A Reconfigurable Computing Framework for Multi-scale Cellular Image Processing

Reid Porter, Jan Frigo, Al Conti, Neal Harvey, Garrett Kenyon, Maya Gokhale

Abstract

Cellular computing architectures represent an important class of computation that are characterized by simple processing elements, local interconnect and massive parallelism. These architectures are a good match for many image and video processing applications and can be substantially accelerated with Reconfigurable Computers. We present a flexible software / hardware framework for design, implementation and automatic synthesis of cellular image processing algorithms. The system provides an extremely flexible set of parallel, pipelined and time-multiplexed components which can be tailored through reconfigurable hardware for particular applications. The most novel aspects of our framework include a highly pipelined architecture for multi-scale cellular image processing as well as support for several different pattern recognition applications. In this paper we will describe the system in detail and present our performance assessments. The system achieved speed-up of at least 100x for computationally expensive sub-problems and 10x for end-to-end applications compared to software implementations.

1 Introduction

Cellular computing architectures, like Cellular Automata, are characterized by simple processing elements, local interconnect and massive parallelism [1]. These architectures can be significantly accelerated using hardware platforms which can exploit fine grain parallelism. In the 1980's these platforms were custom chip, or multi-processor based systems [2], [3]. In the 1990's these platforms were Reconfigurable Computers (RCC) [4], [5]. RCC leverages commercial-off-the-shelf (COTS) devices and promises to provide increased design longevity compared to custom hardware solutions. Cellular automata (CA) are perhaps the simplest cellular architecture and are typically characterized by a regular array of Boolean processing elements with local interconnect. With increases in reconfigurable device capacity, several RCC-based frameworks for Cellular Automata implementation have been proposed [6], [7], [8].

Cellular architectures are an excellent match for many image and video processing applications [9]. For example, a two-dimensional array of linear processing elements with local interconnect implements a convolution. One of the first RCC implementations of convolution was on the Splash-2 [10]. Cellular Nonlinear Networks (CNN) [11] define a more general class of cellular architecture for image processing where the processing element contains both linear functions and nonlinear functions similar to a neural network. CNN have been implemented with custom analog devices [12], and with RCC [13]. The basic two-dimensional cellular architecture can be extended to a three-dimensional array, where each layer (a two-dimensional array) interacts with other layers, to perform more complex tasks. This multi-layered architecture is a good match for image processing where complex algorithms are often decomposed into a sequence of primitive operations. Several RCC-based frameworks have been proposed for this type of architecture [14], [15], [16], [17]. A multi-layered CNN for retina modeling was implemented with RCC by Nagy and Szolgay [18]. In previous work we implemented a generalized multi-layered convolution neural network for multi-spectral image classification [19].

To solve complex image processing tasks most traditional image processing algorithms depart from the cellular architecture paradigm. For example, many algorithms require multiple passes through an image and communication is not local e.g. histogram techniques and connected component labeling. One way to extend a cellular architecture to solve more complex image processing tasks is to implement a multi-scale hierarchy. This is a multi-layered cellular architecture in which the number of processing elements is incrementally reduced from one layer to the next. Multi-scale hierarchies can provide many of the benefits of global communication, while maintaining many of the advantages of local, low latency, communication. The Neocognitron was one of the first multi-scale cellular architectures proposed for image processing and was inspired by biological models of cat retina [20]. Multi -scale cellular architectures are now an active research area and many researchers are working to apply them to increasingly complex visual tasks [21], [22].

This paper describes a RCC-based framework for multi-scale cellular architectures called HARPO: Hierarchical Architectures for Rapid Processing of Objects. As far as we are aware, this framework provides a novel implementation of multi-scale architectures in which multiple scales are processed in parallel. Existing custom hardware [23] and RCC [24] multi-scale frameworks process different image scales in different execution passes. The image is sub-sampled, by modifying the memory address generator, and the image processing pipeline is applied to a reduced data volume. Since data volume typically reduces by a factor of four at each scale the total execution time with this approach is 1.33 times greater than for a single image pass. In our framework, we process multiple scales within a single processing pipeline. This means all scales can be processed in the time taken for a single image pass (with some increase in latency). Pipelining has been suggested for existing multi-scale implementations, but the performance improvement is negated by the fact that processing units executing at reduced scales are not operating at full capacity [23]. In an RCC based system we can customize the implementation at run-time and obtain close to 100% utilization. Pipelined multi-scale processing is particularly useful for cellular image processing for two reasons:

1. Many cellular image processing solutions have strong local dependencies between different scales [25]. It is possible that without parallel execution of multiple-scales it will be difficult to exploit the parallelism within a single scale.
2. In most multi-scale cellular architectures the number of processing layers is increased as the scale is reduced [20]. This means that the total data volume that must be processed does not decrease and hence 100% utilization is essential.

In Section 2 we introduce the fundamental building block for cellular image processing algorithms: local neighborhood functions. We describe the most common operations, provide a brief overview of implementation strategies, and motivate the HARPO design choices. Due to the local communication constraints, algorithm development for cellular image processing is typically more difficult than for traditional image processing. One solution is to use unsupervised and supervised learning algorithms as part of algorithm development. In Section 3 we describe the HARPO system and describe how top-level software and hardware components interact as a practical image processing tool via supervised learning. The details of the RCC API and implementation are provided in Section 4. HARPO aims to automatically build top-level hardware pipelines in VHDL from high level specifications. This requires highly parameterized hardware modules with accurate timing and resource utilization estimators which are described in Section 5. The system performance is assessed for large multi-layered, multi-scale applications in Section 6. We conclude in Section 7 with a discussion of future work.

2 Background

Local neighborhood functions, or sliding window functions, are the fundamental building blocks for cellular image processing. These functions are applied at a particular pixel location and their output depends on a finite spatial, temporal and spectral neighborhood. Typically the same function is applied to all pixel locations in parallel. Neighborhood functions include a large number of traditional image processing algorithms. Image functions such as spectral averaging, clipping, thresholding and pixel scaling can be considered a subclass of local neighborhood functions without spatial extent. Local neighborhood inputs can come from multiple input images such as color channels or spectral dimensions. Neighborhood functions can also receive multiple images in time (as in Figure 1) e.g. for finite impulse response filters, the neighborhood window has a finite temporal extent and slides through time as the function is applied at each step. For infinite impulse response filters the neighborhood window includes a finite number of state variables. When a neighborhood function is applied at the edge of the image, some inputs will be undefined. In our system we temporarily increase the input image size (*pad* the input image) by reflecting pixel values across each edge.

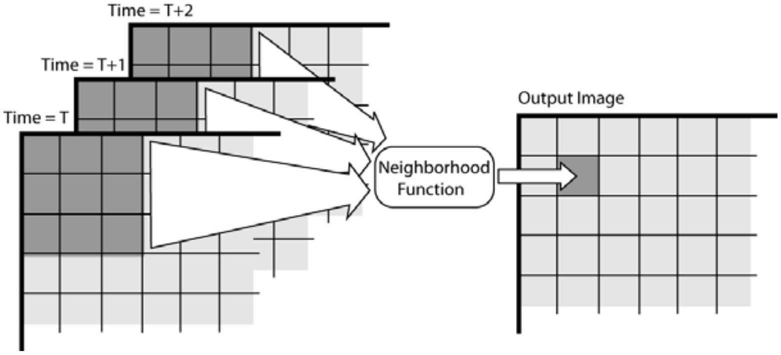


Figure 1: Example of local neighborhood function inputs and outputs.

2.1 Typical Neighborhood Functions

Perhaps the most well known local neighborhood function is convolution, defined in Equation 1. A multiplicative weight $W_{k,l}$ is associated with each location of the neighborhood k,l collectively known as the kernel K. The output $F_{i,j}$ of the function is the accumulated weighted sum of the kernel applied at each pixel location i, j of the image I :

$$F_{i,j} = \sum_{k,l} W_{k,l} I_{i\text{-}k,\,j\text{-}l} \qquad \text{Equation 1}$$

In many cellular image processing architectures, such as cellular automata and cellular nonlinear networks, neighborhood functions update state variables associated with each processing element, or cell. For example, in Equation 2 the state variable $F_{i,j}(t)$ is update based on a weighted sum of the input image and a weighted sum of the state variables within a local neighborhood. In traditional CNN formulations state variable update is a continuous function, but we restrict our attention to discrete time updates suitable for RCC implementation.

$$F_{i,j}(t+1) = \sum_{k,l} W_{k,l} I_{i\text{-}k,\,j\text{-}l} + \sum_{k,l} V_{k,l} F_{i-k,\,j-l}(t) \qquad \text{Equation 2}$$

To maintain full precision with fixed point arithmetic, neighborhood functions such as Equation 1 and Equation 2 must increase the pixel bit width. This can be a problem when neighborhood functions are cascaded. Thresholding is an important class of nonlinear operation in image processing and is often used between successive local neighborhood functions. Thresholding can also be used to reduce the data bit-width. In the HARPO system we implement a flexible piecewise linear thresholding function defined in Equation 3 and Equation 4. The input data is first shifted by a constant w_1. Second, an absolute value is conditionally applied depending on the value of a Boolean parameter abs. Third, the data is shifted by a second constant w_2. Finally, the data is shifted by $m \in \{0..5\}$ bits and values out of a pre-specified range are clipped.

