Accelerating Adobe Photoshop with Reconfigurable Logic

Satnam Singh
Xilinx Inc.
San Jose, California, U.S.A.
Satnam.Singh@xilinx.com

Robert Slous
Xilinx Inc.
San Jose, California, U.S.A.
Robert.Slous@xilinx.com

Abstract

This paper presents the results of a project designed to produce a commercial application for reconfigurable logic. We describe how we took the popular image processing application Adobe Photoshop and used its plug-in technology to devise a set of FPGA-based filters to accelerate colour space conversion and image convolution operations. Some of the barriers that make it difficult to produce portable FPGA-based filters are explored.

1 Introduction

In the recent IEEE Computer article *Seeking Solutions in Configurable Computing* [5] it was reported that 'no companies are known to use reconfigurable computing for a competitive advantage.' This paper describes one of several projects by the authors to develop real-world applications for FPGA technology which address this problem. In particular, we demonstrate how we designed and implemented several FPGA-based plug-ins to accelerate the popular Adobe Photoshop application which is widely used for image processing.

This paper gives an overview of the type of image processing operations one can perform with Adobe Photoshop and then goes on to describe various filtering operations that are suitable for hardware acceleration. We describe the state of the art in software and hardware based Photoshop filters, and then review what reconfigurable computing options are available for speeding up Photoshop. The design and implementation of several filters using a Xilinx XC6200 FPGA on a PCI card is presented. We describe the procedure for developing the hardware and present details of how the plug-ins were developed and integrated into Photoshop.

The initial performance results for the FPGA-based filters are very encouraging. The work presented here could form the basis of a commercial system for accelerating Adobe Photoshop using a general purpose FPGA-card (sold by multiple vendors). This could be a high profile reconfigurable application that addresses the concerns of the authors of *Seeking Solutions in Configurable Computing*.

However, based on experience with using the Photoshop filters on various platforms, we describe some of the software and standards infrastructures that must be in place before composite hardware-software entities can be sold as commodity items that run one systems sourced by multiple vendors.

2 Adobe Photoshop

Adobe Photoshop is a widely used image processing package which provides a modular architecture for extending its functionality based on plug-ins. Images to be processed are often in true-colour (24-bits) and may be sampled from a photograph at a high resolution. Photoshop provides filters that can manipulate an image in various ways including colour manipulation and filtering (e.g. Gaussian blur). For large images these filters can take a long time to run, and there is already a market for specialised DSP-based cards which can be used with plug-ins to accelerate Photoshop.

By using an off-the-shelf system like the VCC XC6200 FPGA system, one can produce filters that are realised on an FPGA. One can distribute image processing circuits as plug-ins, making them a commodity item that is conveniently packaged. If high speed filters can be produced then there may be a market for the VCC boards in the desktop publishing niche.

Related to this project is the PostScript/FPGA project which is using the VCC XC6200 cards to accelerate PostScript rendering. This is also targeted at desktop publishing. A studio using a VCC card for image processing and PostScript rendering provides a compelling example of dynamic reconfiguration. Reconfiguration occurs at the application level (switching between PhotoShop support and PostScript) and within an application (switching between different filters or rendering algorithms).

The principal reason for us to produce an Adobe Photoshop plug-in is to provide an interesting example of dynamic reconfiguration which may evolve into a real product.

3 Performance Limits

Before considering which filters would be suitable for realisation on the XC6200 PCI cards, it is instructive to calculate the upper bounds for the performance of a system based on the VCC XC6200 FPGA cards.

The maximum on-board SRAM is 8MB (23-bits of address space). Allowing 3 bytes per pixel this can represent about 2.7 million pixels or an area of 1672 pixels square. For a 600 dpi target this represents an area which is 3 square inches. For filters that can mutate the original image the area is halved. Unfortunately 8MB of SRAM is probably not enough for realistic image sizes, but it was enough to allow us to perform experiments to establish feasibility of this approach. 64MB of RAM would allow more realistically sized images to be manipulated.

For comparison, an existing correlator implemented on the XC6200 [9] uses a 32x16 mask which can be used to correlate a 512x512 image in about 30ms (about 8.7 mega-samples per second). Reconfiguration occurs in a few milliseconds (or less).

4 Image Processing Examples

This section describes some of the problems associated with digitally scanned images and presents the notion of filtering for smoothing images.

