ACCELERATING DESIGN AND IMPLEMENTATION OF
EMBEDDED VISION SYSTEMS

A Dissertation in
Computer Science and Engineering

by

Ahmed Al Maashri

© 2012 Ahmed Al Maashri

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2012
The dissertation of Ahmed Al Maashri was reviewed and approved* by the following:

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

Yuan Xie  
Professor of Computer Science and Engineering

Robert T. Collins  
Associate Professor of Computer Science and Engineering

Richard A. Behr  
Director, Smart Spaces Center for Adaptive Aging in Place  
Charles and Elinor Matts Professor of Architectural Engineering

Lee D. Coraor  
Associate Professor of Computer Science and Engineering  
Director of Academic Affairs for the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School
ABSTRACT

In today’s world, embedded systems have become a necessity in our daily lives. Ranging from digital watches to factory controllers, these systems are dedicated to handle a particular task both efficiently and reliably. The advancements in computer vision and machine learning, in conjunction with the abundance of computational power, made it possible for these systems to perform other tasks such as image processing and video analytics; giving rise to embedded vision systems.

Being the heart of embedded vision systems, computer vision is the key technology enabler for image processing and video analytics. However, the computational modalities that are prominent in computer vision algorithms are usually inconsistent with those modalities most exploited in contemporary computer architectures. As a result, execution inefficiencies are observed when algorithms are processed by such architectures.

Furthermore, recent studies have shown that biologically-inspired (neuromorphic) vision algorithms can be robust alternatives, due to their detection and recognition capabilities. Interestingly, these neuromorphic algorithms experience similar execution inefficiencies when running on general purpose processors.

On the other hand, domain-specific computing is believed to be the solution to the challenges presented above. Developing customized hardware accelerators targeting specific workloads, is the key to achieving the desired performance while operating at a lower power budget. However, domain-specific computing may be an unfavorable route to many, due to a whole host of challenges associated with hardware design and implementation.

This dissertation addresses the issues presented above by describing a hardware-software framework that offers a flexible, reliable, and high performance accelerator infrastructure. The underlying hardware complexity of the framework is abstracted from the user through a
standardized Application Programming Interface, API. Furthermore, the dissertation presents a software automation tool for expediting the process of building embedded vision systems and mapping them to prototyping platforms.

In addition, this dissertation discusses the hardware architecture of several vision accelerators, including neuromorphic accelerators, which are mapped to the accelerator framework described above. In particular, this dissertation presents accelerators for a neuromorphic vision algorithm, HMAX. This algorithm can be used as a feature extractor for multiple recognition tasks. Results reveal that the neuromorphic accelerators can deliver as much as 7.6× speedup, and are up to 12.8× more power efficient when compared to contemporary CPU and GPU platforms. Additionally, results indicate that the neuromorphic accelerators can achieve up to 90% classification accuracy on some of the standard datasets. Furthermore, this dissertation discusses the hardware architecture of other non-neuromorphic embedded vision systems, where results show that these accelerators can speed up the execution time of some of the computer vision algorithms by up to 100× when compared to a CPU platform.
# TABLE OF CONTENTS

List of Figures .................................................................................................................. vii  
List of Tables .................................................................................................................... ix  
Acknowledgements ............................................................................................................ x  
Chapter 1 Introduction ....................................................................................................... 1  
Chapter 2 A Hardware-Software Framework for Embedded Vision Accelerators .......... 5  

  Framework’s Computational and Communicational Requirements ............................. 5  
  Accelerator Framework Components .............................................................................. 7  
    The Stream Operator (SOP) ......................................................................................... 9  
    The Switch Attached Processor (SAP) ......................................................................... 10  

Chapter 3 Cerebrum: A Software Tool for Embedded Vision System Prototype  

  Composition and Automation ........................................................................................ 14  

  Background .................................................................................................................... 15  
  Cerebrum Features and Software Architecture ............................................................. 16  
    Cerebrum Frontend (GUI) .......................................................................................... 16  
    Cerebrum Backend (EDA) ........................................................................................ 18  

Chapter 4 Accelerating Neuromorphic Vision Algorithms ............................................. 22  

  Neuromorphic Vision Algorithms .................................................................................. 22  
  The HMAX Computational Model ................................................................................ 24  
    Model Description ....................................................................................................... 24  
    The Computational Complexity of HMAX .................................................................... 26  
    Extensions of the HMAX Model .................................................................................. 30  
  The Architecture of the HMAX Accelerators ............................................................... 34  
    S1 Accelerator ........................................................................................................... 34  
    C1 Accelerator ........................................................................................................... 36  
    S2/C2 Accelerator ...................................................................................................... 36  
  Experimental Evaluation ............................................................................................... 39  
    Nallatech Development System .................................................................................. 39  
    The Neuromorphic Acceleration System ..................................................................... 41  
    CPU and GPU Platforms ........................................................................................... 43  
    Datasets Used for Evaluation ..................................................................................... 43  
    Classification Accuracy .............................................................................................. 45  
    Speed ........................................................................................................................... 48  
    Power Efficiency ........................................................................................................ 49  
    Configurability of Accelerators and Design Tradeoffs ................................................ 51  
    Discussion of Results .................................................................................................. 52  
    Resource Utilization .................................................................................................... 54  
  Hardware- vs. Software-controlled SAP-PE ................................................................. 55
Chapter 5 Hardware Accelerators for Embedded Vision Systems ..........................59

High-Performance Hardware Accelerators for Retail Environments..........................59
  Overview of PCV Automation System ...............................................................61
  Hardware Architecture of Accelerators............................................................65
  Experimental Evaluation.....................................................................................72

Scalable Hardware Architecture for Connected Component Labeling ......................77
  Background .........................................................................................................77
  Sliced Connected Component Labeling Algorithm .................................................79
  SCCL Hardware Architecture ............................................................................81
  Experimental Evaluation.....................................................................................87

Chapter 6 Conclusions and Future Work..................................................................90

Bibliography ...........................................................................................................93
LIST OF FIGURES

Figure 2-1. Accelerator system topologies and configurations ........................................... 8
Figure 2-2. NIF-SOP Architecture ................................................................................. 9
Figure 2-3. NIF-SAP Architecture .............................................................................. 10
Figure 2-4. SAP-PE Architecture ............................................................................. 12
Figure 3-1. The frontend (GUI) and backend (EDA) of the Cerebrum tool ..................... 17
Figure 3-2. Cerebrum graphical user interface. ............................................................ 18
Figure 4-1. The computational layers of the HMAX model ......................................... 24
Figure 4-2. HMAX execution time using different configurations .................................. 27
Figure 4-3. Pseudo code of $S_2$ stage ........................................................................ 29
Figure 4-4. Speedup in $S_2$ execution time when increasing number of threads .......... 29
Figure 4-5. The increase in power consumption as the number of threads increases ...... 30
Figure 4-6. A multi-purpose recognition system using HMAX features ....................... 33
Figure 4-7. $S_1$ accelerator ........................................................................................ 35
Figure 4-8. $C_1$ accelerator ....................................................................................... 35
Figure 4-9. $S_2/C_2$ accelerator .................................................................................. 38
Figure 4-10. Nallatech™ Development System ........................................................... 40
Figure 4-11. Data flow paths for the HMAX neuromorphic system ......................... 41
Figure 4-12. Representative samples from the datasets ............................................. 46
Figure 4-13. Classification accuracy for the Caltech256 dataset ............................... 47
Figure 4-14. Classification accuracy for the PASCAL VOC2007 dataset .................... 48
Figure 4-15. Task-level parallelism as exploited by the accelerator system ............... 49
Figure 4-16. Speedup across the three platforms ....................................................... 49
Figure 4-17. Improvement in power efficiency (fps-per-watt) across the three platforms .... 51
Figure 4-18. The influence of number of scales on classification accuracy. .......................... 52
Figure 4-19. The influence of number of scales on performance. ........................................ 52
Figure 4-20. Pseudo code of the SAP-PE Egress (a) and SAP-PE Ingress (b). ....................... 56
Figure 5-1. Conceptual illustration of PCV automation system. ........................................ 61
Figure 5-2. Automation system for PCV. ........................................................................... 62
Figure 5-3. Representative samples from the Products Database. ........................................ 62
Figure 5-4. Pipeline stages of the SURF and Vector Matcher accelerators. ....................... 66
Figure 5-5. Two intuitive approaches to implement the Integral Image cache................. 67
Figure 5-6. The Fast Hessian Accelerator ........................................................................... 68
Figure 5-7. SCCL algorithm example with 2 slices............................................................ 80
Figure 5-8. SCCL Architecture.......................................................................................... 81
Figure 5-9. Operations performed by the Slice Processor unit ........................................ 83
Figure 5-10. The architecture of the Coalescing Unit....................................................... 84
Figure 5-11. SCCL execution time as a function of image size........................................ 87
Figure 5-12. SCCL speedup............................................................................................. 88
LIST OF TABLES

Table 4-1. Execution time for each layer in the HMAX model, in percentage, out of the total execution time ..........................................................................................................................28

Table 4-2. A description of the parameters of the spatiotemporal Gabor filter as defined in Equations (3) – (6): .................................................................................................33

Table 4-3. The interactions between CMP and accelerators ...........................................42

Table 4-4. Datasets used for evaluation: Accelerated HMAX is used to extract features from the datasets, which is used as an input to a regularized least-square (RLS) classifier ..................................................................................................................44

Table 4-5. The resource utilization of the neuromorphic accelerators. Accelerators are mapped to four Virtex-5 SX-240T FPGAs. The numbers in brackets show the utilization in percentage .................................................................................................55

Table 4-6. Execution time of hardware- vs. software-controlled SAP-PE. ......................57

Table 4-7. Resource utilization comparison between software- and hardware-controlled SAP-PE .......................................................................................................................................58

Table 5-1. A comparison of execution time (ms) between software and proposed SURF and Vector Matcher accelerators (Average keypoints per model is 531, and 5415 keypoints per input frame) ..........................................................................................................................74

Table 5-2. Resource utilization, in percentage, for the SURF and Vector Matcher accelerators ........................................................................................................................................75

Table 5-3. Determining labels in single pass CCL ................................................................78

Table 5-4. Comparing SCCL with other approaches .........................................................89

Table 5-5. Resource utilization of the SCCL hardware according to number of slices .......89
ACKNOWLEDGEMENTS

I would like to start by thanking my dissertation advisor, Prof. Vijay Narayanan. It was a great privilege to be your student. I learnt a lot from you, and I really appreciate your guidance and support throughout those five years.

To Prof. Yuan Xie, Prof. Robert Collins, and Prof. Richard Behr; it was a great honor to have you in my research committee. Your invaluable suggestions and comments are highly appreciated.

To Prof. Chaitali Chakrabarti: although we never met face to face, but I have learned so much from you. Thank you!

To Dr. Kevin Irick and Dr. Michael DeBole: I appreciate everything you taught me. I am really thankful for everything you did for me from the first day I joined the MDL lab. You are great mentors and friends.

To the “boys and girls” in the ReALM group, Sungho, Matt, Yang, Nandhini, Srinidhi, Prasanth, Siddarth, Vikram, Mi Sun, Chris, Anusha, and Karthik: Thanks for making the lab a fun place to work in!

To Cindy, Beth, Karen, Marcella, Macy, Kathy, Lena, and Annie: thank you for your help in the paperwork. Cindy and Beth, I will miss those chats a lot!

To the best IT support; Tony, Barb, Mark, Kem, Eric, and Christen: thanks for always being there to help!

Also, I would like to express my sincere gratitude to the Government of Oman and Sultan Qaboos University for the scholarship and for the support throughout the past five years.
To my sisters: no words can express my gratitude for your endless support and encouragement.

To my mother: thank you for giving me life, love, and for always being there for me. I love you and I hope I make you proud.

To my kids, Iman, Leen, and Saeed: You are my inspiration! I love you so much!

To Rana, my wife: You are the love of my life; without your support, dedication, and encouragement I wouldn’t have made it.

Finally, I would like to dedicate this work to the loving memory of my father, “Saeed Al Maashri”. I hope you are proud of me.
Chapter 1

Introduction

Embedded vision systems employ image processing and video analytics to process captured scenes and video sequences. These systems can be utilized in a variety of application domains including autonomous cars, cell tracking, space exploration, and surveillance and security [1,2,3,4]. In addition, these systems can be the key building blocks to ubiquitous computing [5], and smart environments [6]. These systems require real-time performance, while operating at low power budget.

Current approaches in developing embedded vision systems rely on traditional computer vision algorithms to perform tasks such as features extraction and tracking. However, recent breakthroughs in understanding the visual path in the primate’s visual cortex have inspired a number of computer vision algorithms that are collectively referred to as “biologically-inspired vision algorithms” or “neuromorphic vision algorithms”. In particular, some of these algorithms were developed by reverse-engineering the human’s brain to produce vision algorithms that are both efficient and robust. As a step towards exploring how the brain efficiently processes visual information, a brain-inspired feedforward hierarchical model, HMAX, [7] has become a widely accepted abstract representation of the visual cortex. HMAX originates from the current understanding of the mammalian ventral visual stream, a hierarchy of brain areas responsible for carrying out object recognition in the mammalian vision system. HMAX mimics the feedforward path of object recognition in the visual cortex, accounting for the first 150 milliseconds of processing within the primate’s visual cortex [7,8,9].
One common attribute that is shared between traditional computer vision algorithms and neuromorphic vision algorithms is that their software implementation is usually inefficient when executed on general purpose processors. This results in relatively slow execution time, while operating at high power budgets. This may be attributed to the fact that the computational modalities that are prominent in these algorithms are inconsistent with those modalities most exploited in contemporary computer architectures.

On the other hand, domain-specific computing is proposed as an adequate solution to high-performance and low-power applications. Domain-specific accelerators are customized hardware architectures specialized in executing certain applications. These customized accelerators execute more efficiently while operating at lower frequencies—leading to a reduction in overall power consumption. Therefore, rather than relying on general purpose processors to execute specialized workloads – such as the ones described above – instead, a shift towards a heterogeneous integration of these cores within processor architectures will be required in order to attain the desired performance and reduce burgeoning power budgets [10,11,12,13].

However, hardware design and implementation is laborious and requires knowledge of Register Transfer Level, RTL, and certain skills in Hardware Description Language, HDL. In addition, the lack of standardized interfaces and standard means of inter-accelerator communication introduces additional complexities to hardware architecture design. Furthermore, mapping the design to digital circuits involves many obstacles such as resolving timing violations and routing failures. These challenges impede the advancements of customized hardware acceleration, and as a result may have an impact on the development of embedded vision systems.

To address the aforementioned challenges, this dissertation presents a hardware-software accelerator framework that provides the necessary infrastructure for inter-accelerator communication. The framework implements a software abstraction layer that hides the underlying
hardware details from the user. Accordingly, the users may configure and operate the accelerators via high-level Application Programming Interface, API.

In addition, this dissertation explores and discusses the hardware architecture of vision accelerators. First, we explore a neuromorphic vision algorithm, HMAX, as being one of the most widely accepted models of the primate’s visual cortex. The computational structure of this algorithm is investigated in depth; followed by a discussion of the neuromorphic accelerators that speed up the execution time of HMAX and improve the power efficiency of the neuromorphic vision system. These configurable accelerators are used in four application domains: (1) Object Recognition, (2) Face Identification, (3) Facial Expression Recognition, and (4) Human Action Recognition.

Furthermore, this dissertation investigates other vision algorithms that can be employed in many application domains. The algorithms that are considered for acceleration are as follows: (1) Speeded-Up Robust Features, SURF, [14] a feature extraction algorithm, (2) Brute-Force Vector Matcher, a feature matching algorithm, and (3) Connected Component Labeling [15], a process of identifying disjoint pixel regions in an image and assigning a unique label to each of these regions.

Field-Programmable Gate Arrays, FPGAs, are used as prototyping platforms to validate the proposed hardware architectures. Towards that end, a software automation tool is developed to help users in rapidly creating prototypes of embedded vision systems and mapping them to multi-FPGA systems. The tool employs a multi-FPGA partitioning and mapping algorithm to efficiently allocate the accelerators to the FPGAs.

In summary, this dissertation discusses a number of aspects that may contribute to future advancements in embedded vision systems and related application domains. In particular, this dissertation describes a hardware-software framework that offers a flexible, reliable, and high performance accelerator infrastructure. The underlying hardware complexity of the framework is
abstracted from the user through a standardized API. Furthermore, the dissertation presents a software automation tool for expediting the process of building embedded vision systems and mapping them to prototyping platforms. Furthermore, this dissertation discusses the hardware architecture of several vision accelerators. For instance, results indicate that the proposed neuromorphic accelerators can achieve up to 7.6× speedup while delivering up to 12.8× more power efficiency when compared to contemporary CPU and GPU platforms. Additionally, this dissertation discusses the hardware architecture of other non-neuromorphic embedded vision systems, where results show that these accelerators can speed up the execution time of some of the computer vision algorithms by up to 100× when compared to a CPU platform.

