The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

SYSTEM-ON-CHIP INTEGRATION OF HETEROGENEOUS ACCELERATORS FOR PERCEPTUAL COMPUTING

A Dissertation in

Computer Science and Engineering

by

Sungho Park

© 2013 Sungho Park

Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

December 2013
The dissertation of Sungho Park was read and approved\(^1\) by the following:

Vijaykrishnan Narayanan  
Professor of Computer Science and Engineering  
Advisor  
Chair of Committee

Mary Jane Irwin  
Professor of Computer Science and Engineering

Chita R. Das  
Professor of Computer Science and Engineering

Dongwon Lee  
Associate Professor of Information Sciences and Technology

Kevin M. Irick  
Special Member

Lee Coraor  
Associate Professor of Computer Science and Engineering  
Graduate Officer for the Department of Computer Science and Engineering

\(^1\)Signatures on file in the Graduate School.
Abstract

Traditional microprocessor design has seen radical shifts over the past few years. The challenges of excessive power consumption led to the shift from faster and more complex processors to multiple cores on the same chip. More recently, there has been a growing trend towards integrating multiple customized cores instead of homogeneous arrays of processors. The distinction between embedded heterogeneous System-on-Chip (SoC) Architectures and mainstream processor architectures is blurring. A key challenge in both these domains is to efficiently integrate these accelerators in a single chip.

This dissertation contributes towards making the design of system-on-a-chip architectures more flexible, more programmable, and easier to develop and verify. Specifically, a communication and interface framework to integrate heterogeneous accelerators for this domain is proposed. This framework has been incorporated to develop SoC designs for two different perceptual computing applications, visual perception and wireless body-area networks (WBANs). Perceptual computing applications perceive intent by sensing and monitoring different activities of a person and their environments. To support visual perception, a system for detecting, tracking and recognizing objects has been built using the proposed framework. A system has also been developed for supporting compressed sensing of medical signals from the human body for perceptual medical diagnostic applications. These two frameworks demonstrate the flexibility of the framework to compose different systems.

This dissertation also contributes to the design of approximate computing techniques for design of energy-efficient systems. These techniques leverage the programmable aspect of the
proposed communication/interface framework. First, the complexity of computation is varied based on relative salience of an object in a visual scene to expend non-uniform effort on an entire scene while providing a quality of output similar to expending same effort across the scene. Second, mathematical approximations are employed to reduce the effort of computation for reconstruction of compressed signals without significant loss of accuracy.

The proposed framework has also been validated through adoption by other researchers in their SoC integration efforts. This research opens new directions in dynamic configuration of accelerators that will form part of future research.
# Table of Contents

List of Tables .............................................................................. x

List of Figures ........................................................................... xii

Acknowledgments ....................................................................... xv

Chapter 1. Introduction ................................................................. 1

Chapter 2. Visual Perception on Vortex Framework ................... 3

  2.1 Introduction ........................................................................... 3

  2.2 Related Works ....................................................................... 4

  2.3 Design Methodology and Architectural Approaches ............... 7

    2.3.1 Requirements for Interconnection Network ...................... 7

      2.3.1.1 Flexibility ................................................................. 8

      2.3.1.2 Scalability ................................................................. 8

      2.3.1.3 Programmability ....................................................... 8

      2.3.1.4 High Bandwidth ....................................................... 9

    2.3.2 Proposed Architectural Approach for Interconnection Network ... 9

      2.3.2.1 Communication Fabric ........................................... 10

      2.3.2.2 Vortex: A Framework to Integrate Heterogeneous Accelerators for Perceptual Computing ................................. 11

      2.3.2.3 Meeting Neuromorphic Vision Requirements ............. 25
2.3.3 Requirements for Customized Accelerators

2.3.3.1 Exploiting Parallelism

2.3.3.2 Power Efficiency

2.3.3.3 Highly Parameterizable Design

2.3.3.4 Composability and Programmability

2.3.4 Architectural Details of Customized Accelerators

2.3.4.1 SAP Processing Element (PE)

2.3.4.2 Composability (ChipMONK for SOPs)

2.3.5 The Realization of the Neuromorphic SoC

2.4 Case Study

2.4.1 Retina Preprocessing

2.4.2 Visual Saliency (AIM)

2.4.3 Feature Extraction (HMAX)

2.5 Experimental Setup and Results

2.5.1 Classification Accuracy

2.5.2 Performance

2.5.3 Discussion of Performance Results

2.6 Highlights in Vortex and Comparisons to other system

2.6.1 Five major aspects in Vortex

2.6.1.1 Intuitive dataflow mapping with abstraction

2.6.1.2 Latency Hiding

2.6.1.3 Composability and Reusability

2.6.1.4 2D DMA, built in network interface under abstraction
Chapter 4. Acceleration of Signal Reconstruction by Compressed Sensing for Body Signals

4.1 Introduction .................................................................................................................. 90

4.2 Optimizations and Approximations for Implementing Hardware Accelerators .......... 91

4.2.1 BSBL-BO Algorithm ................................................................................................. 91

4.2.2 Computing $PBP$ ....................................................................................................... 92

4.2.3 Matrix Multiplication (MM) ....................................................................................... 95

4.2.4 Computing $b$ ............................................................................................................. 95

4.2.4.1 $\Theta_{iq}$ ................................................................................................................... 98

4.2.4.2 $\Omega_{iq}$ .................................................................................................................. 99

4.3 Micro Architecture of Accelerators ............................................................................... 102

4.3.1 PBP .......................................................................................................................... 102

4.3.2 Matrix Multiplication (MM) ....................................................................................... 103

4.3.3 Compute $b$ ................................................................................................................. 104

4.3.3.1 Compute $\Theta$ ......................................................................................................... 105

4.3.3.2 Compute $\mu_X$ ........................................................................................................ 106

4.3.3.3 Compute $\Omega$ ........................................................................................................ 107

4.4 Experimental results .................................................................................................... 107

4.4.1 The complete hardware-acceleration system .............................................................. 108

4.4.2 Accuracy, power, and performance for EEG application ........................................ 110

4.5 Conclusion .................................................................................................................... 112
Chapter 5. Conclusion and Future Work ........................................... 114

Bibliography ..................................................................................... 117
## List of Tables

2.1 Various Transaction Types ........................................... 14
2.2 A summary of resource utilization of the router with various number of bidirectional ports on the Xilinx XCV6SX475T FPGA device ......................... 24
2.3 A summary of resource utilization of NIF-SAP with various number of handler-pair on the Xilinx XCV6SX475T FPGA device .......................... 24
2.4 A summary of resource utilization of NIF-SOP with various number of output channels on the Xilinx XCV6SX475T FPGA device ....................... 24
2.5 A summary of resource utilization for neuromorphic retina processor ........ 40
2.6 Comparisons among architectural approaches to implement AIM .................. 42
2.7 Schedule of timeline to operate AIM .................................. 46
2.8 A summary of resource utilization for AIM accelerator on XCV6SX475T ....... 47
2.9 A summary of resource utilization for the HMAX accelerator on XCV6SX475T 49
2.10 List of Caltech101 categories used in the experiments .......................... 51
3.1 Exploration result on various orientation parameters in HMAX ................. 81
3.2 Exploration result on various scale parameters in HMAX ........................ 82
3.3 Exploration result on various coverage of prototypes in HMAX ................. 83
3.4 Sample configurations .................................................. 85
3.5 Results of sample configurations .................................... 86
3.6 Top 3 energy-efficient configuration for accuracy-critical applications (Target accuracy = 82%) .................................................. 86
3.7 Results of top 3 configurations for accuracy-critical applications (Target accuracy = 82%) .......................... 87

3.8 Top 3 energy-efficient configuration for energy-critical applications (Target accuracy = 78%) ..................... 87

3.9 Results of top 3 configurations for energy-critical applications (Target accuracy = 78%) ..................... 87

4.1 Resource consumption on three FPGAs(V6SX475T) ........................................ 110

4.2 Evaluation of hardware acceleration system for 64-channel EEG signals .... 112
## List of Figures

2.1 Example Dataflow Graph ........................................... 7
2.2 Accelerator System Topologies and Configurations ................ 13
2.3 Diagram of NIF-SAP Architecture .................................. 15
2.4 Diagram of NIF-SOP attached with Custom SOP .................. 18
2.5 Example Dataflow on Vortex ...................................... 19
2.6 Transaction Scenarios with and without NIF-MEM ............... 20
2.7 Diagram of NIF-MEM ........................................... 21
2.8 Example Dataflow Graph (1-to-N, N-to-1) ....................... 22
2.9 Illustration of ROI and terms for Window Transfer .............. 22
2.10 Window Transfer Table ......................................... 23
2.11 Multi-channel SOP ............................................ 25
2.12 SAP-PE $\mu$ Architecture ..................................... 33
2.13 Hierarchical composition of an SOP ............................. 34
2.14 Neuromorphic system for visual processing ..................... 36
2.15 Multi-FPGA mapping of visual processing system .............. 37
2.16 Retina processor pipeline ....................................... 38
2.17 Dataflow graph representation of AIM .......................... 43
2.18 Scale-Concurrent AIM architecture ............................... 44
2.19 Block Diagram of AIM architecture ............................... 45
2.20 A computational template of HMAX model showing the HMAX stages ............................... 48
2.21 Interaction between the host and HMAX accelerator .......................... 49
2.22 Experimental setup of visual perception system ................................. 50
2.23 Object classification accuracy for a number of accelerated HMAX configurations. 52
2.24 A comparison of performance between CPU and accelerated retina processor 53
2.25 A comparison of performance between GPU and accelerated AIM processor 54
2.26 A comparison of performance between CPU, GPU, and accelerated HMAX for two configurations ................................................................. 56
2.27 A comparison of power efficiency between CPU, GPU, and accelerated HMAX for two configurations ................................................................. 57
2.28 A Dataflow Graph targeted on Imagine Stream Processor [28] ............... 61
2.29 Architecture of Imagine Stream Processor [28] .................................. 62
2.30 Flow Diagram, introduced in [42] ......................................................... 65
2.31 Architecture of BONE-V5 [42] .......................................................... 67
2.32 CogniServe accelerator programming and communication: device driver interface (a) and virtual memory-based acceleration (b) [26] ......... 68
3.1 Typical processing flow across major computational blocks in object recognition system ................................................................. 72
3.2 5 patches and their rankings suggested by the saliency algorithm on a natural scene ............................................................. 75
3.3 Saliency Confidence Score over rankings of salient regions for 900 annotated images from TU Graz-02 Database ....................................... 80
3.4 Analysis of Accuracy to Energy Ratio for design space of HMAX .......... 84
3.5 Accuracy loss $\Delta$ and Energy savings of top 3 configurations for accuracy-critical and energy-critical applications

4.1 Pseudo-code of BSBL-BO Algorithm

4.2 Approximation to compute $PBP$

4.3 Software Execution Time Analysis of BSBL-BO Algorithm

4.4 Approximation to compute $\Theta$

4.5 Approximation to compute $\Omega$

4.6 Micro-architecture of $PBP$ accelerator

4.7 Micro-architecture of matrix multiplication accelerator

4.8 Micro-architecture of Compute$_b$ accelerator

4.9 Elements of $H\Phi\{i,i\}$ which involve in the computation of $\Theta_{i0}$

4.10 Micro-architecture of Compute$_\Theta$ module

4.11 Micro-architecture of Compute$_{\mu_s}$ module

4.12 Micro-architecture of Compute$_\Omega$ module

4.13 System composition on three FPGAs

4.14 Accuracy of hardware acceleration for one channel of EEG signal
Acknowledgments

I have been extremely lucky to meet these three persons since I came here across the Pacific Ocean: Dr. Vijay, Dr. Irick and Dr. Maashri. To Dr. Vijay, million thanks to you. I sincerely appreciate your consistent support and guidance. You are the one who actually made me stand here at this moment. To Dr. Kevin Irick, your inspiration and enthusiasm have affected me too much. I’ll miss the days and nights at the lab with pizza and you! To Dr. Ahmed Al Maashri, you had been my true friend here and always tried to help me in any situations, including when I was trying to find an used mini-van and when I had to pull off three teeth at a time. Sorry for the spicy kimchi that I prepared for you without knowing that you might freak out.

To all my research committees, Dr. Irwin, Dr. Das, Dr. Lee, and Dr. Irick: Your advise and guidance finally helped shape me up as who I am now. Your kind help will never be forgotten! Thank you again!

To my mom and dad, who have been supporting me throughout my whole life: My school years are almost over, finally! Let me try to support you back from this point on.

To my lovely wife, Eun Jeong: You have been my best friend and supporter for the last 16 years already. I truly appreciate your sacrifice for my 5 years of Ph.D. life with two children. It’s never going to be forgotten, and I know it’s time to pay them back. To my daughter, Siyeon: You were only an 18-month-old baby when I started my study, but now you are already almost a 7-year-old girl. I still remember you sleeping in the narrow bassinet in the airplane that brought us here from Korea. Thank you for growing nice and pretty. To my son, Logan: You were not
even here when I started, but you are already a 4-year-old boy! Thank you as well for being nice and gentle. To my third one in his/her mommy’s belly: You are finally the only child of mine who won’t see his dad going to school every morning! Lucky! I cannot wait to see you!

To all the MDLers including Mike, Peter, Aarti, Matt, Yang, Misun, Siddharth, Nandhini, Huichu, Chris, Moonseok: It was very nice to meet and work with all of you. Wish you the best, and I will definitely miss the time we spent together, here at 351 IST. (The only problem of 351 is there’s no window to see if it’s raining or snowing outside!!)

To my brother: Wish the best for your starting school years! Good luck with your master’s at UIUC!

To all my families, relatives, and friends in Korea: Your constant support and pray definitely encouraged me to arrive here. Wish we could enjoy our re-union sooner or later.
Chapter 1

Introduction

The rapid evolution of CMOS technology has resulted in mobile devices such as smartphones and tablets becoming more ubiquitous and essential in modern life. Consequently applications that were once targeted for powerful laptop and desktops are expected to function equally well on mobile devices. Emerging applications such as ultra-high definition video streaming, real-time augmented reality, and advanced biometric recognition will be standard on mobile devices in the near future. Because of their potential for high energy efficiency and throughput, domain specific accelerators are indispensable in satisfying the performance and quality requirements of such applications executing on energy constrained mobile devices.

However, hardware-based accelerators have been generally regarded as inflexible, non-programmable, difficult to develop and verify, and impossible to mend after production. This dissertation aims at reshaping this thinking about design of custom accelerators. Specifically, this dissertation introduces Vortex [46], a reconfigurable communication platform suitable for composing accelerators into systems. Its main distinguishing attributes are the optimized network interfaces that ease the mapping of dataflow graphs onto SoCs. The network interfaces provide a transport layer on top of a packet-switched NoC to support frame-level transactions while abstracting the underlying physical interconnection. Further, it maps accelerators into two categories: Switch Attached Processor (SAP) and Streaming OPerator (SOP) to provide a scalable framework for incorporating different accelerators.
The proposed Vortex framework has been used to develop two different SoC designs to demonstrate the effectiveness of the proposed features. An object detection, tracking and recognition system has been designed using this framework for supporting visual perception applications. The framework has also been applied for designing a system for signal reconstruction in a medical body-area-network. In the design of these systems, novel energy and performance optimizations have been proposed leveraging the programmability of the SoC framework.

The rest of this document is organized as follows. Chapter 2 introduces the proposed framework to compose heterogeneous accelerators. This chapter also shows how this framework is used to design a SoC for visual perception, and compares to other perceptual systems. Further energy optimization on the visual perception system is discussed in chapter 3 to highlight the programmability of the framework. Finally, we show the flexibility of the framework by applying it to build an acceleration system for WBANs application in chapter 4.
Chapter 2

Visual Perception on Vortex Framework

2.1 Introduction

While machine vision research has improved multi-fold in recent decades, it still falls short of the capabilities and efficiencies of the primate visual cortex. The primate brain excels at comprehending and interacting with complex natural environments. In energy use, the brain is estimated to consume 20 Watts with all of its functionality including complex scene understanding. While there is much consensus on the superiority of neuromorphic vision systems over machine vision on most vision tasks, debates continue over which computational approaches might lead to better efficiencies and flexibility akin to the visual cortex. Consequently, there are significant ongoing algorithmic advances emerging in both neuroscience and machine vision.

Along with algorithmic advances, hardware fabrics that realize these algorithms efficiently are essential for achieving the speed and energy efficiencies of the brain. Current processing elements based on general purpose processors and Graphics Processing Units, GPUs, often do not meet the performance and power constraints of embedded vision applications. This has triggered interest in the design of domain-specific accelerators that support a broad set of high-level vision algorithms. The availability of low-power, high-speed and high-accuracy vision systems that detect and recognize, can enable a variety of embedded applications in health care, surveillance, automobiles and e-business. Consequently, there is an emergence of customized System-on-Chip, SoC, designs that support neuromorphic vision algorithms. While
there are active works on realizing brain-like hardware fabrics based on analog neuronal arrays and synaptic cross-bars, this work is focused on realizing hardware using digital accelerators.

In this chapter, I present a design framework along with associated design automation tools to facilitate the development of neuromorphic vision systems. First, we focus on the communication architecture required to support streaming data computations in vision applications. Next, we identify and design the computational primitives that mimic the operations of various stages of the visual cortex. Finally, we present customized accelerators that implement various stages of the visual cortex — giving support to object detection and recognition applications. The results indicate that domain-specific accelerators offer a promising approach to bridging the gap between efficiencies of digital hardware vision systems and the brain.

The rest of this chapter is organized as follows: Section 2.2 discusses related work. Section 2.3 proposes guidelines to be considered when mapping neuromorphic algorithms to hardware. Section 2.4 details the implementation of a neuromorphic vision system mapped to a multi-FPGA platform while section 2.5 presents experimental results of the system.

2.2 Related Works

There has been a thread of research that focuses on identifying characteristics of media applications in general and methods to achieve efficient acceleration of these applications. Numerous academic and commercial systems have been developed as the fruit of these research efforts. The Imagine stream processor[44] adapts the stream programming model by tailoring a bandwidth hierarchy to the demands of the particular media application. By passing a stream from a central register file through an array of 48 32-bit floating-point arithmetic units, it targets the parallelism and locality of such applications. Similar to [44], Storm-1[29] defines a
stream processor that treats streams and kernels as part of the instruction-set architecture (ISA) to exploit data parallelism and manage on-chip memories. Both approaches map data streams onto many kernels executing concurrently but execution remains inefficient due to load/store instructions that do not perform actual computation.

While [44][29] have developed stream processors for bandwidth-efficient media processing utilizing clusters of ALUs that process large data streams, [53] analyzed and profiled major applications in media processing to reveal performance bottlenecks. Their analysis found that about 80 percent of dynamic instructions were supporting instructions to feed the computational units rather than meaningful computational instructions. In order to overcome such inefficiency, the MediaBreeze architecture was introduced with more customized hardware support for address generation, loop, and data reorganization. In summary, [53] focused on improving the utilization of computational units rather than increasing the number of these units as done by [44][29].

In [30] the authors detail the implementation of a multi-object recognition processor on an SoC. They present a biologically inspired neural perception engine that exploits analog-based mixed-mode circuits to reduce area and power. Moreover, they utilize a Network-on-Chip, NoC, as the interconnection fabric among all cores. However, except for the visual attention engine and the vector matching processors, all other algorithm acceleration is performed on multiple SIMD processors executing software kernels.

A majority of the prior art significantly relies on software execution on a large number of processing cores. Still, the use of a control-oriented processing paradigm to implement naturally streaming applications such as neuromorphic vision has several limitations. First, the organization of processing units must be considered by the programmers when implementing the
kernels to maximize overall efficiency. Only programmers with intimate knowledge of both the application and the structure of the underlying hardware can expect to achieve efficient implementations. Second, the granularity of optimization remains at the instruction level — neglecting the efficiency achievable from domain-specific customized hardware and their associated ISA. Third, due to overheads associated with moving data and resolving data-dependencies, computational units are underutilized.

In summary, optimizations gained from gate-level customization yield higher energy-efficient realizations on an SoC when compared to coarse grain instruction-based architectures presented in other works.

