Using Design Patterns to Overcome Image Processing Constraints on FPGAs

K. T. Gribbon, D. G. Bailey, C. T. Johnston
Institute of Information Sciences and Technology
Massey University, Private Bag 11 222,
Palmerston North, New Zealand
k.gribbon@massey.ac.nz, d.g.bailey@massey.ac.nz, c.t.johnston@massey.ac.nz

Abstract

The mapping of image processing algorithms to hardware is complicated by several hardware constraints including limited processing time, limited access to data and limited resources of the system. These constraints often force the designer to reformulate the algorithm. To aid in the process this paper details the application of design patterns to image processing algorithm development. Design patterns embody experience and through reuse provide tools for solving particular mapping problems. The effectiveness of design patterns for overcoming constraints in the mapping process is illustrated in the context of a real world example that focuses on the development of a real-time object tracking algorithm implemented on an FPGA.

1. Introduction

Continual advances in the size and functionality of FPGAs over recent years has resulted in an increasing interest in their use as implementation platforms for image processing applications, particularly real-time video processing [1].

The physical structure of FPGAs allows them to exploit the parallelism inherent in low-level image processing operations. This parallelism exists in two major forms [2]: spatial parallelism, in which the image is divided into multiple sections and processed concurrently, and temporal parallelism, where the algorithm may be represented as a time sequence of simple concurrent operations. FPGA implementations have the potential to be parallel using a mixture of these two forms.

Pragmatically, the degree of parallelization is subject to the processing mode and hardware constraints imposed by the system. Based on previous work [3,4] we believe there are three processing modes: stream, offline and hybrid processing. We have also identified the following constraints: timing (limited processing time), bandwidth (limited access to data), and resource (limited system resources) constraints. These constraints are inextricably linked and manifest themselves in different ways depending on the processing mode. Managing constraints makes the mapping of image processing algorithms to hardware more challenging.

1.1. The mapping process

In this paper, mapping is defined as the process of taking an image processing algorithm (usually represented as a serial algorithm as implemented on a serial computer) and specifying it in some hardware language which can then be subsequently compiled into a netlist and implemented on an FPGA.

Traditionally, this has been an involving low-level process as the designer must manually manage the constraints. The design is often specified at the register transfer level (RTL) using a data flow approach to help maintain better control of the constraints and allow for flexible optimization [4]. Working at this level draws heavily on past design experience to address problems resulting from the imposed constraints and often requires detailed knowledge of the underlying hardware. Reusability is performed in a somewhat ad hoc fashion, consisting of libraries of functions that remain specific to the target device and system.

In recent years, high-level languages for hardware have emerged that attempt to address some of the limitations of the low-level approach by automating parts of the mapping process. Many of these are based on popular procedural languages like C [5,6]. Compilers automatically extract parallelism from the source code using optimization techniques such as loop unrolling to exploit spatial parallelism and automatic pipelining to exploit temporal parallelism. Any speedup over an equivalent implementation on a serial processor is deemed useful.

This provides a more algorithmic approach to hardware design and appears to present a solution for image processing, which already has a large stable code base of well-defined software algorithms for implementing many common image processing operations [7]. Thus a working hardware design can be produced by porting existing software image processing libraries. This also makes it easy for image processing experts who are used to programming in a software language to make the transition from algorithmic source code to a gate-level representation (netlist) by abstracting away from the underlying low-level hardware issues [8].

The problem with this approach is that high-level languages for hardware give the illusion of software programming, which can reinforce the software 'mindset'. Offen [9] stated that the classical serial architecture is so central to modern computing that the architecture-algorithm duality is firmly skewed towards this type of architecture. If direct mapping of a software algorithm to hardware is performed, compiler optimizations will only improve the speed of what is fundamentally a sequential-based algorithm. The resulting implementation is functionally correct and in some cases real-time video processing rates are achieved or exceeded. However, it does not necessarily represent the most optimal algorithm to use for certain processing modes and there is no guarantee that compiler optimizations will meet the explicit timing constraints demanded by video rate processing.

This is most apparent when dealing with software algorithms that access memory in a way which cannot easily be achieved under stream processing. Chain coding is an example of such an operation as it requires random access to memory [10]. The algorithm must be rewritten without the requirement of random access to memory using either single or multiple passes through the image. In these cases development must revert back to low-level mapping.