The rest of this dissertation is organized as follows; Chapter 2 describes a hardware-software accelerator framework that provides a configurable, flexible, and programmable accelerator-rich environment. Chapter 3 presents Cerebrum, a software automation tool for rapidly building prototypes of accelerated embedded vision systems. Chapter 4 explores the acceleration of the HMAX algorithm. Chapter 5 discusses the acceleration of other computer vision algorithms that may be considered as essential ingredients of vision systems. Finally, Chapter 6 concludes this dissertation and discusses few directions for future work.
Chapter 2

A Hardware-Software Framework for Embedded Vision Accelerators

In future embedded vision systems, CPUs and domain-specific accelerators will have to co-exist and interact to accomplish the tasks assigned to the system. In such an accelerator-rich heterogeneous system, many issues have to be addressed such as how will these accelerators communicate with one another? Should these accelerators be void of any sophisticated control logic, and focus only on data processing? Or, should these accelerators possess the necessary decision-making logic—making them more autonomous? Moreover, how would chip multiprocessors, CMPs, interact with these accelerators?

This chapter discusses a hardware-software framework that attempts to find solutions to the questions presented above. The chapter presents a high-bandwidth, configurable routing infrastructure that facilitates the inter-accelerator communication. In addition, the chapter discusses a proposed abstraction layer for accelerators—making them both autonomous in their decisions and programmable from the user viewpoint. For the sake of completeness, this dissertation describes in details all aspects of the accelerator framework. However, it should be noted that the main contribution of this dissertation is the design and implementation of the accelerators’ abstraction layer (i.e. SAP-Processing Element) as discussed later in the subsection titled “The Switch Attached Processor (SAP)”.

Framework’s Computational and Communicational Requirements

The operations carried out by the vision accelerators sets certain requirements when designing a framework that hosts such accelerators. These requirements are as follows:
High Performance: Accelerators are expected to operate in real-time, while processing large volumes of data. Supporting such performance requires high-bandwidth communication infrastructure. For example, feature extractors produce feature maps that can be large sets of data. In order for the system to meet the real-time requirement, the infrastructure must have enough aggregate bandwidth to transfer these large sets of data without introducing undesirable latency.

Communication: Accelerators need to communicate with each other, whether to share data or synchronize with one another. The acceleration framework must implement the necessary logic to support data transfer mechanisms (e.g. Direct Memory Access, DMA, transfers), as well as synchronization primitives (e.g. messaging, barriers, etc…)

Configurability: Some of the vision applications may require frequent change to the parameters that govern the operations of the embedded vision systems. For example, when operating at low battery, an application may tradeoff speed for low power consumption to save energy. This may trigger a subset of the accelerators to operate in less a computationally-complex mode (e.g. smaller input frame, lower frame rate, etc…). Therefore, the framework should be capable of being configured and re-configured in such a way that it can be re-purposed towards multiple variants of an application.

Programmability: Vision accelerators vary in the functions they perform, the numerical properties of the input and output they operate on, and the types of transactions they execute in order to read/write data from/to other accelerators. As systems scale in terms of the number and types of accelerators they host, managing this system becomes a challenging task to developers. To address this
issue, an additional layer of APIs needs to be developed that abstracts the underlying complexities of the hardware. This layer of abstraction simplifies the task of controlling these accelerators and allows developers to focus more on the application, and less on the hardware details.

The following section presents the accelerator framework that supports the aforementioned requirements.

**Accelerator Framework Components**

The accelerator framework is composed of three components: (1) Routing infrastructure, (2) Network interfaces, and (3) Processing nodes.

The routing infrastructure, referred to as Vortex, is responsible for the inter-accelerator communication. Vortex is a packet-switched, reconfigurable and highly programmable Network-on-Chip, NoC, platform for vision applications with considerations to the requirements addressed previously. Vortex uses Router Configuration Table, RCT, which determines the routing behavior of the router. The RCT table is run-time configurable, giving the user the flexibility to change traffic routes while the system is running.

The network interface, NIF, is an optimized hardware that abstracts the handshaking and signaling details with the Vortex router from the attached accelerators. 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 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.

The network interfaces use virtual circuits in order to control data flow between the SAPs and SOPs. Generally speaking, SOPs are used to process input in a streaming fashion. The SAPs, on the other hand, are explicitly targeted towards those computations requiring control of data flow and processing non-contiguously stored data. As shown in Figure 2-1, the virtual circuits allow data packets to be grouped based on application needs—allowing one type of application to use one set of accelerator modules, and another application to use another set, with the circuits being dynamically setup before use. The use of virtual circuits, denoted as a flow, in this instance provides several benefits, including the ability to time-duplex SOPs across several flows and removing the need to destroy the circuit between uses. Flows are configured and tracked through

![Diagram](image)

**Figure 2-1. Accelerator system topologies and configurations**
An example system with 5 accelerators attached to Vortex router. (a) Physical Topology: showing accelerators physically attached to the Vortex router. To support scalability, the system allows cascading chains of switches. (b) Virtual Circuit Topology: data flows from an I/O device attached to the source SAP, via SOPs, destined to a sink SAP. (c) Flow Table Configurations: In this configuration, three flows have been configured. Flows 1 & 2 time-share SOPs 1 & 2, flows 0 & 2 time-share SOP 3, flow 1 exclusively accesses SOP 4, and all flows eventually terminate at SAP 5.
a distributed routing scheme (through flow tables) to avoid a centralized table lookup and a potential performance bottleneck.

The Stream Operator (SOP)

The SOP attaches to the Vortex router through the network interface for a SOP, NIF-SOP, as shown in Figure 2-2. The NIF-SOP architecture is composed of three components: (1) A Depacketizer that depacketizes incoming packets, (2) A Packetizer that packetizes packets before being sent across the network, and (3) A flow table that decodes an opcode based on the incoming flow. Once depacketized, data are streamed to the custom SOP cores through the egress interface. The egress interface exposes the data and associated opcode through a simple asynchronous handshaking protocol. Once processed, data are directly forwarded to the ingress interface and re-packetized to match the underlying packet format. The custom SOP cores are typically pre-defined IP included in the hardware. Note that SOPs can also be chained using flows, providing dynamically configurable functionality through the composition of multiple SOPs.

![Diagram of NIF-SOP Architecture]

Figure 2-2. NIF-SOP Architecture
This example shows that the custom SOP supports vector-vector multiplication, addition, and subtraction. The opcode governs the type of operation performed on data.
The Switch Attached Processor (SAP)

Figure 2-3 illustrates the architecture of network interface for a SAP, NIF-SAP. The NIF-SAP implements three interfaces: (1) master interface, (2) slave interface, and (3) message interface. The master interface allows a SAP to initiate a transaction and provides the means to send 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. The NIF-SAP uses a pool of handlers to manage data transactions and message passing. As shown in Figure 2-3, the Window Table is used to keep track of 2D Region of Interest, ROI, data transfer, while the Flow Table is consulted to determine the next hop to forward the packet to.

The SAP is suitable for carrying out computations that are structurally iterative and operate on non-contiguous blocks of data. 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 state 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, this dissertation proposes augmenting the SAPs with an additional layer of abstraction. This abstraction layer 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 introduce additional layer 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 term SAP-Processing Element, SAP-PE, is used to refer to the SAP accelerator combined with the additional layer of abstraction described above.

The SAP-PE architecture, depicted in Figure 2-4, has been partitioned into two distinct sections: (1) a control path that provides instruction-based control over the movement of data and the custom accelerator, and (2) a data path—the implementation of the custom accelerator logic and functions. In the control path, the main driver of operation is the Light-Weight Processor, LWP, which provides several standard mechanisms for control such as branching, looping, and basic Arithmetic and Logic Unit, 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 scratchpad 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

![Diagram](image_url)

**Figure 2-4. SAP-PE Architecture**

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.
custom accelerator. These commands can be used to communicate specific instructions to the accelerator. For instance, the user may use a command to start computations, and another command to stop the computations.

- 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.

A slightly different approach to implement the control path is to replace the LWP and the instruction memory with a state machine to speed up the execution time of the control path. However, this approach is not recommended for the following reasons: (1) Our preliminary experiments indicate that the state machine is not significantly faster than the proposed approach, especially when execution time is dominated by operations executed in the SAP-PE data path, and (2) Using hardware logic to implement the instruction fetch and dispatch unit yields a rigid system that is not flexible to changes and modifications.

The SAP-PE data path, on the other hand, as illustrated in Figure 2-4 (bottom) is made up of the custom accelerator hardware and is directly controlled through specific instructions issued by the control path. Data are transferred to the custom logic, directly through the NIF-SAP slave interface, using the DMA transfer instructions described above.
Chapter 3

*Cerebrum*: A Software Tool for Embedded Vision System Prototype Composition and Automation

The previous chapter introduced a hardware-software framework for hosting accelerators for embedded vision systems. The functionality of these accelerators and the framework can be initially verified in simulation, giving confidence to the developer that these accelerators are ready to be validated on actual integrated circuits. Application-Specific Integrated Circuit, ASIC, is one option to validate the design. However, ASIC design and development cycle may require millions of US dollars for Non-Recurring Engineering, NRE, costs, and it may take a year or more before the design is actually fabricated on chip. A cheaper and a more rapid time-to-market alternative is to use a prototype Field-Programmable Gate Array, FPGA. This alternative is even more convenient to developers since there are commercial tools that provide FPGA to ASIC conversion with Zero-NRE costs [16].

Prototyping a design to FPGAs requires certain skills and expertise in Hardware Description Language, HDL and Register-Transfer Level, RTL, some knowledge of memory hierarchies, and using the proper tools to generate and download the architecture to FPGAs. This might be challenging to anyone who is interested in building an embedded system but lacks the knowledge and skills. Even those with the required expertise may find the process of system composition laborious, redundant, and error-prone.

This chapter presents a software tool that automates the process of composing embedded vision systems and mapping them to FPGA platforms. This tool is referred to as *Cerebrum.*
Background

HDL development has always been a laborious task and it required certain skills that are acquired over long periods of time, let alone mapping this HDL to ASICs or FPGAs. Therefore, a number of previous works and commercial tools were proposed in the last decade in an effort to assist HDL developers in their quest. In fact, Neely et al. [17] have discussed three categories of tools used for accelerating the FPGA design process. The first category of tools aims at reducing NRE costs due to IP core development. Examples of this category include Impulse C [18], and Catapult C [19]. Although these tools raise the level of abstraction from HDL, however, they are limited in scope, as they fail to elevate the level of abstraction beyond the individual core. The second category aims to provide system design methodology similar to ASICs (e.g. Xilinx Platform Studio, XPS [20]). These tools provide the designers with peripheral, bus, and application IP. However, the onus is on the designer to construct the system in an appropriate fashion. The third and last category aims to offer the user abstractions at the system level. Examples include Xilinx System Generator [21] and ShapeUp [17], where IP modules are encapsulated in a higher-level language and module parameters are provided as a means for performing operations such as static type checking. These black-box modules can then be composed either programmatically, or graphically. However, these tools do not attempt to provide standardized interfaces for IP components, nor address the issue of inter-IP communication.

Recently, tools have appeared that are a hybrid of the first two categories described above. Cong et al. [22] describe the use of AutoPilot [23], a C-to-FPGA synthesis solution that is coupled with the XPS platform design tool offered by Xilinx. The authors show that using the tool yields an 11 - 31% reduction in FPGA resource usage compared to hand-coded designs.
However, the authors do not discuss the ability of the tool to map components to multi-FPGA systems, MFS.

**Cerebrum Features and Software Architecture**

*Cerebrum* was developed to allow users with little or no knowledge of hardware and RTL development to compose accelerators for various vision algorithms with minimal effort. The tool standardizes MFS specifications and uses high-level metadata to deliver an application-level design experience to the user. *Cerebrum* includes a library of optimized IP cores, and a multi-FPGA mapping algorithm. Furthermore, the tool automates the process of synthesizing/compiling the composed system and downloading the hardware/software to the FPGAs. Most importantly, the tool is extendable, where it allows importing third-party IP, for example.

The software tool allows users to compose systems that are based on the framework infrastructure presented in the previous chapter. Therefore, the tool recognizes the notion of SOPs and SAPs, and infers the number of Vortex routers that are required to furnish the communication between the accelerators in the system. Additionally, the tool is capable of generating the routing tables to configure the Vortex routers.

The *Cerebrum* software architecture, illustrated in Figure 3-1, is partitioned into frontend components (i.e. Graphical User Interface, GUI), and the backend components (i.e. Electronic Design Automation, EDA). The following subsections detail each component.

**Cerebrum Frontend (GUI)**

The frontend of *Cerebrum* offers the users a graphical interface for composing a system and automating the backend process. The GUI provides the user with access to a library of
highly-optimized IP cores, which can be dragged-n-dropped to the design canvas to compose a system.

The user subsequently defines connections between IP cores, which specify the communication among cores and guides the mapping to an MFS. Cores are categorized as either stream-oriented (i.e. SOP) or compute-oriented (i.e. SAP) and are described by an XML file called the “IP Core Specification”. This file specifies interfaces and contents of the core and is composed of two sections. The first section, `<Software>`, has several fields that determine how the core is exposed to the Cerebrum designer. The `<Hardware>` section details the internals of the core and is separated into three subsections: interface type, pcore set, and clocks. The interface type indicates the type of network interface to be used when attached to the Vortex router. The pcore set describes the library components that make up the core. Finally, the clocks subsection specifies required clock attributes such as frequency and phase.

Figure 3-1. The frontend (GUI) and backend (EDA) of the Cerebrum tool
To create a system, the user drags-n-drops compute-based and stream-oriented cores onto the design canvas and connects them as necessary. Compute-based cores allow users to create transactions and are programmed using small ANSI C programs called codelets. Accelerator functions are provided through the APIs that are specified along with the accelerator, as discussed in Chapter 2. Reprogramming stream data flows and making modifications to the codelets are permissible at any time, and do not trigger system synthesis. Figure 3-2 shows a screenshot of Cerebrum’s GUI.

**Cerebrum Backend (EDA)**

The Cerebrum backend performs the following tasks: (1) Mapping IP cores to an MFS, (2) Invoking third-party tools for synthesizing the cores, and (3) Codelet compilation and merging the bytecode with the hardware configuration file.

---

Figure 3-2. Cerebrum graphical user interface.
The library on the left populates neuromorphic cores that can be dragged and dropped to the design canvas.
The *Cerebrum* backend uses a number of specification files to accomplish these tasks. These files can be either one of the following categories: (1) *platform specification*: includes XML files that define I/O, resources, interconnections, and required interfaces of the target platform, (2) *design specification*: includes XML files which describe the IP cores, their interconnections, and any design parameters, and (3) *project options*: category consists of files that specify backend tool options. The backend uses an in-house-developed multi-FPGA accelerator-mapping algorithm that computes the optimal placement of IP cores onto the MFS.

**MFS mapping algorithm**

MFS partitioning is the process of breaking up the circuit and allocating these sub-circuits across the FPGAs. The last two decades witnessed a plethora of works that investigate the optimal partitioning. These works propose algorithms that vary in terms of partitioning granularity (e.g. gate-level vs. function-level), as well as the maximization/minimization function (e.g. maximizing performance vs. minimizing number of FPGAs). Both flattened and structural gate-level partitioning was investigated in previous works [24,25,26,27,28,29]. For instance, [24] used functional replication technique which performs circuit replication at the functional-cluster level to minimize delay and interconnect. This algorithm takes into account path delays and design structural information, to achieve high performance and high-density multi-FPGA partitioning. Furthermore, [26] proposed a two-phased natural and adaptive clustering method, followed by a simulated-annealing-based N-way partitioning process for timing optimization of multi-chip designs.

Although fine-granular partitioning at the gate-level may improve FPGA-density utilization, however, it may degrade the overall performance. For instance, partitioning a pipeline across two or more FPGAs may incur high volume inter-FPGA traffic. Alternatively, other works
proposed a more coarse-grain partitioning [30,31,32]. Vahid et al. [30] proposed using functional specification to direct the partitioning, while resources are assumed to be allocated to a shared bus. While this algorithm may be suitable for bus-based infrastructures, it may not be the right fit for NoC infrastructures. The SPARCS system [30,31] serves as a MFS partitioning algorithm along with high-level synthesis tool. While SPARCS has been shown to work on case studies, however, the authors of this system do not indicate how pre-mapping (i.e. user manually pre-allocates some of the components of the FPGAs) and component-grouping (i.e. user groups two or more components to be allocated to the same FPGA) can be supported with this system. The lack of these two features makes this system less flexible in attending to custom-user mapping constrains.