4.1 Anti-Aliasing

Digitised photographs exhibit aliasing which occurs because a continuous light signal has been mapped onto a discrete signal of possibly limited colour depth (i.e. a 2D array of pixels). By suitably sampling the continuous source signal the original image can be reconstructed from the finite data. However, sampling an infinite amount of data down to a finite amount of data will loose information and cause artifacts on the reconstructed image which are not present on the source image.

One such problem is aliasing which manifests itself as jaggy lines and sharp edges where rapid frequency variations in the source signal could not be captured in the discrete signal. Anti-aliasing is the process of removing these artifacts from the sampled image. For example, consider the triangle in Figure 1 (a) which has been sampled as shown in part (b). The jaggy edge can be smoothed by used different shades of grey to reflect how much of a pixel is occupied by the source signal, as shown in part (c).

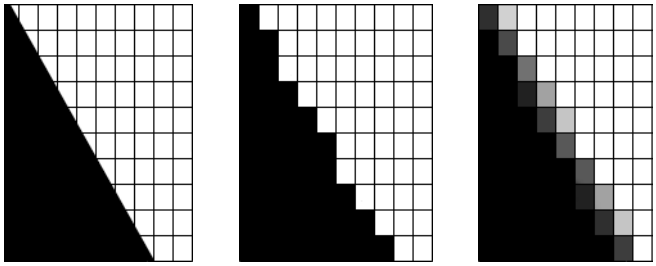


Figure 1 Anti-Aliasing Graphics

On colour output devices anti-aliasing is performed by using different shades of black to give a smooth outline for fonts, as shown in Figure 2.

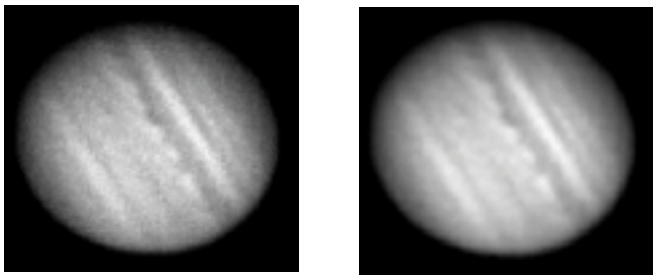
Figure 2 Anti-Aliasing text

4.2 Image Filtering by Convolution

In order to capture faithfully a source signal of frequency *f*, it must be sampled at a minimum of frequency 2*f* (called the *Nyquist* frequency). If the number of high frequency signals can be reduced, then fewer samples need to be taken of the source image, but the sampled image will be of a poorer quality. This is a common compromise which is known as *bandwidth limiting* or *band limiting* the signal. This technique is implemented with low-pass filters which inhibit high frequency signals. The perfect filtering function *S* of some signal *u* which is to be constrained in the band *-k* to *k* is defined as:

$$S(u) = \begin{cases} 1, when \;\; -k \le u \le k \\ 0, elsewhere \end{cases}$$

Figure Figure shows how low-pass filtering of a noisy CCD image of Jupiter improves the visibility of fine details at the cost of a blurrier image.



(a) (b)

Figure 3 Low-Pass Filtering of Jupiter

In principal to perform low-pass filtering the image signal has to be transformed from the spatial domain into the frequency domain, multiplied by a suitable pulse function (to mask off the unwanted frequencies) and then transformed back into the spatial domain. The computational complexity of this task can be reduced by noting that multiplying two Fourier transforms in the frequency domain corresponds to *convolution* on their inverse Fourier transforms in the spatial domain.

The convolution of two signals $f(x)$ and $g(x)$ is written as $f(x) * g(x)$ and represents a weighted average of the neighbourhood around each point of the signal $f(x)$. The weights are specified by the signal $g(x)$ and the neighbourhood is the domain over which the filter is nonzero (called the *support* of the filter). A filter that is nonzero over a finite domain is said to have *finite support*. The filtered signal $h(x)$ is specified by:

$$h(x) = f(x) \times h(x) = \int_{-\infty}^{\infty} f(\tau) g(x - \tau) d\tau$$

The filter function is called the *convolution kernel* or *filter kernel*.

Implementing convolution involves repeated multiplications and additions which makes this algorithm suitable for realisation on FPGAs. The implementation can also be pipelined, giving further performance gains. Furthermore, convolution in software of RGB colour images requires independent convolution of the red, green and blue channels.

A common filter kernel is Gaussian Blur, which is a standard filter in Photoshop. An example of a Gaussian 7x7 filter is:

1	4	8	10	8	4	1
4	12	25	29	25	12	4
8	25	49	58	49	25	8
10	29	58	67	58	29	10
8	25	49	58	49	25	8
4	12	25	29	25	12	4
1	4	8	10	8	4	1

This filter places a lot of weight around the pixels near the centre of the mask, but tapers off sharply

Using a square filter of size n pixels on a square image of size m pixels requires n^2 multiplications and additions. Each RGB channel is at most 8 bits, which is also a reasonable upper bound for the filter component values.

