Reconfigurable Computing Aspects of the Cray XD1

Craig Ulmer, Ryan Hilles, and David Thompson

Sandia National Laboratories*
Livermore, California USA
cdulmer@sandia.gov

ABSTRACT: *Reconfigurable Computing (RC) refers to the use of reconfigurable hardware devices to accelerate the computational performance of a system for particular applications. Cray's new XD1 computer presents an appealing substrate for RC research because it places Field-Programmable Gate Arrays (FPGAs) in close proximity to host processor memory. In this paper we present our early experiences with the XD1 in the context of RC. In order to gain more insight into the inner mechanics of the architecture, we have constructed four simple FPGA-based applications: a data transfer engine, a linear sorting array, a data hashing function, and a distance calculation kernel that involves double-precision floating-point operations.*

Keywords: XD1, FPGA, double-precision floating point

1. Introduction

In 2004 Cray Canada (formerly OctigaBay) of Cray, Inc. brought to market a new multiprocessor system called the Cray XD1 [1]. The XD1 is an attractive system for many in the scientific community because it provides a dense computing platform that is based on a familiar "cluster computing"-style programming environment. While there is high demand for this type of integrated system, there is another aspect of the XD1 that warrants a closer examination of its architecture. The XD1 is one of the first commercial high-performance computing (HPC) systems to include Field-Programmable Gate Arrays (FPGAs) as user-programmable accelerators that are located in close proximity to the host CPU's main memory. These FPGAs function as a means of offloading key operations into hardware, and enable the XD1 to serve as a platform for Reconfigurable Computing research.

1.1 Reconfigurable Computing

Reconfigurable Computing (RC) [2,3] refers to the practice of utilizing reconfigurable hardware devices to accelerate the computational performance of a system for a particular application. In this work a reconfigurable hardware device is programmed to emulate application-specific circuitry specified by the user. By adapting an algorithm to function as custom hardware, researchers have been able to achieve significant speedups over approaches that implement the algorithm in software [4]. While many reconfigurable hardware architectures have been proposed over the years, most RC research is based on Field-Programmable Gate Arrays (FPGAs). Commercial FPGAs are readily available and offer vast amounts of reconfigurable logic for emulating user-defined circuitry. For example, Xilinx offers multiple FPGA products [5] that operate at moderate clock rates (100-300 MHz) and can house multi-million logic gate designs.

1.2 RC Challenges

Reconfigurable hardware is an appealing option for HPC users because it enables application designers to construct custom processing architectures that can be optimized to fit the characteristics of their applications. However, it is important to observe that RC researchers have always had to face three main challenges in their work:

1. **System Integration**: In order for any accelerator to be relevant to end users, it must be integrated into a system in a way that enables data to be exchanged with the accelerator in an efficient manner. While many FPGA accelerator projects have produced exceptional on-chip results, performance is often lost because data is exchanged between the FPGA and host processor using a slow I/O interface such as PCI. What is needed is a system architecture

* Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94-AL8500.

that enables a tight coupling between the system's FPGAs and host processors.

2. **Limited FPGA Capacity**: The fact that FPGAs have finite resources places an upper bound on the size of the design that can be emulated in a single chip. This bound essentially limits the number of concurrent operations that can be performed in the FPGA and therefore affects performance.
3. **Programming Environments**: Hardware design is significantly more time consuming than software design. While compilers for higher-level languages exist [6], hardware designers typically implement their designs by hand using traditional hardware description languages (HDLs).

While these challenges have delayed widespread use of RC, recent events suggest that a new environment is emerging where RC can be utilized in practical situations. The integration challenge is being addressed by Cray and others with the development of systems that elevate the position of FPGA accelerators in the system architecture. The capacity challenge is being addressed by the FPGA industry, which is steadily increasing FPGA capacity and capabilities in order to meet market demand. While the programming environment challenge is likely to be a long term effort, a number of researchers are active in this area and are steadily advancing compiler technology.

2. Cray XD1 Architecture

The Cray XD1 is a new multiprocessor system that is a mix of commodity parts and custom design. Each 3U chassis in the XD1 architecture houses six compute blades, a high-speed interconnection network called the Rapid Array Fabric, six storage devices, and a service processor for monitoring the system's health. Each compute blade provides two AMD Opteron processors, memory, and network interface hardware to connect the blade to the system's Rapid Array Fabric. Internally, this fabric utilizes commodity InfiniBand [7] components for managing low-level data transfers. However, the XD1 employs its own communication library on top of this fabric, making it incompatible with the InfiniBand standard.