$$F_{i,j} = \begin{cases} S\left(I_{i,j} + w_1\right) & if\ abs = 1 \\ S\left(\left|I_{i,j} + w_1\right| + w_2\right) & otherwise \end{cases} \tag{Equation 3}$$

$$S\left(x\right) = \begin{cases} Max\,range & if\ x > Max\,range \\ Min\,range & if\ x < Min\,range \\ {x}/{2^m} & otherwise \end{cases} \tag{Equation 4}$$

Mathematical morphology defines another large family of nonlinear neighborhood functions used extensively in image processing. In this case, the weighted sum in Equation 1 is replaced with an order statistic. In morphology the kernel K is also known as a structuring element and it defines which pixels are included in the order statistic. The simplest filters are erosion and dilation. Erosion is defined as the minimum value pixel from the neighborhood window (see Equation 5) and dilation is the maximum (see Equation 6). Each weight $W_{k,l}$ is a Boolean flag and indicates which neighborhood pixel values are included in the maximum (or minimum) operation. Another common morphological filter is the median filter. Morphological functions are closely related to digital logic functions. They avoid multipliers and can be efficiently implemented with RCC [26].

$$F_{i,j} = \forall_{W_{k,l}>0} Min\left(I_{i\text{-}k,\,j\text{-}l}\right) \tag{Equation 5}$$

$$F_{i,j} = \forall_{W_{k,l}>0} Max\left(I_{i\text{-}k,\,j\text{-}l}\right) \tag{Equation 6}$$

2.2 Implementing Neighborhood Functions

Local neighborhood functions demand exceptionally high bandwidth to data. For example, for a modest 3 band 256 pixel wide by 256 pixel high color video sequence, a (typical) 7 by 7 spatial neighborhood size and a 3 frame temporal window, would require access to 441 pixel values at each image location to implement the most general neighborhood function. To obtain real time processing rates at 30 frames per second would require access to approximately 870 million pixels per second. Most image processing applications are composed of large numbers of local neighborhood functions and therefore the communication bandwidth requirement quickly exceeds what general purpose computing can provide. Fortunately, due to the regular nature of the communication across the image array, there are also many opportunities to optimize the bandwidth. Reconfigurable computers are ideal platforms to tailor high-bandwidth memory hierarchies and implement algorithm specific address generation. There are two main ways that local neighborhood functions are implemented:

Data Parallel: Many implementations exploit the parallel nature of cellular array directly by implementing a large number of processing elements, each associated with a specific pixel location. At the extreme of this approach all pixels are processed in parallel, each cell has direct access to neighboring pixels at all times, and the entire array is updated in 1 clock cycle. This implementation is illustrated on the left in Figure 2. RCC can implement a programmable, maximally parallel implementation of a cellular array, but can only efficiently implement a small numbers of cells. Large arrays require time multiplexing, and the time taken to initialize the array and communicate results is typically much larger than the computation time.

Instruction Parallel: When implementing synchronous systems (as we are in RCC) it is also possible to achieve parallelism via pipelining. At the extreme of this approach only one cell is implemented for an entire array and each pixel location within the array is updated sequentially. The output stream from the first array is then directly tied to the input stream of a second array and therefore a long chain of instructions can be executed each clock cycle. Each instruction must access its local neighborhood at the same time and therefore image data must be cached within the computing device in large shift registers. This implementation is illustrated on the right in Figure 2.

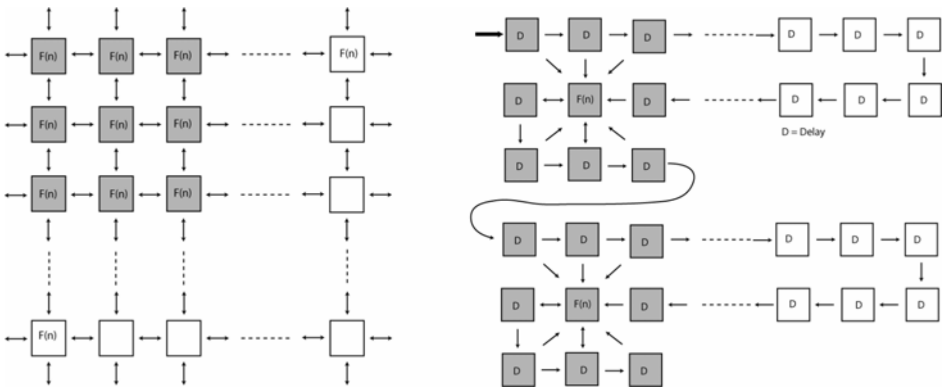


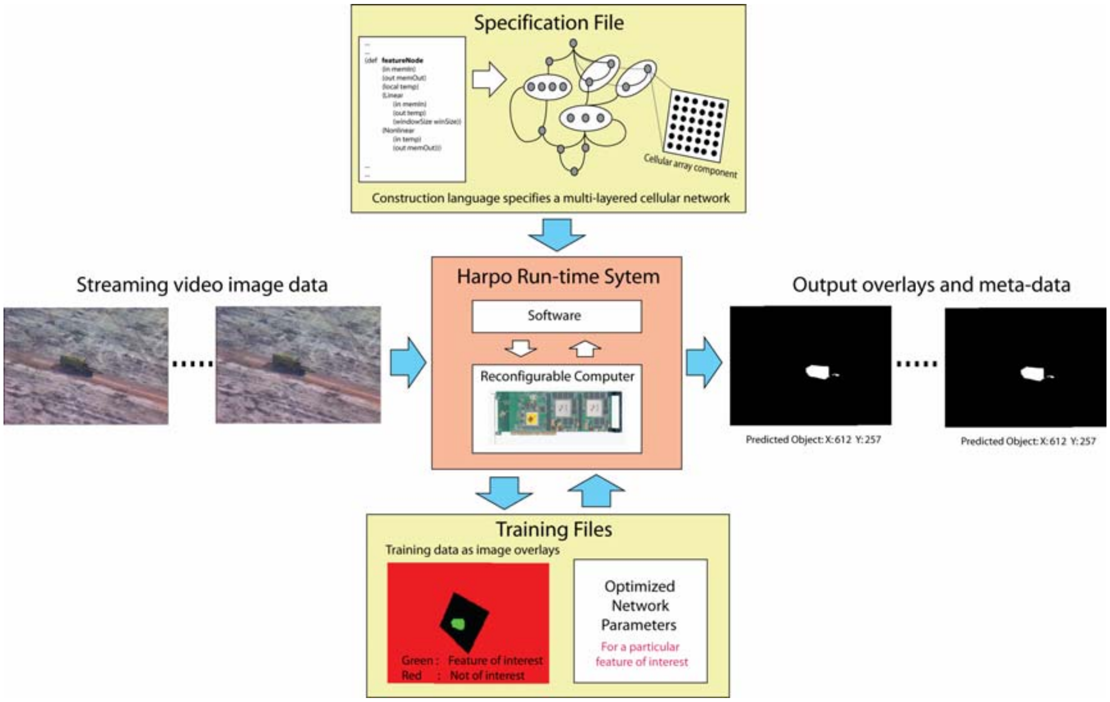
Figure 2: Data versus Instruction Parallel

We have described the two extremes of a pixel parallel versus instruction parallel design space. In practice any combination of these two extremes can be used. The optimal design point is dictated largely by the memory bandwidth that is available both on-chip and off-chip. Most modern FPGA devices now offer large quantities of on-chip memory which is ideal for implementing the large shift registers required in pipelining. By exploiting internal memory bandwidth, an instruction parallel design places less demand on external memory bandwidth (only 1 pixel / clock cycle is required). The instruction parallel approach has some inefficiency due to pipeline latency and edge conditions. For an image W pixels high by W pixels wide, each instruction introduces a αW latency, where α is proportional to the local neighborhood size. Also, for each neighborhood instruction the valid output image is reduced in size by α pixels due to edge conditions. For P pipelined instructions this means approximately $P\alpha W$ pixel calculations are invalid. The total execution time for the pipeline is W^2 and therefore for practical image sizes, $W \sim 1000$, the impact on performance is usually not significant.

The HARPO framework adopts the instruction parallel approach at the extreme and reads one pixel location from memory each clock cycle. This read includes multiple pixels, but each pixel comes from a different input image. The main motivation for this approach is to accelerate extremely large networks where the external bandwidth would be completely utilized by a large number of input images. Depending on the application this design choice may not be optimal and external bandwidth and computational resources may go unused. In this case it would be relatively straight forward to generalize the HARPO system to include data parallel pipelines.

3 System Overview

An overview of the HARPO system is shown in Figure 3.



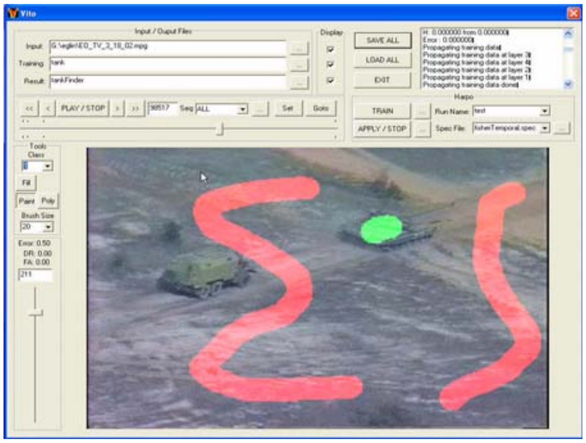
```
...
...
(def featureNode
     (in memIn)
     (out memOut)
     (local temp)
     (Linear
         (in memIn)
         (out temp)
         (windowSize winSize))
     (Nonlinear
         (in temp)
         (out memOut)))
...
...
```

Specification File
Cellular array component
Construction language specifies a multi-layered cellular network
Streaming video image data
Harpo Run-time Sytem
Software
Reconfigurable Computer
Output overlays and meta-data
Predicted Object: X: 612 Y: 257
Predicted Object: X: 612 Y: 257
Training Files
Training data as image overlays
Green : Feature of interest
Red : Not of interest
Optimized Network Parameters
For a particular feature of interest

Figure 3 – HARPO system overview

HARPO operates in two modes: *run-time* mode and *training* mode. In the *run-time* mode HARPO is applied as an online system. Video frames are loaded sequentially and passed to the system one at a time. After a fixed number of frames latency (the latency depends on the application) HARPO will start producing an output video stream. The output is a video overlay where high pixel values indicate objects or features of interest e.g. particular vehicles, land cover etc. HARPO can also produce additional meta-data such as predicted locations, percentage cover etc. which accompanies the output video stream. Figure 3 shows the typical video input and output frames for a vehicle segmentation task.

