

# Real-time H/W Implementation of the Approximate Discrete Radon Transform

M. T. Frederick, N. A. VanderHorn, and A. K. Somani  
Dependable Computing and Networking Laboratory  
Department of Electrical & Computer Engineering  
Iowa State University, Ames, IA, 50011, USA  
E-mail: {freds, nvander, arun}@iastate.edu

## Abstract

The Radon transform (RT) is a widely studied algorithm used to perform image pattern extraction in fields such as computer graphics, medical imagery, and avionics. Real-time implementation of the discrete RT (DRT) is extremely difficult due to its use of complex trigonometric functions and  $O(N^3)$  time complexity, making its use in video applications difficult. A  $O(N^2 \lg N)$  approximate discrete (ADRT) has been presented in literature [1] that allows highly parallel computation. This paper presents an architecture that uses the ADRT to create a computation architecture known as the  $x$ ADRT. Performance analysis indicates that it can achieve a refresh rate of 10 frames per second for use in real-time image processing applications.

## 1 Introduction

The Radon Transform (RT) is an essential component of many pattern recognition and extraction methods and is used extensively on applications such as computer graphics and medical imagery. The  $O(N^3)$  time complexity of the RT makes real-time execution difficult and thus, its use in video applications computationally expensive.

The standard RT computation of an  $N \times N$  image sampled at  $M$  distinct angles requires  $O(N^2 M)$  time, with  $M$  on the order of  $N$ . Its execution also relies on the use of complex, time consuming trigonometric functions. Minimizing the computational complexity and running time of the RT requires an algorithm that lends itself to parallel computation and minimal computational complexity. The Approximate Discrete Radon Transform (ADRT) [1] outlines a method that computes an approximation of the RT and requires only addition operations.

The research reported in this paper is partially supported by the Jerry R. Junkins endowment at Iowa State University and the Lockheed Martin Corporation.

## 2 Approximate Discrete Radon Transform

To speed up the discrete RT it is necessary to observe that for discrete images, each point in Radon space is not completely independent of neighboring points. As observed in [1], for neighboring angles, different strips may share large subsets of pixel, as shown in Fig. 1, excerpted from the same paper. In this example, each angular line shares the bottom half of the vertical line originating from the same pixel, and the top half of the vertical line originating from the pixel to the left. Thus, these partial sums can be computed iteratively, and their values used in multiple Radon points. Partial sums are computed in a sequential manner and used in successive iterations to optimize computational reuse.

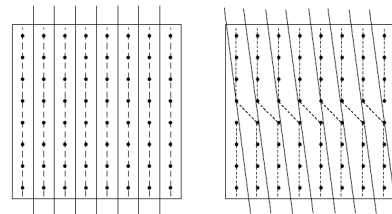


Figure 1: Shared partial sums [1]

### 2.1 ADRT Algorithm Optimizations

When considering an optimized hardware implementation it is important to take into account memory accesses. Because pixel information located in memory must be read into the Radon Calculation Engine (RaCE) and written back to storage memory, it is beneficial to perform all calculations that consume data from the same memory location before writing the results back.

The original ADRT algorithm first computes all vertical rays (“vertical” additions) from every start point ( $a$  is even) before computing any angular components or “diagonal” additions ( $a$  is odd). This requires reading the value

of each pixel twice. By exchanging the  $a$  and  $y$  loops of the original algorithm [1], as in Alg. 2.1, both vertical and diagonal additions for the same pixel can occur by reading each pixel only once. Consequently, when a row is read into the RaCE it is completely consumed and can be overwritten in memory. In the original algorithm, the image would have to be buffered while all of the rows perform the vertical addition until the diagonal addition is performed. Interchanging realizes two optimizations; 1) number of memory accesses is reduced as rows are only read once per iteration, and 2) the need to buffer large amounts of data is eliminated.

