A FPGA Hardware Solution for Accelerating Tomographic Reconstruction

Jimmy Xu

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Electrical Engineering

University of Washington

2009

Program Authorized to Offer Degree:
Department of Electrical Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a master's thesis by

Jimmy Xu

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____

Scott A. Hauck

_____

Adam M. Alessio

Date: _____

University of Washington

Abstract

A FPGA Hardware Solution for Accelerating Tomographic Reconstruction

Jimmy Xu

Chair of the Supervisory Committee:
Professor Scott A. Hauck
Electrical Engineering

Field-programmable gate array (FPGA) based hardware coprocessors offer the potential to enable dramatic acceleration in performance for certain applications. In this thesis, we evaluate the performance and design process for a coprocessor application by implementing a Computed Tomography (CT) image reconstruction algorithm on an FPGA coprocessor. Specifically, we document the design process and performance of two separate methods to parallelize the CT reconstruction algorithm for acceleration with the XtremeData XD1000 FPGA coprocessor. Along the way, we make comparisons between the traditional VHDL based FPGA programming model and a C-to-FPGA toolflow called Impulse C in terms of performance and design effort. We show that a VHDL implementation is 1.69x faster than an Impulse C implementation, at the cost of significantly increased design effort with VHDL. In addition, the results of this thesis demonstrate that the FPGA coprocessor is capable of achieving a 103x improvement over multi-threaded software (8 threads) using parallel ray-by-ray reconstruction.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Impulse Accelerated Technologies (Kirkland, WA) and the Washington Technology Center (Seattle, WA) for giving me the opportunity to perform this research by providing the funding that makes it possible. I would also like to express my deep thanks to Professor Scott Hauck and Professor Adam Alessio for patiently providing the guidance and assistance I needed; and Nikhil Subramanian for being a great research partner and friend. Finally, I would like to thank the family and friends that have supported me through the difficulties I faced; without them, this thesis would not have been possible.

1. Introduction

With sequential computing failing to keep up with the continued scaling of transistor density, interest in spatial processors, like Field-Programmable Gate Arrays (FPGA), is being revitalized. FPGAs offer large numbers of simple programmable logic that, when combined, is capable of dramatically accelerating the execution of certain programs. This acceleration is achieved by exploiting the parallelism inherent in certain applications and distributing the computation across the various computation units. While this method of parallel hardware acceleration can offer great benefits, it is often difficult to translate applications written in sequential software code into parallel hardware.

While traditional methods of designing FPGA applications have relied on schematics or HDL, the FPGA community has shown much interest in C-to-FPGA tool flows that allow users to design FPGA hardware in C. C-to-FPGA tool flows allow for wider adoption of FPGAs by reducing the hardware expertise required on the part of the designer, at the cost of abstracting away some degree of fine control. It is our goal to compare the results of a hand-coded VHDL application against an Impulse C version [1] in terms of execution speed, ease of implementation, and resource usage.

A great candidate for this purpose is Computed Tomography (CT). Computed Tomography is a medical imaging technique which uses computer processing to create cross-sectional images from two-dimensional X-ray images taken around a single axis of rotation. This reconstruction technique generates a tremendous amount of data that needs to be processed, which is extremely taxing on traditional sequential processors. In addition, the CT algorithm is well suited for a parallel implementation, as it is mainly composed of independent computations that can be executed in parallel. These traits make CT backprojection a perfect

candidate for a study of the design and implementation of a parallel hardware accelerator.

Previous works on using hardware accelerators [3,4,8] to accelerate the backprojection process have demonstrated that parallel execution on appropriate hardware can achieve performances that are orders of magnitude higher than comparable sequential micro-processor based systems, although analysis on the design process of such algorithm is rarely made. The process of tailoring the computation to maximize the usage of available resources is complex, yet vital, knowledge for a good hardware designer. To that end we aim to design and implement a CT back projector on a FPGA, using hand-coded VHDL, in order to explore the hardware design process for an extremely compute intensive application.

This thesis offers an exploration of different algorithms and approaches for designing a FPGA-based CT Back-projection accelerator. In addition, comparisons versus a multi-threaded C program and an Impulse C FPGA implementation will be made. The following sections of this thesis are organized as follows:

- **Chapter 2: Background** provides background information on FPGAs, CT scanners, backprojection, and coprocessors.
- **Chapter 3: Benchmarks** presents the results of various initial benchmarks prior to starting our backprojector design.
- **Chapter 4: XD1000 Reference Design** details the reference design supplied by XtremeData, which serves as a basis for our hardware design.
- **Chapter 5: Design** provides insight on the parallel algorithm design.
- **Chapter 6: System Design** provides detailed information on the actual hand-coded hardware design.
- **Chapter 7: Results** presents the results of our designs.

2. Background

2.1 Tomographic Reconstruction

Tomographic reconstruction obtains cross-sectional images of an object from a set of measurements taken at various angles. The measurement data records integrals of information along rays traced through the object of interest and is commonly organized as a sinogram, which is the Radon transform [28] of the object . In traditional tomographic systems, the primary computational demand is the backprojection of the sinogram to reconstruct the scanned object. In this case, backprojection can be viewed as the mapping of data from the sinogram space to the image space.

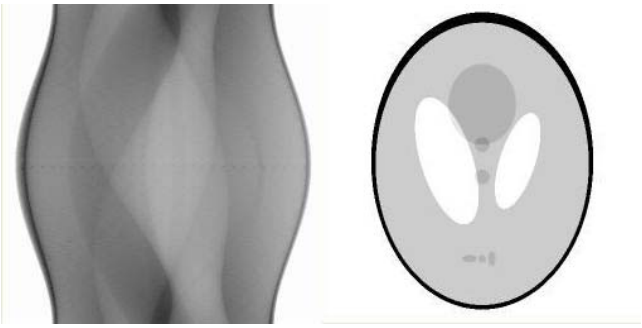

Figure 1 Sinogram space (left), corresponding image space (right)

Tomographic reconstruction is used in many fields and applications, ranging from medical to military. In the field of medical imaging, Computed Tomography, which uses the principles of tomographic reconstruction, is of particular interest.

2.1.1. Computed Tomography

Computed Tomography (CT) is a medical imaging technique which uses tomography created by x-rays, and processed on a computer, to take cross-sectional images of patients. The patients normally lie on a platform within the gantry of the CT and are positioned in the path of the x-rays. Since different parts

of the human body have different attenuation coefficients for the incident x-rays, the viewer is able to differentiate between bones and different soft tissues in the reconstructed image. An example of an image produced by a CT scanner is shown in Figure 2.

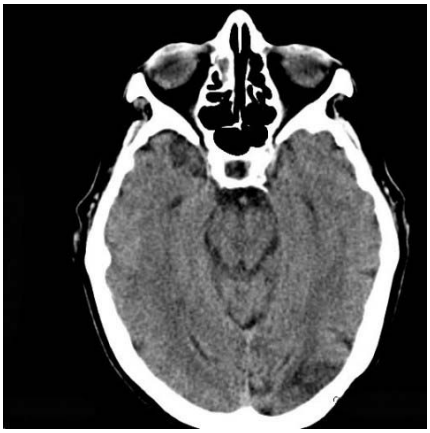

Figure 2 CT Image [27]

First generation CT scanners were introduced with a single x-ray source and detector pair. As detector and source technology advanced, improved generations of CT scanners were developed with difference configurations of sources and detector arrays. Fourth generation CT scanners are equipped with a fixed detector array that completely surrounds the patient. At present, all commercial CT scanners are third generation systems with a single source and detector array mounted on opposing sides of a rotating gantry. With the introduction of newer technology with finer detector sampling, the algorithm associated with reconstructing the cross-section increases in complexity. However, a simplification can made which eliminates much of the trigonometric calculations associated with a single point source and multiple detectors. Basically, the fan-beam projections can be interpolated into parellel-beam projections. This parallel-beam backprojection calculation is less computationally intensive than the single source fan-beam, but still serves as an adequate model for the backprojection process.

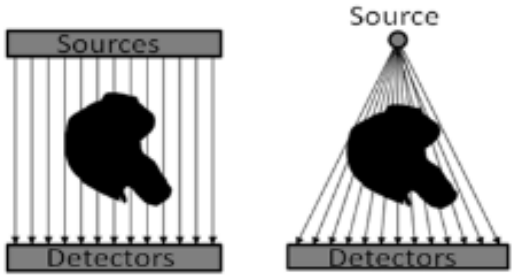

Figure 3 Paralle-beam (left), Fan-beam (right) [1]

2.2 Backprojection

For parallel-beam back projection, the sinogram represents line integrals of the attenuation coefficients along rays perpendicular to the sensor bar at many angles. During the reconstruction process, the collected ray values are re-distributed across the image at the angle which they were collected. As seen in Figure 4**,** the value at sensor S of the sensor bar is distributed along the ray perpendicular to the sensor at angle θ. When this is performed for all entries in the sinogram, enough data is distributed across the image pixels to recreate the image.

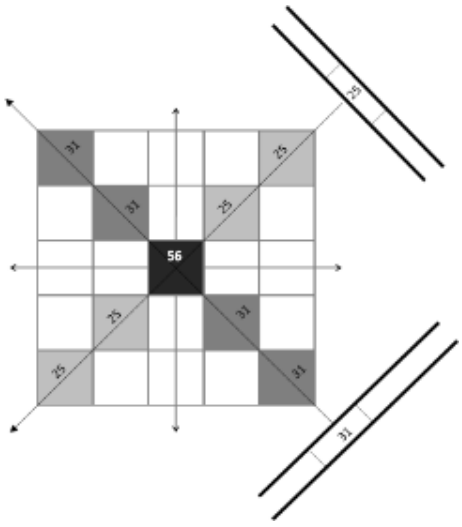

Figure 4 Backprojecting along rays

As shown in Figure 4**,** the pixel at which the two rays intersect takes on a value that is the sum of the two sensor values. As the backprojection algorithm iterates through all sinogram angles, the summation of sensor values along the perpendicular rays is what reconstructs the image.

Iterating through the sinogram to distribute its data across the image is a method of backprojection known as ray-by-ray backprojection. Another approach is to iterate through the image one pixel at a time and examining all S and θ values of the sinogram that contribute to it; this is pixel-by-pixel backprojection. These two methods produce identical results, and only differ in their execution order.

The process of mapping a pixel data point to a sinogram data point can be described by the equation:

$$x\cos\theta + \mathrm{y}\sin\theta = \mathrm{S}$$

The result of this calculation describes the specific sensor S along the sensor array on which the traced ray at angle θ will land. This also indicates the pixels (x,y) that the particular sensor S, at the particular angle θ, will affect. The value S produced by this equation does not always map directly to one sensor, as it is not limited to only integers. In cases where a fractional component of S is present, the fractional component is used to perform interpolation between the two closest sensors. Finally, the sinogram values at the appropriate sensor S for all angle θ are summed to create the final reconstructed image pixel value.

Thus, computations that must be performed for backprojection include: trigonometry to find the sensor index, interpolation, and accumulation with the pixel data from other sinogram projections. These three computations steps alone may not seem overly complex, but it is the sheer number that must be performed to reconstruct an image that contributes to the computational intensity of CT backprojection.

2.2.1 Filtered and Iterative Backprojection

Simple backprojection of the sinogram space into the image space will generate images with a blurring effect around the edges of objects. While additional projection angles and denser sensor arrays will provide more data points for reconstruction, which will give a closer approximation to the original object, the blurring effect will persist unless specifically addressed.

To eliminate the low frequency noise and reduce the blurring effect, one technique often used is the application of a high-pass filter to the sinogram data prior to backprojection. This technique is known as Filtered Backprojection (FBP).. The reconstruction process is essentially solving the inverse Radon transform of the sinogram data [32]; the true solution for this inverse requires the filtering step with a Ramp filter. This Ramp filter, in Fourier transform domain, accentuates high frequency components. In practice, some variant of the Ramp filter is applied, such as a Ramp apodized with a Hanning or Butterworth window, in order to offer a compromise between the true solution and noise reduction.

Another promising approach for eliminating noise in a reconstructed image is iterative reconstruction. This technique involves repeated forward and backprojections between the sinogram space and the image space, with each iteration producing an image that is closer to the scanned object. This method can produce images with improved signal to noise results compared to Filtered Backprojection, but can be orders of magnitude more computationally demanding than FBP.

