A C-to-FPGA Solution for Accelerating Tomographic Reconstruction

Nikhil Subramanian

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

Abstract

A C-to-FPGA Solution for Accelerating Tomographic Reconstruction

Nikhil Subramanian

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

Computed Tomography (CT) image reconstruction techniques represent a class of algorithms that are ideally suited for co-processor acceleration. The Filtered Backprojection (FBP) algorithm is one such popular CT reconstruction method that is computationally intensive but amenable to extensive parallel execution. We develop an FPGA accelerator for the critical backprojection step in FBP using a C-to-FPGA tool flow called Impulse C. We document the strategies that work well with Impulse C, and show orders of magnitude speedup over a software implementation of backprojection. We contrast the ease of use and performance of Impulse C against traditional HDL design, and demonstrate that Impulse C can achieve nearly the same performance as hand coded HDL while significantly reducing the design effort.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank the Washington Technology Center (Seattle, WA) and Impulse Accelerated Technologies (Kirkland, WA), for the funding that made this research possible. Many thanks to Adam Alessio for providing expert guidance and the FBP software benchmarks; and Jimmy Xu for being a patient co-worker and creating the HDL version of the design. I would like to express my deepest and most sincere gratitude to my advisor, Professor Scott Hauck, for being a source of tremendous inspiration. Finally, I would like to thank my family for their constant support and encouragement; without them, this thesis could not have been completed

1. Introduction

Computed Tomography (CT) is a medical imaging technique used to create cross-sectional images from x-ray transmission data acquired by a scanner. In tomographic systems, the primary computational demand after data capture by the scanner is the backprojection of the acquired data to reconstruct the internal structure of the scanned object (Figure 1.1). Backprojection can be viewed as the mapping of raw data into the image space and has a complexity of $O(n^3)$ when generating 2D cross-sections and $O(n^4)$ for full 3D reconstruction. In a typical CT system, the data has $\sim 10^6$ entries per cross-section and the process of tracing each datum through the image space is demanding. As a result, hardware acceleration of this process has been the focus of numerous studies.

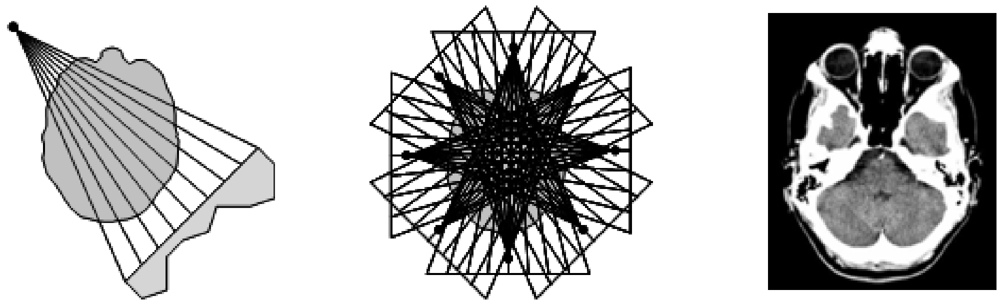


Figure 1.1 Simplified Tomography flow. Scanning (left) produces a set of aggregate data on probe lines. Through backprojection (center) multiple samples are brought together to reconstruct the internal structure of the patient (right).

Previous work on backprojection [3-10] has shown that hardware accelerators can achieve much higher performance than microprocessor-based systems. Field Programmable Gate Arrays (FPGAs) are often used to build such accelerators. The traditional way to design an FPGA-based accelerator is to describe the hardware performing the computation using a schematic or a hardware description language (HDL). For some years now there has been interest in C-to-FPGA tool flows that allow users to describe the computation in C and automatically generate the hardware description. This makes the process of creating FPGA designs easier and allows for wider adoption of FPGAs, even by those with little or no hardware engineering expertise.

The general perception has been that while such high level tool flows are simple to use, they do not provide the same level of performance as hand coded HDL. We investigate this notion by profiling a C-to-FPGA tool flow called Impulse C (Impulse Accelerated Technologies, Kirkland, WA, USA). Our software benchmark was an implementation of the filtered backprojection (FBP) algorithm running on a processor. We partitioned the design such that the compute intensive backprojection step was run on an FPGA while the user interface and some filtering operations were performed on the host processor. The XD1000 (XtremeData Inc., Schaumburg, IL, USA) development system was chosen for the implementation because it offers a tightly coupled CPU-FPGA platform. Details of the Impulse C tool flow and the XD1000 development system can be found in Chapter 2.

This thesis provides a case study on how a C-to-FPGA tool flow can be used to design an efficient FPGA-based accelerator by documenting some of the strategies that work well and some common pitfalls to avoid. It is organized as follows:

- Chapter 2: Background introduces the filtered backprojection algorithm, FPGA-based accelerators and the Impulse C tool flow.

- Chapter 3: Design Considerations discusses issues related to porting a design to Impulse C and also describes the implemented FPGA design.

- Chapter 4: Results presents the performance of the various versions of the design in comparison to software and hand coded HDL.

- Chapter 5: Conclusions is a summary of the lessons learned and a discussion of possible future research.

2. Background

2.1. Tomographic Reconstruction

Tomographic reconstruction is the process of generating cross-sections of an object from a series of measurements acquired from many different directions. In Computed Tomography (CT or “CAT” scans), the measurements are x-ray attenuation data. We now discuss CT systems and the reconstruction process.

2.1.1. CT Systems

A CT system comprises of a scanner that is housed in a radiation safe scanning room and an adjoining computer suite to generate, view and print images. The scanner has a movable platform and a ring shaped gantry that carries x-ray sources on one side and detectors on the other side. The gantry rotates about an axis perpendicular to the platform, enabling measurements at various angles. The sources and detectors in the gantry can come in two configurations, leading to distinct beam geometries. The simpler of these is the parallel beam geometry, which arises from a linear array of sources and detectors. Fan beam geometry is the result of a single source and a linear array of detectors.

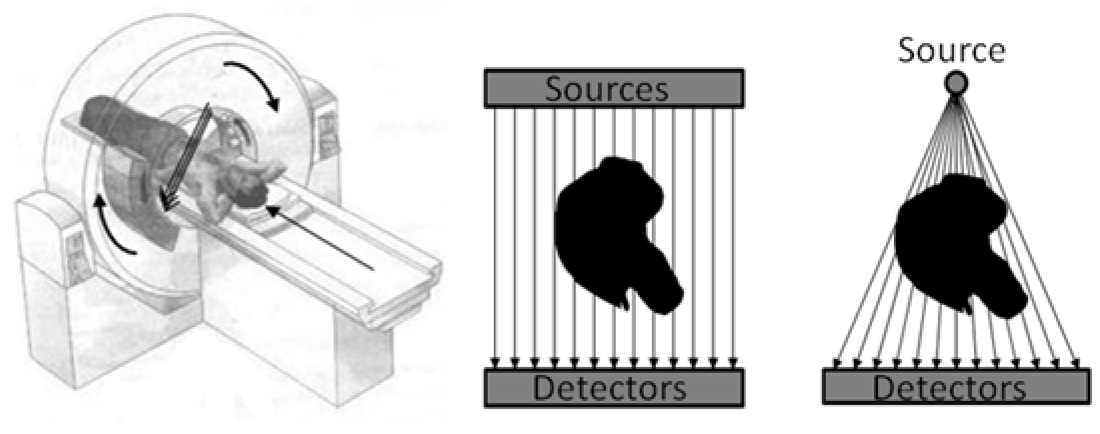


Figure 2.1 CT System. Scanner [14] with movable patient platform and ring shaped rotating gantry. Parallel Beam (center) and Fan Beam (right) geometry.

Patients lie down on the platform, which is moved into the gantry to position the scan area in the path of the x-rays. The sources then generate and direct x-rays through the patient. Different parts of the human anatomy, such as bones and soft tissue, attenuate the incident x-rays by varying amounts. The detectors record the received intensity. The set of detector intensity readings for any angle is called a projection. The gantry rotates and collects several projections at equidistant angles over a 180-360 degree span. From this set of projections a single image slice can be generated. In a typical CT exam, multiple closely spaced image slices are generated to image an entire organ.

First generation CT scanners consisted of a single source and single detector. Both the source and detector needed to translate to collect a single projection; then the source/detector mechanism rotated to collect more projections. In second generation CT scanners, multiple detectors were included to reduce the number of translations and rotations required. Third generation CT scanners have a large array of detectors opposite a single source on a fixed rotating gantry; these scanners do not require a translating step leading to dramatically reduced scan times. Fourth generation CT systems have a fixed detector array that completely surrounds the patient; only the CT tube rotates. The increased sophistication of modern scanners naturally increases the complexity of the reconstruction process; however, the core computation is still conversion of the detector data into an image. One way to solve the image reconstruction problem is with the popular filtered backprojection (FBP) algorithm.

2.1.2. Filtered Backprojection (FBP)

The set of all projections acquired by a scanner is a two dimensional matrix of data. This data represents a 2D spatial function termed a *sinogram*, because each point in the image has contributions from several points that lie on a sine wave in the 2D spatial function. The problem of reconstruction can be viewed as the problem of transforming the sinogram into the image space (X,Y). A plot of the sinogram data on a grayscale is shown in figure 2.2 (left). The simplest way one can imagine solving this is through backprojection which is also known as the summation method or linear superposition method.

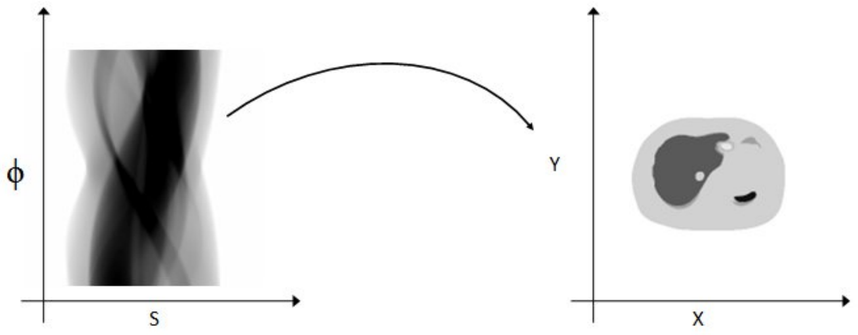


Figure 2.2 Reconstruction [17]. Transformation from sinogram (left) to image space (right)

Backprojection simply reverses the measurement process and can be thought of as smearing the projections back across the reconstructed image. Figure 2.3 below provides some intuition for how this works. In the simplest case, projections are taken in only two orientations. The reconstruction is obtained by wiping each projection back across the image at the angle at which it was acquired. The additive effect of overlapping regions makes those parts of the image brighter.

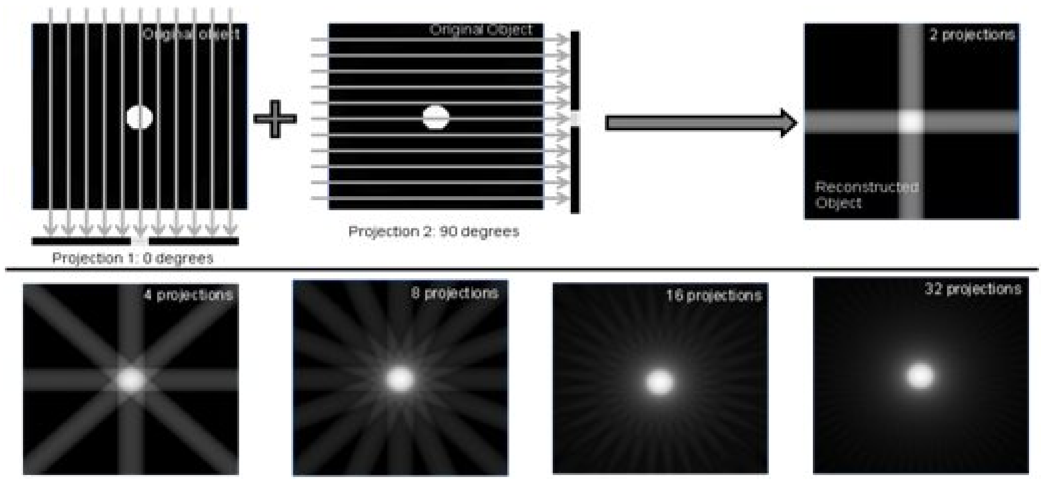


Figure 2.3 Backprojection [18]. In the simple case an image can be generated from just two projections (top). Images generated from 4, 8, 16 and 32 projections (bottom)

As seen in figure 2.3, more projections give a better reconstruction. However, even for a large number of projections, there is blurring around the object. The streaking and star-like artifacts in the image arise because backprojection is not the exact inverse operation of the image acquisition process. To eliminate the low frequency blur, the acquired data (sinogram) is high-pass filtered before backprojection. This is Filtered Backprojection (FBP). The Ramp Filter is often used in FBP because it provides ideal reconstruction in the absence of noise [11]. If noise is present, the Ramp Filter leads can lead to excessively noisy images because it accentuates the high frequency components in the data. Since measured data can contain some noise, the Ramp Filter is often apodized to reduce high frequency contributions. Smooth apodization filters are generally better when viewing soft tissue and sharper filters aid high resolution imaging. Figure 2.4 shows the difference in image quality when backprojection is performed with and without filtering.

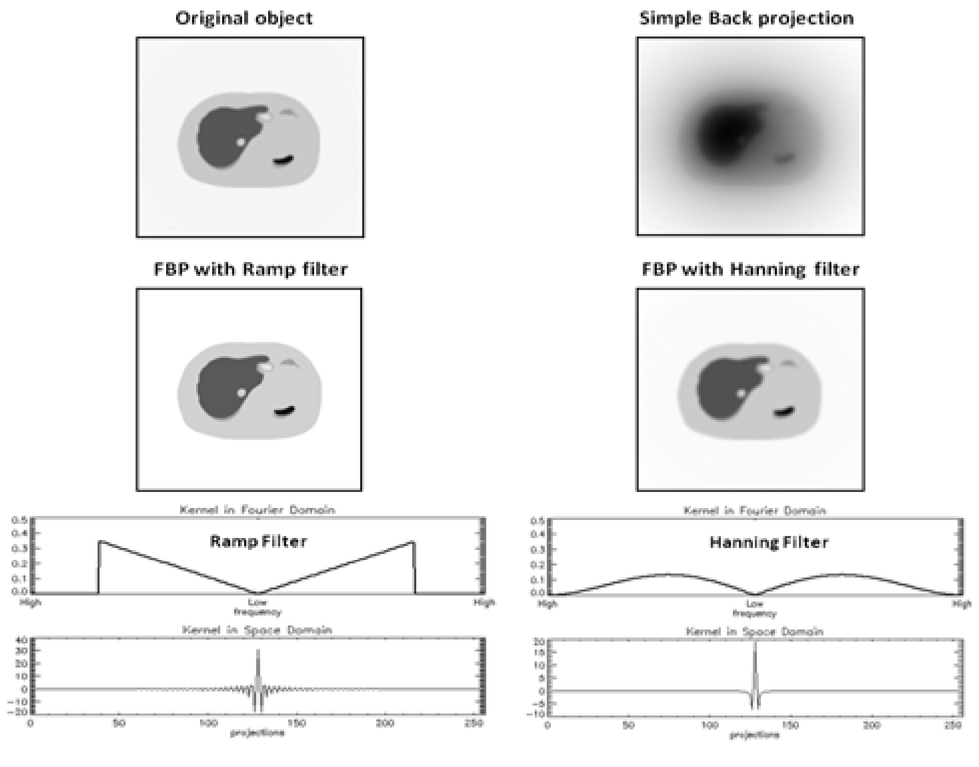


Figure 2.4 Backprojection with and without filtering [17]

The flowchart (figure 2.5) below shows the sequence of operations involved in a simple software implementation of FBP. There is strong motivation to accelerate the backprojection stage that has a computational complexity of O(n^3).

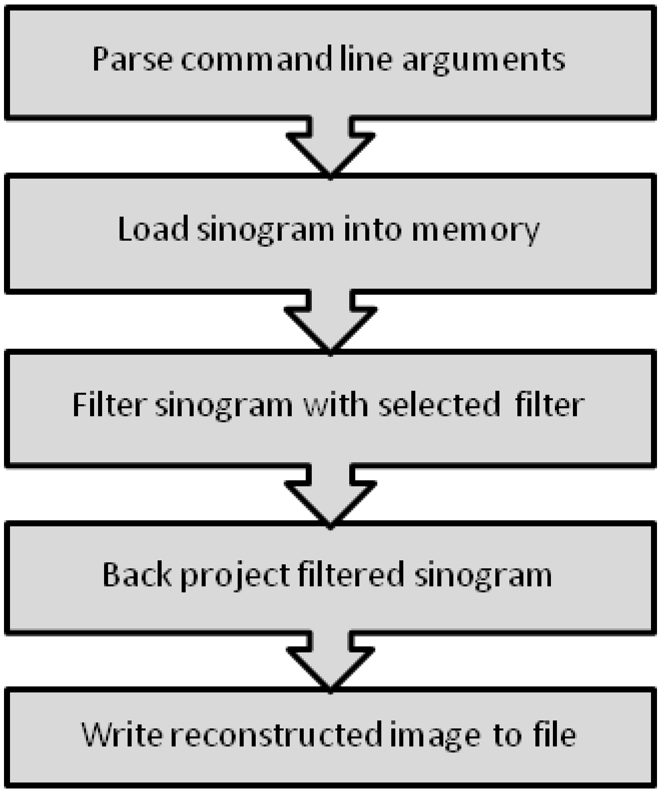

Parse command line arguments
Load sinogram into memory
Filter sinogram with selected filter
Back project filtered sinogram
Write reconstructed image to file

Figure 2.5 Flow of operations in software implementation of FBP

The transformation of generating a sinogram by taking line integrals through the image is known as the x-ray transform which in two dimensions is the same as the Radon transform. FBP is one analytical reconstruction method to implement the inverse Radon transform. Reconstruction can also be performed through other analytic image reconstruction techniques such as Direct Fourier Methods or iterative reconstruction techniques such as Maximum-Likelihood Expectation-Minimization (ML-EM). For a formal mathematical treatment of the analytical and iterative image reconstruction concepts, the reader is directed to this excellent reference [1].

