IPSJ Transactions on System LSI Design Methodology Vol.5 71–95 (Aug. 2012)

[DOI: 10.2197/ipsjtsldm.5.71]

Invited Paper

System-On-Chip for Biologically Inspired Vision Applications

Sungho Park[1,a)] Ahmed Al Maashri[1] Kevin M. Irick[1]
Aarti Chandrashekhar[1,†1] Matthew Cotter[1] Nandhini Chandramoorthy[1]
Michael Debole[1,†2] Vijaykrishnan Narayanan[1]

Received: March 28, 2012, Released: August 6, 2012

Abstract: Neuromorphic vision algorithms are biologically-inspired computational models of the primate visual pathway. They promise robustness, high accuracy, and high energy efficiency in advanced image processing applications. Despite these potential benefits, the realization of neuromorphic algorithms typically exhibit low performance even when executed on multi-core CPU and GPU platforms. This is due to the disparity in the computational modalities prominent in these algorithms and those modalities most exploited in contemporary computer architectures. In essence, acceleration of neuromorphic algorithms requires adherence to specific computational and communicational requirements. This paper discusses these requirements and proposes a framework for mapping neuromorphic vision applications on a System-on-Chip, SoC. A neuromorphic object detection and recognition on a multi-FPGA platform is presented with performance and power efficiency comparisons to CMP and GPU implementations.

Keywords: neuromorphic vision, system-on-chip, dataflow process networks, visual saliency, object recognition

1. Introduction

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

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

In this work, we present a design platform along with associated design automation tools to facilitate the development of neuromorphic vision systems. First, we focus on the communication architecture required to support streaming data computations in vision applications. Next, we identify and design the computational primitives that mimic the operations of various stages of the visual cortex. We present design automation tools that are key enablers towards composing communication and computation primitives into an SoC. These automation tools give the ability to experiment with various model perturbations and parameter variations. This allows neuroscientists and computer vision experts to explore various models of neuromorphic vision and quickly develop working systems that process live imagery. Finally, we present customized accelerators that implement various stages of the visual cortex—giving support to object detection and recognition applications. Our results indicate that domain-specific accelerators offer a promising approach to bridging the gap between efficiencies of digital hardware vision systems and the brain.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 proposes guidelines to be considered when mapping neuromorphic algorithms to hardware. Section 4 details the implementation of a neuromorphic vision system mapped to a multi-FPGA platform. Section 5 presents experimental results of the system. Section 6 concludes the paper.

[1] Computer Science and Engineering, Pennsylvania State University, University Park, Pennsylvania 16802, USA
[†1] Presently with Intel Corporation in Folsom, CA
[†2] Presently with IBM, System and Technology Group in Poughkeepsie, NY
[a)] szp142@cse.psu.edu

2. Related Works

There has been a thread of research that focuses on identifying characteristics of media applications in general and methods to achieve efficient acceleration of these applications. Numerous academic and commercial systems have been developed as the fruit of these research efforts. The Imagine stream processor [1] adapts the stream programming model by tailoring a bandwidth hierarchy to the demands of the particular media application. By passing a stream from a central register file through an array of 48 32-bit floating-point arithmetic units, it targets the parallelism and locality of such applications. Similar to Ref. [1] Storm-1 [2] defines a stream processor that treats streams and kernels as part of the instruction-set architecture (ISA) to exploit data parallelism and manage on-chip memories. Both approaches map data streams onto many kernels executing concurrently but execution remains inefficient due to load/store instructions that do not perform actual computation.

While Refs. [1], [2] have developed stream processors for bandwidth-efficient media processing utilizing clusters of ALUs that process large data streams, Ref. [3] analyzed and profiled major applications in media processing to reveal performance bottlenecks. Their analysis found that about 80 percent of dynamic instructions were supporting instructions to feed the computational units rather than meaningful computational instructions. In order to overcome such inefficiency, the MediaBreeze architecture was introduced with more customized hardware support for address generation, loop, and data reorganization. In summary, Ref. [3] focused on improving the utilization of computational units rather than increasing the number of these units as done by Refs. [1], [2].

In Ref. [4] the authors detail the implementation of a multi-object recognition processor on an SoC. They present a biologically inspired neural perception engine that exploits analog-based mixed-mode circuits to reduce area and power. Moreover, they utilize a Network-on-Chip, NoC, as the interconnection fabric among all cores. However, except for the visual attention engine and the vector matching processors, all other algorithm acceleration is performed on multiple SIMD processors executing software kernels.

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

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

This work focuses on algorithmic abstractions of the primate visual cortex and the corresponding implementations on digital CMOS substrate. In contrast, other works attempt to mimic the visual cortex using spiking neural networks and artificial synapse implementations [5], [6]. These approaches employ specialized analog circuitry to implement the computational components of the brain model. Unlike digital circuits, these analog systems utilize power transistors and lower supply voltages which results in highly specialized circuits that are extremely sensitive to parameter selection and variation [7]. Since many of the models of the visual cortex are still being discovered and refined, the stringent constraints of analog design do not allow free exploration of model parameters once the substrate has been created. In systems consisting of both analog and digital components, converting stimuli and response between the domains can be extremely challenging and is still an active research area. Compared to pure analog and mixed-signal approaches, our focus is to leverage CMOS technology scaling to circumvent the design challenges while offering a high-degree of exploration and configurability.

3. Design Methodology and Architectural Approaches

3.1 Requirements for Interconnection Network

Support for streaming dataflows is fundamental towards implementing neuromorphic vision algorithms, where most processing procedures have a directly inferable dataflow graph representation. An example of a system-level dataflow graph is shown in **Fig. 1**.

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

3.1.1 Flexibility

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

3.1.2 Scalability

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

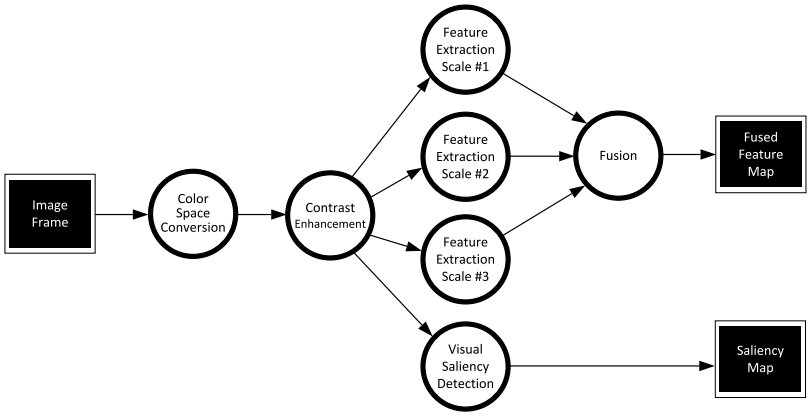


Fig. 1 Example dataflow graph.

3.1.3 Programmability

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

3.1.4 High Performance

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

3.2 Proposed Architectural Approach for Interconnection Network

In consideration of the aforementioned requirements, qualitative comparisons among major communication mechanisms are discussed in Section 3.2.1. Subsequent sections introduce Vortex [8]: a reconfigurable communication platform suitable for composing neuromorphic vision systems.

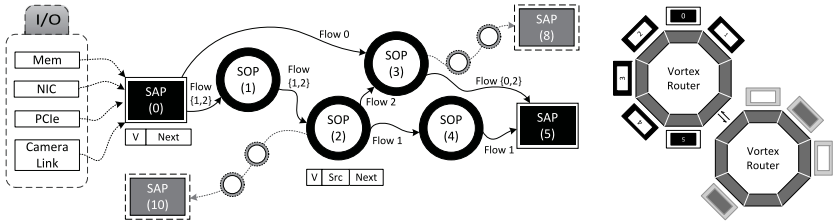
3.2.1 Communication Fabric

Shared buses are common communication mechanisms used in SoC design. ARM's AMBA [9] and IBM's CoreConnect [10] are popular shared-bus architectures deployed in many SoCs. Shared buses moderately satisfy the flexibility requirement by providing uniform master and slave access interfaces to any Intellectual Property, IP, core that conforms to the bus protocol. The dataflow from a source node to a target node can be accomplished by the source node issuing a write into the target node's slave interface. Alternatively, a centralized DMA engine can be tasked with reading data from the source node's slave interface and writing to the target node's slave interface. If the bus architecture supports broadcasting, then source nodes with multiple output channels can send data to multiple targets concurrently. Note, however, that multiple nodes concurrently sourcing a target node with multiple input channels is not supported. This is because shared buses do not allow writing by more than a single device simultaneously and generally limit communication to a single source and target pair at a time. Therefore the shared bus becomes the performance bottleneck in dataflow oriented processing.

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

The NoC paradigm has gained significant attention as the number of heterogeneous cores being integrated on a single SoC increases [11], [12].

By transferring data packets across a system of interconnected switches, the NoC allows communication between all devices



(a) DataFlow graph topology (b) Physical topology

DevID = 0

Flow	V	Next
0	1	3
1	1	1
2	1	1
3	0	

DevID = 1

Flow	V	Src	Next
0	0		
1	1	0	2
2	1	0	2
3	0		

DevID = 2

Flow	V	Src	Next
0	0		
1	1	0	4
2	1	0	3
3	0		

DevID = 3

Flow	V	Src	Next
0	1	0	5
1	0		
2	1	0	5
3	0		

DevID = 4

Flow	V	Src	Next
0	0		
1	1	0	5
2	0		
3	0		

(c) Flow table configuration

Fig. 2 Accelerator system topologies and configurations.

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

3.2.2 Vortex: Reconfigurable NoC Platform for Vision Applications

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

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

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

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

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

The Switch Attached Processor, SAP, is either the source of data streams or the sink of data streams. In addition, SAPs represent computational nodes that need autonomy in initiating transactions. As illustrated in **Fig. 3**, the network interface for an SAP, NIF-SAP, hides the details of accessing high-level application functionality through low-level network protocols. The NIF-SAP has three interfaces to accomplish this: *master interface*, *slave interface*, and *message interface*. The *master interface* allows an SAP to initiate a transaction and provide or receive data directly through a simple FIFO-like handshaking mechanism, or in-

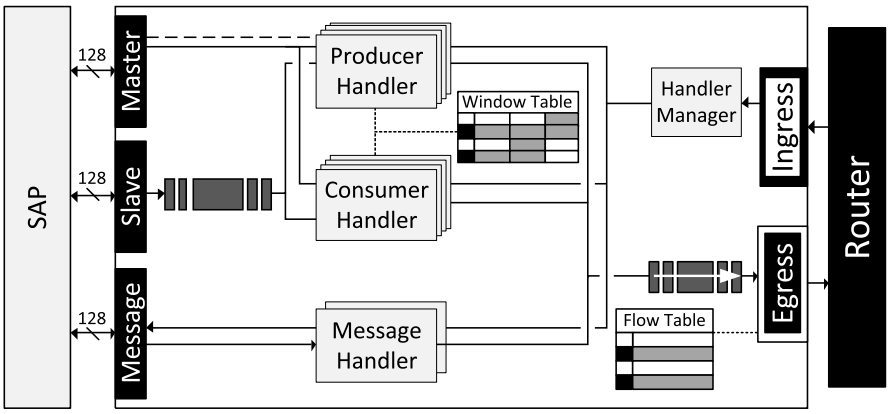


Fig. 3 Diagram of NIF-SAP architecture.

Table 1 Various transaction types.

Master Request Type	Stream Source		Stream Destination	
	Device	Interface	Device	Interface
1. Master Write	Initiator	Master	Target	Slave
2. Master Read	Target	Slave	Initiator	Master
3. Slave Write	Initiator	Slave	Target	Slave
4. Slave Read	Target	Slave	Initiator	Slave
5. Master Write-Lite	Initiator	Master	Target	Slave
6. Master Read-Lite	Target	Slave	Initiator	Master

directly 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.

An NIF-SAP allows an SAP to initiate 6 different types of transactions as listed in **Table 1**. As shown in the table, the Initiator denotes the SAP that initiates a transaction through its *master interface*, while Target denotes the SAP that represents the endpoint of the transaction. The *flow-id* is specified by the Initiator during the request such that the NIF-SAP can establish the virtual connection between the startpoint, endpoint, and all intermediate nodes. The channel setup phase results in an efficient data transfer phase as most of the control information is maintained in the startpoints and endpoints and not contained in the header of each transaction packet.

