High Level Programming for Real Time FPGA Based Video Processing

K Benkrid*, D Crookes*, J Smith* and A Benkrid*

*School of Computer Science, The Queen’s University of Belfast, Belfast BT7 1NN, UK
*VisiCom Laboratories, 10052 Mesa Ridge Court San Diego, CA, 92121 USA

ABSTRACT

The inherent reprogrammability of Field Programmable Gate Arrays (FPGAs) gives them some of the flexibility of software while keeping the performance advantages of an application specific hardware solution. However, the main disadvantage of FPGAs is the low level of their programming model. Although software tools have been drastically improved since the early days of this new technology, they still require the user to think at the hardware level rather than at the algorithmic level.

To bridge the gap between the application and implementation levels, we present a high level software environment for FPGA-based real time video processing, which aims to hide hardware details completely from the user. Our approach is to provide a flexible FPGA-based Image Processing Coprocessor with a very high level programming interface based on the core operators of Image Algebra.

Our system has been successfully implemented on VISICOM’s Vigravision™ PCI board giving real time processing of video data.

1. INTRODUCTION

FPGA technology has been proposed as a way of obtaining high performance for computationally intensive DSP applications such as Image Processing (IP) at a relatively low cost, even under real time requirements [1]. Although this solution seems to enjoy the advantages of both a dedicated hardware solution and a software based one, many are still reluctant to move toward this new technology because of the low level programming model offered by FPGAs. Even with libraries of common functions like adders and multipliers, the user still has to think at the logic level rather than at the algorithmic level.

In this paper, we present a high level software environment for FPGA-based image processing, which hides hardware details as much as possible from the application developer. The high level programming model is based on providing a range of image level operations, including point operations and a set of neighbourhood operations based on the core operators of Image Algebra [2]. Our software tools can generate efficient FPGA configurations from very high level descriptions. Although our approach is not tied to a particular type of FPGA, our current implementation of the IP Coprocessor (IPC) is based on Xilinx XC4000 FPGAs.

This paper first presents the application-oriented programming model. Section 3 then gives details of the hardware architectures used to implement the high level abstractions. The overall software environment is then outlined in Section 4. Finally, some conclusions are drawn and future research directions indicated.

2. THE PROGRAMMING MODEL

Common image processing operations can be classified in terms of the extent of their data access requirements into three categories: point operations (such as image subtraction), neighbourhood operations (such as convolution), and global image operations (such as histogramming). For the purposes of this paper, we will concentrate here on neighbourhood operations. Many neighbourhood operations can be described by a template (static windows with user defined weights) and a set of Image Algebra operators [2]. Indeed, most common neighbourhood operations can be split in two stages:

- A ‘local’ operator applied between an image pixel and the corresponding window coefficient.
- A ‘global’ operator applied to the set of local operation results to generate a result image pixel.

The set of local operators includes ‘Add’ (+) and ‘multiplication’ (*), whereas the global operators include ‘Accumulation’ (Σ), ‘Maximum’ (Max) and ‘Minimum’ (Min) operators. With these local and global operators, the following neighbourhood operations can be built:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution</td>
<td>*</td>
<td>Σ</td>
</tr>
<tr>
<td>Additive maximum</td>
<td>+</td>
<td>Max</td>
</tr>
<tr>
<td>Additive minimum</td>
<td>+</td>
<td>Min</td>
</tr>
<tr>
<td>Multiplicative maximum</td>
<td>*</td>
<td>Max</td>
</tr>
<tr>
<td>Multiplicative minimum</td>
<td>*</td>
<td>Min</td>
</tr>
</tbody>
</table>

For instance, a simple Laplace operation would be performed by doing convolution (i.e. Local Op. = * and Global Op. = Σ) with the following template:

\[
\begin{array}{ccc}
\sim & -1 & \sim \\
-1 & 4 & -1 \\
\sim & -1 & \sim
\end{array}
\]

In practical image processing applications, many algorithms comprise more than a single operation. Such complex operations can be broken into a number of primitive simple instructions. An example of such complex operations is Sobel edge detection algorithm, which can be approximated by adding the absolute results of two separate convolutions as shown below:

\[
\text{Sobel-horiz} = \begin{bmatrix}
1 & -1 & -1 \\
2 & -2 & -2 \\
1 & -1 & -1
\end{bmatrix}
\]

\[
\text{Sobel-ver} = \begin{bmatrix}
1 & 2 & 1 \\
-1 & -2 & -1
\end{bmatrix}
\]

\[
\text{Sobel} = \text{Convolution} + \text{Convolution}
\]
3. HARDWARE ARCHITECTURES

As mentioned previously, any neighbourhood operation involves passing a 2-D window over an image, and carrying out a calculation at each window position. To allow each pixel to be supplied only once to the FPGA, line delays are required. These line delays synchronise the supply of input values to the processing elements, ensuring that all the pixel values involved in a particular neighbourhood operation are processed at the same instant [3]. Figure 1 shows the architecture of a generic 2-D neighbourhood operation with an N by M window template where ‘L’ is the local operation and ‘G’ is the global one.

