SHARC: A Streaming Model for FPGA Accelerators and its Application to Saliency

Srinidhi Kestur, Dharav Dantara and Vijaykrishnan Narayanan
Microsystems Design Laboratory (MDL), Department of Computer Science and Engineering,
The Pennsylvania State University, University Park, PA - 16802
{kesturvy, djd300, vijay}@cse.psu.edu

***Abstract*—Reconfigurable hardware such as FPGAs are being increasingly employed for accelerating compute-intensive applications. While recent advances in technology have increased the capacity of FPGAs, lack of standard models for developing custom accelerators creates issues with scalability and compatibility.**

We present SHARC - Streaming Hardware Accelerator with Run-time Configurability, for an FPGA-based accelerator. This model is at a lower-level compared to existing stream processing models and provides the hardware designer with a flexible platform for developing custom accelerators. The SHARC model provides a generic interface for each hardware module and a hierarchical structure for parallelism at multiple levels in an accelerator. It also includes a parameterization and hierarchical run-time reconfiguration framework to enable hardware reuse for flexible yet high throughput design.

This model is very well suited for compute-intensive applications in areas such as real-time vision and signal processing, where stream processing provides enormous performance benefits. We present a case-study by implementing a bio-inspired Saliency-based visual attention system using the proposed model and demonstrate the benefits of run-time reconfiguration. Experimental results show about 5X speedup over an existing CPU implementation and up to 14X higher Performance-per-Watt over a relevant GPU implementation.

I. Introduction

Many signal processing and synthetic vision applications require high-performance processing of incoming signal or image data streams. This I/O *stream processing* often includes filtering, convolution, decimation and interpolation operations. In addition to demanding processing requirements, stream processing applications frequently need to address system-level issues including size, weight and power (SWAP), time to deployment, field upgradeability and dynamic, on-the-fly reconfigurability. Modern FPGAs with their reconfigurability, large gate counts, embedded DSP units and built-in high-speed I/O ports have emerged as a credible alternative to custom ASICs or DSPs.

While FPGAs have been used extensively in application acceleration [1], there is a no general model for building custom accelerators. The design techniques, HDL coding styles, signaling specifications and data representations used by hardware designers are completely customized to achieve their end goal of speeding up the application. This lack of standardization creates compatibility issues while reusing/migrating accelerator cores from one application to another. Further, the parameterization strategy adopted by one designer may vary from others and hence a standard reconfiguration framework for hardware accelerators does not exist.

Most of the existing stream processing architectures are at a system-level [2] [3] [4]. As far as we know, there is no published resource at the hardware-module level which assists a hardware designer in developing flexible and compatible accelerators.

In a typical stream processing application some feedback information may pass from later to earlier stages, but the majority of data movement is in a unidirectional stream [5]. Further, with shrinking of transistor sizes, the amount of logic and memory resources on a single FPGA is increasing with every new generation of FPGAs and a lot more computation can be accomplished in a single FPGA with minimal accesses to external memory. Hence, a streaming FPGA accelerator would receive data from external memory or I/O, process it in a sequence of streaming operations and do a single final write to external memory or I/O. Most of the intermediate buffering would be handled by the huge amounts of local BRAMs on the FPGA and intermediate accesses to external memory would be minimal if not absent. This assumption is reasonable in the domain of DSP and vision applications. Hence, existing stream processing models would be an overkill in terms of the resource overhead for memory I/O or intra-FPGA communication I/O.

978-3-9810801-7-9/DATE11/ © 2011 EDAA

In this paper, we propose *SHARC - Streaming Hardware Accelerators with Run-time Configurability*, which is a general framework for developing hardware accelerators. SHARC employs a simple signaling interface for each hardware module and a hierarchical structure to support parallelism at each level along-with a run-time configuration strategy. The vision for SHARC is to standardize hardware accelerators to enable plug-and-play hardware accelerators and a GNU-style sharing for accelerator cores as utilities/apps.

II. SHARC model

A. Hardware Module

Each hardware module in the SHARC model has the interface description as shown in Figure 1. We utilize the simplest form of the Xilinx LocalLink interface [6] for inter-module signaling. LocalLink is a high-performance, synchronous Xilinx standard interface for point-to-point connections and includes simple handshake signals and frame delimiter signals. Each hardware module in SHARC has 3 independent LocalLink (*LL*) ports/interfaces namely - input, output and control. The input LL port is a LL destination interface to receive data from the previous module in the pipeline and the output LL port is a LL source interface to transmit data to the next module in the pipeline. The control LL port is a LL destination interface to receive control data at run-time. Every hardware module has an input buffer, output buffer and configuration registers to provide temporary buffering of input data, output data and run-time parameters respectively.

