A Reconfigurable Real-time Reconstruction Engine for Parallel MRI

Ishaan L. Dalal
Fred L. Fontaine

Technical Report CU/EE/S-TR07M02

Center for Signal Processing, Communications and Computer Engineering Research [S*ProCom²]

Department of Electrical Engineering
The Cooper Union
51 Astor Pl., New York, NY 10003.

This paper won First Prize at the IEEE Region 1 Graduate Student Paper Contest Farmingdale State University May 4, 2007, Long Island, NY.

S*ProCom²
January 2007
CONTENTS

I  Introduction ........................................... 2

II  Partial-Reconfiguration on the FPGA ................. 3
   II-A  The FPGA Platform  .................................. 3
   II-B  Memory Subsystem and the PLB Bus ............... 4
   II-C  Dynamic Partial Reconfiguration .................... 4

III  Reconfigurable 2D-FFT  ............................... 5
   III-A  Block Floating-point FFT  .......................... 6
   III-B  Calculating the Absolute Value .................. 8
   III-C  Fast Matrix Transpose  ............................ 10

IV  Next-Neighbor Regridder  .............................. 12

V  Discussion ........................................... 16

References ............................................. 19
A Reconfigurable Real-time Reconstruction Engine for Parallel MRI

Abstract

Parallel MRI acquisitions are generally reconstructed into images off-line, on PCs and computer clusters. Here, we present an innovative single-FPGA engine that performs real-time 2D-FFT image reconstruction from arrays of up to 16 coils. Partial reconfiguration enables rapid switching of FPGA modules for maximal flexibility and lower hardware cost. If the engine is integrated on to an FPGA-based receiver, more complicated parallel MRI algorithms can replace the receiver logic after an acquisition is complete. The engine can also function as a standalone hardware accelerator. As a proof-of-concept for real-time non-cartesian reconstruction, a reconfigurable module for next-neighbor regridding of spiral MRI is also demonstrated.

I. INTRODUCTION

A magnetic resonance imaging (MRI) scanner consists of a back-end and a front-end. The back-end includes static and gradient magnetic-field generators that excite H\textsuperscript{+} nuclei in the body, as well as coils (antennae) that pick up RF signals emitted by the nuclei when the external fields relax. The front-end is comprised of receivers that demodulate and digitize these RF-MR signals, followed by a reconstruction engine that converts the frequency-domain RF-MR data into spatial images.

The acquisition time of a classical single-coil scanner is limited by electrical and physiological factors. Current MRI research focuses on parallel imaging (pMRI) techniques such as SMASH, SENSE and GRAPPA \cite{1}. These speed up acquisition by employing multiple-coil phased-arrays, followed by data interpolation based on the individual coil sensitivities (radiation pattern). Therefore, pMRI requires multiple receiver channels as well as more sophisticated image reconstruction hardware.

Commercial MRI front-ends use analog RF/IF receivers, baseband sampling and PCs for reconstruction. Other experimental front-ends for pMRI \cite{2, 3} have used specialized digital receivers, bandpass sampling and computer clusters for image reconstruction. Recently, a field-programmable gate array (FPGA)-based 16-channel digital receiver has also been demonstrated \cite{4}. General-purpose computers/CPUs are ill-suited for dedicated
pMRI reconstruction algorithms such as the 2D-FFT, requiring clusters of multiple computers for real-time pMRI. This solution is neither economical nor very efficient.

Our reconstruction engine generates real-time images (via the 2D-FFT) from arrays of up to 16 coils (or channels). It is built around a single FPGA, and includes a partial-reconfiguration framework that allows different reconstruction modules to be dynamically loaded/unloaded as needed. Such hardware re-use opens the door for real-time reconstruction with more complicated pMRI algorithms that may be too large to fit on the FPGA. These algorithms can usually be partitioned into sub-modules that operate sequentially upon data; hence, partial reconfiguration allows them to be fully implemented without having to move to larger or multiple FPGAs. For example, if the engine and the digital receiver of [4] are integrated on one FPGA, then as soon as an acquisition has been buffered in on-board memory, the receiver logic is no longer necessary and can be replaced by, say, SENSE reconstruction. To the best of our knowledge, this is the first single-FPGA based pMRI reconstruction engine and has a superior performance-cost ratio than existing solutions.

