding elements of each matrix can be done in parallel and also all the elements of the resultant matrix can be computed in parallel after they have been scaled. The manner in which the operation is parallelized will have implications for the resource requirements and performance of any design that implements it.

4.3.1 Design Alternatives for Implementing Matrix Addition

In block partitioning, the output matrix is partitioned into N chunks, where N is the number of processors, and each processor performs the addition on its portion of the output matrix. This requires that each input matrix is partitioned in the same manner and the chunks distributed to the processors' local memories. Also each processor needs a copy of the set of scaling constants. The data distribution for this scheme is shown in Figure 20.

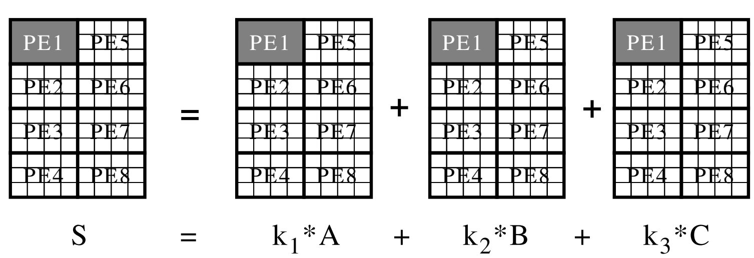


Figure 20 Block Partitioning

In the WILDCHILD system, each PEX FPGA could act as a processor, reading matrix elements from its local memory, multiplying each with the corresponding constant and accumulating the result in a local register. The result can then be written back to memory. This way each processor functions independently of the other processors. The advantage of this design is that the size of the operation is limited only by the size of each processor's memory and does not depend on the number of processors available.

The disadvantage of this design arises from the following: the memories in the WILDCHILD system have a non-zero bus turnaround time. This means that for each change between read and write modes, a clock cycle is lost in turning the bus around. This may not be significant for matrix additions involving several matrices, as the percentage of wasted clock cycles may not be large. But for smaller number of matrices this wastage could be significant. For example, for addition of five matrices, there are 2 wasted clock cycle for every 5 clock cycles which leads to a 30% loss in performance. One way to reduce this wastage is to buffer the results of several sets of additions and write the entire buffer in one shot.

An alternative data partitioning involves distribution of entire matrices on separate processor memories. Each processor reads a matrix element from its local memory, scales it with the locally stored constant and adds it to the sum it receives from the previous processor. It then passes this sum on to the next processor. The last processor reads the incoming sum and writes it to its local memory. The processors can then be classified as Reader, Adder and Writer.

This scheme was implemented on the WILDCHILD system. The advantage of this design is that since each FPGA will either always read or write memory, there is no penalty for bus turn-around as in the previous design. The disadvantage of this design is that one of the FPGAs does the trivial operation of only writing incoming data to memory and does not do any processing. The more serious limitation is that the size of the matrix equation is limited to a maximum of seven terms.

4.3.2 Implementation of the Matrix Adder Function on WILDCHILD

Architecture of Matrix Adder on WILDCHILD

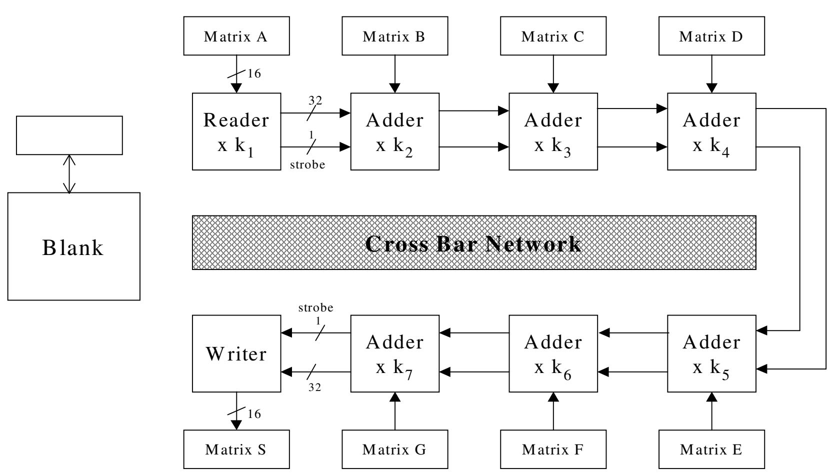


Figure 21 Architecture of Matrix Adder Function

The architecture for the scaled matrix adder function is shown in Figure 21. The matrix Adder is comprised of three separate designs - the Reader, Adder and Writer. Each of these units was targeted for the PEX FPGAs and the PE0 FPGA was left blank.

The Reader reads in a 16-bit matrix element every clock cycle from memory, multiplies it with a constant and sends out the 32-bit product on the right systolic bus. The constant is read from memory during the initialization phase and stored in a local register. The multiplier is a single clock cycle multiplier generated by the synthesis tool. The Reader asserts the strobe line on the right bus to signal to the next processor that valid data is available on the bus.