In the *training* mode HARPO receives additional training images. An example training image is shown in the bottom left of Figure 3. These image overlays are associated with a number of frames in the input video stream. During training mode the HARPO system uses supervised learning methods to minimize error between the output overlays and the training images. Training image overlays are typically generated by the user with the HARPO Graphical User Interface (GUI) shown in Figure 4. The GUI allows the user to navigate video sequences and using a collection of *Paint Program* tools (like paint brush and polygon), specify features of interest in green and non interesting features in red. For the two class classification problem features of interest are indicated with green markup and examples of the non-feature (or background) are indicated by red markup. Pixels that are not marked-up, and left black, are assumed 'don't care' and do not contribute to error during training. The user can provide training mark-up for as many frames as they wish for any particular problem. The HARPO system translates red and green mark-up into +1 and -1 class labels for a standard classification problem.



Input: G:\eglin\EO_TV_3_18_02.mpg
Training: tank
Result: tankFinder
98517 Seg: ALL
Run Name: test
Spec File: fisherTemporal.spec
Class: 6
Brush Size: 20
Error: 0.50
DR: 0.00
FA: 0.00
211

```
H: 0.000000 from 0.000000
Error : 0.000000
Propagating training data
Propagating training data at layer 3
Propagating training data at layer 4
Propagating training data at layer 2
Propagating training data at layer 1
Propagating training data done
```

Figure 4: Screen-shot of video mark-up tool used to specify features of interest.

HARPO provides a very flexible collection of software and hardware components to solve feature extraction problems. The system depends upon a tight coupling between a host processor and the RCC to implement these components. The host processor is responsible for image and video file I/O, control and synchronization of the RCC. The host processor also maintains a library of software modules which provide bit-accurate implementations of all HARPO hardware components. At any point in time the host processor can execute any network entirely in software, but typically, a network will be implemented with some combination of software and hardware components.

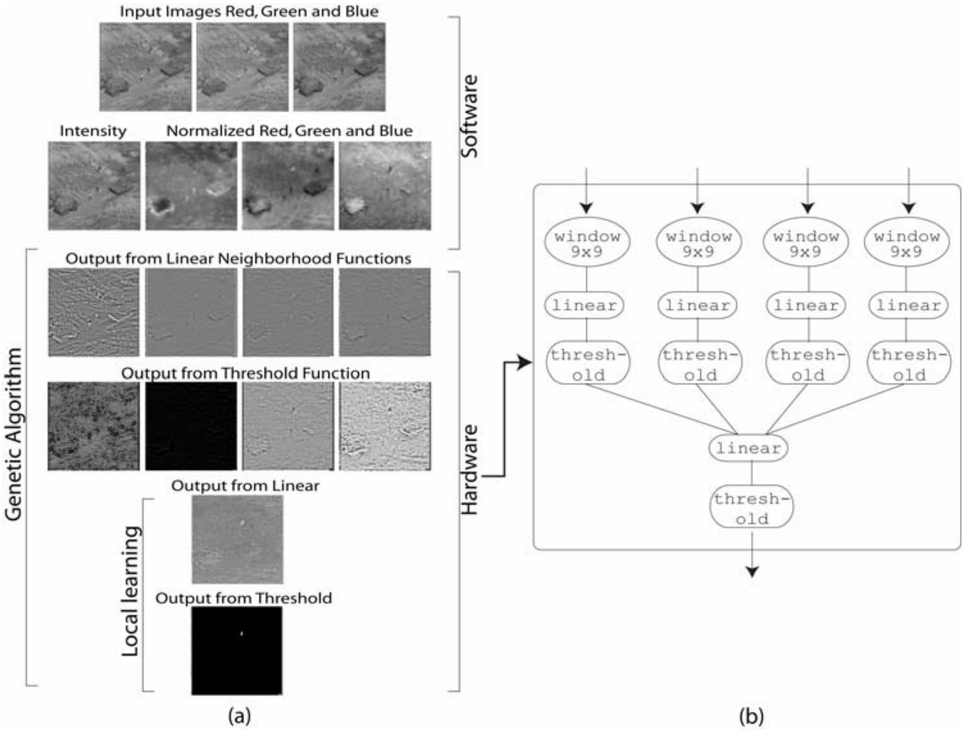


Figure 5: HARPO network example. a) Output images from processing layers and b) hardware components used in RCC sub-network.

Figure 5 presents a typical cellular algorithm used to solve feature extraction problems. The basic building block is a two-dimensional grid of processing elements which we call a processing layer, or, layer. Processing layers can be implemented in software or hardware and include, amongst other things, the neighborhood functions described in Section 2.1. The output images produced by each processing layer are shown in Figure 5a. The first four layers, executed in software, calculate an intensity image and three intensity normalized color images. These layers require square root and division. The four output images are sent to the RCC and the remainder of the network is executed in hardware. The partitioning between RCC and software is currently performed by the user. Figure 5b shows the hardware sub-network in more detail. First four different convolutions (linear neighborhood functions) are applied. Each has a 9 by 9 window. Second a threshold operation is applied to each convolution output. Finally, the four thresholded outputs are combined with a 4 input linear function and then a final threshold applied. The RCC implementation of these components is described in detail in Sections 4 and 5.

The RCC and software components are tailored for a particular application using two levels of configurability. The first level of configurability, shown at the top in Figure 3, is the Specification File. The specification file defines the solution space for a particular problem class e.g. vehicle detection or terrain type. The second level of configurability, shown at the bottom of Figure 3, is the Parameter File. The parameter file defines the additional parameters and settings that solve problem instances e.g. truck versus car detection. For a particular specification file, there are multiple parameter files, each tuned to particular features of interest. Parameter File values are usually derived using supervised learning algorithms.

One of the main motivation for the two levels of configurability is the long reconfiguration times associated with large RCC devices (84us for the Virtex 2 Pro 100 FPGA). Different specification files will typically use different RCC bit-streams. Different parameter files use the same bitstream. The 1st level of configurability requires VHDL to be generated and synthesized using vendor tools, and hence, is performed offline. Typically these bit-streams are loaded once when HARPO is first initialized. The 2nd level of configurability involves writing on-chip registers, control lines and other types of programmed flexibility. The 2nd level of configurability is very quick and is performed online as the application is running. The two levels of configurability are illustrated in Figure 6.

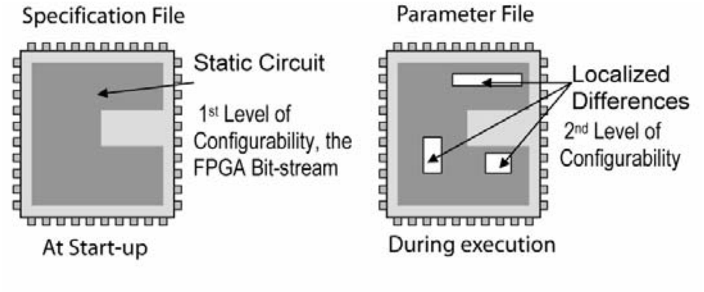


Figure 6: The two-levels of configurability

In the next two sections we describe how the two levels of configurability are both expressed and tailored in more detail.

3.1 Network Specification File

The Specification file is a compact description for large and complex multi-layered cellular networks. A processing layer can receive an arbitrary number of images as input and produce an arbitrary number of images as output. The connectivity between layers define data paths that pass image planes or arrays of image planes from one layer to the next. We use a simple scripting language called Scale Language (SL) to specify the particular layers and the data paths. In its present form, SL uses a simple s-expression syntax. This is a matter of convenience for parsing, and could easily be replaced by a more procedural syntax with infix notation. The important features of the language are:

- declare data planes and vectors of data planes,
- define new composite layers with parameters and data planes as input and data planes as output,
- use arbitrary arithmetic constructs, especially to index vectors of data planes,
- replicate arbitrary objects in space and time,

- concisely express hierarchical pipelines.

At the most basic level, the language has constructs to allocate image planes and to instantiate and interconnect layers using those planes. There are two types of input for each layer: data planes and parameters. Each layer returns data planes as output. A simple layer such as image differencing takes two input planes and returns an output plane. More complex layers might take vectors of image planes as input in addition to various parameters, and return vectors of image planes. Often a layer might accept and/or return a vector of arbitrary length, where each component of the vector is itself an image plane. Parameter semantics of course depend on the layer. Some layers (eg. `Linear`) accept a `trainFlag` keyword that indicates whether the layer should find coefficient values via learning when the system is in training mode.

