High-Speed Compressed Sensing Reconstruction on FPGA Using OMP and AMP

Lin Bai, Patrick Maechler, Michael Muehlberghuber, and Hubert Kaeslin
Integrated Systems Laboratory, ETH Zurich, Switzerland
Email: lbai@ee.ethz.ch, {maechler, mbgh, kaeslin}@iis.ee.ethz.ch

Abstract—**Compressed sensing allows to reconstruct sparse signals sampled at sub-Nyquist rates. However, reconstruction of the original signal requires high computational effort, even for problems of moderate size. Especially for applications with real-time requirements, software realizations are not fast enough. We therefore present generic high-speed FPGA implementations of two fast reconstruction algorithms: orthogonal matching pursuit (OMP) and approximate message passing (AMP). Our implementations also support less sparse signals, which makes them suitable for, e.g., image reconstruction. The two implementations are optimized for highly parallel processing on FPGAs and have similar hardware structures, which allows comparisons in terms of resource usage and performance.**

I. Introduction

Compressed sensing (CS) [1] enables the acquisition of a signal with fewer measurements than the Nyquist rate suggests. In order to reconstruct the original signal, this signal must be sparse, i.e., must be represented by a small number of non-zero coefficients in a certain basis. Reducing the number of measurements can reduce the time or cost of signal acquisition, which led to a large interest in CS for many applications. Prominent examples include image acquisition, magnetic resonance imaging (MRI), wireless communication, and radar. While CS reduces the hardware requirements of signal acquisition, digital signal processing, necessary to recover the original signal, becomes much more involved. Even though much progress has been made in the development of fast recovery algorithms, the computational complexity remains very high.

Many of the above applications would benefit from fast real-time reconstruction, which makes the development of fast digital reconstruction hardware necessary. Hardware implementations are either targeted at a specific problem, which may allow to exploit fast transforms that exist for certain bases (e.g. Fourier, wavelet), or are general-purpose solvers, which can be applied to any (unstructured) CS problem. The target of this paper is to develop reconfigurable general-purpose architectures for field-programmable gate arrays (FPGA).

Prior work: In order to speed up complex reconstruction algorithms, a number of implementations on graphics processing units (GPU) [2] and a small number of implementations on application-specific integrated circuits (ASICs) or reconfigurable FPGAs were reported so far. The first ASICs implementing the greedy CS reconstruction algorithms matching pursuit (MP), gradient pursuit (GP), and OMP were presented in [3], [4] for channel estimation in wireless communication systems. An FPGA implementation of OMP for generic CS problems of dimension 32×128 was developed in [5], processing signals up to a sparsity of $K = 5$. In [6], a reconstruction algorithm similar to OMP is implemented on an FPGA to reconstruct band-sparse signals acquired by the modulated wideband converter. An OMP-like implementation for problems of size 64×256 was proposed in [7], which, however, does not orthogonalize the estimation in every iteration. The first AMP designs [8] perform audio restoration and solve CS problems of size 512×1024.

Contributions: In this paper, we present highly-parallel FPGA implementations of the two CS reconstruction algorithms OMP [9] and AMP [10]. Running on a Xilinx Virtex-6 FPGA, these are the two fastest general purpose CS solvers on FPGAs known so far. In contrast to earlier VLSI implementations, our designs can also deal with less sparse signals, which enables, e.g., fast image reconstruction. Finally, we also provide a comparison of hardware complexity.

Notation: Bold lower and upper case symbols stand for column vectors and matrices, respectively. $\boldsymbol{A}_i$ represents the i*th* column and $\boldsymbol{A}_{ji}$ represents the entry at the j*th* row and i*th* column of matrix $\boldsymbol{A}$. $\boldsymbol{A}_\Gamma$ is a subset of matrix $\boldsymbol{A}$ consisting of the columns defined in the set Γ.