The input LL port and output LL port can be of any width, so that the module can operate on multiple I/O streams at a time. For multiple streams, the module can either share the same, or have independent, frame delimiter signals for each stream. The module is independent of the source of the the input streams and the destination of the output streams, as long as they follow the LL specification. The control LL port also has frame delimiter signals to incorporate the notion of *frame* on the run-time parameters. Each hardware module must have a specific number and sequence of configuration parameters that need to be loaded at run-time, and must internally handle initialization of the configuration registers each time the parameters are loaded. This way, the SHARC interface

abstracts a streaming hardware module with a reasonable degree of complexity.

Example: A 4×4 switch has 4 input data streams and 4 output data streams and this can be supported by having 4-wide LL ports for the input and output. The run-time parameters could be the entries of the switching table which can be input back-to-back into the control LL port to create a control stream.

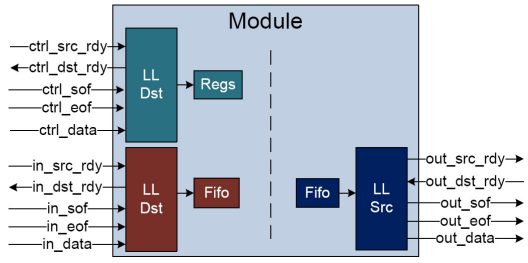

Fig. 1. The interfaces for each hardware module

B. The Accelerator hierarchy

An FPGA accelerator derives its throughput from the parallelism it provides. In order to provide a generic structure with massive parallelism in the design, the hardware modules in the accelerator are classified based on their position in the design hierarchy.

- At the bottom of the hierarchy are the *utilities*, which are basic arithmetic or logic functions such as - multiply-accumulator, adder tree, comparator, dynamic shifter, divider and resources such as FIFO buffers. These utilities can be custom implementations or from an IP library and are oblivious to the frame-level details.
- At the next level are the *user-IPs*, which are hardware realizations for frame-level operations and are mostly custom implementations. Example of such operations are - 2D convolution, down-sampler, image up-sampler. A user-IP itself can include instances of utilities, other user-IPs and custom logic.
- At the next level are the *cores*, which are frame-level operations at a higher scale. A core can include multiple instances of a single user-IP or different user-IPs along with custom logic.
- At the next level are the *wrappers*, which are basically composed of multiple cores. The additional function of a wrapper is to parse the configuration stream during run-time configuration and route the data to the appropriate cores in it.
- Finally, the top-most level in the hierarchy is a *stream-pipe* which is the streaming pipeline composed of wrappers and custom logic. During, run-time configuration, a stream-pipe broadcasts the configuration stream to all the wrappers instantiated and does not handle parsing the stream. This is explained in detail in later sections.

All hardware modules from the user-IPs to the stream-pipes have the LocalLink interface ports as shown in Figure 1. However, the utilities do not have the LocalLink interfaces, but have simple data and valid signals. A target application might include a suite of algorithms and each algorithm would have an associated hardware realization as a stream-pipe.

This hierarchical structure allows parallelism at every stage - i.e both fine-grain and coarse-grain parallelism are incorporated. Since each hardware module from utility to the stream-pipe is pipelined, temporal parallelism is inherent in the accelerator. Each user-IP and core can be designed with any degree of parallelism - example: a 1D FIR filter user-IP can be implemented by performing all the multiplications in parallel - and this is invisible to the higher-level wrapper. The wrapper however provides coarse-grain parallelism by allowing multiple instances of a core to operate in parallel.

C. Parameterization

A unique feature of an FPGA accelerator is the ability to be reconfigured to suit the needs of the application. This reconfiguration can be at several stages -

1) Compile-time reconfiguration - this is the simplest form of reconfiguration and is realized by modifying the HDL parameters in the design - which would require synthesizing the design to re-generate the bitstream.
2) Run-time reconfiguration - this is realized by modifying the run-time parameters in the design online. This would however require the hardware to support the entire range of the run-time parameter hence requiring the hardware to be designed for the worst-case value.
3) Partial reconfiguration - this is a complicated form of reconfiguration and is realized by regenerating the bitstream for only a portion of the design - which requires specific guidelines to be followed for the static and the reconfigurable portions of the design [7].

The SHARC framework currently supports compile-time and run-time reconfiguration (1 and 2 above). In order to utilize this ability completely, the hardware design needs to be completely parameterized so that nothing is hard-wired at design-time.

The bit-widths of the data bus and registers, latency of the primitives such as multipliers and adders, the depths of the buffers, number of fraction bits in the fixed-point representation, the fields in a control packet and further algorithm-specific parameters which determine the type of hardware, are all declared as HDL parameters (compile-time). As such, the type, size and functionality of the synthesized design is highly dependent on these parameters. *Generate* statements in Verilog or VHDL are used to instantiate the desired number of instances of a utility/user-IP/core/wrapper in a pipeline which are HDL parameters as well. The other variables in the design which do not need modification of the underlying hardware are all run-time parameters. These may include the data size, select lines for multiplexers, constants for data manipulation, initial values for registers etc.