SL allows the user to define new layers as compositions of pre-defined and primitive layers. Figure 7 shows the definition of a new `simple-pipe1` layer. The keywords `input` and `output` indicate data planes as input or output respectively. The keyword `local` signifies data planes local to the operator. Vectors of data planes can also be declared, as illustrated with the local variable `pipe` that is a vector of two planes indexed from 0 to 1. In this example, three pre-defined layers are instantiated, two Linear layers and a Threshold.

```
(def simple-pipe1
     (input mem-in)
     (output mem-out2)
     (local (pipe 0 1))
     (Linear (input mem-in) (output (index pipe 1)) (trainFlag 1))
     (Linear (input (index pipe 1))
             (output (index pipe 2))
             (paramType     DoG)
             (windowSize    11)
             (centerWidth   0.05)
             (surroundWidth 1.0)
             (DoGRatio      10.0))
      (Threshold (input (index pipe 2))
                 (output mem-out2)
                 (trainFlag             1)
                 (threshType            linear))
      )
```

Figure 7: Specification example for a simple pipe

As the example shows, some layers are extremely versatile, taking a large number of optional parameters. The first instance of Linear simply specifies input and output planes along with the `trainFlag` set to 1, whereas the second instance is much more tailored. It specifies a `paramType` (parameter type) of `DoG` (Difference of Gaussians), a `windowSize` of `11`, as well as several additional parameters. This example also illustrates how the operators are interconnected. The first instance of Linear takes as input the `simple_pipe1` input plane called `mem-in`. Its output is `pipe[0]`, which is input to the second instance of `Linear`. Similarly, the second `Linear` outputs `pipe[1]`, which in turn is input to `Threshold`. The `Threshold` instance also uses indexed data planes and operator-specific parameters.

To allow the user to specify a collection of layers, the Scale Language includes a `replicate` construct, as illustrated in Figure 8. In this example, a new layer `increase-scale-unit` is defined. This operator takes as input a parameter `n` as well as a data plane `mem-in` and returns a data plane `mem-out1`. It has a local vector `mem-loc` of data planes. The size of the vector is `n`, the input parameter to the layer, allowing the size of the vector to be specified by any layer that instantiates `increase-scale-unit`. Different instantiations with different values for `n` will result in different sizes of `mem-loc`. The `set` statement is an assignment setting `mem-loc[0] = mem-in`. The `replicate` statement takes as the first argument a range of replications. The first parameter (`i` in the example) is an identifier that is bound to each index in the replication range. Following is a list of layers to be instantiated, each of which is replicated over the range. In the example, there is a single layer `incr-scale` that takes as input `mem-loc[i-1]` and returns `mem-loc[i]`. This is a concise definition of an arbitrary size increase-scale-unit. The example illustrates a spatial replication. Replication can also occur in the time domain by replacing the keyword `replicate` with `loop`.

```
(def increase-scale-unit (param n)
     (input mem-in )
     (output mem-out1 )
     (local ( mem-loc 0 n))

     (set (index mem-loc 0) mem-in)
     (replicate (i 1 n)
                (incr-scale
                 (input (index mem-loc (- i 1)))
                 (output (index mem-loc i))
                 )
       )
```

Figure 8: Specification example for Replicate

HARPO assumes a feed-forward pipeline and will execute layers in the order they appear in the SL specification. In video applications the HARPO run-time system loads a frame and associates the data with the top-level `memIn` identifier. The network is executed and HARPO then writes out the images referenced by the top-level `memOut` identifier. To build networks that exploit the time domain and/or introduce state variables the user must explicitly instantiate a predefined layer called a `Buffer.` Figure 9 shows how the `Buffer` layer is used to implement a three frame temporal filter. State variables can be implemented in a similar way by using a layer's buffered output as one of its inputs. Since temporal processing is entirely in the hands of the user it can be customized for various modalities and data types without adding complexity to the HARPO run-time system.

```
(def FIR-filter (input memIn) (output memOut)
(local temp)
(Linear (input memIn) (output memOut)
        (weights ~1))
(Buffer (input (index memIn 1)) (output (index memIn 2)))
(Buffer (input (index memIn 0)) (output (index memIn 1))))

(def main-net (input memIn)(output memOut)
(evolve (FIR-filter (input memIn) (output memOut))))

(local (memTaps 0 2))
(local memIn)
(local memOut)
(set (index memTaps 0) memIn)
(main-net  (input memTaps) (output memOut))
```

Figure 9: Specification example for evolve

3.2 Network Parameter File

The second level of configurability tailors a network specification for a particular problem instance by modifying layer parameters at run-time. For example, a `Linear` layer has a number of integer weights defined in Equation 1 and a morphological layer has Boolean valued weights defined in Equations 5 and 6. Parameters have default values but can also be given specific values via the Specification file. Alternatively, parameter values can be derived in *training mode*. HARPO supports two types of training: layer specific local learning and general purpose global learning. Local learning algorithms must be provided by the user who develops the layer and will typically affect a predefined subset of parameters. For example, for one `Linear` layer in Figure 5 the `trainFlag` keyword is `TRUE` and weights will be optimized with Fisher's linear discriminant. There are also several general purpose training algorithms defined in the HARPO runtime system which can be applied to any type of layer. For example, in Figure 9, the `evolve` keyword indicates that free parameters found in the `FIR-filter` sub-network should be optimized with an Evolutionary Algorithm. HARPO uses the GAlib genetic algorithm package, written by Matthew Wall at the Massachusetts Institute of Technology for these algorithms. A useful strategy for optimizing network parameters is to use a combination of global learning and local learning algorithms. For example, for each evaluation of the GA in figure 5, parameter values for the four convolutions and threshold functions are randomly generated, applied to the input image, and then combined with a Fisher discriminant and a minimal error threshold. A population of 20 candidates evolved for 20 generations (with a simple generational GA) to obtain the images shown in Figure 5a.

4 Hardware Overview

We have provided a broad overview of the HARPO system as an application and have outlined how cellular algorithms can be specified and optimized to solve particular tasks. In this section we provide more detailed description of how the RCC interacts with the system. In our prototype the host processor is a conventional desktop workstation and the RCC is attached through a global PCI bus. This may not be the best configuration and to support alternatives we have defined both hardware and software application programming interfaces (API).

4.1 Hardware API

The hardware API abstracts the underlying RCC and defines interfaces for top-level modules that are customized for each application. A block diagram of the API components is illustrated in Figure 10.

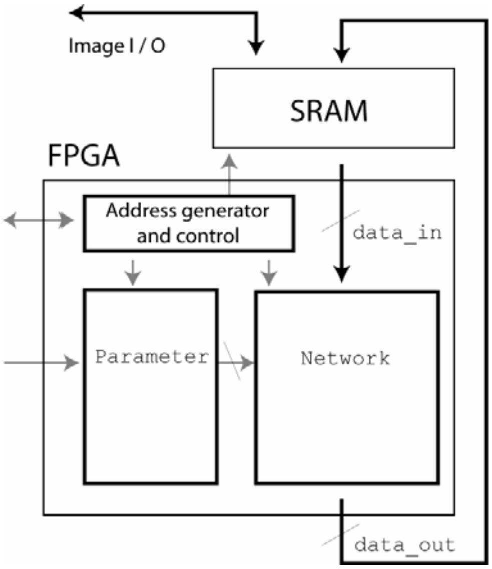


Figure 10: Top level hardware architecture

The RCC memories are interfaced to the `Network` as one large dual port memory. Each clock cycle the `Network` expects new data on `data_in` and produces new data on `data_out`. The address generator provides the same read address to all memories each clock cycle, which typically corresponds to raster scan order. The only deviation from sequential access is when the address generator is configured (via control registers) to automatically *pad* the input images as described in Section 2. The address generator also provides the same write addresses to all RCC memories each clock cycle. As the output images are produced only valid, *non-padded* data is written to memory. When padding is used the address generator also introduces latency into the pipeline to ensure that memory writes do not overwrite the memory contents before they are read.

The HARPO hardware system currently supports the WildStar II Pro (WS2Pro) Reconfigurable Computer from Annapolis Microsystems (AMS) [27]. The WS2Pro has two Xilinx Virtex2P100-5 FPGAs with six 8 Mbyte quad data-rate SRAMs surrounding each chip providing a total dual port bandwidth of 192 bits per clock cycle. The board communicates to the host workstation via a 64 bit, 66 MHz PCI bus. The HARPO system could be easily retargeted to other SRAM based RCC platforms; however SDRAM is not currently supported. The resources used to implement the HARPO hardware API on the WS2Pro are shown in Figure 11 . This includes the address generator, memory interfaces, control registers, and the DMA interface. All resource utilization results were obtained via Synplify's Synplicity v8.0 synthesis and Xilinx's ISE Foundation 7.1 place and route.

	Resources	% Usage
On-chip RAM	4 / 444 Blocks	0.9
Logic	3187 / 44096 Slices	7.2

Figure 11: Resources used to implement the Hardware API on the WS2Pro RCC

4.2 Software API

We also define a software API for the HARPO system that abstracts the underlying host-RCC interface. The interface is defined as a C++ object and provides methods for initializing bit-streams (the 1st level of configurability) and writing parameter values (the 2nd level of configurability). Methods are also provided to read and write image data, configure address generation and control pipeline execution. The typical application control flow is illustrated with pseudo code in Figure 12. RCC initialization and bit-stream loading is typically performed once at application start-up. Large volume image transfers to and from the RCC are via DMA. For each image the host may execute the hardware pipeline several times. Each time different parameter values may be written from the host to configure the pipeline differently.

```
Reset and initialized RCC
Write control registers
For f = 1 to total_number_of_frames
     Write image data to RCC
     For i = 1 to number_of_passes
          Write parameter values
          Initiate processing
          Wait until done
     End For
     Read image data from RCC
 End For
Close RCC
```

Figure 12: HARPO Host Pseudo Code: Applications Control Flow

The performance measurements for transactions across the 64-bit / 66MHz PCI bus are shown in Figure 13.

