Implementing image processing algorithms on FPGAs

Christopher Troy Johnston
Massey University
19 PUBLICATIONS 308 CITATIONS
SEE PROFILE

Donald Graeme Bailey
Massey University
170 PUBLICATIONS 1,140 CITATIONS
SEE PROFILE

Implementing Image Processing Algorithms on FPGAs

C. T. Johnston, K. T. Gribbon, D. G. Bailey

Institute of Information Sciences & Technology, Massey University
Private Bag 11-222, Palmerston North, NEW ZEALAND

C.T.Johnston@massey.ac.nz, K.Gribbon@massey.ac.nz, D.G.Bailey@massey.ac.nz

Abstract: FPGAs are often used as implementation platforms for real-time image processing applications because their structure is able to exploit spatial and temporal parallelism. Such parallelisation is subject to the processing mode and hardware constraints of the system. These constraints can force the designer to reformulate the algorithm. This paper presents some general techniques for dealing with the various constraints and efficient mappings for three types of image processing operations.

Keywords: FPGA, image processing, algorithms, operations

1. INTRODUCTION

Real-time image processing is difficult to achieve on a serial processor. This is due to several factors such as the large data set represented by the image, and the complex operations which may need to be performed on the image. At real-time video rates of 25 frames per second a single operation performed on every pixel of a 768 by 576 colour image (PAL frame) equates to 33 million operations per second. This does not take into account the overhead of storing and retrieving pixel values. Many image processing applications require that several operations be performed on each pixel in the image resulting in a even large number of operations per second.

One alternative is to use a field programmable gate array (FPGA). Continual growth 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].

An FPGA consists of a matrix of logic blocks that are connected by a switching network. Both the logic blocks and the switching network are reprogrammable allowing application specific hardware to be constructed, while at the same time maintaining the ability to change the functionality of the system with ease. As such, an FPGA offers a compromise between the flexibility of general purpose processors and the hardware-based speed of ASICs. Performance gains are obtained by bypassing the fetch-decode-execute overhead of general-purpose processors and by exploiting the inherent parallelism of digital hardware.

Parallelism in image processing algorithms exists in two major forms [2]: spatial parallelism and temporal parallelism. FPGA implementations have the potential to be parallel using a mixture of these two forms. For example, in order to exploit both forms the FPGA could be configured to partition the image and distribute the resulting sections to multiple pipelines all of which could process data concurrently. In practice, such parallelisation is subject to the processing mode and hardware constraints of the system. This in turn forces the designer to deal with hardware issues such as concurrency, pipelining and priming, which many image processing experts are unfamiliar with.

To overcome this, research has focused on the development of high-level languages and their associated compilers such as in [3-5]. A common goal of these languages is to hide many of the low-level details from the developer by allowing the compiler to automatically extract parallelism using optimisation techniques such as loop unrolling to exploit spatial parallelism and automatic pipelining to exploit temporal parallelism. Thus a working hardware design may be as simple as making a few syntactic modifications to an existing program, compiling it and downloading the resulting configuration file to the FPGA. This appears to be a perfect solution for image processing, which already has a large stable code base of well-defined software algorithms for implementing many common image processing operations [6]. 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) without any knowledge of the underlying hardware [7].

However, such an approach restricts the developer to the software programming 'mindset'. Offen [8] has 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 optimisations will only improve the speed of what is fundamentally a sequential-based algorithm. This may not be the best algorithm to use for certain processing models on an FPGA, which could benefit from a completely different and more efficient mapping of the conceptual algorithm to hardware.

The mapping of operations to hardware can fall into several categories. In some cases the mapping is simple, and there is little difference from a functionally equivalent software implementation. In others a data structure and caching arrangement needs to be implemented but apart from this the algorithm itself may only have minor changes from a software implementation. However, in some cases the standard algorithm developed for software implementation does not map efficiently to hardware because it uses too many resources or accesses memory in a way which cannot be supported. In these 'hard' cases the algorithm needs to be rewritten to meet the constraints imposed by the hardware.

Manual mapping means the designer must be prepared to deal with the hardware constraints and the implications that follow. For example, the constraint of real-time processing introduces a number of additional complications. These include such issues as limited memory bandwidth, resource conflicts, and the need for pipelining.

