Accelerating Image Algorithm Development using Soft Co-Processors on FPGAs

Tiantai Deng, Danny Crookes, Roger Woods and Fahad Siddiqui
School of EEECS, Queen's University Belfast, Belfast BT7 1NN, UK

***Abstract*—FPGAs can offer high performance with low power and low hardware usage. However, with current software, FPGAs are hard to program, and lengthy re-synthesis times make them unsuitable for the algorithm experimentation which is typical of developing image processing applications. In this paper, we present a system model based on a set of Soft Co-Processors, each of which implements a basic image-level operation (or a common combination of such operations) based on the high-level operators in Image Algebra. Both 'debug' (generic but unoptimised) and 'release' (specific and optimised) versions of the Soft Co-Processors can be used. The advantage of debug mode is that no re-synthesis is required during algorithm experimentation. For release mode, a novel macro-based transformation tool enables the creation of a set of reusable HLS skeleton co-processors which require the user only to write a C function to obtain a new, special-purpose Soft Co-Processor.**

Initial experiments with several algorithms show that the area and speed overheads for using debug (rather than release) mode are both around 25-30%, thus enabling algorithm development to take place on the FPGA itself. For creating function-specific Co-processors using our macro-based tool, the overheads compared with an expert hardware design are around 20%.

***Index Terms*—Image Processing, FPGA, Image Algebra**

I. INTRODUCTION

Image processing algorithms are used in many applications, such as image classification, medical image processing, video surveillance and target detection and tracking [1-3]. Using the concepts of Image Algebra (IA) [4], many image processing algorithms can be expressed as a combination of basic image-level operations, including: point operations, neighbourhood operations and global operations.

Image processing applications usually require the processing of large amounts of data, in some cases in real-time. Possible hardware platforms include multi-core CPUs, Graphic Processing Units (GPUs), and Field Programmable Gate Arrays (FPGAs). FPGAs can offer high computation capability and high bandwidth, and can also have the benefit of low power [5, 6]. Currently, however, implementation on FPGAs is very design intensive and require hardware design knowledge, so FPGAs are usually only used to run a stable image processing algorithm. Unfortunately, the development of image processing algorithms tends to be experimental and iterative. Even if it was possible to speed up the FPGA design process, the usual lengthy synthesis time is not acceptable for algorithm experimentation and tuning.

Several tools have been designed to bridge the gap between hardware design and high-level programming. Vivado HLS allows developers to use C syntax to develop dedicated hardware on FPGAs. However, it is difficult to tune architectures using HLS without re-synthesising the whole system on an FPGA [5, 6]. In addition, despite using C syntax, a developer must still think in terms of hardware design. The developer must understand what the software tools will generate, in case they write code which the tools cannot handle efficiently. Also, the IP cores generated by HLS need to be integrated with the rest of the system using Vivado. If we use a soft-processor on FPGAs (such as the Xilinx MicroBlaze), which is a common approach, we have to trade performance for programmability, because the multiple fetch-execute cycles interrupt the dataflow stream processing.

FPGAs have a lot of computing resources but limited memory, and the efficient use of memory resources is crucial to system performance. A skilled developer can choose the optimal memory management approach from a vast range of possibilities in a way that existing tools are incapable of doing.

Given the current state of the art, three main challenges for developing image processing systems on FPGAs are:

- The long synthesis time during iterative algorithm development and tuning is not acceptable.
- Balancing programmability against performance.
- The hardware design time for new algorithms.

In order to address these challenges, this paper presents a system model based on a set of IA based Soft Co-Processors on an FPGA, with an AXI-Stream interconnection-based system, which allows users to develop and experiment with their algorithm without having to re-synthesize the whole system. We call this 'debug' mode. We secondly provide a simple code transformation tool to enable the development of optimized co-processors with minimal coding effort. This gives our 'release' mode. More specifically, the main contributions are as follows:

- A set of parameterized Soft Co-Processors on FPGAs for each of the core IA types of image-level operation: point operations, neighbourhood operations, and global operations. We also provide co-processors for common multi-function operations, such as a neighbourhood operation followed by a point operation. The co-processors use optimized memory management, and can be linked together for full algorithm development.
- A flexible AXI-Stream-based system which allows users to link co-processors in any pattern, corresponding to the required dataflow model of the application.