2.3 FPGA

Field programmable gate arrays (FPGAs) offer a flexible approach to applications traditionally dominated by application-specific integrated circuits (ASICs) and

computer software executed on sequential processors. ASICs provide highly optimized resources specifically tuned for a particular application, but it is permanently configured to only one task and includes an extremely high non-recurring engineering cost, which can run into the millions of dollars [30]. FPGAs on the other hand offer programmable logic blocks and interconnects that eliminate the NRE cost associated with ASICs, at a cost of 5 to 25 times the area, delay and performance [30]. While software provides the flexibility to execute a large number of tasks, typically its performance when run on a sequential processor is orders of magnitude slower than ASICs and FPGAs. These two factors combined make FPGAs an attractive middle ground for applications that require performance that can't be achieved by software running on sequential processors, but yet require a degree of flexibility not possible with an ASIC implementation.

Traditionally, FPGAs have been used for applications that process large streams of data, where processes can be executed in parallel and independent of each other. In fact, FPGAs can achieve several orders of magnitude faster processing speeds compared to microprocessors by exploiting the parallelism in applications. Since many signal processing, networking, and other scientific computations have inherent parallelism profiles that are easily exploitable, implementing them using FPGAs is a relatively easy way to accelerate their computation.

FPGAs function similarly to building a breadboard circuit from standard parts. It consists of programmable logic blocks and interconnects that can be configured to resemble a variety of complex logic circuits. Combinational logic is implemented using look up tables (LUTs), and sequential logic is implemented using registers. Additional special elements, such as large memories and multipliers, are embedded within the FPGA fabric to boost capacity and speed.

For the purposes of this research, Altera's Stratix-II EP2S180 FPGA is used. This device is the largest member of the Stratix-II FPGA family and is included in the XD1000 coprocessor module.

2.3.1 LUT

The look up table (LUT) is the computational heart of the FPGA. It is formed with a combination of a N:1 multiplexer and an N-bit memory [30], and implements a truth table that is capable of expressing Boolean equations. As Boolean equations are capable of representing any computation, the LUT in the FPGA have become the basic building block for most commercial FPGAs [30].

2.3.2 Memory

The Stratix-II FPGA includes three different types of embedded memories, known as Block RAMs. The largest is the MRAM, which has a capacity of 576 Kbits. Correspondingly, MRAMs are the slowest of the available Block RAMs, and the Stratix-II only has 9 of these available. M4K RAM blocks are the next largest in terms of capacity. They can each hold 4 Kbits, and there are 768 blocks available on the Stratix-II. Finally, the smallest, but most widely distributed, Block RAM is the M512 RAM. Each M512 RAM can hold 512 bits of data, and there are 930 available in the Stratix II FPGA.

Table 1 Stratix II Memory Capacities

Memory	Capacity	Available Units	Functionality
MRAM	576 Kbits	9	Single Port Simple Dual Port True Dual Port
M4K	4 Kbits	768	Single Port Simple Dual Port True Dual Port
M512	512 bits	930	Single Port Simple Dual Port

With the exception of the M512 Block RAM, memories on the FPGA can be configured to be Single Port, Simple Dual Port, or True Dual Port. Single Port memories allow either one write or one read per cycle, but not both. Simple Dual Port memories allow both a read and a write operation to be performed in the same cycle, and True Dual Port memories allow two writes or two reads per cycle.

2.3.3 DSP

Dedicated arithmetic blocks are also included in most modern FPGAs. These custom logic are designed to more efficiently execute DSP arithmetic compared to LUT logic. The Stratix-II FPGA has a total of 768 DSP blocks, each of which can be configured to be one 36x36 multiplier, four 18x18 multipliers, or eight 9x9 multipliers.

2.4 Coprocessors

In sequential processing systems, frequently occurring or complex computations can overwhelm the primary processor. A solution to this problem is utilizing coprocessors to shoulder some of the burden, allowing the CPU to offload these computations. A common example of this system is the Graphics Processing Unit (GPU). GPUs are coprocessors designed to handle the complex calculations used in 3-D graphics rendering. They are especially adept at manipulating computer graphics and accelerating the memory-intensive work of texture mapping and polygon rendering [31].

Another promising type of hardware coprocessor uses FPGAs as the primary processing units for computations. These coprocessors offer increased flexibility compared to GPU processors, as they are not limited to the rigid vector processing structure of GPUs. However, limited market penetration as compared to GPUs has limited the development of FPGA coprocessors, and this technology has been relegated to be a very niche market [1].

Several key issues prevent wider adoption of FPGA coprocessors. Traditionally, a primary concern in accelerating applications using FPGA coprocessors is the cost of transferring data between the CPU and the FPGA board. Since traditional FPGA coprocessors often utilized the PCI or other lower bandwidth busses to communicate with the CPU, the cost of transferring data to the accelerator may overcome the benefit of offloading the computation.

Alternatively, FPGA co-processors such as the XtremeData XD1000 and the Accelium platform [26] from DRC Computer offer tightly-coupled communication interface between the CPU and co-processor. Compared to coprocessor solutions from companies like Pico Computing, which utilizes the Express Card interface with a bandwidth of 250 MB/s [25], it is apparent that the approach taken by these tightly-coupled FPGA coprocessors may greatly reduce the data transfer cost of communicating between the CPU and FPGA.

2.4.1 XD1000

Unlike many other FPGA coprocessor boards, the XtremeData XD1000 uses a very tightly-coupled FPGA to CPU interface, with the FPGA coprocessor module situated in one of the CPU sockets in a dual processor AMD Opteron motherboard. This approach allows the FPGA module to communicate with the CPU via the HyperTransport bus, which has a bandwidth of up to 1.6 GB/s. Comparing this bandwidth to a FPGA board such as the Pico E-16 from Pico Computing, which uses an Express Card interface with a bandwidth of 250 MB/s [25], it is apparent that the approach taken by the XD1000 may make the CPU-FPGA communication link much less of a bottleneck than traditionally feared.

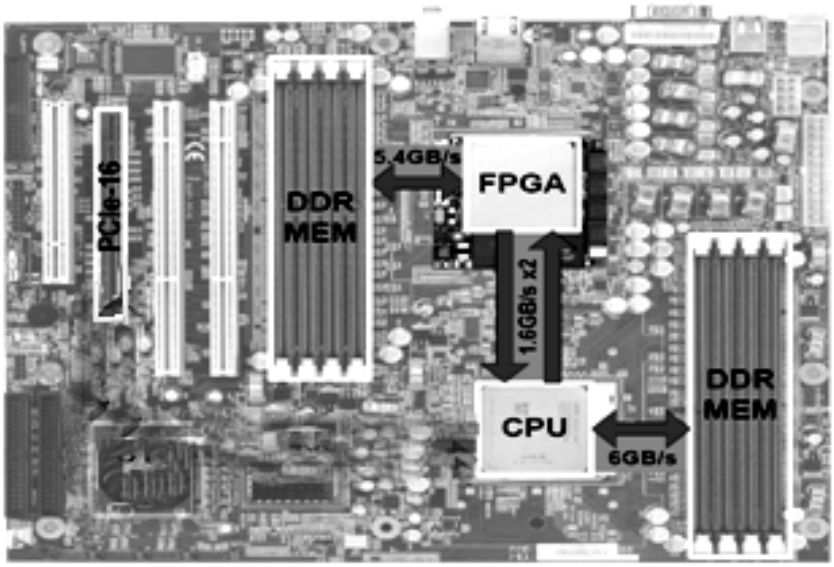

Figure 5 XD1000 Development System [24]

The particular XD1000 system used in this study includes 4 MB of SRAM onboard the XD1000 module, and 4 GB of DDR RAM available each to the Opteron CPU and the Stratix II FPGA. Details of available resources and maximum transfer rates as provided by XtremeData can be found in Figure 5.

3. Benchmarks

3.1 XD1000 Hardware Benchmarks

Prior to starting development on the backprojector application, we modified and ran the included XD1000 reference design in order to benchmark the available inter-chip bandwidth. The data is collected by running and timing each of the individual resource tests within the reference design. Details of the resource tests can be found in [23]. The result of this benchmark test is broken down by resource and shown in Table 2**.**

Table 2 XD1000 Initial Hardware Benchmark Results

Resource	Min Speed	Max Speed
CPU -> FPGA	31 MB/s	534 MB/s
FPGA -> DRAM	50 MB/s	4956 MB/s
CPU -> SRAM	19 MB/s	19 MB/s
SRAM -> CPU	3.6 MB/s	3.6 MB/s
FPGA -> SRAM	800 MB/s	800 MB/s

The results are separated by minimum transfer speed and maximum transfer speed. This distinction is the result of the HyperTransport bus, and the associated data transfer overheads, favoring block transfers with large amounts of data. The minimum speed is achieved by transferring the minimum allowable data size, while the maximum speed is achieved by transferring the maximum data size allowed.

3.2 Single-Threaded Software Backprojector Benchmark

The backprojection software source code [21], provided by the Radiology Department of the University of Washington, is a parallel beam CT backprojector that supports a variety of common projection filters. It is single-threaded and uses 32-bit floating-point operations to perform most of the operations involved

in the backprojection process. While the original code is not designed with performance in mind, the software can still be used to provide a baseline for performance comparisons.

We began our analysis of this backprojection algorithm by profiling the single-threaded software benchmark in order to determine which portion of the code would benefit the most from hardware acceleration. As seen in Table 3**,** the backprojection stage takes significantly more time to execute compared to the filter and file IO stages. This suggests that the backprojection loop is the best candidate for FPGA hardware acceleration, which relegates the CPU of the XD1000 to the file I/O and filtering stages.

Table 3 Software source code execution time profile

Execution Stage	512 Projections	1024 Projections
Filter	0.89 s	3.56 s
Backprojection	3.85 s	10.03 s
File IO	0.03 s	0.03 s
Total	4.77 s	13.62 s

Another reason for selecting the backprojection stage for hardware acceleration is the benefits it would provide to iterative backprojection. As outlined in Chapter 2, iterative backprojection relies on repeated forward-projection and backprojection iterations that do not involve either the filtering or the file I/O stage. Since the forward-projection process is very similar to backprojection, by including the backprojection, and forward-projection, loop in the FPGA, we can keep the iterative backprojection computation entirely within the FPGA and bypass the communication overhead to the CPU.

3.3 Multi-threaded Software Backprojector Benchmark

As multicore processors overtake their singlecore counterparts, there is an opportunity to achieve better performance through parallel processing. For the backprojector application, we were interested in the performance of the single-threaded benchmark when run with multiple parallel threads on a multicore processor. This would provide yet another point of comparison for our hardware implementation and would be much more indicative of real world performance of the software backprojector application.

To implement the multi-threaded version of the software benchmark, we decided to use the OpenMP API. OpenMP supports shared memory multiprocessing programming in C and C++ on both UNIX and Windows platforms, which fit our development requirements. OpenMP provides an easy way for developers to achieve multithreading and parallel execution. It does so by automating the process of forking the master/serial thread into a specified number of slave/parallel threads, effectively dividing the task among several processing units.

OpenMP requires the section of code that is meant to run in parallel to be marked accordingly in the code. This is done with a preprocessor pragma that readies the slave threads before the marked code is run. With the OpenMP library included in the compilation, the pragma simply declares the data within the marked code that will be shared between threads, and the data that will be unique for each thread. This is done to prevent data read and write hazards from causing erroneous results.

For the backprojector benchmark the code that would benefit the most from parallel processing is the backprojection loop. Analysis on the algorithm of this loop reveals that since the outer loop iterates through the image space by pixel, each thread would be responsible for completing the reconstruction of one pixel.

Therefore, no write-after-write data hazard would be present. However, many variables present in this backprojection loop are incremented through iterations of the outer and inner loops. While this may be computationally more efficient than re-calculating the values of these variables, it does not lend itself well to parallel execution. Since parallel execution of a loop in OpenMP effectively unrolls each iteration into individual threads the order of execution of the loop iterations is no longer guaranteed.

For example, the outer loop in Figure 6 iterates through the X coordinates of an image, but when unrolled into three threads by OpenMP, it is not guaranteed that i=0 will be the first thread to be executed. Therefore, if x was iterated in the outer loop instead of assigned a value, the value passed down to process() would be incorrect.