This work focuses on algorithmic abstractions of the primate visual cortex and the corresponding implementations on digital CMOS substrate. In contrast, other works attempt to mimic the visual cortex using spiking neural networks and artificial synapse implementations [50][55]. These approaches employ specialized analog circuitry to implement the computational components of the brain model. Unlike digital circuits, these analog systems utilize power transistors and lower supply voltages which results in highly specialized circuits that are extremely sensitive to parameter selection and variation [3]. Since many of the models of the visual cortex are still being discovered and refined, the stringent constraints of analog design do not allow free exploration of model parameters once the substrate has been created. In systems consisting of both analog and digital components, converting stimuli and response between the domains can be extremely challenging and is still an active research area. Compared to pure analog and mixed-signal approaches, our focus is to leverage CMOS technology scaling to circumvent the design challenges while offering a high-degree of exploration and configurability.
2.3 Design Methodology and Architectural Approaches

2.3.1 Requirements for Interconnection Network

Support for streaming dataflows is fundamental towards implementing neuromorphic vision algorithms, where most processing procedures have a directly inferable dataflow graph representation. An example of a system-level dataflow graph is shown in Fig. 2.1.

![Dataflow Graph Diagram](image)

Fig. 2.1: Example Dataflow Graph

Each node in the graph represents an atomic operation, or process, that accepts data from one or more input channels and produces data into one or more output channels. Processes can be defined hierarchically; meaning its operation is defined by another dataflow graph consisting of processes and channels. The following subsections discuss the communication infrastructure requirements that allow efficient mapping of these dataflow processes to an SoC.
2.3.1.1 Flexibility

The communication infrastructure must be flexible enough for various functional processes to be instantiated without affecting one another. Each process has specific requirements including number of input and output channels, input and output data-rates, and data format (i.e. fixed-point representation of data elements). To support the large dimension of requirements imposed by participating processes, the communication system must provide flexibility at the appropriate granularity without becoming a significant resource and performance bottleneck.

2.3.1.2 Scalability

Dataflow representations vary in size ranging from tens of nodes to hundreds of nodes. Moreover, the fan-in and fan-out of each node can have large variations. This variation is present at every recursive level of the graph hierarchy. Therefore, the communication infrastructure must be scalable enough to support the large disparities in graph structure, while maintaining uniformity of performance across configurations.

2.3.1.3 Programmability

Exploration of neuromorphic vision reveals that many of the models consist of processes whose behaviors change over time in response to input stimuli, environmental variations, or top-down task directives. For example, coarse scale Gabor feature extraction is utilized when performing time-critical salient region detection. Conversely, feature extraction is performed at all scales for the subsequent process of region classification. Therefore, an algorithm may have multiple configurations that exhibit different functionality, accuracy, and performance. One could create a unique dataflow graph for each of the configurations; however the resulting system
would unnecessarily consume an abundance of resources. This is especially true in the common case that no more than one dataflow configuration is active at any instant. A more resource efficient approach reconfigures the dataflow graph as necessary to match the appropriate configuration. The underlying communication infrastructure must therefore allow runtime structural adaptation to dynamic processing requirements.

2.3.1.4 High Bandwidth

Real-time requirements impose bandwidth constraints on constituent algorithm processes. In these bandwidth-sensitive vision applications, the additive bandwidth demands of parallel processes dictates whether performance objectives are met. For example, scale invariant algorithms, such as feature extraction, generate large sets of intermediate feature maps corresponding to each level of an image pyramid. Each of these feature maps targets memory and shares the bandwidth of the underlying communication infrastructure. The communication infrastructure must have sufficient aggregate bandwidth to ensure system performance constraints are met.

2.3.2 Proposed Architectural Approach for Interconnection Network

In consideration of the aforementioned requirements, qualitative comparisons among major communication mechanisms are discussed in Section 2.3.2.1. Subsequent sections introduce Vortex[45]: a reconfigurable communication platform suitable for composing neuromorphic vision systems.
2.3.2.1 Communication Fabric

Shared buses are common communication mechanisms used in SoC design. ARM’s AMBA[4] and IBM’s CoreConnect[25] are popular shared-bus architectures deployed in many SoCs. Shared buses moderately satisfy the flexibility requirement by providing uniform master and slave access interfaces to any Intellectual Property, IP, core that conforms to the bus protocol. The dataflow from a source node to a target node can be accomplished by the source node issuing a write into the target node’s slave interface. Alternatively, a centralized DMA engine can be tasked with reading data from the source node’s slave interface and writing to the target node’s slave interface. If the bus architecture supports broadcasting, then source nodes with multiple output channels can send data to multiple targets concurrently. Note, however, that multiple nodes concurrently sourcing a target node with multiple input channels is not supported. This is because shared buses do not allow writing by more than a single device simultaneously and generally limit communication to a single source and target pair at a time. Therefore the shared bus becomes the performance bottleneck in dataflow oriented processing.

The point-to-point communication approach is preferred when maximum efficiency is required in terms of both circuit area and power consumption. Point-to-point channels provide the most efficient communication fabric between neighboring nodes in a dataflow graph; particularly when the graph is simple with relatively low fan-in and fan-out per node. Point-to-point architectures achieve their efficiency by having structural attributes such as bit-width, operating frequency, and signaling protocol, appropriately set at design time; leading to optimal trade-offs between bandwidth and circuit area. However, as the complexity of the dataflow graph increases, the area and power consumption increases exponentially because each pair of neighboring nodes
must have dedicated point-to-point channels. Communication is only allowed between those pair
of cores that have static channels allocated between them at design time. Consequently, point-
to-point architectures suffer from lack of flexibility, programmability, and scalability. Still, if
utilized at the appropriate granularity, point-to-point communication can be effective in many
aspects of neuromorphic vision SoCs.

The NoC paradigm has gained significant attention as the number of heterogeneous cores
being integrated on a single SoC increases[31][16].

By transferring data packets across a system of interconnected switches, the NoC allows
communication between all devices that are capable of performing the necessary packetization
and depacketization functions at the expense of overheads related to additional routing infor-
mation and data serialization. NoCs utilize standardized interfaces to achieve the same level of
core interoperability as offered by bus architectures. However, the distributed interconnection
topology offers scalability well beyond that of shared bus and point-to-point communication ar-
chitectures. Programmability comes without additional burden on the communication fabric as
all cores may communicate with each other through the network. The performance of the NoC
often depends on the various parameters of the network such as topology, routing algorithm,
flit size, and buffer depths. To minimize communication latency and subsequently maximize
performance potential, careful consideration must be taken when selecting these parameters.

2.3.2.2 Vortex: A Framework to Integrate Heterogeneous Accelerators for Perceptual
Computing

Vortex is a reconfigurable and highly programmable NoC platform for vision applica-
tions with considerations to the requirements addressed previously. Although Vortex is based on
the NoC paradigm, its main distinguishing attributes are the optimized network interfaces that ease the mapping of dataflow graphs onto SoCs. The network interfaces provide a transport layer on top of a packet-switched NoC to support frame-level transactions while abstracting the underlying physical interconnection.

The analysis of various neuromorphic vision algorithms reveals the need for two categories of processing nodes: Switch Attached Processor (SAP) and Streaming OPerator (SOP). Accordingly, Vortex provides two types of network interfaces. Regardless of the type of attached network interface, Vortex uses a 16-bit device address, device-id, to refer to an interface attached to one of its ports.

A major contribution of the Vortex platform is the integrated network awareness and support for application flows. A flow describes any sequence of operations required to complete a designated computation. Flows offer three major benefits. First, a large sequential computational process can be decomposed into multiple small operators, where each operator is a general purpose and reusable component. Second, by overlapping the computation of data with the transport of that data between endpoints (i.e. memory-to-memory transfer) the potential to hide computational latency with communication latency increases. Finally, the dataflow representation of a computation can be easily mapped to the network architecture, making design automation tractable. In addition, the support of non-trivial flow patterns including converging and diverging flows significantly extends the applicability of dataflow processing in SoCs.

A flow identifies a path that a data stream takes from initiation to termination. A flow can start from on-chip or off-chip memory mapped SAPs; travelling through one or a sequence of SOPs according to its flow; and finally terminating at the destination memory. The 10-bit flow-id allows users to allocate 960 unique application flows on the network, with an additional
64 flow-ids reserved as system flows. The flow-id is run-time configurable and is associated with an initiator device-id, a terminator device-id, and one or several next-hop device-ids. The network interface of each intermediate node decodes the flow-id to obtain the next hop to which the current packet should be routed. Therefore, individual SAP and SOP nodes do not retain any information about any other nodes in the flow. In fact, the nodes are oblivious of almost all network control information including their own device-id and are only responsible for properly completing their assigned task or computation.

![Diagram](image)

Fig. 2.2: Accelerator System Topologies and Configurations

Fig. 2.2 depicts an example of a system, showing the associated dataflow graph topology, and physical topology. In this configuration, three flows have been configured. Flow 1 and flow 2 time-share SOP 1 and SOP 2, flow 0 and flow 2 time-share SOP 3, flow 1 exclusively accesses SOP 4, and all flows eventually terminate at SAP 5. Example contents of the flow table at
each network interface are also shown in Fig. 2.2(c), where valid bit and next-hop device-id are specified for every flow-id at each node.

The Switch Attached Processor, SAP, is either the source of data streams or the sink of data streams. In addition, SAPs represent computational nodes that need autonomy in initiating transactions. As illustrated in Fig. 2.3, the network interface for an SAP, NIF-SAP, hides the details of accessing high-level application functionality through low-level network protocols. The NIF-SAP has three interfaces to accomplish this: master interface, slave interface, and message interface. The master interface allows an SAP to initiate a transaction and provide or receive data directly through a simple FIFO-like handshaking mechanism, or indirectly from its own local memory space. The FIFO mechanism is more suitable for interfacing with devices such as cameras that output streams of pixel data in raster-scan fashion. The slave interface provides address/data style of handshaking, which is consistent with memory controllers including those for SRAM and DRAM memories. Finally, the NIF-SAP provides a light-weight message interface enabling message passing among SAPs. This is very useful for synchronizing the operation of different SAPs within the system.

<table>
<thead>
<tr>
<th>Master Request Type</th>
<th>Stream Source Device</th>
<th>Stream Source Interface</th>
<th>Stream Destination Device</th>
<th>Stream Destination Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Master Write</td>
<td>Initiator</td>
<td>Master</td>
<td>Target</td>
<td>Slave</td>
</tr>
<tr>
<td>2. Master Read</td>
<td>Target</td>
<td>Slave</td>
<td>Initiator</td>
<td>Master</td>
</tr>
<tr>
<td>3. Slave Write</td>
<td>Initiator</td>
<td>Slave</td>
<td>Target</td>
<td>Slave</td>
</tr>
<tr>
<td>4. Slave Read</td>
<td>Target</td>
<td>Slave</td>
<td>Initiator</td>
<td>Slave</td>
</tr>
<tr>
<td>5. Master Write-Lite</td>
<td>Initiator</td>
<td>Master</td>
<td>Target</td>
<td>Slave</td>
</tr>
<tr>
<td>6. Master Read-Lite</td>
<td>Target</td>
<td>Slave</td>
<td>Initiator</td>
<td>Master</td>
</tr>
</tbody>
</table>

An NIF-SAP allows an SAP to initiate 6 different types of transactions as listed in Table 2.1. As shown in the table, the Initiator denotes the SAP that initiates a transaction through its
Fig. 2.3: Diagram of NIF-SAP Architecture

*master interface*, while Target denotes the SAP that represents the endpoint of the transaction. The *flow-id* is specified by the Initiator during the request such that the NIF-SAP can establish the virtual connection between the startpoint, endpoint, and all intermediate nodes. The channel setup phase results in an efficient data transfer phase as most of the control information is maintained in the startpoints and endpoints and not contained in the header of each transaction packet.

Each NIF-SAP hosts a set of hardware components responsible for managing transactions. Each one of these components is referred to as a *handler*, while the set of handlers is referred to as the *handler pool*. Therefore, data channels are established between a producer handler at the initiator NIF-SAP and a consumer handler at the target NIF-SAP. On each transaction, the participating NIF-SAPs dynamically allocate producer and consumer handlers from their local handler pool. If the handler pool of either the Initiator or Target is exhausted, then the transaction request is terminated with an appropriate error code. Exactly how these errors
are handled is delegated to the SAP. For instance, the SAP may retry the failed transaction indefinitely, or it may reschedule other pending transactions that do not share dependency with the failing transaction. The dynamically allocated handler pool architecture has several benefits: (1) the SAP can be involved in multiple outstanding transactions, thus maximizing task-level concurrency; (2) the number of handlers can be configured at design time to trade-off resource utilization and application performance. For example, to maximally utilize the bandwidth offered by current DDR3 components while operating at conservative yet achievable clock frequencies, the memory controller can utilize multiple handlers to issue and respond to multiple concurrent read/write transactions. In addition to single initiator-target transactions, the NIF-SAP allows data streams to converge or diverge such that multiple initiators or targets can be involved in a single transaction. This provides for an efficient support for SIMD and MIMD operations. In these transactions, the NIF-SAP safely detects and mitigates deadlock conditions, where cancellation packets are issued when one or more targets deny a request while others have allocated handlers and accepted the request.

Master transactions, i.e. request types 1, 2, 5, and 6 in Table 2.1, can be utilized when the Initiator SAP wants to write (read) directly to (from) the master interface in a streaming fashion. In these scenarios, the SAP directly produces or consumes data via a simple FIFO-like handshaking protocol across the SAP/NIF-SAP interface. In particular, request types 5 and 6 are more suited for low-latency, small-size transactions. In these cases the Initiator-Target setup phase is established without regard to the Target(s) handler availability. This is useful for post boot configuration since most devices are free to accept packets. Configuration accesses can benefit from back-to-back transfers that avoid setup overhead for each small configuration packet. Slave transactions, i.e. request types 3 and 4, are similar to traditional DMA transfer;
where access to the initiator and target SAPs are at memory-mapped locations utilizing SRAM-like handshaking protocol.

A Stream Operator, SOP, is a processing node that operates on data in a streaming fashion. The NIF-SOP provides FIFO-like handshaking interfaces to the input and output channels of an SOP. Along with the simple handshaking signals, the NIF-SOP is equipped with side-band signals to denote the beginning and ending of frames. An SOP with more than a single input/output channel can connect to the network in one of two ways: (1) physically through multiple NIF-SOPs or, (2) virtually by time sharing a single NIF-SOP. In the former, scheduling conflicts are minimized since all processing nodes within the SOP can work concurrently in a data driven fashion.

The internal architecture of the NIF-SOP is shown in Fig. 2.4. It consists of three components: a depacketizer that deserializes data from the network for consumption by the SOP; a packetizer that serializes data originating from the SOP for presentation into the network; and a flow-id table which decodes an incoming flow-id into a local SOP opcode and next-hop device-id. Once depacketized, data is streamed to the custom SOP core through the egress interface. The egress interface exposes the data and associated opcode through a simple asynchronous handshaking protocol allowing back-pressure to be exerted to the input as necessary. As the SOP processes data, the output is forwarded to the ingress interface and re-packetized for injection into the network. During packetization, the incoming flow-id is used to update the packet header with the proper next-hop device-id. The NIF-SOP replaces the source device-id field in the packet header with the pre-configured device-id associated with the incoming flow-id as shown in Fig. 2.2c. Collectively the Vortex streaming framework allows maximum parallelism,
Fig. 2.4: Diagram of NIF-SOP attached with Custom SOP

which is obtained through coarse-grain pipelining across the network in addition to fine-grain pipelining within each SOP.

An example dataflow network on Vortex is illustrated in Fig. 2.5. The network interfaces, i.e. NIF-SAP and NIF-SOP, resolve the 10-bit application flow-id into a next-hop physical device-id within the network. The flow-id is encoded in the header of every packet that belongs to a frame. Since the flow-id to device-id translation is handled within the network interface, processing nodes are not aware of the context in which their outputs are used. The notion of information hiding is a hallmark of high-level programming paradigms and the key to flexible and scalable application development. Accordingly, the static or runtime configuration of flow-id tables within the network interfaces allows arbitrary dataflow graphs to be implemented without consideration by the constituent processing nodes. For SOPs, an additional 16-bit opcode is presented along with incoming data. This opcode is virtually coupled to the flow-id associated with the incoming data allowing the processing node to distinguish and offer flow specific variants of operations.
For example, Gabor feature extraction is performed selectively at scales of interest. The finest scale, scale 1, is associated with opcode 0x0010 while the coarsest scale, scale 5, is associated with opcode 0x0014. Two approaches can be taken for handling the situation of performing fine-grain Gabor feature extraction followed by coarse-grain Gabor feature extraction. The first method involves allocating two flow-ids say 0x2F0 and 0x2F4, respectively and configuring the NIF-SOP to translate the first and second flow-id to opcodes 0x0010 and 0x0014, respectively. During system operation, the data is streamed through the Gabor processor using flow-id=0x2F0 and subsequently with flow-id=0x2F4. This static method of configuration is suitable if all modes of operation that will be utilized during system operation are known a priori and there are sufficient flow-ids available. In the second approach, a single flow-id is allocated and initially associated with opcode 0x0010. During system operation, data is first streamed through the Gabor processor using flow-id=0x2F0. Next, the NIF-SOP flow-id table is reconfigured to
translate \texttt{flow-id}=0x2F0 to \texttt{opcode} 0x0014. Finally the data is streamed through the Gabor processor using \texttt{flow-id}=0x2F0. Since resolving \texttt{flow-id} to \texttt{opcode} occurs once at the beginning of a new frame, reprogramming of the \texttt{flow-id} table could be overlapped with data processing as the next translation will be synchronized at the start of the next frame. This runtime configuration method is useful when system behavior is dynamic and not known at system design time.

Memory is treated as a type of SAP, utilizing the NIF-SAP slave interface to expose a globally accessible memory mapped device. The NIF-MEM augments the functionality of the NIF-SAP by expanding the base \textit{message interface} with a message-triggered \textit{request manager}. The \textit{request manager} parses memory-request commands and subsequently initiates slave transactions between its local memory-connected slave interface and the slave interface of any other remote SAP(s) (including the NIF-MEM itself in the case of memory-to-memory copy). If a memory transfer requires barrier synchronization, the request manager sends completion notifications per transaction. Because the DMA functionality is incorporated within the NIF-MEM, network utilization can be reduced as compared to traditional shared DMA architectures that implement a read-store-write style of DMA, as illustrated in Fig. 2.6.
The request manager in Fig. 2.7 decodes memory-request messages, extracting information including source and target base addresses, transaction length, and flow-id. To start the stream, the NIF-MEM initiates a transaction request through its master interface to invoke a local memory read. This scenario is depicted in Fig. 2.8(a), and is distinguished as a write-transaction because the initiator desires to write a data stream into the network. Fig. 2.8(b) illustrates a read-transaction. The initiator wants to read a data stream from the network. In this case, Memory 4 is the initiator of the transaction even though it is the endpoint of the data stream. Therefore the memory request message is sent to Memory 4.

---

1Dashed lines indicate presence of intermediary SOPs (not shown in figure)
Notice that the initiator SAP (or memory) is only aware of the flow-id and device-id(s) of the terminating SAPs (or memory) with no knowledge of the intermediate SOPs within the flow. This abstraction hides intermediate data manipulation from the perspective of the startpoint and endpoint SAPs. This reduces the complexity of the SAP behavior and subsequently its programming. The system level configuration for each device determines the physical path that each stream follows to reach the endpoint. Modifying the path of the data stream does not affect the programs running on startpoint/endpoint SAPs.