978-1-5386-6046-1/18/$31.00 ©2018 IEEE

- A simple code transformation tool in Vivado HLS, which allows users to define their own customized Soft Co-Processors. Unlike the tool in [7], it is not a code generation tool, but a lightweight macro-based transformation tool which exploits the C preprocessor.

II. Background and Related Work

A. Image Algebra

Image Algebra (IA) [4] is a mathematical theory concerned with the transformation and analysis of digital images at the whole image (rather than pixel) level. The main goal is the establishment of a comprehensive and unifying theory of image transformations, image analysis, and image understanding. Basic IA operations can be classified as: point operations, neighbourhood operation, and global operations.

1) Point Operation (P2P): In a point operation, the same operation is applied at every input pixel position. Operations can be binary or unary; they include relational (e.g. '>', '='), arithmetic (e.g. '+', '×'), and logical (e.g. 'and', 'or') operations. It must normally generate one output pixel for each corresponding input pixel position. A point operation produces an output image of the same size as the input image(s).

2) Neighbourhood Operation (N2P): A neighbourhood (window) operation is applied to each (potentially overlapping) region of image. It is common to use a 3×3 or 5×5 window. A new pixel value will be generated for each window position. The same operation is applied at each window position. The size of the result image may be slightly different from the input image size because the window at boundary pixel positions may exceed the image limits. The neighbourhood operation at each window position has two phases: an initial point operation, and a secondary reduction operation. For example, for convolution, the two operations are multiplication and summation. In the original IA, the configuration of windows can be location-dependent, but implementations often restrict windows to a fixed configuration (as we do).

3) Global Operation (R2S or R2V): A global operation is a reduction operation which is applied to the whole image and produces a scalar (R2S) or a vector (R2V). For example, the global maximum will produce one scalar value, whereas histogram will produce a 256-element vector.

B. Implementing Image Processing Algorithms on FPGAs

When an image processing algorithm is implemented on FPGAs, the algorithm usually processes a stream of pixels in order to increase task parallelism and save memory resources. It is necessary to arrange internal memory differently according to the operations. For example, point operations do not need buffers, but neighbourhood operations require line buffers to hold the relevant pixels within the window.

There is much existing work to indicate how to implement neighbourhood operations on FPGAs. In [8-11], several comparisons have been carried out, showing the advantage of line buffering for efficient data management. In [12], Yu and Leeser implemented a highly-parallel system for edge detection. To get the best performance, they used two line-buffers and two off-chip memories to increase the bandwidth. They also describe a tool, called SWOOP, for implementing sliding window operations (Sliding window operation optimization).

C. Current tools to accelerate the design process on FPGAs

There are many high level synthesis (HLS) tools which aim to accelerate the design process on FPGAs. Both academic and commercial HLS tools have been developed recently. Some are for general purpose applications, such as Vivado HLS from Xilinx [5] and LegUp from University of Toronto [13]. There are also some application-specific HLS tools such as DK Design Suite for image processing [14]. Using these tools, developers can program FPGAs in a high-level language syntax and achieve potentially acceptable performance. However, even using HLS tools, developers have to be aware of how the hardware is utilised by the tool, and must optimize the code carefully to achieve acceptable performance, especially in image processing. Schmid solves the problem of memory arrangement by using a code generation tool combined with a C based library of image processing functions [7, 15]. The library covered most of the functions in OpenCV, which is important for developers from a purely software background.

In [16-18] Crookes, Benkrid et al used hardware skeletons to accelerate the design process of image processing algorithms. They also provide several hardware skeletons and use the language Prolog to describe hardware. The skeleton handles all the memory management details, while the developer supplies the function applied at each window position. Users can generate hardware within a very short time. Similarly, Fernando and Wijtvliet use sequential C code to describe hardware using hardware skeletons [19].