	Mbytes/sec
DMA Read	223
DMA Write	298
Register Read / Write	256

Figure 13: WS2Pro RCC communication performance measurements

5 Hardware Components

In the next few sections we will describe the HARPO library components used to build the `Network` and `Parameter` modules. HARPO aims to automatically generate `Network` and `Parameter` modules from hardware library components. The most important design consideration was therefore modularity and flexibility. At the top-level the library is parameterized by:

`image_width` = image width
`pad_width` = number of pixel rows/columns used to pad image

`bit_width` = input data bit width

These quantities are fixed at run-time. The `image_height` can be changed at run-time and is limited only by the RCC memory depth. For the WS2Pro the SRAM depth is 2Mwords which means the maximum image height is $2M/\texttt{image\_width}$ pixels. We use two's complement fix bit representation throughout the design. The hardware sub-components can be roughly divided into four different types. The first deals with the local memory management required by neighborhood functions. We have developed a versatile and programmable `window` module which can be instantiated and used for any local neighborhood function at any scale. This module is described in Section 5.1. The second type of component includes specific neighborhood functions, such as convolution and is described in Section 5.2. The third type of component, described in Section 5.3, is used for data sequencing and includes down sampling and up sampling. The fourth component is used in the `Parameter` module and provides the interface between the `Network` module and the host processor to provide run-time configurability.

5.1 Neighborhood Memory Access

Figure 14 provides a block diagram of a pipelined neighborhood memory access for 3 by 3 neighborhood convolution. The image input arrives one pixel per clock cycle in raster scan order into three pipelined registers: $P_{3,3}, P_{3,2}, P_{3,1}$. The output from $P_{3,1}$ is input to a shift register, implemented on-chip, whose length is equal to `image_width` - 3. The strategy is repeated for the second row and the second shift register output fed into a final row of neighborhood registers $P_{1,3}, P_{1,2}, P_{1,1}$.

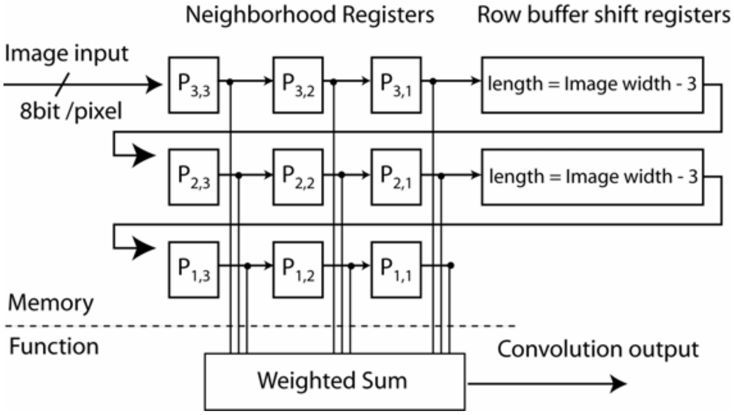
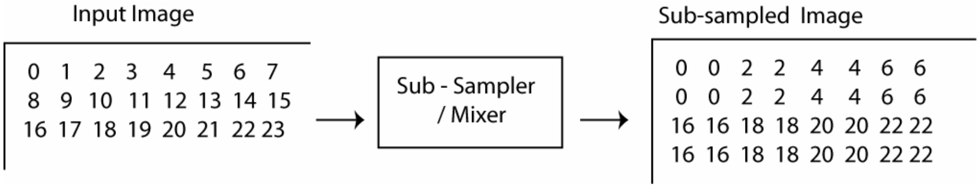


Figure 14: Interface for the neighborhood memory access.

The latency for the window buffer is given by Equation 7.

$$L_{win} = (\texttt{image\_width}) * ((\texttt{window\_width-1})/2) + (\texttt{window\_width-1})/2 + 1 \quad \textbf{Equation 7}$$

HARPO aims to cascade multiple neighborhood functions, each operating at different image resolutions, within a single pipeline. This means different parts of the pipeline process different quantities of image data. Parts of the pipeline that are applied to the highest resolution operate at 1 pixel per clock cycle and parts of the pipeline applied to reduced resolution data will have multiple clock cycles available per pixel. Figure 15 illustrates how sub-sampling affects the image pixel streams. Numbers in both images represent pixel values. Sub-sampling selects the upper left pixel from each group of four, and duplicates its value in both horizontal and vertical directions.



0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23

0	0	2	2	4	4	6	6
0	0	2	2	4	4	6	6
16	16	18	18	20	20	22	22
16	16	18	18	20	20	22	22

Figure 15: Block diagram of multi-scale window memory access.

The cycles in which data is duplicated can be used to compute multiple instructions per pixel. We generalize the pipelined neighborhood memory access to exploit this redundancy. This approach is illustrated in Figure 16 for an image reduced by a factor of two. This component is an array of carefully allocated shift registers made from flip flops and block RAM. Small shifters separate adjacent pixel values while block RAMs are used to delay data across the width of the image. This architecture produces a new window of data for a sub-sampled image each clock cycle.

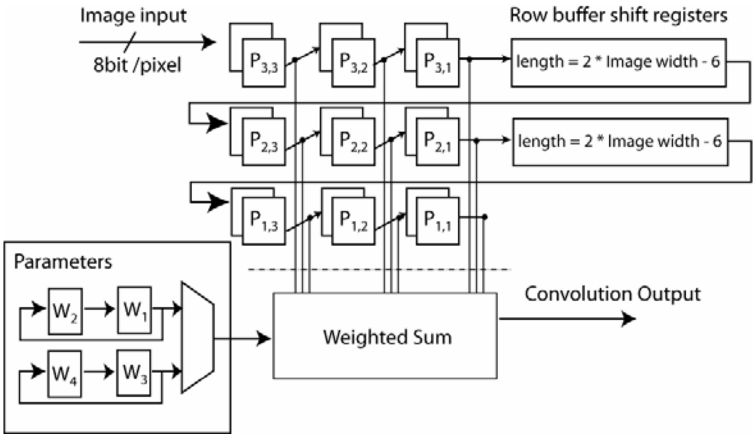


Figure 16: Block diagram of multi-scale window memory access.

The latency for this generalized window buffer is given by Equation 8.

$$L_{win} = \texttt{sample\_rate*}(\texttt{image\_width})\texttt{*}(\texttt{window\_width-1})/2 + \texttt{sample\_rate*}((\texttt{window\_width-1})/2+1) \qquad \text{Equation 8}$$

The interface for the `window` module is shown in Figure 17. It is a versatile, highly parameterized component which can be used within a multi-scale pipeline without delay lines or state machines. The hardware that is generated depends on the values of the generic parameters that must be known at compile time. `Sample_rate` sets the size of the internal shifters which changes the scale of the data which is routed to the output ports and operated on by either a compute function or data sequencer.

```
component window
  generic (
    window_width   : natural;
    window_height  : natural;
    sample_rate    : natural;
    opt            : natural);
  port (
    clk            : in std_logic;
    reset          : in std_logic;
    run            : in std_logic;
    data_in_valid  : in std_logic;
    data_out_valid : out std_logic
    data_in        : in std_logic_vector(bit_width-1 downto 0);
    data_out       : out std_logic_vector(window_width*window_height*bit_width-1 downto 0);
  );

end component;
```

Figure 17: Interface for generalized window component.

An image that has been down sampled by a factor of two has a data redundancy factor of four. In practice, the memory windows that will be propagating redundant data can be identified at compile time. Opt is a parameter that selects one of two implementation optimizations which can minimize the amount of logic and block RAM used for shifting redundant data. In the first optimization a by-sample_rate clock divider is used to slow the propagation of pixel data thus cutting the amount of necessary Block RAM by a factor of sample_rate. The second optimization works on the same principle. A by-sample_rate clock division in combination with a special type of data sequencer reduces the amount of necessary block RAM by sample_rate*sample_rate.

Figure 18 shows the resource and post place and route timing of a standard (no optimization) 3 by 3 window while varying data bit_width. Figure 19 shows how the window component scales as a function of two other generic parameters, window dimension and opt. For these experiments we fix sample_rate=2. A target frequency of 100 MHz and the default placement effort was used. Logic utilization scales linearly as a function of the window size while block RAM scales linearly with the window dimension.

	8 bits	16 bits	32 bits
clk frequency	116 MHz	109 MHz	105 MHz
slices	224	346	590
blockRAM	1	2	4

Figure 18: Resource utilization and timing of window component as bit width is increased.

		opt 0 (standard)	opt 1 (reduce by 2)	opt 2 (reduce by 4)
3x3	clk frequency	116 MHz	106 MHz	57 MHz
	slices	224	168	114
	block RAM	1	1	1
4x4	clk frequency	123 MHz	125 MHz	51 MHz
	slices	375	257	141
	block RAM	2	1	1
5x5	clk frequency	110 MHz	107 MHz	70 MHz
	slices	566	376	259
	block RAM	2	1	1
6x6	clk frequency	109 MHz	101 MHz	61 MHz
	slices	800	652	303
	block RAM	3	1	1
7x7	clk frequency	102 MHz	111 MHz	66 MHz
	slices	1076	693	432
	block RAM	3	2	1
8x8	clk frequency	110 MHz	103 MHz	57 MHz
	slices	1534	890	496
	block RAM	3	2	1
9x9	clk frequency	100 MHz	101 MHz	52 MHz
	slices	1758	1118	661
	block RAM	4	2	1
10x10	clk frequency	101 MHz	101 MHz	53 MHz
	slices	2211	1418	743
	block RAM	4	2	1

Figure 19: Resource utilization and timing of window component as window width increases and optimizations are applied.

