6-9-2016

Fast and Scalable Architectures and Algorithms for the Computation of the Forward and Inverse Discrete Periodic Radon Transform with Applications to 2D Convolutions and Cross-Correlations

Cesar Carranza

Recommended Citation

Carranza, Cesar. "Fast and Scalable Architectures and Algorithms for the Computation of the Forward and Inverse Discrete Periodic Radon Transform with Applications to 2D Convolutions and Cross-Correlations." (2016). https://digitalrepository.unm.edu/ece_etds/44

Cesar Alberto Carranza De La Cruz

Candidate

Electrical and Computer Engineering

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

MARIOS PATTICHIS , Chairperson

RAMIRO JORDAN

VINCE CALHOUN

DANIEL LLAMOCCA

Fast and Scalable Architectures and Algorithms for the Computation of the Forward and Inverse Discrete Periodic Radon Transform with Applications to 2D Convolutions and Cross-Correlations

by

Cesar Carranza

B.Sc., Electrical Engineering, Pontificia Universidad Católica del Perú, 1994
M.Sc., Computer Science, Centro de Investigación Científica y de Educación Superior de Ensenada, 2010
M.Sc., Computer Engineering, University of New Mexico, 2012

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Engineering

The University of New Mexico

Albuquerque, New Mexico

May, 2016

Dedication

To my family, for their support.

Acknowledgments

I would like to thank my advisor, Professor Marios Pattichis, for his advice.

Fast and Scalable Architectures and Algorithms for the Computation of the Forward and Inverse Discrete Periodic Radon Transform with Applications to 2D Convolutions and Cross-Correlations

by

Cesar Carranza

B.Sc., Electrical Engineering, Pontificia Universidad Católica del Perú, 1994

M.Sc., Computer Science, Centro de Investigación Científica y de Educación Superior de Ensenada, 2010

M.Sc., Computer Engineering, University of New Mexico, 2012
PhD., Engineering, University of New Mexico, 2016

Abstract

The Discrete Radon Transform (DRT) is an essential component of a wide range of applications in image processing, e.g. image denoising, image restoration, texture analysis, line detection, encryption, compressive sensing and reconstructing objects from projections in computed tomography and magnetic resonance imaging. A popular method to obtain the DRT, or its inverse, involves the use of the Fast Fourier

Transform, with the inherent approximation/rounding errors and increased hardware complexity due the need for floating point arithmetic implementations. An alternative implementation of the DRT is through the use of the Discrete Periodic Radon Transform (DPRT). The DPRT also exhibits discrete properties of the continuous-space Radon Transform, including the Fourier Slice Theorem and the convolution property. Unfortunately, the use of the DPRT has been limited by the need to compute a large number of additions $O(N^3)$ and the need for a large number of memory accesses.

This PhD dissertation introduces a fast and scalable approach for computing the forward and inverse DPRT that is based on the use of: (i) a parallel array of fixed-point adder trees, (ii) circular shift registers to remove the need for accessing external memory components when selecting the input data for the adder trees, and (iii) an image block-based approach to DPRT computation that can fit the proposed architecture to available resources, and As a result, for an $N \times N$ image (N prime), the proposed approach can compute up to N^2 additions per clock cycle. Compared to previous approaches, the scalable approach provides the fastest known implementations for different amounts of computational resources. For the fastest case, I introduce optimized architectures that can compute the DPRT and its inverse in just $2N + \lceil \log_2 N \rceil + 1$ and $2N + 3 \lceil \log_2 N \rceil + B + 2$ clock cycles respectively, where B is the number of bits used to represent each input pixel. In comparison, the prior state of the art method required $N^2 + N + 1$ clock cycles for computing the forward DPRT. For systems with limited resources, the resource usage can be reduced to $O(N)$ with a running time of $\lceil N/2 \rceil (N + 9) + N + 2$ for the forward DPRT and

$\lceil N/2 \rceil (N+2) + 3 \lceil \log_2 N \rceil + B + 4$ for the inverse.

The results also have important applications in the computation of fast convolutions and cross-correlations for large and non-separable kernels. For this purpose, I introduce fast algorithms and scalable architectures to compute 2-D Linear convolutions/cross-correlations using the convolution property of the DPRT and fixed point arithmetic to simplify the 2-D problem into a 1-D problem. Also an alternative system is proposed for non-separable kernels with low rank using the LU decomposition. As a result, for implementations with enough resources, for a an image and convolution kernel of size $P \times P$, linear convolutions/cross correlations can be computed in just $6N + 4\log_2 N + 17$ clock cycles for $N = 2P - 1$.

Finally, I also propose parallel algorithms to compute the forward and inverse DPRT using Graphic Processing Units (GPUs) and CPUs with multiple cores. The proposed algorithms are implemented in a GPU Nvidia Maxwell GM204 with 2048 cores@1367MHz, 348KB L1 cache (24KB per multiprocessor), 2048KB L2 cache (512KB per memory controller), 4GB device memory, and compared against a serial implementation on a CPU Intel Xeon E5-2630 with 8 physical cores (16 logical processors via hyper-threading)@3.2GHz, L1 cache 512K (32KB Instruction cache, 32KB data cache, per core), L2 cache 2MB (256KB per core), L3 cache 20MB (Shared among all cores), 32GB of system memory. For the CPU, there is a tenfold speedup using 16 logical cores versus a single-core serial implementation. For the GPU, there is a 715-fold speedup compared to the serial implementation. For real-time applications, for an 1021x1021 image, the forward DPRT takes 11.5ms and 11.4ms for the

inverse.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Discrete Radon Transform (DRT) is an essential component of a wide range of applications in image processing [4, 5]. Applications of the DRT include the classic application of reconstructing objects from projections in computed tomography, radar imaging, and magnetic resonance imaging [4, 5]. Furthermore, the DRT has also been applied in image denoising [6], image restoration [7], texture analysis [8], line detection in images [9], and encryption [10]. More recently, the DRT has been applied in erasure coding in wireless communications [11], signal content delivery [12], and compressive sensing [13].

A popular method for computing the DRT involves the use of the Fast Fourier Transform (FFT). The basic approach is to sample the 2-D FFT along different radial lines through the origin and then use the 1-D inverse FFT along each line to estimate the DRT. This direct approach suffers from many artifacts that have been discussed in [6]. Assuming that the DRT is computed directly, Beylkin proposed an exact inversion algorithm in [14]. A significant improvement to this approach was proposed by Kelley and Madisetti by eliminating interpolation calculations [15].

The initial motivation of the current dissertation is to investigate the development of DPRT algorithms that are both fast and scalable. Here, I use the term *fast* to refer to the requirement that the computation will provide the result in the minimum number of cycles. Also, I use the term *scalable* to refer to the requirement that the approach will provide the fastest implementation based on the amounts of available resources. The dissertation also develops new, efficient, parallel algorithms for computing the DPRT and its inverse on GPUs and multi-core CPUs.

The scope of the dissertation was then expanded to cover convolution and cross-correlation applications. The interest in convolution and cross-correlation is due to the fact that these operations are essential in a wide range of applications in the field of image and video processing. The performance of most image processing systems is directly affected by the speed at which we can perform 2-D convolutions. There is thus perennial interest in developing fast methods for computing 2-D convolutions. There is also renewed interest in developing fast convolution methods that can fit in new devices. The dissertation presents novel implementations of 2-D convolutions and cross-correlations that can be computed as fast as $O(N)$ clock cycles provided that we have the available resources.

1.1 Thesis Statement

I believe that it is possible to develop fast and scalable architectures and algorithms for the computation of the Discrete Periodic Radon Transform (DPRT) and its inverse (iDPRT), that will enable the application of the DPRT in different areas where its use was limited due the lack of a fast implementation (e.g., in 2D convolutions and cross-correlations). Furthermore, I believe that it is possible to develop highly efficient, parallel DPRT and iDPRT algorithms on existing GPUs and multi-core CPUs.

1.2 Contributions

A list of the main contribution includes:

- A scalable and fast framework for computing the DPRT and its inverse. The dissertation develops a set of parallel algorithms and associated scalable architectures to compute the forward and inverse DPRT of an $N \times N$ image that allows effective implementations based on different constraints on running time and resources. In terms of resources and running time, the scalable framework provides optimal configurations in the multi-objective sense. In terms of performance, the fastest architecture computes the DPRT in linear time (with respect to N). This is the fastest implementation to date.

- A scalable and fast framework for computing convolutions and cross correlations for relatively large image sizes (of the order of the image size). Scalability is based on the scalable DPRT framework, the scalable computation of 1D convolutions in the DPRT domain, LU decompositions, and the use of an overlap and add approach. Similar to the FFT, the DPRT can be used for computing linear convolutions using zero-padding. To compute 2-D linear convolutions between an image of size $P_1 \times P_2$ and a relatively large (of the order of the image size) and non-separable kernel $Q_1 \times Q_2$, we can use DPRTs of size of $N \times N$ where $N = \texttt{NextPrime}(\texttt{max}(P_1+Q_1-1, P_2+Q_2-1))$. In terms of resources and running time, each solution (by itself) is optimal in the multi-objective sense. When the rank of the non-separable kernel is low, the framework based on the LU decomposition becomes the optimal solution, and for high-rank kernels, the framework based on the DPRT is the optimal solution. In terms of performance, the fastest architecture based on the DPRT computes the 2-D linear convolution in linear time (with respect to N). And for low rank kernels, the fastest architecture based on the LU decomposition computes the 2-D linear convolution in linear time.

- A scalable and fast framework for computing the forward and inverse DPRT using GPUs and multi-coure CPUs. Scalability on the GPUs is a function of the number of multi-processors (MIMD) and their associated cores (SIMD). For the CPUs, scalability is a function of the number of cores (MIMD).

1.2.1 Specific contributions for the Scalable and Fast DPRT and its inverse

Specific contributions over the best previous algorithms, architectures and practical implementations of the DPRT are as follows:

- ***Fast and scalable architecture that can be adapted to available resources***: The proposed approach is designed to be fast in the sense that column sums are computed on every clock cycle. In the fastest implementation, a prime direction of the FDPRT is computed on every clock cycle. More generally, the approach is scalable, allowing to handle larger images with limited computational resources.

- ***Pareto-optimal DPRT and iDPRT based on running time and resources***: The proposed approach is shown to be Pareto-optimal in terms of the required cycles and required resources. Thus, as compared to previous approaches, the scalable approach provides the fastest known implementations for the given computational resources. As an example, in the fastest case, for an $N \times N$ image (N prime), the DPRT is computed in linear time ($2N + \lceil \log_2 N \rceil + 1$ clock cycles) requiring resources that grow quadratically ($O(N^2)$). In the most limited resources case, the running time is quadratic ($\lceil N/2 \rceil (N + 9) + N + 2$ clock cycles) requiring resources that grow linearly ($O(N)$). A Pareto-front of optimal solutions is given for resources that fall within these two extreme cases. All prior research in this area focused on the development of a single architecture. Furthermore, when sharing comparable computational resources, the proposed approach is always better than previously published approaches. For example, in [2], the authors reported the fastest previous implementation that required $N^2 + N + 1$ clock cycles requiring resources that grow quadratically with N. Similar results are obtained for the inverse DPRT, although results for this case were not previously reported.

- ***Fastest possible implementation of the FDPRT and iFDPRT***: For the fastest case, it is shown that the scalable architecture can be further reduced to obtain the FDPRT and iFDPRT in $2N + \lceil \log_2 N \rceil + 1$ and $2N + 3\lceil \log_2 N \rceil + B + 2$ cycles respectively (B is the number of bits used to represent each input pixel).

- ***Parallel and pipelined implementation***: Parallel and pipelined implementations are proposed providing an improvement over the sequential algorithm proposed by [16] and used in [1],[2]. For $H = 2, \ldots, N$, the scalable approach computes $N \times H$ additions in a single clock cycle. Furthermore, shift registers are used to make data available to the adders in every clock cycle. Then, additions and shifts are performed in parallel in the same clock cycle.

- ***Unique fast transposition method***: A RAM-based architecture and associated algorithm that provides a complete row or column of the input image in one clock cycle. Using this parallel RAM access architecture, transposition is avoided since the image can be accessed by either rows or columns.

- ***Generic architectures***: The proposed architectures are not tied to any particular hardware. They can be applied to any existing hardware (e.g., FPGA or VLSI).

1.2.2 Specific contributions for the Fast 2-D Convolutions and Cross-Correlations Using Scalable Architectures

The contributions are listed in terms of the DPRT and LU based methods. We begin with contributions for both the DPRT and LU frameworks and then present the specific contributions for each framework.

Specific contributions over the best previous 2-D convolution/cross-correlation systems for non-separable and relatively large kernels are as follows:

- ***Fast and scalable architectures that can be adapted to available resources for both frameworks***: The proposed approaches are designed to be fast because of the mapping of 2-D convolutions into fast 1-D convolutions. For the DPRT framework, scalability comes from the control of the number of 1-D convolution kernels. For the LU framework, scalability comes from the separability of the kernels and its decomposition into low-rank 1D kernels. For the fastest implementations, a throughput of N convolved pixels per clock cycle is achieved.

- ***Pareto-optimal 2-D convolutions and cross-correlations based on running time and resources for both frameworks***: The proposed approaches are shown to be Pareto-optimal in terms of the required cycles and required resources. Thus, as compared to previous approaches, the scalable approach provides the fastest known implementations for the given computational resources. For each framework, a Pareto-front of optimal solutions is given for resources that fall within the fastest and the slowest running time. The proposed approach is always better than previously published approaches for large and non-separable kernels.

- ***Generic architectures for both frameworks***: The proposed architectures are not tied to any particular hardware. They can be implemented in an FPGA.

- ***Custom SRAM architectures to provide fast transposition and accumulation of the results for computing fast cross-correlations and the LU framework***: The dissertation presents different RAM-based architectures and associated algorithms that allow access, storage or accumulation of the results from a row or column in a single clock cycle. Using the custom SRAM architectures, transposition is avoided since the partial results can be accessed by either rows or columns.

- ***Parallel and pipelined 1-D circular convolvers for the DPRT framework***: The DPRT framework loads N pixels in a single clock cycle and computes one output pixel per clock cycle.

- ***Parallel and pipelined 1-D linear convolvers for LU framework***: The LU framework loads a complete row of pixels in a single clock cycle and computes one output pixel per clock cycle.

1.2.3 Specific contributions for the computation of the DPRT and its inverse on multi-core CPUs and GPUs

The contributions are as follows:

- ***Parallel implementations of the DPRT and iDPRT on multi-core CPUs***: The parallel implementations distribute processing of the prime directions among all the logical cores available. Compared to a serial (single-core) implementation, the proposed approach achieved a tenfold speedup with 16 logical cores (C and Pthreads implementation).

- ***New parallel and memory efficient DPRT and iDPRT implementations on GPUs***: The proposed approach distributes the computation of the prime directions among the multiprocessors (MP). Within each MP, the rays associated with each prime direction are distributed among the cores. The proposed algorithms were coded in C/CUDA where: (i) parallel threads were synchronized to always read the same row of pixels at the same time, (ii) efficient memory access is achieved by enforcing row-major ordering for reads and writes, and (iii) further specific optimizations for the GPU Nvidia GeForce GTX980 were applied (pixel width, optimal concurrency of warps, fast address calculation and modulo operation masking). For 16 multi-process with 128 cores each, compared to a serial implementation, the speedup is of the order of 715 (max=853x for DPRT, max=873x for inverse DPRT).

1.3 Overview of Dissertation

This dissertation is organized into 5 chapters. The work described in each chapter is presented below.

Chapter 2 presents a new scalable approach to compute the DPRT and its inverse that balances the use hardware resources versus execution time.

Chapter 3 presents a fast and scalable framework for computing 2-D linear convolutions and cross-correlations for relatively large and non-separable kernels.

Chapter 4 presents new parallel algorithms for the computation of the forward and inverse DPRT on CPUs and GPUs. The proposed algorithms are implemented in currently available hardware.

Chapter 5 presents conclusions and future work.

Also, on the appendices I include a summary of my research (App. A), two algorithms for the computation of flip-flops, full adders and muxes in adder trees (App. B, C) and the source code for the CPU and GPU implementations of the DPRT and its inverse (App. D, E,F).

Chapter 2

Scalable Fast Discrete Periodic Radon Transform and its inverse

This chapter describes Fast and Scalable architectures and algorithms for computing the DPRT and its inverse (iFDPRT). The work presented here has been published in:

C. Carranza, D. Llamocca, and M. Pattichis, “Fast and scalable computation of the forward and inverse discrete periodic radon transform,”, IEEE Transactions on Image Processing, vol. 25, no. 1, pp. 119-133, Jan 2016.

C. Carranza, D. Llamocca, and M. Pattichis, “A scalable architecture for implementing the fast discrete periodic radon transform for prime sized images,” in 2014 IEEE International Conference on Image Processing (ICIP), Oct 2014, pp. 1208-1212.

C. Carranza, D. Llamocca, and M. Pattichis, “The fast discrete periodic radon transform for prime sized images: Algorithm, architecture, and vlsi/fpga implementation," in 2014 IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI), April 2014, pp. 169-172.

In what follows, the content of the first journal have been merged with extracts of the other two in order to provide a better understanding to the reader.

The discrete periodic radon transform (DPRT) has been extensively used in applications that involve image reconstructions from projections. Beyond classic applications, the DPRT can also be used to compute fast convolutions that avoids the use of floating-point arithmetic associated with the use of the Fast Fourier Transform. Unfortunately, the use of the DPRT has been limited by the need to compute a large number of additions and the need for a large number of memory accesses.

This chapter introduces a fast and scalable approach for computing the forward and inverse DPRT that is based on the use of: (i) a parallel array of fixed-point adder trees, (ii) circular shift registers to remove the need for accessing external memory components when selecting the input data for the adder trees, (iii) an image block-based approach to DPRT computation that can fit the proposed architecture to available resources, and (iv) fast transpositions that are computed in one or a few clock cycles that do not depend on the size of the input image.

As a result, for an $N \times N$ image (N prime), the proposed approach can compute up to N^2 additions per clock cycle. Compared to previous approaches, the scalable approach provides the fastest known implementations for different amounts of computational resources. For example, for a 251×251 image, for approximately

25% fewer flip-flops than required for a systolic implementation, the scalable DPRT is computed 36 times faster. For the fastest case, optimized architectures are presented that can compute the DPRT and its inverse in just $2N + \lceil \log_2 N \rceil + 1$ and $2N + 3 \lceil \log_2 N \rceil + B + 2$ cycles respectively, where B is the number of bits used to represent each input pixel. On the other hand, the scalable DPRT approach requires more 1-bit additions than for the systolic implementation and provides a trade-off between speed and additional 1-bit additions.

2.1 Introduction

Fixed point implementations of the DRT can be based on the Discrete Periodic Radon Transform (DPRT). Grigoryan first introduced the forward DPRT algorithm for computing the 2-D Discrete Fourier Transform as discussed in [17]. In related work, Matus and Flusser presented a model for the DPRT and proposed a sequential algorithm for computing the DPRT and its inverse for prime sized images [16]. This research was extended by Hsung et al. for images of sizes that are powers of two [18].

Similar to the continuous-space Radon Transform, the DPRT satisfies discrete and periodic versions of the the Fourier slice theorem and the convolution property. Thus, the DPRT can lead to efficient, fixed-point arithmetic methods for computing circular and linear convolutions as discussed in [18]. The discrete version of the Fourier slice theorem provides a method for computing 2-D Discrete Fourier Transforms based on the DPRT and a minimal number of 1-D FFTs (e.g., [17, 19]).

A summary of DPRT architectures based on the algorithm described by [16] can be found in [20]. In [16], the DPRT of an image of size $N \times N$ (N prime) requires $(N + 1)N(N - 1)$ additions. Based on the algorithm given in [16], a serial and power efficient architecture was proposed in [1]. In [1], the authors used an address generator to generate the pixels to add. The DPRT sums were computed using an accumulator adder that stores results from each projection using N shift registers. The serial architecture described in [1] required resources that grow linearly with the size of the image while requiring $N(N^2 + 2N + 1)$ clock cycles to compute the full DPRT.

Also based on the algorithm given in [16], a systolic architecture implementation was proposed in [2]. The architecture used a systolic array of $N(N+1)(\log_2 N)$ bits to store the addresses of the values to add. The pixels are added using using $(N+1)$ loop adder blocks. The data I/O was handled by $N + 1$ dual-port RAMs. For this architecture, resource usage grows as $O(N^2)$ at a reduced running time of N^2+N+1 cycles for the full DPRT.

The motivation for the current chapter is to investigate the development of DPRT algorithms that are both fast and scalable. Recall that *fast* refers to the requirement that the computation will provide the result in the minimum number of cycles. On the other hand, *scalable* refers to the requirement that the approach will provide the fastest implementation based on the amounts of available resources.

This chapter is focused on the case that the image is of size $N \times N$ and N is prime. For prime N, the DPRT provides the most efficient implementations by requiring the minimal number of $N + 1$ primal directions [21]. In contrast, there are $3N/2$ primal directions in the case that $N = 2^p$ where p is a positive integer [22]. On the other hand, despite the additional directions, it is possible to compute the directional sums faster for $N = 2^p$, as discussed in [23, 24]. However, it is important to note that prime-numbered transforms have advantages in convolution applications. Here, just like for the Fast Fourier Transform (FFT), zero-padding can be used to extend the DPRT for computing convolutions in the transform domain. Unfortunately, when using the FFT with $N = 2^p$, zero-padding requires the use of FFTs with double the size of N. In this case, it is easy to see that the use of prime-numbered DPRTs is better since there are typically many prime numbers between 2^p and 2^{p+1}. For example, it can be shown that the n-th prime number is approximately $n \log(n)$ which gives an approximate sequence of primes that are $n \log(n), (n+1)\log(n+1)$ which is a lot more dense than what can be accomplished with powers of two $2^n, 2^{n+1}$ [25]. As a numerical example, there are 168 primes that are less than 1000 as opposed to just 9 powers of 2. Thus, instead of doubling the size of the transform, It can be used a DPRT with only a slightly larger transform.

This chapter introduces a fast and scalable approach for computing the forward and inverse DPRT that is based on parallel shift and add operations. Preliminary results were presented in conference publications in [26, 27]. The conference paper implementations were focused on special cases of the full system discussed here, required an external system to add the partial sums, assumed pre-existing hardware for transpositions, and worked with image strip-sizes that were limited to powers of two. The current chapter includes: (i) a comprehensive presentation of the theory and algorithms, (ii) extensive validation that does not require external hardware for partial sums and transpositions, (iii) works with arbitrary image strip sizes, and also includes (iv) the inverse DPRT. In terms of the general theory presented in this chapter, the conference paper publications represented some special cases. The contributions of the current chapter over previously proposed approaches are summarized in the following paragraphs.

Overall, a fundamental contribution of the chapter is that it provides a fast and scalable architecture that can be adapted to available resources. The approach is designed to be fast in the sense that column sums are computed on every clock cycle. In the fastest implementation, a prime direction of the DPRT is computed on every clock cycle. More generally, the approach is scalable, allowing to handle larger images with limited computational resources.

Furthermore, this chapter provides a Pareto-optimal DPRT and inverse DPRT based on running time and resources measured in terms of one-bit additions (or 1-bit full-adders) and flip-flops. Thus, the proposed approach is shown to be Pareto-optimal in terms of the required cycles and required resources. Here, Pareto-optimality refers to solutions that are optimal in a multi-objective sense (e.g., see [28]). Thus, in the current application, Pareto-optimality refers to the fact that the scalable approach provides the fastest known implementations for the given computational resources. As an example, in the fastest case, for an $N \times N$ image (N prime), the DPRT is computed in linear time ($2N + \lceil \log_2 N \rceil + 1$ clock cycles) requiring resources that grow quadratically ($O(N^2)$). In the most limited resources case, the running time is quadratic ($\lceil N/2 \rceil (N + 9) + N + 2$ clock cycles) requiring resources that grow linearly ($O(N)$). A Pareto-front of optimal solutions is given for resources that fall within these two extreme cases. All prior research in this area focused on the development of a single architecture. Similar results are obtained for the inverse DPRT, although results for this case were not previously reported.

In terms of speed, this chapter describes the fastest possible implementation of the DPRT and inverse DPRT. For the fastest cases, assuming sufficient resources for implementation, the fast DPRT (FDPRT) and the fast inverse DPRT (iFDPRT) are introduced to compute the full transforms in $2N + \lceil \log_2 N \rceil + 1$ and $2N + 3 \lceil \log_2 N \rceil + B + 2$ cycles respectively (B is the number of bits used to represent each input pixel).

To achieve the performance claims, parallel and pipelined implementations are described providing an improvement over the sequential algorithm proposed by [16] and used in [1],[2]. To summarize the performance claims, let the $N \times N$ input image be sub-divided into strips of H rows of pixels. Then, for $H = 2, \ldots, (N-1)/2$, this scalable approach computes $N \times H$ additions in a single clock cycle. Furthermore, shift registers are used to make data available to the adders in every clock cycle. Then, additions and shifts are performed in parallel in the same clock cycle.

In addition, the use of fast transpositions is presented. A unique transpositions method is proposed based on a RAM-based architecture and associated algorithm that provides a complete row or column of the input image in one clock cycle. Using this parallel RAM access architecture, transposition is avoided since the image can be accessed by either rows or columns.

Finally, a generic family of architectures is provided. Thus, the proposed architectures are not tied to any particular hardware. They can be applied to any existing hardware (e.g., FPGA or VLSI).

The rest of the chapter is organized as follows. The mathematical definitions for the DPRT and its inverse along with previous DPRT implementations are given in section 2.2. The proposed approach is given in section 2.3. Section 2.4 describes the architecture implementation details. Section 2.6 presents the results. Conclusions and future work are given in section 2.7.

2.2 Background

The purpose of this section is to introduce the basic definitions associated with the DPRT and provide a very brief summary of previous implementations. The notation is introduced in section 2.2.1. Then the definitions of the DPRT and its inverse are produced in section 2.2.2. A summary of previous implementations is given in section 2.2.3.

2.2.1 Notation summary

Consider $N \times N$ images where N is prime. Let Z_N denote the non-negative integers: $\{0, 1, 2, \ldots, N-1\}$, and $l^2(Z_N^2)$ be the set of square-summable functions over Z_N^2. Then, let $f \in l^2(Z_N^2)$ be a 2-D discrete function that represents an $N \times N$ image, where each pixel is a positive integer value represented with B bits. Also, subscripts are used to represent rows. For example, $f_k(j)$ denotes the vector that consists of the elements of f where the value of k is fixed. Similarly, for $R(r, m, d)$, $R_{r,m}(d)$ denotes the vector that consists of the elements of R with fixed values for r, m. Here, note that all but the last index is fixed. $\langle\alpha\rangle_\beta$ is used to denote the modulo function. In other words, $\langle\alpha\rangle_\beta$ denotes the positive remainder when α is divided by β where $\alpha, \beta > 0$.

To establish the notation, consider the following an example. For an 251 × 251 8-bit image, we have $N = 251$, $B = 8$, and f represents the image. Then, $f_1(j)$ represents the row number one in the image. In 3-dimensions, $R_{1,2}(d)$ denotes the elements $R(r = 1, m = 2, d)$, where d id allowed to vary. For the modulo-notation, $\langle 255\rangle_{251} = 4$ represents the integral remainder when 255 is divided by $N = 251$. $R(m, d)$ is used to denote the DPRT of f and $R'(r, m, d)$ to index the r-th partial sum associated with $R(m, d)$. Here, R' is used for explaining the computations associated with the scalable DPRT.

2.2.2 Discrete Periodic Radon Transform and its Inverse

The definition of the DPRT and its inverse (iDPRT) based on [18] is given as follows. Let f be square-summable. The DPRT of f is also square summable and given by:

$$R(m,d) = \begin{cases} \sum_{i=0}^{N-1} f(i, \langle d + mi\rangle_N), & 0 \le m < N, \\ \\ \sum_{j=0}^{N-1} f(d, j), & m = N, \end{cases} \tag{2.1}$$

where $d \in Z_N$ and $m \in Z_{N+1}$. The row vector k of $R(m, d)$ is denoted as $R_k(d)$ which represents the k projection of $f(i, j)$. Fig. 2.1 provides an illustration of the

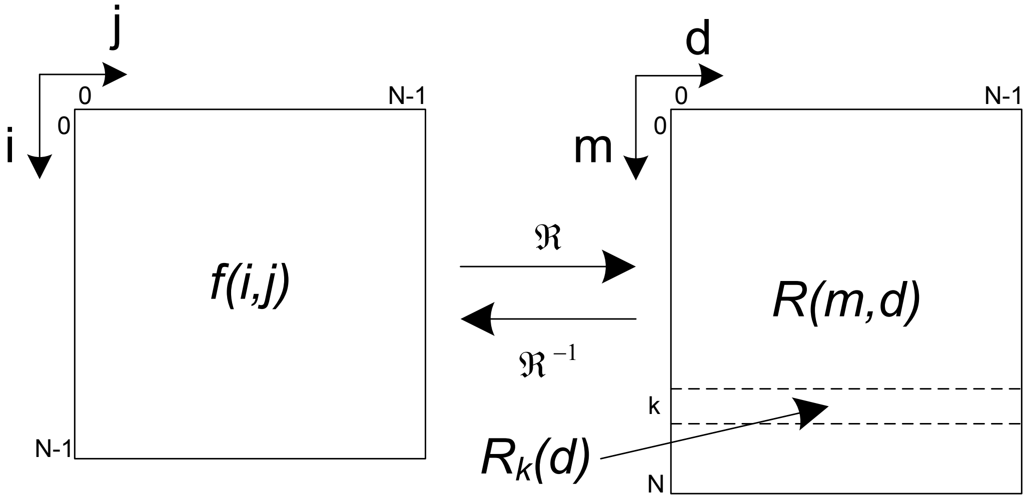


Figure 2.1: Illustration of the DPRT and its iDPRT for a function f of size $N \times N$, where N is prime. Each row of $R(m, d)$, denoted as a vector $R_k(d)$, $k = 0, \ldots, N$, represents a projection of $f(i, j)$.

DPRT applied to a discrete image $f(i, j)$ of size $N \times N$, where N is prime; also the k projection of $f(i, j)$, $R_k(d)$, is shown, $k = 0, \ldots, N$.

Observe that the summations in (2.1) are done over the discrete periodic line segment parameterized by $(i, \langle d + mi \rangle_N)$ for the projections captured with $0 \leq m < N$ and (d, j) for the projection captured with $m = N$. The projections are given along the directional vectors $(1, m)$ for $0 \leq m < N$ and $(0, 1)$ for $m = N$ which represent the *prime directions*. The prime directions of an 7×7 image $f(i, j)$ are shown in Fig. 2.2(a), and an example of one periodic line segment (defined by the prime direction (1,2)) used to compute $R(2, 0)$ is shown in Fig. 2.2(b).

The iDPRT recovers the input image as given by:

$$f\left(i,j\right) = \frac{1}{N}\left[\sum_{m=0}^{N-1} R\left(m, \langle j - mi \rangle_N\right) - S + R\left(N, i\right)\right] \tag{2.2}$$

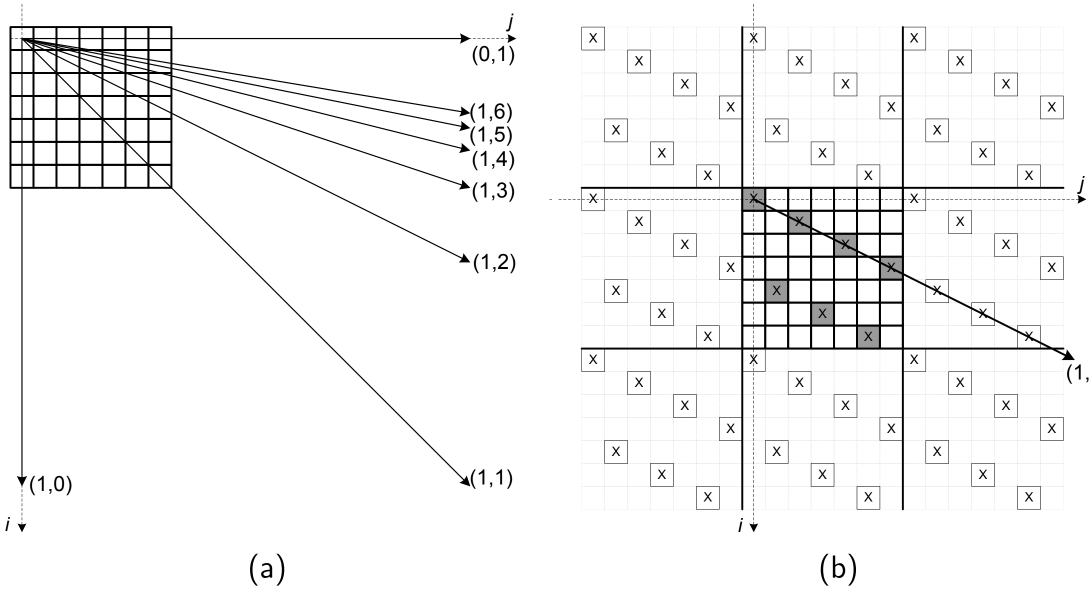


Figure 2.2: DPRT Example for a 7 × 7 image. (a) Prime directions; (b) Main image (at center with bold boxes) and its periodic extensions. Pixels marked with ×: samples along periodic line for prime direction $(1, 2)$, pixels marked with × in grey boxes are added to compute $R(2, 0)$

where:

$$S = \sum_{j=0}^{N-1} \sum_{i=0}^{N-1} f(i, j). \tag{2.3}$$

From (2.3), it is clear that S represents the sum of all of the pixels. Since each projection computes the sums over a single direction, the results can be added from any one of these directions to compute S as given by:

$$S = \sum_{d=0}^{N-1} R(m, d). \tag{2.4}$$

Note that the DPRT as given by (2.1) requires the computation of $N + 1$ projections. All of these projections are used in the computation of iDPRT as given in (2.2). In (2.2), the last projection computes $R(N, i)$ that is needed in the summation.

2.2.3 previous DPRT implementations

DPRT implementations have focused on implementing the algorithm proposed in [16]. The basic algorithm is sequential that relies on computing the indices i, j to access $f(i, j)$ that are needed for the additions in (2.1). For each prime direction, as shown in (2.1), the basic implementation requires N^2 memory accesses and $N(N-1)$ additions. For computing all of the prime directions $(N + 1)$, $(N + 1)N^2$ memory accesses and $(N + 1)N(N - 1)$ additions are required.

Based on [16], hardware implementations have focused on computing memory indices, followed by the necessary additions [1], [2]. An advantage of the serial architecture given in [1] is that it requires hardware resources that grow linearly with N (for and $N \times N$ image). Unfortunately, this serial architecture leads to slow computation since it computes the DPRT in a cubic number of cycles ($N(N^2+2N+1)$ clock cycles). A much faster, systolic array implementation was presented in [2]. The systolic array implementation computes N indices and N additions per cycle. Overall, the systolic array implementation requires hardware resources that grow quadratically with N while requiring $N^2 + N + 1$ clock cycles to compute the full DPRT.

The proposed architecture does not require memory indexing and computes the additions in parallel. Furthermore, the new architecture is scalable, and thus allows to consider a family of very efficient architectures that can also be implemented with limited resources.

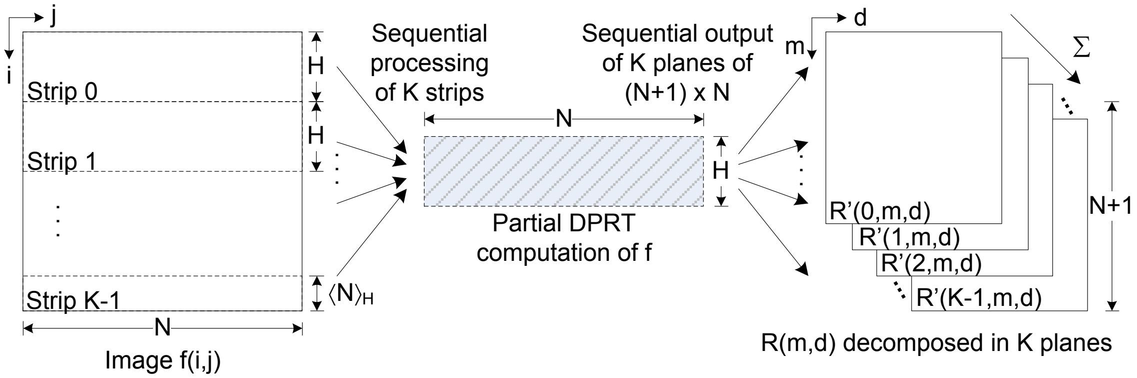

j
i
Strip 0
Strip 1
Strip K-1
H
H
⟨N⟩H
N
Image f(i,j)
Sequential processing of K strips
N
H
Partial DPRT computation of f
Sequential output of K planes of (N+1) x N
d
m
Σ
R'(0,m,d)
R'(1,m,d)
R'(2,m,d)
R'(K-1,m,d)
N+1
R(m,d) decomposed in K planes

Figure 2.3: Scalable DPRT concept. The input image is divided into K strips. The DPRT is computed by accumulating the partial sums from each strip.

2.3 Methodology

This section presents a new fast algorithm and associated scalable architecture that can be used to control the running time and hardware resources required for the computation of the DPRT. Additionally, the approach to the inverse DPRT (iDPRT) is extended. At the end of the section, an optimized architecture implementation that computes the DPRT and iDPRT in the least number of clock cycles is provided

.

2.3.1 Partial DPRT

For the development of scalable architecture implementations, the concept of the partial DPRT is introduced. The basic concept is demonstrated in Fig. 2.3 and formally defined below.

The idea is to divide f into strips that contain H rows of pixels except for the last one that is composed of the remaining number of rows needed to cover all of the

N rows (see Fig. 2.3). Here, note that the height of the last strip will be $\langle N\rangle_H \neq 0$ since N is prime. Now, let K be the number of strips, then $K = \lceil N/H \rceil$. In what follows, let r denote the r-th strip. the DPRT over each strip is computed using:

$$R(m,d) = \begin{cases} \sum_{r=0}^{K-1} \sum_{i=0}^{L(r)-1} f(i+rH, \langle d+m(i+rH)\rangle_N), \\ \qquad\qquad\qquad\qquad\qquad\qquad 0 \leq m < N \\ \\ \sum_{r=0}^{K-1} \sum_{j=0}^{L-1} f(d, j+rH), \qquad\qquad m = N \end{cases} \tag{2.5}$$

where

$$L(r) = \begin{cases} H, & r < K-1 \\ \\ \langle N\rangle_H & r = K-1. \end{cases} \tag{2.6}$$

Let $R'(r,m,d)$ denote the r-th partial DPRT defined by:

$$R'(r,m,d) = \begin{cases} \sum_{i=0}^{L(r)-1} f(i+rH, \langle d+m(i+rH)\rangle_N), \\ \qquad\qquad\qquad\qquad\qquad\qquad 0 \leq m < N \\ \\ \sum_{j=0}^{L(r)-1} f(d, j+rH), \qquad\qquad m = N \end{cases} \tag{2.7}$$

where, $r = 0, \ldots, K-1$ is the strip number. Therefore, the DPRT can be computed as a summation of partial DPRTs using:

$$R(m,d) = \sum_{r=0}^{K-1} R'(r,m,d). \tag{2.8}$$

Similarly, the partial iDPRT of $R(m,d)$ is defined using

$$f'(r,i,j) = \sum_{m=0}^{L(r)-1} R(m+rH, \langle j-i(m+rH)\rangle_N) \tag{2.9}$$

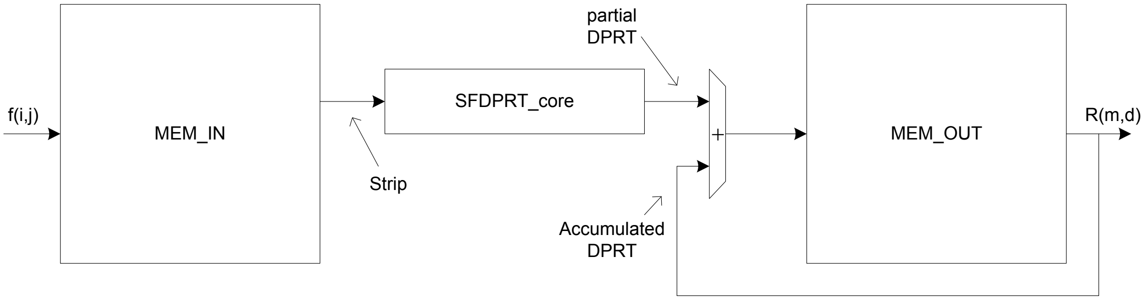


Figure 2.4: Top level system for implementing the Scalable and Fast DPRT (SFDPRT). The `SFDPRT_core` computes the partial sums. `MEM_IN` and `MEM_OUT` are dual port input and output memories. A Finite State Machine (`FSM`, not shown in the figure) is needed for control. See text in Sec. 2.3.2 for more details.

which allows to compute the iDPRT of $R(m,d)$ using a summation of partial iDPRTs:

$$f(i,j) = \frac{1}{N}\left[\sum_{r=0}^{K-1} f'(r,i,j) - S + R\left(N,i\right)\right]. \tag{2.10}$$

In what follows, let $n = \lceil \log_2 N \rceil$, $h = \lceil \log_2 H \rceil$, and $R'_{r,m}(d)$ be an N-th dimensional vector representing the partial DPRT of strip r.