However, all these approaches suffer from an underlying problem that, although the developer may be using a high-level syntax, the design thinking is often still somewhat at the hardware level. Another key disadvantage of the above systems is that a modification to, or tuning of, the high level description of the algorithm generally requires re-synthesis of the FPGA architecture. This can be very time-consuming. The development of image processing algorithms is particularly experimental in nature, and involves many design iterations. Thus a simplistic approach to architecture synthesis can become frustrating, and can reduce productivity.

Now we present our novel approach for developing image processing hardware on FPGA. This is based on providing a set of Soft Co-Processors. Each co-processor implements a single, highly parameterised core IA operation, or a common compound operation. Co-processors can be linked together as in a data flow graph. There are FIFO buffers between connected co-processors, which addresses the stream synchonisation problem. The approach supports two levels of system use: (i) a 'debug' mode where the FPGA is configured with a fixed (but user-selected) mixture of generic, parameterised co-processors, rather like an FPGA having embedded functional units. Changing co-processor parameters and their interconnection does not require re-synthesis; and (ii) a 'release' mode where, once the algorithm development has stabilised, equivalent algorithm-specific co-processors can replace the more generic 'debug' co-processors by writing C functions within a chosen skeleton co-processor. This requires re-synthesis, but results in greater efficiency. This is facilitated by providing a simple macro-based code transformation tool, which allows users to define their own function and extend the AXI-Stream interconnection system without touching the hardware.

III. Soft Co-Processor Based System

In order to provide optimized memory allocation on point, neighbourhood and global operations, we provide three types of co-processor based on the core IA operations. Each co-processor instance can connect through an AXI-Stream interface. User will be able to edit the application dataflow graph in terms of IA co-processors without re-synthesis.

A. *Image Algebra-based Core Classes*

We define several types of parameterised IA co-processor, corresponding to each class of IA operation. For each image operation to be performed by the system, the user includes one of the available co-processor instances, and supplies the appropriate parameters (including the lower level functions).

- Point Operation Co-processors: these have two modes: Image-to-Scalar (I2S) and Image-to-Image (I2I). When I2S is used, the co-processor takes a streamed input image and a scalar value (a parameter), and applies the specified point function (also a parameter) at all pixel positions in the input image. When I2I is used, the hardware takes two input images, and performs the specified function on all pairs of input pixels.

The supported point operations include:

">", "<", "=", "!", "+", "-", "--", "×", "/", "and", "or", "not".

"--" performs B-A rather than A-B.

If these basic operations cannot meet the requirements of users, users can create their own point co-processors using a simple C language function.

To perform a point operation in the code, the user first acquires one of the available point co-processors. Then the parameters of the IA operation are sent to the co-processor. For example, if we want merely to threshold an image (from a defined input source channel) using a threshold value of 120, sending its output to some other channel, we could create and configure a co-processor as shown in Figure 1:

```
int main()
{
    pointOP Thresh; // A point co-processor
    Thresh.initPoint();
    Thresh.setMode("I2S");
    Thresh.setOp(2, ">", 120); // Channel 2
    Thresh.setOut(3);  // Output channel 3
}
```

Figure 1. *Code for defining a thresholding (Point) Operation*

In the code, we create and initialise a new co-processor called "PointOp1". We set its mode to "I2S" and define its function parameters to be '>' and the scalar parameter to be 120. Then we define the output image channel to which the result image will be streamed: in the above, we have merely specified some channel "3" in the AXI-Stream interconnection system.

A Point co-processor essentially consists of the set of functional units and the FIFO for storing outputs. Its size is small, but most of its area will be unused in any one operation.