2.1.3. Backprojection

As mentioned earlier, different parts of the scanned object attenuate the incident x-rays by varying amounts. Thus, the scanning process can be modeled as detectors acquiring line integrals of the attenuation along the scanned object. Figure 2.6 shows this model of data acquisition in the image space. The value of an image pixel is proportional to the attenuation at that point. The highlighted pixels along the ray contribute to the measured value, which we will call the "ray-value". These values are aggregated as entries of the sinogram.

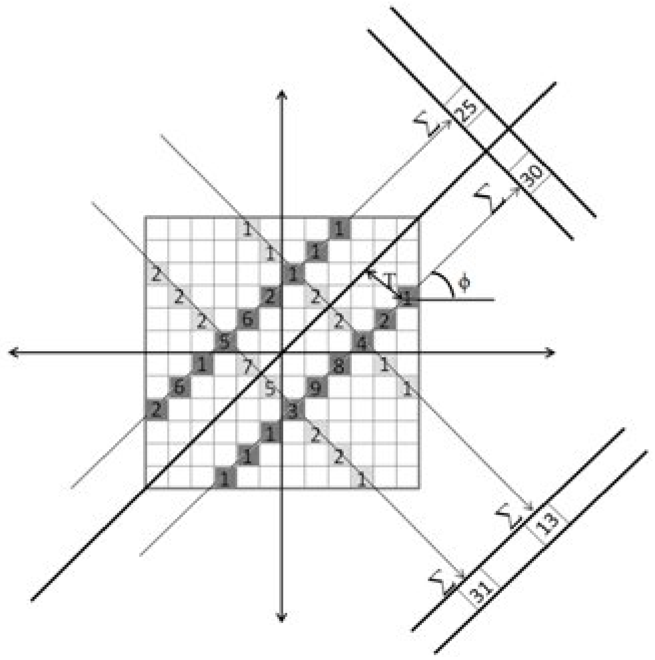


Figure 2.6 Image Pixels contributing to detector values

When reconstructing, we start with an image initialized to 0. Corresponding to each entry of the sinogram (i.e. a ray-value) we identify all the pixels that lie on the ray. The values of those image pixels are then incremented by the corresponding ray-value (figure 2.7). When this process is performed for all entries in the sinogram, we have reconstructed the image. This is ray-by-ray reconstruction. Another way to approach this is to reconstruct one pixel at a time. In this scheme, given a pixel and an angle of measurement, the ray-value that contributes to the pixel needs to be identified. If we go through all angles, identifying all rays that the pixel was on, and add the ray-values together, that pixel is fully reconstructed. We can then go ahead and reconstruct all the other pixels in the same way. This is pixel-by-pixel reconstruction.

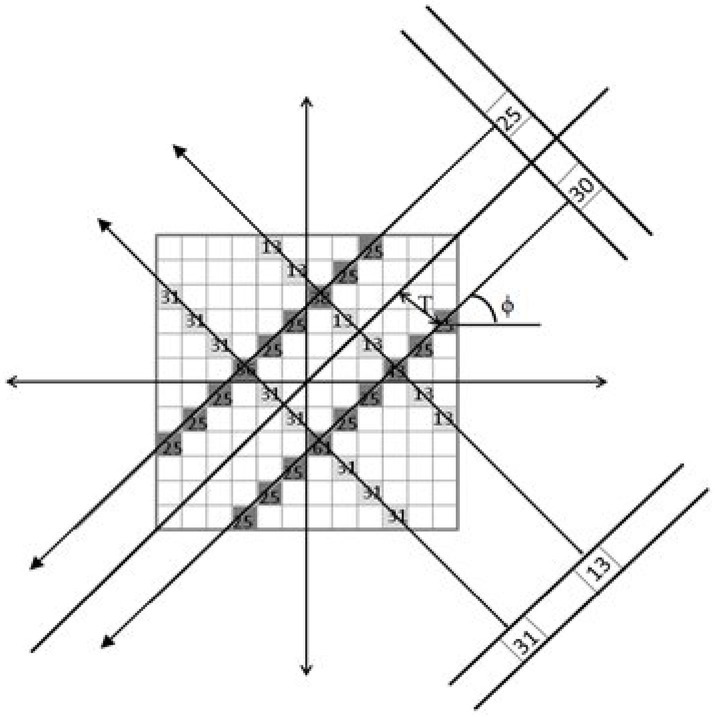


Figure 2.7 Backprojecting along rays

The ray-by-ray and pixel-by-pixel methods produce identical results, and primarily differ in their implications for resources required for implementation. We will carry forward the discussion with the pixel-by-pixel method, realizing that similar concepts apply to the ray-by-ray approach.

When we reconstruct a pixel at a time, the operation that was informally introduced as “smearing” projections back across the image actually involves summing the detector values that contributes to the pixel at each angle. Recall that the sinogram is a 2D matrix with a row of detector values for each angle. Hence, we need the index of the detector value that contributes to the pixel being reconstructed. For a given pixel (x,y) and angle *theta*, the index of the sinogram entry that contributes to that pixel value can be identified using the formula

$$T = x*\cos(theta) + y*\sin(theta)$$

The value of T derived from the computation above does not always result in a unique integer index corresponding to a single ray-value. For example, if T=3.7, the value at index 3 and at index 4 contribute to the image pixel. The fractional part of T (.7 in this case), is used to perform interpolation between the two ray values at index 3 and 4.

The computations involved in reconstructing a pixel are: simple trigonometry to find detector index for each angle, interpolation and accumulation. These procedures by themselves are simple. The computational complexity of FBP does not stem from the complexity of the individual operations, but rather from the sheer number of them. CT scanners acquire between 512 and 1024 projections (with ~1024 entries each) per slice, and so reconstructing a pixel involves performing the computations 512 – 1024 times. Typically 512x512 images are reconstructed, and thus during backprojection each pixel in a 512 by 512 matrix goes through 512-1024 of the simple computations mentioned above.

There is further motivation to focus on the acceleration of backprojection. Filtered backprojection produces the correct reconstructed image from noise-free, continuous data. However, the acquired data in tomographic systems are subject to various random phenomenon and irregularities which FBP does not account for. In order to employ a “closer to real” model of the acquisition process, iterative techniques are employed [15]. From a computation standpoint, these techniques involve repeated transformations between the image space and the projection space in contrast to FBP which makes just one transform of the projection data into the image space (with associated filtering). As a result, depending on the number of iterations used, iterative techniques can be orders of magnitude more computationally demanding than FBP. In FBP, backprojection accounts for 70-80% of the total execution time. In iterative techniques, backprojection and forward projection can account for as much as 70% to 90% of the execution. As a result hardware acceleration of the backprojection and the forward projection step has been the focus of numerous studies [3-10].

2.2. Coprocessor Accelerators

Frequently occurring or complex computations can overwhelm the primary microprocessor (host or CPU) in a computer system. While the host processor performs useful functions such as the user interface and interfacing with peripherals, a coprocessor is better suited to undertake heavy workloads. The most common coprocessor is the Graphics Processing Unit (GPU), also known as a Graphics Card or Graphics Accelerator. GPUs are specially designed chips used to accelerate 3D graphics rendering and texture mapping. With the proliferation of high definition computer displays and rich 3D content, their utility is justified in most modern computing scenarios.

With increasing demand for higher computational performance and increased power efficiency, there are many new and promising coprocessor technologies. The Cell Broadband Engine (CBE) (IBM Corporation, Armonk, NY, USA), is a novel general purpose architecture optimized for distributed computing and is suitable for performing data parallel operations [5]. The Ambric (Nethra Imaging Inc., Santa Clara, CA, USA) is a Massively Parallel Processor Array (MPPA) with 336 processing cores that process data in parallel. It is a streaming architecture with an asynchronous array of processors and memories. These coprocessors can accelerate a wide variety of applications.

For custom or “niche market” computations that require acceleration, an off-the-shelf coprocessor might not provide adequate performance or have the desired power efficiency. For such applications, there are two choices. The first is to design a full custom Application Specific Integrated Circuit (ASIC), which is a piece of custom silicon that can perform the computation. For any given computational task, a well designed ASIC fabricated in the current process technology will provide the best performance. However, this performance comes at a very high cost that can, usually, only be offset by large scale mass production of the design. Since the economic model of a “niche market” computation might not lend itself to mass production, people turn to another choice - Field Programmable Gate Arrays (FPGAs).

FPGAs are prefabricated Integrated Circuits (ICs) that can be programmed and reprogrammed to implement arbitrary logic. The logic circuit to be implemented is usually described using a Hardware Description Language (HDL), which is mapped to the resources on the FPGA by a compiler. Modern FPGAs have millions of gates of logic capacity, millions of memory bits and may even contain one or more embedded microprocessors. Although FPGA performance generally does not rival that of an ASIC, they can provide much higher performance than a standard processor for complex computing tasks. The advantage of FPGAs is that they do not have the prohibitively high costs associated with creating and fabricating ASICs. Hence FPGAs are often used to implement custom coprocessor accelerators.

2.2.1. Coprocessor Acceleration of Backprojection

One of the obvious observations about backprojection is that the reconstruction of any one pixel is completely independent of any other. Thus, multiple pixels can be reconstructed in parallel on hardware that supports parallel processing. This is pixel parallelism. The other option is to reconstruct a single pixel from multiple projections in parallel. This is projection parallelism. An *n*-way parallel system (pixel or projection parallelism) is *n* times faster than a system that processes one pixel or one projection at a time. However, there is a significant difference in the required memory bandwidth. If *n* pixels are reconstructed in parallel, pixel parallelism requires *n* times the memory bandwidth required for the reconstruction of a single pixel, whereas projection parallelism does not require higher bandwidth than a non-parallel implementation [3].

On an FPGA, projection parallelism can be easily exploited. By pipelining the trigonometry, interpolation and summing operations we can further increase the number of operations running in parallel. An FPGA implementation of a 16-way projection parallel implementation with a 7 stage compute pipeline was demonstrated by Leeser et al. [3]. The implementation was on an Annapolis Micro Systems Firebird board comprising one Xilinx Virtex2000E FPGA chip and 36 Mbytes of on board SRAM. They could process a 512x512 image from 1024 projections in 250ms, which represented over 100x speedup compared to their software implementation running on a 1GHz Pentium.

GPUs have been employed to accelerate medical imaging reconstruction [7-10]. One of the main advantages of GPU as a co-processor is that they are relatively ubiquitous in computer systems. Xue et al. [7] demonstrated that backprojecting a 256x256 image from 165 equiangular projections having 512 detectors was 21x faster if implemented using 32 bit fixed point on an Nvidia GPU when compared to floating point on a CPU. Further, a 16 bit fixed point implementation on the same GPU was 44x faster than the CPU.

Table 2.1, adapted from Kachlerieβ et al. [5] below, shows the performance of the CBE in comparison with other custom hardware implementations of FBP The numbers in the table are scaled for the reconstruction of 512x512 images from 512 projections with 1024 detector channels and for advances in process technology. The table also shows the bit-width of the data path employed with *i* representing integer and *f* representing floating point. Agi et al. [4] demonstrated the first ASIC to implement backprojection. The ASIC performance is superior to that of other custom hardware platforms for reasons described in the previous section. The custom hardware accelerators consistently outperform the software benchmarks in all the studies.

Table 2.1 Custom hardware implementations of FBP [5]. Time taken to reconstruct 512x512 images from 512 projections; each having 1024 detector channels.

Group	Type	Hardware	Time
Leeser et al. [3]	i09	CPU	4.66 s
	i09	FPGA (Virtex -2)	125 ms
Agi et al. [4]	i12	ASIC (1um)	.7 ms
Kachelrieß et al.[5]	*f*32	CPU (reference)	5.2 s
		Cell	7.9 ms
Ambric [6]	i16	MPPA	54 ms
Xui et al. [7]	f32	CPU	7.13 s
	i32	FPGA (Altera)	273 ms
	i32	GPU (Nvidia GF 7800)	295 ms

2.2.2. Field Programmable Gate Arrays

FPGAs come in various capacities, ranging from a few thousand gates of logic capacity, to several million. A designer makes the decision on which chip to use based on the requirements of the design. In FPGAs, speedup is not achieved by operating the design at very high clock frequencies, but rather by exploiting the parallelism in the design. Many signal processing and scientific computations have parallelism profiles that make them good targets for parallel computation.

FPGA resources often include distributed memories called Block RAMs, logic cells, registers, and hard wired digital signal processing (DSP) blocks. Each vendor uses different terminology to refer to these resources. This thesis uses the Altera (Altera Corporation, San Jose, CA, USA) names for FPGA resources. Other

FPGA vendors generally have comparable functionality in their chips. The Stratix series is Altera's range of high-end chips, and is frequently used in high performance computations.

Table 2.2 Altera Stratix series [19]

Device Family	Stratix	Stratix II	Stratix III	Stratix IV
Year of Introduction	2002	2004	2006	2008
Process Technology	130 nm	90 nm	65 nm	40 nm
Equivalent Logic Elements	10,570 to 79,040	15,600 to 179,400	47,500 to 338,000	105,600 to 681,100
Adaptive Logic Modules	N/A	6,240 to 71,760	19,000 to 135,200	42,240 to 272,440
Total RAM (Kbits)	899 to 7,253	410 to 9,163	1,836 to 16,272	8,244 to 22,977
DSP Blocks	6 to 22	12 to 96	27 to 112	48 to 170

The basic building block of logic in an FPGA is a "slice", which contains look up tables (LUTs) to implement combinational logic, registers to implement sequential logic, and in many cases a carry chain to implement adders. Altera calls this an Adaptive Logic Module (ALM) in its Stratix series of FPGAs. The term *adaptive* comes from the fact that the ALM can be put in different modes that gives it flexibility to implement arithmetic between the LUTs in the ALM or certain many-input combinational functions with input sharing.

The Block RAMs in FPGAs come in various capacities. Altera calls its largest on-chip memory M-RAM, each of which has 576 Kbits. There are a small number of these distributed around the chip. M4K RAM blocks, each of which has 4.5 Kbits, are much more widespread. There are also several instances of very small memories that are useful as local scratch space or to store constants. M512 RAM blocks are an example of these, and they have 576 bits each. Table 2.2 shows the total memory bits available in each Stratix chip.

The memories can be configured as Single Port, Simple Dual Port, or True Dual Port. A Single Port memory can be read from or written to once every clock cycle. A Simple Dual Port memory can be written to and read from in the same cycle. In addition to this functionality, a True Dual Port memory can have 2 values written to it or read from it in a single cycle. The M512 memory blocks in the Altera chips can operate only in Single Port or Simple Dual Port mode. The M4ks and M-RAMs support all three modes.

The third important resource available in most FPGAs today is hard-wired DSP blocks, which are custom logic that implements DSP arithmetic more efficiently than ALM-based implementations. The Stratix-II DSP block can be configured either as one 36x36 multiplier, four 18x18 multipliers or eight 9x9 multipliers. They can also be configured to perform Multiply-Accumulate (MAC) operations, which are frequently encountered in DSP applications.

2.2.3. FPGA Accelerator Platforms

The term "platform" is somewhat arbitrary, but generally refers to a known, previously verified hardware configuration that may be used as the basis for of one or more specific applications [12]. FPGA Accelerator Platforms have an FPGA coprocessor connected to a host through a high speed communication protocol. Traditionally, the utility of FPGA coprocessors was limited by the fact that the relatively slow links between the host and FPGA eliminated the gains achieved through acceleration. Today multi-gigabit, low latency communication links have made it attractive to offload data parallel applications. The Accelium platform made by DRC Computer and the XD development systems by XtremeData are good examples of devices with high capacity CPU-FPGA links. We chose the XD1000 development system for our implementation.

The XD1000 system has a dual Opteron® motherboard populated with one AMD Opteron processor and one XD1000 FPGA coprocessor module [13]. The coprocessor module has an Altera Stratix II FPGA and a 4 Mbytes of SRAM. The coprocessor communicates to the CPU on the HyperTransport bus. Both the CPU and the coprocessor also have access to additional DDR3 RAM. Figure 2.8 shows the system architecture annotated with theoretical maximum data rates.