1.2. Using design patterns

It is clear that there are shortcomings in the mapping process for both approaches described above. Low-level mapping is labor intensive, requires detailed knowledge, and places little emphasis on design reusability. In high-level mapping the software 'mindset' often results in sub-optimal mappings.

Reflection over previous work [3,4,10] has led to the identification of common challenges, in the mapping process under imposed constraints. To address this, we propose applying the concept of design patterns, a common design methodology in software engineering [11] (and originally borrowed from architectural engineering [12]), to the application domain of image processing on FPGAs.

The use of generalized solutions in the mapping process is not a new concept. Benkrid, Crookes et al [13] discuss hardware skeletons which are defined as a parameterized description of a task-specific architecture. However, design patterns differ in that they are more abstract. Hardware skeletons can be directly embodied in code, but only examples of patterns can be embodied in code. Other researchers in the field of reconfigurable computing advocate the use of design patterns [14] but we opt for a narrower view focused on image processing and the constraints that make the mapping process difficult.

Design patterns, as we envisage applying them, identify possible techniques for managing the constraints in different situations and focus on key elements of the solution which may be reusable in subsequent mappings.

We believe design patterns address the limitations of both low and high-level mapping that were outlined above in the following ways:

- Recorded patterns provide a way to convey design experience in a structured and informative manner by capturing the essence of the solution
- Patterns encourage design reusability as they represent generalized solutions which can be reapplied
- Considering a range of patterns for a particular problem means the choice of pattern(s) can be made according to the constraints and the system and application requirements ultimately leading to more efficient mappings
- The generalized solutions that form the basis of the methodology provide a suitable abstraction that can be applied to emerging languages and hardware in this fast moving area

Section 2 addresses issues of categorization for design patterns. A list of example patterns with associated tradeoffs is provided at the end of this paper (see Table 1). To show how design patterns can aid in the mapping process section 3 will apply design patterns in the context of a real-world example. Section 4 closes with a summary of the paper.

2. Example design patterns

In essence, design patterns are generalized solutions to recurring problems. An important aspect in our proposal of applying design patterns is taxonomy. As there are many design patterns (discovered, documented and undiscovered) and each can vary in its level of abstraction and purpose, there needs to be a way to categorize them. Categorizing should be performed according to the application domain that design patterns will be applied to. In architectural engineering Alexander, Ishikawa, et al [12] opted for a hierarchical categorization, viewing the connection between patterns as a linear and dependent sequence of categories from the general (e.g. towns) to the specific (e.g. rooms). Gamma, Helm et al [11] on the other hand have opted for classification according to purpose, which reflects what a pattern does. Thus a pattern can have creational, structural, or behavioral purpose depending on what effect the pattern has on the (programming) object.

Thus, we propose categorizing design patterns according to the constraints that they address because in our target application domain of image processing we are concerned with how to manage the constraints effectively during the mapping process.

Where possible design patterns should cover a range of possible solutions so a user can base their choice of patterns in the context of how the problem is presented and the imposed constraints. To provide some insight into the range of design patterns available a categorized list of example patterns is provided in Table 1. This list is by no means exhaustive. A brief description and the trade offs of applying each pattern are provided.

3. Real-world example

One of the best ways to illustrate the effectiveness of design patterns is to show how they solve particular problems imposed by the constraints in the context of a real-world example. Recently completed work on an FPGA-based remote object tracking algorithm [15] will serve as a means for conveying these benefits. This example will ignore some of the details of the image processing algorithm and the reader is referred to [15] if these details are of interest. Instead, this section will focus on the algorithm development and show how various design patterns were employed in the mapping process. Alternative design patterns will also be considered.

3.1. System overview

The tracking algorithm is broken into four stages: color conversion, segmentation, filtering, and object location (see Figure 1).