This paper first discusses the design of the partial-reconfiguration framework. Considering that image quality is paramount in medical MRI, the implementation of an efficient-yet-accurate fixed-point 2D-FFT, including a fast matrix transpose is discussed next. Finally, we demonstrate the utility of the engine beyond conventional (cartesian) MRI with a novel implementation of next-neighbor regridding [5] for spiral MRI. Relevant performance analyses and results are also presented.

II. PARTIAL-RECONFIGURATION ON THE FPGA

A. The FPGA Platform

The engine is built around a Xilinx Virtex-II Pro XC2VP30 FPGA that has \( \approx 13,500 \) configurable logic blocks (CLB), 136 \( 18 \times 18 \)-bit multipliers and two embedded PowerPC 405 microprocessors. The FPGA sits on a XUPV2P development board that provides physical-layer ICs such as a Fast Ethernet transceiver to facilitate I/O via multiple interfaces. One of the PowerPCs acts as a system supervisor and also runs a lightweight TCP/IP stack and FTP server for transferring data/images over Ethernet; the other PowerPC is currently unused.
B. Memory Subsystem and the PLB Bus

The engine has two kinds of memory: primary memory and secondary memory. Primary (or ‘core’) memory is a limited amount (2.5 megabits) of fast, single-cycle-latency static RAM (SRAM) inside the FPGA that is reserved for use by logic modules. Secondary memory is a slower but much larger consumer-grade 512 megabyte DDR dynamic RAM (DRAM) module [6] used mostly for storing for MRI acquisitions. A DRAM memory controller on the FPGA handles low-level interfacing to the DDR RAM module to enable simplified memory access.

The PowerPC communicates with the DRAM and other high-bandwidth logic peripherals such as the fast ethernet MAC over the 64-bit Processor Local Bus (PLB) [7]. Transferring data to/from DRAM over the PLB incurs some transactional overhead in addition to DRAM’s native latencies. Hence, transfers of long bursts of contiguous data are much more efficient than single reads/writes. The DRAM access cost can be linearly modeled as

\[ \tau_{\text{read}} + m \tau + mt \]  

where \( \tau \) is the initial latency, \( m \) is the number of data units and \( t \) is the transfer time/unit. With 64-bit units at a 100 MHz PLB/DRAM frequency, \( \tau_{\text{read}} = 100 \) ns, \( \tau_{\text{write}} = 50 \) ns and \( t = 10 \) ns, leading to burst transfer rates of up to 800 megabytes/s. Read/write latencies are asymmetric since the DRAM controller can only pipeline writes. We limit burst transfers to a maximum of 256 32-bit ‘words’ since maximum burst lengths may vary based on the physical structure of the DRAM module.

C. Dynamic Partial Reconfiguration

The FPGA is programmed in ‘frames’—addressable units that physically stretch from ‘top’ to ‘bottom’. This enables dynamic partial reconfiguration [8], i.e. reprogramming portions (sets of frames) of the FPGA while the remainder of the device is still operational (Fig. 1).

A crucial consideration in PR design is ensuring seamless communication between partially-reconfigurable modules (PRMs) and existing static logic (as well as other PRMs). Signals can only cross PR boundaries via logic bridges called bus macros which must be
hard-wired upon initial programming. While bus macros provide physical interconnects, the PR framework on a multiple-module system such as the reconstruction engine must be structured around a logical bus for consistency. Since reconstruction PRMs need direct access to MRI data stored in DRAM (which is connected to the PLB), PLB is used as the framework bus.