5 Software and DSP Filters

Most Adobe Photoshop filters are implemented in software, although DSP based Photoshop accelerator cards are also available. A DayStar Gensis MP600 workstation with four 150MHz PowerPC 604 processor can perform a Gaussian Blur in about a third of the time taken by a PowerMac 9500 and about roughly the same speed as a PCI PowerShop DSP card designed to accelerate PostScript. The DayStar system used specially written plug-ins that exploit the multiple processors.

In a benchmark test, a Gensis MP600 workstation performed a 100-pixel radius Gaussian blur of a 50MB RGB image in 39.5 seconds (about 0.4 mega-samples per second). For the same machine and image size, RGB to CMYK colour space conversion took 10.9 seconds and RGB to Lab conversion took 3.7 seconds.

Other high end platforms include Sun SPARC workstations with the visual instruction set (VIS) which provide MMX style pixel-parallel operations. High end work stations are approaching or surpassing the performance achieved from specialised DSP based accelerator cards. A careful analysis is required to establish that a XC6200 based implementation of PhotoShop filters would be competitive against such high end workstations.

New DSP based PhotoShop accelerators would use the latest DSP chips e.g. the Texas Instruments TMS320C54x fixed-point digital signal processor [3]. This can support 40 million multiply-adds per second. Accounting 25ns for the cost of being a part of a PCI-based system and allowing 3 times 25ns for each channel this gives a total delay of 100ns for each multiply-accumulate. For a size 100 mask this yields a minimum execution time of around 277 seconds. This seems to suggest that high end microprocessors that support pixel-parallel and multiply-accumulate operations still have an edge over DSP based systems.

We wrote a straight forward software implementation of a Gaussian Blur filter in Ada on Pentium Pro 150MHz machine running under Windows/NT. This took about 4 min-

utes to execute for an 80MB image with a size 7 mask, which indicates that PhotoShop filters are highly optimised.

Even if the XC6200 based filters do not offer better performance with respect to very expensive top of the range workstations, these filters may still be viable when compared with typical desktop machines since the cost is only $1000 for the card.

6 XC6200 FPGA/PCI Card Based Filters

There is certainly enough resource on the XC6216 to implement several 8-bit by 8-bit multiply and add units. This architecture would read in the filter values on each clock tick and a new source image value every n^2 ticks. In this case the filter coefficients would also reside in the on board SRAM and two memory reads would be required for each pixel multiply-and-accumulate. This is prohibitively expensive.

A faster alternative is to store the filter values on the FPGA itself. Ideally, one would like to use n^2 constant coefficient multipliers, since the kernel is usually fixed and we could exploit the existing reconfigurable multiplier design. An 8-bit reconfigurable multiplier and accumulator with a 16-bit result is 16 cells high and 16 cells wide. At most 32 of these multiplier can be accommodated on a XC6216. For small filter kernels e.g. size 5 (e.g. 25 multiply-accumulates), it may be possible to implement a systolic high speed convolution. However, this can only be accomplished by using serial multipliers for small values of n.

Using a 40MHz 3-stage pipelined multiplier designed we can expect a critical path of around 50ns (accounting for other I/O and memory access delays). This would require 4.9e-7 seconds to process each pixel if we can realise a systolic array based filter which has been subjected to retiming, slowdown and hold-up. Regardless of the mask size, such a circuit would produce 20 mega-samples per second (with larger latencies for larger masks). However, we can not accommodate so many multipliers at once on the FPGA for mask sizes greater than 3.

A compromise architecture would be to use a pipelined multiplier for which the source signal is shifted in serially but the coefficients are available in parallel. A multiplier of this type has been designed at Xilinx which takes 16 clock ticks to produce a 16-bit multiplier result. Assuming a critical path of 15ns for the serial/parallel multiplier, this yields a throughput of about 4 mega-samples per second which is still ten times faster than the Gensis MP600 workstation.

7 Accelerating Adobe Photoshop

This section presents a concrete illustration of a novel application of the XC6200 FPGA. In particular, we show how the FastMap interface greatly simplifies the design of systems that comprise of communicating hardware and software.