To help maintain better control of these constraints many hardware designs are still coded at the register transfer level (RTL). This has been corroborated through previous work which has shown that it is beneficial to maintain a data flow approach at the register transfer level [9,10]. The 'better control' is a function of the low-level programming model used, which in the software domain is analogous to programming in assembly language. Like assembly language, however, design at the RTL level becomes difficult and cumbersome for large and complex algorithms.

The ultimate goal then is a development environment that automatically manages the hardware issues without the restrictions of a 'software mindset'. A step towards this requires understanding what the constraints are, under what circumstances they are imposed, and their effect. Based on previous work [9,10] we have identified the following constraints: timing, bandwidth, and resource constraints. These constraints are inextricably linked and driven by the data rate requirements of the application as section two will show. This section discusses the constraints and their effect under different processing modes. Section three discusses some general techniques for mapping algorithms to hardware that should always be considered. In section four, three different classes of operations and their implementations are discussed with respect to the issues illustrated in the paper.

2. CONSTRAINTS

The constraints outlined above manifest themselves in different ways depending on the processing model. We believe there are three modes: stream, offline and hybrid processing.

In stream processing, data is received from the input device in a raster nature at video rates. Memory bandwidth constraints dictate that as much processing as possible be performed as the data arrives.

In offline processing there is no timing constraint. This allows random access to memory containing the image data. This mode is the easiest to program, as a direct mapping from a software algorithm can be used. The speed of execution in most cases is limited in by the memory access speed.

The hybrid case is a mixture of stream and offline processing. In this case, the timing constraint is relaxed so the image is captured at a slower rate. While the image is streamed into a frame buffer it can be processed, such as extracting a region of interest. This region of interest can then be processed by an offline stage which would allow random access to the region's elements.

2.1. Timing Constraints

If there is no requirement on processing time then the constraint on timing is relaxed and the system can revert to offline processing. This is often the result of a direct mapping from a software algorithm. The constraint on bandwidth is also eliminated because random access to memory is possible and desired values in memory can be obtained over a number of clock cycles with buffering between cycles. Offline processing in hardware therefore closely resembles the software programming paradigm; the developer need not worry about constraints to any great extent.

This is the approach taken by languages that map software algorithms to hardware. The goal then, is to produce hardware that processes the input data as fast as possible given various automatic and manual optimisation techniques. The view here is that any speedup over an equivalent implementation on a serial processor is useful. This is the approach offered by [3].

In contrast to this, when an image processing application demands real-time processing at video rates, the timing constraints become crucial. For example, video display generation has deadlines on the order of one pixel every 40 ns (VGA output). Stream processing constrains the design into performing all of the required calculations for each pixel at the pixel clock rate. Producing one pixel every 40 ns for non-trivial applications, such as lens distortion correction [10] is difficult because for each pixel complex expressions must be evaluated. These can introduce significant propagation delay, which may easily exceed a single pixel clock cycle. A pipelined approach is thus needed that 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.

Pipelining is a relatively easy optimisation to perform, since it does not require that the algorithm be modified. Given enough resources, any desired throughput can be achieved by pipelining, at the expense of added latency. A number of higher-level languages already offer automatic pipelining capabilities [3,5]. In some cases real-time video processing rates are achieved or exceeded using automatic mapping to hardware but there is no guarantee that compiler optimisations will meet the explicit timing constraints demanded by video rate processing. This is not the focus of such an approach.

Under stream processing, some operations require that the image be partly or wholly buffered because the order that the pixels are required for processing does not correspond directly to the raster order in which they are input. This is true for spatial operations and local filter operations. Consequently developers are forced to deal with resource and bandwidth constraints.

2.2. Bandwidth Constraints

Frame buffering requires large amounts of memory. The size of the frame buffer depends on the transform itself. In the worst case (rotation by 90º, for example) the whole image must be buffered. A single 24-bit (8-bits per colour channel) colour image with 768 by 576 pixels requires 1.2 MB of memory. FPGAs have very limited amounts of on-chip RAM (Xilinx calls this BlockRAM). The logic blocks themselves can be configured to act like RAM (termed distributed RAM) but this is usually an inefficient use of the logic blocks.

