THE HARDWARE SUBROUTINE APPROACH TO DEVELOPING CUSTOM CO-PROCESSORS

A THESIS SUBMITTED TO THE GRADUATE DIVISION OF THE
UNIVERSITY OF HAWAI`I IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

INFORMATION AND COMPUTER SCIENCES

MAY 2001

By

Mark F. Waterson

We certify that we have read this thesis and that, in our opinion, it is satisfactory in scope and quality as a thesis for the degree of Master of Science in Information and Computer Sciences.

THESIS COMMITTEE

____

Chairperson

____

____

Acknowledgements

Support for this work by the administration and faculty of the University of Hawaii Institute for Astronomy, and the assistance of the university support programs of Research Systems, Inc, Xilinx, Inc, Aldec, Inc, and Synplicity, Inc. are gratefully acknowledged.

Abstract

The Hardware Subroutine Approach is a process for developing a reconfigurable, custom co-processor as a direct replacement for a subroutine in a larger program. The approach provides a framework that helps the developer analyze the benefits of using hardware acceleration, and a design procedure to guide the implementation process.

To illustrate the design process a Hardware Subroutine (HWS) implementation of a derivative estimation function is described. In this context, it is shown that key performance parameters of the HWS can be estimated well before the design is finished. Performance of this example is compared to the software-only implementation and to estimates developed during the design process. Due to limitations in a vendor-supplied driver, the performance of this implementation is found to be a factor of 60 slower than the target subroutine's performance.

A convolution function is analyzed and performance estimates presented as a second example. Analysis of 5 potential designs indicate that two of them would be expected to produce better performance than the original software routine. Convolution of a 256x256 element data array and 16x16 kernel is predicted to take 128 milliseconds using the HWS system, while the same computation takes 201 milliseconds in software.

HWS coprocessors can provide performance improvements for problems which are computationally intensive, but the technique is limited by data transfer overhead. The approach also shows potential as a research platform for investigation of hardware/software integration issues and novel processing techniques.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ALU	Arithmetic Logic Unit
API	Application Program Interface
DEC	Digital Equipment Corp.
DT	Decision Table
DMA	Direct Memory Access
FM	Functional Memory
FPGA	Field-Programmable Gate Array
HWS	Hardware Subroutine
HWSA	Hardware Subroutine Approach
IDL	Interactive Data Language
PE	Processing Element
RAM	Random-access Memory
RPC	Remote Procedure Call
SIMD	Single-Instruction Multiple-Data
VHDL	VHSIC (Very-high-speed Integrated Circuit) Hardware Description Language
mS	Millisecond
uS	Microsecond
nS	Nanosecond

1.0 Introduction

Analyses of the performance of large programs used for reduction and analysis of scientific data frequently show that a few relatively small program steps consume most of the processing time required by the application. While this is often due to limitations on data transfer bandwidth in the processing hardware, the sequential architecture of today's general-purpose computers is not always optimal for implementing the algorithms required. The processing time required by these applications not only impedes obtaining research results, but also slows the development and testing of new analysis methods. Utilization of custom co-processing hardware designed to more efficiently implement resource-intensive areas of the program can substantially improve performance for some problems.

Access to hardware co-processing allows the researcher to consider applying alternative computational techniques that are not feasible within the constraints of the sequential processor environment. Examples such as implementing searching using content-addressable memory [1] or the substitution of a Discrete Polynomial Transform for the normal Fast Fourier Transform (FFT) algorithm illustrate approaches which are not reasonable on a general-purpose computer system because they are not computationally efficient on a purely sequential architecture.

The techniques needed to implement a candidate problem as a hardware logic array have not, however, been easily accessible to the typical developer of scientific analysis programs. There are two components to this problem. First, while many approaches to implementing processing problems in discrete hardware have been discovered, implementing these circuits generally requires a significant commitment of time and resources to developing and testing the detailed designs. In addition, integrating such a processor system with a general-purpose computer requires implementing the equivalent of a new peripheral interface for the system, a challenging process. Before committing to an effort of this size, a researcher must be able to determine whether the investment in learning the technology and developing an implementation will result in sufficient performance improvement.

1.1 The Hardware Subroutine Approach

The Hardware Subroutine Approach (HWSA) is a framework and a philosophy for constructing custom hardware coprocessors and integrating them with conventional computers and software programs. The approach provides a common architecture and a design procedure that will help a developer or scientist wishing to investigate the application of reconfigurable computing techniques. The procedure allows the user to develop a hardware-based replacement for a subroutine that can be directly substituted into an existing program to provide enhanced performance or alternative processing methods.

The HWSA leverages advances in available programmable logic devices, development tools, and software engineering techniques to avoid the need for designers to "program down to the transistors" to produce a hardware processor design. A further key component of the HWSA is the use of performance estimation techniques to guide the designer's implementation decisions from an early stage of a project so that resources are not wasted on approaches that are unlikely to be successful.

To permit consistent estimates of performance across projects, implementations must be based on comparable types of design and should be developed with a consistent process as well. To support such consistency, the Hardware Subroutine Approach provides a flexible design architecture based on a hierarchy of interfaces and processing elements. A typical HWS architecture, shown in Figure 1, consists of a hardware Processing Element (PE) connected to a host computer by a standard bus and to a parent program by interface software arranged into hierarchical layers.

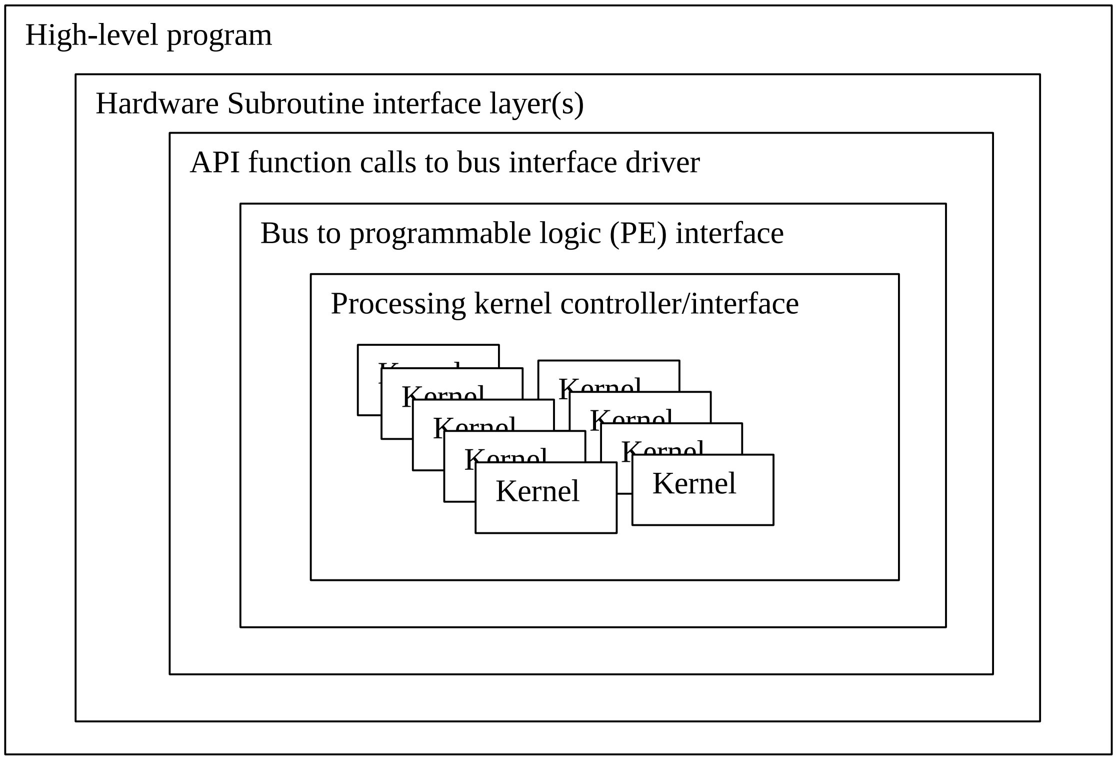
High-level program
Hardware Subroutine interface layer(s)
API function calls to bus interface driver
Bus to programmable logic (PE) interface
Processing kernel controller/interface
Kernel
Kernel

Figure 1. A Generic Hardware Subroutine Architecture

1.2 Thesis objectives

The goal of this work is to develop a coherent process for designing and implementing integrated software/hardware computing systems containing novel and customized coprocessing elements. In this paper, I show that complete subroutines from a high-level data analysis program can be replaced by hardware implementations of the algorithms. Comparison of the measured performance of the final integrated hardware-software system with estimates derived from simulation of parts of the system and other proxy measures show the value of early performance estimates. I also show how these early estimates can be used to guide design efforts. Thus, the claims of this thesis are:

- *The Hardware Subroutine Approach represents an improved process for designing and evaluating integrated systems of custom hardware and software;*
- *The Hardware Subroutine is a cost- effective way to implement a custom hardware coprocessor;*
- *The Hardware Subroutine Approach provides the designer with early predictions of a design's performance that can indicate the probable effectiveness of the resulting system.*

1.3 Contributions

This work provides a bridge between the purely software description of computation algorithms and the implementation of those algorithms in application-specific hardware processing units. In identifying a set of commercially available tools and a design framework through which a Hardware Subroutine processing element can be developed, the HWSA helps the software developer community to utilize current hardware design technology.

The HWSA can also be used to aid research into alternative computational techniques by providing an easier way to leverage the capabilities of a conventional computer as a development support environment. By performing such essential but mundane tasks as data storage, input/output and user interface display on a host machine while isolating the novel computation to a subroutine context the HWSA removes the need to create a complete computer and operating system simply to test a computational architecture. This approach allows the researcher to focus their attention on the most productive areas and encourages investigation of synergistic combinations of novel hardware architectures, software structures, and design techniques.

1.4 Document structure

The remainder of this paper will describe the aspects of the HWS Approach in detail. Chapter 2 reviews related work that forms the foundation for the HWS approach. Chapter 3 expands on the overall design philosophy behind the HWSA and the architectural hierarchy of a Hardware Subroutine implementation. The characteristics of computational problems suitable for successful HWS implementation and the requirements imposed on the host processor and software environment will also be discussed. Chapter 4 describes the design process used to progress from a purely software source algorithm to the final hardware image and interface code. The specific tools used at each step will be introduced in the context of implementing a derivative estimation algorithm. Performance of the resulting system is compared to that of the original software routine and the preliminary estimates. Chapter 5 presents a design study for a more complex convolution function and compares the predicted performance of design alternatives to the software implementation of the function. Conclusions regarding the utility and limitations of the HWS approach are presented in Chapter 6, along with some characteristics of the commercial hardware and software that can have a significant impact on the results. Several avenues for further research are suggested in Chapter 7.

2.0 Related Work

The development of high-density field reprogrammable logic gate arrays (FPGAs) has encouraged the design of custom processors that do not use fixed hardware implementations. While this allows more flexibility in developing and using logic-based external computing resources, the techniques for designing and implementing the needed hardware circuits have remained significantly different from those of software development.

Research into use of programmable logic has resulted in several approaches to connecting high-level algorithms to the detailed register-level logic needed for implementation in gate arrays. Many of these efforts have focused on converting complete programs into hardware, as researchers attempted to develop alternatives to von Neumann processing architectures that would be able to perform general-purpose computations in a more efficient manner by executing simple computations in parallel. For some applications, particularly those involving processing streams of data values, interfacing, and real-time control, this approach has proven successful and application-specific processors implemented in FPGAs are widely used in these areas.

2.1 Instrument control using FPGAs

FPGA-based hardware acceleration has been applied to problems in the field of astronomy in several ways. The most successful application of the technology has been to interface control and data-formatting problems that are difficult to accomplish on a conventional workstation due to real-time event constraints.

