The Advantages and Limitations of High Level Synthesis for FPGA Based Image Processing

Donald G. Bailey
School of Engineering and Advanced Technology
Massey University
Palmerston North, New Zealand
D.G.Bailey@massey.ac.nz

ABSTRACT

High level synthesis (HLS) tools can provide significant benefits for implementing image processing algorithms on FPGAs. The higher level (usually C based) representation enables algorithms to be expressed more easily, significantly reducing development times. The higher level also makes design space exploration easier, making it easier to optimise the trade-off between resources and processing speed. However, one danger of using HLS is simply porting existing image processing algorithms onto an FPGA platform. Often, better parallel or pipelined algorithms may be may be designed which are better suited to the FPGA architecture. Examples will be given from image filtering, to connected components analysis, to efficient memory management for 2-D frequency domain based filtering.

CCS Concepts

•Hardware → High-level and register-transfer level synthesis; Reconfigurable logic applications; •Computing methodologies → *Computer vision problems;*

Keywords

rapid development; design space exploration; architecture exploration; smart camera

1. INTRODUCTION

With increased processing power, there has been a trend towards performing image processing within the camera itself, rather than using a separate computer. Such "smart cameras" no longer simply capture images, but also perform much of the image processing and communicate the processed images or even just the data extracted from the images [10]. However, even with advances in device technology, low power general purpose processors often struggle with the processing demands of image processing applications. Initially, digital signal processors were commonly used to implement smart cameras [9, 36]. More recently, field programmable gate arrays (FPGAs) have been gaining popularity for image processing within smart cameras [11, 16, 27] because modern FPGAs now have sufficient resources to implement all of the processing for a complete application within a single low cost FPGA [3].

The conventional approach to developing the firmware for such FPGA based embedded vision applications is to first develop and test the algorithms on using a general purpose software based image processing system. This is because algorithm development, and in particular testing, is much easier to perform on a software platform than it is directly in hardware [3]. These high level algorithms are effectively behavioural models of the design, and are usually written in C or C++ because such languages provide efficient system descriptions, and are supported by a broad range of compilers, development and debugging tools. Once the algorithm is functioning as desired, it is then ported or mapped to a hardware description language such as VHDL or Verilog for FPGA implementation. Such lower level design is usually at the register transfer level (RTL). Recoding of complex algorithms as RTL designs is both tedious and error prone. This, coupled with the large software code base of image processing algorithms, has led to development of compilers which can directly synthesise the RTL hardware from the high level software designs [1]. It is this characteristic that is the basis of many of the advantages of high level synthesis (HLS). There is a wide range of HLS tools available, which differ significantly in both their ease of use and the quality of hardware derived [25].

There are five key challenges that such HLS tools must overcome [17]. These form the basis of many of the limitations of HLS:

- Hardware in inherently concurrent, whereas software representations are sequential. HLS must map the sequential algorithm onto concurrent hardware.
- Timing is implicit in software in the sequence of instructions used. Synchronous hardware must deal with timing constraints, with controlling and synchronising operations at the clock cycle level.
- In software, the word length is fixed (8, 16, 32 or 64 bits), but in hardware, it is usually optimised for the task being performed [8, 13].
- The software model of memory is as a single block with a monolithic address space, with almost all data items stored in memory. On an FPGA, local variables are stored in registers, with multiple distributed memory

ICDSC '15, September 08 - 11, 2015, Seville, Spain
© 2015 ACM. ISBN 978-1-4503-3681-9/15/09…$15.00
DOI: http://dx.doi.org/10.1145/2789116.2789145

blocks, each with their own independent address space. In such an environment, pointers have little meaning, and dynamic memory allocation is very difficult.

- Communication in software is usually through shared memory, whereas on an FPGA it relies on constructing appropriate hardware, from implicit (within stream processing), to simple token passing, to using dedicated FIFOs to manage flow control.

Modern HLS tools generally perform the following steps: dataflow analysis (to determine the operations that need to be performed); resource allocation (to determine how many hardware operators are needed to execute all of the source code operations); resource binding (to allocate source code operations to hardware resources); and scheduling (to determine when each source code operation is executed on the allocated hardware).