**Algorithm 2.1:** MODIFIED ADRT( $R_0$ )

```

for  $i \leftarrow 1$  to  $\lg N$ 
  for  $y \leftarrow 0$  to  $N - 2^i$  step  $2^i$ 
    for  $a \leftarrow 0$  to  $2^i - 1$ 
      for  $x \leftarrow 0$  to  $2 \cdot N - 1$ 
         $R_i(x, y, a) = R_{i-1}(x, y, \lfloor \frac{a}{2} \rfloor) + R_{i-1}(x - \lceil \frac{a}{2} \rceil, y + 2^{i-1}, \lfloor \frac{a}{2} \rfloor)$ 

```

## 2.2 Exploiting Parallelism

An inspection of the memory access structure for the modified ADRT algorithm reveals an additional opportunity for performance improvement. Parallelism of the original algorithm is discussed in [2], however, the parallel implementation discussed here has been performed using the modified ADRT algorithm given by Alg. 2.1 and with an eye towards an efficient hardware implementation.

Fig. 2 depicts how each memory location in a  $16 \times 16$  image, in hexadecimal, is accessed during each sweep of the calculation. On the first iteration, row 0 is added to row 1 and the sum saved in row 0. Row 0 is also added to a rotated row 1 and the sum saved in row 1. The figure represents this as the initial row 0 contributing to rows 0 and 1 in pass 1. The memory access pattern reveals that the first iteration only needs to access 8 sets of 2 adjacent row pairs. The second iteration, the algorithm needs to access 4 sets of 4 rows, the third 2 sets of 8 rows, and the final iteration all 16 rows. This allows the memory needed to perform the whole computation to be partitioned, with each partition providing an additional memory port to be used in parallel.

In Fig. 2, the memory is partitioned into quarters, with the first and second passes of the algorithm operating sequentially within a quadrant. The  $3^{rd}$  iteration operates on two of the partitions, and the 4th operates on all 4 partitions. If 4 partitions are chosen, iterations 1 through  $\lg \frac{N}{4}$  use 1 of the partitions, while iteration  $\lg \frac{N}{2}$  uses 2 of them, and iteration  $\lg N$  uses all 4 of them.

If  $m$  partitions are chosen, the first 1 to  $\lg \frac{N}{m}$  iterations can perform parallel computation of  $\frac{1}{m}^{th}$  of the image, and the final  $\lg m$  iterations each compute of  $\frac{2^j}{m}$ ,  $j = 1, \dots, \lg m$  partitions in parallel. In this manner, the bulk of the computation can be parallelized, while only the final iterations exhibit increasingly sequential behavior.

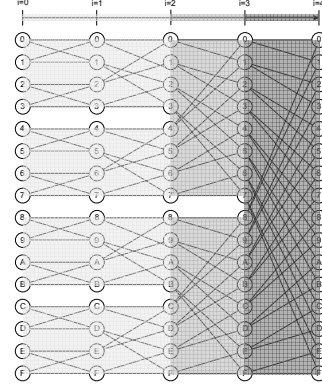


Figure 2: Memory access pattern for  $N = 16$

## 3 Radon Calculation Engine (RaCE)

The general architecture of the xADRT assumes that the data source switches between two input buffers—buffer  $i + 1$  is filled while buffer  $i$  is consumed. The four separate quadrants are computed in parallel using identical RaCEs by using the original image (Eqn. 1) and performing a main diagonal flip (Eqn. 2),  $\frac{\pi}{2}$  clockwise rotation (Eqn. 3), and flip over horizontal axis (Eqn. 4).

$$I_0[x][y] = I[x][y] \quad (1)$$

$$I_1[x][y] = I[y][x] \quad (2)$$

$$I_2[x][y] = I[N - y - 1][x] \quad (3)$$

$$I_3[x][y] = I[x][N - y - 1] \quad (4)$$

The modified ADRT algorithm (Alg. 2.1) allows each memory location to be read, computed, and written once per pass through the image. That means the number of serial cycles spent performing memory transfers and computation,  $C$ , can be modeled by Eqn. 5, where  $b$  is the memory width in pixels,  $N$  the image size, and  $p$  the pixel size in bytes. The equation reflects the product of the number of pixels in the image,  $N \cdot 2 \cdot N$ , the 3 cycles needed to read, compute, and write the pixel, and the number of bytes per pixel,  $p$ , divided by the width of the memory interface,  $p \cdot b$ . This result is then summed over the  $\lg N$  passes through the image that the algorithm needs to compute the ADRT.