2.3.2 Scalable Fast Discrete Periodic Radon Transform (SFDPRT)

In this section, the scalable DPRT hardware architecture is developed by implementing the partial DPRT concepts presented in Fig. 2.3. The top-level view of the hardware architecture for the scalable DPRT is presented in Fig. 2.4 and the associated algorithm in Fig. 2.5. Refer to Fig. 2.3 for the basic concepts. The basic idea is to achieve scalability by controlling the number of rows used in each rectangular strip. Thus, for the fastest performance, the largest pareto-optimal strip size that can be implemented using available hardware resources is chosen. The final result is computed by combining the DPRTs as given in (2.7).

An overview of the architecture is presented in Fig. 2.4. Here, we have three basic hardware blocks: the input memory block (`MEM_IN`), the partial DPRT computation block (`SFDPRT_core`), and output/accumulator memory block (`MEM_OUT`). The input image f is loaded into the input buffer `MEM_IN` which can be implemented using a customized RAM that supports access to each image row or column in a single clock cycle. Partial DPRT computation is performed using the `SFDPRT_core`. `SFDPRT_core` is implemented using an $H \times N$ register array with B bits depth so as to be able to store the contents of a single strip. Each row of the `SFDPRT_core` register array is implemented using a Circular Left Shift (CLS) register that can be used to align the image samples along each column. Each column of this array has a H-operand fully pipelined adder tree capable to add the complete column in one clock cycle. The output of the adder trees provide the output of the `SFDPRT_core`, which represents the partial DPRT of f. This combination of shift registers and adders allows the computation of $H \times N$ additions per clock cycle with a latency of h. At the end, the outputs of the `SFDPRT_core` are accumulated using `MEM_OUT`. The required computational resources are summarized in section 2.6.

A fast algorithm for computing the DPRT is summarized in Fig. 2.5. Also a detailed timing diagram for each of the steps is presented in Fig. 2.6. For the timing diagram, note that time increases to the right. Along the columns, each step and the required number of cycles is labeled. Furthermore, computations that occur in parallel will appear along the same column. To understand the timing for each

```
 1: Load_shifted_image (f) in MEM_IN
          using CLS registers of SFDPRT_core.
 2: for $r = 0$ to $K - 1$ do
 3:     Load_strip(r,‘row_mode’) into the SFDPRT_core
 4:     for $k = 0$ to $N - 1$ do
 5:         Shift in parallel all the $H$ rows:
                $CLS_a(H \cdot r + a)$, $a = 0, \ldots, H - 1$
 6:         Compute in parallel $R'_{r,k}(d)$
 7:         Add_partial_result: $R_k(d) = R_k(d) + R'_{r,k}(d)$
                in MEM_OUT
 8:     end for
 9: end for
10: for $r = 0$ to $K - 1$ do
11:     Load_strip (r,‘column_mode’) into the SFDPRT_core
12:     Compute in parallel $R'_{r,N}(d)$
13:     Add_partial_result: $R_N(d) = R_N(d) + R'_{r,N}(d)$
            in MEM_OUT
14: end for
```

Figure 2.5: Top level algorithm for computing the scalable and fast DPRT (SFDPRT). Within each loop, all of the operations are pipelined. Then, each iteration takes a single cycle. For example, the Shift, pipelined Compute, and the Add operations of lines 5, 6, and 7 are always computed within a single clock cycle. Refer to section 2.2.1 for the notation.

computation, recall that N denotes the number of image rows, K denotes the number of image strips where each strip contains a maximum of H image rows.

Furthermore, to explain the reduced timing requirements, note the special characteristics of the pipeline structure. First, dual port RAMs (MEM_IN and MEM_OUT) are used to allow to load and extract one image row per cycle. Thus, the computation of the first projection can be started while shifting (also see the overlap between the second and third computing steps of Fig. 2.6). Second, note the use of fully pipelined adder trees which allows to start the computation of the next projection without requiring the completion of the previous projection (see overlap in projection

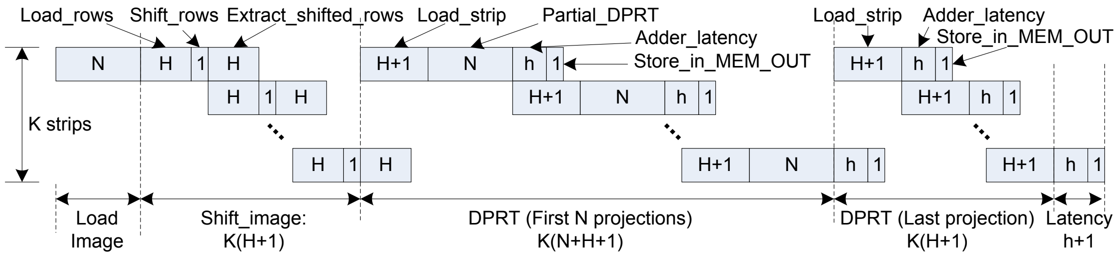


Figure 2.6: Running time for scalable and fast DPRT (SFDPRT). In this diagram, time increases to the right. The image is decomposed into K strips. Then, the first strip appears in the top row and the last strip appears in the last row of the diagram. Here, H denotes the maximum number of image rows in each strip, $K = \lceil N/H \rceil$ is the number of strips, and $h = \lceil \log_2 H \rceil$ represents the addition latency.

computations in Fig. 2.6).

Next, a summary of the entire process is depicted in Figs. 2.5 and 2.6. Initially, a shifted version of the image is loaded into MEM_IN. The significance of this step is that the stored image allows computation of the last projection in a single cycle without the need for transposition. Here, note that rows and columns of MEM_IN can be accessed in a single clock cycle. In terms of timing, the process of loading and shifting in the image requires $N + K(H + 1)$ cycles.

Then, the first N projections are computed by loading each one of the K strips (outer loop) and then adding the partial results (inner loop). The partial DPRT for the strip r is computed in the inner loop, (see lines 4 - 8 in Fig. 2.5). For computing the full DPRT, the partial DPRT outputs are accumulated in MEM_OUT. In terms of timing, each strip requires $N + H + 1$ cycles as detailed in Fig. 2.6. Thus, it takes a total of $K(N + H + 1)$ cycles for computing the first N projections.

For the last projection, note the requirement for special handling (see lines 10-14 in Fig. 2.5). This special treatment is due to the fact that unlike the first

N projections that can be implemented effectively using shift and add operations of the rows, the last projection requires shift and add operations of the columns. For this last projection requires $K(H+1)+h+1$ cycles which brings the total to $K(N+3H+3)+N+h+1$ cycles for computing the full DPRT. Furthermore, the DPRT is represented exactly by using $B+\lceil \log_2 N \rceil$ bits where B represents the number of input bits.

2.3.3 Inverse Scalable Fast Discrete Periodic Radon Transform (iSFDPRT)

The scalable architecture for the iDPRT is given in Fig 2.7, and the associated algorithm is given in Fig. 2.8. Here, we have three basic hardware blocks: (i) the input memory block (`MEM_IN`) (optional), (ii) the partial inverse DPRT computation block (`iSFDPRT_core`), and (iii) the output/accumulator memory block (`MEM_OUT`). The functionality of this system is the same as the SFDPRT (see Sec. 2.3.2) with the exception of the extra circuit that performs the accumulation and normalization of the output. Since there are many similarities between the DPRT and its inverse, the focus is on explaining the most significant differences. The list of the most significant differences include:

- **Input size:** The input is $R(m,d)$ with a size of $(N+1)\times N$ pixels.
- **No transposition and optional use of `MEM_IN`:** A comparison between (2.1) and (2.2) shows that second term of (2.1) is not needed for computing (2.2). Thus, the horizontal sums that required fast transposition are no longer

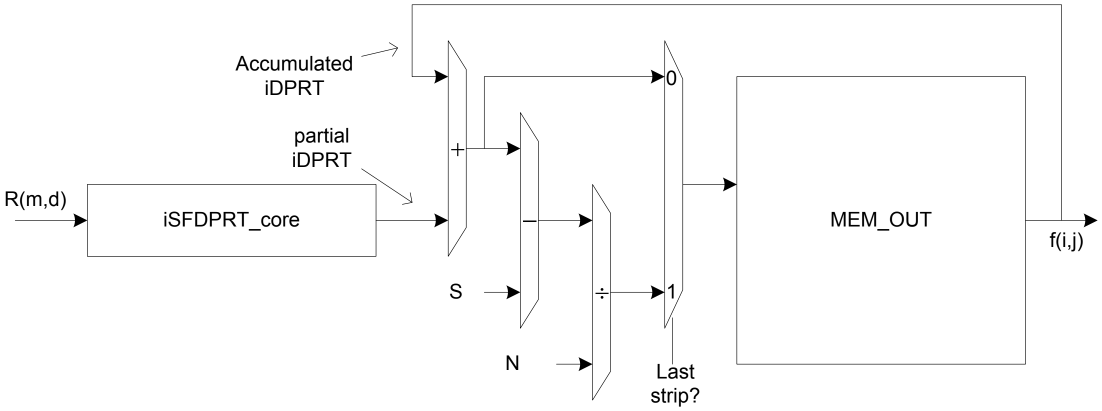


Figure 2.7: System for implementing the inverse, scalable and fast DPRT (iSFDPRT). The system uses the `iSFDPRT_core` core for computing partial sums. The system uses dual port input and output memories, an accumulator array and a Finite State Machine for control. See text in Sec. 2.3.3 for more details.

needed. As a result, `MEM_IN` is only needed to buffer/sync the incoming data. In specific implementations, `MEM_IN` may be removed provided that the data can be input to the hardware in strips as described in Fig. 2.8.

- **Circular right shifting replaces circular left shifts:** A comparison between (2.1) and (2.2) shows that the iDPRT index requires $\langle j - mi \rangle_N$ as opposed to $\langle d + mi \rangle_N$ for the DPRT. As a result, the circular left shifts (CLS) of the DPRT becomes circular right shifts (CRS) for the iDPRT.

In terms of minor differences, note the special iDPRT terms of $R_N(d)$ and S in (3.5) that are missing from the DPRT. These terms needed to added (for $R_N(d)$) and subtracted (for S) for each summation term. Refer to Fig. 2.8 for details.

An optimized implementation that uses pipelined dividers with a latency of as many clock cycles as the number of bits needed to represent the dividend is consid-

```
 1: for $r = 0$ to $K - 2$ do
 2:     Load strip $r$ into the iSFDPRT_core
 3:     if $r = 0$ then
 4:         Compute $S$
 5:     end if
 6:     for $k = 0$ to $N - 1$ do
 7:         Shift in parallel all the $H$ rows:
                $CRS_a(H \cdot r + a)$,
                $a = 0, \ldots, H - 1$
 8:         Compute in parallel $f^{'}_{r,k}(j)$
 9:         Add partial result $f_k(j) = f_k(j) + R^{'}_{r,k}(j)$
                in MEM_OUT
10:     end for
11: end for
12: Load last strip into the iSFDPRT_core
13: for $k = 0$ to $N - 1$ do
14:     Shift in parallel $\langle N \rangle_H$ rows:
                $CRS_a(H \cdot r + a)$,
                $a = 0, \ldots, \langle N \rangle_H - 1$
15:     Compute in parallel $f^{'}_{r,k}(j) + R_N(d)$
16:     Add partial result: $f_k(j) = f_k(j) + f^{'}_{r,k}(j) + R_N(d)$
17:     Subtract S: $f_k(j) = f_k(j) - S$
18:     Normalize by N: $f_k(j) = f_k(j)/N$
                and store in MEM_OUT
19: end for
```

Figure 2.8: Top level algorithm for computing the inverse Scalable Fast Discrete Periodic Radon Transform $f(i, j) = \Re^{-1}(R(m, d))$. With the exception of the strip operations of lines 2 and 12, all other operations are pipelined and executed in a single clock cycle. The strip operations require H clock cycles where H represents the number of rows in the strip. See section 2.2.1 for the notation.

ered. Then, the total running time is $K(N + H) + h + 3 + B + 2n$ as illustrated in Fig. 2.9. Resource requirements are given in section 2.6.

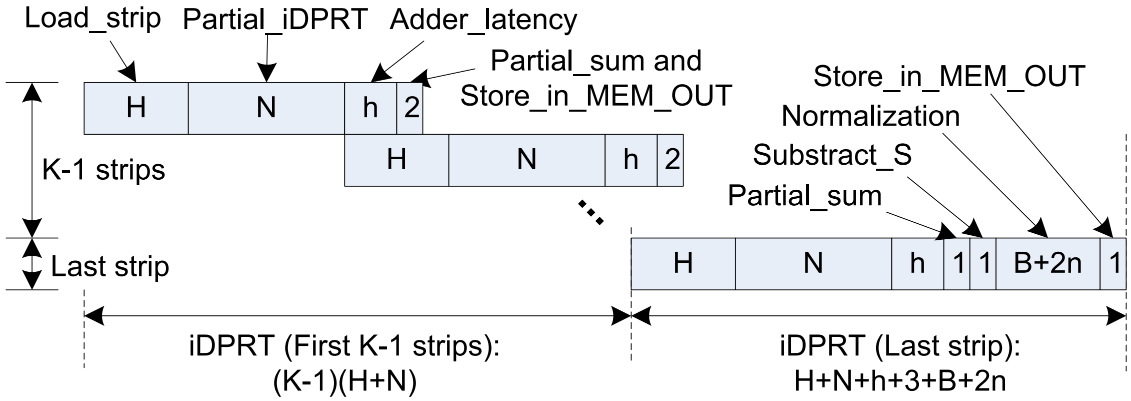


Figure 2.9: Running time for computing the inverse, scalable, fast DPRT (iSFDPRT). Here, H denotes the maximum number of projection rows for each strip, $K = \lceil N/H \rceil$ is the number of strips, $h = \lceil \log_2 H \rceil$ is the addition latency, $n = \lceil \log_2 N \rceil$, and $B + 2n$ is the number of bits used to represent the results before normalization.

2.3.4 Fast Discrete Periodic Radon Transform (FDPRT) and its inverse (iFDPRT)

When there are sufficient resources to work with the entire image, there is no need to break the image into strips. All the computations can be done in place without the need to compute partial sums that will later have to be accumulated. In this case, the use of the RAM is eliminated and simply hold the input in the register array. For this case, the terms FDPRT and iFDPRT are used to describe the optimized implementations. For the FDPRT, the register array is also modified to implement the fast transposition that is required for the last projection (transposition time=1 clock cycle) as described in [26].

The basic idea to compute the FDPRT is to use a shift register architecture to align the image samples that need to be added for each projection. Then, along each shifted direction, an effective adder-tree approach can be used to provide for an effective method for performing the additions.

To introduce the approach, consider projections along the prime directions given by $(1, p), p = 0, ..., N-1$ (note that the prime direction $(0, 1)$ is not included yet). Then, to compute the vector projection $R_p(d) = R(p, d)$, $d = 0, ..., N-1$, fixed p, begin with the first prime direction $(1, 0)$. The first projection is given by:

$$R_0(d) = \sum_{i=0}^{N-1} f(i, d).$$

The computation of $R_0(d)$ is illustrated in Fig. 2.10(a) for a 7×7 image example. Along each column, the addition can be carried out effectively using a multi-operand fully-pipelined adder tree (see [29] and Fig. 2.10(c) for more details). As described in [29], each adder tree generates an output per clock cycle with a latency of $\lceil \log(N) \rceil$ clock cycles. As shown in Fig. 2.10(b), the adder trees can work in parallel. Thus, the architecture outlined in Fig. 2.10(b,c) can compute $R_0(d)$ in one clock cycle.

While the inputs to the adder tree are aligned for the direction of $(1, 0)$, it is important to note that they will not be properly aligned for the next prime direction of $(1, 1)$. To see this, note that the projection for $(1, 1)$ is given by:

$$R_1(d) = \sum_{i=0}^{N-1} f(i, \langle d+i \rangle_N).$$

See Fig. 2.10(d) for an illustration. To re-align the adder tree inputs, circular left shift registers (CLS) are used, as illustrated in Fig. 2.10(e,f). The basic idea is to apply different shift levels to each row so as to properly align the image samples with the adder trees. The appropriate shift amount is given by: $\texttt{CLS}(i) =$

`CircularLeftShift`(i). The structure of the shift registers is clearly illustrated in Fig. 2.10(f,g). All shifts are performed in parallel in a single clock cycle.

The remaining prime directions can also be implemented using the circular shift register array of Fig. 2.10(f).

The entire algorithm for computing the FDPRT is given in Fig. 2.11. For all of the projections, except for the primal direction of $(0, 1)$, the additions can be carried out across each column. For $(0, 1)$, the additions need to be done along the rows, and there is no shift operation that would allow the additions to be carried out along the columns. Thus, a transposition is needed. In general, the transposition requires c cycles depending on the implementation. In an optimized implementation, the transposition can be performed in a single cycle. Furthermore, it can be performed in the same clock cycle with the last shift reducing c to $c = 0$. The result (last projection) is then stored in $R_N(d)$.

To achieve the fastest implementation of the algorithm described in Fig. 2.11 additional optimizations must be enforced:

- Since the N-operand adder trees are fully pipelined and those N adder trees work in parallel, in the same clock cycle the shift for the next projection can be started. This means that Step 2 and Step 4 can start in the same clock cycle; similarly, inside the loop, Step 5 (iteration p) and Step 4 (iteration $p + 1$); and Step 5 and Step 7 for the last loop iteration.

- The transpose in Step 8 can take several clock cycles, to reduce that to one clock cycle, the register array holding $f(i, j)$ can have the capability of move

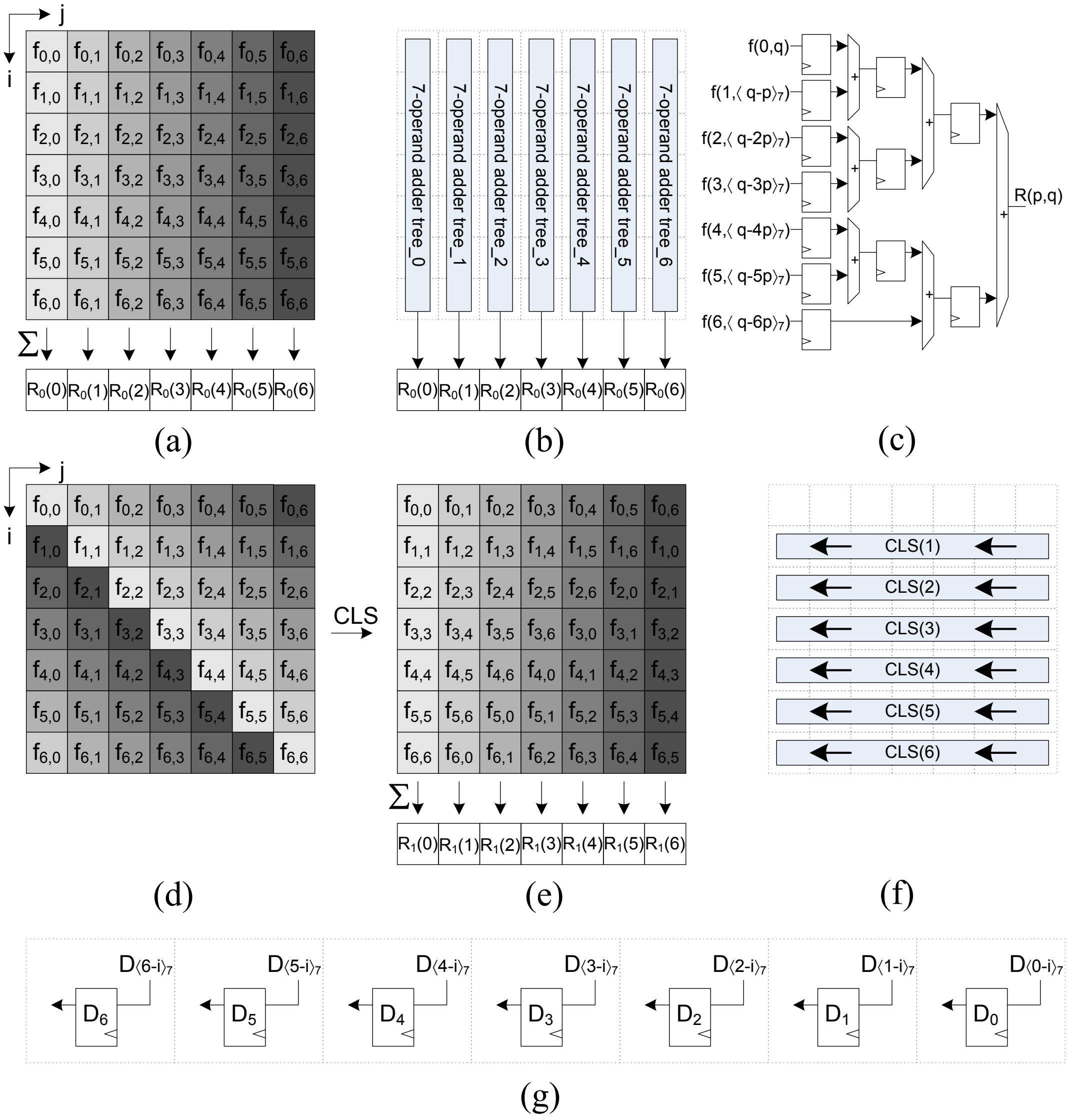

j
i
f0,0 f0,1 f0,2 f0,3 f0,4 f0,5 f0,6
f1,0 f1,1 f1,2 f1,3 f1,4 f1,5 f1,6
f2,0 f2,1 f2,2 f2,3 f2,4 f2,5 f2,6
f3,0 f3,1 f3,2 f3,3 f3,4 f3,5 f3,6
f4,0 f4,1 f4,2 f4,3 f4,4 f4,5 f4,6
f5,0 f5,1 f5,2 f5,3 f5,4 f5,5 f5,6
f6,0 f6,1 f6,2 f6,3 f6,4 f6,5 f6,6
Σ
R0(0) R0(1) R0(2) R0(3) R0(4) R0(5) R0(6)
(a)
7-operand adder tree_0
7-operand adder tree_1
7-operand adder tree_2
7-operand adder tree_3
7-operand adder tree_4
7-operand adder tree_5
7-operand adder tree_6
R0(0) R0(1) R0(2) R0(3) R0(4) R0(5) R0(6)
(b)
f(0,q)
f(1,⟨ q-p⟩7)
f(2,⟨ q-2p⟩7)
f(3,⟨ q-3p⟩7)
f(4,⟨ q-4p⟩7)
f(5,⟨ q-5p⟩7)
f(6,⟨ q-6p⟩7)
+
R(p,q)
(c)
CLS
f0,0 f0,1 f0,2 f0,3 f0,4 f0,5 f0,6
f1,1 f1,2 f1,3 f1,4 f1,5 f1,6 f1,0
f2,2 f2,3 f2,4 f2,5 f2,6 f2,0 f2,1
f3,3 f3,4 f3,5 f3,6 f3,0 f3,1 f3,2
f4,4 f4,5 f4,6 f4,0 f4,1 f4,2 f4,3
f5,5 f5,6 f5,0 f5,1 f5,2 f5,3 f5,4
f6,6 f6,0 f6,1 f6,2 f6,3 f6,4 f6,5
R1(0) R1(1) R1(2) R1(3) R1(4) R1(5) R1(6)
(d)
(e)
CLS(1)
CLS(2)
CLS(3)
CLS(4)
CLS(5)
CLS(6)
(f)
D⟨6-i⟩7 D6
D⟨5-i⟩7 D5
D⟨4-i⟩7 D4
D⟨3-i⟩7 D3
D⟨2-i⟩7 D2
D⟨1-i⟩7 D1
D⟨0-i⟩7 D0
(g)

Figure 2.10: Projection computation example for the first two prime directions for a 7×7 image. (a) Pixels are added along each column using an adder tree for prime direction $(1, 0)$. (b) Array of 7-operand adder tree for performing the additions. (c) Detailed architecture of the *7-operand adder tree_p* (fully pipelined) to compute the projection p, element q. (d) For prime direction (1,1), pixels sharing the same gray-scale value need to be added but are not aligned along the columns. (e) Pixels are properly aligned along each column following the required number of circular, left, shifts. (f) Circular Left-Shift (CLS) structure for aligning image samples for prime direction of $(1, 1)$, all shifts are performed in parallel in a single clock cycle.. (g) Detalied architecture for CLS(i).

```
1: Load image $f(i,j)$
2: Compute and Store in parallel $R_0(d)$
3: for $p = 1$ to $N-1$ do
4:     Shift in parallel the last $N-1$ rows
           $CLS(i), i = 1, ..., N-1$
5:     Compute and Store in parallel $R_p(d)$
6: end for
7: Shift in parallel the last $N-1$ rows
       $CLS(i), i = 1, ..., N-1$
8: Transpose of the image
9: Compute and Store in parallel $R_N(d)$
```

Figure 2.11: Algorithm for computing the Fast Discrete Periodic Radon Transform $R(m,d) = \Re(f(i,j))$.

all the values to the transpose location in parallel in one clock cycle. Moreover, this transposition can be started in the same clock cycle of Step 5 and Step 7 making the effective number of clock cycles for the transpose equal to zero [26].

In terms of computational complexity, the timing diagram is depicted in Fig. 2.12. Here, Note that time increases to the right. As before, the different computational steps are depicted along the columns. Cycles associated with parallel computations appear within the same column.

The details of the process and the total running time in terms of the required number of cycles is given as follows. Initially, the image is loaded row-by-row. Thus, image loading requires N cycles as depicted in the timing diagram of Fig. 2.12. Shifting is performed in a single cycle along each row. The shifted rows are then added along each column. Due to the fully pipelined architecture, it only takes $N-1$ cycles to compute the first $N-1$ projections. The last two projections only require

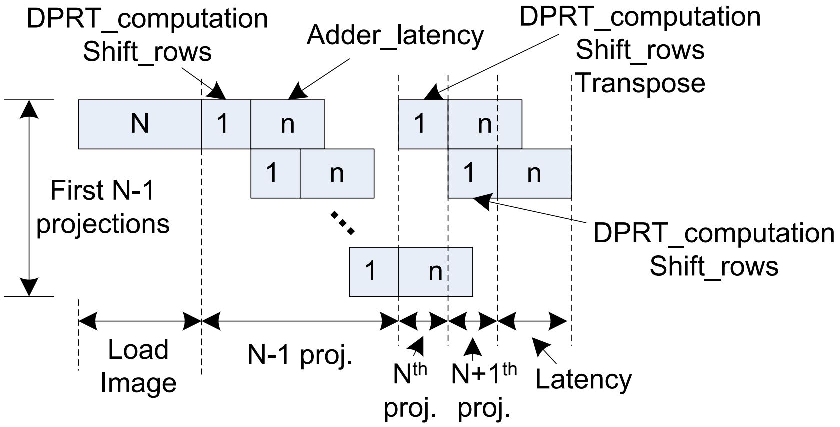


Figure 2.12: Running time for fast DPRT (FDPRT). In this diagram, time increases to the right. The DPRT is computed in $N+1$ steps (projections). Each projection takes $1+h$ clock cycles. Here, $n = \lceil \log_2 N \rceil$ represents the addition latency. Pipeline structure: Since fully pipelined adder trees are used, the computation of subsequent projections can be started after one clock of the previous projection.

two additional cycles. Note that the transposition is perfomed in the same clock cycle of the last shift. Then, the final result is is only delayed by the latency associated with the last addition ($n = \lceil \log_2 N \rceil$). Thus, overall, it only takes $2N + n + 1$ cycles to compute the FDPRT.

For the iFDPRT, let $R(m, d)$ be the DPRT of $f(i, j)$. To compute the iFDPRT of $R(m, d)$, the architecture used for the FDPRT can be used again, with the following changes:

- The number of projections needed to perform the iFDPRT is now N.
- The summation $\sum_{m=0}^{N-1} R\left(m, \langle j - mi \rangle_N\right)$ in Eq. 2.2 is done now over periodic line segments perpendicular to the ones in the FDPRT. Therefore, a Circular Right Shifting (CRS) is neeed to align the data to be added by columns. This

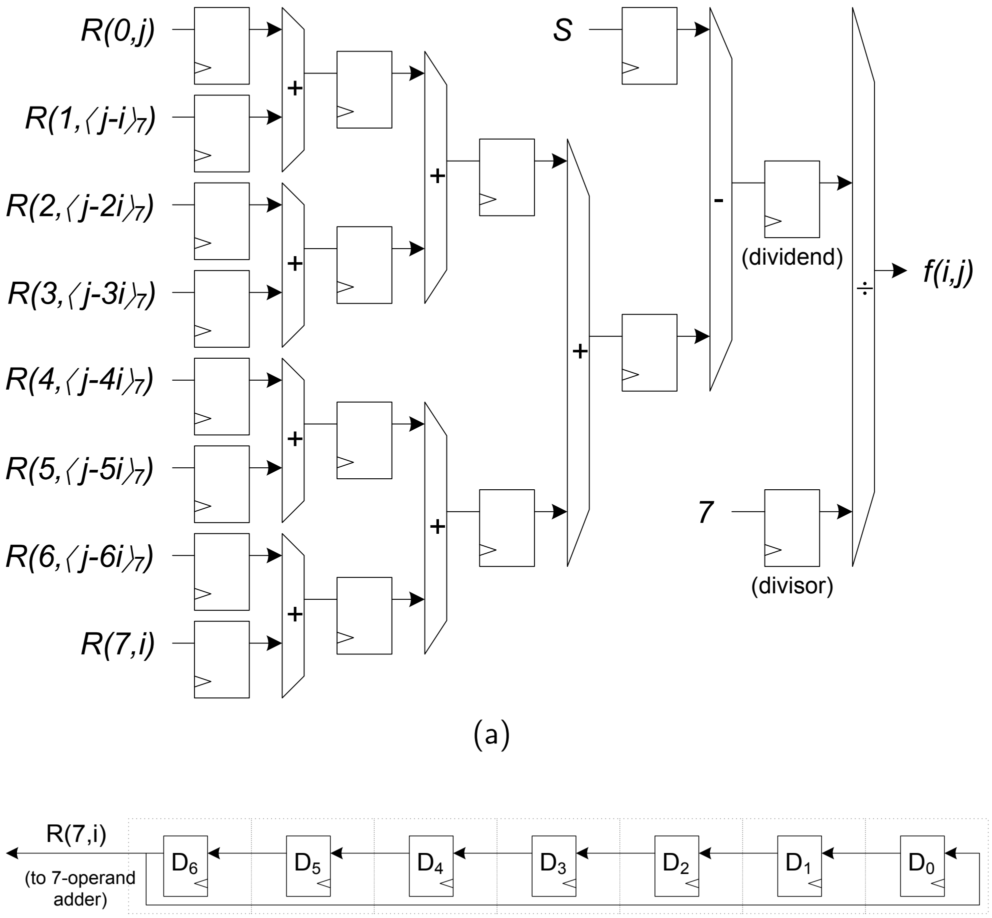
indicates that the architecture used for the FDPRT can be used for the iDPRT with the change of the CLS registers by CRS registers.

- When loading the image, the last row ($R_N(d)$) must not be included (this last row will not be shifted). Instead, this vector must be stored in an additional CLS(1) register; and when computing the projection i, the value $R(N,i)$ must be passed as an additional input for each adder tree (see Fig. 2.13(a), bottom input, and Fig. 2.13(b) for an illustration).

- The value S (Eq. 2.3) must be computed before starting the computation of the projections. The most convenient way to compute S is when $R(m,d)$ is being loaded, using and additional N-operand adder to do the summation $S = \sum_{d=0}^{N-1} R(m,d)$ with $m = 0$. S must be substracted from every output of the adder tree. See Fig. 2.13(a), for an illustration.

- The normalizing factor $1/N$ in Eq. 2.2 must be applied to every value obtained after the substraction of S. See Fig. 2.13(a), division, for an illustration.

Therefore, taking into account all the differences described above, the N-operand adder tree needes to be modified (see Fig. 2.13 for a 7×7 example), the shift registers now shift the data to the right; and the algorithm to perform the iFDPRT (shown in Fig. 2.14, based on the algorithm for the FDPRT) is as follows: In parallel, R will be loaded and S will be computed (Steps 1 and 2), after the first row $R_0(d)$ is loaded, the computation of S can start and it will take $\lceil \log_2 N \rceil$ clock cycles, on the other hand the loading takes one clock cycle per row, therefore a total of N cycles.



(a)

(b)

Figure 2.13: (a) Adder architecture example for $N = 7$. The fully pipelined 7-operand adder tree used for the FDPRT is now modified to compute the iFDPRT (i projection, element j): After the shift registers align the data, the adder tree receives all the terms to compute $\sum_{m=0}^{6} R\left(m, \langle j - mi \rangle_7\right) + R(7,i)$. Note that $R(7,i)$ is an additional term for the 7-operand adder tree (provided by an additional `CLS(1)` holding $R_N(d)$). After all the terms are added, S needs to be substracted and then divide the result by 7 to obtain $f(i,j)$. In a full implementation, 7 of those adders ($j = 0, \ldots, N-1$) are used to be able to compute in parallel one projection, to obtain one complete row of f. (b) Additional modified `CLS(1)` to hold $R_N(d)$ and provide $R(N,i)$ to all of the 7-operand adder trees on each projection i.

Once R is loaded, the computation of each projection will start (note that the first projection does not need shifting), since a fully pipelined adder tree is used, only one clock cycle is required to start the computation of one projection (with a latency of $\lceil \log_2 N \rceil + 1 + d$, where d accounts for the latency of the divider at the end of the adder tree), note note that the shifting can be performed in the same starting clock cycle.

To sum up, the architecture is basically reduced to the CRS registers plus the adder trees, where the additional CLS(1), the substraction of S and the normalizing factor ($1/N$) can be embedded inside the adder trees (see Fig. 2.13 for a 7×7 architecture example). The total running time to compute the iFDPRT is now $2N + 3n + B + 2$, using a full pipelined divider with a latency of $d = B$ clock cycles.

The required resources for both FDPRT and iFDPRT are summarized in section 2.6.

```
Load $R(m,d)$,
      $0 \le m, d \le N-1$
Compute $S$
Load $R_N(d)$
Compute in parallel $f_0(j)$
**for** $i = 1$ to $N-1$ **do**
    Shift in parallel the last $N-1$ rows
          $CRS_k(k), k = 1, \ldots, N-1$
    Compute in parallel $f_i(j)$
**end for**
```

Figure 2.14: Algorithm for computing the Inverse Fast Discrete Periodic Radon Transform $f(i,j) = \Re^{-1}(R(m,d))$. Refer to section 2.2.1 for the notation.

2.3.5 Pareto-optimal Realizations

For the development of scalable architectures, restrict the attention to implementations that are optimal in the multi-objective sense. A similar approach was also considered in [30].

Basically, the idea is to expect that architectures with more hardware resources will also provide better performance. Here, consider architectures that will give faster running times as the hardware resources are increased.

The set of implementations that are optimal in the multi-objective sense forms the Pareto front [28]. Formally, an implementation is considered to be sub-optimal if another (different) implementation can be found and can run at the same time or faster for the same or less hardware resources, excluding the case where both the running time and computational resources are equal. The Pareto front is then defined by the set of realizations that cannot be shown to be sub-optimal.

For deriving the Pareto-front, the image size is fixed to N. Then, we want find the number of rows in each image strip (values of H) that generate Pareto-optimal architectures. Now, since N is prime, it cannot be divided by H exactly. The number of strips is given by $\lceil N/H \rceil$ which denotes the ceiling function applied to N/H. To derive the Pareto-front, we require that larger values of H will result in fewer strips to process. In other words, we require that:

$$\left\lceil \frac{N}{H} \right\rceil < \left\lceil \frac{N}{H-1} \right\rceil . \tag{2.11}$$

In this case, using H rows in each strip will result in faster computations since fewer strips are being processed and also processing a larger number of rows per strip. The Pareto front is then defined using:

$$\texttt{ParetoFront} = \{H \in S \quad \text{s.t.} \quad H \quad \text{satisfies eqn} \quad (2.11)\} \tag{2.12}$$

where $S = \{2, 3, \ldots, (N-1)/2\}$ denotes the set of possible values for the number of rows. To solve (2.11) and derive the `ParetoFront` set, simply plug-in the different values of H and check that (2.11) is satisfied. Beyond the scalable approach, note that an optimal architecture for $H = N$ was covered in subsection 2.3.4. The Pareto front will be presented in the Results section.

2.4 Implementation Details

In this section, the implementations details are presented for the scalable and fast DPRTs and their inverses. A top-down description of the scalable architecture in section 2.4.1 is provided. Then, in section 2.4.2 a discussion of the changes with respect to the SFDPRT to obtain the the inverse DPRTs is presented. This is because, the architectures for the forward DPRTs are closely related to the architectures for the inverse DPRTs but simpler.

2.4.1 Scalable Fast Discrete Periodic Radon Transform (SFDPRT)

In this section, the different processes that were presented in the top-down diagram of Fig. 2.4 are described in detail. At the top level, block diagrams are presented for the the memory components (MEM_IN and MEM_OUT) in Fig. 2.15.

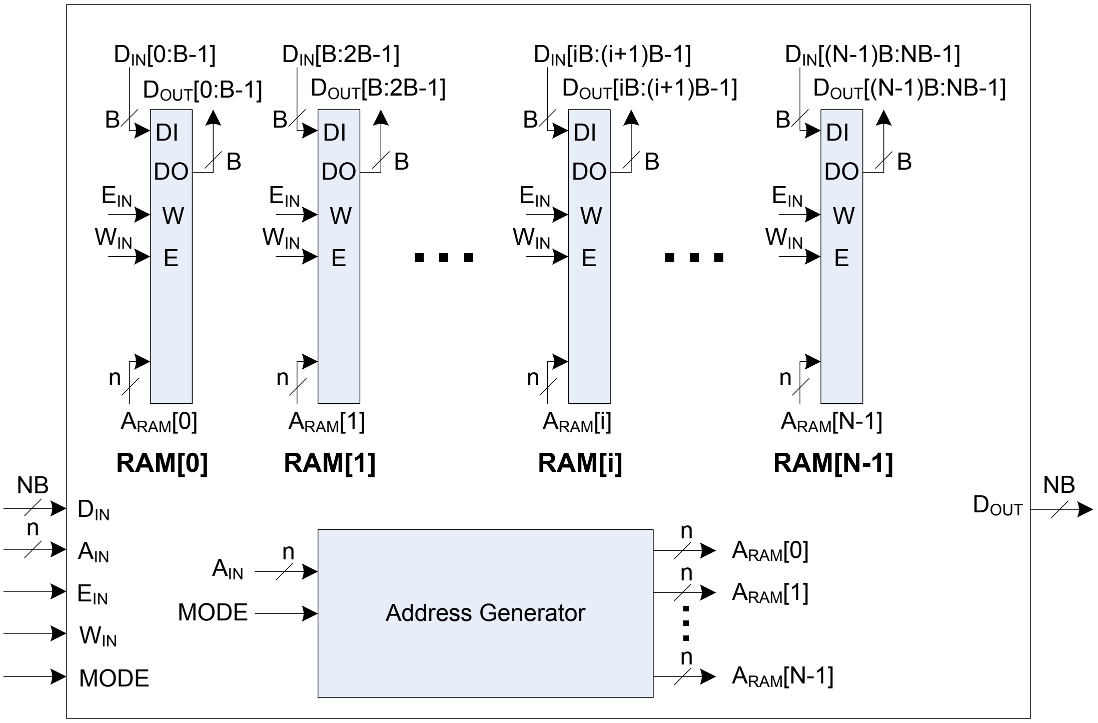


Figure 2.15: Memory architecture for parallel read/write. For the parallel load, refer to Fig. 2.16. The memory allows to avoid transposition as described in Fig. 2.17. The memory architecture refers to MEM_IN and MEM_OUT in Fig. 2.4. Each **RAM** is a standard Random Access Memory with bus address $A[0:n-1]$, separate data buses $DI[0:B-1]$ and $DO[0:B-1]$, and control signals W and E to select Read/Write cycles and enable the memory respectively. MEM_IN is a memory with bus address $A_{IN}[0:n-1]$, separate data buses $D_{IN}[0:NB-1]$ and $D_{OUT}[0:NB-1]$ with control signals W_{IN}, E_{IN} and $MODE$ to select Read/Write cycles, enable the memory and two addressing modes respectively. The $MODE$ signal selects between row or column reading, in other words, provides a complete row or column of f.