Bennett [2] describes a high-speed data acquisition system designed for use with photon-counting detectors to construct images from extremely faint objects. The hardware co-processing board contains a large Xilinx FPGA and 4MB of static RAM and is used to control timing and capture of photon events from two types of detectors. The system can buffer and combine data to reduce the event rate, so that a normal workstation can be used for supervisory control. The benefit of the approach is integrating data from high-speed detectors and GPS timing sources in real time without relying on a computer to capture each event, permitting data capture rates of up to 8 Mbytes/sec with 0.1-microsecond resolution. The system's logic is defined in Handel-C, a compile-to-hardware system available from Embedded Systems, Ltd [3].

An application of an FPGA-based coprocessor to controlling an instrument used in Solar astronomy is described by Shand [4]. The approach taken to integrate the interfaces between external camera hardware and the controlling computer, based on the Digital Equipment Corp. (DEC) PAMette reconfigurable processing board [5], focused on external logic implementation of only those functions that could not be handled by the conventional computing resources. These functions were primarily data synchronization, low-level control signals, data formatting, and real-time sequencing tasks that were not compatible with the multi-tasking operating system used on the workstation. All computation was done in the 64-bit ALU of the workstation's DEC Alpha microprocessor. However, novel bit-packing schemes were used to achieve performance goals. The Alpha chip was thus used as a SIMD parallel processor, requiring some low-level assembly-language programming of the microprocessor. Comparison of this system to an alternate approach that used a dedicated high-performance image-processing subsystem shows that the flexibility of reconfigurable hardware resulted in lower system costs and faster development.

The philosophy guiding Shand's projects, expressed as "if it can be done on a microprocessor, it will be done on a microprocessor," reflects the reluctance of scientists to utilize a technology if it offers only marginal benefits. Continuing advances in workstation speed, and the greater degree of skill required to complete a hardware-based processing scheme, combine to make it more difficult to show a compelling benefit in using external hardware for computation.

2.2 FPGAs as data processing elements

The use of FPGAs to provide expression-processing resources to a conventional microprocessor system has also been explored. Halverson and Lew describe an architecture for attaching programmable logic to a microprocessor as a memory-mapped peripheral termed Functional Memory (FM) [6, 7]. Consisting of expression-level processors attached to a microprocessor as addressable memory locations, a computation is triggered by writing a data value to one of these locations. An FM-based processing element appears to the host CPU as a normal memory location. However, the values read back from the location contain processed results of the data written in earlier cycles. While constrained by the limited size of FPGAs available at the time, they show that a

3x3 convolution engine implemented as an FM processor could outperform a specialized signal processing chip by reducing the number of processing steps needed.

Halverson and Lew also describe a complete processor architecture implemented as Functional Memory. A mechanism for controlling program execution, based on a decision table representation developed in an earlier paper [8], is shown to be effective in this application as well. In the complete design, the host microprocessor is only required to load and fetch data values from the FPGA system.

While the application of programmable logic as a computation engine has been shown to be advantageous in areas where the characteristics of the problem permit parallel data operations, the cost and risk involved in implementing the coprocessor logic is an obstacle. In one case study, Shand explores a searching problem and shows that even problems that contain a high degree of parallelism cannot always be solved faster in reconfigurable logic-based hardware processors [9]. He describes several techniques for speeding up processing of an exhaustive-search problem in hardware, but goes on to show that they are equally applicable to an approach that uses the 64-bit ALU in the DEC Alpha processor in novel ways. Despite employing substantial amounts of parallel computation, the hardware approach results in performance that is comparable to that of the conventional CPU-based system. The benefit of parallel computation is offset by overhead components that are not required by computations performed entirely within the processor’s local cache environment. Given the greater difficulty of developing hardware-based processing systems, the author feels the overall cost of the approach is not justifiable in this case. However, it is not as clear that these specialized ALU techniques would be appropriate in other areas.

2.3 The "compile-to-hardware" approach

A significant challenge to use of FPGAs as computing resources is that the techniques for manually designing efficient hardware logic are quite specialized. Scientists and programmers who might find hardware coprocessing useful are generally not familiar with this field and thus face a significant barrier to using this technology.

This need has motivated research into methods for transforming algorithms written in common programming languages directly into the low-level descriptions required for programming the FPGA. The high demand for improved hardware design tools has pushed this research out into the commercial sector and several companies now offer such "compile-to-hardware" systems. These systems use specialized libraries to permit compiling an algorithm written in C or Java directly into FPGA–ready logic descriptions. The drawback to this approach for the experimenter is that the structure of the logic executing the algorithm is fixed by the compiler and libraries, restricting the kinds of processing architectures that can be expressed. For example, the A/RT Builder ("Algorithm to Register Transfer") libraries and compiler from Frontier Technologies [10] are extensions to the C programming language that permit the user to automatically generate a VLIW (Very Large Instruction Word) processor and custom sequencing microcode from a C-language program. While the resulting FPGA processor is specialized to execute that particular algorithm, the user is not able to influence the underlying architecture of the implementation. This limits the benefit of these tools to problems which are good fits to the generated architecture, and does not assist the investigation of alternative designs.

3.0 Hardware Subroutines – A Better Way

The work described here is an approach to developing custom hardware coprocessors and integrating them with high-level programs that avoids the limitations of the "compile-to-hardware" techniques and builds on the Functional Memory architecture. In the basic design, an FPGA-based reconfigurable coprocessor is attached to a host computer's system bus and is treated as a peer peripheral rather than a memory unit to permit transfers of data in multiple-word bursts to and from the host computer. This model supports serial stream-based processing techniques as well as work with entire arrays, which can be stored in memory local to the co-processor since the necessary addressing logic can be implemented in the FPGA along with the computational processing circuitry. The FPGA processor logic is expressed in the industry-standard hardware description language VHDL [11], allowing use of simulation and implementation tools supplied by the device vendors, and permitting the design ideas to be expressed in a high-level, behavioral manner.

3.1 The Hardware Subroutine

A Hardware Subroutine (HWS) is a system consisting of a custom hardware coprocessing device and several layers of interface software, designed to directly replace a candidate subroutine in a larger computer program. The objective of the HWS approach is to provide the user the ability to "plug-in" a performance improvement or novel processing appliance into an existing program with minimum effect on the rest of the system. By localizing changes in the “parent” program to the subroutine level modular design philosophies are respected and debugging the resulting system is considerably eased.

3.1.1 A Functional Memory model

Of many approaches to hardware-based external processors, the implementation of co-processors as Functional Memory offers advantages to the non-specialist wishing to develop an attached hardware processor. This design pattern provides a good model for implementing Hardware Subroutines as transformation engines operating on the input parameters. For application to typical scientific problems, however, this technique needs to be extended to allow operations on vector and array data types. The presence of an intermediate layer of host processor code in the HWS architecture provides the ability to implement these more complex parameter operations, and allows use of a simpler internal processing kernel. The additional overhead cost of such pre- and post processing must be carefully considered, however.

Applying the FM model to the internal organization of the HWS suggests an array of identical processing kernels implementing the core algorithm of the application in hardware, permitting parallel computation on blocks of input data. Among the desirable features of this model are its regular structure, which permits easy generation of multiple kernels using built-in VHDL generation schemes, a modular design which is easier to understand and debug, and a “thin” architecture which reduces clock and signal delays within the FPGA device.

3.1.2 The Functional Memory timing model

The premise of a Functional Memory architecture is that, as in a spreadsheet, computations are triggered by modification of an input value and appear to occur instantaneously to the user. While truly instantaneous computations are impossible to achieve, parallel hardware processing elements can be arranged so that their processing activity overlaps with writes to succeeding elements. If the time required to finish a computation is short enough that it is completed before the host processor is able to read the result, to the host the computation appears to consume no extra time. Under these conditions, which I term the Functional Memory timing model, the computation is limited only by the time required to transfer the input data to the external device and to read back the results. Whereas the time required to complete a computation on a sequential processor is the sum of write, compute, and read components, the parallel hardware elements of a FM model system complete their work while the host computer is still busy transferring data to succeeding processing elements. This time depends on the transaction speed of the bus as well as the number of parallel elements.

3.1.3 The HWS architecture

The basic architecture for implementing a HWS can be described as a custom Processing Element (PE) implemented in an FPGA. This element is attached to the host processor by an interface that maps the input and output of the PE (typically an array of 32-bit registers) to consecutive memory addresses in the bus-address space of the host. The details of this mapping are hidden from the user by the driver software, typically provided at the lowest levels by the hardware board vendor and supplemented by customized higher-level interfaces developed as part of the HWS design process.

The internal structure of the PE is customized to fit the application and can include features such as an array of processor kernels as shown in Figure 1, local memory, and other functional elements such as address generation logic, FFT engines, or content-addressable memory arrays. Restricting the PE interface addressing to consecutive address blocks permits the data transfers to be done in bursts of data words, and more importantly allows the internal structures of the PE to be automatically generated.

3.2 Requirements and limitations

The HWS is not a universal solution to creating a custom computing appliance. Restrictions on both the type of problem and host environment suitable for use with the HWSA result from limitations in the interface and the HWS system model.

3.2.1 Appropriate HWS applications

Application domains and algorithms suitable for conversion to HWS implementations share several characteristics, key among them being computational parallelism. Moreover, the complexity of the individual computations is an extremely important issue because the slow data transfer rate of commonly available external system buses imposes a heavy time cost to moving data out of the host system’s CPU/memory space. Applications that involve simple computations on a few operands at a time are poor candidates for HWS conversion even if the operations can be performed in parallel. The benefit of performing the computation on parallel hardware can be significant, however, if the number of operations on a set of data elements is large compared to the number of data transfers required to move the operands to and from the HWS environment. Recursive algorithms also are usually poor choices unless the storage requirements for their intermediate contexts are known before the program is run.

Applications which normally would be considered suitable candidates would involve processing large arrays or streams of data and include computations which access many data elements for a single result. Examples that have been suggested include image convolution, searching using content-addressable memory, FFT-based image processing, and data compression.

3.2.2 Host system language and hardware requirements

Implementation of the HWS requires that the programming language of the high-level system provide mechanisms for accessing external system resources on the host computer as well as a suitably general hardware co-processor peripheral device. Most commonly used languages do provide such facilities, often in the form of packages for implementing Remote Procedure Calls (RPC) to access processing resources on separate machines. In fact, the HWS approach can be viewed as a form of RPC, although the remote processor would typically be located on a bus within the local computer. Similar operations to prepare and transport data to and from the HWS are required, and many of the drawbacks of the RPC approach are common to HWS as well. In particular, the overhead of transporting the procedure data outside of the processor-memory context is a significant issue.

The ultimate performance of a HWS depends on components of program overhead, data transfer requirements, and hardware computation speed. The key to a successful HWS implementation is minimizing the time required to transfer data across the system bus to the external hardware device. This is limited by bus setup time and basic bus clock speed.

3.2.3 Programmable logic hardware

The resources available on the external board constrain the applications that can be executed successfully on the system. A typically available FPGA today can contain more than 150,000 equivalent gates, and 4-million-gate FPGAs have been announced [12]. Complex functional elements such as internal memory arrays and flexible routing structures are also commonly available. This level of resources, combined with access to large amounts of inexpensive memory, permits the implementation of complete processing systems capable of handling multi-megabyte data arrays and complex functionality.

3.3 Evaluation of potential performance

The feasibility of using HWS in a given application rests on demonstrating that a suitably complex processing engine can be implemented within the constraints of available hardware platforms, and that the performance of such a device will be significantly better than performing the same computations on a conventional computer.