This subsection discusses an algorithm that was developed to allocate IP cores to MFS. The objective of this algorithm is to determine the most optimal placement of accelerators across the MFS. To achieve this, the algorithm employs a graph-based approach. Let $G_c = (V_c, E_c)$ be a graph defining the IP resource and connectivity, with $V_c$ and $E_c$ are defined as follows:

$V_c = \{v_j|v_j \text{ represents a component in the design which consumes a set of resources, } R}\}.$

$E_c = \{e_j|e_j \text{ is an edge from } v_j \text{ to } v_{j+1} \text{ and represents a communication between each node}\}.$

The physical resources of the FPGAs and their connectivity are represented with another directed graph, $G_F = (V_F, E_F)$, with the set of vertices, $V_F$, and edges, $E_F$, defined as follows:

$V_F = \{v_m|v_m \text{ represents an FPGA present in the design which provides a set of resources, } R\}.$

$E_F = \{e_m|e_m \text{ is an edge from } v_m \text{ to } v_{m+1} \text{ and represents the interconnectivity between each FPGA}\}.$

The direction of each edge defines the link direction; uni- or bi-directional.
Mapping Problem Formulation: Given the physical FPGA resource and connectivity graph \( G_F = (V_F, E_F) \) and the component graph \( G_c = (V_c, E_c) \) representing a design, find a mapping of the components, \( G_c \), to the FPGAs, \( G_F \), that does not exceed the resources available on any one FPGA.

Feasibility Check: A first-pass feasibility check is performed prior to mapping. During this pass it is guaranteed that enough resources exist across all FPGAs. If satisfied, the mapping proceeds through four phases:

I. **Component Grouping:** For all \( G_c \) vertices determine if there have been any that must be placed within the same FPGA. If so, annotate each group with the FPGA they are to be mapped onto.

II. **I/O Distance Calculation:** These distances will be used to map those components directly interfacing with I/O close to the source (sink).

III. **Pre-Map Allocation:** Place components that have been pre-mapped to FPGA’s.

IV. **Un-mapped Allocation:** A greedy approach is used to iterate through the groups and assigned to an FPGA vertex based on the available resources, and the I/O distance of each of the vertices, \( v_m \). As each vertex is visited, a check is performed to see if there are sufficient resources available. The FPGA that provides enough resources and has the lowest combined I/O distance is chosen.

**System Synthesis**

*Cerebrum* supports a number of FPGA platforms and devices. It has been used to create systems targeting the FPGA boards that host Xilinx Virtex-5 [33], and Virtex-6 devices [34]. These FPGA boards are: ML510 [35], Nallatech [36], ML505 [37], ML605 [38], and The Dini Group [39] development systems.
Chapter 4

Accelerating Neuromorphic Vision Algorithms

Neuromorphic vision algorithms are biologically inspired models that follow the processing that takes place in the primate’s visual cortex. While some of these algorithms exhibit efficiency and robustness, however, the computational complexity of these algorithms results in reduced performance when executed on general purpose processors. This chapter discusses a number of hardware neuromorphic accelerators that are developed to speed up HMAX—a neuromorphic feature extractor that is used to perform a variety of recognition tasks, including object recognition, face identification, facial expression recognition, and human action recognition.

Neuromorphic Vision Algorithms

The cognitive and visual processing capabilities of primates – in general – and humans – in particular – have always been source of inspiration to scientists, engineers, and technology makers. Take the human’s brain for example, a massively parallel processor consisting of $10^{11}$ billion individual processing elements, or neurons. The enormity in number of neurons translates to an unparalleled processing rate of $10^{16}$ FLOPS. 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 Nageswaran et al. [40], include sub-threshold spiking operations, sparse-energy efficient codes for signaling, and proper balance of analog computation and digital signaling.

While still an active research area, neuroscientists have proposed several computational models that are believed to represent the processing that takes place in the mammalian visual
cortex. One such model is that proposed by Itti et al. [41], which rapidly computes the conspicuous locations in a scene. Another example is HMAX [7,8], which is a brain-inspired feed-forward hierarchical model that has become a widely accepted abstract representation of the visual cortex. While similar in structure to Convolutional Neural Networks (CNNs) [42] and other Artificial Neural Networks (ANN), HMAX originates from the current understanding of the mammalian ventral visual stream—a hierarchy of brain areas responsible for carrying out object recognition. In fact, HMAX has been shown to correctly predict the output of similar units in experimentally captured read-outs from the monkey IT cortex [7]. From a computer vision perspective, HMAX is a scale-, orientation-, and shift-invariant feature extractor used for object recognition, with classification accuracy that competes with state-of-the-art machine vision algorithms as presented in [8,43].

Previous attempts to implement neuromorphic vision algorithms, such as HMAX, on CPUs and GPUs have shown that these platforms may not be the most suitable platforms, due to power and computational inefficiencies. These inefficiencies are mainly attributed to the disparity in the computational modalities prominent in these algorithms and those modalities most exploited in contemporary computer architectures. Furthermore, the use of rigid, non-configurable domain-specific accelerators may hinder the flexibility needed to allow parameter variations of these computational models. Therefore, the acceleration of neuromorphic algorithms requires strict adherence to a different set of computational and communicational requirements.

This chapter presents hardware accelerators for speeding up the HMAX algorithm. This algorithm was chosen for the following reasons:

1. The algorithm, as stated earlier, represents an accepted model of the processing in the visual cortex, carrying many of the distinct features of a biologically-inspired model. In addition, the reported classification accuracy scored by the model makes it an attractive component in embedded vision systems.
2. The inherent parallelism of the model makes it an ideal candidate for acceleration. Unlike other platforms, domain-specific accelerators can exploit this parallelism to enhance the overall performance of the algorithm.

3. As presented later in this chapter, several studies have proposed extensions to the HMAX model. This allows HMAX features to be used in other recognition tasks, such as face identification, facial expression recognition, and action recognition.

**The HMAX Computational Model**

**Model Description**

Figure 4-1 shows a computational template of HMAX. The model consists of a preprocessing stage, $S_0$, followed by two distinct types of computations, convolution and pooling (i.e. non-linear subsampling). The convolution and pooling stages correspond to the simple, $S$, and complex, $C$, cell types found in the visual cortex, respectively. Following the orientation-tuned $S_1$ layer, processing proceeds through alternating layers of complex pooling and simple template-matching. A number of HMAX implementations exist; where each implementation introduces variations in the layers of the model. Here, we use an implementation derived from the extension developed by Mutch & Lowe [43,44]. The following text describes the details of each layer in the model.

---

**Figure 4-1. The computational layers of the HMAX model**
**S₀ (Preprocessing) layer:** This layer is used for preprocessing the input image to ensure the uniformity of inputs and provide scale invariance. First, the image is converted to grayscale and then pixel values are normalized to the range [0, 1]. A 12-scale pyramid is created by downscaling the input image to sizes of 256×256 and smaller. The interpolation method can vary, however no noticeable improvement was gained using more complex techniques (e.g. bicubic) over simpler ones (e.g. nearest-neighbor) that are more favorable for hardware implementations.

**S₁ (Gabor filter) layer:** The S₁ layer, corresponding to the V1 simple cells, is based on an accepted model of the simple receptive fields of the cortex, Gabor filters [8]. These filters produce outputs for each scale at all desired orientations. The Gabor filters are 11×11 in size and are described by equation (1):

\[ G(x, y) = \exp - \left( \frac{(X^2 + Y^2)}{2\sigma^2} \right) \times \cos \left( \frac{2\pi x}{\lambda} \right) \]

Where \( X = x\cos(\theta) + y\sin(\theta) \), and \( Y = -x\sin(\theta) + y\cos(\theta) \). The model follows [8] and varies \( x \) and \( y \) between -5 and 5, and \( \theta \) between 0 and \( \pi \), while the wavelength (\( \lambda \)), width (\( \sigma \)), and aspect ratio (\( \gamma \)) are 5.6, 4.6, and 0.3, respectively.

**C₁ (local invariance) layer:** The C₁ layer – modeled after the complex cells of the V1 – pools over the outputs of the S₁ layer. This maximum-value pooling over local windows of adjacent scales provides both local scale-invariance and reduces the processing units required in subsequent layers by sampling only a subset of the S₁ output.

**S₂ (Tuned features) layer:** The S₂ layer models V4 or posterior IT by matching a set of randomly sampled 4×4×\( m \), 8×8×\( m \), 12×12×\( m \), and 16×16×\( m \) prototypes. The value of \( m \) represents the number of orientations extracted from the image in the S₁ layer. These prototypes make up a dictionary of \( k \) patches used as fuzzy templates for simple position- and scale-invariant features. S₂ then computes the response of a C₁ layer output patch, \( X \), to a particular S₂ feature prototype,
$P$, of size $n \times n \times m$ (typical $n = \{4, 8, 12, 16\}$). The number of patches, $k$, is determined through a learning phase, performed offline, that randomly selects feature prototypes of varying sizes from a set of images that represent the categorization task. If a general model is desired, the training set should contain images not related to any specific categorization task. The $S_2$ layer computes the final response using Normalized Dot Product, NDP, as shown in equation (2).

$$R(X, P) = \frac{X, P}{\sqrt{\sum x_i^2 - (\sum x_i)^2}}$$

As shown in the numerator of equation (2), the $S_2$ layer accumulates the responses from all orientations within each scale. The normalization stage, as shown in the denominator of equation (2), is performed by computing the normalization patch, which is computed from a windowed average of the current scale. The variable $x_i$ denotes a single pixel in the $C_1$ layer output patch. Then, pixel-wise division is performed by dividing the accumulated responses by the normalization patch.

$C_2$ (Global invariance) layer: The final layer provides global invariance by finding the per-prototype global maxima over all scales and positions, thus removing all position and scale information. The resulting complex feature set can then be used for classification to perform final object recognition.

The Computational Complexity of HMAX

The computational structure of the HMAX model exhibits a number of inefficiencies when executed on a general purpose processor; resulting in performance degradation. This subsection presents a detailed empirical study of the HMAX algorithm’s performance when
subjected to certain variations, on varying CPU configurations. This study helps in making adequate architectural design decisions when developing the hardware accelerators for the model.

In order to study the performance of the HMAX model, the HMAX implementation from [44] that was further extended to allow thread-level parallelism. The extended version was executed on an Intel Xeon-based system, consisting of two 2.4 GHz quad-core processors, with 12 GB system memory. Intel’s HyperThreading was enabled on all cores, providing a total of sixteen logical processors. Furthermore, the HMAX implementation made use of the SSE2 instruction set extension to benefit from the Single Instruction Multiple Data, SIMD, resources available on the cores.

Figure 4-2 shows the overall execution time of the model under two variations related to the number of orientations, while executing at different levels of parallelism (i.e. 1 to 16 threads). The figure shows that HMAX with 12 orientations is 2.5× – 2.8× slower when compared to HMAX with 4 orientations. This observation is expected since increasing the number of orientations increases the computational complexity of the $S_1$, $C_1$ and $S_2$ layers. Also, the figure shows that doubling the number of threads is associated with a consistent improvement in speed up by a factor of approximately 2×. An exception to this is the 16-thread configuration, where a

![Figure 4-2. HMAX execution time using different configurations](image-url)
speedup of only 1.3× is observed when compared to the 8-thread configuration. To explain this behavior, we study the per-layer execution time and its contribution to the overall execution time as demonstrated in Table 4-1.

Table 4-1. Execution time for each layer in the HMAX model, in percentage, out of the total execution time

<table>
<thead>
<tr>
<th># Threads</th>
<th># Orientations</th>
<th>S₁</th>
<th>C₁</th>
<th>S₂</th>
<th>C₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1.62</td>
<td>0.19</td>
<td>97.61</td>
<td>0.53</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>1.77</td>
<td>0.21</td>
<td>97.82</td>
<td>0.19</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1.69</td>
<td>0.22</td>
<td>97.45</td>
<td>0.56</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>1.79</td>
<td>0.22</td>
<td>97.78</td>
<td>0.19</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1.89</td>
<td>0.30</td>
<td>97.15</td>
<td>0.50</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>1.90</td>
<td>0.26</td>
<td>97.60</td>
<td>0.17</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>3.54</td>
<td>0.58</td>
<td>94.97</td>
<td>0.60</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>2.96</td>
<td>0.40</td>
<td>96.27</td>
<td>0.25</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>4.71</td>
<td>0.87</td>
<td>93.26</td>
<td>0.76</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>3.08</td>
<td>0.56</td>
<td>95.92</td>
<td>0.29</td>
</tr>
</tbody>
</table>

The table demonstrates how the contribution of each layer is influenced by the number of threads and the number of orientations. Note that the preprocessing stage, S₀, is not included in the table due to its insignificant contribution. The table shows that the S₂ layer dominates the total execution time with an average of 96.4% across all thread configurations for both 4 and 12 orientations. This explains the relatively low increase in speedup of the 16-thread configuration compared to the 8-thread configuration. Performing input-to-prototype correlation constitutes the critical path in the S₂ layer, as shown in Figure 4-3. Similarly, Figure 4-4 reveals that S₂ layer takes advantage of thread-level parallelism in gaining more speed up for both 4 and 12 orientations. However, the performance benefit associated with the increase in number of threads starts to decrease as number of threads exceeds a certain threshold (i.e. 8 threads in this case), as
convolution-like operations dominate the computation, exacerbated with the large number of prototypes in the $S_2$ dictionary. Similarly, Figure 4-5 shows the increase in power consumption as the number of instantiated threads increases.

For each scale, $s = 1:1:11$

// Compute normalization patches

For each orientation, $o = 1:1:m$

For each window, $w = \{4 \times 4, 8 \times 8, 12 \times 12, 16 \times 16\}$

$$\text{sum}_{\text{norm}}(w) = \text{corr}(C_1[s,o], \text{ones}[w])$$

$$\text{sum}^2_{\text{norm}}(w) = \text{corr}(C_1[s,o]^2, \text{ones}[w])$$

$$\text{len}(w) = \text{window} \_ \text{size}^2 \times m$$

$$\text{norm}(w) = \sqrt{\text{sum}^2_{\text{norm}}(w) - \text{sum}_{\text{norm}}(w) / \text{len}(w)}$$

end;

end;

For each prototype, $p = 1:1:5120$

For each orientation, $o = 1:1:m$

$$\text{result}_{\text{correlate}} += \text{corr}(C_1[s,o], \text{proto}[p,o])$$

end;

$$\text{result}_{S_2}[s, p] = \text{result}_{\text{correlate}} / \text{norm}$$

end;

end;

Figure 4-3. Pseudo code of $S_2$ stage

![Bar chart showing speedup in $S_2$ execution time when increasing number of threads. The figure shows speedup when increasing number of threads for both 4 and 12 orientations. Values are normalized to the single-thread configuration.](image-url)
From the observations presented above, it is evident that the HMAX model would benefit from parallelism. However, to further speed up the algorithm, application-specific acceleration is required for non-linear operations, in general, and convolutions, in particular. Also, profiling the model shows that $S_2$ layer is the hotspot and therefore, accelerating this layer will improve the overall execution time the most.

Extensions of the HMAX Model

Neuroscientists have observed that the primates’ visual system often shares a general, early-level processing structure, which eventually branches off into more specific higher-level representations. This serves as a motivation to customize the HMAX model to implement a variety of recognition problems beyond object classification. For example, to support face identification and facial expression recognition, Meyers et al. [45] proposed adding a center-surround stage of processing to model the ‘center-on surround-off’ processing that is present in

![Bar chart showing power consumption for various orientations and thread counts.](chart.png)

Figure 4-5. The increase in power consumption as the number of threads increases. The figure shows power consumption increase for both 4 and 12 orientations. Values are normalized to the single-thread configuration.
the retinal and Lateral Geniculate Nucleus (LGN) of the thalamus \[45\]. The center-surround is computed prior to pyramid generation and helps to eliminate intensity gradients due to shadows. The processing is done by placing a 2D window at each position in the input image that is identical in size to the filter used for \(S_1\) (i.e. 11×11). The output is then computed by dividing the current pixel’s intensity by the mean of the pixel intensities within the window. In addition, the model does not perform \(S_2\) and \(C_2\) stages in order to maintain visual features localized to a particular region in space.

While HMAX was originally limited to a model for the ventral stream, a model of the dorsal stream is useful for analyzing motion information. Jhuang et al. \[46\] have proposed augmenting the original HMAX model with the dorsal path as it can then be applicable to motion-recognition tasks, such as identifying actions in a video sequence. Computationally, this is done by integrating spatiotemporal detectors into \(S_1\), while adding two additional layers, \(S_3\) and \(C_3\), which track features over time, providing time-invariance to the structure.