We selected an off the shelf application for FPGA-based acceleration. Our choice was Adobe Photoshop, which is used extensively for high quality image processing. This application provides a collection of 'filters' that perform various image processing operations. The filter menu of Photoshop is shown in Figure 4 below. The available filters are not fixed, but instead are read as plug-ins to Photoshop. This means that a user can purchase more filters or develop more filters and extend the functionality of Photoshop without access to the source code of the application.

We used the publicly available Photoshop Software Development Kit (SDK) to implement a variety of filters that use the XC6200 to accelerate the image processing time. We have been concentrating on colour space conversion (RGB to greyscale conversion) and convolution style calculations (e.g. Gaussian Blur). Gaussian Blur is one of the slowest operations in Photoshop and is often used as a benchmark when assessing the performance of desk-top publishing systems.

Figure 4 The Filter Menu of Adobe Photoshop

The filter plug-ins were developed in C++ and compiled as Windows DLLs. A circuit for converting RGB to greyscale was designed (by performing a weighted average), as shown in Figure 6. The binary programming information for the circuit is compiled into the DLL as a Windows resource.

The interesting thing about this circuit is that it uses the wirless I/O feature of the XC6200. The input RGB is written by the plug-in software into the lower 24-bits of column 0. This write operation automatically triggers a clock pulse which after a suitable delay latched the greyscale result into a 24-bit register in column 14. This register is then read by the plug-in software and is used to create the new greyscale image. In this model of operation, the image is written and read a pixel at a time to the FPGA which resides on a PCI card. A more efficient version of this circuit was also designed which reads and writes the image from a local

SRAM frame buffer. This version operates at 20 million pixels per second.

8 Developing Photoshop Plug-Ins

The plug-ins are added to Photoshop by simply copying them into a filter directory. When Photoshop runs, it scans this directory and then loads the plug-ins. Some of the plug-ins developed at Xilinx are shown in the Photoshop filter menu in Figure 5.

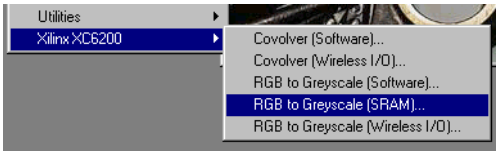


Figure 5 Some of the Xilinx filters developed for Photoshop

9 A Colour to Greyscale Filter

9.1 Colour Space Conversion

A simple way to produce a greyscale image from a colour RGB (red, green, blue) format image is to modify each RGB value in the target to be the simple average of the red, green and blue intensities. However, the human eye is more sensitive to some frequencies than others. A better result is achieved if a weighted average is used. We implemented several filters based around the following average which produced good results: grey = 0.29*red + 0.587*green + 0.114*blue. In hardware, this was implemented using a series of shifts and adds that give a close enough approximation.

9.2 A Software Only Filter

We first produced a software only version of a plug-in that peforms this weighted average. This operation of this filter was analysed and we measured a performance of around 0.05 mega-pixels per second. Plug-in filters spend a considerable amount of time getting a copy of the image from Photoshop in the format required and then submitting the result image (which Photoshop converts back into its internal format). All of the filter speeds presented in this paper including the time spent communicating the image to the plug-in and the overhead of the plug-in interface protocol.

Direct access to the image data used by Photoshop would eliminate the time taken to copy large images to and from the plug-in buffers, and also avoids having to create so many large temporary image buffers too.

The software greyscale filter that we developed is not quite as fast at the built in image mode conversion function of Photoshop. This is in part due to the fact that this operation is not implemented as a filter in Photoshop (it is a special menu item), so there is no image communication overhead.

10 A Wireless I/O Greyscale Filter

A layout of a circuit that converts a 24-bit RGB value to a 24-bit greyscale value (8-bit weighted average on each channel) is shown in Figure 6 on a XC6216 device (64 by 64 cells). The interesting thing about this circuit is that the inputs and outputs use wireless I/O (also called FastMap). The input RGB value is written by the plug-in software into the lower 24-bits of column 0. This write operation automatically triggers a clock pulse which after a suitable delay latches the greyscale result into a 24-bit register in column 14. This register is then read by the plug-in software and is used to create the new greyscale image. In this mode of operation, the image is written and read a pixel at a time to and from the FPGA which resides on a PCI card.