Custom FPGA logic modules including PRMs must connect to the PLB via an IP Interface (IPIF); each PRM must have a dedicated IPIF. We have designed a simplified IPIF that uses significantly fewer resources than Xilinx’s IPIF, but relies on the PRM to provide buffers and a finite state machine (FSM) for data transfers. The IPIF is a combined master/slave PLB device supporting techniques important for real-time applications such as bus mastering and variable-length bursts. When the FPGA is initially programmed, a fixed number of IPIFs are placed on PR boundaries and permanently connected to the PLB as shown in Fig. [1] Upon reconfiguration, a PR module is automatically linked to its boundary IPIF. The FPGA’s internal structure places certain constraints upon the placement of PRMs. Therefore, PRMs that include resources in fixed locations inside the FPGA—such as external I/Os, multipliers and SRAM—usually have some overhead compared to a non-PR version.

III. RECONFIGURABLE 2D-FFT

The real-time 2D-FFT consists of a 1D block floating-point FFT, absolute value (CORDIC) and a fast matrix transpose as shown in Fig. [2]
Fig. 2. 2D-FFT: 1D block floating-point (BFP) FFT, absolute value (CORDIC) and fast matrix transpose, with the process flow at the bottom.

Fig. 3. A magnitude plot of the k-space MRI data-set FatWater

A. Block Floating-point FFT

MRI data-sets belong to the complex spatial-frequency domain, or *k-space*. Data-sets are generally acquired on cartesian trajectories and reconstructed via the 2D Fast Fourier Transform (FFT) into spatial images. The 2D-FFT is decomposed into 1D ‘row’ and ‘column’ passes. At each stage of a radix-2 FFT, the intermediate output can grow by as much as $1 + \sqrt{2} \approx 2.414$, causing overflows in fixed-point architectures; the outputs of one or more stages must be scaled to prevent overflow. When the nature of the FFT input is unknown, a common scheme is to scale each stage by 2 (1 bit) regardless of whether
overflow would have occurred. If a large proportion of the input data are small compared to the FFT’s precision, accumulated roundoff errors from scaling-by-2 severely affect the final output; this problem is compounded with the dual row-column FFTs for 2D data. Fig. 3 is a magnitude plot of FatWater, a 16-bit 256 × 256-point k-space data-set. Characteristically, the data are very small except for the central region (low frequencies). Reconstructing k-space data-sets with a 16-bit scaling-by-2 fixed-point FFT results in extremely poor image quality. Better quality can be obtained by increasing the FFT precision or through multi-pass normalization at the expense of higher run-times and/or resource usage.

There is a third alternative: at a given precision, the block floating-point FFT (BFP-FFT) [9] is better than fixed- and floating-point, and only slightly more computationally expensive than fixed-point. Each BFP-FFT stage scales intermediate outputs only when they would overflow (Fig. 4) and tracks overall scaling with a block exponent $b$ common to the entire input block. Multiplying the output block by $2^b$ recovers unity-gain values.

Both 16- and 24-bit fixed-point and block floating-point FFTs were tested extensively

---

**TABLE I**

<table>
<thead>
<tr>
<th>Data-set</th>
<th>Metric</th>
<th>FxP/16</th>
<th>BFP/16</th>
<th>FxP/24</th>
<th>BFP/24</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>noise</strong></td>
<td>PSNR (dB)</td>
<td>47.59</td>
<td>66.76</td>
<td>74.02</td>
<td>91.52</td>
</tr>
<tr>
<td><strong>FatWater</strong></td>
<td>PSNR (dB)</td>
<td>27.83</td>
<td>62.02</td>
<td>61.42</td>
<td>86.30</td>
</tr>
<tr>
<td></td>
<td>SSIM</td>
<td>0.5414</td>
<td>0.9993</td>
<td>0.9992</td>
<td>1.0000</td>
</tr>
<tr>
<td><strong>Cardiac</strong></td>
<td>PSNR (dB)</td>
<td>22.80</td>
<td>62.45</td>
<td>60.70</td>
<td>86.75</td>
</tr>
<tr>
<td></td>
<td>SSIM</td>
<td>0.4244</td>
<td>0.9998</td>
<td>0.9998</td>
<td>1.0000</td>
</tr>
</tbody>
</table>
with a number of MRI data-sets. Table I compares the quality of the final 2D images from the different FFTs to a reference 64-bit floating-point MATLAB reconstruction. The test data-sets are all 256×256 complex 16-bit: uniformly distributed random numbers (noise), an artificial phantom (FatWater) and an actual acquisition (Cardiac). The PSNR metric is derived from the mean-squared error (or $L^2$ norm) of the pixel-by-pixel difference between the ‘distorted’ image and the reference. While used extensively in image coding studies, PSNR is not always indicative of an image’s perceived visual quality. Hence, we also employ the *Structural Similarity* (SSIM) metric that considers both the underlying structure of (i.e. objects in) the image and the human visual system model to calculate an SSIM index between 0–1 (higher is better).