The \(S_1\) units are composed using spatiotemporal Gabor filters described in \[47\]. In essence, these are Gabor filters that are extended by adding a third temporal dimension to the receptive fields. Computationally, this layer becomes \(n\)D convolution across a sliding window of past, present, and future frames, where \(n\) is total number of frames. The receptive field function of a spatiotemporal filter \(g_{v,\theta,\phi}(x, y, t)\) is defined by the following set of equations:

\[
g_{v,\theta,\phi}(x, y, t) = \frac{\gamma}{2\pi\sigma^2} \exp\left(\frac{-(\bar{x} + \mu_v t)^2 + y^2 \bar{y}^2}{2\sigma^2}\right) \\
\cdot \frac{1}{\sqrt{2\pi\tau}} \exp\left(\frac{-(t - \mu_t)^2}{2\tau^2}\right) \cdot U(t) \\
\cdot \cos\left(\frac{2\pi}{\lambda} (\bar{x} + \nu t) + \phi\right)
\]

\[\bar{x} = x \cos \theta + y \sin \theta\]
Equation (3) shows that the spatiotemporal response is a product of the following:

- A Gaussian envelope function that restricts $g_{v,\theta,\varphi}(x,y,t)$ in the spatial domain.
- Another Gaussian function that depends only on the time $t$ and determines the temporal decay of $g_{v,\theta,\varphi}(x,y,t)$
- A step function $U(t)$ which ensures that the filter based on $g_{v,\theta,\varphi}(x,y,t)$ considers inputs only from the past.
- A cosine wave traveling with a phase speed $v$ in direction $\theta$

Table 4-2 lists all the parameters that appeared in the equations (3) to (6) above along with a description of each parameter [48].

It is worth noting that other works such as [49] have also used these Gabor motion filters in performing Facial Expression Recognition. While the approach is conceptually sound, as both spatial and temporal processing of the visual signal can be performed to capture the emotions of the face, however, the execution time of this approach may be slower compared to other approaches.

On the other hand, the $S_3$ unit responses are obtained by temporally matching the output of $C_2$ features to a dictionary, similar to $S_2$, where each patch represents a sampled sequence of frames. $C_3$ unit responses are the maximum response over the duration of a video sequence. The authors of [46], however, indicated that adding the $S_3$ and $C_3$ layers result in marginal improvement in the classification accuracy.

Figure 4-6 illustrates a proposed multi-purpose recognition vision system that is based on the features extracted from the HMAX algorithm.

\[
\begin{align*}
\tilde{y} &= -x \sin \theta + y \cos \theta \\
U(t) &= \begin{cases} 
1 & \text{if } t \geq 0 \\
0 & \text{if } t < 0 
\end{cases}
\end{align*}
\]
Table 4-2. A description of the parameters of the spatiotemporal Gabor filter as defined in Equations (3) – (6):

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \gamma )</td>
<td>Spatial aspect ratio</td>
<td>Specifies the ellipticity of the support of the Gabor function</td>
</tr>
<tr>
<td>( \sigma )</td>
<td>Filter size</td>
<td>The size of the filter kernel</td>
</tr>
<tr>
<td>( v_c )</td>
<td>Envelope speed</td>
<td>The speed at which the center of the spatial Gaussian envelope moves along the x axis</td>
</tr>
<tr>
<td>( \tau )</td>
<td>Mean receptive field duration</td>
<td>Determined by the standard deviation of the temporal Gaussian</td>
</tr>
<tr>
<td>( \mu_t )</td>
<td>Mean time delay</td>
<td>The mean of the temporal decay Gaussian function</td>
</tr>
<tr>
<td>( v )</td>
<td>Speed</td>
<td>The preferred speed of the filter</td>
</tr>
<tr>
<td>( \theta )</td>
<td>Orientation</td>
<td>The preferred spatial orientation and direction of motion</td>
</tr>
<tr>
<td>( \varphi )</td>
<td>Phase offset</td>
<td>Determines the symmetry of the filter in the spatial domain with respect to its (moving) center</td>
</tr>
<tr>
<td>( \lambda )</td>
<td>Spatial wavelength</td>
<td>Wavelength of the cosine factor</td>
</tr>
</tbody>
</table>

Figure 4-6. A multi-purpose recognition system using HMAX features. 

\( S_2 \) accepts multiple dictionaries depending on the current recognition task. For example, Prototypes 1 is used to extract features for object recognition task, while Prototypes 2 is used for action recognition task.
The Architecture of the HMAX Accelerators

This section discusses the design and implementation of the hardware accelerators for the HMAX model.

$S_1$ Accelerator

The $S_1$ layer performs a streaming convolution on its input. The $S_1$ accelerator, as illustrated in Figure 4-7, is a SOP that consists of a serial-to-parallel FIFO, a 2D streaming convolution engine that feeds an adder tree, and a bank of Gabor coefficients. The serial-to-parallel FIFO reconstructs the serially streamed image pixels into row-wise parallel output that is fed to the 2D filter. The convolution engine is a 2D $11\times11$ filter that feeds an adder tree. The accelerator utilizes a scratchpad memory (coefficients bank) to store the Gabor coefficients for all supported orientations. These coefficients are selectively accessed and loaded to the convolution engine based on the incoming flow associated with the current frame. Note that there is an initial latency to fill the serial-to-parallel FIFO with input image, proportional to the image width, in addition to the latency to compute the filter response. After that latency, output results are produced at a rate of one filter response per cycle.

For the spatiotemporal Gabor filter, the $S_1$ accelerator is expended by iteratively sending consecutive frames and using an additional scratchpad memory structure to store the intermediate outputs.
Figure 4-7. $S_1$ accelerator.
(a) The accelerator consists of a serial-to-parallel FIFO to convert the streaming serial input into parallel output that feeds the 2D systolic filter. The filter supports reloadable coefficients. (b) The coefficients are stored in a multi-bank scratchpad memory.

Figure 4-8. $C_1$ accelerator.
(a) The accelerator consists of units that keep track of the current $x$ and $y$ coordinates and processors that perform windowed pooling in the $x$ and $y$ positions. (b) The Horizontal-Window processor is a 1D window processor in the $x$ direction.
**C₁ Accelerator**

The C₁ accelerator, as illustrated in Figure 4-8, is a SAP that performs a windowed pooling on a stream of inputs. The accelerator contains several local memories used to store x and y position information for each window. This information is compared against the current incoming pixel’s location in order to determine to which window a current pixel belongs. Since windows can overlap in both x and y dimensions, multiple compare units exist, with each responsible for comparing a separate sub-window in x and y in a hierarchical fashion. At the lowest level, Horizontal-Window processors are responsible for a 1D horizontal window in x, which tracks the current sub-windows assigned to it. Alternatively, a Vertical-Pixel router keeps track of which Horizontal-Windows are currently active in order to handle overlaps in y.

**S₂/C₂ Accelerator**

The S₂ layer performs template matching through correlation between the outputs of the C₁ layer and pre-stored prototypes. The outputs of this layer are pooled on by the C₂ layer to find the global maximum across all positions and scales for each prototype in the S₂ dictionary. Careful analysis of the data flow between S₂ and C₂ layers shows that combining these two layers into a single SAP-PE accelerator leads to several benefits. First, this allows for the pooling operation to occur immediately following the computation of the current S₂ output without a delay. Second, combining these two layers can effectively decrease the amount of data required to be sent across the network by:

\[
\sum_{S=0}^{S-1} (X_S \times Y_S) \times N_{\text{proto}}[X_S, Y_S] \\
(N_{\text{proto}} \times 2)
\]

Here S is the number of input image scales at the S₂ layer, Xₜ (Yₜ) is the dimension of scale S in the x (y) dimension, and N_{\text{proto}} is the number of prototypes. The notation N_{\text{proto}}[Xₜ, Yₜ] is used to
denote total number of prototypes that can be correlated with that scale. With a dictionary of 5120 prototypes, this results in a data transfer reduction of 4,154× when operating on 12 scales at 12 orientations.

Figure 4-9 illustrates the architecture of the S₂/C₂ accelerator. First, the input scale (i.e. C₁ output) is buffered in the Image Memory, which is a scratchpad memory with enough capacity to store all orientations of the largest scale. Since template matching is an iterative process across all prototypes in the S₂ dictionary, the local buffering of the input scale can dramatically reduce the amount of data sent across the network by:

\[ \frac{\sum_{S=0}^{S=1}(X_S \times Y_S) \times N_{orient} \times N_{proto}[X_S, Y_S]}{\sum_{S=0}^{S=1}(X_S \times Y_S) \times N_{orient}} \]

Here, \( N_{orient} \) is number of orientations. Buffering the image reduces the amount of data sent across the network by approximately 5005× when using 12 scales, 12 orientations, and 5120 prototypes.

The image buffer feeds an array of convolvers, allowing the accelerator to parallelize the template-matching process across these convolvers. Each one of these convolvers is a 2D, multi-tap systolic engine with reloadable coefficients, where coefficients are the fuzzy templates in the S₂ dictionary. Each convolver can support all four kernel sizes (i.e. 4, 8, 12, and 16), which can be configured at runtime by the accelerator’s controller (not shown in figure). The prototypes are stored in a SRAM memory due to their relatively large size (~24 MB). To hide the latency of the SRAM memory, the accelerator overlaps memory reads and convolution, where FIFOs are used to buffer prototypes from the memory.
The $S_2$ layer accumulates the correlation output across all orientations within the same scale for the current prototype. This is accomplished through the Accumulation pipeline stage, where the output of the convolution is processed in a streaming fashion. A temporary buffer, Accumulation Memory, stores the current output of each correlation. When the convolution engine produces the output of the next orientation, the hardware logic reads the corresponding accumulated output of the previous correlation from memory, updates it with the current output, and writes the result back to the Accumulation Memory. The logic was designed such that accumulation is done in a streaming fashion to sustain a throughput of one accumulation per cycle. Only after processing the last orientation, does the Accumulation stage stream the accumulated output to the Normalization stage. The Normalization stage normalizes the

Figure 4-9. $S_2/C_2$ accelerator. The Image Memory buffers the $C_1$ outputs, and feeds the $N$ 2D convolution engines. The convolution engines load the $S_2$ dictionary stored in SRAM. The Accumulation stage performs pixel-wise accumulation across correlation outputs for each orientation within the same scale. The Normalization stage normalizes the accumulated output. The $C_2$ stage performs global max over all positions and scales for each prototype independently. The accelerator can be configured and controlled through the SAP-PE’s ASR and ASI commands.
accumulated correlation output to the pre-computed normalization patch, which is the output of a windowed averaging function applied on the current input scale. This stage operates in streaming fashion, where in each cycle the logic reads the corresponding normalization pixel from the *Normalization Patch Memory*, performs normalization, and then transfers the results to the C₂ pooling stage.

The last stage in the accelerator performs the global pooling to find the maximum response across all positions and scales for each prototype. Upon receiving a new response from the previous stage, the hardware logic in the C₂ stage fetches the corresponding response from the *C₂ Tables*. If the current value is larger than the value stored in the table, then the logic will update table with the current value. When scales and prototypes are fully processed, a read request can be made to the S₂/C₂ accelerator, which transfers the feature vector stored in the C₂ tables by DMA.

**Experimental Evaluation**

The hardware accelerators discussed in the previous section were implemented in synthesizable HDL. To validate the functionality and performance of the implemented accelerators, a multi-FPGA system is used as a prototyping platform. The platform was also used to measure the power consumption of the accelerators. Furthermore, CPU and GPU implementations of HMAX model are used to compare with the implemented accelerators.

**Nallatech Development System**

A Multi-FPGA platform from Nallatech [36] is used to validate the neuromorphic accelerators discussed above. The platform houses an Intel S7000FC4UR motherboard with a
Quad-Core Xeon processor running at 1.6 GHz, with a total of 24 GB system memory. The Intel motherboard interfaces to acceleration modules using the Intel QuickAssist technology [50] enabling optimized use and deployment of FPGAs on Intel platforms. The platform makes that interface possible using a high-speed bus called Front-Side Bus, FSB, which is a 64-bit bus running at a frequency of 1066 MHz allowing data transfer rates of up to 5.8 GB/s sustained read (i.e. System Memory to FPGA), and 2.8 GB/s sustained write (i.e. FPGA to System Memory). Access to the Bus is facilitated using a software framework that is based on Intel’s Accelerator Abstraction Layer, AAL, [51] and Xilinx Accelerated Computing Platform, ACP, [52]. The system contains two computing modules; each houses two Xilinx Virtex-5 SX240T FPGAs [33]. Figure 4-10 illustrates the Nallatech development system.

Figure 4-10. Nallatech™ Development System
(a) The Intel motherboard houses an accelerator socket, where base module (used as an interface) and computing modules (used for acceleration) are vertically stacked on the socket. (b) The diagram shows how the system interfaces to the computing modules through a base module that is responsible for transferring data to and from the computing modules. There are two computing modules; each contains two Virtex-5 SX240T FPGAs.
The Neuromorphic Acceleration System

Each neuromorphic accelerator was individually validated both using cycle-accurate simulator and on the FPGA prototyping platform. However, due to resource limitation on the prototyping platform, not all the accelerators can be mapped simultaneously. Additionally, Table 4-1 shows that the execution time of $C_1$ stage is less than 1% of the total execution time. Therefore, only $S_1$ and $S_2/C_2$ accelerators are mapped to the Nallatech development system, as these two stages constitute the top two hotspots of the system, while $C_1$ stage is computed in software that is executed on the host machine. Note that the total execution time of the system includes the overhead latency due to data transfer between the CMP and the FPGAs.

Figure 4-11 shows the data flow paths for the HMAX neuromorphic acceleration system. In addition to the SOP $S_1$ and SAP-PE $S_2/C_2$, the system includes the following components: (1) SAP-PE Ingress, which is responsible for transferring data from CMP, through FSB, and to the

![Figure 4-11](image)

Figure 4-11. Data flow paths for the HMAX neuromorphic system.
(a) Data flow path for processing both 2D Gabor and spatiotemporal Gabor filter. (b) Data flow path of the template-matching and global max stages. Note that when CMP requests HMAX feature vector, the SAP-PE Ingress sends a notification message to the SAP-PE Egress, which in turn makes a DMA read request from the $S_2/C_2$ accelerator.
desired destination, (2) SOP FL2FX, a stream-based core for converting values from floating-point to multiple fixed-point representations. The exact fixed-point representation can be selected on-the-fly based on the current flow, (3) SOP FX2FL, which is another stream-based core that converts the output values from fixed-point to floating-point representation, and (4) SAP-PE Egress, which transfers data from accelerators – through FSB – to the CMP. The CMP and the accelerators interact with each other following the steps listed in Table 4-3.

According to the table, the execution of face identification and facial expression recognition applications involve only the steps 1, 2, and 5. In contrast, the execution of object recognition and human action recognition applications involve all the steps in the table.

The following subsection presents two additional platforms that are used to compare with the prototyping platform.

Table 4-3. The interactions between CMP and accelerators

<table>
<thead>
<tr>
<th>Step</th>
<th>Operation</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Configure Vortex router, SOPs’ and SAPs’ flow tables</td>
<td>Once at initialization, and when a route/flow information need an update</td>
</tr>
<tr>
<td>2</td>
<td>Write Gabor coefficients to S1 accelerator</td>
<td>Once at initialization</td>
</tr>
<tr>
<td>3</td>
<td>Write S2 dictionary to S2/C2 accelerator</td>
<td>Once at initialization, and when a new dictionary is available</td>
</tr>
<tr>
<td>4</td>
<td>Configure ASR registers of S2/C2</td>
<td>Once at initialization, and when ASRs update is required</td>
</tr>
<tr>
<td>5</td>
<td>Write input scales to S1 and read results (See solid line in Figure 4-11 (a))</td>
<td>Every new input frame</td>
</tr>
<tr>
<td>6</td>
<td>Write C1 output to S2/C2 (See solid blue line in Figure 4-11 (b))</td>
<td>Every new input frame (Object recognition, and Action recognition application only)</td>
</tr>
<tr>
<td>7</td>
<td>Request S2/C2 result (See dotted green line in Figure 4-11 (b))</td>
<td>Every new input frame (Object recognition, and Action recognition application only)</td>
</tr>
<tr>
<td>8</td>
<td>Read HMAX feature vector (See solid blue line in Figure 4-11 (b))</td>
<td>Every new input frame (Object recognition, and Action recognition application only)</td>
</tr>
</tbody>
</table>
**CPU and GPU Platforms**

A CPU implementation of the HMAX model [44] is used to evaluate the performance of HMAX when executed on the CPU platform. The workload was further extended to allow thread-level parallelism across scales, orientations, or prototypes. The workload was compiled with the highest possible optimization and using the SSE2 instruction set extension. The workload was executed on an Intel Xeon machine, consisting of a 12-Core processor clocked at 2.4 GHz. This section assumes CPU performance when all 12 cores are utilized for executing the workload.

