The Pennsylvania State University
The Graduate School

A FINE-GRAINED DATAFLOW LIBRARY FOR RECONFIGURABLE STREAMING ACCELERATORS

A Thesis in
Electrical Engineering
by
Aarti Chandrashekhar

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

December 2011

The thesis of Aarti Chandrashekhar was reviewed and approved* by the following:

Vijaykrishnan Narayanan
Professor of Computer Science and Engineering
Thesis Advisor

Suman Datta
Professor of Electrical Engineering

Kultegin Aydin
Professor of Electrical Engineering and Interim Department Head

*Signatures are on file in the Graduate School.

Abstract

In this thesis, a library of basic operators for accelerating complex algorithms on an Field Programmable Gate Array (FPGA) is proposed. The components of this custom Register Transfer Level (RTL) hardware library are specifically designed to provide fine-grained control over resources while accelerating algorithms on an FPGA. Furthermore, the library is extensible allowing designers to develop custom operators. A hardware framework to ease the composition of systems using the components of this library is also presented. Such an approach facilitates the use of dataflow programming at the application level for mapping an algorithm to the hardware components. This framework is highly modular and configurable in terms of hardware resources, bit-width allocation, and accuracy. In addition, the hierarchical nature of this framework allows recursive definitions of custom operators. This allows complex operators to be built using the library operators. The framework is well-suited to image processing tasks since it takes into account the streaming requirements of such applications. The initial architecture of this framework and the associated drawbacks are discussed and a new improved architecture which overcomes these drawbacks is also presented. Biologically-inspired vision processing algorithms with applications such as saliency detection and object recognition are studied as a use case of the framework. In particular, the implementation of a bio-inspired architecture of Retinal and Lateral Geniculate Nucleus (LGN) processing stages using the proposed framework is detailed. All the hardware examples are synthesized and verified on a Xilinx© Virtex-6 SX475T FPGA. The FPGA implementation is also compared to a multi-core CPU implementation of the algorithm and it is shown that the FPGA-based implementation outperforms the CPU-based implementation by an order of 10.

Table of Contents

Chapter 6

Chapter 7

List of Figures

List of Tables

Acknowledgments

I would like to express my sincere gratitude to my advisor, Dr. Vijaykrishnan Narayanan, for his constant encouragement, unabated enthusiasm and effective guidance throughout my research. I would also like to thank my committee member, Dr. Suman Datta for his insightful comments and guidance. I express my sincere appreciation to the staff of the Department of Electrical Engineering and Department of Computer Sciences and Engineering for their constant support in my administrative matters.

I would also like to take this opportunity to thank my mentor, Dr. Kevin Irick. His enthusiasm, inspiration, and relentless effort to explain hardware and software concepts simply and clearly has made me love hardware design. I would also like to thank my project members and colleagues at the Microsystems Design Laboratory (MDL), at Penn State, who have always been there to help me at every stage of my research. Special thanks to our collaborators at Teledyne Technologies, Inc. I would also like to acknowledge DARPA for funding this research work.

I thank my closest friends Anuj Jaiswal, Prasoon Joshi, and Shruti Gupta who have constantly provided me encouragement, inspiration, support, and a push, whenever I have been in need. Lastly, I am grateful to my family who have always encouraged me to study further and supported me at every step of this journey.

Dedication

This thesis is dedicated to the loving memory of my beloved father, who taught me to be a good engineer and a better human being.

Chapter 1

Introduction

Reconfigurable accelerators such as Field Programmable Gate Arrays (FPGAs) are popular due to their inherent massive parallelism [1], low power consumption [2, 3] and high computational density [4]. Furthermore, FPGAs allow fine-grained control of device computational resources where each gate within an FPGA can be independently controlled. Due to these advantages, computational-intensive algorithms, especially image processing tasks, can be greatly accelerated when they are off-loaded from general-purpose CPUs to FPGAs [5, 6, 7, 8]. However, developing efficient hardware implementations on FPGAs requires a significant effort compared to other solutions [9] based on, e.g., General-Purpose Graphics Processing Units (GPGPUs). Firstly, the designer is required to possess in-depth knowledge of Hardware Description Languages (HDL), such as VHDL [10, 11] and Verilog [12, 13], which are typically used to describe the behavior and structure of systems and circuit designs for a given implementation. Secondly, a thorough understanding of FPGA architecture allows the designer to create HDL code that effectively uses FPGA system features. In addition, experience in dealing with the intricacies of synthesis and place-and-route tools provides an added advantage in generating an optimal solution. Lastly, verification of the implemented design on FPGA is an important aspect in generating a correct solution. Such complexities involved in the design and implementation of algorithms on an FPGA can be a deterrent to researchers from exploiting FPGAs for accelerating complex signal and image processing applications [9]. Therefore, there is a need for design methodologies that ease the composition of systems - making the process accessible to those working at the algorithm, application, and system levels.

All complex algorithms employ basic operations such as multiplication, convolution, histogram computation, data scaling and data binning. A significant portion of the programming effort in the development of such algorithms, for an FPGA, is spent on developing these basic operators and integrating them for a desired solution. In case of hardware implementation of image processing algorithms, real-time performance is desired. Therefore, additional complexities are introduced when the design requires pipelining and worst-case buffering. Furthermore, it is desirable that the hardware implementation on reconfigurable computing systems be highly flexible. In case of image processing, the implementation must support multiple image sizes, variable bit-widths, variable number of pixels per clock cycle etc. While larger bit-widths in intermediate computation stages leads to better accuracy, they come at the cost of increased resource utilization. Similarly, more pipelined stages leads to better timing, and therefore, higher operating frequency but at the cost of latency. Therefore, the availability of configurable basic modules having uniform interfaces simplifies the exploration and development of larger systems. Frameworks that combine these basic modules effectively are required to build complex solutions from these basic building blocks. In addition, software frameworks can be used to combine these modules in a dataflow programming manner, ultimately reducing the time and effort to build and modify such systems. Such an approach will encourage researchers who are not trained in HDL to implement any algorithm in a short period of time.

1.1 Problem Statement

To reduce the effort in the implementation of complex algorithms on FPGAs, this thesis proposes a hardware framework consisting of a library of basic arithmetic and image processing modules. The proposed framework also provides the capability to combine multiple operators efficiently to build large streaming systems.

1.2 Related Work

In this section, related work on dataflow programming and FPGA performance are presented.

1.2.1 Dataflow Programming

Dataflow computers were proposed as an architecture offering massive parallelism. Dataflow architectures [14, 15, 16, 17] maximize concurrent execution by utilizing local memory and by executing instructions as soon as data operands are ready. For an excellent review of the history of dataflow architecture, programming, issues and dataflow future, please refer to [18].

In recent years, there have been numerous studies on design and implementation of vision applications on FPGAs using dataflow. NeuFlow [19, 20, 21] is a dataflow computer for vision processing which is suitable for convolutional neural networks. Janneck et al. [22] illustrate hardware synthesis from dataflow programs to implement a MPEG-4 simple profile decoder. Single Assignment C (SA-C) [23, 24] is a high-level language for writing image processing algorithms that can then be compiled and executed on an FPGA.

1.2.2 Performance Studies on FPGAs

Thomas et al [25] discuss the future of parallel computing by comparing CPU, GPU, Massively Parallel Processor Array (MPAA) and FPGA performance for random number generation. Their study shows that the performance per joule of an FPGA was orders of magnitudes greater than any other platform for this task. Cope et al [8] compared the performance of 2D convolution on FPGAs and GPUs. Their results indicate that FPGA performance is much higher than GPUs. In addition, their study shows that for 2D convolution of size 7x7, GPUs were incapable of meeting their target throughput rates of 8MP/s and FPGAs were the only current solution for this task. A second study [26], again, found that FGPA performance was at least three times greater than GPUs. Both these studies [8, 26], however, did not take into account other advantages offered by FPGAs such as performance per joule which would have shown that FPGAs offer unrivalled performance in addition to significantly lower power consumption. Chase et al [9] similarly studied real-time optical flow calculations of FPGAs and GPUs. Their experimental evaluation again showed that FPGAs have unmatched Input/Output (I/O) and computation capabilities while GPUs are sensitive to compute to I/O ratios. Secondly, their work shows that high performance for Multiple Input Multiple Output (MIMO) operations on GPUs is dependent on high data buffering while FPGA implementations of MIMO modules had lower latency. Lastly, their work highlighted that the drawbacks of using FPGAs was the greater development time and skill level required. Che et al [5] undertook a performance study where three applications ("Gaussian elimination", "Data encryption standard (DES)" and "Needleman-Wunch") where evaluated on FPGA, GPU and multi-core CPU platforms. Their study again show that FPGAs offer unmatched performance over all other platforms for these applications.

1.3 Research Contributions

The primary contributions of this thesis are as follows:

- A novel hardware framework that accelerates implementation of complex algorithms on FPGAs is introduced. The proposed framework can be used to build FPGA solutions that either use minimal resources and/or provide greater accuracy. In case of image processing applications, the framework provides solutions that have high (real-time) processing rates. This framework also provides the capability for a non-hardware designer to compose the algorithm using dataflow programming at the application level, thus reducing complexity of generating hardware implementations.

- The capabilities and effectiveness of the proposed framework architectures are demonstrated by implementing the Retinal and LGN processing stages of a neuromorphic vision algorithm. The first framework architecture is highly modular with different operator modules having similar interfaces. This framework architecture allows recursive definition of complex operators using the library operators. Moreover, the framework architecture is highly configurable and enables a designer to choose between greater computation accuracy, resource minimization or high input data processing rates. However, such a framework architecture results in hardware implementations that have large resource overheads and operators do not exploit all features provided. The second proposed framework architecture overcomes the large resource overheads by optimizing operator modules to use only those features that are essential.

- The performance results of the FPGA-based implementation are compared to that of a CPU-based implementation to demonstrate the advantages of using FPGAs over multi-core CPUs. The experimental evaluation demonstrates that the FPGA-based implementation outperforms a CPU-based implementation by an order of 10.

1.4 Organization of this Thesis

The rest of the thesis is organized as follows: Chapter 2 introduces dataflow programming concepts and its application to the proposed framework design. Chapter 3 presents a hardware framework for implementing complex algorithms on an FPGA. The drawbacks of the initial framework are also outlined in this chapter. Chapter 4 presents the optimized framework which was redesigned based on the lessons learned after the design of the framework illustrated in Chapter 3. A bio-inspired vision algorithm for image preprocessing is introduced in Chapter 5 and forms the use case for both the frameworks. Experimental setup is discussed in Chapter 6 along with a comparison of the accuracy of a fixed-point implementation on an FPGA to that of a double-precision implementation using Matlab. This chapter also presents a comparison with the performance of the same algorithm on a CPU. Finally, the conclusion and future work are discussed in Chapter 7.

Chapter 2

Dataflow Library

The current state-of-the-art FPGA devices contain high-speed I/Os, specialized Digital Signal Processing (DSP) units, abundant on-chip memories, and embedded processor cores [27, 28]. In addition, FPGAs allow massive parallelism as compared to CPUs, thereby greatly accelerating tasks such as matrix multiplications [29, 30]. However, it is tedious for non-hardware algorithm developers to exploit in-built processing cores of an FPGA for creating an optimal solution [9]. Therefore, there is a need to create libraries of basic fine-grained processing units which can then be connected together to implement a given task. Complex processing units can be built using these basic processing units, thereby introducing hierarchy and modularity in the design. Furthermore, at the application level, the algorithm can be mapped to these basic processing units in hardware by using a dataflow program or language. Such an approach accelerates the composition of systems on reconfigurable accelerators. This chapter introduces dataflow concepts and a hardware library of basic fine-grained dataflow nodes/operators.

2.1 Dataflow Programming Concepts

A dataflow program is a directed graph of data flowing between operations. The graph consists of nodes which represent operations and edges which represent data paths between the operations. Programming languages such as VHDL, Verilog, Single Assignment C (SA-C), LabView, Simulink are some examples of dataflow programming languages.

Consider a set of operations given in following equations

$$x <= a + b; \tag{2.1}$$

$$y <= (x - (a * b))^2; \tag{2.2}$$

$$z <= (a * b) + y; \tag{2.3}$$

These equations can be represented using a dataflow graph as presented in Figure 2.1.