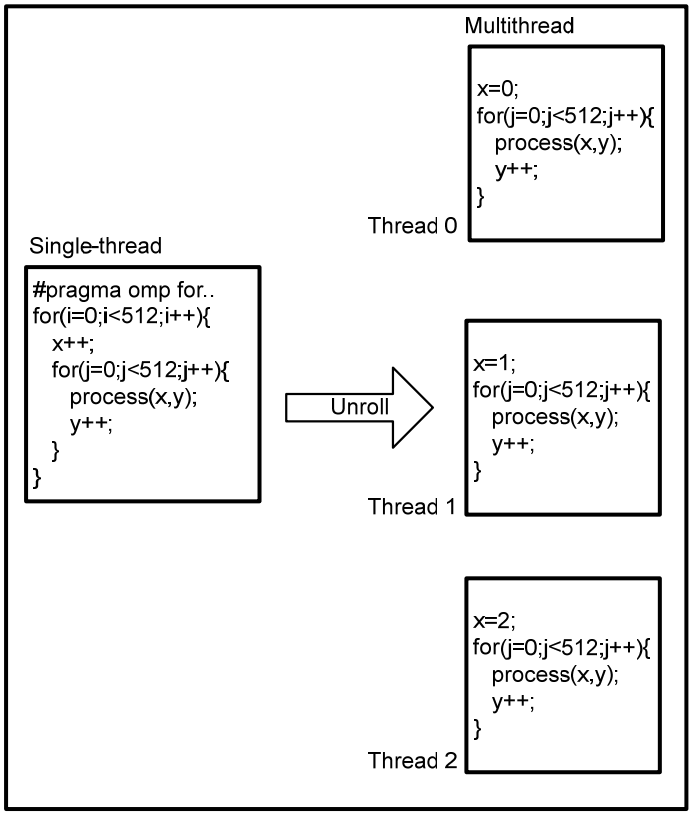

Figure 6 OpenMP Loop Unrolling

While OpenMP has many additional features for multithreading programs, the simple structure of the loop we are parallelizing only required the basic parallel FOR loop structure. After making the necessary changes to avoid data hazards, we ran this multithreaded benchmark on a system with two quad-core Intel Xeon L5320 processors, which allowed us to allocate 8 threads to OpenMP. The results of this benchmark can be found in the Results chapter.

4. XD1000 Reference Design

To aid hardware designers developing for the XD1000, XtremeData includes a VHDL reference design with example code to access and test all of the available resources. This reference design connects all components in the system via Altera's Avalon Communication Fabric, and provides documentation to assist the designer with integrating his own code into the pre-existing hardware wrappers.

4.1 Software

The software environment supplied as part of the XD1000 reference design includes necessary drivers and C++ test code for developing FPGA co-processor applications. The included device driver is implemented as a Linux kernel module, and is loaded by the system at boot if the FPGA is found and configured. The included C++ test application is designed to test the FPGA configuration in the reference design package.

The test application, when run, presents the user with a menu-driven set of tests, each verifying the functionality of one of the FPGA modules. For example, the "HT test (FPGA <-- CPU)" initiates a data transfer test which sends randomly generated data from the CPU to the FPGA over HyperTransport, with the FPGA then verifying and acknowledging back to the CPU that the data is received correctly. For the purposes of the backprojector applications, we determined the most logical development step for creating the software controller was to build upon this test application.

The code in place for the "CPU <--> SRAM" and the "FPGA <--> CPU" tests were both suitable for the backprojector software controller. Both of these tests offered a way to send large amounts of data from the CPU to the FPGA (sinogram) and back (reconstructed image). Since the SRAM is not a shared resource between the FPGA and the CPU, and the datapath from the CPU to the SRAM passes through the FPGA, both of these tests utilize the same HT controller on the

FPGA. This meant the choice between the two starting points relied only on how the sinogram should be stored. For a streaming model, where pieces of the sinogram is transferred to the FPGA (from the CPU) as the processing engines require input data, the "FPGA <--> CPU" code base is sufficient, as the SRAM is not needed for buffering. But as initial hardware benchmarks show, the HT bus has a much higher bandwidth for block transfers; therefore, the SRAM buffer was determined to be necessary, and the "CPU <--> SRAM" was selected for initial implementation.

Many changes were necessary to adopt the "CPU <--> SRAM" test code into the software controller for the backprojector application. First, initial hardware benchmarks indicated that the CPU to SRAM datapath is extremely inefficient, especially compared to the CPU to FPGA data path which achieved a max transfer rate of over 500 MB/s. This was a puzzling result, as the bottleneck of this transfer (the HyperTransport) is shared between both tests. Additional analysis revealed that the "CPU <--> SRAM" test in the reference design sent single word data packets to the FPGA and the SRAM, and required a message signaled interrupt (MSI) acknowledgement with each transfer. Compared to the block transfer style of the "FPGA <--> CPU" test, this was extremely inefficient. A change was needed to introduce bulk transfers directly to the SRAM.

On the software side, bulk transfer across the HT required the software controller to use the xd_mem_lock function on the entire sinogram data prior to transfer. This prepares the Linux drivers to expect the correct amount of data to be transferred. An MSI which initiates the transfer is then generated from the CPU to the FPGA, telling the FPGA to begin receiving data. The software then begins iterating through the memory addresses locked by xd_mem_lock, and sends the data across the HT in word sizes of 64 bits. When finished, another MSI is sent which initiates the backprojection process.

Bulk transfer from the SRAM/FPGA back to the CPU uses a slightly different process. An MSI is sent from the FPGA to the CPU to initiate transfers, and the Linux drivers will begin storing the incoming data into the DDR RAM accessible by the CPU. When transfer is complete, a pointer to the head of the block of memory containing the received data is returned.

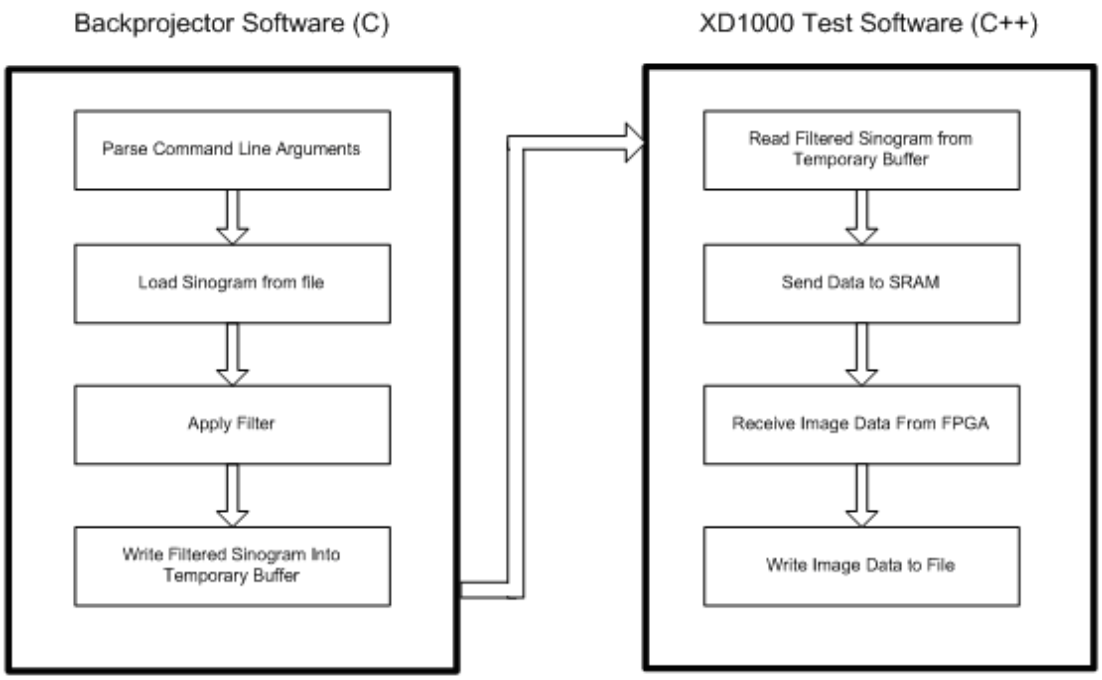

Figure 7 Software Flow

The second major change to the XD1000 software test application was to integrate it into the existing backprojector software. This framework includes the initial sinogram processing and filtering stages of the backprojector. Since the XD1000 test software is written in C++ and calls on specific XD1000 drivers, it is not easily converted into C. Likewise, the backprojector software framework is written in C, and has library calls which made it difficult to port to C++. The backprojector software controller requires elements from each, so a temporary solution was to run each separately, and pass the filtered sinogram from the

backprojector software to the test software through a temporary file buffer. This method is far from ideal, but serves as an adequate proof-of-concept.

4.2 Hardware

The hardware side of the XD1000 reference design contains modules, written in VHDL, that access and test various hardware components connected to the XD1000. These modules create an entry point for the application designer by providing example code for the hardware controllers. The modules are memory-mapped and connected together by the Altera Avalon bus. For the backprojector application the main hardware modules of interest are the HyperTransport send/receive controller and the ZBT SRAM controller. An outline of functionalities of each module, as supplied in the reference design, is given in the following sections. For details on changes made to each in the backprojector design, please see the System Design chapter.

4.2.1 HyperTransport Controller

The HyperTransport Controller consists of HT_DATA_CHECK and HT_DATA_GEN controllers. Both are memory-mapped on the Avalon bus as slaves, with HT_DATA_CHECK responsible for processing data received over HT and HT_DATA_GEN responsible for processing data to be sent over HT.

In the original reference design both HT_DATA_CHECK and HT_DATA_GEN contain random number generators that are complements to the software on the other side of the HT bus. This allows the HT_DATA_CHECK to check whether the data received is correct, and the HT_DATA_GEN to generate data that can be verified by the receiving software.

4.2.2 ZBT SRAM

ZBT (Zero Bus Turnaround) is a feature where there is a 0 cycle delay to change access to the SRAM from READ to WRITE. Like the HT controllers, the ZBT SRAM

controller supplied by XtremeData is memory-mapped on the Avalon bus as a slave. The controller includes the necessary Avalon controllers, as well as the control signals for physical layer access to the SRAM. Memory-mapped registers for write/read data, as well as control signals, effectively serve as a bridge for communication between the Avalon bus master and the SRAM.

5. Design

5.1 Algorithms

Since both pixel-by-pixel and ray-by-ray backprojection methods require the compute loop to touch every angle of the sinogram for every point on the image (an $O(n^3)$ operation), the main difference between these two methods is the outer loop of the processing algorithm. As seen in Table 4**,** the pixel-by-pixel approach places the x-y loops outside of the angle loop, with the opposite being true for the ray-by-ray method. Both of these methods can be made parallel by introducing processing blocks into the image and sinogram respectively. For the pixel-by-pixel method, each column of the image can be assigned to a separate processing engine. An example is a system which divides the image into four 128x512 blocks, which will require 128 processing engines. The ray-by-ray method can be made parallel by partitioning the sinogram into blocks of 128x1024 (θxS), with each angle of the sinogram block assigned to a separate processing engine.

Table 4 Backprojection Algorithm Loops

	Pixel –by-pixel	**Ray-by-ray**

Serial	for(x=0 to 511 by 1)
	for(θ=0 to 1023 by 1)
	for(y=0 to 511 by 1)
	compute(x,y,θ)
		for(θ=0 to 1023 by 1)
		for(x=0 to 511 by 1)
		for(y=0 to 511 by 1)
		compute(x,y,θ)

128 way Parallel	for(bx = 0 to 511 by 128)
	for(θ=0 to1023 by 1)
	for(dy=0 to 511 by 1)
	compute(bx..bx+127,y,θ)
		for(bθ=0 to1023 by 128)
		for(x = 0 to 511 by 1)
		for(y=0 to 511 by 1)
		compute(x,y,bθ..bθ+127)

The total run-time for the backprojector is composed of the number of cycles to write the sinogram to the SRAM, the cost of loading the sinogram from the SRAM, the number of compute cycles, and the number of cycles to read the image back

out to the host application. This can then be divided by the operating frequency of the FPGA to find the total run-time required for the particular algorithm.

Since the image we're reconstructing will be 512 by 512 pixels, the time it takes to read the image back to the host application will be constant, at 262144 cycles or 2.6 ms with a 100 MHz FPGA. For the pixel-by-pixel method of image reconstruction this can be masked within the compute time, as the loop structure fully reconstructs a block of the image before moving on to the next block. This allows the FPGA to send off the processed image block back to the CPU while the next block is still being computed.

The number of cycles to write the sinogram to the SRAM from the CPU is also the same between ray-by-ray and pixel-by-pixel, and only dependent on the size of the sinogram we're using to reconstruct the image. This is a straightforward data transfer that can't be optimized beyond hardware limitations.

The inner loop of both pixel-by-pixel and ray-by-ray contain the same set of computations, and do not differ significantly in their performance. Instead, the different outer loops have significant effects on the sinogram load and computation stages of the reconstruction process. This is where we can optimize the algorithms for parallel execution, and where we spent significant design effort. Detailed analysis of each of the algorithms is presented in the following sections.

5.1.1 Pixel-by-pixel

In the pixel-by-pixel method of image backprojection each processing engine is tasked with reconstructing one pixel of the image from all relevant sinogram data. These processing engines are independent of each other, but may share the same sinogram data for their computations, depending on the given angle. Since the bandwidth of the path to load the sinogram data (SRAM to FPGA) is fixed and limited by the physical constraints present, we should maximize the reuse of

sinogram data between the processing engines, which will minimize the number of accesses to the SRAM during the computation.