It is often necessary in image processing applications to access a subset of a 2D array of data, which is referred to as Region of Interest, or ROI. In many occasions, this may require accessing non-contiguous chunks of the data. One approach to accessing an ROI is to issue
multiple transaction requests targeting these chunks of data. The disadvantage of this approach is that network arbitration and packet overheads may degrade performance, especially when these chunks are relatively small in size. Additionally, more complex logic is required to handle out-of-order arrival of these chunks. In contrast, the NIF-SAP supports read and write window transfers to handle ROI access. The NIF-SAP uses a window descriptor to specify the details of ROI access. A window descriptor includes row size, row stride, and row count information to describe an access pattern for fetching a rectangular sub-region, as illustrated in Fig. 2.9. The NIF-SAP includes a run-time configurable window descriptor table to associate ROI window descriptors to a given flow-id as shown in Fig. 2.10. This region can begin at any offset within a memory-mapped space. When a transaction referencing a window flow is issued, the initiator and target NIF-SAPs transparently fetch and store data according to the access pattern while fully utilizing the payload capacity of each packet. Since window transfers are handled exclusively within the NIF-SAP, intermediary nodes maintain a simplified 1D streaming view of data.
Table 2.2: A summary of resource utilization of the router with various number of bidirectional ports on the Xilinx XCV6SX475T FPGA device

<table>
<thead>
<tr>
<th># bidirectional ports</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice Register</td>
<td>3192</td>
<td>5880</td>
<td>8335</td>
<td>10788</td>
<td>13288</td>
<td>15796</td>
<td>18317</td>
<td>37424</td>
</tr>
<tr>
<td>Slice LUT</td>
<td>1590</td>
<td>3721</td>
<td>5626</td>
<td>7539</td>
<td>10794</td>
<td>13025</td>
<td>15280</td>
<td>15519</td>
</tr>
<tr>
<td>BRAM</td>
<td>9</td>
<td>17</td>
<td>26</td>
<td>34</td>
<td>43</td>
<td>51</td>
<td>60</td>
<td>68</td>
</tr>
</tbody>
</table>

Table 2.3: A summary of resource utilization of NIF-SAP with various number of handler-pair on the Xilinx XCV6SX475T FPGA device

<table>
<thead>
<tr>
<th># handler-pair</th>
<th>1(wo/msg)</th>
<th>1(w/msg)</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice Register</td>
<td>2607</td>
<td>2980</td>
<td>4281</td>
<td>5599</td>
<td>6685</td>
</tr>
<tr>
<td>Slice LUT</td>
<td>4222</td>
<td>4932</td>
<td>7600</td>
<td>10629</td>
<td>12404</td>
</tr>
<tr>
<td>BRAM</td>
<td>8</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>DSP</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 2.4: A summary of resource utilization of NIF-SOP with various number of output channels on the Xilinx XCV6SX475T FPGA device

<table>
<thead>
<tr>
<th># output channels</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice Register</td>
<td>422</td>
<td>478</td>
<td>535</td>
<td>592</td>
<td>652</td>
<td>710</td>
<td>765</td>
<td>821</td>
</tr>
<tr>
<td>Slice LUT</td>
<td>485</td>
<td>668</td>
<td>823</td>
<td>909</td>
<td>1109</td>
<td>1270</td>
<td>1470</td>
<td>1516</td>
</tr>
<tr>
<td>BRAM</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>18</td>
</tr>
</tbody>
</table>

Vortex supports multiple concurrent applications by sharing the NoC fabric. As long as the accumulated bandwidth does not reach the peak bandwidth supported by the infrastructure, multiple flows can share SOPs in a time multiplexed fashion. In order to support such functionality, the NIF-SOP contains multiple output queues in the packetizer, as shown in Fig. 2.4. The number of queues is parameterizable at design time to trade-off resource consumption and maximum outstanding flows that share the SOP. Table 2.2 presents the resource utilization of the Vortex router, when mapped to a Xilinx FPGA device, using various number of bidirectional ports. Similarly, Table 2.3 and Table 2.4 show the resource utilizations for different configurations of the NIF-SAP and NIF-SOP, respectively.
2.3.2.3 Meeting Neuromorphic Vision Requirements

1. Flexibility

The NIF-SAP and NIF-SOP support the connection of computational nodes found in a dataflow representation. Multiple input and output channels are supported through multiple network interfaces connected to a single processing node as shown in Fig. 2.11. This allows for multiple streams to progress concurrently. Vortex also supports a hierarchy of dataflow graphs, which means that an SOP can be composed of smaller recursively-defined operators. To aid in the composition of SOPs, Vortex provides a composition library of modules, namely, input and output port adaptors, inter-operator link modules, address decoders, and hierarchical control units.

2. Scalability

Vortex inherits its scalability attributes from the NoC paradigm. Routers can be cascaded either on-chip or across-chip boundaries via platform specific inter-chip links. By utilizing a 16 bit *device-id*, the system can distinguish 64k individual processing nodes. Moreover,
by adopting a table-based routing scheme at each router, no particular network topology or hierarchy is implied or required.

3. Programmability

Runtime configuration of flow-id and opcode at each network interface maximizes the programmability of the system in terms of dataflow and behavior of processing nodes even on post-silicon designs. Once primitive operations are attached to the network as a form of SOP, various algorithms are mapped by appropriately programming the distributed flow-id tables.

4. High Bandwidth

The network interfaces utilize a 128-bit flit between the interface and the attached SOP or SAP. Experiments conducted on FPGA prototyping platforms show that the Vortex system easily achieves 200 MHz operating frequency when targeting the lowest speed grade device in the Xilinx Virtex-6 SX FPGA family[58]. Note that the maximum achievable clock-frequency may be significantly higher when targeting an ASIC. Moreover, by supporting independent clocking domains for each SOP and SAP, the maximum operating frequency of a particular node does not affect those of other nodes in the system. In fact, the maximum bandwidth measured on an FPGA emulation platform is 3.2 GB/s. Internally, routers utilize a 256-bit flit size and operate at 400 MHz when targeting the lowest speed grade device in the Xilinx Virtex-6 SX FPGA family. Such a high network capacity ensures that Vortex satisfies the interconnection infrastructure requirements.
2.3.3 Requirements for Customized Accelerators

There are several requirements that an algorithm must meet before it becomes mapped to hardware. The following subsections highlight these requirements.

2.3.3.1 Exploiting Parallelism

The human brain is a massively parallel processor consisting of 100 billion individual processing elements, or neurons. The enormity in number of neurons translates to an unparalleled processing rate of $10^{16}$ FLOPS. While this work does not attempt to provide a neuron-level modeling of algorithms, the brain remains a metric of fidelity when defining architectures for accelerating neuromorphic algorithms.

Delivering high performance hardware architectures requires a deep understanding of the given algorithms. For example, some algorithms exhibit potential for data-level parallelism (DLP) (e.g. convolution-like operations). Others exhibit iterative processing behavior on independent data sets, which can be accelerated using task-level parallelism (TLP). A number of hardware accelerators that make use of DLP and TLP are illustrated in later sections. Such architectures are possible due to the abundance of parallel resources either on reconfigurable computing platforms (e.g. FPGAs) or dedicated hardware (e.g. ASICs).

2.3.3.2 Power Efficiency

Interestingly, the human brain delivers its massive computing capacity while maintaining a relatively low power budget of roughly 20 Watts. The reasons for such ultra-low power consumption—compiled by [37]—include sub-threshold spiking operations, sparse-energy efficient codes for signaling, and proper balance of analog computation and digital signaling.
In contrast, the computational power of the contemporary single core CPU has been limited by the power wall dominated by high dynamic power consumption. While the typical operating frequency remains around 3GHz, the multi-core CPU paradigm is the current resolution to the power wall dilemma. However, even multi-core CPUs are unable to meet the performance requirements for a number of application classes. As such, domain-specific accelerators are gaining popularity as solutions that deliver high performance within small power envelopes. Domain-specific accelerators go beyond the task-level parallelism exploited by general purpose CPUs, by taking advantage of gate-level customization and parallelism. The result is highly optimized processing pipelines that can be clocked at lower frequencies resulting in high energy efficiency.

2.3.3.3 Highly Parameterizable Design

As knowledge of biological processes continues to grow, neuromorphic models of the primate visual cortex are continuously being refined and redefined. Consequently, neuromorphic vision algorithms are in a constant state of developmental flux. Accordingly, accelerators must support a wide range of configurations allowing them to be re-tasked to a large number of algorithm contexts. Configurability should be supported at design time, runtime, or both. Design time configurations are specified at the time of system synthesis and determine structural aspects of accelerators. These structural aspects may include accelerator composition, statically defined operand data widths, pipeline depth, and worst-case architectural features. Design time parameters allow synthesis tools to optimize around statically specified non-changing design parameters, leading to optimal resource utilization. To support rapid algorithm exploration, however, accelerators must support a modest degree of runtime configurability while maintaining a
reasonable resource profile. Runtime configuration allows aspects of accelerator operation to be modified without time consuming re-synthesis. This comes at the expense of additional logic resources to support such flexibility. Runtime configurable parameters are typically reserved for non-structural aspects such as convolution kernel coefficients, image dimensions, and accelerator control parameters. However, accelerators such as variable size convolution engines that expose runtime configurability in structural aspects of their architecture are stronger candidates for hardware mapping.

2.3.3.4 Composability and Programmability

To maximize component reuse and hierarchical system composition, the most frequently referenced algorithm components are identified and used to populate a library of hardware building blocks. The granularity of these building blocks is multi-tiered: ranging from fine-grain primitives such as adders and subtractors; mid-grain primitives such as convolvers and statistical operators; and macro operators such as retina preprocessors, saliency detectors, and feature extractors. Profiling neuromorphic vision models reveals that many of the algorithms can be mapped to a streaming mode of processing. Stream processing is benefited by a minimal requirement for storage and control as these aspects are implicit in the dataflow nature of the processing. For these reasons, all operations that can be mapped to a streaming modality are considered for hardware implementation. To compose these streaming algorithms, accelerators are constructed from a set of streaming operators mapped to a dataflow process network[32]. The nodes of this process network are realized by specific streaming operators while the dataflow paths are realized by the Vortex interconnection system of on-chip routers and communication channels.
Still, there is a significant number of algorithms that exhibit either non-streaming characteristics or a hybrid of streaming and non-streaming characteristics. Iterative control constructs, complex state transitions, and arbitrary memory utilization and accesses present in these algorithms are not effectively mapped to a purely streamed processing architecture. Consequently, the dataflow process network is augmented to include processing elements that maintain a Von Neumann model of sequentially executed instruction streams. The specific definitions of these instructions are reserved to the implementation of each processing element. In this way each class of processing element has a unique ISA for which the associated processing architecture is optimized to execute in an accelerated fashion. Instruction streams are executed concurrently by any number of processing elements, each scheduling arbitrary process flows across streaming accelerators as necessary. Therefore system composition is defined by three aspects: the static allocation of SIMD stream accelerators; the static allocation of custom ISA processors; and the orchestration of highly temporal control and virtual interconnection of both. In this way, complex notions of iteration and functional composition can be described on a static network of accelerator resources.

2.3.4 Architectural Details of Customized Accelerators

2.3.4.1 SAP Processing Element (PE)

The SAP is suitable for carrying out computations that are structurally iterative and operate on non-contiguous blocks of data. Earlier sections discussed the features exposed by the NIF-SAP, which includes data movement and messaging capabilities and how the NIF-SAP conceals the underlying network details from the SAP developer. However, SAP developers may
still find it laborious to implement the necessary logic for handshaking with the NIF-SAP. Similarly, developers may observe undesired redundancy; where the same hardware logic used to interface with the NIF-SAP is not being reused from one SAP implementation to another. Moreover, controlling SAP accelerators in their current status require the additional implementation of finite state machine, FSM, to orchestrate the operations of the accelerator — making the SAP less flexible and harder to reconfigure.

To address the issues presented above, SAP-PE is designed by [1] to add an additional layer of abstraction to the SAP. This layer of abstraction serves the following purposes:

- Standardizing how the SAP is used and accessed. As a result, developers focus more on the custom accelerator development and worry less about the complexities of interfacing with the NIF-SAP. Additionally, standardizing the SAP allows developers to reuse their code, hence boosting their productivity.

- Abstracting the hardware details from the user and exposing a set of pre-defined software primitives (APIs) that can be used to control operations. This API is coded in C/C++, allowing non-HDL developers to program these accelerators. A standard C/C++ tool chain is used to compile the written code into a bytecode that is stored in the SAP for subsequent execution. Using this API, the user can perform DMA transactions, synchronize operations across SAPs, issue specific instructions to the SAP, or configure the SAP’s register file.

Henceforth, the acronym SAP-PE is used to refer to the SAP accelerator combined with the additional layer of abstraction described above.
Fig. 2.12 illustrates the architecture of the SAP-PE. The architecture is partitioned into a control path, providing instruction-based control over the movement of data and the custom accelerator, and a data path, consisting of the implementation of the custom accelerator logic and functions. In the control path, the main driver of operation is the *Light-Weight Processor*, or LWP. The LWP provides several standard mechanisms for control such as branching, looping, and basic ALU functions for simple address manipulation (e.g. addition, subtraction, shifting, etc...). The LWP is deliberately void of complex arithmetic logic as the majority of the computation is intended for the custom accelerator hardware, rather than an instruction based processor.

The LWP fetches instructions from a scratch-pad memory that is loaded with the user’s instruction sequence. Consequently, the LWP decodes the fetched instructions and issues the corresponding command to one of the available command handlers. Each one of these handlers carries out a specific task as follows:

- **DMA Read/Write, DMA Rd/Wr, handler**: Issues a DMA transaction request to the NIF-SAP Master interface
- **Master Read/Write, MS Rd/Wr, handler**: Issues a Single transaction request to the NIF-SAP Master interface
- **Message, Msg, handler**: Issues a message read/write request to the NIF-SAP message interface
- **Accelerator-Specific Instruction, ASI, handler**: Communicates one of up to 256 accelerator-specific commands. The control path is oblivious of the interpretation of these commands. Therefore, the exact interpretation of these commands must be handled by the custom
Fig. 2.12: SAP-PE µArchitecture [1] The architecture is split into two paths: control and data. The control path abstracts the underlying hardware complexities and exposes a set of APIs for the user to control the accelerator’s operations. The data path is where the custom accelerator resides.

- Accelerator-Specific Register, ASR, handler: Provides access to the register file implemented within the custom accelerator. These registers can be used to configure the accelerator. For instance, the user can write configurations to a register in order to change the kernel size of the convolution engine implemented within the custom accelerator.

On the other hand, the SAP-PE data path, illustrated in Fig. 2.12, is made up of the custom accelerator hardware and is directly controlled through specific instructions issued by the control path. Data is transferred to the custom logic, directly through the NIF-SAP slave interface, using the DMA transfer instructions described above.
2.3.4.2 Composability (ChipMONK for SOPs)

Fig. 2.13 highlights the hierarchical streaming accelerator structure. The root of the hierarchy is the streaming accelerator. The accelerator is composed of one or more primitive streaming building blocks. These building blocks include convolution engines, arithmetic operators, statistical operators, transform operators, and non-linear function approximation modules. Collectively, the primitive building blocks are referred to as the Base Library of Operators. The interconnectivity, programmability, and control of these primitives are supported by a set of utility components that include input and output port adaptors, inter-operator link modules, address decoders, and hierarchical control units. An accelerator may support many modes of operation that may differ in runtime parameters or to some degree sequence of computation. For example, the Retina/LGN preprocessing accelerator can operate in Retina, LGN, and Retina-LGN modes. Each of these modes is different in the way data flows through the pipeline and can be chosen dynamically. To support rapid algorithm exploration, an automation framework called
ChipMONK is proposed to combine computational primitives and utility primitives to construct SOPs. ChipMONK performs the tasks of interconnecting primitives, resolving bit-width configurations, calculating and allocating appropriate buffering, address space partitioning, and control state machine synthesis.

2.3.5 The Realization of the Neuromorphic SoC

In order to validate accelerators mapped to the SoC framework, a multi-FPGA System, MFS, is used as a prototyping platform. However, designing a hardware architecture that targets an MFS requires considerable skill and expertise. Moreover, HDL development becomes laborious and error prone as the target systems grows in size and complexity. Hence emerges the need for design automation tools that assist users in building their design easily and efficiently — saving them both time and effort.

A software tool for automating the process of mapping hardware designs to an MFS is also presented in [1], namely Cerebrum. Cerebrum abstracts the details of RTL coding as well as communication and memory hierarchy partitioning. Moreover, it automates execution of the synthesis and implementation phases of the design process. Further details about the tool can be found in [1], and the system composition has been done through the tool for this work.

2.4 Case Study

Fig. 2.14 illustrates a neuromorphic system capable of performing object detection and recognition. The system receives imagery from an attached streaming device (e.g. a camera).
Fig. 2.14: A neuromorphic system for visual processing. The system pre-processes the input images for contrast enhancement. The system operates on the enhanced input to detect salient objects and classify them.

The image enhancement block performs contrast enhancement and eliminates undesired common illumination. The enhanced image is stored in memory and forwarded to the saliency detector, which identifies salient regions in the image. The saliency detector communicates the coordinates of ROI to the feature extraction block. The feature extractor uses the coordinates to request, from memory, the salient regions of the enhanced image. The feature extraction block produces a feature vector that is representative of the processed ROI. Finally, a trained classifier classifies the ROI using the feature vector produced by the feature extractor. This neuromorphic system is realized on a Multi-FPGA system as depicted in Fig. 2.15. The following subsections discuss each of these processes in detail and highlight the accelerator architectures that allow execution in real-time.

2.4.1 Retina Preprocessing

In the human visual system, the retina performs preconditioning of imagery for high-level vision tasks such as attention and object recognition[24][51][38]. In the visual pathway,
the retina consists of photoreceptive cells—rods and cones—that perform the transduction of light striking the eye to neural impulses that are forwarded through the optic nerve for subsequent processing in the visual cortex. These neural responses are a function of the competing interactions between stimuli generated by spatially co-located rods and cones. A key artifact of these inhibitory interactions is the enhancement of contrasting structures within the visual field. Ultimately these peak responses become the primary features for perception.

A streaming retina processor was presented in [10] utilizing the SOP composition methodology, discussed in Section 2.3.4.2, and is illustrated in Fig. 2.16. The input to the retina preprocessor is the YIQ color space representation of the original RGB image. Each of Y, I, Q, Negated I, and Negated Q are extracted as independent image channels. The first stage of retina
processing performs contrast enhancement, normalization, common illuminant removal, and dynamic range compression on each channel independently. The retina preprocessor uses a model of center-surround competition within pixel neighborhoods to enhance contrast locally while removing common illuminants[20]. The subsequent image is normalized and its dynamic range compressed using a sigmoid operator that is adaptive to the global image statistics.

The second role of retina processing is to fuse the responses of independent channels in a way that either extracts complementary information (decorrelation) or enhances channel similarities[56]. This inter-channel fusion is also performed using a model of center-surround competition in a fashion similar to the color opponent process between rods and cones in the human eye. The retina processor produces four channels that represent the cross channel fusion of the contrast enhanced and normalized Y, I, Q, Negated I, and Negated Q channels. These channels are used in the subsequent saliency and feature extraction processing stages.
The common operation across the two functions of the retina is an operator referred to as double opponency. As the name suggests, double opponency performs the center-surround opponent computation between two input channels. Internally, the double opponency operator performs Difference of Gaussian, DoG, between its center and surround inputs followed by adaptive dynamic range compression using a non-linear sigmoid-like transform. In the case of channel enhancement, a single channel is replicated and presented to the center and surround inputs identically. This configuration is appropriate because the intent is to enhance pixel values that are local maxima within the neighborhood of the given pixel (intra-channel enhancement). In the case of channel fusion the intent is to either extract or enhance the responses of different channels. Therefore, the center and surround input to the double opponency are the two channels to be considered. Because of its high frequency of use and its streaming nature the double opponency process is realized as a hardware accelerated component in the SOP Base Library of Operators.

There are two stages within the double opponency accelerator. In the first stage, the weighted surround channel is computed by convolving the surround input with a Gaussian filter having a runtime configurable sigma coefficient. For each pixel, the difference between the center channel and the Gaussian weighted surround channel is computed and normalized by the total weighted response in the pixel neighborhood. The resulting image is a local contrast enhanced version of the original image normalized to the neighborhood response.

In the second stage, the contrast enhanced image is remapped to the full output range (0 to 65,535). The non-linear remapping is performed by a sigmoid function approximation modulated by the global mean of the enhanced image. The sigmoid operator is used frequently and so it too is realized as a hardware accelerated component in the SOP Base Library of Operators.
Since retina processing is the first step in the visual pipeline and because it is implemented as a streaming operator, it is configured to receive images directly from the PCI Express interface: alleviating the need to buffer incoming images in onboard memory. The host system captures images from a high-resolution GigE camera and buffers it in the host’s physical memory. To process an image frame, the host sends a read request message to the host-to-board DMA engine on FPGA ‘C’ (See Fig. 2.15). The request specifies the source physical memory address and flow-id: the target addresses within DRAM D and DRAM F are pre-configured and associated with the flow-id. The DMA engine subsequently begins fetching data from the image buffer in host physical memory and injects it into the network targeting the retina processor. Consequently, no additional latency is incurred for onboard buffering as the retina processing is overlapped with the transfer of data from system memory. The full and scaled images are injected into the network utilizing dedicated NIF-SOP interfaces taking independent paths to DRAMs D and F, respectively.

The resource utilization of the neuromorphic retina processor on the prototype platform is presented in Table 2.5.

While CPUs and GPUs operate on 8-, 16-, 32-, and 64-bit variables, FPGA and ASIC fabrics support arbitrary bit-widths for each variable in the design. By adjusting the bit widths according to the precision requirements, significant reduction in the silicon area cost of arithmetic units and bandwidth requirement between different hardware modules can be achieved.