$$C_1 = \sum_{i=1}^{\lg N} \frac{N \cdot 2 \cdot N \cdot 3 \cdot p}{p \cdot b} = N^2 \cdot \frac{6 \cdot \lg N}{b} \quad (5)$$

The serial memory accesses for parallelized computation of 2 and 4 memory partitions are shown in Eqns. 6 and 7. Eqn. 6 is an extension of Eqn. 5, but takes into account the memory transactions that occur in parallel for  $\lg \frac{N}{2}$  steps for 2 memory partitions. Similarly, Eqn. 7 is an extension of Eqn. 6, but takes into account the memory transactions that

occur in 4-parallel for  $lg\frac{N}{4}$  steps for 4 memory partitions. It is observed that  $C_1 \approx 2 \cdot C_2 \approx 4 \cdot C_4$ , because each level of memory partitioning allows approximately twice the number of memory transactions to occur in parallel.

$$C_2 = \frac{6 \cdot N^2}{b} + \frac{1}{2} \cdot \sum_{i=1}^{lgN-1} \frac{6 \cdot N^2}{b} = N^2 \cdot \frac{3 \cdot lgN + 3}{b} \quad (6)$$

$$C_4 = \frac{6 \cdot N^2}{b} + \frac{1}{2} \cdot \frac{6 \cdot N^2}{b} + \frac{1}{4} \cdot \sum_{i=1}^{lgN-2} \frac{6 \cdot N^2}{b} = N^2 \cdot \frac{\frac{3}{2} \cdot lgN + 6}{b} \quad (7)$$

These equations present optimal cases for the number of cycles needed to perform the computation and do not account for the overhead incurred through state management, address computation, and all other control sequences. They give a minimum bound for performance, assuming all control logic can be hidden, and the critical path of computation is purely the procurement, processing, and storage of data.

Several challenges to designing the RaCE around the flow of data must be addressed. The first design challenge is to limit the amount of memory necessary to compute the ADRT. The original algorithm instantiates up to  $2^{i-1}$  arrays in each of the  $lgN$  passes through the image. The xADRT requires only the memory needed to store  $N \cdot 2N = 2N^2$  pixels of size  $p$  bytes per pixel for the image and its padding, yielding a total of  $2 \cdot p \cdot N^2$  bits.

Second, several read after write (RAW) errors need to be handled. The first such error occurs on the row level, where rows are read non-sequentially, computed, and are stored back to row locations sequentially, in the process overwriting unconsumed data. Another RAW error occurs due to the offset within a row where pixels are read and computed, and then written-back sequentially beginning at the start of the row. The  $\frac{N}{2}$  pixel offset data buffer stores deals with intra-row RAW errors and brings the total required memory resources of the RaCE to  $p(2 \cdot N^2 + \frac{N}{2})$  bits.

The RaCE, shown in Fig. 3, handles the bulk of the xADRT calculation. Each quadrant uses an identical RaCE that operates on a different image orientation as explained in the previous section. The RaCE is comprised of different logical blocks designed to create an efficient flow of data, as outlined below.

### 3.1 Finite State Machine (FSM)

The FSM drives the memory accesses of the RaCE on a logical level. It's granularity is only row level in the image, meaning that it sees memory,  $M$ , as the exact image, organized in rows and columns. The FSM asks the memory controller for two rows, rotates them equally, adds them "vertically," rotates the second row, adds them "diagonally,"

and writes back sequentially in memory. Alg. 3.1 shows memory transactions that occur governed by the variables  $i$ ,  $y$ , and  $a$ . The location in memory,  $M[m]$ , is found by summing [2] the values of  $y$  and  $a$ , as defined by Alg. 2.1.

**Algorithm 3.1:** FSM()

```

for  $i \leftarrow 1$  to  $lgN$ 
  for  $y \leftarrow 0$  to  $N - 2^i$  step  $2^i$ 
    for  $a \leftarrow 0$  to  $2^i - 1$ 
       $M[y + a] = M[y + \lfloor \frac{a}{2} \rfloor] + M[y + \lfloor \frac{a}{2} \rfloor + 2^{i-1}]$ 

```

**Address aliasing** is a combinatorial process that remedies the FSM of the RAW errors that occur due to writing to the memory sequentially. In order to avoid the RAW errors, writing back to the exact same memory locations that were read is necessary. Data is overwritten immediately after it is consumed making the only additional memory needed is the  $\frac{N}{2}$  pixel long offset data buffer. This, however, creates a problem in the next iteration of the FSM because the addition results are not written back sequentially, and the FSM will find the image scrambled.