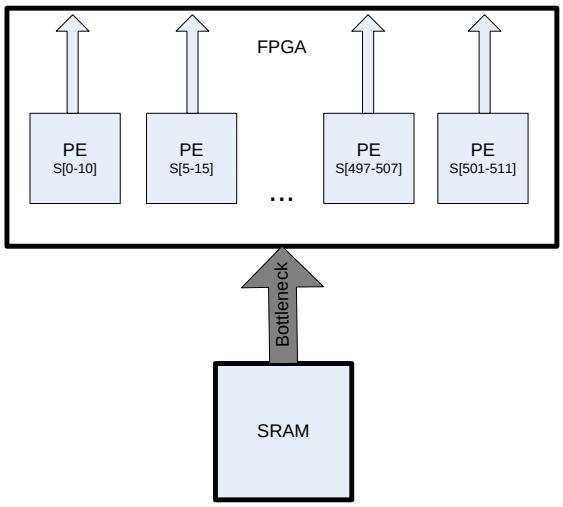

Figure 8 Backprojection Bottleneck

Reuse in the pixel-by-pixel method comes from multiple pixels requiring the same sinogram data for a given angle. As seen in Figure 9**,** multiple pixels along the axis perpendicular to the sensor bar share the same sinogram data, which suggests that only one load is required per row/column. This relationship is of course based on the angle of the sinogram and is observed in different extent across all angles.

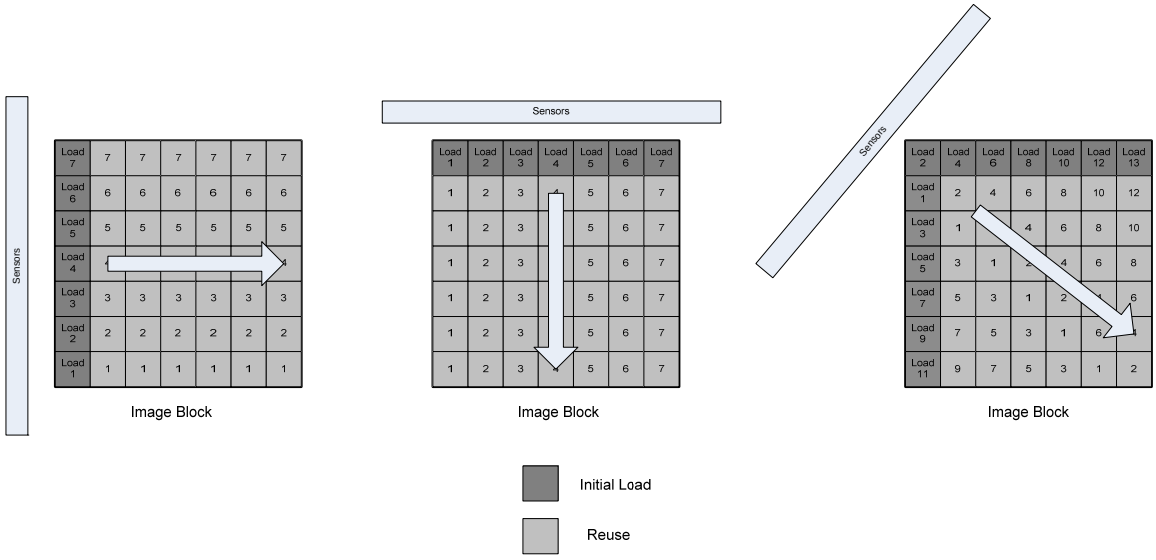

Figure 9 Sinogram Data Reuse

The following sections present the various methods of dividing up the image for parallel processing, along with their best and worst projected run-times. These numbers reflect the amount of sinogram data reuse each method is capable of achieving. The calculations below are performed using a 1024 by 1024 sinogram as the input for the backprojector. Note that there are two separate values for each method of grouping the pixels, presented as lowest and highest. These two numbers reflect the shape of the pixel block, with the lowest column representing the SRAM load calculation based on just the width of the block, and the highest column representing the amount of loads based on the diagonal of the block. Since the sinogram data represents a rotating bar of sensors that circle around the block, the actual number of loads for each angle will fall somewhere between the lowest and highest calculations. These calculations are presented simply as a pre-implementation evaluation of the different pixel blocking methods.

5.1.1.1 Per pixel (1x1)

The first set of calculations presented in Table 5 represents the serial computation of processing one pixel at a time, thereby exploiting no parallelism.

Since the width and diagonal of this method of blocking the image is the same (1x1), the highest and lowest numbers of SRAM loads are also the same. The equation used to calculate the lowest and highest loads uses the dimension of the image x * y, and multiplies that by the ratio of the number of sensors to the number of pixels to find the number of sinogram data points that are needed for a given angle of the sinogram. Multiplying this value with the number of angles will then produce the number of sinogram data points needed to completely reconstruct the image.

$x \times y \times (sensors / pixels) \times \theta$ = number of SRAM loads

Table 5 Number of SRAM Loads for 1x1

Lowest	Highest
$512 \times 512 \times 2 \times 1024 = 5.3x10^8$	$512 \times 512 \times 2 \times 1024 = 5.3x10^8$

5.1.1.2 Per block (128x128)

Blocking the image into a series of squares is perhaps the most natural division for parallel processing. Since the block of pixels is a square, the difference between the highest and lowest columns of Table 6 is simply the relationship between the diagonal and the side of the square. The equation used to calculate the number of loads is composed of the size of the block, in pixels, multiplied by the ratio of sensors to pixels, multiplied by the number of blocks and the number of angles. Notice the overlap from grouping pixels together in a square represents a significant savings as compared to performing the backprojection calculation for one pixel at a time.

$blockx \times (sensors / pixels) \times \theta \times numberofblocks$ = number of SRAM loads

Table 6 Number of SRAM Loads for 128x128

Lowest	Highest
$128 \times 2 \times 1024 \times 16 = 4x10^6$	$\sqrt{2} \times 128 \times 2 \times 1024 \times 16 = 5.9x10^6$

5.1.1.3 Per block strip (128 x 512)

We made the observation that reuse in the pixel-by-pixel method is derived from pixels that are aligned on an axis perpendicular to the sensor bar at a given angle using the same sinogram data. This would suggest that the size of the image pixel block will directly affect the amount of reuse we can achieve. However, we are limited by hardware constraints in the number of processing engines we can realistically implement. With the pixel-by-pixel algorithm, the number of processing engines essentially dictates the width of the block; this means that the height of the block is limited by the size of the sinoRAM local to each processing engine in order to accommodate for the angle which requires the most sinogram data points. This angle is usually dictated by the length of the diagonal of the block, as is the case of the 128x128 blocking approach presented. This means that if we are DSP-constrained, and have limited processing engines, a rectangular blocking scheme is essentially free in terms of hardware usage, but can provide additional sinogram reuse and reduce SRAM loads.

The highest number of SRAM loads is derived from the case where all sensor values from a given angle is loaded to the local sinoRAM, which corresponds to the angle at which the sensor bar is parallel to the length of the block (512). The lowest number of SRAM loads corresponds to the angle at which the sensor is parallel to the width of the block (128).

Table 7 Number of SRAM Loads for 128x512

Lowest	Highest
$128 \times 2 \times 1024 \times 4 = 1x10^6$	$512 \times 2 \times 1024 \times 4 = 4x10^6$

5.1.2 Ray-by-ray

The ray-by-ray method of image backprojection can be visualized as stepping through each point in the sinogram and calculating all points on the image this sinogram point will contribute to. This is the opposite of the pixel-by-pixel method, and focuses on the sinogram as the outer loop of the reconstruction algorithm. Because of this, ray-by-ray has a constant number of SRAM accesses, as each data point in the sinogram is only iterated through once. With a sinogram size of 1024x1024, this value is naturally 1048576, which is the same as the best lower bound in pixel-by-pixel.

Leeser et al. [4] performed detailed analysis on the best method to implement a ray-by-ray parallel algorithm, and concluded that assigning each processing engine to a sinogram angle and processing in blocks of angles is the most efficient way to exploit parallelism in the reconstruction. This effectively divides the outer loop (angle) into blocks, and carves the sinogram into rectangles each spanning the entire S dimension and covering a portion of the θ dimension.

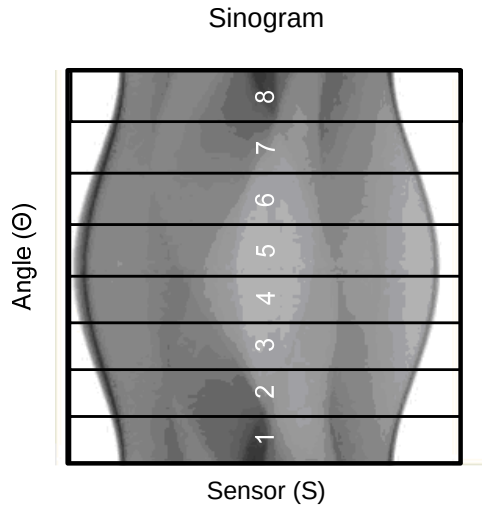

Figure 10 Ray-by-ray Sinogram Division

While loading the sinogram is not a focus of our optimization efforts, a potential bottleneck for ray-by-ray still exists in accumulation of the image data. Assigning one angle to a processing engine will create 512 or 1024 partially reconstructed images that will need to be summed before the image is fully reconstructed. In order to avoid multi-cycle accumulations when many processing engines are working parallel, an adder tree must be used to sum the results of all processing engines. This unifies the individual processing engines to form a processing block, and the partial image output from this is the result of all the angles covered by the processing block.

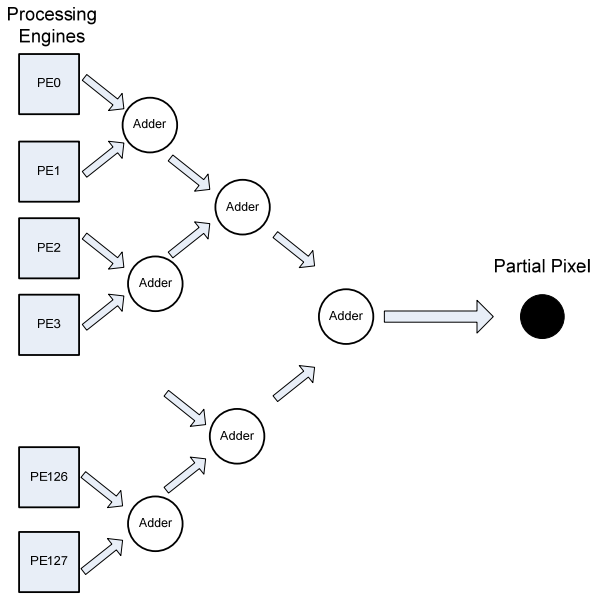

Figure 11 Ray-by-ray Adder Tree

5.2 Hardware Considerations

Since we have limited hardware resources onboard the Stratix-II FPGA, the level of parallelism is constrained by the least available resource. In the case of this backprojector application, each of the processing engines require a certain number of Block RAM for sinogram and image storage, and a certain number of DSP units for multiplication in the interpolation step. While logic could also be a limiting resource, the relatively simple datapath of the backprojector places much more emphasis on the Block RAMs and DSPs.

5.2.1 Block RAM

Both of the reconstruction methods discussed require the same amount of distributed memories to store the sinogram. For the pixel-by-pixel method, all processing engines compute on the same angle at any given time, but since they target different pixels of the image, the sinogram data needed for each processing engine will be different. This requires a unique instantiation of the Block RAM

storing the sinogram (sinoRAM) for each processing engine to provide the two sensor values required for interpolation. For the ray-by-ray method, separate sinoRAMs are required as well, since each processing engine is working on its own angle in the sinogram.

Another important point of consideration for both ray-by-ray and pixel-by-pixel is the bandwidth needed for sinoRAM. Since the interpolation process requires two sinogram points to interpolate between, each processing engine's sinoRAM would need to provide two sinogram values per cycle of processing. This calls for either using two sinoRAMs per processing engine, which would be resource inefficient, or configuring the sinoRAMs to true dual-port, with the ability to output two values per clock cycle.

For the parallel ray-by-ray method of reconstruction, the image memory (imgRAM) does not have to be distributed because only one image pixel is computed every clock cycle, so the central image memory is written to only once a cycle from all processing engines. Since we would like to store as much of the image as possible, so that we do not have to pause the computation to send parts of the reconstructed image off chip, this memory will have to be high capacity. This low bandwidth and high capacity requirement naturally points to the large MRAM blocks on the Stratix-II.