Figs. 5a and 6a show FatWater and Cardiac reconstructed with the 16-bit BFP-FFT that was chosen based on its high quality and negligible resource overhead (< 10%) compared to 16-bit fixed-point. Both metrics are excellent, and the difference images Figs. 5b and 6b (with respect to the MATLAB reference) had to be multiplied by extremely large factors so that difference detail would be visible to the naked eye.

Clocked at 200 MHz, Xilinx’s radix-2 ‘minimum resources’ FFT core calculates a 256-point transform in 5.6μs; operating it in bit-reversed addressing mode allows synchronous load/unload. A fast matrix transpose (section III-C) performs natural reordering and transposition. If all the block-exponents $B = \{b_1, \ldots, b_n\}$ from the first 1D-FFT (rows) are not equal, each row $i$ is divided by $2^{\max(B) - b_i}$ so that the column inputs to the second 1D-FFT have the same relative gain. After the second FFT, each output column is multiplied by $2^{b_i}$ as the next stage (absolute value; section III-B) has a higher precision (24-bit). These block-exponent normalizations are accomplished by using the embedded multipliers as single-cycle barrel shifters.

### B. Calculating the Absolute Value

The most common way to form a spatial image is by taking the absolute value (magnitude) of the complex 2D-FFT outputs $X + jY$. This can be computed efficiently in hardware by the recursive CORDIC algorithm using only shifts and adds (no multiplies). CORDIC iteratively rotates the vector $(X, Y) \rightarrow (X', Y')$ until $Y' = 0$, thus calculating magnitude $X' = \sqrt{X^2 + Y^2}$ and phase (not used). Output precision increases
Fig. 5. **Cardiac**; PSNR = 62.45 dB, SSIM = 0.9993

Fig. 6. **FatWater**; PSNR = 62.02 dB, SSIM = 0.9998
by \( \approx 1 \) bit per iteration, and iterations can be unrolled and pipelined for faster throughput.

Xilinx’s vector rotation core could not provide 24-bit outputs at 200 MHz on our device. Hence, we use two multipliers and an adder (Fig. 2) to calculate \( X^2 + Y^2 \), followed by a simplified hyperbolic mode CORDIC which calculates \( \sqrt{X^2 + Y^2} \). The circuit’s performance can be summed up with the observation that running various MRI data-sets (including those in Table I) through it resulted in negligible PSNR decreases of \(< 0.02 \) dB over MATLAB’s absolute value \((\text{abs})\) function.

C. Fast Matrix Transpose

As Fig. 2 illustrates, the 2D-FFT requires two transpositions: one after the ‘rows’ and another after the ‘columns’. Common \( N \times N \) matrices for MRI applications are \( N = 128 \) and \( N = 256 \), but more complicated algorithms (such as the regridder of section IV) may use up to \( N = 4096 \). Transposing large matrices stored ‘out-of-core’, i.e. in secondary memory (DRAM), requires an intermediate SRAM buffer. Data are read/written in bursts of \( M \) 32-bit words, and the maximum burst transfer length is \( M_{\text{max}} \) (in our case, 256 words). For simpler notation, read/write latency \( \tau_{\text{RW}} = \tau_{\text{read}} + \tau_{\text{write}} \). Derivations for the transposition run-times that follow are detailed in the appendix.