Development of a sufficiently detailed model to allow accurate functional simulations is not a trivial task, as it requires generation of a complete behavioral description of the system and bus interface as well as the actual processing logic. If such a VHDL model of the interface is available, combining this model with the PE logic into a complete model can produce very accurate performance and timing data. However, reasonable estimates of the eventual performance of the system can be made without such detailed simulation by utilizing performance data measured on the target system and proxy measures for the size and speed of a single processing element. The system's overall performance can then be approximated by adding the external call overhead of the high-level program, the time needed to transfer data to and from the external processor, and the cycle time of the hardware computation.

3.3.1 Proxy measurements

Some aspects of the overhead involved in accessing external resources are independent of the particular algorithm implementation. The time involved in making the external call from the high-level language can be measured for various "stubbed" calls to determine the base cost of making the context switch as well as dependence of the call overhead on parameter type and size. These measurements are straightforward and can be expected to scale with the characteristics of the computer, allowing, for example, estimation of the impact of increasing base processor clock speed. For most common problems, however, this source of overhead is insignificant compared to the time required to transfer data between the HWS and the host processor.

Parameter conversion and reformatting costs are also measurable independent of the hardware implementation. The time required to transform floating-point values into an integer representation, for example, is a cost that depends primarily on the amount of data passed to the subroutine and can be extrapolated from a basic set of measurements. Given specific hardware platform characteristics such as bus interface and FPGA type, standard values for data throughput to and from the board as well as configuration and setup time can be determined. A standard test program can be designed to measure these types of overhead components and provide a set of proxy estimates, parameterized by data type and array size, describing the external call performance of that host system.

3.3.2 Simulations

From a standard representation of the logic in the subroutine to be translated into hardware, a base implementation description can be generated in VHDL. Simulation of this hardware description using the characteristics of the FPGA device to be used provides confirmation of the correct operation of the logic as well as timing information. It is neither practical nor necessary to simulate processing of an entire input array to obtain performance data for a single processing kernel as value would be expected to scale directly with array size.

3.3.3 Performance comparison

Combining these values with the characteristics of the data to be processed allows the user to determine the relative benefit of translating the subroutine into hardware by directly comparing the actual performance of the software-only implementation with the estimates. It is expected that these estimates can be made lower bounds on the HWS performance, as hand-optimization of the hardware design can yield further improvement. The main objective at this point, however, is determining whether continuing with the translation is reasonable.

3.4 Design support framework

Design and testing of a HWS system is supported by a framework of interfaces, drivers, and implementation tools. These tools can be seen as a development environment supporting the researcher in the design and test of the implementation. Integrating a complete set of such tools into a single development environment is difficult due to the inconsistent nature of their interfaces, however efforts to utilize scripting and project management tools to automate some of the processing have been successful. Several of the vendors provide “design flow” tools that allow configuration of processing steps and tool options to occur in a single high-level shell. Similarly, makefiles can be used to record and standardize options for a particular device or configuration.

3.5 The HWS design process

The design of a HWS begins with a functioning software algorithm, expressed as a high-level subroutine, and proceeds through iterative cycles of estimation, implementation, and evaluation to arrive at a fully functional hardware replacement for the original routine. Estimates are used to determine the potential performance resulting from the characteristics of a proposed implementation before committing effort to the detailed design. Various software tools are used in a series of design stages to translate a behavioral description of the computational algorithm into a gate-level logic image defining the internal configuration of the FPGA. Simulation of the design description, using a commercial VHDL simulation package, can be used to verify the operation and performance of the design before trying to test it in the combined system, reducing debugging requirements.

A modified version of the original subroutine software is written to prepare input data and call the HWS version of the routine. This new routine maintains the same interface signature and parameters as the original version but contains code to parse the input parameters, make any needed conversions, and issue calls to the external hardware interface API. This permits the HWS to be directly linked into programs using the original software subroutine.

Finally, the hardware implementation is tested by direct comparison with the original software routine for both accuracy and performance. Results of these tests can be used to drive further iterations of the design if needed.

4.0 Hardware Subroutine Design

The following sections detail the progression of steps that are taken to design, implement, and evaluate a HWS instance. Several key decision nodes are identified at points in the process where the designer has access to data that can help focus following efforts in the most productive direction. The process is not intrinsically limited to any specific vendor's tools or hardware components, and several choices are available for most of the elements including the hardware card. To provide a clearer illustration of the design process an example implementation in terms of specific tools and hardware is presented.

4.1 HWS development environment

The tools and components required to implement a HWS instance are identified in the following sections, and the specific elements chosen for this project are described.

4.1.1 FPGA hardware

A suitable hardware peripheral is essential to successfully implementing a HWS is. In addition to a large programmable logic array, a standard, high-speed interface such as PCI or CardBus is needed, and a host system driver library implementing a basic Application Programming Interface (API) must be obtained or developed. Additional resources that can be useful depending on the application include local memory and external input/output ports. While it is possible to construct such a device from scratch, many commercial alternatives are available.

The hardware chosen for this project is the WildCard FPGA prototyping system available from Annapolis Micro Systems [13]. The internal architecture of the card is shown in Figure 2, below, as it might be configured for a typical HWS application. The card contains a single Xilinx Virtex XCV300E FPGA, two 256KB memory arrays, and a CardBus interface controller to interface it with the host computer. The system provides a software driver library to support C program calls through the bus interface and a VHDL model of the interface controller and the pad-to-bus interface of the card.

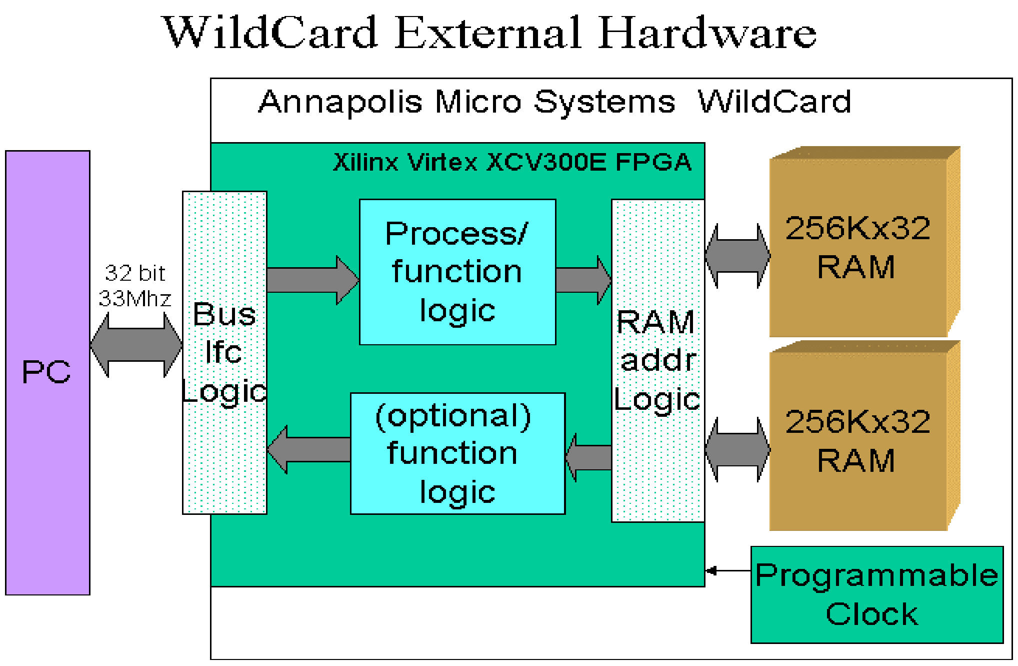

Figure 2. Block Diagram of the Annapolis Micro Systems WildCard

4.1.2 Implementation tools

The process of turning a hardware description into an FPGA programming image requires several specialized software tools. Most fundamental is the vendor-supplied “fitter” or place-and-route tools that map a logic netlist into a programming bit-stream for the FPGA. Xilinx provides a free toolset for implementing designs in the XCV300E chip that includes a design manager, timing analysis program, and place-and-route program [14]. This chip is also supported by their more complete Alliance series of tools. Most major EDA vendors, including Xilinx, Synopsis, Mentor Graphics, and Synplicity provide tools for synthesis of the detailed logic network description from high-level VHDL code; the Synplify Pro package from Synplicity was used for this project [15]. Finally, VHDL simulation tools can be used to validate the high-level logic and verify system timing; the Active-HDL4.2 program from Aldec is an example [16]; similar products are available from other vendors including Mentor Graphics. Unfortunately, the highest quality (and in some cases the only) options are usually expensive commercial products. It is possible, however, to obtain several of the software tools as evaluation or university program versions at significant discounts.

4.1.3 Operating system and host processor environment

The unique configuration of the WildCard in a PC-Card (PC_MCIA) format made it especially advantageous for this project. While Linux drivers are available for the card, the Windows 98 environment was (reluctantly) chosen for compatibility with the other tools required. Source editing and program builds were done with GNU EMACS and

Visual C++6.0. This project was implemented on a 333MHz Pentium II Compaq Armada laptop. The 32-bit CardBus interface used in this laptop PC specifies a 33 MHz bus clock and the WildCard vendor provides timing information indicating the setup requirements and transfer timing in terms of bus cycles.

4.2 Example subroutine description

The HWS design process begins with a working software program that contains one or more subroutines that are candidates for acceleration. Selection of these subroutines is made through use of conventional performance profiling techniques to identify program elements that consume significant amounts of processing time performing computations.

The test case chosen is a subroutine found in a large data analysis program for deriving magnetic field maps from spectrographic images of the sun. This program was written by scientists at the University of Hawaii Institute for Astronomy using the Interactive Data Language (IDL), a high-level data analysis language developed by Research Systems, Inc for use in analyzing large image, remote-sensing, and astronomical data sets [17].

The function computes an estimate of the first and second spatial derivatives of an image, using a simple shift-and-difference method. For historical reasons, the sign of the resulting derivatives is reversed from the strict mathematical definition. This algorithm, while simple, requires numerous array copies (12 total), and accounts for significant processing overhead in the larger program as it is repeatedly called during iterative curve-fitting procedures at several points in the analysis. The code is shown below:

```
;+
; calculates first and second spatial derivatives, for IVM seeing-pol
; correction, Simple shift-and-difference method.

FUNCTION IDERIV, I , nterms=nterms
;
; INPUT PARAMETERS
;     I = image to compute derivatives from
;
; OPTIONAL INPUT KEYWORDS
;     nterms = number of derivative terms to compute, default is 5. Other choice is 3.
;-
;calculate derivatives
    didx = (SHIFT(I,-1,0) – SHIFT(I,1,0) ) / 2
    didy = (SHIFT(I,0,-1) – SHIFT(I,0,1) ) / 2

    dixx = (SHIFT(didx,-1,0) – SHIFT(didx,1,0) ) / 2
    diyy = (SHIFT(didy,0,-1) – SHIFT(didy,0,1) ) / 2

    dixy = (SHIFT(I,-1,-1) + SHIFT(I,1,1) – SHIFT(I,-1,1) – SHIFT(I,1,-1) ) / 4

    IF KEYWORD_SET(nterms) THEN BEGIN
        IF( nterms EQ 3) THEN $
            derv = [[[didx]],[[didy]],[[dixx + diyy]]]
        IF( nterms EQ 5) THEN $
            derv = [[[didx]],[[didy]],[[dixy]],[[dixx]],[[diyy]]]
        IF( nterms NE 3 AND nterms NE 5) THEN $
            MESSAGE, ‘Ideriv: nterms must be 3 or 5.’
    ENDIF ELSE BEGIN
        Derv = [[[didx]],[[didy]],[[dixy]],[[dixx]],[[diyy]]]
    ENDELSE
    RETURN, derv
END
```

This subroutine is called against an image array at several points during the reduction of a typical magnetogram data set. These images are typically 512 x 512 pixels in size and are manipulated in floating point variables for convenience though the original data is 16-bit unsigned, containing approximately 10-12 significant bits of information. A complete image set contains up to 100 pairs of images and requires 512KB per image, a total of up to 100 MB per set. While the workstation used for reducing this data has ample memory to contain the entire image set at one time, the individual images are large enough that they exceed processor cache sizes and will result in main memory access during processing.