A brief description of the memory components is provided. Each RAM-block is a standard Random Access Memory with separate address, data read, and data write buses. The $MODE$ signal is used to select between row and column access. For row access, the addresses are set to the value stored in $\mathtt{A_{RAM}}[0]$. Column access is only supported for MEM_IN. The addresses for column access are determined using:

$\mathtt{A_{RAM}}\left[\mathtt{i}\right] = \left\langle \mathtt{A_{RAM}}\left[\mathtt{0}\right] + \mathtt{i}\right\rangle_{\mathtt{N}}, \mathtt{i} = \mathtt{1}, \ldots, \mathtt{N} - \mathtt{1}.$

The main process of Fig. 2.5 is summarized in four steps:

1. An $N \times N$ image is loaded row-wise in MEM_IN as shown in line 1 of Fig. 2.16.

2. Image strips are loaded into SFDPRT_core, shifted and written back to MEM_IN as described in Fig. 2.16). At the end of this step, the image is rearranged so that each diagonal corresponds to an image column. This allows to get each row of the transposed image in one cycle.

3. Image strips are loaded into the SFDPRT_core and then left-shifted once as described in Fig. 2.17. For the first N projections, accumulate the results from partial DPRTs computed for each strip as described in Fig. 2.18. To compute the accumulated sums, use an adder array. Also, for pipelined operation, MEM_OUT is implemented as a dual port memory.

4. For the last projection, to avoid transposition, access the input image in column mode. The rest of the process is the same as for the previous N projections.

The Transform is computed using exact arithmetic using $B + \lceil \log_2 N \rceil$ bits to represent the output where the input uses B-bits per pixel.

To further ilustrate this process, full details of the design of each block of the system is provided and also full explanation of each step including specific examples is described.

```
*Load* image $f$ into MEM_IN
**for** $z = 0$ to $K - 1$ **do**
    **for** $y = 0$ to $H - 1$ **do**
        ***Move*** row $(z * H + y)$ of $f$ into SFDPRT_core
          in reverse-order (flipped)
        **if** $z > 0$ **then**
            ***Move*** the top row from SFDPRT_core to
              MEM_IN in reverse-order at $((z - 1) * H + y)$
        **end if**
    **end for**
    ***Shift*** in parallel all the $H$ rows into SFDPRT_core
      registers: $CLS(z * H + a)$, $a = 0, \ldots, H - 1$.
**end for**
**for** $y = 0$ to $H - 1$ **do**
    ***Move*** the top row from SFDPRT_core to
      MEM_IN in reverse-order at $((K - 1) * H + y)$
**end for**
```

Figure 2.16: The implementation of ***Load_shifted_image(f)*** of Fig. 2.5. The process shifts the input image during the loading process in order to avoid the transposition associated with the last projection. The shifting is performed using the circular left shift registers that are available in `SFDPRT_core`.

```
**for** $z = 0$ to $H - 1$ **do**
    **if** $M ==$ '$row\_mode$' **then**
        ***Move*** MEM_IN row $(r \cdot H + z)$, mode $M$
          into SFDPRT_core.
    **else**
        ***Move*** MEM_IN row $(r \cdot H + z)$, mode $M$
          into SFDPRT_core in reverse-order (flipped).
    **end if**
**end for**
Shift in parallel all the $H$ rows:
      $CLS_a(H \cdot r + a)$, $a = 0, \ldots, H - 1$
```

Figure 2.17: Process for implementing ***Load_strip(r, M)*** of Fig. 2.5.

SFDPRT_core design

The objective of this core is to compute Eq. (2.7) at high speed. An exaustive analysis of Eq. (2.7) reveals that for the first N projections, the data to be added can

```
Read accumulated $R_k(d)$ from MEM_OUT
if $k = N$ then
    Flip $R_k^{'}(d)$
end if
Add $R_k(d) = R_k(d) + R_k^{'}(d)$
Store $R_k(d)$ in MEM_OUT
```

Figure 2.18: The implementation of ***Add_partial_result*** of Fig. 2.5. The process is pipelined where all the steps are executed in a single clock cycle.

be column aligned via CLS as follows: for the first projection the data on the strip is already column aligned, then, all the H pixels on each column must be added, for this purpose, it is used N H-operand fully pipelined adder trees (see Fig. 2.19(a)) that will add all the pixels in one clock cycle with a latency of $h = \lceil \log_2 H \rceil$ generating $R'_{r,0}(d)$. The next projection needs to CLS the data in the strip by the following amount: $CLS_a(H \times r + a)$, where r is the strip number, and $a = 0, \ldots, H-1$ is the row position inside the strip r (see Fig. 2.19(b)). Once the CLS is performed, the data on each column can be added, obtaining $R'_{r,1}(d)$. All the subsequent projections need to apply the same CLS and do the additions. An optimal architecture can perform on the same clock cycle the colum additions and the CLS. Therefore, the running time for the total computation of one partial DPRT is N clock cycles. Note that the last projection is still missing, this is because it is not possible to align the data for the last projection, to solve this issue, the transpose of f needs to be loaded, and just perform the additions as it was done for the first projection (no CLS is needed), this is possible to do without any additional hardware by using MEM_IN in column mode.

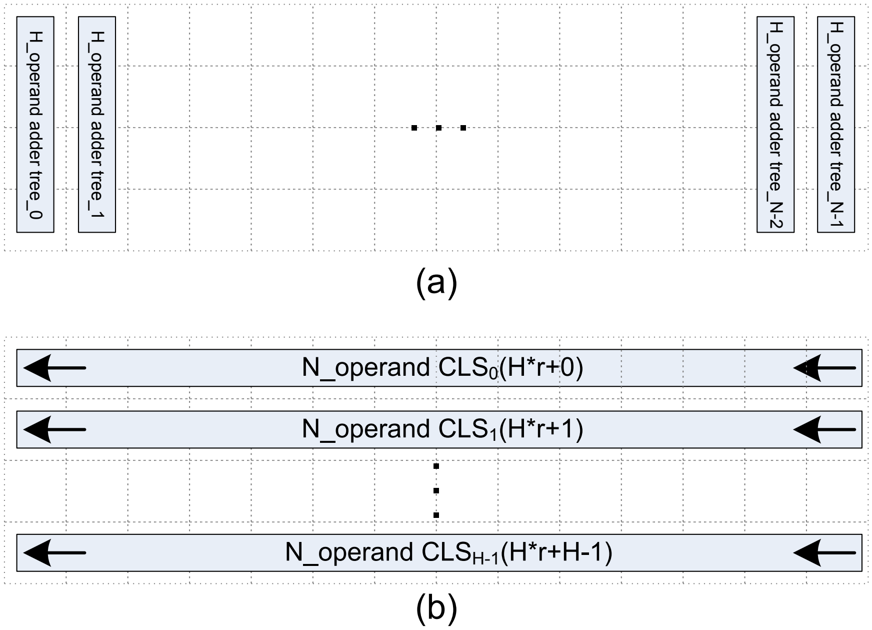
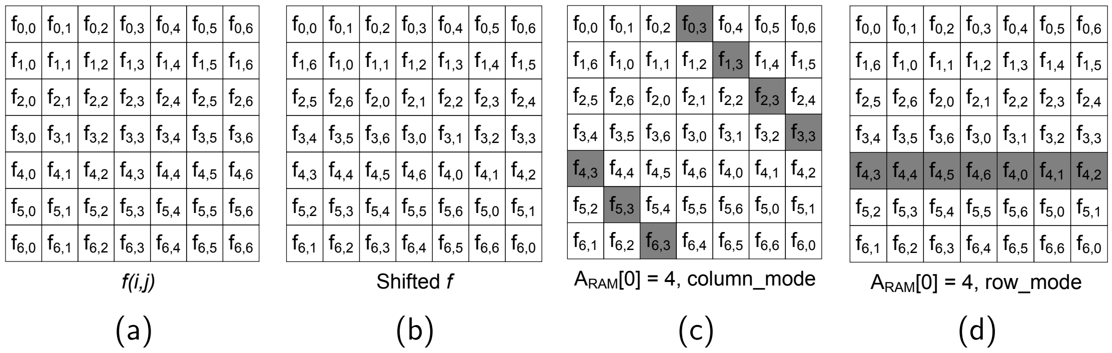


Figure 2.19: `SFDPRT_core` architecture. (a) Array of H-operand adder tree for performing the $H \times N$ additions in parallel in one clock cycle. (b) Circular Left-Shift (CLS) structure for aligning image samples, all shifts are performed in parallel in a single clock cycle..

`MEM_IN` and `MEM_OUT` design

`MEM_IN` is a memory that holds f, able to provide a complete row (or column) of f in one clock cycle. Recall $n = \lceil \log_2 N \rceil$, **RAM** be a Random Access Memory with standard address, data and control buses; and recall that each pixel of f is represented with B bits. `MEM_IN` is defined as an array of **RAM**s: $RAM\left[0 : N-1\right]$ (see Fig. 2.15 for a detailed description), where the Address Generator (via the signal $MODE$) generates the effective address for each RAM as follows:

- *row_mode*: Same address for each $RAM\left[i\right]$, i.e. if the address for $RAM\left[0\right]$ is $A_{RAM}\left[0\right]$, then the address for the rest of the RAMs is the same: $A_{RAM}\left[i\right] =$



$f_{0,0}$	$f_{0,1}$	$f_{0,2}$	$f_{0,3}$	$f_{0,4}$	$f_{0,5}$	$f_{0,6}$
$f_{1,0}$	$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	$f_{1,4}$	$f_{1,5}$	$f_{1,6}$
$f_{2,0}$	$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	$f_{2,4}$	$f_{2,5}$	$f_{2,6}$
$f_{3,0}$	$f_{3,1}$	$f_{3,2}$	$f_{3,3}$	$f_{3,4}$	$f_{3,5}$	$f_{3,6}$
$f_{4,0}$	$f_{4,1}$	$f_{4,2}$	$f_{4,3}$	$f_{4,4}$	$f_{4,5}$	$f_{4,6}$
$f_{5,0}$	$f_{5,1}$	$f_{5,2}$	$f_{5,3}$	$f_{5,4}$	$f_{5,5}$	$f_{5,6}$
$f_{6,0}$	$f_{6,1}$	$f_{6,2}$	$f_{6,3}$	$f_{6,4}$	$f_{6,5}$	$f_{6,6}$

f(i,j)

(a)

$f_{0,0}$	$f_{0,1}$	$f_{0,2}$	$f_{0,3}$	$f_{0,4}$	$f_{0,5}$	$f_{0,6}$
$f_{1,6}$	$f_{1,0}$	$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	$f_{1,4}$	$f_{1,5}$
$f_{2,5}$	$f_{2,6}$	$f_{2,0}$	$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	$f_{2,4}$
$f_{3,4}$	$f_{3,5}$	$f_{3,6}$	$f_{3,0}$	$f_{3,1}$	$f_{3,2}$	$f_{3,3}$
$f_{4,3}$	$f_{4,4}$	$f_{4,5}$	$f_{4,6}$	$f_{4,0}$	$f_{4,1}$	$f_{4,2}$
$f_{5,2}$	$f_{5,3}$	$f_{5,4}$	$f_{5,5}$	$f_{5,6}$	$f_{5,0}$	$f_{5,1}$
$f_{6,1}$	$f_{6,2}$	$f_{6,3}$	$f_{6,4}$	$f_{6,5}$	$f_{6,6}$	$f_{6,0}$

Shifted *f*

(b)

$f_{0,0}$	$f_{0,1}$	$f_{0,2}$	$f_{0,3}$	$f_{0,4}$	$f_{0,5}$	$f_{0,6}$
$f_{1,6}$	$f_{1,0}$	$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	$f_{1,4}$	$f_{1,5}$
$f_{2,5}$	$f_{2,6}$	$f_{2,0}$	$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	$f_{2,4}$
$f_{3,4}$	$f_{3,5}$	$f_{3,6}$	$f_{3,0}$	$f_{3,1}$	$f_{3,2}$	$f_{3,3}$
$f_{4,3}$	$f_{4,4}$	$f_{4,5}$	$f_{4,6}$	$f_{4,0}$	$f_{4,1}$	$f_{4,2}$
$f_{5,2}$	$f_{5,3}$	$f_{5,4}$	$f_{5,5}$	$f_{5,6}$	$f_{5,0}$	$f_{5,1}$
$f_{6,1}$	$f_{6,2}$	$f_{6,3}$	$f_{6,4}$	$f_{6,5}$	$f_{6,6}$	$f_{6,0}$

$A_{RAM}[0] = 4$, column_mode

(c)

$f_{0,0}$	$f_{0,1}$	$f_{0,2}$	$f_{0,3}$	$f_{0,4}$	$f_{0,5}$	$f_{0,6}$
$f_{1,6}$	$f_{1,0}$	$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	$f_{1,4}$	$f_{1,5}$
$f_{2,5}$	$f_{2,6}$	$f_{2,0}$	$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	$f_{2,4}$
$f_{3,4}$	$f_{3,5}$	$f_{3,6}$	$f_{3,0}$	$f_{3,1}$	$f_{3,2}$	$f_{3,3}$
$f_{4,3}$	$f_{4,4}$	$f_{4,5}$	$f_{4,6}$	$f_{4,0}$	$f_{4,1}$	$f_{4,2}$
$f_{5,2}$	$f_{5,3}$	$f_{5,4}$	$f_{5,5}$	$f_{5,6}$	$f_{5,0}$	$f_{5,1}$
$f_{6,1}$	$f_{6,2}$	$f_{6,3}$	$f_{6,4}$	$f_{6,5}$	$f_{6,6}$	$f_{6,0}$

$A_{RAM}[0] = 4$, row_mode

(d)

Figure 2.20: 7×7 pattern example for storing f in `MEM_IN`. (a) Original f. (b) Shifted f. (c) Accessing column number 4 of f, note that each value belongs to a different RAM, therefore all the values can be retrieved in one clock cycle. (d) Accessing row number 4 of f, observe that the data is shifted and needs to be un-shifted before computing the DPRT.

$A_{RAM}\left[0\right], i = 1, \ldots, N-1.$

- *column_mode*: Incremental address for each $RAM\left[i\right]$, i.e. if $RAM\left[0\right]$ has the base address $A_{RAM}\left[0\right]$, then address for the other RAMs is $A_{RAM}\left[i\right] = \left\langle A_{RAM}\left[0\right] + i\right\rangle_N, i = 1, \ldots, N-1$.

For writing data into `MEM_IN` only row mode is needed. At this point, the architecture of `MEM_IN` have been completely described, however, the mechanics of how to store and access the data inside the memory is explained in the following sub-sections.

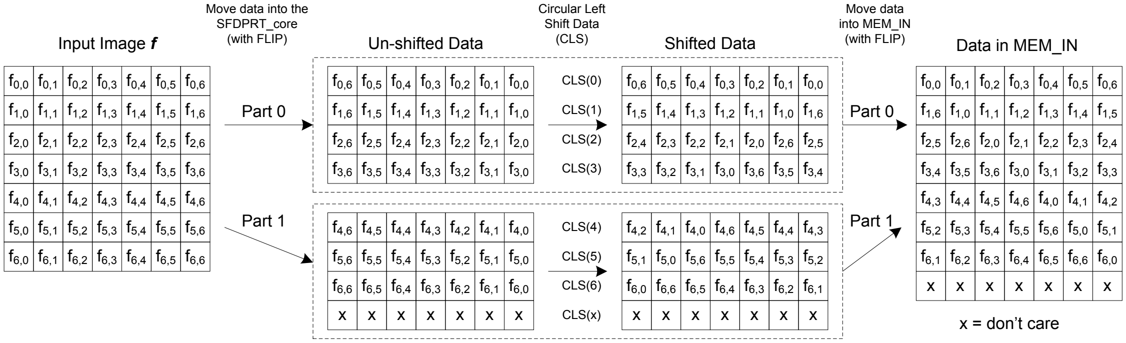
`MEM_OUT` is similar to `MEM_IN` with the only difference that only needs to read rows of f. Finally, it is suggested to use RAMs with dual ports to reduce clock cycles when reading/shifting/writing data.

Loading image to `MEM_IN`

The particular design described in section 2.4.1 has a specific purpose for the SFD-PRT: to able to read one complete row or column of f in one clock cycle, this capability allows to avoid the transposition of the pixels for the last projection when computing the DPRT. Since `MEM_IN` is an array of RAMs, reading a complete row is straightforward: Put the same address (*row_mode*) on all RAMs and retrieve the N pixels of a specific row. However, there is the need to read a complete column in one clock cycle, which is not trivial. If f is stored in `MEM_IN` in the same pattern as Fig. 2.20(a), it is not possible to read one column in one clock cycle, this is because each RAM can only access one value per clock cycle, and all the values needed belong to the same RAM. But, storing f in a shifted pattern (see Fig. 2.20(b)) and using the *column_mode* to read `MEM_IN`, it is possible to read a complete column of f in one clock cycle, because now, each value of the column needed belongs to different RAMs (gray boxes in Fig. 2.20(c)). Observe that storing f with shift, also shifts the data that is read in *row_mode* (see Fig. 2.20(d)), that means that a correction (undo shifting) when reading rows must be applied. Finally, the hardware needed to do the shifting to store or read the image is the `SFDPRT_core` itself, therefore no additional hardware is needed.

Then, for an $N \times N$ image f, $H = 2, \ldots, N$ (height of the block in `SFDPRT_core`), $K = \lceil N/H \rceil$ (number of blocks in which the image f is divided), the algorithm to Load Image f in `MEM_IN` is shown in Fig. 2.16.

In a optimized implementation (including a dual port RAM), Steps 4-7 can be



$f_{0,0}$	$f_{0,1}$	$f_{0,2}$	$f_{0,3}$	$f_{0,4}$	$f_{0,5}$	$f_{0,6}$
$f_{1,0}$	$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	$f_{1,4}$	$f_{1,5}$	$f_{1,6}$
$f_{2,0}$	$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	$f_{2,4}$	$f_{2,5}$	$f_{2,6}$
$f_{3,0}$	$f_{3,1}$	$f_{3,2}$	$f_{3,3}$	$f_{3,4}$	$f_{3,5}$	$f_{3,6}$
$f_{4,0}$	$f_{4,1}$	$f_{4,2}$	$f_{4,3}$	$f_{4,4}$	$f_{4,5}$	$f_{4,6}$
$f_{5,0}$	$f_{5,1}$	$f_{5,2}$	$f_{5,3}$	$f_{5,4}$	$f_{5,5}$	$f_{5,6}$
$f_{6,0}$	$f_{6,1}$	$f_{6,2}$	$f_{6,3}$	$f_{6,4}$	$f_{6,5}$	$f_{6,6}$

Un-shifted Data

$f_{0,6}$	$f_{0,5}$	$f_{0,4}$	$f_{0,3}$	$f_{0,2}$	$f_{0,1}$	$f_{0,0}$
$f_{1,6}$	$f_{1,5}$	$f_{1,4}$	$f_{1,3}$	$f_{1,2}$	$f_{1,1}$	$f_{1,0}$
$f_{2,6}$	$f_{2,5}$	$f_{2,4}$	$f_{2,3}$	$f_{2,2}$	$f_{2,1}$	$f_{2,0}$
$f_{3,6}$	$f_{3,5}$	$f_{3,4}$	$f_{3,3}$	$f_{3,2}$	$f_{3,1}$	$f_{3,0}$
$f_{4,6}$	$f_{4,5}$	$f_{4,4}$	$f_{4,3}$	$f_{4,2}$	$f_{4,1}$	$f_{4,0}$
$f_{5,6}$	$f_{5,5}$	$f_{5,4}$	$f_{5,3}$	$f_{5,2}$	$f_{5,1}$	$f_{5,0}$
$f_{6,6}$	$f_{6,5}$	$f_{6,4}$	$f_{6,3}$	$f_{6,2}$	$f_{6,1}$	$f_{6,0}$
x	x	x	x	x	x	x

Shifted Data

$f_{0,6}$	$f_{0,5}$	$f_{0,4}$	$f_{0,3}$	$f_{0,2}$	$f_{0,1}$	$f_{0,0}$
$f_{1,5}$	$f_{1,4}$	$f_{1,3}$	$f_{1,2}$	$f_{1,1}$	$f_{1,0}$	$f_{1,6}$
$f_{2,4}$	$f_{2,3}$	$f_{2,2}$	$f_{2,1}$	$f_{2,0}$	$f_{2,6}$	$f_{2,5}$
$f_{3,3}$	$f_{3,2}$	$f_{3,1}$	$f_{3,0}$	$f_{3,6}$	$f_{3,5}$	$f_{3,4}$
$f_{4,2}$	$f_{4,1}$	$f_{4,0}$	$f_{4,6}$	$f_{4,5}$	$f_{4,4}$	$f_{4,3}$
$f_{5,1}$	$f_{5,0}$	$f_{5,6}$	$f_{5,5}$	$f_{5,4}$	$f_{5,3}$	$f_{5,2}$
$f_{6,0}$	$f_{6,6}$	$f_{6,5}$	$f_{6,4}$	$f_{6,3}$	$f_{6,2}$	$f_{6,1}$
x	x	x	x	x	x	x

Data in MEM_IN

$f_{0,0}$	$f_{0,1}$	$f_{0,2}$	$f_{0,3}$	$f_{0,4}$	$f_{0,5}$	$f_{0,6}$
$f_{1,6}$	$f_{1,0}$	$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	$f_{1,4}$	$f_{1,5}$
$f_{2,5}$	$f_{2,6}$	$f_{2,0}$	$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	$f_{2,4}$
$f_{3,4}$	$f_{3,5}$	$f_{3,6}$	$f_{3,0}$	$f_{3,1}$	$f_{3,2}$	$f_{3,3}$
$f_{4,3}$	$f_{4,4}$	$f_{4,5}$	$f_{4,6}$	$f_{4,0}$	$f_{4,1}$	$f_{4,2}$
$f_{5,2}$	$f_{5,3}$	$f_{5,4}$	$f_{5,5}$	$f_{5,6}$	$f_{5,0}$	$f_{5,1}$
$f_{6,1}$	$f_{6,2}$	$f_{6,3}$	$f_{6,4}$	$f_{6,5}$	$f_{6,6}$	$f_{6,0}$
x	x	x	x	x	x	x

Figure 2.21: 7×7 example of Loading image f into `MEM_IN` using the algorithm described in Fig. 2.16 with $H = 4$, $N = 7$. Then, $K = \lceil N/H \rceil = 2$, and the loading of f into `MEM_IN` is divided in two parts.

done all in parallel in one clock cycle, similarly Step 12. Therefore, the total running time is $K(H + 1) + H$. An example of filling f into `MEM_IN` for a 7×7 image and $H = 4$ is given in Fig. 2.21.

Loading one strip to `SFDPRT_core`

According to the algorithm in Fig 2.5, a total of 2K strips are loaded into the `SFDPRT_core`, the first K strips are loaded in *row_mode*, and the rest in *column_mode*. To correctly load the strip into the `SFDPRT_core`, the data read from `MEM_IN` must be shifted and/or flipped before starting the partial computation of the SFDPRT.

Formally, when invoking the *Load_strip*(r, M) procedure (r = strip number, M = mode), the effective base address to read the strip is $A = r \times H$. Then, accessing `MEM_IN`, at address A_R (*row_mode*), it returns the row vector $f_{A_R}(j)$ Circular Right Shifted A_R positions ($CRS(A_R)$), then to revert the shifting a $CLS(A_R)$ must be performed and $f_{A_R}(j)$ with no shift is recovered. On the other side, accessing `MEM_IN` at address A_C (*column_mode*), it returns the column vector $f_{A_C}(i)$ Circular Left

Shifted A_C positions $(CLS(A_C))$, then to revert the shifting a $CRS(A_C)$ must be performed and $f_{A_C}(i))$ with no shift is recovered. However, the SFDPRT_core can do only CLS operations, therefore, the data, when loaded/extracted to/from the SFDPRT_core, must be flipped, this flipping converts the CRS into a CLS. The algorithm to perform the load of a strip to the SFDPRT_core is shown in Fig. 2.17. In a optimized implementation, Steps 2-6 can be executed in parallel in one clock cycle, and Step 8 takes one clock cycle. Therefore, the total running time to load one strip into the SFDPRT_core takes $H + 1$ clock cycles.

Recall that when extracting data from SFDPRT_core (inserted in *column_mode*), it must be flipped. A final note, when the data inside the SFDPRT_core is aligned to compute the last projection (data loaded in 'column_mode'), the elements transferred to the adders are shifted, however, this can be ignored since the addition is commutative/associative. A complete example to load all the strips for a 7×7 image with $H = 4$ is shown in Fig. 2.22 for *Load_strip*(k,'*row_mode*'), and in Fig. 2.23 for *Load_strip*(k,'*column_mode*').

Adding partial results to compute R(m,d)

Once the SFDPRT_core computes the partial result $R'_k(d)$, this value needs to be added with the accumulated sum of the previous partial results stored in MEM_OUT and then stored back in MEM_OUT. Therefore, the steps to perform the accumulative addition are described in the algorithm shown in Fig. 2.18.

Note that MEM_OUT must be initialized to zero before starting the computation

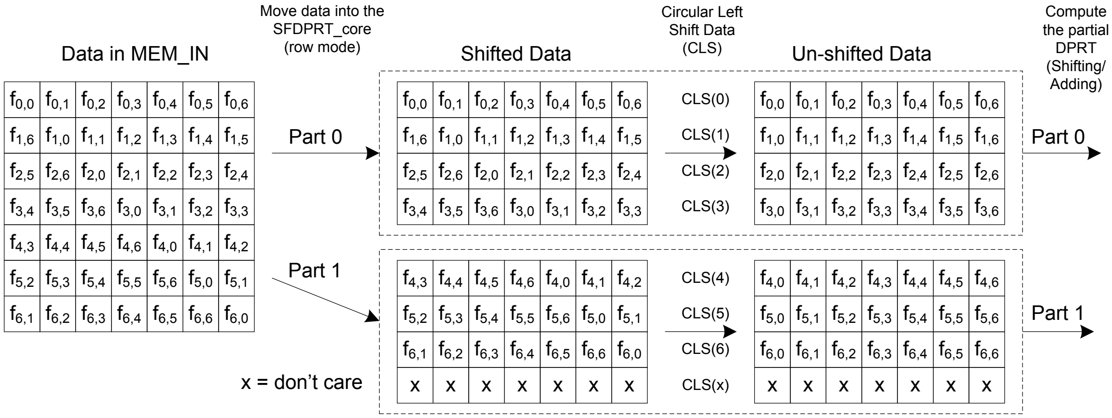


Figure 2.22: 7 × 7 example of Loading strips of f, into `SFDPRT_core` using the algorithm described in Fig. 2.17 with $H = 4$, $N = 7$, *row_mode*. Then, $K = \lceil N/H \rceil = 2$, and the loading of f into `SFDPRT_core` is divided in two parts.

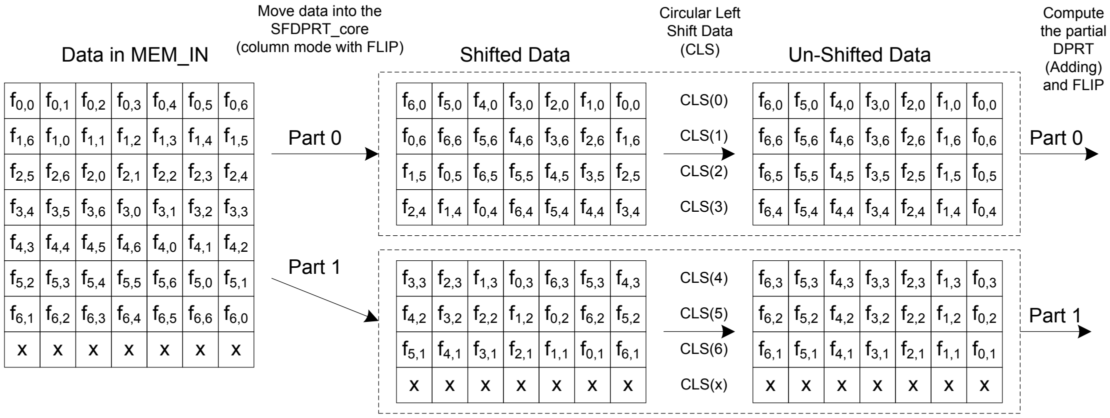

Data in MEM_IN
Move data into the SFDPRT_core (column mode with FLIP)
Shifted Data
Circular Left Shift Data (CLS)
Un-Shifted Data
Compute the partial DPRT (Adding) and FLIP
Part 0
Part 1
CLS(0)
CLS(1)
CLS(2)
CLS(3)
CLS(4)
CLS(5)
CLS(6)
CLS(x)

Figure 2.23: 7 × 7 example of Loading strips of f, into `SFDPRT_core` using the algorithm described in Fig. 2.17 with $H = 4$, $N = 7$, *column_mode* and $K = \lceil N/H \rceil = 2$. Note that the loading of f into `SFDPRT_core` is divided in two parts.

of the SFDPRT, k is the address to access `MEM_OUT`; and observe that in the last projection ($k = N$) the output of the `SFDPRT_core` must be flipped (See previous sub-section, *column_mode* for more details). In a optimized implementation, all the steps can be done in parallel, therefore the total running time is 1 clock cycle.

2.4.2 Inverse Scalable Fast Discrete Periodic Transform Implementations

In this section the necessary changes on the SFDPRT architecture and algorithm to obtain the iSFDPRT is provided.

- As described in Sec. 2.3.3, at the top-level implementation, `MEM_IN` becomes optional because the horizontal prime direction is not needed. In case `MEM_IN` is used, only needs to have the *row_mode*.

- The procedure *Load_shifted_image* is not needed, the image just need to be loaded into `MEM_IN` (if used), if not, the image strips should be directly loaded into `iSFDPRT_core`. This implies that the procedure *Load_strip_r* simply moves the data from `MEM_IN` (or received externally if `MEM_IN` is not used), no shifting or flipping is needed at all.

- An extra hardware is needed to compute the term S, the simple way to do it is to grab the first row of the input image and add all the values. This suggests the use of an extra N-operand adder tree.

- The accumulation of the partial DPRTs follows the same procedure as the SFDPRT. But, at the end an extra step is required. The final output (per pixel) needs to substract S, add $R(N, i)$ (i is the row in `MEM_OUT`) and normalize by N. An example of this procedure is depicted in Fig. 2.13, just note that for the scalable case, the operation is performed only with the last strip.

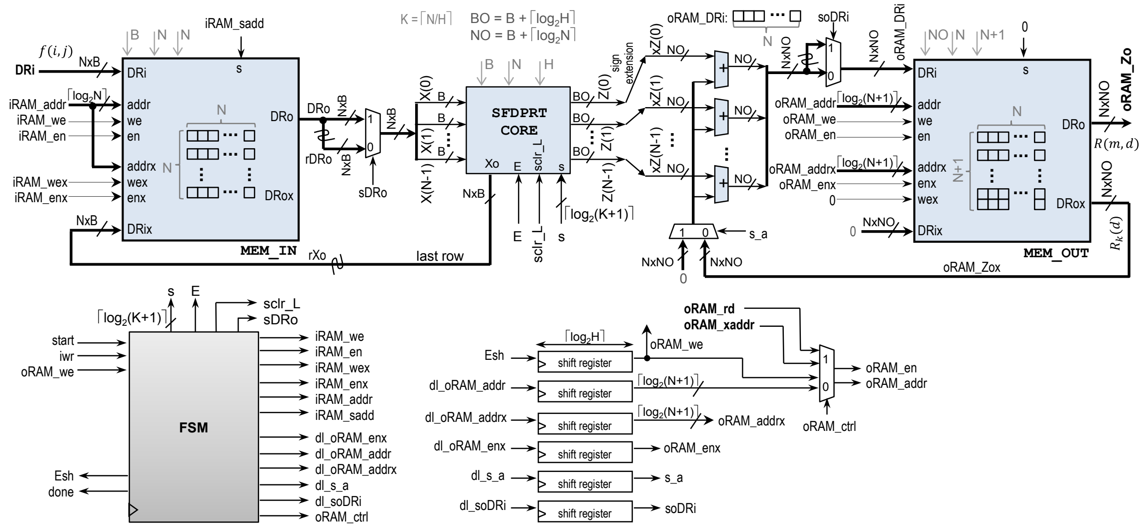


Figure 2.24: System for implementing the Scalable and Fast DPRT (SFDPRT). The `SFDPRT_core` computes the partial sums. `MEM_IN` and `MEM_OUT` are dual port input and output memories. A Finite State Machine (`FSM`) is used for control. See text in Sec. 2.3.2 for more details.

- Bit-width for perfect reconstruction. Assuming the number of bits of each input coefficient as $B' = B + \lceil \log_2 N \rceil$ before normalization the number of bits grows up to $B + 2\lceil \log_2 N \rceil$, and after normalization drops to B. Where B representes the original number of bits of the image before applying the DPRT.

2.5 FPGA Implementation

In this section, an FPGA implementation is presented to show the applicability of the proposed system. The implementation was done by Daniel Llamocca [3].

The top-level view of the hardware implementation for the scalable DPRT in shown in Fig. 2.24 which pairs with the algorithm in Fig. 2.5 and the description in 2.3.2.

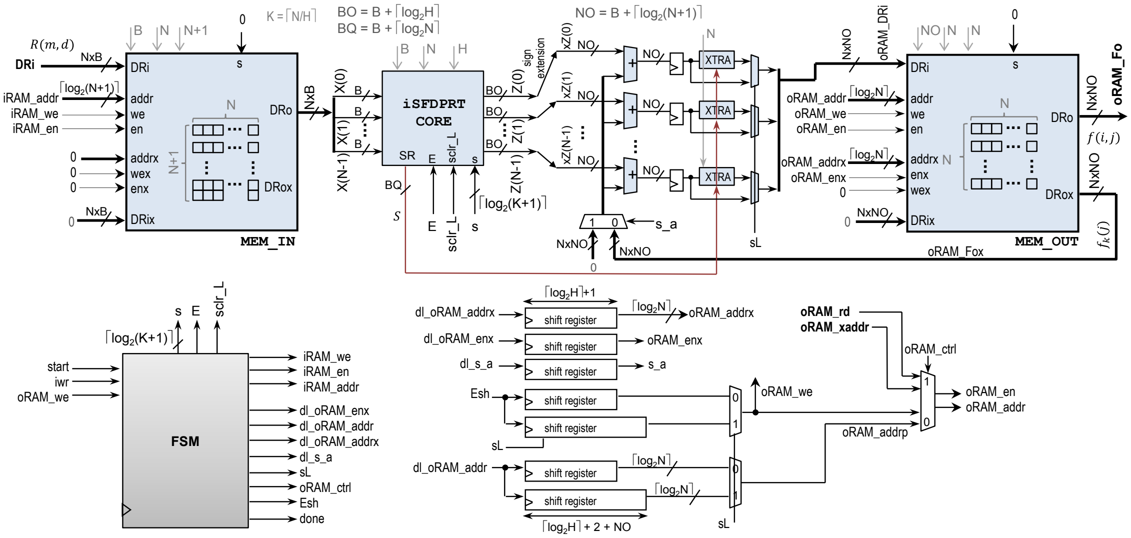


Figure 2.25: System for implementing the inverse, scalable and fast DPRT (iSFDPRT). The system uses the `iSFDPRT_core` core for computing partial sums. The system uses dual port input and output memories, an accumulator array and a Finite State Machine for control. See text in Sec. 2.3.3 for more details.

Similarly, the top-level view of the hardware implementation for the scalable iDPRT in shown in Fig. 2.25 which pairs with the algorithm in Fig. 2.8 and the description in 2.3.3.

An example of the FDPRT hardware implementation on an FPGA for an 7 × 7 image size is presented in Fig.2.26. that pairs with the description on 2.3.4

For the Inverse Fast Discrete Periodic Radon Transform (iFDPRT), the core of a FPGA implementation for a 7 × 7 image is shown as an example in Fig. 2.27 which pairs with the algorithm in Fig. 2.14 and the description in 2.3.4.

A brief overview of the different components is provided. We use 2-input MUXes to support loading and shifting as separate functions. The vertical adder trees generate the $Z(i)$ signals. A new row of $Z(0), Z(1), \ldots, Z(N-1)$ is generated for every

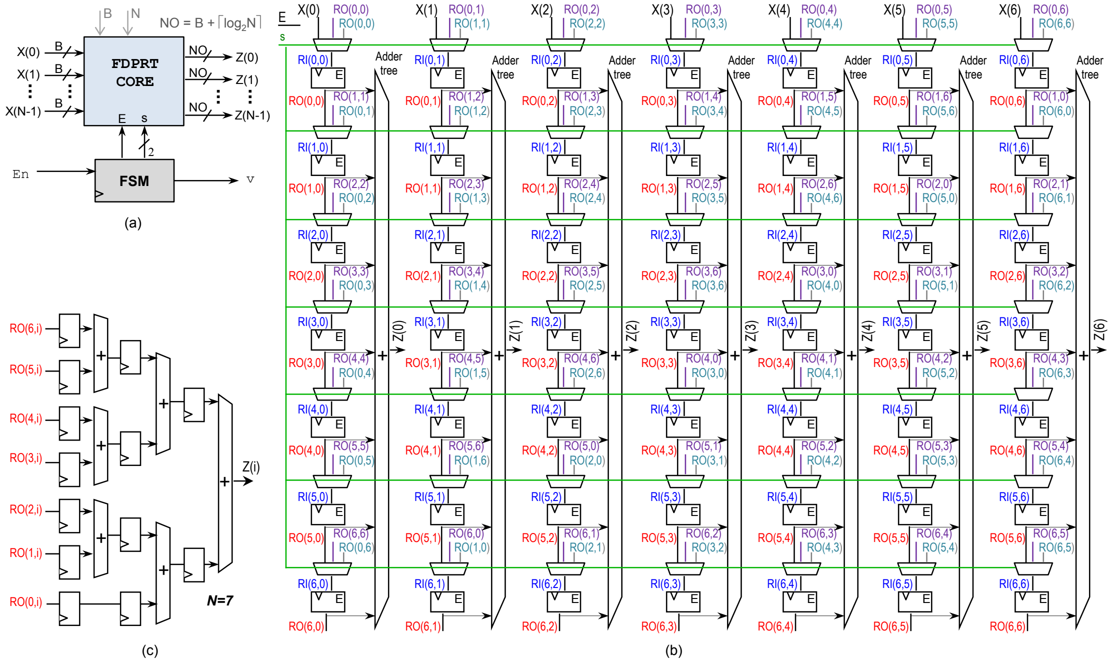

B
N
NO = B + ⌈log2N⌉
X(0)
X(1)
X(N-1)
B
FDPRT CORE
NO
Z(0)
Z(1)
Z(N-1)
E
s
2
En
FSM
v
(a)
RO(6,i)
RO(5,i)
RO(4,i)
RO(3,i)
RO(2,i)
RO(1,i)
RO(0,i)
+
Z(i)
N=7
(c)
E
s
X(0) RO(0,0) RO(0,0)
X(1) RO(0,1) RO(1,1)
X(2) RO(0,2) RO(2,2)
X(3) RO(0,3) RO(3,3)
X(4) RO(0,4) RO(4,4)
X(5) RO(0,5) RO(5,5)
X(6) RO(0,6) RO(6,6)
Adder tree
RI(0,0) RO(0,0) RO(1,1) RO(0,1)
RI(0,1) RO(0,1) RO(1,2) RO(1,2)
RI(0,2) RO(0,2) RO(1,3) RO(2,3)
RI(0,3) RO(0,3) RO(1,4) RO(3,4)
RI(0,4) RO(0,4) RO(1,5) RO(4,5)
RI(0,5) RO(0,5) RO(1,6) RO(5,6)
RI(0,6) RO(0,6) RO(1,0) RO(6,0)
RI(1,0) RO(1,0) RO(2,2) RO(0,2)
RI(1,1) RO(1,1) RO(2,3) RO(1,3)
RI(1,2) RO(1,2) RO(2,4) RO(2,4)
RI(1,3) RO(1,3) RO(2,5) RO(3,5)
RI(1,4) RO(1,4) RO(2,6) RO(4,6)
RI(1,5) RO(1,5) RO(2,0) RO(5,0)
RI(1,6) RO(1,6) RO(2,1) RO(6,1)
RI(2,0) RO(2,0) RO(3,3) RO(0,3)
RI(2,1) RO(2,1) RO(3,4) RO(1,4)
RI(2,2) RO(2,2) RO(3,5) RO(2,5)
RI(2,3) RO(2,3) RO(3,6) RO(3,6)
RI(2,4) RO(2,4) RO(3,0) RO(4,0)
RI(2,5) RO(2,5) RO(3,1) RO(5,1)
RI(2,6) RO(2,6) RO(3,2) RO(6,2)
RI(3,0) RO(3,0) RO(4,4) RO(0,4)
RI(3,1) RO(3,1) RO(4,5) RO(1,5)
RI(3,2) RO(3,2) RO(4,6) RO(2,6)
RI(3,3) RO(3,3) RO(4,0) RO(3,0)
RI(3,4) RO(3,4) RO(4,1) RO(4,1)
RI(3,5) RO(3,5) RO(4,2) RO(5,2)
RI(3,6) RO(3,6) RO(4,3) RO(6,3)
RI(4,0) RO(4,0) RO(5,5) RO(0,5)
RI(4,1) RO(4,1) RO(5,6) RO(1,6)
RI(4,2) RO(4,2) RO(5,0) RO(2,0)
RI(4,3) RO(4,3) RO(5,1) RO(3,1)
RI(4,4) RO(4,4) RO(5,2) RO(4,2)
RI(4,5) RO(4,5) RO(5,3) RO(5,3)
RI(4,6) RO(4,6) RO(5,4) RO(6,4)
RI(5,0) RO(5,0) RO(6,6) RO(0,6)
RI(5,1) RO(5,1) RO(6,0) RO(1,0)
RI(5,2) RO(5,2) RO(6,1) RO(2,1)
RI(5,3) RO(5,3) RO(6,2) RO(3,2)
RI(5,4) RO(5,4) RO(6,3) RO(4,3)
RI(5,5) RO(5,5) RO(6,4) RO(5,4)
RI(5,6) RO(5,6) RO(6,5) RO(6,5)
RI(6,0) RO(6,0)
RI(6,1) RO(6,1)
RI(6,2) RO(6,2)
RI(6,3) RO(6,3)
RI(6,4) RO(6,4)
RI(6,5) RO(6,5)
RI(6,6) RO(6,6)
Z(0)
Z(1)
Z(2)
Z(3)
Z(4)
Z(5)
Z(6)
(b)

Figure 2.26: Fast DPRT (FDPRT) hardware. (a) FDPRT core and finite state machine (FSM). (b) Structure of the FDPRT core including: pipelined adder trees, registers, multiplexers (for shifting and fast transposition) for $N = 7$. (c) Pipelined adder tree architecture for $N = 7$.

cycle. The horizontal adder tree computes SR. We recall that the SR computation is the same for all rows as shown in (2.3). The latency of the horizontal adder tree is $\lceil \log_2 N \rceil$ cycles. Note that SR is ready when $Z(i)$ is ready, as the latency of the vertical adder trees is $\lceil \log_2 (N+1) \rceil$. The SR value is fed to the 'extra units', where all $Z(i)$'s subtract SR and then divide by N (pipelined array divider [31] with a latency of BO cycles). The term $R(N, j)$ is included by loading the last input row on the last register row, where the shift is one to the left. Note that it is always the same element (the left-most one) that goes to all vertical adders.