Similarly, an optimized CUDA implementation of the HMAX model [53] was executed on an Nvidia Tesla M2090 board [54], which houses a 1.3 GHz Tesla T20A GPU, with a 1.34 GB global memory. A 3 GHz 12-core Xeon processor is used to access the GPU, with a total of 49 GB system memory.

**Datasets Used for Evaluation**

As discussed earlier, the HMAX model can be used as a feature extractor for a variety of classification tasks; namely, object recognition, face identification, facial expression recognition, and action recognition. Table 4-4 lists a number of datasets that are used to measure the classification accuracy of the model for different application domains. The fifth column in the table shows the classification accuracy scored by the accelerated HMAX for that particular dataset.

The Caltech-101 [55], and Caltech-256 [56] datasets are used to test the accuracy of object classification using all 102, and 257 different categories, respectively. Similarly, the In-House dataset consists of 16 categories with a variety of objects including vehicles, aircrafts, military equipment and background scenery. In addition, the Pascal Visual Object Classes Challenge 2007 [57] was also used to measure the accuracy of object recognition.
Three datasets were used to measure the accuracy for face processing application domain.

The ORL dataset [58], used for testing face identification accuracy, contains a collection of close-up images of the faces of 40 different individuals from varying viewing angles. The FERET dataset [59,60], also used for face identification accuracy measurements, includes 1208 individuals from which a random subset of 10 individuals was chosen for evaluation. The JAFFE dataset [61] is used for testing the accuracy of facial expression recognition. Six different expressions were tested—anger, disgust, fear, happiness, sadness, and surprise. Finally, the Weizmann dataset [62] is used for testing human action recognition. This dataset includes video sequences of 10 different categories of action: bending, jumping jacks, vertical jumping, horizontal jumping, skipping, running, side-stepping, walking, one-hand-waving, and two-hand-waving. Figure 4-12 shows representative samples of each dataset.

Table 4-4. Datasets used for evaluation: Accelerated HMAX is used to extract features from the datasets, which is used as an input to a regularized least-square (RLS) classifier.

<table>
<thead>
<tr>
<th>Application Domain</th>
<th>Dataset</th>
<th># Categories</th>
<th># Test samples</th>
<th>Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Recognition</td>
<td>Caltech-101</td>
<td>102</td>
<td>4543</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>Caltech-256</td>
<td>257</td>
<td>30607</td>
<td>See Figure 4-13</td>
</tr>
<tr>
<td></td>
<td>In-House</td>
<td>16</td>
<td>1382</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>Pascal VOC2007</td>
<td>20</td>
<td>4952</td>
<td>See Figure 4-14</td>
</tr>
<tr>
<td>Face Identification</td>
<td>ORL</td>
<td>40</td>
<td>200</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>FERET</td>
<td>10</td>
<td>60</td>
<td>70</td>
</tr>
<tr>
<td>Facial Expression</td>
<td>JAFFE</td>
<td>6</td>
<td>60</td>
<td>86.7</td>
</tr>
<tr>
<td>Recognition</td>
<td>Weizmann</td>
<td>10</td>
<td>40</td>
<td>77.7</td>
</tr>
<tr>
<td>Action Recognition</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Classification Accuracy

The classification accuracy is a measure of how well an algorithm is capable of correctly classifying an object, emotion, or even an action. The fifth column in Table 4-4 shows the classification accuracy for some of the datasets using the feature vector extracted by the accelerated HMAX. The recognition accuracy across all the platforms (i.e. FPGA, CPU, and GPU) was similar. However, a slight degradation in accelerators accuracy was observed (i.e. \( \leq 2\% \)) since these accelerators use fixed-point to represent values. This degradation is mainly attributed to the truncation of the fixed-point representation during the multiply & accumulate operations.

Next, we compare the classification accuracies reported in Table 4-4 with other algorithms. For example, Zhang et al. [63] achieved up to 66\% on Caltech-101 using 30 training images per category. Accelerated HMAX scored 70\% while using more training images per category. Furthermore, Ahonen et al. [64] scored 98\% on the ORL dataset; more than 13\% better than accelerated HMAX. Similarly, the algorithm proposed by Shih et al. [65] achieved 94.13\% on the JAFFE dataset when using cross-validation strategy. In contrast, accelerated HMAX scored 86.7\% on the same dataset. Finally, accelerated HMAX scored 77.7\% classification accuracy on the Weizmann dataset, which is more than 22\% less than that scored by Gorelick et al. [66] on the same category using leave-one-sequence-out strategy.
Figure 4-12. Representative samples from the datasets
Additional experiments were conducted to analyze the sensitivity of accelerated HMAX to number of training images. Figure 4-13 illustrates the classification accuracy for the Caltech-256 dataset for both 4 and 12 orientations, where the number of scales is fixed to 12. The figure shows the influence of the number of training images on the overall accuracy. For example, when using 40 images to train the classifier, the overall accuracy is 25.95% (23.23%) for 12- (4-) orientation configuration. Compared to other object recognition algorithms, we find that the accuracy of HMAX is at most 12% less than that reported by Griffin et al. [56]. However, we are unable to determine exactly how much of that loss in accuracy is attributed to HMAX feature extractor, as we use a different classifier.

Figure 4-14 shows the classification accuracy for the PASCAL VOC2007 dataset. A multiple-classifier voting scheme was used for generating the probabilities (i.e. confidence rates). A total of 10 independent classifiers were used, each trained using equal-sized disjoint subsets of the training data, with a total of 15662 objects, 1566 objects per subset, extracted from 5011 images. The classifications and generated probabilities are fed to the PASCAL evaluation tool in order to retrieve the Average Precision, AP, rate. The AP rate measures the mean precision at a

![Bar Chart](image)

**Figure 4-13.** Classification accuracy for the Caltech256 dataset

Accuracy was measured using 4 and 12 orientations with varying number of training images.
set of equally spaced recalled level under the Precision/Recall curve.

In conclusion, it is found that HMAX performs well as a generic feature extractor for diverse workloads. However, its performance on any specific workload may not be necessarily competitive with other state-of-the-art feature extractors.

**Speed**

Frames (segments) processed per second, fps, is used as a metric to compare the speedup gained by each platform. The term “segment” here to refer to a group of 20 frames extracted from a video sequence for action recognition application. Figure 4-15 shows the speedup in fps when accelerating HMAX on the FPGA platform by splitting the workload across 1 to 4 $S_2/C_2$ units. This figure shows how the framework can be easily used to exploit task-level parallelism by controlling the number of active accelerators in the framework.

Figure 4-16 shows a speedup comparison between the three platforms in terms of fps for multiple recognition tasks. The FPGA prototyping platform demonstrates a speedup of up to $7.6 \times$
(4.3×) when compared to the CPU (GPU) platform. The FPGA platform exhibits increased performance improvement in the action recognition application. This is due to the cumulative effect of per-frame performance of the $S_1$ stage. Since each video segment consists of 20 frames, the FPGA accelerator sees a linear increase in performance with each frame.

**Power Efficiency**

The experimental setup included power consumption measurements for all three

![Figure 4-15. Task-level parallelism as exploited by the accelerator system.](image)

![Figure 4-16. Speedup across the three platforms.](image)

A comparison across the three platforms for each application domain. The figures are normalized to the CPU platform. Values in the yellow boxes show the fps of the accelerators for each application domain.
platforms. For the GPU platform, the command-line tool “nvidia-smi -q” is used to retrieve the power consumption from a power sensor found on the GPU board. For the CPU and FPGA platforms, power consumption was measured using an accurate meter for power consumption measurements. The meter provides continuous and instantaneous reading of power drawn by the platform with 99.8% accuracy. The power consumption for all platforms is measured only after the platform reaches steady-state to obtain the baseline power measurement. Then, the workload is executed and peak power is measured throughout the duration of the workload execution. For example, the power measurements show that when running HMAX for object recognition, the GPU, CPU and FPGA platforms consume 144, 116 and 69 Watts, respectively.

Speedup is not the only metric used for evaluating the performance and efficiency of an embedded design. Power efficiency, measured in fps-per-watt, is another metric that is used to assess the amount of performance that can be delivered within an available power budget. As shown in Figure 4-17, the HMAX accelerators demonstrate a significant performance-per-watt benefits, ranging from $5.3\times$ to $12.8\times$ ($3.1\times$ to $9.7\times$) when compared to CPU (GPU) platform.

HMAX accelerators implement customized architectures that deliver high throughput while operating at a low frequency. This is the main driver for lower power consumption, and consequently higher power efficiency.

These speedup and power efficiency benefits are achieved by mapping the neuromorphic accelerators to 65nm SRAM-based FPGA devices operating at 100 MHz clock, to 45nm and 40nm technologies used with CPU and GPU platforms, respectively. It is expected that implementing the neuromorphic accelerators in silicon rather than on an FPGA platform will accentuate such benefits. For instance, Kuon et al. [68] show that at 90nm fabrication process, moving from SRAM-based FPGA to CMOS ASIC architectures improves critical path delay by $3\times$ – $4.8\times$, and dynamic power by $7.1\times$ – $14\times$. 
Configurability of Accelerators and Design Tradeoffs

Like other computational models, HMAX provides a wide spectrum of input parameters and configurations that has a direct impact on the final output. The HMAX accelerators allows for such configurability at run time. This can come in handy, for example, when it is desirable to trade off accuracy for higher performance. Therefore, we experimented further with HMAX accelerators to study the impact of reduced accuracy on the execution time. For example, Figure 4-13 shows that changing the number of orientations from 12 to 4 results in 2.72% degradation in classification accuracy for 40-image training set. At the same time, this change in number of orientations improved the execution time by 2.2×. In another experiment, the numbers of input scales was varied, while observing its influence on accuracy and speedup using the In-House dataset. Figure 4-18 shows that as the number of scales decreases the classification accuracy also decreases until it reaches ~70% when using 5 input scales. On the other hand, Figure 4-19 shows a consistent improvement in speedup and power efficiency as number of scales is decreased,

Figure 4-17. Improvement in power efficiency (fps-per-watt) across the three platforms. A comparison across the three platforms for each application domain. The figures are normalized to the CPU platform. Values in the yellow boxes show the power efficiency of the accelerators for each application domain.
effectively reaching 15.4× speedup (i.e. 18.5 fps) and 15.4× power efficiency (i.e. 2.6 fps-per-watt) when using only 5 scales compared to 12-scale configuration. Permitting such trade-off analysis makes the proposed accelerator very suitable for studies in modeling refinements and vision algorithm tuning.

**Discussion of Results**

There are a number of factors that played a role in the speedup and power efficiency exhibited by the neuromorphic accelerators. These factors are listed below:

![Figure 4-18](image1.png)

**Figure 4-18.** The influence of number of scales on classification accuracy. As the number of input scales decreases, the classification accuracy decreases.

![Figure 4-19](image2.png)

(a)  
(b)

**Figure 4-19.** The influence of number of scales on performance. As the number of input scales decreases, both speedup (a) and power efficiency (b) increase. The values in the figures are normalized to the 12-input-scale configuration.
1. **Fully pipelined and customized streaming architecture**: These customized architectures allow for data reuse, hence avoiding unnecessary data fetching. For instance, the systolic correlation filter implemented within the S₂/C₂ accelerator exploits data reuse where pixels are propagated across multiple multiply-and-accumulate units.

2. **Exploitation of parallelism**: FPGAs offer high degree of parallelism that is often not available on other platforms. For example, the S₂ correlation filter performs 256 multiply-and-accumulate operations simultaneously, providing a 256× increase in performance over sequential operation. This high degree of parallelism is not achievable on general purpose CPU architectures. Even contemporary architectures with explicit vector-processing extensions lack the number of functional units and optimized memory infrastructure to exploit the immense data-level locality inherent in the many convolution operations of both the S₁ and S₂/C₂ accelerators.

3. **Custom numerical representation**: All accelerators use fixed-point representation with varying bit widths suitable for the current operation. For instance, the S₁ accelerator uses 18 bits to represent each Gabor coefficient, while 22 bits are used to represent each pixel in the input image. This ability to operate on varying bit width operands is unmatched by CPUs and GPUs.

4. **Task-level parallelism**: In order to improve the execution time of HMAX, four instances of the S₂/C₂ accelerator were mapped to the FPGAs. Each instance operates on a subset of the prototypes, and therefore, total execution time was reduced to approximately 25%.

5. **Efficient use of memory hierarchies**: The HMAX accelerators made use of multiple hierarchies of memory. For instance, the S₂/C₂ accelerator used SRAM
memory that feeds a queue buffer in order to overlap data fetch and computation. This mechanism hides the latency of data fetch and hence improves overall performance. Similarly, the accelerator used a local buffer to store the input images in order to reduce communication latency. Moreover, these customized memory hierarchies improves data reuse and reduces the unnecessary and expensive data fetch from main memory. Other platforms, such GPUs, lack such memory hierarchies and therefore are less efficient in terms of memory access.

6. **Reliable communication infrastructure**: The accelerator framework, presented in Chapter 2, offers high and reliable communication infrastructure. Measurements taken from the prototype platform show that this network can achieve up to 1.6 GB/s (3.2 GB/s) bandwidth when clocked at 100 MHz (200 MHz). Additionally, using flows to associate incoming packets with operations reduces the latency that would otherwise be incurred had the accelerators been required to be reconfigured. For instance, the $S_1$ accelerator determines the current orientation to process by simply examining the flow of the current frame. Hence, the user is not required to pre-configure the accelerator for each orientation.

**Resource Utilization**

The neuromorphic accelerators were validated on four Virtex-5 SX-240T [33] FPGAs. Table 4-5 shows the amount of resources utilized by the HMAX accelerators when mapped to the FPGAs. The slice registers and slice Lookup Tables, LUTs, were mostly utilized for implementing registers and control logic. The Block-RAM, BRAM, units are mainly dedicated
for scratchpad memories and FIFO-like structures, while the DSP48E slices were mainly allocated for multiply-and-accumulate operations that took place in the convolution engines.

Table 4-5. The resource utilization of the neuromorphic accelerators. Accelerators are mapped to four Virtex-5 SX-240T FPGAs. The numbers in brackets show the utilization in percentage.

<table>
<thead>
<tr>
<th>Slice Registers</th>
<th>Slice LUTs</th>
<th>BRAM</th>
<th>DSP48E</th>
</tr>
</thead>
<tbody>
<tr>
<td>316,794</td>
<td>133,611</td>
<td>623</td>
<td>2,206</td>
</tr>
<tr>
<td>(52.88%)</td>
<td>(22.3%)</td>
<td>(30.18%)</td>
<td>(52.23%)</td>
</tr>
</tbody>
</table>

**Hardware- vs. Software-controlled SAP-PE**

In Chapter 2, the SAP-PE was introduced as a programmable hardware accelerator that can be controlled by software called codelet. As discussed earlier, the software can be replaced by hardware state machine in an effort to speed up the execution time of the control path. It was argued that this approach is not recommended due flexibility issues. Here, we give more solid proofs that the software-controlled SAP-PE is not considerably less efficient than the hardware-controlled SAP-PE.

In this chapter, we presented two SAP-PEs; namely, SAP-PE Ingress and SAP-PE Egress. These two SAP-PEs are responsible for orchestrating data movement between the CMPs and accelerators, and both are controlled by software rather than hardware logic. To compare between the two approaches, we re-implement both SAP-PEs to be controlled via a state machine. Figure 4-20 shows a pseudo code of the Ingress and Egress codelets.
First, we compare the execution time of the software-controlled SAP-PE to the hardware-controlled version using a cycle-accurate simulator. Table 4-6 shows an instruction-level

while (true)

  // Wait for message from Ingress
data = get_msg(msg_id)

  // prepare read request based on msg info
  // then read feature vector from S2/C2
dma_read(data.info, first_space_address)

  // read second space address
dma_read(data.info, second_space_address)
end;

(a)

while (true)

  // read custom core (blocking statement)
data = cc_read()

  // check the request type, act accordingly
  if (data.type = WR_REQ)
    // Write request to desired destination
dma_wrte(data.info)
end;

else

  // forward configuration packet to desired destination
direct_write(data.info)

  // Check if this is read request from CMP to S2/C2
  if (data.info.destination = S2/C2 and data.info.address_space = request_results)
    // send a message to Egress
    msg_send(egress_address, msg_id, data.info)
  end;
end;

(b)

Figure 4-20. Pseudo code of the SAP-PE Egress (a) and SAP-PE Ingress (b).
execution time of each approach. The table indicates that the software-controlled Ingress (Egress) is slower than the hardware-controlled version by 740ns (290ns). This degradation in execution time is insignificant for the following reasons: (1) This difference is at best less than 0.002% of the total execution time of HMAX, (2) The control path is infrequently invoked – only to orchestrate data transfer and synchronize accelerators, and (3) In some of the application (e.g. facial identification) that control path may not be invoked at all after configuring the accelerator at initialization stage.