The subroutine computes the first and second spatial derivative terms in each axis and the first cross derivative of an image array, thus the minimum data transfer for this function is one NxN input array and five NxN output arrays. The core computation is the derivative estimate in a given axis (row or column). The second and cross terms can be obtained by repeating the estimation on corresponding outputs of the first derivative computation.

The most basic approach to design of a processing kernel would implement the computation of a single derivative term, and require a call for each derivative term or five iterations total. Since there is little difference between this and the sequential software-only computation, one would not expect any significant benefit from this implementation. An improved implementation might use two parallel processing elements to produce both the first and second derivatives from the same stream of input data, and obtain the xy cross derivative by processing the first result array again after transposition, eliminating one array write for each pair of derivatives.

4.3 Preliminary evaluation

Before committing effort to a complete design cycle for a HWS, the potential performance should be determined by a combination of measurement and estimation. Time requirements due to the overhead of making the external call can be measured with a specialized program, while the delay components involved in processing the computation can be approximated from the complexity of the logic required and performance of similar functional components. By decomposing the core algorithm into its functional processing steps, the delay requirements can be obtained for each and a total delay estimated based on clock cycle requirements for standard functions. Alternate implementation approaches can be compared in this manner as well.

4.3.1 Measurements

For the specific case described in this paper I have written a test shell in IDL that automates measurement of the execution time required by a candidate routine as well as other overhead components. A similar program would be written in C, for example, if that language were used for the main program. Characteristics of the host system measured in this way include the basic subroutine call overhead, data access time (measured as a per-element time for a large array), and the actual time taken to process test data using the original candidate software subroutine. Performance measurements are also made on the hardware peripheral to determine the data transfer rates achievable with the available driver software and interface. Because maximum theoretical rates are not often attainable in practice, these measurements are critical to preparing accurate predictive estimates.

4.3.2 Hardware performance parameters

The main overhead components consist of data conversions, data transfer, and subroutine setup times. In most cases, the initialization and configuration of the card can be performed in parallel with other processing tasks by calling a routine to initialize the device in advance of its use, and thus does not impact the performance comparison. However, if several configurations for the processing element were used in a single program the time required to reconfigure the HWS might become an issue. Program download time for this card is measured at ~65 milliseconds, and the entire configuration and reset cycle takes 230 milliseconds including reading the FPGA image file from disk. Data throughput is measured using a utility program which performs a burst write and read of a 256KByte (65535 word) data block to the onboard memory of the WildCard in programmed and DMA modes. As shown in Table 1, DMA transfers are approximately 3 to 7 times faster than programmed-I/O mode transfers but require a complex internal address generator to be programmed into the WildCard PE. For purpose of this work, however, this type of circuit is considered to be beyond the level of complexity likely to be implemented by the non-specialist.

While the ultimate throughput of a PCI or CardBus interface is theoretically 32MB/sec, measured rates of only ~20MB/s are typically seen, implying a minimum transfer time of roughly 190ns per 32-bit word, in or out of the accelerator board, using

DMA transfers. Under program-controlled register read or write accesses the transfer time is significantly slower, typically only 3 to 6 MB/sec (corresponding to 625 nS to write and 1.3 uS to read a 32-bit word to or from the external device). Modern FPGA chips are capable of operating with a clock speed of more than 100Mhz, or less than 10ns / cycle, and while one can expect there to be several levels of logic delay in the device circuits, this delay will still be much less than the data transfer overhead from the host system. While there is some extra overhead required in an external subroutine call compared to a standard function call, the context-switching overhead is comparable and would not be expected to result in a noticeable penalty.

Table 1. Performance test report, Annapolis Micro Systems WildCard/E

```
WILDCARD(tm) Performance Test
Setting the repetition value to 50
TESTING USING PARAMETERS:
Clock Frequency = 100.000000
# of Iterations = 50
Device Number   = 0

****************************************************
*           Configuration Information                                   *
****************************************************
Processing Element Information:
PeXilinxType  = XCV300E
PePackageType = PKG_BG352
SpeedGrade    = 6

Memory Information:
Bank 0:
SizeDwords = 65536 (262144 bytes), Speed = 10ns RAM
Bank 1:
SizeDwords = 65536 (262144 bytes), Speed = 10ns RAM

Version Information:
API 1.6, Driver 1.1, Firmware 2.4

******************************************************************
* TEST RESULT SUMMARY:                                           *
*                                                                *
*  Non-Burst PIO Performance:                                    *
*     Read Elapsed time  =   4.340 sec ( 86.8 ms/Iteration) *
*     Read Transfer Rate  =   3.0 MB/s                           *
*     Write Elapsed time  =   2.090 sec ( 41.8 ms/Iteration)  *
*     Write Transfer Rate =   6.4 MB/s                           *
*                                                                *
*  DMA Read (Board-to-Host) Performance:                         *
*     Elapsed time  =   1.200 sec ( 24.0 ms per Iteration)       *
*     Transfer Rate =  21.8 MB/s                                 *
*                                                                *
*  DMA Write (Host-to-Board) Performance:                        *
*     Elapsed time  =   0.220 sec (  4.4 ms per Iteration)       *
*     Transfer Rate = 119.2 MB/s                                 *
*                                                                *
*  Processing Element Programming Performance:                   *
*     Avg. Time per PE Program =  64.8 milliseconds              *
*       Total Time for PE Prog. Test  =   3.240 sec              *
******************************************************************
```

4.3.3 Baseline subroutine performance

The target performance of the original subroutine is measured using a test program to exercise the routine with a generic input set. Results parameterized by input sample size show both the small base call overhead as well as the per-element processing time (Figure 3). In addition, results for a version of the code that computes only a single derivative is shown in Figure 4 as a comparison for the kernel processing element's performance.

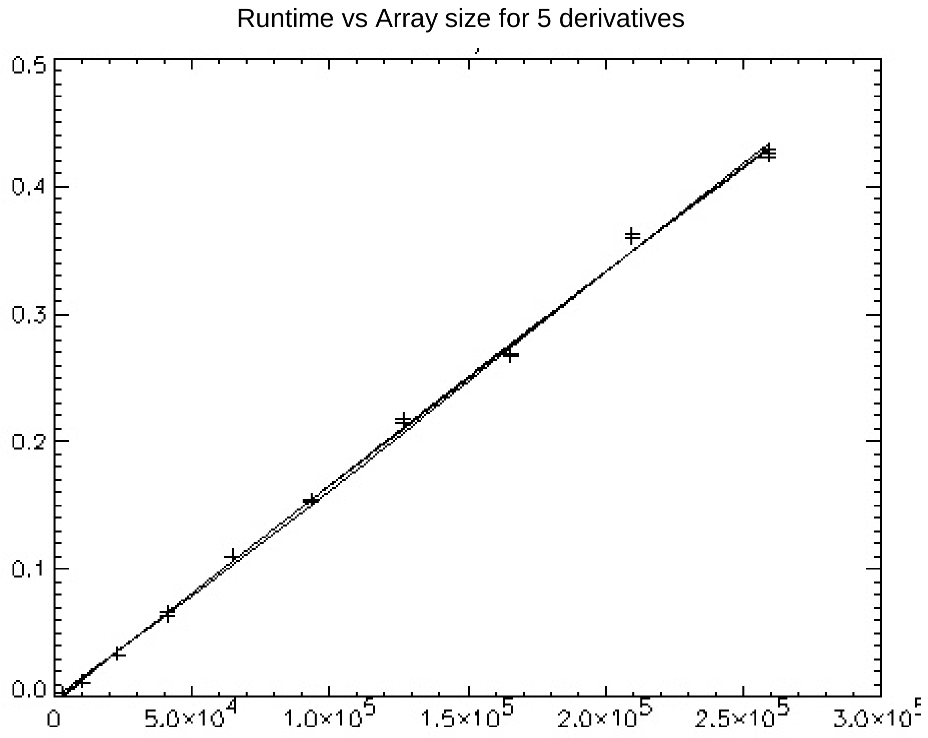

Figure 3. Computation time for complete ideriv.pro (5 derivative terms)

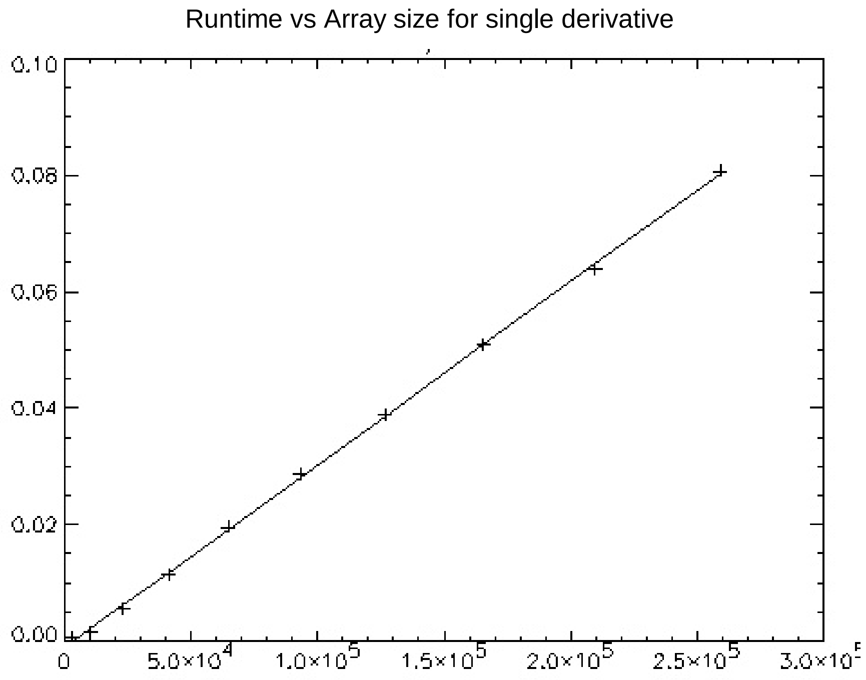

Figure 4. Run time vs. data array size for a single derivative term

The performance of the base subroutine can be computed from curves fitted to these measured values as shown below. Note that this is parameterized as a function of the array dimension N rather than the total number of elements in the data array.

t(derivative function) = 1.31e-006 * $N^{2.015}$ + 0.002 seconds

t(call_overhead) = 1.4e-007 *$N^{1.96}$ - 0.0006 seconds

The program used to make these tests called a function that made a simple assignment of every array element, thus the second equation includes both call overhead and minimum processing cost. At roughly 140 nS per data element, this value includes the time to pass a data element into any external routine, and is significantly less than the time to transfer data to the external hardware.

4.3.4 Feasibility determination

At this point, a basic feasibility assessment can be made. An estimate of the performance of the HWS combines the additional external-call overhead measured by the test shell, any data formatting or type conversion needed, and an estimate of the time required to transfer data to and from the HWS processing elements based on optimum use of the data bus. If the actual HWS computation can be accomplished within the time needed to transfer an input data block to the PE, the Function Memory timing model will apply, and the total HWS performance will be limited by data transfer overhead. If the time required to execute one element's computation using the base software routine is less than the transfer time for an equivalent input dataset, one immediately can conclude that the HWS implementation of this subroutine will not result in any performance benefit. An extension of this comparison parameterizes the result in terms of the input data set size. In general, the HWS approach becomes more beneficial as data size increases since the call overhead components are a smaller fixed percentage of the total for a larger input set, while the transfer overhead is a constant factor.