Typically some sort of off-chip memory is used but this only allows a single access to the frame buffer per clock cycle, which can be a problem for the many operations that require simultaneous access to more than one pixel from the input image. For example, bilinear interpolation (see [9] for more details) requires simultaneous access to four pixels from the input image. This will be on a per clock cycle basis if real-time processing constraints are imposed.

Possible alternatives to deal with this problem are the use multiport RAM, multiple RAM banks in parallel or using a faster RAM clock to read multiple locations in a single pixel clock cycle. These alternatives all have significant disadvantages. Multiport RAM is specialized and expensive. The use of multiple banks is clumsy because the added redundancy is expensive in both cost and space. Finally, using a faster RAM clock requires expensive high speed memory and introduces synchronization issues.

One solution that avoids all these complications is to cache data read from the frame buffer that is likely to be used in subsequent calculations. Creating this data structure introduces additional complications; the need for mechanisms that stall, add to and remove data from the buffer. Managing bandwidth constraints often changes the way one must conceptually view and approach the algorithm as section 4.2 will illustrate.

2.3. Resource Constraints

Resource contention arises due to the finite number of available resources in the system such as local and off-chip RAM or other function blocks implemented on the FPGA. If there are a number of concurrent processes that need access to a particular resource in a given clock cycle then some sort of scheduling must be performed. The worst case involves redesigning the underlying algorithm. Care must also be taken to ensure that concurrent processes avoid writing to the same register during a given clock cycle.

Pipelining results in an increase in logic block usage. This is caused by the need to construct pipeline stages and registers rather than being able to reuse the small number of sequential computing elements (ALU and registers), as can be done with offline processing. Flip flops introduced by pipelining typically incur a minimum of additional area on an FPGA, as they are mapped onto unused flip flops within logic blocks that are already used for implementing other combinatorial logic in the design [11].

While the impact of pipeline registers on logic block usage will be minimal, care must still be taken to make efficient use of the available logic blocks. Implementations of more complex arithmetic operations such as squares and square roots consume large resources and also increase combinatorial delays if not pipelined. These are especially important consideration for designs using small low cost FPGAs.

3. GENERAL TECHNIQUES

There are a number of techniques for dealing with expressions that inefficiently map to hardware.

Simplistically, where it is possible, division and multiplication by powers of two should be used. This is equivalent to a shift left or right. In hardware, multiplication or division by a constant power of two may be achieved by simply rewiring the output of a logic block. For more complex expressions, such as square roots or multiplying and dividing by an arbitrary number, look-up tables (LUT) and raster-based incremental methods can be used.

3.1. Look-up Tables

In its simplest form the LUT method involves pre-calculating the result of an expression or function for various input combinations that represent a range of values. Upon execution the resulting values are loaded into either local memory on the FPGA (in BlockRAM) or in off-chip memory. This step is usually performed offline but another possibility is when spare clock cycles are available, such as in the vertical blanking period of the display. This allows the LUT to be dynamically updated during execution.

During execution the LUT, evaluates the expression, by retrieving the table entry that corresponds to the current input combination (memory address). The access time is constant for all input combinations. There are a number of considerations that must be made before using a LUT. The amount of resources (typically limited BlockRAM) and delays in routing and LUT access time must be balanced against those of building hardware for run-time evaluation. LUTs also have limited access per clock cycle, typically one or two read accesses. Consequently, if many processes wish to access the data simultaneously, more than one LUT is needed.

The accuracy of a LUT is related to its size. For example, if there are 256 elements in the LUT, the expression or function can only be evaluated to an input of 8 bits. To improve the accuracy, linear interpolation between adjacent table entries can be used. This has an added cost in terms of logic block usage and in the number of clock cycles needed for evaluation.

In [10] a 256 element LUT was used with linear interpolation. In this paper, both the required table entry and the next entry are retrieved using the most significant 8 bits of 16-bit operands. The least significant 8 bits of the operand are used to interpolate between the two table entries. This improved the resolution to approximately 15 bits.

3.2. Raster-based Methods

This technique, also used in [10], reduces the amount of hardware required by exploiting how data is presented in the stream processing mode. When an image is read in a raster fashion from top left to bottom right the column (x) and line (y) changes can be used to incrementally update the running total of an expression. For example, to calculate x^2 or y^2 without multiplication equation (1) can be used

$$(z+1)^2 = z^2 + 2z + 1 \tag{1}$$

This uses few resources but cannot be implemented when x^2 or y^2 is needed randomly.