For the parallel pixel-by-pixel method, the amount of memory required to store the image pixels depends on the amount of parallelism used. Since this method computes a block of image pixels in parallel, each processing engine is responsible for one column of the image. Thus a system that is 128 way parallel will require 128 individual processing engines, each needing enough memory for one column of the image. We implemented this with unique memories local to each processing engine, although a global imgRAM implemented using MRAM is possible with clever concatenation of data.

Finally, the last pieces of the system which require Block RAM are the sine and cosine trigonometry memories. In order to avoid complex trig functions, sine and cosine values needed in the computation can be pre-computed and stored in easy to access Block RAM as read-only-memory. For ray-by-ray, unique sine and cosine values will be required for each processing engine, since each is working on a different θ. For pixel-by-pixel, some savings can be had as only the cosine memory is required to be unique for each processing engine, as both θ and y are the same across all processing engines.

5.2.2 DSP

The 96 DSP blocks in the Stratix-II EP2S180 FPGA can be configured as one 36-bit, four 18-bit, or eight 9-bit multipliers. For pixel-by-pixel, the multiplication step used to calculate the sensor value S ($x*\cos\theta + y*\sin\theta = S$) requires two separate 16 bit multipliers per processing engine. These are implemented with 18-bit DSP blocks. For ray-by-ray, a system was developed by Leeser et al. [4] where addition with offsets can be used to replace the multiplication required to find S.

In our algorithm, we use bilinear interpolation since sinogram rays intersect the image lines in between individual pixel locations. This indicates that we also require two separate 16-bit DSP multipliers for the interpolation step.

Should the number of DSP blocks be insufficient for the amount of multipliers needed, multipliers constructed using logic elements can be used as a substitute. Multipliers constructed using logic elements are slower than dedicated DSP multipliers. As a point of comparison, a 16-bit multiplier will consume 2 DSP 9-bit multipliers versus 294 LUTs in the Stratix-II.

5.2.3 Hardware Summary

Prior to hand-optimizations, each processing engine in the backproject datapath requires two 16-bit multipliers, which together will consume 4 DSP 9-bit

multipliers. For a 512x512 ray-by-ray backprojector, the same processing engine will consume 2 M4K Block RAMs to implement sinoRAM, and 2 M4K Block RAMs to implement the local imgRAM. With 128 processing engines, this amounts to 512 consumed M4K RAMs and 512 DSP 9-bit multipliers (64 DSP blocks). Hand-optimizations allow us to factor our two of the DSP 9-bit mulipliers outside of the processing engines, reducing the DSP usage; however, the Block RAM usage cannot be further reduced. This coupled with the fact that we can substitute in LUT multipliers in place of the DSP multipliers should the need arise, shows that the resource constraint for our designs lie in the Block RAMs instead of the DSP multipliers.

6. System Design

Since we began our project focusing on the pixel-by-pixel reconstruction method, the following sections will be focusing on this particular method of parallel reconstruction. However, areas where the two methods differ significantly will be noted.

The system level view of the hardware backprojector system can be divided up into two distinct regions, with one spanning the CPU and the other covered by the FPGA. The numbers in Figure 12 indicate the order of operations, starting with the software application benchmark passing control over to the XD1000 test software.

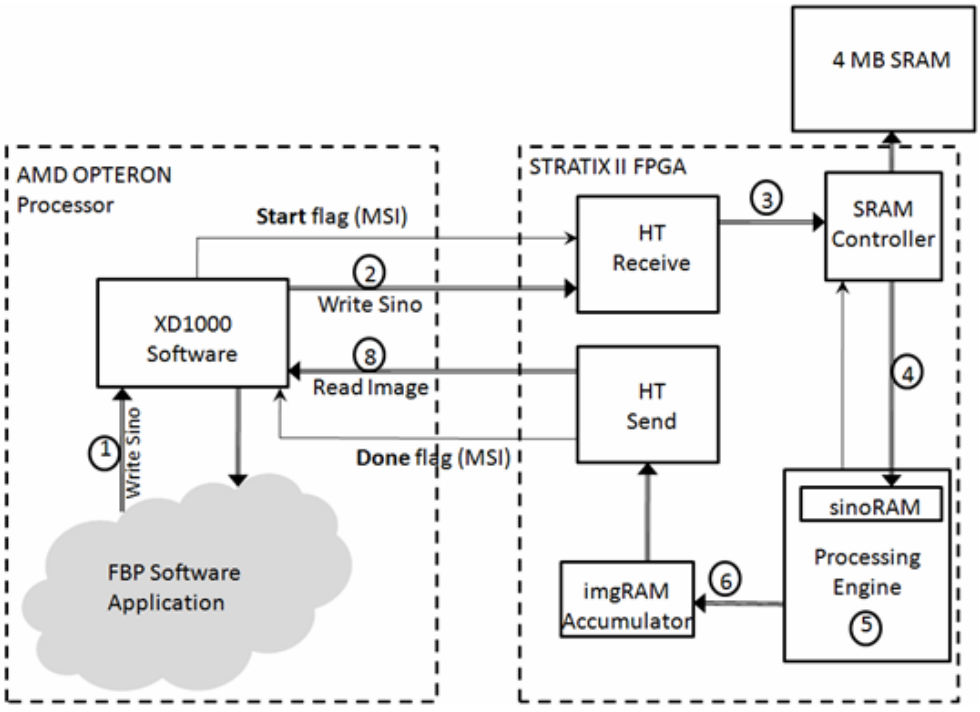

Figure 12 Hardware Block Diagram

The sequence of operations for the pixel-by-pixel system is outlined below:

1) The FBP software application filters the sinogram data, and passes the data to the XD1000 control software.

2) The XD1000 control software writes the data to the FPGA through HyperTransport, and signals the FPGA using a message signaled interrupt when it is finished.

3) The HT Receive controller passes the data to the SRAM controller, which then writes it to the SRAM.

4) After all the data is transferred to the SRAM, the processing engine starts requesting data.

5) The processing engines interpolate the sinogram data accordingly.

6) For the pixel-by-pixel method, the image data is accumulated in local imgRAM.

7) Repeat 4-6 for the entire sinogram and image.

8) After the image is complete, the FPGA sends the data back to the XD1000 control application, which passes it to the FBP software application.

The sequence of operations for the ray-by-ray system follows steps 1-3 of the pixel-by-pixel system. The ray-by-ray system differs in the subsequent processing steps:

4) After all the data is transferred to the SRAM, the processing engine starts requesting data. For the ray-by-ray method, 128 angles of the sinogram are read into their respective processing engine's sinoRAMs.

5) The processing engines interpolate the sinogram data accordingly.

6) For the ray-by-ray method, the image data is accumulated in global imgRAM.

6.2 Hardware System Overview

Figure 13 shows the flow of tasks in the backprojector system. Software tasks are performed by the CPU; while Processing Engine and support tasks are performed by the FPGA. The hardware portion replaces the computation loop that existed in the software backprojector benchmark. The following sections will focus on the hardware portion of the system, which covers the main backprojection computation loop.

The hardware portion is split into two types of tasks: support and processing. Support tasks are distributed across various hardware controllers and state machines. Processing tasks beging with the Load Angle task handled by the SRAM_controller and sino_load_control modules. This task is responsible for loading the stored sinogram into the sinoRAM in each of the processing engines, effectively providing input to the computations. The iterate angle, iterate x-y, and iterate sensor tasks are all performed by the state_control module, which provides the x, y, and θ inputs to each of the processing engines. Send Image Data to CPU is performed by the HyperTransport controller, the details of which are covered in subsequent sections.

The three Processing Engine tasks are implemented in the computation path of each processing engine. These processing engines are then instantiated multiple times to exploit the parallelism of the backprojection algorithm.

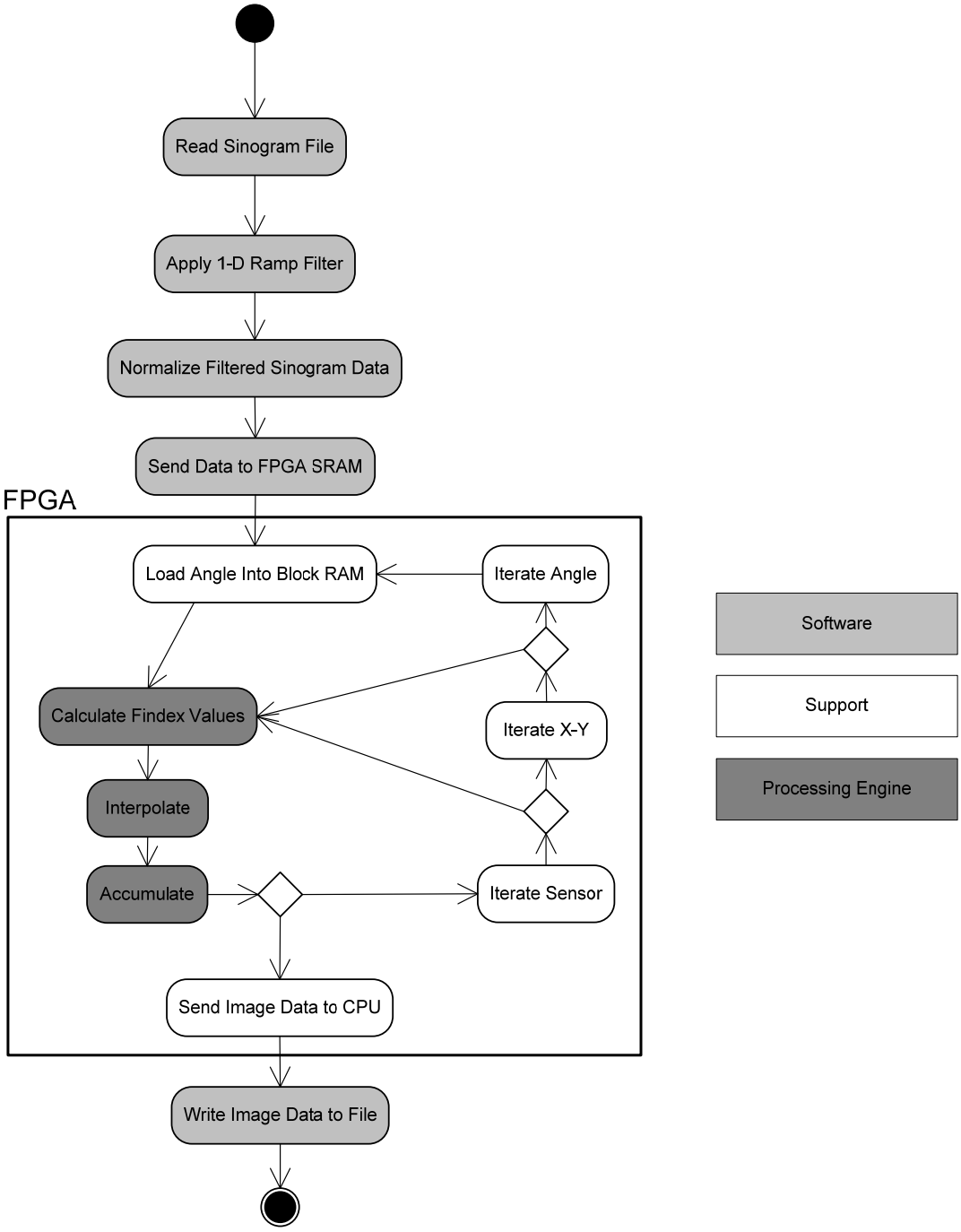

Figure 13 Detailed System Diagram

6.3 State_control

The hardware backprojector's processing flow, for both ray-by-ray and pixel-by-pixel, is broken up into 5 states. For all system states, the control signals for every part of the backprojector is generate in the state_control module.