Table 4-6. Execution time of hardware- vs. software-controlled SAP-PE.

The measurements are taken from a cycle-accurate simulator in clock cycles (CC) and converted to nanoseconds (shown in brackets) assuming 100 MHz clock frequency.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Hardware Exec. Time CC (ns)</th>
<th>Software Exec. Time CC (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAP-PE Ingress</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CC RD 1</td>
<td>1 (10)</td>
<td>10 (100)</td>
</tr>
<tr>
<td>CC RD 2</td>
<td>1 (10)</td>
<td>10 (100)</td>
</tr>
<tr>
<td>Direct WR</td>
<td>1 (10)</td>
<td>32 (320)</td>
</tr>
<tr>
<td>SEND MSG</td>
<td>1(10)</td>
<td>15 (150)</td>
</tr>
<tr>
<td>DMA WR</td>
<td>2 (20)</td>
<td>13 (130)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>6 (60)</strong></td>
<td><strong>80 (800)</strong></td>
</tr>
<tr>
<td>SAP-PE Egress</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GET MSG</td>
<td>1 (10)</td>
<td>10 (100)</td>
</tr>
<tr>
<td>DMA RD 1</td>
<td>1 (10)</td>
<td>11 (110)</td>
</tr>
<tr>
<td>DMA RD 2</td>
<td>1 (10)</td>
<td>11 (110)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>3 (30)</strong></td>
<td><strong>32 (320)</strong></td>
</tr>
</tbody>
</table>

Next, we compare the resource utilization of the two approaches. Table 4-7 shows the utilization for both approaches when mapped to Virtex-5 SX240T device. As expected, the software approach requires more resources as it hosts a Light Weight Processor, LWP, to be
instantiated within the control path of the SAP-PE. However, this increase is considered marginal when considering the flexibility and programmability benefits of software-controlled SAP-PE.

Table 4-7. Resource utilization comparison between software- and hardware-controlled SAP-PE

<table>
<thead>
<tr>
<th>Resource</th>
<th>Hardware Count (%)</th>
<th>Software Count (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAP-PE Ingress</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slice Registers</td>
<td>2,583 (1.72)</td>
<td>2,655 (1.77)</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>2,512 (1.68)</td>
<td>3,008 (2.01)</td>
</tr>
<tr>
<td>BRAM</td>
<td>3 (0.58)</td>
<td>4 (0.78)</td>
</tr>
<tr>
<td>DSP48E</td>
<td>0 (0)</td>
<td>3 (0.28)</td>
</tr>
<tr>
<td>SAP-PE Egress</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slice Registers</td>
<td>942 (0.63)</td>
<td>2,224 (1.49)</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>1,048 (0.7)</td>
<td>2,818 (1.88)</td>
</tr>
<tr>
<td>BRAM</td>
<td>3 (0.58)</td>
<td>4 (0.78)</td>
</tr>
<tr>
<td>DSP48E</td>
<td>0 (0)</td>
<td>3 (0.28)</td>
</tr>
</tbody>
</table>

In addition, the software approach has other benefits when considering the times it takes to synthesize the system. For example, mapping the HMAX accelerators to one of the the FPGA devices takes approximately 5 hours to complete synthesis and implementation. In contrast, compiling the codelet and merging it with the hardware bitstream takes less than a minute, resulting in 300× improvement.

Finally, the software approach is much more efficient in terms of line-of-code count and the complexity of code development and debugging. For example, the Ingress codelet was written in 97 lines of ANSI C, while the Ingress state machine required 518 lines of Verilog code, including 21 FSM states.
Chapter 5

Hardware Accelerators for Embedded Vision Systems

This chapter discusses a number of hardware accelerators for embedded vision systems that provide real-time performance. The first part of this chapter presents a hardware-software system for retail environment applications, while the second part discusses the design and implementation of a hardware accelerator for Connected Component Labeling—a computer vision technique that has many applications in embedded vision systems.

High-Performance Hardware Accelerators for Retail Environments

Since the dawn of the Internet, online markets have given a new and personalized experience to consumers around the globe. Electronic shopping cart, history tracking, and product search; are only few features offered by these online markets, and have successfully made a shift in the marketing paradigm. This has forced traditional brick and mortar stores, BMS, to find innovative ways of remaining relevant and profitable. Current BMS face significant challenges related to inventory management which attempts to mitigate profit losses due to out-of-stock and misplaced items. In a typical U.S. Grocery store, out-of-stock conditions result in roughly $800 weekly revenue losses, as highlighted in a study by Gruen & Corsten [69]. Moreover, out-of-stock items contribute to 20% of time wasted by shoppers. The effects of profit loss and shopper dissatisfaction extend beyond retailer and are ultimately felt by product manufacturers and suppliers. The same study indicates that the root causes are primarily due to insufficient management practices of the retail store; such as inadequate forecasts, ordering, and shelves replenishment. Similarly, misplacing items could also lead to losses, where shoppers may not find an item where it should be located, or even worse, perishable items may rot on the shelf.
Nowadays, many stores use planograms to keep track of the products that reside on the shelves and aisles. Planogram are inventory management tools that specify the placement and quantity of products on store shelves. The process of verifying that the current layout of products complies with the Planogram is often referred to as Planogram Compliance Verification, PCV. In existing retail operations, planogram updates are performed infrequently by store staff or inventory specialists. These infrequent updates to the planogram result in retailers being blind to the actual availability of products which leads to shelf depletion scenarios.

A number of approaches have been proposed in an effort to automate PCV. For instance, Decker et al. [70] and Ferreira et al. [71] propose the use of Radio Frequency Identification, RFID, technology to track items on the shelves. Such approaches require tagging all items with a unique RFID tag, and equipping shelves with RFID readers. The infrastructure costs associated with equipping shelves with RFID readers can be significant and infeasible for many retail stores. In addition, tagging millions of products with unique RFID tags may not be a scalable solution, at least in the foreseeable future. In contrast, approaches based on visual analysis do not mandate any infrastructure changes or tagging overhead, and therefore, is a more realistic solution for PCV.

This section presents a hardware accelerated system that provides real-time automated product identification and counting to assist in the PCV process. A conceptual view of the system is depicted in Figure 5-1. In essence, the system uses computer vision to perform scene analytics on the scanned shelves.
Overview of PCV Automation System

There are three activities that contribute to planogram compliance in a retail store: (1) estimation of product quantities on a shelf, (2) detecting the depletion of items, and (3) detection of misplaced items. This section focuses on the hardware acceleration aspects of a system that utilizes computer vision techniques complemented with high-level data analytics to enable automated PCV. To give context to the discussion, the following subsection highlights the complete automated system.

System Components

Figure 5-2 illustrates the automated PCV system consisting of the following components:
1. **Products Database**: This database hosts representative models of each product. Each product is represented by one or more viewpoints in order to increase the likelihood of detection under varying positions and orientations on a store shelf. A small subset of model views is designated as “Markers” and is used as best case reference models for quickly localizing a product group within a shelf image. Figure 5-3 shows samples of the model views for a number of items. This database was populated from actual aisles and shelf imagery taken at a local store.

![Diagram of automation system for PCV](image)

**Figure 5-2.** Automation system for PCV. The system undergoes an initialization phase, denoted by dotted arrows, to extract features from models stored in databases. The memory location/address at which these extracted features are stored is dependent on the position at which these feature were detected. Solid arrows denote the data processing during runtime operation.

![Sample products from the Products Database](image)

**Figure 5-3.** Representative samples from the Products Database. The figure shows multiple views of some of the products. The value in brackets below each product denotes the total number of detected keypoints.
2. **CMOS Image sensors**: The system is equipped with one or more image sensors. These sensors are directed towards the store’s shelves and supplies the system with input imagery for which inventory counting is performed.

3. **Preprocessing Unit**: This unit performs two main tasks: (1) Color space conversion—providing RGB/BGR to gray-scale conversion, and (2) retinal processing on the input frames—providing image-enhanced input to the system. Note that retinal processing can be turned off in environments where variants introduced by factors such as lighting are minimal.

4. **Feature Extraction Unit**: This unit extracts features from the input frame. The Speeded-Up Robust Features, SURF, [14] is employed to extract features from input images. SURF is a feature extractor that detects and describes local features. The algorithm utilizes integral images and fast Hessian filters in order to approximate the 2\textsuperscript{nd} order Gaussian response while detecting key features in an image. This work uses a total of 3 octaves (8 intervals) with a step size of 1. The keypoint descriptor can be either 64 or 128 values; where the latter yields better accuracy at the expense of slower execution time [14].

5. **Vector Match Accelerator**: The matcher performs brute-force matching between keypoints belonging to an item in the product database and the keypoints extracted from the input image. The matching can be applied to the entire features detected in an image, or alternatively on a Region of Interest, ROI, within the image (i.e. a subset of the detected features).

6. **Smart Memory Controller**: The traditional memory controller has been augmented to provide ROI keypoint access, resulting in efficient keypoint fetching and subsequent matching.
7. **Postprocessor:** This stage consists of high-level heuristic driven software modules that operate on the detected items reported by the Vector Matcher. These modules vary in their functionalities and goals. For example, some modules compute inventory statistics of the products that are detected and act as a “bookkeeper” of all reported products. Other modules perform online scene analytics of shelves in order to determine whether a particular item is out-of-stock or if an item is misplaced. Similarly, other modules may operate at the managerial level; sending alerts to suppliers and staff when certain shelves need to be replenished, for example.

**System Operation**

The system operates in two modes; namely, Initialization & Learning Mode, ILM, and Execution Mode, EM. During ILM mode, the system retrieves model views of products and metadata from the Products Database, extracts their features, and stores them in memory. These features are later used by the system when localizing product instances in an image. Note that this mode is performed only once, or when an update is made to the Database. Once the system is initialized, it switches to EM mode. Similar to the ILM mode, features are extracted from the frames captured by the image sensors. However, in this mode, extracted features are supplied to the Vector Matcher, which performs the rapid matching process. The Vector matcher reports identified objects to the Postprocessor to perform further analysis, as described earlier.
**Profiling the Automation System**

The system depicted in Figure 5-2 was entirely developed in software and was executed on a CPU platform. The platform consists of a 2.8 GHz Intel Core Duo T9600 processor, with a total of 8 GB system memory. This platform will be used as a reference platform, and henceforth will be referred to as CPU-Ref.

Using a dataset of images collected from a local store, the execution time of the system was measured. These measurements reveal that both the Feature Extractor and the Vector Matcher units dominate the execution time: 1.21 seconds per frame for feature extraction, and 134 ms per model view for feature matching. Consequently, a pure software implementation of the system is not suitable for real-time performance. Therefore, the following subsections detail the hardware architectures for accelerators that significantly improve the performance of the feature extraction and feature matching stages.

**Hardware Architecture of Accelerators**

This subsection discusses the hardware architecture of the accelerators that are implemented to speed up the PCV automation system. Power efficiency, speed, and configurability are the three attributes that guide the architecture and design choices for these accelerators.

**SURF Accelerator**

Figure 5-4 illustrates the pipelines stages of the SURF accelerator. This accelerator is designed to support the following runtime configurations:
- **Input Image Size**: can support up to 1024×1024.
- **Hessian Threshold**: giving more flexibility in number of detected keypoints.
- **Oriented-SURF**: turning this option off yields Upright-SURF, where orientation-invariance is limited to ±15°.
- **Extended-SURF**: when on, the accelerator generates 128-value descriptors, otherwise, 64-value descriptors are produced.

The SURF accelerator pipeline is composed of the following stages:

1. **IIS**: The Integral Image Stage computes the integral image in a streaming fashion. The Control & Address Generation unit, CAG, is responsible for resetting the accumulation logic and generating the proper addresses to the last-row table. This table stores the computed integral image pixels of the last processed row. The initial latency of this stage is 4 clock cycles but maintains an input/output rate of 1 at steady-state.

2. **FHS**: The Fast Hessian Stage computes both the response and Laplacian of each point in the input image at 8 different intervals, covering the first 3 octaves described in [14]. Due to the memory access irregularity when accessing the

---

Figure 5-4. Pipeline stages of the SURF and Vector Matcher accelerators.
integral image, data reuse is minimal and redundant memory fetches slow down execution. Compared to other stages in the SURF pipeline, FHS is the slowest as pointed out in [72]. There are two intuitive approaches to address this irregularity. The first approach is to implement a zero-latency 2D array of registers that allow filter units to access any integral image pixel arbitrarily. However, the high resource utilization of implementing a large 2D array of registers is prohibitive. For example, if the kernel size of the filter is 100 and integral image pixel width is 28-bits, then 100×100×28 registers are required. The second approach is to store the integral image in a 2D scratchpad memory. While this approach is feasible in terms of resource requirements, the latency incurred due to serialized access to a limited number of scratchpad access ports is, again, prohibitive. Figure 5-5 illustrates these two approaches.

To circumvent these resource and performance bottlenecks, a novel integral image buffer is proposed. This approach is based on the observation that only a small subset of the integral image pixels within the maximum filter window (i.e. 100×100) is needed in any given clock cycle. The buffer is a hybrid combination of SRAM FIFOs and registers that ensures that the minimum subset of pixels

Figure 5-5. Two intuitive approaches to implement the Integral Image cache
(a) The use of 2D zero-latency registers is expensive. (b) The use of 2D scratchpad memory prevents the accelerator from processing data in streaming fashion
required by all filters is available in a zero-latency register. All other local pixels whose values will be required in the short term are maintained in the FIFOs. As the integral image is streamed into the FHS, the integral image buffer rearranges the pixels into a 2D structure. All 8 fast Hessian filters run in parallel, and on each cycle they are granted access to the necessary pixels to compute the Haar wavelet response.

Figure 5-6. The Fast Hessian Accelerator
(a) Integral Image cache is a hybrid of FIFOs and registers. This cache allows accelerator to process incoming data in streaming fashion. (b) The Fast Hessian filters.
The proposed integral image buffer provides an extremely efficient reuse of incoming data, and maintains the streaming nature of the pipeline. Note that there is an initial latency to fill the buffer with pixels, which is proportional to the width of the image, in addition to the latency of computing the response. After that latency, output results are produced at a rate of one response per cycle. Figure 5-6 illustrates the proposed hardware architecture for the FHS stage.

3. **LOCS:** The Localization Stage performs the non-maxima suppression and interest point interpolation immediately on the fast Hessian output in a streaming fashion. The stage utilizes FIFO queues to buffer the incoming responses to exploit data reuse. The non-maxima suppression is performed through a comparator tree, while interpolation logic computes the X, Y position and scale offsets. All octaves are operated on in parallel. This stage is fully pipelined, and only an initial latency is incurred to fill the FIFOs and compute the offsets.

4. **OAS:** The Orientation Assignment Stage computes the dominant angle of the detected interest point. The design supports 32 orientation bins; giving higher precision in computing the angle. Note that when the Upright-SURF option is activated, the hardware will assign the default orientation to the interest point before passing it to the next stage.

5. **EXS:** The feature EXtractor Stage generates the 64-value or 128-value descriptors. This stage employs an image buffer that is used to compute the descriptors to avoid making unnecessary memory fetches. The operations of the OAS and EXS are overlapped to hide their individual latencies.
Vector Matcher Accelerator

The matcher supports high throughput in two modes of operation; namely Marker Detection and Localization. In both modes, instruction-level parallelism is exploited by replication of the distance compute pipelines. In the Marker Detection mode the objective is to visually segment an image of a shelf into product regions by finding the “Marker” for each product type. Task-level parallelism is exploited by enabling a cascade of Vector Matcher accelerators to search for several product markers within a single stream of keypoints extracted from the image. To support matcher cascading, each accelerator allows simultaneous processing and forwarding of image keypoints to downstream accelerators. Note that the Marker Detection mode can occur in parallel with the initial SURF feature extraction process: SURF features are simultaneously forwarded to the matcher to identify markers, and to memory for later use in product localization.

