

# Parameterized Convolution Filtering in a Field Programmable Gate Array

Richard G. Shoup

Interval Research  
Palo Alto, California 94304

## *Abstract*

*This paper discusses the simple idea of parameterized program generation of convolution filters in an FPGA. Applications in image processing include real-time video and desktop publishing. An example 2-D filter pipeline is assembled from a set of multipliers and adders, which are in turn generated from a canonical serial-parallel multiplier stage. The need for more powerful FPGA development tools is discussed using an analogy to software development.*

## INTRODUCTION

Many image processing operations such as scaling and rotation require resampling or convolution filtering at each pixel in the image. Depending on the bandpass and desired quality (i.e. the width of the filter kernel), computing this weighted sum of neighboring pixels can require significant amounts of computation, thus suggesting a highly parallel implementation in special-purpose hardware.

Current high-performance image transformation hardware such as that used in video production ("special effects") devices performs these calculations, but often with compromises in quality, especially for high scale factors. Most such devices (and indeed most software codes used in computer graphics rendering) have a fixed, sometimes quite small (5 or less) maximum filter width and simply ignore the aliasing and lost information that results. However, as video resolution increases (e.g. high-definition television) and other interactive applications such as graphic arts and page composition become more demanding, the quality of a significantly scaled or transformed picture must remain high. Furthermore, in many such applications multiple cascaded transformations are typical, and any losses at each stage become cumulative.

It is worth noting that, despite many differences, image processing applications in both the video and hard copy domains often require similar high performance. For real-time video, the average pixel rate required is at least

$$640 \text{ pixels} * 480 \text{ pixels} * 30 \text{ frames/sec} = 9.2 \text{ Mpixels/sec.},$$

with a peak rate of 14.3 Mpixels/sec. for the (American) NTSC standard. In the case of an image processing copier, scanner, or printer the rate desired is generally on the order of

$$3000 \text{ pixels} * 4000 \text{ pixels} * 1 \text{ frame/sec} = 12 \text{ Mpixels/sec.}$$

Typically, the number of operations needed at each pixel is 10 to 100 or more, and thus the desired performance is well beyond the capabilities of even the fastest single-processor workstations today. For applications such as high-definition video or high-quality film printing, the numbers are much greater still. Clearly these challenges can only be met using highly parallel computing structures. We can employ parallelism in two ways: in space (replication: multiple instances operating concurrently), and in time (pipelining: multiple steps of the computation operating concurrently). Both of these are available to the designer using FPGAs.

In image processing functions such as convolution filtering, high performance can be achieved by exploiting parallelism, but different filter widths and thus potentially different hardware structures are needed for different applications. It is therefore difficult to make a fixed parallel structure efficient. In an application involving spatial scaling of images, for example, a larger filter kernel would be required for large scale factors, a small one for modest scaling. It would be expensive to implement the entire largest desired filter kernel, and wasteful for small scale factors. On the other hand, a fixed small kernel could be duplicated for high performance (the usual choice), but at a sacrifice in quality for larger scale factors. In addition, since the fan-in from input pixels and fan-out to output pixels changes with filter kernel size, considerable flexibility is needed with respect to pixel memory accessing and I/O data paths.

## **AN FPGA CONVOLUTION PIPELINE**

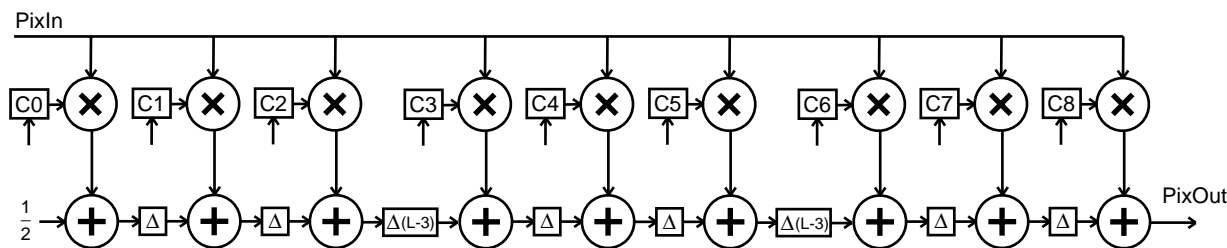
In the context of 2-dimensional image processing, convolution is a standard operation often used for filtering (blurring, sharpening, compression, noise cleaning) or resampling (compositing, scaling, rotation, warping, texture mapping), etc. (Pratt, 1991). The function at each pixel is the weighted sum of neighboring pixels given by

$$V'(x,y) = \sum_{i,j} C(i,j) * V(x+i,y+j),$$

where  $C$  represents the convolution filter kernel, usually a small 2-dimensional array of weights. If the filter size is  $m \times m$ , then each output pixel depends on  $m^2$  adjacent pixels, and thus  $m^2$  multiplies and adds are required at each site. (Although many 2-D functions can be computed separably as the combination of two 1-D functions, we will concentrate here on the general 2-dimensional case.)