Example: A filter of size W which performs the all multiplications in parallel can have W as a HDL parameter since it determines the number of multipliers to be synthesized. However, the actual filter coefficients can be run-time parameters.

D. Run-time Reconfiguration

Run-time reconfiguration is achieved by allowing hardware modules at all levels of the hierarchy to receive parameters at run-time to modify their functionality. Since the complexity and functionality of each hardware module increases as we go up the hierarchy, the number and type of control parameters required for each module also varies. SHARC however provides a generic interface to each module irrespective of the position in the hierarchy, so that the modules can serve as plug-and-play logic and also make system integration an easier task.

Each hardware module - user-IP/core/wrapper - is provided with a control port which follows the LocalLink interface. This control port is used to receive run-time parameters from a higher module in the hierarchy. A global packet format for the configuration data is determined as shown in Figure 2(b)

- which consists of the control data along with the following headers -

- pipe-ID - an identifier for the stream-pipeline.
- core-ID - identifies the type of core
- instance-ID - identifies the specific instance of the core, since each instance can have a different set of parameters

While, the width of each field can be changed by the user, the baseline SHARC control packet is 64-bits wide with 32-bits of control data and 8-bits each for pipe-ID, core-ID and instance-ID. The most significant 8 bits are reserved for application specific flags. This allows up to 256 pipelines each having up to 256 different cores with each core having up to 256 instances. Hence, this packet format is sufficient to support a fairly complex application with massive parallelism.

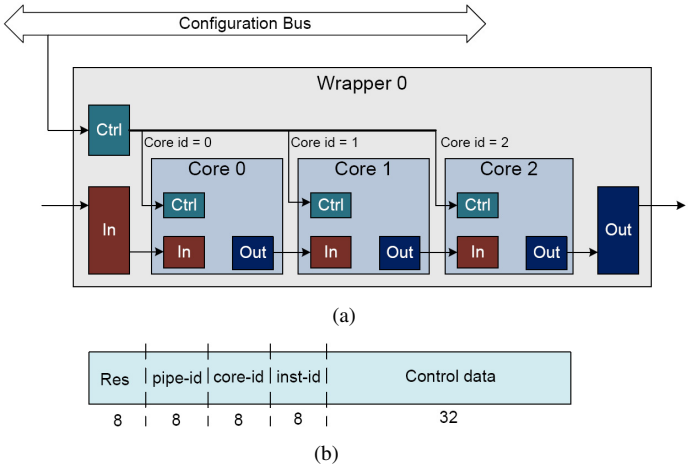

Fig. 2. (a) Hierarchical run-time configuration and (b) packet format

A configuration stream is formed by concatenating the run-time parameters for each hardware module in the design. This configuration structure is streamed into the control port of the top-most module in the hierarchy namely the stream-pipe. The stream-pipe broadcasts the control packets on a configuration bus to all the wrappers in the design. The wrappers parse the configuration packets to determine if the pipe-ID and core-ID correspond to the cores in itself. If it does, it extracts the control data from the packets and passes it to each core corresponding to the instance-ID as shown in Figure 2(a). The core handles the forwarding of the control data to the user-IPs. Thus the configuration parameters trickle down from one large structure into every hardware module in the hierarchy.

Every hardware module must have a pre-determined set of run-time parameters to be included as part of the design specification. A software script can be written in C++ / MATLAB, to build the configuration stream for a particular accelerator. This stream has to be loaded into the design once the bistream has been downloaded on to the FPGA and also if parameters need to be changed during run-time. One advantage of the SHARC framework is that it allows the hardware design to scale to a bigger or smaller structure. Additional cores or wrappers can be inserted or existing modules can be removed in hardware and the configuration structure updated to realize a different functionality with minimal effort.

E. System Integration

An FPGA accelerator usually consists of communication interfaces for both intra-chip and inter-chip communication. Intra-chip communication is usually achieved using local routers or system-buses such as PLB and inter-chip communication is achieved by interfaces such as PCI express, Gigabit Ethernet or inter-FPGA links. In order to realize an end-to-end system using a SHARC-based accelerator, the SHARC pipeline requires interface modules to DDR memory, a system bus such as PLB and external I/O such as PCI express or inter-FPGA links. These interface modules need to be implemented once and can be reused for any application. An end-to-end system using a SHARC-based accelerator is shown in Fig 3.