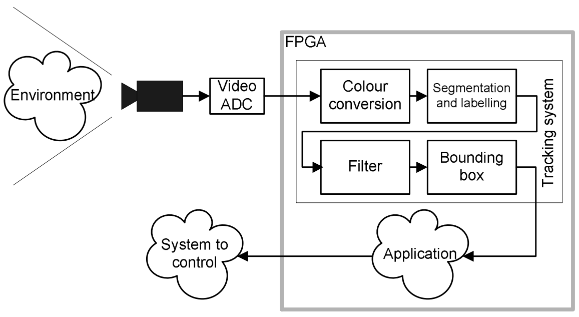

Figure 1. Block diagram of tracking system

The pixel stream from the image capture sub-system is converted to a modified YUV color space to remove the inherent interdependence between luminance and chrominance in RGB color space. Segmentation is performed using color thresholding to associate each pixel with a particular color class. A morphological filter is then used to remove any noise pixels that are not part of object regions. Finally, objects are detected by constructing a bounding box which encloses all pixels within an object region so that position, size, orientation, and other useful information may be determined for each object.

3.2. Image capture

The digitized video input stream of 16-bit RGB pixels (5:6:5) is interlaced with successive fields providing the odd and even lines of the PAL frame. A strict timing constraint is imposed with pixels arriving at a rate of 13.5 MHz, leaving approximately 74 ns per pixel to perform all four operations in Figure 1. Simply performing the operations on each pixel in sequence leads to long combinatorial delays which can easily exceed the input rate. One option is to imitate serial systems and clock the design at a much higher frequency so that multiple sequential operations can be performed in the time between input pixels. This would require running at a clock speed several times the input rate, increasing power consumption.

Our system provides a 27 MHz clock to run the design, effectively providing one input pixel every two clock cycles. This inevitably requires applying coarse-grain (between operations in the processing chain) and fine-grain pipelining (within operations) patterns to ensure timing constraints are met.

The FINE-GRAIN PIPELINING pattern accepts an input pixel value from the stream and outputs a processed pixel value every clock cycle with several clock cycles of latency, equal to the number of pipeline stages, between the input and output. This allows several pipeline stages each for the evaluation of complex expressions and functions contained in the processing chain (see Figure 1).

Here, latency must be traded off against potential speedup and this is an important consideration as delayed tracking information could cause instability in closed-loop control systems. In our implementation we minimize latency by pipelining as little as possible to meet timing constraints but in other applications long pipelines may be acceptable.

High temporal resolution is an important property of our tracking algorithm because we desire to track fast moving objects and provide real-time control. Thus we can apply the INPUT PROCESSING pattern to the interlaced input stream because each field effectively provides an independent time sample of the environment. The tracking algorithm therefore operates independently on each field of the interlaced frame. The effective image size is subsequently reduced to 768 by 288 pixels lowering spatial resolution in the vertical direction but increasing temporal resolution to a frequency of 50 samples a second. The decrease in spatial resolution is justified because tracked objects are large in relation to the size of the frame. It also avoids "tearing" resulting from the rapid movement of objects between successive fields.

The DOWNSAMPLING pattern could also be applied if extra clock cycles are needed to process pixels. For example, processing could be performed on every second pixel (downsampling by a factor of two), effectively providing an extra clock cycle in which to perform processing at the expense of decreasing spatial resolution in the horizontal direction .

Applying the INPUT PROCESSING pattern also avoids potential bandwidth constraints resulting from frame buffering, as intermediate storage for deinterlacing the input stream is not required and the process of extracting tracking information reduces data volume.

Input processing may be ineffective for some operations such as geometric transforms if the order that the pixels are required for processing does not correspond directly to the raster order in which they are input. In these cases the image must be partly or wholly buffered. Consequently developers are forced to deal with resource and bandwidth constraints.

If geometric transforms are to be performed, processing is shifted to the output end and a suitable frame buffering pattern such as MULTIPLE RAM BANKS or MULTI-PORT RAM (see Table 1) can be applied, depending on the hardware available. Full frame processing also requires frame buffering to deinterlace the input video stream. Spatial resolution is increased in exchange for a reduction in temporal resolution.

3.3. Color space transformation

RGB space is inadequate for color thresholding because of the interdependence between the luminance and chrominance components [16]. YUV space provides a more robust alternative. It was specifically designed for broadcasting and transmission and also contains perception-based properties that give more precision to brightness than colour [17].