5.2 Neighborhood Functions

The window component is connected to at least one pipelined neighborhood function. These functions must adhere to the standard interface defined in Figure 20. The functions implemented in the current system include convolution, thresholding and morphology. This is a very limited set of functions since our effort has focused on combining these functions in a flexible way. Implementing additional neighborhood functions could be easily automated with existing C to VHDL compilers since the neighborhood function has been separated from the neighborhood memory access.

```
component function
  port (
    clk       : in std_logic;
    valid_in  : in std_logic;
    valid_out : out std_logic;
    param_in  : in std_logic_vector(param_width-1 downto 0);
    data_in   : in std_logic_vector((num_inputs*bit_width)-1 downto 0);
    data_out  : out std_logic_vector(result_width-1 downto 0)
  );
end component;
```

Figure 20: Standard interface for a neighborhood function.

5.2.1 Convolution

The convolution function must compute a weighted sum of the input data every clock cycle. The library module currently exploits the hard MULT18x18s multiplier cores available in the Virtex2Pro. If the `bit_width` for the design is 18 bits, or less, then num_input multipliers will be used. If the `bit_width` is larger then multiple cores are cascaded for each multiplication. The latency for the convolution function is given by Equation 9.

$$L_{conv} = Ceil\left(\log_2\left(\texttt{num\_inputs}\right)\right) + 1 \qquad \text{Equation 9}$$

Excluding multiplications, resource usage is dominated by the adder tree and pipeline registers. Based on Register Transistor Logic (RTL) diagrams of the function unit we estimate resource utilization by Equation 10. This equation is based on the assumption that logic and registers are not packed into the same CLB.

$$Slices_{conv} = \texttt{bit\_width}\left(\texttt{num\_inputs}+2^{Ceil\left(\log_2\left(\texttt{num\_inputs}\right)\right)} - 1\right) \qquad \text{Equation 10}$$

5.2.2 Morphology

The morphological function implements a comparator tree each clock cycle. The latency for the function is given by Equation 11.

$$L_{conv} = Ceil\left(\log_2\left(\texttt{num\_inputs}\right)\right) \qquad \text{Equation 11}$$

At each node in the comparator tree two inputs are compared and one selected. The module is parameterized to select either the maximum or the minimum based on a flag. A binary weight associated with each input selects between either the input data or a constant value. Resource usage is estimated by Equation 12.

$$Slices_{morph} = \texttt{bit\_width}\left(\texttt{num\_inputs}+2^{Ceil\left(\log_2\left(\texttt{num\_inputs}\right)\right)} - 1\right) \qquad \text{Equation 12}$$

5.2.3 Threshold

The offsets, shifting, range correction, absolute value and clipping described in Section 2.1 are collectively called the `Threshold` function. The function has a latency of 4 and the resource usage is estimated by Equation 13.

$$Slices_{thresh} = 12.\texttt{bit\_width} \qquad \text{Equation 13}$$

5.3 Data Sequencing

We have described the basic computational units involved in the multi-scale pipeline. In this section we describe the additional components which are used to manipulate the pipeline data streams and enable flexible implementation of a large variety of network topologies.

5.3.1 Down Sampling

The `down_sampler` is used to reduce the resolution of an image by a factor `sample_ratio` which is fixed at compile time. This factor is restricted to powers of two. The `down_sampler` is a preceded by a `window` module with neighborhood size `sample_ratio` by `sample_ratio`. The function uses two levels of multiplexing to route `window` input data to the output port based on comparators and a state machine. Latency is two clock cycles for all possible instantiations of this component.

Figure 21 shows results from the synthesis of instantiations of the `down_sampler` module with ranging values for `bit_width` and `sample_ratio`. While the clock frequency remains relatively constant across the different instantiations, the logic requirement scales linearly as a function of the size of the source window.

	sample_ratio = 2	sample_ratio = 4	sample_ratio = 10
bit_width = 8	138 MHz, 52 slices	153 MHz, 85 slices	153 MHz, 356 slices
bit_width = 16	138 MHz, 81 slices	138 MHz, 144 slices	138 MHz, 699 slices
bit_width = 32	138 MHz, 141 slices	138 MHz, 269 slices	138 MHz, 1378 slices

Figure 21: Synthesis results for `down_sampler` component as `sample_ratio` and `bit_width` are varied

5.3.2 Stream Splitting

The `splitter` module de-interlaces a time-multiplexed image stream into independent, redundant image streams. For example, in Figure 22 an image has been down sampled by a factor of two by the `down_sampler` component. The image stream is input to a window function which computes four functions and produces four unique results for each pixel. The `splitter` module takes the time-interleaved result image as input and produces 4 independent output images. The module introduces redundancy into each stream so that they can be easily cascaded into subsequent window functions.

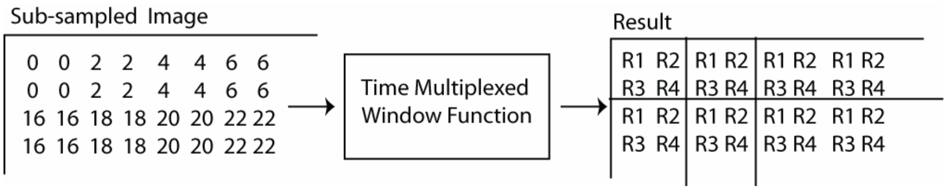


Figure 22: Block diagram of multi-scale window memory access.

Latency for the `splitter` component is two clock cycles for all possible instantiations of this component. Figure 23 shows synthesis results for instantiations of the `splitter` module with ranging values for `bit_width` and `sample_ratio`. Unlike the previous module, the maximum clock frequency decreases as the `bit_width` and `sample_ratio` increased.

	sample_ratio = 2	sample_ratio = 4	sample_ratio = 10
bit_width = 8	140 MHz, 110 slices	125 MHz, 307 slices	79 MHz, 2458 slices
bit_width = 16	128 MHz, 173 slices	113 MHz, 399 slices	64 MHz, 3064 slices
bit_width = 32	115 MHz, 303 slices	101 MHz, 713 slices	61 MHz, 4317 slices

Figure 23: Synthesis results for `splitter` component as `sample_ratio` and `bit_width` vary.

5.3.3 Stream Mixing

The `mixer` module is the inverse of the `splitter` module. A time interleaved image stream is created by sampling a number of independent image streams. The internals of this component are very similar to the other two sequencers and the added latency is two clock cycles. If the resolution of multiple image streams is to be reduced in parallel it is possible to combine the functionality of the `down_sampler` and and `mixer` components. We will see an example of this in Section 6. This lowers resource utilization and reduces routing. As of the current version of HARPO, this module has yet to be fully parameterized. Figure 24 indicates that clock rate is not affected by `bit_width.`

	sample_ratio = 2
bit_width = 8	138 MHz, 34 slices
bit_width = 16	138 MHz, 42 slices
bit_width = 32	138 MHz, 58 slices

Figure 24: Synthesis results for `mixer` component as `bit_width` varies.

5.3.4 Up Sampling

The final component required for most multi-scale cellular algorithms is up-sampling. There is no specialized hardware component for up-sampling since it can be implemented with the existing `window` component. Up-sampling is restricted to powers of two and the default implementation implements linear interpolation.

5.4 Parameter Module

The fourth and final type of hardware component is the `Parameter` module. Each neighborhood function has a number of adjustable parameters which can be configured at run-time. Parameters are stored as a collection of `sub_param` sub-components, each associated with a particular neighborhood function. When the `sub_param` component is associated with a time-multiplexed pipeline component the sub-components are responsible for shifting multiple parameters in sequence depending on the `sample_ratio.`

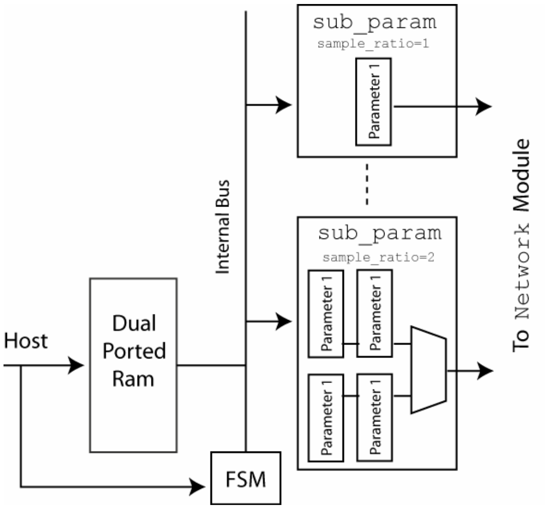


Figure 25: Block diagram of the Parameter module

A block diagram of the `Parameter` module is shown in Figure 25. The host processor crosses the bus and FPGA clock domains with a dual ported Block RAM. Through the `Parameter` state machine the host processor has the ability to reset or load any `sub_param` component. The state machine arbitrates between `sub_param` instantiations via a unique `parameter_id` which is assigned at compilation. The architecture enables as many `sub_param` components as are required for a particular design. The primary resources required by each `sub_param` component are the shift registers. The number of shift registers required is `sample_ratio.` The length of each shift register is `sample_ratio.` The width depends on the parameter width associated with each neighborhood function. Figure 26 shows results from place and route experiments as the `sample_ratio` and total number of parameter bits were varied. The clock rate was consistently above 100MHz.

sample_ratio	parameter bits	frequency	slices
1	20	104 MHz	103
1	40	108 MHz	144
1	60	107 MHz	183
1	80	122 MHz	224
2	20	105 MHz	221
2	40	117 MHz	338
2	60	118 MHz	478
2	80	118 MHz	596
3	20	105 MHz	301
3	40	131 MHz	445
3	60	130 MHz	579
3	80	134 MHz	724
4	20	131 MHz	384
4	40	128 MHZ	565
4	60	130MHz	746
4	80	120 MHz	925

Figure 26: Synthesis results for `parameter` component showing how timing and resource utilization scale with `sample_ratio` and the total number of parameter bits.