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

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

A Stream Operator, SOP, is a processing node that operates on data in a streaming fashion. The NIF-SOP provides FIFO-like handshaking interfaces to the input and output channels of an SOP. Along with the simple handshaking signals, the NIF-SOP is equipped with sideband signals to denote the beginning and ending of frames. An SOP with more than a single input/output channel can connect to the network in one of two ways: (1) physically

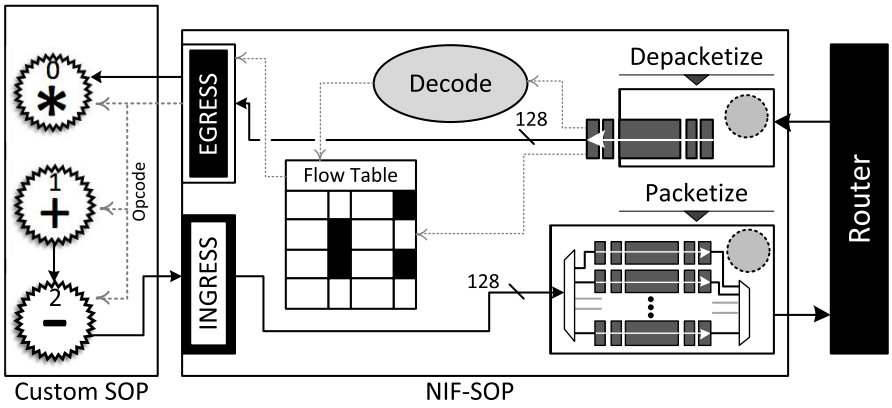


Fig. 4 Diagram of NIF-SOP attached with custom SOP.

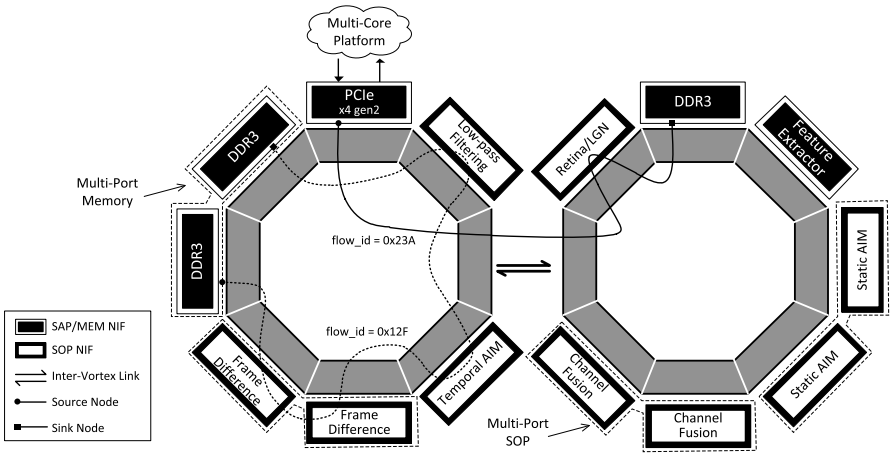


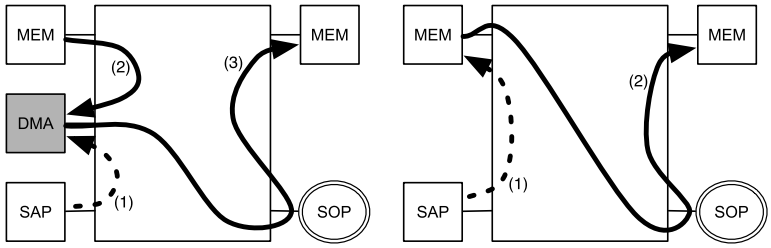
Fig. 5 Example dataflow on Vortex.

through multiple NIF-SOPs or, (2) virtually by time sharing a single NIF-SOP. In the former, scheduling conflicts are minimized since all processing nodes within the SOP can work concurrently in a data driven fashion.

The internal architecture of the NIF-SOP is shown in **Fig. 4**. It consists of three components: a depacketizer that deserializes data from the network for consumption by the SOP; a packetizer that serializes data originating from the SOP for presentation into the network; and a *flow-id* table which decodes an incoming *flow-id* into a local SOP opcode and next-hop *device-id*. Once depacketized, data is streamed to the custom SOP core through the egress interface. The egress interface exposes the data and associated *opcode* through a simple asynchronous handshaking protocol allowing back-pressure to be exerted to the input as necessary. As the SOP processes data, the output is forwarded to the ingress interface and re-packetized for injection into the network. During packetization, the incoming *flow-id* is used to update the packet header with the proper next-hop *device-id*. The NIF-SOP replaces the source *device-id* field in the packet header with the pre-configured *device-id* associated with the incoming *flow-id* as shown in Fig. 2 (c). Collectively the Vortex streaming framework allows maximum parallelism, which is obtained through coarse-grain pipelining across the network in addition to fine-grain pipelining within each SOP.

An example dataflow network on Vortex is illustrated in **Fig. 5**. The network interfaces, i.e., NIF-SAP and NIF-SOP, resolve the 10-bit application *flow-id* into a next-hop physical *device-id* within the network. The *flow-id* is encoded in the header of every packet that belongs to a frame. Since the *flow-id* to *device-id* translation is handled within the network interface, processing nodes are not aware of the context in which their outputs are used. The notion of information hiding is a hallmark of high-level programming paradigms and the key to flexible and scalable application development. Accordingly, the static or runtime configuration of *flow-id* tables within the network interfaces allows arbitrary dataflow graphs to be implemented without consideration by the constituent processing nodes. For SOPs, an additional 16-bit opcode is presented along with incoming data. This opcode is virtually coupled to the *flow-id* associated with the incoming data allowing the processing node to distinguish and offer flow specific variants of operations.

For example, Gabor feature extraction is performed selectively at scales of interest. The finest scale, scale 1, is associated with *opcode* 0x0010 while the coarsest scale, scale 5, is associated with *opcode* 0x0014. Two approaches can be taken for handling the situation of performing fine-grain Gabor feature extraction followed by coarse-grain Gabor feature extraction. The first method involves allocating two *flow-ids*—say 0x2F0 and 0x2F4, respectively—and configuring the NIF-SOP to translate the first and second *flow-id* to *opcodes* 0x0010 and 0x0014, respectively. During system operation, the data is streamed through the Gabor processor using *flow-id* = 0x2F0 and subsequently with *flow-id* = 0x2F4. This static method of configuration is suitable if all modes of operation that will be utilized during system oper-



(a) Without NIF-MEM (b) With NIF-MEM

Fig. 6 Transaction scenarios with and without NIF-MEM.

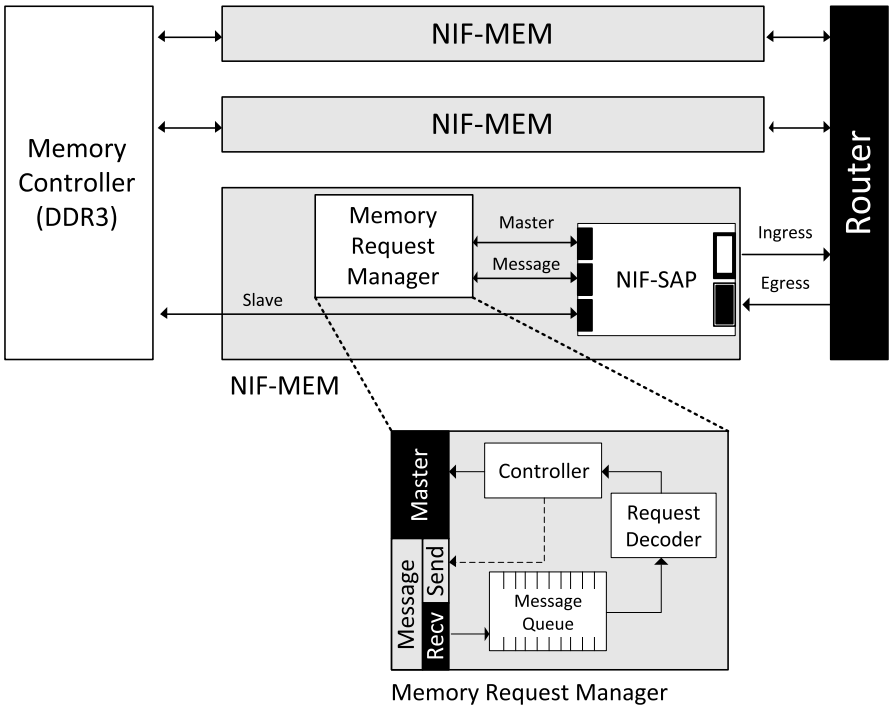


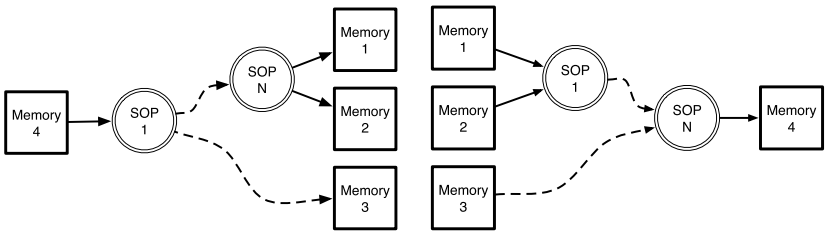
Fig. 7 Diagram of NIF-MEM.

ation are known a priori and there are sufficient *flow-ids* available. In the second approach, a single *flow-id* is allocated and initially associated with *opcode* 0x0010. During system operation, data is first streamed through the Gabor processor using *flow-id* = 0x2F0. Next, the NIF-SOP *flow-id* table is reconfigured to translate *flow-id* = 0x2F0 to *opcode* 0x0014. Finally the data is streamed through the Gabor processor using *flow-id* = 0x2F0. Since resolving *flow-id* to *opcode* occurs once at the beginning of a new frame, reprogramming of the *flow-id* table could be overlapped with data processing as the next translation will be synchronized at the start of the next frame. This runtime configuration method is useful when system behavior is dynamic and not known at system design time.

Memory is treated as a type of SAP, utilizing the NIF-SAP slave interface to expose a globally accessible memory mapped device. The NIF-MEM augments the functionality of the NIF-SAP by expanding the base *message interface* with a message-triggered *request manager*. The *request manager* parses memory-request commands and subsequently initiates slave transactions between its local memory-connected slave interface and the slave interface of any other remote SAP(s) (including the NIF-MEM itself in the case of memory-to-memory copy). If a memory transfer requires barrier synchronization, the request manager sends completion notifications per transaction. Because the DMA functionality is incorporated within the NIF-MEM, network utilization can be reduced as compared to traditional shared DMA architectures that implement a read-store-write style of DMA, as illustrated in **Fig. 6**.

The *request manager* in **Fig. 7** decodes memory-request messages, extracting information including source and target base addresses, transaction length, and *flow-id*. To start the stream, the NIF-MEM initiates a transaction request through its *master interface* to invoke a local memory read. This scenario is depicted in **Fig. 8** (a), and is distinguished as a *write-transaction* because the initiator desires to write a data stream into the network. Figure 8 (b) illustrates a *read-transaction*. The initiator wants to read a data stream from the network. In this case, *Memory 4* is the initiator of the transaction even though it is the endpoint of the data stream. Therefore the memory request message is sent to *Memory 4*.

Notice that the initiator SAP (or memory) is only aware of the *flow-id* and *device-id*(*s*) of the terminating SAPs (or memory) with no knowledge of the intermediate SOPs within the flow. This abstraction hides intermediate data manipulation from the perspective of the startpoint and endpoint SAPs. This reduces the complexity of the SAP behavior and subsequently its program-



(a) Dataflow graph (1-to-N) (b) Dataflow graph (N-to-1)

Fig. 8 Example dataflow graph (1-to-N, N-to-1). Dashed lines indicate presence of intermediary SOPs (not shown in figure).

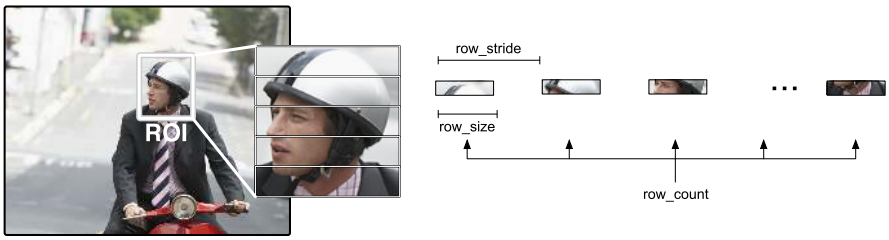


Fig. 9 Illustration of ROI and terms for window transfer.

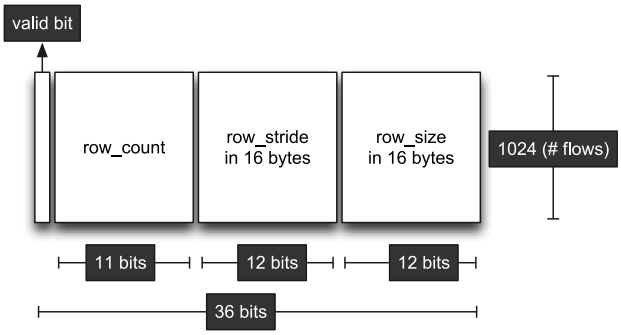


Fig. 10 Window transfer table.

ming. The system level configuration for each device determines the physical path that each stream follows to reach the endpoint. Modifying the path of the data stream does not affect the programs running on startpoint/endpoint SAPs.

