Hardware Acceleration for Sparse Fourier Image Reconstruction

Quang Dinh, Yoram Bresler, Deming Chen
Department of Electrical and Computer Engineering, University of Illinois at Urbana Champaign
Email: {qdinh2, ybresler, dchen}@uiuc.edu

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.

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 paper, we consider an analysis and implementation on reconfigurable hardware of a 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 paper 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 transfer 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 This Paper

Our contributions can be summarized as follows:

(a) 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.

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

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

(d) We design an efficient Application-Specific Vector Processor architecture that provides performance and scalability, and can be generalized to other applications.

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

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 can be commonly encountered for differential measurements, when only small parts of an object change between measurements.

In Figure 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.

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

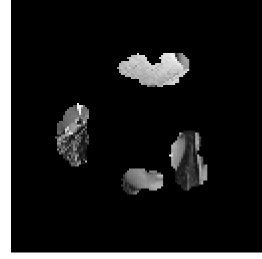

Figure 1. Image with Small Support

2.1 Outline of the Reconstruction Algorithm

Let $\hat{D}$ be the unknown image support.

Let $\hat{v}(x)$ be the unknown pixel values ($x \in \hat{D}$).

Let Φ be the known (sparse) set of 2-D frequency sample locations.

Then what we get from the measurements are the noisy measured samples:

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

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:

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

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 would be in gross error in practice. 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.

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 process is shown in the following pseudo-code:

Initialize D_0
for $k = 1$ to *Number_of_Iteration*
  Find v_k to minimize $\mathrm{C}\left(D_{k-1}, v_k\right)$
  Find D_k to minimize $\mathrm{C}\left(D_k, v_k\right)$
end for

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.

2.2 The Level-Set-based Algorithm

In this step, given a fixed v, we find support D to minimize the cost function $C_v(D) = C(D, v)$.

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.

2.3 Conjugate Gradient (CG) Algorithm

In this step, given a fixed D, we find pixel values v to minimize the cost function $C_D(v) = C(D, v)$.

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

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

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; $\circ$ 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 Figure 2. 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.

This is where most of the computation in our reconstruction algorithm takes place. A typical reconstruction needs about 500 CG iterations. With two 2-D FFT operations per iteration, the overall complexity is $O(N^2 \log_2 N)$ for image size $N \times N$. For our image size of 128×128, this totals up to 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 sections.

Figure 2: Dataflow Graph of One CG Iteration

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 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 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 multiplica-tions.

We find out that by doing two CORDIC vectoring mode operations in locked-step, we can calculate the result 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 the 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 more resources, which limits parallelism. Obviously, narrow bit widths are preferred because they reduce logic consumption and allow faster clock. On the other hand, we need sufficient bit widths to achieve adequate-precision 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, elements in matrix $\boldsymbol{r}$ and $\boldsymbol{d}$ get smaller and smaller after each iteration. We can exploit this behavior of the changing dynamic ranges of $\boldsymbol{r}$ and $\boldsymbol{d}$ to improve the efficiency of their fixed-point representation.

An illustration is shown in Figure 3. 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. 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.

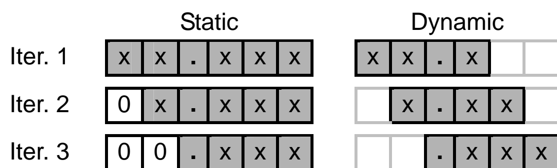

Figure 3. Static and Dynamic Fixed-Point Schemes

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.

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 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.

The SRC-6E contains two Xilinx Virtex II FPGAs (xc2v6000) running at 100MHz. Each FPGA can support massive amount of parallelism: about 33000 logic slices (each contains two 4-input LUTs), and 144 18×18 multipliers [13]. There are 6 independent SRAM memory banks with a total capacity of 24MB and a total bandwidth of 48 bytes per clock cycle (Figure 4).

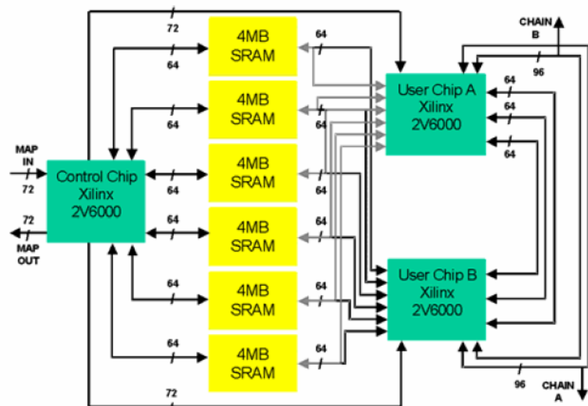

Figure 4. SRC-6E Hardware Architecture

4.2 Parallelization Approach

There are two basic parallel models that we can use for our CG iteration. The first is the *pipeline model* (Figure 5a). 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* (Figure 5b). 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 (Figure 5c). 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.

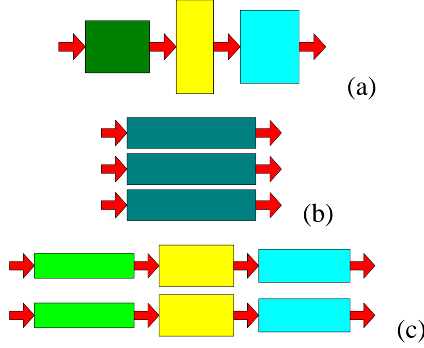

Figure 5. (a) Pipeline Model,
(b) Loops Distributed Model, and (c) Hybrid Model

The most computationally intensive operations in the CG iteration are the two 2-D FFTs. One way to carry out an $N \times N$ 2-D FFT is to separate it into N row-wise 1-D FFTs, and then N column-wise 1-D FFTs [12]. In the CG iteration, this 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. With this data locality, we can pipeline rows (or columns) between operations.