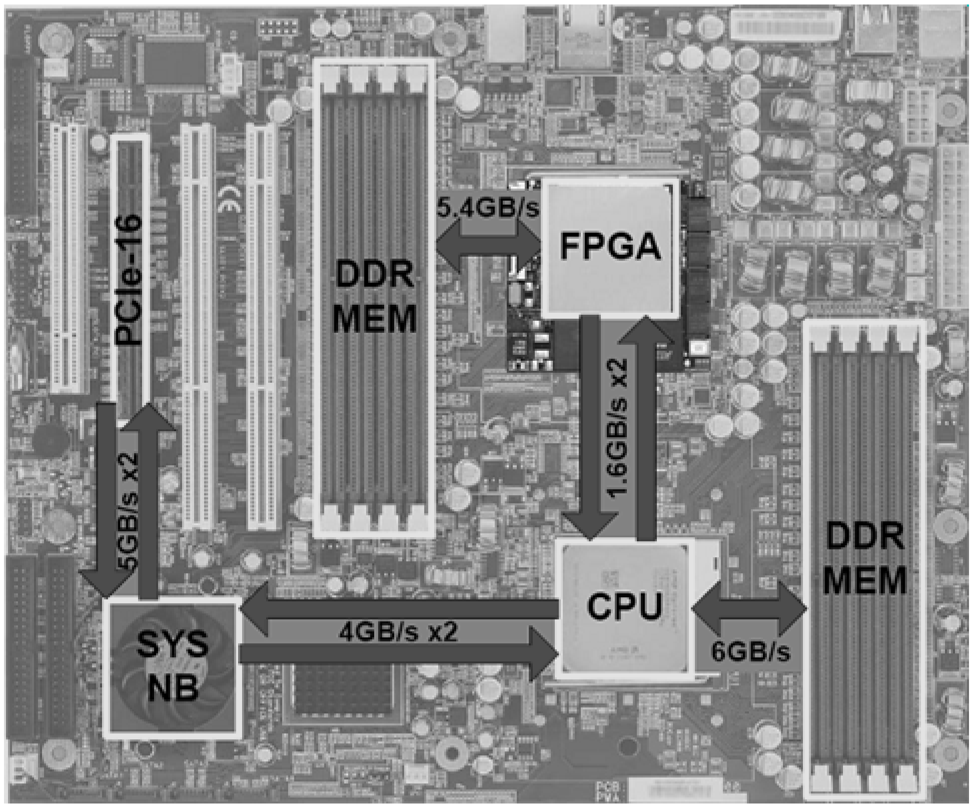


Figure 2.8 XD100 Development System [13]

2.3. C-to-FPGA Design Tools

FPGA designs are usually created by drawing a schematic or writing a hardware description in Verilog or VHDL. Current C-to-FPGA design tools allow users to create designs using C language semantics with added library functions to invoke specific hardware operations. The compiler generates HDL design files from the C code. Special compiler directives in the C code can be used to modify the outcome of the hardware generation process. Once the HDL is available, normal FPGA tool flows supplied by the FPGA vendor can be used to synthesize and map the design to the target chip.

The DK design suite by Agility Design Solutions Inc (Palo Alto, CA, USA) [24], the Mitrion-C compiler by Mitrionics, Inc (Lund, Sweden) [25], Catapult C by Mentor Graphics (Wilsonville, Oregon, USA) [26], and Impulse C by Impulse Accelerated Technologies (Kirkland, WA, USA) [27] are examples of commercially available C-to FPGA solutions. Our goal was to benchmark the performance of Impulse C against software and hand-coded HDL, and to provide a case study on how a C-to-FPGA flow might be applied to accelerate CT reconstruction.

2.3.1. Impulse C

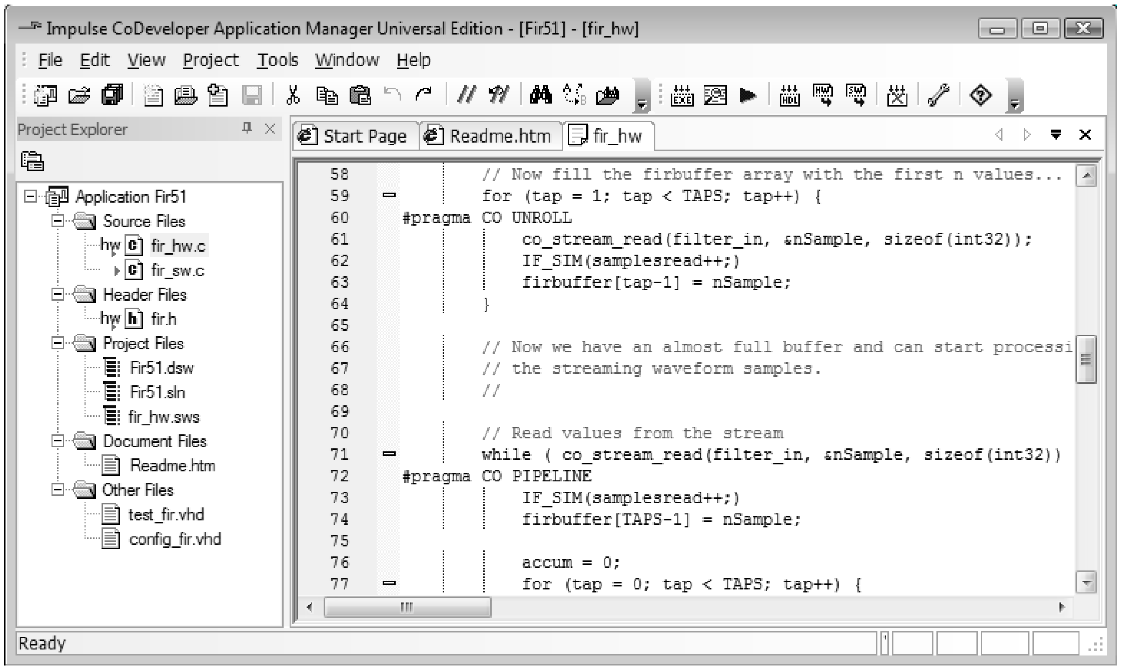
The Impulse C tools include the CoDeveloper C-to-FPGA tools and the CoDeveloper Platform Support Packages (PSPs). Table 2.3 gives a summary of the C-to-FPGA tools. PSPs add platform-specific capabilities to the Impulse CoDeveloper programming tools. With the PSP, users can partition the application between the host and coprocessors, make use of other on board resources, and automatically generate the required interfaces. The interfaces are implemented as a wrapper function around the user logic. We used the Impulse C tools (version 3.20.b.6) with the XtremData XD1000 PSP (version 3.00.a).

Table 2.3 CoDeveloper C-to-FPGA tools

Tool Name	Function
CoDeveloper Application Manager	Project Management and Design Entry.
CoMonitor Application Monitor	Monitoring designs as they execute.
CoBuilder Hardware Generation	Generate HDL from the C processes designated as “hw”.
Stage Master Explorer	Graphical tool to analyze effectiveness of compiler in parallelizing code.
Stage Master Debugger	Graphical debugging tool to observe cycle-by-cycle behavior of C code.

Figure 2.9 shows a screenshot of the Application manager. As seen in the figure, each source file is designated as “sw” or “hw” to indicate if they run on the host processor or on the FPGA. The files also have the “►” symbol just below the “sw”

or “hw” tag, indicating that they are targets for desktop simulation. Desktop simulation can simulate the entire design, including the interaction between software and hardware processes. While not cycle accurate, it is useful for functional verification of the design.



```
        // Now fill the firbuffer array with the first n values...
        for (tap = 1; tap < TAPS; tap++) {
#pragma CO UNROLL
            co_stream_read(filter_in, &nSample, sizeof(int32));
            IF_SIM(samplesread++;)
            firbuffer[tap-1] = nSample;
        }

        // Now we have an almost full buffer and can start processi
        // the streaming waveform samples.
        //

        // Read values from the stream
        while ( co_stream_read(filter_in, &nSample, sizeof(int32))
#pragma CO PIPELINE
            IF_SIM(samplesread++;)
            firbuffer[TAPS-1] = nSample;

            accum = 0;
            for (tap = 0; tap < TAPS; tap++) {
```

Figure 2.9 CoDeveloper Application Manager Screenshot

The *CoMonitor* Application Monitor allows the designer to observe the application as it executes by capturing messages, stream data values and other information. Code must be instrumented with special commands to indicate which variables and stream buffers need to be tracked by *CoMonitor*. Another useful tool is the *Stage Master Explorer* which is a graphical tool used to analyze how effectively the compiler was able to parallelize the C language statements. The tool also provides pipeline graphs, showing the estimated impact of various pipelining and compilation strategies.

CoBuilder generates synthesizable VHDL or Verilog only from processes in the files designated as “hw”. It also generates various hardware/software interface files, including a C runtime library and various hardware components that enable hardware/software communication on the XD1000. It creates a script that can be run to invoke Altera *Quartus* tools and synthesize the design. The software generation provides a software project directory that can be built and executed on the host Opteron processor.

The Impulse C programming model has two main types of elements: processes and communication objects. Processes are independently executing sections of code that interface with each other through communication objects. A software process is designated to run on a conventional processor and is constrained only by the limitations of the target processor. A hardware process is designated to run on an FPGA and is typically more constrained. It must be written using a somewhat narrowly-defined subset of C to meet the constraints of the Impulse *CoBuilder* FPGA compiler.

The following constraints are imposed on FPGA hardware processes [12]:

a. No recursion. A hardware process or function may not call itself, either directly or indirectly.
b. Limited use of function calls. A hardware process may call only the following types of C functions:
    - Impulse C API functions (named co_*)
    - Hardware primitive functions (using `#pragma CO PRIMITIVE`)
    - External HDL functions
c. Pointers must be resolvable at compile time
d. Limited support for C structs
e. No support for unions in C

Impulse C is designed for dataflow-oriented applications, but is also flexible enough to support alternate programming models including the use of shared memory as a communication mechanism. The programming model that is selected will depend on the requirements of the application and on the architectural constraints of the selected programmable platform target. A shared memory model is the preferred communication method between the host and FPGA on the XD1000 when using the Impulse C XD1000 PSP. Shared memory is implemented in the 4 MB SRAM found on the XD1000 module, and provides significantly better performance than streaming data between the CPU and FPGA.

In a software or hardware process, predefined Impulse C functions that perform inter-process communication may be referenced. These functions operate on communication objects to share data among processes. For example, read/write operations on a shared memory can be performed by calling functions on co_memory objects within the processes. The Impulse CoBuilder compiler with XD1000 PSP generates synthesis compatible hardware descriptions (HDL code compatible with *Quartus* synthesis tools) for the Stratix II FPGA, as well as a set of communicating processes (in the form of C code compatible with the target cross-compiler) to be implemented on the Opteron processor.

Impulse C defines several mechanisms for communicating among processes, as shown in Table 2.4. Our FBP implementation on the XD1000 exclusively used shared memories to share data between the host and FPGA. We also employed signals for synchronization.

Table 2.4 Impulse C communication objects [12]

Communication Object	Function
Streams	Buffered, fixed-width data streams
Signals	One-to-one synchronization with optional data
Semaphores	One-to-many synchronization
Registers	Un-buffered data or control lines
Shared memories	Memory shared between hardware and software

Every Impulse C application has a configuration function which is used to define the overall architecture of the application. It includes the declaration of all processes and their communication. To use the communication mechanisms shown in figure 2.4, the programmer declares objects in the configuration function. Processes can take these objects as arguments and call functions on them to communicate with other processes. The processes and signals need to be statically configured at compile time (i.e. no new processes can be generated during the execution). The configuration function used in the FBP implementation is discussed in section 3.1.2.

The summary of the Impulse C design flow to target the XD1000 is shown in figure 2.10.

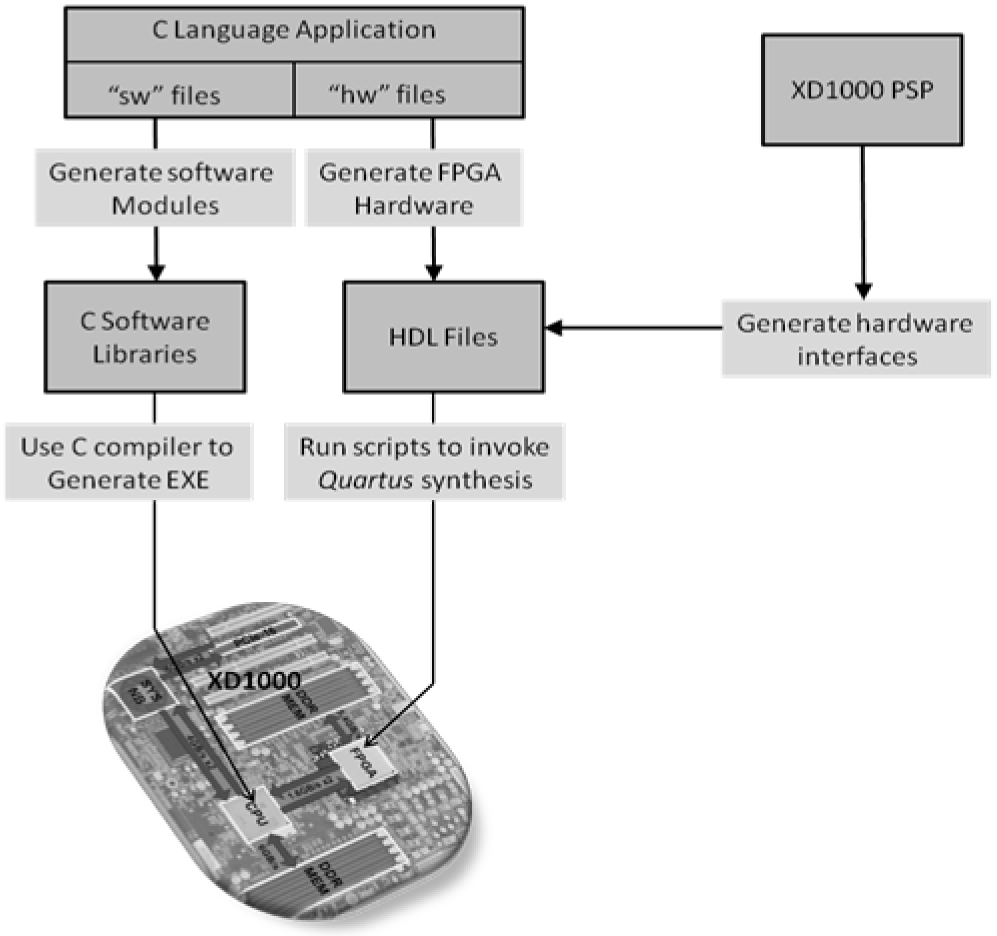


Figure 2.10 Design flow to target XD1000 with Impulse C

3. Design Considerations

Acceleration on an FPGA is achieved by exploiting parallelism in the application. Designers need to consider how the hardware resources will be used by the computing elements and how the data flows through the system. Further, when partitioning a design across a host and coprocessor, the designer has to define how the host communicates and hands off the computation to the coprocessor. There is a tradeoff between the time it takes to transfer the data required to perform a certain computation and the acceleration that can be achieved by doing so.

3.1. From C to Impulse C

This section describes how Impulse C was used to accelerate the FBP software by partitioning the design between the host and FPGA on the XD1000 development system. The backprojection was carried out on the FPGA while the filtering, file I/O and other miscellaneous operations were performed on the CPU.

3.1.1. Filtered Backprojection Software Benchmark

The FBP software source code [20] is a C implementation of FBP compatible with parallel beam CT systems. It handles the file IO, and supports a variety of common filters. It uses 32-bit floating-point operations to perform computations. A floating-point implementation has sufficient precision to be considered near-perfect reconstruction in the absence of noise, and can be used as the standard to verify the accuracy of the hardware implementation. Table 3.1 summarizes the features of the FBP software source code.

Table 3.1 FBP source code features

FBP Software Benchmark Features	
Filters Supported	Ramp, Hamming, Hanning
Beam Geometry supported	Parallel beam
No. Of projections supported	Adapts to input Sinogram
Image size produced	User defined at run time
Data-path Width	32-bit floating-point
Interpolation Type	Bi-linear

We profiled the software source code by running it on the 2.2 GHz Opteron-248 processor in the XD1000 system. Data from profiling the source code (table 3.2) shows that the backprojection is the slowest step, accounting for 70-80% of the execution time. The filter step accounts for most of the remaining execution time with the file I/O and miscellaneous operations account for less than 0.6%. Both the filter and backprojection operations are good targets for FPGA acceleration and could both be run on the FPGA, thereby utilizing the CPU for just the file I/O and miscellaneous operations.

Table 3.2 Software source code execution time for the reconstruction of a 512x512 image

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

Efficient structures to implement high performance filters on FPGAs are well understood and are often provided by FPGA vendors as reference designs. Backprojection, on the other hand, brings up some interesting issues related to data flow, memory bandwidth and access patterns, making it a better choice for studying the performance of the Impulse C hardware generation process. As mentioned in section 2.1.3, there is additional motivation to focus on the acceleration of backprojection because of its utility in iterative reconstruction techniques. Based on these factors, we decided to offload only the backprojection to the FPGA coprocessor and have the rest of the application run on the Opteron processor. This required us to build an FPGA version of the backprojector, which we undertook with Impulse C. It also required the creation of Impulse C software processes to interface the computations running on the CPU with processes running on the FPGA.

