2010 Fifth IEEE International Symposium on Electronic Design, Test & Applications

Algorithm Transformation for FPGA Implementation

Donald G Bailey and Christopher T Johnston
School of Engineering and Advanced Technology
Massey University
Palmerston North, New Zealand
e-mail: D.G.Bailey@massey.ac.nz, chris@johnston.geek.nz

Abstract— **High level hardware description languages aim to make hardware design more like programming software. These languages are often used to accelerate legacy software algorithms by porting them to an FPGA based hardware implementation. Porting does not always result in efficient architectures as the original algorithms are usually developed and optimised to run on a serial processor. To obtain an efficient hardware architecture, one that makes use of the available parallelism, the algorithms need to be transformed. Eleven such transformations are identified and explained. While some of these are straightforward, and have been implemented by some compilers, many cannot be automated because they require detailed knowledge of the algorithm.**

Keywords- FPGA, algorithm, architecture, automated design, compilation, hardware description languages

I. INTRODUCTION

Image processing is a well-developed field, with a wide range of operations and a significant code base in the form of algorithms. Recently, many researchers have investigated porting software image processing algorithms onto FPGAs; the two main reasons being to accelerate the algorithm to enable a real-time implementation, or to physically reduce the size and power consumption for an embedded vision application. Image processing, and in particular the low-level operations, are amenable to a parallel implementation on FPGAs [1].

Porting an existing algorithm to a hardware implementation is often very time consuming. From an algorithmic perspective, traditional hardware description languages (VHDL, Verilog) are quite low level, and require a hardware development mindset. In contrast, image processing is predominantly seen as a software development activity. To overcome this disparity and to increase the productivity of developers, a number of high level languages have been introduced that aim to simplify the porting of algorithms onto FPGAs [2]. Two approaches are taken by such languages.

One is to extend a common software language with parallel and hardware constructs. These enable them to be used as algorithmic hardware description languages, with the added advantage that they abstract or hide much of the low-level control. An example typical of languages in this group is Handel-C [2].

The second approach is to compile "standard" serial code to a parallel FPGA structure. With these, the compiler analyses the serial code to identify potential parallelism, then exploits this in mapping the code to hardware. Examples of languages in this second group are the C-based SA-C [3] and Matlab based Match [4], (or AccelDSP as it is marketed). A characteristic of these languages is that they hide the parallel nature of the hardware completely, allowing existing serial algorithms to be used directly.

The limitation of both approaches is that the underlying algorithm is serial. This is due to the serial architecture being so central to modern computing that all of the effort is placed in developing an efficient algorithm, rather than optimising the underlying architecture for the necessary computation [5]. Consequently, many algorithms so tightly coupled to a serial architecture that the parallelism that can be exploited is often limited. While many image processing operations are inherently parallel, particularly for low-level image to image operations, the existing code, and often the underlying algorithm, has been optimised for a serial implementation.

Consequently, using FPGAs to accelerate image processing algorithms can often yield disappointing results. To achieve a significant speed improvement, a large fraction of the algorithm must be able to be parallelised (Amdahl's law). As a result, the performance of a parallel implementation is particularly sensitive to the quality of the design [6]. Therefore, particular attention needs to be made when implementing an algorithm onto an FPGA to not only exploit the obvious and relatively straight forward transformations, but also to transform the underlying algorithm to exploit other types of parallelism that are compatible with the dataflow.

II. ALGORITHM TRANSFORMATION PROCESS

The key to an efficient hardware implementation is not to port an existing serial algorithm, but to transform the algorithm. This is a three step process.

First, the underlying algorithmic architecture is analysed to make the coupling (and flow of data) both within and between the image processing operations more efficient. It is important to look at the whole algorithm, not just the individual operations.

The second step is to design the computational architecture that is implied by the algorithm architecture. This transforms the algorithm to use a resource efficient architecture that can effectively exploit the parallelism available on FPGAs.

978-0-7695-3978-2/10 $26.00 © 2010 IEEE
DOI 10.1109/DELTA.2010.17

The final step is to map the algorithm onto the architecture to obtain the resultant implementation.

This process is usually iterative with the algorithm and architecture mappings being updated as different design decisions are made.