Table 2.5: A summary of resource utilization for neuromorphic retina processor on XCV6SX475T

<table>
<thead>
<tr>
<th></th>
<th>Slice Register</th>
<th>Slice LUT</th>
<th>BRAM</th>
<th>DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retina Resource</td>
<td>158,568</td>
<td>133,342</td>
<td>434</td>
<td>390</td>
</tr>
</tbody>
</table>
The image input to the retina utilizes a 0:1:15 fixed-point format. Intermediate representations within the pipeline are set appropriately by ChipMONK by performing static fixed-point bit allocation. Ultimately, the difference between the fixed-point implementation and the double-precision equivalent implementation was in the range of $10^{-5}$ to $10^{-6}$.

### 2.4.2 Visual Saliency (AIM)

The attention mechanism of the human vision system allows the brain’s perceptual and cognitive resources to be focused on the most important regions in the visual field. Otherwise, the cognitive system would be overwhelmed by the tremendous amount of visual information arriving from the optic nerve. In synthetic vision systems, this attention mechanism is termed Saliency detection. The process guides the application of object recognition to relatively small subsets of the image field, maximizing system efficiency in terms of processing latency and resource utilization.

A variant of the Attention by Information Maximization, AIM, algorithm proposed by Neil Bruce and John Tsotsos[9] is implemented in the vision processing system. The premise of the algorithm is that the most salient areas are those that have the most information content. The algorithm operates in two phases. In the first phase, AIM transforms the input into an image in which each pixel is an n-dimensional vector. Each element $i$ in the vector is a coefficient representing the contribution of the $i^{th}$ basis vector in an orthonormal basis. The basis or feature space is either learned by an Independent Component Analysis, ICA, process or explicitly defined by a filter such as the Gabor kernel. Two of the four scaled channels (e.g. $0',1',2'$, or $3'$) originating from the retina preprocessor are dynamically selected to generate 48 independent feature maps,
or response maps, by employing two-dimensional complex convolutions with four scales and six orientations of the Gabor kernel.

In the second phase, the probability density is computed for each pixel in each feature map. The 48 Gabor feature maps are used to construct 48 histograms to estimate the probability density function. Finally, once the histograms have been constructed, the likelihood \( L_{i,j,k} \) of the pixel on the \( i^{th} \) row and \( j^{th} \) column appearing in the \( k^{th} \) response map is determined by indexing the \( k^{th} \) histogram with the value of the pixel. The information content or Self-Information of each pixel is then computed by the summation of \( \log(L_{i,j,k}) \) across all \( k \) response maps. The result is the saliency map.

Fig. 2.17 shows the dataflow of the visual saliency model. Note that indexing a histogram is only possible after the histogram has been constructed from the entire response map. Therefore the dataflow graph is split representing the two phases of the algorithm. Implementing each in a fully parallelized fashion achieves latency proportional to \( 2|I| \) where \( |I| \) is the number of pixels in the image. If the saliency processor operates at 100MHz on \( 512 \times 384 \) imagery, the total latency is roughly 4 ms. The pipeline, however, consumes approximately 2,800 multiplier resources which may not be feasible for smaller platforms. In terms of satisfying real-time

### Table 2.6: Comparisons among architectural approaches to implement AIM

<table>
<thead>
<tr>
<th></th>
<th>Resource consumption</th>
<th>Latency(ms)</th>
<th>Frame rate (fps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully parallel</td>
<td>2800</td>
<td>3.9</td>
<td>256</td>
</tr>
<tr>
<td>A channel per iteration</td>
<td>1400</td>
<td>5.9</td>
<td>169.5</td>
</tr>
<tr>
<td>A scale per iteration</td>
<td>972</td>
<td>17.7</td>
<td>56.5</td>
</tr>
<tr>
<td>An orientation per iteration</td>
<td>424</td>
<td>25.6</td>
<td>39.1</td>
</tr>
<tr>
<td>Fully iterative</td>
<td>162</td>
<td>49.2</td>
<td>20.3</td>
</tr>
</tbody>
</table>

Resource consumption is represented by number of multiplications in convolutions. Latency and frame rate is based on following conditions:
- Input Frame resolution = \( 2048 \times 1536 \).
- Sub-sampled Retina/LGN output resolution = \( 512 \times 384 \).
- Operating clock frequency = 100 MHz.
Fig. 2.17: Dataflow graph representation of AIM

constraints of 30 fps, the fully parallel pipeline far exceeds the requirements. There certainly exists a more balanced trade-off between resource consumption and latency. Table 2.6 shows several architectural approaches with estimated latency and resource consumption in number of multiplications.

Alternatively, a fully iterative architecture processes one response map at a time for each combination of channel, scale, and orientation, requiring 48 iterations to obtain the saliency map. The iterative approach only requires 162 multiplier resources, however, it fails to meet real-time constraints: achieving only 10 fps for a $512 \times 384$ image when operating at 100MHz. Within the extremes of these two approaches there is a large exploration space for trading-off performance and resource utilization.

The prototyped implementation of AIM is configured to process four scales concurrently across two channels and six orientations iteratively. Fig. 2.18 highlights the methods of concurrency and iteration in the adopted implementation.
The architecture is partitioned into two pipelines that operate concurrently: the build pipeline for computing the coefficient density and the index pipeline for computing the self-information map. Each response map generated by each of the four Gabor convolvers must be propagated to both the associated histogram, for histogram construction, and to external memory, for recall during histogram indexing. Under these conditions a total of four memory transactions are active simultaneously: fetching the current channel, storing the current output of the four Gabor convolvers, fetching the previous output of the four Gabor convolvers, and storing the current partial saliency map. Assuming $512 \times 384$ imagery with 64-bit input representation and 16-bit response map representation, the saliency processor demands roughly 1.7 GB/s of memory bandwidth to maintain 30 fps throughput. This constitutes 20% of the peak bandwidth.
Fig. 2.19: Block Diagram of AIM architecture

of DDR3-1066. Given that the multiplier resources are relatively low, the memory and network bandwidths can be reduced dramatically at the expense of doubling the number of multiplier resources as shown in Fig. 2.19.

DRAM F, in Fig. 2.15, contains the sub-sampled version of the preprocessed image originating from the retina preprocessor. Each of the four channels has a 16-bit pixel representation. Four pixels are packed into a single 64-bit data flit. For each of the iterations, the AIM processor selects the appropriate channel from the incoming data stream and forwards it to the build and index pipelines. By duplicating the four Gabor convolvers in the index pipeline, the response map is recomputed on-demand obviating the need for storing intermediate maps in external memory. In other words, instead of burdening the network and memory system by storing and fetching the
intermediate response maps, the architecture calculates the response a second time for indexing the histogram. The calculation of the previous response map for indexing is overlapped with the calculation of the current response map for building. The Dual-Histogram stream operator allows concurrent building and indexing of two internal, independent and alternating, histogram tables. The schedule of the pipeline is shown in Table 2.7.

For each of the iterations, the AIM processor produces a likelihood map by applying \( \log(x) \) operator to the output of the indexed histograms: producing likelihood maps for four scales simultaneously at a particular channel and orientation. To produce the partial saliency map, the four maps are accumulated pixel-wise with the partial saliency map computed in previous iterations. Rather than store the partial saliency map in external memory, the architecture utilizes a local SRAM to maintain the pixel-wise accumulation. Network and memory bandwidth are minimized as the processor does not generate outgoing network traffic until all partial saliency maps are computed. Consequently, only 620 MB/s total memory bandwidth is required to maintain 30 fps: a moderate 7% of the peak bandwidth of DDR3-1066.

The host system orchestrates the iterative saliency process by issuing 12 memory-to-memory transaction requests. The source memory is the location of the sub-sampled retina output while the target memory is the location at which the saliency map will be stored. Note that the target memory is only updated after the 12\textsuperscript{th} iteration when the final saliency map is being computed. Each transaction references a unique \textit{opcode} that allows the pipeline to distinguish

<table>
<thead>
<tr>
<th>Table 2.7: Schedule of timeline to operate AIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Channel #, Orientation #) for each iteration</td>
</tr>
<tr>
<td>Build</td>
</tr>
<tr>
<td>Index</td>
</tr>
</tbody>
</table>
which channel and orientation combination to be processed for the given transaction. Once
the saliency map has been computed and stored into the host memory, a host software process
performs ROI extraction using a connected components algorithm. The coordinates of the ROI
within the full scale preprocessed image are forwarded to the feature extractor. Table 2.8 shows
the resource utilization for the AIM accelerator when mapped a Virtex 6 FPGA device.

2.4.3 Feature Extraction (HMAX)

HMAX (“Hierarchical Model and X”) [48][52] is a model of the ventral visual pathway
from the visual cortex to the inferotemporal cortex, IT. This model attempts to provide space and
scale invariant object recognition by building complex features from a set of simple features in
a hierarchical fashion.

Fig. 2.20 shows a computational template of HMAX. The model primarily consists of
two distinct types of computations, convolution and pooling (non-linear subsampling), corre-
sponding to the Simple, S, and Complex, C, cell types found in the visual cortex. The first S-
layer, S₁, is comprised of fixed, simple-tuning cells, represented as oriented Gabors. Following
the S₁ layer, the remaining layers alternate between max-pooling layers and template-matching
layers tuned by a dictionary encompassing patterns representative of the categorization task.

The exact implementation of HMAX is determined from what is considered to be the
most biologically plausible. This work uses a specific implementation by [1] for the object

<table>
<thead>
<tr>
<th>AIM Resource</th>
<th>Slice Register</th>
<th>Slice LUT</th>
<th>BRAM</th>
<th>DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>92395</td>
<td>66404</td>
<td>368</td>
<td>182</td>
</tr>
</tbody>
</table>
Fig. 2.20: A computational template of HMAX model showing the HMAX stages recognition algorithm developed by Mutch and Lowe[36], as it represents the current understanding of the ventral stream and produces good results when used for classification. This model is represented by a total of five layers, an image layer and four layers corresponding to the alternating S and C units. Further details about the architecture of HMAX accelerator, used in this work, can be found in [1].

Fig. 2.21 illustrates the interactions that occur between the Host processor and HMAX accelerators. Although the figure shows a virtual topology, all components are mapped to the physical topology shown in Fig. 2.15.

As discussed earlier in Section 2.4.1, the retina preprocessor produces a full resolution contrast-enhanced image to be used by the feature extractor. This enhanced image is buffered in the memory attached to FPGA ‘A’, See Fig. 2.15. The host processor schedules an ROI transfer of the enhanced image from the memory to the $C_1$ accelerator, where the latter is mapped to FPGA ‘B’. This ROI flows through $S_1$ stream processor, mapped to FPGA ‘A’. This process is repeated for every two adjacent scales across all orientations. The output of the $C_1$ accelerator is broadcasted to all $S_2/C_2$ accelerators, mapped to FPGAs ‘A’, ‘B’, ‘D’ and ‘E’ — exploiting task-level parallelism across all the prototypes in the $S_2$ dictionary. Then, the host processor will schedule a read transfer of results from all $S_2/C_2$ accelerators, sequentially, one after the other.
Fig. 2.21: Host-HMAX accelerator interaction. The Host schedules the SAP Memory to write contrast-enhanced images to C\textsubscript{1} SAP-PE flowing through the S\textsubscript{1} SOP. The C\textsubscript{1} SAP-PE broadcasts its output to multiple instances of the S\textsubscript{2}/C\textsubscript{2} accelerator (figure shows 4 instances). Finally, the Host schedules a read request from each S\textsubscript{2}/C\textsubscript{2} accelerator sequentially and aggregates the incoming results from these accelerators.

Since each S\textsubscript{2}/C\textsubscript{2} accelerator is operating on a different set of prototypes, the host processor will have to merge these outputs as it receives them. Finally, the host processor tests the aggregated feature vector using a linear classifier, where the classification decision is finally made.

Table 2.9 summarizes the HMAX resources utilization when mapped to FPGAs ‘A’, ‘B’, ‘D’ and ‘E’. Note that the numbers in the table includes the four instances of the S\textsubscript{2}/C\textsubscript{2} accelerator.

Table 2.9: A summary of resource utilization for the HMAX accelerator on XCV6SX475T

<table>
<thead>
<tr>
<th>HMAX Resource</th>
<th>Slice Register</th>
<th>Slice LUT</th>
<th>BRAM</th>
<th>DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>316,794</td>
<td>133,611</td>
<td>623</td>
<td>2,206</td>
</tr>
</tbody>
</table>
2.5 Experimental Setup and Results

This section discusses both the accuracy and performance of the implemented accelerators running on the multi-FPGA platform. Furthermore, a cross-platform performance comparison is presented. Fig. 2.22 shows the experimental setup used in this work.

2.5.1 Classification Accuracy

The output of the HMAX model (i.e. features vector) is used as an input to a classifier for recognition purposes. To test the classification accuracy of the neuromorphic accelerators, a
total of 10 categories from the Caltech101 [21] data set are used. Table 2.10 lists the categories and number of images used for training and testing the classifier.

Table 2.10: List of Caltech101 categories used in the experiments. The third and fourth columns show the number of images used for training and testing the classifier, respectively. Note that there is no overlap between training and test images.

<table>
<thead>
<tr>
<th>Category ID</th>
<th>Category Name</th>
<th># training images</th>
<th># test images</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>airplanes</td>
<td>400</td>
<td>400</td>
<td>800</td>
</tr>
<tr>
<td>2</td>
<td>car_side</td>
<td>62</td>
<td>61</td>
<td>123</td>
</tr>
<tr>
<td>3</td>
<td>chandelier</td>
<td>54</td>
<td>53</td>
<td>107</td>
</tr>
<tr>
<td>4</td>
<td>grand_piano</td>
<td>50</td>
<td>49</td>
<td>99</td>
</tr>
<tr>
<td>5</td>
<td>helicopter</td>
<td>44</td>
<td>44</td>
<td>88</td>
</tr>
<tr>
<td>6</td>
<td>ketch</td>
<td>57</td>
<td>57</td>
<td>114</td>
</tr>
<tr>
<td>7</td>
<td>laptop</td>
<td>41</td>
<td>40</td>
<td>81</td>
</tr>
<tr>
<td>8</td>
<td>motorbikes</td>
<td>399</td>
<td>399</td>
<td>798</td>
</tr>
<tr>
<td>9</td>
<td>revolver</td>
<td>41</td>
<td>41</td>
<td>82</td>
</tr>
<tr>
<td>10</td>
<td>watch</td>
<td>120</td>
<td>119</td>
<td>239</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>1268</td>
<td>1263</td>
<td>2531</td>
</tr>
</tbody>
</table>

One of the features offered by the acceleration framework, and discussed earlier in this work, is the ability to modify the parameters of the accelerators without the need to re-synthesize the system. This allows for exploring the accelerator’s configurations, while not affecting the productivity of the user. Using this feature, we study how the number of scales and orientations of the HMAX model impact the classification accuracy. Fig. 2.23 shows the classification accuracy for a number of HMAX configurations. The figure illustrates the impact of changing the number of input scales — for the same number of orientations — on the overall accuracy. For the 4 orientation set (4-Orient), increasing the number of input scales results in improved accuracy. On the other hand, the 12 orientation set (12-Orient) exhibits a varying, but consistent, improvement in accuracy when number of input scales is increased within the 8- to 11-scale configuration sets. However, the 12-scale configuration results in 0.48% less accurate classification when compared to the 11-scale configuration. This insignificant (< 1%) degradation in
Fig. 2.23: Object classification accuracy for a number of accelerated HMAX configurations.

recognition is attributed to the frequent truncation of the fixed-point representation during the multiply-accumulate operation within convolution.

Compared to a CPU implementation[35] of the HMAX algorithm, it is found that the classification accuracy of the FPGA implementation is at most 2% less accurate than the CPU implementation. Again, the reason for the discrepancy is that the neuromorphic accelerators use fixed-point format to represent numerical values, compared to floating-point format used by the CPU implementation.

2.5.2 Performance

This subsection discusses the performance of the proposed accelerators in terms of speed and power efficiency. Additionally, a quantitative comparison is performed between the accelerators and a multi-core CPU and GPU implementations.
The performance of the accelerated retina processor is compared to a CPU implementation developed using OpenCV [6] and executed on a 2.4 GHz Intel Xeon processor. Fig. 2.24 shows a frame-rate comparison between the CPU and accelerated retina processor. For the smallest scale, $384 \times 272$, the accelerated retina delivers 393 fps, 5.24X speedup compared to CPU. The speedup is even larger when processing the largest scale, $2048 \times 1536$, where accelerated retina processor outperforms the CPU by 15X.

![Fig. 2.24: A comparison of performance (frame rate) between CPU and accelerated retina processor](image)

Similarly, a GPU implementation of the AIM algorithm is used for comparison with the AIM accelerator. The GPU implementation is executed on an Nvidia GeForce GTS 250 with graphics and processor clock frequencies of 738 MHz and 1.84 GHz, respectively. Fig. 2.25 demonstrates the performance gain, in fps, of the AIM accelerator compared to the GPU. The figure shows the AIM accelerator outperforming the GPU by 2.7X for a $2048 \times 1536$ image and 11X for a $320 \times 240$ image.
On the other hand, the performance of the HMAX accelerators is compared to a software implementation based on [35] running on a 12-core Xeon 2.4 GHz processor. The software implementation was parallelized across all 12 cores and utilized SSE instruction set extension. Additionally, the performance is compared to an optimized GPU implementation coded in CUDA running on a Nvidia Tesla M2090 platform[39], which houses a Tesla T20A GPU, clocked at 1.3 GHz. Fig. 2.26 illustrates a performance comparison of execution time in fps for two configurations; namely, 4 and 12 orientations. Compared to the CPU, the HMAX accelerator delivers a speedup of 7.2X (7.6X) for 4-orientation (12-orientation) configuration when compared to CPU. Similarly, the HMAX accelerators deliver a speedup of 1.1X (1.24X) for the 4-orientation (12-orientation) configuration when compared to GPU.

Power efficiency in fps-per-watt of the HMAX accelerators is compared to CPU and GPU. The power consumption of the CPU while executing HMAX was measured and found
to be 116 Watts, while the GPU operated at a measured power consumption of 144 Watts. For
the purpose of measuring power consumption, the HMAX accelerators were also mapped to
a Virtex-5 [57] platform equipped with power measurement capabilities. The measurements
show a total of 69 Watts power consumption. Fig. 2.27 shows a power efficiency comparison,
where the HMAX accelerators outperformed the CPU by 12.1X (12.8X) for 4-orientation (12-
orientation) configuration. Moreover, the neuromorphic accelerators outperformed the GPU by
2.3X (2.6X) for 4-orientation (12-orientation) configuration.

The GPU implementation of HMAX is done using CUDA 4.0[40]. Performance im-
provement on the GPU is a combination of two factors: optimizing memory throughput and
maximizing thread-level parallelism. Nvidia’s Tesla M2090 GPU based on the SIMT (Single
Instruction Multiple Thread) architecture has vast parallel computing resources, ideal to accel-
erate the highly parallel, compute-intensive HMAX algorithm. The algorithm is implemented
in 4 stages: \( S_1 \), \( C_1 \), \( S_2 \), and \( C_2 \) sequentially, where the \( S \) stages involve convolution kernels and
the \( C \) stages involve pooling kernels. In the \( S_1 \) stage, convolution of the input layer with all
12 orientations is computed in parallel, where spatial convolution is used. In the \( C_1 \) stage, all
11 pairs of adjacent scales are processed in parallel. The \( S_2 \) stage is the most computation and
memory intensive operation and takes 19.8% of total GPU time as analyzed by the CUDA vi-

cual profiler[41]. The first 4 scales are processed in parallel followed by the next 7 and for each
kernel launch computation with all prototype patches is done in parallel. Memory throughput
is optimized as far as possible by using coalesced accesses to global memory and utilizing the
faster per-block shared memory wherever possible. Overall memory throughput seems to be
higher for the \( C \) stages than the \( S \) stages according to our results from the visual profiler. \( C_2 \) is
also implemented in 2 phases, with scales 0-5 pooled in parallel followed by scales 5-10.
2.5.3 Discussion of Performance Results

There are a number of factors that contribute to the speedup gained by the implemented neuromorphic accelerators compared to the CPU and GPU counterparts. First, the underlying communication infrastructure offers a high bandwidth transfer rate of up to 1.6 GB/s (3.2 GB/s) when operating the design at 100 MHz (200 MHz) clock frequency. However, the speedup achieved by the hardware accelerators is primarily contributed to the fully pipelined and customized streaming architecture. These customized architectures allow for data reuse, hence avoiding unnecessary data fetching.

For instance, in the retina processor, the architecture provides pixel-level parallelism concurrently across all operations in each stage. This high degree of parallelism is not achievable on general purpose CPU architectures as each sub-operation of the retina is executed sequentially. Contemporary CPU architectures with explicit vector-processing extensions lack the number of
Fig. 2.27: A comparison of power efficiency between CPU, GPU, and accelerated HMAX for two configurations. (Values are normalized to CPU)

functional units and optimized memory infrastructure to exploit the immense data-level locality inherent in the many convolution operations of both retina and AIM accelerators. Moreover, the tightly coupled pipeline stages of convolution and histogram building/indexing eliminates the overhead of storing intermediate convolution results.