II. Compressed Sensing

CS allows the acquisition of sparse signals at sub-Nyquist sampling rates [1]. Measurements $\boldsymbol{y} = \boldsymbol{\Phi x}$ with $\boldsymbol{y} \in \mathbb{R}^m$, $\boldsymbol{\Phi} \in \mathbb{R}^{m\times N}$ are non-adaptive linear projections of a signal $\boldsymbol{x} \in \mathbb{R}^N$ with typically $m \ll N$. Recovering the original signal $\boldsymbol{x}$ means solving an under-determined linear equation with usually no unique solution. If, however, $\boldsymbol{x}$ has a sparse representation $\boldsymbol{s}$ in a basis $\boldsymbol{\Psi} \in \mathbb{R}^{N\times N}$ ($\boldsymbol{x} = \boldsymbol{\Psi s}$), it can, under certain conditions, be recovered from the measurements

$$\boldsymbol{y} = \boldsymbol{\Phi x} = \boldsymbol{\Phi \Psi s} = \boldsymbol{\Theta s}. \tag{1}$$

Here, $\boldsymbol{\Theta} \in \mathbb{R}^{m\times N}$ is the measurement matrix $\boldsymbol{\Theta} = \boldsymbol{\Phi\Psi}$. The sparsity of $\boldsymbol{s}$ is measured by the number of non-zero coefficients K, often written as $\|\boldsymbol{s}\|_0$.

III. Reconstruction Algorithms

Solving (1) by finding the sparsest possible $\boldsymbol{s}$ that complies with the measurements results in an NP-hard problem and is computationally intractable. It is, however, possible to relax

978-1-4673-1260-8/12/$31.00 ©2012 IEEE

Algorithm 1 Orthogonal matching pursuit (OMP)

```
Input: matrix $\boldsymbol{\Theta}$, measurements $\boldsymbol{y}$, sparsity $K$
Output: sparse reconstruction $\boldsymbol{s}^K$
1: $\boldsymbol{r}^0 = \boldsymbol{y}$ and $\Gamma^0 = \emptyset$
2: for $i = 1, ..., K$ do
3:     $\lambda^i \leftarrow argmax_j |\langle \boldsymbol{r}^{i-1}, \boldsymbol{\Theta}_j \rangle|$ ▷ Find best fitting column
4:     $\Gamma^i \leftarrow \Gamma^{i-1} \bigcup \lambda^i$
5:     $\boldsymbol{s}^i \leftarrow argmin_{\boldsymbol{s}} \|\boldsymbol{r}^{i-1} - \boldsymbol{\Theta}_{\Gamma^i} \boldsymbol{s}\|_2^2$    ▷ LS optimization
6:     $\boldsymbol{r}^i \leftarrow \boldsymbol{r}^{i-1} - \boldsymbol{\Theta}_{\Gamma^i} \boldsymbol{s}^i$    ▷ Residual update
7: end for
```

Algorithm 3 Approximate message passing (AMP)

```
Input: matrix $\boldsymbol{\Theta}$, measurements $\boldsymbol{y}$
Output: sparse reconstruction $\boldsymbol{s}^{I_{max}}$
1: $\boldsymbol{r}^0 = \boldsymbol{y}$ and $\boldsymbol{s}^0 = \boldsymbol{0}_{N \times 1}$
2: for $i = 1, ..., I_{max}$ do
3:     $\theta \leftarrow \lambda \frac{1}{\sqrt{m}} \|\boldsymbol{r}^{i-1}\|_2$
4:     $\boldsymbol{s}^i \leftarrow \eta_\theta(\boldsymbol{s}^{i-1} + \boldsymbol{\Theta}^T \boldsymbol{r}^{i-1})$    ▷ Thresholding
5:     $b^i \leftarrow \frac{1}{m} \|\boldsymbol{s}^i\|_0$
6:     $\boldsymbol{r}^i \leftarrow \boldsymbol{y} - \boldsymbol{\Theta} \boldsymbol{s}^i + b^i \boldsymbol{r}^{i-1}$    ▷ Residual update
7: end for
```

the reconstruction problem to a convex optimization problem, known as basis pursuit [11].

$$\hat{\boldsymbol{s}} = \arg\min \|\boldsymbol{s}\|_1 \text{ such that } \boldsymbol{y} = \boldsymbol{\Theta}\boldsymbol{s} \tag{2}$$

Convex optimization algorithms solving (2), such as interior-point methods, are still too complex for real-time recovery and are not suitable for hardware integration. Greedy algorithms, such as MP or OMP [9], iteratively approximate the solution of (2) by adding the best fitting element in each iteration. They are much less complex than convex optimizers but are not guaranteed to converge to the optimal solution. Iterative thresholding algorithms, such as AMP, are another class of algorithms that refine the estimation in each iteration by a thresholding step. We chose OMP and AMP as two of the best performing representatives of each class for implementation. Short descriptions of both algorithms are given below together with a discussion of implementation aspects.