The Adder reads in a 16-bit matrix element from the memory every clock cycle and multiplies it with a constant. It then adds this 32-bit product to the 32-bit number read from the left systolic bus during the same clock cycle and sends the 32-bit sum to the right systolic bus. Though the start of computation is dependent on the input strobe from the previous processor, the end of computation is determined by a count that is loaded from memory during the initialization phase.

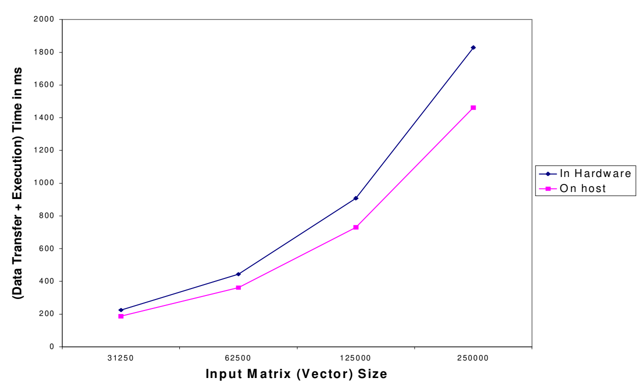
The Writer reads in a 32-bit number every clock cycle from the left systolic bus and sends out the most significant 16 bits on the local memory bus. The Writer waits on the input strobe signal from the preceding processor to start writing data to memory and stops as soon as the input strobe is turned off. It also asserts the interrupt signal briefly to signal the end of computation.

4.3.3 Results

The Matrix Adder function was tested on various data set sizes and the configuration and compute times are shown in Figure 22(a) and the results of the hardware version are compared against the C language version in the graph in Figure 22(b). Since the Adder module can run only at 9 MHz, it determined the system clock. It is evident from the data that the configuration and data transfer times dominate the evaluation time by several orders of magnitude. In this case, the large data transfer time defeats the advantages of high hardware execution speeds and there is no speed-up compared to the C version running on the host. The scaled add operation, being a very fine-grain parallel operation, runs very efficiently on the host due to efficient pipelining in the processor. Also, the data accesses are very regular leading to very good cache performance.

Data Size (Total elements)	Configuration Time (ms)	Data Transfer Time (Download+Readback) (ms)	Compute Time (ms)	Compute Time of C code on host (ms)
31250	3300	195+28=223	0.8	186
62500	3300	386+55=441	1.8	361
125000	3300	793+110=903	3.7	730
250000	3300	1600+220=1820	7.5	1462

(a)



(b)

Figure 22 FPGA/Host Execution Times

4.4 Fast Fourier Transform Function

This function can compute the Fast Fourier Transform of up to 1024 points where each point is a complex number with real and imaginary parts in 8-bit fixed-point fractional 2's complement format. In MATLAB, the Fast Fourier Transform on a vector of input data is computed by using the *fft* function as shown below.

$$
\begin{aligned}
&Y = fft(X) \\
&where\ X\ is\ vector. \\
&Y = fft(X, n) \\
&returns\ the\ n - POINT\ \ fft\ of\ \ vector\ X
\end{aligned}
$$

The FFT operation exhibits a high level of parallelism during the initial stages of computation but communication between processors becomes high during the final stages. Since communication between processors in the multi-FPGA WILDCHILD system is easily achieved using a mesh of interconnecting wires, there is no extra cost in communication though it does lead to more complex logic on the processors.

The FFT computation, the so-called butterfly graph, can be represented graphical as shown in Figure 23. (a) shows the original graph and (b) shows the graph re-ordered to exhibit repetitive functional blocks and the corresponding data flow.

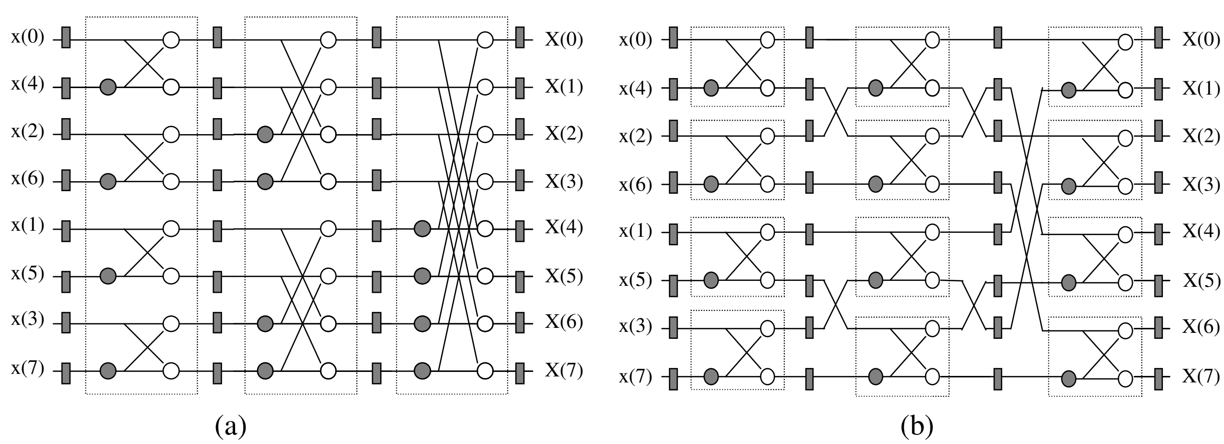


Figure 23 Butterfly Graph for Computing FFT (a) and a Re-ordered Graph (b)