- Neighbourhood Operation Co-Processor

We provide several types of neighbourhood co-processor. The most basic implements just a single operation (such as convolution) using a fixed kernel size (e.g. 3x3). Users need to define the kernel weights, and the two functions (point and reduction) which define the neighbourhood operation. The range of reduction operations include: Min, Max, Σ, |Σ| (absolute value of the final sum). For example, to obtain a convolution, the user would supply the parameters "×" and "Σ" and a vector of 9 kernel weights. To erode (or dilate) a binary input image, the first operator would be "AND", and the reduction operator would be MIN (or MAX).

We also provide more powerful neighbourhood co-processors which implement more complex, multi-function patterns. One such co-processor (NeighOp2) applies the kernel operation in several rotated orientations of the kernel, in parallel, and then combines the pixel result from each orientation using a third function. Many edge detection and morphological operations have this pattern. Rotations should be multiples of 45 degrees. This co-processor type also provides for a final point operation (it could be NOP if no extra function is required).

For example, to specify a complete Sobel edge detection operator, we apply a 3x3 window in two orientations – horizontal and vertical. We supply the kernel once (the vertical one, say) and specify two orientations, with a rotation step of 90. The two neighbourhood function parameters are "×" and "|Σ|" and the vector of kernel weights is [-1, 0, 1, -2, 0, 2, -1, 0, 1]. The operation to combine the two window outputs is '+' (combining the vertical and horizontal edge strengths). Finally, we could use the option of a final point operation to perform thresholding with a scalar value supplied as a parameter.

This more complex type of co-processor can be implemented relatively efficiently because it needs only one line buffer. The only difference between this and the previous simple neighbourhood co-processor is in the sophistication of the function which acts on the 3x3 window. We would thus code the Sobel operation, in 'debug' mode, as shown in Figure 2.

```
int main()
{
    // Define kernel, etc. …
    NeighOP2 Sobel;
    Sobel.initNeighOP();
    Sobel.setKernel(Kernel, 2, 90);
    Sobel.setOp(2, "×", "|Σ|","+", ">", 120);
    Sobel.setOut(3);  // Output channel 3
}
```

Figure 2. *Code for Sobel using Neighbourhood Operation*

- Global Operation Co-Processor

Global operations can reduce a streamed input image to either a scalar result or a vector result. We therefore have two types of global co-processor: R2S and R2V. The main reduction operations are Min, Max, Σ, |Σ|, Count and Average. These can be applied to give either a scalar or vector result. An image histogram can be obtained by setting mode R2V, and specifying Count as the function. The results of the global operation will be stored in a fixed address (in BRAM); users can get the result by accessing that address directly.

The advantage of this 'debug' mode is that the FPGA can be pre-configured with a user-defined selection of multiple copies of each type of co-processor, and provided these resources are sufficient, any system can be run without any re-synthesis.

The disadvantage, though, is that each co-processor must have hardware for a wide range of functions, and most of these will not be utilised. The penalty for this 'debug' mode is thus primarily in requiring extra area on the chip, and some speed penalty compared to a custom-designed core (see section V).

B. User-defined Co-Processors and Code Transformation

This section describes the mechanism which our software environment provides for producing an optimised 'release' version of a co-processor (and which we have used to develop the more generic IA-based co-processors above).

A particular image processing application may use a combination of IA operations which in theory could be performed by a single, complex, co-processor, but for which we have no single co-processor available in our library. When developing in 'debug' mode, we typically would use several co-processors to implement the more complex function. For a more efficient implementation, we enable the user to easily create a new, customised image co-processor by merely providing the C code for the inner operation itself, with all the housekeeping provided by pre-prepared skeleton co-processors. This approach might also be used if we wish to produce an optimised version of a generic co-processor (such as for implementing Sobel). The motivation for this would be to optimise the area, and to a lesser extent, speed.

To assist this, we have developed a simple code transformation tool, so that users can embed their pixel-level function in a pre-prepared skeleton co-processor, and generate a new co-processor which is compatible with the rest of the system including the AXI-Stream interconnection system.