2.1 XD1 Blade Architecture

The architecture for a compute blade in the XD1 is depicted in Figure 1. Each blade features two 64-bit AMD Opteron processors that are connected through a cache-coherent HyperTransport [8] link operating at 3.2+3.2 GB/s. Each processor has four DIMM sockets for memory, enabling a blade to have up to 32 GB of memory. A second HyperTransport link connects one CPU directly to a network interface (NI) chip This NI is currently implemented in a Xilinx Virtex-II/Pro FPGA. Due to signaling limitations for this FPGA, the CPU-NI link operates at 1.6+1.6 GB/s. The NI has two connections to the Rapid Array Fabric, with each connection operating at 1+1GB/s (i.e., 4x InfiniBand).

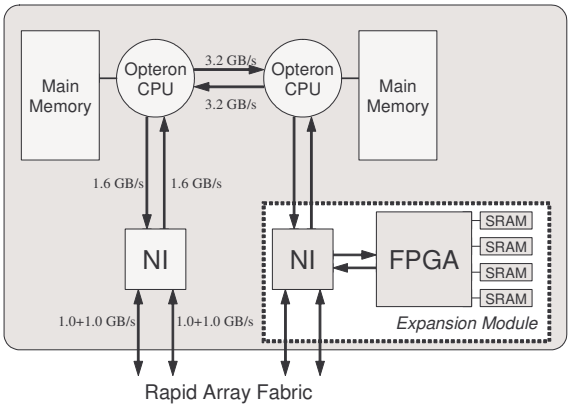

Figure 1: XD1 Blade Architecture

2.2 FPGA Expansion Board

In order to make the blade architecture more expandable, Cray Canada includes a socket interface on the blade that enables a custom add-on expansion board to be connected to the second processor's unused HyperTransport channel. The first expansion board that Cray has made available for this interface contains both networking and RC resources. In terms of networking, the board provides the blade with a second NI that has an additional pair of network connections. These connections can be attached to a second (add-on) plane of Rapid Array Fabric in order to increase the communication performance of the XD1.

The expansion board also provides a user-programmable FPGA that can be used as an accelerator for RC applications. As Figure 1 illustrates, the user FPGA is connected to the NI through a simplified version of HyperTransport. This interface enables the FPGA to read and write the host's memory, as well as respond to memory requests issued by the host. The expansion board is equipped with four banks of quad data rate (QDR) memory that is connected directly to the FPGA. This memory allows moderate amounts of data to be stored near the FPGA in order to improve performance. Cray provides a reasonable amount of support for utilizing the FPGA accelerator on the expansion board. Users are supplied with pre-built hardware cores for both the HyperTransport and QDR memory interfaces, as well as example designs that demonstrate how to use these units. Cray also provides a basic host-level device driver for the

FPGA board. In addition to loading and resetting the FPGA, this driver performs basic memory management functions (e.g., memory pinning and address translation).

2.3 Test System

After participating in an evaluation program with OctigaBay and Cray, we purchased an entry level XD1 to further investigate the system's architecture for RC research. The XD1 described in this paper is a single chassis system with twelve Opteron 248 processors that run at 2.2 GHz. Each of the six blades in the system is equipped with an expansion board, although the system is not equipped with the second plane of Rapid Array Fabric. The expansion boards are loaded with Xilinx Virtex-II/Pro 50-7 (V2P50) FPGAs. While this FPGA has only half the capacity of the largest V2P FPGA, it operates at the highest speed grade. Configuration files for the V2P50 are greater than 2MB and have load times of approximately 1.8 seconds.

2.4 Paper Organization

The intent of this paper is to document our early experiences with the XD1 in the context of reconfigurable computing. In order to better observe low-level performance characteristics of this architecture, we have constructed four simple applications for the FPGA accelerator. First, we describe a DMA data transfer engine in section 3 which exposes the rate at which the FPGA can exchange data with host memory. In section 4 we discuss a data hashing algorithm that computes the MD5 message digest identifier for an arbitrary length of data. Section 5 reports on a sorting algorithm that sorts 64-bit row values in a fixed-width matrix. In section 6 we describe our experiences with double-precision floating point for an algorithm that computes the length of a triangle's hypotenuse. Finally, we provide general observations and concluding remarks for this work.