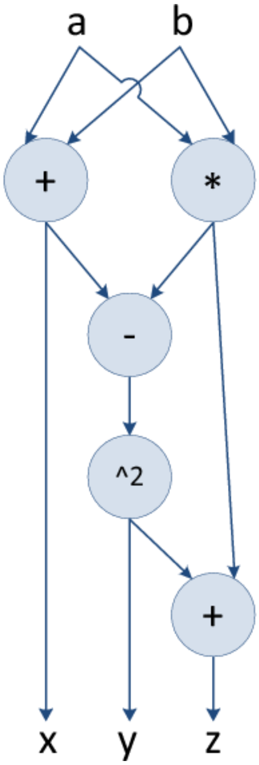

Figure 2.1. Implementation of set of equations using dataflow graph

To implement the above set of equations on a reconfigurable accelerator, a library of optimized operators has been developed to represent each of the computational nodes of the dataflow graph. Section 2.2 describes the features and functionality of dataflow computational nodes in detail. Such a fine-grained library enables the developer to have greater control over the flexibility of the design and resource utilization of an FPGA. Once the computational nodes are instantiated, the connections between them are established using a framework discussed in Chapter 3.

2.2 Dataflow Operators

A dataflow library consisting of basic arithmetic and logical operations/functions such as addition, subtraction, multiplication, division, thresholding, etc., have been developed. In addition, the library also provides image processing operators such as convolution, histogram computation, logarithmic and hyperbolic tangent (tanh) function, subsampler, etc. Furthermore, operators for computing image statistics such as mean and standard deviation are also provided. All operators in the library have a uniform streaming architecture and support parameterizable operand bit-widths. Operators also support both signed and unsigned operations and have the ability to operate on a single datum or multiple data simultaneously. The amount of data processed per clock cycle is configurable at design-time. Lastly, each operator has the ability to perform computations between either two vectors, a vector and a scalar, or a vector and an unsized array.

Each operator uses the same XpressLink protocol at its interface allowing increased consistency and modularity. Using the same protocol at the operator interface also allows ease of automation when using a software framework, at the application level, to combine multiple operators. Operators may also have a configuration space that is accessed via a configuration interface at run-time. This configuration space may be used to store operator-specific run-time parameters such as constants, convolution kernels and image sizes.

A set of operators can be combined to generate more complex operators. For example, a sigmoid operator can be generated using library operators such as tanh, multiplier, subtractor, etc. Designers can also create custom operators that can then be integrated with other library operators to develop a complex solution. To simplify the development of custom operators, the operator interface consists of a minimal set of signals that are required for the correct functioning of the operator. Each operator processes only its inputs and is unaware of other operators that may be connected to it. However, such an approach introduces additional complexities which are discussed in section 2.3.

2.3 Composition of Complex Systems

Operators begin processing when both the operands become available at their input. Therefore, when composing any algorithm using operators in the dataflow library, developers often face issues such as synchronization of operands. The operator may not function properly if both its operands are not available at the input at the same time. In addition, when using fixed-point representation for the operands, operators such as adders and subtractors output a correct computed result only when both their operands have the same number of fractional bits. Therefore, operands need to be properly scaled before they can be input to the operator. The composition of complex systems using basic operators is, therefore, difficult due to issues discussed above. To ease the composition of systems, a framework is presented in Chapter 3 where synchronization and scaling requirements at both the input and output are abstracted away from the operator. This abstraction simplifies operator design since designers need not focus on scaling/ synchronization when building custom operators.

Chapter 3

Initial Framework for Composition of Complex Systems

Complex algorithms usually consist of sequences of transformations such as multiplication, convolution, log and histogram computation. A modular framework is proposed for implementing such complex algorithms on FPGAs, where the operations or transformations are implemented by a set of modules. The input to each module is a set of data while the output is a set of data on which a transformation or operation has been performed. When such modules are connected sequentially to solve a complex task, each module requires the previous data set to be partially or completely available before its output can be generated. Such modular architectures are sequential because some modules will be dependent on the output of other connected modules. However, parallelism can be introduced in such a framework because each module on an FPGA can operate on multiple data of the input data set in a single clock cycle, e.g., multiple adjacent pixels of an input image. Figure 3.1 illustrates a modular bio-inspired vision system consisting of the Retinal and LGN processing stages followed by the Primary Visual Cortex (V1) processing stage.

In this chapter, a configurable hardware framework that consists of processing units named Streaming Vector Processors (SVPs) is introduced. In an SVP, computation is performed on vectors at the data element level with bounded latency from the input of a datum to its corresponding output. SVP processing is similar to execution of Single Instruction Multiple Data (SIMD) instructions on

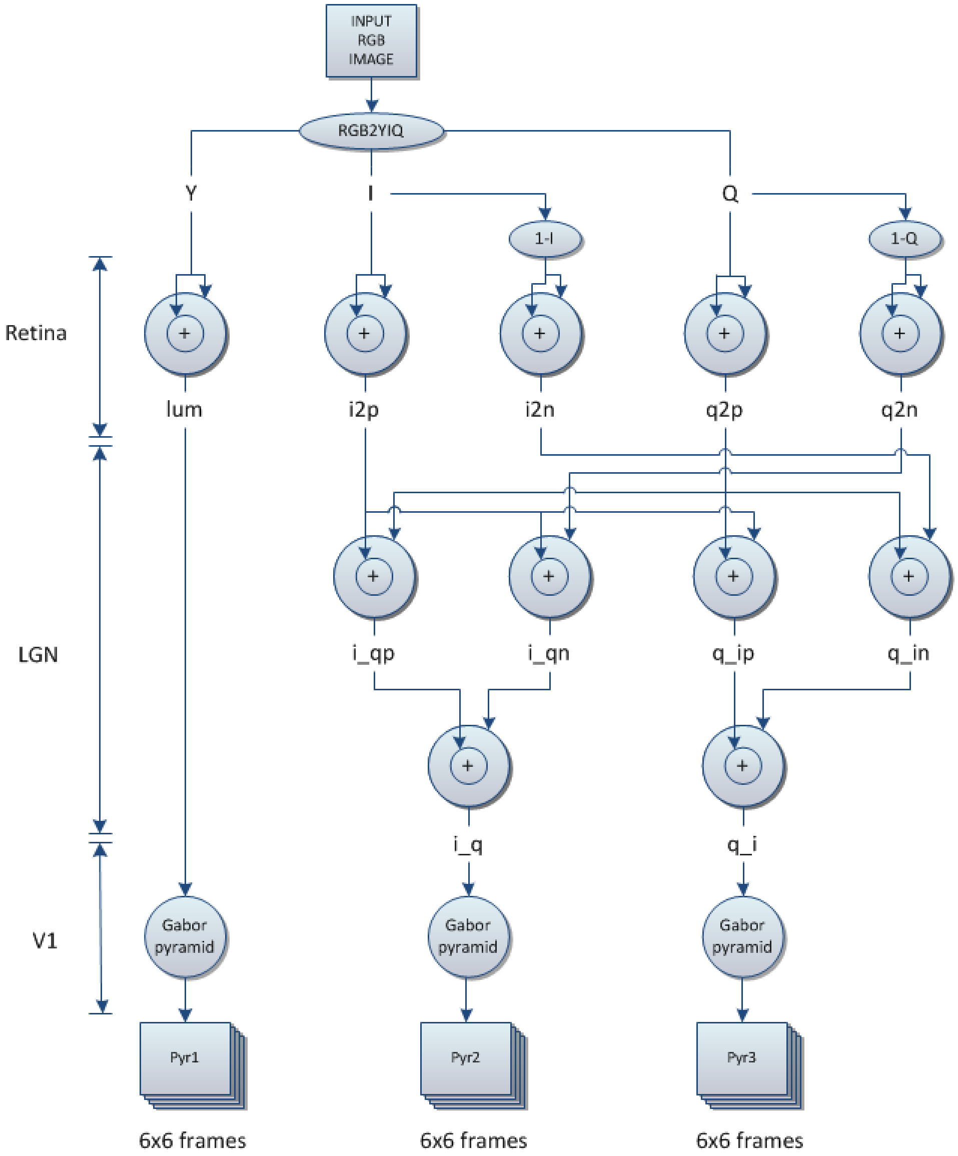

Figure 3.1. Modular system

microprocessors. For image processing applications, SVPs are ideal implementation modality for common pixel-level operations including numerical conversion, color space conversion, coordinate system conversion, data scaling, histogram computation, vector differencing, and data binning. Multiple cascaded SVPs can be utilized to build very powerful systems where the output of one SVP serves as the input to another SVP and so on. An example of a cascade of SVPs is illustrated in Figure 3.2.

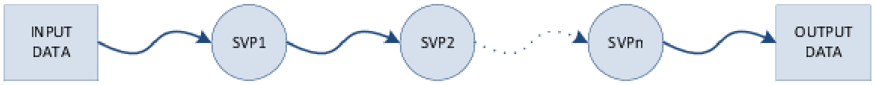

Figure 3.2. Cascade of SVPs

The SVP provides a configurable pipeline for performing Vector-Vector, Vector-Immediate, and Vector-Transform operations on streaming data. These modes of operations are discussed in detail in section 3.1.2. Each SVP is composed of a custom operator and a common SVP wrapper. Operators within an SVP process the input data received without using any buffers for synchronization of multiple input operands. An SVP wrapper is a run-time configurable module that provides buffering at the input and output of each operator. The advantage of such an architecture is that the designer can create custom operators and integrate it with an SVP wrapper to form a set of SVPs designed for an algorithm. An implementation of the set of operations described by equations 2.1, 2.2 and 2.3 using the framework is depicted in Figure 3.3. Another advantage of this approach is that the designer does not need to focus on synchronizing the operands of an operator since synchronization is handled by the SVP wrapper. Lastly, the designer does not need to scale operands of the operator (assuming the operands have fractional bits) since the SVP wrapper contains in-built configurable scaling blocks. The architecture and functioning of the SVP wrapper is described in section 3.1.

3.1 SVP Wrapper

The proposed hardware framework consists of an SVP wrapper that encapsulates library or custom operators to create an SVP. This allows designers to build new operators, and removes the burden of providing buffering within each new operator for operand synchronization. The wrapper provides buffering at both input and output, data-width and format conversion to each operator, thereby simplifying system composition using these operators. To enable the design to be configurable at run-time, intermediate bit-widths within the SVP wrapper were fixed at 8-, 16- or 32-bits. The architecture and components of the SVP wrapper are discussed in section 3.1.1. A block diagram of the wrapper is presented in Figure 3.4.

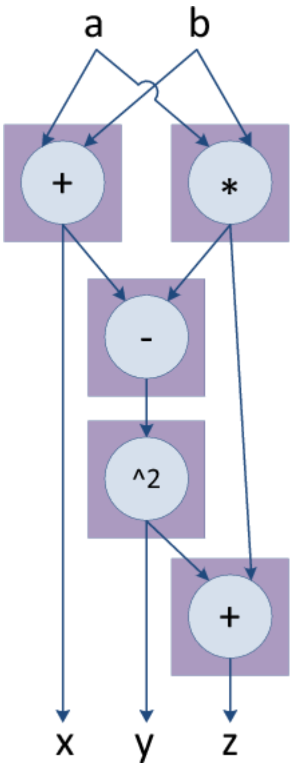

Figure 3.3. Implementation of set of equations using framework

3.1.1 Pipeline Components

The need and functioning of the components of the SVP wrapper are discussed in the following sections. The connections between these components are illustrated in Figure 3.4.

3.1.1.1 Control Unit

The Control Unit orchestrates the flow of data from the input of the SVP through the operators and finally into the output queue. The Control Unit grants access to each of the input and output queues, configures the functional units based on the opcodes, configures the data-path based on the opcode and arbitrates access to the output interface. The Control Unit uses the opcodes that are provided when the input queues are requested to determine the configuration of all pipeline elements. The Control Unit is configured at run-time via a Configuration Interface. The Configuration Interface exposes a memory mapped view of any configuration registers that need to be externally visible.

