HARDWARE ACCELERATION FOR SPARSE FOURIER IMAGE RECONSTRUCTION

BY

QUANG SY DINH

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Adviser:

Professor Yoram Bresler

ABSTRACT

Several supercomputer vendors now offer reconfigurable computing (RC) systems, combining general-purpose processors with field-programmable gate arrays (FPGAs). The FPGAs can be configured as custom computing architectures for the computationally intensive parts of each application. In this paper we present an RC-based hardware accelerator for an important medical imaging algorithm: iterative sparse Fourier image reconstruction. We transform the algorithm to exploit massive parallelism available in the FPGA fabric. Our design allows different ways of chaining custom pipelined vector engines, so that different computations can be carried out without reconfiguration overhead. Actual runtime performance data show that we achieve up to 10 times speedup compared to the software-only version. The design is estimated to provide even more speedup on a next-generation RC platform.

To my family

ACKNOWLEDGMENTS

I would first like to thank my adviser, Prof. Bresler, whose valuable thoughts and vision have made this thesis possible. His guidance and encouragement are also a powerful inspiration to me. Second, I would like to thank Prof. Deming Chen for his support during my thesis writing. Thanks to the people in the Innovative Systems Laboratory of the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign for their support and equipment. And finally, thanks to my parents for all their love, support, and encouragement. The thesis is dedicated to them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Recent advances in FPGA technology have enabled the emerging field of reconfigurable computing (RC). Companies such as Celoxica, Mitrionics, and SRC Computers, now offer FPGA-based RC platforms and HLL-to-HDL compiler technology, enabling RC development using high-level languages [1], [2], [3]. By exploiting massive parallelism available in the FPGA fabric, certain types of applications can potentially run much faster on these RC platforms than on traditional computers.

Medical imaging is an important class of such applications. In fact, FPGA implementation of the well-established filtered backprojection algorithm (a fundamental image reconstruction algorithm) has been studied by several groups [4], [5], [6]. It shows that with FPGA implementation, they can achieve up to 100 times speedup [4].

In this thesis, we consider an analysis and implementation on reconfigurable hardware of a novel, newly developed and complex image reconstruction algorithm, recently presented in [7]. Our goal is to find a scalable implementation of the algorithm that maximizes parallelism, thus maximizing speedup, on our given target RC platform.

1.1 Fourier Image Reconstruction Background

In applications such as magnetic resonance imaging (MRI), the measured data are samples of the Fourier transform of the image, not directly of the image itself. The reconstruction problem is to recover the image from its measured Fourier samples. With sufficient number of measurements, the image can be obtained by simply applying the inverse Fourier transform.

In many practical situations, however, we want to be able to reconstruct the image from only a small number of Fourier samples. This would enable faster acquisition of the data, which is especially important in dynamic imaging applications such as cardiac MRI. However, with sparse sampling, the simple inverse Fourier reconstruction method produces only low-quality results. Nonetheless, Venkataramani and Bresler [8] showed that high quality sparse reconstruction is possible when certain conditions are met.

Building on the theoretical results in [8], Ye, Bresler, and Moulin [7] proposed a novel nonlinear iterative level-set-based reconstruction algorithm. To produce high-quality results, this algorithm executes many iterations to converge to the optimal solution. However, each iteration itself is the combination of two different algorithms. It is this complexity that makes the algorithm slow and computationally intensive.

Practical applications demand a fast and high-quality reconstruction, when large volumes of medical data are processed, or when real-time response is needed. One way to satisfy the speed requirement is by accelerating the algorithm with an RC implementation.

1.2 Related Applications on RCs

As mentioned before, the RC implementation of the filtered backprojection algorithm for speeding up medical image processing formation has been studied before in industry and academia [4], [5], [6]. Comparing to the backprojection algorithm, the level set reconstruction algorithm implemented in this work is more complicated (about 10 times longer in terms of C source code), and harder to parallelize. Indeed, backprojection belongs to the class of “embarrassingly parallel problems” – for which the computational graph is disconnected, making parallelization straightforward. In contrast, the iterative nature of our reconstruction algorithm makes it more challenging to parallelize.

The conjugate gradient (CG) algorithm is one important component in our reconstruction algorithm. Previous work on RC implementation of a conjugate gradient solver [9] only considered running the matrix multiplication operations on the

FPGA, which requires many bandwidth-limited data transfers between the FPGA and other host-based operations. In our design, we move the entire CG algorithm to the FPGA. Of course, we also need to address other issues, such as the scheduling of multiple operations and the partitioning of data into different memory banks.

1.3 Overview of the Work

Our contributions can be summarized as follows:

1. We carry out several mathematical transformations on the original level-set-based algorithm [7], so that it exhibits more parallelism and is better suited to FPGA implementation.

2. We develop a dynamic fixed-point scheme, so that we can get better precision at reduced bit-width.

3. We develop a method to find the maximum pipeline parallelism for this algorithm, which is extendable to other similar algorithms.

4. We design an efficient application-specific vector processor architecture that provides performance and scalability, and can be generalized to other applications.

In the next chapter, we present the essentials of the level-set-based reconstruction algorithm. Chapter 3 describes algorithm transformations, which also include our fixed-point scheme. Next, the parallel architecture is presented in Chapter 4. Finally we present the implementation results and conclusions.

CHAPTER 2

LEVEL-SET-BASED IMAGE RECONSTRUCTION

The goal of the algorithm is to reconstruct an image from sparse samples of its Fourier transform. Applications of this problem can be found in magnetic resonance imaging (MRI), synthetic aperture radar (SAR), and radio astronomy.

It was shown [7] that practical reconstruction from sparse Fourier samples is possible if the image consists of objects supported on a small unknown set D. Fortunately, such cases are commonly encountered for differential measurements, when only small parts of an object change between measurements.

In Fig. 1, for example, our image is mostly black (zero value pixels), except for four small regions. The union of the regions (where the pixels have non-zero values) is called the *support* of the image. This support can also be interpreted as the binary version of the image, where pixel values can only be either 0 or 1.

The reconstruction problem then becomes a nonlinear optimization problem, which can be solved by a gradient-based technique as summarized in the next section. For more detailed explanations, please refer to the original paper [7].