3. Data Exchange

System integration is a key challenge in RC research because the manner in which FPGAs are inserted into a system's architecture dictates the rate at which the FPGAs can exchange data with other system resources. Historically, the most common path for integrating FPGA resources into a workstation has been through the use of peripheral device add-on cards. These cards facilitate communication between the FPGA and the host processor through standard I/O interfaces such as PCI or PCI-X. While these interfaces enable RC researchers to work with commodity parts, their low communication performance makes it challenging to implement a system where there is a tight coupling between FPGA accelerator and host processors.

The Cray XD1 is one of the first systems to connect FPGAs to the system using the high-speed HyperTransport interface. Our first application for examining the XD1's performance is one that measures the rate at which data can be exchanged between the FPGA and host memory. For this work we have constructed a programmable DMA engine for the FPGA that performs FPGA-initiated data transfers. The engine references host memory using physical addresses and thus requires a user application to pin and translate a block of memory using the FPGA device driver before work can begin. For comparison to host-initiated transfers, we have also constructed a host application that exchanges data with the FPGA using standard memcpy() operations.

3.1 XD1 FPGA Interface to Host Memory

Cray provides a communication core for the FPGA that enables data to be transferred between main memory and the FPGA. From the FPGA user's perspective this core is comprised of two separate interfaces: one for host-initiated transfers and another for FPGA-initiated transfers. Each interface has ports for read and write transactions. The *host-initiated* transfer interface is relatively straightforward to utilize because the FPGA user's circuits simply need to accept incoming write data and generate replies for incoming read requests. While reads involve the use of tag identifiers to correlate read replies to read requests, these tags can be managed with simple delay registers.

FPGA-initiated transfers are slightly more complex because the user's FPGA circuitry is responsible for orchestrating the transfers. In addition to supplying the physical address of the host memory used in the transfer, the user must adhere to HyperTransport's rules regarding alignment and burst size. Specifically, users cannot issue read or write transactions that cross 64 byte boundaries. This rule limits the maximum transfer size of a burst to 64 bytes and forces transfers that cross these boundaries to be broken into multiple transactions. Additionally, HyperTransport requires that host addresses be 64-bit aligned. Transfers to unaligned addresses can be performed through a mode that enables byte-lanes in a 64-bit word to be flagged as valid or invalid, but these transfers are limited to a maximum size of 32 bytes.

3.2 DMA Engine

The characteristics of HyperTransport make it awkward to work with the FPGA-initiated transfer interface. Therefore a basic DMA engine was constructed to automate the process of exchanging large blocks of data between the host and the FPGA. For the initial version of this engine, it is assumed that users will only exchange one or more 64-bit words of data with host memory that is 64-bit aligned. The engine is designed to automatically break transfers that cross the 64-byte boundaries of host memory into multiple transactions. The DMA engine provides signals to notify the user's logic when a particular transfer has completed.

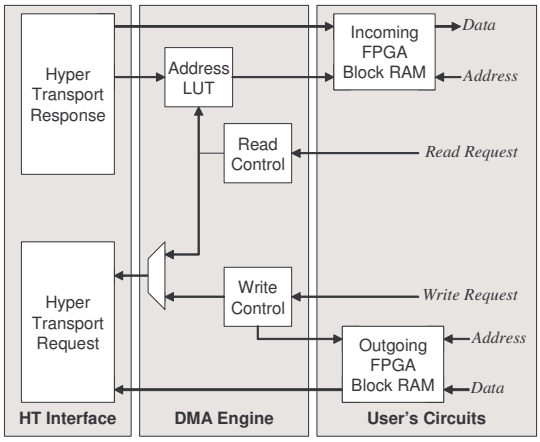

Figure 2: An FPGA DMA Engine

As illustrated in Figure 2, the DMA engine employs a memory interface for exchanging data with the user's circuits, and has separate ports for read and write requests. The memory interface enables the engine to manage its DMA transactions without having to perform complex synchronization signaling with the user's circuits. The process for a user circuit to perform a DMA transfer is as follows. For the (read/write) port, a user provides the DMA engine with the physical address of the host memory (source/target), the (target/source) address of the FPGA memory port, and the number of 64-bit words to be transferred. For write transactions, the DMA engine pulls data from the FPGA memory interface to fill the data section of the outgoing transfer. Read transactions are a two step process where (1) DMA requests are issued to the host and (2) incoming results are written to FPGA memory. In order to correlate requests to replies, a small lookup table is used to associate a transaction with a particular FPGA memory address. This table is necessary because it is possible for HyperTransport requests to be processed out of order. The DMA engine provides signaling to notify the user of when a read/write transaction has been fully issued, and when all of the replies for a read request have arrived.