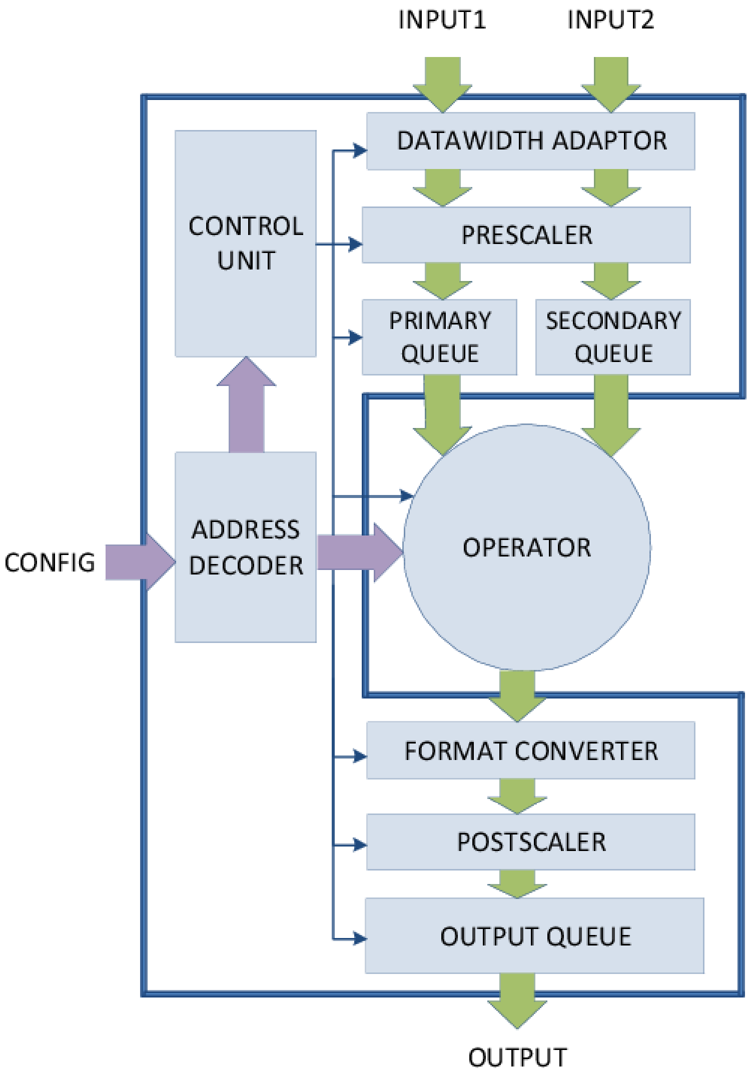

Figure 3.4. Block diagram of SVP wrapper

3.1.1.2 Address Decoder

Each operator as well as its wrapper has its own configuration space. The operator configuration space can be used to store the constants in case of an operation between a vector and a scalar, kernel values in case of a convolution etc. The wrapper configuration space, on the other hand, is used to describe the behavior of the buffers for different modes of operations and for data scaling and format conversion blocks. Each of these configurations can be accessed using an opcode discussed in section 3.1.3. The Address Decoder transfers the run-time configuration parameters sent through the configuration interface of the SVP to the configuration spaces of the operator and the wrapper respectively. In a system of multiple SVPs, the Address Decoder also manages configuration spaces of each SVP.

3.1.1.3 Datawidth Adaptor

The SVP Wrapper provides an input interface of 128-bits or 256-bits. However, the data-path width may be configured internally to meet performance requirements, maintain resource constraints, and/or match output bandwidth. The Datawidth Adaptor converts the SVP input 128/256-bit data-bus into the internal pipeline data width.

3.1.1.4 Prescaler

The Prescaler is used to scale the input data when the two inputs have different number of fractional bits. The Prescaler supports scaling of both signed and unsigned numbers. The amount and direction of scaling is set at run-time by a configuration parameter to the Prescaler.

3.1.1.5 Primary and Secondary Queues

The input queues consist of the Primary Queue and the two Secondary Queues. In Vector-Vector mode, data is streamed into the Primary Queue and one of the two Secondary Queues throughout the duration of the computation. In Vector-Immediate mode, data is streamed into the Primary Queue throughout the duration of the computation. However, one of the Secondary Queues is streamed once with data which is used throughout the duration of computation. In addition, the loading of the Secondary Queue is performed before the Vector-Immediate computation is started. In Vector-Transform mode, data is streamed into either the Primary Queue or the Secondary Queue throughout the computation. Modes of operation are discussed in more detail in section 3.1.2.

The Primary and Secondary Queues are accessed via dedicated XpressLink interfaces. In Vector-Immediate mode, data enqueued in a Secondary Queue is reused as the Secondary Queue supports non-destructive dequeueing. Dual Secondary Queues allows the overlap of enqueueing and dequeueing of either queue. This is particularly useful when performing a Vector-Immediate operation between the Primary Queue and one of the Secondary Queues as the unused Secondary Queue can be simultaneously enqueued with data for another computation.

3.1.1.6 Format Converter

Image processing operators typically use full-precision in representing the computed output data in fixed-point representation. However, there are cases where the range of the computed data values from that operator may not be as large as the bits reserved for representing the computed values. In such cases, the hardware resources allocated for storing data values are greater than required. Format Converter unit is used in such cases to scale the data bit-widths down by discarding a fixed number of integer bits which are configured at run-time.

3.1.1.7 Postscaler

Each operator in the SVP pipeline has an associated Postscaler that performs optional scaling of results output from the operator. The specific amount of scaling performed is input as a run-time parameter.

3.1.1.8 Output Queue

The results of the computation after performing format conversion and scaling get enqueued into an Output Queue. The width of the Output Queue is either 128-bits or 256-bits.

3.1.2 Modes of Operation

There are three modes of operation - Vector-Vector, Vector-Immediate and Vector-Transform which are described in the following sections.

3.1.2.1 Vector-Vector Mode

In Vector-Vector mode, operations are performed element-wise between operands enqueued in the Primary Queue and either of the Secondary Queues. Operands are fixed at 8-bits, 16-bits or 32-bits, though actual data width of an operand may only consist of a subset of the bit width. For example, if the actual data is 24 bits wide, a 32-bit register is used to store this data. The data occupies the lower 24 bits of the register and the upper 8 bits are set to zeros.

3.1.2.2 Vector-Immediate Mode

In Vector-Immediate mode, operations are performed element-wise between the operands in the Primary Queue and either of the Secondary Queues. Unlike Vector-Vector mode, data in the secondary queue is preloaded and consumed and re-consumed in a circular fashion. Once the last operand in the secondary queue is used, computation continues reusing the first operand in the secondary queue. In this way, computation may continue indefinitely as long as there is data available in the Primary Queue.

3.1.2.3 Vector-Transform Mode

In Vector-Transform mode operations are performed on each element in the Primary Queue. In this mode, outputs are a function of input from the Primary Queue and any pre-configured parameters that are associated with the current operation as defined by an opcode. Opcodes are discussed in detail in section 3.1.3.

3.1.3 Opcodes

To allow the SVP to be virtualized for many simultaneous computation contexts, many sets of configurations for the wrapper and the operator can be set via the configuration interface. To access a particular set of configurations during run-time, an opcode is utilized. The opcode specifies the behavior of the queues, data-path, and operator for the current operation. In addition, for operations requiring data in both the Primary Queue and Secondary Queue, the opcode allows the Control Unit to ensure that operands in either queue are matched and appropriate for the given operation.

3.2 Drawbacks

While the framework architecture provided massive flexibility to a designer, however it had a few drawbacks. The first drawback of this architecture was that the framework supported only two-input operators. Secondly, the resource overhead of the SVP wrapper was significant as compared to the operators themselves. The wrapper provided flexibility to the design and a variety of functionalities such as scaling and data-width conversion, However, not all operators took advantage of all the functionalities provided by the wrapper. Lastly, the necessity to maintain fixed bit-widths at the intermediate stages to make the design completely configurable at run-time ultimately led to an increase in resource utilization. In addition, this also led to a loss of accuracy when data was scaled down at the intermediate stages.

As an example, the resource utilization of a sigmoid function implemented using the proposed framework is presented in Tables 3.1 and 3.2. The sigmoid function was implemented on a Xilinx Virtex-6 FPGA. Table 3.1 shows the resource utilization of the operators that were used to build a sigmoid function. Table 3.2 presents the resource utilization of the operators and their wrappers that were used to build the sigmoid function.

Table 3.1. Resource utilization of operators of sigmoid function

Logic Utilization	Used	Utilization
Slice Registers	3681	0%
Slice LUTs	2863	1%
Block RAM/FIFOs	2	0%

Table 3.2. Resource utilization of operators and wrapper of sigmoid function

Logic Utilization	Used	Utilization
Slice Registers	9812	1%
Slice LUTs	6557	2%
Block RAM/FIFOs	26	2%

The results presented in Tables 3.1 and 3.2 show that the resource overhead of the SVP wrapper is significant even for simple functions, e.g., sigmoid. Moreover, since these functions do not utilize many of the capabilities provided by the wrapper, such a framework design is inefficient. Thus when developing a solution, the overhead of the framework may lead to the design requiring significant resources that may be unavailable on an FPGA. The framework architecture was, therefore, revised and a new optimized architecture was developed that is presented in Chapter 4.

Chapter 4

Optimized Framework for Composition of Complex Systems

To overcome the drawbacks of the SVP wrapper discussed in the section 3.2, the framework architecture was modified to reduce resource utilization while providing the same flexibility and functionalities of the previous archtecture. In addition to the basic operators present in the library, new operators were introduced for data scaling and format conversion. However, unlike the previous implementation where data-width conversion features were provided to each operator, a new format converter operator is provided which abstracts data-width conversion away from each operator. The format converter operator can be introduced in a pipeline whereever it is required, thus reducing resource utilization. Furthermore, to efficiently combine multiple operators, another component called Operator Link is introduced which is capable of concatenating incoming data from multiple sources, broadcasting incoming data to multiple destinations while providing buffering. The new framework architecture is also capable of supporting multiple input operators, unlike the initial architecture which only supported two-input operators. Moreover, the optimized framework is not constrained to maintaining fixed intermediate bit-widths (i.e., 8, 16 or 32-bits) and can operate on any bit-width. Such an approach leads to a significant reduction in the resource utilization and also prevents loss of accuracy as compared to the previous framework architecture. An SVP implementation of the set of operations described by equations 2.1, 2.2 and 2.3 using the optimized framework is depicted in Figure 4.1.

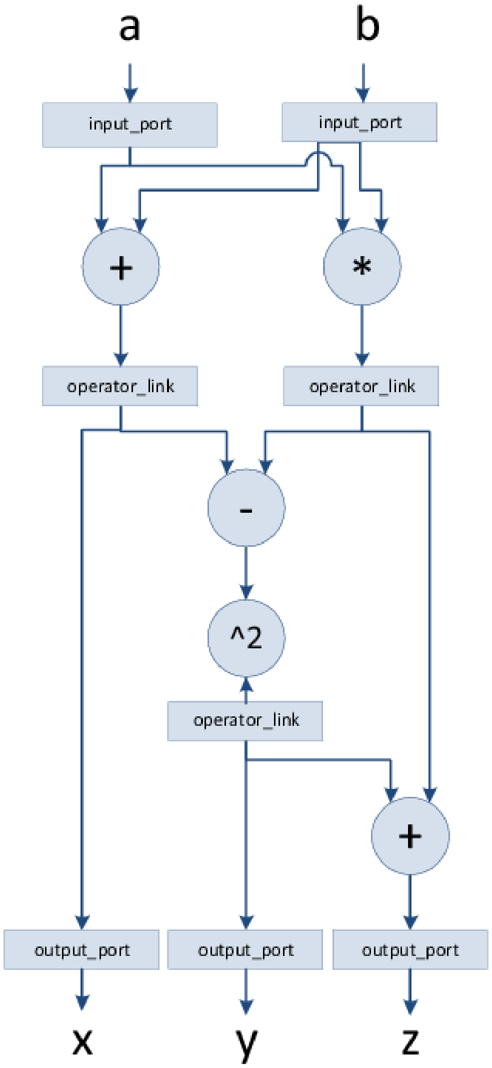

Figure 4.1. Implementation of set of equations using optimized framework

4.1 Pipeline Components

Some of the components from the previous design such as the operator library, control unit and address decoder are modified in the new framework. The new components in the revised framework are presented below.

4.1.1 Input/Output Port

Input and Output ports facilitate the flow of data from the input to the output of each SVP. They provide a range of functionalities such as data-width conversion, synchronization and broadcast of incoming data to multiple targets and convergence of data from multiple sources to a single target. These ports can be configured at run-time via a configuration interface. The configuration space provided within each port facilitates multiple configurations to be stored and selection of a specific configuration at run-time by using an opcode. These ports are also responsible for keeping track of the amount of data flowing in and out of the SVP.