A summary of bitwidth requirements for *perfect reconstruction* is provided. We begin by assuming that the Radon transform coefficients use B'-bits. The number of

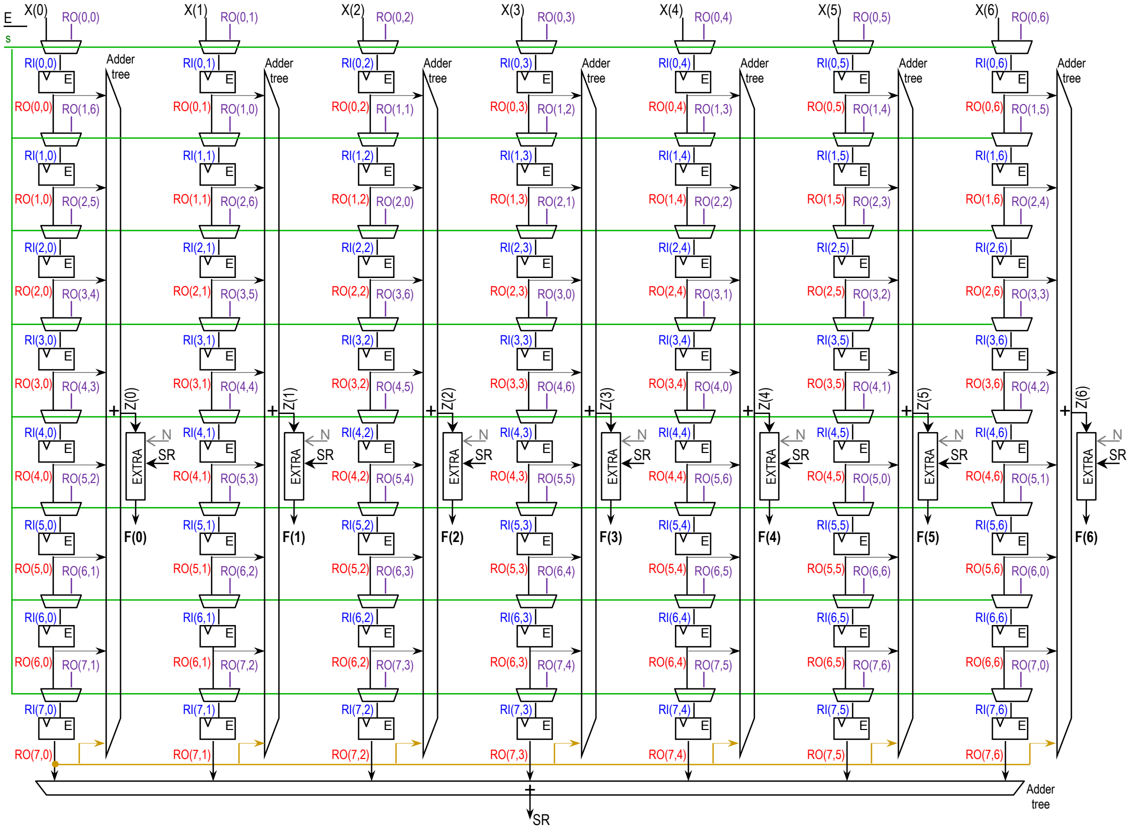


Figure 2.27: The fast inverse DPRT (iFDPRT) hardware implementation. The iFDPRT core shows the adder trees, register array, and 2-input MUXes. Here, we note that the $Z(i)$ correspond to the summation term in (2.2) (also see Fig. 2.14). We note that the 'extra circuit' is not needed for the forward DPRT. Also, for latency calculations, we note that the 'extra circuit' has a latency of $1 + BO$ cycles.

bits of the vertical adder tree outputs $Z(i)$ are then set to $BO = B' + \lceil \log_2(N+1) \rceil$. The number of bits of SR need to be $BQ = B' + \lceil \log_2 N \rceil$. Assuming that the input image f is B bits, we only need B bits to reconstruct it and the relationship between B' and B needs to be: $B' = B + \lceil \log_2 N \rceil$ bits. For the subtractor, note that $Z(i) = \sum_{m=0}^{N-1} R\left(m, \langle j - mi \rangle_N\right) + R\left(N, i\right)$ and then $Z(i) \geq SR$ since $f(i,j) \geq 0$. Thus, the result of $Z(i) - SR$ will always be positive requiring BO bits. Thus, for perfect reconstruction, the result $F(i)$ needs to be represented using BO bits.

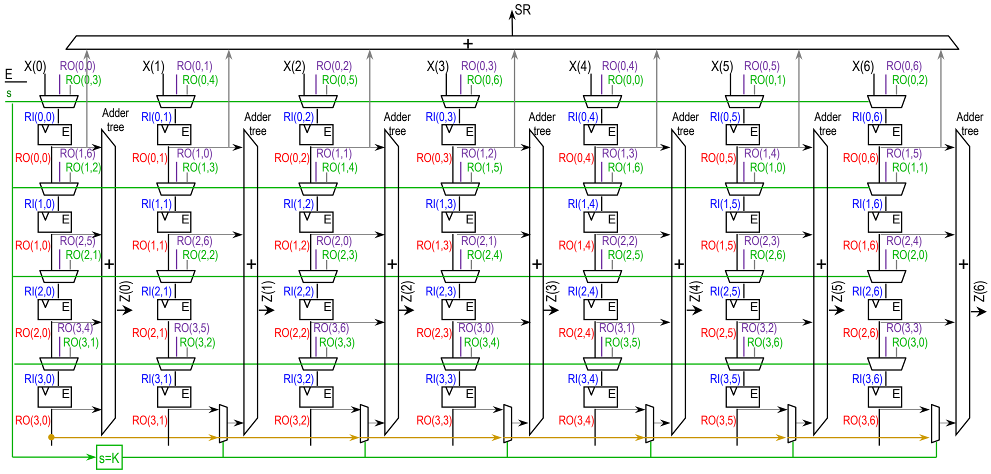


Figure 2.28: The inverse scalable DPRT `iSFDPRT_core` architecture for $N = 7$, $H = 4$. Here, we note that the $Z(i)$ correspond to the summation term in (2.10) (also see Fig. 2.8).

Next, a summary of the core implementation for the Inverse Scalable Fast Discrete Periodic Transform core (`iSFDPRT_core`) is presented in Fig. 2.28. It shows an instance of the `iSFDPRT_core` for $N = 7$ and $H = 4$. Note that the core only generates the partial sums $Z(i)$. We still need to accumulate the partial sums, subtract SR from it and divide by N.

A summary of the required hardware is provided. For each strip, we need to be able to implement different amounts of right shifting. This is implemented using $(K+1)$-input MUXes. Since N is always prime, we will have at-least one row of the register array that will be unused during computations for the last strip. The unused row is used to load the term $R(N, j)$. The vertical MUXes located on the last valid row of the last strip ensure that the term $R(N, j)$ is considered only when the last strip is being processed. Here, for the last row of the last strip, we require the shift to be one to the left. Also, the remaining unused rows are fed with zeros.

Recall that we presented the entire system in section 2.3.3. Beyond the `iSFDPRT_core`, we have the input and output memories, an array of adders and divisors, and ancillary logic. Here, we do not need to use the diagonal mode of the memories, flip the data, or rearrange the input memory. The basic process consists of loading each strip, processing it on the `iSFDPRT_core`, accumulating it to the previous result, and storing it in the output memory. For the last strip, we accumulate the result, but we also need to subtract SR and divide by N.

2.6 Results and discussion

2.6.1 Results

Comprehensive results for both the scalable and the fast DPRTs and their inverses are provided. Also, a comparison between the proposed approaches versus previously published methods is presented.

First, the results as a function of image size are presented. Running times are summarized (in terms of the number of cycles) for the forward and inverse DPRT in Tables 2.1 and 2.2 respectively. For the forward DPRT, they are compared against hardware implementations given by the serial implementation in [1] and the systolic implementation in [2]. For the inverse DPRT, the computation times are similar. However, there are no exact values to compare against. Also, comparative running times for $2 < N < 256$ for $B = 8$ bits per pixel in Fig. 2.29 are presented.

A summary of the computational resources are provided in Table 2.3. Also de-

Table 2.1: Total number of clock cycles for computing the DPRT. In all cases, the image is of size $N \times N$, and $H = 2, \ldots, N$ is the scaling factor for the SFDPRT.

Method	**Clock cycles**
Serial [16],[1]	$N^3 + 2N^2 + N$
Systolic [16],[2]	$N^2 + N + 1$
Proposed Approaches:	
- SFDPRT	$\lceil N/H \rceil (N + 3H + 3) + N + \lceil \log_2 H \rceil + 1$
- SFDPRT ($H = 2$) lowest resource use	$\lceil N/2 \rceil (N + 9) + N + 2$
- SFDPRT ($H = N$) fastest running time	$5N + \lceil \log_2 N \rceil + 4$
- FDPRT	$2N + \lceil \log_2 N \rceil + 1$

Table 2.2: Total number of clock cycles for computing the iDPRT. Here, the image size is $N \times N$. B bits per pixel are used, and $H = 2, \ldots, N$ is the scaling factor of the iSFDPRT. Add N clock cycles in the scalable version if `MEM_IN` is used.

Proposed Approaches	Clock cycles
iSFDPRT	$\lceil N/H \rceil (N + H) + 2 \lceil \log_2 N \rceil$ $+ \lceil \log_2 H \rceil + B + 3$
iSFDPRT ($H = 2$) lowest resource usage	$\lceil N/2 \rceil (N + 2) + 2 \lceil \log_2 N \rceil + B + 4$
iSFDPRT ($H = N$) fastest running time	$2N + 3 \lceil \log_2 N \rceil + B + 3$
iFDPRT	$2N + 3 \lceil \log_2 N \rceil + B + 2$

tailed resource functions usage $\mathtt{A_{ff}}$, $\mathtt{A_{FA}}$, and $\mathtt{A_{mux}}$ are provided for the 8-bit 251×251 images in Fig. 2.30. In Fig. 2.30, resources as a function of the number of rows (H)

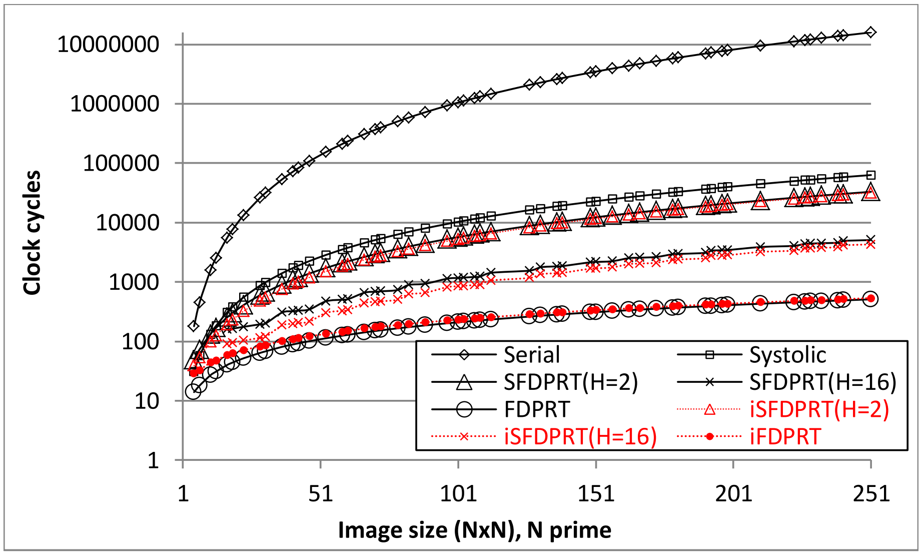


Figure 2.29: Comparative running times for the proposed approach versus competitive methods. Running times in clock cycles for: (i) the serial implementation of [1], (ii) the systolic [2], and (iii) the FPGA implementation of the SFDPRT for $H = 2$ and 16 are presented. The measured running times are in agreement with Tables 2.1 and 2.2.

stored in each image strip are shown. Also, for $N = 251$ and $B = 8$, the required number of RAM resources and the total number of MUXes in Table 2.4 are shown. For comparing performance as a function of resources, the required number of cycles are presented as a function of flip-flops in Fig. 2.31, and as a function of 1-bit additions in Fig. 2.32.

A summary of the results for the inverse DPRT is also presented. For the fast version (iFDPRT) running time and resources, refer back to Fig. 2.29 and Table 2.3. For the number of input bits, recall that $B' = B + \lceil \log_2 N \rceil$. Thus, overall, the iFDPRT implementations require more resources and slightly more computational times. Similar comments apply for the scalable, inverse DPRTs (iSFDPRT) shown in Fig. 2.29 and Table 2.3.

Table 2.3: Resource usage for different DPRT and inverse DPRT implementations. Here, consider an image size of $N \times N$, B bits per pixel, $n = \lceil \log_2 N \rceil$, $h = \lceil \log_2 H \rceil$, $K = \lceil N/H \rceil$, and $H = 2, \ldots, N$. For the adder trees, define $\mathtt{A_{ff}}$ to be number of required flip-flops, and $\mathtt{A_{FA}}$ to be the number of 1-bit additions. For the register array, define $\mathtt{A_{mux}}$ to be the number of 2-to-1 MUXes. $\mathtt{A_{ff}}$, $\mathtt{A_{FA}}$, and $\mathtt{A_{mux}}$ grow linearly with respect to N and can be computed using the algorithm given in the appendix (Fig. B.1). For the inverse DPRT, note that each divider is implemented using $3(B+2n)^2$ flip-flops, $(B+2n)^2$ 1-bit additions, and $(B+2n)^2$ 2-to-1 MUXes [3]. Here, the term "1-bit additions" refers to the number of equivalent 1-bit full adders.

	Resources			
	Register array	Adder trees		Others: Dividers ($B+2n$ bits) or
	(in bits)	Number of flip-flops	1-bit additions	RAM(in bits), 2-to-1 MUXes
Serial [16],[1]	$N(B+n)$	$3B+2n$	$(B+n)$	RAM: N^2B
Systolic [16],[2]	$N(N+1)n$	$(N+1)(3B+2n)$	$(N+1)(B+n)$	RAM: $N(N+1)(B+n)$
SFDPRT	NHB	$N\mathtt{A_{ff}}(H,B)$	$N\mathtt{A_{FA}}(H,B)$ $+N(B+n)$	RAM: $N^2B+N(N+1)(B+n)$ MUX: $NH\mathtt{A_{mux}}(K+1,B)$
SFDPRT ($H=2$) lowest resource usage	$2NB$	$N(B+1)$	NB $+N(B+n)$	RAM: $N^2B+N(N+1)(B+n)$ MUX: $2N\mathtt{A_{mux}}(\lceil N/2 \rceil+1,B)$
SFDPRT ($H=N$) fastest running time	N^2B	$N\mathtt{A_{ff}}(N,B)$	$N\mathtt{A_{FA}}(N,B)$ $+N(B+n)$	RAM: $N^2B+N(N+1)(B+n)$ MUX: N^2B
FDPRT	N^2B	$N\mathtt{A_{ff}}(N,B)$	$N\mathtt{A_{FA}}(N,B)$	MUX: $2N^2B$
iSFDPRT	$NH(B+n)$	$(N+1)\mathtt{A_{ff}}(H,B+n)$ $+3N(B+2n)$	$(N+1)\mathtt{A_{FA}}(H,B+n)$ $+2N(B+2n)$	RAM: $N^2(B+2n)$, Dividers: N MUX: $NH\mathtt{A_{mux}}(K+1,B+n)$
iSFDPRT ($H=2$) lowest resource usage	$2N(B+n)$	$(N+1)(B+n+1)$ $+3N(B+2n)$	$(N+1)(B+n)$ $+2N(B+2n)$	RAM: $N^2(B+2n)$, Dividers: N MUX: $2N\mathtt{A_{mux}}(\lceil N/2 \rceil+1,B+n)$
iSFDPRT ($H=N$) fastest running time	$N^2(B+n)$	$(N+1)\mathtt{A_{ff}}(N,B+n)$ $+3N(B+2n)$	$(N+1)\mathtt{A_{FA}}(N,B+n)$ $+2N(B+2n)$	RAM: $N^2(B+2n)$, Dividers: N MUX: $N^2(B+n)$
iFDPRT	$N^2(B+n)$	$(N+1)\mathtt{A_{ff}}(N,B+n)$ $+N(B+2n)$	$(N+1)\mathtt{A_{FA}}(N,B+n)$ $+N(B+2n)$	Dividers: N MUX: $N^2(B+n)$

For results on the FPGA implementation, we present the required number of slices for a Virtex-6 implementation in Fig. 2.33. As N increases, we observe linear growth in the number of slices as expected from our analysis in Table 2.3. On the other hand, for smaller values of N, we have quadratic growth. The trends are due to the optimizations performed by the Xilinx synthesizer. Overall, since Virtex-6

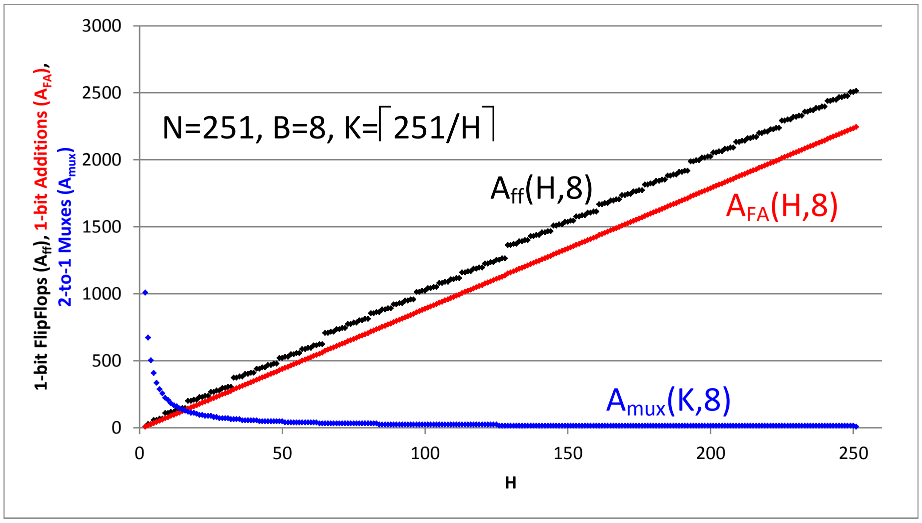


Figure 2.30: Resource functions: (i) number of adder tree flip-flops $\mathtt{A_{ff}}(.)$, (ii) number of 1-bit additions $\mathtt{A_{fa}}(.)$, and (iii) number 2-to-1 multiplexers $\mathtt{A_{mux}}(.)$ for $N = 251$, B=8. Refer to Table 2.3 for definitions.

devices use 6-input LUTS, implementations that utilize all 6 inputs provide better resource optimization than implementations that use fewer inputs. For the entire system, we have clock frequencies of 100 MHz for the Xilinx 6-series and 200 MHz for the Xilinx 7-series (Virtex-7, Artix-7, Kintex-7).

Table 2.4: Total number of resources for RAM (in 1-bit cells) and MUXes (2-to-1 muxes). The resources are shown for $N = 251$. Except for the MUXes for the SFDPRT, the values refer to any H. The number of MUXes for the SFDPRT refer to values of H that lie on the Pareto front*.

Method	**RAM**	**MUXes**
Serial [16],[1]	$504,008$	Unknown
Systolic [16],[2]	$1,012,032$	Unknown
SFDPRT	$1,516,040$	$506,016^*$
FDPRT	0	$1,008,016$

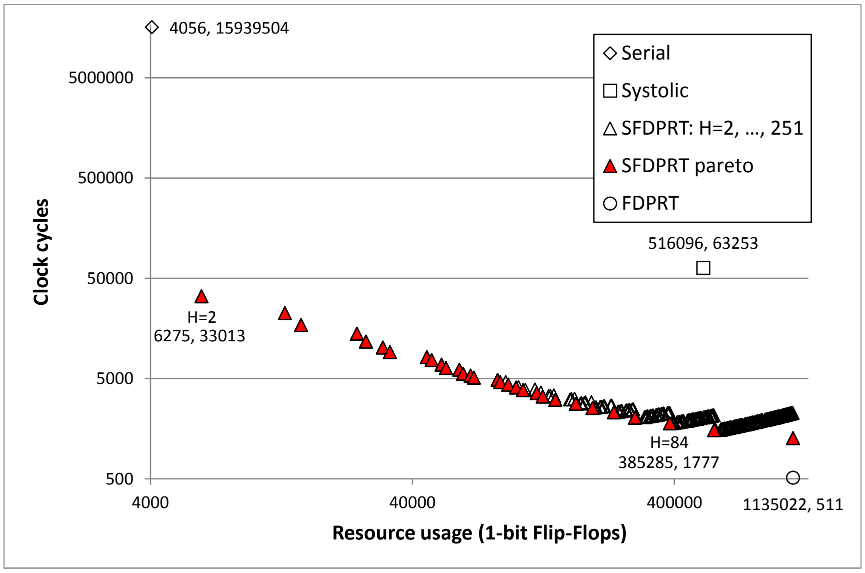


Figure 2.31: Comparative plot for the different implementations based on the number of cycles and the number of flip-flops only. Refer to Fig. 2.32 for a comparative plot for the different implementations based on the number of cycles and the number of 1-bit additions. Also, refer to Table 2.4 for a summary of RAM and multiplexer resources. The plot shows the Pareto front for the proposed SFDPRT for $H = 2, \ldots, 251$, for an image of size 251×251. The Pareto front is defined in terms of running time (in clock cycles) and the number of flip-flops used. For comparison, the serial implementation from [1], and the systolic implementation [2] is shown. The fastest implementation is due to the FDPRT that is also shown.

We also provide a summary of our results for the inverse DPRT. For the fast version (iFDPRT) running time and resources, we refer back to Figs. 2.29 and 2.33. For the number of input bits, we recall that $B' = B + \lceil \log_2 N \rceil$. Thus, overall, the iFDPRT implementations require more resources and slightly more computational times. Similar comments apply for the scalable, inverse DPRTs (iSFDPRT) shown in Figs. 2.29 and 2.33.

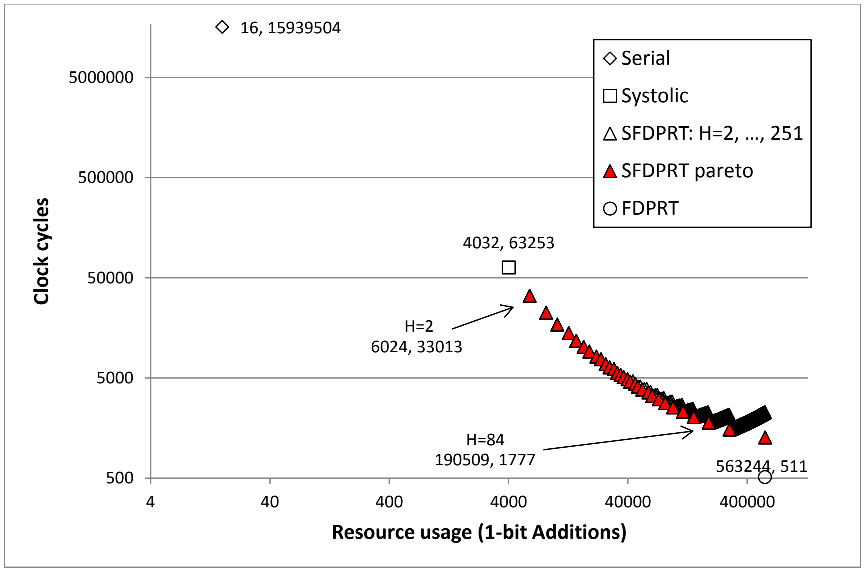


Figure 2.32: Comparative plot for the different implementations based on the number of cycles and the number of one-bit additions only (or equivalent 1 bit full adders). Refer to Fig. 2.31 for a similar comparison based on the number of flip-flops. Pareto front for the proposed SFDPRT for $H = 2, \ldots, 251$, for an image of size 251×251. The Pareto front is defined in terms of running time (in clock cycles) and the number of 1-bit additions. For comparison, the serial implementation from [1], and the systolic implementation [2] is shown. The fastest implementation is due to the FDPRT.

2.6.2 Discussion

Overall, the proposed approach results in the fastest running times. Even in the slowest cases, the running times are significantly better than any previous implementation. Scalable DPRT computation has also been demonstrated where the required number of cycles can be reduced when more resources are available. Significantly faster DPRT computation is possible for fixed size transforms when the architecture can be implemented using available resources. Furthermore, these results have been extended for the inverse DPRT. However, in some cases, the better running times come at a cost of increased resources. Thus, it is also needed a discuss how the

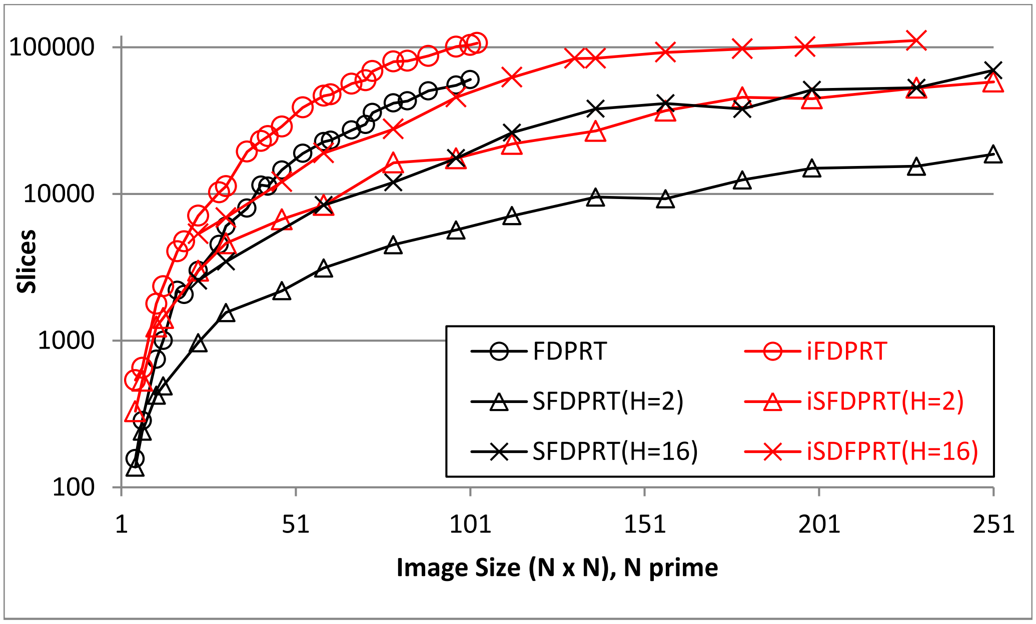


Figure 2.33: FPGA slices for a Virtex-6 implementation for both the forward and inverse DPRTs for $H = 2, 16$, N prime and $2 \leq N \leq 251$.

running times depend on the number of required resources.

For an $N \times N$ image (N prime), the proposed approaches can compute the DPRT in significantly less time than N^2 cycles. The fastest architectures (FDPRT and iFDPRT) compute the forward DPRT and inverse DPRT in just $2N + \lceil \log_2 N \rceil + 1$ and $2N + 3 \lceil \log_2 N \rceil + B + 2$ cycles respectively (where B is the number of bits used to represent each input pixel). When resources are available, the scalable approach can also compute the DPRT in a number of cycles that is linear in N. In the fastest case, the scalable DPRT requires $2N + \lceil \log_2 N \rceil + 1$ clock cycles. However, when very limited resources are available, the number of required clock cycles increases to $\lceil N/2 \rceil (N + 9) + N + 2$ for the case where only two image rows per strip $H = 2$ are used.

Based on Fig. 2.31, the number of cycles as a function of the required number of flip-flops are compared. From the Figure, note that systolic implementation requires $516,096$ flip flops to compute the DPRT for a 251×251 image in $63,253$ clock cycles (square dot in Fig. 2.31). In comparison, with 25% less resources for $H = 84$, the scalable DPRT is computed 36 times faster than the systolic implementation. On the other hand, for the serial implementation, note that the proposed scalable DPRT approaches are much faster but require more resources. The fast DPRT implementation requires only 511 cycles that is vastly superior to any other approach.

Based on Fig. 2.32, the number of cycles are compared as a function of the number of 1-bit additions. As expected, the serial implementation requires a single 16-bit adder. However, the serial implementation is very slow compared to all other implementations. The systolic implementation requires only 4,032 1-bit additions that is close to the two-row per strip ($H = 2$) implementation of the scalable DPRT. However, in all cases, the systolic implementation is significantly slower than all of the proposed implementations. Essentially, the scalable approach improves its performance while requiring more 1-bit additions for larger values of H.

As detailed in section 2.3.5, the interest is only in Pareto-optimal implementations. Here, the Pareto-optimal cases represent scalable implementations that always improve performance by using more resources. The collection of all of the Pareto-optimal implementations form the Pareto-front and are shown in Fig. 2.31 and Fig. 2.32.

The proposed system can also be expanded for use in FPGA co-processor systems where the FPGA card communicates with the CPU using a PCI express interface. Clearly, the advantage of using the proposed architecture increases with N since the image transfer overhead will not be significant for larger N. To understand the limits, assume a PCI express 3.x bandwidth of about 16 GB/s and a general-purpose microprocessor that achieves the maximum performance of 10 Giga-flops using 4 cores at 2.5 GHz. In terms of CPU memory accesses, assume a 32GB/second bandwidth for DDR3 memory. Furthermore, suppose that the interest is in computing the DPRT of a 251×251 image. In this case, image I/O requires about 3.67 micro-seconds per image transfer from the DDR3 to the FPGA card. The DPRT requires $N^2 * (N+1) \approx 15.88$ mega floating point operations for the additions. The additions can be performed in 1.479 milli-seconds (1479 micro-seconds) on the CPU. In addition, the CPU implementation will need $2N^2$ DDR3 memory accesses for implementing the transposition and retrieving the matrix in shifted form. However, assuming that these memory accesses are implemented effectively using DDR3 memory, that only requires 3.67 micro-seconds. Hence, the CPU computation will be dominated by the additions. On the other hand, DPRT computation on an FPGA operating at just 100 MHz for older devices (at half the 200 MHz of more modern FPGA devices), will only require 2*251+9 cycles in about 5.11 micro-seconds. Thus, the speedup factor is above $(3.67 + 1479)/(3.67 + 5.11) \approx 169$.

2.7 Conclusions

The chapter summarized the development of fast and scalable methods for computing the DPRT and its inverse. Overall, the proposed methods provide much faster computation of the DPRT. Furthermore, the scalable DPRT methods provide fast execution times that can be implemented within available resources. In addition, fast DPRT methods that provided the fastest execution times among all possible approaches are presented. For an $N \times N$ image, the fastest DPRT implementations require a number of cycles that grows linearly with N. Furthermore, in terms of resources, the proposed architectures only require fixed point additions and shift registers.

The proposed architectures are not tied to any existing hardware, so it can be applied to any current or future hardware architectures (like FPGAs or VLSI). This work is the first one that presented a complete solution for the iDPRT. This work introduced the novel concept of 'partial DPRT' and 'partial iDPRT' (math equations, algorithm and architecture), which leads to be able to compute the DPRT/iDPRT from a scalable point of view. Several optimizations have been proposed to the architecture to be able to compute the DPRT in parallel at high speed. Including simultaneous additions, shifting and transpositions, fast and fully pipelined adders, substractors and dividers, and customized RAMs to read a complete row or column of an image in one clock cycle.

Chapter 3

Fast 2-D Convolutions and Cross-Correlations Using Scalable Architectures

This chapter describes Fast and Scalable architectures and algorithms for computing 2-D linear convolutions for relatively large and non-separable kernels. The work presented will be submitted to:

C. Carranza, D. Llamocca, and M. Pattichis, "Fast 2-D Convolutions and Cross-Correlations Using Scalable Architectures," *IEEE Transactions on Image Processing.*

Convolution and cross-correlation are essential tools for a wide range of applications in the field of image and video processing. Example applications include medical imaging, computer vision, image restoration, multimedia, etc. [32, 33]. Further examples include feature Extraction [34], template matching [35], pattern recognition

[36], edge detection, filtering, deconvolution, segmentation, and denoising.

The performance of most image processing systems is directly affected by the speed at which the 2-D convolutions can be performed. There is thus perennial interest in developing fast methods for computing 2-D convolutions. There is also renewed interest in developing fast convolution methods that can fit in new devices.

3.1 Introduction

The goal of the current chapter is to develop fast and scalable architectures that can compute real-time convolutions with relatively large kernels. Above all, the paper is focused on the development of fast architectures that can reduce the latency of the convolution components of the system to be significantly below the number of elements in the image. At such speeds, it is clear that larger image processing systems can benefit significantly from the use of much faster convolution components. It is also clear that this requirement significantly exceeds the standard use of 2D FFT architectures that serially compute $O(N^2 \log N)$ flops for $N \times N$ images. Similarly, our basic design requirement also exceeds the standard use of systolic array implementations that process one pixel per cycle. The proposed transform-domain design also outperforms spatial domain methods, as it is explained later in this section.

To support implementations in modern devices (e.g., FPGAs), there is also an interest in scalable architectures. The basic idea is to make efficient use of hardware resources to deliver the best possible performance. The scalability requirement is split into performance and data scalability. For performance scalability, there is a need to investigate implementations that can be fitted within available resources. Thus, the current paper describes families of different hardware implementations that can be fitted to available hardware resources. For data scalability, it is expected that the hardware can be used to perform fast convolutions on large images without having the data processing resources grow with image size.

Additionally, there is an interested in real-time applications where the convolution kernel can be varied in real-time. As an example, consider adaptive filtering applications. During operation, a filter may have to be adjusted to meet different constraints on power, energy, or accuracy. An example of such an application can be found in our own research on Dynamically Reconfigurable Architectures for Time-Varying Image Constraints (DRASTIC) [37]. In video processing applications, the filtering coefficients can be adjusted to preserve energy, or provide more accurate results during a scene change [38]. For real-time convolution applications, it is thus desirable to be able to change the convolution kernel in real-time, without the requirement for any offline computations.

A standard approach for developing efficient architectures for 2-D convolutions would be to build the systems based on 2-D FFTs. As is well-known (e.g., see [39, 40]), for sufficiently large kernels, the use of 2-D FFTs will give better results than direct approaches. Unfortunately, the direct implementation of 2-D FFTs in hardware requires the use of expensive (hardware-intensive) processing and control units to implement complex arithmetic using floating point numbers. As a result, the hardware scalability of using 2-D FFTs is fundamentally limited by the number of 1-D FFT processors that can be fitted in any given hardware device. Refer to [41, 42] for details of the latest implementation of this approach. As shown in [41], performance can be improved by including up to 8 1-D FFT processors. Beyond 8 1-D FFT processors, performance degrades due to I/O issues.

Two-dimensional convolutions can also be computed in the transform domain using the 2-D Discrete Periodic Radon Transform (DPRT). The DPRT can be computed using summations along different directions [43]. Similar to the FFT, the DPRT approach requires to first apply the DPRT of the image and the 2-D kernel. Then, along each DPRT direction, 1-D convolutions between the DPRTs of the image and the 2-D kernel are computed. The 2-D convolution result can then be computed by taking the inverse DPRT of the previous result. For large, non-separable kernels, it is developed a scalable family of DPRT-based architectures that leads to very fast computations that range from $O(N)$ to $O(N^2)$ cycles for $N \times N$ images. As expected, the scalable family will produce faster implementations at the expense of more computational resources.

In the spatial-domain, an important alternative comes from the use of systolic arrays [44]. The standard systolic array implementation of 1-D convolutions computes an output every clock cycle. Without using separability, a direct extension of the 1-D systolic array approach would require to keep several image rows in memory [45, 46]. As a result, the application of non-separable systolic array implementations has been limited to small kernels. Furthermore, it is clear that parallelization by using multiple copies of these non-separable implementation is impractical for larger kernels, i.e in [45] the resource grows as $O(N^3)$.

In the spatial domain, there is a relatively recent emergence of fast convolution using a sliding window [47]. At each image pixel, a sliding window of the same size as the kernel is applied to compute one output pixel of the convolved image [48]. This comes at a cost of using as many multipliers and adders as the coefficients in the kernel, and thus grows linearly with the number of coefficients in the kernel.

An important extension of the systolic approach can be derived by using separable approximations of non-separable kernels [49]. The basic idea is to express non-separable kernels as a sum of separable kernels. Then, scalable hardware implementations can be derived by controlling the number of efficient 1-D processors. Furthermore, as it will be demostrated later in this paper, this approach can also reduce the required number of 1-D kernels using the singular value decomposition (SVD) [50].

Overall, the dissertation describes the development of two families of scalable architectures that map 2-D convolutions into fast 1-D convolutions. For the first family, the approach requires the pre-computation of (i) a spatial-domain separable decomposition based on an SVD-LU factorization (see Chapter 18 of [49]), and (ii) a transform-domain separable decomposition of the Discrete Periodic Radon Transform (DPRT). The first family is ideally suited for implementing non-separable filterbanks where the filters can be computed ahead of time. For the second family, the dissertation considers efficient architectures that compute all DPRTs and the inverse DPRT in real-time so as to avoid any offline computations. The second family is ideally suited for implementing cross-correlations, adaptive filterbanks, or implementing filterbanks without requiring any preprocessing.

A summary of the primary architectural elements of the proposed design are:

- ***An array of circular-shift-registers:*** The image data is stored and processed using an array of circular shift registers. The memory array is implemented using a row of SRAMs where each SRAM stores a column of the image.