A. Orthogonal Matching Pursuit

OMP [9] is a greedy reconstruction algorithm, which adds in each iteration the best fitting column of the matrix $\boldsymbol{\Theta}$ to the current estimate (Alg. 1). A least squares (LS) optimization is then performed in the subspace spanned by all previously picked columns. This ensures orthogonality of the estimate to the residual.

We solve the LS optimization problem in line 5 by QR decomposition (QRD) $\boldsymbol{\Theta}_{\Gamma^i} = \boldsymbol{Q}\boldsymbol{R}$, which yields an unitary matrix $\boldsymbol{Q}$ and an upper-triangular matrix $\boldsymbol{R}$. QRD allows an iterative updating of the decomposition, which reuses the $\boldsymbol{Q}$ and $\boldsymbol{R}$ matrices calculated in the last OMP iteration. The QRD algorithm is implemented by the modified Gram-Schmidt (MGS) algorithm [12] (Alg. 2), which is well suited for hardware implementation. The incremental update saves many operations compared to a full QRD in each iteration but requires the storage of matrices $\boldsymbol{Q}$ and $\boldsymbol{R}$. Having performed a QRD, the residual update can be considerably simplified (see [5]) by replacing line 6 of Alg. 1 with the following:

$$\boldsymbol{r}^i \leftarrow \boldsymbol{r}^{i-1} - \boldsymbol{Q}_i(\boldsymbol{Q}_i)^T \boldsymbol{r}^{i-1}$$

In order to calculate the final solution, back substitution is used, which easily solves the inversion of matrix $\boldsymbol{R}$:

$$\boldsymbol{s}^K = \boldsymbol{R}^{-1}\boldsymbol{Q}^T\boldsymbol{y}$$

Algorithm 2 Incremental QRD by Modified Gram-Schmidt

```
Input: new column $\boldsymbol{\Theta}_{\lambda^i}$; last iteration's $\boldsymbol{Q}^{n-1}$, $\boldsymbol{R}^{n-1}$.
Output: $\boldsymbol{Q}^n$ and $\boldsymbol{R}^n$.
1: $\boldsymbol{R}^n \leftarrow \begin{bmatrix} \boldsymbol{R}^{n-1} & 0 \\ 0 & 0 \end{bmatrix}$, $\boldsymbol{\xi}^n \leftarrow \boldsymbol{\Theta}_{\lambda^i}$
2: for $i = 1, ..., n-1$ do
3:     $\boldsymbol{R}^n_{jn} \leftarrow (\boldsymbol{Q}^{n-1})^H_j \boldsymbol{\xi}^n$
4:     $\boldsymbol{\xi}^n \leftarrow \boldsymbol{\xi}^n - \boldsymbol{R}^n_{jn} \boldsymbol{Q}^{n-1}_j$
5: end for
6: $\boldsymbol{R}^n_{nn} = \sqrt{\|\boldsymbol{\xi}^n\|_2^2}$
7: $\boldsymbol{Q}^n = \left[\boldsymbol{Q}^{n-1} \ \boldsymbol{\xi}^n / \boldsymbol{R}^n_{nn}\right]$
```

B. Approximate Message Passing

AMP [10] is an iterative soft thresholding algorithm with very fast convergence. Its pseudo code is listed in Alg. 3, which follows the exposition in [8]. In each iteration, a soft thresholding operation with threshold θ is performed:

$$\eta_\theta(a) = \operatorname{sign}(a) \max\{|a| - \theta, 0\}$$

The threshold θ is set proportionally to the regularization parameter λ and the root mean square error (RMSE) of the residual (see [8]). During the update of the residual on line 6 of Alg. 3, a correction term involving the residual of the previous iteration is applied. This correction term leads to a very fast convergence, typically within 10 ∼ 100 iterations, depending on the measurement matrix, size, and sparsity of the reconstruction problem.

IV. Hardware Implementation

In this section, we present FPGA designs of OMP and AMP. Using similar architectures allows to compare hardware complexity and performance. Both designs are highly pipelined and optimized for high-throughput processing. Control is realized by multiple cooperating finite state machines (FSMs).