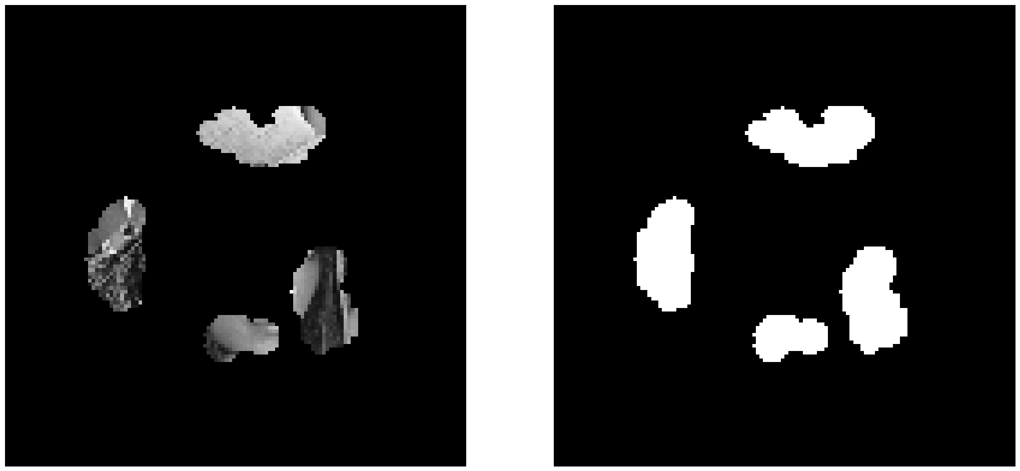

Fig. 1. An Image with Small Support (Left) and Its Support (Right)

2.1 Image Reconstruction Problem Formulation

Let $\hat{D}$ be the unknown image support, $\hat{v}(x)$ be the unknown pixel values ($x \in \hat{D}$), and Φ be the given (sparse) set of 2-D frequency sample locations. Then what we get from the measurements are the noisy measured samples:

$$y(\mathrm{f}) = F^{\hat{D}}\hat{v}(\mathrm{f}) + n \tag{1}$$

where $\mathrm{f} \in \Phi$ is the (frequency) location of a measured sample, $F^{\hat{D}}\hat{v}(\mathrm{f})$ denotes the 2-D Fourier transform of $\hat{v}(x)$ with the support $\hat{D}$, and n is the noise component.

Our goal is to find D, an estimate of $\hat{D}$, and v, an estimate of $\hat{v}$, to minimize the following cost function:

$$\mathrm{C}(D,v) = \frac{1}{2}\left\| y - F^D v \right\|_\Phi^2 + \lambda \int_\Gamma d\Gamma \qquad (2)$$

For $\lambda = 0$, minimization of the first term would find D and v that are least-square estimates of $\hat{D}$ and $\hat{v}$, respectively. However, because of the sparse measurements, these estimates are non-unique, and in practice would be grossly in error. The objective of the second term, $\lambda \int_\Gamma d\Gamma$, is to regularize the solution and make it unique and well-behaved, by penalizing the length of the boundary Γ of the support D with a regularization constant λ.

2.2 Outline of Reconstruction Algorithm

This nonlinear optimization problem can be solved by *alternating minimization* of the cost function with respect to the support D and the pixel values v separately. This iterative process is illustrated in Fig. 2.

```
Initialize $D_0$
for $k = 1$ to *Number_of_Iteration*
    **CG** step: Find $v_k$ to minimize $\mathrm{C}(D_{k-1}, v_k)$
    **LS** step: Find $D_k$ to minimize $\mathrm{C}(D_k, v_k)$

end for
```

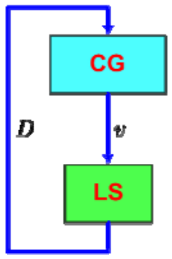

Fig. 2. Iterative Reconstruction Algorithm, Divided into CG and LS Steps

Each reconstruction iteration is separated into two different sub-problems. In the *CG step* (conjugate gradient algorithm, detailed in Section 2.4), given support D_{k-1} from the previous iteration, we find pixel values v_k that minimize the cost function. Then this v_k is used in the *LS step* (level-set-based algorithm, detailed in Section 2.3) to find an updated support D_k. *Number_of_Iteration* can either be a fixed value obtained from experiments or be determined adaptively.

We present these two minimization techniques in the next two sections. Because most of the computation takes place inside the CG steps, we mainly focus on this CG algorithm.

2.3 Level-Set-Based Algorithm

In this step, given a fixed v, we find support D to minimize the cost function $\mathrm{C}_v(D) = \mathrm{C}(D, v)$.

This algorithm is fully described in [7]. Here, we only mention the novel idea behind it: the use of the level-set method [10]. First, the support D is represented by a 2-D *level-set function* $\phi(x, y)$. Specifically, $D = \{(x, y) \mid \phi(x, y) \geq 0\}$, and the boundary of D is the zero level set $\Gamma = \{(x, y) \mid \phi(x, y) = 0\}$. Then, instead of directly manipulating the shape of D or its boundary Γ, we manipulate the level-set function ϕ to implicitly change D. The advantage of using the level-set method is that we can handle topological change (merging, splitting of regions) in D easily.

2.4 Conjugate Gradient (CG) Algorithm

In this step, given a fixed support D, we find pixel values v to minimize the cost function $\mathrm{C}_D(v) = \mathrm{C}(D, v)$.

This is the minimization of a quadratic function with large dimensions and therefore is preferably handled by the conjugate gradient algorithm. The essence of this algorithm is the CG iteration, which updates two 2-D matrices $\boldsymbol{r}$ and $\boldsymbol{d}$ according to the following equations:

$$\begin{aligned}
\alpha &= \left\langle \boldsymbol{r}_k, \boldsymbol{r}_k \right\rangle \Big/ \left\langle \boldsymbol{d}_k, \boldsymbol{d}_k \right\rangle \\
\boldsymbol{t} &= \boldsymbol{m}_D \circ \mathbf{iFFT}(\boldsymbol{d}_k) \\
\boldsymbol{r}_{k+1} &= \boldsymbol{r}_k + \alpha \boldsymbol{t} \\
\beta &= \left\langle \boldsymbol{t}, \boldsymbol{r}_{k+1} \right\rangle \Big/ \left\langle \boldsymbol{d}_k, \boldsymbol{d}_k \right\rangle \\
\boldsymbol{d}_{k+1} &= \beta \boldsymbol{d}_k - \boldsymbol{m}_\Phi \circ \mathbf{FFT}(\boldsymbol{m}_D \circ \boldsymbol{r}_{k+1})
\end{aligned} \tag{3}$$

where matrix $\boldsymbol{r}$ is the residual, matrix $\boldsymbol{d}$ relates to the search direction, $\boldsymbol{m}_D$ and $\boldsymbol{m}_\Phi$ are binary matrix representations of D and Φ, $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$ denotes vector inner product, $\circ$ denotes element-wise matrix multiplication, and α and β are scalar values.