In typical situations, the overhead of additional subroutine context switches and slow data transfer speeds limit usefulness of the HWSA to applications whose processing time is very large, so that the fixed overhead components are averaged over a large number of elemental computations. In comparing potential applications, these overhead components have a much greater impact on a calculation involving a 400-point data set than for the 1 million elements in a 1024 x 1024 image array. Parameterization of the timing estimates in terms of input size will illustrate this effect, if it is significant.

4.3.5 Initial performance estimate

To make a complete estimate of the HWS performance, we must decompose the processing cycle into its basic components and measure or approximate them. The overall processing time required to complete a computation can then be determined by summing the elements that are determined to have a significant impact. Until a design is complete, estimated performance must be based on some assumed data I/O model. Three modes are available for this platform, of which two, the faster register mode and the programmed input-output (PIO) mode measured by the performance program are useable in this type of application. We can estimate that the register-mode data transfer timing would be 250 nS for either read or write based on bus cycle data provided for the WildCard peripheral, and the time to complete a PIO read or write can be computed using the measured throughput values from Table 1. Some additional overhead may be accumulated in the driver software calls, however this will not be determined until the system can be tested. The cycle time required for each mode is summarized below.

Table 2. Data transfer timing for WildCard/E IO modes

Mode	**T(write)**	**T(read)**	**T(total)**
Register	250 nS	250 nS	500 nS
PIO	625 nS	1.33 uS	1.96 uS

A basic computation core circuit for this algorithm would take no more than 3 bus clock cycles to complete, so we can estimate the longest per-element processing time, assuming a single element is computed between each data transfer and the PE clock is bus synchronous at 33 MHz (30.3 nS period), to be 91 nS.

The total processing time is estimated as the sum of the data transfer time (read plus write) and any processing time which does not overlap these transfers. In the case of a FM timing model system, the computations are faster than the transfer, and the processing time can be ignored unless there are significant setup requirements. For the core described above, computation time is much shorter than data transfer time even for register-mode transfers, and the total time will be dominated by transfer and setup requirements. Note that the card initialization is not included as this can be done ahead of time.

The ultimate performance that can be obtained for this function would require buffering the entire array on the card, thus permitting only one array write and five array reads, and would use block DMA to speed the transfers. The processing time in this case will be somewhat larger as the complexity of the processing is greater and will require transferring data from the onboard memory in addition to computations. It will require at least two additional clock cycles per pixel to transfer data to and from the memory buffer, as well as some additional overhead to set up the block transfers. Since the amount of memory on the card is not enough to store a whole 1024x1024 array, the computation would have to be done in eight blocks, adding to the complexity and processing time required in the calling software. In addition, it is probably not feasible to overlap data readout with processing because of memory access conflicts. The total time estimate would then be the sum of the data transfer time (at DMA rates) plus the element processing required for the five derivatives.

Table 3 summarizes these performance estimates by design, identifying the original subroutine (“Software”), the performance of basic core which does meet the FM timing model using register-mode (“Core reg”) and programmed-IO mode (“Core PIO”) data transfers, a design which computes both first and second derivative terms at once (“2-deriv”), and the optimum hardware design discussed above (“Optimum”).

Table 3. Summary of timing estimates for derivative estimator

Design	**T(write)**	**T(compute)**	**T(read)**	**T(total)**	**T(sec)**
Software	-	-	-	1.3e-06*N^(2.015)+0.002	1.53
Core reg	5 *X*t(wr)	91 nS/point	5*N*t(rd)	5*X*[t(wr)+t(rd)] (reg)	2.62
Core PIO	5*X*t(wr)	91 nS/point	5*N*t(rd)	5*X*[t(wr)+t(rd)] (PIO)	10.25
2-deriv	3*X*t(wr)	91 nS/point	5*N*t(rd)	X*[3*t(wr)+5*t(rd)] (reg)	2.09
Optimum	X*t(wdma)	5*N*(150 nS)	X*t(rdma)	X*[t(w_dma)+t(r_dma)] + t(proc)	1.25

Note: X is total input pixels, N is array width, total time est. for 1024x1024 input image,

t(wdma) = 67nS/pt, t(rdma) =360ns/pt, 30nS PE clock assumed.

In the best-case assumption, in which two derivatives can be computed for each write to the processing element, this implies that the HWS will not quite equal the speed of the software version of the code. *This is a key result and indicates that in a real situation, the conversion of this function into a HWS should not be pursued further with this architecture.* In this example case, the results also indicate that this function is effectively a lower bound on the complexity of a function that can be successfully converted into a HWS. Despite these performance implications, by completing the implementation and comparing the actual HWS performance with this estimate, we will obtain a useful validation of the prediction and a demonstration of the design process.

The estimates indicate that, if realizable, the optimum hardware design would result in performance which is about 20% faster than the original software. However, implementing the function in this way would require a significant effort to develop the hardware design, as an on-board memory controller and multiple line stores would be needed. For smaller array sizes, the benefit of the approach is reduced as well.

In the event that register-mode transfers cannot be implemented, we must use the programmed I/O speeds reported by the performance testing to estimate the data throughput. In this case, the data transfer time will dominate the total, and as shown, the results clearly indicate that the HWS would not be useful under these circumstances.

4.4 Processing Element development

The next step in the process is the design of a suitable hardware computation kernel or Processing Element (PE). The computations of the candidate subroutine should be broken down into the core-processing algorithm required to produce a single element of output data. This unit is then expressed in VHDL as a behavioral description of the algorithm, which will be synthesized into a hardware representation following a verification step. The design of this core element is constrained by the bus interface of the chosen system and the resources available on the co-processor card.

At the same time, a bus-to-PE interface design is written to instantiate an array of PE cores and connect them to the system bus signals. This code can be parameterized with VHDL generic structures to permit adjusting the size of the PE array as needed. The Functional Memory model of the HWS guides the design of this interface into a straightforward address and data connection design.

The core algorithm for the derivative estimator can be expressed as a processing pipeline, as shown in Figure 5. Input data elements are written into the pipe registers in sequence along a row or down a column, and for each input value, an output is read back. Parallelism can be exploited in two ways in this case: since row (or column) processing is independent, several can be computed in parallel, allowing burst writes and reads to be used, and by computing the second derivative directly from the outputs of the first kernel using a second processing kernel. In order to correctly compute the second derivative values, however, care must be taken to correctly pre-load the input data values as the registers are not directly accessible. A further development not undertaken here would provide direct-load access to these registers.

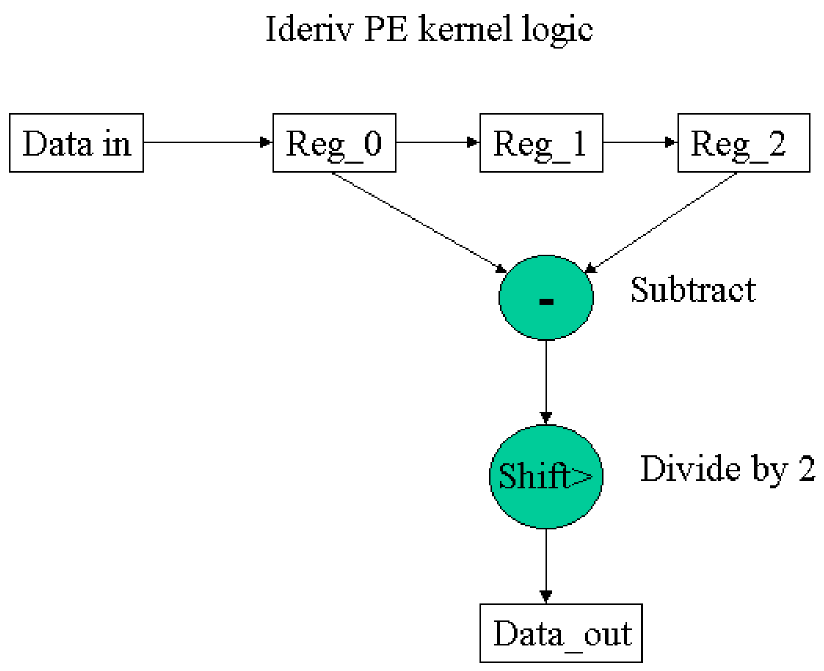

Figure 5. Kernel logic for approximating a derivative

This PE/kernel is expressed in about 120 lines of VHDL code and contains three 32-bit registers, a signed adder, and a Finite State Machine to control data transfer and processing. Another instance of the same PE could be connected to the output of the first to provide the second derivative computation. Alternatively, the input data can simply be run through the processor again, which is the approach taken here.

The PE interface layer provides connection of the kernels and automates generation of the kernel array. The VHDL fragment shown in Figure 6 illustrates this technique, in which a set of instances of a predefined component (here called deriv_est_krnl) are created and connected to sets of input and output signal ports.

Individual members of the sets are denoted by their indexes, for example the internal new_data register of the second kernel element will be connected to the register dk_new_data(2) in the generated context. Specifications of type and size for all of the referenced items are defined earlier in the VHDL file.

```
begin  -- de_fm1
    -- create the array of processing elements
    dk_gen : for i in 0 to (proc_els -1) generate
        dk: deriv_est_krnl
            port map (
                l_reset =>       dk_resets,
                p_clk =>         PE_clk,
                data_ready =>    dk_data_ready(i),  -- done flag reg
                new_data =>      dk_new_data(i),
                Data_in =>       dk_data_in(i),
                Data_out =>          dk_data_out(i),
                Data_rb0 =>          open,           -- dk_rb0(i),
                Data_rb1 =>          open,           -- dk_rb1(i),
                Data_rb2 =>          open);          -- dk_rb2(i)
    end generate dk_gen;
```

Figure 6. Example VHDL code for generating a set of kernel components

4.5 Design verification

A VHDL simulation tool is used to verify correct functioning of the processing kernel by preparing a VHDL driver program or "testbench" which provides inputs and outputs to the PE core. At this point, no system bus elements need be considered, and the element is tested as a stand-alone processor. The simulator program provides the ability to probe the model at the bit/waveform level as well as at higher levels of abstraction, so that both control signal timing and data values can be stimulated and checked.

If a complete behavioral model of the hardware peripheral is available (as it is for the WildCard system), a further design verification step can be carried out to test the function of the entire HWS system. By writing a high-level testbench using VHDL implementations of the host API system calls, the HWS can be exercised at the system level, including the bus interface, and proper operation verified.

Timing analysis can be carried out at this level to determine whether the logic design will operate at the desired clock speed to support the burst data transfer model assumed in the feasibility estimates. Timing data based on assumptions about the target FPGA can be added into the simulation to provide detailed results. Alternatively, this step can be deferred or skipped if the PE complexity is low, since more accurate timing information will be developed during the next phase of the process.

4.6 Netlist synthesis

The hierarchical VHDL description of the processing elements and interface are next synthesized into register-transfer-level (RTL) logic based on the resources of the specific FPGA to be used. The goals of this step are to obtain a complete representation of the HWS processing element that meets the timing requirements for maintaining synchronism with the host system bus and fits within the available resources of the FPGA. The input and output pins which connect to the bus and internal control circuits of the system and the critical timing paths associated with the external interface are specified in a constraints file supplied by the vendor as part of the interface package.

The synthesis tool (Synplify Pro) used in this project provides reports detailing the results of the synthesis run as well as RTL and device-native views of the circuits. The timing report shows the user critical timing path information about the design. Paths that do not meet the required latency constraints are clearly indicated, allowing the user to modify the VHDL design or apply specific timing constraints to the path to bring it into compliance.