Likewise, the HMAX accelerators exhibited significant power efficiency benefits. Since these accelerators are based on customized, pipelined architectures, high throughput can be achieved while operating at low frequency. Operating at low frequency is the main driver of low power consumption, and consequently high power efficiency.

The reader is reminded that performance results are obtained from mapping the accelerators to an FPGA platform. Increased speedup and power benefits will be realized if the accelerators are implemented in silicon (e.g. ASIC).
2.6 Highlights in Vortex and Comparisons to other system

Section 2.6.1 discusses five major aspects of Vortex which attempt to solve problems in composing heterogeneous accelerators for perceptual computing. Based on them, how other recent works deal with the problems is also identified and compared in section 2.6.2.

2.6.1 Five major aspects in Vortex

2.6.1.1 Intuitive dataflow mapping with abstraction

It has been observed that many visual perception algorithms can be mapped onto a form of dataflow process networks [32] in which nodes of basic building blocks feed streams of data to each other through uni-directional FIFO channels. Such dataflow process networks are a very powerful, yet simple model of computation. The dataflow graph naturally implies all the synchronization information by itself, so direct mapping of the graph into a system eliminates most of burdensome and error-prone tasks in design and verification of the system.

NIFs in Vortex are designed in consideration of direct mapping of those streaming accelerators onto NoC paradigm. Moreover, the virtual flow-id provides strong abstraction and autonomy by translating a virtual number to next-hop device address. It inherently eliminates the need of being aware of locations of other processing nodes in the communicational chain because those accelerators do not even need to know where they are located on the network. Obviously, they do not need to know where other processing nodes are located as well. All they care is the virtual number, flow-id since network interface must be pre-configured prior to initiating a transaction (this configuration happens at runtime) and network interface plays a role to translate the flow-id into next-hop device address. Only those SAPs at the very beginning of a transaction
and at the very end of the transaction need to know each other to set up the transaction, but even in this case they do not need to know what are the intermediate SOPs (streaming operators) involved in the transaction. This is how Vortex abstracts the dataflow for intuitive mapping.

2.6.1.2 Latency Hiding

One of the goals of Vortex is the support of computational chains. This literally means all happens concurrently; reading data from a SAP or a memory to initiate a transaction, one of more intermediate computations on the stream, and writing the final output into a SAP or a memory. As the size of a stream gets larger, it is more likely that all of the events occur simultaneously, and it promises the maximum degree of parallelism. Traditionally, a processing node takes three sequential steps to perform a job; 1) reading input data (or a chunk of it) into its local buffer over the network or wait until an external node, e.g. DMA, writes the data onto its local buffer instead, 2) perform its own computation on the input data and stores back the result into its local buffer, 3) write output data over the network to the next processing node or synchronize with external control to schedule a DMA on behalf. Thus, computation does not typically occur simultaneously with communication.

Vortex inherently hides the communication latency under computation latency. Assuming a data stream is large enough, all the computations in the chain occur simultaneously with communication.

2.6.1.3 Composability and Reusability

In section 2.3.4.2, I described how primitive building blocks compose an application-specific accelerator using an automation tool, called ChipMONK. Design and verification of an
accelerator takes relatively longer time and bigger effort compared to its software implementation, so re-using those building blocks at appropriate granularity is very critical. The very simple interface protocol in Vortex allows an easy composition of a complicated computing component in a timely manner.

2.6.1.4 2D DMA, built in network interface under abstraction

2D memory access happens very frequently in visual application and matrix computations, e.g. access of ROI (Region of Interest) in an image frame or sub-matrix of a large matrix. Such memory access pattern is programmable at run-time and associated with a virtual flow-id. Network interface automatically generates memory addresses based on the preconfigured burst length, burst stride, and burst count. Since such information is hidden under network interface and associated with the particular flow-id, not a single node in a full computational chain needs to know whether 2D memory access is currently happening.

2.6.1.5 Operational Mode, traveling with streams

Opcode, given to an SOP with a stream of data, is translated from the flow-id as configured. According to the opcode, an SOP may perform different tasks. It essentially allows for data packets to carry its own operational mode, represented by virtual flow-id. In addition, it removes redundant configuration time between different modes of operations at runtime, especially for latency-critical applications. Pre-configuration of potential modes with different flow-ids makes an equivalent effect at runtime by initiating a transaction with proper flow-id when necessary.
2.6.2 Comparison to other systems

2.6.2.1 Stream Processor [28]

William J. Dally introduced stream processor architectures, Imagine [27] and Merri-mac [15], starting in 1995. He founded SPI (Stream Processors Incorporated) in 2004 by delivering the Imagine Stream Architecture, the Stream programming model, software development tools, programmable graphics and real-time media applications, VLSI prototype, and Stream processor development platform. The company ceased operations in 2009.

His ideas were inspired by streaming notion captured in a dataflow graph model of media and signal processing applications as shown in Fig. 2.28. When a set of computation is mapped onto a dataflow graph model, which most image processing applications are naturally well-fit to, it is not very hard to identify locality and concurrency.

![Dataflow Graph targeted on Imagine Stream Processor](image)

Fig. 2.28: A Dataflow Graph targeted on Imagine Stream Processor [28]

In Stream Processor, Locality is exploited by three-level hierarchy of memory structure as shown in Fig. 2.29; Local Register File (LRF), Stream Register File (SRF), and external
DRAM for global memory access. LRF is used for temporary results that only need to be transferred between scalar operations within a kernel. LRF reduces expensive global communication bandwidth. SRF is shared among an array of ALU clusters in which its own LRFs are contained independently. Imagine contains 128 Kbyte of SRF with 22 simultaneous accesses (implemented as expensive flip-flops). The address space of SRF is directly exposed to all ALU clusters so the allocation of stream for each ALU cluster must be carefully done at compile time. SRF is also connected to DRAM Interface which sits on the top layer of memory hierarchy. The bandwidth between SRF and DRAM is only used for fetching input, storing output and passing global parameters. Accumulated bandwidths from the top layer of memory hierarchy are 16.9 Gbps, 92.2 Gbps, and 1,570 Gbps in Imagine Stream Processor.

![Imagine Stream Processor Architecture](image)

**Fig. 2.29: Architecture of Imagine Stream Processor [28]**

Concurrency is exploited through hierarchy as well in Imagine. Application processor is on the top hierarchy and resides outside of Imagine processor. Imagine processor supports Host
Interface to interact with Application processor which executes the application-level code and hands off the sequencing of operations to Stream Controller through Host Interface. Kernel execution unit (KEU) executes kernels and provide local communications within the kernel. KEU contains a single micro-controller which handles 8 ALU clusters in a single-instruction, multiple-data (SIMD) fashion. Each ALU cluster has its own interface to SRF and consists of 6 ALUs so KEU supports 48 ALUs in total. Kernel program is compiled together with application and transferred to the instruction memory of micro-controller in KEU at the beginning of operation.

The idea of Kernel execution unit (KEU) is also supported by Vortex when wrapped as a form of Stream-Attached Processor (SAP). Actually a very similar form of processing element is reported to be designed for Vortex, named SAP-PE [1] which consists of light-weight processor (LWP) and custom accelerators rather than arrays of general purpose ALU. Since custom accelerators can be replaced by arrays of ALUs, SAP-PE is more generic than KEU.

The first drawback of Stream Processor is that it does not naturally support the independent operation of streaming processing kernels. They are controlled by a single micro-controller. In fact, they are compiled for the micro-controller along with other kernels as a form of sequential software even though concurrent computations happen at each ALU clusters with coarse-grained synchronization. In addition, their data streams share the same SRF so their base addresses are allocated at compile-time in consideration of all other kernels’ needs. This also means the system is not easy to adapt dynamic variation of operation at runtime since their interactions are pre-determined at compile-time.

The second drawback comes from the array of general-purpose ALU clusters as a base operational unit. It intrinsically draws a theoretical limit of degree of optimization in terms of performance and energy consumption. Sequential instructions running on general purpose ALUs
hardly perform as customized hardware pipeline. Such an architecture also has clear benefits; no further synthesis, easy to debug using SDK, short development time, etc. However, high energy efficiency and performance do not easily come along with those benefits.

In-depth comparison to Stream Processor in the five aspects of Vortex follows:

1. Intuitive dataflow mapping: Because of the array structure of general purpose ALU with single micro-controller in Stream Processor, there is no intuitive dataflow mapping on hardware even though the streaming flow is implemented on software layer. Memory allocation and other system parameters are determined at compile time with consideration of a whole system, so generated binary for each kernel is already affected by other kernels. This fact hurts the autonomy of kernels in Stream Processor.

2. Latency Hiding: They argue a stream processor hides memory latency by fetching a stream of records with a single stream load instruction. It is true in a loose sense because fetching a chunk of data in burst manner absolutely shortens the fetching time rather than fetching a scalar value in-between subsequent operations. However, Vortex naturally allows the first token, e.g. pixel, gets operated on operator N while the second token is on operator (N-1) and so on. At the particular point of time, memory at the very beginning of the stream may be still fetching the (N+1)th token. Therefore, memory access and sequential operations literally performs concurrently to maximize the utilization of computational resources on Vortex.

3. Composability and Reusability: Since there is no hardware accelerator performing an application-specific operation, no composition or reuse of core is possible in a strict sense.
However, each kernel is written in modular StreamC language, so those building blocks may be reused for other applications.

4. 2D DMA: There is no hardware support reported in their articles for 2D memory access patterns. In case, each kernel may have to generate addresses by themselves.

5. Switching Operational Mode: Stream processor does not support hardware-oriented mode switching.

2.6.2.2 BONE-V5 [42]

An attention-based object-recognition algorithm for energy efficiency with a heterogeneous multicore architecture for data- and thread-level parallelism has been introduced in [42]. Its flow diagram is shown in Fig. 2.30.

![Flow Diagram of the proposed attention-based object-recognition algorithm](image)

Fig. 2.30: Flow Diagram, introduced in [42]
Its recognition process starts with Machine-learning engine (MLE) where attention map is generated for the input image by combining saliency map, object familiarity map, and temporal familiarity map. The resulting attention map is used to determine the location of ROI tiles to feed the subsequent recognition stage. Each tile is then processed in SMT-enabled feature extraction cluster (SFEC) which handles two tiles simultaneously, and the system consists of four SFECs interacting through System NoC in the backplane. Each SFEC also consists of one dual-vector processing element (DVPE) and four scalar processing elements (SPE) through a Local NoC, forming a star-ring topology in networking. SFEC performs the feature-detection and keypoint-description operations to generate the SIFT descriptors. Finally, feature-matching processor (FMP) matches extracted keypoints to the database utilizing customized matching hardware, which is the only major customized hardware in the whole system where other computations are still executed by sequential instructions on general purpose processors. The architecture of BONE-V5 is shown in Fig. 2.31.

It is noticeable that network interface which is used to attach FMP to System NoC supports 2D DMA which automatically generates the addresses for 2D data access for the data transactions of vision applications. This is very similar to 2D DMA built in network interface of Vortex.

In-depth comparison to BONE-V5 Processor in the five aspects of Vortex follows:

1. Intuitive dataflow mapping: Similarly to Stream Processor, there is no intuitive dataflow mapping on hardware because the system is composed of several heterogeneous but general purpose processors. Their data exchange pattern is determined by interactions among those processors defined solely by their software.
pipeline other recognition algorithms by programming SIMD, MIMD, and MP cores, respectively.

BONE-V5: SMT-enabled heterogeneous multicore processor

Figure 3 depicts the details of the BONE-V5 object-recognition processor, which uses a low-power heterogeneous multicore architecture with SMT operation. BONE-V5 consists of the main processing cores, a dynamic resource controller (DRC), and an external interface. The main processing cores include the throughput-optimized SFEC, the latency-optimized FMP, and the power-optimized machine learning engine (MLE) for different SIFT operations.

Once an ROI tile is allocated to an SFEC consisting of one dual-vector processing element (DVPE) and four scalar processing elements (SPEs), the SFEC performs the feature-detection and keypoint-description operations to generate the SIFT descriptors. The FMP compares the descriptors with those of the target objects in the database through the external interface. The MLE performs the attention operation by generating saliency and familiarity maps, and performing the machine-learning operation for dynamic resource management of DRC.

2. Latency Hiding: For feature-matching processor (FMP), cache- and database-based matching and locality-sensitive hash function are used to decrease the impact of database access latency. Although these techniques reduced the overall execution time on the only pipelined hardware, FMP, the communication infrastructure itself does not support concurrent processing and communication naturally.

3. Composability and Reusability: Similarly to Stream Processor, its major processing components are general purpose processors like DVPE or SPE. Unlike Stream Processor, DVPE or SPE has its own instruction memory so BONE-V is classified as heterogeneous multi-processor system on a chip. However, it still lacks of application-specific custom cores since there is only one hardware hard-IP, Feature-matching processor. Thus, it is not
designed in consideration of composability or reusability, while its low-level C-functions may be reused as a building block.

4. 2D DMA: Network interface that is used for FMP has its built-in 2D DMA functionality to generate memory addresses automatically.

5. Switching Operational Mode: 2D Reconfigurable PE cluster in BONE-V5 supports on-line reconfiguration to facilitate different precision levels with its own parallelism accordingly.

2.6.2.3 CogniServe [26]

CogniServe is a highly efficient recognition server, introduced by Intel, for large-scale recognition that employs a heterogeneous architecture to provide low-power, high-throughput cores, along with application-specific accelerators [26].

CogniServe is composed of many small cores (based on Intel’s Atom core) and several types of application-specific accelerators for visual or speech recognition: Gaussian mixture model(GMM), matching, and interest point detection(IPD).

Fig. 2.32: CogniServe accelerator programming and communication: device driver interface (a) and virtual memory-based acceleration (b) [26]
Their approach for further optimization was focused on reducing latencies of utilizing accelerators in the viewpoint of a whole system including operating system. Traditionally, application software handles device driver on OS kernel to interact with a particular accelerator and kernel buffer involves in data-exchange between user application to hardware accelerator. During the data transfer, redundant data copy from user buffer to kernel buffer, vice versa, hurts overall latency and throughput. To avoid such redundant procedure, they introduced new accelerator ISA (AISA) extensions in the host core to reduce unnecessary device driver overheads. At the same time, accelerator memory management unit (AMMU) is designed for user applications to be executed in the virtual-memory domain. The AMMU automatically translates virtual memory address into the corresponding physical address.

In-depth comparison to CogniServe in the five aspects of Vortex follows:

1. Intuitive dataflow mapping: Accelerators in CogniServe communicates through system bus and their interactions are governed by high-level software running on operating system. However, the authors share the motivation for a more efficient and intuitive programming interface for accelerators. They tried to eliminate significant overhead in system calls and data copies and enabled a direct user interface to accelerators via new instructions. Architectural supports let cores and accelerators operate in the same virtual-memory domain through AMMU.

2. Latency Hiding: Their approach to reduce communication latency with AISA and AMMU definitely enhances overall performance, but it still does not overlap communication time with computation time. Thus, it is hard to claim that CogniServe hides communication latency in a strict sense.
3. Composability and Reusability: Even though CogniServe is equipped with several accelerators, they are internally very tightly-coupled pipeline and not designed in modular manner. This prevents sub-blocks of accelerators from being reused for another.

4. 2D DMA: No hardware support for 2D memory access is reported upon CogniServe.

5. Switching Operational Mode: No hardware-oriented operational mode switching is reported in CogniServe.

2.7 Conclusion

We have analyzed the characteristics of neuromorphic vision algorithms to propose a methodology of implementing such algorithms on an SoC in a structural and efficient manner. As a communication fabric among neuromorphic accelerators, the interconnection network requires flexibility, scalability, programmability, and high performance. In order to meet these requirements, a reconfigurable NoC platform is proposed, in which special network interfaces provide frame-level granularity and application-level abstraction. We demonstrate how the Vortex platform fits very well to mapping dataflow graphs onto networks of domain specific custom accelerators.

After analyzing the requirements for customized accelerators, we have found that the degree of parallelism, power efficiency, parameterization, composability, and programmability are the key factors in realizing the full potential of neuromorphic algorithms. We propose a methodology for composing streaming operators in a hierarchical and automated fashion. In addition, we standardize the way in which compute-intensive accelerators are attached to the network and interact with other processing elements.
A system-level automation tool, *Cerebrum* presented in [1], assists users in prototyping and validating designs easily and efficiently on multi-FPGA systems on top of Vortex framework. *Cerebrum* supports system composition even for users with little knowledge of hardware system design by performing mapping, synthesis, and configuration. It also highlights the aspect of Vortex framework that makes higher-level automation tractable in an efficient manner.

A case study of a neuromorphic system is demonstrated and evaluated on a multi-FPGA system. The neuromorphic system performs retina preprocessing, visual saliency, and object recognition in a single pipeline. Significant performance gains are achieved compared to multi-core CPU and GPU implementations.

Finally, other systems are compared to Vortex framework in its five major aspects. Though some of the highlighted aspects of Vortex are found similar in other works, Vortex can be evaluated as the first framework to integrate heterogeneous accelerators in the application domain, equipped with all the aspects in itself.
Chapter 3

Energy Optimization of Multi-object Recognition

3.1 Introduction

Embedded vision systems are employed in many applications including surveillance[7], automotive[2], and robotics[49]. However, the computational power required by these systems mandates large power consumption.

A recent study of vision processing reveals that biologically inspired neuromorphic algorithms exhibit promising energy-efficiency while maintaining the level of accuracy in comparison with traditional computer vision algorithms[17]. However, human brain or primate visual cortex is known to be energy-efficient in orders of magnitude than any of the artificial developments so far. Thus, further effort to achieve better energy-efficiency is required on such a computationally expensive applications.

![Figure 3.1: Typical processing flow across major computational blocks in object recognition system](image)

Fig. 3.1: Typical processing flow across major computational blocks in object recognition system

Major computational blocks in neuromorphic vision domain are actively under study to mimic mammal visual pathway, but for the best of our knowledge, the relationship among
those major blocks has not been exhaustively studied, at least in terms of optimization across the blocks. Typical processing flow across these blocks is shown in Fig.3.1. In this work, we explore the design space of HMAX for biologically inspired feature extraction and classification in the viewpoint of energy efficiency. Based on the result, a novel method to further reduce energy consumption is proposed while maintaining the level of accuracy. It was achieved by utilizing and propagating the findings from visual saliency in an effective way to control the effort-level of next stages in HMAX.

By investigating insightful optimization across the saliency model and HMAX, this work proposes an approach to achieve better energy-efficiency of overall visual perception system. We first examine the confidence measure of the salient regions according to the ranking, or order, of the region suggested by saliency algorithm. The design space of HMAX is independently explored to inspect the impact of each design parameter on classification accuracy and energy consumption. Based on the findings, we extract a novel energy-efficient configuration scheme by handling each candidate Region of Interest, ROI, differently according to the saliency output, as the ranking of regions. Hardware accelerators and their architectures for the saliency and HMAX are already discussed in chapter 2.

Section 3.2 highlights the niche in the state-of-the-art of neuromorphic object recognition implementations in terms of energy efficiency. Experimental methods to obtain saliency confidence score and to explore the design space of HMAX are discussed in section 3.3.

Section 3.4 presents the results of our experiments while section 3.5 discusses the interpretation of the results and proposes a configuration scheme to further reduce the energy consumption. Section 3.6 summarizes and concludes this work.
3.2 Motivation

Exhaustive search in a relatively large image by applying independent feature extraction and classification for every possible sliding window involves unrealistic amount of computations for real-time processing. Thus, the saliency model is mainly used to extract the most prominent patches, or regions, to be propagated to the next processing block; feature extraction like HMAX.

Typically, individual patches are treated equally by HMAX with pre-determined parameters; e.g. the number of orientations, the number of scales, and the number of prototypes to be involved in the computation. These parameters actually dictate the effort-level of HMAX by imposing direct impact on energy consumption and classification accuracy. Furthermore, saliency algorithms do not necessarily guarantee the presence of “true” objects within targeting object-categories in various applications in every detected ROI. Thus, we propose a dynamic reconfiguration scheme for HMAX to leverage the effort-level based on the saliency ranking of a patch in order to achieve better energy efficiency in embedded platforms.