In the Localization mode, a guided search is performed for less prototypical instances of each product type. For each product type, visual search for all viewpoints associated with the type proceeds in multiple horizontal and vertical directions originating from the product marker. Matching is performed iteratively in breadth first fashion with ROI offsets that are consistent with the known dimensions of the product packaging under the viewpoint detected in the previous iteration. Task-level parallelism is exploited by enabling multiple matcher instances to (1) match different viewpoints of a product concurrently, and (2) evaluate neighboring candidate ROIs concurrently. Concurrency is achieved either by operating matchers independently or as a cascade. In the following discussion the term image keypoints refers to the keypoints extracted from the shelf image. Accordingly, model keypoints refers to the keypoints extracted from a specific viewpoint of a product in the Products Database.
Figure 5-4 (bottom right) highlights the Vector Matcher accelerator which consists of an Image Keypoint channel, Model Keypoint channel, and Distance Compute pipelines. The number of Distance Compute pipelines is parameterized with each pipeline sharing the load of computing the squared Euclidean distance between image keypoints and a preloaded set of model keypoints.

The input channels prepare incoming keypoints for either loading the model keypoints or processing the image keypoints. The keypoint converter performs floating-point to fixed-point conversion of the keypoint descriptors. The datawidth adaptor deserializes the incoming keypoint to match the bitwidth of the distance compute pipeline. This allows the keypoint to be efficiently input into the accelerator in a serial fashion while allowing fully parallel computation. Additionally, the image keypoint channel contains a keypoint filter that discards image keypoints that lie outside the bounds of a configurable rectangular ROI.

Each distance compute pipeline includes a Model Keypoint RAM, MK-RAM, to facilitate local pipelined fetching of product keypoints. Prior to performing a matching iteration, model keypoints are distributed into each MK-RAM. The RAM controller maintains the storage such that keypoints belonging to the same class are accessible consecutively without delay. This is useful, for example, when performing SURF feature matching which requires that keypoints share the same Laplacian designation (class) to be comparable. If keypoints are not organized to allow optimal fetching of same-class keypoints, the matching pipeline would stall frequently as the MK-RAM is sequentially indexed and incompatible keypoints unnecessarily fetched and discarded.

The vector distance module is a fully pipelined arithmetic unit that computes the squared Euclidean distance between image and model keypoint descriptors. It consists of a cascade of vector difference, vector squaring, and summation operators. The output of the summation operator is the likeness score between an image and model keypoint pair. Each pipeline is
parallelized to compute a single 64-descriptor keypoint score per cycle. The summation operator allows partial sums to be propagated to vector distance calculators in adjacent distance compute pipelines. This feature allows adjacent pairs of distance compute modules to act as a single 128-descriptor pipeline. As such, the Vector Matcher can be utilized in 64/128-descriptor SURF or 128-descriptor Scale-Invariant Feature Transform, SIFT, [73] applications.

Each distance compute pipeline has a match table that maintains the two best matching image keypoints for each model keypoint. An entry in the match table maintains fields that specify the first-best and second-best matching image keypoints for the model keypoint associated with that entry. The fields include distance score and image keypoint ID for each of the two matches, respectively. For a given keypoint class, scores are generated in identical order across all iterations of image and model keypoint distance calculation. The architecture exploits this observation to reduce the match table indexing scheme: an entry in the match table is fixed and assigned sequentially in the order that the entry is created. Since this order is maintained for the same set of model keypoints, the match table indexing logic is reduced to a simple up-counter. The counter is initialized in consideration of the keypoint class designation to allow different class entries to reside in the same match table without statically pre-partitioning the table. Once the match table has been updated with all distance scores, the contents of the table are egress to the accelerator output. The egress process may optionally filter entries that do not show significant difference between the scores of the first-best and second-best matches. Post-processing software determines the quality of match based on the contents of the match table.

**Experimental Evaluation**

The architecture for the SURF and Vector Matcher accelerators are implemented in synthesizable HDL and are mapped to a DiniGroup DNV6F6PCIe FPGA platform [39]. A
software implementation of the SURF and Vector Matcher, executing on a CPU, is used as a reference to verify the accelerator output and to measure the performance gain.

**Dataset**

To evaluate the accuracy and performance of the proposed system, a dataset of products was compiled from a set of high-resolution still images captured from a local grocery store. The images were originally captured at 5184×3456 resolution (i.e. 18 MegaPixels). These images were later downscaled and cropped to populate the Products Database. This database includes more than 300 different products such as canned food, boxed items, soft drinks, and detergents.

**Experimental Setup**

The accelerators were prototyped on Virtex-6 SX-475T FPGAs [34] operating at 100 MHz. Data are transferred to and from the FPGAs through high-speed PCI Express ×4 Gen 2 interface. The reference software is developed using the OpenCV library [74] and is executed on the CPU-Ref platform.

**Performance**

- **Accuracy:** The product matching accuracy of the hardware accelerators was checked visually against the CPU-Ref and found to be the same.

- **Speed:** Table 5-1 presents the execution time of the accelerators for different image sizes compared to CPU-Ref. At 100 MHz, the SURF accelerator is capable of processing a 1024×1024 frame in 11.15 ms, yielding 108× speedup compared
to the CPU platform. Also, Table 5-1 demonstrates the execution time of the Vector Matcher accelerator compared to a software version running on the CPU-Ref platform.

Table 5-1. A comparison of execution time (ms) between software and proposed SURF and Vector Matcher accelerators (Average keypoints per model is 531, and 5415 keypoints per input frame)

<table>
<thead>
<tr>
<th>Image Size</th>
<th>CPU-Ref (ms)</th>
<th>Accelerator (ms)</th>
<th>Speedup (×)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128×128</td>
<td>19</td>
<td>0.37088</td>
<td>51.2</td>
</tr>
<tr>
<td>256×256</td>
<td>83</td>
<td>0.92896</td>
<td>89.3</td>
</tr>
<tr>
<td>512×512</td>
<td>322</td>
<td>3.02816</td>
<td>106.3</td>
</tr>
<tr>
<td>640×480</td>
<td>354</td>
<td>3.54528</td>
<td>99.9</td>
</tr>
<tr>
<td>800×600</td>
<td>581</td>
<td>5.35648</td>
<td>108.5</td>
</tr>
<tr>
<td>1024×1024</td>
<td>1210</td>
<td>11.15872</td>
<td>108.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># Pipelines</th>
<th>Accelerator (ms)</th>
<th>Speedup (×)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>28.77</td>
<td>4.66</td>
</tr>
<tr>
<td>2</td>
<td>14.39</td>
<td>9.31</td>
</tr>
<tr>
<td>3</td>
<td>9.60</td>
<td>13.97</td>
</tr>
<tr>
<td>4</td>
<td>7.20</td>
<td>18.61</td>
</tr>
<tr>
<td>5</td>
<td>5.76</td>
<td>23.26</td>
</tr>
<tr>
<td>6</td>
<td>4.80</td>
<td>27.90</td>
</tr>
</tbody>
</table>

- **Resource utilization**: Table 5-2 lists the resource requirements when mapping the accelerators to the FPGA platform. With regards to the Vector Matcher accelerator, the number of distance compute pipelines determines the number of keypoints that can be matched simultaneously. As expected, increasing number of pipelines leads to an increase the utilization of resource. BRAM utilization is dominated by the depth of the MK-RAM which are fully parameterized but chosen to be either 320 or 2048 in this study. The size of the MK-RAM
determines the maximum number of model keypoints that can be locally cached in the matcher. An undersized MK-RAM ultimately affects system accuracy as the number of model keypoints have to be truncated for proper operation. However, as shown in Figure 5-3, for this application an MK-RAM depth of around 512 should reasonably tradeoff resource and accuracy.

Table 5-2. Resource utilization, in percentage, for the SURF and Vector Matcher accelerators

<table>
<thead>
<tr>
<th>Stage</th>
<th>Slice Registers</th>
<th>Slice LUTs</th>
<th>Block RAM</th>
<th>DSP48Es</th>
</tr>
</thead>
<tbody>
<tr>
<td>IIS</td>
<td>0.04</td>
<td>0.06</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>FHS</td>
<td>4.59</td>
<td>10.54</td>
<td>9.40</td>
<td>8.73</td>
</tr>
<tr>
<td>LOCS</td>
<td>8.58</td>
<td>10.27</td>
<td>5.08</td>
<td>52.98</td>
</tr>
<tr>
<td>OAS</td>
<td>3.25</td>
<td>13.57</td>
<td>24.62</td>
<td>0.10</td>
</tr>
<tr>
<td>EXS</td>
<td>3.29</td>
<td>24.11</td>
<td>37.69</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Vector Matcher (MK-RAM Depth = 320)

<table>
<thead>
<tr>
<th>No. Pipelines</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.56</td>
<td>5.41</td>
<td>7.26</td>
<td>9.12</td>
<td>10.98</td>
<td>12.83</td>
</tr>
<tr>
<td></td>
<td>6.95</td>
<td>8.66</td>
<td>10.59</td>
<td>12.51</td>
<td>14.40</td>
<td>16.27</td>
</tr>
<tr>
<td></td>
<td>6.95</td>
<td>10.24</td>
<td>13.53</td>
<td>16.82</td>
<td>20.11</td>
<td>23.40</td>
</tr>
<tr>
<td></td>
<td>6.35</td>
<td>12.70</td>
<td>19.05</td>
<td>25.40</td>
<td>31.75</td>
<td>38.10</td>
</tr>
</tbody>
</table>

Vector Matcher (MK-RAM Depth = 2048)

<table>
<thead>
<tr>
<th>No. Pipelines</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.83</td>
<td>5.70</td>
<td>7.56</td>
<td>9.43</td>
<td>11.29</td>
<td>13.16</td>
</tr>
<tr>
<td></td>
<td>7.56</td>
<td>9.53</td>
<td>11.46</td>
<td>13.46</td>
<td>15.31</td>
<td>17.23</td>
</tr>
<tr>
<td></td>
<td>16.17</td>
<td>16.54</td>
<td>22.93</td>
<td>16.82</td>
<td>20.11</td>
<td>23.40</td>
</tr>
<tr>
<td></td>
<td>6.35</td>
<td>12.70</td>
<td>19.05</td>
<td>25.40</td>
<td>31.75</td>
<td>38.10</td>
</tr>
</tbody>
</table>
Discussion of Results

There are several factors that contribute to the high performance exhibited by the accelerators. First, the novel architecture of the FHS stage sustained the desired streaming performance, resulting in a throughput of up to 8 fast Hessian results per cycle. Second, all the Hessian filters are operating on the incoming integral image in parallel, and their results are immediately pipelined to the LOCS stage, which is also operating on the output from all the filters in parallel. Third, the latency of generating the descriptors of the interest points is masked by overlapping the operations of the OAS and EXS stages.

Comparison to Other Approaches

There have been several proposed hardware architectures to accelerate SURF feature extractor [75,76,77,78,72]. In contrast, the proposed SURF accelerator exploits a fully pipelined and streaming architecture that maximizes throughput while minimizing accelerator-to-memory and accelerator-to-CPU communication. The accelerator supports configurable input frame sizes; supports configurable feature descriptor size (64 or 128); and supports normal or Upright-SURF modes of operation.

Lee et al. [76] and Battezzati et al. [72] have proposed two hardware architectures for a brute-force matching accelerator for SURF-based applications. In contrast, the proposed accelerator described in this chapter is capable of matching any of SURF-64, SURF-128, or SIFT-128 features. Moreover, the proposed accelerator supports configurable degrees of both instruction-level and task-level parallelisms. When coupled with a memory controller capable of ROI access, the matcher exhibits high performance and efficiency while performing matching bounded to specific spatial locations.
Performance of the proposed accelerators is compared to the accelerators proposed in [72]. The authors of [72] indicate that the execution time of their proposed SURF accelerator is 3.03ms for an image of 640×480, while operating the FPGAs at 200 MHz. This implementation uses 100 MHz only as a matter of convenience when interfacing system components not related to the accelerator pipelines. Therefore, to highlight raw performance, the results of the proposed system are scaled to be consistent with a 200 MHz operating frequency. It is found that at 200 MHz, the proposed SURF accelerator can process 640×480 image in 1.77ms. Therefore, the proposed SURF accelerator outperforms the accelerator proposed in [72] by 1.71×.

**Scalable Hardware Architecture for Connected Component Labeling**

This section discusses a stream-based hardware architecture for Connected Component Labeling. The architecture implements a scalable processor that can be tuned to match the available I/O bandwidth on the computing platform that hosts the hardware. The architecture is mapped to an embedded prototyping platform and performance is measured.

**Background**

Connected Component Labeling, CCL, [15] is the process of identifying disjoint pixel regions in an image and assigning a unique label to each of these regions. A disjoint pixel region is an image patch in which each foreground pixel has zero or more adjacent foreground pixels. CCL has a wide range of applications in computer vision, including blob detection and tracking. In addition, CCL can be used to obtain some additional information from the connected regions; including: (1) Coordinates of connected region’s bounding box, (2) Coordinates of connected region’s geometric center (i.e. centroid), and (3) Area of connected region.
A number of studies [79, 80, 81, 82] have proposed different approaches to accelerate CCL. However, these approaches impose some input constraints that are unsuitable for real-time systems. For example, Rasquinha et al. [79] describes a parallel hardware implementation of CCL. The design represents a two pass algorithm capable of processing a single 128\times128 image in 900 microseconds. In contrast, Johnston and Bailey [82] implemented streaming architecture based on a single pass labeling algorithm. However, the architecture has the following drawbacks: (1) The pipeline of the architecture is limited to process only a single pixel per cycle, which may restrict the means by which the input can be streamed into the hardware, and (2) The architecture relies on the presence of video blanking periods to perform costly merge operations at the end of each row. While video blanking periods are common to most CCD and CMOS camera interfaces, they do not exist when interfacing directly to a memory subsystem for which data can be accessed back-to-back with negligible delay.

Table 5-3. Determining labels in single pass CCL
The table summarizes how the single pass CCL algorithm determines the label of the current pixel. In the column labeled “Example”, white boxes represent current pixel, while gray boxes represent neighboring pixels.
This section discusses an enhanced version of the single pass algorithm motivated by the drawbacks discussed above. The proposed architecture offers a support for systems that require more diverse pixel input rates, interfaces and protocols.

**Sliced Connected Component Labeling Algorithm**

Rosenfeld et al. [15] describes a simple algorithm for labeling connected components. The algorithm determines the label of the current pixel by looking at its four neighboring pixels; namely, left (L), upper-left (UL), upper (U) and upper-right (UR) neighbors. Table 5-3 summarizes the labeling algorithm.

This subsection describes a Sliced Connected Component Labeling, SCCL, algorithm. The algorithm slices an input image into ‘s’ number of slices, each of which is operated on independently and concurrently.

**Slice and Merge Algorithm**

The SCCL algorithm starts by slicing the input image into ‘s’ slices. Each slice will be processed and assigned labels independent from other slices. Once all slices are processed, the algorithm coalesces the slices and merges any connected regions that extend across slices. Figure 5-7 illustrates an example of how SCCL algorithm works.

As shown in Figure 5-7 (a), the original image contains three connected region; labeled ‘A’, ‘B’ and ‘C’. The SCCL algorithm slices the images into two slices as shown in Figure 5-7 (b). Henceforth, we refer to these slices as top and bottom. The location of a connected region determines if further processing is required. For example, connected region ‘1’ does not lie on either first or last row of the slice, hence, no further processing is required when merging the
slices. On the other hand, labels 2, 3, 4, and 5 are stored for later processing at the merge stage to determine if any of these labels would coalesce. The algorithm establishes an association between the labels from the last row of the top slice and the first row of the bottom slice. It is clear from Figure 5-7 (b) that labels 2, 4, and 5 must coalesce since they stretch across slices. Note that the labels indicated in Figure 5-7 are for illustration purposes only. The actual order of assigning labels is different from the one shown in the figure.

Coalescing is performed in two phases. In the first phase, the association between labels lying at the last row of top slice and the first row of bottom slice are recorded. In the second phase, the recorded associations are resolved. This is accomplished as follows:

- If the connected region does not stretch across slices, then its bounding box is detected and is committed to a table that is referred to as ‘Global Bounding Box’ table.
- If the connected region stretches across slices, then the bounding box of the coalesced region must be updated in the ‘Global Bounding Box’ table.

Although the process described above refers to the example shown in Figure 5-7, yet the

---

**Figure 5-7. SCCL algorithm example with 2 slices.**
(a) Original image contains three connected regions, (b) SCCL slices the image into two slices, each of which is operated on independently, (c) Slices are merged by analyzing the last row of the top slice and the first row of the bottom slice, labels are updated accordingly, (d) Bounding boxes (in red) are detected for the three connected regions.
same process is applied to SCCL algorithm with more than two slices. In this case, the process is repeated for all slice boundaries until all bounding boxes are committed.

The following subsection describes a hardware implementation of the SCCL algorithm.