The simplified data flow graph of the equations in (3) for one CG iteration step is shown in Fig. 3. In this diagram, the thick arrows represent matrices, while the thin arrows represent scalar values. Matrix $\boldsymbol{r}$ is updated through the left path, and matrix $\boldsymbol{d}$ is updated through the right path.

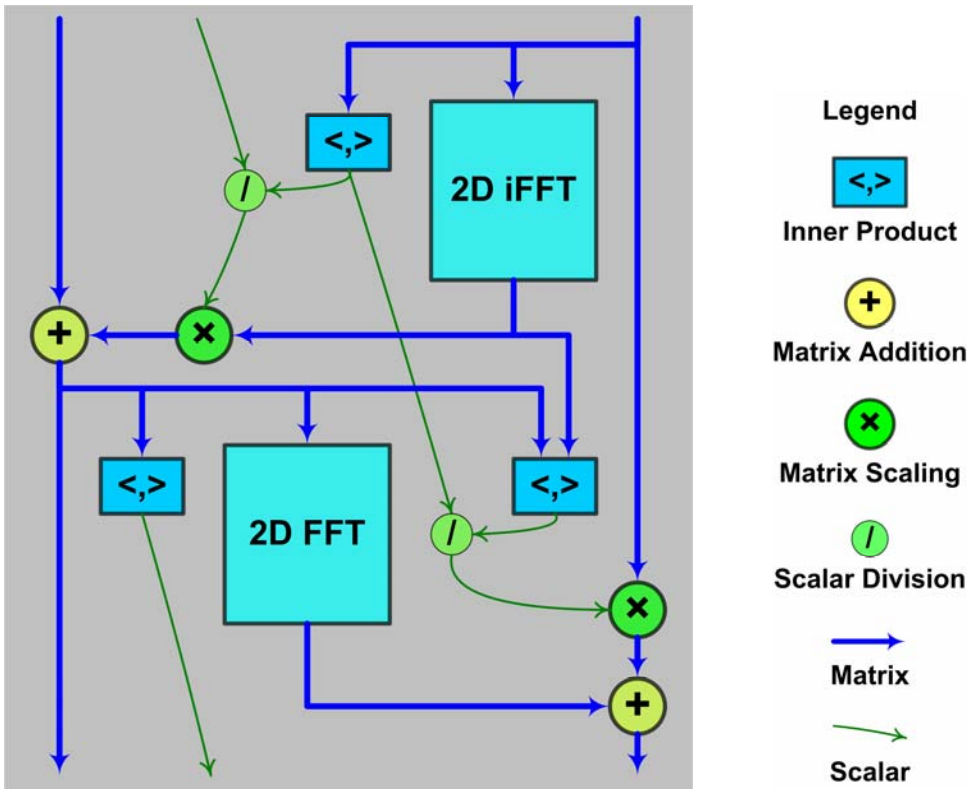

Fig. 3. Simplified Dataflow Graph of One CG Iteration

The CG iteration is where most of the computation in our reconstruction algorithm takes place. On average, one reconstruction iteration requires the computation of about five CG iterations. A typical reconstruction needs about 100 iterations, which requires about 500 CG iterations. With two 2-D FFT operations per iteration, the overall complexity is $O\left(N^2 \log_2 N\right)$ for image size $N \times N$. For our image size of 128×128, this totals about 0.3 GFLOP (giga-floating-point operations) for one reconstruction.

Because this is such a crucial step in terms of computational workload, we mainly focus on the CG iterations in the following chapters.

CHAPTER 3

ALGORITHM TRANSFORMATION

Directly mapping the algorithm to RC hardware will not result in an efficient implementation. Equations may be mathematically transformed into functionally equivalent forms that are more parallelizable or more suitable to FPGA implementation. Another important modification is using fixed-point arithmetic, which is faster and consumes fewer resources than floating-point arithmetic. We present an example of mathematical transformations in Section 3.1 and an interesting fixed-point scheme in Section 3.2.

3.1 Mathematical Transformation

In this section, we present one of several mathematical transformations that we have discovered. This particular transformation makes the algorithm more FPGA-implementation friendly.

One of the key calculations (required for each element of the image) in the level-set-based algorithm is the following expression:

$$\frac{ax+by}{\sqrt{a^2+b^2}} \tag{4}$$

where a, b, x, and y are derived from adjacent values of the 2-D level-set function. If implemented directly, this would require 4 multipliers, 1 square root, and 1 division, which would take up a significant amount of FPGA logic. The CORDIC algorithm [11], which requires only shift and add operations, can be used to efficiently handle the square root, but we are still left with divisions and multiplications.

If we define $\rho = \arctan(b/a)$, then (4) can be written as $x\cos\rho + y\sin\rho$. This turns out to be the result when we rotate vector (x,y) by the angle ρ. Both computing angle ρ and rotating vector (x,y) can be implemented in CORDIC. Therefore, by doing two CORDIC vectoring mode operations in locked-step (same rotation angle ρ), we can calculate the expression without any divisions or multiplications. We start with two vectors, (a,b) and (x,y). Applying the CORDIC vector mode operations to (a,b), we end up with vector $\left(\sqrt{a^2+b^2},0\right)$. Then the same sequence of rotation steps applied to vector (x,y) will produce $\left(\frac{ax+by}{\sqrt{a^2+b^2}},\frac{ay-bx}{\sqrt{a^2+b^2}}\right)$, providing us the value in (4).