A. Algorithm Analysis

A high-level dataflow analysis is one common method of identifying the underlying computational architecture [7, 8]. This gives the developer an overview of the design and indicates how it might be transformed.

Most software algorithms use a random access processing mode, in which data is accessed from anywhere in memory as needed by the operation. Consequently, the underlying algorithm tends to be memory bound, and can suffer a severe speed penalty when implemented on a lower clock speed FPGA. For example, an operation reads the pixels from memory, processes them, and writes the results back to memory. Since each operation performs this, many clock cycles are required to process each pixel, with the speed of the algorithm limited by the number and speed of memory accesses. The available memory bandwidth provides an upper limit on the processing throughput. Throughput can be increased by maximising the clock speed. This may be accomplished by using low-level pipelining to reduce the combinatorial delay of the computation by spreading it over several clock cycles. The algorithm may be further accelerated by examining the dataflow to identify parallel branches.

Stream processing can overcome the memory bandwidth bottleneck by pipelining operations. The input data is read once from memory, or streamed from the camera and passed into the first operation. Rather than write the results back to main memory, they are directly passed on to the next operation. This will eliminate at least two memory accesses for each stage of the pipeline.

For embedded vision applications, as much processing should be performed on this data while it is passing through the FPGA before it is written to memory. Similarly, if an image is being displayed, the pixels need to be streamed to the display generator. Again, much benefit can be gained by performing as much processing on the fly as the data is being streamed out.

In the context of image capture and display, one of the key characteristics of stream processing is a fixed clock rate, usually one clock cycle per pixel. Consequently, the design of stream processing systems is usually synchronous, at least on the input and output. If every stage of the pipeline can be operated synchronously, then the system design is simplified.

Stream processing is well suited to the low-level image processing operations such as point operations and local filters. For local operations, it is necessary to design buffers for each operation to cache data from adjacent pixels so that it will be available when needed. To accomplish this, operations may need to be significantly redesigned from that conventionally used in software.

The best performance can usually be achieved by minimising accesses to frame buffer memory. To achieve this, the algorithms for each operation may need to be redesigned to make them architecturally compatible. The image processing algorithm may also need to be modified to avoid using operations that do not integrate well with the others.

In some applications, it may not be possible to implement the entire algorithm using streamed processing. The introduction of one or more frame buffers will usually increase the latency of the algorithm. However, for some operations where each output pixel may depend on data from anywhere within the image, this cannot always be avoided.

B. Computational Architecture Design

The computational architecture defines how the computational aspects of the algorithm are implemented. A particular focus is the form of parallelism being exploited to accelerate the algorithm. Low-level image processing operations that can readily exploit parallelism are ideal for hardware implementation.

Many high-level image processing operations are characterised by a lower volume of data and more complex control structures. These provide limited opportunities for exploiting parallelism and often the computation path is dependent on the data being processed. While it is possible to implement such tasks in hardware, it can be more efficient in terms of resource utilisation to implement them in software [9]. Consequently, the overall algorithm for many applications will be partitioned between hardware and software.

The mapping of the algorithm onto the computational architecture is not necessarily one-to-one. There may be several variations on the algorithm that may be implemented using the same (or very similar) hardware architecture.

III. Algorithm Transformations

Three transformations that are currently used by compilers for mapping a serial algorithm onto parallel hardware are low level pipelining, loop unrolling, and strip mining. In some applications, these transformations can give impressive speedups, in spite of the fact that the underlying algorithm is still serial. An additional eight transformations are also described in this section. Many of these are much harder to automate because the way in which many of these transformations are employed depends strongly on the actual algorithm being optimised.

A. Low-level pipelining and retiming

In virtually all non-trivial applications, the propagation delay for the logic required to implement all the image processing operations exceeds the pixel clock period. Pipelining maps a sequence of data dependent operations onto a sequence of concurrent processors, with the output of one processor passed to the input of the next processor in the sequence. Throughput is increased because once the first processor has finished with one item of data, it can immediately begin processing the next item. The length of a pipeline section is usually limited by branch and loop constructs within the code. Both SA-C and Match use directed graphs to identify the sections in the design that can

be pipelined. Automated tools are especially good at low-level pipelining by automatically separating expressions and optimising to maximise clock frequency.