Unlike the code generator in [7], we exploit the potential of macros and the C/C++ preprocessor, instead of writing our own transformation system. We provide a library of macros which act like simple programming language extensions. We actually provide different implementations of the same set of macros; this enables the user to run the program either on a PC or on an FPGA (and potentially on a GPU platform) simply by including the appropriate macro definition files. For example, if the user includes "Macros_PC.h", then the code will be suitable for a PC and after the macro expansion, all of the code will appear as normal C/C++. However, if the file "Macros_Virtex7.h" is included, after the macro expansion, the code will appear as HLS C/C++ code, with synchronization and pragmas in the code. Users can, if they wish, debug and verify their code on a PC, and then synthesise and export their code without change. This also enables multiple FPGA families to be made available.

C. Design Flow of our System

In 'debug' mode, which is shown in the left side of figure 3, the user initially defines how many of each type of co-processor they want to have available. These are then synthesised (if necessary), and system development can proceed as long as the number of each type of co-processor is not exceeded. Once the final system is stable, users can define only the precise number of co-processors required, and move to 'release' mode for further optimisation.

IV. Case Study on Creating a New Co-Processor

In this section, we illustrate the production of an optimised co-processor (a 'release' version) specifically for convolution. We then give results for the two versions (debug and release) in terms of both speed and area, to demonstrate the trade-offs when using debug mode as opposed to using release mode.

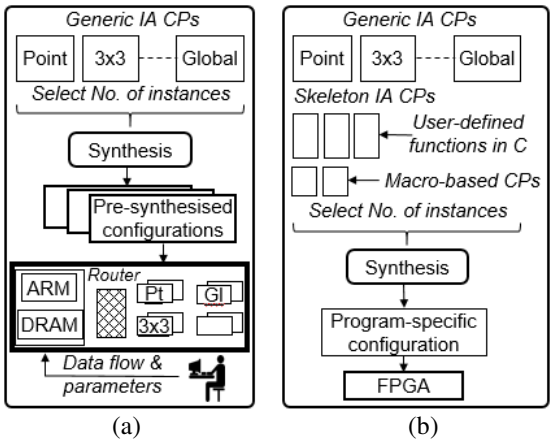

Figure 3. *Design Flow for the System, for (a) Debug mode and (b) initial extension for release mode*

The key macros included in the macro library are for:

- creating objects, such as line buffers and array objects (`_NEW_LINE_BUFFER, _NEW_2D_ARRAY`)
- starting and ending the neighbourhood processing of a streamed image (`_STREAM2D_N_FOR_START, _STREAM2D_N_FOR_END`)
- Read a pixel from the line buffer (`_GET_LINEBUFFER`)

`_STREAM2D_N_FOR_START` acts like the start of a double (2D) for loop. It initially fills the line buffer from the supplied input stream, and sets the coordinates of the first complete window position. (The coordinate variables idxRow and idxCol are actually the coordinates of the bottom right hand corner of the window – i.e. the most recently input pixel).

`_STREAM2D_N_FOR_END` acts like the end of the for loop. It normally outputs the supplied output value and stores the next input pixel in the line buffer. It will also handle border pixels, with options including zero fill and extension by reflection. If however, it has reached the end of the input stream, it sends the appropriate signal to the output stream and ends the loop.

Figure 4 shows the code we use for any neighbourhood operation. The macros all begin with '_'. The code can handle an input image of any size up to a defined maximum size, _IMAGE_X_MAX by _IMAGE_MAX_Y. The cost of this is that the width of the line buffer will be the maximum image width, even if this is not all used. But the advantage is that we can process other image sizes without re-synthesising the co-processor. The window size is defined by two constants (KERNEL_X and KERNEL_Y). Changing these will require re-synthesis.

The heart of the co-processor code is the 2D for loop. Inside this, the first step is to extract a 3x3 region of the line buffer into a separate 3x3 array (window). In some cases, this may be a bit of an overhead; but is it more efficient if any pixel is accessed more than once because it uses registers rather than BRAMs.