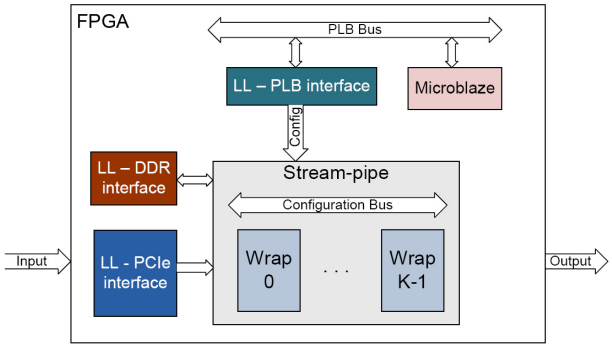

Fig. 3. A SHARC based system

III. Related Work and Discussion

There has been considerable interest in developing FPGA-based accelerators and stream processors. FPGA accelerators can be realized by complete hardware [1] or as embedded MPSoCs [8].

In addition to application accelerators, a stream processor for convolutional networks is described in [3] [9], which has a regular array of stream operators and an interconnection network composed of local and global routers for on-chip and off-chip communication. A streaming library composed of multi-port memory controllers was presented in [2]. A similar multi-FPGA system for composing accelerators was presented in [4]. Virtual interfaces for FPGAs [10] have been studied before.

These stream processing models are at a system-level and are generic enough to be extended to various applications, but they impose huge resource overheads in terms of the communication framework. In comparison, SHARC uses a simple point-to-point communication using a generic signaling interface without the use of routers for each core and hence has very low resource overhead. Further, SHARC-based system includes interfaces to standard off-chip communication interfaces such as PCIe links or inter-FPGA links to realize a multi-FPGA system. Also, the communication interfaces can be customized based on the target board and device.

Run-time parameterization has been explored recently [7], however existing schemes require the use of the embedded CPU and the system bus. Instead, the SHARC accelerator is still a full-hardware solution with minimal overhead for run-time configuration.

Further, the SHARC model is independent of the target device or the target application. It is a generic model that can be used across the spectrum for hardware accelerators. This model can be extended based on the needs of the application and might allow application developers, who are not hardware experts, to plug-and-play accelerators if they have the necessary interfaces.

IV. Case Study: Saliency-based Visual Attention

Attention models are being used in various applications such as object recognition, image/video compression and robotic

control. Among the various bio-inspired visual attention models, the bottom-up saliency model proposed by Itti et al. [11] and extended by Walther et al. [12] is widely used.

A. Saliency Algorithm

As shown in Figure 4, input is provided in the form of static color images which are pre-processed to obtain the intensity map (I) and the red-green (RG) and blue-yellow (BY) color opponency maps. Nine spatial scales are created using dyadic Gaussian pyramids, which progressively low-pass filter and subsample the input image, yielding horizontal and vertical image-reduction factors ranging from 1:1 (scale 0) to 1:256 (scale 8), to obtain 3 pyramids $I(\sigma)$, $RG(\sigma)$ and $BY(\sigma)$, where $\sigma \epsilon \{0..8\}$. Local orientation information is obtained from I using oriented Gabor pyramids $O(\sigma, \theta)$ where $\theta \epsilon \{0^0, 45^0, 90^0, 135^0\}$ is the preferred orientation and $\sigma \epsilon \{0..8\}$.

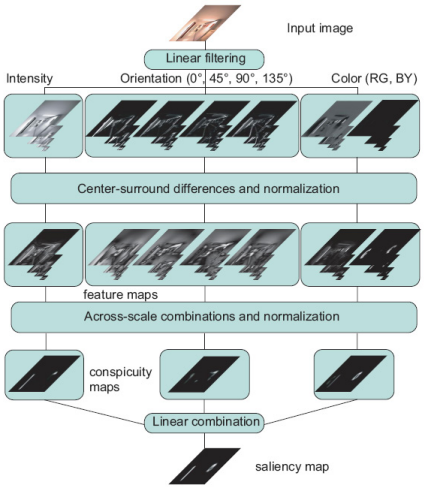

Fig. 4. The Saliency-based bottom-up attention model [11]

Each feature is computed by a set of linear *center-surround* operations. Center-surround is implemented as the difference between fine (center) and coarse (surround) scales: The center is a pixel at scale $c \epsilon \{2, 3, 4\}$ - and the surround is the corresponding pixel at scale $s = c + \delta$, with $\delta \epsilon \{3, 4\}$. The across-scale difference between two maps, denoted by Θ, is obtained by interpolation to the finer scale and point-by-point subtraction. $I(c, s) = |I(c) \Theta I(s)|$ $RG(c, s) = |RG(c) \Theta RG(s)|$ $BY(c, s) = |BY(c) \Theta BY(s)|$ $O(c, s, \theta) = |O(c, \theta) \Theta O(s, \theta)|$

Each feature map is now normalized by an operator $N(.)$ which consists of - (i) finding the map's global maximum M and computing the average of all its other local maxima u and (ii) globally multiplying the map by $\frac{(M-u)^2}{M}$.