6 Application Case Studies

We have described all hardware components implemented within the HARPO framework and provided our assessment of component performance. In this section we describe two practical applications that have been implemented within the HARPO framework. The applications illustrate how components are combined within larger networks and they allow us to better quantify the performance of hardware sub-components when they combined into a larger RCC design. The networks are applied to practical pixel classification problems illustrated in Figure 27. The input is a single 704 by 480 three color image from airborne video camera (left) and contains three people and two vehicles. The first problem (middle) is to segment pixels belonging to a particular person. The second, more difficult problem (right) is to segment pixels associated with all people. In the training overlays white indicates pixel of interest, gray indicates background pixel and black indicates no training data for that pixel.

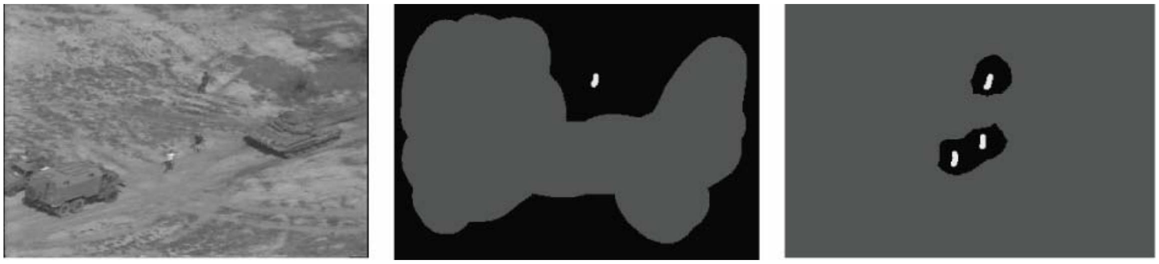


Figure 27: 3 color training image and training overlays for one individual (middle) and all individuals (right).

6.1 Application I

The first application was illustrated in Figure 5 of Section 3. The network was optimized for the first training problem in Figure 27. The network has both software and hardware components. The largest hardware components (in terms of resources) are four large 9 by 9 convolutions. For each convolution there are 81 8-bit parameter values stored within a `sub_param` component. Because convolutions are applied at the maximum resolution the `sample_ratio` is one and no parameter time-multiplexing is required. All parameter values are sent to the `Parameter` modules at start-up and remain constant. Resource utilization for the network is summarized in Figure 28.

	Number of Resources	**% Utilization**
Observed		
MULT18x18s	328	73.9
RAMB16s	12	2.7
Slices	12988	29.5

Figure 28: Multi-layer application resource utilization including hardware API.

Multipliers and Block RAM components are deterministic and easily predicted. In contrast, logic utilization proved difficult to predict accurately for all parameterizations. We synthesized the same network at various neighborhood window sizes and results are presented in Figure 29. Investigating the accuracy beyond 9 by 9 was not investigated due to lack of hard multiplier cores.

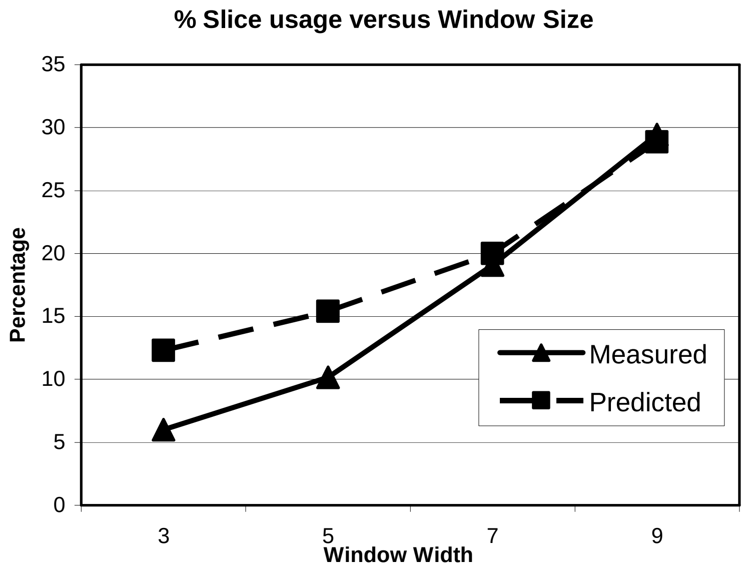


Figure 29: Comparison of predicted and measured resource usage as window size increases.

As shown in Figure 29 there is a significant difference between our predicted resource usage and the observed usage for small window sizes. Accurate resource prediction is difficult in RCC implementations due to the long vendor specific tool chain used to synthesize designs. We hypothesize that unpredictable packing of logic elements (utilizing both the look-up-table and the flip-flop within each cell) could be a significant contributor.

The software implementation was written in C++ and compiled with Microsoft Visual C++ version 7.1 with execution speed optimizations enabled (/O2 compiler flag). The software convolutions were implemented with `for` loops for each pixel, and for each neighborhood element. The software 9 by 9 convolution therefore executes the same number of operations as the hardware implementation: 9*9*704*480 multiply-accumulates. The host processor was a Xeon 3.18GHz workstation with 512KByte and 1MByte level 2 / 3 cache and 4Gbyte of main memory. We estimated throughput at 58.9 million multiply-accumulates per second for a 9 by 9 convolution.

The maximum clock speed for the hardware design, as reported in the Xilinx timing report, was 103MHz. Figure 30 provides a breakdown of execution time for the various hardware and software components. Of theoretical interest is the RCC pipeline execution time which was approximately 742x faster than the Xeon implementation. Of more practical interest is the sub-network speed-up when the host / RCC communication and RCC specific data processing is included. RCC specific data processing includes all computations that would not be required in a software implementation and includes packing and unpacking image data into DMA memory buffers. We estimate speed-up at 102x the Xeon. The final quantity of interest is the speed-up obtained for the complete application. The additional network layers took approximately one third of a second to execute in software. The application speed-up obtained by using hardware and software was approximately 9.6x software.

	Xeon 3.18GHz (s)	RCC (s) 100MHz	Speed-up (x)
Hardware Pipeline	2.969	0.004	742
Communication	-	0.010	
Data Preparation	-	0.015	
Sub-Total	2.969	0.029	102
Software layers	0.313	0.313	
Complete Application	3.282	0.342	10

Figure 30: Processing times for Application I with and without RCC.

6.2 Application II

The second application example extends the previous network by adding additional processing layers to the hardware sub-network. The most significant difference is the use of multi-scale processing components and time-multiplexed resource sharing. This application highlights a situation where 100% utilization is required to maintain throughput. That is, without resource sharing there would be insufficient multipliers to implement the network and two pipeline passes would be required.

Example output images from the network are shown in Figure 31a (images have been rotated for clarity). This application uses the same software processing layers that were used in application I and similar learning strategies. A population of 20 candidates evolved for 20 generations to solve the second problem in figure 27: detect all people in the image. The hardware sub-network is shown in figure 31b. The outputs from the first four threshold functions are down sampled by sub_sampler/mixer component. This produces a single stream with no redundancy. A 9 by 9 window function and threshold is then applied to each of the four streams with four different sets of parameters. The window and sub-param components are instantiated with a sample ratio of two. The time multiplexed results from the threshold function are then split into four redundant streams and then up-sampled via linear interpolation. The four up-sampled images are then combined with the original thresholded outputs (all eight images shown in figure 31a are combined) with a linear layer, and a final threshold. This shows how local dependencies between image data at different scales can be exploited in a single pass.