An iterative process is pursued to find the largest set of PE kernels that will fit in the FPGA, while still meeting the timing and routing constraints. The architecture of the HWSA generally makes it easy to meet timing constraints because of its bus-synchronous characteristics. The 30.3 nS bus clock period implies a mean delay path of approximately 10 gate levels for signals that must meet single clock latency. As the number of PE kernels is increased, the number of clock cycles between data input (host write) and output (host read) increase correspondingly. Thus, for the 15-element design implemented in the current system, there is a minimum delay of 15 clock edges between read and write, allowing complex sequential circuits to be implemented. At some point, of course, available FPGA resources will limit the number of elements and thus one cannot implement an arbitrarily large array in order to gain more cycle time.

Coverage reports showing device resource utilization are generated during the synthesis process based on the target device specified during synthesis setup. The design shown here is the result of three design / synthesis cycles and uses approximately 50% of the available FPGA resources. Additional kernel sections could be added with a design change to provide a larger status register

4.7 Device place-and-route

A program image for use with the FPGA hardware can now be produced by translating the netlist produced by the synthesis tool into a bit-stream for the FPGA. Programs supplied by the chip vendor assign logic elements of the synthesized design to specific locations in the FPGA and define the interconnections between these elements. During this process design rules are checked to ensure that the design is physically realizable and detailed timing and resource assignment reports are generated. The procedure is automated with a makefile and the user is only required to intervene in the event of an error. The final routing report shows the FPGA resources used (Table 4). A timing report is automatically generated and in this case showed that all timing constraints were met easily.

Table 4. Mapping report summary for derivative estimator design

```
Xilinx Mapping Report File
Copyright (c) 1995-2000 Xilinx, Inc.  All rights reserved.

Design Information
------------------
Command Line   : map -p xcv300e-6-bg352 -o map.ncd
pe_deriv_est.ngd
pe_deriv_est.pcf
Target Device  : xv300e
Target Package : bg352
Target Speed   : -6
Mapper Version : virtexe -- D.19
Mapped Date    : Sun Feb 25 20:04:58 2001

Design Summary
--------------
   Number of errors:      0
   Number of warnings:  172
   Number of Slices:             1,596 out of  3,072   51%
   Number of Slices containing
      unrelated logic:               0 out of  1,596    0%
   Number of Slice Flip Flops:    2,628 out of  6,144   42%
   Number of 4 input LUTs:       2,505 out of  6,144   40%
   Number of bonded IOBs:           54 out of    260   20%
      IOB Flip Flops:                              108
   Number of GCLKs:                  4 out of      4  100%
   Number of GCLKIOBs:               3 out of      4   75%
   Number of DLLs:                   3 out of      8   37%
   Number of Startups:               1 out of      1  100%
Total equivalent gate count for design:  61,074
Additional JTAG gate count for IOBs:  2,736
 …
```

The final output of this step is a binary file containing a configuration image for the FPGA. This 229KB file is stored on the host computer hard disk and loaded onto the FPGA processor card under host control during the HWS/PE initialization process.

4.8 Integration

Several steps must be performed to prepare higher-level interface programs for replacing the original subroutine. A new version of the subroutine must be written in the high-level language used (IDL in this example). This routine is basically a shell which parses the subroutine parameters, prepares the input data, calls the external HWS driver, and performs any needed transformations on the output data before returning it. The external HWS driver is a C program which extracts the parameter data and actually issues the API calls to read and write data to the hardware. This small program is compiled into a shared object library and is dynamically linked when the IDL call is made.

In this example, the original IDL subroutine computes five derivative terms while the HWS estimator as built returns only one. Therefore, the replacement subroutine requires some additional code to transpose the input array and make five calls to the hardware PE. This can be done at either the IDL or C interface level, and in fact I have used IDL array operations to transpose the data for the cross derivatives while calling the hardware processor twice in the C interface code to obtain the second derivative, illustrating the flexibility of the layered HWS approach.

The result of the implementation process is a set of files containing the HWS processing element image, the C interface wrapper dynamic-link library file, and a replacement for the original IDL (or other high-level program) subroutine.

4.9 Test and evaluation

The final step in the process is to evaluate the HWS performance and accuracy as a replacement for the original subroutine. In most cases the test shell program used to evaluate the original subroutine can also be used to measure the performance of the hybrid HWS, providing a simple and direct comparison of results. The accuracy of the HWS output can be directly compared with the original routine for the same input. Satisfactory results indicate that the HWS can be used directly in the context of the original program by simply replacing the original software instance with the version that calls the HWS and re-linking the program. In the dynamically compiled IDL environment, no additional re-linking step is necessary, and the necessary files can simply be copied into the appropriate location for use with the original program.

Derivative estimates, computed with the base software and the hardware coprocessor, are shown in the images below (Figure 7). Differences in the output images are at the quantization threshold of the display program (brightness variations are a display artifact).

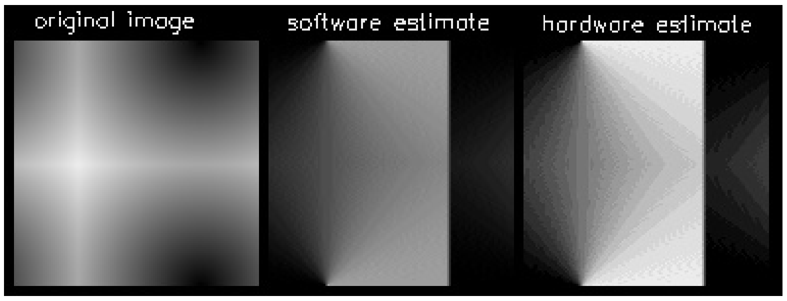

Figure 7. Comparison of column derivative (di/dy) of an image

In some cases, this comparison might be inadequate and direct subtraction of the result, for example, could be used to determine the level of compliance, as shown below. The differences are single-bit integer truncation errors; in some applications, this would be a problem and would require a different design for the PE kernel or input pre-processing to scale the data.

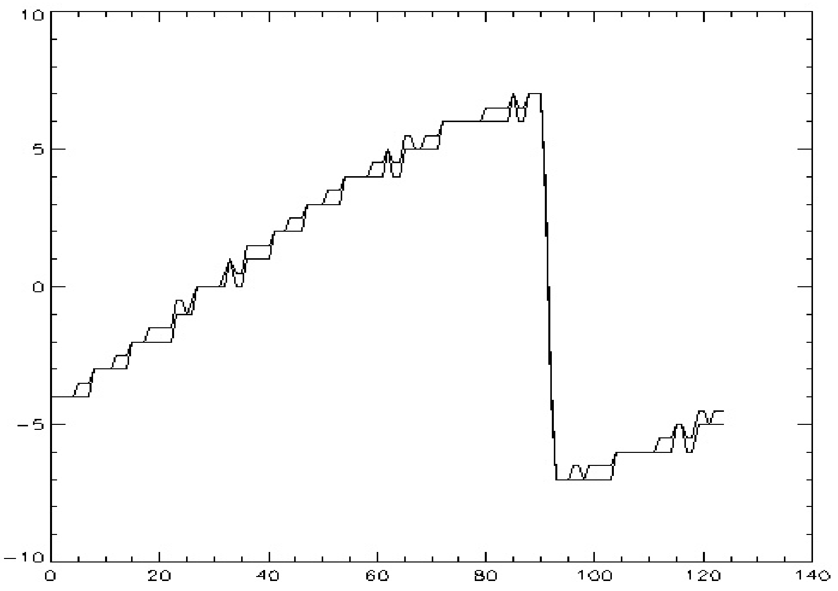

Figure 8. Comparison of a single row of the images shown in Figure 7

4.10 Performance results

HWS performance is determined by measuring the processing time required by the hardware version of the subroutine using the original test program. At this point, a design which does not meet expectations can be revised and a new estimation and design iteration begun. Comparison of real performance with the original estimates is important to validating the estimation assumptions as well as helping to determine whether any performance problem is due to a HWS design flaw or a system level problem.

During the implementation process for the example case, it was found that the vendor-supplied low-level interface was being incorrectly optimized during the synthesis step, preventing use of the burst-mode register-access I/O mode of the WildCard API. As the initial estimates show, use of this fast access mode is crucial to obtaining useful performance from the HWS design, because the alternative programmed IO mode transfer rates are an order of magnitude slower. Despite continuing to work with the vendor, no solution to this problem has been found, and the results shown here reflect use of the slower protocol.

Performance of the HWS derivative estimator implementation is compared to the measured run-times for the IDL base code in Figure 9. The impact of slow data transfer resulting from driver overhead is clear.

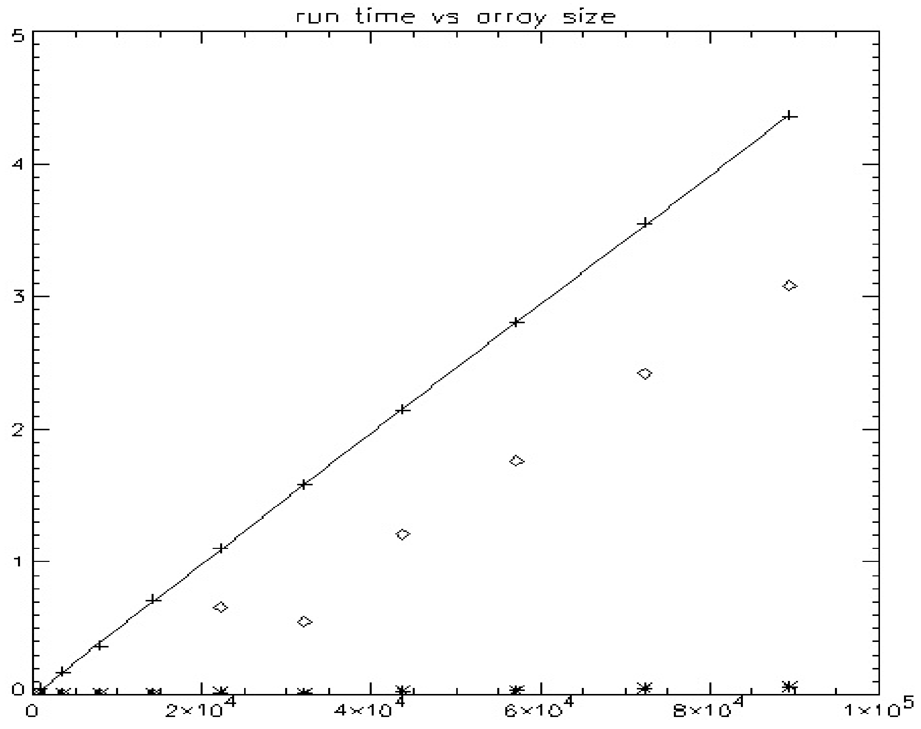

Figure 9. Performance of software and hardware versions of the derivative estimator

The upper line plots hardware-processed data. Stars plot software-processed data.

Diamonds denote t(software) multiplied by 55 for comparison with hardware results.

The performance that could be obtained from use of the expected data transfer protocol can be estimated by compensating the measured results by a factor reflecting the difference in write performance. The factor is estimated from measurements of data transfer speeds using another PE image and program provided as an example file by the card vendor.

4.11 Hardware design alternatives

There are several other design approaches which could further improve the performance of this implementation. The decision to pursue these efforts should be guided by performance estimates. Since the input data for which this program is designed is fundamentally 16-bit unsigned intensities and the PCI bus (and the processor) are 32 bit architectures, it would be straightforward to pack two pixels into each read and write and implement two sets of 16-bit pipelines in the hardware, one connected to the low two bytes and one to the high bytes. This would permit processing two rows or columns simultaneously in each bus transaction, reducing the data transfer overhead time immediately by a factor of two. However, this would complicate the interface software layer, and might pose underflow problems in the case of large operands.