4.1.2 Operator Link

The Operator Link facilitates connections in the intermediate stages of the pipeline between the streaming operators. The Operator Link provides functionalities essential for streaming operations such as convergence and divergence of incoming data and synchronization of data through buffering. Links are configurable to support different modes of operation as presented in the following sections.

4.1.2.1 Modes of Operation

Operator links can function in three modes namely multicast, pack and unpack.

4.1.2.1.1 Multicast An Operator Link can multicast an incoming datum to many different modules connected at its output. This is useful in applications where the output data from one operator flows to multiple connected operators. An example of an Operator Link operating in multicast mode is shown in Figure 4.2. The Operator Link also synchronizes incoming data when it is broadcasting to all targets.

4.1.2.1.2 Pack An Operator Link can also pack data received from multiple sources to a single target. An example of an Operator Link operating in pack mode is shown in Figure 4.3.

4.1.2.1.3 Unpack When the input to the operator link is a packed array of data, the link can work in a segment mode where it unpacks the input and sends each datum to a different target as shown in Figure 4.4.

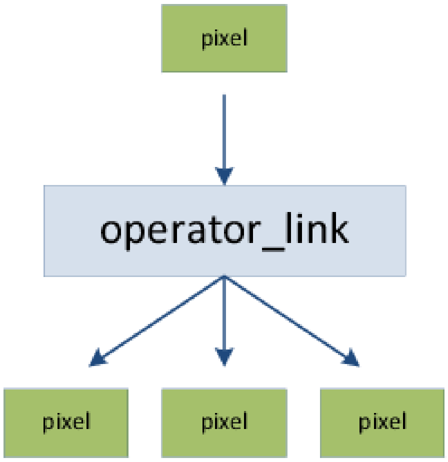

Figure 4.2. Multicast mode of operator link

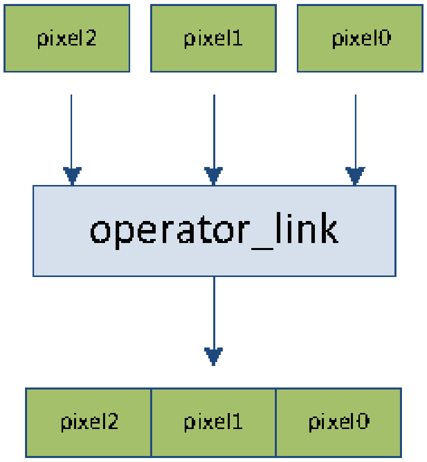

Figure 4.3. Pack mode of operator link

4.1.2.2 Link Buffering

To enable synchronization of data between operators, the Operator Link has an in-built buffer with a parameterizable depth. The depth of the buffer in the link can be set at design-time or the buffer can be completely disabled if not required.

Links can also be configured to perform double-buffering as shown in Figure 4.6. Consider a case where it is required to compute streaming statistics such as mean or standard deviation for an image processing algorithm. The mean of an image can be computed only when the mean operator processes the entire input image. This introduces a latency of at least one image frame during the processing. This latency is greatly increased in a pipelined design especially when there are many

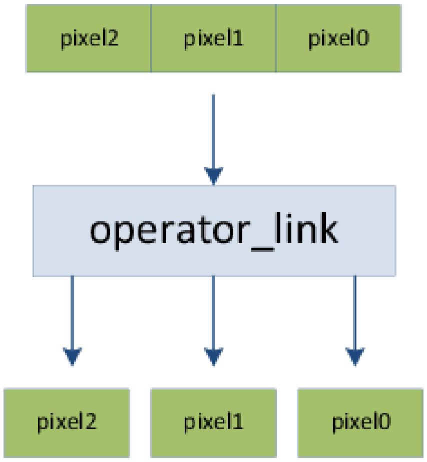

Figure 4.4. Unpack mode of operator link

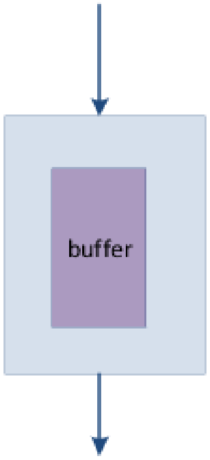

Figure 4.5. Operator link with single buffer

such statistics that must be computed in order to generate the required output. In a completely streaming architecture, the computed statistics of the previous image frame can be used for processing the current frame. Such an approach reduces the latency in computing the mean statistics thereby making the implementation completely streaming. However, in such cases, double buffering must be performed by the link where the computed statistic of the last frame is enqueued in one buffer and utilized in the current frame. At the same time, the computed statistic for the current frame must be enqueued in a second buffer which can then be used for processing the next image frame.

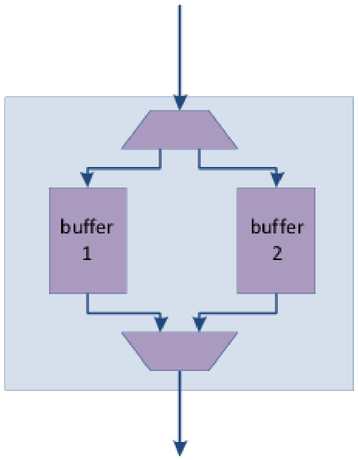

Figure 4.6. Operator link with double buffer

4.1.3 Configurability of Links and Ports

The Input/Output Port and Operator link can be either pre-configured or configured at run-time. For example, the resource utilization of a sigmoid function implemented on a Xilinx Virtex-6 FPGA with pre-configured ports and links is compared to that of an implementation with run-time configured ports and links in Tables 4.1 and 4.2 respectively.

Table 4.1. Resource utilization of sigmoid function with pre-configured ports and links

Logic Utilization	Used	Utilization
Slice Registers	4290	0%
Slice LUTs	3705	1%
Block RAM/FIFOs	3	0%
DSP48Es	14	0%

Table 4.2. Resource utilization of sigmoid function with run-time configured ports and links

Logic Utilization	Used	Utilization
Slice Registers	3397	0%
Slice LUTs	3789	1%
Block RAM/FIFOs	3	0%
DSP48Es	14	0%

When the ports and links are pre-configured, the utilization of look-up tables

(LUTs) on an FPGA is decreased in comparison to when they are configured at run-time. However, the utilization of registers on an FPGA is increased. This allows fine-grained control of the resource utilization of an FPGA along with the ability to change the degree of configurability of a design.

The framework architecture discussed in this chapter is utilized to implement a bio-inspired vision processing algorithm. Chapter 5 introduces the biological principles behind the algorithm. A bio-inspired vision processing architecture is also presented. The basic processing units of the algorithm are implemented using the operator library and the framework and are new additions to the operator library. The entire vision processing algorithm is implemented by recursively using these custom operators with the framework.

Chapter 5

Case Study: Retinal and LGN Processing of Visual System

In this chapter, the retinal and LGN processing of the visual system is discussed. Vision processing is one of the most complex functions performed by primates. In the human visual system, the retina, lateral geniculate nucleus (LGN), and primary visual cortex (V1) stages perform image preprocessing for various high-level vision tasks such as salient region extraction and object recognition [31]. A part of the vision processing is performed in the retina, which is the tissue lining the back of the eye. The axons of the retinal ganglion cells forms the optic nerve which transmits the visual information through the optic chiasm to the Lateral Geniculate Nucleus (LGN) which is located in the thalamus of the brain. The LGN not only relays the information received from the retina to the Primary visual cortex (V1), but also performs some vision processing on the received data [31, 32]. Finally, the messages are sent to the brain where they get analyzed and interpreted, thereby leading to object recognition. The visual system of primates is shown in Figure 5.1.

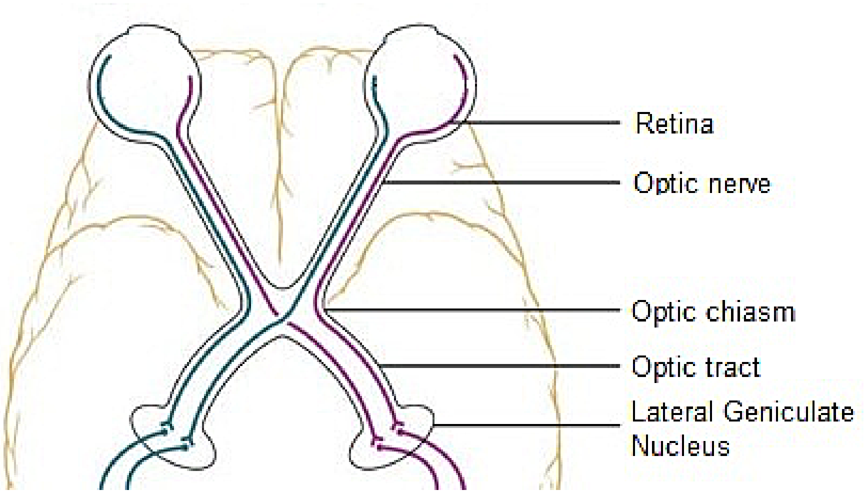

Figure 5.1. Visual system of primates

5.1 Biological Principles

5.1.1 Retina

The retina of primates and humans is the tissue lining the back of the eye. It contains two basic photoreceptor cells - rods and cones. Rods are extremely sensitive and active even at low light levels and support black-and-white vision while cones are active at higher light levels and are responsible for color vision. Cones are further sub-divided into three classes - short, medium and long wavelength sensitive cones. Humans achieve perception of color through the combinations of these three sensors sets [33, 34].

When light falls on the photoreceptors, they send their response to the bipolar cells, which in turn signal the retinal ganglion cells. The photoreceptors are connected to the bipolar cells by the horizontal cells, which run parallel to the retinal layers. Similarly, the amacrine cells connect the bipolar cells to the retinal ganglion cells [31, 32] as illustrated in Figure 5.2.

There are about 125 times more photoreceptor cells than ganglion cells, thus implying that each photoreceptor is not necessarily connected directly to a bipolar cell and each bipolar cell is not directly connected to a ganglion cell [31]. The pit of the retina, called the fovea, is the region responsible for maximum acuity of vision. It is in this region that there is a one-to-one connection between the receptors,

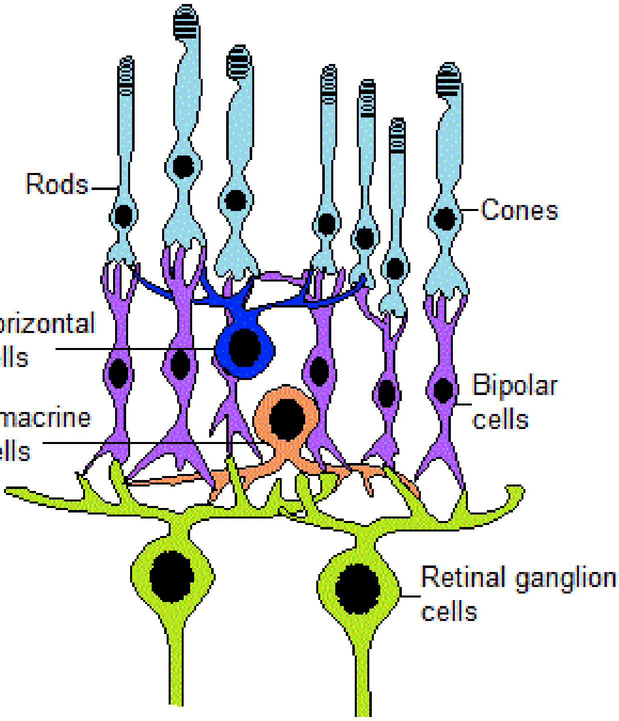

Figure 5.2. Retinal cells

bipolars and ganglion cells. As we further outwards from the fovea towards the periphery of the retina, more receptors converge on bipolars and more bipolars converge on ganglion cells [31].

5.1.2 Lateral Geniculate Nucleus

The LGN is located in the thalamus of the brain and is the the primary relay station for the visual data received from the retina of the eye [35]. There is an LGN in both, the right and left hemispheres of the brain of primates, each having six distinctive layers of neurons. The axons from the LGN travel to the primary visual cortex through the optic radiation. Both the LGNs receive input from the retinas of both eyes, however, each LGN processes information from only one half of the visual field. The LGN also receives input back from the primary visual cortex [31, 32].