Figure 1 shows an example pipeline implementing a  $3 \times 3$  convolution filter for video applications. Pixels arrive in raster-scan order, and each is simultaneously multiplied by all 9 filter coefficients. Partial sums are then accumulated by the adder chain, with a delay of one pixel between adjacent adds and a delay of the image line length minus 3 pixels between

rows to allow proper alignment. The partial sums may be accumulated at double precision to minimize loss. Because the sum is initialized to  $1/2$ , the result is rounded automatically when it is truncated at the output.



**Figure 1** A 3x3 convolution pipeline

Although there are various ways to implement a convolution filter, note that this pipeline is optimal in the sense that each input pixel is accessed only once, regardless of the  $m^2$  fan-in. Output values are produced at the same rate, so maximum parallelism is achieved for the given I/O bandwidth. Also, because of the pipeline delay, output pixels may be rewritten directly into the source image memory and no double buffering is necessary.

### Multiplier generation

Using a macro building utility, we can assemble copies of various macros and primitive elements into a multiplier, and then compose several multipliers and adders into the complete filter pipeline. Conventions as to expansion and mapping of signal names (e.g.  $X_i$  becomes  $X0, X1, \dots$ , etc.) allow appropriate connections to be made automatically by the router.

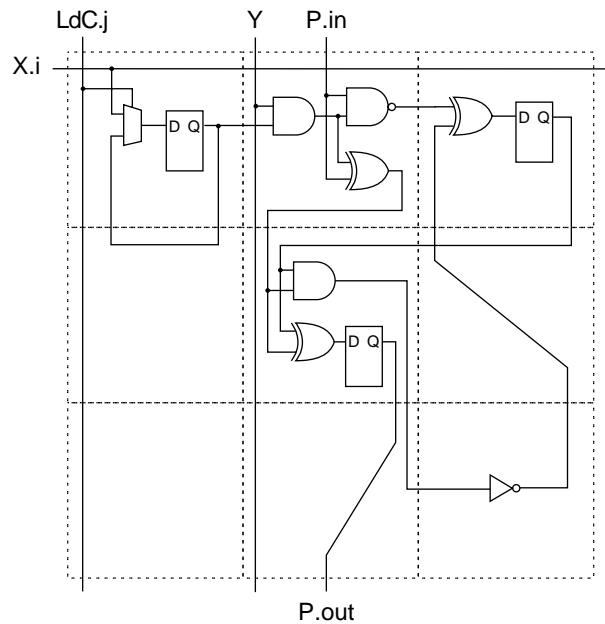
Figure 2 shows one stage of the serial-parallel carry-save multiplier with one bit of coefficient register implemented in 9 cells of an FPGA (CLI 1992). In this particular multiplier design (after Lyon 1976), one operand ( $X$ ) is supplied in parallel to several stages, while the other operand ( $Y$ ) arrives in bit-serial fashion to all stages simultaneously. Stages are stacked vertically to the arithmetic length desired, with the high-order bit at the top and the low-order bit at the bottom, and intermediate results propagating downward via path  $P$ . Result bits emerge serially from the bottom of the multiplier, lsb first. Not shown are common clock and reset lines.

In order to assemble a complete multiplier from this macro cell, we need only stack the appropriate number of these stages, and perform a simple mapping of signal names to make the proper connections. Each  $P.out$  signal connects to  $P.in$  of the stage below. Each stage receives one bit of the parallel operand, so input  $X_i$  of the  $i$ 'th stage is connected to the coefficient bus signal  $X_i$ . The serial operand  $Y$  is bused vertically to all stages in the multiplier, as is the common signal  $LdC_j$ , which enables loading of the entire coefficient register.

### Filter kernel generation

The complete filter is then assembled from  $m^2$  multipliers placed side by side, each accompanied by an accumulating adder. In addition, we must insert appropriate delays

between adders to provide alignment of partial results. All of these components are placed as hard macros at fixed locations in the array, with appropriately expanded signal names, and routing software is used to make the proper connections.