The feature maps are then combined into three conspicuity maps $\overline{I}$ for intensity, $\overline{C}$ for color and $\overline{O}$ for orientation at the scale $\sigma = 4$ of the saliency map. They are obtained through across-scale addition (denoted by $\bigoplus$), which consists of resampling each map to scale 4 and then point-by-point addition. For orientation, four intermediary maps are first created by combination of the six feature maps for a given θ and are then combined into a single orientation conspicuity map. $\overline{I} = \bigoplus_{c=1}^{3} \bigoplus_{s=c+3}^{c+4} N\{I(c, s)\}$ $\overline{C} = \bigoplus_{c=1}^{3} \bigoplus_{s=c+3}^{c+4} [N\{RG(c, s)\} + N\{RG(c, s)\}]$ $\overline{O} = \sum_{\theta} N[\bigoplus_{c=1}^{3} \bigoplus_{s=c+3}^{c+4} N\{O(c, s, \theta)\}]$

The three conspicuity maps are normalized and summed into the final saliency map - $S = \frac{1}{3}[N(\overline{I}) + N(\overline{C}) + N(\overline{O})]$

B. SHARC Library for Saliency

As described above, the Saliency algorithm is highly complex and very compute-intensive. However, it has several common operations across all the channels including 2D convolution, image re-sampling, normalization etc and also has inherent parallelism. We follow a bottom-up design methodology by first implementing the building blocks and then composing them to realize the full Saliency algorithm.

The functional elements are classified based on hierarchy as in Table I. The building blocks for the Saliency algorithm such as 2D convolution are classified as user-IPs and bigger modules such as pyramid generator as cores. Each core has a wrapper to support multiple instances and then a Saliency pipeline to compute the conspicuity map of a channel for N_{cs} center-surround scales is the stream-pipe (If N_c and N_s are the number of center and surround levels per channel, then N_{cs} = $N_c \times N_s$ is the total number of center-surround scales).

TABLE I
SHARC HIERARCHY FOR SALIENCY PIPELINE

Stream pipe	Wrapper	Core	User-Ip
Saliency	Pyramid wrapper	Pyramid generator	2D convolution
	CSD wrapper	CSD	down-sampler
	MaxNorm wrapper	MaxNorm	up-sampler
	Resampler wrapper	Gabor filter	global-max
	Gabor wrapper	Resampler	local-max
		Across-scale Adder	steerable filter
			scaling

We utilize the SHARC model to implement a hardware library of functional elements which can be used to realize Saliency or similar computer vision algorithms. The library includes all the cores and user-IPs listed in Table I and are highly optimized and fully parallel implementations. Implementing this library provided several case studies of using SHARC model for hardware accelerators. The 2D convolution/filter module includes window buffers and 1D filter user-IPs to realize a separable 2D filter. The Gabor filter core is realized by a steerable filter approximation which includes a sinusoidal modulation of the Laplacian filter response followed by a separable 2D filter [13]. Here, the Laplacian filter and the 2D filter are user-IPs. The pyramid generator module is a core and consists of multiple instances of the 2D convolution and down-sampler user-IPs to generate a multi-resolution dyadic pyramid from an input image. It can generate either a Gaussian or a Laplacian pyramid, which can be specified using a run-time parameter.

The CSD module is a core which computes the difference between a high resolution center image and a low-resolution surround image, by up-sampling the surround to the same size as the center before performing a subtraction. The local-max and global-max user-IPs compute a 2D local maxima and global maximum over an image respectively. The MaxNorm core includes global-max and local-max user-IPs to compute the normalization factor and a scaling module to perform the actual scaling of the image with the norm factor. The Resampler core includes multiple instances of the up-sampler or down-sampler user-IPs to realize multi-level image re-scaling.

C. Saliency Pipeline and Run-time Reconfiguration

The basic operations common to all channels - intensity(I), color(C) and orientation(O) - are pyramid generation, center-surround difference(CSD), normalization, across-scale addition by re-sampling to output scale. However, each channel may require a different set of parameters. Example: the intensity and color channels require a Gaussian pyramid while the orientation channel requires a Gabor pyramid.