HLS offers the promise of enabling software engineers to implement hardware [1, 28]. In some ways, this is true, and these are explored in section 2 looking at the benefits of HLS. In other ways, HLS does not live up to its promises, and the limitations of current HLS tools are analysed in section 3. Section 4 presents three case studies of standard image processing operations where a naïve HLS approach fails. Section 5 summarises with some concluding remarks.

2. BENEFITS OF HLS

With high level synthesis, code can be relatively easily ported from software to a hardware implementation. Many of the current generation compilers can work with standard C code, and compile from C to RTL, and rely on the FPGA vendor specific tools for synthesising this onto an FPGA [6]. However, simply compiling software code for FPGA is seldom sufficient. The software algorithm has often been optimised for implementation on a CPU based processor. It is usually necessary to restructure the algorithm for it to better suit hardware. However, rather than perform this step manually in the translation from C to RTL, with HLS this restructuring can be performed directly on the high level source, which is easier and less error prone [6].

One of the key benefits of working directly in a high level language is that the control path is often implicit in the language representation. HLS tools analyse the structure of the algorithm (loops, branches, etc.) to automatically extract and build the control path. In contrast with this, manual implementation in RTL requires explicitly coding the control path as well as the data path. For complex algorithms, designing the control path can take as much effort as designing the data path.

Modern synthesis tools are able to exploit parallelism in three main ways [18]:

- By performing a dataflow analysis to determine data dependencies. From this, the processing sequence can be inferred, and pipelined if necessary to meet desired timing constraints.

- By analysing loops which involve significant processing with limited dependencies between iterations. This enables a pipelined architecture to be inferred, where a subsequent iteration can be started before the previous iteration has completed. Such processing fits well with a streamed type of architecture.

- By unrolling loops. Multiple parallel hardware blocks are built to enable the iterations to be parallelised.

Design space exploration involves exploring different combinations of these to make explicit the trade-off between speed and resources. This is usually accomplished through a combination of source code optimisation and synthesis directives, depending on the particular tool [25]. Within HLS, software profiling tools can help to identify processing bottlenecks, enabling more effort to be concentrated on where it can potentially achieve the greatest gains. Many of the different factors involved in a design space exploration are controlled by synthesis directives . Coupled with this, the HLS tools can provide a reasonably accurate estimate of resources without having to synthesise the resulting RTL [6]. Such estimates enable the effect of different combinations of parallelism and source code optimisations to be quickly evaluated, facilitating early design exploration. In contrast, manual RTL coding will generally require considerable recoding to change both the data and control paths, making design space exploration at the RTL level both time consuming and error prone.

For each of the designs generated, simulation or verification is significantly faster within the high level tools. This is because the verification takes place at a higher level [31]. However, it is still necessary to validate the final design at the RTL level to ensure that the algorithm transformations are correct [31]. In some tools, the RTL test-benches can be automatically generated from the high level verification code facilitating this step [25].

In co-processor environments, where the computation is shared between an FPGA and CPU, HLS enables the same source code to be compiled either as hardware or software (for example [29, 35]). This enables different modules to be rapidly swapped between hardware and software implementation, making it easier to find the optimum mix.

For appropriately structured code, modern HLS tools can generate designs that are as efficient as hand-coded RTL in terms of both resources and processing speed [6, 25, 37]. Given this, why would anyone still code in RTL?

3. LIMITATIONS OF HLS

Unfortunately, it is not as simple as just compiling code for hardware. With most tools, the algorithm must be written in a particular style to enable the synthesis tools to identify and exploit parallelism [18]. This requires restructuring the code. Without such restructuring, the HLS tools can still derive a hardware realisation, but the resulting hardware can be bloated and suffer from poor performance [20].

The key factor here is that FPGA based design is hardware, not software design. The synthesis language (whether RTL or HLS) is describing hardware, even though many languages strongly show their software roots. Each statement describes hardware which must be built, rather than providing a set of instructions to be executed. While modern HLS tools can reuse hardware for different statements, this is dependent on the compiler being able to recognise that they occur at different times so they can be successfully scheduled. With many high level languages for synthesis, it is very easy to slip into a software mind-set, especially if describing algorithms and the developer has had significant software experience. Treating hardware description languages like software can often lead to inefficient use of hardware. While algorithmic representation for software is mature, for hardware realisation it is still in its relative infancy in spite of ongoing research in this area (see for example [7, 14, 21]).