3.3 Data Transfer Tests

A simple host application was constructed to observe the XD1's performance in exchanging data between the FPGA and host memory located on the local blade. The first set of these tests utilized *host-initiated* data transfers by simply memory mapping the FPGA into the host application's address space and using memcpy() to move blocks of memory. By default, the XD1's FPGA driver enables write combining for these types of transfers. Write-combining relaxes the processor's memory consistency model and enables Programmed I/O writes to be handled in burst transactions that are more efficient than individual writes. The second set of tests utilized the DMA engine to perform *FPGA-initiated* data transfers. In these tests, the host application sent a command to the FPGA to perform an FPGA-initiated transfer between pinned memory and FPGA memory. Following the transfer the FPGA used the DMA engine to write a data value to a particular address in host memory to signify the completion of the transfer.

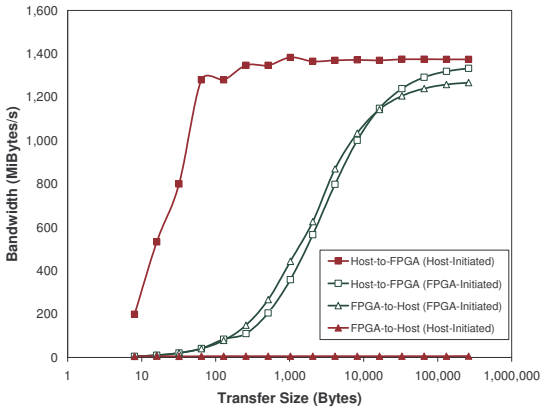

Figure 3: Data Transfer Performance

The results of these tests are presented in Figure 3. As expected, transactions that are performed with write transactions provide the best results. For host-to-FPGA transfers, write combining enables Host-initiated writes to have better performance than FPGA-initiated reads. Host-initiated writes are also advantageous because data can be sent from any virtual address, as opposed to having to come from pinned memory. For FPGA-to-host transfers, the Host-initiated reads gave terrible performance as expected, while FPGA-initiated writes excelled. It is important to note that with the exception of Host-initiated reads, all of the transfers approached the maximum achievable bandwidth of 1,400 MiBytes/s.

3.4 DMA Clock Frequency

The XD1 employs programmable frequency generators to drive the global clock signal of each FPGA in the system. While internal digital clock managers (DCMs) can be used to generate additional clock signals within the FPGA, doing so involves careful planning when moving data between different clock domains. Due to this complexity, it is tempting for FPGA designers to instead use the reference clock to drive all of the logic, and set the design to run at a clock frequency that does not exceed the rate at which the slowest components in the system can operate. However, doing so directly affects the performance of the DMA engine. In order to observe the effect of FPGA clock frequency on DMA performance, the FPGA-initiated memory transfer

program for FPGA-to-host transfers was run using FPGAs that were clocked at different speeds for each test.

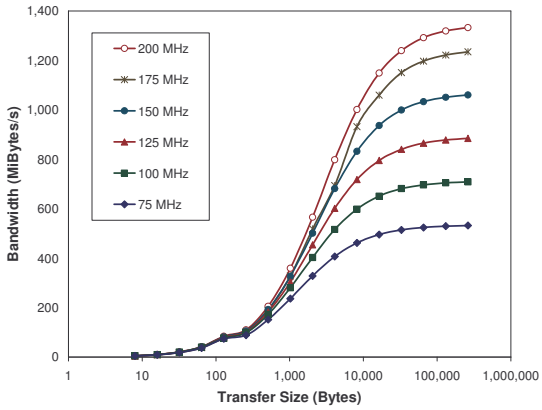

Figure 4: FPGA-to-Host Data Transfer Performance for Different Clock Rates

As Figure 4 illustrates, reducing the clock rate of the DMA engine reduces the rate at which data is transferred to the host. Similar measurements were observed for the other methods of data transfer. Given that it is unlikely that all user circuits will be able to run at 200 MHz, the next step in developing the DMA engine would be to enable the unit to operate at a different clock rate than the user's circuitry. This effort would require signaling to protect data as it moves between clock domains. However, this signaling would only need to be performed for the control signals of the API, as data is exchanged through a memory interface that is typically connected to a dual-ported memory block that allows reader and writer ports to be clocked at different rates.