In some instances, the propagation delays of the different stages in a pipeline can vary significantly. The clock rate of a pipeline is limited by the propagation delay of the slowest stage. These imbalances may be improved through retiming [10]. This moves some of the logic from the stages with a long propagation delay through the registers to be executed in either an earlier or later clock cycle. Retiming can be automated, for example it is provided as an optimisation option when compiling Handel-C designs.

Pipelining achieves acceleration not only through having each stage in the pipeline working in parallel, but also by using intermediate registers to hold data between stages. This saves writing the intermediate results to memory only to read them again later.

B. Loop unrolling

Loop unrolling replaces the innermost loops of an algorithm with multiple sequential copies of the operations within the loop construct. This allows longer pipelines to be constructed at the expense of additional hardware. For short loops, this can enable the whole loop to be implemented in parallel. With many image processing operations, these innermost loops are inherently parallel (for example weighting the pixels within a linear filter kernel) and it is only the sequential nature of a serial processor that required a loop within the algorithm.

Since most of the time in an algorithm is spent in the inner loops, accelerating the innermost loops can have a significant impact on the total execution time. Both SA-C and Match can automate loop unrolling.

C. Strip mining

Strip mining is related to loop unrolling in that it creates multiple parallel copies of the loop body. The difference is that each copy operates on different sections of the data. The computation is accelerated by partitioning the input data over different hardware processors. This is effective if each partition can be processed independently, minimising communication between processors. The optimal allocation of resources to maximise throughput subject to resource constraints is still an active research question [11].

With a fixed size input and fixed processing time (for example low level image processing operations) the image may be partitioned in advance. However, if the processing time varies significantly depending on the content of a block, an alternative is to use a processor farm approach [1] to allocate data dynamically to available processors.

D. Pipelined stream processing

The original serial algorithm is usually based on random access processing. However, if the input data is being streamed from a camera, an obvious transformation is to use stream based processing and pipeline the early operations.

Many low-level image processing operations are implemented with the outer loops performing a raster scan through the image. Stream processing serialises the spatial parallelism of the image processing operation. A local operation is able to take the input pixels as a stream, and produce a corresponding stream of output pixels. A sequence of such local operations (contrast enhancement, filtering, thresholding, etc) can be readily pipelined by feeding the output from one operation as the input of the following operation.

Point operations, where the output depends only on a single input pixel, are ideal for stream processing. Filtering is a little more complex; for each output pixel, multiple input pixels are required. This requires that the computational architecture be designed using an appropriate caching scheme so that each input pixel is only input once.

While current analysis tools are good at automating low-level pipelining, high-level pipelining usually requires the algorithm to be constructed as a stream process (eg PixelStreams [12]). Most of the focus within the literature has been on the efficient implementation of individual image processing operations, rather than looking at complete algorithms. The problem is that in software the individual image processing operations are usually separated by memory buffers. Automated tools cannot easily modify the design to remove the buffers and combine the processing into one streamed pipelined design.

E. Operation specific caching

Many image processing operations require multiple input pixels for each output pixel. A naïve implementation would read each pixel from memory as it is required, limiting the operating speed by the memory bandwidth. Caching provides temporary storage for data items that are potentially used multiple times, significantly reducing memory bandwidth [13].

Consider a 3x3 window filter: each output sample is a function of the nine pixel values within the window. In software, the input pixels are accessed though a pair of nested loops. These loops can be unrolled and the required pixels accessed in parallel though the use of a caching arrangement. Pixels adjacent horizontally are required in successive clock cycles, so may simply be buffered and delayed in registers. A row buffer caches the pixel values of previous rows to avoid having to read the pixel values in again.

The regular access pattern of filters makes design of the appropriate caches straight forward (and is built into some languages such as SA-C). However, for more complex operations, an efficient cache design cannot readily be inferred from the source code.

F. Strip rolling and multiplexing

Often, within image processing, data is being extracted from multiple regions simultaneously, especially when using stream processing. An example is measuring the bounding box of each component within a labeled image.

A strip-mining transformation would result in a separate processor for each region. However, a closer look at the algorithm reveals that each pixel belongs to only one region.