- ***Row-level parallel I/O:*** The scalable architectures load the image into memory using a sequence of parallel loads of rows of pixels. Thus, for an image with N rows, The entire image can be loaded into memory in N cycles.

- ***Row-level parallel and pipelined processing:*** The proposed scalable architectures are designed to process multiple rows at the same time. Thus, for FPGA implementations, the idea is to implement as many row-processing units as they can fit in the device. Then, each row-processor uses a pipelined architecture that produces results after each cycle after an initial latency.

- ***Fast transpositions:*** A significant reduction in the transposition overhead using an additional output memory array. The output memory array uses dual-port memories to allow us to write the output results and read intermediate values at the same time. Based on the proposed approach, rows and columns can read and write in a single cycle as needed. Overall, in the pipelined design, the net effect is that transposition is performed during computation and will thus not require any additional cycles.

The scalability characteristics of the proposed architectures include:

- ***Data scalability using overlap and add:*** The overlap and add property is used to allow processing of larger images. Furthermore, by controlling the image block size, a trade-off between performance and resources is provided. Convolutions can be computed faster by processing larger blocks subject to available resources.

- ***Performance scalability by controlling the number of row-processors in the DPRT and the 1-D convolutions:*** Refer to [3] for the scalable DPRT implementation.

- ***Pareto optimality:*** Pareto-optimal designs are presented in the sense that the proposed family of architectures provide the fastest implementations based on available resources. In other words, additional resources always yield faster performance.

- ***Fast 2-D Convolutions:*** The fastest architectures can compute 2-D convolutions in $O(L \cdot N)$ cycles where N represents the number of pixels processed with each row and L represents the total number of image blocks. On the other hand, in the worst case scenario, with very limited resources, 2-D convolutions can be computed in $O(N^2)$ cycles.

In addition, for the implementation of fast cross-correlations and convolutions without pre-processing, the fast and scalable implementations of the DPRT that were presented in [3] will be used.

The rest of the paper is organized as follows. The mathematical definitions for the DPRT, its inverse, and the transformation property of the DPRT are given in section 3.2. The proposed approach is given in section 3.3. Section 3.4 presents the results. Conclusions and future work are given in section 3.5.

3.2 Background

3.2.1 Basic notation

Let $g(i,j)$ denote an image of P_1 rows with P_2 pixels per row be of size $P_1 \times P_2$ with B bits per pixel. The image $g(i,j)$ is indexed using $0 \leq i \leq P_1 - 1$ and $0 \leq j \leq P_2 - 1$. The access to an entire row is done using $g_i(j)$ and an entire column using $g_j(i)$. For the convolution kernels, the symbol h is used and assume a size of $Q_1 \times Q_2$ with C bits per pixel. $f(i,j)$ is used for the output, with size $N_1 \times N_2$ where $N_1 = P_1 + Q_1 - 1$ and $N_2 = P_2 + Q_2 - 1$, and simply use N for the special case of $N_1 = N_2$.

3.2.2 Separable decomposition for non-separable kernels

First, the 2-D Z-transform of the convolution kernel h is given by:

$$H(z_1, z_2) = \sum_{i=0}^{Q_1-1} \sum_{j=0}^{Q_2-1} h(i,j) z_1^{-i} z_2^{-j}. \tag{3.1}$$

To allow for separable decompositions, consider a matrix re-formulation of (3.1) [49]:

$$H(z_1, z_2) = \begin{bmatrix} 1 & z_1^{-1} & z_1^{-2} & \dots & z_1^{-(Q_1-1)} \end{bmatrix}$$
$$\begin{bmatrix} h(0,0) & h(0,1) & \dots & h(0,Q_2-1) \\ h(1,0) & h(1,1) & \dots & h(1,Q_2-1) \\ \vdots & \vdots & \vdots & \vdots \\ h(Q_1-1,0) & h(Q_1-1,1) & \dots & h(Q_1-1,Q_2-1) \end{bmatrix}$$
$$\begin{bmatrix} 1 \\ z_2^{-1} \\ \vdots \\ z_2^{-(Q_2-1)} \end{bmatrix}$$
$$= \mathbf{Z_1}^T \, \mathbf{H} \, \mathbf{Z_2} \tag{3.2}$$

where all of the filter coefficients have been placed in $\mathbf{H}$, and for $i = 1, 2$ $\mathbf{Z_i} = [1\, z_i^{-1}\, z_i^{-2} \dots z_i^{-(Q_i-1)}]$. Now that the filter coefficients are in matrix form, it can be considered separable matrix approximations to $\mathbf{H}$. First, consider the singular value decomposition (SVD) for $\mathbf{H}$: $\mathbf{H} = \mathbf{U\Sigma V}^T$. Then, $\mathbf{H}$ can be simplified by zeroing out the smallest singular values of $\mathbf{\Sigma}$. Let $\mathbf{\Sigma_m}$ denote the resulting $\mathbf{\Sigma}$ after zeroing-out small singular values, an effective reconstructed approximation to $\mathbf{H}$ using $\mathbf{H_r} = \mathbf{U\Sigma_r V}^T$ where the r larges singular values of $\mathbf{H}$ are kept. In this case, using the LU decomposition of $\mathbf{H_r}$ to get [49]:

$$H_r(z_1, z_2) = \sum_{k=1}^{r} \left(\sum_{i=0}^{Q_1-1} l_{ki}^m z_1^i \right) \left(\sum_{j=0}^{Q_2-1} u_{jk} z_2^j \right) \tag{3.3}$$

where r also denotes the rank of $\mathbf{H_r}$. In (3.3), it is expressed the original 2-D convolution into a sum of r separable 1-D convolutions along the rows and columns. Furthermore, it is clear that the separable decomposition also applies to non-separable 2-D kernels.

3.2.3 The discrete periodic radon transform (DPRT)

The DPRT of an image f of size $N \times N$, N prime, using [18] is defined as:

$$F(m,d) = \begin{cases} \sum\limits_{i=0}^{N-1} f(i, \langle d+mi \rangle_N), & 0 \leq m < N, \\ \\ \sum\limits_{j=0}^{N-1} f(d,j), & m = N, \end{cases} \tag{3.4}$$

where $d = 0, 1, \ldots, N-1$, $m = 0, 1, \ldots, N$, and $< . >_N$ denotes the positive remainder when an integer division by N is performed (e.g., $< 128 >_{127} = 1$). In (3.4), m indexes the prime directions. Along each prime direction, the pixels are added up along each ray. In (3.4), d is used to index each the rays of each direction.

The inverse DPRT can be used to reconstruct the image from the forward DPRT using:

$$f\left(i,j\right) = \frac{1}{N}\left[\sum_{m=0}^{N-1} F\left(m, \langle j-mi \rangle_N\right) - S + F\left(N,i\right)\right] \tag{3.5}$$

where:

$$S = \sum_{j=0}^{N-1}\sum_{i=0}^{N-1} f(i,j). \tag{3.6}$$

As noted in the definition, the size of the transform needs to be restricted to prime numbers. This restriction is not imposed directly to the image and kernel sizes, but to the result of the linear convolution of size $N_1 \times N_2$, with $N_1 = P_1 + Q_1 - 1$ and $N_2 = P_2 + Q_2 - 1$. Therefore, a minimal (or even none) zero padding is required if the input sizes are selected conveniently. There are several reasons for imposing this restriction. Most importantly though, for prime N, the DPRT provides the most efficient implementations by requiring the minimal number of $N+1$ primal directions [21]. It is important to note that prime-numbered transforms have advantages in convolution applications. Here, just like for the Fast Fourier Transform (FFT), zero-padding is used to extend the DPRT for computing convolutions in the transform domain. Unfortunately, when using the FFT with $N = 2^p$, zero-padding requires to use FFTs with double the size of N. In this case, it is easy to see that the use of prime-numbered DPRTs is better since there are typically many prime numbers between 2^p and 2^{p+1}.

Refer to [3] for fast and scalable implementations of the DPRT. In the fastest case, the full DPRT can be computed in just $2N + \lceil \log_2 N \rceil + 1$ clock cycles with $O(N^2)$ growth in resource usage. In the scalable DPRT implementation, it is required $\lceil N/2^h \rceil N + 2N + h$ cycles where h is used as the scalability parameter. A family of scalable DPRT implementation is defined using $h = 2, \ldots, \lceil \log_2 N \rceil$ with a resource usage that varies between $O(N)$ for $h = 2$ and $O(N^2)$ for $h = \lceil \log_2 N \rceil$.

3.2.4 Circular convolutions using the DPRT

Consider the 2-D circular convolution $f = g \otimes h$ given by:

$$f(k,l) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} g(i,j) h(\langle k-i \rangle_N, \langle l-j \rangle_N). \tag{3.7}$$

To define the DPRT convolution property, let m denote a prime direction and define the DPRTs along the m-direction using: $F_m(d) = F(m,d), G_m(d) = G(m,d)$, $H_m(d) = H(m,d)$. Then, the m-direction DPRTs are related through 1-D dimensional circular convolution in the transform domain as given by [43]:

$$F_m(d) = \sum_{k=0}^{N-1} G_m(k) H_m(\langle d-k \rangle_N) \tag{3.8}$$

Thus, the result of 2-D circular convolution can be computed in the transform domain using 1-D circular convolutions along all of the prime directions as given by (3.8). After computing the DPRT of the result along each direction, the inverse DPRT can be applied to recover f.

3.3 Methodology

3.3.1 Computing 1-D circular convolutions using circular shifts

In this section, 1-D circular convolutions using circular shifts is reformulated. Here, the primary application will be to compute the circular convolutions in the DPRT domain. Thus, the DPRTs of f, g, h are used in the derivation. Let $F_m(d), G_m(d)$, $H_m(d)$ denote the DPRTs of f, g, h along the m-th prime direction.

A special flip operation $\breve{H}_m = H_m$ is defined by:

$$\breve{H}_m(d) = H_m(N-1-d), \quad d \geq 0,$$

and the circular right shift (CRS) by n using $H_m^n = H_m$ that is defined by:

$$H_m^n(d) = H_m(\langle d + n \rangle_N).$$

Then, using (3.8) to derive a shifted representation of the circular convolution:

$$\begin{aligned}
F_m(d) &= \sum_{k=0}^{N-1} G_m(k)\, H_m(\langle d - k \rangle_N) \\
&= \sum_{k=0}^{N-1} G_m(k)\, H_m(\langle N - 1 - k + d + 1 \rangle_N) \\
&= \sum_{k=0}^{N-1} G_m(k)\, H_m^{d+1}(N - 1 - k) \\
&= \sum_{k=0}^{N-1} G_m(k)\, \breve{H}_m^{d+1}(k). \qquad (3.9)
\end{aligned}$$

From (3.9), note that $F_m(d)$ can be expressed as the dot product between G_m and a flipped and circular right shifted by $d+1$ positions version of H_m (denoted as $\breve{H}_m^{d+1}$).

3.3.2 Fast 1-D circular convolution hardware implementation

In this section, a fast hardware implementation based on (3.9) is derived. The hardware architecture is presented in Fig. 3.1, the associated algorithm in Fig. 3.2, and the timing diagram in Fig. 3.3.

The sequence of operations is given in Fig. 3.2. Initially, parallel loads to transfer both of the DPRTs to the G and H registers in a single clock cycle are used. Note that flipping H_m into $\breve{H}_m$ is performed by simply wiring the inputs in reverse as shown in the upper register portion of Fig. 3.1. Starting with last convolution output, there is a 3-step sequence of parallel multiplies, addition of the results, and a circular right shift to prepare for the next output (lines 3-5). The multiplications are performed in parallel in a single cycle using the parallel fixed-point multipliers of Fig. 3.1 and added using a pipelined tree structure in just $\lceil \log_2(N) \rceil$ clock cycles (e.g., see [3]). The resulting outputs are left-shifted in, one output sample at a time, into the output F register shown in the lower-right portion of Fig. 3.1. A single cycle is also needed to perform the circular right shift of H using the top-left register of Fig. 3.1.

To derive the timing requirements, refer back to Fig. 3.3. Using a fully pipelined approach, the next output sample computation starts after the parallel multiplication occurs. It is easy to see that after the initial latency for the first sample, an output sample is computed at every cycle. After adding the latency for the initial loads, the adder latency, and the final left shift, a total of just $N + \lceil \log_2(N) \rceil + 2$ clock cycles is needed.

3.3.3 Fast and scalable 2-D linear convolutions and cross-correlations

In this section, the architectures, algorithms, bit requirements, and computational efficiency for 2-D convolutions and cross-correlations are developed. Most importantly, the scalability of the proposed approach that allows for the most efficient implementations based on available resources is discussed.

In what follows, refer to the sequence of operations for computing fast and scalable

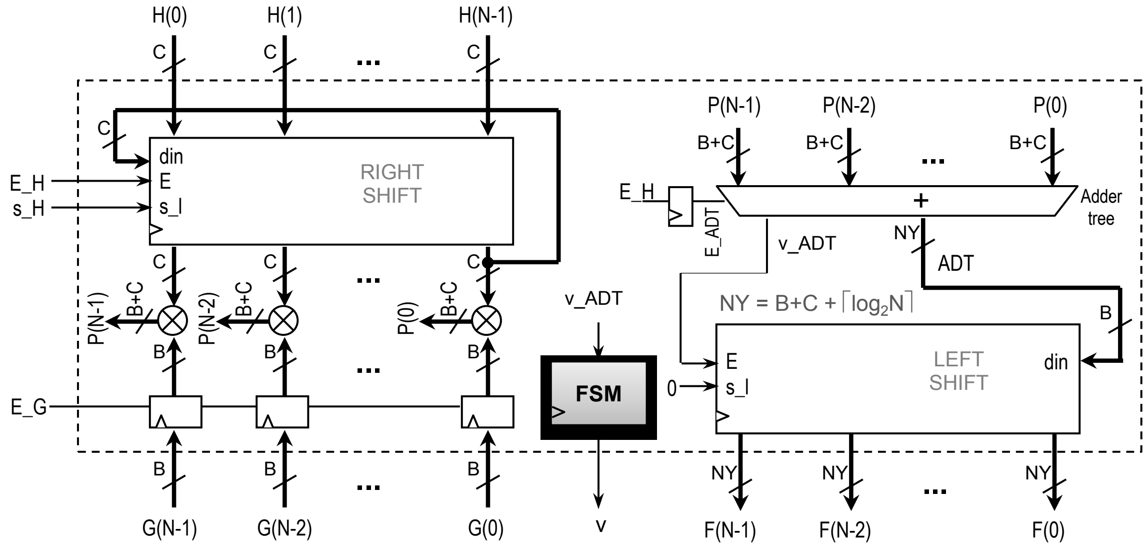


Figure 3.1: Architecture for computing the 1-D circular convolution $F_m = G_m \otimes H_m$.

```
Parallel load $G = G_m$, flipped load $H = \breve{H}_m$
for $d = N - 1$ downto 0 do
    Parallel mult. $P(k) = G(k)H(k),\ 0 \leq k \leq N - 1$.
    Parallel add $F(d) = \sum_{k=0}^{N-1} P(k)$
    CRS by one $H = H^1$
end for
Parallel output $F$
```

Figure 3.2: Algorithm for computing the 1-D circular convolution $F_m = G_m \otimes H_m$.

2D convolutions and cross-correlations as shown in Fig. 3.4. In the most efficient implementation, the convolution kernel is available ahead of time thus the kernel DPRT is pre-computed and stored in memory as shown in the hardware architecture of Fig. 3.5. On the other hand, for adaptive filterbank applications, the DPRT of the zero-padded convolution kernel can be computed in real-time. Furthermore, for cross-correlation computations, a fast transposition is also needed as described in Fig. 3.4.

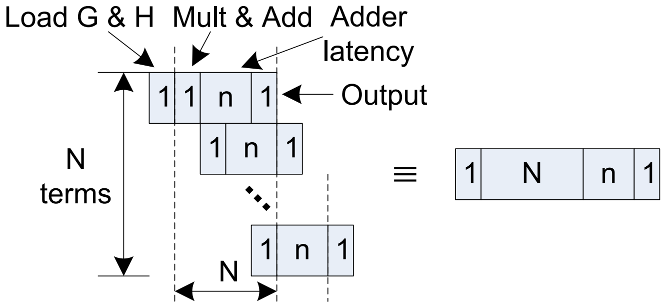


Figure 3.3: Running time for the implementation of the fast architecture for computing 1-D Circular convolutions. In this diagram, time increases to the right. The number of clock cycles for each term of $F_m(d)$ is shown on each strip. The strip on the right represents the total running time. $n = \lceil \log_2 N \rceil$ represents the addition latency.

Scalability is achieved by setting: (i) J which is the number of 1D circular convolutions that can be computed in parallel, and (ii) H which is the number of image rows that can be processed in parallel in the DPRT blocks as described in [3]. Following the computation of the 1D circular convolutions, an inverse DPRT is applied for computing the final result.

A derivation of bit requirements is also presented. Let the input image g be of size $P_1 \times P_2$ with B bits per pixel and the kernel h of size $Q_1 \times Q_2$ with C bits per pixel. Then, it is easy to see that bit requirements include: (i) $B + n$ bits for the DPRT of g, $C + n$ bits for the DPRT of h where $n = \lceil \log_2 N \rceil$ (also see [3]), (ii) $B + C + 3n$ bits for the convolutions, and (iii) $B + C + 4n$ bits just before the normalization step of the inverse DPRT [3], and $B + C + 2n$ bits for the final result. For simplicity, assume square image blocks. Then $N = \texttt{NextPrime}(\max(P_1 + Q_1 - 1,\ P_2 + Q_2 - 1))$. Next, to compute the required number of cycles based on J, H, and N (for square images), computational complexity needs to be derived for: (i) the DPRT of the image, (ii) the circular-convolutions, and (iii) the inverse DPRT for the final result. Here, for adaptive filterbank computation, the DPRTs of the image and the convolution kernel can be computed in parallel without any additional latency. Furthermore, cross-correlation computation would only add an extra cycle for the transposition.

As summarized in section 3.2.3 and [3], scalable DPRT computation requires $\lceil N/2^h \rceil N + 2N + h$ clock cycles that reduces to $2N + \lceil \log_2 N \rceil + 1$ clock cycles for the fast DPRT. For computing the number of cycles required for the circular convolutions, refer to Figs. 3.6 and 3.7. As shown in Fig. 3.6, by using J convolution blocks working in parallel, it is required $J + N + n + 1$ clock cycles to compute J convolutions, where $n = \lceil \log_2 N \rceil$ represents the addition latency. To compute outputs for all of the $N + 1$ required DPRT directions, it is loaded and processed up to J blocks at a time. After the J cycles required for the first set, we wait for N additional cycles until the next set, and so on. Overall, it is required a total of $L(J + N) + n + 1 - (J - J^{'})$ clock cycles where $L = \lceil (N + 1)/J \rceil$, $J^{'} = \langle N + 1 \rangle_J$, and $n = \lceil \log_2 N \rceil$. Depending on available resources, the fastest running time for $J = N + 1$ takes $2N + n + 2$ clock cycles with resource usage of $O(N^2)$, and the slowest for $J = 1$ at $(N + 1)^2 + n + 1$ clock cycles with the lowest resource usage $O(N)$. A detailed analysis of computational resources is provided in section 3.4.

Lastly, the inverse DPRT is applied using the `iSFDPRT_System` module. Similar to the forward DPRT, scalability is controlled by H, the number of image rows processed in parallel [3]. For this step, as mentioned earlier, the input data uses $B + C + 3n$ bits per pixel. Depending on available resources, the inverse DPRT can

```
1: Precompute/Compute
      $H = \texttt{DPRT}\{\texttt{ZeroPad}\{h\}\}$
      and store the results in memory.
   ▷ For cross-correlation, replace step 1 with:
   ▷     Transpose $h$ in a single cycle
   ▷        using custom memory and
   ▷     Compute $H = \texttt{DPRT}\{\texttt{ZeroPad}\{h\}\}$
2: Compute $G = \texttt{DPRT}\{\texttt{ZeroPad}\{g\}\}$
3: for $p = 0$ to $L - 1$ do
4:     Compute $J$ directions in parallel:
         $F_{pJ+i} = G_{pJ+i} \otimes H_{pJ+i}, \quad$ for $i = 0, \ldots, J-1$.
5: end for
6: Compute $f = \texttt{DPRT}^{-1}\{F\}$
```

Figure 3.4: Fast and scalable algorithm for computing 2-D linear convolutions and cross-correlations between $g(i, j)$ and $h(i, j)$ using the architecture depicted in Fig. 3.5.

be computed in just $2N + 5n + B + C + 2$ for $H = N$ with $O(N^2)$ resource usage, or as as slow as $\lceil N/2 \rceil (N + 2) + 4n + B + C + 4$ for $H = 2$ for just $O(N)$ resource usage.

Overall, in the fastest case, convolutions and cross-correlations can be computed in just $O(N)$ clock cycles. Even in the slowest case, with the lowest resource requirements, it is only required $O(N^2)$ clock cycles. In section 3.4, carefull analysis for the required amounts of resources and corresponding clock cycles for each case is done.

3.3.4 Scalable 2-D Linear Convolution using LU decomposition (S2DLCLU)

Consider the 2-D linear convolution between g of size $P1 \times P2$ and a non-separable kernel h of size $Q1 \times Q2$. From section 3.2.2, a non-separable kernel can be decom-

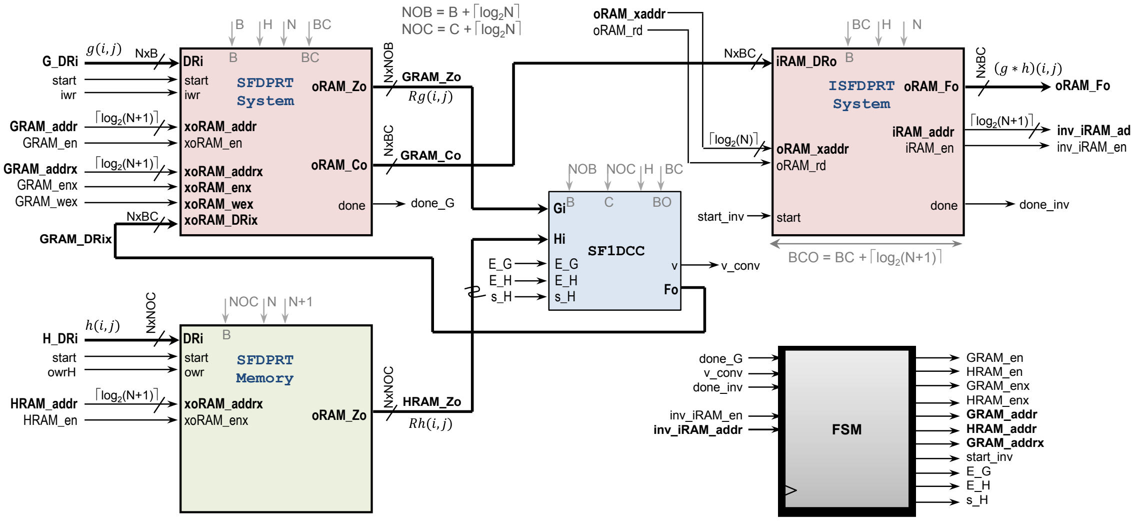


Figure 3.5: Fast and scalable architecture system for computing 2D convolutions. A modification is needed for computing fast cross-correlations (see below). Refer to Fig. 3.4 for the sequence of operations. The DPRT is computed by a fast and scalable block denoted by `SFDPRT_System`. `SFDPRT_System` computes the DPRT of the zero padded input image g. For regular convolution kernels, the DPRT of the zero-padded convolution kernel h can be pre-computed and stored in the `SFDPRT Memory` block as shown here. Alternatively, in adaptive filterbank applications, it can be introduced an extra `SFDPRT_System` block for computing the DPRT in real time. Furthermore, for computing cross-correlations in real-time, a fast transposition is needed before applying the DPRT. It is computed J circular convolutions in parallel (row-wise) using the `SF1DCC_System` block. Control is performed by a finite state machine (`FSM` block).

posed into r separable kernels. The 2-D convolution can then be computed using r 1-D convolutions with the separable kernels and accumulating the results.

The new S2DLCLU system is based on a modified version of the F1DCC convolver presented in section 3.3.2. The standard steps are then: (i) apply 1-D linear convolution between each row of the image and the row-kernel, storing the results in a temporal RAM, (ii) transpose the resulting image, (iii) apply 1-D linear convolution between each row of the resulting image and the column-kernel, accumulating the results in an output RAM, (iv) repeat (i)-(iii) r times for the complete 2-D linear

Table 3.1: Resource usage for different 1-D Circular Convolutions implementations. Here, there are two zero padded images g and h of size $N \times N$, B and C bits per pixel respectively and $n = \lceil \log_2 N \rceil$. For the adder tree, it is defined $\mathtt{A_{ffb}}$ to be number of required flip-flops including input buffers, and $\mathtt{A_{FA}}$ to be the number of 1-bit additions. $\mathtt{A_{ffb}}$ and $\mathtt{A_{FA}}$ grow linearly with respect to N and can be computed using the algorithm given in the appendix (Fig. C.1). For the multipliers, note that each one is implemented using two inputs of size $B+n$ and $C+n$ bits and an output of $B+C+2n$ bits. Here, the term "1-bit additions" refers to the number of equivalent 1-bit full adders.

	Resources		
	Number of flip-flops	1-bit additions	Multipliers
F1DCC	$N(2B+2C+5n)$ $+\mathtt{A_{ffb}}(N, B+C+2n)$	$\mathtt{A_{FA}}(N, B+C+2n)$	N
SF1DCC	$JN(2B+2C+5n)$ $+J\mathtt{A_{ffb}}(N, B+C+2n)$	$J\mathtt{A_{FA}}(N, B+C+2n)$	JN

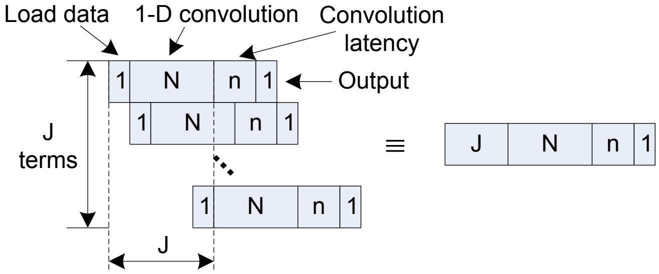


Figure 3.6: Running time for computing J circular convolutions in parallel using J fast convolution blocks (see basic block structure in Fig. 3.1). In this diagram, time increases to the right. Here, it takes one cycle to perform a parallel load for each block. Overall, it is required $J+N+n+1$ to compute everything, where $n = \lceil \log_2 N \rceil$ represents the addition latency.

convolution.

However, the load of the data (row or column) and the transposition can slowdown the processing. To develop a high-speed solution, the key is to avoid I/O limitations and eliminate the transposition step. To this end, an SRAM system is used in

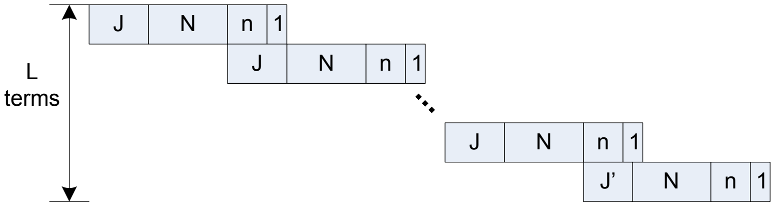


Figure 3.7: Running time for computing $N + 1$ 1-D circular convolutions using J fast convolution blocks operating in parallel. In this diagram, time increases to the right. The convolution blocks need to be reloaded L times, and is given by $L = \lceil (N + 1)/J \rceil$. For the last load, only $J^{'} = \langle N + 1 \rangle_J$ if $\langle N + 1 \rangle_J \neq 0$ convolution blocks are needed. Each row shows the running time for performing J convolutions as described in Fig. 3.6.

four steps. First, `MEM_IN` provides a full row of the image per clock cycle. Second, `MEM_KER` provides a full row or column of the filter coefficients in one clock cycle. Third, `MEM_TMP` is used for holding the temporal row-convolution, and is modified to receive up to $P1$ values of the convolved rows per clock cycle, and also to provide a full column per clock cycle. Fourth, `MEM_OUT` is used for accumulating the final result, while it can also add up to $P2 + Q2 - 1$ values of the convolved image per clock cycle and perform reads of a full row of pixels in a single cycle.

All listed memories follow the architecture described in Fig. 3.8 and are configured as shown in Table 3.2. Without loss of generality, from here assume that $P2 \geq P1$, $Q2 \geq Q1$, and consequently $N2 \geq N1$. Note that the DPRT is not needed in this case.

In the worst case scenario (h is full rank, and all singular values are kept), $r = Q1$. The 1-D linear convolution is carried out by the Fast 1-D Linear Convolver (F1DLC). The block diagram of the F1DLC is presented in Fig. 3.9 and the associated algo-

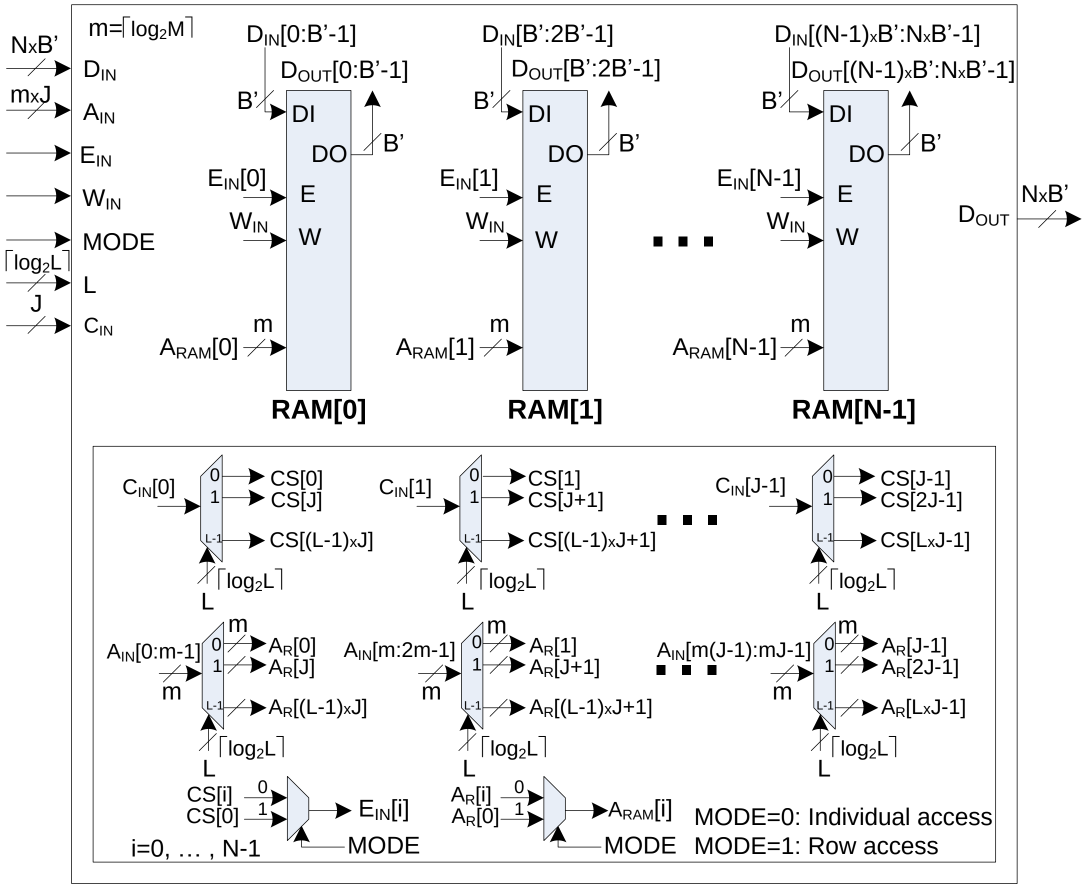


Figure 3.8: Custom SRAM architecture of size $M \times N$, B' bits depth to hold an image of M rows and N columns, capable to read/write a full row in one clock cycle (MODE=1) and allows individual access up to J SRAMs per clock cycle (MODE=0). See Table 3.2 for configuration details.

rithm in Fig. 3.10. The running time is the same as F1DCC, replacing N by SG and n to $\lceil \log_2 SH \rceil$, i.e. $SG + \lceil \log_2 SH \rceil + 2$ clock cycles. The resources grow linearly with respect to SG (see Table 3.3 for exact values).

The complete S2DLCLU system to compute the 2-D linear convolution between the image $g(i,j)$ and the non-separable kernel $h(i,j)$ using LU decomposition with r separable kernels is shown in Fig. 3.11 and the associated algorithm in Fig. 3.12. The scalability is controlled by the parameter J (number of F1DLC convolvers).

Table 3.2: SRAM System configuration. The Word size listed is for maximum accuracy. Orientation refers to each SRAM holding either for a full row or column of the image. The Accumulate mode needs external adders to perform the accumulation and dual-port SRAMs for full speed. Consider B as the number of bits of the input image, C bits for kernel coefficients. $q1 = \lceil \log_2 Q1 \rceil$, $q2 = \lceil \log_2 Q2 \rceil$

SRAM	`MEM_IN`	`MEM_KER`	`MEM_TMP`	`MEM_OUT`
Quantity	$P2$	$Q2$	$P1$	$P2 + Q2 - 1$
Size	$P1$	$2Q2$	$P2 + Q2 - 1$	$P1 + Q1 - 1$
Word B'	B	C	$B + C + q2$	$B + 2C + q1 + q2 + r$
Function	g	h_R, h_C	g'	f
MODES	1	1	0/1	0/1
Orientation	Column	Column	Row	Column
WriteMode	Store	Store	Store	Accumulate

Let $J'_R = \langle P1 \rangle_J$ if $\langle P1 \rangle_J \neq 0$, otherwise $J'_R = J$. Let $J'_C = \langle P2 + Q2 - 1 \rangle_J$ if $\langle P2 + Q2 - 1 \rangle_J \neq 0$, otherwise $J'_C = J$. Let $L_R = \lceil P1/J \rceil$. And let $L_C = \lceil (P2 + Q2 - 1)/J \rceil$. The running time is given by the row processing $(L_R(J + P2 +$

Table 3.3: Resource usage for different Linear Convolvers implementations. Here, all the quantities are given for maximum accuracy. For the adder tree, define $\texttt{A}_{\texttt{ffb}}$ as the number of required flip-flops including input buffers, and $\texttt{A}_{\texttt{FA}}$ to be the number of 1-bit additions. $\texttt{A}_{\texttt{ffb}}$ and $\texttt{A}_{\texttt{FA}}$ grow linearly with respect to $Q2$ and can be computed using the algorithm given in the appendix (Fig. C.1). For the multipliers, note that each one is implemented using two inputs of size $B + C + q2$ and C bits and an output of $B + 2C + q2$ bits. Here, the term "1-bit additions" is used to refer to the number of equivalent 1-bit full adders. Recall $N2 = P2 + Q2 - 1$.

	Resources		
	Number of flip-flops	1-bit additions	Multipliers
F1DLC	$N2 \times (B + C + q2) + Q2 \times C$ $+\texttt{A}_{\texttt{ffb}}(Q2, B + 2C + q2)$	$\texttt{A}_{\texttt{FA}}(Q2, B + 2C + q2)$	$Q2$
S2DLCLU	$J \times (N2 \times (B + C + q2) + Q2 \times C$ $+\texttt{A}_{\texttt{ffb}}(Q2, B + 2C + q2))$	$J \times \texttt{A}_{\texttt{FA}}(Q2, B + 2C + q2)$	$J \times Q2$

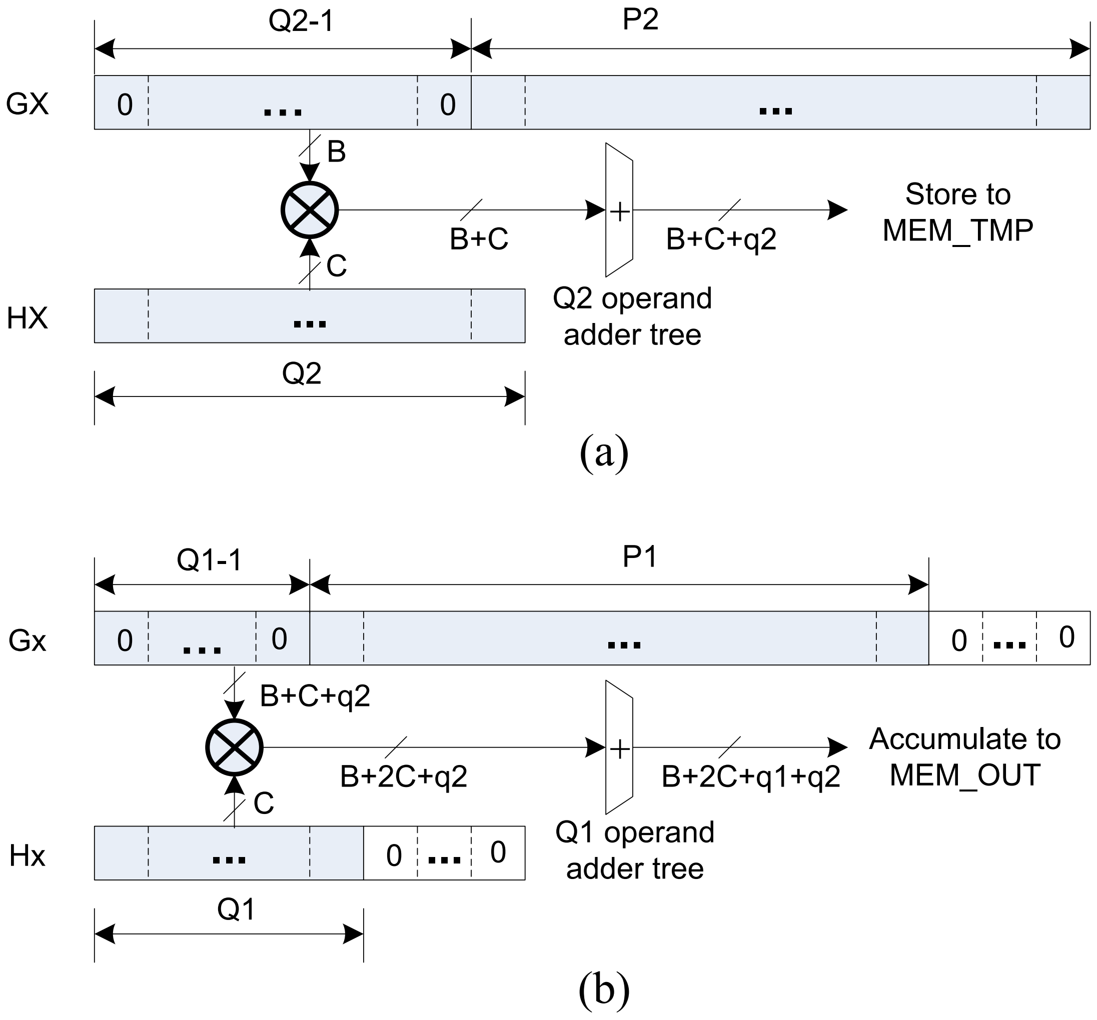


Figure 3.9: Fast 1-D linear convolver (F1DLC) block representation. Assume $P_2 \geq P_1$, $Q_2 \geq Q_1$. GX size is $P_2 + Q_2 - 1$, HX size is Q_2. Gray boxes denotes the usage of the F1DLC. Bit usage is for full accuracy. Recall, B is the number of bits for the input image, C for the kernel, $q1 = \lceil \log_2 Q_1 \rceil$ and $q2 = \lceil \log_2 Q_2 \rceil$. The set of Q_2 multipliers is represented by the $\otimes$ symbol, the input and output bits for each one is indicated in the. All the multipliers are connected to a Q_2-operand adder tree. (a) Convolver processing rows. (b) Convolver processing columns.

$Q2-1)-(J-J'_R))$, plus the column processing $(L_C(J+P1+Q1-1)-(J-J'_C))$, both added and repeated r times, plus the latency of the adder tree $(\lceil \log_2 Q1 \rceil + 1)$. Consider $N = max\{P2+Q2-1, P1+Q1-1\}$, then $J = 1$ gives the minimum

```
procedure F1DLC($GIN(x), SG, SH, MEM$)
    load $GX[SH-1 : SG-1] = GIN(x)$
    load $GX[0 : SH-2] = 0$
    for $s = 0$ to $SG - 1$ do
        in parallel multiply $P(a) = GX[a]HX[a]$
            $a = 0, \ldots, SH-1$
        in parallel add $F[s] = \sum_{j=0}^{SH-1} P[j]$
            and store or accumulate in $MEM$
        CLS by one $GX$
    end for
    return void
end procedure
```

Figure 3.10: Algorithm for computing the 1-D linear convolution between the signal GIN and the preloaded row or column kernel HX. The output is stored in MEM = MEM_TMP for rows, or accumulated in MEM = MEM_OUT for columns. SG is the final size of the convolved signal, SH is the size of the current kernel and $x = 0, \ldots, SG - SH$.