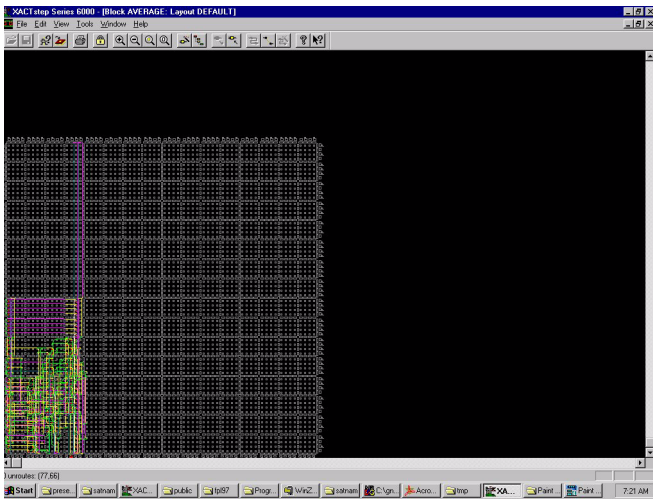


Figure 6 An RGB to Greyscale Conversion Circuit using FastMap

This circuit can be operated at around 20 mega-pixels per second on the VCC Hotworks XC6200 FPGA cards that we used. However, the PCI bottleneck reduces the perceived performance to just 0.12 mega-pixels per second. This is about twice as fast as the software only version, but very much slower than the circuit's potential. Unfortunately, the card and PC system we used delivered a very poor PCI performance, although VCC have subsequently greatly improved the PCI interface.

The programming information for this circuit was compiled into plug-in as a binary resource. The plug-in itself is a windows Dynamically Linked Library (DLL). The plug-in code contains calls to objects of a C++ class that provides a convenient interface to the FPGA and board resources. Assuming the user of such a system already has a card installed with its device driver, we need only supply one file which is used to accelerate Photoshop. This one convenient module contains both the hardware and software needed to add accelerate Photoshop.

The core part of the C++ filter code is shown below. The code uses an instance of a class that represents the XC6200 board and its capabilities (called board). The code includes two nested for loops that iterate over the image. For each pixel, a 24-bit word is read from the source image buffer (srcPtr) and written to the FastMap register in column 0 on the board. When the write occurs, the circuit is automatically clocked (using the CBUF feature of the XC6200), which means by the time the processor is ready to execute the next instruction the greyscale value has already been deposited into the register in column 14. Indeed, the next instruction reads the content of this register and places the greyscale result in the appropriate part of the destination image buffer.

```
for (i=0; i < rows; i++)
  for (j=0; j < columns; j++)
  { board.setColumn (0, *((word*) srcPtr)) ;
    *((word*)dstPtr) = board.getColumn (14) ;
    srcPtr += gStuff->planes+1;
    dstPtr += gStuff->planes+1;
  }
```

The FastMap feature of this device greatly simplifies the task of providing an FPGA co-processor its input data and reading the result. Indeed, the plug-in code above is good for any wireless I/O filter that computes a destination pixel based purely on the information in the corresponding source pixel.

10.1 Using the on-board SRAM

The wireless I/O greyscale filter has a disappointing performance gain with respect to the software version. Most of the time is taken sending pixels back and forth over the PCI bus and in executing many lines of C code for each pixel transaction.

By using the SRAM available on the VCC XC6200 card we can send the image to the board in blocks (benefiting from versions of the PCI interface that support block transfers) and we only execute a few lines of C code for each block that is processed. Then the FPGA can process the on-board image without software intervention. This allowed us to design circuits that execute at around 20 million pixels per second, but these systems were far more complicated to design because we also had to implement the SRAM interface and control logic on the XC6200.

The layout of the SRAM greyscale circuit is shown in Figure 7. The core circuit is the same as the circuit used in the wireless I/O version. However, extra circuitry was needed for address generation, and for generating the correct control signals to the SRAM and for latches to hold in-coming and outgoing data. This circuit fares much better than other SRAM based XC6200 circuit which are clocked off a CBUF pulse, thus limiting the speed of the circuit to the speed at which a C loop can be executed in software. One problem with this implementation is that it has proved to be sensitive to different types of XC6200 PCI cards. Some cards have different SRAM chips with slightly different timing behaviour. This gives unreliable performance, since sometimes the filter produces a blank image (or an image in which only some of the output channels are correctly processed).