The actual function which is to be applied at each window position (`window_function`) is not shown here. This function is written in pure C code. Our 'debug' IA co-processors are implemented simply by writing more generic functions which take the various function operands as parameters. To illustrate the process, Figure 5 shows the code for a basic convolution function (output is cropped to 0..255).

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "Macros_HLS.h"

#define IMAGE_X_MAX     2048
#define IMAGE_Y_MAX     2048
#define KERNEL_X        3
#define KERNEL_Y        3
#define BORDER_VAL      0 // Border handling strategy
#define IMAGE_BIT       8 // Bits per pixel
#define PIXEL_SIZE      8

_COPROCESSOR(Kernel_skeleton,inStream,outStream,
             IMAGE_X,IMAGE_Y,
              char Kernel[KERNEL_X][KERNEL_Y])
{
  _NEW_LINE_BUFFER(linebuff,_PIXEL_TYPE(PIXEL_SIZE),
                   KERNEL_Y,IMAGE_Y_MAX);
  _NEW_2D_ARRAY(window,_INT(10),KERNEL_X,KERNEL_Y);
  _PIXEL_TYPE(PIXEL_SIZE) pixIn;
  _PIXEL_TYPE(PIXEL_SIZE) valOutput;
  _PIXEL_TYPE(PIXEL_SIZE) val;

  _STREAM2D_N_FOR_START(idxRow, idxCol, IMAGE_X,
                        IMAGE_Y,instream, linebuff)
#pragma HLS PIPELINE
    // Window centre is [idXRow-1, idxCol-1]
    for (int i=0;i<KERNEL_X;i++)
    {
      for (int j=0; j<KERNEL_Y; j++)
      {
        val = _GET_LINE_BUFFER
               (linebuff,i,(j+idxCol-(KERNEL_Y-1)));
        window[i][j] = val;
      }
    }
    valOutput = window_function(window, Kernel);
  _STREAM2D_N_FOR_END(idxRow, idxCol, BORDER_VAL,
                      outStream, valOutput)
}
```

Figure 4. *Code for a neighbourhood co-processor skeleton*