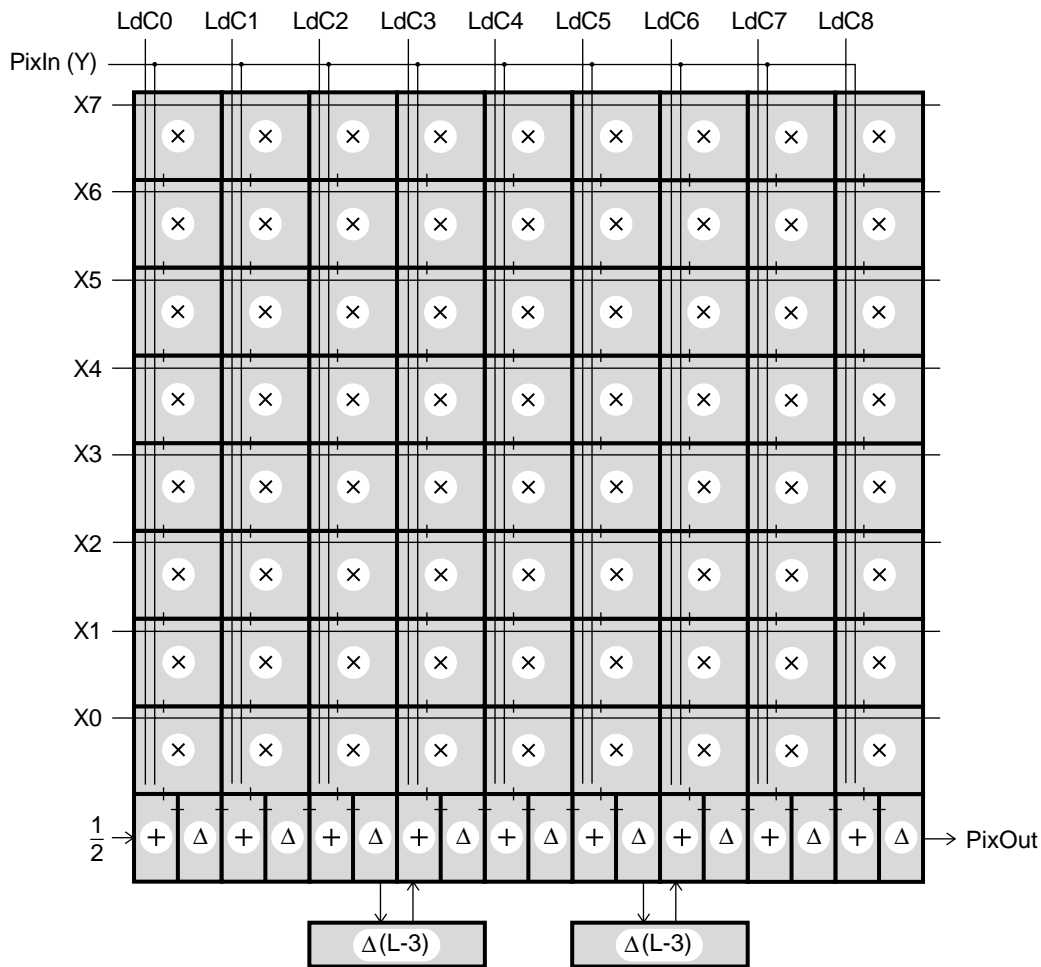
**Figure 2** FPGA multiplier stage with coefficient register

In our implementation of the above 3x3 convolution pipeline, 9 of the serial-parallel multipliers are used, each operating on 8-bit pixel values with an 8-bit two's complement coefficient, and 16-bit intermediate sums. Figure 3 shows this complete 3x3 convolution in its FPGA form. (Similar multiplier and filter designs using the same FPGA have been developed separately by Rupp and Andraka (1993)).

The line-length-minus-3-pixel delays between adders 2 and 3 and between adders 5 and 6 may be implemented in several ways: as a shift register made up of register cells within the FPGA array, as a separate shift register circuit, or as a circular buffer in a separate RAM. We utilize the latter approach, interleaving writes and reads to the local RAM, with sequential addresses offset by an appropriate amount. The memory addressing circuitry is also a macro that is automatically placed in relationship to the multiplier array.

## Operation

In this design, the 9 filter coefficients are assumed to be loaded before the processing of the input begins using buses X0 to X7 and the LdCj signals along the top of the array. Input pixels arrive serially on the Y line, each padded with an additional 8 zeroes to allow the complete 16-bit result to be computed and shifted into the adder stage at the bottom of each multiplier. Each adder simply computes the bit-serial sum of the stream arriving from the left and the input from its multiplier above and outputs the result to the right, where it is delayed by 16 bit times to align with the next partial result.



**Figure 3** 3x3 convolution pipeline in FPGA form

After 9 such accumulations, the complete weighted sum emerges from the rightmost adder and may then be registered and truncated to give a properly rounded 8-bit result. Pixels are thus produced at the same rate as they are consumed, namely one per 16 bit times. At a (modest) clock rate of 16 MHz., for example, one copy of this filter computes all 9 multiplies and 9 adds every 1  $\mu$ sec., or at a rate of roughly 18 Mops/sec. Using the multiplier stage of Figure 2, a complete 3x3 filter (not including the memory interface) occupies about 1/4 of the present FPGA. Faster operation could, of course, be achieved by using fully parallel multipliers, at the expense of more circuitry.