In Fig. 3.2, for example, bounding boxes of top 5 patches on a natural image are presented with their rankings by our saliency implementation. The first and second ranking patches correctly detect children in the scene, but others unfortunately do not include any object of our interest. Likewise, depending on the number of target objects in natural scenes, higher ranking patches generally introduces better accuracy in identifying “true” objects, thus it is a natural optimization to allocate more processing power on such higher ranking patches rather than patches with lower probability of including objects of interest. This optimization is inspired by the anatomical fact that rods and cones are not uniformly distributed on the retina of the eye.
3.3 Experimental Methods

In order to derive meaningful configuration scheme, two experiments were conducted: (1) analysis of “saliency confidence” according to the ranking of patches by saliency algorithm with a number of natural images, and (2) analysis of accuracy to energy trade-off in the design space of HMAX.

3.3.1 Saliency Confidence Score

The evaluation is founded on Vortex framework to explore the design space for energy efficiency. All the results are based on measurements from the prototype executing on a Xilinx
Virtex-6 SX475T platform[58], and it consumes 92,395 slice registers, 66,404 slice LUTs, 368 BRAMs, and 182 DSPs.

Instead of applying large kernels trained by ICA as suggested by [8], empirically equivalent 2D Gabor kernels are used to reduce the computational load. Building and indexing histograms naturally invoke two-pass of processing to prevent large internal frame-buffer of image size, but dual-histogram architecture allows indexing of previously-built histograms simultaneously as building new histograms for different set of kernels for other orientations and scales of Gabor filter. In this way, building and indexing histograms are fully pipelined and almost equivalent to single-pass processing overall. The distribution of histogram for each kernel gives likelihood at each pixel location for the kernel, and joint-likelihood is calculated by sum of log-likelihood to optimize the computation and the precision instead of multiplying all likelihood values.

The saliency implementation returns a saliency map that contains self-information measures at each pixel of the original image associated with its surroundings. A scheme was developed to extract bounding boxes (patches) in the order of highest saliency score. This order naturally gives a ranking among multiple bounding boxes in a scene. The size of these patches is fixed to $256 \times 256$ as they are fed to the HMAX model.

It is expected that the output of such a scheme based on the saliency model may contain false positives (i.e. no objects in the target categories are included in a patch). Therefore, it is very important and meaningful to analyze and quantify the possibility of hitting “true” objects, called saliency confidence score, with relation to the ranking, or order, of the region suggested by saliency map.
TU Graz-02 database[43] was used in the experiments to explore saliency confidence score because it has rich annotations for most objects of interest. It has 900 annotated images for three categories; car, bike, and person. In order to pick the most salient regions, when saliency map is given by the algorithm, $256 \times 256$ of bounding box around the highest salient point is first extracted, and then all the self-information values in the bounding box are set to zero, not to be considered as salient point for next iterations. This simple procedure iteratively picks salient regions in order until the number of regions reaches the predefined maximum, which was 7 in our experiment. By applying this scheme, 7 ranked-patches are extracted in all 900 images in the database.

For each salient region, we examined whether the highest salient point in the region is included in any bounding box of annotations that are already marked as object in the database. If it does not fall in any of the bounding boxes, it is regarded as “a false positive” by the saliency algorithm. By accumulating “true positives” separately for the ranking of the region picked in the iteration, we were able to analyze the confidence score based on the rankings. Intuitively, this confidence score tells us how correctly each ranking picks up a “true” object in a scene.

### 3.3.2 Exploration of HMAX Design Space

This section discusses the three major parameters in the design space of a typical HMAX implementation as well as how we control them in our experiments. The implementation of HMAX [1] is mapped to a multi-FPGA platform, which consists of a quad-core Xeon processor with four Virtex-5 SX240T FPGAs[57] through a Front-Side Bus, FSB, and it utilizes 316,794 slice registers, 133,611 slice LUTs, 623 BRAMs, and 2,206 DSPs in total. To identify the impact
of each dimension in the design space of HMAX, we used in-house vehicle dataset with 1,382 images in 16 categories, e.g. vehicles, aircrafts, military equipments, and background scenery.

3.3.2.1 Number of orientations

In the $S_1$ stage of HMAX, the $256 \times 256$ input patch is convolved with $11 \times 11$ kernels in sliding window fashion. The $S_1$ output consists of feature response maps as many as the number of orientations at each scale. Regardless whether those orientations, or set of coefficients, are processed in parallel or in sequence, the number of orientations is one of the major design parameters to trade-off between classification accuracy and energy consumption. We controlled the number of orientations as either 4 or 12 due to the nature of symmetry while maintaining the number of scales as 12 and the number of prototypes as 5120.

3.3.2.2 Number of scales

The number of scales determines the scale-space pyramid that can potentially be processed in parallel at every computational stage of HMAX. Reducing the number of scales significantly brings down the amount of computation, which in turn reduces total energy consumption. However, since the pyramid plays an important role in extracting ‘scale-invariant’ features, reducing number of scales degrades the classification accuracy.

In the experimental setup, we vary the number of scales in order to quantify the impact of varying number of scales on both classification accuracy and energy consumption. We started with only 4 scales, and incrementally increased the number of scales until the maximum number of scales (i.e. 12) is reached. Both classification accuracy and energy consumption are measured in each configuration.
3.3.2.3 Prototypes coverage

$S_2$ stage of HMAX consists of prototype matching of the most prominent and invariant features produced by $C_1$ stage against a large dictionary of prototypes that was populated off-line. The $S_2$ stage dominates the execution time in the HMAX model. Reducing the number of prototypes decreases the overall execution time of $S_2$ stage. However, reducing the size of the dictionary has a negative impact on the overall classification accuracy.

Our initial setup used a dictionary that contained 5120 prototypes, out of which only 10% was used initially. Measurements of classification accuracy and execution time were done by adding additional 10% at every run until 100% coverage of the prototypes are used. The number of scales and orientations are fixed to 12 throughout the experiment.

3.4 Result

3.4.1 Saliency Confidence Score

Fig. 3.3 shows the reliability of salient regions based on the ranking suggested by the saliency algorithm from our experiment using 900 images in TU Graz-02 database. As shown in the graph, the chance of hitting “true” objects drops dramatically when taking lower ranking regions. It does not need to be interpreted as we may only need top 3 regions from this saliency algorithm for subsequent recognition processing because the absolute range of confident rankings still depend on various factors like size of the original images as well as the average number of target objects appeared in every images in the database we performed on. However, we want to emphasize that no matter how many regions we feed to the subsequent recognition process, it is inevitable to end up wasting redundant energy if we treat them in equal effort because lower
ranking regions do not contribute to overall expected accuracy as much as higher ranking ones. Further discussion on this matter is elaborated in section 3.5.

Fig. 3.3: Saliency Confidence Score over rankings of salient regions for 900 annotated images from TU Graz-02 Database

### 3.4.2 Exploration Results on the design space of HMAX

We have observed the impact of three independent design parameters in terms of energy efficiency: the number of orientations, the number of scales, and prototype coverage. In each configuration, the classification accuracy, per-frame processing time, and the average power consumption are measured to calculate the per-frame energy consumption as well as accuracy-to-energy ratio. Accuracy-to-energy ratio is defined as the achieved accuracy rate per unit energy(1J).
Throughout the whole experiments, average power consumption was measured and was found to be 372 Watts on average since the experimental system was executed on a server-type platform where host machine and FPGAs are tightly coupled via FSB. Because batch job of processing all images in our database was executed back to back, power consumption was not much affected while execution-time per frame became the major variation according to the configuration.

### 3.4.2.1 Number of orientations

Only two different number of orientations were explored mainly due to the nature of symmetry of Gabor filters. As shown in Table 3.1, we observed only 1.1% of accuracy-drop when number of orientations decreased from 12 to 4, which ended up saving 56.7% of energy. Throughout this article, fr stands for frame.

<table>
<thead>
<tr>
<th>Orientation</th>
<th>Accuracy (%)</th>
<th>Runtime (ms/fr)</th>
<th>Energy (J/fr)</th>
<th>Accuracy to Energy Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>81.9</td>
<td>420</td>
<td>155.82</td>
<td>0.526% / J</td>
</tr>
<tr>
<td>12</td>
<td>83.07</td>
<td>970.9</td>
<td>360.20</td>
<td>0.231% / J</td>
</tr>
</tbody>
</table>

### 3.4.2.2 Number of scales

Table 3.2 shows a summary of the design exploration results obtained by varying the number of scales. Classification accuracy dropped from 83.07% down to 70.69% when number of scales decreased from 12 to 5. However, energy consumption dropped significantly during the variation, from 360J/fr to 25J/fr. Consequently, accuracy-to-energy ratio dramatically increased.
from 0.23%/fr to 2.78%/fr, which indicates that the number of scales may provide a good trade-off for energy-efficiency for applications that do not require a very high accuracy rate. More comprehensive discussion with other results can be found in section 3.5.

Table 3.2: Exploration result on various scale parameters in HMAX

<table>
<thead>
<tr>
<th>Scale</th>
<th>Accuracy (%)</th>
<th>Runtime (ms/fr)</th>
<th>Energy (J/fr)</th>
<th>Accuracy to Energy Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>70.69</td>
<td>68.6</td>
<td>25.45</td>
<td>2.777 %/J</td>
</tr>
<tr>
<td>6</td>
<td>77.06</td>
<td>99.8</td>
<td>37.02</td>
<td>2.081 %/J</td>
</tr>
<tr>
<td>7</td>
<td>77.57</td>
<td>160.5</td>
<td>59.54</td>
<td>1.302 %/J</td>
</tr>
<tr>
<td>8</td>
<td>79.31</td>
<td>235.3</td>
<td>87.29</td>
<td>0.908 %/J</td>
</tr>
<tr>
<td>9</td>
<td>79.96</td>
<td>333.8</td>
<td>123.83</td>
<td>0.645 %/J</td>
</tr>
<tr>
<td>10</td>
<td>81.19</td>
<td>499.7</td>
<td>185.38</td>
<td>0.437 %/J</td>
</tr>
<tr>
<td>11</td>
<td>83.14</td>
<td>705.7</td>
<td>261.81</td>
<td>0.317 %/J</td>
</tr>
<tr>
<td>12</td>
<td>83.07</td>
<td>970.9</td>
<td>360.20</td>
<td>0.230 %/J</td>
</tr>
</tbody>
</table>

3.4.2.3 Prototype coverage

As prototype coverage was decreased from 100% to 10%, the classification accuracy has dropped significantly from 83% down to only 4%. However, energy consumption was reduced by about 63% – from 360 J/fr down to 135 J/fr – which pulled down the accuracy-to-energy ratio to 0.03 %/J as shown in Table 3.3.

3.5 Discussion

Analysis on saliency confidence score shown in Fig. 3.3 proves that multiple regions in an image frame are not worth being treated equally in multi-object recognition scenario. Fig. 3.3 shows that the possibility of hitting a “true” object drops below 10% beyond top three regions suggested by the saliency algorithm in the dataset we have chosen. The absolute number of confident rankings is highly affected by multiple factors; e.g. the image size in database, average
Table 3.3: Exploration result on various coverage of prototypes in HMAX

<table>
<thead>
<tr>
<th>Prototype Usage (%)</th>
<th>Accuracy (%)</th>
<th>Runtime (ms/fr)</th>
<th>Energy (J/fr)</th>
<th>Accuracy to Energy Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4.12</td>
<td>363.1</td>
<td>134.71</td>
<td>0.031 %/J</td>
</tr>
<tr>
<td>20</td>
<td>4.49</td>
<td>427.8</td>
<td>158.71</td>
<td>0.028 %/J</td>
</tr>
<tr>
<td>30</td>
<td>28.80</td>
<td>495.5</td>
<td>183.83</td>
<td>0.157 %/J</td>
</tr>
<tr>
<td>40</td>
<td>31.26</td>
<td>562.9</td>
<td>208.84</td>
<td>0.150 %/J</td>
</tr>
<tr>
<td>50</td>
<td>54.70</td>
<td>632.0</td>
<td>234.47</td>
<td>0.233 %/J</td>
</tr>
<tr>
<td>60</td>
<td>61.43</td>
<td>698.4</td>
<td>259.11</td>
<td>0.237 %/J</td>
</tr>
<tr>
<td>70</td>
<td>67.87</td>
<td>764.6</td>
<td>283.67</td>
<td>0.239 %/J</td>
</tr>
<tr>
<td>80</td>
<td>72.72</td>
<td>833.9</td>
<td>309.38</td>
<td>0.235 %/J</td>
</tr>
<tr>
<td>90</td>
<td>82.05</td>
<td>901.8</td>
<td>334.57</td>
<td>0.245 %/J</td>
</tr>
<tr>
<td>100</td>
<td>83.07</td>
<td>970.9</td>
<td>360.20</td>
<td>0.231 %/J</td>
</tr>
</tbody>
</table>

number of objects appeared in each image, and the quality of annotations in database. However, the relative trend of saliency confidence according to the ranking or priority needs to be noted.

For feature extraction and classification as in HMAX, we learn that reducing prototype coverage does not help improving energy-efficiency since the classification accuracy drops much earlier than the energy consumption does. Other two configuration parameters, number of orientations and scales, actually show improvement of accuracy-to-energy ratio when we attempt to reduce the effort level as clearly shown in Fig. 3.4. Thus, further discussion will ignore prototype coverage as a control parameter.

With two dimensional configuration space (number of orientations/scales), a configuration can be defined as followings:

\[ conf_i := \{ j_i, k_i \} \]  

\[ conf := \{ conf_1, conf_2, conf_3, \ldots, conf_N \} \]
where $j_i$ denotes the number of orientations for $i^{th}$-ranking region suggested by saliency algorithm while $k_i$ denotes the number of scales for the ranking. In other words, $conf_i$ means a particular combination of orientations and scales for $i^{th}$-ranking region. Accordingly, $conf$ is defined as a global configuration for the full rankings of $N$ regions.

Since we know that HMAX classification accuracy is only dependent on $conf_i$ for $i_{th}$-ranking region, and the region has its own probabilistic confidence score depending on the given ranking, expected(average) accuracy $[A_{conf}]$ can also be defined as following once $conf$ is fixed:

$$[A_{conf}] = \sum_{i=1}^{N} C_i A_{cons_i} \quad (3.3)$$

where $C_i$ denotes the probability mass function(pmf) of saliency confidence score by normalization to make the sum equal to 1. Now the pmf of saliency confidence score plays a role to distribute the contributions of accuracy across the rankings according to their own probability, and $A_{cons}$ is still a good measure of accuracy as a whole.
In addition to the definition of expected accuracy, total energy consumption for a certain configuration, $E_{conf}$, is defined as following:

$$E_{conf} = \sum_{i=1}^{N} E_{conf_i}$$ (3.4)

Total energy consumption $E_{conf}$ is sum of energy consumption of each region based on its own configuration. The best effort case for all the regions, for example as taken earlier, consumes 1801 J which may sound even unrealistically large just because we ran the experiments on server-based FPGA platform.

An analysis of expected accuracy $A_{conf}$ and total energy consumption $E_{conf}$ with the results of our experiments has been done with $N = 5$, $j_i = (4,12)$, $k_i = (5,6,7,8,9,10,11,12)$. Since there are 2 possible configuration choices of $j_i$ for the number of orientations of a region for a particular ranking $i$ and similarly 8 choices of $k_i$ for the number of scales, we have 16 unique configuration of HMAX for each ranking $i (= 1 \ldots 5)$. Such an environment gives $16^5 = 1,048,576$ unique configurations for the full HMAX configuration as we pick configuration of $i^{th}$-ranking completely independent of each other. Table 3.4 lists several sample cases of full configuration, and their analytical results are given in Table 3.5.

<table>
<thead>
<tr>
<th>Config. (orient/scale)</th>
<th>1st Region</th>
<th>2nd Region</th>
<th>3rd Region</th>
<th>4th Region</th>
<th>5th Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>A (best effort)</td>
<td>(12,12)</td>
<td>(12,12)</td>
<td>(12,12)</td>
<td>(12,12)</td>
<td>(12,12)</td>
</tr>
<tr>
<td>B</td>
<td>(4,12)</td>
<td>(4,12)</td>
<td>(4,12)</td>
<td>(4,12)</td>
<td>(4,12)</td>
</tr>
<tr>
<td>C</td>
<td>(4,8)</td>
<td>(4,8)</td>
<td>(4,8)</td>
<td>(4,8)</td>
<td>(4,8)</td>
</tr>
<tr>
<td>D</td>
<td>(12,12)</td>
<td>(12,9)</td>
<td>(12,7)</td>
<td>(4,7)</td>
<td>(4,5)</td>
</tr>
<tr>
<td>E</td>
<td>(4,5)</td>
<td>(4,7)</td>
<td>(4,7)</td>
<td>(4,7)</td>
<td>(4,5)</td>
</tr>
<tr>
<td>F (least effort)</td>
<td>(4,5)</td>
<td>(4,5)</td>
<td>(4,5)</td>
<td>(4,5)</td>
<td>(4,5)</td>
</tr>
</tbody>
</table>
Table 3.5: Results of sample configurations

<table>
<thead>
<tr>
<th>Cfg.</th>
<th>Expected Accuracy</th>
<th>Energy Cons.</th>
<th>Accuracy Loss Δ</th>
<th>Energy Saving</th>
<th>Accuracy to Energy Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>83.06 %</td>
<td>1801 J</td>
<td>0.00 %</td>
<td>0.00 %</td>
<td>0.00 %</td>
</tr>
<tr>
<td>B</td>
<td>81.90 %</td>
<td>779 J</td>
<td>1.16 %</td>
<td>56.75 %</td>
<td>0.1051 %/J</td>
</tr>
<tr>
<td>C</td>
<td>78.19 %</td>
<td>189 J</td>
<td>4.87 %</td>
<td>89.51 %</td>
<td>0.4141 %/J</td>
</tr>
<tr>
<td>D</td>
<td>80.54 %</td>
<td>580 J</td>
<td>2.52 %</td>
<td>67.80 %</td>
<td>0.1388 %/J</td>
</tr>
<tr>
<td>E</td>
<td>73.92 %</td>
<td>580 J</td>
<td>9.14 %</td>
<td>67.80 %</td>
<td>0.1274 %/J</td>
</tr>
<tr>
<td>F</td>
<td>69.69 %</td>
<td>55 J</td>
<td>13.37 %</td>
<td>96.95 %</td>
<td>1.2661 %/J</td>
</tr>
</tbody>
</table>

Firstly, we set two different accuracy targets in terms of expected accuracy $A_{conf}$, depending on the need of the application: accuracy-critical (82%) or energy-critical (78%). Because configuration of HMAX is dynamically changing at every transition of rankings, moving the full configuration to another set for dynamic adaptation of environmental change is also naturally supported. Such a run-time configuration retrieval is a unique feature of the communication infrastructure on which our experiment is performed [46]. When $A_{conf}$ for all possible combination of full configuration is calculated, all the configurations with less than the target accuracy are discarded. Secondly, remaining configurations are sorted according to the ratio of expected accuracy to total energy consumption ($A_{conf}/E_{conf}$) in order to find the optimum in terms of energy efficiency. Top 3 configurations are listed with their analytical results in Table 3.6 and 3.7 for accuracy-critical applications as well as in Table 3.8 and 3.9 for energy-critical applications, in the order of $A_{conf}/E_{conf}$.