4. Data Hashing Example

Our second example application for investigating the RC characteristics of the XD1 is data hashing. Data hashing algorithms examine an arbitrary amount of data and generate a unique identifier that can be used to reference the data and validate its integrity. A good hashing algorithm includes every byte of input data in the generation of the hash value, and performs sufficient permutations to make it non-trivial for a user to construct a dataset that has the same hash value as another data set. Because of these characteristics, data hashing algorithms can require moderate processing times on host CPUs. Given that data hashing algorithms are used as building blocks in many computer science applications, it is worthwhile to examine means by which their performance can be improved. For this work we focus on the MD5 message digest function.

The MD5 message digest function was created by Rivest in 1992 [9]. While better quality hashing algorithms have been developed since then, MD5 is still commonly utilized today to validate data sets. MD5 takes an arbitrary length of data and generates a 128-bit identifier. Internally the algorithm operates on a 512-bit block of input data at a time and runs the block through 64 computational steps. Each step modifies the 128-bit hash identifier using a computation that is comprised of a small number of Boolean operations, an addition, and a rotation. After streaming the entire data set through the computation, a zero-padded length field is passed through the computation to generate the final hash value.

4.1 Balancing the Hardware Implementation

The serial nature of the MD5 algorithm makes it challenging to construct a hardware implementation that exploits concurrency. While a block computation has 64 stages, these stages cannot overlap or be arranged in parallel because each stage reads and updates the hash value. Concurrency can be found and exploited at the block level, by fetching the next 512-bits of data while the current block is being computed. However, if it takes 64 cycles to perform a block operation, the fetch operation only requires data to be obtained at a rate of 8-bits per clock. Given the wide buses and high data-transfer rates in the XD1, it should be expected that this operation will be a compute bound problem for the FPGA.

Another complication in this design is that the computations in the 64 individual stages require multiple operations. Performing these operations in a single clock cycle decreases the maximum clock rate of the system. Conversely, splitting individual operations into multiple clock cycles improves clock rate but scales the overall number of cycles required to complete the operation. We constructed two designs to observe these characteristics. The first design performs each of the 64 stages in a single clock cycle, while the second design performs each stage in five steps, resulting in a total of 320 clocks to complete a block operation. Both designs utilize the same data exchange operations to obtain data from the host and store results back to host memory. For this application, the host program writes data into FPGA memory using write-combining Programmed I/O operations. The FPGA DMAs the result to host memory when the operation completes.

4.2 MD5 Results

Both MD5 designs were compiled for the XD1's FPGAs. As expected, the design that performed the computations in a single cycle had a much lower clock rate than the multi-cycle design (66 MHz vs. 190 MHz). A test application was constructed to measure the amount of time required for a host application to perform the MD5 computation for data sets of varying lengths using software alone and the two hardware accelerators. The results of these tests are presented in Figure 5. Unfortunately, the serial data flow of the computation did not result in an architecture that was faster than the host

application. As these measurements indicate, the improvements in clock rate obtained by the multi-cycle design were not sufficient to compensate for the larger number of clock cycles required to complete the operation.

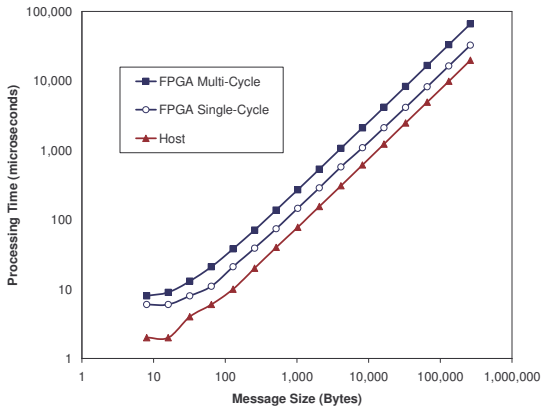

Figure 5: Processing Time for the MD5 Algorithm

5. Sorting Example

Our third application is a simple hardware implementation of a sorting algorithm for fixed width matrices. Sorting is a fundamental task in computer science for which many algorithms have been proposed. In general, the best performing sorting algorithms are efficient because they are optimized for the von Neumann style of processing architecture that is the basis for modern CPUs. This architecture assumes that a CPU has a small number of comparison units, a small amount of internal memory, and that it is expensive to bring data values into the CPU. An example of an algorithm that is customized to these assumptions can be found in quicksort [10]. Quicksort compares unsorted values to a pivot point instead of other unsorted values in order to reduce the number of data values that are pulled into the CPU. FPGAs free designers from the constraints of the von Neumann architecture and enable us to consider custom architectures for an algorithm. Our approach to implementing a sorting engine is to create a linear array of simple sorting elements that operate in a streaming manner. While this approach can only sort as many values as the FPGA has capacity for sorting elements, the hardware can be utilized as a building block for more complex operations.