Algorithms based strongly on pointers and pointer arithmetic do not synthesise well to hardware [37]. In software, a pointer is an address of a variable in memory. There are two difficulties this poses in an FPGA implementation. First, on-chip memory is distributed, with many relatively small, independent blocks, each with their own address space. Second, in hardware many variables are stored in registers. While it is possible to emulate pointers within such a system, it is not particularly transparent and the resulting hardware is somewhat unwieldy. Another problem with pointers is that they can obscure the data-flow, making it difficult for the HLS tools to identify data dependencies. If possible, such algorithms need to be restructured to use array references, with the relevant arrays mapped into one or more distributed memory blocks.

Recursion is another software technique that does not translate well to hardware. Functions or procedures in software store local variables on the stack, so that each call has separate space for variable storage. On an FPGA, a function is implemented as a block of hardware, with a procedure call multiplexing that hardware for each invocation. There is no stack; local variables are generally stored in registers, which are shared with every call of the procedure. Reuse of registers by successive calls limits the use of recursive functions to tail-recursion. In general, recursive algorithms must be restructured to use an equivalent iteration.

In general, concurrent system design is difficult because of the need to design synchronisation. Single-threaded software design does not have this problem, because everything is implicitly sequential. With multiple threads, it is necessary to ensure synchronisation at key points. This is usually managed at a relatively course grained level though mailboxes or through fork/join constructs, enabling multiple cores or processors to be exploited. In hardware concurrency is much finer grained, with every logic block effectively running concurrently. At this finer level, there are many more tasks to synchronise. While HLS does reasonably well at scheduling and pipelining sequences of operations, and synchronising distribution of computation over multiple parallel processors, it struggles with more complex synchronisation.

The RTL produced by HLS tools is not particularly human-readable making the code very difficult to modify at that level [32]. In general, the purpose for using HLS is to avoid needing to do any RTL programming. However, it is still necessary to verify the RTL output [31, 32]. If RTL verification indicates failure, it can be very difficult to determine what exactly is causing the problem in the HLS [32].

Regardless of representation, the best algorithms for hardware realisation are not necessarily the same as those used in software [3, 23]. While HLS tools will produce hardware for realising the algorithm, there may be better or more appropriate algorithms. In general, software is memory based, with all data structures stored in a single monolithic memory (at least logically, if not in reality). Many software algorithms, particularly for image processing, are therefore memory bound with their execution speed limited by memory bandwidth. On FPGAs, stream based processing is efficient for image processing [3], especially for operations close to the camera (pre-processing) or display where the input and output data are naturally streamed. In some cases, algorithms which rely on random data access (the paradigm normally used by software) can be restructured to enable stream processing (see for example [2]).

4. CASE STUDIES

The main limitation of using HLS is that it does not remove the need for hardware design. If it is realised that the language is not software, but actually describing hardware, then it is possible to use the high level languages to also describe relatively low level constructs where necessary. The first case study gives an example of where this is the case. The second study demonstrates that algorithm transformation can result in significant improvements over an existing software algorithm. In this example, the transformation is beyond what can be performed by synthesis tools. The third example explores both algorithm transformation, and memory mapping to improve the performance of an algorithm.

4.1 Image filtering

Image filtering is one of the most commonly used examples of the benefits of HLS. This is because it is one of the most common image processing operations, and the design of efficient filters is non-trivial. Several HLS tools can readily identify the data access patterns associated with filtering, and build the necessary row buffers and windowing to enable efficient streamed implementation (see for example [12, 19, 26]). Other HLS tools identify the access patterns associated with filters, and substitute an optimised, pre-defined windowing architecture (such as those designed in [33]).

The problem with filtering comes when handling the image borders. When part of the filter window extends past the edges of the image, the access pattern in perturbed. In software, this is commonly managed by having conditional code within the filter to handle the different cases around each image border. HLS struggles to integrate these with the standard access pattern, resulting in failure to recognise the windowing access pattern, separate hardware being built for each case, or inefficient reuse of hardware. Because of the number of exceptions (four edges and four corners of the image), it is easy for this additional hardware to be significantly larger than that for handling the normal case.