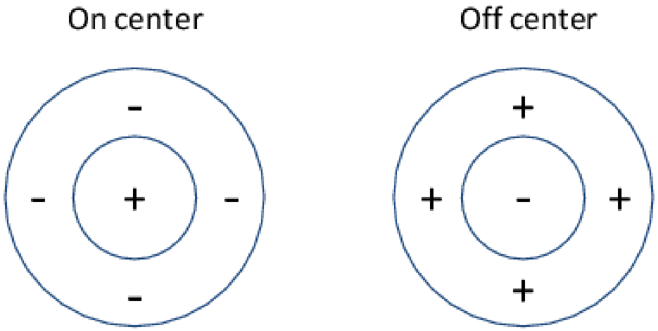

Figure 5.3. Receptive fields of ganglion cells

5.1.3 Receptive Fields of Retina and LGN

It is well-known that the retinal ganglion cells fire action potentials at a steady rate even in the absence of any stimulation [36]. Around 1950, Stephen Kuffler studied the activity of these cells under stimulation by light. Kuffler noticed that when light falls on the receptive fields of these cells, the firing rate is either increased or decreased depending on where the light fell [36]. This led to the discovery of two distinct types of the retinal ganglion cells - on-center cells and off-center cells. On-center cells have a receptive field with an on center and inhibitory surround while off-center cells have an off center with excitatory surround as shown in Figure 5.3. Therefore, on-center cells get stimulated when the center is exposed to light and inhibited when the surround is exposed to light, and vice-versa for off-center cells as shown in Figures 5.4 and 5.5. When both the center and the surround of these cells are not stimulated by light, both the on center and off center cells do not fire as depicted in Figure 5.6. On the other hand, the response from both cells is weak when both center and surround are exposed to light and this is shown in Figure 5.7. During the firing of action potentials by a retinal ganglion cell, the firing of nearby or surround ganglion cells are inhibited [36]. Such center-surround inhibition leads to edge or contrast enhancement. The center-surround structure of the retinal ganglion cells leads to contrast enhancement and edge detection of objects within the visual field of the retina. The image that is spatially encoded by the center-surround structures is then sent out the optical nerve through the optic chiasm to the LGN [31].

The receptive fields of lateral geniculate cells have the same center-surround organization as the retinal ganglion cells that feed into them. While the retina

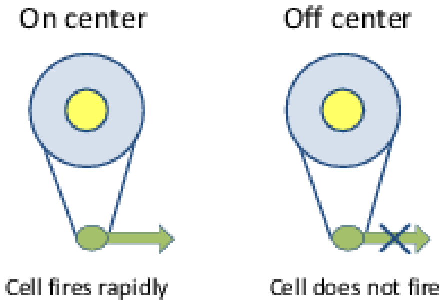

Figure 5.4. Behavior of on-center and off-center cells with only center exposed to light

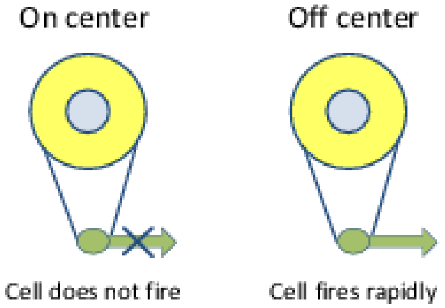

Figure 5.5. Behavior of on-center and off-center cells with only surround exposed to light

accomplishes spatial de-correlation through center surround inhibition, the LGN accomplishes temporal de-correlation [37].

5.2 Bio-inspired Retinal-LGN Architecture

The input image is first converted from RGB color space to a biologically plausible color space such as YIQ color space. The luminance i.e. Y and chrominance channels i.e. I and Q channels are then processed individually. The basic architecture for the Retinal and LGN processing consists of three distinct processing stages as shown in Figure 5.8. In the first processing stage which is the retina, within-band image enhancement and normalization is performed by utilizing a non-linear neural network or a shunting image operator. This produces contrast enhancement, dynamic range calibration, and normalization of input images. The second stage which is sigmoid performs a normalization on the output of retina. The third stage which is LGN adopts the use of the same shunting image operator to produce between-band de-correlation, information enhancement and fusion. The

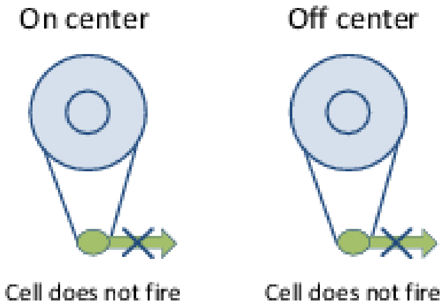

Figure 5.6. Behavior of on-center and off-center cells with both center and surround not exposed to light

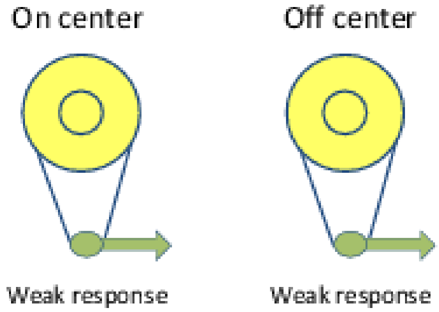

Figure 5.7. Behavior of on-center and off-center cells with both center and surround exposed to light

shunt combinations of the third stage provides four unique sets of information rich images.

5.2.1 Shunting Image Operator

The shunting operator basically performs Difference of Gaussians (DoG) which is a contrast enhancement algorithm. The input image is smoothed by convolving the original image with Gaussian kernels having differing standard deviations. The difference of the two Gaussian smoothed images is used for contrast enhancement and edge detection in an image. However, in neuro-physiological systems and in the algorithm, the non-linear operator has a very narrow spatial window providing a better-tuned de-correlation. In addition, the operator is modulated by more globally defined statistical characteristics of the input that produce normalization, smoothing, and between-band calibration.

There are two basic stages within the shunting image operator. The first stage performs contrast enhancement operation by Difference of Gaussians method. The

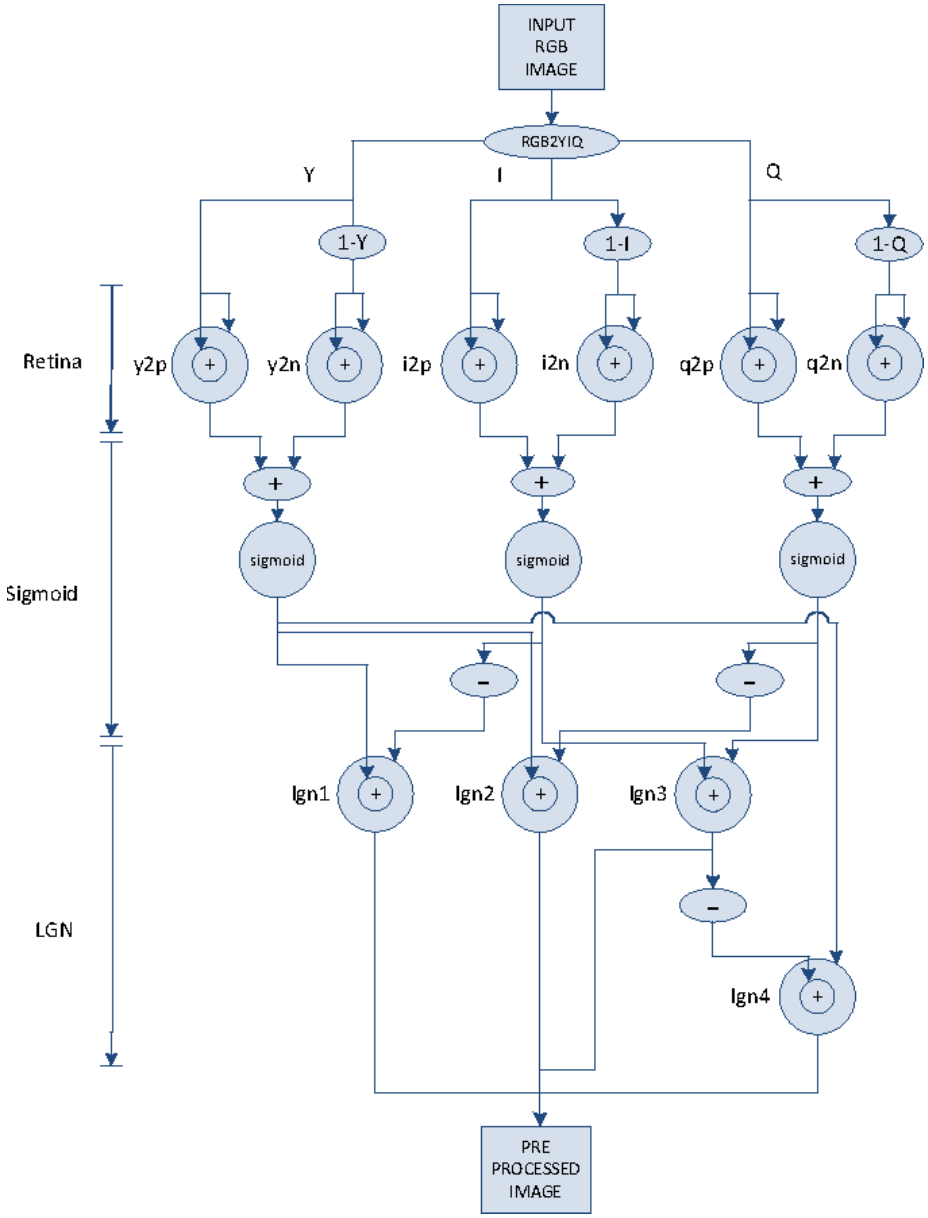

Figure 5.8. Biologically-based retinal-LGN architecture

second stage normalizes and re-maps the resulting contrast enhanced values to the target range by employing a sigmoid operator with a relatively steep slope. In effect, the combination of these two stages leads to the dynamic range compression of the input image in conjunction with contrast enhancement.

5.2.1.1 Implementation of Sigmoid Operator using Optimized Framework

Figure 5.9 shows the implementation of a sigmoid operator using library operators such as tanh, multiplier, adder from the dataflow library along with the connectivity framework.

5.2.1.2 Implementation of Shunting Image Operator using Optimized Framework

Figure 5.10 presents the shunting image operator which basically consists of two processing stages. The first stage performs a Difference of Gaussians using the convolution and multiplication operators from the dataflow library, along with some clipping and clamping operators for thresholding. The second stage is the sigmoid operator which was a custom operator created using the library operators and is used for normalization of the output from the first stage to the required range. In the case where the shunting operator is used for within-band image enhancement, both the center and the surround inputs are derived from the same band. In the case in which the operator is used to combine two bands, the inputs mapped to the center and surround are derived from each of the input images. Here, band 1 is mapped to the center and each of the pixels from band 1 are used to drive the excitatory input of their corresponding shunting operator. Then, a corresponding area of the image for band 2 is used as the surround input that is fed into the same shunt operator. The result is the contrast enhancement of information in band 1 as matched against band 2.

5.2.2 Retinal-LGN Architecture

The sigmoid operator was first created using the library operators and framework. This new custom operator for sigmoid function was then utilized to build the shunting image operator along with other library operators and the framework. Finally, the retinal-LGN processor was implemented using the same framework and the shunting image operator as well as the sigmoid operator as shown in Figure 5.11.