5.1 Sorting Array

The hardware constructed for this work is a one-dimensional array of sorting elements. Each sorting element is comprised of a register for storing a single 64-bit data value, a comparison unit, and logic for sequencing data flow. The sorting element has two modes of operation: evaluate and flush. During the evaluate mode, a sorting element compares data placed on the unit's input to the internal value. It then places the larger of the two values in the element's internal register and the smaller on the unit's output port. In the flush mode, each element first moves its internal value to the output port, and in the following cycles moves data on the input port to the output port. The sorting element is trivial to describe in a hardware description language (90 lines of Verilog), and is easily replicated to build a sorting array.

The sorting array is a one-way pipeline of n sorting elements. A new value can be inserted into the array each cycle until the array reaches capacity. Once the final data value is inserted, the array needs n-1 clock cycles to allow all of the values time to bubble into place, and then an additional n clock cycles to push the stream of sorted data values out of the array. The advantage of this approach is that a large number of data values are compared against other data values at the same time, resulting in a total delay of 3n clock cycles to sort and store n numbers. This approach is also advantageous because it operates in a streaming manner that does not require the entire input data set to be available before sorting can begin.

5.2 XD1 Implementation

A design was constructed for the XD1 FPGA to enable the sorting of a fixed-length series of 64-bit numbers. This design instantiates a DMA engine, a sorting array, and a state machine for controlling the flow of data through the system, as illustrated in Figure 6. Once the FPGA is loaded with the design, a host application copies an unsorted array of data values to a region of pinned host memory and sends a command to the FPGA that specifies the address of the data, the address of where the sorted data should be stored, and the number of n-length blocks that need to be sorted. Upon receiving a command, the FPGA will issue DMA reads to pull each block of data into the sorting array. Once a block is sorted, it is moved into an outgoing FPGA buffer for DMA transfer to the host. Outgoing and incoming DMA transfers are scheduled to overlap in order to hide overhead when there is more than one block to sort.

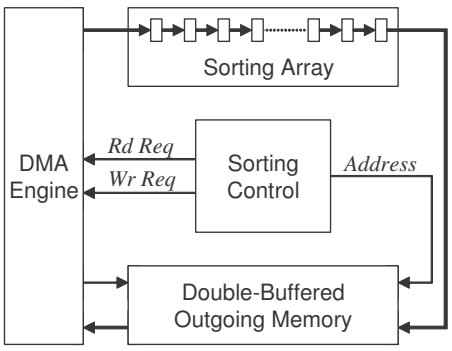

Figure 6: FPGA-based Sorting Engine

5.3 Performance Measurements

Through different experiments, we found that the maximum array size that could be implemented in a single Xilinx V2P50 FPGA was 128 elements. The entire design consumed approximately 87% of the V2P50's logic resources and had a maximum clock rate of 130 MHz. A host application was constructed to compare the design's performance against a CPU implementation of quicksort. In these tests a series of 128 word blocks were sorted to determine the sorting rate of both the FPGA and quicksort. Figure 7 presents the measured rates for the host and two FPGA scenarios. The first is for normal use ("Pre/Post Copy"), where the measurement includes the amount of time required for the host application to copy data into and out of pinned memory that the FPGA can work with. The second is for raw performance ("No Copy"), where unsorted and sorted data is resident in pinned host memory and does not need to be copied. Performance is reported in terms of millions of 64-bit words sorted per second.

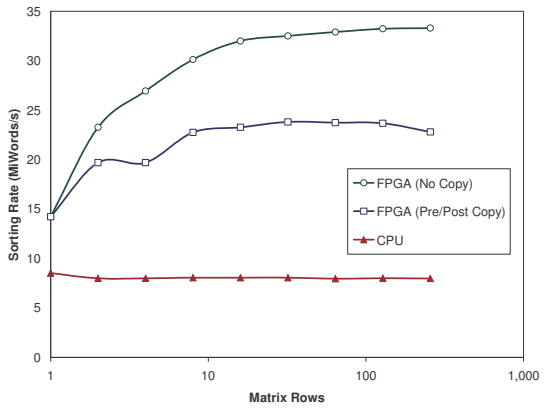

Figure 7: Sorting Performance

The host-based algorithm in these tests provides a constant sorting rate because the number of elements in a block does not change. FPGA performance increases with the number of blocks because multiple blocks enable host memory reads and writes to overlap. Overall, the FPGA accelerator gives a performance gain of 2-4x over the software implementation. Based on our experiments with smaller block sizes, this gain increases as block size increases. There are two areas of future work that are immediately clear. First, a simple mergesort engine could be placed after the array to buffer results and increase the sorting capacity of the FPGA. Second, we believe the design could be optimized to enable more sorting elements to be housed in the FPGA. Our approach in this paper has been to implement simple hardware and rely on the synthesis tools to produce a usable design. Hand placing sorting element logic would likely yield a denser design with faster logic.