A more complex approach to implementing the function would involve processing the data from host to card memory using a PE design similar to the present example, followed by reprogramming the FPGA and using DMA transfers to move the data back to the host in a large block of data. While significant time is required to re-program the PE image, the increased data write speed might compensate by reducing the largest component of overhead in the present design, reading back the computation results. For data arrays of up to 256x256 elements, the hybrid design described could accumulate the entire result for two derivatives in on-board memory, resulting in a processing time approximately equal to the time to write the data plus the time to program and DMA the memory array back. Potential performance in this case would be estimated (using the parameters established in section 4.3) at 520 mS to compute each pair of derivatives for a 256x256 image. However, the measured software-only performance for this array size is only 25 mS – clearly, the effort required by this more complex approach will not be justifiable.

Finally, the ultimate speed depends on the hardware - if a system supporting the 66MHz PCI bus is available, and faster gate arrays utilized, additional reductions in both transfer time and computation time are possible.

5.0 Example II - Convolution

The convolution operation is commonly used for filtering communications signals as well as images and provides an input recovery tool in situations where there is useful prior knowledge about either the signal disturbance or input. In astronomy, an image taken through an imperfect optical system is often processed by convolving the sampled data with a conjugate model of the optical system to remove fixed distortion components. Convolution has been proposed as an application which would be suitable for implementation on a Functional Memory processor and its characteristics suggest that a HWS realization would also be advantageous [7].

The 2 dimensional convolution operation for an input image A and an *l x l* kernel K is defined by the equation

$$\mathbf{Y}(t,u) \; ? \; \frac{1}{S} \overset{l?1}{\underset{j?0}{?}} \; \overset{l?1}{\underset{k?0}{?}} \mathbf{A}_{t \; ? \; j,\, u \; ? \; k} \mathbf{K}_{j,\,k} \qquad [18]$$

A direct implementation of the function involves summing the products of each of the kernel elements with each of the input array elements, an order O(N^2) operation. An alternative approach uses Fast Fourier Transform (FFT) techniques to convert the function to a multiplication in the transform domain, leveraging the O(N log(N)) behavior of the FFT to improve performance [19]. An implementation using the HWS approach can also take advantage of parallelism in the computations to provide a performance improvement.

5.1 Performance estimation

The performance of a conventional convolution function depends strongly on the size of the filter kernel array, and for moderately sized kernels measured processing time on a 256 x 256 image begins to exceed the transfer time that would be required to pass the same data to the HWS over the system bus. The implementation goal is a HWS which satisfies the FM model of having essentially no additional computation time above the time required for data transfer. In this case, computation of a result will be completed in less time than is required to read back the previous result, and the HWS computation will be O(n) with respect to kernel size (where n is the total number of data elements in the kernel; expressed as a function of the kernel dimension for an NxN array, it is $O(N^2)$). Even though the data transfer rate is severely limited, this will result in a performance advantage for the HWS above some data array size. Figure 10 compares predicted and measured subroutine runtimes as a function of kernel size, and shows that for a 256x256 image this crossover occurs at a kernel size of 13x13.

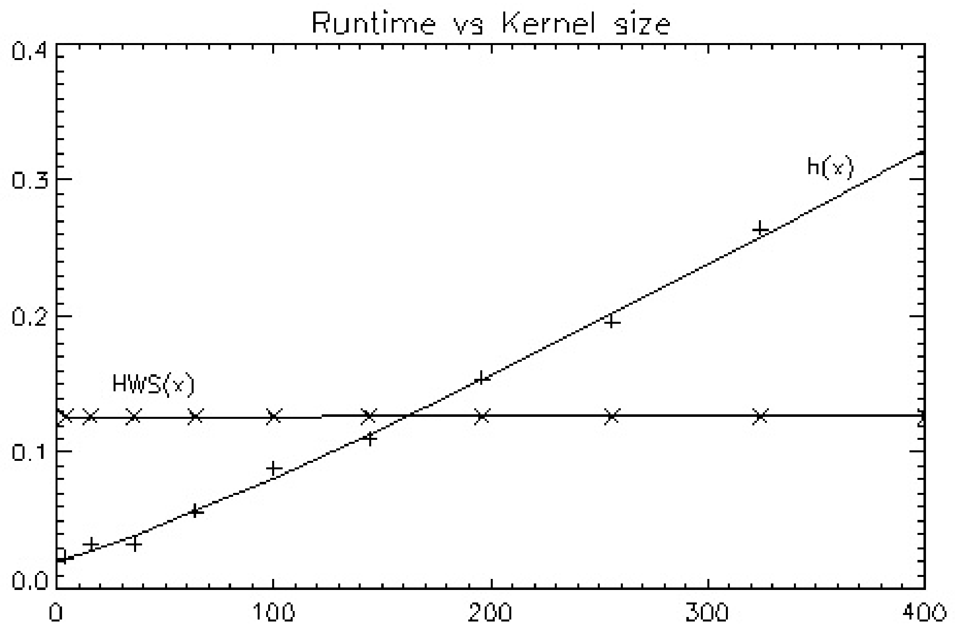

Figure 10. Comparison of convolution for ideal HWS and software-only (h(x))

Given that it is possible for an ideal HWS to perform better than a purely software implementation, we must next find an internal architecture which supports the Functional Memory timing constraint and can be physically realized with the available hardware resources. Again, estimates of potential performance are used to evaluate whether an implementation can achieve the primary FM timing requirement.

5.2 Design alternatives

There are two basic approaches which could be implemented, an FFT-based transform solution and direct computation of the convolution terms, and several architectural options for both designs. The FFT approach is based on a two-dimensional implementation of the standard “block convolution” technique used in signal processing, and would use a pre-designed FFT module available from Xilinx. The direct computation approach would use a large array of multiplier circuits to compute the products in parallel and a cascade of adders to combine the products into the final sum. In either case the host will alternate between writing blocks of input data to the HWS and reading back blocks of results. For the purposes of the example, the problem can be constrained to use a 16x16 kernel (a convenient indexing size) and a 256 x 256 element array of 16-bit input data. The timing constraints for both approaches will first be discussed, and then the implementation risks compared for the cases which could be used.

5.2.1 FFT-based design

The block-convolution process requires a two-dimensional forward FFT of a block of data, a point by point complex multiplication with the transform of the kernel, and an inverse transform to provide a block of results. The transform block size is usually chosen as an even power of two to take advantage of internal symmetries in the FFT algorithm. In this case a 64-bit FFT core available from Xilinx [20] can be used for both of the transforms. Since a two-dimensional transform is required, a pair of cores can be used to process the row transform in parallel with the column computations; each core will consume 1161 logic slices, or about 15% of the total available. A pair of 64x16 memory buffers, implemented with internal Virtex structures, are needed for each to permit overlapping read and write operations. Alternatively, a single core can be used to perform the FFT on the array twice with a row-to-column transpose between the passes, at the cost of twice the computation time.

To arrive at a timing estimate, the approximate timing for the parts of the computation can be added. This circuit will not require any setup other than preloading the kernel data, which will be forward transformed off-line and the result stored as part of the configuration. Alternatively, to permit changing the kernel dynamically, the transform could be computed by the host system and the result written to a register array. The small size of the kernel would make the time required to complete this insignificant. The transform computation is specified to take 192 clock cycles per 64 element row to complete. Each block of data will need 64 FFT computations in each axis, and an additional 4 clocks would be required for multiplying the 64 x 64 sets of row and column results into the full FFT. The transformed block and the kernel are multiplied by a further parallel set of multipliers, and the same 64*(192) + 4 clocks is needed for the inverse transform. Assuming that a 50MHz clock (20 nS period) is achievable with the logic complexity required, the time required to compute an entire data block will be approximately 490 uS.

For a 64 point FFT using a 16 x16 kernel, the data block size will be 47 x 47 points, and 30 blocks will be needed to complete the computation. At the measured PIO rates of t(write) = 625 nS and t(read) = 1.3 us, transfer times will be 1.4 mS to write each block and 2.9 mS to read the result. The ~.5 mS computation time is therefore small compared to the time required to transfer the data block, indicating that the FM timing model can be used. Total process time will be dominated by data transfer, with some additional overhead to initialize and complete the first computation.

5.2.2 Direct computation design

Direct computation architectures compatible with the HWS interfaces might take several forms, noted DC-1 to DC-4 for reference to Table 5. The simplest internal logic would result from directly addressing a 16 x 16 array of multiply/add circuits to compute a single output, requiring explicitly writing each 16x16 sub-image to the HWS (DC-1). This would mean 256 writes for each result element, obviously an enormously high overhead. A second possibility would implement an entire column of cross-connected add/multiply webs, allowing writing a column of data and reading a column of results sequentially (DC-2). The timing requirements for this circuit are not aggressive because an entire column will take 256 x t(write) to transfer, giving the first output approximately 160uS to complete. The large number of adders will result in increased combinatorial logic delays, slowing this circuit more than the FFT architecture, so a slower 33MHz clock is assumed. In this case, the 7 levels of adders and the multiply will take only about 1 uS to complete, assuming the parallel multiply requires 4 clocks and 4-clock pipelined adders are used. However, this circuit would need 4096 16-bit registers (an array of 16 x 256) just to hold the data, as well as internal registers in the multiplier circuits, exceeding the available number of flip-flops on the device.

An architecture which uses the same 16x16 web of the first option but buffers the input data in a local RAM would require more complexity in the form of an address controller for the memory but would eliminate the redundant writes which limit the performance of the DC-1 approach (shown as DC-3). Data would again be written a column at a time, however in this case data already in the compute array would be shifted up one row and the empty row filled with the new element, along with 15 previous row elements from the RAM buffer. The resulting output would be written to a 256x1 buffer, to be read back in a burst after the column is completed. Some additional overhead would be needed to pre-load the proper data into the RAM buffer before the first array could be processed, however this is necessary in all of the cases.

A final possibility might leverage the reconfiguration ability of the HWS to copy the entire image array to the RAM in a single DMA operation (DC-4). The processing element would then be loaded with the processing logic, and the data moved back from RAM into the PE, and results read back into the host. Here reprogramming time becomes a limiting factor, as the 65mS to load the program and ~40 mS to reset the card will equal most of the processing time required by the competing options.

5.3 Design analysis - complexity and other risks

The expected performance of these alternative designs is summarized in Table 5. In order to make a direct comparison with the measured performance of the software implementation the total time required to compute the convolution of a 256 x 256 data array of 32-bit data with a 16 x 16 element kernel is listed in the right-hand column.

Table 5. Summary of timing estimates for convolution architectures

Design	**T(write)**	**T(compute)**	**T(read)**	**T(total)**	**Total**
Base	-	-	-	.0003 X^2.038 + .0203	201mS
FFT	N*t(wr)	4 uS/block	N*t(rd)	N*[t(wr)+t(rd)]	128mS
DC-1*	N*16*t(wr)	~1uS/ point	N*t(rd)	N*[16*t(wr)+1uS +t(rd)]	808mS
DC-2	N*t(wr)	~1uS/ point	N*t(rd)	N*[t(wr)+t(rd)]	128mS
DC-3	N*t(wr)	~1.1uS/point	N*t(rd)	N*[t(wr)+t(rd)]	128mS
DC-4	N*t(dma)+120mS	~1.1uS/point	N*t(rd)	N*[t(dma)+t(rd)]+120mS	211mS

Note: t(wr) = 625nS, t(rd) = 1.33uS, t(dma) = .067uS; time computed for 16x16 kernel, 256x256 data all except DC-1 overlap compute with T(read), base software-only time measured with test program.

From the design estimates, it is clear that options DC-1 and DC-4 cannot meet the timing requirements, and DC-2 can not be implemented, therefore they should be eliminated from further consideration. Either the FFT implementation or the memory-buffered direct computation (DC-3) will meet the O(n) performance goal if it can be implemented according to the estimates. Further analysis regarding complexity, data formatting requirements and complexity should be considered to determine the most promising approach to pursue.