For example, aliasing allows rows 0 and 3 to be read, computed and then written back to locations 0 and 3 instead of sequentially, where they would be written back to rows 0 and 1. The FSM expects the results to be in rows 0 and 1 for the sake of the next pass through the image. To allow memory write back to the same locations that were read, the memory address accesses were studied and a solution transforming current logical row addresses to previous pass actual addresses was found.

$$RD_i(Actual) = LSBflip(RD_i(Logical), i - 1) \quad (8)$$

Given the logical row addresses of iteration  $i$ , the location where the row should be written within the current pass is the logical address with the least and most significant bits flipped over  $i - 1$ . Therefore the rows to be read in pass  $i$  are located at the requested logical row address with bits on either side of  $i - 1$  flipped. For example in pass  $i = 4$ , if logical rows 5 (0b0101) and 13 (0b1101) are requested, those rows should have been written in rows 3 (0b0011) and 11 (0b1011) in pass  $i = 3$ , but were instead actually written to rows 5 (0b0101) and 13 (0b1101). The exchange of the middle bits of 5 (0b0101) and 13 (0b1101) yield, 3 (0b0011) and 11 (0b1011), and the actual location of the desired rows in memory.

The memory controller performs **row blocking and rotation** by dividing the rows up into blocks of pixels to be consumed because it is difficult to read, rotate, and write entire rows of up to 1024 pixels at a time. Memory interfaces aren't wide enough, and the mass rotation and addition row operations necessary to operate on entire rows at

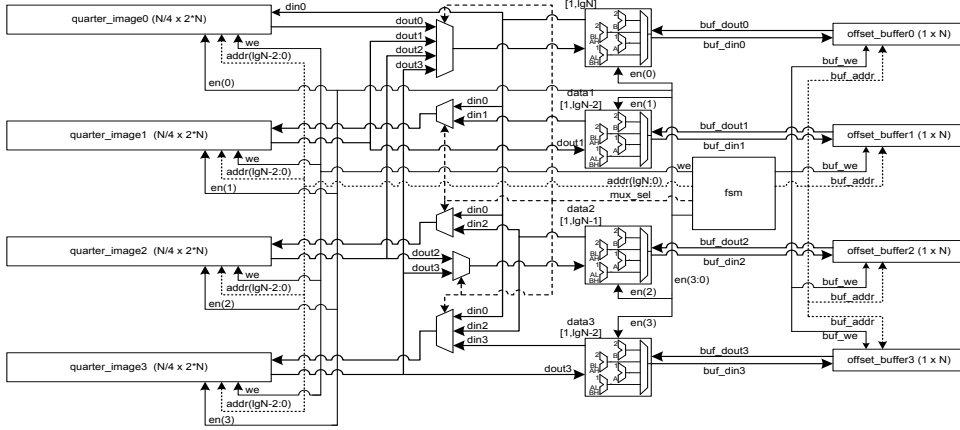


Figure 3: Radon Calculation Engine (RaCE) for 4 Memory Partitions

a time are limited by available combinational logic, memory, and routing resources. This is done by virtually rotating rows of pixels by calculating the proper pixel offset as part of the read address for a block and managing the reads between the memory and the offset data buffer.

A problem with row rotation using address calculation is that the offset of each row leaves pixels behind at the start of each row. When pixels are read at an offset, they are immediately consumed and the results written back sequentially in the same row. This means that if, in an arbitrary row, block 1 is the first block consumed in the row, the result of its calculation will be written back to block 0 in the same row before the unconsumed data in block 0 is read. This is a read after write (RAW) error that is solved through the use of the offset data buffer.