6. Distance Computation

Our final test application for the XD1 is a simple distance computation that is performed in double-precision floating point. This computation determines the length of a triangle's hypotenuse from the lengths of the other two sides of the triangle. Based on the Pythagorean theorem, this computation equates to $c=\text{sqrt}(a^2 + b^2)$.

6.1 Floating Point in FPGAs

Floating point operations have historically been a weakness for FPGAs for multiple reasons. In general, floating-point computations are relatively complex operations that consume significant amounts of resources in an FPGA. Floating-point development is complicated by the fact that the IEEE standard for floating point contains a number of subtle behaviors for specific cases (e.g., rounding) that make full compliance non-trivial. While many researchers have implemented floating point units for FPGAs in the past, we do not know of any publicly available floating-point libraries that offer a complete set of operations that function at high speeds.

As a means of enabling RC research, Keith Underwood and K. Scott Hemmert at Sandia National Laboratories in New Mexico have developed an internal library of single- and double-precision floating-point cores for use with Xilinx FPGAs. These cores were written in low-level JHDL and hand-placed to maximize the performance of the hardware. Double precision floating point operations consume roughly 800-3,000 V2P slices (4-12% of a V2P50), and operate at speeds between 130-200 MHz. The library contains operations for add, multiply, divide, and square root. More information about this work can be found in [11,12].

6.2 XD1 Implementations

The availability of the SNL/NM floating-point cores enabled us to construct the distance computation in the XD1's FPGA hardware. The architecture of the unit is presented in Figure 8. Similar to the MD5 example, this design double buffers input data in a block of FPGA memory and requires the host to fill the buffer using Programmed I/O operations. The FPGA's input memory is 128-bits wide and 1,024 entries deep. This width enables two 64-bit data values to be passed to the computational circuitry every clock cycle. The V2P50 FPGA has enough capacity to implement the full data flow for this computation. Two multiplier circuits are used to compute a^2 and b^2 in parallel, followed by an addition circuit and a square root circuit. The total number of pipeline stages for the floating-point computation is 88. Result data is collected in an outgoing buffer and then transferred to host memory when a full block of data is available.

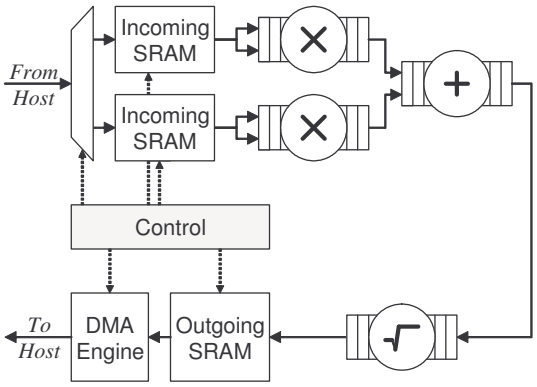

Figure 8: FPGA-Based Distance Calculation

6.3 Performance

The distance design was compiled for the XD1's V2P50 FPGA, and found to occupy roughly 39% of the chip. The maximum clock rate for the design was 159 MHz. A host application was constructed to exchange data with the FPGA in blocks of data that contained 512 pairs of input data values. Performance results are presented in Figure 9 for both the FPGA and an application that performs the distance calculation using the CPU. From these results we see that the large block size chosen for data exchange results in degraded performance until 2,048 computations are requested. At this point the data exchange pipeline is filled and overhead is overlapped. The CPU provides better performance until 64K computations are requested, at which point the FPGA provides slightly better performance. The drop in CPU performance is likely due to cache effects.