It is often necessary in image processing applications to access a subset of a 2D array of data, which is referred to as Region of Interest, or ROI. In many occasions, this may require accessing non-contiguous chunks of the data. One approach to accessing an ROI is to issue multiple transaction requests targeting these chunks of data. The disadvantage of this approach is that network arbitration and packet overheads may degrade performance, especially when these chunks are relatively small in size. Additionally, more complex logic is required to handle out-of-order arrival of these chunks. In contrast, the NIF-SAP supports read and write window transfers to handle ROI access. The NIF-SAP uses a *window descriptor* to specify the details of ROI access. A window descriptor includes row size, row stride, and row count information to describe an access pattern for fetching a rectangular subregion, as illustrated in **Fig. 9**. The NIF-SAP includes a run-time configurable window descriptor table to associate ROI window descriptors to a given flow-id as shown in **Fig. 10**. This region can begin at any offset within a memory-mapped space. When a transaction referencing a window flow is issued, the initiator and target NIF-SAPs transparently fetch and store data according to the access pattern while fully utilizing the payload capacity of each packet. Since window transfers are handled exclusively within the NIF-SAP, intermediary nodes maintain a simplified 1D streaming view of data.

Vortex supports multiple concurrent applications by sharing the NoC fabric. As long as the accumulated bandwidth does not reach the peak bandwidth supported by the infrastructure, multiple flows can share SOPs in a time multiplexed fashion. In order to support such functionality, the NIF-SOP contains multiple output queues in the packetizer, as shown in Fig. 4. The number of queues is parameterizable at design time to trade-off resource consumption and maximum outstanding flows that share the SOP. **Table 2** presents the resource utilization of the Vortex router, when mapped to a Xilinx FPGA device, using various number of bidirectional ports. Similarly, **Table 3** and **Table 4** show the resource utilizations for different configurations of the NIF-SAP and NIF-SOP, respectively.

3.2.3 Meeting Neuromorphic Vision Requirements

(1) Flexibility

The NIF-SAP and NIF-SOP support the connection of computational nodes found in a dataflow representation. Multiple input and output channels are supported through multiple network interfaces connected to a single processing node as shown in **Fig. 11**. This allows for multiple streams to progress concurrently. Vortex also supports a hierarchy of dataflow graphs, which means that an SOP can be composed of smaller recursively-defined operators. To aid in the composition of SOPs, Vortex provides a composition library of modules, namely, input and output port adaptors, inter-operator link modules, address decoders, and hierarchical control units.

(2) Scalability

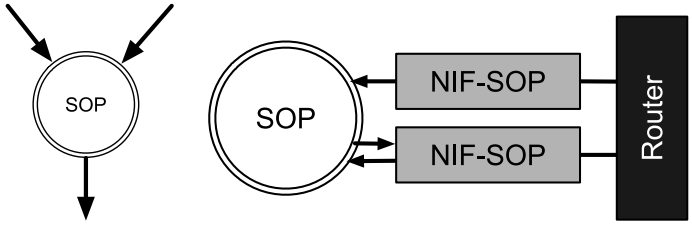
Vortex inherits its scalability attributes from the NoC paradigm. Routers can be cascaded either on-chip or across-chip boundaries via platform specific inter-chip links. By utilizing a 16 bit *device-id*, the system can distinguish 64 k

Table 2 A summary of resource utilization of the router with various number of bidirectional ports on the Xilinx XCV6SX475T FPGA device.

# bidirectional ports	1	2	3	4	5	6	7	8
Slice Register	3,192	5,880	8,335	10,788	13,288	15,796	18,317	37,424
Slice LUT	1,590	3,721	5,626	7,539	10,794	13,025	15,280	15,519
BRAM	9	17	26	34	43	51	60	68

Table 4 A summary of resource utilization of NIF-SOP with various number of output channels on the Xilinx XCV6SX475T FPGA device.

# output channels	1	2	3	4	5	6	7	8
Slice Register	422	478	535	592	652	710	765	821
Slice LUT	485	668	823	909	1,109	1,270	1,470	1,516
BRAM	4	6	8	10	12	14	16	18



(a) DataFlow graph notation of a process (SOP) with two-input channel

(b) Corresponding network attachment with two NIF-SOPs

Fig. 11 Multi-channel SOP.

Table 3 A summary of resource utilization of NIF-SAP with various number of handler-pair on the Xilinx XCV6SX475T FPGA device.

# handler-pair	1 (wo/msg)	1 (w/msg)	2	3	4
Slice Register	2,607	2,980	4,281	5,599	6,685
Slice LUT	4,222	4,932	7,600	10,629	12,404
BRAM	8	12	12	12	12
DSP	2	2	4	6	8

individual processing nodes. Moreover, by adopting a table-based routing scheme at each router, no particular network topology or hierarchy is implied or required.

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

(4) High Performance
The network interfaces utilize a 128-bit flit between the interface and the attached SOP or SAP. Experiments conducted on FPGA prototyping platforms show that the Vortex system easily achieves 200 MHz operating frequency when targeting the lowest speed grade device in the Xilinx Virtex-6 SX FPGA family [13]. Note that the maximum achievable clock-frequency may be significantly higher when targeting an ASIC. Moreover, by supporting independent clocking domains for each SOP and SAP, the maximum operating frequency of a particular node does not affect those of other nodes in the system. In fact, the maximum bandwidth measured on an FPGA emulation platform is 3.2 GB/s. Internally, routers utilize a 256-bit flit size and operate at 400 MHz when targeting the lowest speed grade device in the Xilinx Virtex-6 SX FPGA family. Such a high network capacity ensures that Vortex satisfies the interconnection infrastructure requirements.

3.3 Requirements for Customized Accelerators

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

3.3.1 Exploiting Parallelism

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

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

3.3.2 Power Efficiency

Interestingly, the human brain delivers its massive computing capacity while maintaining a relatively low power budget of roughly 20 Watts. The reasons for such ultra-low power consumption—compiled by Ref. [14]—include sub-threshold

spiking operations, sparse-energy efficient codes for signaling, and proper balance of analog computation and digital signaling.

In contrast, the computational power of the contemporary single core CPU has been limited by the power wall dominated by high dynamic power consumption. While the typical operating frequency remains around 3 GHz, the multi-core CPU paradigm is the current resolution to the power wall dilemma. However, even multi-core CPUs are unable to meet the performance requirements for a number of application classes. As such, domain-specific accelerators are gaining popularity as solutions that deliver high performance within small power envelopes. Domain-specific accelerators go beyond the task-level parallelism exploited by general purpose CPUs, by taking advantage of gate-level customization and parallelism. The result is highly optimized processing pipelines that can be clocked at lower frequencies resulting in high energy efficiency.

3.3.3 Highly Parameterizable Design

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

3.3.4 Composability and Programmability

To maximize component reuse and hierarchical system composition, the most frequently referenced algorithm components are identified and used to populate a library of hardware building blocks. The granularity of these building blocks is multi-tiered: ranging from fine-grain primitives such as adders and subtractors; mid-grain primitives such as convolvers and statistical operators; and macro operators such as retina preprocessors, saliency detectors, and feature extractors. Profiling neuromorphic vision models reveals that many of the algorithms can be mapped to a streaming mode of processing. Stream processing is benefited by a minimal requirement for storage and control as these aspects are implicit in the dataflow nature of the processing. For these reasons, all operations that can be mapped to a streaming modality are considered for hardware implementation. To compose these streaming algorithms, accelerators are constructed from a set of streaming operators mapped to a dataflow process network [15]. The nodes of this process network are realized by specific streaming operators while the dataflow paths are realized by the Vortex interconnection system of on-chip routers and communication channels.

Still, there is a significant number of algorithms that exhibit either non-streaming characteristics or a hybrid of streaming and non-streaming characteristics. Iterative control constructs, complex state transitions, and arbitrary memory utilization and accesses present in these algorithms are not effectively mapped to a purely streamed processing architecture. Consequently, the dataflow process network is augmented to include processing elements that maintain a Von Neumann model of sequentially executed instruction streams. The specific definitions of these instructions are reserved to the implementation of each processing element. In this way each class of processing element has a unique ISA for which the associated processing architecture is optimized to execute in an accelerated fashion. Instruction streams are executed concurrently by any number of processing elements, each scheduling arbitrary process flows across streaming accelerators as necessary. Therefore system composition is defined by three aspects: the static allocation of SIMD stream accelerators; the static allocation of custom ISA processors; and the orchestration of highly temporal control and virtual interconnection of both. In this way, complex notions of iteration and functional composition can be described on a static network of accelerator resources.

3.4 Architectural Details of Customized Accelerators

3.4.1 SAP Processing Element (PE)

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

To address the issues presented above, it is imperative to add an additional layer of abstraction to the SAP. This layer of abstraction serves the following purposes:

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

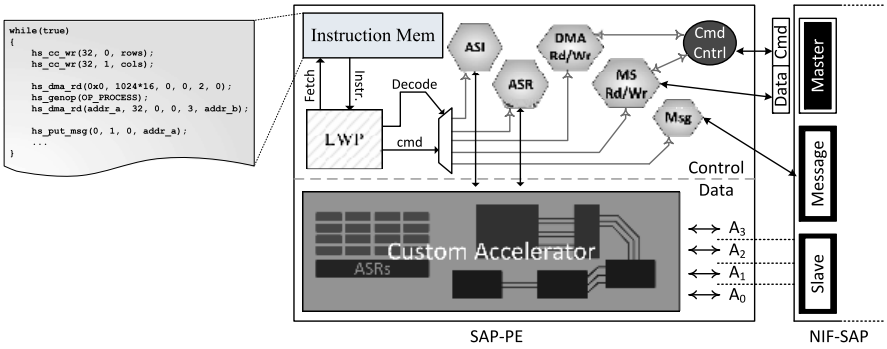


Fig. 12 SAP-PE μArchitecture. The architecture is split into two path; control and data. The control path abstracts the underlying hardware complexities and exposes a set of APIs for the user to control the accelerator's operations. The data path is where the custom accelerator resides.

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

Henceforth, the acronym SAP-PE is used to refer to the SAP accelerator combined with the additional layer of abstraction described above.

Figure 12 illustrates the architecture of the SAP-PE. The architecture is partitioned into a control path, providing instruction-based control over the movement of data and the custom accelerator, and a data path, consisting of the implementation of the custom accelerator logic and functions. In the control path, the main driver of operation is the *Light-Weight Processor*, or LWP. The LWP provides several standard mechanisms for control such as branching, looping, and basic ALU functions for simple address manipulation (e.g., addition, subtraction, shifting, etc...). The LWP is deliberately void of complex arithmetic logic as the majority of the computation is intended for the custom accelerator hardware, rather than an instruction based processor.

The LWP fetches instructions from a 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 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 pause 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.

On the other hand, the SAP-PE data path, illustrated in Fig. 12, is made up of the custom accelerator hardware and is directly controlled through specific instructions issued by the control path. Data is transferred to the custom logic, directly through the NIF-SAP slave interface, using the DMA transfer instructions described above.

3.4.2 Composability (ChipMONK for SOPs)

Figure 13 highlights the hierarchical streaming accelerator structure. The root of the hierarchy is the streaming accelerator. The accelerator is composed of one or more primitive streaming building blocks. These building blocks include convolution engines, arithmetic operators, statistical operators, transform operators, and non-linear function approximation modules. Collectively, the primitive building blocks are referred to as the Base Library of Operators. The interconnectivity, programmability, and control of these primitives are supported by a set of utility components that include input and output port adaptors, inter-operator link modules, address decoders, and hierarchical control units. An accelerator may support many modes of operation that may differ in runtime parameters or to some degree sequence of computation. For example, the Retina/LGN preprocessing accelerator can operate in Retina, LGN, and Retina-LGN modes. Each of these modes is different in the way data flows through the pipeline and can be chosen dynamically. To support rapid algorithm exploration, an automation framework called ChipMONK is utilized to combine computational primitives and utility primitives to construct SOPs. ChipMONK performs the tasks of interconnecting primitives, resolving bit-width configurations, calculating and allocating appropriate buffering, address space partitioning, and control state machine synthesis.

3.5 The Realization of the Neuromorphic SoC

In order to validate accelerators mapped to the SoC framework, a multi-FPGA System, MFS, is used as a prototyping platform.

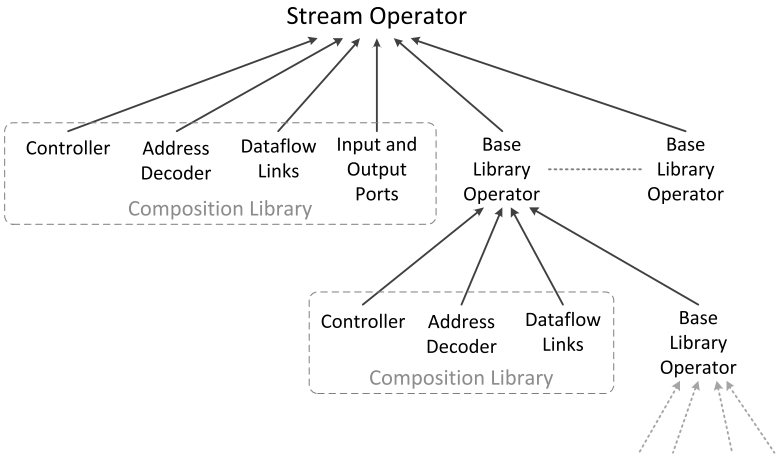


Fig. 13 Hierarchical composition of an SOP.