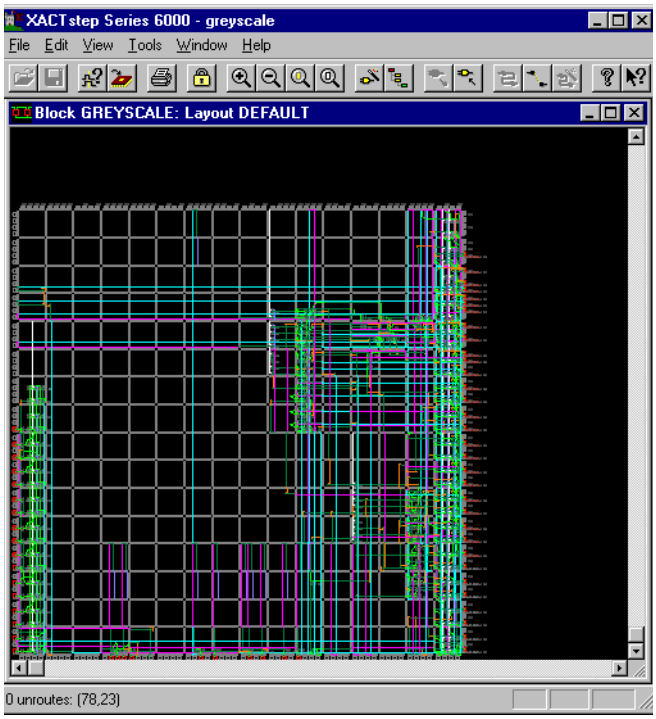


Figure 7 SRAM Version of the Greyscale Circuit

The core of the filter plug-in code is shown below. The size of the image is calculated and is then sent to the on board RAM using the writeRAM member of the board class. The circuit is cleared, and then we wait 60ns for the entire image to be processed (in principal we could vary the time depending on the image size). The circuit will read-write-modify each word of the on-board SRAM image buffer. We then read processed image back from the same location on the on-board SRAM.

```
board.writeRAM (0, (word*)srcPtr, imageSize);
board.clear () ;
Sleep (60) ;
board.readRAM (0, (word*)dstPtr, imageSize);
```

This circuit delivers a performance of around 0.22 mega-pixels per second. However, on the VCC Hotworks board the image is processed at 10 mega-pixels per second. Again, we suffer due to the PCI bus.

11 Convolution Filters

We have also developed a series of convolver circuits for performing operations like Gaussian Blur. These circuits per-

form 1D and 2D convolutions. Some versions just operate over three pixels in one dimension, but perform full precision parallel multiplies and additions (unlike the corresponding MMX implementation). Other versions process a 3 by 3 window for 2D convolution simultaneously on three channels (red, green and blue) performing 27 serial multiplications at a time at up to 8 million pixels per second.

11.1 1D Image Convolution

A one dimensional 3-pixel image convolver was designed using wireless I/O (FastMap) for the image input and output. The layout is shown in Figure 8. This is a systolic design, where each processing stage contains a multiplier and an adder. The multiplier is the constant coefficient multiplier designed by Kean et. al. [2]. This allows us to very quickly reconfigure the multiplier for different weights. The multiplications are 8 bits by 8 bits, with the intermediate results stored to enough precision to give an accurate 8-bit result (assuming that all the coefficients add up to a weight of 255).

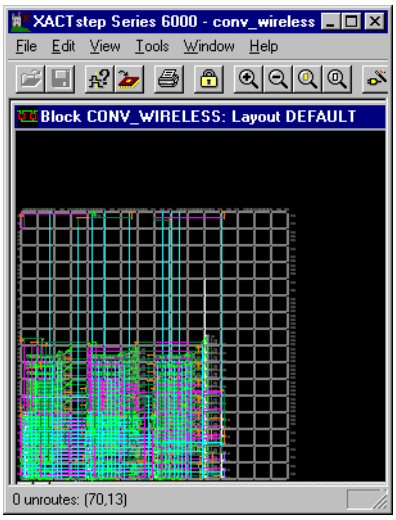


Figure 8 1D 3-pixel Wireless I/O Convolver

A software version of this convolver plug-in runs at 0.02 mega-pixels per second. The wireless I/O version shown above ran at 0.1 mega-pixels per second (once again, performance is limited by the PCI bus). The circuit can be clocked at up to 8MHz.

11.2 2D 3x3 Image Convolver Plug-In

The previous section has shown how we performed a 1D convolution to an image. This convolver operates on each line of the image independently. We have also designed a filter which can perform a 2D Gaussian blur. In order to do this we need a 2D convolver which operates on pixels in a line as well as pixels in a row together to calculate a single pixel result. The size of the convolver, or window, is determined by the radius of the Gaussian blur to be performed. Our first attempt uses a 3X3 window which translates to a 1.5 pixel radius. The 3X3 window gets convolved with a 3X3 table of coefficients. These coefficients are calculated such that they represent a Gaussian distribution curve. The idea is to perform a spatial low pass filter on a grouping of pixels. The centre pixel, which is located at the top of the curve, has the most weight. As you move away from the centre pixel the pixels have less influence and, therefore, have less weight. The actual convolver circuit that we designed is shown in Figure 9.