This implies that with stream processing, a single region processor may be shared for all of the iterations (strip rolling). Separate data registers must be maintained for each region with the registers multiplexed by the pixel label. In this way, a single processor is able to extract features from an arbitrary number of regions in a single pass through the image. Since only one set of registers is only accessed at a time, they may be implemented using a memory-based data table. The memory addressing logic effectively performs the multiplexing for free.

G. *Data reduction through data coding*

With many images, the volume of data may be significantly reduced by appropriate data coding. For example, run-length coding enables multiple consecutive pixels to be processed simultaneously as a unit. This may result in a significant speed improvement, especially when multiple passes through the image are required.

One example where this has been used is in connected component labeling [14]. Stream processing is used in the first pass, where pixels are transformed into runs, and the initial labels assigned. The second pass, assigning the final labels, may be accelerated, because pixels are processed as runs.

In the worst case, processing the run-length encoded stream will take the same time as the original data. However, for typical images there are significantly fewer runs than pixels, so the latency is significantly reduced. The actual latency, though, would depend on the image complexity.

H. *Algorithm rearrangement*

In some algorithms, it is possible to rearrange the order of operations to simplify the processing complexity, or even eliminate some steps. A classic example of this is greyscale morphological filtering followed by thresholding. This sequence of operations is equivalent to thresholding first, followed by binary morphological filtering. The algorithms for binary filtering are significantly simpler than those for greyscale filtering, resulting in considerable hardware savings and often a reduction in latency.

I. *Operation substitution*

A related transformation is to substitute one or more image processing operations with other, functionally similar (although not necessarily equivalent) operations. One example of this is to use a different method for detecting edges within an image.

Other substitutions may involve approximating operations with similar but computationally less expensive operations. Common examples are replacing the L_2 norm in the Sobel filter with the computationally simpler L_1 or L_∞ norm [15], or modifying the coefficients used for colour space conversion to be powers of two [16].

Note that substituting an operation may change the results of the algorithm. In many image processing applications this does not matter, and the substitution may enable other transformations to be applied to improve the computational efficiency.

J. *Data structure selection*

In software, virtually all of the data structures are implemented within a single monolithic computer memory. Hardware allows a much wider range of physical data structures, and many conventional data structures can be implemented more efficiently in parallel hardware.

The storage on an FPGA ranges in granularity from flip-flop based registers, to shallow RAMs, shift registers and FIFO buffers implemented using logic blocks, through to larger blocks of embedded RAM. The outputs of flip-flops are always available to other parts of the circuit, and most of the embedded RAM within the FPGA can be dual ported. The multiplicity of RAM blocks on an FPGA (as opposed to a single monolithic memory architecture) can alleviate bandwidth issues because different blocks may be accessed in parallel.

Transforming a software algorithm to use a more appropriate hardware structures can significantly speed an algorithm. In particular, data stacks and FIFOs may be used to cache items of significance that may be needed later.

K. *Memory optimisation*

Careful analysis of an algorithm may reveal a way of significantly reducing the memory requirements. For example, with connected components analysis the maximum number of labels in use at any one time by a stream processing implementation depends only on the width of the image. Therefore by rearranging the algorithm to relabel each row can significantly reduce the number of labels, and associated data, required [17]. Often a reduction in memory will have an associated reduction in latency.

IV. DISCUSSION AND CONCLUSIONS

Many of the transformations described in this paper significantly modify the algorithm. Many rely on the developer having a detailed knowledge of the algorithm and how it works. While some of the simpler transformations may be automated to a greater or lesser extent by hardware compilers, many of the transformations require a significant design effort and cannot be automated.

The complete algorithm is given at a relatively high level of abstraction, in terms of individual image processing operations. Some of these operations can be quite complex, and all have their own algorithms at a lower level of abstraction. Many of these operations can be implemented using several possible algorithms. The operations within the software based image processing environment will have been optimised for serial implementation within that environment. Simply porting that algorithm onto the FPGA will generally give relatively poor performance, because it will still be predominantly a serial algorithm. For many low-level image processing operations, the serial algorithm may have a relatively simple transformation to make it suitable for parallel hardware. However, for many intermediate-level operations (such as connected components labelling) the underlying algorithm may need to be completely redesigned to make it more suitable for FPGA implementation.