However, designing a hardware architecture that targets an MFS requires considerable skill and expertise. Moreover, HDL development becomes laborious and error prone as the target systems grows in size and complexity. Hence emerges the need for design automation tools that assist users in building their design easily and efficiently—saving them both time and effort. This section introduces *Cerebrum*, a software tool for automating the process of mapping hardware designs to an MFS. *Cerebrum* abstracts the details of RTL coding as well as communication and memory hierarchy partitioning. Moreover, it automates execution of the synthesis and implementation phases of the design process.

3.5.1 Automation Tool (Cerebrum) [16]

In the last decade, several academic and commercial tools have appeared that seek to reduce the skill and expertise required for HDL system development. Neely et al. [17] discuss three categories of tools used for accelerating the FPGA design process. The first category of tools aims at reducing non-recurring engineering (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, 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 [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 categories 1 and 2. Cong et al. [22] describes 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.

Cerebrum was developed to allow users with little or no knowledge of hardware and RTL development (e.g., neuroscientists and researchers) to compose accelerators for various cortical vision algorithms with minimal effort. The tool standardizes MFS specifications and uses high-level meta-data to deliver an application-level design experience to the user. *Cerebrum* includes an IP-based, multi-FPGA mapping algorithm designed to optimally allocate IP components according to resource use, connectivity, and I/O requirements. The following discussion details the *Cerebrum* software architecture: partitioned into front end GUI and the back end EDA components. **Figure 14** illustrates the *Cerebrum* design flow and the interactions between the front end and back end.

3.5.2 *Cerebrum* Front End (GUI)

The front end of *Cerebrum* offers the users a graphical interface for composing a system and automating the back end 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 most important being the port interfaces. The ports are the interfaces to the physical core, initiator/target (compute), input/output (streaming). Ports define how cores can be validly interconnected (for example, a target can only be connected to an initiator or output).

The ⟨*Hardware*⟩ section of the "IP Core Specification" file, 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 infrastructure. The *pcore set* describes the library components that make up the core. The *clocks* subsection specifies required clock attributes such as frequency and phase.

To create a system, the user drags-n-drops compute-based and

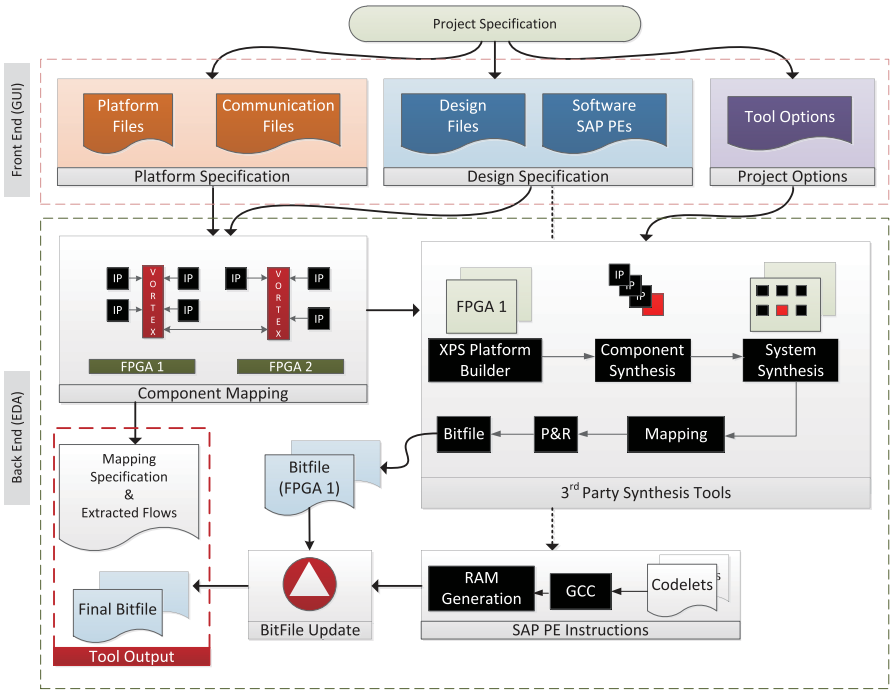


Fig. 14 The front-end (GUI) and back-end (EDA) of the *Cerebrum* tool.

stream oriented cores onto the design canvas and connect 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 instruction set extension APIs that are specified along with the accelerator. Stream oriented modules process data as it streams between compute-based cores. Stream oriented modules may be chained together allowing for scalable stream processing.

The *Cerebrum* front end allows reprogramming stream dataflows and modifying code that executes on compute-based modules. This is accomplished by rewriting *codelets* which does not trigger system synthesis.

3.5.3 *Cerebrum* Back End (EDA)

The *Cerebrum* back end performs the following tasks:

- Mapping IP cores to an MFS
- Invoking 3rd party tools for synthesizing the cores
- *Codelet* compilation and merging executable with hardware configuration files

The *Cerebrum* back end uses a number of specification files to accomplish these tasks. The specification files belong to one of three categories. The first category, *platform specification*, includes XML files that define I/O, resources, interconnections, and required interfaces of the target platform. The *design specification* category includes XML files which describe the IP cores, their interconnections, and any design parameters. Finally, the *project options* category consists of files that specify back end tool options.

The back end uses an in-house-developed multi-FPGA accelerator-mapping algorithm that automatically places IP cores onto the FPGAs and generates the communication network.

3.5.4 System Synthesis

Cerebrum supports a number of FPGA platforms and devices. It has been used to create systems targeting the Xilinx Virtex 5 and Virtex 6 devices [13], [24]: including ML510 [25], Nallatech [26], ML505 [27], ML605 [28] and DiniGroup [29] development systems. Note that although all listed devices are Xilinx based, *Cerebrum* is not restricted to a particular vendor. In fact, *Cerebrum* allows the user—through custom-user scripts—to specify details of a 3rd party software used for system synthesis and implementation.

4. Case Study

Figure 15 illustrates a neuromorphic system capable of performing object detection and recognition. The system receives imagery from an attached streaming device (e.g., a camera). The image enhancement block performs contrast enhancement and eliminates undesired common illumination. The enhanced image is stored in memory and forwarded to the saliency detector, which identifies salient regions in the image. The saliency detector communicates the coordinates of ROI to the feature extraction block. The feature extractor uses the coordinates to request, from memory, the salient regions of the enhanced image. The feature extraction block produces a feature vector that is representative of the processed ROI. Finally, a trained classifier classifies the ROI using the feature vector produced by the feature extractor. This neuromorphic system is realized on a Multi-FPGA system as depicted in **Fig. 16**. The following subsections discuss each of these processes in detail and highlight the accelerator architectures that allow execution in real-time.

4.1 Retina Preprocessing

In the human visual system, the retina performs preconditioning of imagery for high-level vision tasks such as attention and object recognition [30], [31], [32]. In the visual pathway, the retina consists of photoreceptive cells—rods and cones—that perform the transduction of light striking the eye to neural impulses

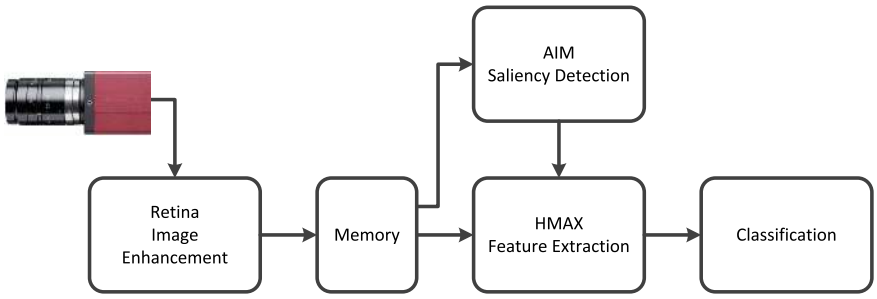


Fig. 15 A neuromorphic system for visual processing. The system pre-processes the input images for contrast enhancement. The system operates on the enhanced input to detect salient objects and classify them.

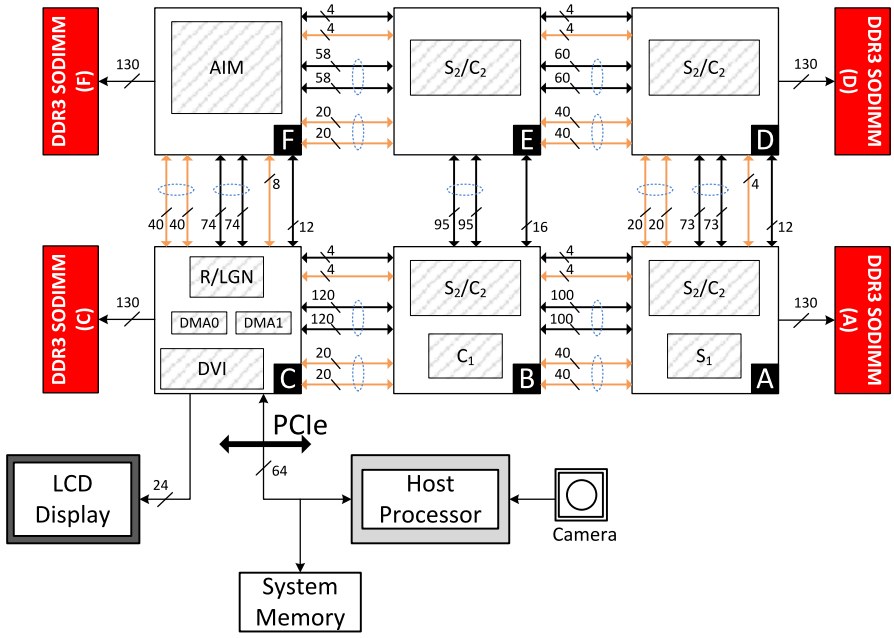


Fig. 16 Visual processing system mapped to a multi-FPGA prototyping platform. Images are captured from an HD camera attached to the host system. The PCI Express interface is used to transfer imagery to the neuromorphic accelerators. The output is displayed on an LCD monitor interfaced to the platform.

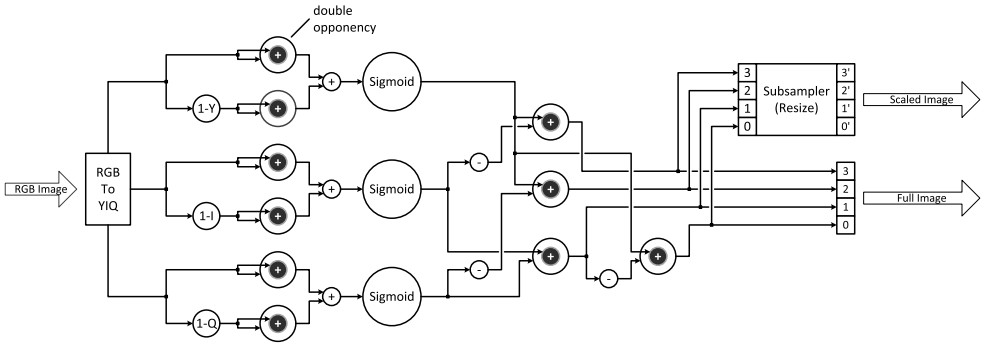


Fig. 17 The retina processor is a completely streaming pipeline implementation. The retina generates a full resolution enhanced image in addition to a scaled image via the subsampler. The primary components are the double opponency operator, sigmoid operator, and subsampler. Each of the operators are members of the Base Library of Operators and are composed of other primitives in the library.

that are forwarded through the optic nerve for subsequent processing in the visual cortex. These neural responses are a function of the competing interactions between stimuli generated by spatially co-located rods and cones. A key artifact of these inhibitory interactions is the enhancement of contrasting structures within the visual field. Ultimately these peak responses become the primary features for perception.

A streaming retina processor was implemented utilizing the SOP composition methodology, discussed in Section 3.4.2, and is illustrated in **Fig. 17**. The input to the retina preprocessor is the YIQ color space representation of the original RGB image. Each of Y, I, Q, Negated I, and Negated Q are extracted as independent image channels. The first stage of retina processing performs contrast enhancement, normalization, common illumi-

nant removal, and dynamic range compression on each channel independently. In traditional machine vision applications, this process is typically performed by Histogram Equalization. Histogram Equalization is generally performed using statistics computed globally across the entire image and therefore is less effective in enhancing local image contrast and tends to artificially introduce a global image bias. The retina preprocessor uses a model of center-surround competition within pixel neighborhoods to enhance contrast locally while removing common illuminants [33]. The subsequent image is normalized and its dynamic range compressed using a sigmoid operator that is adaptive to the global image statistics.