However, with careful design, this need not necessarily be the case. In [4] it is shown that the additional border management logic can be integrated with window creation with modest additional resources (a few multiplexers and registers, as shown in Fig. 1 for border mirroring). It can provide one window per clock cycle, with no blanking delays required between rows or frames). By separating the window formation from the filter function, the logic for the filter function is unchanged. While this could be integrated into predefined windowing architectures, this would require additional directives to select border management.

4.2 Connected components analysis

Many algorithms require multiple processing passes through the image. This necessitates storing the intermediate images from each pass within a frame buffer. A typical example of such an algorithm is connected components analysis.

In software, multiple passes through the image are typically used [30]. First, two passes are required to perform connected component labelling, which labels each set of connected pixels within the image with a unique label. Then each component is selected from the image for processing to

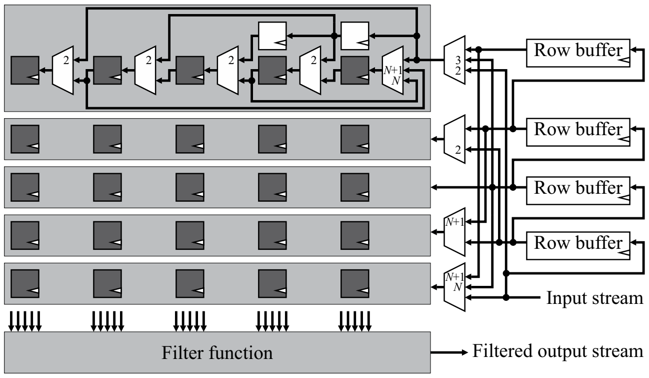

Figure 1: Border management by mirroring pixels with a 5×5 window, from [4]. Window registers are shaded dark grey. Row logic is repeated for each row.

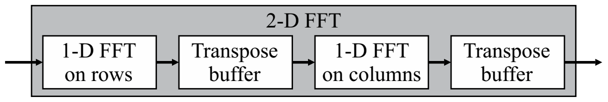

Figure 2: A two dimensional Fourier transform implemented separably as a row transform followed by a column transform.

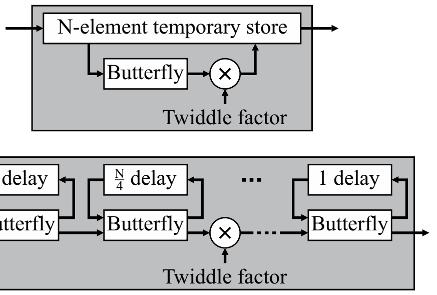

Figure 3: Architectures for 1-D FFT. Top: sequential in-place algorithm, using on-chip memory for the temporary store. Bottom: a pipelined radix-2^2 algorithm.

extract features of that component, such as its area, centre of gravity, shape, etc.

Since the algorithm is largely sequential, the data dependency on the current pixel's predecessors makes parallelisation non-trivial. When directly synthesised for hardware, such an algorithm would retain the same sequential computational architecture, at least for the first two passes. Feature extraction for each component is independent of the other components, so is readily parallelisable, at the expense of building multiple copies of the feature extraction logic.

While a sequential algorithm is adequate for processing on a CPU based processor, there exist better parallel algorithms. Unless one of these parallel algorithms is coded, it is difficult, if not impossible, for the synthesis tools to efficiently parallelise. For stream processing, a single pass connected components analysis algorithm is described in [22] and is further optimised in [24]. These algorithms make use of the observation that if features are extracted in the first pass, then it is not necessary to produce a labelled image, making the second pass unnecessary. With stream processing, features for multiple connected regions are extracted in parallel; however the logic for feature accumulation can be shared because each pixel can only belong to one connected component. Such a single pass algorithm significantly reduces both resources (it does not need to save the image in a frame buffer within the processing) and latency (because with a single pass algorithm, the results can be output as soon as the object is completed).

This example illustrates that simply porting or restructuring a software algorithm is insufficient for an efficient hardware implementation. It is necessary to transform the algorithm [5] to gain the maximum benefits from the hardware available. The key transformations require a high level understanding of the algorithm and involve an element of design. Such transformations and optimisations are beyond the capabilities of current HLS tools. However, there is nothing preventing the resulting transformed design from being expressed using a high level language.