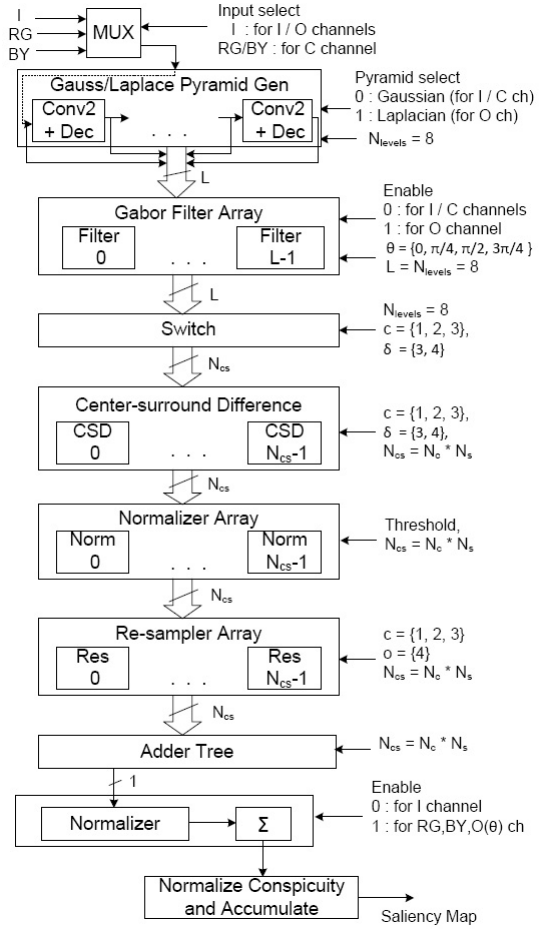

Fig. 5. Streaming pipeline for Saliency based on SHARC

The center-surround computations must be completed for 4 orientations, 2 color and 1 intensity channels (totally 7 channels), where each channel comprises of N_{cs} center-surround scales. If resources permit, each channel can be computed in parallel using independent pipelines. In effect, we would need 7 parallel pipelines to compute the Saliency map in single-pass.

However, this would need a huge amount of FPGA resources and might not fit on a single device. In this regard, we have developed a novel streaming saliency pipeline based on SHARC. Instead of having separate pipelines to implement each channel, we can instantiate a single SHARC-based saliency pipeline which can be configured at run-time as to which channel to implement. This would require instantiating all the cores and wrappers required to compute each channel at compile-time and running the pipeline for multiple iterations while configuring it to compute a particular channel each time around. Hence, run-time reconfiguration needs to be performed 7 times for each input image (once after each iteration). This provides a highly parallel and flexible design which scales with resource availability.

The Saliency pipeline as shown in Figure 5 can compute the conspicuity map for a single channel from all the center-surround scales in parallel. Hence, each wrapper would have N_{cs} instances of cores to handle the N_{cs} center-surround scales.

The input image *img* pre-processed to generate the intensity channel *I* and color channels *RG* and *BY*. The I input is selected for all orientations and the intensity channel while RG and BY are chosen one at a time for the 2 color channels.

For the intensity and color channels, the pyramid generator is configured to generate a Gaussian pyramid and the Gabor filter wrapper is disabled/bypassed. For the orientation channels, the pyramid generator is configured to generate a Laplacian pyramid and the Gabor filter wrapper is enabled.

The center-surround difference for all scales is computed by running N_{cs} instances of the *CSD* module in parallel. The Normalization block for each channel includes N_{cs} instances of the *MaxNorm* module. Next, the Resampler is used to resample each center-surround map to output scale. This is followed by an across-scale adder to accumulate the center-surround maps to obtain the conspicuity map. Further, the conspicuity normalizer and accumulator are enabled for the orientation and color channels to combine orientation/color maps over multiple iterations.

Finally, the conspicuity maps for the intensity, color and orientation channels are normalized and added to obtain the final Saliency map.

While, the Saliency pipeline is designed for processing all the N_{cs} center-surround scales in parallel, the Switch allows dynamic selection of the center and surround levels. If this design does not fit on the target FPGA, a subset of the center-surround scales can be computed in parallel which would require multiple iterations of the pipeline for just one channel. Thus, the SHARC framework allows the degree of parallelism to vary when resources available are different.

V. Experimental Setup and Results

Our target FPGA platform is the Xilinx ML605 board [14] which consists of a Virtex6 LX240T FPGA along with 512 MB of DDR3 memory and Compact Flash. Our test platform includes the Micro-blaze processor for software-based control, the PLB bus for on-chip communication, the Multi-port memory controller (MPMC) for accessing the external DDR3 memory and the SysAce Compact Flash card for external I/O. The Saliency pipeline is imported into this platform by implementing the PLB-LocalLink interface. Xilinx EDK is used to generate the bitstream, which is then downloaded on to the FPGA.

The configuration stream with run-time parameters is generated by executing a MATLAB script. The Compact Flash (CF) card is used for off-line transfer of the input images and the config file from a host to the DDR3 memory on the ML605 and output saliency map back to the host. The Saliency pipeline reads the config parameters from memory and initializes all the hardware modules in a hierarchical fashion. The input images are then loaded back-to-back into the Saliency core which computes the saliency map.