The second role of retina processing is to fuse the responses of independent channels in a way that either extracts complementary information (decorrelation) or enhances channel similarities [34]. This inter-channel fusion is also performed using a model of center-surround competition in a fashion similar to the color opponent process between rods and cones in the human eye. The retina processor produces four channels that represent the cross channel fusion of the contrast enhanced and normalized Y, I, Q, Negated I, and Negated Q channels. These channels are used in the subsequent saliency and feature extraction processing stages.

The common operation across the two functions of the retina is an operator referred to as double opponency. As the name suggests, double opponency performs the center-surround opponent computation between two input channels. Internally, the double opponency operator performs Difference of Gaussian, DoG, between its center and surround inputs followed by adaptive dynamic range compression using a non-linear sigmoid-like transform. In the case of channel enhancement, a single channel is replicated and presented to the center and surround inputs identically. This configuration is appropriate because the intent is to enhance pixel values that are local maxima within the neighborhood of the given pixel (intra-channel enhancement). In the case of channel fusion the intent is to either extract or enhance the responses of different channels. Therefore, the center and surround input to the double opponency are the two channels to be considered. Because of its high frequency of use and its streaming nature the double opponency process is realized as a hardware accelerated component in the SOP Base Library of Operators.

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

In the second stage, the contrast enhanced image is remapped to the full output range (0 to 65,535). The non-linear remapping is performed by a sigmoid function approximation modulated by the global mean of the enhanced image. The sigmoid operator is used frequently and so it too is realized as a hardware accelerated component in the SOP Base Library of Operators.

Table 5 A summary of resource utilization for neuromorphic retina processor on XCV6SX475T.

	Slice Register	Slice LUT	BRAM	DSP
Retina Resource	158,568	133,342	434	390

The sigmoid operator was first created using operators within the Base Library of Operators including mean and standard-deviation operator and a generic function approximation module for computing $Logist(x)$. The sigmoid operator, along with other components, was then utilized to build the double opponency operator. Finally, the retina processor was implemented using the double opponency, sigmoid, and image subsampler operators as shown in Fig. 17. The subsampler resizes each output channel by a configurable factor in both image width and height. The retina outputs both the full resolution and scaled images for processing by the feature extractor and saliency detector, respectively.

Since retina processing is the first step in the visual pipeline and because it is implemented as a streaming operator, it is configured to receive images directly from the PCI Express interface: alleviating the need to buffer incoming images in onboard memory. The host system captures images from a high-resolution GigE camera and buffers it in the host's physical memory. To process an image frame, the host sends a read request message to the host-to-board DMA engine on FPGA 'C' (See Fig. 16). The request specifies the source physical memory address and *flow-id*: the target addresses within DRAM D and DRAM F are pre-configured and associated with the *flow-id*. The DMA engine subsequently begins fetching data from the image buffer in host physical memory and injects it into the network targeting the retina processor. Consequently, no additional latency is incurred for onboard buffering as the retina processing is overlapped with the transfer of data from system memory. The full and scaled images are injected into the network utilizing dedicated NIF-SOP interfaces taking independent paths to DRAMs D and F, respectively.

The resource utilization of the neuromorphic retina processor on the prototype platform is presented in **Table 5**.

While CPUs and GPUs operate on 8-, 16-, 32-, and 64-bit variables, FPGA and ASIC fabrics support arbitrary bit-widths for each variable in the design. By adjusting the bit widths according to the precision requirements, significant reduction in the silicon area cost of arithmetic units and bandwidth requirement between different hardware modules can be achieved. The image input to the retina utilizes a 0:1:15 fixed-point format. Intermediate representations within the pipeline are set appropriately by ChipMONK by performing static fixed-point bit allocation. Ultimately, the difference between the fixed-point implementation and the double-precision equivalent implementation was in the range of 10^{-5} to 10^{-6}.

4.2 Visual Saliency (AIM)

The attention mechanism of the human vision system allows the brain's perceptual and cognitive resources to be focused on the most important regions in the visual field. Otherwise, the cognitive system would be overwhelmed by the tremendous amount of visual information arriving from the optic nerve. In synthetic

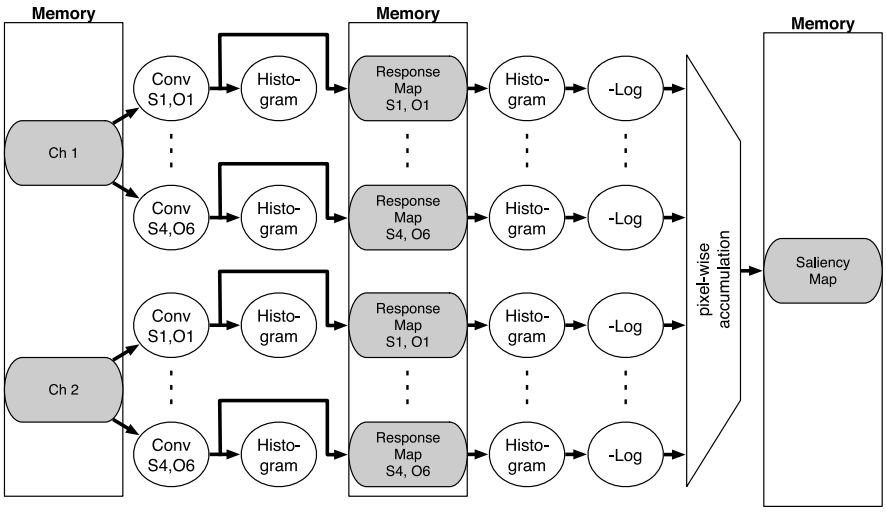


Fig. 18 Dataflow graph representation of AIM.

Table 6 Comparisons among architectural approaches to implement AIM.

	Resource consumption	Latency (ms)	Frame rate (fps)
Fully parallel	2,800	3.9	256
A channel per iteration	1,400	5.9	169.5
A scale per iteration	972	17.7	56.5
An orientation per iteration	424	25.6	39.1
Fully iterative	162	49.2	20.3

Resource consumption is represented by number of multiplications in convolutions.
Latency and frame rate is based on following conditions:
Input Frame resolution = 2,048 × 1,536.
Sub-sampled Retina/LGN output resolution = 512 × 384.
Operating clock frequency = 100 MHz.

vision systems, this attention mechanism is termed Saliency detection. The process guides the application of object recognition to relatively small subsets of the image field, maximizing system efficiency in terms of processing latency and resource utilization.

A variant of the Attention by Information Maximization, AIM, algorithm proposed by Neil Bruce and John Tsotsos [35] is implemented in the vision processing system. The premise of the algorithm is that the most salient areas are those that have the most information content. The algorithm operates in two phases. In the first phase, AIM transforms the input into an image in which each pixel is an n-dimensional vector. Each element i in the vector is a coefficient representing the contribution of the i^{th} basis vector in an orthonormal basis. The basis or feature space is either learned by an Independent Component Analysis, ICA, process or explicitly defined by a filter such as the Gabor kernel. Two of the four scaled channels (e.g., 0′,1′,2′, or 3′) originating from the retina preprocessor are dynamically selected to generate 48 independent feature maps, or response maps, by employing two-dimensional complex convolutions with four scales and six orientations of the Gabor kernel.

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

Figure 18 shows the dataflow of the visual saliency model. Note that indexing a histogram is only possible after the histogram has been constructed from the entire response map. Therefore the dataflow graph is split representing the two phases of the algorithm. Implementing each in a fully parallelized fashion achieves latency proportional to $2|I|$ where $|I|$ is the number of pixels in the image. If the saliency processor operates at 100 MHz on 512 × 384 imagery, the total latency is roughly 4 ms. The pipeline, however, consumes approximately 2,800 multiplier resources which may not be feasible for smaller platforms. In terms of satisfying real-time constraints of 30 fps, the fully parallel pipeline far exceeds the requirements. There certainly exists a more balanced trade-off between resource consumption and latency. **Table 6** shows several architectural approaches with estimated latency and resource consumption in number of multiplications.

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

The prototyped implementation of AIM is configured to process four scales concurrently across two channels and six orientations iteratively. **Figure 19** highlights the methods of concurrency and iteration in the adopted implementation.

The architecture is partitioned into two pipelines that operate concurrently: the build pipeline for computing the coefficient density and the index pipeline for computing the self-information map. Each response map generated by each of the four Gabor convolvers must be propagated to both the associated histogram, for histogram construction, and to external memory, for recall during histogram indexing. Under these conditions a total of four memory transactions are active simultaneously: fetching the current channel, storing the current output of the four Gabor convolvers, fetching the previous output of the four Gabor convolvers, and storing the current partial saliency map. Assuming 512×384 imagery with 64-bit input representation and 16-bit response map representation, the saliency processor demands roughly 1.7 GB/s of memory bandwidth to maintain 30 fps throughput. This constitutes 20% of the peak bandwidth of DDR3-1066. Given that the multiplier resources are relatively low, the memory and network bandwidths can be reduced dramatically at the expense of doubling the number of multiplier resources as shown in **Fig. 20**.

DRAM F, in Fig. 16, contains the sub-sampled version of the preprocessed image originating from the retina preprocessor. Each of the four channels has a 16-bit pixel representation. Four pixels are packed into a single 64-bit data flit. For each of the iterations, the AIM processor selects the appropriate channel from the incoming data stream and forwards it to the build and index pipelines. By duplicating the four Gabor convolvers in the index pipeline, the response map is recomputed on-demand obviating the need for storing intermediate maps in external memory. In other words, instead of burdening the network and memory system by storing and fetching the intermediate response maps, the architecture calculates the response a second time for indexing the histogram. The calculation of the previous response map for indexing is overlapped with the calculation of the current response map for building. The Dual-Histogram stream operator allows concurrent building and indexing of two internal, independent and alternating, histogram tables. The schedule of the pipeline is shown in **Table 7**.

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

The host system orchestrates the iterative saliency process by issuing 12 memory-to-memory transaction requests. The source memory is the location of the sub-sampled retina output while the target memory is the location at which the saliency map will be stored. Note that the target memory is only updated after the 12^{th} iteration when the final saliency map is being computed. Each transaction references a unique *opcode* that allows the pipeline to distinguish which channel and orientation combination to be processed for the given transaction. Once the saliency map has been computed and stored into the host memory, a host software process performs ROI extraction using a connected components algorithm. The coordinates of the ROI within the full scale preprocessed image are forwarded to the feature extractor. **Table 8** shows the resource utilization for the AIM accelerator

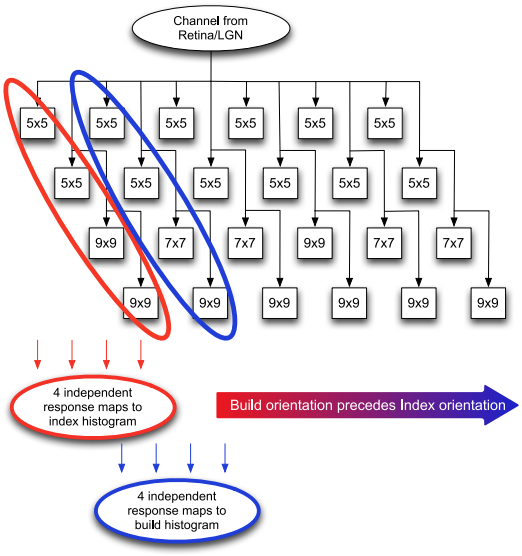


Fig. 19 Scale-Concurrent AIM architecture.

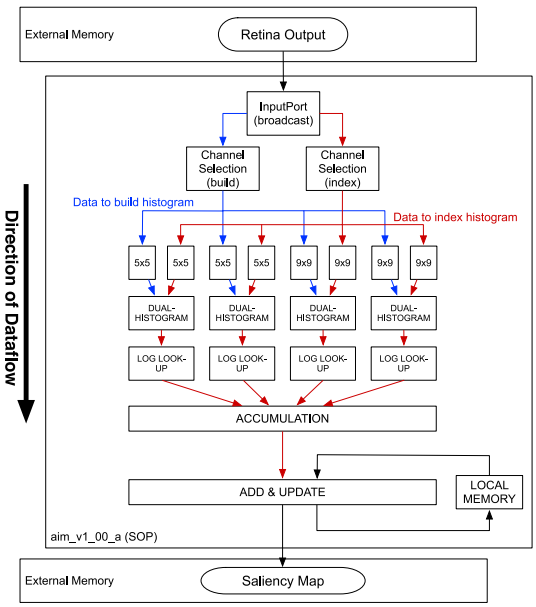


Fig. 20 Block Diagram of AIM architecture.

Table 7 Schedule of timeline to operate AIM.

	(Channel #, Orientation #) for each iteration												
Build	1,1	1,2	1,3	1,4	1,5	1,6	2,1	2,2	2,3	2,4	2,5	2,6	—
Index	—	1,1	1,2	1,3	1,4	1,5	1,6	2,1	2,2	2,3	2,4	2,5	2,6

when mapped a Virtex 6 FPGA device.