It is worth clarifying why backprojection takes only 2.5 times longer when the number of projections is doubled (table 3.2), even though it is an O(n^3) operation. The reason for this is that we reconstruct the same number of image pixels (512x512) in each case. As we move from 512 to 1024 projections, we increase the number of detector channels per projection from 512 to 1024 but this does not affect the number of computations performed as we are reconstructing pixel-by-pixel. In the base case (512 projections) shown in table 3.2 we perform (512x512x512) computations; in the other we perform (512x512x1024) operations. If we were to reconstruct a 1024x1024 image from 512 and 1024 projections, the backprojection stage would take 4 and 8 times longer respectively.

One of the important advantages of C-to-FPGA tools is the ability to reuse existing C codebases. The non-critical steps that are not targets for acceleration continue to run unmodified on the host. Only the functions that are targeted for acceleration need to be modified or rewritten to achieve an efficient implementation. We will use the term “refactoring” to refer to the process of modifying and reordering statements in a function in order to obtain an efficient hardware implementation. The next section describes the process of porting the above benchmark to the Impulse C framework. After that we discuss steps to refactor the backprojection function.

3.1.2. Porting to Impulse C

The first step in porting a C application to Impulse C is to compile and run the existing code within the Impulse C framework. This is done by importing the code base and designating all source files as targets for desktop simulation. Desktop simulation simply compiles the application using a standard C compiler (gcc by default or a standard third-party compiler of the user's choice) into a native Windows executable that may be run directly or executed with a standard third-party debugger. Any ANSI compliant C code base should work in this framework. Once the existing code is in the framework, it is easy to add Impulse C processes to accelerate the desired functions.

The next step is to define the system level architecture (using a configuration function - see figure 3.2) and entry points for Impulse C processes. As mentioned earlier, based on the profiling data, we decided to implement the backprojection function as a hardware process on the FPGA. A software process is required on the host to interface with the hardware. The natural entry point for these processes is the function in the existing code that performs backprojection. That function can simply be replaced by a call to the Impulse C *run-function*. The *run-function* executes the software and hardware processes in the Impulse C application.

The software process serves as the bridge between the functions in the application and the computation executing on the FPGA. To the software process, we can pass both data structures from the C application and also the communication objects that enable CPU – FPGA interaction. Further, any global data structures or functions in the application are visible inside the software process. Once the hardware process has finished processing the data, the software process reads the results from the FPGA and presents it back to functions in the application that perform post processing and file output.

In the FPGA there are opportunities to overlap execution with memory accesses. With this in mind, the backprojector design is divided into two hardware processes. The “Memory Engine” performs memory accesses. The other, named “Processing Engine” performs the computations. This serves as a logical partitioning between the stages of execution.

Every Impulse C application running on a platform has one or more software processes running on the host and one or more hardware processes running on the FPGA. The system architecture that governs how they communicate among each other and synchronize their operations is largely determined by the architecture of the platform. It is also governed by the relative efficiency of different communication links on the platform. As mentioned in chapter 2, the most efficient dataflow model that Impulse C implements on the XD1000 platform is shared memory through the on-board SRAM. Based on this, we decided to go with the shared memory model for our application. Figure 3.1 shows the data transfer rates to the SRAM.

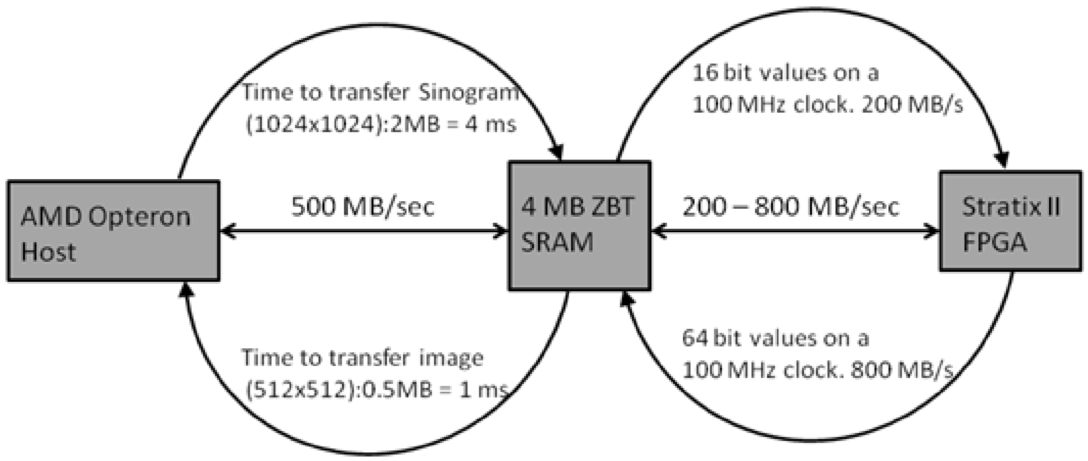
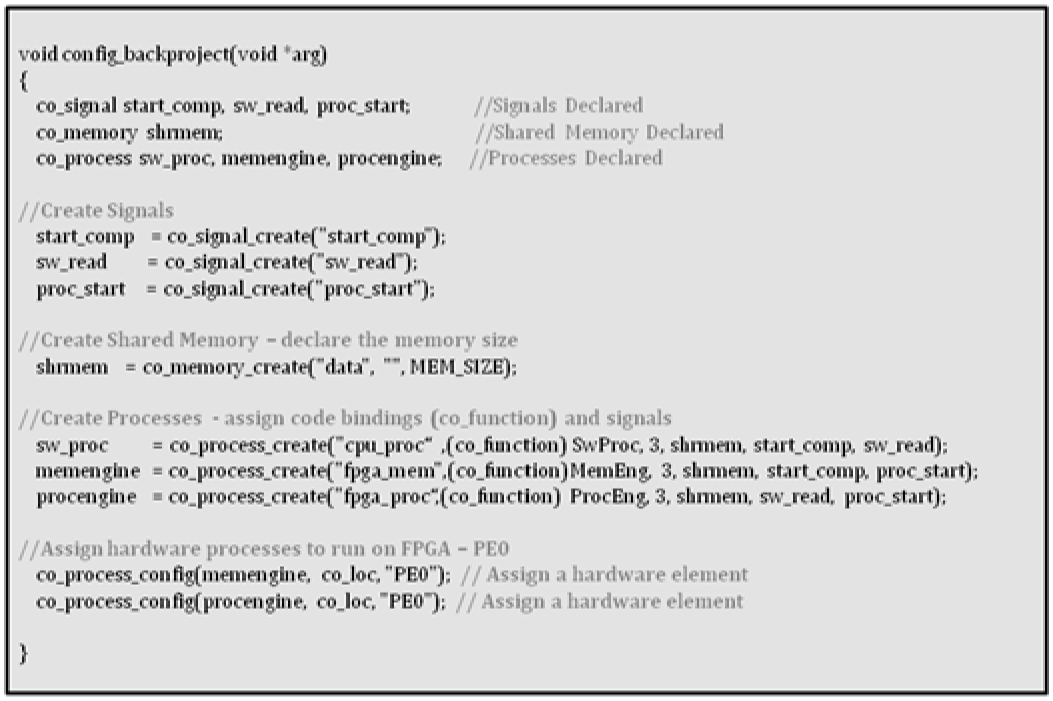


Figure 3.1: Data rates to Shared Memory

Comparing the transfer speed in figure 3.1 to the execution times in table 3.2, we see that the time it takes to move data to and from the shared memory is negligible compared to the time taken carry out backprojection in software.

Therefore, any acceleration that can be achieved by the FPGA implementation should speed up the overall execution. This is in contrast to situations where data transfer is slower than the execution time. In such cases, the decision to move the computation off-chip should be revisited.

Figure 3.2 shows the configuration function used to declare the various processes, signals, and shared memories in the design. As mentioned in section 2.3.1, the configuration function defines the architecture of the application.



```
void config_backproject(void *arg)
{
   co_signal start_comp, sw_read, proc_start;          //Signals Declared
   co_memory shrmem;                                    //Shared Memory Declared
   co_process sw_proc, memengine, procengine;          //Processes Declared

//Create Signals
   start_comp   = co_signal_create("start_comp");
   sw_read      = co_signal_create("sw_read");
   proc_start   = co_signal_create("proc_start");

//Create Shared Memory - declare the memory size
   shrmem   = co_memory_create("data", "", MEM_SIZE);

//Create Processes - assign code bindings (co_function) and signals
   sw_proc      = co_process_create("cpu_proc" ,(co_function) SwProc, 3, shrmem, start_comp, sw_read);
   memengine  = co_process_create("fpga_mem",(co_function)MemEng, 3, shrmem, start_comp, proc_start);
   procengine  = co_process_create("fpga_proc",(co_function) ProcEng, 3, shrmem, sw_read, proc_start);

//Assign hardware processes to run on FPGA - PE0
   co_process_config(memengine, co_loc, "PE0"); // Assign a hardware element
   co_process_config(procengine, co_loc, "PE0"); // Assign a hardware element

}
```

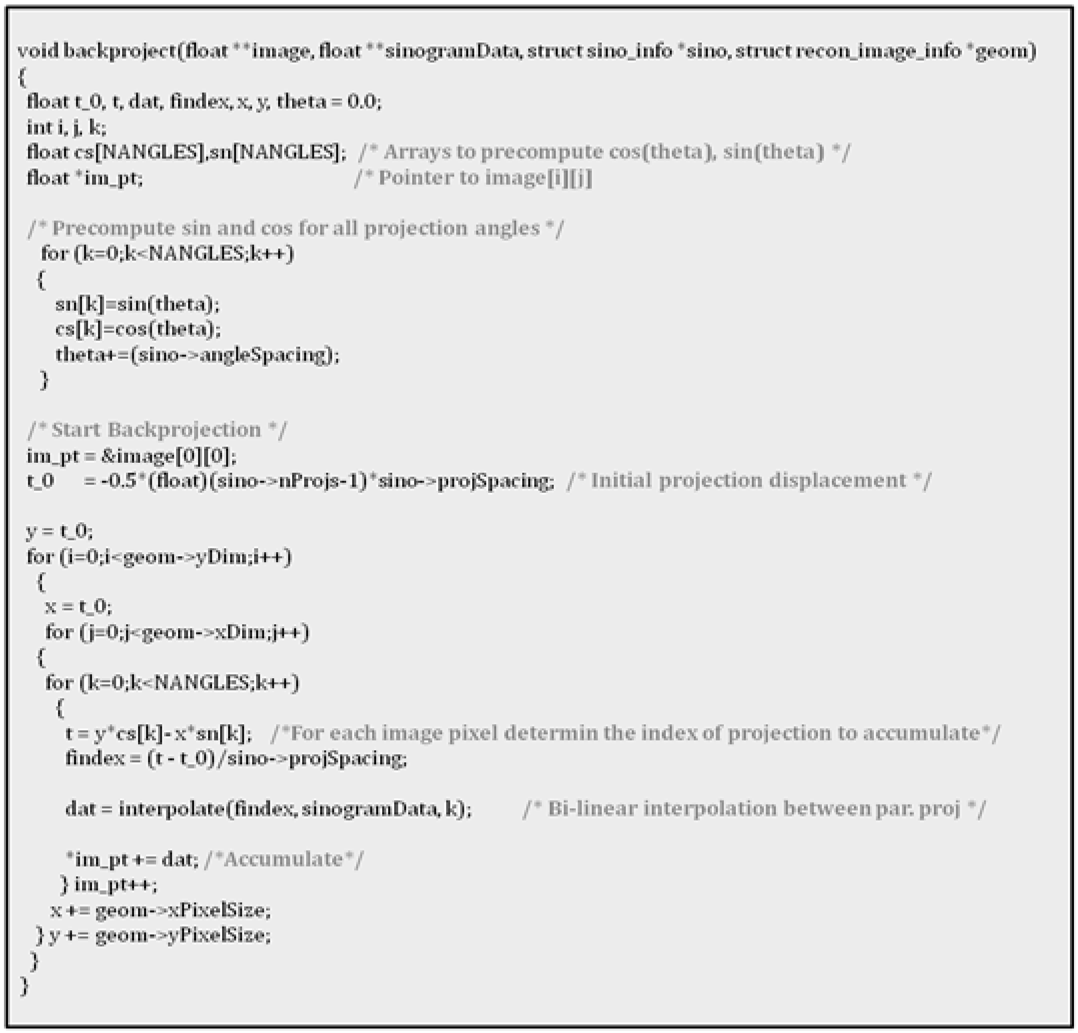
Figure 3.2 Configuration Function code snippet

Creating efficient code bindings for the hardware processes is crucial. It is this code that the compiler maps to HDL and is synthesized to run on the FPGA; its efficacy largely determines the achieved acceleration. The remainder of this chapter discusses issues pertinent to this task.

It is useful to clarify the following terms. In this thesis, we refer to the architecture of the entire system, including the CPU-SRAM-FPGA communication interfaces, by the term “system architecture”, and the architecture of the hardware design implemented on the FPGA as “design architecture”.

3.2. Refactoring Code for Hardware Generation

Figure 3.3 shows a code snippet (from the software source code) that performs backprojection. Pre-computed sine and cosine values are stored in arrays. The triple nested loop structure traverses the image pixel by pixel. The inner loop cycles through the angles and determines the index of the projections that contributes to the pixel for each angle. The values are then interpolated and added to the pixel.



```
void backproject(float **image, float **sinogramData, struct sino_info *sino, struct recon_image_info *geom)
{
 float t_0, t, dat, findex, x, y, theta = 0.0;
 int i, j, k;
 float cs[NANGLES],sn[NANGLES];  /* Arrays to precompute cos(theta), sin(theta) */
 float *im_pt;                   /* Pointer to image[i][j]

 /* Precompute sin and cos for all projection angles */
   for (k=0;k<NANGLES;k++)
  {
     sn[k]=sin(theta);
     cs[k]=cos(theta);
     theta+=(sino->angleSpacing);
  }

 /* Start Backprojection */
 im_pt = &image[0][0];
 t_0    = -0.5*(float)(sino->nProjs-1)*sino->projSpacing;  /* Initial projection displacement */

 y = t_0;
 for (i=0;i<geom->yDim;i++)
  {
   x = t_0;
   for (j=0;j<geom->xDim;j++)
  {
   for (k=0;k<NANGLES;k++)
    {
     t = y*cs[k]- x*sn[k];   /*For each image pixel determin the index of projection to accumulate*/
     findex = (t - t_0)/sino->projSpacing;

     dat = interpolate(findex, sinogramData, k);        /* Bi-linear interpolation between par. proj */

     *im_pt += dat; /*Accumulate*/
    } im_pt++;
   x += geom->xPixelSize;
  } y += geom->yPixelSize;
 }
}
```

Figure 3.3 Backprojection code from SW benchmark

C code written to run on a processor does not automatically convert into an efficient FPGA implementation if passed without modification through the Impulse C hardware generation process. When hardware was generated from the code in figure 3.3 (after adding support for shared memory to enable communication), the resulting design was significantly slower than simply executing the code on the host processor. The code that eventually produces a good FPGA version must use the "special" resources (Distributed memories, DSP blocks etc) that enable parallelism on the FPGA. Obviously, if there is little or no parallelism, the FPGA cannot be faster at the task.

Loops in the C code are natural targets for parallel computation. A loop repeatedly performs the same computation on different pieces of data. If the hardware performing the loop computation is replicated as many times as the loop runs, the entire loop could theoretically be executed in the time it takes to perform a single iteration. This process is called *loop unrolling*. There might be dependencies that prevent parallel execution and care must be taken to avoid or work around these. Memory accesses inside the loop need to be streamlined because operations running in parallel cannot access the same memory at the same time. The memory either needs to broken up or replicated.

Loop unrolling in the context of hardware generation is not the same as the traditional software engineering *loop unroll* optimization where loops are unrolled to avoid loop overhead and gain limited execution overlap. In software execution there is no replication of the computing resources, whereas in hardware generation the data-path is replicated. In the hardware generation process, a loop with *n* iterations requires *n* times the resources when unrolled, but can be up to *n* times faster.

Loop unrolling is performed in Impulse C using the `CO UNROLL` pragma. Selecting the “Scalarize array variables” option during hardware generation replaces arrays (memories) by registers so that they can be accessed simultaneously. This works in simple cases. However, we might need the memories to be broken up into smaller memories instead of being converted into registers. In such cases, the `CO UNROLL` pragma may not be able to automatically generate the segmented memories required. Further, there may not be enough resources to completely unroll the loop. In such situations, we could partially unroll the loops. This might complicate the logic and memory structure. If there is a nested loop structure that needs to be partially unrolled with memory segmentation, `CO UNROLL` cannot be efficiently used. In such cases, we still have the option of manually defining separate smaller memories and manually performing a partial unroll of the loops. In our implementation the triple nested for-loop structure seen in figure 3.3 was partially unrolled by hand. This is described in more detail in the context of the design architecture in section 3.3.2.