We use a nominal operating frequency of 200 MHz for the FPGA. For a 256×256 image, the Saliency system gives a throughput of 371 frames per sec which satisfies real-time requirements. We provide a performance comparison of our system with existing Saliency implementations in Table II. The CPU and GPU implementations are the closest to our implementation since they follow the bottom-up saliency model as proposed by Itti [11]. The CPU implementation is by Peters et al. [15] and the GPU implementation is by Xu et al. [16] on a multi-GPU system. The table shows that our FPGA implementation provides close to 5X speedup when compared to the CPU version and almost identical throughput for the GPU when normalized to a single-GPU performance.

The Table II also provides a comparison of Performance-per-Watt in terms of Frames-per-sec-per-Watt. The GPU performance is normalized to a single GPU for this calculation.

TABLE II
PERFORMANCE OF SALIENCY IMPLEMENTATIONS FOR 640×480 IMAGES

	CPU impl [15]	GPU impl [16]	Our FPGA impl
Hardware	Intel Xeon processor	4 Nvidia GeForce 8800(GTX)	Virtex6 LX240T
Frequency	2.8 GHz	1.35 GHz	200 MHz
Precision	floating-point	floating-point	fixed-point
Frame rate	19.48 fps	313 fps	89.6 fps
Power(TDP)	80 W [17]	155 W [18]	10 W [19]
FPS-per-Watt	0.2435	0.608	8.96

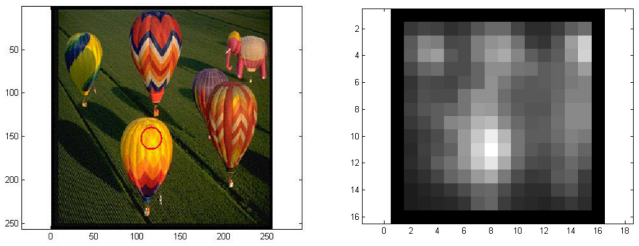

Fig. 6. Saliency map and the most salient location for 256x256 image

We use the Thermal Design Power (TDP) of the CPU and GPU for approximate device power consumption. For the FPGA, the power number is extracted from the data sheet [19]. The table shows that our FPGA implementation provides 37X and 14X times higher Performance-per-Watt compared to the CPU and GPU implementations respectively. Figure 6 shows the saliency map for a sample image - the brighter regions correspond to more salient locations in the image.

In order to demonstrate the scalability of the SHARC-based Saliency implementation, we provide projected performance for different FPGAs in the Virtex6 family. We choose SX475T and LX760 as the other Virtex6 devices for comparison. The Saliency system is synthesized on each of these FPGAs to obtain resource utilization numbers, from which the number of Saliency pipelines that can fit in parallel can be determined. Table III provides the resource utilization numbers for the SX475T FPGA for $N_{cs} = 6$.

Figure 7 provides a comparison of the throughput on the ML605 with the projected throughput on the SX475T and LX760 for a 256×256 image for 3 configurations of N_{cs}. For smaller N_{cs}, the number of computations and resource requirement is lower and so is the accuracy of the saliency map. Hence, multiple pipelines can be fit on a single device, which gives higher throughput. For higher N_{cs}, the Saliency map is more accurate, but the resource requirement is huge. Smaller FPGAs might not be able to process all center-surround scales in parallel and will have to run multiple iterations per channel. The SX475T and LX760 are both much bigger devices and hence can provide more parallelism and hence can speedup the application further.

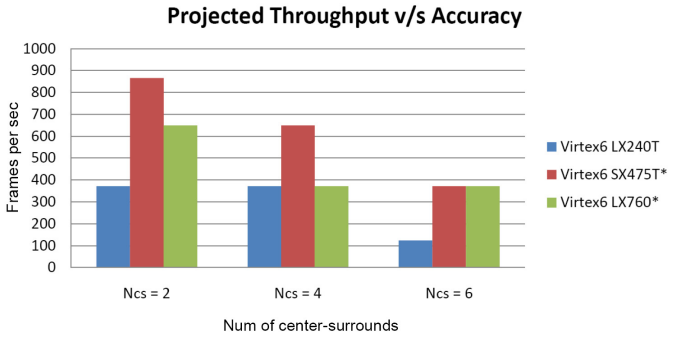

Fig. 7. Projected throughput of SHARC-Saliency system for 256x256 image

TABLE III
RESOURCE UTILIZATION ON VIRTEX6 SX475T FOR $N_{cs} = 6$

Slice Regs	Slice LUTs	BRAMs/FIFOs	DSP48E1
153832 (25%)	160253 (53%)	430 (40%)	590 (29%)

VI. CONCLUSION

In this paper, we have presented SHARC - a streaming model for hardware accelerators with run-time configurability - a simple yet powerful model for custom accelerator design. It provides a simple signaling interface based on Xilinx LocalLink interface with point-to-point communication along-with a hierarchical structure to enable parallelism at every stage in the pipeline. This work further presents a hierarchical run-time reconfiguration framework to enable hardware reuse and scalability. We apply this framework to implement an algorithm for Saliency-based Visual attention. The SHARC-based streaming architecture is highly parallel, completely pipelined and scales well to different parameter sets. Experimental results on Virtex6 FPGA platforms show about 5X speedup over an existing CPU implementation and up to 14X higher Performance-per-Watt over a relevant GPU implementation. Future work includes implementing the interfaces for SHARC to PCI express and supporting partial reconfiguration.