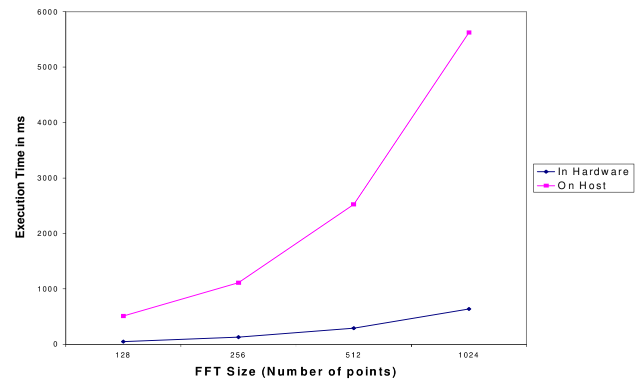
An FFT of size *N* has *N * log(N)* computation 'cells'. Each cell is comprised of a complex multiplier (one input is a constant), one complex adder and one complex subtractor. Potentially, each of the constants in the complex multipliers is a distinct non-trivial quantity and the constants are different for different sized FFTs. Each complex multiplication involves 4 real multiplications and 2 real additions. Each complex addition involves 2 real additions.

4.4.1 Results

The 8-bit FFT function was tested on various input vector sizes and the configuration and compute times are shown in Figure 24(a) and the results of the hardware version are compared against the C language version in the graph in Figure 24(b). The clock speed for the design was 8 MHz. The data transfer times are in milliseconds while the computation times are in microseconds. Thus the data transfer times are a full 3 orders of magnitude larger than the execution times.

FFT size	Configuration Time (ms)	Data Transfer Time (ms)	Compute Time (us)	Compute Time of C code on host (us)
128	3656	105+46=151	50	510
256	3656	106+47=153	130	1111
512	3656	107+48=155	290	2528
1024	3656	109+48=157	640	5624

(a)



(b)

Figure 24 FPGA/Host Execution Times

5 Conclusion

In this paper, the design and implementation of a set of library functions for matrix and signal processing operations and their interfaces to a MATLAB compiler was presented. The MATCH compiler compiles a source program in MATLAB into parallel code that can be executed simultaneously on a collection of COTS hardware like FPGAs, DSPs, and embedded processors that form an experimental heterogeneous test bed. The FPGA component of the system comprises of the WILDCHILD system, a multi-FPGA reconfigurable computer with 9 Xilinx FPGAs on a single VME-compatible board. We described in detail our implementations of a library function for filtering operations and briefly mentioned the architecture and implementation of three other functions for matrix and signal processing operations. We presented the execution times for each function, which reflected the performance gains of the function in hardware compared to its traditional implementations.

6 References

[1] The MATCH Project
http://www.ece.nwu.edu/cpdc/Match/Match.html

[2] The WILDCHILD Custom Computing Engine
http://www.annapmicro.com/wfhtm.html

[3] "The Cameron Project: High-Level Programming of Image Processing Applications on Reconfigurable Computing Machines" by W. Najjar et al (PACT Workshop on Reconfigurable Computing, October 1998) http://www.cs.colostate.edu/cameron/

[4] WASPP: WILDFORCE(tm)-based Adaptive Signal Processing Project
http://www.annapmicro.com/WASSP.html

[5] CHAMPION: A Software Design Environment for Adaptive Computing Systems
http://microsys6.engr.utk.edu/~bouldin/darpa/

[6] "Using Xilinx FPGAs to Design Custom Digital Signal Processing Devices" by Gregory R. Goslin, Xilinx Inc.

[7] "Using Programmable Logic to Accelerate DSP Functions" by Steven K.Knapp, Xilinx Inc.

[8] “Essential VHDL: RTL Synthesis Done Right” by Sundar Rajan
http://www.clbooks.com/specials/vhdl/

[9] “The Designer’s Guide to VHDL” by Peter J.Ashenden, Morgan Kaufmann

[10] “Digital Signal Processing in VLSI” by Richard J.Higgins, Prentice Hall

[11] “Digital Signal Processing with the TMS320C25” by Chassaing, Wiley-Interscience

[12] “HDL Chip Design” by Smith, Doone Publications

[13] "Library Functions in Reconfigurable Hardware for Matrix and Signal Processing Operations in MATLAB" by Suresh Periyacheri, MS Thesis. CPDC Technical Report CPDC-TR-9905-008.
http://www.ece.nwu.edu/cpdc/HTML/techreports.html