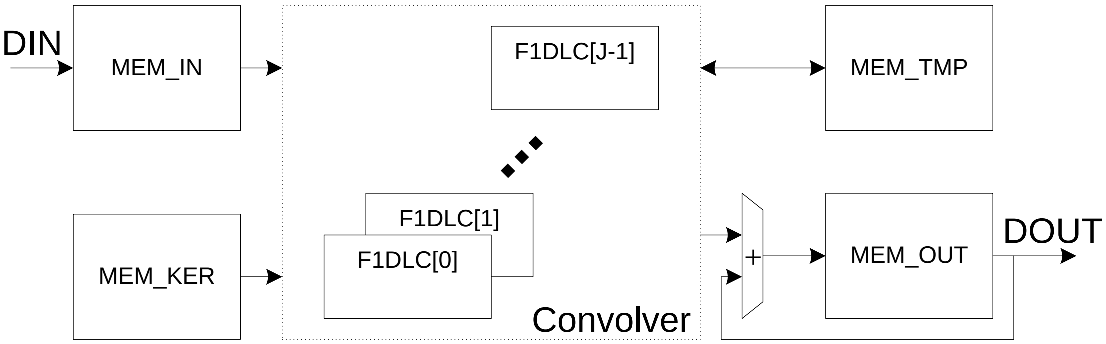


Figure 3.11: S2DLCLU System (top level diagram). Bus width is for maximum accuracy. DANIEL must provide the final version with more detail.

resource usage $O(N)$ with a running time of $O(N^2)$, $J = P2 + Q2 - 1$ gives the fastest running time $O(N)$ with resource usage $O(N^2)$. Refer to Tables 3.2 and 3.3 for detailed resource usage.

```
for $q = 0$ to $r - 1$ do
    load $HX[0 : Q2 - 1] = h_{Rq}(j)$
    for $p = 0$ to $L_R - 1$ do
        in $\|$ F1DLC($g_{pJ+a}(j)$, $P2 + Q2 - 1$, $Q2$, MEM_TMP)
            $a = 0, \ldots, J - 1$
    end for
    load $HX[0 : Q1 - 1] = h_{Cq}(j)$
    for $p = 0$ to $L_C - 1$ do
        in $\|$ F1DLC($g'_{pJ+a}(i)$, $P1 + Q1 - 1$, $Q1$, MEM_OUT)
            $a = 0, \ldots, J - 1$
    end for
end for
```

Figure 3.12: Algorithm for computing the 2-D linear convolution between the image $g(i, j)$ and the non-separable kernel $h(i, j)$ decomposed into r separable kernels $h_R(i, j)$ for rows and $h_C(i, j)$ for columns. $g'(i, j)$ holds the results of the row-convolution in `MEM_TMP`. The output is stored in `MEM_OUT`.

3.3.5 Overlap and Add for larger images

In a limited resources system, the SF2DLC or S2DLCLU systems that can compute the linear convolution for $P_1 \times P_2$ image sizes with $Q_1 \times Q_2$ filter size, using the overlap and add property for 2-D linear convolution. For this purpose, the original input image or kernel or both are divided in smaller blocks and they are processed sequentially.

For overlap-and-add, the original image is subdivided into U smaller blocks of size $P_1 \times P_2$. The each block is processed serially through the SF2DLC or S2DLCLU systems. Thus, the total running time is the product of U multiplied by the processing time for each block. In terms of resources, there is no need of significant additional resources for the convolution itself. Here, either SF2DLC or S2DLCLU can perform the overlap and add operation on the output memory. The system is implemented by expanding the FSM to provide the necessary control signals.

3.4 Results

The SF2DLC and S2DLCLU systems are compared in terms of running time and resources against the following previous work: (i) convolution using serial systolic arrays [45] (SerSys), (ii) convolution using scalable and parallel systolic arrays [46] (ScaSys), (iii) convolution using sliding windows [51] (SliWin), and (iv) convolution using the Fast Fourier Transform radix-2 [41] (FFTr2). Distributed arithmetic (DA) solutions are not included because the internal ROM required for the DA operation grows exponentially with the kernel size, making them not suitable for large kernels [52]. For FFTr2, parallelism is achieved by using different numbers of 1-D FFTs.

3.4.1 Experimental setup

Consider the 2-D linear convolution $f(i, j) = g(i, j) * h(i, j)$ where g is of size $P1 \times P2$ and h is non-separable of size $Q1 \times Q2$. The output f is of size $N1 \times N2$ with $N1 = P1 + Q1 - 1$ and $N2 = P2 + Q2 - 1$. To compare the different approaches for large kernels set $P1 = P2 = Q1 = Q2 = P$, $p = \lceil \log_2 P \rceil$. Then the output $f(i, j)$ has a size of $N \times N$, where $N = 2P - 1$, $n = \lceil \log_2 N \rceil$. The image pixels use $B = 8$ bits. Kernel coefficients use $C = 12$ bits. After SFDPRT, the results are stored using 12 bits (not exact). The outputs of the multipliers and the adders also use 12 bits.

In the SF2DLC system, to balance the speed between the SFDPRT and the

F1DCC, set $J = H$, except for $H = N$, where $J = N + 1$ (optimal solution) is used. Note that N is always an odd number, which is fine for SerSys, SliWin and S2DLCLU. However, for SF2DLC, N is restricted to be a prime number. Similarly, in the case of the FFTr2, N needs to be a power of two. We thus compare methods for the case when N is prime and $N + 1 = 2^m$ for some m. For ScaSys, P needs to be a composite number expressed as $P = P_A \times P_B$. Thus, for ScaSys, $P = 2 \times 64, 4 \times 32, 8 \times 16, 16 \times 8, 32 \times 4$. In all cases, it is assumed that the input data is provided at the required rate.

3.4.2 Running time

For SF2DLC, the slowest case with the lowest resources occurs when $J = H = 2$. For the fastest case, $J = N+1$ and the system is based on the FDPRT and iFDPRT (instead of the SF2DLC). For the S2DLCLU, the slowest case occurs when $J = 1$ and the fastest for $J = N$. For all cases, the rank of h is taken as $r = 2, N$. Note that for $J = H = 2$ and N odd, $\lceil N/2 \rceil = \lceil (N+1)/2 \rceil = (N+1)/2$

For the FFTr2, [41] does not provide detailed running time for the complete convolution. Here, it is assumed that the point-to-point multiplication can be done in parallel with the FFT computation (i.e. assume 0 clock cycles for that). Also, for the latency of the FFTr2, $4N$ (load and output for rows and columns) is added.

Table 3.4 shows the running time as a function of N. In Figs. 3.13 and 3.14, it is used $N = 3, \ldots, 255$ and show the running time for different cases. For S2DLCLU, $J = 1$ with full rank represents the serial solution of the convolution problem in the spatial domain with time complexity of $O(N^3)$. The FFTr2 with $D = 1$ is also a serial solution but lowers the time complexity to $O(N^2 \log_2 N)$ in the frequency domain. In terms of time complexity, the non-scalable solutions SliWin and SerSys along with the scalable solutions ScaSys and SF2DLC at their lowest speed and the S2DLCLU at two modes: $J = 1$, $r = 2$ and $J = N$ require $O(N^2)$ cycles. The scalable SF2DLC, S2DLCLU ($r = 2$) and Scasys have fastest speeds of $O(N)$ clock cycles and the FFTr2 with $D = N$ (impossible to implement) at $O(N \log_2 N)$ cycles.

The S2DLCLU solutions offer good performance for non-separable kernels with low rank. Then, as the rank increases, the SF2DLC becomes a better choice. For non-separable, full-rank kernels and cross-correlation applications, the SF2DLC is definitely a better choice. To achieve these speeds by alternatively methods, we will need prohibitively large amounts of resources as demonstrated for ScaSys ($P_B = 4$), SF2DLC ($J = N + 1$ and FDPRT) and FFTr2 ($D = N$). Even with prohibitively large resources, the proposed S2DLCU is still a better choice. Furthermore, in terms of speed only, SerSys and SliWin produce implementations that fall between the faster and slower realizations of the proposed, scalable implementations.

Due to the fact that there are much more primes than powers of 2 within a given interval, we have much more size choices for the proposed scalable DPRT implementations. Fro example, the FFTr2 can only be implemented for $N = 4, \ldots, 256$, only 7 compared to 53 available sizes for SF2DLC and 127 for S2DLCLU. For example, for a convolved image of size 130, the FFTr2 needs to zero-pad to the next power of two for $N = 256$ which wastes a lot of computational resources [53]. In contrast, the

Table 3.4: Running time for a 2-D linear convolution between an image $g(i,j)$ and a large non-separable kernel $h(i,j)$ with rank r, both of size $P \times P$. The convolved result $f(i,j)$ has a size of $N \times N$, where $N = 2P - 1$, $n = \lceil \log_2 N \rceil$ and $p = \lceil \log_2 P \rceil$. For ScaSys P needs to be a composite number $P = P_A \times P_B$. For FFTr2, $D = 1, \ldots, N$ represents the number of 1-D FFT units running in parallel.

Method	Running time (in clock cycles)
SF2DLC (proposed) $J = H = 2$	$(N+1)(3N+13)/2 + N + 2n + 19$
SF2DLC (proposed) $J = N+1$,FDPRT	$6N + 4n + 17$
S2DLCLU (proposed) $J = 1$	$r \times ((N+1)(N+P)) + p + 1$
S2DLCLU (proposed) $J = N$	$r \times (3N+P) + p + 1$
SerSys [45]	$N^2 + 2P - 2$
ScaSys [46]	$\lceil N^2/P_A \rceil + 2P_A + P_B + \lceil \log_2(P \times P_A) \rceil$
SliWin [51]	$N \times P + N^2 + 2\lceil \log_2 P \rceil + 1$
FFTr2 [41]	$(N^2 \log_2 N)/D + 4N$

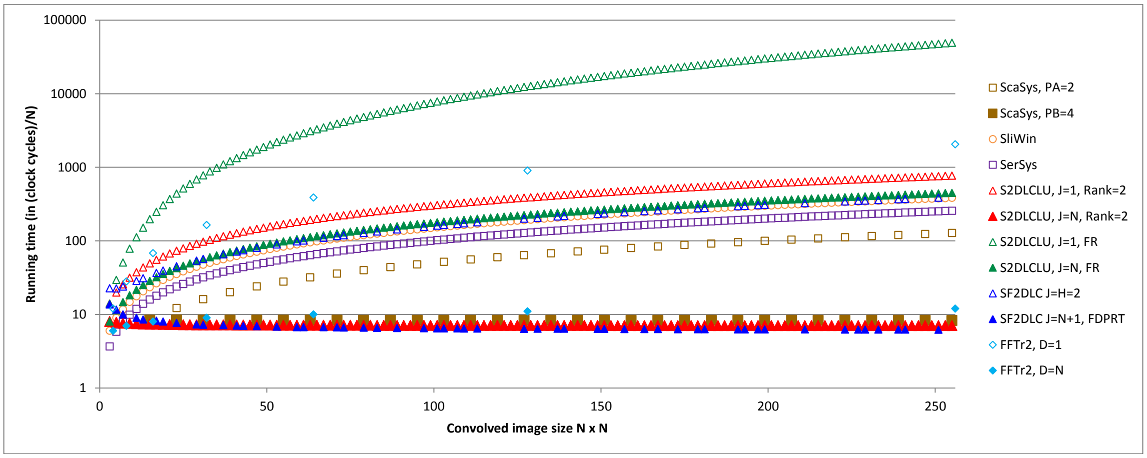


Figure 3.13: Running time in clock cycles (normalized by image size N) versus convolved image size for all methods.

scalable approach works with $N = 131$ which is the next prime number.

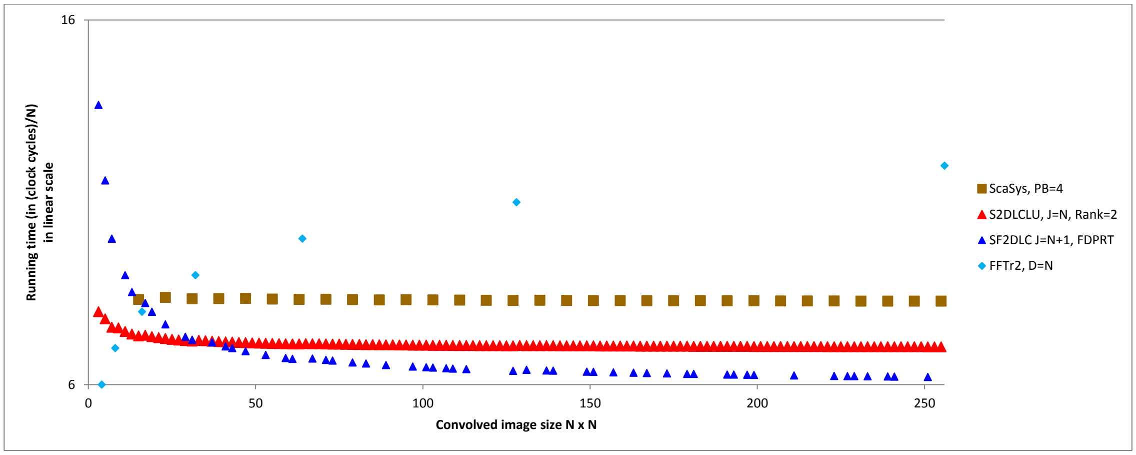


Figure 3.14: Running time in clock cycles (normalized by image size N) versus convolved image size for the fastest methods.

3.4.3 Pareto comparisons

This section provides comparisons of the amounts of resources and running times associated with the different methods. For the FFT implementations, there is a requirement for expensive floating point hardware that grow very large as compared to what is needed for fixed-point implementations. For FFT implementations, the comparisons will be made in terms of required FPGA resources.

The majority of the required resources can be summarized in terms of the numbers of: 1-bit flip-flops, 1-bit additions (full-adders), Multipliers and SRAM. Other resources, such as the Finite State Machine, I/O with other systems and ancillary logic will be accounted for in the next section after the target technology is selected. For the running time, the results from Sec. 3.4.2 are used. The actual running time (seconds) is computed in the next section after the target technology is selected and the frequency of the system is set.

For resource usage, refer to Table 3.5. For the SF2DLC using the SFDPRT and iSFDPRT, only two memories are needed. The two memories are reused as input and output memory for each stage. Also, for the FDPRT and iFDPRT, no memories are needed since the internal registers act as a memory. For the S2DLCLU, the resources do not change for different kernel ranks.

The Pareto front is displayed in terms of resources and running times. The Pareto plots in Figs. 3.15,3.16,3.17 show comparisons in terms of 1-bit FlipFlops, 1-bit Additions, and Multipliers. Memory comparisons are given in in Table 3.6.

It is clear that our proposed systems are Pareto-optimal among all methods. The proposed systems offer a better trade-off between running time and resources. SliWin, ScaSys and SerSys are not in the Pareto front. Among our proposed solutions, S2DLCLU dominates the Pareto front for low rank kernels. However, as noted earlier, as the rank increases, SF2DLU becomes a better solution. For high-speed, the SF2DLC using the FDPRT is the best choice. In terms of memory usage, our methods use more memory, but still grow as $O(N^2)$ which is not a limitation for current technologies. Also, our proposed systems and the SliWin have the advantage that the kernel can be changed in running time.

3.5 Conclusions

This chapter describes two scalable and fast systems (SF2DLC and S2DLCLU) for computing 2-D Linear convolutions with relatively large non-separable kernels. When the non-separable kernel rank is low, the S2DLCLU works best. The chapter also

Table 3.5: Resource usage for a 2-D linear convolution between an image $g(i,j)$ and a large non-separable kernel $h(i,j)$, both of size $P \times P$. The convolved result $f(i,j)$ has a size of $N \times N$, where $N = 2P - 1$, $n = \lceil \log_2 N \rceil$ and $p = \lceil \log_2 P \rceil$. For ScaSys P needs to be a composite number $P = P_A \times P_B$. Define $\mathtt{A_{ffb}}\,(a,b)$ to be number of required flip-flops inside the a-operand of b bits adder tree including input buffers, $\mathtt{A_{ff}}\,()$ without input buffers and $\mathtt{A_{FA}}\,()$ to be the number of 1-bit additions, all grow linearly with respect to N and can be computed using the algorithm given in the appendix (Fig. C.1).

Method	FlipFlops (1-bit)	1-bit Addtitions	Multipliers	Memory
SF2DLC (proposed) (using SFDPRT and iSFDPRT)	$J(36N + \mathtt{A_{ffb}}(N,12))$ $+N(8H + \mathtt{A_{ff}}(H,8))$ $+12N(H+3) + (N+1)\mathtt{A_{ff}}(H,12)$	$J\mathtt{A_{FA}}(N,12)$ $+N\mathtt{A_{FA}}(H,8) + 12N$ $+(N+1)\mathtt{A_{FA}}(H,12) + 2N(12+n)$	$J \times N$	$24N(N+1)$ Ker: $12N(N+1)$
SF2DLC (proposed) (using FDPRT iFDPRT, J=N+1)	$(N+1)(36N + \mathtt{A_{ffb}}(N,12))$ $+N(8N + \mathtt{A_{ff}}(N,8))$ $+12N^2 + (N+1)\mathtt{A_{ff}}(N,12) + N(12+n)$	$J\mathtt{A_{FA}}(N,12)$ $+N\mathtt{A_{FA}}(N,8)$ $+(N+1)\mathtt{A_{FA}}(N,12) + N(12+n)$	$J \times N$	ker: $12N(N+1)$
S2DLCLU (proposed)	$J(36P + \mathtt{A_{ffb}}(P,12))$	$J(\mathtt{A_{FA}}(P,12) + 12)$	$J \times P$	$8P^2 + 12N(N+P)$ Ker: $24P^2$
SerSys [45]	$4P^3 + 34P^2 - 10P - 12$	$12P(P+1)$	P^2	Ker: $12P^2$
ScaSys [46]	$P_A(20P^2 + \mathtt{A_{ffb}}(P_AP,12)) + 8P(P_A^2 + P_A - 1)$	$P_A(12P^2 + \mathtt{A_{FA}}(P_AP,12))$	P_AP^2	Ker: $12P_AP^2$
SliWin [51]	$20P^2 + \mathtt{A_{ffb}}(P^2,12)$	$\mathtt{A_{FA}}(P^2,12)$	P^2	$8PN + 8P^2 + 12N^2$

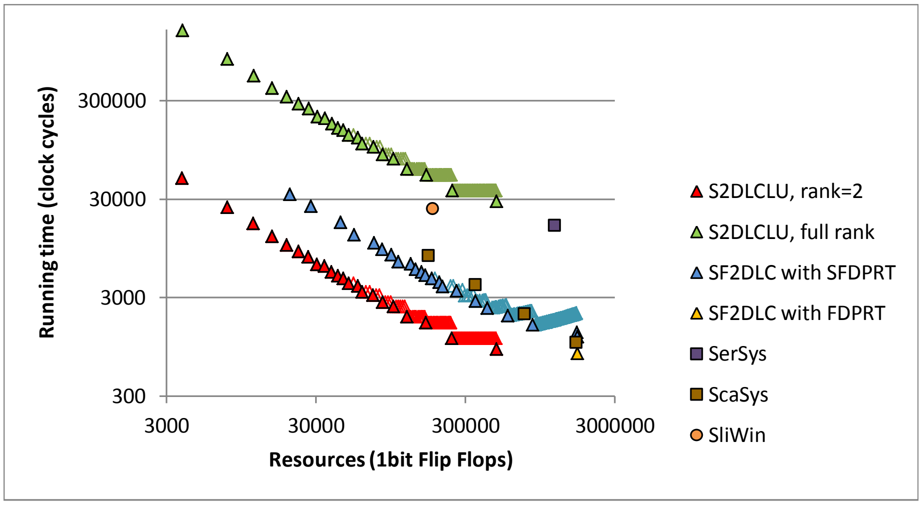


Figure 3.15: Resources (1-bit FlipFlops) vs Running time.

presents a novel parallel memory access system (by row/columns) that uses standard SRAMs to provide high speed transfers and avoid transpositions.

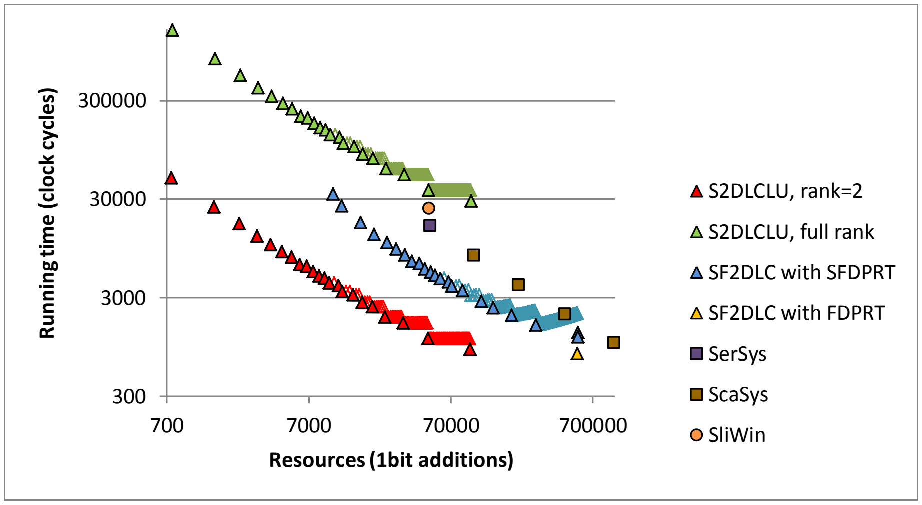


Figure 3.16: Resources (1-bit Additions) vs Running time.

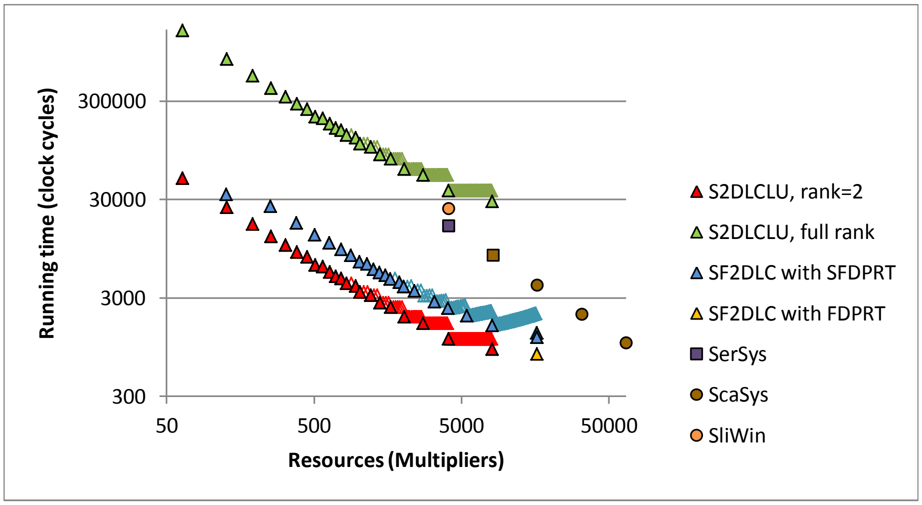


Figure 3.17: Resources (Multipliers) vs Running time.

Table 3.6: Memory usage for a 2-D linear convolution between an image $g(i, j)$ and a large non-separable kernel $h(i, j)$ both of size 64×64. For ScaSys $P_A = 2, 4, 8, 16$.

Method	Memory (bits)
SF2DLC (proposed) $J = H$	585216
SF2DLC (proposed) $J = N + 1$, FDPRT	195072
S2DLCLU (proposed)	422156
SerSys [45]	49152
ScaSys [46]	49152$\times P_A$
SliWin [51]	291340

Chapter 4

Discrete Periodic Radon Transform implementation on GPUs and multi-core CPUs

Graphics Processing Units (GPUs) have been established as important alternatives to general purpose microprocessors for performing large/complex computations. Real-time image processing applications can significantly benefit from the hardware resources available on GPUs. Similarly, real-time image processing applications can also benefit from the emergence of multi-core CPUs.

Current algorithms to compute the forward and inverse DPRT are designed for serial implementations [16],[18]. There are many I/O issues associated with parallelizing these serial implementations. Instead, the proposed SFDPRT and iSFDPRT presented in Chapter 2 avoid I/O issues by removing address calculations. Unfortunately, such a direct approach is not possible on modern CPU/GPUs architectures. Instead, this Chapter introduces a new set of algorithms that are specifically targeted towards CPU/GPU implementations. As before, the ultimate goal is to minimize the I/O bottleneck due to memory accesses.

In what follows, an architecture overview for CPUs and GPUs including an analysis of the I/O for memory accesses is presented in Section 4.1. The proposed parallel algorithms for implementing the forward and inverse DPRT for CPU and GPU architectures are presented in Section 4.2. The implementation of the proposed algorithms on a Xeon processor (CPU) and GM204 processor (GPU) are described in Section 4.3. Finally, results and conclusions are given in Section 4.4.

4.1 Architecture overview for multi-core CPUs and GPUs

A typical architecture for parallel processing using CPUs and Graphic Processing Units (GPUs) is shown in Fig. 4.1. A general description of the main components given next.

The HOST system: The system consists of the Central Processing Unit (CPU) using a Von-Neumann / Harvard architecture with M_C cores, with a standard memory hierarchy (Cache L1/L2/L3, Main Memory and Storage memory) and a Peripheral Component Interconnect bus (PCI). A dispatch unit can provide different instructions to the M_C cores in parallel (MIMD). Each core has its own own set of registers and local Fast SRAM (Cache L1/L2). The L3 cache is shared across cores. Inside each core, there is an Arithmetic Logic Unit that can perform fixed-point or floating-point multiplication/addition operations in one clock cycle.

The DEVICE system (the GPU): The system consists of M_P Multiprocessors (MP), connected with a shared Cache and a Device memory. Inside each MP, there are N_P processors (cores), each one with their own set of registers, a local Fast SRAM (shared with all the processors inside the MP) and an Instruction unit capable to dispatch, in parallel, the same instruction to the N_P cores. Inside each core, there is an Arithmetic Logic Unit, capable to perform a simple multiplication/addition in one clock cycle. The operation can be fixed-point or floating point.

In what follows, let f_D denote the clock frequency for each core. Also, in terms of timing operations on the DEVICE, consider:

- Arithmetic operations: Each core is capable of performing one basic operation per clock cycle.
- Load/Store operations: The following cases need to be considered separately:
    1. Operations between registers require one clock cycle.
    2. Fast SRAM access requires T_S cycles (a few clock cycles).
    3. Cache access requires T_C cycles (tens of clocks cycles).
    4. Cache misses cost T_M cycles (hundreds of clock cycles).

For fast implementations, small images can be loaded in the fast SRAM of a GPU. Unfortunately, the fast SRAM is not shared along all the MPs. Instead, fast

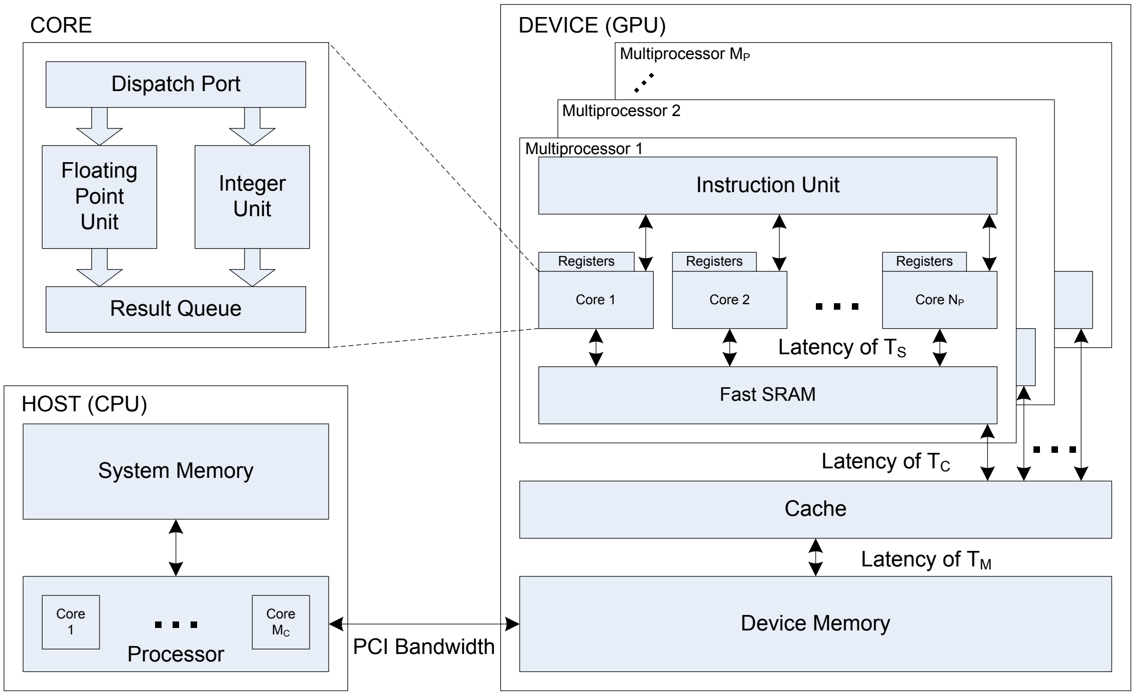


Figure 4.1: Top level block diagram of the CPU and GPU architecture. The block on the lower-left represents the **HOST** system (the CPU). The block on the right represents the **DEVICE** where all the computations are performed (the GPU). Top-left shows the detail of one **CORE**.

SRAM is shared by the cores associated with each MP. On the other hand, cache memory is shared along all MPs. For the HOST, consider the L1/L2 cache of the HOST as being analogous to the fast SRAM on the DEVICE.

4.2 Parallel Algorithms for computing the forward and inverse DPRT

When computing the DPRT $R(m, d)$ of an $N \times N$ image $f(i, j)$ (Eq. (3.4)), the process involves computing $N-1$ additions of N rays per prime direction, for a total

```
for $m = 0$ to $N - 1$ do
    for $d = 0$ to $N - 1$ do
        $sum = f(0, d)$
        for $i = 1$ to $N - 1$ do
            $sum = sum + f(i, \langle d + m \times i \rangle_N)$
        end for
        $R(m, d) = sum$
    end for
end for
for $d = 0$ to $N - 1$ do
    $sum = f(d, 0)$
    for $j = 1$ to $N - 1$ do
        $sum = sum + f(d, j)$
    end for
    $R(N, d) = sum$
end for
```

Figure 4.2: Serial algorithm for computing the forward Discrete Periodic Radon Transform $R(m, d)$ of the image $f(i, j)$ of size $N \times N$.

of $N + 1$ prime directions. The entire DPRT requires $(N - 1)N(N + 1)$ additions. For comparison, a serial algorithm based on [16] to compute the DPRT is presented in Fig. 4.2.

Similarly, when computing the inverse DPRT (iDPRT) $f(i, j)$ of an $(N + 1) \times N$ radon space $R(m, d)$, the process involves computing $N - 1$ additions of N rays per prime direction, for a total of N prime directions. Then, the entire iDPRT requires $(N - 1)N^2$ additions. Additionally, each output pixel in the row i requires two extra additions and a division (terms $-S$, $R(N, i)$ and divide by N from Eq. (3.5) respectively). For the baseline case, a serial algorithm based on [16] is presented in Fig. 4.3.

For the serial implementations, the number of required cycles is much higher than

```
$S = \sum_{d=0}^{N-1} R(m,d)$, with $m = 0$
for $i = 0$ to $N-1$ do
    for $j = 0$ to $N-1$ do
        $sum = R(0,j)$
        for $m = 1$ to $N-1$ do
            $sum = sum + R(m, \langle j - m \times i \rangle_N)$
        end for
        $f(i,j) = (sum - S + R(N,i))/N$
    end for
end for
```

Figure 4.3: Serial algorithm for computing the inverse Discrete Periodic Radon Transform $f(i,j)$ of the radon space $R(m,d)$ of size $(N+1) \times N$.

the number of additions because of:

1. The overhead in the loops (3 levels).

2. The computation of the address of the pixel to be read from memory.

3. The memory latency to load the pixel to be added. The latency can be in the order of 1, T_S, T_C, T_M clock cycles, which depends on the position of the pixel in the memory hierarchy.

4. Storing in memory the output pixel (similar to (3) but for writing).

To parallelize the DPRT, consider a system with p processors where each processor requires $(N-1)N(N+1)/p$ additions. Ideally, this will give a speedup factor of p. The same speedup is expected for the iDPRT.

Now, to develop a parallel algorithm for computing the DPRT, there are different scenarios on how to parallelize the DPRT and iDPRT depending on the number of p available processors. Recall that the DPRT needs to compute $(N+1)N$ rays. Thus, on a system with p processors:

1. If $(N+1)N > p$, each processor will be assigned to compute $\lceil (N+1)N/p \rceil$ rays and possibly 1 less for some cores.

2. If $(N+1)N \leq p$, only $N(N+1)$ processors are needed where each one computes one ray.

For the iDPRT. only N^2 rays need to be computed as given by:

1. If $N^2 > p$, each processor will be assigned to compute $\lceil N^2/p \rceil$ rays and possibly 1 less for some cores.

2. If $N^2 \leq p$, only N^2 processors are needed where each one computes one ray.

However, in both cases, the ideal speedup of p is not achievable with current parallel architectures because of:

1. The overhead of launching, synchronizing and terminating the parallel processing. Fortunately, this is of constant cost that is independent of N.

2. The need for concurrent reads/writes at block levels in the memory hierarchy.

The development of a solution to the problem of performing synchronized and concurrent reads/writes within the memory hierarchy is the primary contribution of this chapter.

4.2.1 Analysis of the DPRT and iDPRT properties to parallelize the processing

The development of parallel algorithms relies on the following:

- Pixel usage: For each prime direction, each pixel is used only one time on the computation of the N rays. Therefore, for each prime direction all the pixels are needed. This implies that trying to partition the image in smaller blocks for partial processing is not optimal.

- Address calculation: According to the radon equation (Eq. (3.4)), when computing the address of the pixel to be added, the pattern of memory access for each pixel changes per prime direction. Therefore, trying to speed up the memory access by fetching blocks of neighbors will not work for all prime directions.

- When adding pixels along one ray, the memory distance between pixels is fixed. Therefore, the computation of the addresses of the pixels can be simplified to just adding an offset and a modulo operation (due to the periodicity of the transform).

- Word size: Because of the additions, the word size increases from B (for B bits per pixel) to $B+\lceil\log_2 N\rceil$ for the DPRT outputs, and up to $B+2\lceil\log_2 N\rceil$ before normalization of the iDPRT. Then, word sizes need to be selected according to the image size for full precision.

- When $N^2 \leq p$, the system can assign each ray computation to a different processor, which leads to a linear running time of $O(N)$. When $N^2 > p$, there are not enough processors for each ray. Thus, the theoretical running time increases from linear (at $p = N^2$) to quadratic (at $p = N$). Beyond this case ($p = N$ to $p < N$ to $p \ll N$), running time starts to move from quadratic to cubic order. Thus, as N increases, computational complexity is of order that is: linear ($N^2 \leq p$), moves to linear-quadratic ($N^2 > p > N$), and then becomes quadratic-cubic (from $N = p$ to $N \gg p$).

4.2.2 Parallel DPRT and iDPRT on a multi-core CPU system

On current architectures for the HOST, a set of cores can also be used for parallel processing. The multi-core CPU of the HOST is characterized as Multiple Instruction, Multiple Data (MIMD) [54]. Each core has a separate instruction and data access to a (shared or distributed) program and data memory. In each step, each core loads a separate instruction and a separate data element, applies the instruction to the data element, and stores a possible result back into the memory. The processing elements work asynchronously and do not communicate with each other.

Let M_C be the number of cores available on the HOST. Then, this model is suitable to partition the DPRT serial algorithm into a set of M_C threads, each one processing asynchronously a set of prime directions $\lceil (N+1)/M_C \rceil$. In some architectures, cores can process more than one thread. In this case, instead of M_C, the total number of parallel cores is given by (Number of hardware cores) × (number of parallel threads per core).

```
Partition the set of $N + 1$ prime directions into $M_C$ sets of consecutive prime directions
Launch $M_C$ threads, assing each partitioned set to each thread to compute $R(m, d)$
Wait for threads to finish
```

Figure 4.4: Main parallel algorithm for computing the forward Discrete Periodic Radon Transform $R(m, d)$ of the image $f(i, j)$ of size $N \times N$ on a CPU with M_C cores.

```
procedure $fDPRT\_HOST\_Kernel(dirIni, dirEnd)$
    if $dirEnd = N$ then
        for $d = 0$ to $N - 1$ do
            $sum = f(d, 0)$
            for $j = 1$ to $N - 1$ do
                $sum = sum + f(d, j)$
            end for
            $R(N, d) = sum$
        end for
        $dirEnd = dirEnd - 1$
    end if
    for $m = dirIni$ to $dirEnd$ do
        for $d = 0$ to $N - 1$ do
            $sum = f(0, d)$
            for $i = 1$ to $N - 1$ do
                $sum = sum + f(i, \langle d + m \times i \rangle_N)$
            end for
            $R(m, d) = sum$
        end for
    end for
end procedure
```

Figure 4.5: Kernel algorithm for each core on the HOST to compute one set of prime directions of the Discrete Periodic Radon Transform $R(m, d)$ of the image $f(i, j)$ of size $N \times N$. Consecutive prime directions $dirIni$ through $dirEnd$ are computed.

The proposed parallel algorithm to compute $R(m, d)$, the DPRT of an $N \times N$ image f, using a HOST with M_C cores is presented in fig. 4.4. Furthermore, the kernel for each core is presented in fig 4.5.

```
Partition the set of $N$ prime directions into $M_C$ sets of consecutive prime directions
Compute $S = \sum_{d=0}^{N-1} R(m, d)$, with $m = 0$
Launch $M_C$ threads, assing each partitioned set to each thread to compute $f(i, j)$
Wait for threads to finish
```

Figure 4.6: Main parallel algorithm for computing the inverse Discrete Periodic Radon Transform $f(i, j)$ of the radon space $R(m, d)$ of size $(N + 1) \times N$ on a CPU with M_C cores.

```
procedure $iDPRT\_HOST\_Kernel(dirIni, dirEnd)$
    for $i = dirIni$ to $dirEnd$ do
        for $j = 0$ to $N - 1$ do
            $sum = R(0, j)$
            for $m = 1$ to $N - 1$ do
                $sum = sum + R(m, \langle j + m \times i \rangle_N)$
            end for
            $f(i, j) = (sum - S + R(N, i))/N$
        end for
    end for
end procedure
```

Figure 4.7: Kernel algorithm for each core on the HOST to compute one set of prime directions of the inverse Discrete Periodic Radon Transform $f(i, j)$ of the radon space $R(m, d)$ of size $(N + 1) \times N$. Consecutive prime directions $dirIni$ through $dirEnd$ are computed.

Similarly, the proposed parallel algorithm to compute $f(i, j)$, the iDPRT of an $(N + 1) \times N$ radon space $R(m, d)$, using a HOST with M_C cores is presented in fig. 4.6. Furthermore, the kernel for each core is presented in fig 4.7.

4.2.3 Parallel DPRT and iDPRT on a GPU

Current DEVICEs (GPUs) provide several cores for parallel processing. GPUs consist of a set of M_P Multiprocessors where each one can run a set of threads independently of the others (like a MIMD architecture). However, inside each MP, there is a set of N_P cores that work according to the Single Instruction, Multiple Data (SIMD) model [54]. Alternatively, the programming model is also called Single Instruction Multiple Threads (SIMT) [55]. Each core has a private access to a shared memory. On the other hand, there is only one program memory from which a special control processor fetches and dispatches instructions. For each step, each core obtains from the control processor the same instruction and loads a separate data element through its private data access on which the instruction is performed. Thus, the instruction is synchronously applied in parallel by all cores to different data elements. For applications with a significant degree of data parallelism the SIMD approach can be very efficient [56].

The increased complexity of the architecture requires a more complex algorithm to fully exploit the parallelism of the DEVICE. The proposed DPRT algorithm on a GPU is derived as follows (the inverse is described later):

1. Consider a GPU with M_P MPs based on a MIMD architecture model. Each MP has N_P cores with fast SRAM shared among local cores based on the SIMD model. For all MPs, there is a shared Cache and a shared Device memory.