An example is shown in Figure 6. Assume that we have three different operations **F**, **G**, and **H** to 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 to block **H**. 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.

Figure 6. 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 block, reducing hardware 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 can not be pipelined together. The three inner products produce scalar values, thus can not be pipelined with following operations. The two pairs of row and column FFTs also cannot be pipelined together, as they require a transposition memory in between.

By grouping these five pipeline breaks into two barriers, as shown in the top two dark bands in Figure 7, 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 Figure 7. In this way, we are able to reduce to 2 pipelines per CG iteration.

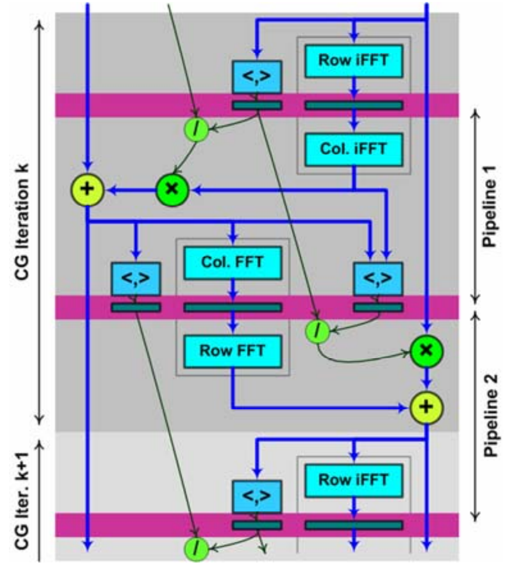

Figure 7: Pipelines and Barriers

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

(1) Identify all non-pipelinable operations.
(2) Group those operations into as few pipeline barriers as possible.
(3) The maximum pipelines are all the operations between those barriers.

With the pipelines determined, we can map them to one single datapath (Figure 8). This is because the two pipelines are relatively similar: 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 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 MUXes 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.

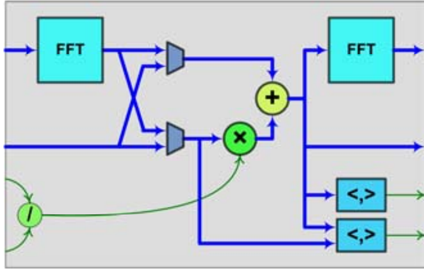

Figure 8: Datapath Design

4.4 Scalability Analysis

For an image size $N \times N$ and I/O bandwidth B, we have:

Throughput for 1 CG iteration is $O(B/N^2)$. We have linear speed-up with respect to I/O bandwidth: with double the I/O bandwidth, the design runs twice faster.

The number of concurrent FFT multipliers is $O(B \times \log_2 N)$. This is also our resource requirement. We can have more parallelism with larger images, or with increased I/O bandwidth.

5. Implementation Results

Table 1 shows the runtimes for different image sizes. The total runtime on the RC platform (with FPGAs clocked at 100MHz) 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 2.5GHz Pentium IV PC with 1GB RAM and 512KB 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

Table 2 shows the amount of FPGA resource used by our implementation. We use only one of the two Vertex xc2v6000s available, because the algorithm is bound by the available memory I/O bandwidth (48 bytes/clock).

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 (150MHz) 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.

6. Conclusion

This study shows that the level-set-based image reconstruction algorithm can be implemented on RC platform with much higher performance compared to software-only implementation. Several insights on 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 type of imaging applications that involve many large vector and matrix operations.

Acknowledgment

The SRC-6E platform and development environment are supported by the Innovative Systems Laboratory, National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana Champaign. The work of Yoram Bresler on this project has been supported by an NCSA Faculty Fellowship. The work of Quang Dinh has been supported by a VEF Fellowship.

References

[1] Celoxica Ltd., http://www.celoxica.com

[2] Mitrionics Inc., http://www.mitrionics.com

[3] SRC Computers Inc., 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*, 39, p.295 (2005).

[5] N. Sorokin, "An FPGA-Based 3D Backprojector," *PhD thesis*, Universitat des Saarlandes (2003).

[6] D. Stsepankou, K. Kommesser, J. Hesser, R. Manner, "Real-time 3D cone beam reconstruction," *IEEE Nuclear Science Symp. Conf. Record*, 6, p.3648 (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*, 50, p.253 (2002).

[8] R. Venkataramani and Y. Bresler, "Further results on spectrum blind sampling of 2D signals," *Proc. IEEE Int. Conf. Image Processing*, 2, p.752 (1998).

[9] G. Morris, V. Prasanna, and R. Anderson, "A Hybrid Approach for Mapping Conjugate Gradient onto an FPGA-Augmented Reconfigurable Supercomputer," *Proc. of the IEEE Symp. on Field-Programmable Custom Computing Machines*, 0, p.3 (2006).

[10] J. Sethian, *Level Set Methods and Fast Marching Methods*. Cambridge University Press (1996).

[11] J. Duprat and J. Muller, "The CORDIC Algorithm: New Results for Fast VLSI Implementation," *IEEE Transactions on Computers*, 42, p.168 (1993).

[12] J. Proakis and D. Manolakis, *Introduction to Digital Signal Processing*, Prentice Hall, 3rd Ed. (1996).

[13] Xilinx Inc., http://www.xilinx.com