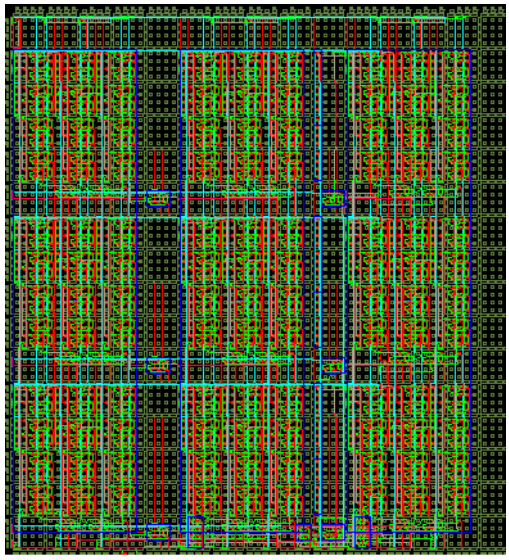


Figure 9 2D 3x3-pixel Convolver Layout

In order to perform a 3X3 convolution you need to do 9 multiplies and 8 adds. The Gaussian coefficients are symmetrical. We can exploit this symmetry and reduce the circuit to 3 multiplies and 8 adds. Because we wanted to be able to perform other filters besides the Gaussian blur we kept the convolver flexible. Our convolver does not exploit symmetry and, therefore, does 9 multiplies and 8 adds. Because the images that our filter operates on are RGB the convolution has to be done on each colour. This means a total of 27 multiplies and 24 adds for each pixel.

We chose a bit-serial approach to minimize the amount of chip real estate. The adders are a carry-save variety and take up 5 cells.

The multipliers are parallel/serial. The coefficient is loaded into a parallel register and the data is shifted through the multiplier serially. The partial products are added using carry-save adders. The size of the multipliers is determined by the width of the coefficient. We used an 8 bit coefficient. This requires 8 AND gates and 8 adders for a total of 48 cells per multiplier.

Data for this filter is provided from the 32 bit dedicated processor interface on the XC6200. The Photoshop plug-in controls the reading and writing of image data to and from the VCC Hotworks board. Because the convolution occurs on a 3 pixel column the plug-in does three write accesses to the board before the filter can calculate a result at which time the plug-in reads back the result. Reading and writing to the XC6200 is done using the FastMap interface feature of the device.

The XC6200 has another unique feature that we used for letting user logic determine when the microprocessor interface has accessed a register. Whenever the microprocessor interface accesses a register a special internal signal gets pulsed. After the plug-in is done writing the filter needs to crunch on the data and produce a result. This pulsed signal is used to start the filters state machine. This pulsing feature prevented us from having to design extra logic to start the state machine as well as extra software which would have slowed the filter down. The data gets written to a shift register on the XC6200. Then the multiply and add functions are performed on the serial data. The final result is shifted back into a shift register where the plug-in reads back the result.

The performance of the 2D convolver is 0.1 mega-pixels per second. Without the PCI bus limit, it would perform at around 8 mega-pixels per second.

12 Filter Development Procedure

The plug-in software was developed using the freely available SDK from Adobe, which contains ready to use projects for use with Microsoft's C++ compiler (Developer Studio). We made extensive use of the XC6200DS C++ class library, which provides high level access to all the board resources including the FPGA, the on-board SRAM and the on-board programming clock generator.

Most of the filters were designed using the Lava hardware description language. This is based on the algebraic hardware description language Ruby [8], and allows us to specify the behaviour and layout of circuits using powerful circuit combinators [7].

13 Limitations

The filters produced worked perfectly on the development PC using the VCC board. However, some of the filters did not work on other versions of the VCC cards, and on some of the Annopolis XC6200 PCI cards, even though all these cards are made to a common reference standard. One problem is that these cards using different SRAM chips (some by Toshiba some by NEC) which have slightly different timing diagrams and speeds. This means that some of the SRAM-based filters do not port properly.

The performance of the system is very much bounded by the PCI bandwidth. During the development of this project the PCI interface core was modified by VCC, which resulted in a board that went twice as fast for some of our filters (e.g. the SRAM version of the greyscale circuit).