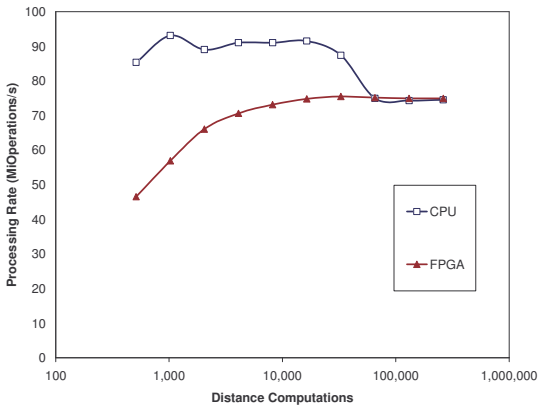

Figure 9: Double-Precision Distance Computation

7. Observations

A number of observations can be made from these experiments. First, the XD1 provides an architecture where FPGA accelerators have high-bandwidth access to host memory. An FPGA in the XD1 can read and write host memory at over 1,300 MiBytes/s, which is roughly 10 times greater than the rate that can be achieved in a system that employs an FPGA on a 32-bit/33MHz PCI interface. However, it is important to note that the HyperTransport link connecting the FPGA to the CPU operates at half the rate of the HyperTranspot link that connects the CPUs. This fact implies that the CPU still has a natural processing advantage that RC researchers must account for when developing FPGA accelerator circuits.

Another observation about this work is that some efforts were successful in obtaining a speedup while others were not. While the MD5 computation performs bit-wise operations that FPGAs typically excel at, the data flow for the algorithm did not enable us to extract enough parallelism to make the FPGA implementation competitive. However, this application demonstrated how a balance must be made between the number of operations performed each clock cycle and the total number of cycles required to complete the algorithm. The sorting and distance calculation algorithms were more successful because they provide parallelism that can be exploited in the hardware design. These examples demonstrate that the FPGA can compete with the host processor, even when large blocks of data have to be exchanged with main memory.

Finally, our experiments indicate that the XD1's V2P50 FPGAs are sufficiently large enough for users to begin considering their use for small but meaningful scientific computations. Using the SNL/NM floating-point library, we found that the V2P50 has the capacity to house a small number (10-20) of double-precision floating-point units. While this is a good start, we encourage Cray to consider newer, larger FPGAs in their future work, such as the Xilinx Virtex4.

8. Conclusions

The Cray XD1 is an appealing platform for Reconfigurable Computing research because it places FPGAs in close proximity to the system's main memory. In this paper we have examined how these FPGAs can be utilized as computational accelerators. Four simple FPGA applications were constructed to examine the underlying performance characteristics of the architecture, and to serve as early experimentation into how accelerator applications should be constructed. These examples demonstrate that the FPGA accelerators can perform useful work that is competitive with host-processor performance for certain types of algorithms.

9. Acknowledgments

The authors acknowledge the help and support of multiple people in this work. Steve Margerm at Cray Canada provided a great deal of assistance with the low-level mechanics of the XD1. In addition to answering numerous questions, he provided useful examples and insight into how to work with the XD1's FPGAs. We also acknowledge and thank Keith Underwood and K. Scott Hemmert at Sandia National Laboratories in New Mexico for allowing us to use their floating-point libraries. These cores worked as promised and operated at speeds greater than their specifications.

References

[1] Cray, Inc. "The Cray XD1 Datasheet," 2005.

[2] W. Mangione-Smith, V. Pasanna, H. Spaanenburg, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, and K. Palem. "Seeking Solutions in Configurable Computing," in IEEE Computer Vol. 30, Iss. 12. 1997.

[3] K. Compton and S. Hauck. "Reconfigurable Computing: A Survey of Systems and Software," in ACM Computing Surveys Vol. 34, Iss. 2. 2002.

[4] W. Smith and A. Schnore, "Towards an RCC-Based Accelerator for Computational Fluid Dynamics Applications," in The Journal of Supercomputing, Vol. 30, No. 3, 2004.

[5] Xilinx, Inc. "Virtex-4 Family Overview," 2005.

[6] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski. "Stream-oriented FPGA computing in the Streams-C high level language," in Proceedings of Field-Programmable Custom Computing Machines, 2000.

[7] InfiniBand Trade Association. "InfiniBand Architecture Specification Release 1.2", 2004.

[8] HyperTransport Consortium. "HyperTransport Link Specification," 2005.

[9] RFC 1321. "The MD5 Message-Digest Algorithm", 1992.

[10] C.A.R. Hoare. "Quicksort," in The Computer Journal, Vol. 5, Issue 1, 1962.

[11] K. Underwood and K. Hemmert. "Closing the Gap: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance," in Proceedings of Field-Programmable Custom Computing Machines 2004.

[12] M. Haselman, M. Beauchamp, A. Wood, S. Hauck, K. Underwood, K. Hemmert. "A Comparision of Floating Point and Logarithmic Number Systems for FPGAs," in Proceedings of Field-Programmable Custom Computing Machines 2005.