3.2 Dynamic Fixed-Point Scheme

To produce an efficient FPGA implementation, fixed-point arithmetic should be used. A floating-point implementation would be considerably slower and use much more FPGA resources, limiting parallelism. Obviously, narrow bit widths are preferred because they reduce logic consumption and allow a faster clock. On the other hand, we need sufficient bit widths to achieve adequately precise results. We determine the allowable quantization level by software simulation of the fixed-point implementation.

Because we are trying to minimize bit-widths while maintaining adequate precision, close study of the CG algorithm leads to an important discovery. In the CG algorithm, as we converge to the optimal solution, the magnitudes of elements in matrix $\boldsymbol{r}$ and $\boldsymbol{d}$ get smaller and smaller after each iteration. At the same time, these elements get more and more accurate, which requires more fractional bits. We can exploit this behavior of the changing of both dynamic range and accuracy of $\boldsymbol{r}$ and $\boldsymbol{d}$ to improve the efficiency of their fixed-point representation.

An illustration is shown in Fig. 4. At the beginning, $\boldsymbol{r}$ and $\boldsymbol{d}$ have small scaling factors (fewer bits after the radix point) so that no overflows occur with the chosen bit-width. Note that at this stage we do not need high precision for $\boldsymbol{r}$ and $\boldsymbol{d}$ values. After each CG iteration, because of smaller elements, we can increase the scaling factors (more bits after the radix point) without causing overflows. With increased scaling factors, $\boldsymbol{r}$ and $\boldsymbol{d}$ are represented more precisely after each CG iteration.

In the simplified example in Fig. 4, the initial values take values up to 2^1 in iteration 1, while the final values requires precision of 2^{-3} in iteration 3. The straight-forward static fixed-point implementation would require 5-bit data. With dynamic fixed-point scheme, we can use only 3-bit data and still meet the precision requirement in the final results.

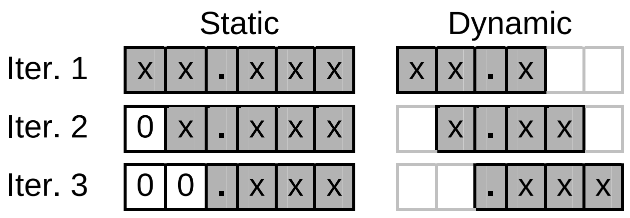

Fig. 4. Static vs. Dynamic Fixed-Point Schemes

Thus, comparing to a simple static fixed-point scheme that has constant scaling factors, our dynamic scheme needs fewer bits for the same required accuracy by adjusting the scaling factors after each CG iteration.

CHAPTER 4

PARALLEL ARCHITECTURE DESIGN

After the algorithm has been transformed into a form suitable for RC implementation, our goal is to design a hardware architecture that maximizes the available parallelism, given the constraints of the target RC platform. We also consider scalability issues for future extensions.

4.1 Target RC Platform

The SRC-6E is a commercial reconfigurable computing platform from SRC Computers Inc. [3]. This platform has a typical RC architecture, which comprises user FPGAs, on-board memory banks, and DMA link to a traditional computer host. SRC's Carte programming environment provides a library to handle host-to-FPGA data communication and other necessary details. This allows us to focus on mapping the algorithm to FPGA.

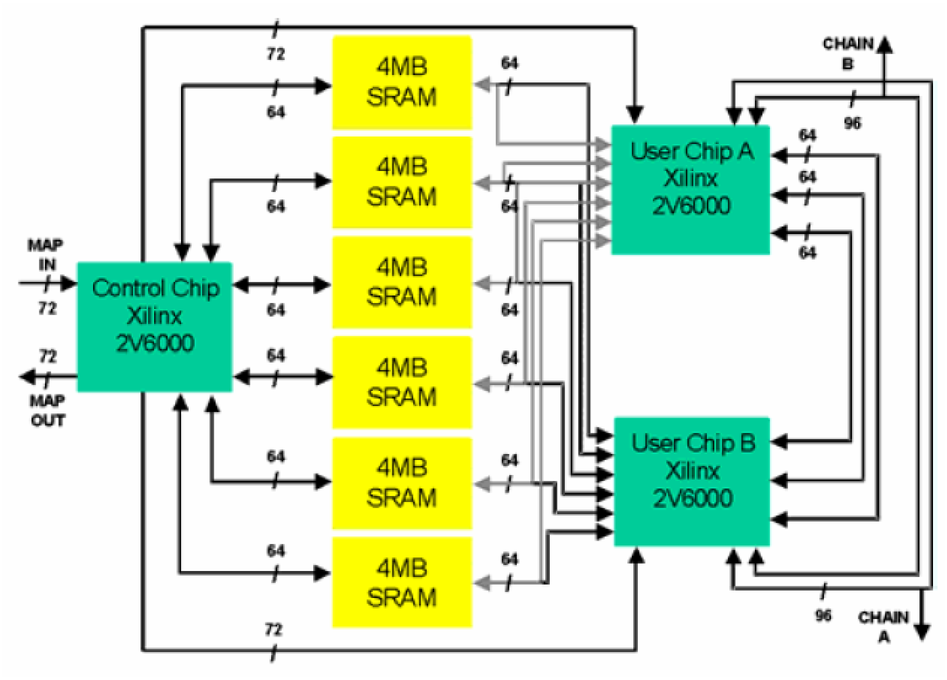

Fig. 5. SRC-6E Hardware Architecture

The SRC-6E contains two Xilinx Virtex II FPGAs (xc2v6000) running at 100 MHz. Each FPGA can support massive parallelism: about 33000 logic slices (each contains two 4-input LUTs), 144 18-kbit block RAM, and 144 18×18 multipliers [12]. The SRC-6E board also provides six independently addressable SRAM memory banks with a total capacity of 24 MB and a total bandwidth of 48 bytes per clock cycle (Fig. 5).

4.2 Parallelization Approach

There are two basic parallel models that we can use to implement our CG iteration. The first model is the *pipeline model* (Fig. 6a). In this model, sequential operations are executed concurrently on different processing blocks. Partial results from one block are forwarded to the next block. This approach has very efficient I/O usage, only at the first and last operations of the chain. There are, however, pipeline barriers, where operations cannot be pipelined.