Table 3.6: Top 3 energy-efficient configuration for accuracy-critical applications (Target accuracy = 82%)

<table>
<thead>
<tr>
<th>Top Config. (orient/scale)</th>
<th>1st Region</th>
<th>2nd Region</th>
<th>3rd Region</th>
<th>4th Region</th>
<th>5th Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(12,11)</td>
<td>(12,6)</td>
<td>(4,11)</td>
<td>(4,6)</td>
<td>(4,6)</td>
</tr>
<tr>
<td>2</td>
<td>(12,9)</td>
<td>(12,8)</td>
<td>(12,6)</td>
<td>(12,6)</td>
<td>(4,11)</td>
</tr>
<tr>
<td>3</td>
<td>(12,8)</td>
<td>(12,8)</td>
<td>(12,8)</td>
<td>(4,11)</td>
<td>(4,6)</td>
</tr>
</tbody>
</table>
Table 3.7: Results of top 3 configurations for accuracy-critical applications (Target accuracy = 82%)

<table>
<thead>
<tr>
<th>Top Cfg.</th>
<th>Expected Accuracy</th>
<th>Energy Cons.</th>
<th>Accuracy Loss Δ</th>
<th>Energy Saving</th>
<th>Accuracy to Energy Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>82.0157 %</td>
<td>530 J</td>
<td>1.05 %</td>
<td>70.57 %</td>
<td>0.1547 %/J</td>
</tr>
<tr>
<td>2</td>
<td>82.0264 %</td>
<td>540 J</td>
<td>1.04 %</td>
<td>70.03 %</td>
<td>0.1519 %/J</td>
</tr>
<tr>
<td>3</td>
<td>82.0544 %</td>
<td>541 J</td>
<td>1.02 %</td>
<td>69.94 %</td>
<td>0.1516 %/J</td>
</tr>
</tbody>
</table>

Table 3.8: Top 3 energy-efficient configuration for energy-critical applications (Target accuracy = 78%)

<table>
<thead>
<tr>
<th>Top config. (orient/scale)</th>
<th>1st Region</th>
<th>2nd Region</th>
<th>3rd Region</th>
<th>4th Region</th>
<th>5th Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(4,9)</td>
<td>(4,8)</td>
<td>(4,7)</td>
<td>(4,6)</td>
<td>(4,6)</td>
</tr>
<tr>
<td>2</td>
<td>(4,9)</td>
<td>(4,9)</td>
<td>(4,6)</td>
<td>(4,6)</td>
<td>(4,6)</td>
</tr>
<tr>
<td>3</td>
<td>(4,8)</td>
<td>(4,8)</td>
<td>(4,8)</td>
<td>(4,7)</td>
<td>(4,6)</td>
</tr>
</tbody>
</table>

Table 3.9: Results of top 3 configurations for energy-critical applications (Target accuracy = 78%)

<table>
<thead>
<tr>
<th>Top Cfg.</th>
<th>Expected Accuracy</th>
<th>Energy Cons.</th>
<th>Accuracy Loss Δ</th>
<th>Energy Saving</th>
<th>Accuracy to Energy Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>78.0024 %</td>
<td>149 J</td>
<td>5.068 %</td>
<td>91.72 %</td>
<td>0.5231 %/J</td>
</tr>
<tr>
<td>2</td>
<td>78.0988 %</td>
<td>155 J</td>
<td>4.971 %</td>
<td>91.38 %</td>
<td>0.5033 %/J</td>
</tr>
<tr>
<td>3</td>
<td>78.0322 %</td>
<td>155 J</td>
<td>5.038 %</td>
<td>91.39 %</td>
<td>0.5033 %/J</td>
</tr>
</tbody>
</table>
Fig. 3.5: Accuracy loss $\Delta$ and Energy savings of top 3 configurations for accuracy-critical and energy-critical applications

Accuracy loss $\Delta(\%)$ and energy savings($\%$) compared to the best effort configuration, where treating all 5 patches with 12 scales and 12 orientations, are presented in Fig. 3.5 showing top 3 configurations for accuracy-critical and energy-critical application. For accuracy-critical applications, we achieved 70% of energy savings while maintaining accuracy loss at around 1%. On the other hand, more than 91% of energy savings were denoted with 5% of accuracy loss for energy-critical applications.

3.6 Conclusion

In multi-object recognition system, neuromorphic approaches find multiple regions of interest by saliency model to pass down to extract features. These features are used to classify the object among pre-learned categories of target objects. Typically, these two stages work separately and the latter processing usually consider multiple regions in an image equally. Since there
exists a ranking, or priority, suggested by the former processing, however, taking the rankings into account leads us to achieve better energy-efficiency by spending limited energy-budget more effectively. This work attempted to maximize energy-savings with minimum accuracy-loss.

In such a system design with dynamic configuration, the configuration scheme itself plays a very important role. If the runtime configuration takes too much time or energy by itself, then we may lose more than what we gain, or real-time requirements may not be met. Because Vortex framework is designed to pre-configure enough number of possible cases, it introduces no extra energy burden to retrieve them at runtime.

From our experiments, we achieved up to 70.57% of energy saving for accuracy-critical applications compared to the case of treating all 5 regions at the best effort while only 1.05% of expected accuracy was lost. Similarly, for energy-critical applications, we have proposed a dynamic configuration which saves 91.72% of energy consumption while 5.07% of expected accuracy drop happened.
Chapter 4

Acceleration of Signal Reconstruction
by Compressed Sensing for Body Signals

4.1 Introduction

Wireless body-area networks (WBANs) are currently an area of very active research [54][13]. Yet, the challenges include the tight energy budget limited by operating the portable health-monitoring device on batteries.

Compressed sensing algorithms exhibit promising capability to tremendously reduce the sensing energy and wireless transmission energy not only on the individual sensors but also on the portable device [34][11]. They essentially enable the sensors to sample at sub-Nyquist rate to reduce the sensing energy [14]. Furthermore, the dimension of samples to transmit is far smaller than that of original signals so wireless transmission energy also decreases proportionally [12].

Compressed sensing, however, requires the receiver to reconstruct the original signal, and many algorithms have been proposed to efficiently solve the under-constrained optimization problem of signal reconstruction, also known as sparse recovery. The algorithms include Approximate Message Passing (AMP) [19], Fast Iterative Shrinkage-Thresholding Algorithm (FISTA) [5], Homotopy algorithm [33], Orthogonal Matching Pursuit (OMP) [47], and Block Sparse Bayesian Learning (BSBL) [59]. Among these algorithms, BSBL is most recently proposed and shows outstanding performance compared to others [59]. Its performance is also evaluated with EEG
signals in this work, and it demonstrates over 60 dB of average accuracy while other algorithms only shows under 20 dB of average accuracy.

However, the amount of computation involved in BSBL makes it difficult to be applied in portable devices with real-time constraints for multi-channel signals like EEG. Thus, this work introduces analytical optimization and approximation of BSBL, tailored at hardware acceleration. Moreover, novel architectures of hardware accelerators and the system composition on NoC communication infrastructure is also proposed, which is the first hardware acceleration system of BSBL to the best of our knowledge.

Section 4.2 begins with pseudo-code of BSBL to introduce the optimizations and approximations we achieved, and micro-architecture of each accelerator is illustrated in section 4.3. Experimental results by prototyping the acceleration system on multi-FPGA board are given in section 4.4, with comparisons in terms of accuracy, power, and performance among several platforms.

4.2 Optimizations and Approximations for Implementing Hardware Accelerators

4.2.1 BSBL-BO Algorithm

Compressed sensing obtains a sample $y$ of dimension $M$ by multiplying original signal $x$ of dimension $N$ ($\gg M$) to a sensing matrix $\Phi$ of dimension $M \times N$ as shown in Eq. (4.1).

$$y = \Phi x$$  \hspace{1cm} (4.1)
The task of signal reconstruction or sparse recovery is to recover the original signal from $y$ and $\Phi$. BSBL-BO (Block Sparse Bayesian Learning with Bounded Optimization) is one of the sparse recovery algorithms which exhibits the best performance among those introduced yet. Fig. 4.1 summarizes the pseudo-code of BSBL-BO.

### 4.2.2 Computing $PBP$

At the beginning of iterations in BSBL-BO algorithm, $PBP$ is calculated by accumulating $\Phi_i\Sigma_0\{i\}\Phi_i^T$ for each block as shown at Line 2 in Fig. 4.1.

$$
\Phi_i\Sigma_0\{i\}\Phi_i^T = \Phi_i\gamma_i B\Phi_i^T
$$

$$
= \gamma_i[\Phi_i^T B\Phi_i^T]
$$

$$
:= PBP_i
$$

(4.2)

Considering the fact that $\Sigma_0\{i\}$ is calculated by multiplying a scalar value $\gamma$ to matrix $B$ for each block $i$ (Line 19 in Fig. 4.1), we can factor out $\gamma$ out of two consecutive matrix multiplications. Furthermore, $B$ is Toeplitz matrix defined by a scalar value $b$. Thus, we can easily draw out each element of $\Phi_i B\Phi_i^T$ ($:= PBP_i(l,m)$) in terms of scalar $b$ with only two rows, $l_{ih}$ and $m_{ih}$, of $\Phi_i$ as described in Eq. (4.3).
1: while iteration < max_iter do
2: \( PBP \leftarrow \sum_{i=1}^{p} [\Phi_i \Sigma_0 \{i\} \Phi_i^T] \)
3: \( H \leftarrow \Phi_i^T \times \{PBP + \lambda I_M\}^{-1} \)
4: \( H_y \leftarrow H \times y \)
5: \( H \Phi \leftarrow H \times \Phi \)
6: for \( i = 1 \rightarrow p \) do
7: \( \mu_i \leftarrow \Sigma_0 \{i\} \times H_{yi} \)
8: \( \Sigma_x \{i\} \leftarrow \Sigma_0 \{i\} - \Sigma_0 \{i\} \times H \Phi \{i, i\} \times \Sigma_0 \{i\} \)
9: \( C_x \{i\} \leftarrow \Sigma_x \{i\} + \mu_i \times \mu_i^T \)
10: end for
11: \( B_0 \leftarrow \sum_{i=1}^{p} C_x \{i\} \)
12: \( b \leftarrow \frac{\text{avg}(d_1)}{\text{avg}(d_0)} \)
13: if \( \text{abs}(b) \geq 0.99 \) then
14: \( b \leftarrow \text{sign}(b) \times 0.99 \)
15: end if
16: \( B \leftarrow \text{Toeplitz}(b) \)
17: for \( i = 1 \rightarrow p \) do
18: \( \gamma_i \leftarrow \gamma_i \times \|B \times H_{yi}\|_2 / \sqrt{\text{trace}(H \Phi \{i, i\} \times B)} \)
19: \( \Sigma_0 \{i\} \leftarrow B \times \gamma_i \)
20: end for
21: end while

\( p \) number of blocks
\( k \) block size \( (k = N/p) \)
\( \Phi \) \( \mathbb{R}^M \times \mathbb{R}^N \) sensing matrix
\( \Phi_i \) \( \mathbb{R}^M \times \mathbb{R}^k \) sub-matrix of \( \Phi \) with \( i^{th} k \)-columns
\( \Sigma_0 \{i\} \) \( \mathbb{R}^k \times \mathbb{R}^k, i = 1 \ldots p \), initialized as \( I_k \)
\( d0 \) diagonal of \( B_0 \) calculated in Line 11
\( d1 \) superdiagonal of \( B_0 \) calculated in Line 11
\( H \Phi \{i, i\} \) \( \mathbb{R}^k \times \mathbb{R}^k \) sub-matrix of \( H \Phi \) with \( i^{th} k \)-columns and \( i^{th} k \)-rows

Fig. 4.1: Pseudo-code of BSBL-BO Algorithm
\[ PBP_i(l, m) = \left[ b^0 \times \{ \Phi_i(l, 1)\Phi_i(m, 1) + \Phi_i(l, 2)\Phi_i(m, 2) \right. \\
+ \cdots + \Phi_i(l, k)\Phi_i(m, k) \} \\
+ b^1 \times \{ \Phi_i(l, 1)\Phi_i(m, 2) + \Phi_i(l, 2)\Phi_i(m, 3) \right. \\
+ \cdots + \Phi_i(l, k-1)\Phi_i(m, k) \} \\
+ b^2 \times \{ \Phi_i(l, 1)\Phi_i(m, 3) + \Phi_i(l, 2)\Phi_i(m, 4) \right. \\
+ \cdots + \Phi_i(l, k-2)\Phi_i(m, k) \} \\
+ \cdots \\
+ b^{k-1} \times \Phi_i(l, 1)\Phi_i(m, k) \] \times \gamma_i \quad (4.3)

Since \( b \) is always less than 1 (Line 13-15 in Fig. 4.1), \( b^n \) decreases exponentially when \( n \to k - 1 \). This allows us to approximate \( PBP \) with only several lower order terms of \( b \). Fig. 4.2 shows average SNR for 64 channels of EEG signals with maximum iteration of 10 by taking a certain number of terms into account, and it depicts that the SNR sharply drops only when we consider single term for calculation. When we approximate the calculation of \( PBP_i \) to first-order polynomial of \( b \), only a SNR drop of 1.60 dB was observed.

Furthermore, when we assume a fixed sensing matrix, the coefficients in the polynomial to calculate \( PBP \) are completely static. This implies we can calculate the coefficients off-line and \( PBP_i \) is only a function of scalar values \( b \) and \( \gamma_i \).
4.2.3 Matrix Multiplication (MM)

Line 3 and 5 in Fig. 4.1 are matrix multiplication with relatively large matrices. According to our analysis of software execution time of BSBL-BO algorithm (Fig. 4.3), these two computations are observed to consume more than half of the whole execution time.

Notice that $H\Phi$ is only used in Line 8 of Fig. 4.1 and $k \times k$ blocks along with the diagonal are necessary. Thus, it is possible to break up the large matrix multiplication into $p$ (number of blocks) small ones. The total number of scalar multiplication reduces from $MN^2$ to $MNk$ ($N \gg M \gg k$).

4.2.4 Computing $b$

Lines 8-12 of Fig. 4.1 describe the process of computing $b$ which is a very important parameter of this algorithm. Essentially, $b$ is the ratio of the average of superdiagonal of $B_0$ to the average of diagonal of $B_0$, and $B_0$ is $k \times k$ matrix which accumulates $C_x\{i\}$ over $\gamma_i$ for each block $i$. 

![Fig. 4.2: Approximation to compute PBP](image-url)
\[ B_0 = \sum_{i=1}^{p} \frac{C_x\{i\}}{\gamma} \]  

(4.4)

\[ = \sum_{i=1}^{p} \left[ \frac{1}{\gamma_i} \Sigma_x\{i\} + \frac{1}{\gamma_i} \left( \mu_i \times \mu_i^T \right) \right] \]  

(4.5)

\[ = \sum_{i=1}^{p} \left[ \frac{1}{\gamma_i} \Sigma_0\{i\} - \frac{1}{\gamma_i} \left( \Sigma_0\{i\} \times H\Phi\{i,i\} \times \Sigma_0\{i\} \right) \right. \]  

\[ + \frac{1}{\gamma_i} \left( \mu_i \times \mu_i^T \right) \]  

(4.6)

\[ = \sum_{i=1}^{p} \left[ B - \gamma_i \left( B \times H\Phi\{i,i\} \times B \right) \right. \]  

\[ + \gamma_i \left( (B \times H_yi) \times (B \times H_yi)^T \right) \]  

(4.7)

When substituting the definition of \( C_x\{i\} \) and \( \Sigma_x\{i\} \) to Eq. (4.4), we obtain Eq. (4.6). \( B_0 \) can be further simplified as Eq. (4.7) by considering the definition of \( \Sigma_0\{i\} \) in Line 19 of Fig. 4.1.

Note that we only need the average of two diagonals of \( B_0 \) to calculate \( b \) as in Line 12 of Fig. 4.1. Thus, we obtain Eq. (4.8) and Eq. (4.9) from Eq. (4.7).
$$\text{avg}(d_0) = \frac{1}{k} \sum_{i=1}^{p} \left[ \text{tr}_0(B) - \gamma \left\{ \text{tr}_0\left( B \times H\Phi\{i,i\} \times B \right) \right\} \right. $$

$$+ \left. \text{tr}_0\left( (B \times H_{yi}) \times (B \times H_{yi})^T \right) \right\} \right]$$

\hspace{1cm} (4.8)$$

$$\text{avg}(d_1) = \frac{1}{k} \sum_{i=1}^{p} \left[ \text{tr}_1(B) - \gamma \left\{ \text{tr}_1\left( B \times H\Phi\{i,i\} \times B \right) \right\} \right.$$ $$+ \left. \text{tr}_1\left( (B \times H_{yi}) \times (B \times H_{yi})^T \right) \right\} \right]$$

\hspace{1cm} (4.9)$$

Here $\text{tr}_q(A)$ is defined as $\sum_{i=1}^{n-q} a_{(i)(i+q)}$. Furthermore, $B$ is Toeplitz matrix defined by scalar $b$ so it can more be simplified as

$$\text{avg}(d_0) = \frac{1}{k} \sum_{i=1}^{p} \left[ k - \gamma \left\{ \text{tr}_0\left( B \times H\Phi\{i,i\} \times B \right) \right\} \right.$$ $$- \left. \text{tr}_0\left( (B \times H_{yi}) \times (B \times H_{yi})^T \right) \right\} \right]$$

\hspace{1cm} (4.10)$$

$$= p - \frac{1}{k} \sum_{i=1}^{p} \left[ \frac{\gamma}{k} (\Theta_{i0} - \Omega_{i0}) \right]$$

\hspace{1cm} (4.11)$$

$$\text{avg}(d_1) = \frac{1}{k} \sum_{i=1}^{p} \left[ (k - 1) b - \gamma \left\{ \text{tr}_1\left( B \times H\Phi\{i,i\} \times B \right) \right\} \right.$$ $$- \left. \text{tr}_1\left( (B \times H_{yi}) \times (B \times H_{yi})^T \right) \right\} \right]$$

\hspace{1cm} (4.12)$$

$$= \frac{k - 1}{k} pb - \frac{1}{k} \sum_{i=1}^{p} \left[ \frac{\gamma}{k} (\Theta_{i1} - \Omega_{i1}) \right]$$

\hspace{1cm} (4.13)$$

$$\approx pb - \sum_{i=1}^{p} \left[ \frac{\gamma}{k} (\Theta_{i1} - \Omega_{i1}) \right]$$

\hspace{1cm} (4.14)$$

where
\[ \Theta_{iq} := \text{tr}_q \left( B \times H \Phi \{ i, i \} \times B \right) \]  \hspace{1cm} (4.15)

\[ \Omega_{iq} := \text{tr}_q \left( (B \times H_{yi}) \times (B \times H_{yi})^T \right) \]  \hspace{1cm} (4.16)

\( b \) can easily be calculated once we obtain \( \Theta_{q0}, \Theta_{q1}, \Omega_{q0} \) and \( \Omega_{q1} \), and further approximation is identified to reduce the amount of computation for them. The following describes the approximation and optimization:

4.2.4.1 \( \Theta_{iq} \)

Since \( B \) is a structural Toeplitz matrix, we identified that \((l, m)\) element of \( B \times H \Phi \{ i, i \} \times B \) had the following meaning when expressed in terms of \( b \),

\[(l, m) = H \Phi \{ i, i \}(l, m) \]

\[+ b \times \text{[sum of 1-hop neighbors of } H \Phi \{ i, i \}(l, m)\text{]}\]

\[+ b^2 \times \text{[sum of 2-hop neighbors of } H \Phi \{ i, i \}(l, m)\text{]}\]

\[+ b^3 \times \text{[sum of 3-hop neighbors of } H \Phi \{ i, i \}(l, m)\text{]}\]

\[+ \cdots\]

\[+ b^{k-1} \times \text{[sum of } (k - 1)\text{-hop neighbors of } H \Phi \{ i, i \}(l, m)\text{]} \]  \hspace{1cm} (4.17)
Here again, higher order terms of $b$ get closer to zero and we empirically evaluated the average SNR for 64 channels of 1-sec long EEG signals, as shown in Fig. 4.4. As we keep decreasing the number of terms from 32 to 1, average SNR does not get affected until $b$ reaches two terms. Interestingly, the SNR shows a slight increase when only two terms are used in the computation and drops very quickly when only one term is used. This shows that the approximation with two terms is not only good for resource and energy optimization, but it also increased the overall accuracy.

![Fig. 4.4: Approximation to compute $\Theta$](image)

### 4.2.4.2 $\Omega_{il}$

Line 7 in the pseudo code calculates $\mu_{v_i}$ by multiplying $\Sigma_0\{i\}$ and $H_{ji}$. Noting $\Sigma_0\{i\} = \gamma_i B$ where $\gamma_i$ is a scalar value for block $i$, calculation of $B \times H_{ji}$ in Eq. (4.16) must be shared. Toeplitz matrix $B$ gives another chance of approximation here. Defining $H_{ji} = [p_1 p_2 p_3 \cdots p_{k-1}]^T$, $B \times H_{ji}$ can also be rearranged in terms of $b$ as following:
\[
\begin{pmatrix}
1 & b & b^2 & b^3 & \cdots & b^{k-1} \\
b & 1 & b & b^2 & \cdots & b^{k-2} \\
b^2 & b & 1 & b & \cdots & b^{k-3} \\
b^3 & b^2 & b & 1 & \cdots & b^{k-4} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
b^{k-1} & b^{k-2} & b^{k-3} & b^{k-4} & \cdots & 1
\end{pmatrix}
\begin{pmatrix}
p_1 \\
p_2 \\
p_3 \\
p_4 \\
\vdots \\
p_{k-1}
\end{pmatrix}
= 
\begin{pmatrix}
p_1 + b p_2 + b^2 p_3 + b^3 p_4 + \cdots \\
p_2 + b(p_1 + p_3) + b^2 p_4 + \cdots \\
p_3 + b(p_2 + p_4) + b^2(p_1 + p_5) + \cdots \\
p_4 + b(p_3 + p_5) + b^2(p_2 + p_6) + \cdots \\
\vdots \\
p_{k-1} + b p_{k-2} + b^2 p_{k-3} + \cdots
\end{pmatrix}
\] (4.18)

Now the number of terms to be included in computation naturally gives an approximation because \(b\) is less than 1, again. The analysis results are shown in Fig 4.5.