The choice between two implementation alternatives can be approached from a risk-benefit perspective and the decision based not only on ultimate performance but also on the expected amount of effort, which is a strong function of internal complexity. In this case, either of the two approaches selected requires a more complicated internal structure than the derivative estimator example, making their implementation non-trivial.

The core FFT function is a tested component which should be trouble-free, however this application's requirement of a 2-dimensional complex multiplication as well as the application of the linear FFT to a two dimensional array are untested innovations. Timing issues which could be a problem include synchronizing the faster processing element with the system bus interface and successfully interleaving access to the input data array during simultaneous cross transforms.

A direct computation (DC) processing element must synchronize the system bus to the PE compute array as well as to the memory buffer; and in addition sequencing logic to control shifting of data between array elements and memory is required. Generation of the large number of product and sum terms can be done programmatically using VHDL generate structures.

Data scaling to avoid internal overflows is a concern with both implementation approaches, however the Xilinx FFT core is designed to handle this issue. The DC version will require the designer to explicitly handle scaling and possibly implement a block-floating-point normalization scheme.

Based on the timing constraints, internal structural complex multiplier, and integration uncertainties surrounding the unfamiliar FFT core modules, I consider the risk of that approach greater than the direct computation model. Completion and testing of a design with this level of complexity will require a considerable amount of time and effort, however, based on the successful implementation of the simpler derivative estimation example there is no indication that the process can not be successfully applied to this design.

6.0 Conclusions

This paper has illustrated an improved approach to designing and integrating a hardware coprocessor system. By constraining the implementation to replace a subroutine in a larger program context, the HWSA leverages the capabilities of the host computer and provides a well-defined interface to the coprocessor. Two example cases were explored to demonstrate the design process and the potential performance of this technique.

6.1 HWS examples

A Hardware Subroutine replacement for a simple derivative estimation function was successfully implemented and tested. While a theoretical design with better performance than the original subroutine was found, a physically realizable design showed much poorer performance because of limitations in the data transfer functions of the card vendor's driver software. While this implementation did not produce any useful performance improvement, the design process illustrated by this example is repeatable and can be followed to implement other more complex functions.

Analysis of several designs for a HWS implementing a convolution function showed that the estimated performance of the HWS would be faster than a software-only version of the function for sufficiently large data and kernel arrays. The characteristics of the convolution function make the HWS coprocessor increasingly advantageous as the size of the kernel increases, and calculations show that, for the hardware and computer used, the hardware approach would provide a 35% performance improvement for a 16x16 kernel and 256x256 data array. Larger kernel sized would realize greater benefits providing the required logic can be fit into the FPGA.

6.2 Advantages of the HWS approach

The cost-effectiveness of this type of hardware acceleration can be evaluated on several levels. In strict comparison to the obvious alternative of simply buying a faster computer, the roughly $2500 (discounted) cost of hardware and software added to development time might be difficult to justify only in terms of improving computational performance. However, as a platform for exploring the development of alternative computing resources and hardware/software co-development, I believe this approach to be very cost effective in both direct expenditure as well as development effort required as compared to the traditional approach to creating a hardware coprocessing system. The flexible yet powerful Functional Memory architecture and the clearly defined interfaces of the HWS model permit the developer to focus their efforts on implementing the core logic of their algorithm.

The estimation approach I outline here provides the designer a means to evaluate the likely outcome of a project before committing large amounts of effort. As seen in the derivative estimator case study, preliminary estimates indicated that successfully achieving a large performance improvement with this approach was not likely, as experience confirmed. The estimation technique was also shown to be useful in selecting among design options based on their potential performance.

6.3 Limitations and other concerns

Accurate performance estimation depends on reliable prediction of the performance of the individual components making up the system. If any one element deviates significantly from expectations, the effects can be quite significant. The heavy reliance of the HWS design methodology on commercial software tools exposes the user to the risk that problems with these tools can seriously limit the performance of the design and thus success of the project. In the first case studied here, actual performance of the system was an order of magnitude worse than expected due to several problems that combined to prevent host writes from completing in the predicted time. This could have a tremendous cost impact on a project if such an erroneous prediction leads to significant effort being misdirected into developing a system that cannot work.

The speed of the FPGA device has an impact on the performance of a HWS implementation by limiting the depth of the logic which can meet the FM timing constraint. For the examples treated here this has minimal effect due to the extremely high data transfer overhead and low internal complexity in the PE core logic. More complex functions, possibly including the convolution example might be limited by computation delays as the techniques used to reduce computation delays require additional logic in the form of pipeline registers and additional computation blocks. However, the base speed of the XCV300E chip is sufficient to permit 100+ MHz internal clock speeds, which should be sufficient to implement many applications.

Design of complex PE cores will increase the risk and effort involved in completing the implementation. While the same estimation process described here would provide a rough indication of whether the result will be useful, more detailed analysis to determine the system's potential performance would be justified by the greater potential cost of the effort. Such analysis and estimation cycles might be performed at several points during the design process to confirm that further effort is warranted as the design becomes more complete and estimates that are more accurate can be obtained.

As the internal complexity of a design increases, device resource limitations may also become a concern. While an arbitrarily complex design can be written and simulated, finite numbers of internal device elements and routing resources can make completing the translation of the design into an FPGA image impossible. The designer's only recourse in this situation is revising the design to reduce its complexity or use of specific features, or obtaining a hardware device better suited to the application.

The relative benefit of implementing a HWS coprocessor will continue to depend on the capabilities of the host computer against which it is compared. However, as conventional computer capabilities continue to increase, concurrent improvements in programmable logic devices, driven by the same fundamental technology advances, should permit the Hardware Subroutine to continue to be a competitive approach to performance improvement as well as a viable platform for research into novel computational techniques.

7.0 Further work

While I have shown that the HWS approach is feasible, the implementation procedure is still somewhat complicated and use of the technique at this point requires the developer to invest some effort in learning both the environments and the languages. Widespread use of the approach would require more integration in the design flow and standardization of interfaces. Other areas of investigation that might be considered include further work to expand and integrate the tools available to the HWS developer and effort to explore and define the domain of problems to which the HWS approach is suited. A complete framework for supporting the development of HWS might also provide tools for transforming the high-level code of the selected subroutine into a configuration for the PE kernel elements, generation of interface code to connect the data structures of the high-level program with the co-processor, and test facilities to validate that the alternate subsystem functions correctly.

7.1 Tools for automated production of HWS interface programs

The interface programs which connect the HWS processing elements to the parent program are similar in many ways to the "stubs" which are used to implement a Remote Procedure Call access to a network connected computing resource. Scripts could be developed to produce the parameter checking, conversion and marshalling routines needed to convert a base software routine into a HWS. This would simplify an aspect of developing a HWS implementation and reduce the chance of defect introduction.

7.2 Characterization of the useful problem domain

The limitations of data transfer overhead impose severe restrictions on the set of problems for which the HWS approach is useful and continuing advances in conventional microprocessor speed compete with much slower advances in interface technology. A more complete understanding of how to choose suitable problems and techniques for optimizing HWS performance would increase the utility of the technique.

7.3 Other applications for the HWS approach

This work establishes a foundation for developing hardware coprocessing systems that also can be viewed as an environment for research into the use of novel computing technologies. In addition to exploring application of the HWS to other performance accelerator applications, the HWSA can assist investigations into alternative processor architectures and parallel processing by providing a development environment and debugging support for these non-standard processors.

7.3.1 Comparison of alternate coprocessor architectures

The convenience of arbitrarily changing the internal functional logic of the HWS processing elements suggests that the HWS approach could provide a useful testing environment for evaluating novel processor architectures. In contrast to a conventional development environment, the flexibility of the host system to manage data I/O, store large amounts of test data and results, execute complex setup and analysis processing, and provide helpful debugging and user interface displays can enhance the testing and evaluation process. The results presented in this paper themselves benefited substantially from the capability of combining analysis processing with actual testing programs, providing comparative plots of performance results and fitting of predictive equations as outputs of the test software.

7.3.2 Investigation of an optimized DT processor architecture

Description of algorithm logic in Decision Table format provides a common language that can be expressed both in sequential software and directly in hardware. A specially designed Decision Table Processor could be implemented as a HWS processing element, and permit the direct execution of DT programs on a custom hardware platform. The capability of simulating a VHDL description of the HWS processing element before completing the design cycle permits detailed evaluation of the characteristics of such a processor even if hardware resources are not immediately available, and encourages comparison of various design approaches. The HWS provides a design methodology for creating such a processor and can provide a supporting environment for testing and analysis of its performance.

References

[1] "Implementing High-speed search Applications with APEX CAM", Application Note 119, Altera Corp, July 1999, <http://www.altera.com/literature/an/an119.pdf>

[2] Bennett T., Morgan F., Shearer A., Redfern M., "An FPGA- based time resolved data acquisition system for astronomical and other applications", Proceedings of the Irish Signals and Systems Conference, UCD, Dublin, 2000, pp. 336-341.

[3] Embedded Solutions, Ltd. www page: <http://www.embeddedsol.com>

[4] Shand, M. and Moll, L., "Hardware/software Integration in Solar Polarimetry," in *FPGAs for Custom Computing Machines (FCCM'98).* IEEE, April 1998.

[5] Digital Equipment Corp., PCI PAMette WWW page, <http://www.research.compaq.com/SRC/pamette/Background.html>

[6] Halverson, Jr., R. and Lew, A. "FPGAs for Expression Level Parallel Processing," *Microprocessors & Microsystems* **19** (1995), pp. 533-540.

[7] Halverson, Jr. R. and Lew, A., "Programming with Functional Memory," in *Proc. 23rd International Conference on Parallel Processing,* Vol. I, CRC Press, Boca Raton, 1994.

[8] Halverson, Jr., R. and Lew, A., "Programming the Hawaii Parallel Computer," *Proc. ACM 2nd Intl. Workshop on FPGAs*. Feb., 1994.

[9] Shand, M., "A Case Study of Algorithm Implementation in Reconfigurable Hardware and Software," In *Proceedings of 7th International Workshop, FPL '97*, Lecture Notes in Computer Science. Springer-Verlag, Sept., 1997.

[10] Frontier Design, <http://www.frontierd.com/eda/index.html>

[11] Skahill, K., *VHDL for Programmable Logic*, Addison-Wesley, 1996.

[12] Xilinx, 'Virtex-E Data Sheet', Xilinx, Inc, San Jose CA, 2000, <http://www.xilinx.com/partinfo/ds022.pdf>

[13] Annapolis Micro Systems, "Wildcard Reference Manual Rev 2.0," Annapolis Micro Systems, Inc, Annapolis, MD, 2000, <http://www.annapmicro.com/>

[14] Xilinx, "WebPACK ISE data sheet," Xilinx, Inc, <http://www.xilinx.com/products/software/webpowered.htm#webpack>

[15] Synplicity, "Synplify Pro data sheet," Synplicity, Inc, <http://www.synplicity.com/products/synplifypro.html>

[16] Aldec, "Active-HDL 4.2 data sheet," Aldec, Inc., <http://www.aldec.com>

[17] Research Systems, Inc., <http://www.rsinc.com>

[18] Research Systems, Inc. *IDL Reference Guide, Version 5.4,* Research Systems, Inc, 2000, p. 241.

[19] Selesnick, I. W. and Burrus, C. S., "Fast Convolution and Filtering," in *The Digital Signal Processing Handbook,* Madisetti, V., Williams, D., eds.. CRC Press, 1998, pp. 8-2 – 8-5.

[20] Xilinx, “High Performance 64-point Complex FFT / IFFT v1.0.3 Product Specification,” Xilinx, Inc. 1999, <http://www.xilinx.com/ipcenter/catalog/logicore/docs/c_fft64_v1_0.pdf>