4.3 Two dimensional FFT

A two-dimensional FFT can efficiently be implemented in two stages because the FFT is separable. First, a 1-D FFT is performed on each row in the image, followed by a 1-D FFT on each column. This is generally implemented by saving the results of the first stage into a transpose buffer (see Fig. 2), because the column FFT requires data from all of the rows. Similarly a transpose buffer may also be required on the output if a row-streamed output is required.

For the 1-D FFTs, HLS should enable design space exploration within each FFT allowing a mix of parallel and pipelined stages to be explored for the implementation. Since each row is processed independently, HLS should also be able to infer that multiple rows can be transformed in parallel (at the expense of increased resources). However, the size of the transpose buffers means they will need to be implemented using external (to the FPGA) memory and the memory bandwidth for loading and saving the data will ultimately limit the processing speed. Design space exploration can optimise hardware utilisation for the available bandwidth. Such exploration is much easier to perform using HLS than RTL synthesis.

However, given standard software implementations of the FFT where the FFT is performed in-place on data in memory, HLS is likely to infer a variation of the architecture shown in the top panel of Fig. 3. It is unlikely to infer the pipelined FFT algorithm (bottom panel of Fig. 3 [34]) unless the algorithm has been very carefully written to imply such a structure. If a radix-2 algorithm is coded, it will not infer the more efficient radix-2^2 algorithm (which has half the number of complex multiplications).

Even so, this is still only part of the problem. As mentioned, the transpose buffer will need to be located in external memory. The problem comes when this is stored in dynamic RAM (DRAM). Modern synchronous DRAM [15] is structured in rows, with only one row accessible at a time. Before accessing a row, it must first be activated (the contents transferred from internal capacitor storage into a set of parallel sense amplifiers). Once activated, random access to any address within the row is permitted. Finally, the row must be pre-charged, which transfers data back to the inter-

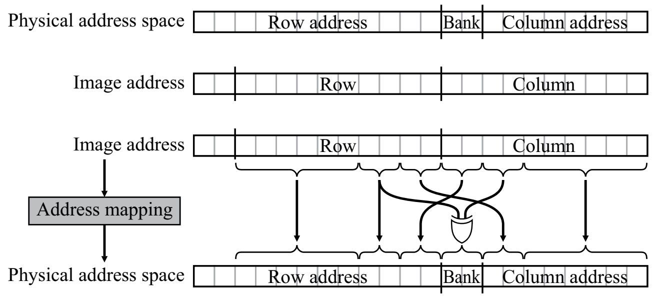

Figure 4: DRAM address mapping. Top: typical interleaved DRAM address space. Middle: worst case example – successive rows are in the same memory bank. Bottom: example address map for solving this problem.

nal capacitor storage and pre-charges the column lines for the next read. In practise, it is a little more complex than this, with the memory divided into multiple banks. One row may be open within each bank, enabling the activation and pre-charge overhead to be hidden by interleaving the memory such that at the end of a row the next logical address is in a different bank. Normally, much of this complexity is hidden by the memory controller.

Accessing the image along the rows is generally straightforward, as burst mode can efficiently access (for reading or writing) successive locations on the same memory row. The overhead in moving from one memory row to the next can be hidden through interleaving.

Accessing by column, however, can incur considerable overhead from activating and pre-charging successive memory rows within the same bank. In the worst case, when successive image rows correspond to different physical memory rows within the same memory bank (as shown in Fig. 4), the overhead cannot be hidden, and the available memory bandwidth can be reduced by over 80% in the worst case. HLS tools are generally unaware of this issue, as the problem depends on both the length of the FFT and also the structure of the physical memory chips (which is hardware dependent).

This can be alleviated (and this overhead hidden) by careful mapping of the logical image address to a physical address. One such mapping is shown in Fig. 4. Some of the image row bits are mapped into the column address space to ensure that multiple successive image rows are within the same memory row. This allows multiple accesses within an open row before the bank is switched. A consequence is that some of the image column bits are mapped into the row address space, and multiple memory rows need to be opened to read or write an image row. A second optimisation is to combine some of the row address bits with the column bits in selecting the bank. This ensures that when traversing a column, successive image rows will not be in the same bank. This allows the next bank to be opened in advance, hiding the activation overhead.