Before describing the architectures for the local and global operators, we first have to address the following key question: should we use bit serial or bit parallel arithmetic?

3.1 Serial vs. parallel arithmetic

While parallel designs process all data bits of an operand simultaneously, bit serial designs process input data one bit at a time. For an n-bit word, the required hardware for a parallel implementation is typically ‘n’ times that of the equivalent serial implementation. On the other hand, the bit serial approach requires ‘n’ clock cycles to process an n-bit word while the equivalent parallel one needs only one clock cycle. In bit serial implementation, there is a need to generate a bit clock from the pixel clock. The bit clock frequency is ‘n’ times the pixels clock one (for an ‘n’ bit pixel). For practical real time video processing, pixel sampling rate frequency must be at least 13 MHz. This implies a bit clock frequency of not less than 100 MHz for 8 bit pixels, and 200 MHz for 16-bit pixels. This range of frequencies is very hard, if not impossible, to achieve, even on the fastest FPGA chips. This has led us to choose parallel arithmetic rather than bit serial, for video processing.

Note that a trade-off in the form of digit serial arithmetic is still possible. However, this implies additional hardware for the digit clock frequency generation, and extra care for data synchronisation. A parallel implementation is easier to implement and can be efficiently implemented using dedicated fast carry logic as described below.

3.2 Basic component architectures

The basic functions required for nearly any signal processing operation include addition/subtraction, shifts and delays. These blocks can then be used to construct the more complicated structures such as multipliers, dividers and maximum/minimum selectors.

- **Parallel adder/subtractor**
  Parallel adders/subtractors are efficiently implemented using dedicated carry logic on XC4000 FPGAs. Each Complex Logic Block (CLB) can implement a two-bit adder slice. An ‘N’ bit adder is implemented in N/2+1 CLBs.

- **Parallel Maximum/minimum selectors**
  Here again, high speed carry logic is used to implement an efficient min/max function. The carry logic in each CLB is set for an A-B subtract function, while the function generators are used to implement a 2:1 mux. The mux is controlled by the carry out of the whole subtraction (CYx) as shown in the following figure.

![Figure 2. A 2-bit slice of a Maximum/Minimum selector](image)

An ‘N’ bit Maximum/Minimum unit occupies N/2+1 CLBs.

- **Parallel constant multiplication**
  Multiplication involves two basic operations: the generation of the partial products and their accumulation. In order to speed up the multiplication process, either the number of the partial products should be reduced, or the accumulation process should be speeded up. Note that the partial products corresponding to ‘0’ bits in the multiplier input are zero, and therefore do not have to be included in the sum. If the number of ‘1’ bits in a constant coefficient multiplier is small, then the multiplication may be realised with shifts and few adders. Moreover, in the case where the multiplier contains a string of consecutive 1’s, a proper recoding of the multiplier can reduce the number of partial products. Consider the example of a multiplication by 15. Note that 15=1111 can be expressed as 16=10001 in Signed Digit notation where \( \bar{1} = -1 \). Thus the multiplication is reduced to a simple subtraction (with input shift). The recoding scheme reduces the number of partial products by a half on average. In common neighbourhood operations, window coefficients are often...
sparse (contain many 0’s) and thus yield to a considerable reduction in hardware complexity. For instance, no multiplication is needed in a Laplacian filter.

Note that the minimal number of partial products required for a given multiplier is equivalent to the problem of finding the smallest number of non-zero digits in a Signed Digit (SD) representation of the multiplier. The algorithm for obtaining the minimal representation of a SD number is called canonical recoding [4].

- **Parallel constant division**

For the division operation, the divisor \( D \) is approximated to a fraction \( \frac{X}{Y} \), where \( Y \) is a power of two. The division is then reduced to a multiplication followed by a right shift. The above multiplication method is used to efficiently multiply by \( X \). For instance, consider the case of the *Averaging* operation given by the following window mask:

\[
\begin{array}{ccc}
1/9 & 1/9 & 1/9 \\
1/9 & 1/9 & 1/9 \\
1/9 & 1/9 & 1/9 \\
\end{array}
\]

The division by 9 can be approximated by multiplication by \( \frac{7}{64} = 2^6 \), which is a multiplication by 7 followed by 6 right shifts. Using the canonical recoding algorithm, the multiplication by \( 7 = 2^3 - 1 \) is performed using only one subtraction block. Therefore, a division by 9 consumes only one subtraction unit.

In general, divisors are optimised offline and stored in a look-up table.