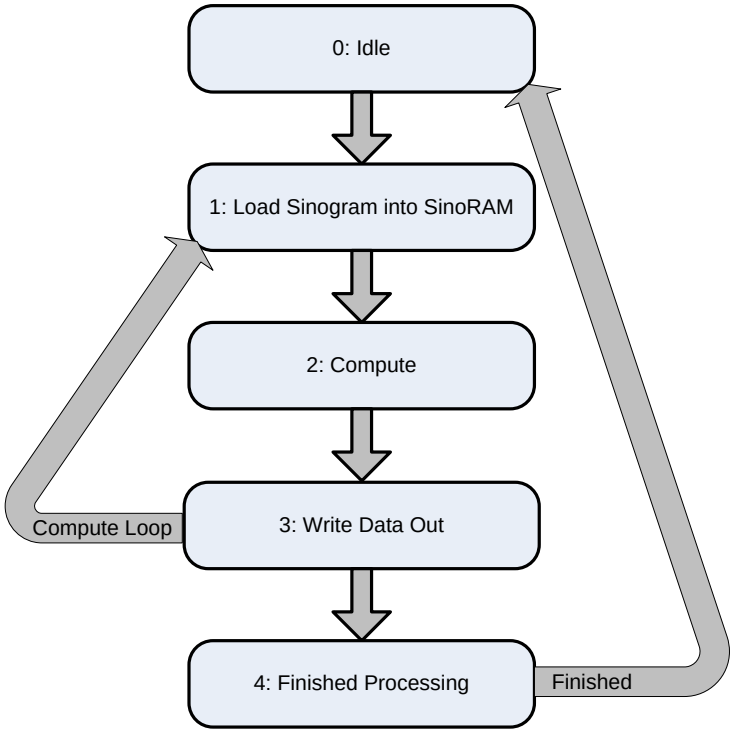

Figure 14 Backprojector System States

The system starts in the idle state and waits for the MSI from the CPU to indicate that the sinogram has been completely transferred to the SRAM. In this state a hardware counter cycles through all imgRAM locations and writes 0's. This mirrors software zeroing the image array prior to accumulation.

When the MSI is received, the system begins the computation loop and loads the first set of sinogram values into the sinoRAM of each processing engine. For pixel-by-pixel, this is all S values in the first angle of the sinogram; for ray-by-ray, this is all S values in 128 angles of the sinogram. State 1 is complete when the

sino_load_control module signals the state_control module by raising the load_ready flag.

The compute stage consists of the actual inner loops of the computation. Hardware counters are in place to cycle through θ and y in pixel-by-pixel, versus x and y in ray-by-ray. These counters are nested, in the same manner as their software loop counterparts. They are incremented every cycle, and their outputs are fanned out to each processing engine to feed the computations. This structure of iterating variables through counters produces a throughput of 1 per processing engine.

After the nested counters in the compute stage finishes counting, an intermediary step of writing the image data is performed. For both ray-by-ray and pixel-by-pixel, this step involves iterating to the next sinogram/image processing block. For pixel-by-pixel, this step also involves sending the completed portion of the image back to the CPU. If the all sinogram/image blocks have been processed, a finished_processing flag is raised and the system is sent into the final state; otherwise the state is set back to the sinogram load state.

The final state for pixel-by-pixel simply sends the finished processing MSI to the CPU to announce completion. For ray-by-ray, this step involves sending the entire processed image back, followed by the finished processing MSI. After this is complete the system is set back into the idle state, and the imgRAM zeroed and ready for the next backprojection task.

6.4 Communications

Outside of designing the system to support the backprojection computation, significant effort was spent on modifying the XD1000 reference design infrastructure to support the type of data transfer we required. The majority of

the focus was directed at the HyperTransport module and the SRAM controller. Details of the modifications and additions made to these modules can be found in the following sections.

6.4.1 HyperTransport protocol

6.4.1.1 HT_DATA_CHECK

The modified data receive controller on the HyperTransport bus is designed with the goal of minimizing the transfer time between the CPU and the SRAM. As noted, the original design supplied by XtremeData did not take advantage of bulk data transfers, and passed ACK signals back to the CPU in-between every word of data received. This represented an unnecessary overhead and needed to be removed to improve transfer speeds.

To introduce bulk data transfers to the HT_DATA_CHECK module, we explored two different options of bridging data on the HT bus to the SRAM. The first option was to embed the SRAM address as part of the packet transferred over the HT bus. The data buffer that collects data from the HT bus has a word size of 64 bits, while the SRAM has a data width of only 32 bits. This allowed us to use the top 32 bits of each word sent over the HT bus to store the SRAM address, and the bottom 32 bits to store the data to be transferred to the SRAM. While this approach offered flexibility and provided us control over where we would like to store a piece of data in the SRAM, we quickly realized that the pattern of transfer for storing the sinogram into the SRAM does not require random access, so the actual SRAM address can be generated locally in the HT_DATA_CHECK module with a simple up-counter. This led us to decide on implementing the second proposed method of bridging the HT and the SRAM for sinogram data transfer.

The second method of transferring the sinogram to the SRAM uses the entire 64 bits of the HT packet for storing sinogram data. Since each data point in the sinogram uses 16 bits, this approach allows us to store 4 data points per packet.

This would equate to transferring half as many packets over HT as compared to embedding the SRAM address within the HT packet.

The process of receiving data on the HT bus and subsequently sending it to the SRAM is then outlined as follows:

1) The host software initiates the data transfer by sending appropriate HT header packet.

2) The HT controller on the FPGA receives the transfer request and generates a transfer request signal over the Avalon bus.

3) The HT_DATA_CHECK module receives the transfer request and initializes the SRAM address counter to 0.

4) Sinogram data is transferred to the HT_DATA_CHECK module in packets each containing 4 data points.

5) For each data_in_valid signal received by the HT_DATA_CHECK module, the 64-bit data packet is split into two separate 32-bit data packets and stored into the SRAM, with the top 32 bits stored in sram_address+1.

6) sram_address is incremented by two for every successful piece of sinogram data received over HT.

7) After the transfer is complete, a MSI is sent from the CPU to the FPGA which initiates the backprojection process.

6.4.1.2 HT_DATA_GEN

The modified HT_DATA_GEN module is composed of two separate modules. The first module, HT_block, is composed of a FIFO that stores and concatenates the reconstructed image values into 64 bits and readies them for transfer over the HT

bus. In addition, the HT_block module also contains a series of registers designed to delay the signal which initiates the image transfer (write_sram). This delay is designed to accommodate for the pipeline on the output of the reconstruction processing engines, as the write_sram signal is generated from the control module instead.

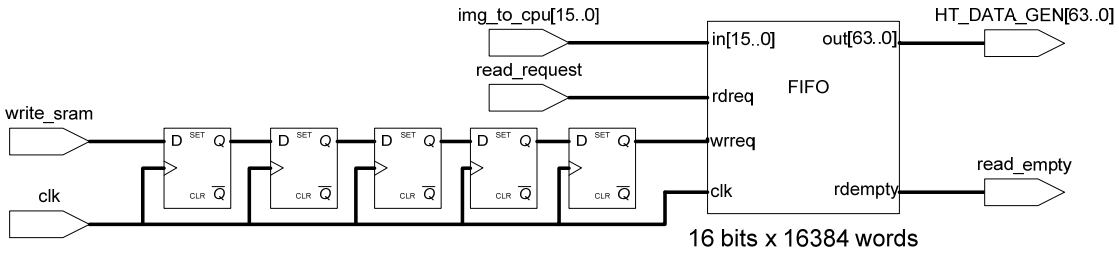

Figure 15 Schematic of HT_block FIFO system for ray-by-ray back projector

The read_request signal in Figure 15 are generated from the HT_DATA_GEN module, and serves as an ACK signal for when the read_empty signal of the FIFO drops low, signaling that there is data available to be read. The FIFO is implemented using the M4K Block RAMs on the Stratix II FPGA. Since the FIFO is 16-bits wide at its input and can store up to 16384 words, it occupies 64 separate M4K RAMs on the FPGA. The depth for the FIFO is chosen to allow sufficient buffering capacity should the HT or Avalon bus be congested with excessive traffic.

The HT_block module is implemented differently for the pixel-by-pixel method of image reconstruction. Since image pixels in a processing block are completely processed after the block has finished executing, the partial image data can be transferred back to the CPU while the next image block is being processed. This calls for a FIFO that is large enough to possibly store the entire 128x512 image pixel block. Since the pixel-by-pixel system does not require any MRAMs in its implementation, the MRAMs on the Stratix II can be used for this FIFO.

The HT_DATA_GEN module is a slave on the Avalon bus that is capable of requesting data transfers from the bus master. This module generates the read_request signal to the image FIFO when the read_empty signal is low, with data then read out in 64 bit wide words on each clock cycle that the read_empty signal remains low.

6.4.2 SRAM_controller

The SRAM_controller design is based off of a ZBT_SRAM controller included in the XD1000 reference design. The controller is positioned on the Avalon bus as a memory-mapped slave and communicates with the SRAM attached to the FPGA. The primary function of the SRAM_controller is to act as a bridge between the HT receive controller to write the sinogram data to the SRAM, and to load the sinogram data into the local sinoRAM Block RAM in the FPGA.

The ZBT_SRAM controller supplied in the XtremeData reference design does not effectively support bulk read/writes to the SRAM. Instead, the original ZBT_SRAM controller is designed for single word read/writes to test SRAM functionality. Since the backprojector application design calls for the sinogram to be stored in the SRAM significant modifications were required to eliminate the latency of single word writes and reads.

The biggest change we made to the ZBT_SRAM controller is the path which data is supplied. Although Avalon is designed to support burst transfers, the bus master module supplied by XtremeData did not implement this feature. Since the SRAM is the only resource of interest, we decided to bypass the Avalon bus, and establish a direct connection between the SRAM_controller and the HT controllers. This gave us flexibility in both the width of this paths and the pace at which data is transferred between the two. Also, since our processing engines are not memory-mapped to the Avalon bus either, we routed a separate bus connecting each processing engine with the SRAM_controller.

The SRAM provided with the XD1000 module has a width of 32-bits, which allows us to store two sinogram values, each 16 bits wide, into one location in the SRAM. Since the SRAM controller operates at a frequency of 200 MHz, this allows us to read/write four sinogram values per cycle of the user clock, which operates at 100 MHz. During read operations these four sinogram values are then buffered in a 1024 word FIFO, the output of which is fanned out to the sinoRAM block memories in each processing engine.

The sino_load_control module also exists as part of the modified ZBT_SRAM controller. This module is responsible for generating the correct addresses for the sinoRAM modules when loading sinogram values from the SRAM. In addition, this module acts as a local control to the FIFOs in the SRAM_controller, and communicates with the State Control module of the backprojector to initiate system state transitions. The write addresses to the sinoRAM in each processing engine is also generated in this module. During sinogram loads this module effectively controls the pace at which sinogram data is written to the sinoRAM of each processing engine.

6.5 Clocks

The XD1000 reference design came supplied with a PLL module that generated a 100 MHz clock for the user logic blocks of the FPGA. This clock is used by the Avalon bus and the controllers for the SRAM and HyperTransport. In addition, the ZBT_SRAM controller takes in another clock signal (zbt_clk) that has a frequency of 200 MHz. This zbt_clk is used for controlling the physical layer of the connection to the SRAM.

Initially, we realized that the processing engine design may not be able to meet the timing requirements of a 100 MHz user clock. This is due to the high number of computations that must be performed in the processing engine data path in a single cycle. We can remedy this problem by pipelining the computation in the

processing engine, effectively breaking up the datapath into many shorter paths separated by registers. However, for an initial design we aimed for a 25 MHz user logic clock to accommodate for the long data path. Once the design has been realized with a 25 MHz user clock, the plan was to then add enough registers to the data path to achieve a 50 MHz user clock, and finally move to the targeted 100 MHz user clock as the final step of the design.

6.6 Processing Engine

The heart of the hardware backprojector lies in the processing engines. This module represents the innermost loop of the algorithm, and is responsible for performing the bulk of the required computations. Along with control signals generated from sino_load_control and state_control modules, the processing engine modules accept the x, y, and θ values from state_control as computation inputs, and SinoAddress_a and SinoAddress_b signals from sino_load_control as addresses for the sinoRAM during the sinogram load stage.

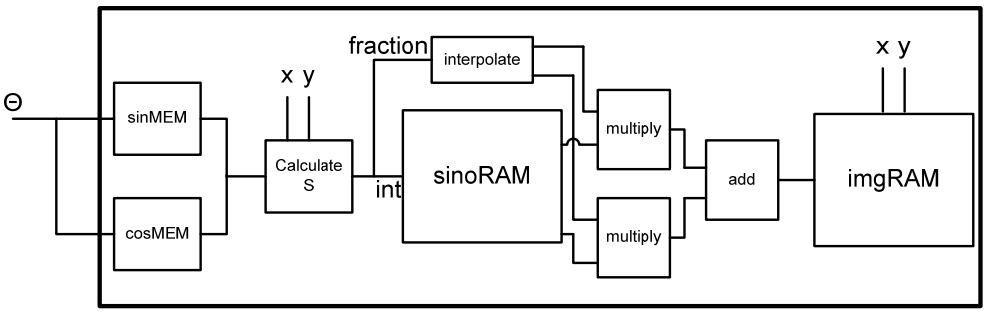

Figure 16 Hardware Block Diagram of Processing Engine

Every cycle, the current set of x, y, and θ values are used in the computation. The θ input value is accepted by the sinMEM and cosMEM ROMs as addresses, and the resulting outputs from these memories constitute the sin θ and cos θ operations. These sin(θ) and cos(θ) values are then multiplied with the y and x inputs respectively to find the complete value of S. This S value is then split into the

integer and fractional portions, with the integer portion going into the sinoRAM as an address input, and the decimal portion going into the interpolate module to be used for interpolation.