2. At the top level, DPRT computation is subdivided equally among the MP multi-processors (similar to the parallel algorithm on the HOST) by having each MP process a set of prime directions.

3. Since each prime direction generates a full row of the radon space, each MP will generate a set of P_d rows of the radon space where $P_d = \lceil (N+1)/M_P \rceil$ for all except the last MP that will process the remaining directions. Each

MP will load on its own fast SRAM a copy of the complete input image $f(i, j)$ (possibly not the whole image at the same time).

4. Inside the MP, the SIMD model is used to process P_d prime directions. For each prime direction, N_P rays are processed in parallel until the completion of computations for N rays. Then, the next prime direction is processed and so on until the P_d prime directions are computed. Thus, per MP, the system will launch $P_d \times N$ threads to be processed by N_P cores.

5. After a core computes one ray, the result is directly stored in the device memory. There is no need to hold the results in the fast SRAM because there is no further use of that result and there is no chance of concurrent writes.

6. During the process, the image $f(i, j)$ is assumed to be on the device memory and the result is stored in the same memory.

The proposed parallel algorithm using a DEVICE with $M_P \times N_P$ cores is presented in Fig. 4.8. The kernel for each core is presented in fig 4.9.

For the iDPRT the process is very similar. The differences are: (i) iDPRT requires only N prime directions (the horizontal one is not needed), and (ii) the final output per ray needs two additional additions and one division. The proposed parallel algorithm is presented in Fig. 4.10 and the kernel for each core is presented in Fig. 4.11.

```
▷ STEP1: Partition the (N+1) prime directions into $M_P$ sets, each set assigned
to a MP $P_i = i$, $i = 0, \ldots, M_P - 1$
$th = \langle N+1 \rangle_{M_P}$
for $i = 0$ to $M_P - 1$ do
    if $(P_i \geq th)$ & $(th > 0)$ then
        $primeSiz = (N + 1 + M_P - 1)/M_P - 1$
        $primeStart_i = th \times (primeSiz + 1) + (P_i - th) \times primeSiz$
    else
        $primeSiz = (N + 1 + M_P - 1)/M_P$
        $primeStart_i = P_i \times primeSiz$
    end if
    $primeEnd_i = primeStart + primeSiz - 1$
end for
▷ STEP2: Launch the threads per MP to compute $R(m, d)$
for $i = 0$ to $M_P - 1$ do
    Launch $(primeEnd_i - primeStart_i + 1) \times N$ threads for each MP $P_i$
        to compute $R(m, d)$. Each thread is indexed by
        $m = primeStart, \ldots, primeEnd$, $d = 0, \ldots, N - 1$
end for
Wait for threads to finish
```

Figure 4.8: Main parallel algorithm for computing the forward Discrete Periodic Radon Transform $R(m, d)$ of the image $f(i, j)$ of size $N \times N$ on a GPU with $M_P \times N_P$ cores.

4.3 Implementation of proposed algorithms on a CPU and GPU processors

In this section, the following implementations are presented:

1. **Serial implementation on the HOST:**: Using only one logical core, a serial implementation using the Algorithms of Fig. 4.2 and Fig. 4.3 for the DPRT and iDPRT respectively are presented. This implementation is used as a baseline to compare other implementations.

```
procedure $fDPRT\_DEVICE\_Kernel(m, d)$
    $sum = 0$
    if $m = N$ then
        for $j = 0$ to $N - 1$ do
            $sum = sum + f(d, j)$
        end for
    else
        for $i = 0$ to $N - 1$ do
            $sum = sum + f(i, \langle d + m \times i \rangle_N)$
        end for
    end if
    $R(m, d) = sum$
end procedure
```

Figure 4.9: Kernel algorithm for each core on the DEVICE to compute one ray of the forward Discrete Periodic Radon Transform $R(m, d)$ of the image $f(i, j)$ of size $N \times N$.

2. **Parallel implementation on the HOST:** Using all logical cores available on the HOST, parallel implementations are developed using the algorithms of Figs. 4.4 and 4.5 for the DPRT and algorithms of Figs. 4.6 and 4.7 for the iDPRT.

3. **Parallel implementation on the DEVICE:** Using a GPU, parallel implementations are developed using the algorithms of Figs. 4.8 and 4.9 for the DPRT and algorithms of Figs. 4.10 and 4.11 for the iDPRT.

The hardware used for the HOST is given by:

- CPU: Intel Xeon CPU E5-2630 v3 @3.2GHz, L1 cache 512K (32KB Instruction cache, 32KB data cache, per core), L2 cache 2MB (256KB per core), L3 cache 20MB (Shared among all cores), 8 cores (16 logical processors via hyperthreading).

```
▷ STEP1: Partition the $N$ prime directions into $M_P$ sets, each set assigned to a
MP $P_i = i$, $i = 0, \ldots, M_P - 1$
$th = \langle N \rangle_{M_P}$
for $i = 0$ to $M_P - 1$ do
    if $(P_i \geq th)$ & $(th > 0)$ then
        $primeSiz = (N + M_P - 1)/M_P - 1$
        $primeStart_i = th \times (primeSiz + 1) + (P_i - th) \times primeSiz$
    else
        $primeSiz = (N + M_P - 1)/M_P$
        $primeStart_i = P_i \times primeSiz$
    end if
    $primeEnd_i = primeStart + primeSiz - 1$
end for
▷ STEP2: compute $S$
$S = \sum_{d=0}^{N-1} R(m, d)$, with $m = 0$
▷ STEP3: Launch the threads per MP to compute $f(i, j)$
for $i = 0$ to $M_P - 1$ do
    Launch $(primeEnd_i - primeStart_i + 1) \times N$ threads for each MP $P_i$
        to compute $f(i, j)$. Each thread is indexed by
        $i = primeStart, \ldots, primeEnd$, $j = 0, \ldots, N - 1$
end for
Wait for threads to finish
```

Figure 4.10: Main parallel algorithm for computing the inverse Discrete Periodic Radon Transform $f(i, j)$ of the radon space $R(m, d)$ of size $(N + 1) \times N$ on a GPU with $M_P \times N_P$ cores.

```
procedure $iDPRT\_DEVICE\_Kernel(i, j)$
    $sum = 0$
    for $m = 0$ to $N - 1$ do
        $sum = sum + R(m, \langle j - m \times i \rangle_N)$
    end for
    $f(i, j) = (sum - S + R(N, i))/N$
end procedure
```

Figure 4.11: Kernel algorithm for each core on the DEVICE to compute one ray of the inverse Discrete Periodic Radon Transform $f(i, j)$ of the radon space $R(m, d)$ of size $(N + 1) \times N$.

- System memory: 64GB (4 x 16GB) 288-Pin DDR4 SDRAM ECC DDR4 2133 (PC4 17000).
- Storage memory: (Primary) Solid State Drive (SSD) Hynix SH920 2.5” 256GB, SCSI device. Secondary: Western Digital RE WD4000FYYZ 4TB 7200 RPM 64MB Cache SATA 6.0Gb/s.
- System bus: PCI Express 3.0, PCI Express configurations x4, 8x, 16x.
- Software: Windows 8.1 Enterprise (64-bit operating system)

The hardware for the DEVICE is a GPU: Nvidia GeForce GTX 980 card (GM204 Maxwell architecture), installed in the PCI Express bus of the HOST, with the following configuration:

- 4 Graphics Processing Clusters (GPCs).
- 16 Streaming Multiprocessors Maxwell (SMM, 4 per GPC).
- 4 Memory controllers (one per GPC), each has 64-bit data width, for a total of 256bit memory bus width.
- 2048 CUDA Cores (128 per SMM) @1367 MHz.
- 1024K 32-bit registers (64K 32-bit registers per SMM)
- L1 Cache: 384KB (24K per SMM, shared by all cores inside a SMM).
- Shared SRAM: 1536KB (96KB per SMM, shared by all cores inside a SMM, only up to 48KB per block of threads).

- L2 Cache: 2048K (512K per Memory controller, shared by all SMMs).

- Device memory (global memory, shared by all SMMs): 4 GB GDDR5, 256bit data width, clock 7010 MHz (effective), bandwidth 224.3GB/s.

- I/O HOST-DEVICE: PCI Bus, type PCI-E 3.0.

System development was based on the following software:

- Microsoft Visual Studio 2013, v12.0.31101.00 Update 4: Integrated Development Environment (IDE) for writing/compiling/executing and debugging programs for the HOST and DEVICE. Compiler for the HOST.

- Nvidia CUDA Toolkit 7.0 (Integrated in the MS VS2010, compiler, libraries, and tools for the DEVICE).

- In all cases, the language used is C.

Since the memory is 1-D, a map from 2-D functions to 1-D memory is needed. Depending on the implementation, row-major or column-major order is needed. Then, for this purpose, for a 2-D image f of size $N_1 \times N_2$, the (i, j) position in the 1-D memory is given by:

- Column-major ordering position $=j * N_1 + i$

- Row-major ordering position $=i * N_2 + j$

4.3.1 Serial implementation of the DPRT and iDPRT on the HOST

The serial implementation directly follows from the algorithms of Figs. 4.2 and 4.3. Column-major order is used. For the iDPRT before starting the computation of the prime directions, the constant S needs to be computed. The complete code is given in App. D.

4.3.2 Parallel implementation of the DPRT and iDPRT on the HOST

To parallelize the DPRT and iDPRT computation on the HOST, set $M_C = 16$ cores (16 logical cores on the Xeon E5-2630, 8 physical cores, each one capable to process 2 threads in parallel). The HOST offers a mix of private and shared address space within the physical CPU (i.e. each physical core on the Xeon E5-2630 has its own cache L1 and L2 and a shared cache L3), and an off-chip System memory. Then a natural programming model for this architecture is a thread model in which all threads have access to shared variables. POSIX threads (also called Pthreads) define a standard for the parallel programming with threads, based on the programming language C. Then, Pthreads combined with the MIMD architecture of the HOST allows us to produce a fast and parallel implementation. Again, column-major order is used. The image is loaded in the system memory and it is available to each thread. The HOST launches $M_C = 16$ threads each one computing a set of prime directions (as described in the algorithms in Figs. 4.4 and 4.6). There are concurrent reads and no concurrent writes. For the iDPRT before starting the computation of the prime directions, the constant S needs to be computed. The complete code is on App. E.

4.3.3 Parallel implementation of the DPRT and iDPRT on the DEVICE

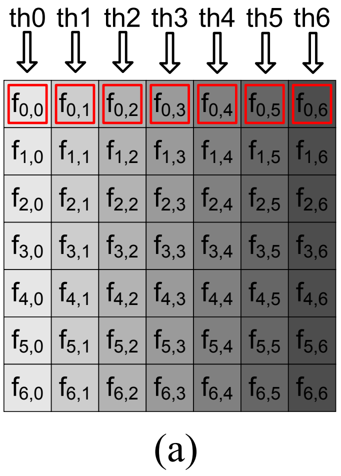
To parallelize the DPRT and iDPRT computation on the HOST, set $T_C = M_P \times N_C$ cores (e.g. 2048 cores on the GTX980). The GPU has the ideal hardware to compute the required additions at a rate of T_C integer additions per clock cycle. However, computing the address of the data to be added and moving the data from the device memory to the core can take several clock cycles. A careful design of an algorithm to minimize the memory access and address computation is required.

General Implementation (DPRT and iDPRT)

As described in 4.1, the transfer of the input image from the device memory to the DEVICE cache and fast SRAM can result in significant processing delays. As a result, it is not possible to efficiently assign a fraction of the input image to one MP. Alternatively, if the complete image can be loaded in the fast SRAM of each MP, a high processing speed will achieved. Unfortunately, current GPUs do not offer a large fast SRAM. For example, the GTX980 has up to 48KB of fast SRAM per block of threads which limits image sizes to $N < (48 \times 1024/4)^{0.5} = 111)$ when using 32-bits per pixel. Thus, an efficient mechanism to load/drop pieces of the input image until the complete image has been processed in one MP is needed.

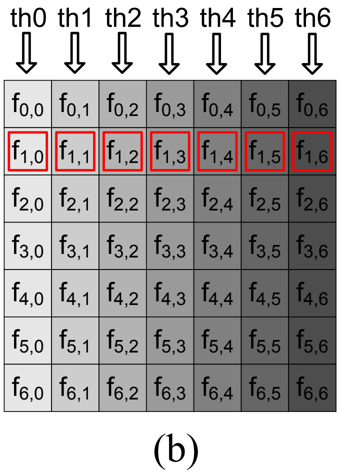
For the DPRT, based on Eq. (3.4), note that each ray of the same prime direction uses exactly one pixel of each row. Thus, if possible, the goal is to align parallel processing so that the threads access the same row of pixels. To illustrate the idea, consider $N = 7$. Then, for the first prime direction ($m = 0$), there are 7 threads running in parallel computing one ray each. In the first step, all threads read the first element of their respective ray (see Fig. 4.12(a)). In the second step, all threads read the second element (see Fig. 4.12(b)), and so on, until the last element (See Fig. 4.12(c)). For each step, a complete row of the image is used. Thus, for the first prime direction, the data must be stored using row-major ordering to accelerate access. Using row-major ordering, the GPU can then move blocks data from device memory to the cores for efficient processing. For example, for the GTX980, one memory access to the device memory transfers 128 bytes. Since each MP has 32 cores, assuming 32 bits per pixel, a single data transfer can move all the data needed for the cores. The same property holds for the rest of the prime directions. For each step, a complete row of the image is accessed as shown in Figs.4.13(a)-(c)).

The results from the parallel computations are also completed synchronously. Thus, block writes are also possible using row-major ordering. On the other hand, note that (non-blocked) concurrent writes are not possible. For address computation, for a fixed prime direction m, pixel offsets are constant. An initial address for the first pixel can be pre-computed before starting ray computations. Then, the constant offset is added for successive pixel addresses. Fig. 4.14 presents the main kernel



th0	th1	th2	th3	th4	th5	th6
$f_{0,0}$	$f_{0,1}$	$f_{0,2}$	$f_{0,3}$	$f_{0,4}$	$f_{0,5}$	$f_{0,6}$
$f_{1,0}$	$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	$f_{1,4}$	$f_{1,5}$	$f_{1,6}$
$f_{2,0}$	$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	$f_{2,4}$	$f_{2,5}$	$f_{2,6}$
$f_{3,0}$	$f_{3,1}$	$f_{3,2}$	$f_{3,3}$	$f_{3,4}$	$f_{3,5}$	$f_{3,6}$
$f_{4,0}$	$f_{4,1}$	$f_{4,2}$	$f_{4,3}$	$f_{4,4}$	$f_{4,5}$	$f_{4,6}$
$f_{5,0}$	$f_{5,1}$	$f_{5,2}$	$f_{5,3}$	$f_{5,4}$	$f_{5,5}$	$f_{5,6}$
$f_{6,0}$	$f_{6,1}$	$f_{6,2}$	$f_{6,3}$	$f_{6,4}$	$f_{6,5}$	$f_{6,6}$

(a) (b) (c)



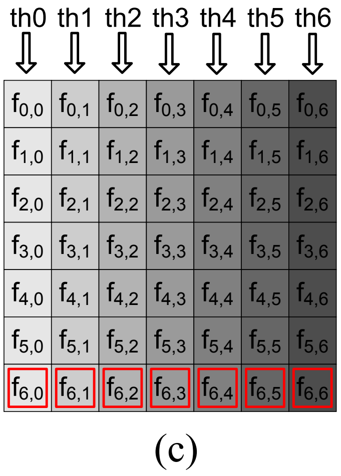
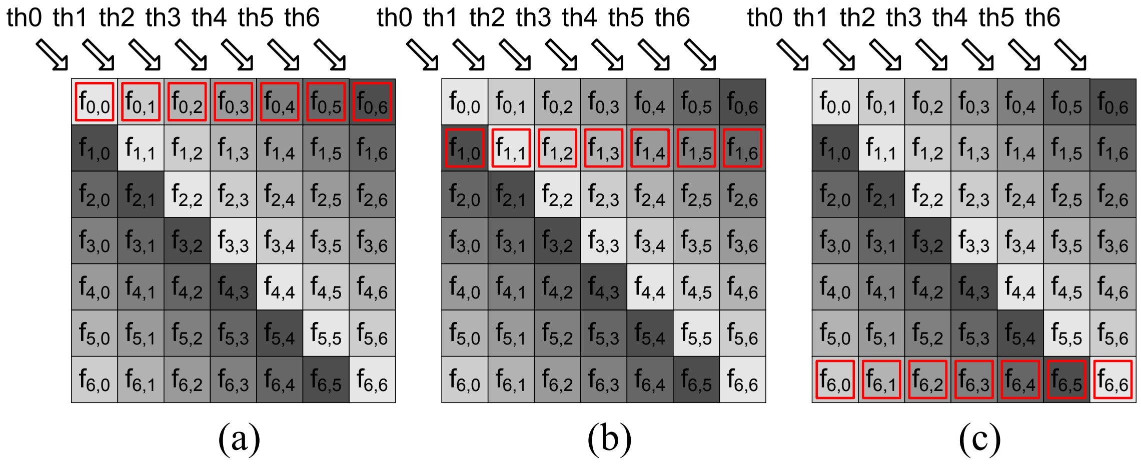


Figure 4.12: Input image of size $N \times N$, $N = 7$. For the prime direction $m = 0$, pixels with the same grayscale level are added to compute one output pixel (radon space), i.e. 7 rays in parallel are computed. (a) 7 threads in parallel start computing 7 rays. Red boxes highlight the first pixel loaded for each thread. (b) Second set of pixels are highlighted. (c) Last set of pixels are highlighted. Assuming the threads are syncronized, note that all threads read the same row of pixels.



th0	th1	th2	th3	th4	th5	th6
$f_{0,0}$	$f_{0,1}$	$f_{0,2}$	$f_{0,3}$	$f_{0,4}$	$f_{0,5}$	$f_{0,6}$
$f_{1,0}$	$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	$f_{1,4}$	$f_{1,5}$	$f_{1,6}$
$f_{2,0}$	$f_{2,1}$	$f_{2,2}$	$f_{2,3}$	$f_{2,4}$	$f_{2,5}$	$f_{2,6}$
$f_{3,0}$	$f_{3,1}$	$f_{3,2}$	$f_{3,3}$	$f_{3,4}$	$f_{3,5}$	$f_{3,6}$
$f_{4,0}$	$f_{4,1}$	$f_{4,2}$	$f_{4,3}$	$f_{4,4}$	$f_{4,5}$	$f_{4,6}$
$f_{5,0}$	$f_{5,1}$	$f_{5,2}$	$f_{5,3}$	$f_{5,4}$	$f_{5,5}$	$f_{5,6}$
$f_{6,0}$	$f_{6,1}$	$f_{6,2}$	$f_{6,3}$	$f_{6,4}$	$f_{6,5}$	$f_{6,6}$

(a) (b) (c)

Figure 4.13: Input image of size $N \times N$, $N = 7$. For the prime direction $m = 1$, pixels with the same grayscale level are added to compute one output pixel (radon space), i.e. 7 rays in parallel are computed. (a) 7 threads in parallel start computing 7 rays. Red boxes highlight the first pixel loaded for each thread. (b) Second set of pixels are highlighted. (c) Last set of pixels are highlighted. Assuming the threads are synchronized, note that all threads read the same row of pixels.

```
procedure $fDPRT\_GPU\_Kernel(radon, img, N, m, d)$
    if $m = N$ then
        $offs = 0$
        $incr = 1$
        $init = d \times N;$
    else
        $offs = d$
        $incr = N$
        $init = 0;$
    end if
    $k = d + m \times N$
    $sum = 0$
    for $i = 0$ to $N - 1$ do
        $sum = sum + img[init + offs]$
        $offs = \langle offs + m \rangle_N$
        $init = init + incr$
    end for
    $radon[k] = sum$
end procedure
```

Figure 4.14: Kernel algorithm for each core on the GPU to compute one ray of the forward Discrete Periodic Radon Transform $R(m, d)$ of the image $f(i, j)$ of size $N \times N$. $R(m, d)$ is mapped to a vector $radon[k]$ and $f(i, j)$ is mapped to a vector $img[k]$, both using row-major order.

(running in one core) for computing one ray d of a prime direction m of the image $f(i, j)$ stored in memory in row-major ordering.

Similar considerations apply for the iDPRT, with the difference being that the number of prime directions is reduced to N. Fig. 4.15 presents the main kernel (running in one core) for computing one ray j of a prime direction i of $R(m, d)$ stored in row-major ordering in $radon(k)$ and storing the results in $img(k)$.

```
**procedure** $iDPRT\_GPU\_Kernel(img, radon, N, S, i, j)$
    $k = j + i \times N$
    $sum = 0$
    $offs = j$
    $init = 0$
    **for** $m = 0$ to $N - 1$ **do**
        $sum = sum + radon[init + offs]$
        $offs = \langle offs - i + N \rangle_N$
        $init = init + N$
    **end for**
    $img[k] = (sum - S + radon[N \times N + i])/N$
**end procedure**
```

Figure 4.15: Kernel algorithm for each core on the GPU to compute one ray of the inverse Discrete Periodic Radon Transform $f(i, j)$ of the radon space $R(m, d)$ of size $(N + 1) \times N$. $R(m, d)$ is mapped to a vector $radon[k]$ and $f(i, j)$ is mapped to a vector $img[k]$, both using row-major order.

Specific Implementation details for GPU: Nvidia GeForce GTX980

This section provides implementation details that are specific to the GPU that was used. Table 4.1 summarizes the technical specifications of the GPU GM204.

The Pixel bit-width for exact computation needs to be set to 32-bit so that there are sufficient bits for all stages of the computation. In terms of bits, the following bitwidths are used: (i) each pixel is assumed to be of $B = 8$ bits, (ii) the DPRT requires $B + \lceil \log_2 N \rceil$, (iii) the inverse DPRT uses up to $B + 2 \lceil \log_2 N \rceil$ before normalization, with a final output of B bits. Arithmetic instructions can use either 8, 16, 32, or 64 bits. Typically, grayscale images or each channel of a color image use 8 bits. Thus, the use of 32-bits allows exact computation for sizes up to $N \times N = 4093 \times 4093$. On the other hand, 16-bits is impractical except for very small image sizes (up to 13×13). Furthermore, the use of 64-bits is also impractical

Table 4.1: Technical specifications for the GPU GM204, compute capability 5.2 (Maxwell Architecture).

Technical specification	Value
Maximum number of threads per block	1024
Warp size	32
Maximum number of resident blocks per multiprocessor	32
Maximum number of resident warps per multiprocessor	64
Maximum number of resident threads per multiprocessor	2048
Number of 32-bit registers per multiprocessor	64K
Maximum number of 32-bit registers per thread block	64K
Maximum number of 32-bit registers per thread	255
Maximum amount of shared memory per multiprocessor	96KB
Maximum amount of shared memory per thread block	48KB
Number of shared memory banks	32
Maximum number of instructions per kernel	512M

since it results in a significant slowdown from 128 additions per clock cycle per MP to 1 addition per clock cycle per MP.

Initially, the $N+1$ prime directions are computed using N threads per prime direction. Thus, there is a total of $(N+1)N$ threads (each one running the *fDPRT_GPU_Kernel*) that are scheduled to be executed on the GPU. From the programmer's point of view, each prime direction is assigned to a block of N threads and each block is assigned to a MP. The partition of the $N + 1$ prime directions into $M_P = 16$ sets is done automatically by the scheduler (the blocks of the grid are enumerated and distributed to MPs with available execution capacity).

By default, the memory hierarchy will move the data from the device memory to the cores using the L2 cache as a buffer for the input image. Recall that the L2 cache memory is shared among all MPs. For each MP in the Maxwell Architecture, there is a choice between the use of fast SRAM (shared memory) or the L1 cache (that needs to be activated). The use of shared memory requires additional coding versus the use of the L1 cache that can be handled automatically. There was no advantage to manually programming the shared memory. Instead, the code uses the L1 cache by compiling the CUDA code using the option **-Xptxas -dlcm=ca**. This compile-time option forces that all reads are cached provided that the input is in read-only mode, as is the case for the input image.

When a MP is given a group of blocks to execute, it partitions them into warps and each warp gets scheduled by a warp scheduler for execution. The way a block is partitioned into warps is always the same; each warp contains 32 threads of consecutive, increasing thread IDs with the first warp containing thread 0. A warp executes one common instruction at a time so full efficiency is realized when all 32 threads of a warp agree on their execution path [55]. For the $fDPRT\_GPU\_Kernel$, the main loop assures all the threads follow the same execution path. An apparent divergence appears when an **if** statement is used to check for the last prime direction. Thus, in the final implementation, a synchronization instruction is issued to ensure that the execution path is properly synchronized after the **if**.

When a warp is scheduled, memory access optimization is also needed. To achieve high bandwidth, either shared memory or cache L1 is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. Let **x** denote the number of addresses that need to be accessed. Any memory read or write request made of the **x** addresses that fall in distinct memory banks can be serviced simultaneously. As a result, the overall bandwidth is **x** times as high as the bandwidth of a single module. Since the warp is 32 threads in parallel, the ideal scenario is to have 32 addresses pointing to 32 different banks. However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary, decreasing throughput by a factor equal to the number of separate memory requests. For the $fDPRT\_GPU\_Kernel$, when computing 32 rays (one warp) all the threads request consecutive 32-bit memory positions and thus avoid bank conflicts. On the other hand, it is possible to have a wrap-around (DPRT periodicity) within those 32 memory accesses, but since N is prime, after the wrap-around, all the 32 pixels will be loaded in different banks.

When a warp stalls (e.g., requiring device memory access), the warp scheduler switches to another warp that is ready to execute. Thus, delays due to memory stalls are minimized. Therefore, overall impact is minimized by having a large number of warps ready for execution. As stated earlier, at launch time, all the rays of all prime directions are scheduled for execution. As a result, each MP will be filled with as many blocks as the MP can handle (e.g., 32 blocks per MP in GTX980). Furthermore, the modulo operation is executed in parallel during a memory access so as to minimize the impact of the main kernel loop. As stated by the GPU manufacturer, the modulo operation maps to around 20 single-cycle assembly instructions which is still much faster than the number of cycles for memory access.

All of the previous considerations also apply for the iDPRT. The two major differences are: (i) N instead of $N + 1$ prime directions, and (ii) the extra two additions and a division at the end of each ray computation. However, these two differences do not change the implementation details.

The source code needed to compute the DPRT and iDPRT using the algorithms of Figs. 4.14 and 4.15 including the necessary considerations is given in App. F.

4.4 Results and Conclusions

In this section, the 6 main algorithms are tested:

- **fSER:** Forward DPRT, serial on the host using one thread (CPU processor, Xeon E5-2630 v3).
- **fCPU:** Forward DPRT, parallel on the host with Pthreads, using 16 threads (CPU processor, Xeon E5-2630 v3).
- **fGPU:** Forward DPRT, parallel on the device (GPU processor, GM204).
- **iSER:** Inverse DPRT, serial the host using one thread (CPU processor, Xeon E5-2630 v3).
- **iCPU:** Inverse DPRT, parallel on the host with Pthreads, using 16 threads (CPU processor, Xeon E5-2630 v3).
- **iGPU:** Inverse DPRT, parallel on the device (GPU processor, GM204).

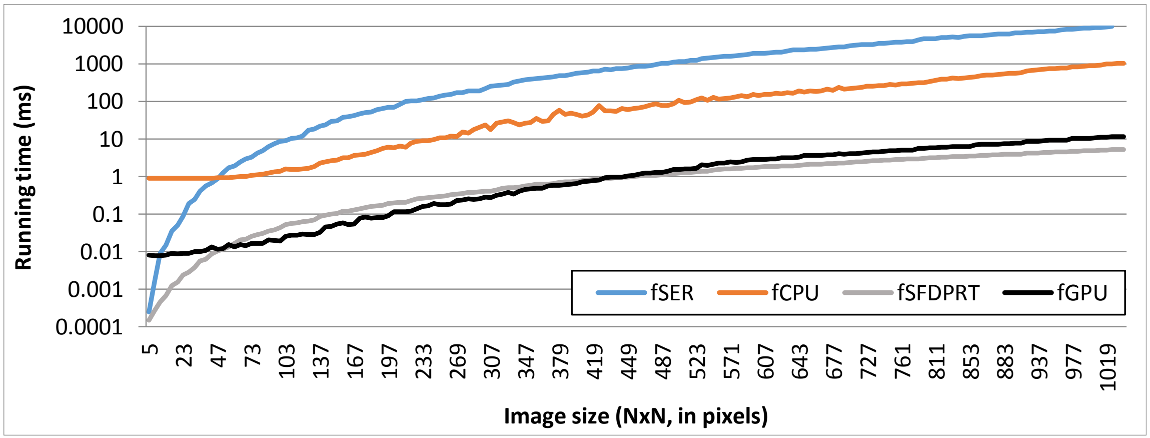


Figure 4.16: Comparative running time for different implementations of the forward DPRT.

To setup the system, forward and inverse DPRT are computed for 170 prime numbers, from 5x5 up to 1021x1021 image sizes. For each case, the input image is filled with random 8-bit integers. Since exact arithmetic is used, all the results have zero error. Additionally, the proposed algorithms are compared against the SFDPRT and iSFDPRT from Chapter 2 with $H = 2$ and $f_{CLK} = 100MHz$.

The results are presented in Figs. 4.16, 4.17, 4.18, and 4.19. Fig. 4.16 shows the running times for the forward implementations. Fig. 4.17 shows the running time for each inverse implementations. Fig 4.18 shows the forward DPRT SpeedUp with respect to the serial implementation. Here, the speedup is defined as the ratio (Forward Parallel Running time) / (Forward Serial Running time). Fig 4.19 shows the inverse DPRT SpeedUp with respect to the serial implementation.

From Fig. 4.16, the serial implementation fSER is the fastest possible for small image sizes. This is because of the typical overhead associated with the parallel implementations. When the image size becomes sufficiently large, the additions

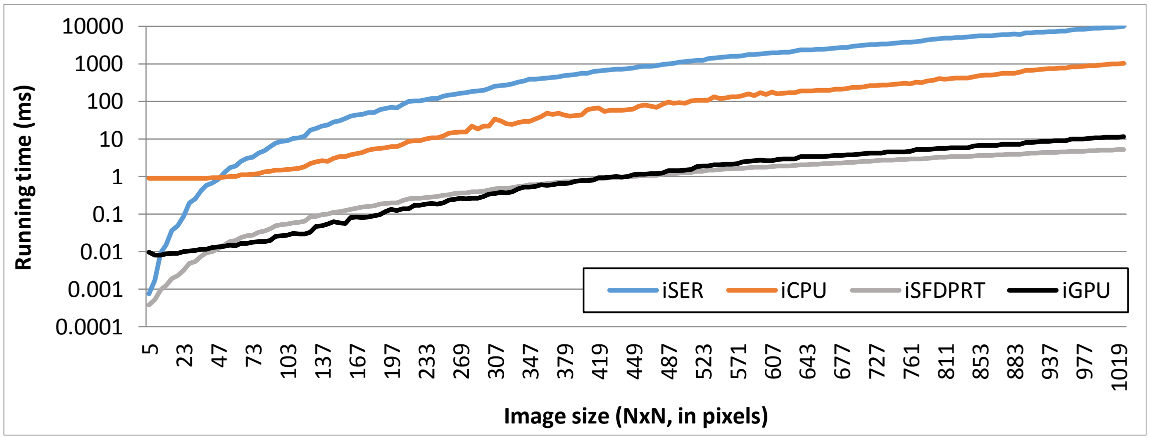


Figure 4.17: Comparative running time for different implementations of the inverse DPRT.

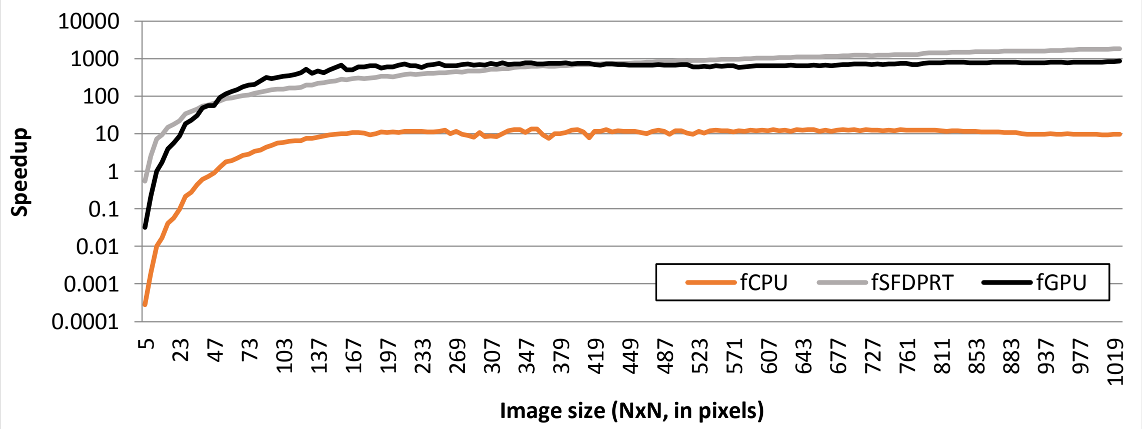


Figure 4.18: Speedup for different implementations of the forward DPRT with respect to the serial implementation (fSER).

dominate the running time. From Fig. 4.18, the fCPU implementation using all the available cores of the CPU gives an speedup of around 10. For the GPU, besides the overhead launching the threads, the processing is noticeable faster for small image sizes because there are enough cores to process all the rays in parallel. As explained earlier, for small image sizes, all of the cores can be used and the L1 cache (or fast SRAM) will not saturate. Beyond $N > 47$, each core starts to process more than

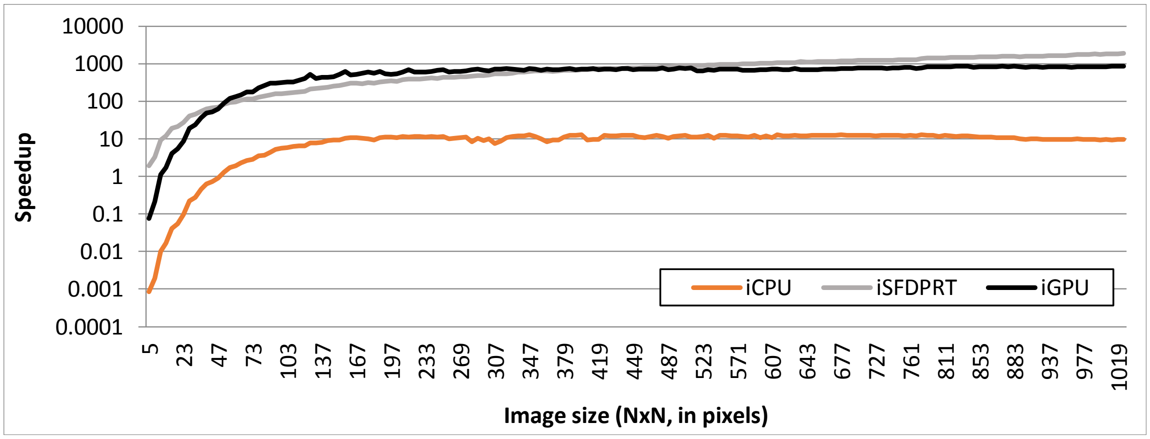


Figure 4.19: Speedup for different implementations of the inverse DPRT with respect to the serial implementation (iSER).

one ray, and around $N > 79$ the L1 cache (or fast SRAM) saturates and there is a requirement to access slower memory. For $N > 167$, the speedup of fGPU levels off around $600 \sim 800$. The hardware-based DPRT has the advantage of not having I/O issues since all the additions are computed without delays. Consequenty, even at a much slower clock frequency, compared with CPUs or GPUs, the running time of the hardware-based DPRT is similar to the GPU.

For real time video processing, Table 4.2 provides maximum image sizes for which performance stays above 30 frames per second. Here, note that the table only takes into account DPRT computation. In an actual real-time application, either the frame-rate or image size may need to be reduced to allow for additional computations. For example, for the GPU implementation, the forward and inverse for a 1021x1021 image requires a total of 22ms, leaving 11ms for further processing.

For the inverse DPRT, the running time is given in Fig. 4.17 and the speedup in Fig. 4.19. The results are virtually the same as the forward DPRT.

Table 4.2: Closest running time to 33.33ms for real time video applications of fSER, fCPU and fGPU

Solution	Time(ms)	Image size
fSER	30.5	151 × 151
fCPU	30.3	353 × 353
fGPU	32.1	1471 × 1472

In conclusion, the GPU implementations achieved significantly more speedup than the CPU-based implementations. In terms of current cost, a Xeon E5-2630 v3 costs US$ 675 vs one GPU GTX980 that is priced at US$ 490 (retail prices at 11/28/2015).

Chapter 5

Conclusions and future work

Overall, my dissertation has led to the development of fast and scalable methods for the computation of the DPRT and its inverse. The fast methods have enabled the application of the DPRT to new areas that were not possible with implementations that required $O(N^3)$ computations. My work offers two paths. First, using the developed hardware implementation, the DPRT and its inverse can be computed in linear time (with respect to N), provided that there are sufficient resources. On the other hand, my methods provide the fastest running times that can be achieved with available resources. Second, using the software implementation on current hardware platforms (multi-core CPUs and GPUs), my parallel algorithms can compute the DPRT in real time.

To demonstrate the application of the new hardware methods for the DPRT, I presented a system to compute 2-D convolutions and cross-correlations with relatively large and non-separable kernels based on the DPRT convolution property.

This approach converts the non-separable 2-D linear convolution/ cross-correlation problem into a sequence of 1-D problems while also significantly reducing the complexity of the calculations. Furthermore, the proposed system is scalable with respect to available resources. Scalability implies that the system can compute the 2-D convolution/cross-correlation in linear time in the fastest case with large resources and slower (down to quadratic time) for cases with fewer hardware resources. In all cases, the methods are Pareto optimal in the sense that they provide the fastest implementations for available resources. Additional improvements can be derived from the use of SVD-LU decompositions for low rank convolution kernels.

Beyond the development of efficient methods for DPRTs for prime N, future research can focus on other values for N. For example, when N is a power of two: $N = 2^m$, the number of additions can be reduced to $N^2 \log_2 N$ while the number of prime directions increases to $3N/2$ and the inverse DPRT needs to be computed iteratively.

Another extension of this work is to explore the multi-objective space defined by the accuracy, performance, and required resources of the DPRT and its applications in 2-D convolutions and cross-correlations. In this case, the goal will be to find Pareto-optimal realizations that balance the different objectives. This type of research involves the determination of optimal parameters such as: (i) the number of bits that are needed at different stages, (ii) the number of singular values kept on the SVD decompositions, and (iii) all other scalability parameters presented in the dissertation. Furthermore, by including accuracy considerations, future research can focus on determining optimality conditions for switching between the DPRT and LU based implementations. Similarly, future research can explore accurate filtering applications based on the DPRT implementations on GPUs.

Appendices

Appendix A

List of publications

This appendix list the publications related with my research.

Prior to University of New Mexico, I worked developing adaptive image restoration methods:

[57] C. A. Carranza, V. Kober, and H. Hidalgo, "Image restoration with local adaptive methods," in Proc. SPIE, Applications of Digital Image Processing XXXIII, vol. 7798, September 2010, pp. 779 827779 82712. [Online]. Available: http://dx.doi.org/10.1117/12.860754

At University of New Mexico and related with the present dissertation, I started parallelizing algorithms to speedup the running time.

[58] D. Llamocca, C. Carranza, and M. Pattichis, "Separable fir filtering in fpga and gpu implementations: Energy, performance, and accuracy considerations," in 2011 International Conference on Field Programmable Logic and Applications (FPL), Sept 2011, pp. 363368.

[59] C. Carranza, V. Murray, M. Pattichis, and E. Barriga, “Multiscale am-fm decompositions with gpu acceleration for diabetic retinopathy screening,” in 2012 IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI), April 2012, pp. 121124.

Also, I started to work on optimization problems.

[60] D. Llamocca, C. Carranza, and M. Pattichis, “Dynamic multiobjective optimization management of the energy-performance-accuracy space for separable 2-d complex filters,” in 2012 22nd International Conference on Field Programmable Logic and Applications (FPL), Aug 2012, pp. 579582.

[61] D. Llamocca, M. Pattichis, and C. Carranza, “A framework for selfreconfigurable dcts based on multiobjective optimization of the power-performance- accuracy space,” in 2012 7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), July 2012, pp. 16.

In 2013, my research focused on the development of an architecture to speedup the computation of the DPRT.

[26] C. Carranza, D. Llamocca, and M. Pattichis, “The fast discrete periodic radon transform for prime sized images: Algorithm, architecture, and vlsi/fpga implementation,” in 2014 IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI), April 2014, pp. 169172.