The **offset data buffer** prevents the overwriting of unconsumed data in the offset position of every other row read by the memory controller. At the start of each iteration, the pixels that are offset past are read into the offset data buffer and saved until needed. The offset data buffer is an  $\frac{N}{2}$  pixel deep memory that is used to store up to half a row of pixels at its peak utilization in the final stages of the last pass of the algorithm where  $i = \lg N$ .

The memory controller begins reading pixels at an offset within each row of the image due to the intra row offset  $\lceil \frac{a}{2} \rceil$  term in  $R_{i-1}(x - \lceil \frac{a}{2} \rceil, y + 2^{i-1}, \lfloor \frac{a}{2} \rfloor)$  of Eqn. 2.1. In this manner, maximum buffer utilization occurs during the last iteration where  $i = \lg N$ , and  $a = \frac{2^{i \lg N - 1}}{2}$ , yielding a total of  $\lceil \frac{a}{2} \rceil = \lceil \frac{2^{i \lg N - 1}}{2} \rceil = \lceil \frac{N-1}{2} \rceil = \frac{N}{2}$  pixels that will need to be stored so that they are not overwritten before they are consumed. As blocks of pixels are read and computed at an offset given by  $x - \lceil \frac{a}{2} \rceil$ , write-back of the results occurs sequentially at  $x$ . The offset data buffer prevents the overwriting of unconsumed data due to sequential write-back.

The FSM coordinates the **parallel computation** of the ADRT. There are three current implementations of the

RaCE, corresponding to three different parallelization levels. The first implementation sees memory as one partition, the second as two, and the third as four. The architecture for the four partition RaCE is shown in Fig. 3. It shows how the FSM provides global management of the entire architecture, including addressing, memory reading/writing, memory enabling, and memory interface control.

There are four data computation modules, with module *data0* being the primary one, because it is used in all passes through the image. Data module *data2* is the secondary one, and is used in all passes through the image with the exception of the final one. The other two data modules are used for  $\lg \frac{N}{4}$  passes through the image.

Each computational module is continually active. Adders 1 and 3 perform the vertical additions, while adders 2 and 4 perform the diagonal ones. The 4 adders add blocks of size  $b$  pixels of size  $p$  in a pipelined manner. The FSM controls which adder has a valid output at any given time. It may occur that the maximum sum of  $N$  pixels exceeds the bit width of pixel storage size. The calculation structure, once the maximum value of  $2^p - 1$  is reached, saturates and remains at that value. For example, if an image with an initial pixel size of 8 bits is input to the RaCE, the designer may wish that 16 bits of storage be available for each summed pixel. For an image of  $N = 512$ , the maximum sum that can be achieved is  $N \cdot 2^p - 1 = 512 \cdot 2^8 - 1 = 130560$ , a number that could only be accommodated by  $\lg 130560 \approx 17 \text{bits}$ . Instead, the calculation structure will saturate the sum of  $N$  pixels of width 8 at the pixel storage width of 16-bits of  $2^{16} - 1 = 65535$ .

## 4 Performance Results

Performance results of the xADRT calculation indicate that this architecture can reach the 100ms/frame threshold for image sizes less than or equal to  $N = 512$ . Block sizes

of  $b = 2, 4, 8$  of  $p = 1, 2$  byte pixels have been tested and are denoted as  $(p,b)$  in Fig. 4. These sizes have been chosen based on 32 and 64-bit memory interfaces. The results for 1 (SERIAL), 2 (PARA) and 4 (PARA4) memory partitions are shown in Fig. 4 each operating at synthesis frequency on a Xilinx Virtex II Pro (V2P) FPGA [3]. For  $N = 512$  and execution speeds of 95.34 and 47.11 ms for 32 and 64-bit memory widths, respectively, are obtained.

As is to be expected, increasing the number of memory partitions and the pixels read per memory transaction increases the system performance, while increasing pixel size decreases it. The P4 implementation actually outperformed the singular memory partition, 2-byte pixel, 2-pixel memory interface (SERIAL (2,2)). Additionally, the number of bytes/pixel in the P4 implementation had no effect on execution as the processor has a 32-bit datapath.