```
_PIXEL_TYPE(PIXEL_SIZE) window_function(
 _PIXEL_TYPE(PIXEL_SIZE) window[KERNEL_X][KERNEL_Y],
    char Kernel[KERNEL_X][KERNEL_Y])
{
  // Implements a basic convolution
  _PIXEL_TYPE(PIXEL_SIZE) valOutput;
  _INT(16) temp = 0;
  for (int i=0;i<KERNEL_X;i++)
    for (int j=0; j<KERNEL_Y;j++)
#pragma HLS PIPELINE
      temp += window[i][j] * Kernel[i][j];
  if (temp>=255) return 255;
  if (temp<0)    return 0;
  return temp;
}
```

Figure 5. *Co-processor code for a convolution function*

The advantage of this co-processor over the more generic IA (debug) version is of course that it is more area efficient, since it does not need the logic for the wide range of point and reduction operations which the generic version has. An additional optimisation when the kernel weights are known would be to unroll the inner reduction loop, using the known weights in the calculation.

V. RESULTS AND COMPARISON

Advances in the performance of tools such as Vivado HLS mean that the code in which we have written the (debug) versions of our IA co-processors is much more acceptable than in the past. Therefore, rather than comparing our architectures with VHDL-coded versions, we compare different versions of the two modes (debug and release).

Our first performance test in this section is based on a simple comparison between the performance of a 3x3 convolution implemented firstly using our generic IA neighbourhood co-processor, and the customised version using the code shown in Figure 4 and Figure 5 (see Table 1).

256 × 256	LUTs	FFs	DSPs	BRAMs	Clk cycles	FPS
Debug	1059	853	0	4	197,284	506
Release	801	638	0	3	131,111	768
[21]	2648	3652	0	97.5	<1,666,800	> 40
512 × 512	LUTs	FFs	DSPs	BRAMs	Clk cycles	FPS
Debug	1404	1104	0	5	786,524	128
Release	700	913	0	3	521,293	192
[15]	14241	10950	45	2	265308	>700

Table 1. *Resources for processing 256×256 and 512×512 images, in debug and release mode, 100MHz ([15] 214.4MHz, [21] 150MHz)*

It can be seen that the optimised (release) version saved about 25% hardware resources over the very generic (debug) implementation, and it performs about 50% faster. Although these savings are significant in a final system, the overhead of using a very flexible, generic system, which does not require repeated re-synthesis, suggests that our proposed approach is a useful compromise between convenience and efficiency.

Compared with similar work in the literature, our design keeps both programmability and performance. Compared with [21] (where clock cycles are calculated from their FPS), our approach is slightly faster using the same platform and similar architecture. Since [21] uses a colour image and does grayscale conversion before the multi-convolution, their performance will be similar to ours if doing the same task. In [15] (FPS is calculated from their clock cycles), with a library-based implementation of convolution for a 512×512 image, they achieved performance of one clock cycle/pixel, with a clock speed of 214.4 MHz, which equates to over 700FPS. However, it uses 20 times the resources to run their hardware at this clock frequency just for doing convolution.

Our second comparison is based on a novel implementation of the Lloyds K-Means Clustering algorithm (publication in preparation). We first developed an efficient hardware-based implementation in HLS. Then we used our higher level macro-based notation described in section IV to develop an equivalent co-processor. In Table 2, we compare the hardware usage of the two designs. Here, we see that the resources for our high level, macro-based coding version is only about 1.2 times that of the hand-coded HLS implementation. The increase is partly because the macro-based version introduces some variables, which are never actually used (such as the coordinates of the current image pixel).

	LUTs	FFs	DSPs	BRAMs
Hand-coded HLS	936	784	4	2
Macro-based	1101	964	4	2

Table 2. *Resources for K-Means clustering, comparing manual hardware design vs. our higher level Macro coding*

In terms of performance, the high-level implementation of K-means takes 0.0045s (222 FPS) to cluster a 256×256 image (with 50 iterations) and output the cluster values. The low-level hardware design using HLS takes 0.0036s (277.8 FPS) to cluster the same image.

From both these tests, we can conclude that:

- The overhead in using the generic co-processors is not very significant during the development cycle (~25-30%). The benefits of not having to re-synthesis during algorithm and system experimentation outweigh the resource and performance overheads.
- The high level, macro-based approach often enables more efficient co-processors to be obtained with a small amount of algorithmic C programming (rather than using C syntax while still thinking in terms of hardware), without significant hardware expertise.
- Our high level macro-based hardware programming tool has also proved beneficial in designing co-processors for important image processing algorithms (such as Kmeans clustering) which are not so easily expressible in IA.

VI. Conclusions

In this paper, we have presented an approach to developing image processing applications based on the concept of Soft Co-Processors. The co-processors are based on implementing image-level operations using an IA-like notation. The complete system comprises a set of interconnected co-processors which reflects the structure of the corresponding data flow graph.