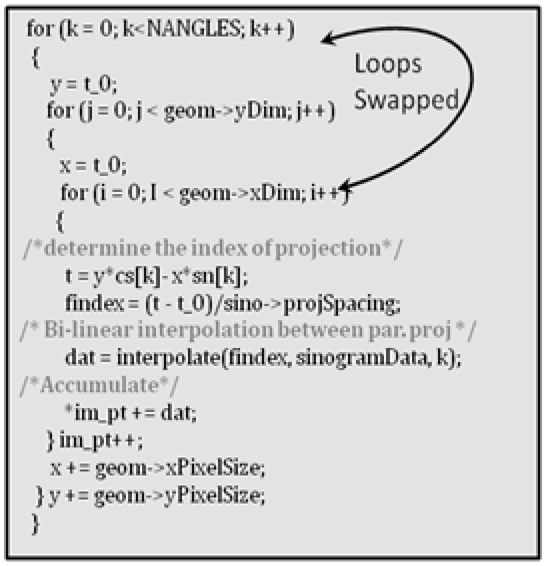
Refactoring C code to implement parallelism generally involves unrolling loops and breaking up the memories (arrays in C code) such that they do not block each other. Another important optimization target is the inner loop computation itself. It is this computation that gets replicated in the process of unrolling, and so refactoring the code to produce the most efficient version of it is desirable. For example, in a system that is 128-way parallel, one adder saved in the inner loop computation saves 128 adders in the overall design. The inner loop computation can also be pipelined to increase the number of computations running in parallel. This is different from the parallelism execution achieved through loop unrolling. Unrolling creates identical copies of the data path that works simultaneously on independent data. On the other hand, pipelining allows different stages of the same data path to work simultaneously. Thus, pipelining and unrolling used together can provide higher speedup than using one or the other. The added advantage of pipelining the data path is that it increases the maximum frequency of operation of the design by breaking up the critical path.

There are performance gains to be had if computations are performed in fixed point instead of floating point. Going to specific bit-widths for each operand can save resources. Other refactoring steps might include performing computations differently to save scarce resources. For example, if multipliers are a scarce resource in the system, any opportunities to save multipliers must be explored.

3.2.1. Index Generation through Offsets

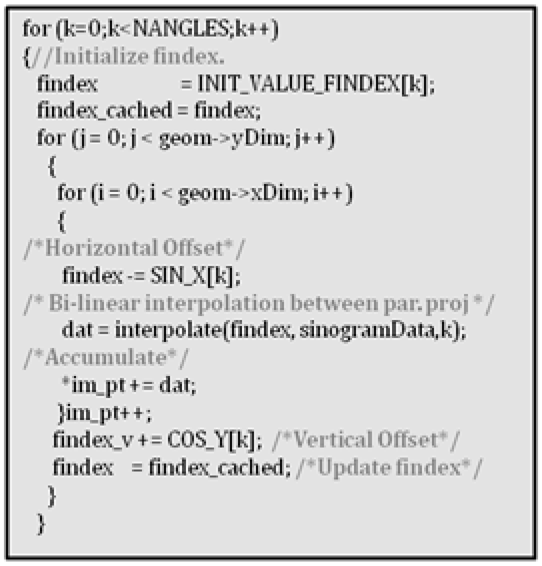
Consider the code in figure 3.3. In the inner loop, the index of the projection (findex) that needs to be accumulated is determined. The fractional part of findex is used to perform Bi-linear interpolation, which requires 2 multipliers and an adder. In the code, determining findex requires two multipliers to find t and one constant denominator division operation to find findex (which can be implemented as a multiply). These three multipliers are used to generate each findex value. If we have *n* parallel computations, *3n* multipliers get utilized in just determining the index of the projection that we need to interpolate. To avoid using these multipliers in computing findex, we generate findex values for each pixel through stored offsets instead of computing it.

One of the things to notice about the *findex* computations in figure 3.3 is that the variables *x*, and *y* are incremented by fixed amounts in the loop structure. Also, *t_0* and *sino->projSpacing* are constant values. Only the sine and cosine values change with the changing angles in the inner loop. However, if we reorder the loops such that the outermost loop traverses the projections, and the inner loops traverse the pixels, the angles remain constant in the inner loop. This reordering of the loops in figure 3.2 is shown in figure 3.4. Now, *findex* changes by a fixed amount in successive iterations of the inner loop.



```
for (k = 0; k<NANGLES; k++)
 {
   y = t_0;
  for (j = 0; j < geom->yDim; j++)
  {
    x = t_0;
    for (i = 0; I < geom->xDim; i++)
   {
/*determine the index of projection*/
      t = y*cs[k]- x*sn[k];
      findex = (t - t_0)/sino->projSpacing;
/* Bi-linear interpolation between par. proj */
      dat = interpolate(findex, sinogramData, k);
/*Accumulate*/
      *im_pt += dat;
   } im_pt++;
   x += geom->xPixelSize;
 } y += geom->yPixelSize;
 }
```

Figure 3.4 Reordering loops from figure 3.2



```
for (k=0;k<NANGLES;k++)
{//Initialize findex.
  findex          = INIT_VALUE_FINDEX[k];
  findex_cached = findex;
  for (j = 0; j < geom->yDim; j++)
   {
    for (i = 0; i < geom->xDim; i++)
    {
/*Horizontal Offset*/
     findex -= SIN_X[k];
/* Bi-linear interpolation between par. proj */
     dat = interpolate(findex, sinogramData,k);
/*Accumulate*/
     *im_pt += dat;
    }im_pt++;
   findex_v += COS_Y[k];  /*Vertical Offset*/
   findex   = findex_cached; /*Update findex*/
   }
 }
```

Figure 3.5 Offset-based index generation

Instead of computing *findex* from scratch in each iteration, we can compute it by storing the initial value of *findex* and the fixed offsets for each angle. This provides a tradeoff between memory required to store the offsets and multipliers needed for findex generation. In our system multipliers in the form of DSP blocks were a scarcer resource than the small memories that would be needed to store the offsets, and hence we implemented the offset method of computing findex. Figure 3.5 shows the code that implements offset-based computing of findex. It is identical to the computation in figure 3.4. The initial value of findex for each angle is stored in the array *INIT_VALUE_FINDEX*. The arrays *SIN_X* and *COS_Y* store the horizontal and vertical offsets respectively. *findex* is computed by making appropriate offsets from the initial value.

To gain some intuition for why the offset method works, consider what the inner loops do. For a given angle, the loops walk through the image pixels, figuring out where on the detector the x-ray through that pixel strikes. Figure 3.6 shows how for a given angle, the impact point on the detector varies for different pixels on the image. As we move horizontally across the image through a distance *xPixelSize*, the x-ray through the pixel strikes at a point *h* away on the detector bar. A vertical movement through a distance *yPixelSize* causes the x-ray to strike *v* away. *h* and *v* are related to xPixelSize and yPixelSize respectively through the simple trigonometric relationship

*h = xPixelSize*cos(theta)*,

v = *yPixelSize*sin(theta)*

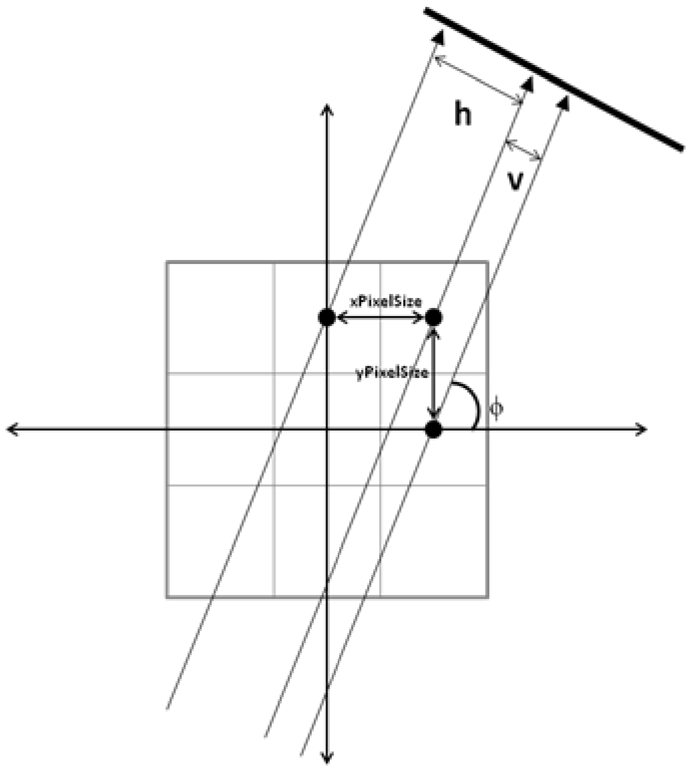


Figure 3.6 Detector offsets when walking through image

Thus, for each projection angle, if we know the position that the x-ray through the first pixel strikes the detector, and the value of *h*and *v* for that angle, we can determine where x-rays through neighboring pixels strike the detector. These positions are the values that index into projection data array, and are known as *findex* in the code snippets. For 512 projections we need to store 512 initial values, 512 *v* values and 512 *h* values. The memory required to store these values is less than 1% the total memory required by the design while the number of DSPs saved by this method is 50% of the total DSPs required by the design, making it a very attractive tradeoff. Leeser et al. [3] use this index generation mechanism in their FPGA implementation.

3.2.2. Fixed Point Representation

The advantage of floating point numbers over fixed point is the wide range of values they can represent. This advantage comes at the cost of complex logic required to implement floating point arithmetic. In applications where the range of values is not large, fixed point representation can be an equally accurate alternative to floating point. Standard C has no native data type to represent fixed point fractional data. One could implement fractional fixed point using *ints* by manually keeping track of where the decimal point is, but this is rarely done. Hence, in many C applications, floating point representations are used even when the data does not have the dynamic range to justify it.

The sinogram values that the FPGA operates on can be pre scaled to the range -1 to +1. In the backprojection stage, the sinogram values only get accumulated. Because we know the number of values that are added, we can determine *a priori* exactly what the theoretical maximum value of the accumulation will be. Based on this we can design the data path with fixed point arithmetic. Leeser et al. [3] performed a detailed study of the effects of quantization introduced through fixed point implementation of backprojection. They demonstrated that a 9-bit representation of the sinogram and 3-bit for the interpolation factor provided a good tradeoff between reconstruction quality and efficient implementation. In our implementation, we employed a more conservative approach using 16-bits to represent the sinogram and 16 bits for the interpolation factor. The internal arithmetic gives rise to 32 bit results (due to multiplying 16 bit numbers), and is scaled back to 16 bits by discarding the less significant bits.

We instrumented our floating point code to track the range and precision at different points of the data path. Table 3.3 shows bit width that was chosen for different variables in the data path. A note about the notation (*SsI.F)* used. This implies there are *S* (1 or 0) sign bits, *I* integer bits and *F* fraction bits. The fixed point format for the sinogram data is 1s1.14. This means that there is one sign bit, one integer bit and 14 fraction bits. The variable *floor* is used to generate memory indices from *findex*. We selected different bit depths and visually assessed the influence on reconstructed images. We choose wide enough bit-widths to generate images with no discernable visual difference compared to the floating point implementation.

Table 3.3: Fixed point format employed

Variable	Format
Sinogram data	1s1.14
Interpolation factor	1s7.8
Findex	1s10.8
Floor	0s11.0
Offsets	1s10.8
Image data	1s8.7

By converting the sinogram data from 32 bit floating point to 16 bit fixed point, we need only half the number of DSPs to perform the interpolation step. This allows us to use the saved DSPs to accommodate more parallel operations, thus resulting in faster execution. The Stratix DSP block can multiply 16 bit fixed point numbers every cycle with no latency, whereas, floating point multiplication has 11 cycles of latency. Further, the fact that the fixed point representation of the sinogram and image are only half the size of their floating point counterparts makes the process of transferring them between the CPU and FPGA twice as fast. The result is that converting the design to fixed point results in at least a 2x speedup.

Impulse C supports fixed point via data types to represent data and macros to perform arithmetic on them. The data types are co_int*x*/co_uint*x*, where *x* represents the number of bits and int/uint represents signed and unsigned numbers respectively. Impulse C also provides an efficient way to extract and insert bits into any position of a 32-bit number through the `co_bit_extract` and `co_bit_insert` functions. The macros only work on data types of certain bit-widths. In our implementation we extensively used the data types but preferred not to use the macros, choosing instead to manually track the number of fraction bits along the data path. Bits can be dropped using the standard C right shit operator “>>”. In Impulse C, when making assignments of unequal bit-widths, bits of the same weight are preserved (i.e. if we assign a 32-bit variable to a 16 bit variable, the bottom 16 bits of the 32 bit number are copied to the 16 bit number). Using a combination of shifts and assignments, we can manually maintain data in the appropriate bit-width. Figure 3.8 shows the various fixed point operations used inside the computing loop.

The integer portion of the variable *findex* is the array index, and the fractional portion of it is the interpolation factor. All legal values of *findex* are positive. We however keep the sign bit because *findex* does come out negative for certain pixels that lie outside the field of view. In those cases, *floor* is an undefined index into the array (memory address). The image pixel is updated with some unknown value. This is not a problem because the pixels that lie outside the field of view are masked in a post processing step performed on the CPU. The compute pipeline (described next) is more efficient if the computation is performed on all image pixels; adding logic to the pipeline to avoid these corner cases reduces the achieved rate.

Figure 3.7 shows the field of view in the image. Only pixels lying in the inscribed circle lie in the field of view. This is because during reconstruction we assume that both the detector and the image are centered at the origin. When we rotate the detector through various angles, a circle is described. The points lying inside the circle are the valid pixels.

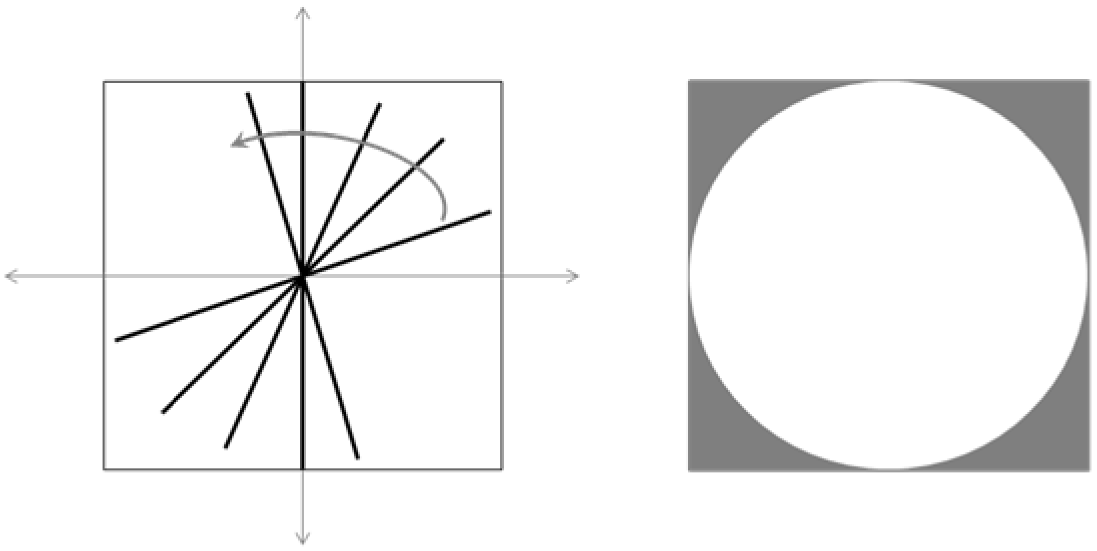


Figure 3.7 Detectors describe a circle in the image space (left).
Valid pixels in the image (right).