The standard RGB to YUV transform is a coordinate rotation involving several computationally expensive floating point multiplications to map the RGB cube onto the Y, U and V axes (see [15] for details of the transformation matrix).

Direct evaluation of the transformation matrix is possible but the imposed timing constraint may cause difficulties that are avoidable if we consider that the colour space is being interpreted at an intermediate stage of the processing chain in Figure 1 and will not be viewed by the human visual system. Thus any perception-based optimisations can be ignored.

Given these considerations, the APPROXIMATION pattern can be applied. The traditional YUV transform can be modified by replacing costly multiplications with simple addition, subtraction and shift operations while still retaining the properties of YUV space that are desirable. The resulting transform is less costly to implement in terms of resources and combinatorial delays (see [15]).

3.4. Region labeling

To perform segmentation and labeling for each object, the colour space components must be independently thresholded using Equation (1):

$$Y'' > Y''_{\min}\ ,\ Y''U'_{\min} < U' < Y''U'_{\max}\ ,\ Y''V'_{\min} < V' < Y''V'_{\max} \quad (1)$$

which takes into account normalization of U and V to compensate for varying illumination (see [15]). According to Equation (1) for N colour classes, $4N$ multiplications and $5N$ comparisons must be made on each pixel after performing the colour transformation.

For stream processing, each color class must have separate hardware because all sets of comparisons and multiplications must be performed per pixel. This can be time consuming and can consume significant resources on the FPGA if performed in parallel.

As an alternative, the LOOKUP TABLE pattern can be used to perform thresholding, normalization and labeling in a single step on any number of color classes (subject to resource limitations) and has a constant processing time of one clock cycle. This saves on logic cell utilization, combinatorial logic delays, and pipeline latency.

One of the disadvantages of lookup tables is that the entries must be pre-calculated. Our implementation requires external non-volatile memory to store the contents of the lookup table because they are dynamic in the sense that the thresholds are not fixed.

3.5. Object detection

One of the simplest forms of extracting information from an object is to use a bounding box. The bounding box encloses a labeled region (all pixels within a color class) from which tracking information can be extracted. Details of the algorithm are given in [15].

All bounding boxes must be calculated in parallel during the processing of each field. A naïve implementation would use separate hardware to keep track of each bounding box. However, as each pixel is labeled as part of a single color class (overlapping regions are invalid) only a single bounding box is adjusted for each pixel. Therefore, the use of duplicated hardware is inefficient as only one instance will be used at a time.

Instead, we can apply the SHARED PROCESSOR pattern. An aggregate data structure for all bounding boxes is multiplexed to a single piece of hardware which calculates the bounding box. Using the SHARED PROCESSOR pattern, data structures can be built using logic cells (array of registers), off-chip memory, or on-chip memory. The advantage of using memory structures is that multiplexing is implicitly performed through the address port reducing logic cell utilization.

In our implementation, we configure logic cells to behave like single-port RAM. This is possible because the clock frequency of the design is twice the video input data rate. Thus a single pixel arrives every two clock cycles and the read-modify-write operations are split appropriately. Dual-port memory could also be used although on our target device there is a limited pool available.

Object tracking using a bounding box requires two processes to operate on the shared bounding box data structure. The first incrementally updates the bounds as pixels arrive. The second extracts tracking information once the bounding box has been constructed. Resource conflict can occur between these processes. One solution is to use two data structures so that the new bounding box can be constructed while tracking information can be extracted from the previous bounding box.

Alternatively the TEMPORARY OFFLINE PROCESSING pattern can be applied to overcome this resource conflict. The pattern exploits the property that for video data there are more clock cycles in a field (or frame) than there are visible pixels if the input (or output) stream contains blanking periods. Thus offline processing can be invoked whilst in the blanking periods and multiple sequential accesses can be made as long as processing is completed before the end of the blanking period.

In our implementation, three processes are used to calculate the bounding box. The first process executes during the visible portion of the video input stream to update the top, right and left bounds of the boxes as pixels arrive. The second executes during the horizontal blanking period to update the bottom bound. The third executes exclusively in the vertical blanking period where tracking information such as velocity, position, and size is extracted and the bounds reset. This removes the need to duplicate data structures.