Since the sinoRAM is a true dual-port memory, capable of reading out data from two locations at once, the sinogram data at location S and S+1 are both retrieved. At the same time, the interpolate module calculates the interpolation factors by subtracting the decimal portion of S from the integer 1. The difference, along with the original decimal value, is then multiplied with the outputs of the sinoRAM to calculate the interpolated contributions from each of the two S sensors. The results are the summed.

For pixel-by-pixel, an accumulator exists here that sums the result from this cycle's computation with what's already in the local imgRAM. For ray-by-ray, the results of all processing engines are first summed together, and then accumulated with the pixel data from the global imgRAM.

The datapath within the processing engine is implemented with a 32-bit precision, with the last 16 bits being dropped after image pixel data accumulation is complete. Since this hardware design is not constrained by this datapath, the 32-bit precision is used to preserve as much accuracy as possible throughout the reconstruction process. A detailed analysis on precision of this datapath and its effects on the reconstructed image can be found in [1].

7. Results

Figure 17 presents the reconstructed images produced from our system. Visually, there is no significant difference between the three different methods of backprojection.

Figure 17 Reconstructed Images. Pixel-by-pixel (left), ray-by-ray (middle), Software (right)

7.1 Performance Comparisons

Table 8 below shows the results of our VHDL implementation and the Impulse C implementation presented in [1], compared to the software benchmark described in section 3.2. The software benchmark scores were obtained using two quad-core Intel Xeon L5320 processors with operating frequencies of 1.86 GHz. The base results represent a serial execution, and the multi-threaded result is obtained using OpenMP and 8 threads distributed across the eight available CPU cores.

Table 8 Execution Times

Design	512 Projections (Speedup vs. Multi-threaded Software)	1024 Projections (Speedup vs. Multi-threaded Software)
Base Software	5.17 s (.2x)	20.93 s (.19x)
Multi-threaded Software	1.06 s (1x)	3.95 s (1x)
Impulse C	31.83 ms (33x)	64.52 ms (61x)
VHDL (Pixel-by-pixel)	22.60 ms (47x)	95.29 ms (41x)
VHDL (Ray-by-ray)	21.68 ms (49x)	38.02 ms (103x)

The major reason for the FPGA implementations being significantly faster than the software benchmark is the ability to exploit additional parallelism. Compared to the multi-threaded software implementation, which was limited to 8-way parallelism from the 8 processor cores, loop-unrolling allowed us to process 128 elements in parallel, resulting in much faster performance than the 8 way parallel of the software implementation. In addition, manual pipelining techniques allowed us to increase our throughput of each processing engine, which provided additional processing speed benefits.

The tight coupling of the FPGA and the CPU in the XD1000 also played a pivotal role in the results we achieved. Due to the high bandwidth and low latency between the CPU and the FPGA, transferring data to the FPGA did not prove to be a bottleneck for this application, and we were able to focus the resources and effort to optimize the datapath of the system.

Comparing the pixel-by-pixel system and the ray-by-ray system, it is evident that while the 512 backprojectors have very similar performance, the 1024 pixel-by-pixel backprojector is significantly slower. This is due to the pixel-by-pixel backprojector's inability to scale to a 128-way 1024 backprojector, so only 64 processing engines were instantiated; the sinoRAM and local imgRAM for pixel-by-pixel proved to be resource constraints, and directly limited the number of processing engines. The ray-by-ray system avoids this bottleneck by largely freeing up the imgRAM requirement with its global imgRAM implemented in 8 MRAM blocks.

Since the Impulse C backprojector was implemented using the ray-by-ray system, the VHDL ray-by-ray backprojector provides an appropriate point of comparison. An analysis of why Impulse C was slower than hand coded VHDL can be found in 7.4 VHDL vs. Impulse C.

7.2 Execution Time Breakdown

The breakdown of the execution time shown in

Table 9 reflects the ray-by-ray 1024 backprojector system. It is provided to demonstrate the benefit of a tightly-coupled FPGA accelerator. It includes improvements we implemented to the HyperTransport and SRAM hardware modules, as well as the control software.

Table 9 Breakup of Execution Time. Ray-by-ray from 1024 projections

Execution Stage	Time	Percentage
Transfer Sinogram from CPU -> SRAM	6.05 ms	16%
Read Sinogram SRAM -> FPGA	5.40 ms	14%
FPGA Computation	20.97 ms	55%
Transfer Image from FPGA -> CPU	2.60 ms	7%
Post Processing	3.00 ms	8%
Total	38.02 ms	100%

Analysis of this breakdown reveals that the data transfer between the FPGA and CPU consisted of only 23% of the total execution time, compared to 55% of time spent in computation. This goes to show the effectiveness of a tightly-coupled system. On a platform with less bandwidth between the CPU and FPGA, the time it takes to transfer the data to and from the FPGA could easily dominate the execution time.

7.3 Resource Utilization

This section is designed to provide a point of comparison, in terms of resource usage, between the pixel-by-pixel and ray-by-ray backprojectors with the same number of processing engines (128).

Table 10 shows the percentage of each resource consumed for each parallel backprojector.

Table 10 Resource Utilizations of 128-way parallel Systems

Resources	Available	Pixel-by-pixel (512x512)	Ray-by-ray (1024x1024)
Logic			
Combinational ALUTs	143,520	22%	20%
Dedicated Logic Registers	143,520	20%	45%
RAM			
M512 (576 bits)	930	33%	74%
M4K (4.5 Kbits)	768	100%	80%
MRAM (576 Kbits)	9	56%	100%
DSP			
9-bit DSP elements	768	100%	67%

In terms of logic, both implementations have fairly consistent resource usages. The increase in register usage for ray-by-ray is caused by the additional registers needed to accommodate for the global imgRAM. Since the global imgRAM is composed of 8 MRAM blocks spread out across the chip, an extensive series of registers were needed to pipeline their inputs and outputs in order for the place & route tool to meet the 100 MHz clock requirement. The local imgRAMs in pixel-by-pixel did not require such extensive buffering.

In terms of distributed Block RAM, pixel-by-pixel consumed more M4K blocks to instantiate its local imgRAMs. Since the ray-by-ray system is backprojecting from a 1024x1024 sinogram, it requires a bigger sinoRAM (implemented using M512 RAM) to accommodate for the larger inputs. This increase in sinoRAM usage is the only significant cost of extending the ray-by-ray to accommodate a 1024x1024 sinogram input; for a 512x512 ray-by-ray, the size of sinoRAM required would be halved, which means the amount of M512 RAM used can be halved as well.

7.4 VHDL vs. Impulse C

The last section of this results chapter is dedicated to the comparison versus the Impulse C backprojector implemented by Subramanian et al. [1]. This section is broken up into two specific points of comparison: performance and development time/effort.

7.4.1 Performance

Since the ray-by-ray backprojector used the same algorithm as the Impulse C backprojector, we decided to use it as the basis for performance and resource usage comparisons against the Impulse C design.

Table 11 Execution Time

Design	1024 Projections	Speedup
VHDL	38.02 ms	1x
Impulse C	64.52 ms	.59x

As shown in Table 11, the VHDL 1024x1024 backprojector is ~1.7x faster than the Impulse C version. This can be attributed to a few different factors, most of which become evident after the results in Table 12 are presented.

Table 12 Breakup of Execution Time (1024 Projections)

Execution Stage	VHDL	Impulse C
Transfer sinogram from CPU -> SRAM	6.05 ms	4.20 ms
Read sinogram SRAM -> FPGA	5.40 ms	11.12 ms
FPGA computation	20.97 ms	42.40 ms
Transfer image from FPGA -> SRAM	2.60 ms	2.50 ms
Transfer image from SRAM -> CPU		1.30 ms
Post processing	3.00 ms	3.00 ms
Total	38.02 ms	64.52 ms

Examining Table 12 reveals that there is one main area where the VHDL version performs significantly better than Impulse C version: FPGA computation. The VHDL design's increased performance in the FPGA computation stage can be attributed to the ability of its processing engines to produce a result every single cycle. The Impulse C design was not able to implement the sinoRAM Block RAMs as true dual-port because of limitations in the Impulse C tool, which limited its compute engines to produce a result only once every two cycles.

A second area where the VHDL version performs faster than the Impulse C version is the transfer of the sinogram from the SRAM to the FPGA. The SRAM controller that is supplied as part of the Impulse C XD1000 Platform Support Package can only achieve the maximum throughput for SRAM to FPGA communications for a data width of 64 bits. Since the both the image and sinogram data are stored in a 16-bit format, several data points must be packed together to achieve the maximum transfer speed. However, the additional logic required to pack and unpack the data caused the Impulse C design to fall short of the 100 MHz timing requirement. This led to the ~2x SRAM to FPGA transfer speed penalty seen by the Impulse C design.

7.4.3 Development Time and Effort

The Impulse C and VHDL backprojectors were created in parallel, by two separate designers. In terms of design time, the initial design for the hardware backprojector took approximately 12 weeks to complete. This includes understanding the algorithm, determining opportunities to exploit parallelism, benchmarking the hardware, learning the Linux drivers and the XD1000 reference design hardware modules, and finally writing the VHDL. In comparison, the initial Impulse C design took approximately 9 weeks, and includes the time it took for the designer to get acquainted with the Impulse C tools, understand the tool flow and design methodology, and finally optimizing the C benchmark for hardware.

The real differentiation comes from the incremental time to extend the design from 512x512 to 1024x1024. This evolutionary design step took the hardware VHDL approach one week to complete, since it required changing much of the VHDL code and hardware structure to accommodate for the increased data size. However, since the Impulse C designer had acquired the knowledge to effectively utilize Impulse C through implementing the first design, this second pass required only 1 day of development and testing time.

It is important to point out that both the Impulse C designer and the VHDL designer had similar FPGA development experience, so these development times are representative of what developers in industry will experience.

A similar story is seen in the comparison of design complexity between Impulse C and VHDL. The VHDL backprojector was designed and coded in approximately 10,000 lines of code, compared to just 3,000 in the Impulse C backprojector. In addition, Impulse C's platform support package abstracted nearly all hardware management away from the developer, which resulted in a much smoother development process.

8. Discussion

8.1 Hardware Design Process

Prior to writing code for the backprojector, we first examined the trade-offs of the pixel-by-pixel and ray-by-ray methods of reconstruction. It was apparent that the pixel-by-pixel structure lends itself better to hardware design and implementation, as it has a hierarchical structure that is very compatible with the design abilities of HDL.

After the 512x512 projector was complete, several observations regarding the performance of the pixel-by-pixel and ray-by-ray methods of reconstruction were made. First, the pixel-by-pixel method would not scale as effectively to 1024x1024 due to resource limitations. Since the ray-by-ray can use the MRAM to store the image data, much more of the M4K and M512 RAMs were available for the sinogram data. We realized that if we use the pixel-by-pixel method to construct a 1024x1024 projector, we would need to decrease the number of processing engines to 64 due to the shortage of M4K and M512 RAMs.

Second, we realized that the transfer time saved by overlapping communication with computation would not be significant compared to the runtime of the compute loop. This is especially evident as the size of the projector scales to 1024x1024. The processing time scales with the sinogram size, but the reconstructed image stays at a constant 512x512, which means that the data that must be transferred back to the CPU does not change in size. The pixel-by-pixel backprojector is able to mask this transfer time since the outer loop will only traverse through each pixel once. So after a pixel has been reconstruced, it can be sent back to the CPU while another pixel is being reconstructed. This is not possible with ray-by-ray, since reconstruction is not complete for any single pixel until all the pixels have been fully reconstructed.

We decided the benefits of pixel-by-pixel reconstruction did not outweigh the cost for the 1024x1024 implementation, so we adopted the ray-by-ray reconstruction method for the 1024x1024 version. This approach allowed us to explore both reconstruction algorithms and compare and contrast their individual results.

8.2 Strengths and Weaknesses of VHDL Design vs. Impulse C.

While the traditional view on C-to-FPGA style programming models indicates that hand-coded hardware design should always be able to produce faster results in terms of performance, this work and that of Nikhil Subramanian [1], indicates this may be wrong. Our hardware implementation is 550x faster than un-optimized single-threaded software (on a 1.86 GHz Intel Xeon L5320 processor), and 103x faster than multi-threaded software with 8 threads, but the Impulse C implementation was still able to come within .59x the performance [1], while keeping almost identical hardware usage. In terms of productivity, we estimate that a designer well versed in Impulse C can produce working designs 2-5 times faster than HDL, particularly for hardware that must interface with C code on the host processor.