3.2.3. Pipelined Data Path

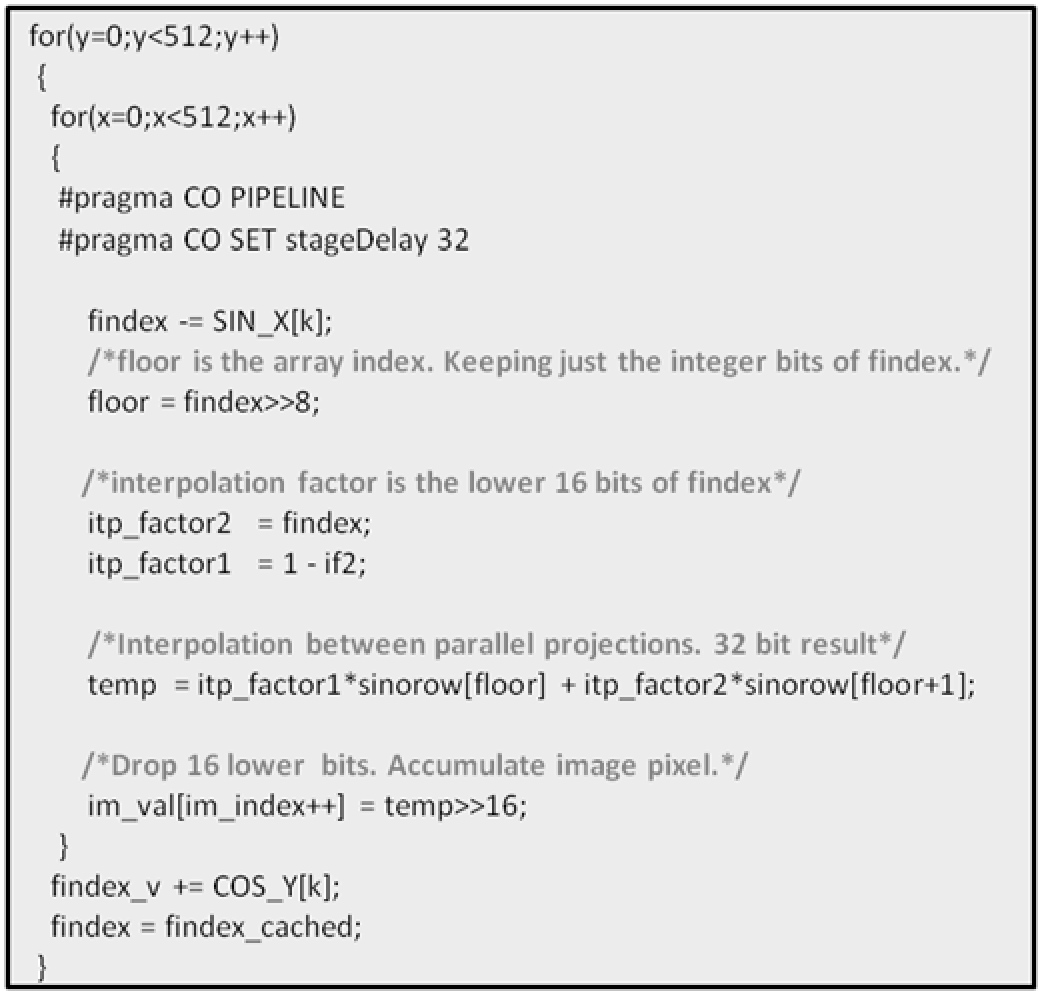
We have discussed two optimizations of the inner loop computation (fixed point conversion and incremental computation of *findex*) and the benefit they provide. The next step is to pipeline the computation. Pipelining is the process of breaking up a complex computation into multiple stages so that the individual stages can run simultaneously. If each stage of the pipeline executes in 1 clock cycle and there are *n* pipeline stages, the first result from the pipeline is produced after n cycles. The time it takes for the first result to appear is called the latency of the pipeline. Every subsequent value from the pipeline is available in successive clock cycles (if there are no dependencies in the pipeline). For computations that are carried out many times in succession, the latency of the pipeline is negligible when compared to the time taken to produce all the results. Pipelining is accomplished in Impulse C using the `CO Pipeline` pragma.

The Impulse C *Stage Master Explorer* tool is used to analyze the performance of pipelines in the design. It uses the following terms in the context of pipelining:

- *Latency:* The number of cycles for inputs to reach the output of the pipeline. It is equal to the number of stages in the pipeline.
- *Stage Delay:* The combinational delay or levels of logic in a single pipeline stage.
- *Rate:* The number of cycles after which the pipeline accepts new inputs. A rate of 1 means the pipeline accepts a new input every cycle. A Rate of *r* implies the pipeline accepts an input ever *r* cycles.

The Impulse C compiler groups all statements that can execute in parallel into a stage. All statements within a stage execute in one cycle. If there are many statements executing in a single stage, it results in a higher stage delay and lowers the maximum frequency of operation. If high frequency operation is desired, the stage delay of the pipeline can be constrained to equal a certain amount by using the `CO SET stageDelay` pragma. The compiler now only groups together those computations that can be performed within the specified stage delay. Another way to control how the stages are generated is using the `co_par_break` statement. This statement can be used to manually control which statements are grouped into a pipeline stage.

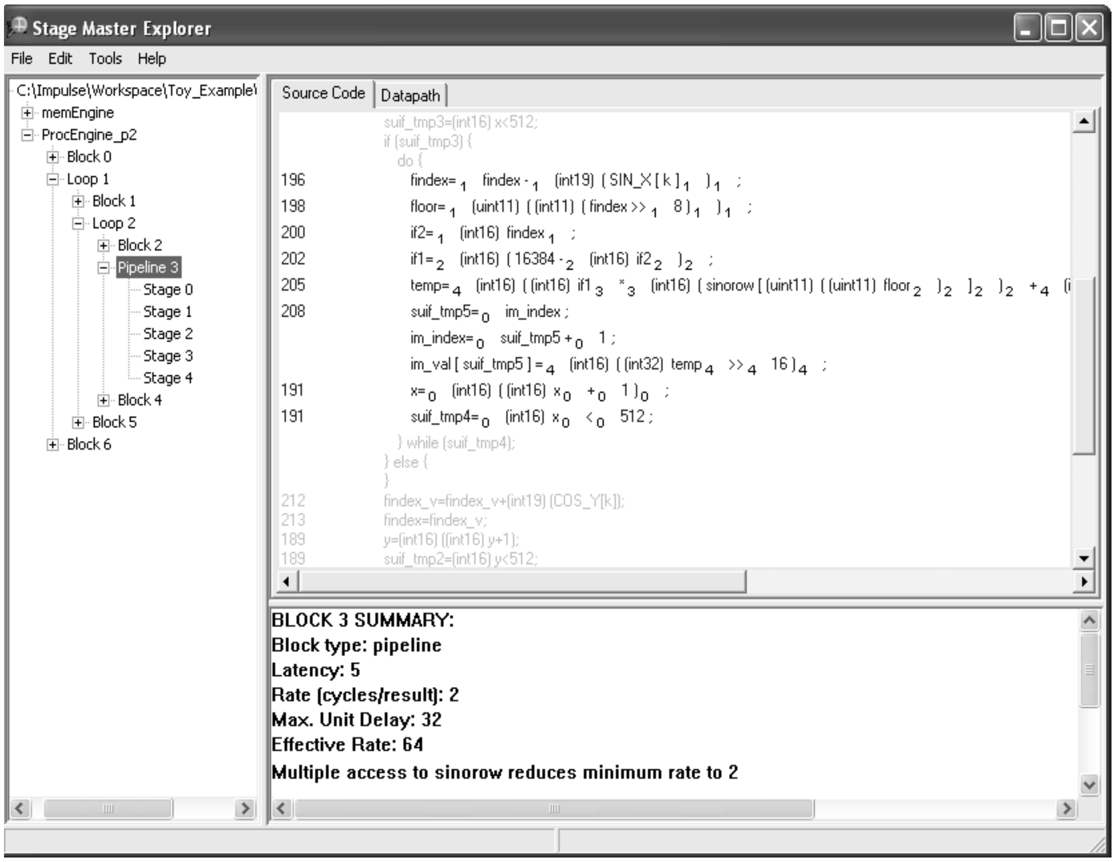
Figure 3.8 shows the backprojection loop with the pipeline pragma. This loop finds the contribution to every pixel in the image from one projection. The function call to `interpolate()` seen in figure 3.3-3.5, has been replaced by the actual interpolation code because Impulse C hardware generation has limited support for function calls. Further, it is easier to exercise control over the pipeline if all operations are visible in the loop. Interpolation requires 2 values to be fetched from memory, 2 multiplies and 1 add.



```
for(y=0;y<512;y++)
 {
  for(x=0;x<512;x++)
  {
   #pragma CO PIPELINE
   #pragma CO SET stageDelay 32

      findex -= SIN_X[k];
      /*floor is the array index. Keeping just the integer bits of findex.*/
      floor = findex>>8;

     /*interpolation factor is the lower 16 bits of findex*/
      itp_factor2   = findex;
      itp_factor1   = 1 - if2;

      /*Interpolation between parallel projections. 32 bit result*/
      temp  = itp_factor1*sinorow[floor]  + itp_factor2*sinorow[floor+1];

     /*Drop 16 lower  bits. Accumulate image pixel.*/
      im_val[im_index++]  = temp>>16;
   }
  findex_v += COS_Y[k];
  findex = findex_cached;
 }
```

Figure 3.8 Pipelining the backprojection code

Impulse C *Stage Master Explorer* provides graphical views of the pipeline generated from the code. The block summary displays the latency, the maximum stage delay, and the achieved rate of the pipeline. Stage Master has a source code view showing which statements in the code were grouped into a stage. The statements are annotated with numbers to show to which stage they belong. Figure 3.9 shows the source code view generated by Stage Master from the code in figure 3.8.



```
                suif_tmp3=(int16) x<512;
                if (suif_tmp3) {
                  do {
196                 findex= 1  findex - 1  (int19) ( SIN_X [ k ] 1  ) 1  ;
198                 floor= 1  (uint11) ( (int11) ( findex >> 1  8 ) 1  ) 1  ;
200                 if2= 1  (int16) findex 1  ;
202                 if1= 2  (int16) ( 16384 - 2  (int16) if2 2  ) 2  ;
205                 temp= 4  (int16) ( (int16) if1 3  * 3  (int16) ( sinorow [ (uint11) ( (uint11) floor 2  ) 2  ] 2  ) 2  + 4  (i
208                 suif_tmp5= 0  im_index ;
                    im_index= 0  suif_tmp5 + 0  1 ;
                    im_val [ suif_tmp5 ] = 4  (int16) ( (int32) temp 4  >> 4  16 ) 4  ;
191                 x= 0  (int16) ( (int16) x 0  + 0  1 ) 0  ;
191                 suif_tmp4= 0  (int16) x 0  < 0  512 ;
                  } while (suif_tmp4);
                } else {
                }
212             findex_v=findex_v+(int19) (COS_Y[k]);
213             findex=findex_v;
189             y=(int16) ((int16) y+1);
189             suif_tmp2=(int16) y<512;
```

BLOCK 3 SUMMARY:
Block type: pipeline
Latency: 5
Rate (cycles/result): 2
Max. Unit Delay: 32
Effective Rate: 64
Multiple access to sinorow reduces minimum rate to 2

Figure 3.9 Impulse C Stage Master Explorer

The inner loop operation executes 512^2 times, and so the latency of the pipeline is irrelevant. Our goal is to achieve a rate of 1 while keeping the stage delay as low as possible to maximize the frequency of operation. One of the restrictions imposed by the Impulse C platform support package for the XD1000 was that the user logic had to be clocked at 100MHz. This made it even more crucial to keep the stage delay as low as possible. The stage delay is calculated as the sum of the unit delays of the individual operations in a stage. Bitwise operations, such as shifts, have a stage delay of 1. Arithmetic operations have a stage delay equal to the bit-width of the widest operand. In our pipeline, the interpolation step has the longest stage delay of 32 because a 32-bit result is generated in that step. Considering that we have a delay of *at least* 32, the idea is to try to divide the stages such that each stage has a delay of *no more than* 32. Hence we use the CO SET pragma to constrain the stage delay to 32, as seen in figure 3.6.

Once the stage delay is determined, we try to achieve a rate of 1 so that the pipeline is accepting inputs and producing results every cycle. However, the pipeline could only achieve a rate of 2 as seen in figure 3.7. Operations within a stage do not have dependencies and so do not affect the rate. The message "Multiple access to sinorow reduces minimum rate to 2" in Stage Master Explorer provides insight about the issue affecting the rate. In the interpolation step, we need to read 2 values form the array *sinorow* in the same cycle. This makes the rate 2 because only one value can be read from the memory at a time.

There are some ways to work around this and achieve the desired rate of 1. The first is to maintain two memories with the same values - *sinorow1* and *sinorow2*. These two distinct memories can be accessed in the same cycle. The disadvantage of this method is that it doubles the memory required to implement the pipeline. As mentioned earlier, when the loop is unrolled, any increase in the resources required to perform the inner loop computation implies a very large increase in the overall design. Memories were a resource bottleneck in our design and so this was not a viable solution. A more efficient method is to implement *sinorow* as a true dual port memory. This enables reading two values from the memory at the same time. The Stratix II M4k RAM block is capable of implementing true dual port. In Impulse C, an array can be explicitly implemented using a particular memory type using the `co_array_config()` function. The declaration of the array is followed by the configuration statement designating it as true dual port

```
co_int16 sinorow[SINO_SIZE];
co_array_config(sinorow, co_kind, “dualsync”);
```

However, the Impulse C compiler does not allow memories accessed in more than one process to be explicitly configured. All operations with a particular memory (that needs explicit configuration) should be performed within the process that declares it. However, in our application we chose to keep the memory read operations in a separate process and the compute operations in another. Though we had initially conceived this partitioning simply as a logical way to break up the tasks to be performed, we found that it had implications for the ability of the synthesized design to meet the timing constraints. When we attempted to perform the memory read operations in the same process as the compute, the design would fail timing. We believe that the partitioning might have helped improve the placement of the design, thus enabling it to meet timing. Hence we decided to keep the memories in the default configuration and maintain the partitioning.

3.3. Design Architecture

The inner loop of the computation represents the task carried out by a single complete data path in the design. The optimization strategy is to create the most efficient hardware implementation of the inner loop possible and then replicate it as many times as the resources on the chip will allow. The previous section discussed all the optimization made to the inner loop. We now discuss replication of the data path and then present a system-level view of how the entire design operates.

3.3.1. Parallel Design Architecture

The loops in figure 3.8 find the contribution to every image pixel from one projection. If we replicate the computation inside the inner loop, we can simultaneously find the contribution to every image pixel from multiple projections. Based on resource availability, we decided to build a 128-way parallel system.

3.3.1.1. Loop Unrolling

The process of replicating the inner loop computation was achieved by manually unrolling the computation by duplicating each statement inside the compute loop 128 times with different variable names. Some of the statements could have been more elegantly unrolled using the `CO Unroll` pragma. However, the memory access patterns inside the inner loop require the compiler to infer that memories need to be broken into smaller distributed units, which the current version of the Impulse C tools is not capable of doing. This computation was simple enough that the manual unrolling process was not too painstaking, and was achieved rather easily using a standard text editor.

With 128 operations running in parallel the last stage of the pipeline generates 128 values which need to be summed. We use a simple 7-stage adder compression tree. The first stage add produces 64 results, the second 32, and so on until the last stage produces the final result. This adder tree adds 7 cycles of latency to the pipeline, but maintains the rate of 1, producing an image pixel every cycle. With 128 projections being reconstructed in parallel, it takes 4 and 8 runs to reconstruct from 512 and 1024 projections respectively. Figure 3.10 shows the pipelined and parallel data path of the design. Leeser et al. [3] implemented a 16-way projection parallel version and predicted that with the advent of new high capacity FPGAs the architecture could scale to achieve more parallelism. Our design confirms this.

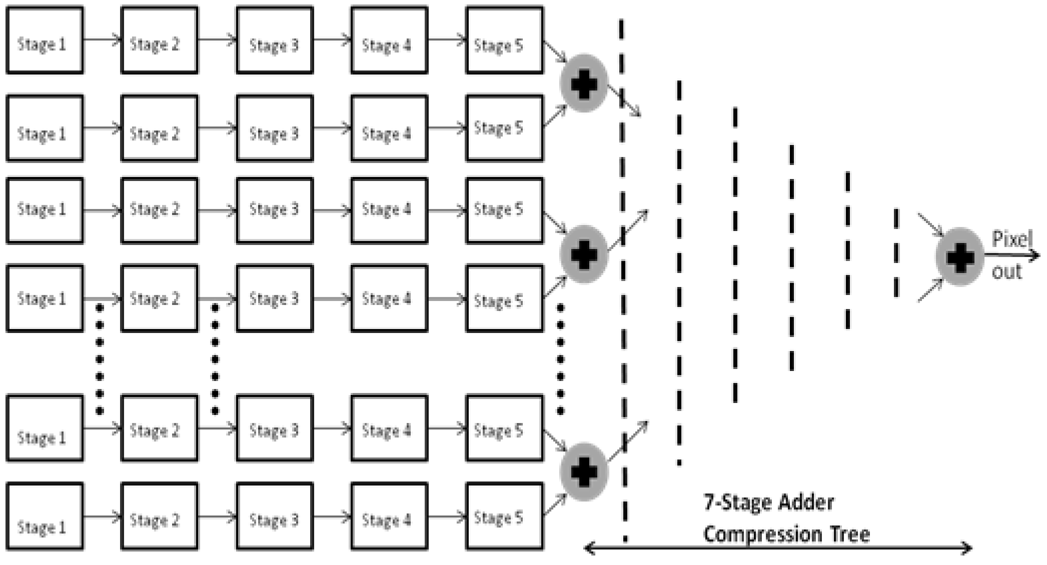


Figure 3.10 Pipelined and Parallel Data Path

As mentioned earlier, we decided to build a 128-way parallel system based on resource availability. The number of projections that can be processed in parallel is constrained by two resources. The first is the availability of memories to store the projections (*sinorow*). The second is the availability of DPS's to perform the multiplications in the interpolation step. Table 3.4 shows a summary of the resources available on the Stratix II EP2S180 chip present in the XD1000.

Table 3.4 Resources on Stratix II EP2S180 FPGA

Logic Resources	
Combinational ALUTs	143,520
Dedicated logic registers	143,520
RAM	
M512 (576 bits)	930
M4k (4.5 Kbits)	768
M-RAM (576 Kbits)	9
Total RAM bits	9,383,040
DSP	
9-bit DSP elements	768
Simple Multipliers (18-bit)	384

3.3.1.2. Memory Utilization

The memory requirements in the design are data dependent. The image memory does not have to be distributed because only one image pixel is computed every clock cycle and the memory is written to only once a cycle. We would however 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. We can store the entire image on chip using the large MRAM blocks. The image requires 4 Mbits of storage. There is a little over 5 Mbits of M-RAM on the chip. Hence we can store the image using the M-RAM blocks.