A. OMP Implementation

The execution of OMP includes two computationally expensive steps, which are the matrix-vector multiplication $\boldsymbol{\Theta}^T\boldsymbol{r}$ on line 3 and the LS optimization on line 5 in Alg. 1. To speed up these operations, a large number of parallel multipliers P is instantiated and configured as a vector multiplication unit (VMU), which is the central computation unit of the

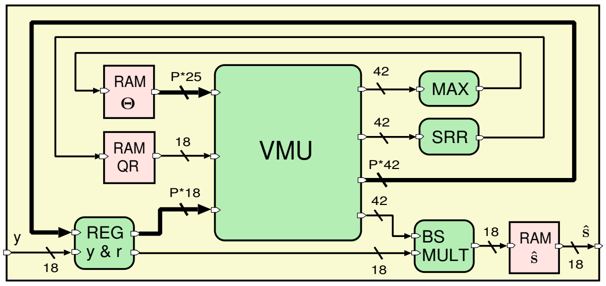


Fig. 1. High-level block diagram of OMP

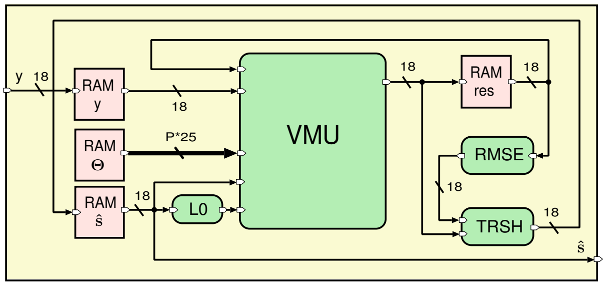


Fig. 2. High-level block diagram of AMP

architecture, as shown in Fig. 1. Thus, the main challenge of this design is to supply the VMU with enough parallel data at high speed. To do so, the memory storing $\boldsymbol{\Theta}$ is split into many block RAMs that can be accessed in parallel. Each block RAM holds one row of $\boldsymbol{\Theta}$. Other memories, such as the ones for the measured vector $\boldsymbol{y}$ and the residual $\boldsymbol{r}$ are implemented as register banks, where each element is accessible in parallel.

While the VMU is shared for most of the computations, a few additional hardware units perform specialized tasks. The MAX unit finds the maximum index from the correlations with columns of $\boldsymbol{\Theta}$. In the BS MULT unit, the back substitution in the LS algorithm is computed using one multiplier and one adder. An operation which also needs specialized hardware is the square root reciprocal (SRR) computing $1/\boldsymbol{R}_{nn}^{n}$ in the QRD, which combines parts of lines 6 and 7 in Alg. 2 and makes explicit division unnecessary.

1) Vector multiplication unit (VMU): The OMP VMU has multiple operation modes that allow to compute all multiplication operations in OMP.

- In the matrix-vector mode, the unit multiplies one column of a matrix with a vector in parallel and adds up all coefficients in a subsequent adder tree. This mode is used for the calculation of $\boldsymbol{\Theta}^T\boldsymbol{r}$ and QRD.
- In the subtraction mode, a second vector is subtracted from a vector-scalar product. This is used during QRD and the residual update.
- In the scalar mode, the VMU multiplies a vector with a scalar and outputs the results in parallel.

2) Square root reciprocal: The SRR is implemented according to [13] using a look-up table to obtain a first estimation, followed by a modified Newton-Raphson step. In- and output signals are scaled to cover a higher dynamic range. Using only one Newton-Raphson iteration proved to be accurate enough for our application.

B. AMP Implementation

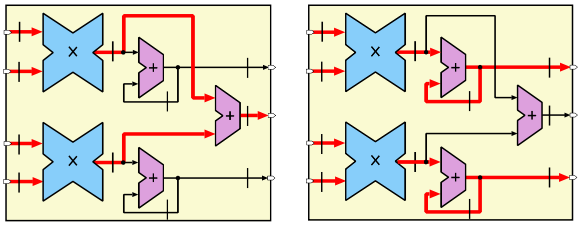
The AMP architecture uses a slightly modified VMU as central processing block. This allows to efficiently compute the matrix-vector multiplications on lines 4 and 6 of Alg. 3. A number of small additional blocks are necessary to compute the remaining operations. The L0 block calculates b by counting the number of non-zero elements. The RMSE unit is responsible for computing the threshold θ, which is then fed into the TRSH block that performs soft thresholding.

