Accelerated Image Processing on FPGAs[1]

Bruce A. Draper, J. Ross Beveridge, A.P. Willem Böhm, Charles Ross, Monica Chawathe, Jeffrey Hammes

ABSTRACT

The Cameron project has developed a language and compiler for mapping image-based applications to field programmable gate arrays (FPGAs). This paper tests this technology on several applications and finds that FPGAs are between 8 and 800 times faster than comparable Pentiums for image based tasks.

1) Introduction

Although computers keep getting faster and faster, there are always new image processing (IP) applications that need more processing than is available. Examples of current high-demand applications include real-time video stream encoding and decoding, real-time biometric (face, retina, and/or fingerprint) recognition, and military aerial and satellite surveillance applications. To meet the demands of these and future applications, we need to develop new techniques for accelerating image-based applications on commercial hardware.

Currently, many image processing applications are implemented on general-purpose processors such as Pentiums. In some cases, applications are implemented on digital signal processors (DSPs), and in extreme cases (when economics permit) applications can be implemented in application-specific integrated circuits (ASICs). This paper presents another technology, field programmable gate arrays (FPGAs), and shows how compiler technology can be used to map image processing algorithms onto FPGAs, achieving 8 to 800 fold speed-ups over Pentiums.

2) Field Programmable Gate Arrays

Field-programmable gate arrays (FPGAs) are non-conventional processors built almost entirely out of lookup tables. In particular, FPGAs contain grids of logic blocks, connected by programmable wires, as shown in Figure 1. Each logic block has one or more lookup tables (LUTs) and several bits of memory. As a result, logic blocks can implement arbitrary logic functions (up to a few bits), or be combined together to form registers. FPGAs as a whole can be used to implement circuit diagrams, by mapping the gates and registers onto logic blocks.

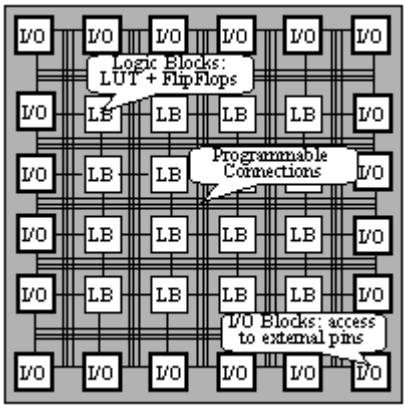

Figure 1: A conceptual illustration of an FPGA. Every logic block contains one or more LUTs, plus a bit or two of memory. The contents of the LUTs are (re)programmable, as are the grid connections. I/O blocks provide access to external pins, which usually connect to local memories.

FPGAs were originally developed to serve as test vehicles for hardware circuit designs. Recently, however, FPGAs have become so dense and fast that they have evolved from simple test and "glue logic" circuits into the central processors of powerful reconfigurable computing systems [1]. A Xilinx XCV-2000E, for example, contains XXX logic blocks, and can operate at up to 180 MHz (depending on the latency of the embedded circuit). The logic blocks can be configured so as to exploit data, pipeline, process, I/O parallelism, or all of the above. In computer vision and image processing, FPGAs have already been used to accelerate real-time point tracking [2], stereo [3], color-based object detection [4], video and image compression [5], and neural networks [6].

The economics of FPGAs are fundamentally different from the economics of other parallel

[1] This work was funded by DARPA through AFRL under contract F3361-98-C-1319.

architectures. Because of the comparatively small size of the image processing market, most special-purpose image processors have been unable to keep pace with advances in general purpose processors. As a result, researchers who adopt them are often left with obsolete technology. FPGAs, on the other hand, enjoy a multi-billion dollar market as low-cost ASIC replacements. Consequently, increases in FPGA speeds and capacities have followed or exceeded Moore's law for the last several years, and researchers can continue to expect them to keep pace with general-purpose processors [7].

Unfortunately, FPGAs are very difficult to program. Algorithms must be expressed as detailed circuit diagrams, including clock signals, etc., in hardware description languages such as Verilog or VHDL. This discourages most computer vision researchers from exploiting FPGAs; the intrepid few who do are repeatedly frustrated by the laborious process of modifying or combining FPGA circuits.

The goal of the Cameron project is to change how reconfigurable systems are programmed from a circuit design paradigm to an algorithmic one. To this end, we have developed a high-level language (called SA-C) for expressing image processing algorithms, and an optimizing compiler that targets FPGAs. Together, these tools allow programmers to quickly write algorithms in a high-level language, compile them, and run them on FPGAs.