From (A.1), the minimum transpose time for a \( N \times N \) matrix \((N \geq M_{\text{max}})\) on our system is

\[
T_{\text{min}} = \frac{N^2}{M_{\text{max}}} (\tau_{\text{RW}} + 2M_{\text{max}} t)
\]

(the factor of 2 is for reading and writing). What sets different transposition algorithms apart is the size \( B \) of the intermediate buffer necessary to achieve this bound, e.g. a direct transpose requires \( B = M_{\text{max}}^2 \), i.e. \( 256^2 \times 32\)-bits, or 2 megabits. Considering the extremely limited SRAM (2.5 megabits) available, our goal is minimizing transpose time across different \( N \) with as small a buffer as possible.

For this purpose, we have adapted Kaushik, et al’s single-radix transposition (SRT) algorithm [12]. SRT is out-of-place and requires temporary storage equal to the original matrix’s in secondary memory; this is not a problem since consumer DRAM modules are available in large capacities at reasonable cost. If \( N \) is factorized as
the SRT algorithm makes \( p \) passes over the entire matrix with \( N/s_i \) steps-per-pass. In each step, \( s_i \) consecutive rows are read from DRAM into a \( B = s_i \times N \) buffer, linearly permuted, and written back as non-consecutive rows. Fig. 7 illustrates the SRT of a 4 \( \times \) 4 matrix. From (3), the number of passes \( p \) depends on the choice of factors \( s_i \); the largest \( s_i \) determines the size \( B \) of the buffer. As the ‘buffer/permute’ parts of Fig. 7 show, permutation can be performed by filling the \( s_i \times N \) buffer column-wise and then emptying it row-wise into DRAM. In hardware this simply requires rewiring the \( \log_2(s_i) \) LSBs of the buffer’s address input as MSBs before it is filled, e.g. rewire \([b_2 b_1 b_0]\) as \([b_0 b_2 b_1]\) for the two-row buffer of Fig. 7.

From (A.2), SRT run-time is \( p \cdot T_{\text{min}} \) and may be reduced by decreasing the number of passes \( p \). For a \( N \times N \), \( N = 2^n \) matrix, SRT is fastest for \( p = 2 \) and factors into two \( s_i \):

\[
\text{even } n: \begin{cases} s_1 = 2^{\frac{n}{2}} \\ s_2 = 2^{\frac{n}{2}} \end{cases} \quad \text{odd } n: \begin{cases} s_1 = 2^{\frac{n+1}{2}} \\ s_2 = 2^{\frac{n-1}{2}} \end{cases} \tag{4}
\]

Such minimal-time implementations require large \( B = s_1 \times N \) buffers. However, with only a slight increase in the complexity of the permutation logic, much smaller \( B = s_1 \times M_{\text{max}} \) buffers can be used without affecting performance (A.3); transfers of length \( M < M_{\text{max}} \) allow for even smaller \( B = s_1 \times M \) buffers with minor increases in run-time. Additionally, the first SRT pass can be performed on the FFT outputs on their way back to DRAM (Fig. 2), effectively making it ‘free’. We call these modifications the compact-SRT (cSRT)

---

Fig. 7. Single-radix transposition of a 4 \( \times \) 4 matrix in 2 passes (steps \( s_1, s_2 \))
Table II shows compact-SRT run-times for different matrix and buffer sizes, compared to the direct algorithm using a 65536 × 32-bit buffer. The 2D-FFT uses a 2048 × 32-bit buffer to achieve a good balance between buffer size and run-time for different $N$. As shown in Fig. 2, two such buffers allow the second transposition pass to occur while the 1D-FFT is being computed. A complete $256 \times 256$ 2D-FFT is calculated in $\approx 4$ ms.