### 3.3 Hardware implementation

Our implementation of the IPC is based on VISICOM’s VigraVision™ PCI board [5]. The functional block diagram of the VigraVision board is given by figure 3.

Due to the limited memory resources on the FPGA chip, the line delays have been implemented using the offchip DRAM. Part of the FPGA is configured as an interface to the onboard DRAM (FIFOs), while the other part is configured to perform the required image processing operation as shown below:

**Figure 3. Block diagram of VigraVision video board**

4. **DESIGN OF THE SOFTWARE ENVIRONMENT**

The goal of our environment is to provide the user with the ability to dynamically create and use, at a very high level, FPGA configurations for a wide range of IP operations. This should enable the user to specify only the IP algorithm to be carried out, and obtain automatically an efficient FPGA configuration dynamically. This is done in two phases: first, the programmer defines the required IP operation and creates its FPGA configuration. Subsequently, he or she applies this configuration to an actual input video stream.

#### 4.1 Defining new operations

The basic instruction set provided includes a general neighbourhood operation constructor (plus point to point image instructions such as \( \text{abs}, >, +, \cdot, /, \ast, \) etc.) For instance, to describe any general neighbourhood operation, the following constructor is used:

\[
\text{neighbourhood}(\text{local}, \text{global}, \text{Window coefficients}).
\]

For example, a Laplace edge detection operation is described as follows:

\[
\text{neighbourhood}(\text{mult}, \text{accum}, \{\sim,-1,\sim\},
[-1,4,-1],
[\sim,0,\sim]/9).
\]

A Blur operation is described as follows:

\[
\text{neighbourhood}(\text{mult}, \text{accum}, \{1,1,1\},
[1,1,1],
[1,1,1])/9.
\]

Similarly, an erosion operation can be described as:

\[
\text{neighbourhood}(\text{add}, \text{min}, \{\sim,0,\sim\},
[0,0,0],
\sim,0,\sim]/9).
\]

In a similar way, compound operations can be described using point to point operators in addition to the *neighbourhood* constructor. For instance, a Sobel edge detection operation is simply described as follows:
Using the above instruction set, the user can describe a multitude of simple and compound neighbourhood operations.

4.2 Applying an operator

The user interfaces to the VigraVision board through a C-callable library (VigraVision ToolBox DLL). The ToolBox includes hardware initialisation and register control functions, image acquisition functions and image processing functions. For instance, the user downloads a particular FPGA configuration by invoking the following function:

\[
\text{u_loadXilinx}(\text{Xilinx Chip ID, Configuration filename})
\]

The user can copy the processed image from the video buffer to the host processor, for further analysis, by:

\[
\text{u_readRect}(\text{LPRECT lpImgRect, LPVOID imgData})
\]

where \(lpImgRect\) specifies the rectangle in the frame buffer where data is to be read from, and \(imgData\) is a pointer to the image transferred to the host memory.

4.3 The Complete Environment

The complete system is given in Figure 5. For internal working purposes, we have developed our own intermediate high level hardware description notation called HIDE [6]. This is Prolog-based [7], and enables highly scalable and parameterised component descriptions to be written.

![Figure 5. Overview of the complete system](image)

As indicated previously, the user programs in C++, or can interact with a graphical user interface specifying the IP algorithm to be carried out on the FPGA (Local and Global operators, window template coefficients etc.). For the time being, we rely on Xilinx Placement And Routing (PAR) tools to place and route the circuit. This may take a considerable amount of time (~1 hr), but the approach has proved sufficient to satisfy the real time timing constraints for the whole range of operations provided to date.

Behind the scenes, when the user gives all the parameters needed for the specific IP operation, a circuit description in the HIDE notation is generated. This description will then go through the HIDE2EDIF tool to generate the FPGA configuration EDIF netlist. Lastly, the configuration netlist will go through the FPGA vendor’s specific tools to generate the configuration bitstream file. The whole process is invisible to the user, thus making the FPGA completely hidden from the user’s point of view.

The whole range of operations described above and many more have been generated using the high level environment described above. The complete system is fully operational on the VigraVision board, and enables rapid and convenient application development for video processing.

5. CONCLUSION

In this paper, we have presented the design of an FPGA-based video coprocessor, along with its high level programming environment. The coprocessor instruction set, although extensible, is based on a core level containing the operations of Image Algebra. Possibly the most significant aspect of this work is that it opens the way to image processing application developers to exploit the high performance capability of a direct hardware solution, while programming in an application-oriented model.

Future directions include upgrading the system to handle other FPGA series (particularly Xilinx Virtex chips), the investigation of a digit serial implementation, adding a fully automatic placement utility to our system in order not to rely on the Xilinx PAR tools, and the extension of the instruction set to handle more sophisticated image processing operations (FFT, DCT, wavelet transform etc.)

6. REFERENCES