The second model is the *loops distributed model* (Fig. 6b). In this model, we simply duplicate the processing blocks. This approach has heavy I/O usage: the more blocks, the greater the I/O needed to supply data to those blocks. In addition, the data feeding into each block have to be independent.

To maximize parallelism under I/O constraints, we use a hybrid model, the combination of the two mentioned above (Fig. 6c). First, we try to pipeline as much as possible, so that we do as many calculations as possible with an I/O operation. Then we use the loops distribution model to duplicate our processing blocks until all available I/Os are used up.

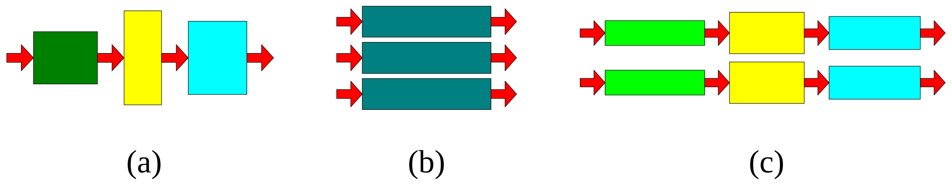

Fig. 6. (a) Pipeline Model, (b) Loops Distributed Model, and (c) Hybrid Model

The most computationally intensive operations in the CG iteration are the two 2-D FFTs. One way to carry out an $N×N$ 2-D FFT is to separate it into N row-wise 1-D FFTs, followed by N column-wise 1-D FFTs [13]. The ordering is interchangeable: we can also do the column-wise FFTs first, then do the row-wise FFTs. This flexible ordering allows us to achieve more parallelism, as shown later in Section 4.3.

In the CG iteration, this separated implementation is the most efficient way because we then have N independent rows (or columns) that can easily be distributed across identical processing blocks. Furthermore, each row (or column) has only N data points, small enough to be stored in the on-chip block RAM of the FPGA. With this data locality, we can pipeline rows (or columns) between operations.

An example is shown in Fig. 7. Assume that we have three different operations **F**, **G**, and **H** to sequentially operate on N independent rows, from row 0 to row $N–1$. Using the loops distribution model, we can have two pipelines, one working on even rows and the other working on odd rows. In each pipeline, one row result from block **G** can be forwarded immediately to block **H**, without having to wait for results of other rows. So, after **G** finishes with data originating from row 0, the result is forwarded to **H**. Now **G** can work on data originating from the next row (row 2) while **H** is working on data originating from row 0. Thus, pipeline operation at row-level is realized.

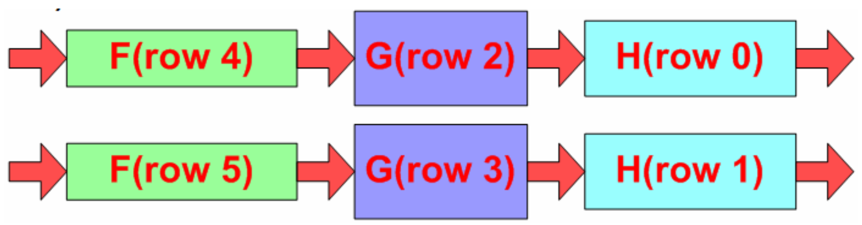

Fig. 7. Parallelism at Row Level

For actual implementation, the separated FFTs have a regular structure that can be mapped into hardware easily. The row-wise FFT and the column-wise FFT can be executed by the same FFT hardware block, reducing logic usage. Another benefit is that the pipelined 1-D FFT IP core is already available.

4.3 Maximum Pipelines

Although we can pipeline the operations at the row- and column-level, some sequences of operations cannot be pipelined together. There are three vector inner products, each produces a single scalar value based on all $N\times N$ elements. Therefore, single row or column data from the previous pipeline stage is not enough, and the inner products cannot be pipelined with following operations. Similarly, the two pairs of row and column 1-D FFTs cannot be pipelined together, as we need data from all *N* rows before we can compute FFT for the first column.

If the ordering of both the inverse FFT (iFFT) and the forward FFT is the same—Row followed by Column (refer to Fig. 8)—then another pipeline barrier exists: the Row FFT cannot start until all column data are available from the Column iFFT. But if we switch the ordering of the forward FFT to Column followed by Row, then the Column FFT can start immediately when it receives a single column from the Column iFFT. This means we can put these Column iFFT and FFT into two pipeline stages of the same pipeline.

By grouping the five pipeline breaks into two barriers, as shown in the top two dark bands in Fig. 8, we can get maximum pipelines. For example, consider the top dark band separating two pipelines. When the first pipeline finishes, the last result from block *Row iFFT* is written to the transposition memory. At that time, the inner product block also outputs its correct result. Only then can we execute the scalar division and start the second pipeline.

We pipeline results from the end of the previous CG iteration to the beginning of the next CG iteration, as shown near the bottom of Fig. 8. In this way, we are able to reduce to two pipelines per CG iteration. Each pipeline has two stages corresponding to the iFFT and FFT operations.