### IV. Next-Neighbor Regridding

Non-cartesian MRI trajectories such as spirals are increasingly used for fast imaging. Regridding refers to the process of mapping the non-cartesian trajectories to cartesian coordinates so that the 2D-FFT can be used for reconstruction. Conventional regridding methods convolve the data with an interpolation kernel before resampling onto a rectangular grid; they are computationally intensive and are run off-line. The next-neighbor (NN) regridding algorithm [5] is less complex and can provide comparable quality. It maps k-space data in a spiral data-set $S$ to a large $2^q \times 2^q$ sparse substitute matrix $\hat{R}$ with minimal interpolation. $\hat{R}$ starts out as all zeros, and k-space positions $(u_s, v_s) \in \mathbb{R}^2$ are quantized to $q$-bit integers to obtain ‘next-neighbor’ indices $(\hat{u}_r, \hat{v}_r) \in \mathbb{N}^2$ in $\hat{R}$. Data $S(u_s, v_s)$ are regredded to $\hat{R}(\hat{u}_r, \hat{v}_r)$, although redundant data points can occur when quantization causes multiple k-space positions $(u_s, v_s)$ to regred to the same location $(\hat{u}_r, \hat{v}_r)$ in $\hat{R}$. The number of redundant points depends on the size of $\hat{R}$, i.e. $q$ and for
spiral scans, on the number of overlapping interleaves. \( \hat{R}(\hat{u}_r, \hat{v}_r) \) containing redundant points must be averaged before FFT reconstruction. Also, if the positions \((u_s, v_s)\) do not have a uniform density in k-space, \( S(u_s, v_s) \) must be multiplied with weighting factors \( W(u_s, v_s) \) before reconstruction.

After quantizing each \((u_s, v_s) \rightarrow (\hat{u}_r, \hat{v}_r)\), the regriddler must check for redundancy by verifying that \( \hat{R}(\hat{u}_r, \hat{v}_r) \) is zero and does not contain any previously regridded data points. A primary concern of our design was efficiently detecting redundancy when the data are stored in secondary memory (DRAM). Even for a contiguous block of \((u_s, v_s)\), the corresponding quantized \((\hat{u}_r, \hat{v}_r)\) are effectively non-contiguous, making naïve read-back of each \((\hat{u}_r, \hat{v}_r)\) for verification too slow. One-to-one occupancy maps such as a \(2^2q\) bit-array are also inefficient since \( \hat{R} \) is sparse. Instead, we use Bloom filters \cite{13, 14}. The Bloom filter is a randomized data structure that compactly represents a set for querying membership. Its space-efficiency can be traded off against a small probability of false positives.

A Bloom filter consists of \(k\) hash functions \((h_i, i = 1, 2 \ldots k)\) and an \(m\)-bit array \((M)\) initially set to all zeros. Given an input \(x\), the hash functions generate outputs \(h_i(x)\) with range \(\{1 \ldots m\}\) that address individual bits of the array \(M\). For a set \(A = \{x_1 \ldots x_n\}\) with \(n\) elements, the filter is programmed by setting the \(k\) bits in \(M\) that correspond to \(h_i(x)\) to one for each \(x \in A\). To check if an item \(y\) is in \(A\), we check the bits in \(M\) corresponding to \(h_i(y)\). If any bit is zero, \(y \notin A\). If all the bits are one, we assume \(y \in A\), with a false positive probability \(P_{FP} \approx (1 - e^{-kn/m})^k\).

Hash functions from the \(H_3\) universal class \cite{15} are computed with only exclusive-OR (XOR, \(\oplus\)) operations, and are thus very efficient for hardware implementation. It is easiest to illustrate \(H_3\) hashes with an example. If the hash inputs are 8 bits and the outputs (addresses) must be 4 bits, we choose a matrix \(D\) of 4-bit random numbers
Fig. 8. Flow-diagram for next-neighbor regridding
TABLE III

<table>
<thead>
<tr>
<th>Size of ( \mathbf{R} (2^q \times 2^q) ):</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>res. Phantom ((n = 32768 \text{ pts, } n \bar{P}_{FP} = 2.56))</td>
<td>(t ) (ms)</td>
<td>2.71</td>
<td>2.57</td>
<td>2.53</td>
</tr>
<tr>
<td></td>
<td>FP Mean</td>
<td>2.41</td>
<td>2.47</td>
<td>2.46</td>
</tr>
<tr>
<td></td>
<td>Red. Pts</td>
<td>500</td>
<td>160</td>
<td>81</td>
</tr>
<tr>
<td>coronary ((n = 65536 \text{ pts, } n \bar{P}_{FP} = 276.39))</td>
<td>(t ) (ms)</td>
<td>5.50</td>
<td>5.15</td>
<td>5.07</td>
</tr>
<tr>
<td></td>
<td>FP Mean</td>
<td>278.51</td>
<td>271.93</td>
<td>274.35</td>
</tr>
<tr>
<td></td>
<td>Red. Pts</td>
<td>1112</td>
<td>302</td>
<td>120</td>
</tr>
</tbody>
</table>