As stated in [1], Impulse C's ability to create applications entirely in C but have them partitioned across the CPU and FPGA presents a very attractive option for the designer. Functional verification of prototype designs is greatly enhanced, and the design effort benefits that Impulse C has over HDL simply cannot be ignored. From initial development time to design verification effort, Impulse C presents a seamless way to integrate software and hardware that allows rapid prototyping of applications targeting hardware coprocessors.

Another difficult aspect of hardware design was the timing issues associated with large designs. As the initial design consumed many of the available resources, the

routing algorithms in the Altera Place & Route tools could not create FPGA configurations that can meet the desired 100 MHz clock frequency. As a result, much hand tweaking was needed to effectively pipeline the dataflow between the various memories, multipliers, and adders. Impulse C eliminates much of this effort by automatically pipelining intensive computations with the included tools.

A definite strength hardware design has over Impulse C still lies in the efficient implementation tricks and flexibility that can be exploited in the hands of a skilled designer. Situations which require simple modifications like adding registers to the input and output of a computation is not possible with Impulse C unless the designer dives into the generated HDL code. This lack of flexibility also makes implementing control logic in the generated pipeline extremely difficult. Further, system parameters such as the operating clock frequency cannot be controlled by the Impulse C designer. This means that the common hardware design technique of incremental designs using slower clock frequencies can not be performed with Impulse C.

8.2 Benefits of a tightly-coupled FPGA accelerator

We found the tightly-coupled FPGA coprocessor offered by the XD1000 is a significant advantage. Our results showed that, contrary to commonly held beliefs, the time spent transferring data to and from the FPGA was not the bottleneck of our application. Instead, data transfer occupied a small fraction of the total execution time. This suggests that as technology moves towards higher bandwidth and lower latency in FPGA coprocessor implementations, designers will no longer be forced to design applications around the time it takes to transfer data to the FPGA, and can instead focus on utilizing available hardware resources to produce the most efficient computation structure. There will always be applications where the communication to computation ratio rules out FPGA

coprocessor implementations, but tightly coupled FPGA-CPU systems allows many more applications to benefit from these systems. This can also be seen in [5,6], where the efficient use of on-chip memory bandwidth was the critical concern.

8.3 Ray-by-ray vs. Pixel-by-pixel

The ray-by-ray and pixel-by-pixel methods of parallelizing the backprojection algorithm each has its advantages and disadvantages. Ray-by-ray is more scalable and faster for larger sinograms. With the resource limitations of the Stratix-II FPGA, we were able to create a 128-way parallel ray-by-ray backprojector, but we were limited to only 64-way parallel pixel-by-pixel backprojector. Ray-by-ray also has fewer SRAM accesses, since the sinogram is loaded only once. This is very beneficial for large sinogram sizes, where repeated access to the SRAM can prove prohibitive.

The biggest disadvantage of ray-by-ray is that it presents additional difficulties when trying to achieve timing closure. The global imgRAM has an extremely large fan-in from all processing engines. Combined with distributing this to 8 separate MRAM blocks instantiated as one single memory, a lot of design effort and registers are needed to have a system that can match pixel-by-pixel in terms of operating frequency.

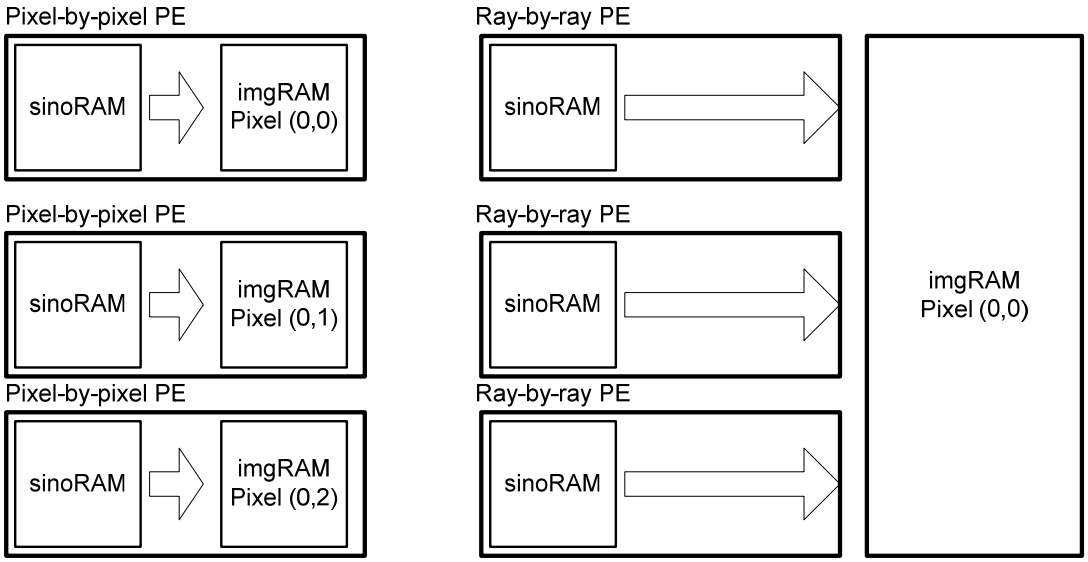

Figure 18. Pixel-by-pixel vs. Ray-by-ray imgRAM configuration

A key advantage of pixel-by-pixel is that it is easier to implement in hardware. It presents a natural processing engine hierarchy, as each processing engine is assigned to one pixel of the image and each contain one sinoRAM and one imgRAM. This structure allows the processing engines to operate independent of how the processing is distributed, thus allowing hierarchies of processing engines to be easily created. Pixel-by-pixel also has a less complex datapath, which makes timing closure easier to achieve. Finally, pixel-by-pixel can overlap computation with transferring the image back to the CPU, which is very beneficial for loosely-coupled systems with less bandwidth between the CPU and FPGA, and it effectively eliminates the cost of the image transfer.

9. Conclusion

9.1 Future Direction

With the base design that we have established, it will be interesting to extend our design to CT forward-projection. We have performed some initial analysis on a forward-projector design, and the XD1000 infrastructure we have in place for the backprojector should easily port over. This includes the HyperTransport controllers and the SRAM controller, as well as the sinoRAM and imgRAM memory structure. In addition, the hierarchical design and trigonometric computation system we implemented can also be utilized for a forward-projector.

Another future research interest is implementing the fan beam or cone beam backprojectors in an FPGA coprocessor. These backprojection algorithms more closely model what the real world CT scanners are actually processing, and they would provide a better representation of a FPGA coprocessor's performance if used in a real world system.

Finally, this work can be extended to implement an iterative backprojection algorithm. Due to the amount of computation required iterative backprojection, processing on sequential processors is not feasible due to the time required to reconstruct the image. As we've demonstrated with the parallel beam backprojector, this roadblock can be resolved with an FPGA coprocessor implementation, as the spatial parallelism available in FPGAs can provide tremendous benefits in terms of processing speed.

9.2 Conclusion

We created a FPGA implementation of a CT backprojection algorithm using VHDL. In the process, we explored two different parallel reconstruction techniques: pixel-by-pixel and ray-by-ray. We concluded that the ray-by-ray method of parallel backprojection was more effective for this application with sinogram size

of 1024x1024 due to its ability to scale more effectively compared to pixel-bypixel. We also compared our results with an Impulse C implementation of the same algorithm [1], as well as a multi-threaded software implementation, and determined that our hand-coded hardware implementation is 1.69x faster than Impulse C and 61x faster than multi-threaded software with 8 threads. Finally, we conclude that FPGA coprocessors, like the XtremeData XD1000, are a viable and effective way to accelerate CT backprojection.

References

[1]N. Subramanian, *A C-to_FPGA Solution for Acclerating Tomographic Reconstruction*, M.S. thesis, University of Washington, Washington, USA, June, 2009.

[2]P. E. Kinahan, M. Defrise, and R. Clackdoyle, “Analytic image reconstruction methods,” in Emission Tomography: The Fundamentals of PET and SPECT, pp. 421–442, Elsevier Academic Press, San Diego, Calif, USA, 2004.

[3] N. GAC, S. Mancini, M. Desvignes, and D. Houzet, “High Speed 3D Tomography on CPU, GPU, and FPGA,” EURASIP Journal on Embedded Systems, vol. 2008, Article ID 930250, 12 pages, 2008.

[4] M. Leeser, S. Coric, E. Miller, H. Yu, and M. Trepanier, “Parallel–beam backprojection: An FPGA implementation optimized for medicalimaging,” Proc. of the Tenth Int. Symposium on FPGA, Monterey, CA,pp. 217–226, Feb. 2002.

[5]N. Johnson-Williams, *Design of a Real Time FPGA-based Three Dimensional Positioning Alogorithm for Positron Emission Tomography,* M.S. thesis, University of Washington, Washington, USA, December 2009.

[6]D. Dewitt, *An FPGA Implementation of Statistical Based Positioning for Positron Emission Tomography*, M. Eng. thesis, University of Washington, Washington, USA, Jan 2008.

[7] I. Agi, P.J. Hurst, and K.W. Current, “A VLSI architecture for high-speed image reconstruction: considerations for a fixed-point architecture,” Proceedings of SPIE, Parallel Architectures for Image Processing, vol. 1246, 1990, pp. 11-24.

[8] Kachelrieb, M.; Knaup, M.; Bockenbach, O., "Hyperfast Parallel Beam Backprojection," Nuclear Science Symposium Conference Record, 2006. IEEE, vol.5, no., pp.3111-3114, Oct. 29 2006-Nov. 1 2006.

[9] Ambric Am2045 CT Backprojection Acceleration White Paper

[10] X. Xue, A. Cheryauka, and D. Tubbs, “Acceleration of fluoro–CT reconstruction for a mobile C–arm on GPU and FPGA hardware: A simulation study,” SPIE Medical Imaging Proc., vol. 6142, pp. 1494–1501, Feb. 2006.

[11] F. Xu and K. Mueller, "Accelerating popular tomographic reconstruction algorithms on commodity pc graphics hardware," IEEE Transaction of Nuclear Science, 2005.

[12] F. Xu and K. Mueller. "Towards a Unified Framework for Rapid Computed Tomography on Commodity GPUs" IEEE Medical Imaging Conference (MIC) 2003

[13] K. Mueller and F. Xu. "Practical considerations for GPU-accelerated CT" IEEE 2006 International Symposium on Biomedical Imaging (ISBI '06) Arlington, VA, April 2006 pp. 1184-1187, 2006.

[14] A. K. Jain. Fundamentals of Digital Image Processing. Prentice-Hall, 1989.

[15] XD1000 Product flyer. http://www.xtremedatainc.com/index.php?option=com_docman&task=doc_details&gid=17&Itemid=129

[16] Wikimedia Commons article: Computed Tomography - CT Scans. http://commons.wikimedia.org/wiki/File:CTScan.jpg

[17] A. Alessio and P. Kinahan, "PET Image Reconstruction," In: Nuclear Medicine (2nd ed.). Henkin et al., Eds., Philadelphia, Elsevier; 2006

[18] Hutton, Brian F. An Introduction to Iterative Reconstruction. Alasbimn Journal 5(18): October 2002. Article N° AJ18-6.

[19] Images generated using the University of Washington Emission Reconstruction Demo (UWERD), © Adam Alessio 2/05.

[20] http://www.slideshare.net/lobelize/ct-2-history-process-scanners-dip Slide 31

[21] FBP Software benchmark designed by Dr. Adam Alessio, Research Asst. Professor, Imaging Research Laboratory, Department of Radiology, University of Washington.

[22] Espasa, R. and Valero, M. 1997. Exploiting Instruction- and Data-Level Parallelism. IEEE Micro 17, 5 (Sep. 1997), 20-27. DOI= http://dx.doi.org/10.1109/40.621210

[23] http://www.impulseaccelerated.com/

[24] "XD1000 Development System Brief", XtremeData, IL, USA.

[25] "Pico E-16 datasheet," Pico Computing, WA, USA.

[26] http://www.drccomputer.com/pdfs/DRC_Accelium_Coprocessors.pdf

[27] http://en.wikipedia.org/wiki/File:Tomographic_fig1.png

[28] http://en.wikipedia.org/wiki/Tomographic_reconstruction

[29] http://www.altera.com/literature/manual/mnl_avalon_spec.pdf

[30] S. Hauck and A. Dehon. Reconfigurable Computing. Morgan Kaufmann. 2008.

[31] http://en.wikipedia.org/wiki/Graphics_processing_unit

[32] A. C. Kak and M. Slaney, *Principles of Computerized Tomographic Imaging*. New York: IEEE Press, 1988.