4.3 Feature Extraction (HMAX)

HMAX ("Hierarchical Model and X") [36], [37] is a model of the ventral visual pathway from the visual cortex to the inferotemporal cortex, IT. This model attempts to provide space and scale invariant object recognition by building complex features from a set of simple features in a hierarchical fashion.

Figure 21 shows a computational template of HMAX. The model primarily consists of two distinct types of computations, convolution and pooling (non-linear subsampling), corresponding to the Simple, S, and Complex, C, cell types found in the visual cortex. The first S-layer, S_1, is comprised of fixed, simple-tuning cells, represented as oriented Gabors. Following the S_1 layer, the remaining layers alternate between max-pooling layers and template-matching layers tuned by a dictionary encompassing patterns representative of the categorization task.

The exact implementation of HMAX is determined from what is considered to be the most biologically plausible. This paper uses a specific implementation for the object recognition task developed by Mutch and Lowe [38], as it represents the current understanding of the ventral stream and produces good results when used for classification. This model is represented by a total of five layers, an image layer and four layers corresponding to the alternating S and C units.

Image layer: This layer is used for preprocessing the image to ensure the uniformity of inputs. First, the image is converted to grayscale and then pixel values are normalized to the range [0, 1]. However, since the full-resolution output of the retina preprocessor conforms to this specific format, this step can be omitted. Then, the input image is downsampled to create an image pyramid of 12 scales, with the largest scale being 256×256. 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.

S_1 (Gabor filter) layer: The S_1 layer corresponds to the V1 simple cells and is computed by performing a convolution with a set of orientations at each position and scale. The number of orientations used in this model is 12, producing 12 outputs per scale and of equivalent size (for a total of 144 outputs). The Gabor filters are 11×11 in size and are described by:

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

where $X = xcos(\theta) + ysin(\theta)$ and $Y = -xsin(\theta) + ycos(\theta)$. The model follows [39] and varies x and y between -5 and 5, and θ between 0 and π, while the wavelength(λ), width(σ) and aspect ratio(γ) are 5.6, 4.6, and 0.3, respectively.

C_1 (local invariance) layer: The C_1 layer provides a model for the V1 complex cells and pools over nearby S_1 units (within the same orientation). Within a scale, each orientation is convolved with a 3D max filter of size $10 \times 10 \times 2$ (10×10 units across in position and 2 in scale). This layer provides scale invariance over large local regions and reduces the number of units used as input to the next layer, as it acts to non-linearly subsample the S_1 output. Due to the cross-scale pooling, the result of this stage is 11 scales $\times$12 orientations structure. However, the output of C_1 stage exhibits smaller spatial extent than the original S_1 output.

S_2 (Tuned features) layer: The S_2 layer models V4 or posterior IT by matching a set of $4 \times 4 \times m$, $8 \times 8 \times m$, $12 \times 12 \times m$, and $16 \times 16 \times m$ prototypes, which have been randomly sampled from a set of representative images. These prototypes make up a dictionary of patches consisting of k entries used as fuzzy templates consisting of simple features that are position and scale invariant. The value of m represents the number of orientations extracted from the original image, which in this case is 12. S_2 computes the response of a patch, X, of C_1 units, to a particular S_2 feature prototype, P, of size $n \times n \times m$ ($n = \{4, 8, 12, 16\}$). The number of patches, k, is determined through a learning phase, which randomly selects feature prototypes of varying sizes from a set of images which represent the categorization task. If a general model is desired, the training set should contain images not related to any categorization task. In S_2, the final response is given by the normalized dot product:

$$R(X, P) = \frac{XP}{\sqrt{\|X\|^2 - \frac{1}{n^2}(\sum x_i)^2}}$$

C_2 (Global invariance): The final layer provides global invariance by taking the maximum response from each of the templates across the scales. The output of this stage removes all position and scale information, leaving only a complex feature set. These complex features can then be used for classification. In this work, a Regularized Least-Square, RLS, classifier was used to perform the classification. Note that other classifiers can also be used as reported by Ref. [37], [38].

Although all HMAX stages have been mapped to hardware ac-

Table 8 A summary of resource utilization for AIM accelerator on XCV6SX475T.

	Slice Register	Slice LUT	BRAM	DSP
AIM Resource	92,395	66,404	368	182

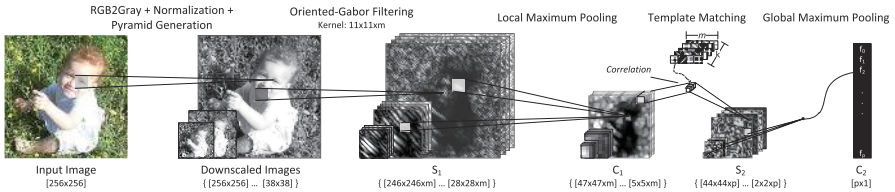


Fig. 21 A computational template of HMAX model showing the HMAX stages.

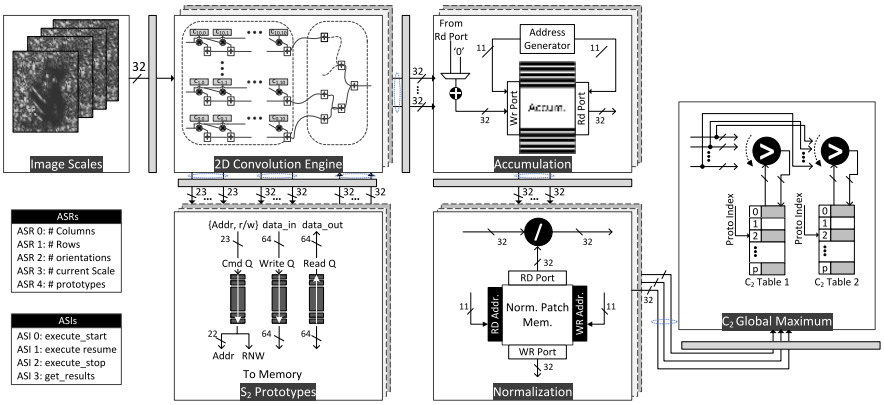


Fig. 22 S_2/C_2 accelerator. The accelerator combines the S_2 and C_2 stages into a single multi-stage pipeline.

celerators, this paper focuses on the architecture of the S_2/C_2 combined architecture as algorithm profiling reveals that S_2 is the most time consuming stage in the HMAX model. However, it is worth mentioning that both the image and S_1 layers were combined into a single SOP. In this case, the ROI streamed into the SOP is first preprocessed to produce the pyramid scales, utilizing a Gaussian smoothing operator and the subsampler operator. Then, these generated scales are subsequently processed by the Gabor filters to produce the orientations per each scale. On the other hand, the C_1 accelerator was mapped to a SAP-PE.

The S_2 and C_2 accelerators were combined into a multi-stage pipeline that resides in a single SAP-PE. There are two reasons for combining these accelerators: (1) The C_2 pooling operation can occur immediately following the computation of a current S_2 feature output without a delay, (2) Combining these two modules effectively decreases the amount of data required to be sent across the network. **Figure 22** depicts the S_2/C_2 architecture. The accelerator performs template-matching on the input image scale across all prototypes in the S_2 dictionary. First, since the template-matching operation is an iterative process, the current input image scale is stored in a local scratchpad memory to reduce the overhead that would have been incurred if the image was streamed over the network in each iteration. The capacity of the scratchpad memory was made large enough to store all orientations of the largest possible scale. For example, if the largest scale is 47×47, using 12 orientations and 4 bytes to represent a pixel, then the local memory should be at least $47 \times 47 \times 12 \times 4$ = 106,032 bytes.

The template-matching operation is essentially a 2D convolution operation. To support this operation, a multi-tap convolution engine is implemented to support kernel sizes 4×4, 8×8, 12×12 and 16×16. Note that the current kernel size is a runtime configurable parameter that can be set using ASR instructions. The prototypes are stored in SRAM memory, which can be accessed through an optimized memory controller. The 2D convolution is followed by the Accumulation stage, where the convolution output from each orientation is pixel-wise accumulated. This is achieved as follows: the output from the first orientation is stored as-is in a local memory. Then, for each subsequent orientation, the proper address is generated to read the pixel value from memory, summed with the corresponding convolution output pixel, and the result is written back to the local memory. This procedure is repeated until all orientations are processed. The next stage in the pipeline, Normalization, normalizes the output of the Accumulation stage. The normalization is done by dividing the output pixel by the pre-computed local average-of-sum of the input image scale. The neighborhood window size of local average-of-sum is determined by the current kernel size.

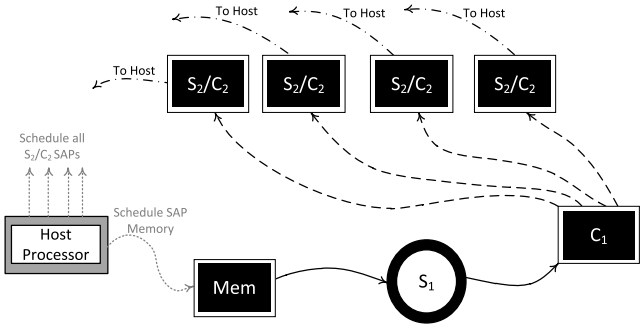


Fig. 23 Host-HMAX accelerator interaction. The Host schedules the SAP Memory to write contrast-enhanced images to C_1 SAP-PE flowing through the S_1 SOP. The C_1 SAP-PE broadcasts its output to multiple instances of the S_2/C_2 accelerator (figure shows 4 instances). Finally, the Host schedules a read request from each S_2/C_2 accelerator sequentially and aggregates the incoming results from these accelerators.

The C_2 stage performs global maximum operation. This stage receives the normalized output from the previous stage and performs a C_2 tables lookup, indexed by the current prototype ID and scale number. If the received output is larger than the value stored in a table, then the indexed cell is updated with that output.

As stated earlier, S_2 stage is the most time-consuming operation in the HMAX model. Therefore, the pipeline stages: Convolution, Accumulation, and Normalization can be duplicated in order to parallelize the template-matching operation. Note that the number of duplicate instances is mainly constrained by the available resources.

Figure 23 illustrates the interactions that occur between the Host processor and HMAX accelerators. Although the figure shows a virtual topology, however, all components are mapped to the physical topology shown in Fig. 16.

As discussed earlier in Section 4.1, the retina preprocessor produces a full resolution contrast-enhanced image to be used by the feature extractor. This enhanced image is buffered in the memory

attached to FPGA 'A', See Fig. 16. The host processor schedules an ROI transfer of the enhanced image from the memory to the C_1 accelerator, where the latter is mapped to FPGA 'B'. This ROI flows through S_1 stream processor, mapped to FPGA 'A'. This process is repeated for every two adjacent scales across all orientations. The output of the C_1 accelerator is broadcasted to all S_2/C_2 accelerators, mapped to FPGAs 'A', 'B', 'D' and 'E'—exploiting task-level parallelism across all the prototypes in the S_2 dictionary. Then, the host processor will schedule a read transfer of results from all S_2/C_2 accelerators, sequentially, one after the other. Since each S_2/C_2 accelerator is operating on a different set of prototypes, the host processor will have to merge these outputs as it receives them. Finally, the host processor tests the aggregated feature vector using a linear classifier, where the classification decision is finally made.

Table 9 summarizes the HMAX resources utilization when mapped to FPGAs 'A', 'B', 'D' and 'E'. Note that the numbers in the table includes the four instances of the S_2/C_2 accelerator.

Table 9 A summary of resource utilization for the HMAX accelerator on XCV6SX475T.

	Slice Register	Slice LUT	BRAM	DSP
HMAX Resource	316,794	133,611	623	2,206

5. Experimental Setup and Results

This section discusses both the accuracy and performance of the implemented accelerators running on the multi-FPGA platform. Furthermore, a cross-platform performance comparison is presented. **Figure 24** shows the experimental setup used in this paper.

5.1 Classification Accuracy

The output of the HMAX model (i.e., features vector) is used as an input to a classifier for recognition purposes. To test the classification accuracy of the neuromorphic accelerators, a total of 10 categories from the Caltech101 [40] data set are used. **Table 10** lists the categories and number of images used for training and testing the classifier.