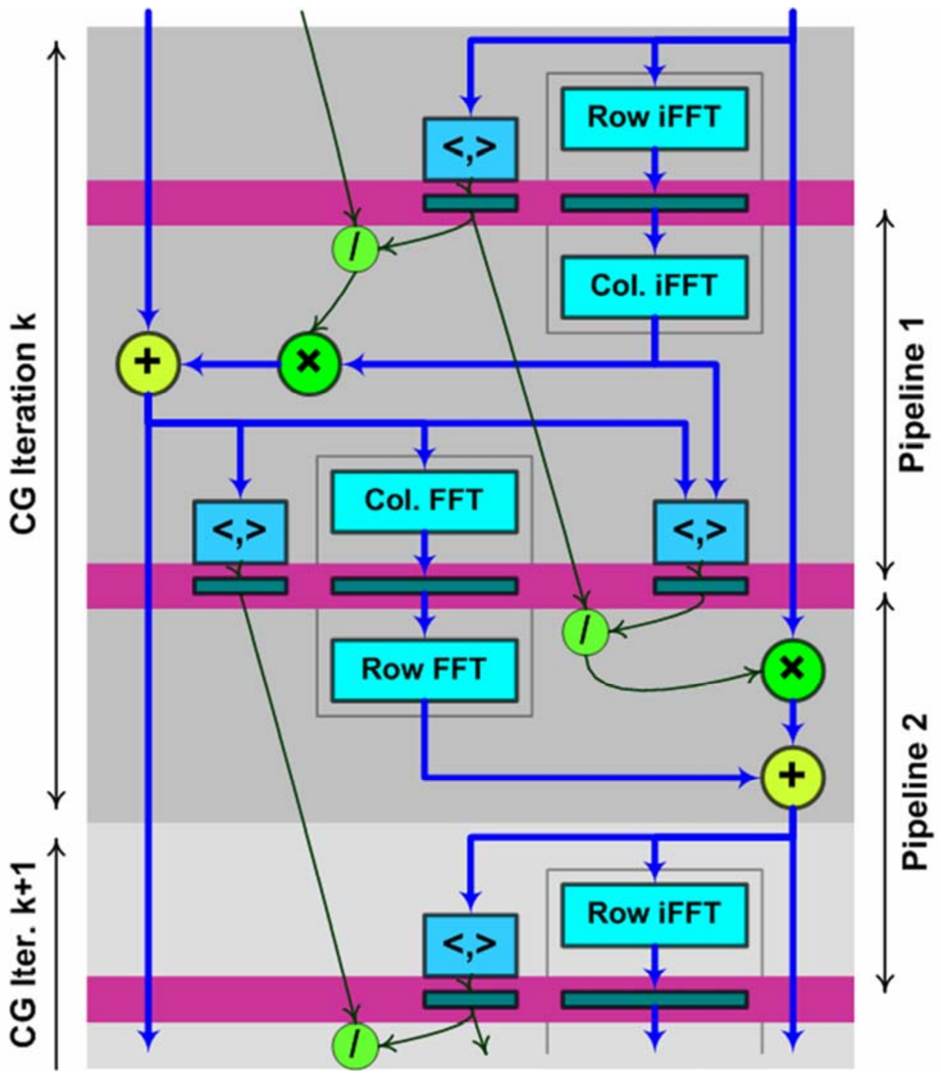

Fig. 8. Pipelines and Barriers

The procedure to determine the maximum pipelines can be summarized as follows:

1. Identify all non-pipelinable operations.

2. Group these operations into as few pipeline barriers as possible.

3. The maximum pipelines are all the operations between those barriers.

In the final implementation, the platform resource can support two such hardware pipelines: one pipeline works on even rows/columns, the other works on odd rows/columns. This is the hybrid model illustrated in Fig. 7. The limiting factor here is the available I/O bandwidth between the FPGA and the on-board memory banks.

Figure 9 shows the timing diagram of one CG iteration. In this figure, c0 denotes column 0, r(*N*-1) denotes row *N*–1. Two pipelines stages, iFFT and FFT, are shown for each hardware pipeline, even and odd. The horizontal axis shows the sequences of operations in time. In a CG iteration, algorithm pipeline 1 operates on *N* columns, then algorithm pipeline 2 operates on *N* rows.

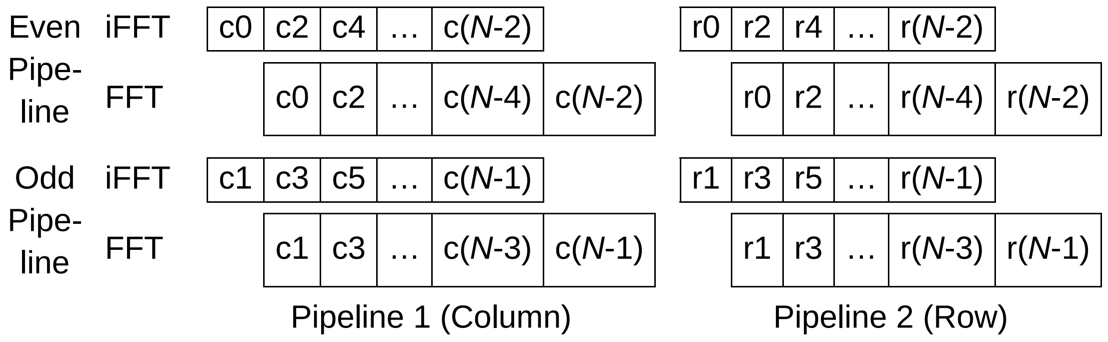

Fig. 9. Pipeline Timing Diagram for One CG Iteration

4.4 Datapath

With the pipelines determined, we can map both Pipeline 1 and 2 to one single hardware datapath (Fig. 10). This is because the two pipelines operate sequentially and they are nearly identical: each has two FFT blocks at input and output, a vector scaling block feeding into a vector addition block, and inner product blocks. The slightly different data flows of the pipelines are enabled by the two MUXes embedded in the datapath.

Our architecture can be viewed as an application-specific vector processor. The datapath works on data of vector type. Different vector instructions correspond to different MUX configurations that execute different data flows. The datapath is highly customized to have specific functional units (FFTs, inner products) and dedicated links between these functional units.

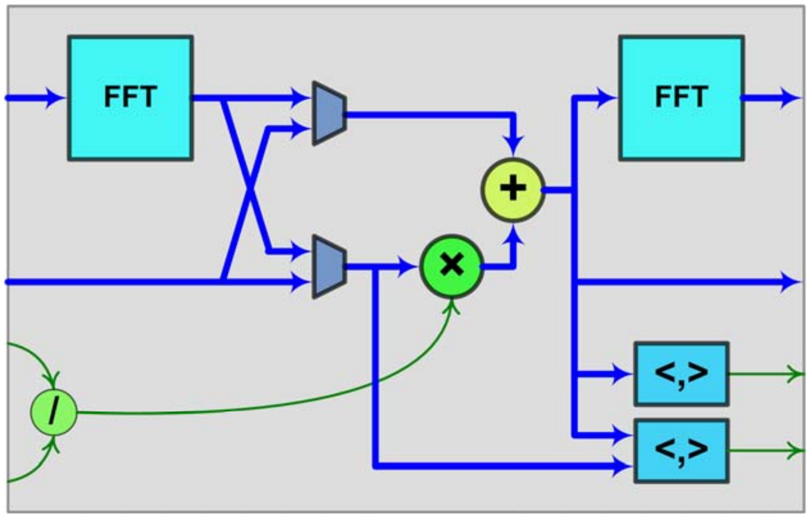

Fig. 10. Datapath Design

4.5 Memory Assignments

Due to their size, the matrix variables in the CG iterations cannot fit into the on-chip block RAM. So they are stored in the on-board memory banks of the SRC-6E. To be able to feed data to the pipeline at maximum rate, we need to consider the flow of data among variables as well as the memory I/O constraints of the underlying platform.