As timing constraints are imposed the two processes that operate during the blanking periods must be scheduled to execute at specific points in time. The EVENT TRIGGERING pattern from Table 1 can be applied for this purpose. The processes remain stalled waiting for a specific event such as the beginning of the blanking period. When this occurs, execution resumes.

3.6. Filtering

After testing the initial algorithm, it was found that even with adequate tuning there were isolated noise pixels associated with each color class. Stray pixels cause the bounding box to encompass the noise pixel, leading to erroneous calculation and consequently the derivation of inaccurate tracking information. To remove these mislabelled pixels we employ a morphological erosion filter with a two-by-two structuring element window.

Filtering imposes memory bandwidth constraints as we need to populate an entire neighborhood of pixels to perform the operation.

A number of design patterns are applicable and can be divided into frame buffering or local buffering arrangements. The choice of frame buffering pattern (see Table 1) is highly dependant on the memory architecture of the system. While each frame buffering pattern aids in the alleviation of memory bandwidth constraints by allowing multiple accesses, the consequences of using any of these patterns must also be weighed. These are primarily increased system cost and space requirements.

As an alternative to frame buffering, a design pattern from the row buffering family can be applied which leads to the arrangement in Figure 2. Input data from the previous row is buffered for when the window is scanned along subsequent lines.

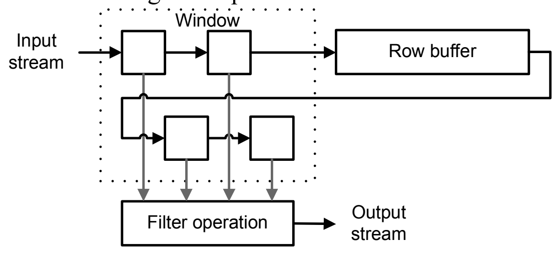

Figure 2. Architecture of row buffering design pattern family

A suitable row buffering pattern must be chosen. We apply the `PIXEL-ADDRESSED ROW BUFFER` pattern as an external counter driving other processes is available and it is more efficient to multiplex this than use a circular memory buffer. A shift register is also another alternative but does not map well to the architecture of the target device due to the large pixel width.

To minimize expected latency and combinatorial delays in filtering, our implementation also applies the `SEPARABILITY` pattern which takes advantage of the property that for some operations, independent processing can be performed in the horizontal and vertical directions. Using this pattern, the two-by-two window is decomposed into two element windows operating in both the horizontal and vertical directions. This simplifies the filter calculations.

4. Summary

FPGAs are often used as implementation platforms for image processing applications because their structure can exploit spatial and temporal parallelism.

In high-level mapping the software 'mindset' often results in sub-optimal mappings. Low-level mapping can overcome the software 'mindset' but the approach requires detailed knowledge and places little emphasis on reusability.

The example in section 3 has shown that even a simple algorithm is quickly complicated by stream processing constraints. Using design patterns facilitates the mapping process and can help overcome the imposed constraints.

5. Acknowledgements

The authors would like to acknowledge the Celoxica University Programme for generously providing the DK3 Design Suite.

6. References

[1] Hutchings, B. and Villasenor, J., "The Flexibility of Configurable Computing," *IEEE Signal Processing Magazine*, vol. 15, pp. 67-84, Sep, 1998.

[2] Downton, A. and Crookes, D., "Parallel Architectures for Image Processing," *IEE Electronics & Communication Engineering Journal*, vol. 10, pp. 139-151, Jun, 1998.

[3] Gribbon, K. T. and Bailey, D. G., "A Novel Approach to Real-time Bilinear Interpolation," *Second IEEE International Workshop on Electronic Design, Test and Applications,* Perth, Australia, pp. 126-131, Jan, 2004.

[4] Gribbon, K. T., Johnston, C. T., and Bailey, D. G., "A Real-time FPGA Implementation of a Lens Distortion Correction Algorithm with Bilinear Interpolation," *Proc. Image and Vision Computing New Zealand,* Massey University, Palmerston North, New Zealand, pp. 408-413, Nov, 2003.