The results of the implementation of the bio-inspired retinal-LGN architecture

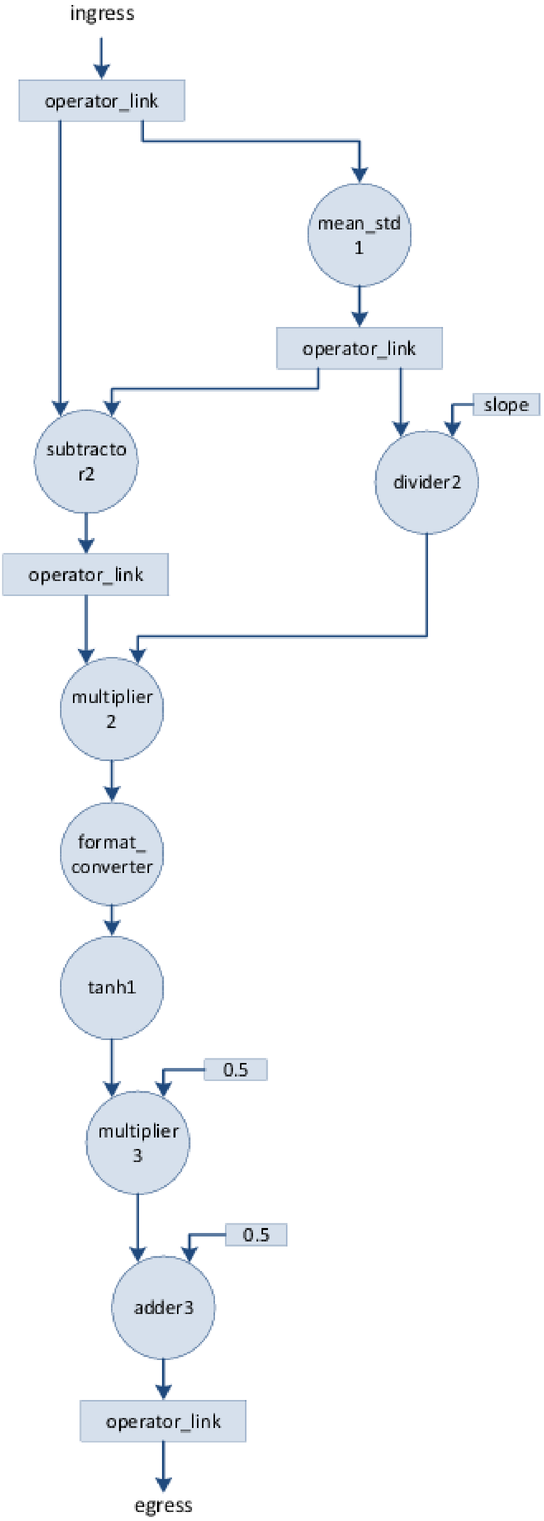

Figure 5.9. Implementation of sigmoid operator using optimized framework

on an FPGA are presented in Chapter 6. To compare the performance of the FPGA-based implementation to that on a CPU, the algorithm was also implemented on a multi-core CPU. The accuracy of the fixed-point implementation on the FPGA with respect to a double-precision implementation in Matlab is also presented in Chapter 6.

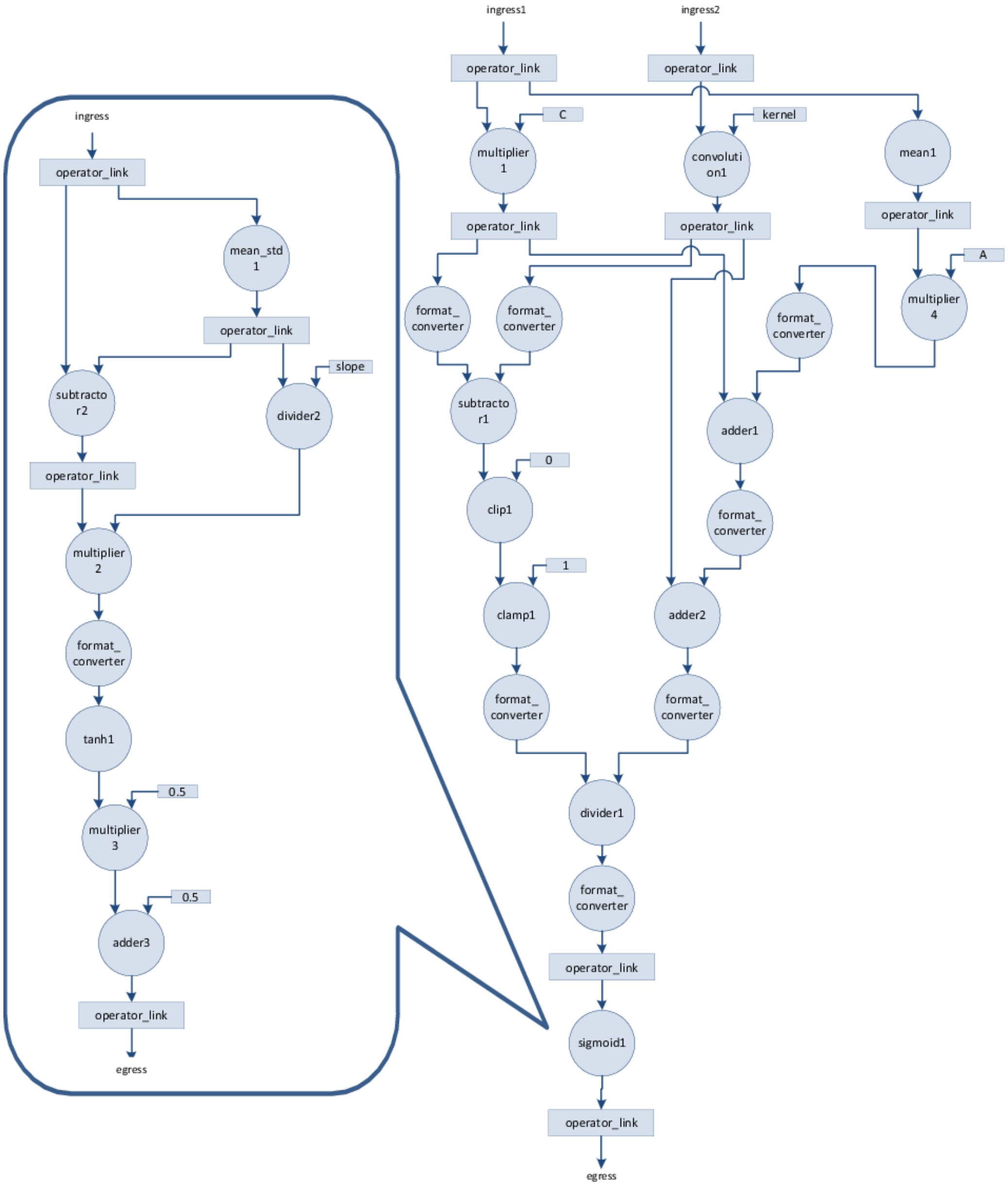

Figure 5.10. Implementation of shunting image operator using optimized framework

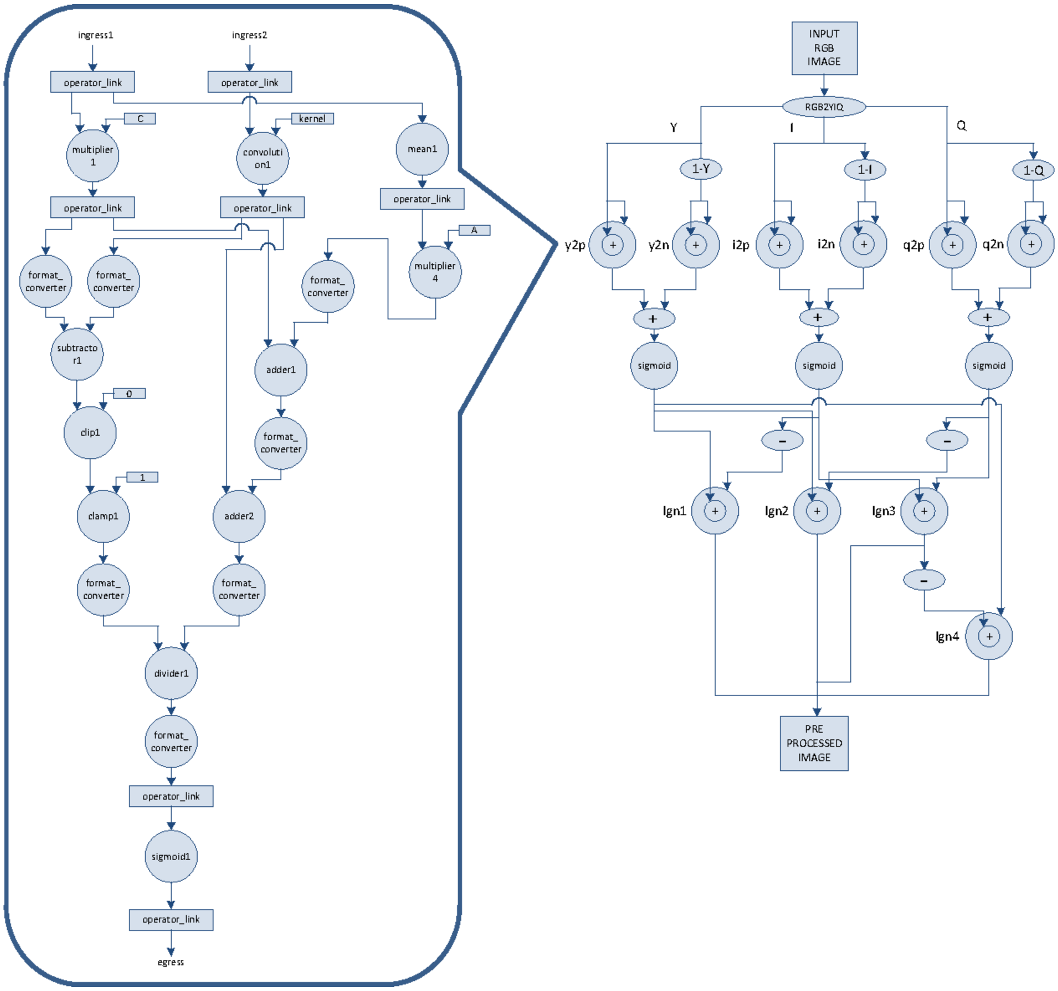

Figure 5.11. Implementation of retinal-LGN processor using optimized framework

Chapter 6

Experimental Setup and Results

The Retinal-LGN processing architecture was implemented in Verilog HDL and synthesized using Xilinx ISE 13.2 design tools. Simulation was carried out using Mentor Graphics ModelSim simulation suite. Software simulation, used for accuracy comparison purposes, was implemented in Matlab.

6.1 Normalization and RGB to YIQ Conversion

The input image is normalized between 0 to 1 and changed to a biologically plausible color space. A color space conversion from RGB to YIQ is performed on the image and the individual Y, I and Q channels are extracted and input to the Retinal-LGN processing architecture. The formulae for RGB to YIQ conversion is given in Equation (6.1).

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.595716 & -0.274453 & -0.321263 \\ 0.211456 & -0.522591 & 0.311135 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{6.1}$$

6.2 Double Precision to Fixed-Point Precision Analysis

While CPUs and GPUs can only handle 8-, 16-, 32-, and 64-bit variables, FPGAs support arbitrary bit-widths for each variable in the design. By adjusting the bit widths according to the precision requirement, significant reduction in the silicon area cost of arithmetic units and bandwidth requirement between different hardware modules can be achieved. Thus, this approach improves overall throughput of the entire system. The image input to the retinal-LGN implementation was fixed to be 16-bit fixed-point unsigned numbers in the range from 0 to 1. Although the range of the input image pixels was fixed, intermediate and final pixel values in the system can have a larger range. Therefore, conversion from a floating-point format to fixed-point format was carried out using the fixed-point toolbox in Matlab. A fixed-point analysis was performed to determine appropriate data bit-width for sufficient computation accuracy for the algorithm. The result of such optimizations was smaller, faster functional units and reduced data storage requirements. The accuracy of the fixed-point implementation as compared to the double-precision equivalent implementation was in the range of 10^{-05} to 10^{-06}.

6.3 Validation on DiniGroup FPGA board

The bio-inspired algorithm studied in Chapter 5 was synthesized and implemented on a FPGA board. The features of the selected FPGA board are described in section 6.3.1.

6.3.1 Introduction to DNV6F6PCIe board

The DNV6F6PCIe is configured with 6 Xilinx Virtex-6, SX475Ts. Each SX475T contains 2,016, multipliers, each 25x18, per FPGA. The DNV6F6PCIe contains 12,096 multipliers in addition to more than 21 million gates of ASIC logic, making this FPGA board ideal for heavy DSP-based algorithmic acceleration and High Performance Computing (HPC) applications. There is an on-board Marvell MV78200 CPU from the Discovery Innovation CPU family. The DNV6F6PCIe can be hosted via PCIe, USB, or Ethernet where FPGA configuration occurs via the host under the control of one of the Marvell CPUs. If the board is used stand-alone, the FPGA configuration files are copied onto a USB stick and FPGA configuration occurs at power up after the Marvell processors have booted.