ACKNOWLEDGMENT

The authors acknowledge the valuable discussions with Dr. Deepak Khosla (HRL Laboratories) during the course of this work. This work was supported in part by NSF 0903432 and DARPA Neovision2 programs.

REFERENCES

[1] S. Hadjitheophanous, C. Ttofis, A. Georghiades, and T. Theocharides, "Towards Hardware Stereoscopic 3D Reconstruction A Real-time FPGA Computation of the Disparity Map," in *Design Automation and Test in Europe Conference (DATE), 2010*, Mar. 2010, pp. 1743 –1748.
[2] A. D. C. Lucas, S. Heithecker, and R. Ernst, "FlexWAFE - A High-end Real-time Stream Processing Library for FPGAs," in *DAC '07: Proc. of the 44th annual Design Automation Conference*. New York, NY, USA: ACM, 2007, pp. 916–921.
[3] C. Farabet, C. Poulet, J. Han, and Y. LeCun, "CNP: An FPGA-based Processor for Convolutional Networks," in *Intl. Conf. on Field Programmable Logic and Applications. FPL 2009.*, aug. 2009, pp. 32 –37.
[4] K. M. Irick, M. DeBole, S. Park, A. Al Maashri, S. Kestur, C.-L. Yu, and N. Vijaykrishnan, "A Scalable Multi-FPGA Framework for Real-time Digital Signal Processing," in *Proc. of SPIE*, Vol. 7444, 2009.
[5] "FPGAs for Stream Processing: A Natural Choice." [Online]. Available: http://www.cotsjournalonline.com/articles/view/100649
[6] "LocalLink User Interface." [Online]. Available: http://www.xilinx.com/products/ipcenter/LocalLink_UserInterface.htm
[7] "FPGA Run-time Reconfiguration: Two Approaches," *Altera White Paper*.
[8] A. Tumeo, F. Regazzoni, G. Palermo, F. Ferrandi, and D. Sciuto, "A Reconfigurable Multiprocessor Architecture for a Reliable Face Recognition Implementation," in *Design Automation and Test in Europe Conference (DATE), 2010*, Mar. 2010.
[9] C. Farabet et. al., "Hardware Accelerated Convolutional Neural Networks for Synthetic Vision Systems," in *Proc. of 2010 IEEE Intl. Symp. on Circuits and Systems, ISCAS '10*, May. 2010.
[10] Y. Ha et. al., "Virtual Java/FPGA Interface for Networked Reconfiguration," in *Asia and South Pacific Design Automation Conference, ASP-DAC '01*, 2001.
[11] L. Itti, C. Koch, and E. Niebur, "A Model of Saliency-based Visual Attention for Rapid Scene Analysis," *IEEE Tran. on Pattern Analysis and Machine Intelligence* , vol. 20, no. 11, pp. 1254 –1259, Nov. 1998.
[12] D. Walther and C. Koch, "Modeling Attention to Salient Proto-objects," *Neural Networks*, vol. 19, no. 9, pp. 1395 – 1407, 2006.
[13] H. Greenspan, S. Belongie, R. Goodman, P. Perona, S. Rakshit, and C. Anderson, "Overcomplete steerable pyramid filters and rotation invariance," in *IEEE Comp. Soc. Conf. on Computer Vision and Pattern Recognition, CVPR '94*, jun. 1994.
[14] "ML605 Documentation." [Online]. Available: http://www.xilinx.com/products/boards/ml605/reference_designs.htm
[15] R. J. Peters and L. Itti, "Applying Computational Tools to Predict Gaze Direction in Interactive Visual Environments," *ACM Trans. Applied Perception*, vol. 5, pp. 9:1–9:19, May 2008.
[16] T. Xu, T. Pototschnig, K. Kühnlenz, and M. Buss, "A High-speed Multi-GPU Implementation of Bottom-up Attention using CUDA," in *ICRA'09: Proc. of the IEEE intl. conf. on Robotics and Automation*. NJ, USA, 2009.
[17] "Intel Xeon." [Online]. Available: http://en.wikipedia.org/wiki/Xeon
[18] "GeForce 8 series." [Online]. Available: http://www.nvidia.com/page/geforce8.html
[19] "Virtex6 Data sheet." [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds152.pdf