One of the features offered by the acceleration framework, and discussed earlier in this paper, is the ability to modify the parameters of the accelerators without the need to re-synthesize the system. This allows for exploring the accelerator's configurations, while not affecting the productivity of the user. Using this feature, we study how the number of scales and orientations of the HMAX model impact the classification accuracy. **Figure 25** shows the classification accuracy for a number of HMAX config-

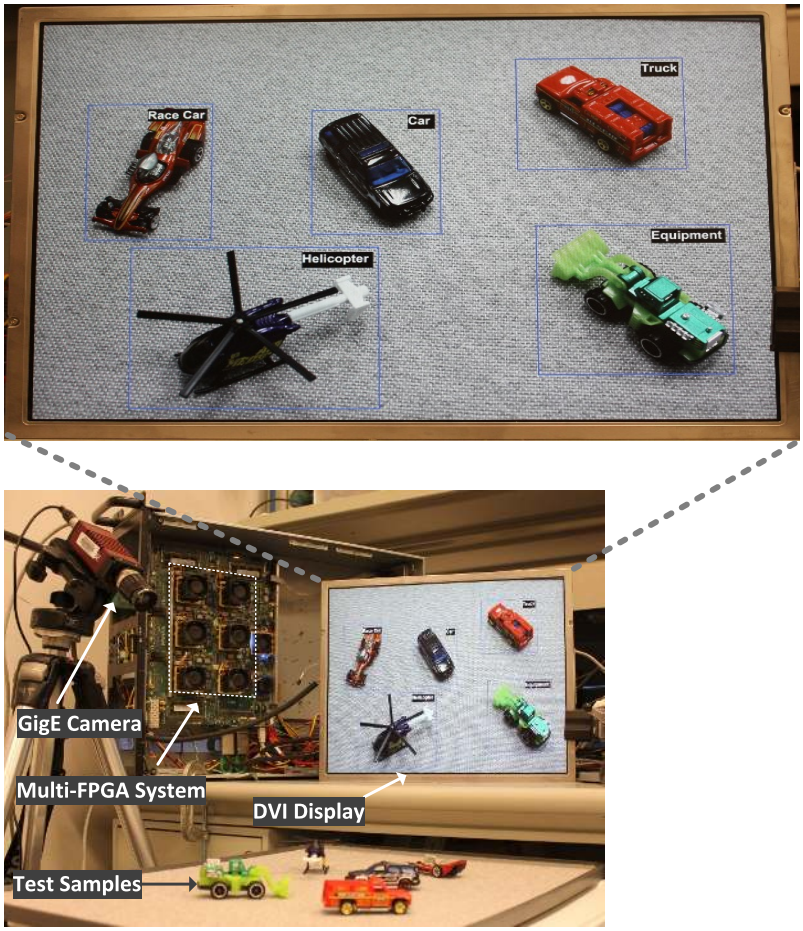


Fig. 24 Experimental Setup. The figure shows toy vehicles and aircrafts used as test samples. A GigE camera is used as a streaming input to the host processor (not shown in figure). A multi-FPGA system is used to process the input frames for object detection and recognition. A DVI display shows result image annotated with bounding boxes and classification labels.

Table 10 List of Caltech101 categories used in the experiments. The third and fourth columns show the number of images used for training and testing the classifier, respectively. Note that there is no overlap between training and test images.

Category ID	Category Name	# training images	# test images	Total
1	airplanes	400	400	800
2	car_side	62	61	123
3	chandelier	54	53	107
4	grand_piano	50	49	99
5	helicopter	44	44	88
6	ketch	57	57	114
7	laptop	41	40	81
8	motorbikes	399	399	798
9	revolver	41	41	82
10	watch	120	119	239
Total		1,268	1,263	2,531

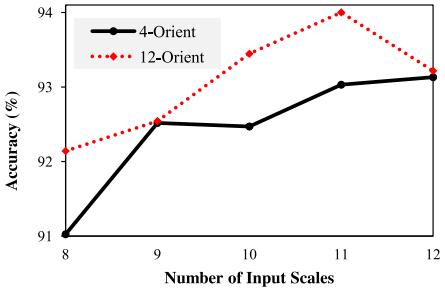


Fig. 25 Object classification accuracy for a number of accelerated HMAX configurations.

urations. The figure illustrates the impact of changing the number of input scales—for the same number of orientations—on the overall accuracy. For the 4 orientation set (4-Orient), increasing the number of input scales results in improved accuracy. On the other hand, the 12 orientation set (12-Orient) exhibits a varying, but consistent, improvement in accuracy when number of input scales is increased within the 8- to 11-scale configuration sets. However, the 12-scale configuration results in 0.48% less accurate classification when compared to the 11-scale configuration. This insignificant (< 1%) degradation in recognition is attributed to the frequent truncation of the fixed-point representation during the multiply-accumulate operation within convolution.

Compared to a CPU implementation [41] of the HMAX algorithm, it is found that the classification accuracy of the FPGA implementation is at most 2% less accurate than the CPU implementation. Again, the reason for the discrepancy is that the neuromorphic accelerators use fixed-point format to represent numerical values, compared to floating-point format used by the CPU implementation.

5.2 Performance

This subsection discusses the performance of the proposed accelerators in terms of speed and power efficiency. Additionally, a quantitative comparison is performed between the accelerators and a multi-core CPU and GPU implementations.

The performance of the accelerated retina processor is compared to a CPU implementation developed using OpenCV [42] and executed on a 2.4 GHz Intel Xeon processor. **Figure 26** shows a frame-rate comparison between the CPU and accelerated retina processor. For the smallest scale, 384 × 272, the accelerated retina delivers 393 fps, 5.24X speedup compared to CPU.

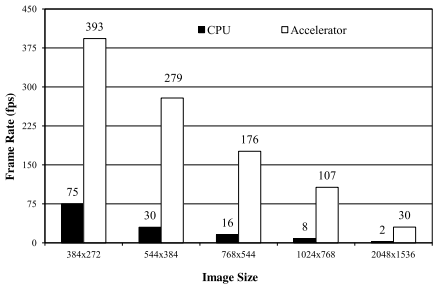


Fig. 26 A comparison of performance (frame rate) between CPU and accelerated retina processor.

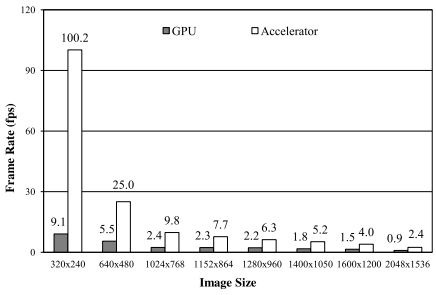


Fig. 27 A comparison of performance (frame rate) between GPU and accelerated AIM processor.

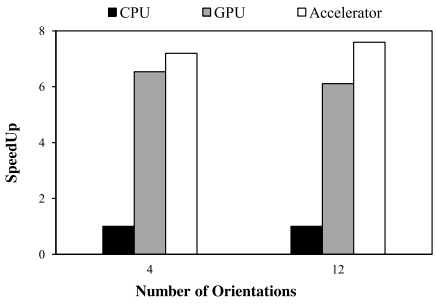


Fig. 28 A comparison of performance (frame rate) between CPU, GPU, and accelerated HMAX for two configurations (Values are normalized to CPU).

The speedup is even larger when processing the largest scale, 2,048 × 1,536, where accelerated retina processor outperforms the CPU by 15X.

Similarly, a GPU implementation of the AIM algorithm is used for comparison with the AIM accelerator. The GPU implementation is executed on an Nvidia GeForce GTS 250 with graphics and processor clock frequencies of 738 MHz and 1.84 GHz, respectively. **Figure 27** demonstrates the performance gain, in fps, of the AIM accelerator compared to the GPU. The figure shows the AIM accelerator outperforming the GPU by 2.7X for a 2,048 × 1,536 image and 11X for a 320 × 240 image.

On the other hand, the performance of the HMAX accelerators is compared to a software implementation based on Ref. [41] running on a 12-core Xeon 2.4 GHz processor. The software implementation was parallelized across all 12 cores and utilized SSE instruction set extension. Additionally, the performance is compared to an optimized GPU implementation coded in CUDA running on a Nvidia Tesla M2090 platform [43], which houses a Tesla T20A GPU, clocked at 1.3 GHz. **Figure 28** illustrates a performance comparison of execution time in fps for two configurations; namely, 4 and 12 orientations. Compared to the CPU, the HMAX accelerator delivers a speedup of 7.2X (7.6X)

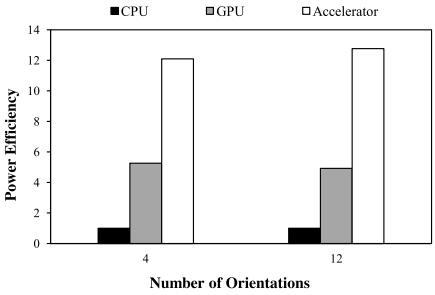


Fig. 29 A comparison of power efficiency between CPU, GPU, and accelerated HMAX for two configurations (Values are normalized to CPU).

for 4-orientation (12-orientation) configuration when compared to CPU. Similarly, the HMAX accelerators deliver a speedup of 1.1X (1.24X) for the 4-orientation (12-orientation) configuration when compared to GPU.

Power efficiency in fps-per-watt of the HMAX accelerators is compared to CPU and GPU. The power consumption of the CPU while executing HMAX was measured and found to be 116 Watts, while the GPU operated at a measured power consumption of 144 Watts. For the purpose of measuring power consumption, the HMAX accelerators were also mapped to a Virtex-5 [24] platform equipped with power measurement capabilities. The measurements show a total of 69 Watts power consumption. **Figure 29** shows a power efficiency comparison, where the HMAX accelerators outperformed the CPU by 12.1X (12.8X) for 4-orientation (12-orientation) configuration. Moreover, the neuromorphic accelerators outperformed the GPU by 2.3X (2.6X) for 4-orientation (12-orientation) configuration.

The GPU implementation of HMAX is done using CUDA 4.0 [44]. Performance improvement on the GPU is a combination of two factors : optimizing memory throughput and maximizing thread-level parallelism. Nvidia's Tesla M2090 GPU based on the SIMT (Single Instruction Multiple Thread) architecture has vast parallel computing resources, ideal to accelerate the highly parallel, compute-intensive HMAX algorithm. The algorithm is implemented in 4 stages:-S_1, C_1, S_2, and C_2 sequentially, where the S stages involve convolution kernels and the C stages involve pooling kernels. In the S_1 stage, convolution of the input layer with all 12 orientations is computed in parallel, where spatial convolution is used. In the C_1 stage, all 11 pairs of adjacent scales are processed in parallel. The S_2 stage is the most computation and memory intensive operation and takes 19.8% of total GPU time as analyzed by the CUDA visual profiler [45]. The first 4 scales are processed in parallel followed by the next 7 and for each kernel launch computation with all prototype patches is done in parallel. Memory throughput is optimized as far as possible by using coalesced accesses to global memory and utilizing the faster per-block shared memory wherever possible. Overall memory throughput seems to be higher for the C stages than the S stages according to our results from the visual profiler. C_2 is also implemented in 2 phases, with scales 0-5 pooled in parallel followed by scales 5-10.

5.3 Discussion of Performance Results

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

For instance, in the retina processor, the architecture provides pixel-level parallelism concurrently across all operations in each stage. This high degree of parallelism is not achievable on general purpose CPU architectures as each sub-operation of the retina is executed sequentially. Contemporary CPU architectures with explicit vector-processing extensions lack the number of functional units and optimized memory infrastructure to exploit the immense data-level locality inherent in the many convolution operations of both retina and AIM accelerators. Moreover, the tightly coupled pipeline stages of convolution and histogram building/indexing eliminates the overhead of storing intermediate convolution results.

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

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

6. Conclusion

We have analyzed the characteristics of neuromorphic vision algorithms to propose a methodology of implementing such algorithms on an SoC in a structural and efficient manner.

As a communication fabric among neuromorphic accelerators, the interconnection network requires flexibility, scalability, programmability, and high performance. In order to meet these requirements, a reconfigurable NoC platform is proposed, in which special network interfaces provide frame-level granularity and application-level abstraction. We demonstrate how the Vortex platform fits very well to mapping dataflow graphs onto networks of domain specific custom accelerators.

After analyzing the requirements for customized accelerators, we have found that the degree of parallelism, power efficiency, parameterization, composability, and programmability are the key factors in realizing the full potential of neuromorphic algorithms. We propose a methodology for composing streaming operators in a hierarchical and automated fashion. In addition, we standardize the way in which compute-intensive accelerators are attached to the network and interact with other processing elements.

A system-level automation tool, *Cerebrum*, is proposed to assist users in prototyping and validating designs easily and efficiently on multi-FPGA systems. *Cerebrum* supports system com-

position even for users with little knowledge of hardware system design by performing mapping, synthesis, and configuration. Additionally, the tool assists users in mapping C/C++ *codelets* that are executed on SAP-PEs, where control-oriented operations run on light-weight processors with tightly coupled customized hardware accelerators.

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

Acknowledgments This work was supported in part by grants from DARPA Neovision2 program, Intel Science and Technology Center on Embedded Computing and NSF Awards 0916887, 0702617, 0829607, 1147388.

References

[1] Owens, J.D., Mattson, P.R., Rixner, S., Dally, W.J., Kapasi, U.J., Khailany, B. and Lagunas, A.L.: A Bandwidth-Efficient Architecture for Media Processing, *IEEE/ACM International Symposium on Microarchitecture*, Vol.0, pp.3+ (online), DOI: 10.1109/MICRO.1998.742118 (1998).