To maximise memory bandwidth, it is necessary to use burst mode to free the control (address) bus, enabling the activation and pre-charge commands to be issued in parallel with data transfer. In this way the overhead is hidden without affecting data throughput. This is natural when accessing the image by row, however when accessing the images by column, it may be necessary to access two or more successive columns (depending on data width) to enable burst mode to be used. If multiple columns are not processed in parallel, then then they can be buffered (cached) in on-chip memory. This may increase the latency slightly, but enable the maximum throughput to be maintained.

Note that such a memory mapping scheme must be designed, and cannot be inferred by current HLS tools. It may also be necessary to have finer control over the memory controller (or design a custom controller to incorporate the mapping).

5. SUMMARY

While using high level synthesis can provide significant advantages for rapid development and design space exploration, it is no substitute for careful design. It is still essential to consider the hardware being built, and treat the HLS tool as a hardware description language and not as software.

The operations within a software based image processing environment have been optimised for CPU based implementation. Simply recompiling that algorithm will give relatively poor performance. It is necessary to restructure the algorithm to enable HLS to identify and exploit the parallelism. HLS is good at exploring the design space through partial loop unrolling and pipelining. This works well for low level image processing operations where the control structure is relatively simple. However, for intermediate-level operations (for example connected components analysis) HLS is not able to redesign the algorithm to make it more suitable for FPGA implementation. Such tasks are still up to the developer.

HLS can be used to represent such redesigned algorithms, and enable more compact and efficient coding than RTL coding. High level code is easier to write, because the algorithm control is implicit within the language constructs, and does not require explicitly designing the separate control logic. This also makes high level code easier to read and maintain, and verification at the higher level is also significantly faster.

In conclusion, HLS offers many benefits over conventional RTL implementation for FPGA based design. However, simply using a high level language does not alleviate the need for appropriate hardware design.

6. REFERENCES

[1] I. Alston and B. Madahar. From C to netlists: Hardware engineering for software engineers? *IEE Electronics and Communication Engineering Journal*, 14(4):165–173, 2002.

[2] D. Bailey. Chain coding streamed images through crack run-length encoding. In *Image and Vision Computing New Zealand*, 6 pages, 2010.

[3] D. G. Bailey. *Design for Embedded Image Processing on FPGAs*. John Wiley and Sons (Asia) Pte. Ltd., Singapore, 2011.

[4] D. G. Bailey. Image border management for FPGA based filters. In *6th International Symposium on Electronic Design, Test and Applications*, pages 144–149, 2011.

[5] D. G. Bailey. Invited paper: Adapting algorithms for hardware implementation. In *7th IEEE Workshop on Embedded Computer Vision*, pages 177–184, 2011.

[6] BDTI. High-level synthesis tools for Xilinx FPGAs. Technical report, Berkley Design Technology Inc., 2010.

[7] A. Benkrid and K. Benkrid. HIDE+: A logic based hardware development environment. *Engineering Letters*, 16(3):460–468, 2008.

[8] D. Boland and G. A. Constantinides. A scalable approach for automated precision analysis. In *International Symposium on Field Programmable Gate Arrays*, pages 185–194, 2012.

[9] M. Bramberger, J. Brunner, B. Rinner, and H. Schwabach. Real-time video analysis on an embedded smart camera for traffic surveillance. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 174–181, 2004.

[10] M. Bramberger, A. Doblander, A. Maier, B. Rinner, and H. Schwabach. Distributed embedded smart cameras for surveillance applications. *IEEE Computer*, 39(2):68–75, 2006.

[11] P. Chalimbaud and F. Berry. Design of an imaging system based on FPGA technology and CMOS imager. In *IEEE International Conference on Field-Programmable Technology*, pages 407–411, 2004.

[12] J. Cong, P. Zhang, and Y. Zou. Optimizing memory hierarchy allocation with loop transformations for high-level synthesis. In *49th Annual Design Automation Conference*, pages 1229–1234, 2012.

[13] G. A. Constantinides, P. Y. Cheung, and W. Luk. Optimum wordlength allocation. In *Symposium on Field-Programmable Custom Computing Machines*, pages 219–228, 2002.

[14] D. Crookes, K. Alotaibi, A. Bouridane, P. Donachy, and A. Benkrid. An environment for generating FPGA architectures for image algebra-based algorithms. In *International Conference on Image Processing*, volume 3, pages 990–994, 1998.