Despite efforts to make programming FPGAs more accessible and more like software engineering [2], efficient FPGA programming is still very difficult. This is because programming FPGAs is hardware design, not software design. Although the languages may look like software, the compilers use the language statements to create the corresponding hardware. Concurrency and parallelism are implicit in the hardware that is built, and the compilers have to build additional control circuitry to make them execute sequential algorithms [18]. Sequential algorithms ported from software will by default run sequentially. Relatively simple changes to the algorithm, such as pipelining and loop unrolling enable it to exploit some of the parallelism available, and this is often sufficient to compensate for the lower clock speed of FPGAs. However, the algorithm is still largely sequential unless it has been specifically adapted for parallel execution. The software-like languages are in reality hardware description languages, and efficient design requires a hardware mindset.

This means that an algorithm transformation process is needed where the designer considers both the algorithm and the architecture. The algorithm is first analysed to determine the underlying structure. The individual operations are then transformed to share a compatible processing mode. The algorithm is then optimised (through the combining or rearranging of steps) before considering the computational architecture. The mapped design may be resource intensive and modifications to both the algorithm and architecture may be possible to reduce this. This process of design transformation is iterative and needs a human in the loop who understands the detailed operation of the whole algorithm and is thinking in terms of efficient hardware design.

REFERENCES

[1] A. Downton and D. Crookes, "Parallel architectures for image processing," *IEE Electronics & Communication Engineering Journal,* vol. 10, pp. 139-151, June 1998 1998.

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

[3] J. Hammes, B. Rinker, W. Bohm, W. Najjar, B. Draper, and R. Beveridge, "Cameron: high level language compilation for reconfigurable systems," in *Proceedings International Conference on Parallel Architectures and Compilation Techniques*, 1999, pp. 236-244.

[4] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky, "A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems," in *2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000, pp. 39-48.

[5] R. J. Offen, "VLSI Image Processing," 1 ed London: Collins, 1985, p. 326.

[6] 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.

[7] M. Sen, I. Corretjer, F. Haim, S. Saha, J. Schlessman, T. Lv, S. S. Bhattacharyya, and W. Wolf, "Dataflow-based mapping of computer vision algorithms onto FPGAs," *EURASIP Journal on Embedded Systems,* vol. 2007, p. 12 pages, 2007.

[8] 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,* vol. 2006, p. 8 pages, 2006.

[9] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys,* vol. 34, pp. 171-210, June 2002.

[10] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica,* vol. 6, pp. 5-35, 1991.

[11] Q. Liu, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, "Combining data reuse exploitation with data-level parallelization for FPGA targeted hardware compilation: a geometric programming framework," in *International Conference on Field Programmable Logic and Applications (FPL 2008)*, Heidelberg, Germany, 2008, pp. 179-184.

[12] Celoxica, *PixelStreams User Manual* vol. 5752-5-0: Celoxica Limited, 2007.

[13] M. Weinhaudt and W. Luk, "Memory access optimisation for reconfigurable systems," *IEE Proceedings - Computers and Digital Techniques,* vol. 148, pp. 105-112, 2001.

[14] K. Appiah, A. Hunter, P. Dickenson, and J. Owens, "An run-length based connected component algorithm for FPGA implementation," in *International Conference on Field Programmable Technology*, Taipei, Taiwan, 2008, pp. 177-184.

[15] I. E. Abdou and W. K. Pratt, "Quantitative design and evaluation of edge enhancement / thresholding edge detectors," *Proceedings of the IEEE,* vol. 67, pp. 753-763, 1979.

[16] G. Sen Gupta and D. Bailey, "A new colour-space for efficient and robust segmentation," in *Image and Vision Computing New Zealand (IVCNZ'04)*, Akaroa, New Zealand, 2004, pp. 315-320.

[17] D. G. Bailey, C. T. Johnston, and N. Ma, "Connected components analysis of streamed images," in *International Conference on Field Programmable Logic and Applications (FPL 2008)*, Heidelberg, Germany, 2008, pp. 679-682.

[18] I. Page and W. Luk, "Compiling Occam into field-programmable gate arrays," in *Field Programmable Logic and Applications*, Oxford, UK, 1991, pp. 271-283.