Tab. 1 indicates the average performance results for both 32 and 64-bit memory interfaces for synthesized frequency, area utilization, and percent of optimal performance. Parallelization seems to have little effect on synthesized frequency, and in most cases, actually yields higher operational frequencies. This is attributable to VHDL optimizations that took place while converting the serial design to parallel. As expected, increasing parallelization and datapath width increases area. Regardless, all designs are capable of being efficiently implemented in FPGAs.

The average percent of the optimal performance is calculated based on Eqns. 5, 6, and 7. The optimal number of clock cycles, though extremely idealistic, provides a rough method of assessing the architecture's efficiency. The optimal cycles computation is idealistic because it doesn't include the effect of control overhead, such as address calculation, state machine execution, and handshaking between modules. Additionally, it doesn't account for memory transaction cycles needed to fill the offset data buffer at the beginning of each row read. Though the calculated optimal performance is affected by these factors and thus is an extremely best case model, it does indicate that there is room for further architecture optimization.

The results of a MATLAB-based DRT [4], and hardware computation results of the ADRT transformed to DRT space are shown for an arbitrary image of vegetables. Fig. 5 indicates the difference between ADRT translated hardware output and the MATLAB DRT is visually imperceptible.

The 32-bit xADRT compares favorably to a general purpose processor (GPP) ADRT implementation. In this case, the ADRT was implemented, compiled using gcc at the highest optimization level, and executed on a 3.0GHz Intel Pentium 4 (P4) system with 1GB of RAM running Linux run-level 1. File I/O was removed from consideration in order to get a fair comparison to the xADRT architecture. The results are shown in Fig. 6 and are normalized to the P4's performance. Execution times for the P4 system were

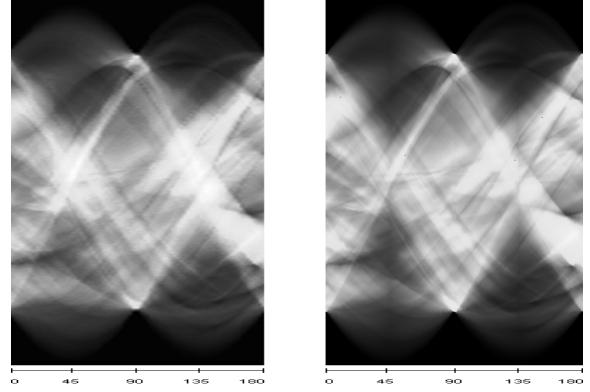


Figure 5: MATLAB (left) and xADRT (right)

obtained by measuring the cycle counters in the processor. The results show the xADRT has an average speedup of 4x over the P4 implementation.

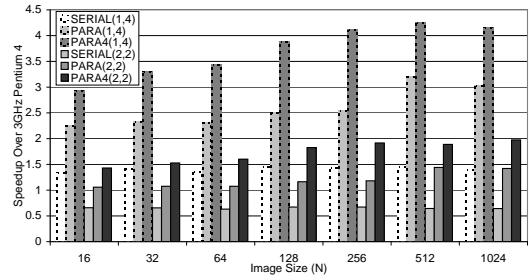


Figure 6: Speedup over P4 vs. Image (lgN)

Several 2-Dimensional Discrete Fourier Transform (DFT) architectures were chosen [5], [6], [7], and [8] as points of comparison because the 2-D DFT has the same  $O(N^2 \lg N)$  algorithmic complexity as the ADRT. The results in Tab. 2 indicate that although the xADRT architecture is capable of achieving 21fps for a 64-bit memory, it performs on the lower end of the published DFT architectures. This is largely a result of state machine inefficiencies and unnecessary memory transactions.

Table 2: Comparison to FPGA-based DFTs

Arch.	Target FPGA	Image Size	Frames/sec
Shirazi et al.	2 x XC4000E	512x512	2.12
Dick	XC4000E	512x512	24
Dilon Eng.	2 xXC2V6000	2048x2048	120
Uzun et al.	XCV2000E	512x512	35
xADRT-SERIAL	V2P 2vp7	512x512	7.13
xADRT-PARA	V2P 2vp7	512x512	15.65
xADRT-PARA4	V2P 2vp7	512x512	21.23