The paper presents two modes of application development: using a set of generic co-processors whose functionality is parameterised, and which can therefore be re-purposed without the need to re-synthesis the whole system. This is designed to speed up the development cycle on FPGAs, which enables algorithm and application development to take place on the FPGA itself, with a performance reduction and resource overhead of typically around only 25%-30%. For development purposes, this is a very significant benefit.

Further, we have demonstrated that, when creating function-specific co-processors, we can raise the programming level considerably without the need to develop and maintain sophisticated software tools, by exploiting the macro facilities of the C/C++ pre-processor. Again, while there is a cost (~25-30%) of this higher level approach over hand-optimised hardware design, this trade-off is a small price to pay for the more rapid design cycle, and the reduction in hardware awareness needed of the developer. By merely using different macro definition files, users can re-target their program for a different FPGA family or different hardware platform.

VII. Acknowledgement

This work was supported by the Chinese Scholarship Council.

References

[1] Ciresan D C, Meier U, Masci J, et al. Flexible, high performance convolutional neural networks for image classification, Proc. IJCAI International Joint Conference on Artificial Intelligence. 2011, 22(1): 1237.

[2] Krizhevsky A, Sutskever I, Hinton G E. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems. 2012: 1097-1105.

[3] Ovtcharov K, Ruwase O, Kim J Y, et al. Accelerating deep convolutional neural networks using specialized hardware. Microsoft Research Whitepaper, 2015, 2(11).

[4] Wilson J N, Ritter G X. Handbook of computer vision algorithms in Image Algebra. CRC press, 2000.

[5] Crockett L H, Elliot R A, Enderwitz M A, et al. The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000. All Programmable Soc. Strathclyde Academic Media, 2014.

[6] Bailey D G. Design for embedded image processing on FPGAs. John Wiley & Sons, 2011.

[7] Reiche O, Schmid M, Hannig F, et al. Code generation from a domain-specific language for C-based HLS of hardware accelerators. Proc IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2014. pp 1-10.

[8] Johnston C T, Gribbon K T, Bailey D G. Implementing image processing algorithms on FPGAs. Proc. 11th Electronics New Zealand Conference, ENZCon'04, 2004. pp. 118-123.

[9] Shokr M E. Evaluation of second-order texture parameters for sea ice classification from radar images. Journal of Geophysical Research: Oceans, 1991, 96(C6): 10625-10640.

[10] Benedetti A, Prati A, Scarabottolo N. Image convolution on FPGAs: the implementation of a multi-FPGA FIFO structure. Proc. Euromicro Conference, 1998. Proceedings. 24th. IEEE, 1998, 1: 123-130.

[11] Monson J, Wirthlin M, Hutchings B L. Optimization techniques for a high level synthesis implementation of the Sobel filter. Proc. International Conference on Reconfigurable Computing and FPGAs (ReConFig), IEEE, 2013. pp. 1-6.

[12] Yu H, Leeser M. Automatic sliding window operation optimization for FPGA-based computing boards. Proc. 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2006. FCCM'06. pp. 76-88.

[13] Canis A, Choi J, Aldham M, et al. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. Proc. 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays. ACM, 2011. pp. 33-36.

[14] DK design suite: Handel-C to FPGA for algorithm design. 2010.

[15] Schmid M, Apelt N, Hannig F, et al. An image processing library for C-based high-level synthesis. Proc. 24th International Conference on Field Programmable Logic and Applications (FPL), 2014. IEEE, 2014: 1-4.

[16] Crookes D, Alotaibi K, Bouridane A et al. An environment for generating FPGA architectures for image algebra-based algorithms. Proc. International Conference on Image Processing (ICIP 98). 1998: 990-994.

[17] Benkrid, K, Crookes, D, Smith, J, & Benkrid, A. High-level programming for FPGA based image and video processing using hardware skeletons. Proc. 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2001. (FCCM'01). pp. 219-226.

[18] Benkrid K, Crookes D. From application descriptions to hardware in seconds: a logic-based approach to bridging the gap. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2004, 12(4): 420-436.

[19] Fernando S, Wijtvliet M, Nugteren C, et al. Accelerator synthesis using algorithmic skeletons for rapid design space exploration. Proc. 2015 Design, Automation & Test in Europe Conference & Exhibition. EDA Consortium, 2015. pp. 305-308.

[20] Nugteren C, Corporaal H, Mesman B. Skeleton-based automatic parallelization of image processing algorithms for GPUs. Proc. IEEE International Conference on Embedded Computer Systems (SAMOS), 2011. pp. 25-32.

[21] Altuncu M A, Guven T, Becerikli Y, et al. Real-time system implementation for image processing with hardware/software co-design on the Xilinx Zynq platform[J]. International Journal of Information and Electronics Engineering, 2015, 5(6): 473.