When \(B \times H_{yi}\) is computed as a \(k \times 1\) vector, \(\left((B \times H_{yi}) \times (B \times H_{yi})^T\right)\) becomes a \(k \times k\) square matrix.

Let \(B \times H_{yi}\) be defined as \([a_1 a_2 \cdots a_k]^T\), then \(\left((B \times H_{yi}) \times (B \times H_{yi})^T\right)\) formulates the following square matrix.
\[
\begin{pmatrix}
    a_1^2 & a_1a_2 & \cdots & \cdots \\
    \cdots & a_2^2 & a_2a_3 & \cdots \\
    \cdots & \cdots & \ddots & \cdots \\
    \cdots & \cdots & \cdots & a_{k-1}a_k \\
    \cdots & \cdots & \cdots & a_k^2
  \end{pmatrix}
\]

Note that we only need two diagonal sums, \(\Omega_{i0}\) and \(\Omega_{i1}\), so they can be reformulated as following:

\[
\Omega_{i0} = a_1^2 + a_2^2 + \cdots + a_k^2 \quad (4.19)
\]

\[
\Omega_{i1} = a_1a_2 + a_2a_3 + a_3a_4 + \cdots + a_{k-1}a_k \quad (4.20)
\]

where

\[
B \times H_{ji} := [a_1a_2 \cdots a_k]^T
\]

Fig. 4.5: Approximation to compute \(\Omega\)
4.3 Micro Architecture of Accelerators

This section describes micro-architecture of three major accelerators; PBP, Matrix Multiplication, and Compute_b. These accelerators are in the form of streaming operators attachable to Vortex framework.

4.3.1 PBP

As we have seen in section 4.2.2, the output matrix $PBP$ is an accumulation across blocks, in which $\gamma$ and $b$ can be factored out. Other terms are completely dependent only on sensing matrix $\Phi$, which we can assume fixed empirically. Furthermore, we have analyzed that two lower-order terms provide enough approximation, as shown in Fig. 4.2.

Fig. 4.6: Micro-architecture of $PBP$ accelerator
Eq. (4.3) depicts that most of the computation can be done off-line except \( b \) and \( \gamma \), that change across iterations. Thus, instead of streaming sensing matrix \( \Phi \) and Toeplitz matrix \( B \) in, two coefficients per each location of output matrix \( PBP \) can now be streamed into the accelerator where \( b \) and \( \gamma \) are configured at runtime and fixed for the iteration. Fig. 4.6(a) shows a block diagram of the \( PBP \) accelerator with two streaming operator interfaces for computing 8 blocks concurrently. As Line 2 of the pseudo code computes \( PBP \) as an accumulation of matrix \( PBP_i \), the accelerator implements an adder-tree to accumulate them at the end of the computational pipeline. Fig. 4.6(b) describes the internal pipeline of \( pbp_{blk} \) where \( \gamma \) and \( b \) are static values configured at the beginning of each iteration and two coefficients are streamed through for each consecutive location of output matrix \( PBP_i \) in a raster-scan fashion.

### 4.3.2 Matrix Multiplication (MM)

Matrix multiplication accelerator is very general for any size matrices. Due to the limitation of resources, however, it divides the large multiplication into multiple small ones. For \( C = A \times B \) as an example, matrix \( B \) is divided into multiple banks, 32 columns each. Runtime configuration is required to notify the accelerator about the dimension of matrices prior to loading columns of matrix \( B \). Micro-architecture of the matrix multiplication accelerator is similar to systolic array architecture [22]. Once the first 32 columns of matrix \( B \) are loaded into column FIFOs simultaneously through 4 network-interfaces as shown in Fig. 4.7, 4 rows of matrix \( A \) are simultaneously fed into the accelerator through the same interfaces. Since the accelerator already knows the size of matrices, it automatically switches its mode from loading-column mode to streaming-row mode. As soon as the 4 rows are streamed through the pipeline, multiplication results \((4 \times 32)\) stream out through the 4 network-interfaces. When the number of rows of matrix
$A$ is larger than 4, the columns must be re-used for next 4 rows so the column values are fed back to the column FIFO to eliminate redundant loadings. Thus, next 4 rows of matrix $A$ can follow immediately to the accelerator to compute next $4 \times 32$ multiplication results. Since the accelerator is built with 128 multiply-accumulate (MAC) instances, it performs 128 multiplications concurrently at each cycle. Once all the rows of matrix $A$ are streamed through the accelerator, it automatically switches its mode to another column-loading mode to load the next 32 columns of matrix $B$, if necessary.

![Diagram of a matrix multiplication accelerator](image)

**Fig. 4.7: Micro-architecture of matrix multiplication accelerator**

### 4.3.3 Compute b

Compute b accelerator outputs $(\Theta_{i0} - \Omega_{i0})$ and $(\Theta_{i1} - \Omega_{i1})$ for each block $i$ by taking charge of most computation to compute $avg(d0)$ and $avg(d1)$ in Eq. (4.11) and Eq. (4.14). The most important parameter being updated through iterations across the algorithm is the scalar value $b$ which is calculated by dividing $avg(d0)$ by $avg(d1)$.
In order to start computation, $\gamma_i$ and old $b$ must be configured prior to streaming in $H\Phi_i$ and $H_{ij}$. As shown in Fig. 4.8, the accelerator consists of three major streaming operators: Compute_Θ, Compute_μ, and Compute_Ω.

### 4.3.3.1 Compute_Θ

Because the approximation in section 4.2.4.1 shows that the best accuracy was achieved when only two terms of $b$ are used in Eq. (4.17), each element of $B \times H\Phi\{i,i\} \times B, (l,m)$, can be calculated by adding $H\Phi\{i,i\}(l,m)$ to $b\times$ [sum of 1-hop neighbors of $H\Phi\{i,i\}(l,m)$]. Furthermore, we only need two diagonal sums since only $\text{avg}(d0)$ and $\text{avg}(d1)$ affect the computation of $b$.

Fig. 4.9 illustrates those elements of $H\Phi\{i,i\}$ which involve in the computation of $\Theta_{i0}$. Red circles depict what we need to accumulate, and blue circles depict what we need to subtract after doubling it, since those overlaps are 1-hop neighbor of two consecutive red-circle elements. Considering similar computation of $\Theta_{i1}$, both $\Theta_{i0}$ and $\Theta_{i1}$ can be computed with 4 diagonal sums; $d_{-1}, d_0, d_1$, and $d_2$ as expressed in Eq. (4.21) and Eq. (4.22).
\[ \Theta_{i0} = d_0 + 2b(d_1 + d_{-1}) \]  (4.21)  \\
\[ \Theta_{i1} = d_1 + 2b(d_0 + d_2) \]  (4.22)

The micro-architecture of Compute_\( \Theta \) is shown in Fig. 4.10. As each element of \( H\Phi_i \) comes into the module, 4 diagonal-sum sub-modules accumulate corresponding elements, and \( \Theta_{i0} \) and \( \Theta_{i1} \) are computed once the entire matrix \( H\Phi_i \) streams into the module. Internal FSM (not shown in the figure) determines the timings and controls the multiplexers accordingly.

### 4.3.3.2 Compute_\( \mu \)

The Compute_\( \mu \) computes \( \mu_{xi} \) and \( B \times H_{yi} \). The reconstructed signal \( \mu_i \) as a concatenation of \( \mu_{xi} \) is the final output of the whole algorithm. At the same time, \( B \times H_{yi} \) as an intermediate vector variable to reach \( \mu_{xi} \) is required and exposed to Compute_\( \Omega \) module to be used in computation of \( \Omega_{i0} \) and \( \Omega_{i1} \). Similar to other modules, \( b \) and \( \text{gamma}_i \) are configured at run-time. Fig. 4.11 illustrates the internal pipeline of the module and it is essentially a 3-tap FIR filter.
which incorporates only two lower-order terms in Eq. (4.18). The multiplexer in Fig. 4.11 inserts 0 before and after valid $H_{yi}$ vector gets inserted in order to deal with boundary issues.

### 4.3.3.3 Compute $\Omega$

Eq. (4.19) and Eq. (4.20) lead us to design the Compute $\Omega$ module as shown in Fig. 4.12. $\Omega_{i0}$ is the sum of squares of all elements in $B \times H_{yi}$ and $\Omega_{i1}$ is the sum of multiplication of every neighbor elements.

### 4.4 Experimental results

The hardware acceleration of BSBL algorithm was evaluated on multi-FPGA Dinigroup board [18]. The board is equipped with six Virtex-6(V6SX475T) FPGAs and connected to a host PC through PCIe. The host PC runs C# implementation of BSBL algorithm with Emgu CV [23], which is a cross platform .Net wrapper to the OpenCV image processing library [6]. Software implementation running on the host PC with dual-core Intel i3-2100 has been evaluated and compared to the aid of hardware acceleration.
4.4.1 The complete hardware-acceleration system

The three major accelerators are distributed to three FPGAs on the board. As the resource consumption reveals in Table 4.1, the distribution across multiple FPGAs was not caused by resource limitation but to facilitate independent design and verification of each accelerator. On-chip communication is based on Vortex infrastructure so all accelerators are designed in the form of streaming operator (SOP) in its definition. Fig. 4.13 describes the system composition of three FPGAs on the board, where the number on every router port depicts the port-number. Every node on the network is referred to its own network address which consists of router ID and port number. Each router is tagged with its router ID in the figure as well. The host PC is interacting
with the FPGA board through PCIe at 3.2 GB/s of transfer rate. External DRAMs (4GB) are connected through MPMC (Multi-port Memory Controller) to both FPGA1 and FPGA3 to store input/output to/from accelerators.

![Fig. 4.13: System composition on three FPGAs](image)

As shown in Fig. 4.13, numerical representation converting modules are also attached on the network to convert fixed-point numbers to single-precision floating-point numbers, and vice versa. When the host PC transfers input matrices in single-precision floating-point format through PCIe, the data-stream immediately visits FLT2FIX (floating-to-fixed conversion) module before reaching at designated DRAM. This enables not only accelerators to see its natural numerical representation as input when in operation, but also hides the converting latency behind the communication latency. It only adds initial latency as much as the converting module consumes, and does not require additional transactions to convert the data in floating-point representation by reading, converting, and writing back to the memory in a different representation. In addition, efficient memory usage is achieved by avoiding redundant memory allocation for both floating-point and fixed-point representations. Similarly, when reading results by accelerators back to the host PC, the result data-stream travels through FIX2FLT (fixed-to-floating conversion) module.
to enable the PC to use the data without any additional conversion but only with simple pointer
type-casting.

Table 4.1: Resource consumption on three FPGAs (V6SX475T)

<table>
<thead>
<tr>
<th>FPGA</th>
<th>Register</th>
<th>LUT</th>
<th>BRAM</th>
<th>DSP48E1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>92,053 (15%)</td>
<td>86,192 (28%)</td>
<td>182 (17%)</td>
<td>48 (2%)</td>
</tr>
<tr>
<td>2</td>
<td>70,765 (11%)</td>
<td>66,683 (22%)</td>
<td>99 (9%)</td>
<td>16 (1%)</td>
</tr>
<tr>
<td>3</td>
<td>127,165 (21%)</td>
<td>120,530 (40%)</td>
<td>339 (31%)</td>
<td>528 (26%)</td>
</tr>
</tbody>
</table>

Table 4.1 shows resource consumption of each FPGA including NoC routers, network interfaces, inter-FPGA communication channels, memory controllers, PCIe adapter, etc. FPGA1 and FPGA3 consume relatively more resources because they need more logic to incorporate more peripherals like DRAM and PCIe.

4.4.2 Accuracy, power, and performance for EEG application

The acceleration system on the FPGAs was evaluated with part of a public EEG database that contains measurements from 64 electrodes placed on the scalp sampled at 256 Hz for 1 second. Table 4.2 summarizes measurements and estimations of accuracy, power, and performance of two platforms; software execution, hardware acceleration of our study on FPGAs. All the measurements are collected and averaged through 64 channels of 1-sec EEG signals. Execution time is given as an average per iteration, and Fig. 4.14(a) shows that the SNR converges around 10 iterations for the BSBL-BO algorithm, which our system accelerates. It also compares the accuracy across multiple sparse recovery algorithms for a particular channel of the EEG database and clearly reveals that BSBL-BO algorithm outperforms others. The reconstructed signal (blue) from our hardware acceleration is given in Fig. 4.14(b) with the original signal (red) for the particular channel. The reconstruction is obtained after 10 iterations so the software implementation
takes 200 ms for a single channel of 1-sec long EEG signal. On the other hand, our acceleration on FPGA reduces the execution time of 10 iterations to \( \sim 36 \) ms (speed-up 5.6).

In real-time applications of EEG, the software implementation can only keep up with 5 channels, but our acceleration on FPGA enables to support up to 27 channels simultaneously.

![SNR comparison over iterations for various algorithms on EEG](image1)

![Original and reconstructed EEG signal by accelerated BSBL-BO](image2)

Fig. 4.14: Accuracy of hardware acceleration for one channel of EEG signal

The accuracy in Table 4.2 is measured as an average SNR for 64 channels of 1-sec EEG signals. The approximation adopted in the hardware acceleration did not harm the overall accuracy but rather enhanced it by a statistically negligible amount. This is mainly due to the \( \Theta \) approximation, mentioned in section 4.2.4.1, which exhibited better accuracy when approximation with only two terms were applied. The amount of energy consumed when reconstructing one channel of 1-sec long EEG signal was measured as \( \sim 13 \) J for pure software execution on dual-core Intel i3-2100 processor, but the hardware acceleration on FPGA only consumed \( \sim 1 \) J of energy. Notice that ASIC implementation of this system can further remove redundant blocks, like inter-FPGA communication, that are only required on multi-FPGA board, so we anticipate even better speed-up and energy consumption beyond the simple scaling of FPGA results.
Table 4.2: Evaluation of hardware acceleration system for 64-channel EEG signals

<table>
<thead>
<tr>
<th></th>
<th>Software</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time</td>
<td>20.33 ms/iter</td>
<td>3.62 ms/iter</td>
</tr>
<tr>
<td>Accuracy</td>
<td>47.71 dB</td>
<td>47.74 dB</td>
</tr>
<tr>
<td>Power</td>
<td>TDP 65 W</td>
<td>26.1 W</td>
</tr>
<tr>
<td>Energy</td>
<td>1.32 J/iter</td>
<td>94 mJ/iter</td>
</tr>
</tbody>
</table>

4.5 Conclusion

Compressed sensing on wireless body-area networks (WBANs) is expected to reduce sensing and wireless transmission energy limited by batteries in portable health-monitoring devices. Among recently introduced algorithms of signal reconstruction, also known as sparse recovery algorithms, Block Sparse Bayesian Learning (BSBL) [59] has shown superior performance compared to others. However, the amount of computation involved in its iterative processes demands hardware acceleration to be applied in mobile devices.

This work introduces the analytical optimization and approximation of the algorithm, tailored at hardware acceleration. To the best of our knowledge, this is the first hardware acceleration system for BSBL algorithm. Novel architecture of three major accelerators and the system composition using Vortex framework were proposed, and their evaluation on FPGAs revealed promising performance speed-up and energy reduction. FPGA implementation showed the speed-up of 5.6x using only 7.7% of energy without any accuracy loss, compared to software implementation for multi-channel EEG signals. Since such a sparse recovery can be applied to any kind of natural signals, our acceleration system is expected to improve the performance and the battery life of portable health-monitoring devices which capture body signals on multiple modalities, e.g. EEG, ECG and respiratory sounds.
In this work, accessing a sub-matrix of a large matrix was very efficiently handled due to the 2D DMA, built in the network interface of SAP. Moreover, BSBL algorithm is essentially a different style of computation from that of vision applications. While vision application is characterized as stream processing of consecutive image frames, BSBL can be categorized into a control-oriented and iterative programming model with massive matrix computations. However, Vortex framework in this work demonstrated that it is not only tailored at stream processing model but also beneficial to others as well.
In chapter 2, we have proposed a framework and a methodology to integrate heterogeneous accelerators for visual perception, especially tailored at neuromorphic vision applications. It has been identified that the interconnection network for visual perception requires flexibility, scalability, high bandwidth, programmability, and composability. In order to meet these requirements, the framework is designed to support frame-level granularity and application-level abstraction. Moreover, system-level automation tool, Cerebrum, demonstrated that the framework is suitable for developing automation tools on top. We also demonstrated how the Vortex framework fits very well to mapping dataflow graphs onto networks of domain specific custom accelerators with significant improvements in performance and power consumption compared to multi-core CPU and GPU implementations.

More in-depth energy optimization is discussed in chapter 3 for multi-object recognition system. Typically, neuromorphic approaches find multiple regions of interest by saliency model to pass down to extract features. These features are used to classify the object among pre-learned categories of target objects. However, these two stages, visual saliency and feature extraction, commonly work separately and the latter processing usually consider multiple regions in an image equally. Since there exists a ranking, or priority, suggested by the former processing, taking the rankings into account leads us to achieve better energy-efficiency by spending limited energy-budget more effectively.
The experimental results demonstrated to achieved up to 70.57% of energy saving for accuracy-critical applications compared to the case of treating all 5 regions at the best effort while only 1.05% of expected accuracy was lost. Similarly, for energy-critical applications, we have proposed a dynamic configuration which saves 91.72% of energy consumption while 5.07% of expected accuracy drop happened.

More importantly in such a system design with dynamic configuration, the configuration scheme itself plays a very important role. If the runtime configuration takes too much time or energy by itself, then we may lose more than what we gain, or real-time requirements may not be met. Since the Vortex framework is designed to pre-configure enough number of possible cases, it turned out to suffer no extra energy burden to retrieve them at runtime.

Finally in chapter 4, we have adopted our infrastructure to the domain of wireless body-area networks (WBANs) and designed custom accelerators for BSBL algorithm to construct a hardware acceleration system. During the work, we introduced analytical optimization and approximation of the algorithm, tailored at hardware acceleration. To the best of our knowledge, it is the first hardware acceleration system for BSBL algorithm. Novel architecture of three major accelerators and the system composition on Vortex framework was proposed, and its evaluation on FPGAs revealed promising performance speed-up and energy reduction. FPGA implementation showed the speed-up of 5.6x with only 7.7% of energy without any accuracy loss, compared to software implementation for multi-channel EEG signals. Since such a sparse recovery can be applied to any kind of natural signals, our acceleration system is expected to improve the performance and the battery life of portable health-monitoring devices which capture body signals on multiple modalities, e.g. EEG, ECG and respiratory sounds.
Vortex framework has shown its applicability for perceptual computing in both vision and WBAN applications. It supplies enough degree of programmability and flexibility to SoC integration for ASIC implementation. However, I foresee a significant potential in conjunction with field programmable gate array (FPGA) which started to support dynamic partial reconfiguration (DPR). DPR substantially carries out an equivalent impact as expanding the chip size to infinity with no increase of manufacturing cost and energy consumption because a fraction of the chip can change anytime even during run-time. Without DPR, the only way to achieve it is to statically place those potential hardware blocks on the same chip, resulting in proportionally increased chip size.

However, DPR still requires interfaces among partial regions and static regions to be static, obviously. It raises critical issues in most of existing integration strategies due to the lack of complete abstraction and modularity. In most cases, heterogeneous accelerators in a system become coupled at a certain level, e.g. prior-knowledge of network address of each other. Such an unintended coupling at system composition is not impossible to get rid of with careful design manner, but framework-level support may handle it more ideally and safely. Vortex is designed toward such a goal with virtual flow notion to reinforce the abstraction of underlying physical structure against reconfigurable high-level management or control. Exploring the benefit of such a framework with DPR of FPGA may be a very interesting and exciting future work. Though ASIC typically gives us better energy efficiency and performance compared to the same implementation on FPGA, it can never change its own circuits, unfortunately.
Bibliography


[57] Xilinx. Virtex-5 family overview. *DS100*(v.5.0), 2009.


Vita
Sungho Park

Education

Ph.D. in Computer Science and Engineering
Advised by Dr. Vijay Narayanan

B.S. in Electrical Engineering

Career Summary

Internship at Intel Corp.
Identifying/resolving technical challenges in process migration toward advanced technology

DARPA Project Experience in RTL design on FPGAs for 3 years
Designing streaming accelerators and communication infrastructure with Network-on-Chip for neuromorphic vision

Work Experience in RTL design on FPGAs for more than a year
Designing deblocking filter and reconstruction module for H.264 Encoder/Decoder, and validating on FPGA

Work Experience as a firmware engineer for more than 3 years
Developing firmware on micro-controllers in mobile devices

Selected Publications