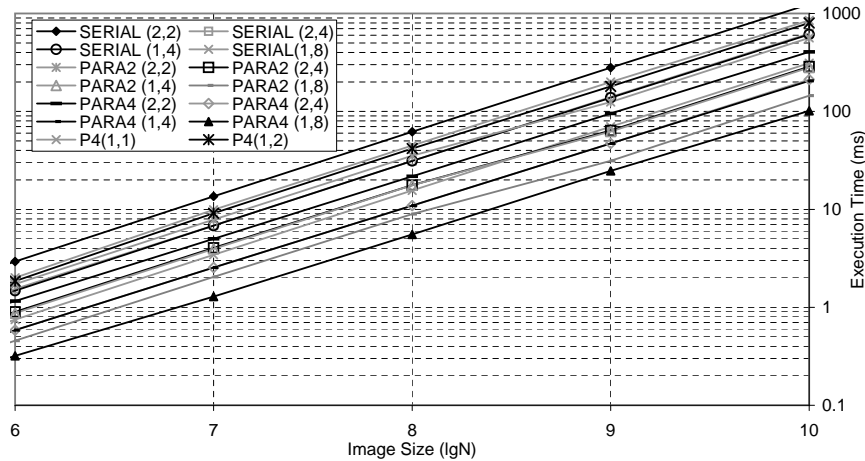


Figure 4: Execution speed (ms) vs. Image size (lgN)

Table 1: System Characterizations

Mem. Width	Freq. (MHz)			Area (V2P Slices)			%Optimal		
	SERIAL	PARA2	PARA4	SERIAL	PARA2	PARA4	SERIAL	PARA2	PARA4
32-bit	107.30	105.99	113.54	811.07	2661.43	2288.07	23.27	25.04	23.37
64-bit	106.62	104.27	111.70	1259.43	4556.71	3822.29	22.96	24.67	23.01

## 5 Future Work and Conclusions

In the future, this project will be integrated into a video vision system. First, the ADRT algorithm and current hardware implementation will have to be optimized to decrease the execution time of the system. The algorithm can be modified to discount pixels that are known to be 0, thus eliminating unnecessary memory transactions. The percent of optimal metric indicates that the RaCE still has room for improvement, mainly through optimization of state management processes and hardware description. Second, a system architecture will have to be identified and the current implementation adapted to it. This includes choosing a suitable memory interface capable of providing adequate bandwidth to the RaCE and the design of a caching system to hide data transfer latencies. Third, an inverse approximate discrete Radon (IADRT) transform algorithm will be developed. Finally, additional system components will be identified and integrated with the ADRT and IADRT.

The Radon transform has many applications in the computer, medical, and avionics fields. Currently, the real-time computation of a Radon transform is computationally expensive, making its use in video applications prohibitive. Through the use of the ADRT algorithm [1] and the optimizations presented in this paper, a solution to this problem has been developed in hardware known as the xADRT that can achieve a computation rate of 21 frames per second.

## References

- [1] M. L. Brady, "A Fast Discrete Approximation Algorithm for the Radon Transform," 1998 Society for Industrial and Applied Mathematics.
- [2] M. L. Brady and W. Yong, "Fast Parallel Discrete Approximation Algorithms for the Radon Transform." SPAA 1992.
- [3] Xilinx Virtex II Pro User Guide, [www.xilinx.com](http://www.xilinx.com).
- [4] MATLAB documentation, [www.mathworks.com](http://www.mathworks.com).
- [5] N. Shirazi et al, "Implementation of a 2-D Fast Fourier Transform on a FPGA-Based Custom Computing Machine," IEEE Symposium on FPGAs for Custom Comp. Mach., Sept. 1999.
- [6] C. Dick, "Computing Multidimensional DFTs Using Xilinx FPGAs," The 8th Int'l Conference on Signal Processing Applications And Technology, Sept. 1998.
- [7] T. Dillon, "Two Virtex-II FPGAs Deliver Fastest, Cheapest, Best High-Performance Image Processing System," Xilinx Xcell Journal 41, 2001.
- [8] I.S.Uzun et al, "FPGA Implementations of Fast Fourier Transforms for Real-Time Signal and Image Processing," Proceedings on the IEEE Int'l Conference on Field-Programmable Technology, Dec. 2003.