[5] Najjar, W. A., Böhm, W., Draper, B. A., Hammes, J., Rinker, R., Beveridge, J. R., Chawathe, M., and Ross, C., "High-level language abstraction for reconfigurable computing," *IEEE Computer*, vol. 36, pp. 63-69, Aug, 2003.

[6] Haldar, M., Nayak, A., Choudhary, A., and Banerjee, P., "A system for synthesizing optimized FPGA hardware from MATLAB," *Proc. International Conference on Computer-aided Design,* San Jose, California, pp. 314-319, 2001.

[7] Webb, J. A., "Steps toward architecture-independent image processing," *IEEE Computer*, vol. 25, no. 2, pp. 21-31, 1992.

[8] Alston, I. and Madahar, B., "From C to netlists: hardware engineering for software engineers?" *IEE Electronics & Communication Engineering Journal*, pp. 165-173, Aug, 2002.

[9] Offen, R. J. *VLSI Image Processing*, London: Collins, 1985.

[10] Johnston, C. T., Gribbon, K. T., and Bailey, D. G., "Implementing Image Processing Algorithms on FPGAs," *Proc. Eleventh Electronics New Zealand Conference,* Palmerston North, New Zealand, pp. 118-123, Nov. 2004.

[11] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, United States of America: Addison-Wesley Publishing Company, 1995.

[12] Alexander, C., Ishikawa, S., Silverstein, M., Jacobsen, M., Fiksdahl-King, I., and Angel, S. *A Pattern Language*, New York: Oxford University Press, 1977.

[13] Benkrid, K., Crookes, D., and Benkrid, A., "Towards a general framework for FPGA based image processing using hardware skeletons," *Parallel Computing*, vol. 28, pp. 1141-1154, Aug, 2002.

[14] DeHon, A., Adams, J., DeLorimier, M., Kapre, N., Matsuda, Y., Naeimi, H., Vanier, M., and Wrighton, M., "Design patterns for reconfigurable computing," *12th Annual IEEE Symp. Field-Programmable Custom Computing Machines,* pp. 13-23, 2004.

[15] Johnston, C. J., Gribbon, K. T., and Bailey, D. G., "FPGA-based Remote Object Tracking for Real-time Control," To appear in *Proc. First International Conference on Sensing Technology*, Nov, 2005.

[16] Sen Gupta, G. and Bailey, D. G., "A new colour-space for efficient and robust segmentation," *Proc. Image and Vision Computing New Zealand,* Christchurch, New Zealand, pp. 315-320, Nov, 2004.

[17] Russ, J. C. *The Image Processing Handbook*, Boca Raton, Florida: CRC Press, 2002.