\(\{d_1, d_2, \ldots , d_8\}\). Then, for an input \(x = 10001010\),

\[
D = \begin{bmatrix}
1100 \\
0001 \\
1111 \\
1110 \\
1001 \\
0100 \\
1101 \\
0110
\end{bmatrix}
\]

\[
\therefore \ h(x) = h(10001010) = 1 \cdot d_1 \oplus 0 \cdot d_2 \oplus 1 \cdot d_3 \oplus 1 \cdot d_4 \\
\quad \quad \quad \quad \quad \quad \oplus 0 \cdot d_5 \oplus 0 \cdot d_6 \oplus 1 \cdot d_7 \oplus 0 \cdot d_8 \\
\quad \quad \quad \quad = 1100 \oplus 1001 \oplus 1101 = 1000
\]

Fig. 8 is a flow-diagram of the regridder. It is optimized for spiral scans with up to \(n = 65536\) total points and regrids them to substitute matrices up to \(4096 \times 4096\), i.e. \(q \leq 12\). First, each quantized \((\hat{u}_r, \hat{v}_r)\) is concatenated to a 24-bit number \((\hat{u}_r \cdot 2^{12} + \hat{v}_r)\) that is input to 8 \((k)\) hashes. The 16-bit hash outputs individually address 64-kbit chunks of the 512-kbit array \(\mathcal{M}\). Since the Bloom filter is always used in a simultaneous query/program mode, the average false positive probability \(\bar{P}_{FP}\) is

\[
\bar{P}_{FP} \approx \frac{1}{n} \sum_{j=1}^{n} (1 - e^{-k \cdot j/m})^k \ll P_{FP}
\]

(5)

Offsets of \(\mathbf{R}(\hat{u}_r, \hat{v}_r)\) containing redundant points are stored and these are averaged after all \(n\) points have been regridded.
The NN regridder was tested with a GE resolution phantom \( (n = 2048 \text{ pts} \times 16 \text{ interleaves}) \) and a coronary scan \( (n = 4096 \times 16) \); both were regridded to different sizes of \( \hat{R} \) as shown in Table III. The eight regridding tests were repeated 1000 times with different hashes, i.e. random numbers obtained from an on-board 32-bit maximal-length linear-feedback shift register. As expected, the mean number of false positive points \( (FP \text{ Mean}) \) is very close to what theory, i.e. \( (5) \), predicts: \( n \bar{P}_{FP} \). The regridding time \( t \) is nearly constant, and asymptotically depends only on the number of points \( n \) to be regridded.

Fig. 9 shows reconstructions of the GE resolution phantom with conventional regridding (Kaiser-Bessel, window width 2.5) and next-neighbor regridding \((2048 \times 2048 \text{ substitute matrix } \hat{R})\). The NN-regridded data was transformed with the on-board 2D-FFT in \( \approx 320 \text{ ms} \).

V. DISCUSSION

The partial-reconfiguration framework was designed in VHDL and synthesized with Xilinx XST. It was then tested on the FPGA by downloading reconfiguration bitstreams from a PC as well as a CompactFlash memory card (standalone mode). Bus-macro functionality and memory throughput were verified by connecting the IPIF to a ‘stress-test’ PR module that interacted with PLB control signals and performed a wide variety of read/write transfers. Preliminary design and functional simulation of the 2D-FFT and NN-regridder were done with Xilinx’s high-level System Generator tool; these modules were then synthesized and simulated with ModelSim using behavioral models of the PowerPC and DDR RAM for bit- and cycle-accurate results.
Table IV shows the resource usage of each module; the 2D-FFT is ‘light’ enough to be integrated with the digital receiver of [4] on an XC2VP30 FPGA as a complete front-end. At $\approx 4$ ms per $256 \times 256$ FFT, a 250 images/sec throughput can be sustained with sum-of-squares interpolation of multiple coil images. Thus, real-time reconstruction at near-movie (16 channels @ 15 fps) to movie (8 channels @ 30 fps) frame-rates is possible for applications such as cardiac MRI. Reconstructed image quality is excellent as demonstrated earlier.