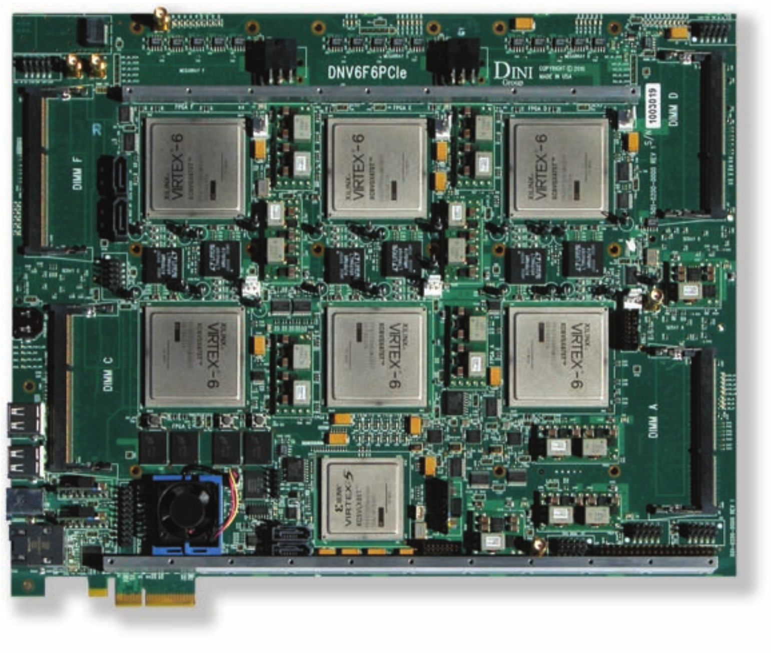

Figure 6.1. DiniGroup 6-FPGA board

6.3.2 Resource Utilization

The resource utilization of the hardware implementations synthesized on a Virtex-6 SX475T FPGA is presented in the following sections.

6.3.2.1 Sigmoid Operator

The utilization of slice registers, LUTs, block RAMs and DSP48Es on a single FPGA by the sigmoid implementation using the initial and optimized frameworks are presented in Tables 6.1 and 6.2 respectively. The optimized framework utilizes about half of the logic and block RAMs used by the initial framework and takes advantage of in-built DSP units on the FPGA.

6.3.2.1.1 Implementation using initial framework The utilization of resources by the sigmoid implementation using the initial framework is presented in Table 6.1.

6.3.2.1.2 Implementation using optimized framework The utilization of resources by the sigmoid implementation using the optimized framework is pre-

Table 6.1. Resource utilization of implementation of sigmoid operator using initial framework

Logic Utilization	Used	Utilization
Slice Registers	9812	1%
Slice LUTs	6557	2%
Block RAM/FIFOs	26	2%
DSP48Es	0	0%

sented in Table 6.2.

Table 6.2. Resource utilization of implementation of sigmoid operator using optimized framework

Logic Utilization	Used	Utilization
Slice Registers	4290	0%
Slice LUTs	3705	1%
Block RAM/FIFOs	3	0%
DSP48Es	14	0%

6.3.2.2 Shunting Image Operator

While maintaining the same level of flexibility offered by the initial framework, the implementation of shunting image operator using the optimized framework requires significantly lesser FPGA resources as compared to the implementation using the initial framework. Tables 6.3 and 6.4 show that the implementation using the optimized framework requires less than one-fifth of the logic utilized by the initial framework.

6.3.2.2.1 Implementation using initial framework The utilization of resources by the shunting image operator implementation using the initial framework is presented in Table 6.3.

Table 6.3. Resource utilization of implementation of shunting image operator using initial framework

Logic Utilization	Used	Utilization
Slice Registers	59863	10%
Slice LUTs	56588	19%
Block RAM/FIFOs	52	4%
DSP48Es	14	0%

6.3.2.2.2 Implementation using optimized framework The utilization of resources by the shunting image operator implementation using the optimized framework is presented in Table 6.4.

Table 6.4. Resource utilization of implementation of shunting image operator using optimized framework

Logic Utilization	Used	Utilization
Slice Registers	12892	2%
Slice LUTs	9221	3%
Block RAM/FIFOs	27	2%
DSP48Es	26	1%

6.3.2.3 Retinal-LGN Processor

The logic utilization of the implementation of the biologically-inspired retinal-LGN architecture using both frameworks is presented in Tables 6.5 and 6.6. The implementation using the initial framework requires more resources than what are available on a single FPGA, whereas, the implementation using the optimized framework requires less than 50% of the resources of one FPGA.

6.3.2.3.1 Implementation using initial framework The utilization of resources by the retinal-LGN implementation using the initial framework is presented in Table 6.5.

Table 6.5. Resource utilization of implementation of retinal-LGN processor using initial framework

Logic Utilization	Used	Utilization
Slice Registers	480828	81%
Slice LUTs	516325	173%
Block RAM/FIFOs	598	56%
DSP48Es	140	7%

6.3.2.3.2 Implementation using optimized framework The utilization of resources by the retinal-LGN implementation using the optimized framework is presented in Table 6.6.

Table 6.6. Resource utilization of implementation of retinal-LGN processor using optimized framework

Logic Utilization	Used	Utilization
Slice Registers	116656	19%
Slice LUTs	120425	40%
Block RAM/FIFOs	434	40%
DSP48Es	390	19%

6.3.3 Performance and frame-rate

The total execution time of Retinal-LGN processor for an image size of 2048×1536 on a Xilinx Virtex-6 SX475T FPGA operating at 100 MHz was 31.45 ms, which translates to approximately 31 frames per second.

For the input image shown in Figure 6.2, the output images from the FPGA implementation are shown in Figures 6.3, 6.4, 6.5 and 6.6.

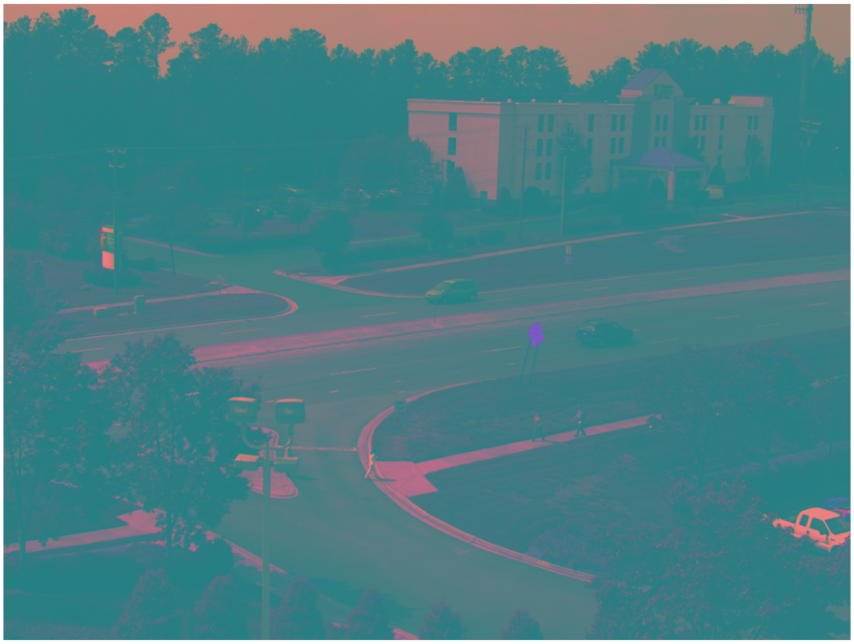

Figure 6.2. Input YIQ image to retinal-LGN processor

6.4 Comparison with CPU

The bio-inspired algorithm was implemented using C and OpenCV 2.3.1 on a Intel Core 2 Quad 2.4Ghz CPU based workstation with 4GB RAM. The total

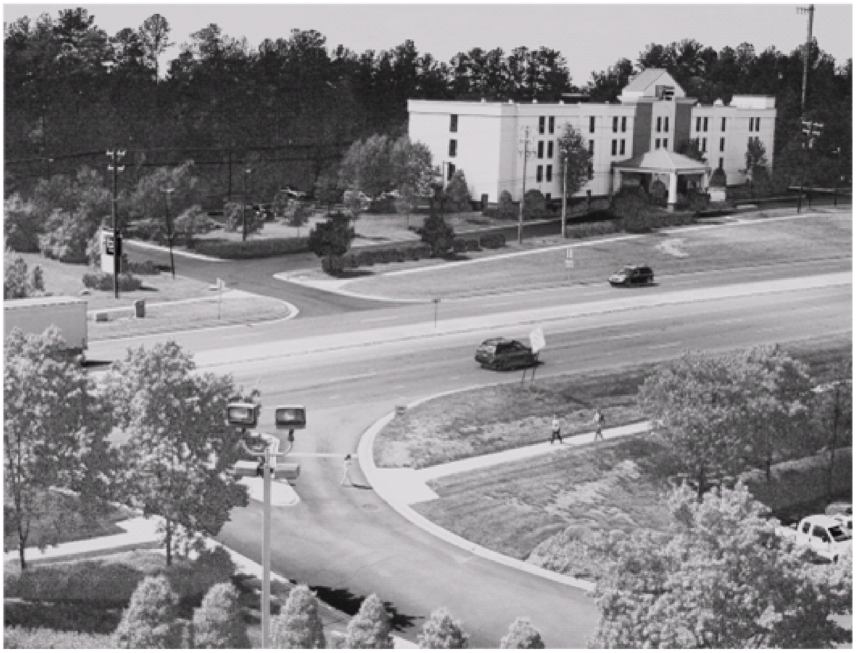

Figure 6.3. LGN1 output of retinal-LGN processor

execution time for an image of size 2048×1536 pixels was 435 ms, which is over 10 times greater then the execution time of the same image on an FPGA. A graph depicting the execution time of the algorithm on a CPU for different image sizes is shown in Figure 6.7.