The sinogram memories have different requirements. Each row of the sinogram needs to be in a separate memory so that computations from multiple rows can proceed in parallel. Hence we need to allocate sinogram memories using the distributed M4k blocks. One row of the sinogram needs 16Kbits of memory. Hence one row of the sinogram takes up 4 M4k blocks. We can store at most 192 rows of the sinogram using up all the M4k blocks. The backprojector design cannot use up all the M4k blocks because some are used by the Impulse C wrapper functions that implement the communication interfaces. These functions take up around 5% of the memory and logic resources on the chip.

Having a system that is n-way parallel, where n is exactly divisible by the number of iterations of the outer loop has its advantages. Consider the case where we are reconstructing from 512 projections. A system that is 128-way parallel will reconstruct the image from 128 projections in each run and will have 4 execution runs to complete the process. Even if it were 150 way parallel, it still needs to run 4 times. In order to reduce the number of runs, the system needs to be at least 171-way parallel. This way it needs to run only 3 times.

The other memories in the design are the small memories needed to hold the offsets to compute *findex*. In the 128-way parallel system, each of these memories holds 4 or 8 values depending on whether the reconstruction is from 512 or 1024 projections. They easily fit in the M512 blocks.

One of the challenges of building a design that uses most of the memories on chip is that as more resources get utilized, the routability is diminished, and achieving timing closure becomes harder. Again, the timing target of 100MHz introduced by the Impulse C platform support package made this an important consideration.

In Impulse C the memories are not explicitly declared as M4k, M512 or M-RAM blocks. They are simply declared as arrays. Based on the array size the compiler, in conjunction with the *Quartus* tools, automatically generates the appropriate memory. Any memory configuration (dual port/single port) preferences imposed through the `co_array_config` function are passed on by the compiler to *Quartus*.

3.3.1.3. DSP Utilization

There are 96 DSP blocks in the Stratix II EP2S180. These can each be configured as eight 9-bit multipliers, four 18 bit multipliers or a single 36-bit multiplier. In the interpolation step, we perform 16-bit multiplication, and these are implemented using 18-bit multipliers in the DSP block. Since each interpolation operation needs two multipliers, we need 256 18-bit multipliers for a 128-way parallel system. These are available in the DSP blocks. It should be noted that if a design needed more multipliers than are available in the DSP blocks, it can still be implemented because the FPGA compiler will simply build the multipliers out of LUTs, if available. The observation about high memory utilization affecting routability also applies to the DSP and logic. We concluded that a 128-way parallel design would comfortably fit on our chip, and offered a good tradeoff between high performance and improved feasibility of achieving timing closure.

3.3.2. System Level View

In the previous sections we have discussed some of the individual elements of the design and how they were created. We now discuss from a system level how the backprojector operates.

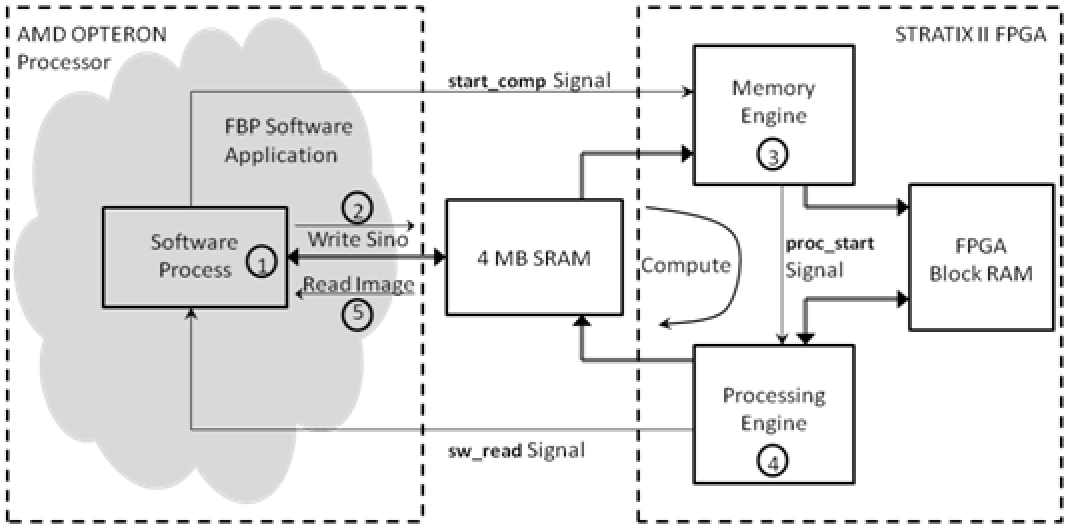


Figure 3.11 System Architecture

Figure 3.11 shows the system level block diagram. The numbers indicate the sequence of operations. The operations are outlined below.

1. The FBP software application parses the command line arguments and reads the specified sinogram input file. It then filters the sinogram with the specified filter and passes control to the Impulse C software process. The process converts the filtered sinogram into the 16-bit fixed point format.

2. The software process then transfers the sinogram data to the shared memory and signals the memory engine (using the *start_comp* signal) that it is done transferring data. The process of data transfer and signaling is performed easily in Impulse C using the functions shown below.
   co_memory_writeblock(datamem, OFFSET , &sinogram, SINO_SIZE);
   co_signal_post(start_comp,0);

"datamem" is the name given to the shared memory. OFFSET indicates the starting location in the shared memory. The function also takes in the address of the data to send to the shared memory and the number of bytes to transfer.

3. The memory engine reads 128 rows of the sinogram from the SRAM into the FPGA block RAM. Once it is done it signals the processing engine through the *proc_start* signal.

4. The processing engine then reconstructs the entire image from 128 projections and places the partially processed image in the FPGA block RAM.

5. At this point we have two versions of the design that deal with the partially processed image differently. The first version sends every row of the partially processed image to the SRAM as it is computed. It is then read by the CPU. Steps 1-4 are repeated. The final image is the sum of the partial images generated in each step. The CPU performs the summation as it retrieves partial images from the SRAM. The disadvantage of the CPU accumulation method is that the processing is stalled when transmitting the image to the SRAM. To avoid this, the second version of the design accumulates the image in the FPGA block RAM itself. In this version only one transfer at the end of the complete processing is required.

Chapter 4 presents the system timing, resource usage and relative performance of all the versions of the design.

4. Results

The previous chapter described how the Impulse C C-to-FPGA tool flow was employed to design an FPGA implementation of backprojection. The PSP capabilities were leveraged to create a software process to fit within the existing source code framework, seamlessly hand off the computation to the FPGA, and recover the results from shared memory. As the design evolved, various optimizations were made to both the design architecture and the system architecture. In this chapter we quantify and compare the effectiveness of those optimizations by presenting the performance and resource utilization of the designs. We also compare how the Impulse C backprojector design performs versus the software backprojector running on the host.

To facilitate a head-to-head comparison between Impulse C and traditional FPGA design methods, a HDL version of the backprojector was created in a related work [23]. The hand coded version was written with a combination of VHDL code and a schematic created using the schematic design tool in Altera *Quartus*. The hand coded version implements the same algorithm to perform backprojection. In this chapter we compare the performance of the HDL design with that of the Impulse C design.

We present both performance of the designs and the resource utilization for reconstruction from 512 and 1024 projections. The reconstructed image size is 512x512 pixels. In our system we assume the number of detector channels is equal to the number of projections, thereby keeping the sinogram a square matrix. Performance is measured as execution time. Resource utilization is measured as the logic, block RAM, and DSPs used.

4.1. Performance and Resource Utilization

We consider two versions of the design. The first version performs the accumulation of the image in the CPU, and the second performs the accumulation in the FPGA itself, as mentioned in section 3.3.2. The disadvantage of having the CPU perform the accumulation is that the processing is stalled to transmit the image to the SRAM as it is being computed. Figure 4.1 shows the system timing diagram for the CPU accumulation version. The timing annotations are for reconstruction from 512 projections. The grey sections indicate the resource is free at a given time and white indicates it is busy. It can be seen that during the processing step, the PROC_ENGINE writes to the SRAM when it computes a row of the image. The CPU then reads the data from the SRAM.

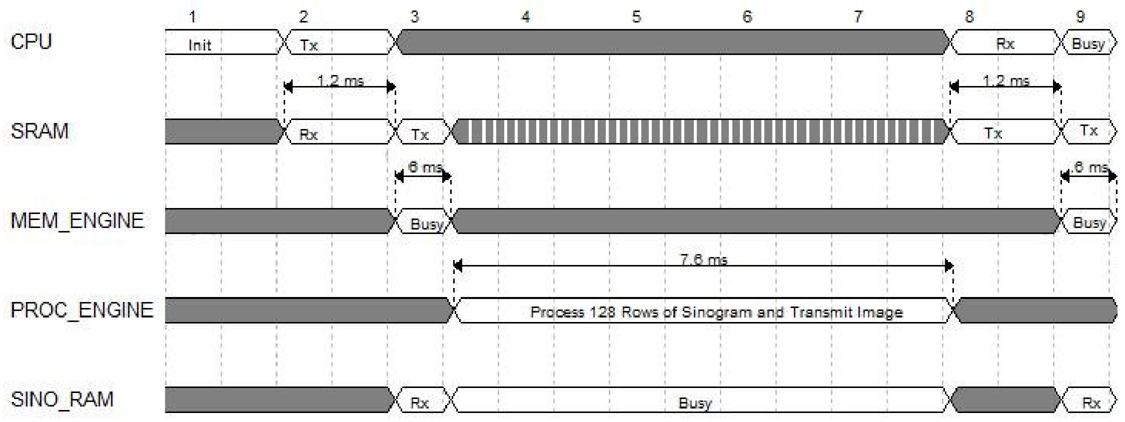


Figure 4.1 System timing for CPU accumulation. Reconstruction from 512 projections

Figure 4.2 shows the system timing of the FPGA accumulation version. It shows how the design can process data without being interrupted by SRAM write and CPU read operations.

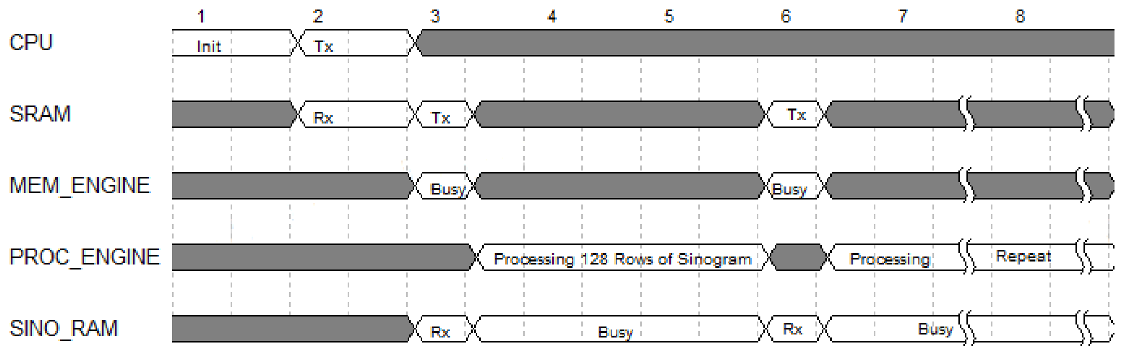


Figure 4.2 System timing for FPGA accumulation. Reconstruction from 512 projections

Table 4.1 and 4.2 gives the break-up of the execution time for the CPU and FPGA accumulation versions of the design. In table 4.1, we see that transferring the partially computed images from the FPGA to CPU (through the shared memory implemented on the SRAM) takes 15.2ms and 30.4ms when reconstructing from 512 and 1024 projections respectively. In the FPGA accumulation (table 4.2) there is only one transfer from the FPGA to CPU at the end of the computation, which takes 3.8ms irrespective of the number of projections, making the overall execution 30% faster than CPU accumulation. The miscellaneous CPU operations include fixed point to floating point transformations, and also the masking operation described in section 3.2.2.

Table 4.1 Break-up of execution time for CPU summation design

Execution Stage	512 Projections	1024 Projections
Transfer Sinogram from CPU->SRAM	1.05 ms	4.20 ms
Read Sinogram SRAM->FPGA	2.78 ms	11.12 ms
FPGA Computation	21.20 ms	42.40 ms
Transfer image from FPGA->SRAM	10.00 ms	20.00 ms
Transfer image from SRAM->CPU	5.20 ms	10.40 ms
Misc CPU operations	3 ms	3 ms
Total	43.25 ms	83.92 ms

Table 4.2 Break-up of execution time for FPGA accumulation version

Execution Stage	512 Projections	1024 Projections
Transfer Sinogram from CPU->SRAM	1.05 ms	4.20 ms
Read Sinogram SRAM->FPGA	2.78 ms	11.12 ms
FPGA Computation	21.20 ms	42.40 ms
Transfer image from FPGA->SRAM	2.50 ms	2.50 ms
Transfer image from SRAM->CPU	1.30 ms	1.30 ms
Misc CPU operations	3 ms	3 ms
Total	31.83 ms	64.52 ms

The main advantage of CPU accumulation is its reduced on-chip memory requirement. The CPU accumulation approach needs to store only one row of the image on-chip, whereas the FPGA accumulation approach needs to cache the entire image on-chip. Table 4.3 and 4.4 shows the resource utilization of the two designs. Caching the image takes up 90% of the M-RAM on-chip. Hence we cannot use this method for images that are much larger than 512x512. The CPU accumulation version can scale to support the reconstruction of much larger images. The other advantage of CPU accumulation is that it is a completely feed-forward design. FPGA accumulation has a feedback loop as we need to update previously cached values of the image when processing a new set of projections. This makes achieving timing closure in the FPGA accumulation design more challenging.

It is seen from table 4.3 that the distributed M4k blocks are the bottleneck in both designs. There are plenty of logic resources left over. To achieve further parallelism, we need more distributed memories.

Table 4.3 Resource utilization for CPU accumulation design

Resources	Available on chip	512 Projections		1024 Projections	
		No. Used	% Used	No. Used	% Used
Logic					
Combinational ALUTs	143,520	32,909	23%	33,319	23%
Dedicated logic registers	143,520	37,301	26%	37,712	26%
RAM					
M512 (576 bits)	930	72	8%	256	28%
M4k (4.5 Kbits)	768	595	77%	768	100%
M-RAM (576 Kbits)	9	0	0%	0	0%
DSP					
9-bit DSP elements	768	512	67%	512	67%

Table 4.4 Resource utilization for FPGA accumulation design

Resources	Available on Chip	512 Projections		1024 Projections	
		No. Used	% Used	No. Used	% Used
Logic					
Combinational ALUTs	143,520	39,748	28%	40,564	28%
Dedicated logic registers	143,520	52,744	37%	53,640	37%
RAM					
M512 (576 bits)	930	72	8%	256	28%
M4k (4.5 Kbits)	768	593	77%	768	100%
M-RAM (576 Kbits)	9	8	89%	8	89%
DSP					
9-bit DSP elements	768	512	67%	512	67%

4.2. Impulse C Vs Software

We compare the performance of the backprojector created with Impulse C to the performance achieved by simply executing the backprojection source code on the host. The Impulse C version of the backprojector performs two orders of magnitude better. When reconstructing 1024 projections with FPGA accumulation, we obtain a 155x speedup.

Table 4.5 Impulse C Vs Software execution time

Design	512 Projections	1024 Projections
Unmodified Source code on host	3.6s	10s
Impulse C design with CPU accumulation	43.25 ms	90.92 ms
Impulse C design with FPGA accumulation	31.83 ms	64.52 ms

Table 4.6 Impulse C Vs Software speedup

Design	512 Projections	1024 Projections
Unmodified Source code on host	1x	1x
Impulse C design with CPU accumulation	83x	110x
Impulse C design with FPGA accumulation	113x	155x

One reason why the software source code is slower than the optimized FPGA design is that the software was created to be flexible enough to support sinograms and images of arbitrary dimensions at run time. The FPGA design, on the other hand, can only support a fixed sinogram and image size at run time. When reconstructing from 1024 instead of 512 projections, the FPGA design has to be recompiled. The advantage we derived by fixing the image and sinogram dimensions is that we could generate address indices as offsets (section 3.2.1). Similar optimizations can be made to the software version. Hand optimization of the software source code was outside the scope of this work.

However, the major reason for the Impulse C design being faster is pipelining and loop unrolling which allows us to process 128 projections in parallel. The high bandwidth to the distributed memory and the availability of logic resources and DSPs to build multiple parallel data paths is not present in the processor. As a result, those optimizations cannot be replicated in the software version running on the processor. We believe that even with significant hand optimization, the Impulse C version of the design will still be much faster than the software.

The FPGA acceleration of backprojection makes the execution time of this stage negligible compared to the filter stage in FBP software benchmark. As mentioned in 3.1.1, backprojection accounts for 70-80% of the execution time in the benchmark. Thus, our partitioned implementation speeds up the overall FBP execution by 4x-5x. In section 3.1.1, we discussed why we chose to perform the filtering operation on the CPU and the backprojection on the FPGA. Nevertheless, the filter can be efficiently implemented on the FPGA and can achieve the same level of acceleration as the backprojection stage. Thus, performing FBP on the FPGA can be up to 155x faster than running FBP on a processor.