4. ALGORITHM MAPPING

Algorithms for image processing are normally classified into one of three levels: low, intermediate or high. Low-level algorithms operate on individual pixels or neighbourhoods. Intermediate-level algorithms either convert pixel data into a different representation, such as a histogram, coordinate or chain code, or operate on one of these higher representations. High-level algorithms aim to extract meaning from the image using information from the other levels. This could be in the form of rejecting a component or identifying where an object is within an image.

When moving from low to the high-level representations there is a corresponding decrease in exploitable parallelism due to the change from pixel data to more descriptive representations. However there is also a reduction in the amount of data that must be processed, allowing more time to do the processing.

Due to their structure, FPGAs are most appropriate for use with computationally intensive tasks which form the vast majority of low and intermediate-level operations. The large data sets and regular repetitive nature of the operations can be exploited. For this reason it has been traditional in many systems for the FPGA to handle the low-level operations and then pass the processed data to a microprocessor which then executes the high-level operations. With increasing FPGA size, it is now possible to implement processor cores on the reconfigurable fabric, which means the FPGA can form the core of the system.

4.1. Point operations

Point operations are a class of transformation operations where each output pixel's value depends only upon the value of the corresponding input pixel [12]. The mapping of point operations to hardware can be achieved by simply passing the image though a hardware function block, that is designed to perform the required point operation. For more complex functions LUTs can be used.

Stream processing essentially converts spatial parallelism into temporal parallelism. Raster-based presentation of the data stream results in the need to perform only one memory access per clock cycle.

4.2. *Window operations*

A more complex class of low-level operations are local filters. Conceptually, each pixel in the output image is produced by sliding an $N \times M$ window over the input image and computing an operation according to the input pixels under the window and the chosen window operator. The result is a pixel value that is assigned to the centre of the window in the output image, as shown below in Figure 1.

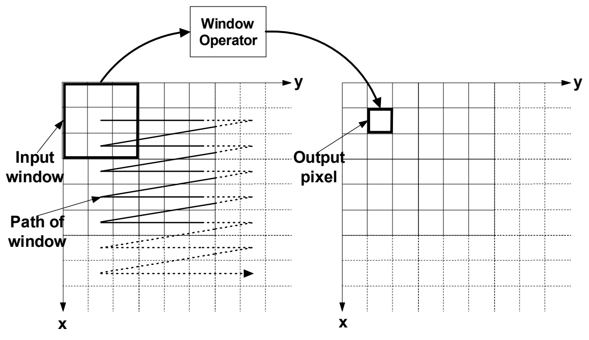

Figure 1. Conceptual example of window filtering

For processing purposes, the straightforward approach is to store the entire input image into a frame buffer, accessing the neighbourhood pixels and applying the function as needed to produce the output image. If real-time processing of the video stream is required $N \times M$ pixel values are needed to perform the calculations each time the window is moved and each pixel in the image is read up to $N \times M$ times. Memory bandwidth constraints make obtaining all these pixels each clock cycle impossible unless some form of local caching is performed. Input data from the previous $N-1$ rows can be cached using a shift register (or circular memory buffer) for when the window is scanned along subsequent lines. This leads to the block diagram shown below in Figure 2.

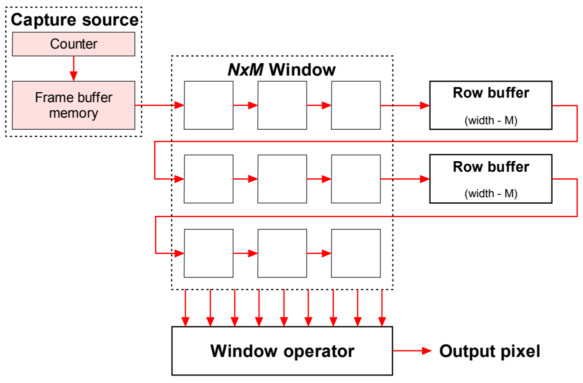

Figure 2. Block diagram for hardware implementation of window filtering

Instead of sliding the window across the image, the above implementation now feeds the image through the window.