The SRC-6E has six on-board memory banks, each of which can be addressed and read/written independently. Each memory bank is 64-bit wide, and the throughput is one read or write access per clock. The access latency, however, is several clocks, which makes switching between reads and writes very costly (7 clocks are required). Therefore, optimal usage of the on-board memory is to fix the access type to each memory bank during the execution of a pipeline. In this way, we can utilize the maximum memory bandwidth of one access per clock. The problem is how to assign the variables in the CG iterations to different memory banks so that we can get this desired behavior.

With the pipelines and datapath described in previous sections, each pipeline requires reads from two matrix variables and writes to two matrix variables. The scalar values are required only once per pipeline and can be ignored. The detailed access patterns of these variables are depicted in the left side of Fig. 11. In this figure, the rounded shapes represent the variables, and the large rectangles represent the pipelines. For example, CG iteration 1 comprises pipeline 1a (column pipeline) and pipeline 1b (row pipeline). This iteration reads in $\boldsymbol{r}_1$ and $\boldsymbol{d}_1$, and writes out $\boldsymbol{r}_2$ and $\boldsymbol{d}_2$. There is also a temporary variable between adjacent pipelines, and this is the transposition memory for the 2-D FFTs.

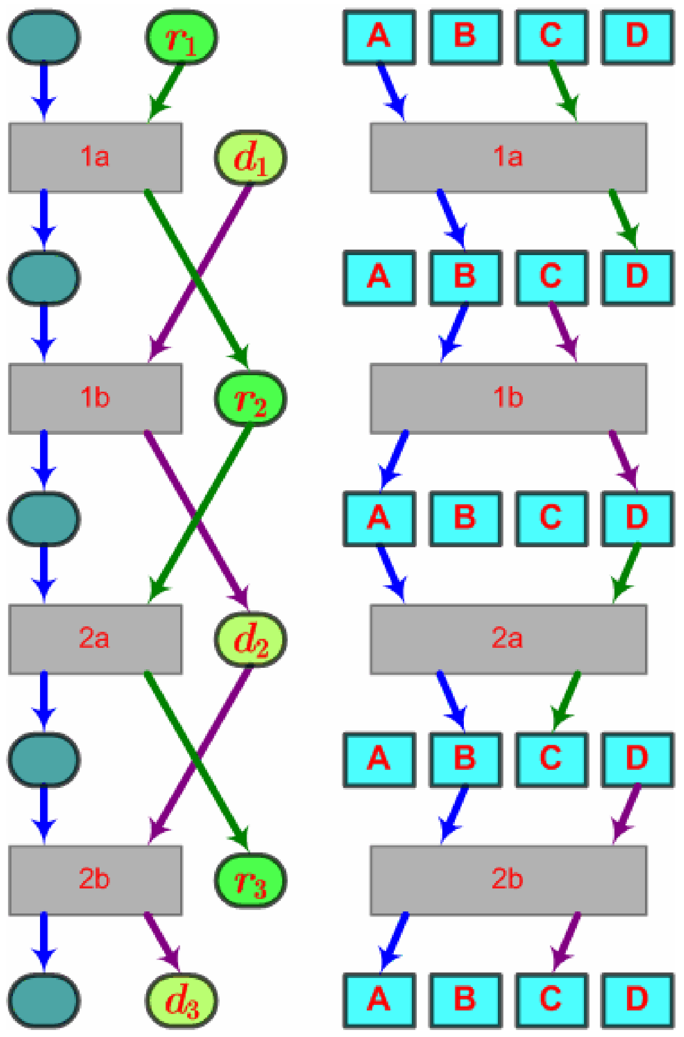

Fig. 11. Variables Flow (Left) and Assignment (Right)

With the pipeline requiring access to 4 different variables, and the 6 memory banks available, we can assign each matrix variable to one memory bank. We cannot use all 6 banks because we cannot share one bank to two variables (this would creates conflicts in later pipelines due to the flow of variables). Because each data element requires 32 bits and one memory bank is 64-bit wide, we can fetch two data elements per clock to feed the odd and even hardware pipelines, as mentioned in Section 4.2.

The variable assignments for our implementation are detailed in Fig. 11, right side. Four memory banks, A, B, C, D, are used to store the variables. In pipeline 1a, $\boldsymbol{r}_1$ is read from bank C, and $\boldsymbol{r}_2$ is written to bank D. Then in pipeline 2a, $\boldsymbol{r}_2$ is read from bank D, and $\boldsymbol{r}_3$ is written to bank C. Meanwhile, the temporary variable is alternated between banks A and B.

4.6 Scalability Analysis

As mentioned in Section 4.5, the current implementation is bound by the available memory I/O bandwidth in the SRC-6E platform. To evaluate the design on different configurations, we can consider an image size $N \times N$ and I/O bandwidth B.

One CG iteration accesses $O(N^2)$ data elements at bandwidth B. Therefore, throughput for one CG iteration is $O(B/N^2)$. If we can double the bandwidth, we will be twice as fast. On the other hand, doubling the image dimensions to $2N \times 2N$ will make runtime four times longer.

The multipliers inside the FFT consume the most logic in the FPGA. Each FFT block uses $O(\log_2 N)$. The number of pipeline duplications is $O(B)$, so the total number of concurrent FFT multipliers is $O(B \times \log_2 N)$. This is roughly the resource requirement. We can have more parallelism with either larger images, or increased I/O bandwidth.

CHAPTER 5

IMPLEMENTATION RESULTS

Table 1 shows the runtimes for different image sizes. The total runtime on the RC platform (with FPGAs clocked at 100 MHz) includes time for configuring the FPGAs and time for transferring data between the host and the FPGAs. The software version is an optimized floating-point implementation running on a 2.5 GHz Pentium IV PC with 1 GB RAM and 512 kB cache (due to scaling overhead, software fixed-point implementation is slower than floating-point implementation).