[2] Khailany, B., Williams, T., Lin, J., Long, E., Rygh, M., Tovey, D. and Dally, W.: A Programmable 512 GOPS Stream Processor for Signal, Image, and Video Processing, *IEEE Journal of Solid-State Circuits*, Vol.43, No.1, pp.202–213 (online), DOI: 10.1109/JSSC.2007.909331 (2008).

[3] Talla, D., John, L.K. and Burger, D.: Bottlenecks in Multimedia Processing with SIMD Style Extensions and Architectural Enhancements, *IEEE Trans. Comput.*, Vol.52, No.8, pp.1015–1031 (online), DOI: 10.1109/TC.2003.1223637 (2003).

[4] Kim, J.-Y., Kim, M., Lee, S., Oh, J., Kim, K. and Yoo, H.-J.: A 201.4 GOPS 496 mW Real-Time Multi-Object Recognition Processor With Bio-Inspired Neural Perception Engine, *IEEE Journal of Solid-State Circuits*, Vol.45, No.1, pp.32–45 (online), DOI: 10.1109/JSSC.2009.2031768 (2010).

[5] Schemmel, J., Fieres, J. and Meier, K.: Wafer-scale integration of analog neural networks, *IEEE International Joint Conference on Neural Networks, 2008, IJCNN 2008* (*IEEE World Congress on Computational Intelligence*), pp.431–438 (online), DOI: 10.1109/IJCNN.2008.4633828 (2008).

[6] Vogelstein, R.J., Mallik, U., Cauwenberghs, G., Culurciello, E. and Etienne-Cummings, R.: Saliency-driven image acuity modulation on a reconfigurable silicon array of spiking neurons, *Advances in Neural Information Processing Systems*, pp.1457–1464, MIT Press (2005).

[7] Annema, A.J., Nauta, B., VanLangevelde, R. and Tuinhout, H.: Analog circuits in ultra-deep-submicron CMOS, *IEEE Journal of Solid State Circuits*, Vol.40, No.1, pp.132–143 (online), available from ⟨http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1374997⟩ (2005).

[8] Park, S., Cho, Y.C.P., Irick, K.M. and Narayanan, V.: A reconfigurable platform for the design and verification of domain-specific accelerators, *2012 17th Asia and South Pacific Design Automation Conference* (*ASP-DAC*), pp.108–113 (online), DOI: 10.1109/ASPDAC.2012.6164928 (2012).

[9] ARM Limited: *AMBA Specification v2.0* (1999). Technical Note.

[10] IBM: *CoreConnect* (1999). Technical Note.

[11] Kumar, S., Jantsch, A., Soininen, J.-P., Forsell, M., Millberg, M., Oberg, J., Tiensyrja, K. and Hemani, A.: A network on chip architecture and design methodology, *VLSI, 2002, Proc. IEEE Computer Society Annual Symposium*, pp.105–112 (online), DOI: 10.1109/ISVLSI.2002.1016885 (2002).

[12] Dally, W. and Towles, B.: Route packets, not wires: on-chip interconnection networks, *Proc. Design Automation Conference, 2001*, pp.684–689 (online), DOI: 10.1109/DAC.2001.156225 (2001).

[13] Xilinx: Virtex-6 Family Overview, (online), available from ⟨http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf⟩ (2012).

[14] Nageswaran, J., Richert, M., Dutt, N. and Krichmar, J.: Towards reverse engineering the brain: Modeling abstractions and simulation frameworks, *VLSI System on Chip Conference* (*VLSI-SoC*), *2010 18th IEEE/IFIP*, pp.1–6 (2010).

[15] Lee, E.A. and Parks, T.M.: Dataflow process networks, *Proc. IEEE*, Vol.83, No.5, pp.773–801 (online), DOI: 10.1109/5.381846 (1995).

[16] DeBole, M., AlMaashri, A., Cotter, M., Yu, C.-L., Chakrabarti, C. and Narayanan, V.: A framework for accelerating neuromorphic-vision algorithms on FPGAs, *2011 IEEE/ACM International Conference on Computer-Aided Design* (*ICCAD*), pp.810 –813 (online), DOI: 10.1109/ICCAD.2011.6105351 (2011).

[17] Neely, C., Brebner, G. and Shang, W.: ShapeUp: A High-Level Design Approach to Simplify Module Interconnection on FPGAs, *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines* (*FCCM*), pp.141–148 (2010).

[18] Impulse: C-to-FPGA Tools from Impulse Accelerated Technologies, (online), available from ⟨http://www.impulseaccelerated.com⟩ (2007).

[19] Calypto: Catapult C Synthesis, (online), available from ⟨http://www.calypto.com/catapult_c_synthesis.php⟩ (2011).

[20] Xilinx: Design Tools, (online), available from ⟨http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm⟩ (2012).

[21] Xilinx: ISE Design Suite: DSP Edition, (online), available from ⟨http://www.xilinx.com/products/design-tools/ise-design-suite/dsp-edition.htm⟩ (2012).

[22] Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K. and Zhang, Z.: High-Level Synthesis for FPGAs: From Prototyping to Deployment, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol.30, No.4, pp.473–491 (2011).

[23] Zhang, Z., Fan, Y., Jiang, W., Han, G., Yang, C. and Cong, J.: AutoPilot: A Platform-Based ESL Synthesis System, *High-Level Synthesis*, Coussy, P. and Morawiec, A. (eds.), pp.99–112, Springer Netherlands (2008).

[24] Xilinx: Virtex-5 Family Overview, (online), available from ⟨http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf⟩ (2009).

[25] Xilinx: Virtex-5 FXT ML510 Embedded Development Platform, (online), available from ⟨http://www.xilinx.com/products/boards-and-kits/HW-V5-ML510-G.htm⟩ (2012).

[26] Nallatech: FSB — Development Systems, (online), available from ⟨http://www.nallatech.com/Intel-Xeon-FSB-Socket-Fillers/fsb-development-systems.html⟩ (2012).

[27] Xilinx: Virtex-5 LXT FPGA ML505 Evaluation Platform, (online), available from ⟨http://www.xilinx.com/products/boards-and-kits/HW-V5-ML505-UNI-G.htm⟩ (2012).

[28] Xilinx: Virtex-6 FPGA ML605 Evaluation Kit, (online), available from ⟨http://www.xilinx.com/products/boards-and-kits/EK-V6-ML605-G.htm⟩ (2012).

[29] DiniGroup: Godzilla's Son-In-Law: ASIC Prototyping Engine Featuring Xilinx Virtex-6, (online), available from ⟨http://www.dinigroup.com/new/DNV6F6PCIe.php⟩ (2012).

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

[31] Schiller, P.H. and Logothetis, N.K.: The color-opponent and broad-band channels of the primate visual system, *Trends in Neurosciences*, Vol.13, No.10, pp.392–398 (1990).

[32] Newman, E.A. and Hartline, P.H.: Integration of visual and infrared information in bimodal neurons in the rattlesnake optic tectum, *Science*, Vol.213, No.4509, pp.789–791 (online), available from ⟨http://www.sciencemag.org/content/213/4509/789.short⟩ (1981).

[33] Ellias, S.A. and Grossberg, S.: Pattern formation, contrast control, and oscillations in the short term memory of shunting on-center off-surround networks, *Biological Cybernetics*, Vol.20, No.2, pp.69–98 (online), available from ⟨http://www.springerlink.com/index/10.1007/BF00327046⟩ (1975).

[34] Waxman, A.M., Aguilar, M., Fay, D.A., Ireland, D.B., Racamato, J.P., Ross, W.D., Carrick, J.E., Gove, A.N., Seibert, M.C., Savoye, E.D., et al.: Solid-State Color Night Vision : Fusion of Low-Light Visible and Thermal Infrared Imagery, *Lincoln Laboratory*, Vol.11, No.1, pp.41–60 (1998).

[35] Bruce, N.D.B. and Tsotsos, J.K.: An Information Theoretic Model of Saliency and Visual Search, *WAPCV*, pp.171–183 (2007).

[36] Riesenhuber, M. and Poggio, T.: Hierarchical models of object recognition in cortex, *Nature Neuroscience*, Vol.2, pp.1019–1025 (1999).

[37] Serre, T., Wolf, L., Bileschi, S., Riesenhuber, M. and Poggio, T.: Robust Object Recognition with Cortex-Like Mechanisms, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol.29, No.3, pp.411–426 (2007).

[38] Mutch, J. and Lowe, D.G.: Object Class Recognition and Localization Using Sparse Features with Limited Receptive Fields, *Int. J. Comput. Vision*, Vol.80, No.1, pp.45–57 (2008).

[39] Serre, T., Kouh, M., Cadieu, C., Knoblich, U., Kreiman, G., Poggio, T., Serre, T., Kouh, M., Cadieu, C., Knoblich, U., Kreiman, G. and Poggio, T.: A theory of object recognition: Computations and circuits in the feedforward path of the ventral stream in primate visual cortex,

AI Memo (2005).
[40] Fei-Fei, L., Fergus, R. and Perona, P.: Learning generative visual models from few training examples, *Workshop on Generative-Model Based Vision, IEEE Proc. CVPR* (2004).
[41] Mutch, J.: hmin: A Minimal HMAX Implementation, (online), available from ⟨http://cbcl.mit.edu/jmutch/hmin/⟩ (2011).
[42] OpenCV: Open Computer Vision Library, (online), available from ⟨http://sourceforge.net/projects/opencvlibrary/⟩ (2012).
[43] Nvidia: Tesla M2090 Board Specification, (online), available from ⟨http://www.nvidia.com/docs/IO/43395/Tesla-M2090-Board-Specification.pdf⟩ (2011).
[44] Nvidia: Nvidia CUDA 4.0 C Programming Guide, (online), available from ⟨http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf⟩ (2012).
[45] Nvidia: Nvidia CUDA 4.0 Visual Profiler User Guide, (online), available from ⟨http://developer.nvidia.com/nvidia-visual-profiler⟩ (2012).

Sungho Park was born in 1979. He received his B.S. degree in Electrical Engineering from Seoul National University, Seoul, Korea in 2007, and currently is pursuing Ph.D. in Computer Science and Engineering at the Pennsylvania State University. His research interests are reconfigurable architecture of communication infrastructure for streaming applications, especially neuromorphic vision, and distributed systems for large-scale video analytics.

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 System (Internetworking) from the University of New South Wales, Sydney, Australia. Currently, he is a Ph.D. candidate and a research assistant at the Pennsylvania State University in the department of Computer Science and Engineering. His research is focused on reconfigurable computing, domain-specific acceleration and high-level synthesis.

Kevin M. Irick is a Research Scientist in the Microsystems Design Lab in the Department of Computer Science and Engineering at Pennsylvania State. He received his B.S. degree in Electronics Engineering Technology from DeVry University in 2002. He earned his M.S. and Ph.D. degrees in Computer Science and Engineering from The Pennsylvania State University in 2006 and 2009 respectively. His research interests include application-specific hardware accelerators, hardware-assisted image processing and recognition, and high-performance computing on reconfigurable architectures.

Aarti Chandrashekhar is a Component Design Engineer at Intel Corporation in Folsom, CA. She received a B.Tech. degree in Electrical Engineering from College of Engineering, Pune, India in 2007 and a M.S. degree in Electrical Engineering from the Pennsylvania State University in 2011. Her research interests include dataflow computing on reconfigurable platforms and high-performance embedded computing.

Matthew Cotter was born in 1981. He received his B.S. degree in Computer Engineering from the Pennsylvania State University in 2008. He is currently a Ph.D. candidate at the Pennsylvania State University in the department of Computer Science and Engineering. His research is focused on emerging devices and their potential for applications in neuromorphic hardware design and applications.

Nandhini Chandramoorthy was born in 1988. She received her B.E. in Electronics & Communication Engineering from Anna University, Chennai, India and is presently a 2nd year Ph.D. student at the Department of Computer Science and Engineering in the Pennsylvania State University. Her research interests include FPGA and GPU architectures for bio-inspired vision algorithms.

Michael DeBole is an Advisory Engineer in IBM's System and Technology Group in Poughkeepsie, NY where he joined in 2012. He received a B.S. degree in Computer Engineering from the Pennsylvania State University in 2006 and a Ph.D. degree in Computer Science and Engineering from the Pennsylvania State University in 2011. Prior to joining IBM, Dr. DeBole was a Research Associate at the Pennsylvania State University and worked on research including high-performance computer architectures for machine vision, neuromorphic FPGA architectures, embedded system design, and multi-FPGA design tools.

Vijaykrishnan Narayanan is a Professor of Computer Science and Engineering and Electrical Engineering at the Pennsylvania State University. Vijay received his Bachelors in Computer Science and Engineering from University of Madras, India in 1993 and his Ph.D. in Computer Science and Engineering from the University of South Florida, USA, in 1998. He is the co-director of the Microsystems Design Lab, which has more than 50 graduate students.

(Invited by Editor-in-Chief: *Hiroyuki Tomiyama*)