4.3. Impulse C Vs HDL

To contrast the performance of Impulse C designs with designs created by traditional methods, a hand coded version of the backprojector was developed in a related work [23]. It uses many design techniques that are unique to the level of detail allowed by hand coding VHDL. It employed the same algorithm as the Impulse C version to perform backprojection from 1024 projections. In addition, it used a separate parallel algorithm to implement the reconstruction from 512 projections which was designed to capitalize on the advantages of the given hardware, and the flexibility of the implementation method. Table 4.7 and 4.8 shows the performance of Impulse C when compared to the hand coded version of the design.

Table 4.7 Impulse C Vs HDL execution time

Design	512 Projections	1024 Projections
HDL version of design	21.68 ms	38.02 ms
Impulse C design with CPU accumulation	43.25 ms	90.92 ms
Impulse C design with FPGA accumulation	31.83 ms	64.52 ms

Table 4.8 Impulse C Vs HDL speedup

Design	512 Projections	1024 Projections
HDL version of design	1x	1x
Impulse C design with CPU accumulation	.50x	.41x
Impulse C design with FPGA accumulation	.68x	.59x

Table 4.9 provides a breakdown of the execution time of the various steps in the backprojection implementation for the Impulse C as well as hand coded version. The HDL version is ~1.7x faster than the Impulse C version.

Table 4.9 Breakup of execution time. Reconstruction from 1024 projections.

Execution Stage	Impulse C	HDL
Transfer Sinogram from CPU->SRAM	4.20 ms	6.05 ms
Read Sinogram SRAM->FPGA	11.12 ms	5.40 ms
FPGA Computation	42.40 ms	20.97 ms
Transfer image from FPGA->SRAM	2.50 ms	2.60 ms
Transfer image from SRAM->CPU	1.30 ms	
Post Processing	3.00 ms	3.00 ms
Total	64.52 ms	38.02 ms

There are few reasons for the better performance of the HDL design. First, the compute pipeline in the HDL version produces a result every cycle, whereas the Impulse C version produces a result only once every two cycles. This makes the hand coded compute pipeline 2x faster. Furthermore, in the hand coded version, the final image is streamed directly to the CPU instead of using the shared memory, making that process 30% faster. Lastly, the hand coded version has a custom SRAM controller that achieves 2x faster data access to the onboard SRAM than the Impulse C version.

As mentioned in section 3.2.3, we were not able to implement the sinogram memories as true dual port because of limitations in the Impulse C compiler. Also, the Impulse C scheduler makes conservative assumptions about the memory accesses to the image cache. These two factors result in the reduced rate of the compute pipeline. We believe that this is a limitation of the compiler that can be resolved in future versions of the Impulse C design tools.

As mentioned earlier, the Impulse C PSP for the XD1000 supports efficient data transfer only through shared memory, and streaming from the FPGA to the CPU is slow. Further, the SRAM controller on the PSP can achieve maximum throughput for FPGA-SRAM communication only for specific data widths (64-bit). The image and sinogram data in our application was 16-bit, and as a result, we took a 2x performance penalty on the SRAM-FPGA throughput. We attempted to circumvent this limitation by trying to pack our 16-bit data into 64-bit words before writing them to FPGA block RAM and unpack them as we used them in the pipeline. However, the overhead imposed by this modification, caused our design to fail the required 100MHz timing requirement.

These limitations can be resolved by enhancing support in the PSP for efficient data transfers of different bit-widths. It is seen from table 4.9 that the Impulse C PSP performs a very efficient data transfer from the CPU to the SRAM. This shows that it should be possible to match HDL performance on the FPGA–SRAM communication as well. By making these improvements to the compiler and PSP, the performance of Impulse C can match that achieved through hand coded HDL for our implementation of backprojection (table 4.10).

Table 4.10 Estimated impact on runtime and speedup by optimizing the compiler and PSP. Reconstruction of 512x512 image from 1024 projections

Implementation	Runtime	Speedup
Actual HDL version	38.02 ms	1x
Actual current Impulse C version	64.52 ms	0.59x
Estimate of Impulse C with PSP optimization to match HDL transfer speed to SRAM	57.60 ms	0.66x
Estimate of Impulse C with compiler optimizations	43.32 ms	0.88x
Impulse C with both optimizations	33.40 ms	1.05x

Table 4.11 compares resource utilization of the Impulse C design with the HDL design. We focus on the reconstruction from 1024 projections. It is seen that both designs have very similar resource utilization. In the HDL version, the M512 RAMs are used to store the horizontal and vertical offsets. The Impulse C version uses all the M4ks before using the M512s. The HDL version is more heavily pipelined and hence uses more registers.

Table 4.11 Comparison of resource utilization

Resources	Available on chip	Impulse C		HDL	
		No. Used	% Used	No. Used	% Used
Logic					
Combinational ALUTs	143,520	40,564	28%	29,043	20%
Dedicated logic registers	143,520	53,640	37%	65,089	45%
RAM					
M512 (576 bits)	930	256	28%	687	74%
M4k (4.5 Kbits)	768	768	100%	611	80%
M-RAM (576 Kbits)	9	8	89%	9	100%
DSP					
9-bit DSP elements	768	512	67%	512	67%

To compare the ease of use of Impulse C to HDL, we compare the design time and lines of code in the design (table 4.12, 4.13). Both the Impulse C and HDL designs were created by designers with similar hardware engineering background and FPGA design experience. Creating the initial version of the design using Impulse C took 25% less time than the HDL version. This includes the time it took us to get acquainted to the Impulse C tools and understand the tool flow and design methodology. Further, in the 9 weeks, we had created two Impulse C versions (CPU and FPGA accumulation) of the design, whereas only one HDL version had been created. The incremental time taken to design, test and debug the 1024 version of the design using Impulse C was much less when compared to HDL. This is because on the second pass, we were better acquainted with the Impulse C tools.

4.12 Comparison of design time

Design Version	Time	Time
512 Projections (Initial Design)	12 weeks	9 weeks
Incremental time to extend design to support 1024 projections	1 week	1 day

The Impulse C design required ~3000 lines of code. As mentioned earlier, we manually unrolled the loops instead of using the `CO UNROLL` pragma. The manual unrolling was accomplished by replicating code statements with different variables names. Of the 3000 lines only ~600 were unique lines of code. Parts of the hand coded design were created with schematics. The equivalent lines of HDL code presented in table 4.12 were estimated by the designer.

4.13 Impulse C versus HDL lines of code

Implementation	Total lines of code
HDL	~10000
Impulse C	~3000

We have compared the performance, resource utilization, and ease of use of the backprojection design created using HDL and Impulse C. There is however, an important distinction between the two implementations. The streamlined Impulse C design flow allowed us to seamlessly integrate the backprojector into the existing codebase, resulting in 5x faster run time than the software implementation. The HDL version, on the other hand, could not be integrated into the existing code. The existing FBP code was first executed to generate the sinogram data. A different code base had to be run to perform the backprojection and collect the results. The FBP code was then re-run to post process the results. In order to achieve a similar seamless integration of the HDL design into the existing codebase, more design time and effort will be required.

The next chapter discusses some of the lessons learned and possible future directions.

5. Conclusions

In the chapter we discuss the strengths and weaknesses of Impulse C, and describe some basic requirements for creating efficient Impulse C designs. WE present our conclusions from the results, and identify possible future research directions.

5.1. Writing Efficient Impulse C

Traditional FPGA design in HDL is fundamentally different from Impulse C design. A Hardware Description Language (HDL) is precisely that; a way to describe the hardware that performs the computation. The designer defines the logic blocks in the circuit, makes sure they are synchronized, and manually defines how they interact. C, on the other hand, is a way to describe a computation. Using Impulse C, we describe computations as one would in traditional C programming and move the low level timing and synchronization tasks to the compiler. We then use compiler optimizations to generate efficient hardware.

An important caveat is that when designing with Impulse C, the designer needs to have an idea of the design architecture that is suitable for the computation. The designer has to consciously structure the C program to help the compiler do an efficient job. For example, the sinogram memories have to be declared as individual memories so that each processing engines can get access to one row of the sinogram. If it is declared as one large memory, then the compiler will place it in the M-RAM block, which does not have the bandwidth to feed all of the computations in parallel.

When creating designs with Impulse C, one does not need to know how the compiler works, but one must have a firm idea of what the compiler does. Specifically, the designer must acquaint himself with the impact of specific changes made to the C code on the design. To this end, analysis tools such as Stage Master Explorer, pipeline graphs, etc., are useful in assessing the effects of various optimization strategies.

The designer also needs to know what resources and how much of each are available on the FPGA. Several design decisions are contingent on available resources, and different implementation strategies offer different resource versus performance tradeoffs.

5.2. The Strengths and Weaknesses of Impulse C

Impulse C is a great design methodology for targeting a hardware platform. The ability to create applications entirely in C but have them easily partitioned across the CPU and FPGA is very attractive. The ability to perform functional verification on the complete design is greatly enhanced. If one were creating the FPGA design separately by writing HDL and C to run on the host, the design verification is much more complex. Further, because the design of the software and hardware is so tightly coupled the eventual implementation is totally seamless. With traditional methods it takes extra effort to seamlessly integrate the software and hardware execution stages.

Impulse C is a big win over HDL when it comes to design effort. It affords rapid prototyping and allows the designer try various strategies within a short span of time. As the design evolves through various optimization strategies a designer using Impulse C will find his task simpler and less time consuming than one using HDL.

The Impulse C support for pipelining computations is excellent. The methods to perform the optimization are intuitive and the tools to analyze pipeline performance are full featured and informative. The unrolling optimization, on the other hand, is not supported very well. We worked around this by manually unrolling the code. Ideally there would be methods to perform partial unrolling of loops and automatically break up memories to support parallel execution. Tools to analyze the impact of unrolling would also be a welcome addition to the tool set.

One of the drawbacks of Impulse C is the loss of fine grained control over the resulting hardware. In certain situations we might want to make simple modifications like adding registers to the input and output of a computation. For example, we discovered that writing to the image cache was a step that failed timing. A simple work around we wanted to implement was to postpone the write to the next clock cycle by adding a register to the input and output of the cache. These sorts of fine grained changes are not easily communicated to the compiler. An upshot of the above drawback is that it is extremely difficult to efficiently implement control logic in the pipeline.

Another disadvantage of Impulse C when targeting platforms is that there is a fixed frequency at which the user logic is clocked. The ability to clock the logic at a desired rate is one of the important features of FPGAs. If the frequency of operation needs to be fixed to support reliable communication with off-chip resources, it would be nice to have 2-3 different fixed clock speed settings. That way the designer can choose the setting that best suits his needs.

5.3. Future Directions

Our work in benchmarking Impulse C provided some insight into how to optimize C-to-FPGA designs and how the performance of Impulse C compares to software and HDL. However, this work represents a single point in the design space. It would be nice to undertake an investigation of the performance of Impulse C versus HDL, over a diverse suite of benchmarks.

There is plenty of opportunity to extend the work on accelerating FBP. The filter can be implemented on the FPGA to further accelerate FBP. This system can be modified to support fan beam detector geometry. Further, the backprojector architecture applies directly to the forward projection task. This work can be extended to perform iterative reconstruction by implementing algorithms such as maximum likelihood expectation minimization (ML-EM) [28, 29]. It will also be interesting to perform a study on what it takes to port the existing design to a different platform such as the XD2000 to assess the portability of Impulse C designs.

5.4. Conclusion

A C-to-FPGA tool flow was employed to create an FPGA implementation of backprojection. The backprojector designed with Impulse C was ~155x faster than the software implementation, resulting in a 5x speedup of the FBP software. Furthermore, the backprojector design is within 1.7x of the performance of hand coded HDL, with significantly reduced design effort. This thesis concludes that C-to-FPGA tools such as Impulse C can be a viable alternative to traditional HDL for creating platform based FPGA designs.

References

[1] P. E. Kinahan, M. Defrise, and R. Clackdoyle. “Analytic Image Reconstruction Methods.” Emission Tomography: The Fundamentals of PET and SPECT. Miles N. Wernick and John N. Aarsvold. San Diego, CA, USA: Elsevier Academic Press, 2004. pp. 421–442.

[2] Nicolas GAC, Stéphane Mancini, Michel Desvignes, and Dominique Houzet. “High Speed 3D Tomography on CPU, GPU, and FPGA.” EURASIP Journal on Embedded Systems. Vol. 2008, Article ID 930250, 2008.

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

[4] Agi, I., Hurst, P.J., and Current, K.W. “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.

[5] Kachelrieb, M.; Knaup, M.; Bockenbach, O. "Hyperfast Parallel Beam Backprojection." Nuclear Science Symposium Conference Record, 2006 IEEE. Vol. 5, Oct. 2006. pp. 3111-3114.

[6] Ambric. “Am2045 CT Backprojection Acceleration White Paper.” July 12, 2007. White Paper.

[7] 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, Feb. 2006. pp. 1494–1501.

[8] F. Xu and K. Mueller, "Accelerating popular tomographic reconstruction algorithms on commodity pc graphics hardware." IEEE Transaction of Nuclear Science. Vol. 52, Issue 3, Part 1, June 2005. pp. 654-663.

[9] F. Xu and K. Mueller. "Towards a Unified Framework for Rapid Computed Tomography on Commodity GPUs." Nuclear Science Symposium Conference Record, 2003 IEEE. Vol. 4, Oct. 2003. pp. 2757-2759.

[10] K. Mueller and F. Xu. "Practical considerations for GPU-accelerated CT." 3rd IEEE International Symposium on Biomedical Imaging: Nano to Macro, 2006. Apr. 2006. pp. 1184-1187.

[11] Jain, Anil K. Fundamentals of Digital Image Processing. Prentice Hall, 1989.

[12] Impulse Accelerated Technologies. Impulse C User Guide. Copyright © 2002-2008, Impulse Accelerated Technologies.

[13] XtremeData Inc, "XD1000 Development System Product Flyer." XtremeData Inc. 06/05/2009 <*http://www.xtremedatainc.com/index.php?option=com_docman&task=doc_details&gid=17&Itemid=129*>.

[14] Sundar. "File:CTScan.jpg - Wikipedia, the free encyclopedia." Wikipedia, the free encyclopedia. 06/05/2009 <*http://en.wikipedia.org/wiki/File:CTScan.jpg*>.

[15] Adam Alessio and Paul Kinahan. "PET Image Reconstruction." (to appear) Nuclear Medicine 2nd Ed. Henkin et al.

[16] Hutton, Brian F. "An Introduction to Iterative Reconstruction." Alasbimn Journal. Year 5, No. 18: October 2002. Article N° AJ18-6.

[17] Dr. Adam Alessio, Research Asst. Professor, Department of Radiology, University of Washington. “University of Washington Emission Reconstruction Demo (UWERD).” © Adam Alessio Feb. 2005

[18] Slideshare, "Ct2 History Process Scanners Dip." SlideShare Inc. 06/05/2009 <*http://www.slideshare.net/lobelize/ct-2-history-process-scanners-dip*>.

[19] Altera Corporation, "FPGA CPLD and ASIC from Altera." Altera Corporation. 06/05/2009 <*http://www.altera.com/*>.

[20] Dr. Adam Alessio, Research Asst. Professor, Department of Radiology, University of Washington. “UW Emission Reconstruction.” © University of Washington, Imaging Research Laboratory Jan 2004.

[21] Espasa, R. and Valero, M. “Exploiting Instruction- and Data-Level Parallelism.” <u>IEEE Micro.</u> Vol. 17, Issue 5, Sep. 1997. pp. 20-27. DOI= http://dx.doi.org/10.1109/40.621210

[22] Bushberg, Seibert, Leidholdt and Boone. <u>The Essential Physics of Medical Imaging.</u> Second Edition, Lippincott Williams and Wilkins, 2002. ISBN 0-683-30118-7

[23] Jimmy Xu, Department of Electrical Engineering, University of Washington. <u>FPGA Acceleration of Backprojection.*</u> Master's Thesis, 2009. (*To be published).

[24] Agility Design Solutions Inc., "C Based Products :: Products :: Agility Design Solutions :: Algorithm to Implementation. Fast." Agility Design Solutions Inc. 06/05/2009 <*http://agilityds.com/products/c_based_products/default.aspx*>.

[25] Mitronics, Inc., "Mitronics - Hybrid Computing." Mitronics, Inc.. 06/05/2009 <*http://www.mitrionics.com/*>.

[26] Mentor Graphics, "Generate Correct-by-Construction, High-Quality RTL, 10-100x Faster - Mentor Graphics." Mentor Graphics. 06/05/2009 <*http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/*>.

[27] Impulse Acclerated Technologies, "Impulse Acclerated Technologies - Software Tools for an Acclerated World." Impulse Acclerated Technologies. 06/05/2009 <*http://www.impulseaccelerated.com/*>.

[28] Dempster A, Laird N, and Rubin D. "Maximum likelihood from incomplete data via the EM algorithm." Journal of the Royal Statistical Society, Vol. 39, 1977. pp. 1-38.

[29] Shepp L and Vardi Y, "Maximum Likelihood Reconstruction for Emission Tomography," IEEE Transactions on Medical Imaging, Vol. MI-1, 1982. pp. 113-122.