Introducing the row buffer data structures adds additional complications. With the use of both caching and pipelining there needs to be a mechanism for adding to the row buffer and for priming, stalling and flushing the pipeline. This is required when operating on video data, due to the horizontal blanking between lines and the vertical blanking between frames. If either the buffer or the pipeline operated during the blanking periods the results for following pixels would be incorrect due to invalid data being written to them. This requires the designer to stop entering data into the row buffers and to stall the pipeline while a blanking period occurs.

Another complication occurs when the window extends outside the image boundary. There are several options for dealing with this; the simplest is to assume one row wraps into the next. A better option is to replicate the edge pixels of the closest border. Such image padding can be considered as a special case of pipeline priming. When a new frame is received the first line is pre-loaded into the row buffer the required number of times for the given window size. Before processing a new row the first pixels are also pre-loaded the required number of times, as is the last pixel of the line and the last line.

4.3. *Global operations*

Intermediate level operations are often more difficult to implement on FPGAs as they convert pixel data to higher-level representations such as chain codes or regions of interest. These algorithms often require random access to memory that cannot easily be achieved in stream processing mode. The algorithm must be rewritten without the requirement of random access to memory using either single or multiple passes through the image. Chain coding is an example of an algorithm for which this must be performed.

A chain code defines the boundary of an object. The ‘standard’ software approach is to scan the image until an unprocessed edge is encountered. From this point, the direction to the next edge pixel is added to the chain code and tracking continues until the whole contour is defined. Tracking the edge therefore requires random access to memory.

The algorithm can be reformulated for a direct raster implementation by keeping track of multiple edges while raster scanning [13]. This is a more efficient mapping to hardware and needs only a single pass through the image.

5. SUMMARY

FPGAs are often used as implementation platforms for real-time image processing applications because their structure can exploit spatial and temporal parallelism. Such parallelisation is subject to the processing mode and hardware constraints of the system.

Using high-level languages and compilers to hide the constraints and automatically extract parallelism from the code does not always produce an efficient mapping to hardware. The code is usually adapted from a software implementation and thus has the disadvantage that the resulting implementation is based fundamentally on a serial algorithm.

Manual mapping removes the 'software mindset' restriction but instead the designer must now deal more closely with timing, resource and bandwidth constraints, which complicate the mapping process.

Timing or processing constraints can be met using pipelining. This only adds to latency rather than changing the algorithm, which is why automated pipelining is possible. Meeting bandwidth constraints on the other hand is more difficult because the underlying algorithm may need to be completely redesigned, an impossible task for a compiler.

This paper presented some general techniques for evaluating complex expressions to help deal with resource constraints by reducing logic block usage. The mapping of three different types of operations was discussed in relation to the hardware constraints.

Implementing any operation under stream processing mode requires that for every clock cycle (1) the required pixel data can be accessed by the processing block and (2) the pixel data is presented correctly to the processing block.

For point operations, both requirements are easily met and the resulting mapping differs little from a functionally equivalent software implementation. Window operations require local caching and control mechanisms but the underlying algorithm remains the same. Global operations such as chain coding require random access to memory and cannot be easily implemented under stream processing modes. This forces the designer to reformulate the algorithm.

6. ACKNOWLEDGEMENTS

The authors would like to acknowledge the Celoxica University Programme for generously providing the DK2 Design Suite.

7. 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 *Electronics & Communication Engineering Journal*, vol. 10, pp. 139-151, Jun, 1998.

[3] 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.

[4] Haldar, M., Nayak, A., Choudhary, A., and Banerjee, P., "A system for synthesizing optimized FPGA hardware from MATLAB ," *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design,* San Jose, California, pp. 314-319, 2001.

[5] Crookes, D., Benkrid, K., Bouridane, A., Alotaibi, K., and Benkrid, A., Design and Implementation of a High Level Programming Environment for Fpga-Based Image Processing *IEE Proceedings-Vision Image and Signal Processing*, vol. 147, pp. 377-384, Aug, 2000.

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

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

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

[9] 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, Jan. 2004.

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

[11] Xilinx, Inc. Xilinx 4 Software Manuals: Libraries Guide. 2002.

[12] Castleman, K. R. *Digital Image Processing*, Upper Saddle River, New Jersey: Prentice-Hall, 1996.

[13] Zingaretti, P., Gasparroni, M., and Vecci, L., Fast chain coding of region boundaries *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 20, no. 4, pp. 407-415, 1998.