Table 1. Runtime Comparison

Image Size	Software (s)	RC platform (s)
128×128	2.0	0.2
256×256	9.5	0.8

Because the image size of 512×512 does not fit into the on-board memory of the SRC-6E, and the FFT requires the size to be power of 2, only resolutions of 128×128 and 256×256 are tested.

Table 2 shows the amount of FPGA resource used by our implementation. We use only one of the two Vertex xc2v6000s available, because the parallelism is bound by the available memory I/O bandwidth (48 bytes/clock), not the available logic resource.

Table 2. FPGA Resource Usage

Slices	60% (20000)
Multipliers	60% (88)
Block RAMs	30% (42)

On the more advanced SRC-7 RC platform, with a higher FPGA clock (150 MHz) and more memory bandwidth (160 bytes per clock cycle), we estimate that the RC-based design will achieve more than a 4-fold speedup over the current SRC-6E implementation and still fit into the available FPGAs.

Figure 12 shows the results from a test run of the implemented algorithm on the RC platform. On top, from left to right, are the support of the image, the reconstructed image (closely resembles the original image), and the original image. At the bottom are the final level-set function, the simple inverse Fourier reconstruction result (virtually unusable), and the plot of errors vs. reconstruction iterations.

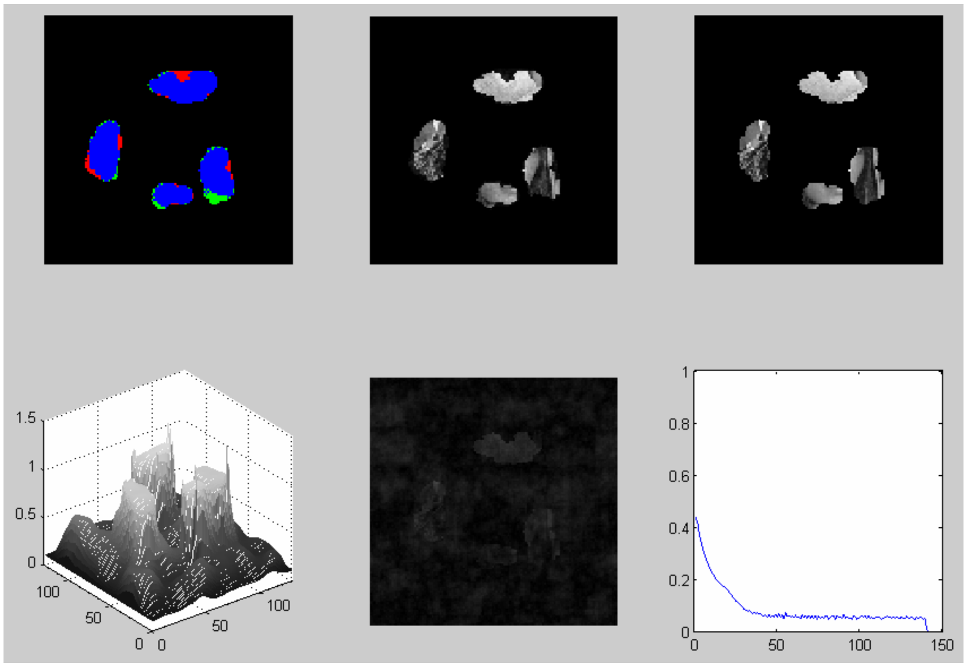

Fig. 12. Result from a Test Run of the Reconstruction Algorithm

CHAPTER 6

CONCLUSION

This study shows that the level-set-based image reconstruction algorithm can be implemented on an RC platform with much higher performance than software-only implementation. Several insights about how to efficiently map the algorithm on to FPGA hardware have been discovered at various stages. Our application-specific vector processor architecture can be generalized for similar types of imaging applications that involve many large vector and matrix operations.

REFERENCES

[1] Celoxica Ltd., Oxfordshire, UK. An innovative architecture, Oct. 2007. [Online]. Available: http://www.celoxica.com/ technology/architecture.html

[2] Mitrionics Inc., CA. Accelerated Computing, Sep. 2007. [Online]. Available: http://www.mitrionics.com

[3] SRC Computers Inc., CO. Delivering programmer-friendly reconfigurable processor-based computers, Aug. 2007. [Online]. Available: http://www.srccomp.com

[4] M. Leeser, S. Coric, E. Miller, H. Yu, and M. Trepanier, “Parallel-beam backprojection: An FPGA implementation optimized for medical imaging,” *Journal of VLSI Signal Processing Systems*, vol. 39, no. 3, pp. 295-311, Mar. 2005.

[5] N. Sorokin, “An FPGA-based 3D backprojector,” Ph.D. dissertation, Universitat des Saarlandes, Germany, 2003.

[6] D. Stsepankou, K. Kommesser, J. Hesser, and R. Manner, “Real-time 3D cone beam reconstruction,” *IEEE Nuclear Science Symposium Conference Record*, vol. 6, pp. 3648-3652, Oct. 2004.

[7] J. Ye, Y. Bresler, and P. Moulin, “A self-referencing level-set method for image reconstruction from sparse Fourier samples,” *International Journal of Computer Vision*, vol. 50, no. 3, pp. 253-270, Dec. 2002.

[8] R. Venkataramani and Y. Bresler, “Further results on spectrum blind sampling of 2D signals,” in *Proceedings of IEEE International Conference on Image Processing*, 1998, pp. 752-756.

[9] G. Morris, V. Prasanna, and R. Anderson, “A hybrid approach for mapping conjugate gradient onto an FPGA-augmented reconfigurable supercomputer,” in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006, pp. 3-12.

[10] J. Sethian, *Level Set Methods and Fast Marching Methods*. New York, NY: Cambridge University Press, 1996.

[11] J. Duprat and J. Muller, “The CORDIC algorithm: New results for fast VLSI implementation,” *IEEE Transactions on Computers*, vol. 42, no. 2, pp. 168-178, Feb. 1993.

[12] Xilinx Inc., CA. Virtex-II Complete Data Sheet, Jun. 2007. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds031.pdf

[13] J. Proakis and D. Manolakis, *Introduction to Digital Signal Processing*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1996.