[15] B. Davis. *Modern DRAM Architectures*. PhD thesis, 2001.

[16] F. Dias, F. Berry, J. Serot, and F. Marmoiton. Hardware, design and implementation issues on a FPGA-based smart camera. In *First ACM/IEEE International Conference on Distributed Smart Cameras*, pages 20–26, 2007.

[17] S. A. Edwards. The challenges of synthesizing hardware from C-like languages. *IEEE Design and Test of Computers*, 23(5):375–383, 2006.

[18] M. Fingeroff and T. Bollaert. *High Level Synthesis Blue Book*. Mentor Graphics Corporation, 2010.

[19] Z. Guo, B. Buyukkurt, and W. Najjar. Input data reuse in compiling window operations onto reconfigurable hardware. *ACM SIGPLAN Notices*, 39(7):249–256, 2004.

[20] M. Herbordt, T. VanCourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello. Achieving high performance with FPGA-based computing. *IEEE Computer*, 40(3):50–57, 2007.

[21] C. T. Johnston. *VERTIPH: A visual environment for real-time image processing on hardware*. PhD thesis, 2009.

[22] C. T. Johnston and D. G. Bailey. FPGA implementation of a single pass connected components algorithm. In *IEEE International Symposium on Electronic Design, Test and Applications*, pages 228–231, 2008.

[23] Y. K. Lim, L. Kleeman, and T. Drummond. Algorithmic methodologies for FPGA-based vision. *Machine Vision and Applications*, 24(6):1197–1211, 2013.

[24] N. Ma, D. Bailey, and C. Johnston. Optimised single pass connected components analysis. In *International Conference on Field Programmable Technology*, pages 185–192, 2008.

[25] W. Meeus, K. V. Beeck, T. Goedeme, J. Meel, and D. Stroobandt. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3):31–51, 2012.

[26] W. Meeus and D. Stroobandt. Automating data reuse in high-level synthesis. In *Conference on Design, Automation and Test in Europe*, 4 pages, 2014.

[27] R. Mosqueron, J. Dubois, and M. Paindavoine. High-speed smart camera with high resolution. *EURASIP Journal on Embedded Systems*, 2007(Article ID 24163):8 pages, 2007.

[28] I. Page. Closing the gap between hardware and software: Hardware-software cosynthesis at Oxford. In *IEE Colloquium on Hardware-Software Cosynthesis for Reconfigurable Systems (Digest No: 1996/036)*, pages 2/1–2/11, 1996.

[29] A. A. H. B. A. Rahman, R. Thavot, M. Mattavelli, and P. Faure. Hardware and software synthesis of image filters from CAL dataflow specification. In *2010 Conference on Ph.D. Research in Microelectronics and Electronics*, 4 pages, 2010.

[30] A. Rosenfeld and J. Pfaltz. Sequential operations in digital picture processing. *Journal of the Association for Computing Machinery*, 13(4):471–494, 1966.

[31] J. Sanguinetti. Understanding high-level synthesis design's advantages. *EE Times Asia*, pages 1–4, 26 April 2010.

[32] W. Savage, D. Garrett, S. Rawat, and O. Gunasekara. Panel discussion: Who drives whom? High-level synthesis or IP reuse? In *Design Automation Conference*, 2014.

[33] M. Schmid, N. Apelt, F. Hannig, and J. Teich. An image processing library for C-based high-level synthesis. In *24th International Conference on Field Programmable Logic and Applications*, 4 pages, 2014.

[34] S. Sukhsawas and K. Benkrid. A high-level implementation of a high performance pipeline FFT on Virtex-E FPGAs. In *IEEE Computer society Annual Symposium on VLSI*, pages 229–232, 2004.

[35] F. M. Vallina and J. R. Alvarez. Using OpenCV and Vivado HLS to accelerate embedded vision applications in the Zync SoC. *XCell Journal*, 83:24–30, 2013.

[36] P. Wills. *The hardware design of a smart camera for the robot soccer environment*. BE(Hons) thesis, 1999.

[37] F. Winterstein, S. Bayliss, and G. A. Constantinides. High-level synthesis of dynamic data structures: A case study using Vivado HLS. In *International Conference on Field Programmable Technology*, pages 362–365, 2013.