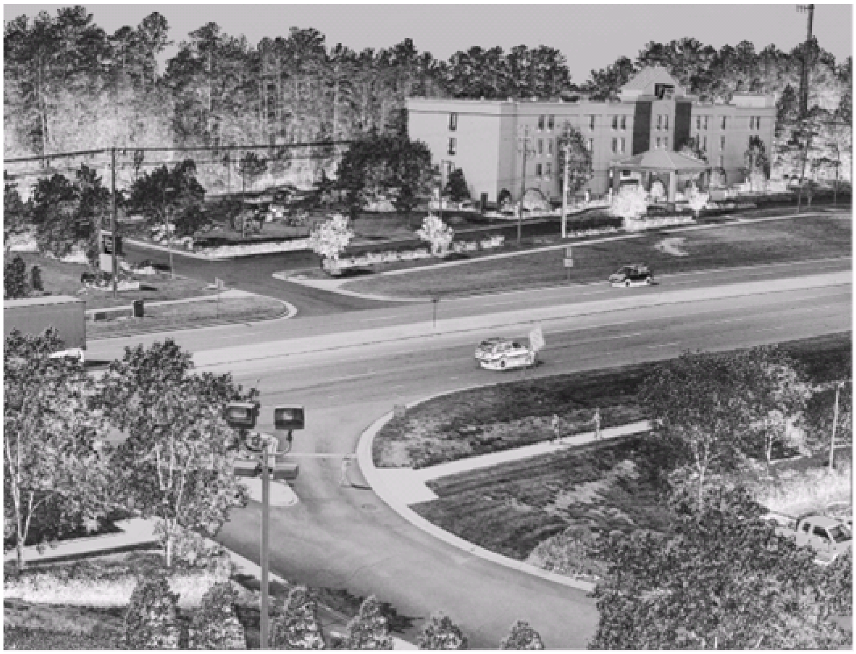

Figure 6.4. LGN2 output of retinal-LGN processor

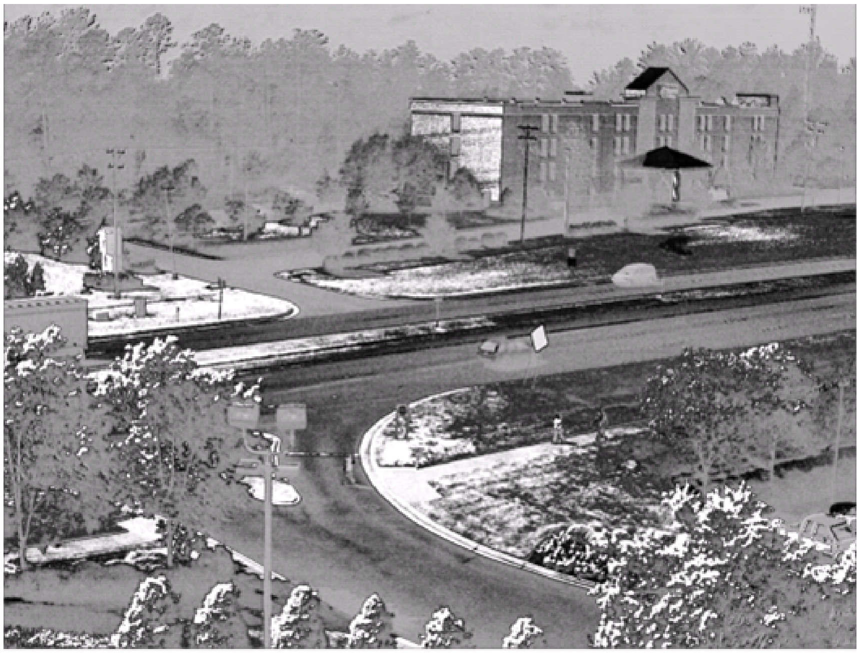

Figure 6.5. LGN3 output of retinal-LGN processor

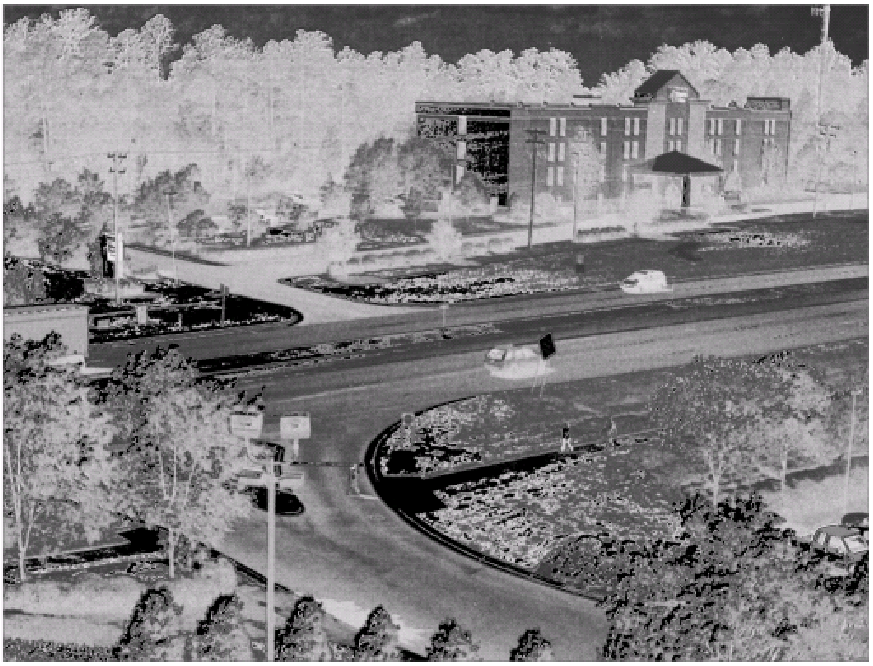

Figure 6.6. LGN4 output of retinal-LGN processor

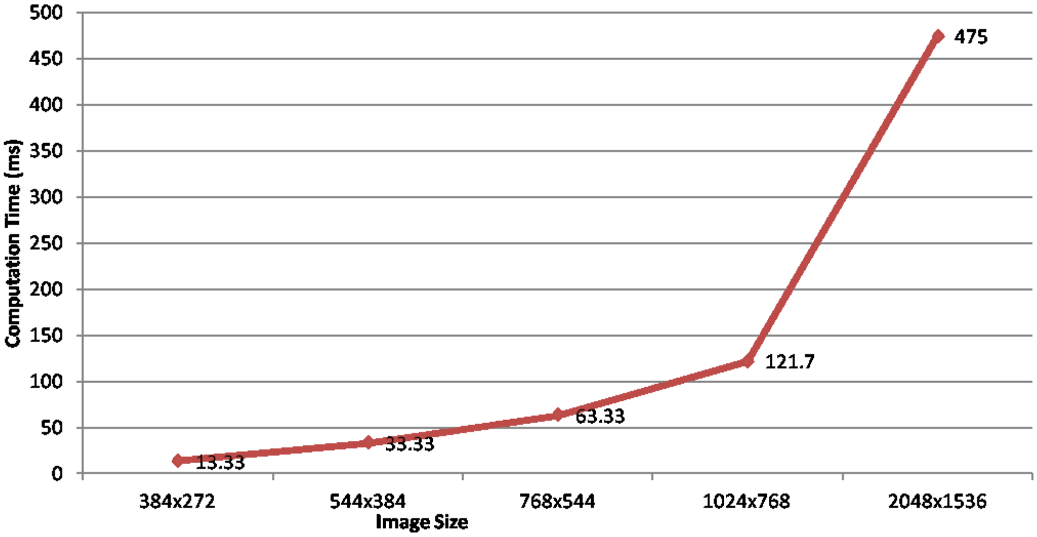

Figure 6.7. Graph of CPU execution time vs image size for retinal-LGN processor

Chapter 7

Conclusion

In this thesis, a fine-grained dataflow framework for complex processing tasks on reconfigurable streaming accelerators has been proposed. A dataflow library has been created to provide basic arithmetic and logical operators such as adders, subtractors and multipliers. In addition, the library also provides image processing operators such as convolution, scaling and histogram computation. Furthermore, the proposed framework also provides the capability to combine these operators effectively for a solution. Such an approach has allowed recursive definitions of custom operators using the library operators and the framework. A bio-inspired vision processing algorithm was been implemented on an FPGA as a use-case for the framework and experimental results have been presented. The algorithm was also implemented on a multi-core CPU and experimental results show that the FPGA-based implementation outperforms the CPU-based implementation by an order of 10.

7.1 Future Work

The scope of future work based on this thesis is immense. Possible avenues are as follows:

- A software framework that maps algorithms to the hardware library using dataflow programming is one area for research. Such a software framework will allow non-hardware developers to effectively utilize the hardware framework to implement complex algorithms on FPGAs.

- Another area for future research is optimizing the hardware framework to minimize power consumption. For example, in this thesis, all computational operators in an algorithm are constantly consuming power. However, there exist algorithms where some operators are unused during part of the computation and can thus be shut off. Optimizing the framework architecture to power down unused operators for a particular period of computation would allow fine-grained control over the power consumption of the design.

Bibliography

[1] Guo, Z., W. Najjar, F. Vahid, and K. Vissers (2004) "A quantitative analysis of the speedup factors of FPGAs over processors," in *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, ACM, pp. 162–170.

[2] George, V. and J. Rabaey (2001) *Low-energy FPGAs: architecture and design*, Springer Netherlands.

[3] Shang, L., A. Kaviani, and K. Bathala (2002) "Dynamic power consumption in Virtex-II FPGA family," in *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, ACM, pp. 157–164.

[4] DeHon, A. (2000) "The density advantage of configurable computing," *Computer*, **33**(4), pp. 41–49.

[5] Che, S., J. Li, J. Sheaffer, K. Skadron, and J. Lach (2008) "Accelerating Compute-Intensive Applications with GPUs and FPGAs," in *Application Specific Processors, 2008. SASP 2008. Symposium on*, IEEE, pp. 101–107.

[6] Asano, S., T. Maruyama, and Y. Yamaguchi (2009) "Performance comparison of FPGA, GPU and CPU in image processing," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, IEEE, pp. 126–131.

[7] Gac, N., S. Mancini, M. Desvignes, and D. Houzet (2008) "High speed 3D tomography on CPU, GPU, and FPGA," *EURASIP Journal on Embedded systems*, **2008**, p. 5.

[8] Cope, B., P. Cheung, W. Luk, and S. Witt (2005) "Have GPUs made FPGAs redundant in the field of Video Processing?" in *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, IEEE, pp. 111–118.

[9] Chase, J., B. Nelson, J. Bodily, Z. Wei, and D. Lee (2009) "Real-time optical flow calculations on FPGA and GPU architectures: A comparison study," in *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on*, IEEE, pp. 173–182.

[10] Ashenden, P. (2008) *The designer's guide to VHDL*, vol. 3, Morgan Kaufmann.

[11] Skahill, K. (1996) *VHDL for programmable logic*, Addison-Wesley Longman Publishing Co., Inc.

[12] Palnitkar, S. (2003) *Verilog HDL: a guide to digital design and synthesis*, vol. 1, Prentice Hall PTR.

[13] Thomas, D. and P. Moorby (2002) *The Verilog hardware description language*, vol. 1, Springer Netherlands.

[14] Dennis, J. and D. Misunas (1975) "A preliminary architecture for a basic data-flow processor," in *Proceedings of the 2nd annual symposium on Computer architecture*, ACM, pp. 126–132.

[15] Davis, A. (1978) "The architecture and system method of DDM1: A recursively structured Data Driven Machine," in *Proceedings of the 5th annual symposium on Computer architecture*, ACM, pp. 210–215.

[16] Dennis, J. (1980) "Data flow supercomputers," *Computer*, **13**(11), pp. 48–56.

[17] Watson, I. and J. Gurd (1982) "A practical data flow computer," *Computer*, **15**(2), pp. 51–57.

[18] Johnston, W., J. Hanna, and R. Millar (2004) "Advances in dataflow programming languages," *ACM Computing Surveys (CSUR)*, **36**(1), pp. 1–34.

[19] Farabet, C., B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello (2010) "Hardware accelerated convolutional neural networks for synthetic vision systems," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, IEEE, pp. 257–260.

[20] Farabet, C., C. Poulet, J. Han, and Y. LeCun (2009) "Cnp: An fpga-based processor for convolutional networks," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, IEEE, pp. 32–37.

[21] Farabet, C., C. Poulet, and Y. LeCun (2009) "An FPGA-based stream processor for embedded real-time vision with convolutional networks," in *Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on*, IEEE, pp. 878–885.

[22] Janneck, J., I. Miller, D. Parlour, G. Roquier, M. Wipliez, and M. Raulet (2008) "Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study," in *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, IEEE, pp. 287–292.

[23] Draper, B., J. Beveridge, A. Bohm, C. Ross, and M. Chawathe (2002) "Implementing image applications on FPGAs," in *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, vol. 3, IEEE, pp. 265–268.

[24] Rinker, R., M. Carter, A. Patel, M. Chawathe, C. Ross, J. Hammes, W. Najjar, and W. Bohm (2001) "An automated process for compiling dataflow graphs into reconfigurable hardware," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, **9**(1), pp. 130–139.

[25] Thomas, D., L. Howes, and W. Luk (2009) "A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation," in *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, ACM, pp. 63–72.

[26] Cope, B., P. Cheung, and W. Luk (2007) "Bridging the gap between FPGAs and multi-processor architectures: A video processing perspective," in *Application-specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, IEEE, pp. 308–313.

[27] Kuon, I., R. Tessier, and J. Rose (2008) "Fpga architecture: Survey and challenges," *Foundations and Trends® in Electronic Design Automation*, **2**(2), pp. 135–253.

[28] Ahmed, T., P. D. Kundarewich, J. H. Anderson, B. L. Taylor, and R. Aggarwal (2008) "Architecture-specific packing for virtex-5 FPGAs," in *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, FPGA '08, ACM, New York, NY, USA, pp. 5–13. URL `http://doi.acm.org/10.1145/1344671.1344675`

[29] Dou, Y., S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev (2005) "64-bit floating-point FPGA matrix multiplication," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, FPGA '05, ACM, New York, NY, USA, pp. 86–95. URL `http://doi.acm.org/10.1145/1046192.1046204`

[30] Jiang, J., V. Mirian, K. P. Tang, P. Chow, and Z. Xing (2009) "Matrix Multiplication Based on Scalable Macro-Pipelined FPGA Accelerator Architecture," *Reconfigurable Computing and FPGAs, International Conference on*, **0**, pp. 48–53.

[31] Hubel, D., J. Wensveen, and B. Wick (1988) *Eye, brain, and vision*, Scientific American Library New York.

[32] Hubel, D. (1963) “The visual cortex of the brain.” *Scientific American.*

[33] Martin, P. (1998) “Colour processing in the primate retina: recent progress,” *The Journal of physiology*, **513**(3), pp. 631–638.

[34] Roorda, A. and D. Williams (1988) “The arrangement of the three cone classes in the living human eye,” *Acoust. Soc. Am*, **83**, pp. 1102–1116.

[35] Grossberg, S. (1995) “The attentive brain,” *American Scientist*, **83**(5), pp. 438–449.

[36] Kuffler, S. (1953) “Discharge patterns and functional organization of mammalian retina,” *Journal of Neurophysiology*, **16**(1), p. 37.

[37] Dong, D. and J. Atick (1995) “Temporal decorrelation: a theory of lagged and nonlagged responses in the lateral geniculate nucleus,” *Network: Computation in Neural Systems*, **6**(2), pp. 159–178.