1) Vector multiplication unit (VMU): Similar to the OMP implementation of the VMU, different operation modes need to be supported. As an important difference, however, AMP needs to perform multiplications with $\boldsymbol{\Theta}$ as well as its transposed $\boldsymbol{\Theta}^T$. As the elements of $\boldsymbol{\Theta}$ are stored column-wise, i.e., only the coefficients of one row can be accessed in parallel, the two operations need to be computed with different operation principles.

- Parallel mode (Fig. 3(a)): In order to perform a multiplication with $\boldsymbol{\Theta}^T$, each entry of the resulting vector is computed by a parallel multiplication of one row of $\boldsymbol{\Theta}^T$ followed by an addition of all multiplication results in an adder tree. As this mode outputs the resulting vector serially, the subsequent thresholding operation can be computed in a simple serial unit.
- Serial mode (Fig. 3(b)): In order to perform a multiplication with $\boldsymbol{\Theta}$, each entry of the resulting vector is computed serially on a single multiplier, configured as a multiply-accumulate (MAC) unit. In this mode, the resulting vector is presented in parallel at the output of the MAC array, which is then serially stored into a RAM.

The support of those two modes makes this implementation especially suitable for measurement matrices stored in RAMs. This is in contrast to the architecture [8] which supports only one mode and requires the measurement matrix to be generated with parallel access on both, rows and columns.

2) Root mean square unit: In order to calculate the RMSE, a multiplier is instantiated to serially square and accumulate the entries of the residual. The square root is then approximated by the method proposed in [14].



(a) VMU in parallel mode

(b) VMU in serial mode

Fig. 3. AMP vector multiplication unit with $P = 2$

C. Fixed-Point Operation

To allow for fast and energy-efficient computations, fixed-point operations are used throughout the implementation. The word-widths (annoted in Fig. 1 and 2) were optimized to simultaneously obtain near-floating-point performance and fit well on the DSP structures available on the FPGA. The input and output signals of both algorithms use 18 bit. The most important parameters are the word-widths of the VMU, which are set to 18 bit and 25 bit on the inputs and 42 bit on the accumulator. The measurement matrix $\mathbf{\Theta}$ is stored with an accuracy of 25 bit.

V. Implementation Results

Both algorithms were implemented on a Xilinx Virtex-6 XC6VLX240T FPGA (speed grade -1) with the same throughput target for problems with a matrix $\mathbf{\Theta}$ of size 256×1024 and with $P = 256$ parallel multipliers. AMP is configured to perform a fixed number of $I_{max} = 40$ iterations. In the same time, OMP can perform 36 iterations, which also sets the supported sparsity to $K = 36$. The implementation results with corresponding resources usages (and device utilizations in parentheses) are shown in Table I. Note that AMP requires less than half the number of slices compared to OMP, which is mainly due to the additional register banks for $\boldsymbol{y}$ and $\boldsymbol{r}$. OMP also requires additional memories for the $\boldsymbol{Q}$ and $\boldsymbol{R}$ matrices. Since the memory storing $\mathbf{\Theta}$ dominates block RAM usage, RAMs for $\boldsymbol{Q}$ and $\boldsymbol{R}$ only lead to a small increase in RAM usage. Further, AMP achieves a higher maximum clock frequency compared to OMP, mainly because the VMU block is used in more configurations than in AMP, which results in more complex input multiplexers.

In order to test the circuitry, a UART interface to a PC has been implemented, which allows to download measurement vectors to the FPGA and to retrieve the reconstructed sparse estimates. An evaluation of the hardware was performed using four times sub-sampled images in blocks of size 32×32 pixels. The images are assumed to be sparse in the wavelet basis. Thus, $\mathbf{\Psi}$ implements a 2-dimensional wavelet transform and $\mathbf{\Phi}$ is a Gaussian matrix. For the processed example image (Lena with size 256×256), OMP achieves a reconstructed signal-to-noise ratio of 23.5 dB, while AMP achieves 21.4 dB. The performance of floating-point Matlab implementations is 0.6 dB and 0.4 dB better for OMP and AMP, respectively.

In general, OMP is faster for very sparse signals, as the processing time increases quadratically with sparsity K. Since the processing time of AMP is not strongly dependent on K, it is more efficient than OMP for problems with larger K.