Once a high speed computation of the DPRT was achieved, the next step was to make it scalable.

[27] C. Carranza, D. Llamocca, and M. Pattichis, “A scalable architecture for

implementing the fast discrete periodic radon transform for prime sized images," in 2014 IEEE International Conference on- Image Processing (ICIP), Oct 2014, pp. 1208-1212.

And the complete solution for the forward and inverse DPRT was presented in 2015.

[3] C. Carranza, D. Llamocca, and M. Pattichis, "Fast and scalable computation of the forward and inverse discrete periodic radon transform," IEEE Transactions on Image Processing, vol. 25, no. 1, pp. 119-133, Jan 2016.

At University of New Mexico I also worked on image registration.

[62] E. Barriga, V. Chekh, C. Carranza, M. Burge, A. Edwards, E. McGrew, G. Zamora, and P. Soliz, "Computational basis for risk stratification of peripheral neuropathy from thermal imaging," in 2012 Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), Aug 2012, pp. 14861489.

[63] V. Chekh, S. S. Luan, M. Burge, C. Carranza, P. Soliz, E. McGrew, and S. Barriga, "Quantitative early detection of diabetic foot," in Proceedings of the International Conference on Bioinformatics, Computational Biology and Biomedical Informatics, ser. BCB13. New York, NY, USA: ACM, 2013, pp. 86:8686:95. [Online]. Available: http://doi.acm.org/10.1145/2506583.2506598

Appendix B

Adder trees resource computation

The architecture inside a fully pipelined X-operand adder tree is not unique. It uses a combination of registers and 2-operand adders interconnected by a binary tree structure. Furthermore, there is not a closed form equation to compute the exact number of resources for an arbitrary X. However, one practical approach is to use the algorithm shown in Fig. B.1 to compute the total number of 1-bit registers and 2-operand B-bits adders used in an X-operand adder tree. Note that if it is assumed each input data is represented with B bits, after each stage, the number of bits is increased by one bit, therefore, on the stage $i = 1, \ldots, h$, each register is $B + i$ bits wide, and the 2-operand adder is a $(B + i - 1)$-bits adder. To normalize the size of the 2-operand adders, a 2-operand $(B+i-1)$-bits adder is expressed as a $(1+(i-1)/B)$ times 2-operand B-bits adder. Additionally, it is included the amount of 2-to-1 muxes used in the register array.

```
procedure Tree_Resources($X, B$)
    $h = \lceil \log_2 X \rceil$
    $\mathtt{A}_{\mathtt{ff}} = \mathtt{A}_{\mathtt{FA}} = \mathtt{A}_{\mathtt{mux}} = 0$
    $a = X$
    for $z = 1$ to $h$ do
        $r = \langle a \rangle_2$
        $a = \lfloor a/2 \rfloor$
        $\mathtt{A}_{\mathtt{FA}} = \mathtt{A}_{\mathtt{FA}} + a \cdot (B + z - 1)$
        $\mathtt{A}_{\mathtt{mux}} = \mathtt{A}_{\mathtt{mux}} + a \cdot B$
        $a = a + r$
        $\mathtt{A}_{\mathtt{ff}} = \mathtt{A}_{\mathtt{ff}} + a \cdot (B + z)$
    end for
    return $\mathtt{A}_{\mathtt{FA}}, \mathtt{A}_{\mathtt{ff}}, \mathtt{A}_{\mathtt{mux}}$
end procedure
```

Figure B.1: Required tree resources as a function of the number of strip rows or number of blocks (X), and the number of bits per pixel (B). Refer to Table 2.3 for definitions of $\mathtt{A}_{\mathtt{ff}}, \mathtt{A}_{\mathtt{FA}}, \mathtt{A}_{\mathtt{mux}}$. For $\mathtt{A}_{\mathtt{ff}}$, the resources do not include the input registers, but do include the output registers since they are implemented in `SFDPRT_core` and `iSFDPRT_core`.

Appendix C

Adder trees resource computation for Convolution

```
 1: procedure Tree_Resources_WIB($N, D$)
 2:     $n = \lceil \log_2 N \rceil$
 3:     $\mathtt{A}_{\mathtt{ffb}} = \mathtt{A}_{\mathtt{FA}} = 0$
 4:     $a = N$
 5:     for $z = 1$ to $n$ do
 6:         $r = \langle a \rangle_2$
 7:         $a = \lfloor a/2 \rfloor$
 8:         $\mathtt{A}_{\mathtt{FA}} = \mathtt{A}_{\mathtt{FA}} + a \cdot (D + z - 1)$
 9:         $a = a + r$
10:         $\mathtt{A}_{\mathtt{ffb}} = \mathtt{A}_{\mathtt{ffb}} + a \cdot (D + z)$
11:     end for
12:     $\mathtt{A}_{\mathtt{ffb}} = \mathtt{A}_{\mathtt{ffb}} + X \cdot D$            ▷ With Input Buffers (WIB)
13:     return $\mathtt{A}_{\mathtt{FA}}, \mathtt{A}_{\mathtt{ffb}}$
14: end procedure
```

Figure C.1: Required tree resources as a function of the zero padded image (N), and the number of bits per pixel (D). Refer to Table 3.1 for definitions of $\mathtt{A}_{\mathtt{ffb}}, \mathtt{A}_{\mathtt{FA}}$. Remove step 12 to compute $\mathtt{A}_{\mathtt{ff}}$ (without input buffers)

Appendix D

Source code for the Serial DPRT and iDPRT on the HOST

```
/*  (c) 2015 Cesar Carranza
  University of New Mexico

  Serial implementation of the forward and inverse DPRT on the
     HOST

  Input data: None. f(i,j) is generated randomly
  Output data: timing.txt (text file with the running time and
     error difference).

  Image sizes: From 2x2 up to 1021x1021
  Data is stored in a vector, column-major ordering.
  top-left pixel (0,0) is the 1st value on the vector.

*/

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>

int forwardDPRT(long *radon, clock_t *runTime, long *img,
                int N, int imgSize);
int inverseDPRT(long *img, clock_t *runTime, long *radon,
                int N, int radonSize);

int main()
{
  const int numImgs = 172; // up to 1021x1021
```

```
// First 172 prime numbers
const int imgSizes[172] = {
  2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
  31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
  73, 79, 83, 89, 97, 101, 103, 107, 109, 113,
  127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
  179, 181, 191, 193, 197, 199, 211, 223, 227, 229,
  233, 239, 241, 251, 257, 263, 269, 271, 277, 281,
  283, 293, 307, 311, 313, 317, 331, 337, 347, 349,
  353, 359, 367, 373, 379, 383, 389, 397, 401, 409,
  419, 421, 431, 433, 439, 443, 449, 457, 461, 463,
  467, 479, 487, 491, 499, 503, 509, 521, 523, 541,
  547, 557, 563, 569, 571, 577, 587, 593, 599, 601,
  607, 613, 617, 619, 631, 641, 643, 647, 653, 659,
  661, 673, 677, 683, 691, 701, 709, 719, 727, 733,
  739, 743, 751, 757, 761, 769, 773, 787, 797, 809,
  811, 821, 823, 827, 829, 839, 853, 857, 859, 863,
  877, 881, 883, 887, 907, 911, 919, 929, 937, 941,
  947, 953, 967, 971, 977, 983, 991, 997, 1009, 1013,
  1019, 1021 };

int N; // Base image size of NxN
int imgSize;
int radonSize;
int status;
long *img; // Pointer to the image data
long *imgOut; // Pointer to the image after the inverse
long *radon; // Pointer to the radon data
int z,y,x;
int errorImg;
int r;
FILE *pun1;

clock_t timeForward = 0;
clock_t timeInverse = 0;

fopen_s(&pun1, "timing.txt", "w");

// Main loop, one iteration per image size
for (x = 0; x < numImgs; x++)
{
  N = imgSizes[x];
  imgSize = N*N;
  radonSize = (N + 1)*N;
  img = (long *)calloc(imgSize, sizeof(long));
```

```
    imgOut = (long *)calloc(imgSize, sizeof(long));
    radon = (long *)calloc(radonSize, sizeof(long));

    // Generate random data for the input image
    for (z = 0; z < imgSize; z++)
    {
      img[z] = rand() % 256;
    }

    // Compute forward DPRT
    status = forwardDPRT(radon, &timeForward, img, N,
                             imgSize);
    if (status != 0) {
      fprintf(stderr, "forward RADON failed!");
      return 1;
    }
    printf("Elapsed Forward time: %li\n", timeForward);

    // Compute inverse DPRT
    status = inverseDPRT(imgOut, &timeInverse, radon, N,
                             radonSize);
    if (status != 0) {
      fprintf(stderr, "inverse RADON failed!");
      return 1;
    }
    printf("Elapsed time Inverse: %li\n", timeInverse);

    // Error, should be zero (it is an exact transform!)
    errorImg = 0;
    for (z = 0; z < imgSize; z++)
    {
      errorImg = errorImg + abs(img[z] - imgOut[z]);
    }
    printf("Error difference for %dx%d size: %d \n\n",N,N,
               errorImg);
    fprintf_s(pun1, "%d,%li,%li,%d\n", N, timeForward,
                  timeInverse, errorImg);
  }

  fclose(pun1);
  free(img);
  free(radon);
  free(imgOut);
    return 0;
}
```

```
int forwardDPRT(long *radon, clock_t *runTime, long *img,
                int N, int imgSize)
{
  int prime, ray, z, incr, init, radxy, sum;
  clock_t start, finish;

  start = clock();
  for(prime=0;prime < N+1; prime++)
  {
    if (prime == N) // Special case for final prime direction
    {
      incr = N; // Increment to get the next value
    }
    else // First N prime directions
    {
      incr = prime * N + 1; // Increment to get the next value
    }
    for(ray=0;ray<N;ray++)
    {
      if (prime == N) // Special case for final prime dir.
      {
        init = ray; // Starting position to add
      }
      else // First N prime directions
      {
        init = ray * N; // Starting position to add
      }
      radxy = prime + ray * (N + 1); // Pos. in the radon
      sum = 0;
      for(z=0;z<N;z++)
      {
        sum = sum + img[init];
        init = (init + incr) % imgSize;
      }
      radon[radxy] = sum;
    }
  }
  finish = clock();
  *runTime = (clock_t)(finish - start);
    return 0;
}

int inverseDPRT(long *img, clock_t *runTime, long *radon,
                int N, int radonSize)
{

  int prime, ray, z, decr, init, radxy, sum;
  int S;
  clock_t start, finish;

  start = clock();

  // Computing S
  S = 0;
  for (z = 0; z < N; z++)
  {
    S = S + radon[z*(N+1)];
  }

  for (prime=0;prime<N;prime++)
  {
    decr = -prime * (N + 1) + 1; // Dec. to get the next value
    for(ray=0;ray<N;ray++)
    {
      init = ray * (N + 1); // Starting position to add
      radxy = prime + ray * N; // Position in the radon output
      sum = 0;
      for(z=0;z<N;z++)
      {
        sum = sum + radon[init];
        init = (init + decr + radonSize) % radonSize;
      }
      img[radxy] = (sum - S + radon[prime*(N + 1) + N])/N;
    }
  }
  finish = clock();
  runTime[0] = (finish - start);
  return 0;
}
```

Appendix E

Source code for the Parallel DPRT and iDPRT on the HOST

```
/*  (c) 2015 Cesar Carranza
  University of New Mexico

  Parallel implementation of the forward and inverse DPRT on
       the HOST

  Input data: None. f(i,j) is generated randomly
  Output data: timing.txt (text file with the running time and
     error difference).

  Image sizes: From 2x2 up to 1021x1021.
  Data is stored in a vector, column-major ordering.
  Top-left pixel (0,0) is the 1st value on the vector.
  Using Pthreads for parallel processing.

*/

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <math.h>
#include <pthread.h>

// For Pthreads, use 16 logical cores, 8 physical
#define cores 16

long *img; // Pointer to the image data
long *imgOut; // Pointer to the image after the inverse
long *radon; // Pointer to the radon data
int N;
int imgSize;
int radonSize;


// Kernels for Pthreads
void *forwardDPRTkernel(void *arg)
{
  int numTh,primeSiz,primeStart,primeEnd, threshold;
  int prime,ray,z,incr,init,radxy,sum;

  numTh = (int)arg; // Kernel ID
  threshold = (N+1)%cores;
  // Some threads have 1 less prime direction
  if (numTh >= threshold && threshold > 0)
  {
    primeSiz = (N+1 + cores-1)/cores - 1;
    primeStart = threshold*(primeSiz+1)+
                       (numTh-threshold)*primeSiz;
    primeEnd = primeStart+primeSiz;
  }  // Below the threshold, Each thread has the full amount
  else
  {
    primeSiz = (N+1+cores-1)/cores;
    primeStart = numTh*primeSiz;
    primeEnd = primeStart+primeSiz;
  }

  for(prime=primeStart;prime < primeEnd; prime++)
  {
    if (prime == N) // Special case for final prime direction
    {
      incr = N; // Increment to get the next value
    }
    else // First N prime directions
    {
      incr = prime * N + 1; // Increment to get the next value
    }
    for(ray=0;ray<N;ray++)
    {
      if (prime == N) // Special case for final prime dir.
      {
        init = ray; // Starting position to add
      }
      else // First N prime directions
      {
        init = ray * N; // Starting position to add
      }
      radxy = prime + ray * (N + 1); // Pos. in the radon
      sum = 0;
      for(z=0;z<N;z++)
      {
        sum = sum + img[init];
        init = (init + incr) % imgSize;
      }
      radon[radxy] = sum;
    }
  }
  pthread_exit((void*) 0);
}

void *inverseDPRTkernel(void *arg)
{
  int numTh,primeSiz,primeStart,primeEnd,threshold;
  int prime,ray,z,decr,init,radxy,sum,S;

  numTh = (int)arg; // Kernel ID

  threshold = N % cores;
  // Some threads have 1 less prime direction
  if (numTh >= threshold && threshold > 0)
  {
    primeSiz = (N + cores - 1)/cores - 1;
    primeStart = threshold*(primeSiz+1)+
                       (numTh-threshold)*primeSiz;
    primeEnd = primeStart+primeSiz;
  }  // Below the threshold, Each thread has the full amount
  else
  {
    primeSiz = (N + cores - 1)/cores;
    primeStart = numTh*primeSiz;
    primeEnd = primeStart+primeSiz;
  }

  // Compute S
  S = 0;
  for (z = 0; z < N; z++)
  {
    S = S + radon[z*(N+1)];
  }

  for (prime=primeStart;prime<primeEnd;prime++)
  {
    decr = -prime * (N + 1) + 1; // Dec. to get the next value
    for(ray=0;ray<N;ray++)
    {
      init = ray * (N + 1); // Starting pos. to add
      radxy = prime + ray * N; // Pos. in the radon
      sum = 0;
      for(z=0;z<N;z++)
      {
        sum = sum + radon[init];
        init = (init + decr + radonSize) % radonSize;
      }
      imgOut[radxy] = (sum - S + radon[prime*(N + 1) + N])/N;
    }
  }

  pthread_exit((void*) 0);
}

int main()
{
  const int numImgs = 172; // Up to 1021x1021
  // First 172 prime numbers
  const int imgSizes[309] = {
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
    31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
    73, 79, 83, 89, 97, 101, 103, 107, 109, 113,
    127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    179, 181, 191, 193, 197, 199, 211, 223, 227, 229,
    233, 239, 241, 251, 257, 263, 269, 271, 277, 281,
    283, 293, 307, 311, 313, 317, 331, 337, 347, 349,
    353, 359, 367, 373, 379, 383, 389, 397, 401, 409,
    419, 421, 431, 433, 439, 443, 449, 457, 461, 463,
    467, 479, 487, 491, 499, 503, 509, 521, 523, 541,
    547, 557, 563, 569, 571, 577, 587, 593, 599, 601,
    607, 613, 617, 619, 631, 641, 643, 647, 653, 659,
    661, 673, 677, 683, 691, 701, 709, 719, 727, 733,
    739, 743, 751, 757, 761, 769, 773, 787, 797, 809,
    811, 821, 823, 827, 829, 839, 853, 857, 859, 863,
    877, 881, 883, 887, 907, 911, 919, 929, 937, 941,
    947, 953, 967, 971, 977, 983, 991, 997, 1009, 1013,
    1019, 1021 };
```

```
int z,y,x;
int errorImg;
int r;
FILE *pun1;

//Parallel version: Using Pthreads
pthread_attr_t attr;
pthread_t hisThr[cores];
void *status1,*status2;
clock_t start, finish;
clock_t timeForward;
clock_t timeInverse;

// Make sure each thread is joinable
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

fopen_s(&pun1, "timing_pthreads.txt", "w");

for (x = 0; x < numImgs; x++)
{
  N = imgSizes[x];
  imgSize = N*N;
  radonSize = (N + 1)*N;
  img = (long *)calloc(imgSize, sizeof(long));
  imgOut = (long *)calloc(imgSize, sizeof(long));
  radon = (long *)calloc(radonSize, sizeof(long));

  for (z = 0; z < imgSize; z++)
  {
    img[z] = rand() % 256;
  }

  start = clock();
  // Compute forward DPRT
  //Launch the threads
  for(z=0;z<cores;z++)
  {
    pthread_create(&hisThr[z], &attr, forwardDPRTkernel,
                         (void *)z);
  }
  //Wait to finish
  for(z=0; z<cores; z++)
  {
      pthread_join(hisThr[z], &status1);
    }
    finish = clock();
    timeForward = finish - start;
    printf("Elapsed Forward time: %li\n", timeForward);

    // Compute inverse DPRT
    start = clock();
    //Launch the threads
    for(z=0;z<cores;z++)
    {
      pthread_create(&hisThr[z], &attr, inverseDPRTkernel,
                           (void *)z);
    }
    //Wait to finish
    for(z=0; z<cores; z++)
    {
      pthread_join(hisThr[z], &status2);
    }
    finish = clock();
    timeInverse = finish - start;
    printf("Elapsed time Inverse: %li\n", timeInverse);

    // Check for zero error
    errorImg = 0;
    for (z = 0; z < imgSize; z++)
    {
      errorImg = errorImg + abs(img[z] - imgOut[z]);
    }
    printf("Error difference for %dx%d size: %d \n\n",N,N,
               errorImg);
    fprintf_s(pun1, "%d,%li,%li,%d\n", N, timeForward,
                  timeInverse, errorImg);
  }

  fclose(pun1);
  free(img);
  free(radon);
  free(imgOut);
    return 0;
}
```

Appendix F

Source code for the Parallel DPRT and iDPRT on the DEVICE (GPU GM204, Maxwell)

```
/*  (c) 2015 Cesar Carranza
  University of New Mexico

  Parallel implementation of the forward and inverse DPRT on
       the DEVICE: Nvidia GM204 - Card: GeForce GTX980

  Input data: None. f(i,j) is generated randomly
  Output data: timing.txt (text file with the running time and
     error difference).

  Image sizes: From 2x2 up to 1021x1021.
  Data is stored in a vector, row-major ordering.
  Top-left pixel (0,0) is the 1st value on the vector.
  Using mixed code:
    HOST (Xeon CPU) for launching the kernels.
    DEVICE (GM203 GPU) executing the kernels.

  use the compilation flag: -Xptxas -dlcm=ca
     to activate Cache L1.

*/

#include <cuda.h>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
```

```
cudaError_t forwardDPRT(long *radon, float *timeKernel,
                        long *img, int N, int imgSize);
cudaError_t inverseDPRT(long *img, float *timeKernel,
                        long *radon, int N, int radonSize);

__global__ void fDPRTKernel(int *radon, const int *img,
                            const int N, const int imgSize)
{
  int init, offs, radxy, z, sum, incr;

  if (blockIdx.x == N) // Special case for final prime dir.
  {
    offs = 0; // Starting position to add
    incr = 1;
    init = threadIdx.x*N;
  }
  else // First N prime directions
  {
    offs = threadIdx.x; // Starting position to add
    incr = N;
    init = 0;
  }
  radxy = threadIdx.x + blockIdx.x * N; // Pos. in rad.
  sum = 0;
  __syncthreads();

  // Add all values on prime dir. blockIdx.x, ray threadIdx.x
  for (z = 0; z < N; z++)
  {
    sum = sum + img[init + offs];
    offs = (offs + blockIdx.x) % N;
    init = init + incr;
  }
  radon[radxy] = sum;
}

__global__ void iDPRTKernel(int *img, const int *radon,
                            const int N, const int radonSize,
                            const int S)
{
  int radxy = threadIdx.x + blockIdx.x * N; //Pos.in img.out
  int z;
  int sum = 0;
  int offs = threadIdx.x; // Starting position to add
  int init = 0;
  // Add all values on prime dir. blockIdx.x, ray threadIdx.x
  for (z = 0; z < N; z++)
  {
    sum = sum + radon[init + offs];
    offs = (offs - blockIdx.x + N) % N;
    init = init + N;
  }
  img[radxy] = (sum - S + radon[N*N + blockIdx.x]) / N;
}

int main()
{
  const int numImgs = 172; // Up to 1021x1021
  // First 172 prime numbers
  const int imgSizes[172] = {
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
    31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
    73, 79, 83, 89, 97, 101, 103, 107, 109, 113,
    127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    179, 181, 191, 193, 197, 199, 211, 223, 227, 229,
    233, 239, 241, 251, 257, 263, 269, 271, 277, 281,
    283, 293, 307, 311, 313, 317, 331, 337, 347, 349,
    353, 359, 367, 373, 379, 383, 389, 397, 401, 409,
    419, 421, 431, 433, 439, 443, 449, 457, 461, 463,
    467, 479, 487, 491, 499, 503, 509, 521, 523, 541,
    547, 557, 563, 569, 571, 577, 587, 593, 599, 601,
    607, 613, 617, 619, 631, 641, 643, 647, 653, 659,
    661, 673, 677, 683, 691, 701, 709, 719, 727, 733,
    739, 743, 751, 757, 761, 769, 773, 787, 797, 809,
    811, 821, 823, 827, 829, 839, 853, 857, 859, 863,
    877, 881, 883, 887, 907, 911, 919, 929, 937, 941,
    947, 953, 967, 971, 977, 983, 991, 997, 1009, 1013,
    1019, 1021};

  int N, i, j;
  int imgSize;
  int radonSize;
  long *img; // Pointer to the image data
  long *imgOut; // Pointer to the image after the inverse
  long *radon; // Pointer to the radon data
  int z, y, x;
  int errorImg;
  FILE *pun1;
```

```
float timeForward = 0;
float timeInverse = 0;

fopen_s(&pun1, "timing_GPU.txt", "w");

for (x = 0; x < numImgs; x++)
{
  N = imgSizes[x];
  imgSize = N*N;
  radonSize = (N + 1)*N;
  img = (long *)calloc(imgSize, sizeof(long));
  imgOut = (long *)calloc(imgSize, sizeof(long));
  radon = (long *)calloc(radonSize, sizeof(long));

  for (z = 0; z < imgSize; z++)
  {
    img[z] = rand() % 256;
  }

  // Compute forward DPRT
  cudaError_t cudaStatus = forwardDPRT(radon, &timeForward,
                                           img, N, imgSize);
  if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "forward RADON failed!");
    return 1;
  }
  printf("Elapsed Forward time: %f\n", timeForward);

  // Compute inverse DPRT
  cudaStatus = inverseDPRT(imgOut, &timeInverse, radon,
                               N, radonSize);
  if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "inverse RADON failed!");
    return 1;
  }
  printf("Elapsed time Inverse: %f\n", timeInverse);

  // Check the error. Should be zero!
  errorImg = 0;
  for (z = 0; z < imgSize; z++)
  {
    errorImg = errorImg + abs(img[z] - imgOut[z]);
  }
  printf("Error difference for %dx%d size: %d \n\n",
             N, N, errorImg);
    fprintf_s(pun1, "%d,%f,%f,%d\n", N, timeForward,
                  timeInverse, errorImg);
  }

  fclose(pun1);
  free(img);
  free(radon);
  free(imgOut);
  return 0;
}


cudaError_t forwardDPRT(long *radon, float *timeKernel,
                        long *img, int N, int imgSize)
{
  int *dev_img = 0;
  int *dev_radon = 0;
  cudaError_t cudaStatus;

  // Allocate GPU buffers for input image and radon output.
  cudaStatus = cudaMalloc((void**)&dev_radon,
                            (imgSize + N) * sizeof(int));
  if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc dev_radon failed!");
    goto Error;
  }
  cudaStatus = cudaMalloc((void**)&dev_img,
                            imgSize * sizeof(int));
  if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc dev_img failed!");
    goto Error;
  }

  // Copy input image from host memory to GPU buffers.
  cudaStatus = cudaMemcpy(dev_img, img, imgSize * sizeof(int),
                            cudaMemcpyHostToDevice);
  if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
  }

  // Timing using cudaEvent
  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord(start);
  // Launch a kernel on with block size of N, and N+1 blocks
  fDPRTKernel << < N + 1, N >> >(dev_radon , dev_img , N,
                                   imgSize);
  cudaEventRecord(stop);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(timeKernel , start , stop);

  cudaEventDestroy(start);
  cudaEventDestroy(stop);

  // Check for any errors launching the kernel
  cudaStatus = cudaGetLastError();
  if (cudaStatus != cudaSuccess) {
    fprintf(stderr , "fDPRTKernel launch failed: %s\n",
                cudaGetErrorString(cudaStatus));
    goto Error;
  }
  // Copy output vector from GPU buffer to host memory.
  cudaStatus = cudaMemcpy(radon , dev_radon ,
       (imgSize + N) * sizeof(int), cudaMemcpyDeviceToHost);
  if (cudaStatus != cudaSuccess) {
    fprintf(stderr , "cudaMemcpy radon failed!");
    goto Error;
  }

Error:
  cudaFree(dev_radon);
  cudaFree(dev_img);

  return cudaStatus;
}

cudaError_t inverseDPRT(long *img, float *timeKernel ,
                        long *radon , int N, int radonSize)
{
  int *dev_img = 0;
  int *dev_radon = 0;
  cudaError_t cudaStatus;

  // Allocate GPU buffers for input radon and image output.
  cudaStatus = cudaMalloc((void**)&dev_radon ,
                            radonSize * sizeof(int));
  if (cudaStatus != cudaSuccess) {
```

```
  fprintf(stderr, "cudaMalloc dev_radon failed!");
  goto Error;
}
cudaStatus = cudaMalloc((void**)&dev_img,
                          (radonSize - N) * sizeof(int));
if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMalloc dev_img failed!");
  goto Error;
}

// Copy radon image from host memory to GPU buffers.
cudaStatus = cudaMemcpy(dev_radon, radon,
     radonSize * sizeof(int), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "cudaMemcpy failed!");
  goto Error;
}

// Computing S
int S;
S = 0;
for (int z = 0; z < N; z++)
{
  S = S + radon[z];
}

// Timing using cudaEvent
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
// Launch a kernel with block size of N, and N blocks
iDPRTKernel << < N, N >> >(dev_img, dev_radon, N,
                             radonSize, S);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(timeKernel, start, stop);

cudaEventDestroy(start);
cudaEventDestroy(stop);

// Check for any errors launching the kernel
cudaStatus = cudaGetLastError();
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "iDPRTKernel launch failed: %s\n",
                cudaGetErrorString(cudaStatus));
    goto Error;
  }
  // Copy img from GPU buffer to host memory.
  cudaStatus = cudaMemcpy(img, dev_img,
       (radonSize - N) * sizeof(int),
        cudaMemcpyDeviceToHost);
  if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy img failed!");
    goto Error;
  }

Error:
  cudaFree(dev_radon);
  cudaFree(dev_img);

  return cudaStatus;
}
```

References

[1] S. Chandrasekaran and A. Amira, “High speed/low power architectures for the finite radon transform,” in *International Conference on Field Programmable Logic and Applications*, Aug 2005, pp. 450–455.

[2] S. Chandrasekaran, A. Amira, S. Minghua, and A. Bermak, “An efficient vlsi architecture and fpga implementation of the finite ridgelet transform,” *Journal of Real-Time Image Processing*, vol. 3, no. 3, pp. 183–193, 2008.

[3] C. Carranza, D. Llamocca, and M. Pattichis, “Fast and scalable computation of the forward and inverse discrete periodic radon transform,” *IEEE Transactions on Image Processing*, vol. 25, no. 1, pp. 119–133, Jan 2016.

[4] S. Deans, *The Radon Transform and Some of Its Applications*, ser. Dover Books on Mathematics Series. Dover Publications, 2007.

[5] A. K. Jain, *Fundamentals of Digital Image Processing.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[6] J.-L. Starck, E. Candes, and D. Donoho, “The curvelet transform for image denoising,” *IEEE Transactions on Image Processing*, vol. 11, no. 6, pp. 670–684, 2002.

[7] D. P. K. Lun, T. Chan, T.-C. Hsung, D. Feng, and Y.-H. Chan, “Efficient blind image restoration using discrete periodic radon transform,” *IEEE Transactions on Image Processing*, vol. 13, no. 2, pp. 188–200, 2004.

[8] K. Jafari-Khouzani and H. Soltanian-Zadeh, “Radon transform orientation estimation for rotation invariant texture analysis,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 6, pp. 1004–1008, June 2005.

[9] N. Aggarwal and W. Karl, “Line detection in images through regularized hough transform,” *IEEE Transactions on Image Processing*, vol. 15, no. 3, pp. 582–591, March 2006.

[10] A. Kingston and I. Svalbe, “Geometric shape effects in redundant keys used to encrypt data transformed by finite discrete radon projections,” in *Digital Image*

Computing: Techniques and Applications, 2005. DICTA '05. Proceedings 2005, 2005, pp. 16–16.

[11] N. Normand, I. Svalbe, B. Parrein, and A. Kingston, "Erasure coding with the finite radon transform," in *2010 IEEE Wireless Communications and Networking Conference (WCNC)*, April 2010, pp. 1–6.

[12] B. Parrein, N. Normand, M. Ghareeb, G. D'Ippolito, and F. Battisti, "Finite radon coding for content delivery over hybrid client-server and p2p architecture," in *2012 5th International Symposium on Communications Control and Signal Processing (ISCCSP)*, May 2012, pp. 1–4.

[13] G.-W. Ou, D.-K. Lun, and B.-K. Ling, "Compressive sensing of images based on discrete periodic radon transform," *Electronics Letters*, vol. 50, no. 8, pp. 591–593, April 2014.

[14] G. Beylkin, "Discrete radon transform," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 35, no. 2, pp. 162–172, Feb 1987.

[15] B. Kelley and V. Madisetti, "The fast discrete radon transform. i. theory," *IEEE Transactions on Image Processing*, vol. 2, no. 3, pp. 382–400, Jul 1993.

[16] F. Matus and J. Flusser, "Image representation via a finite radon transform," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 15, no. 10, pp. 996–1006, 1993.

[17] A. Grigoryan, "Comments on: The discrete periodic radon transform," *IEEE Transactions on Signal Processing*, vol. 58, no. 11, pp. 5962–5963, Nov 2010.

[18] T. Hsung, D. P. K. Lun, and W.-C. Siu, "The discrete periodic radon transform," *IEEE Transactions on Signal Processing*, vol. 44, no. 10, pp. 2651–2657, 1996.

[19] I. Gertner, "A new efficient algorithm to compute the two-dimensional discrete fourier transform," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 36, no. 7, pp. 1036–1050, Jul 1988.

[20] A. Ahmad, A. Amira, H. Rabah, and Y. Berviller, "Medical image denoising on field programmable gate array using finite radon transform," *IET Signal Processing*, vol. 6, no. 9, pp. 862–870, Dec 2012.

[21] A. Kingston and I. Svalbe, "Projective transforms on periodic discrete image arrays," *Advances in Imaging and Electron Physics*, vol. 139, p. 76, 2006.

[22] M. S. Pattichis, "Novel algorithms for the accurate, efficient, and parallel computation of multidimensional, regional discrete fourier transforms," in *10th Mediterranean Electrotechnical Conference, 2000. MELECON 2000.*, vol. 2. IEEE, 2000, pp. 530–533.

[23] M. S. Pattichis and R. Zhou, "A novel directional approach for the scalable, accurate and efficient computation of two-dimensional discrete fourier transforms," *AHPCC2000-019, Albuquerque High Performance Computing Center, The University of New Mexico*, 2000.

[24] M. S. Pattichis, R. Zhou, and B. Raman, "New algorithms for computing directional discrete fourier transforms," in *icip*, vol. 3, 2001, pp. 322–325.

[25] G. H. Hardy and E. M. Wright, *An introduction to the theory of numbers.* Oxford University Press, 1979.

[26] C. Carranza, D. Llamocca, and M. Pattichis, "The fast discrete periodic radon transform for prime sized images: Algorithm, architecture, and vlsi/fpga implementation," in *2014 IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI)*, April 2014, pp. 169–172.

[27] ——, "A scalable architecture for implementing the fast discrete periodic radon transform for prime sized images," in *2014 IEEE International Conference on-Image Processing (ICIP)*, Oct 2014, pp. 1208–1212.

[28] S. Boyd and L. Vandenberghe, *Convex optimization.* Cambridge university press, 2004.

[29] D. Llamocca, M. Pattichis, and G. A. Vera, "Partial reconfigurable fir filtering system using distributed arithmetic," *Int. J. Reconfig. Comput.*, vol. 2010, pp. 4:1–4:14, Feb. 2010. [Online]. Available: http://dx.doi.org/10.1155/2010/357978

[30] D. Llamocca and M. S. Pattichis, "A dynamically reconfigurable pixel processor system based on power/energy-performance-accuracy optimization," *IEEE Transactions on Circuits and Systems for Video Technology*, no. 3, pp. 488–502, March 2013. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6252023

[31] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, 2nd ed. New York, NY, USA: Oxford University Press, Inc., 2009.

[32] A. C. Bovik, *Handbook of Image and Video Processing (Communications, Networking and Multimedia).* Orlando, FL, USA: Academic Press, Inc., 2005.

[33] J. C. Russ and F. B. Neal, *The Image Processing Handbook*, 7th ed. Boca Raton, FL, USA: CRC Press, Inc., 2015.

[34] M. S. Nixon and A. S. Aguado, *Feature Extraction & Image Processing for Computer Vision*, 3rd ed. London, UK: Academic Press, 2012.

[35] M. Anam and Y. Andreopoulos, "Throughput scaling of convolution for error-tolerant multimedia applications," *Multimedia, IEEE Transactions on*, vol. 14, no. 3, pp. 797–804, June 2012.

[36] M. Anam, P. Whatmough, and Y. Andreopoulos, "Precision-energy-throughput scaling of generic matrix multiplication and convolution kernels via linear projections," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 24, no. 11, pp. 1860–1873, Nov 2014.

[37] Y. Jiang and M. Pattichis, "A dynamically reconfigurable architecture system for time-varying image constraints (drastic) for motion jpeg," *Journal of Real-Time Image Processing*, pp. 1–17, 2014. [Online]. Available: http://dx.doi.org/10.1007/s11554-014-0460-8

[38] D. Llamocca and M. Pattichis, "Dynamic energy, performance, and accuracy optimization and management using automatically generated constraints for separable 2d fir filtering for digital video processing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 4, pp. 31:1–31:30, Dec. 2014. [Online]. Available: http://doi.acm.org/10.1145/2629623

[39] D. E. Dudgeon and R. M. Mersereau, *Multidimensional Digital Signal Processing*. Prentice Hall Professional Technical Reference, 1990.

[40] K. R. Rao, D. N. Kim, and J.-J. Hwang, *Fast Fourier Transform - Algorithms and Applications*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[41] I. Uzun, A. Amira, and A. Bouridane, "Fpga implementations of fast fourier transforms for real-time signal and image processing," *Vision, Image and Signal Processing, IEE Proceedings* -, vol. 152, no. 3, pp. 283–296, June 2005.

[42] Xilinx. (2012) Logicore ip, fast fourier transform v8.0. [Online]. Available: www.xilinx.com/support/documentation/ip_documentation/ds808_xfft.pdf

[43] D. P. K. Lun, T.-C. Hsung, and W. C. Siu, "On the convolution property of a new discrete radon transform and its efficient inversion algorithm," in *IEEE International Symposium on Circuits and Systems, 1995. ISCAS '95.*, vol. 3, Apr 1995, pp. 1892–1895 vol.3.

[44] H. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 1, pp. 37–46, Jan 1982.

[45] H. keung Kwan and T. Okullo-Oballa, "2-d systolic arrays for realization of 2-d convolution," *Circuits and Systems, IEEE Transactions on*, vol. 37, no. 2, pp. 267–233, Feb 1990.

[46] B. Mohanty and P. Meher, “Cost-effective novel flexible cell-level systolic architecture for high throughput implementation of 2-d fir filters,” *Computers and Digital Techniques, IEE Proceedings -*, vol. 143, no. 6, pp. 436–439, Nov 1996.

[47] Y. Dong, Y. Dou, and J. Zhou, “Optimized generation of memory structure in compiling window operations onto reconfigurable hardware,” in *Reconfigurable Computing: Architectures, Tools and Applications.* Springer, 2007, pp. 110–121.

[48] J. Fowers, G. Brown, P. Cooke, and G. Stitt, “A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications,” in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays.* ACM, 2012, pp. 47–56.

[49] A. Antoniou, *Digital Signal Processing: Signals, Systems, and Filters.* New York NY, USA: McGraw-Hill Education, 2005.

[50] W.-S. Lu, H.-P. Wang, and A. Antoniou, “Design of two-dimensional fir digital filters by using the singular-value decomposition,” *Circuits and Systems, IEEE Transactions on*, vol. 37, no. 1, pp. 35–46, Jan 1990.

[51] P. Cooke, J. Fowers, G. Brown, and G. Stitt, “A tradeoff analysis of fpgas, gpus, and multicores for sliding-window applications,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 1, pp. 2:1–2:24, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2659000

[52] P. Meher, S. Chandrasekaran, and A. Amira, “Fpga realization of fir filters by efficient and flexible systolization using distributed arithmetic,” *Signal Processing, IEEE Transactions on*, vol. 56, no. 7, pp. 3009–3017, July 2008.

[53] J. Fowers, G. Brown, J. Wernsing, and G. Stitt, “A performance and energy comparison of convolution on gpus, fpgas, and multicore processors,” *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 25:1–25:21, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400684

[54] M. Flynn, “Some computer organizations and their effectiveness,” *Computers, IEEE Transactions on*, vol. C-21, no. 9, pp. 948–960, Sept 1972.

[55] NVIDIA. (2015) CUDA toolkit & SDK. [Online]. Available: http://docs.nvidia.com/cuda

[56] T. Rauber and G. Rnger, *Parallel Programming for Multicore and Cluster Systems*, 2nd ed. New York, USA: Springer Berlin Heidelberg, 2013.

[57] C. A. Carranza, V. Kober, and H. Hidalgo, “Image restoration with local adaptive methods,” in *Proc. SPIE, Applications of Digital Image Processing XXXIII*, vol. 7798, September 2010, pp. 779 827–779 827–12. [Online]. Available: http://dx.doi.org/10.1117/12.860754

[58] D. Llamocca, C. Carranza, and M. Pattichis, "Separable fir filtering in fpga and gpu implementations: Energy, performance, and accuracy considerations," in *2011 International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2011, pp. 363–368.

[59] C. Carranza, V. Murray, M. Pattichis, and E. Barriga, "Multiscale am-fm decompositions with gpu acceleration for diabetic retinopathy screening," in *Image Analysis and Interpretation (SSIAI), 2012 IEEE Southwest Symposium on*, April 2012, pp. 121–124.

[60] D. Llamocca, C. Carranza, and M. Pattichis, "Dynamic multiobjective optimization management of the energy-performance-accuracy space for separable 2-d complex filters," in *2012 22nd International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2012, pp. 579–582.

[61] D. Llamocca, M. Pattichis, and C. Carranza, "A framework for self-reconfigurable dcts based on multiobjective optimization of the power-performance-accuracy space," in *2012 7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, July 2012, pp. 1–6.

[62] E. Barriga, V. Chekh, C. Carranza, M. Burge, A. Edwards, E. McGrew, G. Zamora, and P. Soliz, "Computational basis for risk stratification of peripheral neuropathy from thermal imaging," in *2012 Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, Aug 2012, pp. 1486–1489.

[63] V. Chekh, S. S. Luan, M. Burge, C. Carranza, P. Soliz, E. McGrew, and S. Barriga, "Quantitative early detection of diabetic foot," in *Proceedings of the International Conference on Bioinformatics, Computational Biology and Biomedical Informatics*, ser. BCB'13. New York, NY, USA: ACM, 2013, pp. 86:86–86:95. [Online]. Available: http://doi.acm.org/10.1145/2506583.2506598