**SCCL Hardware Architecture**

The SCCL hardware architecture, illustrated in Figure 5-8, is composed of one or more of Connected Component Processors, CCP, and Slice Processors, SP. Also, the architecture contains a coalescing logic and a bounding box FIFO. The number of CCP and SP units is determined by number of image slices that the SCCL can support. For example, the architecture shown in Figure 5-8 supports four slices; hence it houses four CCP and four SP units.

The following subsections discuss the internal implementation of each of these units.

**Connected Components Processor (CCP)**

Each CCP handles a slice that is composed of a range of rows. Consider an input image of size 240×320 that is split into 8 slices. In this case, rows 0 to 29 are handled by CCP₀, rows 30 to 58 by CCP₁, and so on. Figure 5-8 illustrates this setup. The figure shows how each slice is processed by the respective CCP, and the results are fed into the coalescing logic to generate the final bounding boxes.
to 59 are handled by CCP_1, and so on up to CCP_7 which handles rows 210 to 239. Each CCP fetches memory to read rows of its designated slice. Following our example, CCP_0 starts by fetching row 0. Once CCP_0 is done fetching and while it is processing row 0, CCP_1 starts fetching row 30. Similarly, other CCPs will fetch and process the first row of their designated slice. This mechanism allows the architecture to process a row from each slice in parallel. When CCP_7 is done fetching row 210 (i.e. the first row of its designated slice), memory access is arbitrated to CCP_0, where the latter fetches and processes row 1 (i.e. the second row of its designated slice). This process continues until all the CCPs have fully processed all rows in their designated slice.

**Slice Processor (SP)**

When a CCP detects the end of a bounding box, it will send the box coordinates to the corresponding SP. The SP checks if the box boundary lies on the top or bottom of the slice by analyzing the minimum and maximum row coordinates, respectively. If the bounding box does not stretch across either slice’s boundary, then the box coordinates are enqueued in Bounding Box FIFO. This indicates that bounding box requires no further processing. On the other hand, if the bounding box stretches across either the top or bottom slice boundary, then the SP will place the box coordinates and label in the Coalesce Queue. This indicates that further processing is required in the coalescing stage.

The SP maintains a record of labels assigned in the top row of its designated slice. This is necessary because the labels assigned in the top row may merge amongst themselves, which may result in relabeling connected regions as bounding box coordinates are detected. This scenario is illustrated in Figure 5-9.
**Function 1:** Monitor the labels assigned in the first row in the dedicated range of rows and keep a count.

<table>
<thead>
<tr>
<th>COUNT</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Function 2:**
a) Update a stack when labels assigned in the first row of the range merge within themselves and increment temporary count. 
b) Also increment temporary count when a label assigned in the first row finishes as a box.

**Function 3:** Send `stack_entry_done` signal when `Temp_Count` equals `Count`.

**Function 4:** When `stack_entry_done` is asserted, read the stack, and update merge table in the corresponding CCLP until stack gets empty.

**Function 5:** When `end_of_box` signal is asserted, redirect the box labels to appropriate FIFOs.

---

Figure 5-9. Operations performed by the Slice Processor unit.
**Coalescing Unit (CU)**

The CU is responsible for coalescing regions across slices. Figure 5-10 demonstrates the main components of the CU. The unit consists of the Association FIFO, Common Label (CL) RAM0, CL RAM1 and Global Bounding Box RAM. Consider the scenario in which the last row of slice N-1 is completely labeled. This is when the CU begins the coalescing process. As shown in Figure 5-10, when the CU reads the last row of slice N, it coalesces the boxes in slices N-1 and N. At the same time, the unit records an association between the boundary boxes in slice N and the labels in slice N+1. At the end of the coalescing process, the Global Boundary Box RAM is read along with a valid bit to determine all bounding boxes that are not connected any further and are ready to be sent to the next stage of processing. Note that the Association FIFO does not hold entries for those boundary labels that are not connected to any label in the adjoining slice (top or bottom). As a result, the Global Bounding Box RAM does not have entries for these boundary labels.

On the other hand, the Coalesce Queue has labels of all boxes in slice N-1 that lie either

![Figure 5-10. The architecture of the Coalescing Unit.](image-url)
on the top boundary, bottom boundary, or both. The labels in this queue are read one by one and a connectivity check is done to determine if the labels are connected at the top or bottom. If they are unconnected at both ends, the labels are ready to be committed.

**Common Label (CL) RAMs**

The CU hosts two CL RAMs; namely, upper and lower. These CL RAMs store a common reference label – starting from decimal 1 – for the connected labels between any two slices. The entry registers for A0’ and A1’ are retrieved and the subsequent processing is determined by the following algorithm:

1) If both are 0, then a new common label is written both in the upper CL RAM indexed by A0’ and lower CL RAM indexed by A1’. Also, the new common label value is incremented, while the entry registers corresponding to these labels are set to 1.

2) If entry register corresponding to A0’ is 0 and entry register corresponding to A1’ is 1, then this means that the label A1’ in the bottom slice is already connected to some other label in the top slice, however, the label A0’ in the upper slice is not connected to any other label in the bottom slice. Therefore, the corresponding entry registers are validated and the common label value output indexed by A1’ in the lower CL RAM is written into the upper CL RAM indexed by A0’.

3) If entry register corresponding to A0’ is 1 and entry register corresponding to A1’ is 0 then it means the label A0’ in the upper slice is already connected to some other label in the bottom slice but the label A1’ in the bottom slice is not yet connected to any other label in the top slice. Thus, the corresponding entry registers are validated and the common label value output indexed by A0’ in the upper CL RAM is written into the lower CL RAM indexed by A1’.

4) If both entry registers are 1 then both labels A0’ and A1’ are already connected to some other label(s) in the bottom slice and top slice, respectively. A comparison of the output common labels from both CL RAMS is made to ensure they are the same. If they are not the same, then the lower common label has to replace the higher common label. If not, no update is needed.
Note that after processing every boundary, the two CL RAMs are switched. The entry registers of the current upper CL RAM are zeroed (current `upper_entry_register = 0`) and becomes the lower CL RAM while coalescing slices N and N+1. The current lower CL RAM becomes the next upper CL RAM but its entry registers (current `upper_entry_register = previous bottom_entry_register`) are kept as is, to maintain the connectivity history.

**Updating Global Bounding Box RAM**

The Global Bounding Box RAM stores the Bounding box values of the common labels. While reading the CL RAMs, the Bounding Box data tables from the connected regions are also read. A0’ is used to read the bounding box of the upper slice and A1’ is used to read the bounding box of the lower slice. The 2-bit entry is registered, and in the next cycle, the Global Bounding Box RAM is read with the common label as index. Based on the registered 2-bit value, the update of the Global Bounding Box RAM takes place as explained in the following algorithm:

1) If both values are 0, then there is no entry for the new common label. Therefore, the two bounding box values from the connected region are compared and the updated bounding box is found. With the new common label as index, the value is written to the bounding box. When a new common label is assigned, a box counter is incremented to keep track of the number of boxes that have coalesced.

2) If 01, then the values to be compared are the bounding box from top slice connected region and the bounding box from global Bounding Box RAM. The final bounding box is found and, with the common label as index, the Global Bounding Box RAM is updated.

3) If 10, the values to be compared are the bounding box from bottom slice connected region and bounding box from Global Bounding Box RAM. The final bounding box is found and, with the common label as index, the Global Bounding Box RAM is updated.

4) If both values are 11 and the common labels are different, then both values are read from Global Bounding Box RAM and the smaller of the two labels is updated with the final bounding box. A valid bit has to be set for the higher label, which indicates it is not a valid label anymore. Also the box counter has to be decremented.
Bounding Box Update

At the end of the coalescing stage, the Global Bounding Box RAM is updated with a validity bit for each label either set or reset. Before reading the contents of the Global Bounding Box RAM, the Coalesce Queue is read until empty. If the ID read from the queue is unconnected both at the top and the bottom – obtained from the corresponding entry register and bottom_monitor_current register, respectively – then it is committed immediately by fetching the value from the corresponding bounding box table.

Experimental Evaluation

A Verilog implementation of the SCCL architecture was validated on a Xilinx Virtex-5 FPGA development platform. The ML510 [35] development board hosts a Virtex-5 FX130T FPGA that includes a PowerPC processor. Figure 5-11 illustrates the execution time of SCCL for three different image sizes, while increasing the number slices. The figure shows with 6 slices, the SCCL processes 800×600 input frame in 1.73ms.

![Figure 5-11. SCCL execution time as a function of image size](image-url)
Furthermore, the SCCL performance is compared to a C# implementation of multi-pass CCL. The software is running on a 3.18 GHz dual-core Xeon workstation. Figure 5-12 shows that SCCL architecture with 6 slices, running at 100 MHz, outperforms the CPU by up to 10.9×. Note that the results presented here account for the bandwidth and latency characteristics of a realistic embedded system accessing a DDR2 memory device.

Furthermore, SCCL performance was compared against other hardware implementations as indicated in Table 5-4. The table clearly shows how SCCL outperforms other hardware implementation, especially when increasing the number of slice processors.

Table 5-5 lists the resource utilization of the SCCL hardware when synthesized on a Virtex-5 FX130T device for different number of slices. The accelerator consumes around 20% of LUT slices and 7% of BRAM of that available on the FPGA for a design with 2 slices (2 SP + 2 CCP + CU). Synthesis reports show that for every additional slice (SP + CCP), a 4% (3%) additional LUTs (BRAM) is required.

![SCCL speedup](image)

**Figure 5-12. SCCL speedup.**

SCCL with 6 slices outperforms the CPU platform by up to 10.9×.
### Table 5-4. Comparing SCCL with other approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Algorithm</th>
<th>Frequency (MHz)</th>
<th>Image Size</th>
<th>fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rasquinha et al.</td>
<td>Systolic Array</td>
<td>66.67</td>
<td>128×128</td>
<td>~1000</td>
</tr>
<tr>
<td>Jablonski et al.</td>
<td>8-adjacency</td>
<td>60</td>
<td>512×512</td>
<td>~25</td>
</tr>
<tr>
<td>Appiah et al.</td>
<td>Run length</td>
<td>156</td>
<td>640×480</td>
<td>~253</td>
</tr>
<tr>
<td></td>
<td>Iterative</td>
<td>150</td>
<td>640×480</td>
<td>~81</td>
</tr>
<tr>
<td>Proposed</td>
<td>SCCL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Slices 2, 4, 6</td>
<td>100</td>
<td>320×240</td>
<td>~1200, ~2300, ~3000</td>
</tr>
<tr>
<td></td>
<td>Slices 2, 4, 6</td>
<td>100</td>
<td>640×480</td>
<td>~318, ~617, ~880</td>
</tr>
<tr>
<td></td>
<td>Slices 2, 4, 6</td>
<td>100</td>
<td>800×600</td>
<td>~200, ~400, ~580</td>
</tr>
</tbody>
</table>

### Table 5-5. Resource utilization of the SCCL hardware according to number of slices

<table>
<thead>
<tr>
<th># Slices</th>
<th>Block-RAM Count (%)</th>
<th>Slice LUTs Count (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>21 (~7)</td>
<td>15777 (~20)</td>
</tr>
<tr>
<td>4</td>
<td>36 (~13)</td>
<td>22317 (~28)</td>
</tr>
<tr>
<td>6</td>
<td>51 (~17)</td>
<td>28618 (~35)</td>
</tr>
</tbody>
</table>
Chapter 6
Conclusions and Future Work

This dissertation discussed a number of aspects related to the advancement of embedded vision systems. Specifically, the focus of the dissertation is the exploration of means to achieve real-time performance, while maintaining a low power budget.

Chapter 2 presented a framework for hosting hardware accelerators. This framework provides high-speed communication infrastructure, while keeping the framework both flexible and configurable. The framework supports two types of attachments: (1) Stream-Operators, SOP, for stream-based computation, and (2) Switch Attached Processor, SAP, for compute-based processing. It is concluded that a hardware-controlled SAP has several impacts on the flexibility of the SAPs, and the productivity of developers. To address these issues, the SAP was further extended by adding a software-controlled abstraction layer. The SAP Processing Element, SAP-PE, is an accelerator that can be programmed and reprogrammed using ANSI C API. Based on measurements taken from cycle-accurate simulations, we conclude that the slight degradation in the execution time is negligible compared to the benefits obtained from using software to control SAPs.

As an extension to the accelerator framework, it would be interesting to investigate the integration of the programmable framework with OpenCL [85] standard, which may encourage the adoption of the framework by the research community and Industry.

Chapter 3 described the automation tool Cerebrum. The tool enables developers to rapidly create prototypes of embedded vision systems and map them to multi-FPGA systems for validation and performance measurement. Based on the extensive use of the automation tool for
building working prototypes, it is concluded that the tool has dramatically reduced the time taken
to compose these prototypes, and has simplified the process of creating accelerated vision
systems.

_Cerebrum_ can be further extended by embedding FPGA-to-ASIC conversion tool to
facilitate the seamless migration from prototyping stage to actual deployment.

Chapter 4 explored the acceleration of HMAX, a feature extractor that can be used for
multiple recognition tasks. The computational structure of the algorithm was analyzed using a
multi-core CPU platform. It is concluded that although multi-threading can help in improving the
execution time, however, HMAX algorithm still exhibit certain inefficiencies when executed on
traditional general purpose processors. A number of neuromorphic accelerators were developed to
speed up HMAX and were validated on a multi-FPGA platform. Results reveal that accelerated
HMAX is $7.6\times (4.3\times)$ faster than a CPU (GPU) platform, and $12.8\times (9.7\times)$ more power efficient
when compared with the CPU (GPU) platform. Based on the experiments that were conducted on
the accelerated HMAX, we find that specialized memory hierarchies and the use of parallel
resources are the main reasons for the speedup and improved power efficiency. Additionally, a
study of speed-accuracy, and speed-power efficiency tradeoffs was presented in this dissertation.
Finally, we found that the classification accuracies reported for HMAX are not always
competitive with other state-of-the-art algorithms. We conclude that HMAX can be used as a
generic feature extractor for generic workloads, where it can deliver relatively reasonable
accuracy.

For future work, other neuromorphic algorithms (e.g. Saliency [41], CNN [42]) can be
developed and integrated with the described accelerator framework.

Chapter 5 presented three additional accelerators for three algorithms: (1) SURF, (2)
vector matching, and (3) Sliced Connected Component Labeling. Results indicate that the SURF
and Vector Matcher accelerators outperform a CPU platform by $108\times$ and $28\times$, respectively. We
found that instruction-level parallelism, task-level parallelism, and customized memory hierarchies have greatly contributed to the speedup achieved by these accelerators.

It would be interesting to investigate replacing SURF with SIFT and study both the improvement in accuracy as well as the impact on the speedup.

Furthermore, Chapter 5 discussed the Sliced Connected Component Labeling algorithm, SCCL, and described its hardware implementation. The accelerator outperforms a software implementation of CCL by 11×.

The accelerators discussed in this dissertation should not be thought of as isolated hardware components. In fact, these accelerators can work together to tackle certain tasks. For example, consider surveillance and security in airports, where security personnel needs to detect any suspicious behavior. This can be automated using a vision system that employs a skin-tone detector to detect human’s face and body parts. The SCCL accelerator may then operate on the output of the skin-tone detector to identify ROIs and label them. Then, accelerated HMAX can be used to extract features from these regions. Finally, a classifier can be used to recognize the actions as well as facial expressions, and report any abnormalities to the airport security personnel. Similarly, other application domains can benefit from composing and connecting these accelerators as desired.
Bibliography


[33] Xilinx, "Virtex-5 Family Overview," DS100(v5.0) 2009.

[34] Xilinx, "Virtex-6 Family Overview," DS150 (v2.4) 2012.


[69] T. W. Gruen and D. Corsten, "A comprehensive guide to retail out-of-stock reduction in the fast-moving consumer goods industry," University of Colorado at Colorado Springs,


VITA

Ahmed Al Maashri

Ahmed Al Maashri is a Lecturer at the Department of Electrical and Computer Engineering, Sultan Qaboos University, Oman. Ahmed received his B. Eng. in Computer Engineering in 2002 from Sultan Qaboos University. In 2005, he received his Masters in Information Systems (Internetworking) from the University of New South Wales, Sydney, Australia. Currently, he is a Ph.D. candidate at the Pennsylvania State University in the Department of Computer Science and Engineering, under the supervision of Prof. Vijaykrishnan Narayanan. His research is focused on reconfigurable computing, domain-specific acceleration, and embedded systems. While at Penn State, Ahmed has authored and co-authored 2 journal papers, 11 conference papers, and 2 book chapters.

Selected Publications:

Journal Papers:


Conference Papers:


Book Chapters: