K. T. Gribbon, D. G. Bailey, A. Bainbridge-Smith, 'Development Issues in Using FPGAs for Image Processing', *Proceedings of Image and Vision Computing New Zealand 2007*, pp. 217–222, Hamilton, New Zealand, December 2007.

Development Issues in Using FPGAs for Image Processing

K.T. Gribbon[1], D.G. Bailey[1], and A. Bainbridge-Smith[2]

[1] Institute of Information Sciences and Technology, Massey University
[2] Department of Electrical and Computer Engineering, University of Canterbury

Email: k.gribbon@massey.ac.nz

Abstract

Field programmable gate arrays (FPGAs) offer many performance benefits for executing image processing applications. Developing the algorithms to solve image processing problems is one aspect of using FPGAs; these algorithms must then be mapped to the FPGA. Mapping an algorithm requires building and utilising FPGA-specific hardware (such as fast multipliers and block RAM) on an open architecture platform. This is fundamentally different to the design of software for the fixed architectures of conventional processors. Although software techniques may help define the image processing algorithm and facilitate its programming, they will provide little guidance on how to manage hardware specific issues such as concurrency and pipelining. As part of our aim to define new techniques that address these issues in an image processing context we present the complete design cycle and use it to elucidate some of the important considerations in the progression from problem specification to an FPGA-based implementation.

Keywords: FPGA, design cycle, algorithm development, architecture

1 Introduction

Semiconductor manufacturers are no longer relying upon increasing clock speeds to increase processor performance. Heat dissipation and power consumption become increasingly problematic at high clock speeds (above 3 GHz) calling for other techniques to be considered [1]. One option is to leverage concurrency by building chips incorporating multiple processors. This solution is being widely adopted in the market with multi-core processors now standard on new computers and gaming consoles.

Although consumers are just beginning to experience the benefits of concurrency, parallel architectures have played an important role in image processing (IP) since the 1960s [2]. Systems such as the INMOS transputer and data cube have been used to solve complex vision problems where conventional single-processor architectures have not achieved the desired speed of processing.

Today, FPGAs (field programmable gate arrays) are emerging as a useful parallel platform for executing demanding IP algorithms. FPGAs consist of a large array of parallel logic and a large number of I/O pins for data access, as shown in Figure 1. This general and unspecified structure can be easily exploited to implement both spatial and temporal parallelism. These are common in IP applications and thus significant computation advantages can be achieved.

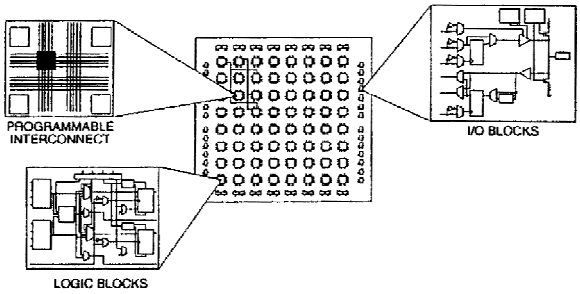

Figure 1: An FPGA consists of logic "islands" in a "sea" of routing, which may be exploited for massive parallelism. Figure taken from Trimberger [3].

Using FPGAs to accelerate IP algorithms presents several challenges. One simply relates to Amdahl's law: a large proportion of the algorithm must lend itself to parallelisation to achieve substantial speedup [4]. Based on this Herbodt et al conclude that FPGA performance is "unusually sensitive to the implementation's quality" (p. 50, [4]). Therefore it is important to develop an appropriate algorithm to exploit available parallelism.

Unfortunately, this is not as simple as parallelizing the code of a software algorithm as one would when porting single-processor applications to multi-core or traditional parallel computing systems. One reason is that current hardware languages and compilers are still maturing [5]. The problem is made more difficult due to an FPGA's general structure which is not limited to two or four fixed processors such as on current dual or quad-core chips. Instead they have the potential to be massively parallel, with the number of processors limited only by the density of the logic. For a given application one could build a large

number of simple parallel processors in a configuration such as a systolic array, a full hardware solution, a soft-core processor or any combination of the above. As an FPGA's structure is not fixed, one must develop both the algorithm and the architecture on which it is implemented. This complicates design because it must performed at the high-level (algorithmic) and low-level (architecture selection, pipelining, memory management) simultaneously.

Therefore, traditional software engineering techniques cannot be relied upon exclusively. To address these issues our research group has been investigating techniques that lower developmental effort when implementing IP algorithms on FPGAs. To date our research efforts have focused on VERTIPH, a visual programming language to help designers conceptualise and implement IP algorithms on FPGAs [6], GATOS, a windowed operating system to aid in the debugging of IP algorithms [7] and an extension of design pattern methodology to capture generalized solutions to FPGA design problems [8].

As part of identifying potential development issues we have found it useful to consider the development process holistically. In this paper we present the complete design cycle which we use to explore several important development issues. Section 2 considers some of the challenges stemming from the differences between software and FPGA design. Section 3 presents the complete design cycle for an FPGA-based system. Sections 4, 5 and 6 examine some of the important issues in each stage of the cycle. Section 7 addresses design flow. Summary and conclusions follow in section 8.

2 Hardware Challenges

Image processing is often viewed as a software engineering problem [9]. Although important in the development and implementation of algorithms, software design is often given prominence over other aspects of the system. Applied image processing is really a systems engineering problem because there are other important aspects to consider such as lighting, optics and integration with supporting hardware and machinery [10]. Design with FPGAs fits well into a systems engineering context because it is performed at several different levels. These include high-level algorithmic design down to bit-level operation design.

Although the flexibility is available to work at the bit-level, designers do not want to spend all their time there. Schematic entry and HDLs are often too low-level as design tools because they do not capture the algorithmic nature of image processing functions adequately. Design at this level is complex, tedious and error prone [11]. An alternative that aids programmer productivity and more closely matches algorithmic design are high-level languages that infer circuitry using a hardware compiler. In this context FPGA configurations may resemble traditional high-level languages like C, but specify hardware not software [4]. One advantage with this approach is that traditional software techniques can be co-opted to help write code. The danger is the temptation to port software algorithms to hardware because of the similarities between the languages. These similarities are often superficial in the sense that the hardware configuration has merely undergone a representational change. This leads to the implementation being 'constrained' by the algorithm because the approach assumes that good software algorithms make good hardware algorithms. This is often untrue for the following reasons:

- Optimal processing modes differ on an FPGA [4]. Random-access and pointer-based operations are efficient in software. A typical processing scenario involves grabbing a frame and moving it to main memory. The processor can then sequentially process pixels, affording random access to the image. On an FPGA this can be highly inefficient and costly.
- Clock speeds are typically an order of magnitude slower than processors due to delay overheads through the general routing matrix. Therefore configurations must exploit parallelism rather than relying solely upon a high rate of processing.
- Sequential processing of software code avoids contention for system resources. An FPGA's potential for massive parallelism frequently complicates arbitration and creates contention for memory and shared processors.
- Lack of an operating system complicates management of 'thread' scheduling, memory, and system devices, which must be managed manually.

Based on these reasons, a more suitable algorithm may exist that can better exploit the available parallelism of the selected architecture. However, modifying an algorithm and designing the computational and memory architecture requires extra development effort on the part of the system designer. Given these challenges we now present the complete design cycle which will allow deeper exploration of some of these issues.

3 Design Cycle

We believe the complete development of an image processing application for an FPGA-based system involves four stages: *problem specification*, *algorithm development*, *architecture selection* and *implementation*. Descriptions and logical relationships between the stages are shown in Figure 2.

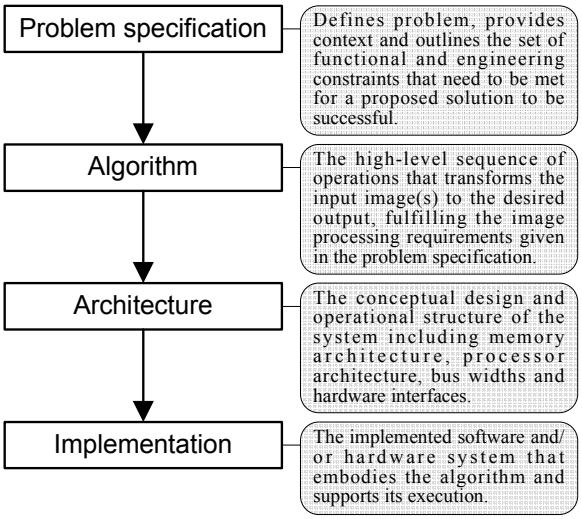

Figure 2: Stages in the design cycle.

As the problem specification stage is common to software engineering design (and indeed any engineering design) we will focus our attention upon the following three stages of Figure 2 and corresponding FPGA specific issues.

4 Algorithm level

The development of an algorithm can be regarded as a problem solving task, where the aim is to find a sequence of mathematical (image processing) operations that progressively transform the image into the desired result, satisfying the functional requirements outlined in the problem specification. Several approaches can be applied to development which have remained largely unchanged since the 1980's [12].

4.1 Selecting Algorithms

A simple mapping of a software implementation into hardware often falls short of the potential benefits offered by an FPGA solution as they do not tend to leverage concurrency. These sequential algorithms must be redesigned.

For example, the classic connected components algorithm requires two raster-scan passes through the image. On the first pass, a label is assigned to each pixel based upon the connection between pixels in a local neighbourhood. Multi-branch ("U" shaped) objects have multiple labels applied during the scan and must be relabelled for consistency in the second pass. The classic algorithm assumes that memory is available to perform image buffering of the intermediate labelled image between the first and second pass.

On an FPGA, on-chip memory resources are limited. Implementing the classic algorithm on an FPGA would require off-chip buffering which adds to the system cost. If we can develop and use a one-pass algorithm we could process images streamed directly from a camera. Given an appropriate stream processing architecture (to convert spatial distribution into a temporal stream) it may then be possible to eliminate image buffering altogether.

To achieve this it is necessary to perform the merging and relabelling operations on the fly and in parallel. As we do not wish to retain image data, the features of each connected component must also be accumulated in parallel as the image is scanned. This removes the need for producing a labelled image and saves having to perform the second, relabelling pass.

5 Architecture level

Figure 2 shares much in common with the traditional software engineering lifecycle. The problem is defined, algorithms are developed and then implemented. However, software design assumes an underlying hardware architecture. With an FPGA we should seek to tailor the architecture for the application. This is the purpose of the architecture stage in Figure 2.

In the connected component labelling application the preprocessing operations (typically local filters and point operations), necessary to segment objects from the background, are ideally suited to stream-based processing without image buffering. By utilising a stream processing architecture (which includes an appropriate memory architecture for the local filters) coupled with the one pass algorithm mentioned above, the requirement of image buffering could be removed altogether. An FPGA-based system such as an embedded smart camera could then process a progressively scanned image directly without any off-chip buffering.

Although most preprocessing operations are amenable to stream processing and can be implemented efficiently on an FPGA, some operations can not. For example, performing recognition tasks on the labelled image using a neural network may not translate well into an FPGA implementation.

In general, higher-level image processing tasks can be harder to implement in custom hardware and tend to underutilise the hardware because they may not be run as often (e.g. recognition tasks run once per frame as opposed to preprocessing operations which need to be run for each pixel in an image scan). This can leave hardware idle for long periods of time which is often an ineffective use of the limited logic resources. An alternative is to implement some of these tasks in software by synergistically combining FPGAs and serial processors in a single system. This is an example of reconfigurable computing, which is becoming an increasingly important paradigm in high performance computing [11].

5.1 System Configuration

The degree to which an FPGA is yoked to a processor is termed coupling. Two useful configurations are hosted and standalone. In a hosted configuration, the FPGA is utilised as a co-processor to a conventional processor. As a co-processor, the FPGA's role is to complement the host computer by accelerating the portion of the algorithm that takes most of the execution time.

These tasks are likely to be low-level (pixel-based) image processing operations, which an FPGA is more suited to performing. The host computer is usually responsible for "front end" tasks such as image capture, display and user interaction. To facilitate co-operation, shared memory is required to transfer the raw and processed image data between the two devices.

In a standalone configuration, the host computer is dispensed with, and all tasks are performed directly on the FPGA.

Hosted and standalone configurations represent two possible options on a wide coupling spectrum. At one end are full hardware solutions, at the other full software solutions; in between are several options at a range of granularities. For example, implementations needn't be executed on off-the-shelf processors. A simple controller like a finite state machine can be built very easily on the FPGA fabric. These may be useful for performing intermediate-level tasks with low processing requirements. If more complicated processing is required a full-blown processor may be required Building a soft-core processor provides several advantages. Instruction sets can be customised for efficiency and wiring delays are minimised. High development costs can be offset against software programmability; the same FPGA configuration can be used for several applications and a custom parallel computer can be built very quickly [13].

5.2 Memory Architecture

Memory configuration is another important architectural issue. Although there are several different memory resources in serial processor systems (processor cache, RAM etc.), when writing software the programmer typically has access to two types of memory: registers and external memory. In contrast a designer has access to a range of different memory resources at differing granularities available for FPGAs. An FPGA may have access to:

- Individual registers within logic blocks
- Logic blocks configured as look up tables
- Small blocks of dedicated contiguous memory
- Off-chip memory

These resources should be exploited appropriately. For example, registers can be used to store an array or buffer intermediate values in a pipeline. On the other hand, the small blocks of dedicated memory (often referred to as block RAM) are ideally suited to row buffering which is necessary for local filtering. Block RAM is usually sufficient to buffer several rows of an image while off-chip memory is often necessary for applications that require frame buffering, such as image warping and geometric transforms.

6 Implementation level

After thinking through the architecture we must think about how to specify the operations efficiently in the presence of timing, resource and bandwidth constraints. Testing of the implementation for debugging purposes is also necessary but is harder to accomplish on an FPGA.

Good design can be accomplished by using techniques such as pipelining (fine and coarse grain), utilising look up tables, CORDIC functions, and making function approximations. It is also useful to draw a data flow diagram early on to identify parallel branches between and within operations in the algorithm. We have been capturing and formally documenting such techniques using hardware design patterns which we believe are an appropriate means for recording and transferring the required knowledge needed to implement these techniques [8].

6.1 Pipelining

Fine-grain pipelining is an important technique to exploit the temporal parallelism inherent in stream data, which in an image processing context, is commonly generated by a video source.

Simply performing the required operations in the algorithm on each pixel in sequence often leads to long combinatorial delays which can easily exceed the input rate of the stream. A pipeline on the other hand accepts an input pixel value from the stream and outputs a processed pixel value each clock cycle with several clock cycles of latency, equal to the number of pipeline stages, between the input and output. At any instant, stages of the pipeline will contain pixels at successive stages of processing. This allows several pipeline stages each for the evaluation of complex operations.

A video stream may contain a mixture of pixel and control data in the form of tokens to indicate the start of a new frame or line. Therefore mechanisms must be available to prime, flush and stall the pipeline. This complicates pipeline management. Priming for image processing operations is often difficult and error prone because of the need to start processing before the first item of valid data is available. Flushing is also difficult as the pipeline must continue to run until all valid data has finished processing.

6.2 Debugging

Once we have developed our FPGA implementation we are likely to want to test it for debugging purposes. Conventional hardware verification paradigms are impractical for this purpose. The problem stems from the large volume of data contained within an image. With complex algorithms, it is extremely difficult to design test vectors that exercise all of the functionality of the system, especially when there may be complex interactions. In image processing, the problem is even more difficult, because the algorithm may be working perfectly as designed, but it may not be appropriate or adequate for the task to which it is applied. Validation using test vectors will only verify that the algorithm is implemented correctly, and not whether the correct algorithm is actually being used. When processing video, this is exacerbated by the dynamic nature of the data being processed.

Simulation of the configuration prior to in-circuit testing is desirable. However, on the development platform (which is most likely a general-purpose computer) simulation is of limited value in judging final performance because a parallel architecture must be simulated on a serial architecture, which is very slow. The performance of a real-time application processing a continuous, noisy, video stream is difficult to evaluate without actually having the final system.

In-circuit debugging is significantly more complex and troublesome on an FPGA particularly in standalone configuration which lacks an operating system to manage basic tasks such as user interaction and peripheral interfacing. For in-circuit testing additional debug circuitry must be built in tandem to perform these tasks even if they are not required in the final system. For example, circuitry to drive a display was needed during development of an object tracking application to perform lens calibration, tuning and to verify correctness despite the output being a set of integer coordinates that are passed on to another subsystem [14]. Building debug circuitry also reduces the amount of logic available to the application.

To address some of these issues we have proposed the Gate Array Terminal Operating System (GATOS) which provides a set of high-level IP blocks to implement the graphical user interface of an interactive windowing operating system [7].

7 Design Flow

The logical progression through the development stages is shown in Figure 2. For example, algorithm development precedes architecture selection because that way the architecture is tailored specifically to the algorithm.

In practice however, this flow may be interrupted. We have alluded to the iterative nature of the development process throughout the paper. Iteration is usually required to achieve acceptable performance and functionality. It is also required due to the interdependency between the stages. Changes made in one stage may inadvertently affect another. For example, significant modifications may be made to an existing algorithm to make it more suitable for hardware implementation, such as in the connected component labelling application. When these changes are made the algorithm should be re-tested (back in the image processing development environment).

In addition to the iterative nature of the practical flow, development stages can be entered out of order. Several practical permutations of the design process exist. For example, consider Figure 3 which reflects the practical flow that we took in the development of the object tracking algorithm mentioned above.

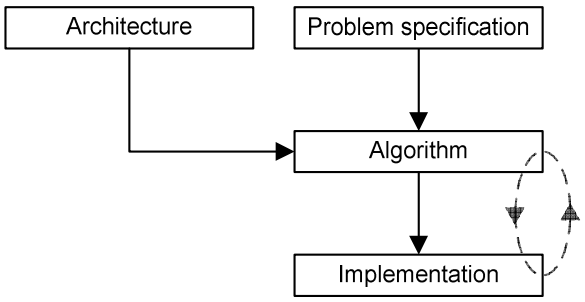

Figure 3: Practical permutation of development process.

In [14] the architecture was implicitly selected because we wanted to use a specific FPGA development board and thus the system architecture was predefined. The algorithm was then shaped to fit the architecture.

To achieve this, the flow alternated between the algorithm and implementation stages, as Figure 3 shows. Each operation in the processing chain was developed, implemented and tested in circuit on the FPGA. The algorithm and implementation were effectively developed in lock-step with one another. This method can be thought of as an attempt to develop the algorithm directly on an FPGA. At the completion of the development process, it was found that a filter operation needed to be inserted into the algorithm to account for troublesome noise [14], requiring further iteration.

8 Summary

FPGAs offer many performance benefits for implementing image processing algorithms. Their general structure can be configured to exploit both spatial and temporal parallelism inherent in images. However, obtaining these benefits can come at significant development cost because an algorithm

and architecture must be simultaneously developed for an application.

This aspect, which is peculiar to FPGA-based development, needs to be reflected in the design cycle. Appropriate hardware design should be performed to select the computational and memory architecture in addition to defining the problem specification, developing an algorithm, and implementing the system (which are also common stages of software design).

Other development issues encountered while following the design cycle include an operating clock frequency one tenth that of a conventional processor, lack of an operating system to oversee common tasks, inefficiency of random access and pointer-based operations and a complex debugging process.

To address these issues we are actively working in several different areas: visual programming languages for hardware [6], windowed operating system for debugging, calibration and tuning [7] and design patterns for capturing and transferring useful implementation techniques [8].

9 References

[1] G. Ottoni, R. Rangan, A. Stoler, M. J. Bridges, and D. I. August, "From sequential programs to concurrent threads," *IEEE Computer Architecture Letters*, vol. 5, pp. 6-9, 2006.

[2] M. Maresca, M. A. Lavin, and H. Li, "Parallel Architectures for Vision," *Proceedings of the IEEE*, vol. 76, pp. 970-981, 1988.

[3] S. M. Trimberger, *Field-Programmable Gate Array Technology*. Massachusetts, USA: Kluwer Academic Publishers, 1999.

[4] M. C. Herbordt, T. VanCourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello, "Achieving High Performance with FPGA-Based Computing," *IEEE Computer*, vol. 40, pp. 50-57, 2007.

[5] I. Alston and B. Madahar, "From C to netlists: hardware engineering for software engineers?," *IEE Electronics & Communication Engineering Journal*, vol. 14, pp. 165-173, 2002.

[6] C. T. Johnston, D. G. Bailey, and P. Lyons, "A Visual Environment for Real-Time Image Processing in Hardware (VERTIPH)," *EURASIP Journal on Embedded Systems*, p. 8, 2006: (Article ID 72962).

[7] D. G. Bailey, K. T. Gribbon, and C. T. Johnston, "GATOS: A Windowing Operating System for FPGAs," in *Proc. of the Third IEEE International Workshop on Electronic Design, Test, and Applications (DELTA 2006)*, Kuala Lumpur, Malaysia, pp. 405-409, 2006.

[8] K. T. Gribbon, D. G. Bailey, and C. T. Johnston, "Using Design Patterns for Image Processing Algorithm Development on FPGAs," in *Proc. of the Third IEEE International Workshop on Electronic Design, Test, and Applications (DELTA 2006)*, Kuala Lumpur, Malaysia, pp. 47-53, 2006.

[9] B. G. Batchelor and P. F. Whelan, "Machine Vision Systems: Proverbs, Principles, Prejudices and Priorities," in *Machine Vision Applications, Architectures and Systems III*, Boston, MA, pp. 374-385, 1994.

[10] D. G. Bailey, "Machine Vision: a Multi-disciplinary Systems Engineering Problem," in *Hybrid Image and Signal Processing* Orlando, Florida: SPIE 939, 1988, pp. 148-155.

[11] J. L. Tripp, M. B. Gokhale, and K. D. Peterson, "Trident: From High-Level Language to Hardware Circuitry," *Computer*, vol. 40, pp. 28-37, 2007.

[12] R. C. Vogt, "Formalized Approaches to Image Algorithm Development Using Mathematical Morphology," in *Proceedings of the Vision '86*, 1986, pp. 5/17-5/37.

[13] S. Y. C. Li, G. C. K. Cheuk, K. H. Lee, and P. H. W. Leong, "FPGA-based SIMD processor," in *11th Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM 2003)*, pp. 267-268, 2003.

[14] C. T. Johnston, D. G. Bailey, and K. T. Gribbon, "Optimisation of a colour segmentation and tracking for real-time FPGA implementation," in *Image and Vision Computing New Zealand*, Dunedin, New Zealand, pp. 422-427, 2005.