Compared to a reconstruction by Matlab on a PC with one 2.6 GHz dual-core CPU, the FPGA implementation is 4000 times faster for OMP and > 5000 times faster for AMP. When scaling our OMP implementation down to the much smaller problem size of [5] and mapping it to a Virtex-5 FPGA, our implementation shows $1.5\times$ faster speed. No other hardware implementations of OMP supporting as high K as ours are known.

TABLE I
FPGA Implementation Results (Xilinx Virtex-6)

	AMP	OMP
Frequency [MHz]	165	100
Proc. time [μs]	$15.81 \cdot I_{max}$	$10.97 \cdot K + 0.59 + \sum_{l=1}^{K} 0.34 \cdot l$
Slices	12113 (32%)	32010 (84%)
Block RAMs	256 (61%)	258 (62%)
DSP slices	258 (33%)	261 (33%)

VI. Conclusions

In this paper, we presented two fast and highly parallel FPGA implementations of the OMP and AMP algorithms. Both implementations solve generic CS problems of size 256×1024 with reconfigurable measurement matrices. While AMP has lower hardware requirements and is more suitable for less sparse problems, OMP performs faster for recovery problems with, in our case, less than 36 non-zero coefficients. With a reconstruction time of only 0.63 ms for a 32×32 image block, these implementations show the feasibility of CS reconstruction for real-time applications.

References

[1] D. Dohono, “Compressed sensing,” *IEEE Trans. Inf. Theory*, vol. 52, no. 4, pp. 1289–1306, Apr. 2006.

[2] M. Andrecut, “Fast GPU implementation of sparse signal recovery from random projections,” *Engineering Letters*, vol. 17, no. 3, pp. 151–158, Aug. 2009.

[3] P. Maechler, P. Greisen, N. Felber, and A. Burg, “Matching pursuit: Evaluation and implementation for LTE channel estimation,” in *Proc. ISCAS*, Paris, France, May 2010, pp. 589–592.

[4] P. Maechler, P. Greisen, B. Sporrer, S. Steiner, N. Felber, and A. Burg, “Implementation of greedy algorithms for LTE sparse channel estimation,” in *Proc. 44th Asilomar Conf. Signals, Systems and Computers*, Pacific Grove, CA, USA, Nov. 2010, pp. 400–405.

[5] A. Septimus and R. Steinberg, “Compressive sampling hardware reconstruction,” in *Proc. ISCAS*, May 2010, pp. 3116–3119.

[6] M. Mishali *et al.*, “Generic sensing hardware and real-time reconstruction for structured analog signals,” in *Proc. ISCAS*, May 2011, pp. 1748–1751.

[7] J. Stanislaus and T. Mohsenin, “High performance compressive sensing reconstruction hardware with QRD process,” in *Proc. ISCAS*, May 2012, pp. 29–32.

[8] P. Maechler, C. Studer, D. Bellasi, A. Maleki, A. Burg, N. Felber, H. Kaeslin, and R. Baraniuk, “VLSI design of approximate message passing for signal restoration and compressive sensing,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 2, no. 3, 2012.

[9] J. Tropp and A. Gilbert, “Signal recovery from random measurements via orthogonal matching pursuit,” *IEEE Trans. Inf. Theory*, vol. 53, no. 12, pp. 4655–4666, Dec. 2007.

[10] D. Donoho, A. Maleki, and A. Montanari, “Message-passing algorithms for compressed sensing,” in *Proc. of the National Academy of Sciences*, vol. 106, no. 45, Sep. 2009, pp. 18 914–18 919.

[11] S. Chen, D. Donoho, and M. Saunders, “Atomic decomposition by basis pursuit,” *SIAM Review*, vol. 43, no. 1, pp. 129–159, Aug. 2001.

[12] G. H. Golub and C. F. Loan, *Matrix Computations*, 3rd ed. Johns Hopkins University Press, 1996.

[13] M. Ercegovac, L. Imbert, D. Matula, J. Muller, and G. Wei, “Improving goldschmidt division, square root, and square root reciprocal,” *IEEE Trans. on Computers*, vol. 49, no. 7, pp. 759–763, Jul. 2000.

[14] I. Park and T. Kim, “Multiplier-less and table-less linear approximation for square and square-root,” in *Proc. ICCD*, Oct. 2009, pp. 378–383.