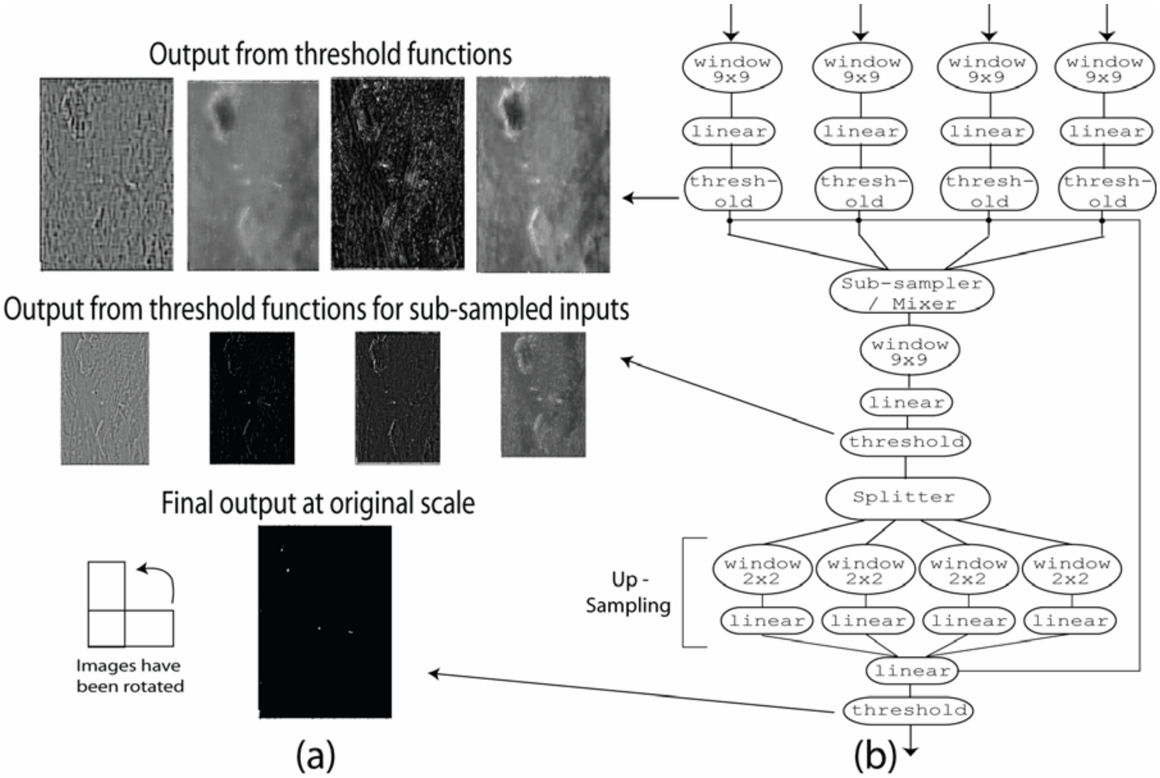


Figure 31: Multi-scale application. a) Output images from processing layers and b) hardware components used in the design.

Figure 32 shows the resource utilization for this pipeline within the HARPO framework. The Xilinx timing tool reported a maximum clock frequency of 68MHz. This is almost half the clock rate reported for individual pipeline components. The precise bottleneck in the design was difficult to determine but we hypothesize place and route tools had difficulty with the high percentage of dedicated multipliers in combination with the modest slice utilization.

	Number of Resources	**% Utilization**
Observed		
MULT18x18s	417	93.9
RAMB16s	22	5.0
Slices	29023	65.8

Figure 32: Measured and predicted resource usage for multi-scale application.

The speed-up of the RCC implementation compared to the software implementation is summarized in Figure 33. Since communication and data preparation account for over 80% of the execution time, the slower 66MHz clock rate had little affect on performance.

	Xeon 3.18GHz (s)	RCC (s) 66MHz	Speed-up (x)
Hardware Pipeline	3.860	0.006	643
Communication	-	0.010	
Data Preparation	-	0.015	
Sub-Total	3.860	0.031	125
Software layers	0.313	0.313	
Complete Application	4.173	0.344	12

Figure 33: Processing times for Application II with and without RCC.

HARPO's performance on both applications is encouraging. We successfully generated hardware sub-networks which utilized a significant portion of the FPGA and achieved significant end-to-end acceleration of the application. The performance gains are due to the extensive use of pipelining which provides significant parallelism with minimal communication overhead. For example, even though the pipeline efficiency of Application II was smaller than Application I due to increased latency and lower clock speeds the application level speed-up was greater.

7 Conclusion

Multi-scale network topologies vary greatly from one application to the next and it can be difficult to exploit parallelism with general purpose accelerators. HARPO enables a seamless transition between streams of image data at different scales through a number of parameterized hardware components. Data can be channeled through any number of processing elements, split, joined, or sampled in virtually any number of ways. This provides a rich design space to trade data throughput, latency and area. RCC is an ideal match for this framework since it can tailor the data-paths for each application to ensure redundancy is minimized and computational units are 100% utilized.

The HARPO system provides tools to semi-automate the process of mapping networks to reconfigurable hardware. The specification file, and particularly our approach to the time domain, helps make this possible by forcing the user to model the data flow used in the hardware implementation. For example, memory bandwidth is explicitly allocated in the specification file via the `Buffer` processing layer. In future work we hope to investigate how design space tradeoffs can be included in the automated generation of hardware pipelines and hence make the most effective use of both RCC and software resources.

8. References

1. Wolfram, S., *Theory and Applications of Cellular Automata*, ed. S. Wolfram. 1986, Singapore: World Scientific.
2. Preston, K. and M.J.B. Duff, *Modern Cellular Automata*. 1984, New York: Plenum Publishing Corporation.
3. Toffoli, T. and N. Margolus, *Cellular Automata Machines : A new environment for modeling*. 1987, Cambridge, Massachusetts: MIT Press.
4. Hoffmann, R., K. Volhmann, and M. Sobolewski, *The Cellular Processing Machine CEPRA-8L*. Mathematical Research, 1994. **81**: p. 179-188.
5. Marchal, P. and E. Sanchez. *CAFCA (Compact Accelerator For Cellular Automata): the metamorphosable machine*. in *IEEE Workshop on FPGA's for Custom Computing Machines*. 1994. Napa Valley, CA, USA. p. 66-71.
6. Halbach, M. and R. Hoffmann. *Implementing cellular automata in FPGA logic*. in *18th International Parallel and Distributed Processing Symposium, IPDPS 2004.* . 2004. Santa Fe, NM, United States. p. 3531-3535.
7. Kobori, T., T. Maruyama, and T. Hoshino. *A Cellular Automata System with FPGA*. in *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*. 2001. Rohnert Park, California. p. 120-129.

8. Corsonello, P., et al. *Efficient implementation of cellular algorithms on reconfigurable hardware*. in *10th Euromicro Workshop on Parallel, Distributed and Network-based Processing*. 2002. Canary Islands, Spain. p. 211-18.
9. Rosenfeld, A., *Parallel Image Processing Using Cellular Arrays.* Computer, 1983. **16**: p. 14-20.
10. Ratha, N.K., A.K. Jain, and D.T. Rover. *Convolution on splash 2*. in *Proceedings of the 1995 IEEE Symposium on FPGAs for Custom Computing Machines*. 1995. p. 204-213.
11. Chua, L.O., *CNN: A Paradigm for Complexity*. 1998: World Scientific Publishing Company.
12. Rodriguez-Vazquez, A., et al., *ACE16k: The third generation of mixed-signal SIMD-CNN ACE chips toward VSoCs.* IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, 2004. **51**(5): p. 851-63.
13. Perko, M., et al. *Low-cost, high-performance CNN simulator implemented in FPGA*. in *Proceedings of the 6th IEEE International Workshop on Cellular Neural Networks*. 2000. p. 277-282.
14. Bosi, B., G. Bois, and Y. Savaria, *Reconfigurable Pipelined 2-D Convolvers for Fast Digital Signal Processing.* IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 1999. **7**(3): p. 299-308.
15. Bouridane, A., et al., *High level FPGA-based abstract machine for image processing.* Journal of Systems Architecture, 1999. **45**(10): p. 809-24.
16. Liang, X. and J.S. Jean, *Mapping of Generalized Template Matching Onto Reconfigurable Computers.* IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2003. **11**(3): p. 485-498.
17. Torres-Huitzil, C. and M. Arias-Estrada, *Real-time image processing with a compact FPGA-based systolic architecture.* Real-Time Imaging, 2004. **10**: p. 177-187.
18. Nagy, Z. and P. Szolgay, *Configurable Multilayer CNN-UM Emulator on FPGA.* IEEE Transactions on Circuits and Systems –I: Fundamental Theory and Applications, 2005. **40**(6): p. 774-778.
19. Porter, R., et al., *Optimizing Digital Hardware Perceptrons for Multi-Spectral Image Classification.* Journal of Mathematical Imaging and Vision, 2003. **19**: p. 133-150.
20. Fukushima, K., *Neocognitron: A Hierarchical Neural Network Capable of Visual Pattern Recognition.* Neural Networks, 1988. **1**: p. 119-130.
21. LeCun, Y., et al., *Gradient-Based Learning Applied to Document Recognition.* Intelligent Signal Processing, 2001: p. 306-351.
22. Serre, T., L. Wolf, and T. Poggio, *A New Biologically Motivated Framework for Robust Object Recognition*. 2004, Massachusetts Institute of Technology, Cambridge, MA, November, 2004.
23. Vanderwal, G.S. and P.J. Burt, *A VLSI Pyramid Chip for Multiresolution Image Analysis.* International Journal of Computer Vision, 1992. **8**(3): p. 177-189.
24. Nuno-Maganda, M.A., M.O. Arias-Estrada, and C. Feregrino-Uribe. *Three video applications using an FPGA based pyramid implementation: Tracking, Mosaics and Stabilization*. in *2003 IEEE International Conference on Field-Programmable Technology (FPT)*. 2003. Tokyo, Japan. p. 336-339.
25. Behnke, S., *Hierarchical Neural Networks for Image Interpretation*. LNCS, ed. Springer. Vol. 2766. 2002. 224.
26. Woolfries, N., et al. *Non Linear Image Processing on Field Programmable Gate Arrays*. in *NOBLESSE Workshop on Non-linear Model Based Image Analysis*. 1998. Glasgow. p. 301-307.
27. Annapolis, M. http://www.annapmicro.com/. 2001.