Table 1. Examples patterns and classification

Constraint	Pattern Family	Pattern Name	Description	Trade offs	
Timing[1]	Lookup table	Basic LUT	Pre-calculate values of a complex expression or function and store them in a lookup table	+ Constant access time	– Pre-calculation required
		Interpolated LUT	Interpolate between elements of a LUT	+ Increased accuracy	– Extra operations needed
	Pipelining	Fine-grain pipelining	Break up long combinatorial paths by inserting registers to store intermediate results	+ Increases design clock speed	– Increased latency – Increased resource use – Priming required
	Temporary offline processing		Utilize blanking period of the input/output video stream to perform offline processing	+ Multiple sequential operations possible	– Requires input/output streams with blanking periods
	Computational architectures	Incremental update	Calculate function using previous results and incrementally update	+ Can directly evaluate complex functions	– Sequential addressing required – Must store previous results
		CORDIC	Use CORDIC algorithm to calculate trigonometry functions	+ Does not require any multiplications	– Iterative algorithm and multiple sequential calculations
	Separability		Process vertical direction and horizontal direction of image independently	+ Simplifies calculations + May remove local buffering	– Intermediate image may need to be buffered
	Approximations		Make an approximation to an operation	+ Helps meet timing constraints	– Less accurate results possible
	Scheduling	Event-triggered	Process waits for the occurrence of some event before executing	+ Simple triggering	– Designer must keep track of applicable events
		Channel	Process waits to synchronize or to pass data over a channel	+ Designer need not keep track of synchronizations	– May be stalled for long periods of time
		Buffered channel	Processes pass data to one another through an intermediate buffer connecting them	+ Processes on either end of the buffer do not stall as often	– Buffer requires resources
		Self-stalling process	Process automatically stalls after execution and awaits triggering	+ Process can be reused	– May not have finished executing by next trigger event/sync
Bandwidth[2]	Row buffering	Shift register	Shift registers are used as buffering elements in the window filtering arrangement	+ Simple buffering arrangement	– Large shift registers may not map well to architecture – Random access not supported
		Circular memory buffer	Circular memory buffers are used as buffering elements in the window filtering arrangement	+ Reduces logic cell utilization as memory used to store pixel data	– May require dual-port RAM – Random access not supported
		Pixel-addressed buffer	Memory addressed by the *x* component of the pixel location is used as buffering elements in the window filtering arrangement	+ Random access possible	– External counters often needed for addressing
		Pre-loading buffer	Decision criteria is used to pre-load row buffer (usually in blanking period) with pixels for filtering on next line	+ May help in attaining required neighborhood pixel values in ‘hard’ cases	– More complex logic required for buffering
	Frame buffering	Multiple RAM banks	Add redundant memory in the form of multiple RAM banks in parallel.	+ Multiple access to pixels	– Increases system cost – Increases space requirements
		Bank switching	Write to one RAM bank and read from the other. Swap at end of frame	+ Provides synchronization	– Requires two RAM banks for essentially one image
		Multi-port RAM	Use RAM that supports multiple address and data ports in order to make multiple read or write accesses	+ Allows multiple access to pixels per system clock cycle	– Requires specialized RAM – Increases system cost
		Fast memory clock	Use faster RAM clock to make multiple sequential accesses per system clock cycle	+ Allows multiple access to pixels per system clock cycle	– High speed RAM required. – Increases system cost – Synchronization issues
	Input processing		Process as much of the input stream as possible	+ Can remove need to buffer image	– Not possible if input order does not match required processing order
	Packing	Pack memory	If a memory location is a multiple of pixel width, multiple pixels can be stored per location	+ Allows multiple pixels to be retrieved per access	– Suitable only for low color resolution or gray-scale pixels
		Interleaved memory	Rearrange an image as it is written to RAM such that when it is read, pixels will be in the desired order for processing e.g. window filtering operations	+ Order of pixels read from memory matches order required for processing	– Image must be rearranged to be written into memory
Resource[3]	Resource controller	Semaphore	A process raises a flag when accessing shared data. Competing processes check to see if flag is raised before accessing shared data	+ Simple to implement with insignificant additional cost to resources	– Flag can be raised at the same time it is checked leading to conflict
		Prioritized access	Shared data access is prioritized among competing processes. Higher priority processes pre-empt lower priority processes when contention occurs	+ Higher priority processes get immediate access to shared resources	– Higher-priority processes may ‘hog’ resource for long periods of time
		Time slot access	Competing processes access shared data for specific amounts of time	+ Access occurs for determinable amount of time	– Contention can still occur if time overlaps
		Event-based access	Competing processes access shared data based on the occurrence of some event	+ Simple access method	– Contention can still occur if events coincide
	Shared processor		A single instance of some function is multiplexed to multiple data	+ Decrease resource utilization as hardware is reused	– Can’t access multiple elements of data structure
	Downsampling		Decrease spatial resolution in vertical or horizontal direction or in both	+ Reduces storage requirements + Reduces ‘real estate’ if images displayed	– Processing performed on lower resolution image

[1]**Timing constraints:** The data rate requirements of the application impose a timing constraint. At video rates all required processing for each pixel must be performed at the pixel clock rate (or faster). Processes must be also scheduled for execution and asynchronous processes may need to synchronize to exchange data.

[2]**Bandwidth constraints:** Images may need to be partially or wholly buffered to perform certain operations. Off-chip memory is often used as images represent large data sets and FPGAs have small and limited amounts of on-chip memory. This places large amounts of data behind limited bandwidth and serialized connections.

[3]**Resource constraints:** The finite number of available resources in the system such as function blocks or local and off-chip RAM imposes a constraint. On an FPGA, there may be a number of concurrent processes that need access to a particular resource in a given clock cycle which can result in contention and/or undefined behavior.