**CONCLUSION**

An innovative multi-channel reconfigurable engine for MR imaging has been presented. The block-floating point 2D-FFT module reconstructs up to 16 channels in real-time with very high quality. Partially reconfiguring the FPGA allows idle acquisition/imaging modules to be dynamically replaced with other modules, increasing hardware reuse and thus flexibility as well as economy. The utility of partial reconfiguration beyond conventional cartesian MRI has also been demonstrated with a fast next-neighbor regridder for spiral MRI. Possible future directions include developing real-time modules for accelerated pMRI techniques such as SENSE and GRAPPA.
APPENDIX: DERIVING TRANPOSITION RUN-TIMES

If length–$M$ burst transfers are used, larger reads/writes are broken up into multiple transfers. Hence, direct transposition requires reading $M$ row-wise blocks (of $M$ units) from DRAM into a $B = M^2$ buffer so that $M$ column-wise blocks (of $M$ units) can be written out to DRAM. Run-time is:

$$T_{\text{direct}} = \frac{N^2}{M}(\tau_{\text{RW}} + 2Mt) \quad (A.1)$$

The absolute minimum time $T_{\text{min}}$ is obtained when $M = M_{\text{max}}$, the maximum possible burst transfer length.

The time for the single-radix algorithm is:

$$T_{\text{SRT}} = \text{passes} \times \frac{\text{steps}}{\text{pass}} \times \frac{\text{rows}}{\text{step}} \times \frac{\text{M-blocks}}{\text{row}} \times \frac{\text{time}}{\text{M-block}}$$

$$= p \times \frac{N}{s_i} \times s_i \times \frac{N}{M} \times (\tau_{\text{RW}} + 2Mt) \quad (A.2)$$

$$= p \frac{N^2}{M} (\tau_{\text{RW}} + 2Mt)$$

and can be minimized to $2T_{\text{SRT}} = 2T_{\text{min}}$ for $p = 2$ and $M = M_{\text{max}}$. This requires a $s_i$-row buffer, i.e. $B = s_i \times N$.

For $p = 2$ and length–$M$ transfers, we can reduce the buffer to $s_1 \times M$ if the SRT permutation logic is modified for the appropriate offsets. Each step $s$ is now divided into $s/k_i$ sub-steps, and in each sub-step, only $k = s_i \cdot M/N$ rows are filled row-wise into the $s_1 \times M$ buffer so that column-wise blocks of length–$M$ can be written to main memory. As noted in section [III-C] transposing the FFT outputs on their way back to DRAM gives us a ‘free’ first pass. These strategies form the basis for the compact-SRT algorithm we have implemented, with run-time:
\[ T_{\text{CSRT}} = \text{passes} \times \text{steps} \times \frac{\text{substeps}}{\text{step}} \times \frac{\text{rows}}{\text{substep}} \times \frac{\text{time}}{\text{row}} \]

\[ = 1 \times \frac{N}{s_i} \times \frac{s_i}{k_i} \times k_i \times \frac{N}{M}(\tau_{\text{RW}} + 2Mt) \]

\[ = \frac{N^2}{M}(\tau_{\text{RW}} + 2Mt) \quad (A.3) \]

Thus, if \( M = M_{\text{max}} \) with buffers \( [B_{\text{CSRT}} = s_1 \times M_{\text{max}}] < [B_{\text{direct}} = M_{\text{max}}^2] \), \( T_{\text{CSRT}} = T_{\text{direct}} = T_{\text{min}} \). If \( M < M_{\text{max}} \) and \( [B_{\text{CSRT}} = s_1 \times M] < [B_{\text{direct}} = M^2] \), \( T_{\text{CSRT}} < T_{\text{direct}} \) since the direct transpose must read and write in length–\( M \) bursts while compact-SRT can always read in length–\( M_{\text{max}} \) bursts (because SRT reads are contiguous rows).

**REFERENCES**