The Hotworks board has 2 Megabytes of SRAM. Bursting large chunks of image data to the SRAM is much more efficient than doing a bunch of single pixel writes to the filter. Once the image data is resident on the board the filter can get very quick access to the data while avoiding PCI transfers. Once the filter has operated on all the data and written the results back to the SRAM the PCI bus can burst the contents of the SRAM back to main memory. A drawback of this approach is the added complexity of the circuit design to control the SRAM.

In the case of the 3X3 convolution we had to do 3 PCI writes to the board before each calculation. Two of the writes are redundant because we didn't use line buffers. If we had created line buffers in the SRAM then the first time a pixel was written it would have been saved in the line buffer and the next two times the pixel was needed it could have been quickly accessed from the buffer. Bursting entire lines to the SRAM would have increased performance even more. A drawback of this method is creating the circuitry to control the address generation for the line buffers, particularly for arbitrary image sizes. Having an embedded microprocessor resident on the board would greatly simplify this issue.

14 Future Work

There is clearly a need to provide operating system level support for such applications. In particular, we need a standard software architecture (or hardware abstraction layer) that allows systems to be built which are independent as possible from the details of a specific board. For example, we would like to be able to abstract from details like the type of SRAM device used. And we would like to have straightforward information about the board resources e.g. how much SRAM is installed.

Another reconfigurable processing board we would like to test is the ACE*card*™ from TSI TeleSys. It has an embedded microprocessor which would be very useful for handle reconfiguration and controlling the memory. It has a lot of memory which makes image storage on the board more flexible. It also has a lot more gates. This board has a pair of XC6264 devices on it. This is eight times the capacity of the board we are using.

Another way to improve performance is to use a card that supports Intel's Advanced Graphics Port (AGP) [1] which allows transfers up to 533Mb/s on a separate bus, with the control signals passing as usual over the PCI bus. To our knowledge AGP is not supported on any commercially available FPGA cards.

We are now working on the implementation of anti-aliasing filters based on simple linear interpolation. We are also considering more compute intensive operations like RGB to

CMYK colour space conversion. This occurs every time a page is printed, and in conjunction with colour correction is a very slow process.

15 Summary

This experiment shows that it is possible to use a generic PCI-based FPGA card which be purchase from various vendors for the acceleration of shrink-wrapped applications via plug-in technology. As each different filter is selected, a new design is downloaded. This makes much more flexible use of the FPGA than is typical in embedded applications. Furthermore, the close association between on-chip registers and program variables greatly simplifies the production of the plug-in software.

However, a lot of work still need to be done before one can produce hardware/software filters and sell them as commodity items. There needs to be greater standardisation of card resources and a higher level run-time support for application software.

16 Acknowledgements

This work was carried out with the support of several people at Xilinx including Jaime Cummins, Raj Patel, Bernie New, Dennis Rose, Stuart Nisbet, Beth Cowie, Tom Kean and Ann Duncan. Finally, thanks to John Gray, who got us hooked in the first place and continues to encourage us.

References

[1] Intel. *Accelerated Graphics Port Interface Specification Revision 2.0*. December 11, 1997.

[2] T. Kean, B. New, B. Slous. *A Multiplier for the XC6200*. Sixth International Workshop on Field Programmable Logic and Applications. Darmstadt, 1996.

[3] H. T. Kung. *Why Systolic Architecture*s. IEEE Computer. January 1982.

[4] Charles E. Leiserson. *Systolic and Semisystolic Design*. IEEE Conference on Computer Design/VLSI In Computers (ICCD'83). 1983.

[5] William H. Mangione-Smith, Brad Hutchings, David Andrews, André DeHon, Carl Ebeling, Reiner Hartenstein, Oskar Mencer, John Morris, Kirshna Palem, Viktor K. Prasanna, Henk A. E. Spaanenburg. *Seeking Solutions in Configurable Computing*. IEEE Computer, December, Vol. 30, No. 12. December 1997.

[6] Satnam Singh and Pierre Bellec. *Virtual Hardware for Graphics Applications using FPGAs*. FCCM'94. IEEE Computer Society, 1994.

[7] Satnam Singh. Architectural Descriptions for FPGA Circuits. FCCM'95. IEEE Computer Society. 1995.

[8] M. Sheeran, G. Jones. *Circuit Design in Ruby*. Formal Methods for VLSI Design, J. Stanstrup, North Holland, 1992.

[9] Xilinx. *XC6200 FPGA Family Data Sheet*. Xilinx Inc. 1995.

[10] J.D. Foley, A. Van Dam. *Fundamentals of Interactive Computer Graphics*. Addison Wesley. 1984.