### Further parameterization

If the filter coefficients are held constant during the entire processing of an image, the multipliers can be constructed as fixed functions with the coefficient values built in. Each multiplier cell of Figure 1 can be simplified in the obvious way (not shown) to implement either multiply by 0 or multiply by 1 and the filter assembled from these elements, with no coefficient registers necessary.

Alternatively, the filter coefficients can be made variable from pixel to pixel, as is necessary for non-integer scaling of an image, for example. In this case, the coefficients can be obtained from a local memory (table) that is addressed according to the subpixel position of the current site (Crow 1984).

A similar technique may be used as well for 1-D separable (2-pass) scaling, rotation and other transformations (Catmull and Smith 1980, Smith 1987). By using a different kernel macro instead of a multiplier, but in a similar arrangement, one can implement many other local image processing functions such as morphology, correlation, pattern matching and feature extraction.

## **BEYOND MACRO GENERATION**

Macro generators of the type described here are simple yet powerful tools for creating regular computational structures. Beyond these, however, we need more powerful and expressive languages and techniques that capture concepts similar to those commonly used in modern software development environments, including self-similar hierarchies, object-oriented structures and functional specification.

By analogy to software development environments, FPGA design tools are now at the stage in which assembly language programming is still the norm, and the first compilers of algebraic languages (e.g. VHDL, etc.) have not yet reached maturity or come into general use. Schematic-based tools have the potential to evolve from engineering CAD tools into powerful visual language editors, but have not yet done so. What is desired is a higher level of tool, accessible to programmers and system designers without the need for detailed hardware engineering knowledge.

It is not so important, for example, to achieve very high percentage utilization of FPGAs, any more than it is important for software compilers to utilize every main memory location efficiently. What is important is that complex computational structures can be specified, implemented, and modified easily and quickly. It should be no more difficult or time consuming to implement a function in FPGA hardware than to write and debug the equivalent software program.

Currently, placement and routing software is often a major bottleneck and source of difficulty. Most designers of FPGAs have given significant attention to connectivity and routing within their arrays to somewhat ameliorate this problem (for example Altera 1992). This problem is considerably harder than the analogous step of linking in software compilation, since mapping (packing) into a plane (or small set of planes) is required. Although simulated annealing and other artificial intelligence techniques have been used in this area, these generally take far too much time to be practical in an FPGA development environment. For the foreseeable future, programs will require additional knowledge and assistance in placement and routing, and this is precisely what is supplied by generators of the type discussed above.

FPGA design tools should preserve hierarchical information such as relative placement within macros at every level of the hierarchy and allow the designer to interact at any level of the structure. They should provide the designer with the ability to indicate placement, layout, and even detailed wiring when this is advantageous, but not require it.

## CONCLUSIONS AND FUTURE RESEARCH

As FPGA technology matures and much larger arrays become practical, techniques that allow the automatic generation of highly-parallel architectures will become central to high-performance computing. We have described some simple techniques for automatic generation of convolution pipelines for image processing and other applications. Higher-level techniques and approaches are also needed, much like those which have developed in software engineering.

However, a bigger long-term problem in the utilization of FPGAs in custom computing architectures is the separation of processing and memory that they perpetuate. FPGAs permit restructurable processing, and restructurable interconnects are also becoming available. But the trend is still towards larger integrated processing resources and larger fixed memory units, making the problem of flexible interconnect of subparts of these resources more difficult.

## ACKNOWLEDGMENTS

This work was supported in part by the Ricoh California Research Center. We also gratefully acknowledge assistance from National Semiconductor, Concurrent Logic, and Atmel Corporation.

## REFERENCES

- Altera FLEX Series Field Programmable Gate Arrays, Altera Corporation, San Jose, California, 1992.
- Andraka, R.J., "FIR Filter Fits in an FPGA using a Bit Serial Approach", Proc. 3rd Annual PLD Conference, CMP Publ., Manhasset, NY, 1993.
- Catmull, E., and Smith, A.R., "3-D Transformations of Images in Scanline Order", Computer Graphics, Vol. 14, No. 3, July 1980.
- CLi6000 Series Field Programmable Gate Arrays, Concurrent Logic, Inc., Sunnyvale, California, 1992. (Now marketed by Atmel Corporation, Santa Clara, California.)
- Crow, F.C., "Summed-Area Tables for Texture Mapping", Computer Graphics, Vol. 18, July 1984.
- Lyon, R.F., "Two's Complement Pipeline Multipliers", IEEE Trans. on Communications, April 1976.
- Pratt, W.K., *Digital Image Processing*, 2nd ed., Wiley, 1991.
- Smith, A.R., "Planar 2-Pass Texture Mapping and Warping", Computer Graphics, Vol. 21, No. 4, July 1987.