Detailed descriptions of the SA-C language and optimizing compiler can be found elsewhere (see [8, 9],or http://www.cs.colostate.edu/~cameron/ for a complete set of documents and publications). This paper only briefly introduces SA-C and its compiler before presenting experiments comparing SA-C programs compiled to a Xilinx XCV-2000E FPGA to equivalent programs running on an Intel Pentium III processor. Our goal is to familiarize applications programmers with the state of the art in compiling high-level programs to FPGAs, and to show how FPGAs implement a wide range of image processing applications.

3) SA-C

SA-C is a single-assignment dialect of the C programming language designed to exploit both coarse-grained (loop-level) and fine-grained (instruction-level) parallelism. Roughly speaking, there are three major differences between SA-C and standard C: 1) SA-C adds variable bit-precision data types and fixed point data types. This exploits the ability of FPGAs to form arbitrary precision circuits, and compensates for the high cost of floating point operations on FPGAs by encouraging the use of fixed-point representations. 2) SA-C includes extensions to C that provide data parallel looping mechanisms and true multi-dimensional arrays. These extensions make it easier to express operations over sliding windows or slices of data (e.g. pixels, rows, columns, or sub-images), and also make it easier for the compiler to identify and optimize data access patterns. 3) SA-C restricts C by outlawing pointers and recursion, and restricting variables to be single assignment. This creates a programming model where variables correspond to wires instead of memory addresses, and functions are sections of a circuit, rather than entries on a program stack.

To illustrate the differences between SA-C and traditional C, consider how the Prewitt edge detector might be written in SA-C, as shown in Figure 2.

```
int16[:,:] main (uint8 image[:,:]) {
  int16 H[3,3] = {{-1,-1,-1}{0,0,0}{1,1,1}};
  int16 V[3,3] = {{-1,0,1}{-1,0,1}{-1,0,1}};
  int16 M[:,:] =
    for window W[3,3] in image {
      int16 dfdy, int16 dfdx =
        for w in W dot h in H dot v in V
        return(sum(w*h),sum(w*v);
      int16 magnitude =
        sqrt(dfdy*dfdy+dfdx*dfdx);
    } return(array(magnitude));
} return(M);
```

Figure 2: SA-C source code for the Prewitt edge detector. The Prewitt edge detector convolves the image with two masks (H & V above), and then computes the square root of the sum of the squares.

At first glance, one is struck by the data types and the looping mechanisms. "int16" simply represents a 16-bit integer, while "uint8" represents an unsigned 8-bit integer. Unlike in traditional C, integers and fixed point numbers are not limited to 8, 16, 32 or 64 bits; they may have any precision (e.g. int11), since the compiler can construct circuits with any precision[2]. Also, arrays are true multi-dimensional objects whose size may or may not be known at compile time. For example, the input argument "uint8 image[:,:]" denotes a 2D array of unknown size.

[2] Earlier versions of SA-C limited variables to no more than 32 bits, but this limitation has been removed.

Perhaps the most significant differences are in the looping constructs. "for window W[3,3] in image" creates a loop that executes once for every possible 3x3 window in the image array. Such windows can be any size, although their size must be known at compile time. In addition to stepping through images, SA-C's looping constructs also allow new arrays to be constructed in their return statements. In this case, "return (array (magnitude))" makes a new array out of the edge magnitudes calculated at each 3x3 window.

Perhaps the least C-like element of this program is the interior loop "for w in W dot h in H dot v in V". This creates a single loop that executes once for every pixel in the 3x3 window W. Since H and V are also 3x3 arrays, each loop iteration matches one pixel in W with the corresponding elements of H and V. This "dot product" looping mechanism is particularly handy for convolutions, but requires that the structures being combined have the same shape and size. A more thorough description of SA-C can be found in [8] or at the Cameron web site.

4) THE SA-C COMPILER

The SA-C compiler translates high-level SA-C code into dataflow graphs, which can be viewed as abstract hardware circuit diagrams without timing information [10]. The nodes in a data flow graph are generally simple arithmetic operations whose inputs arrive over edges. There are also control nodes (e.g. selective merge) and array access/storage nodes.

Dataflow graphs are a common internal representation for optimizing compilers. The SA-C compiler performs standard optimizations including common subexpression elimination, constant folding, operator strength reduction, invariant code motion, function in-lining, and dead code elimination. It also performs specialized optimizations for hardware circuits that reduce I/O bandwidth (e.g. loop fusion, partial loop unrolling), reduce circuit size (e.g. bitwidth narrowing, window narrowing, and temporal common subexpression elimination,) and increase the clock rate (pipelining, lookup tables). These optimizations are described in [11].

After optimization, the SA-C compiler translates data flow graphs into VHDL; commercial tools[3] synthesize and place and route the VHDL to create FPGA configurations. The SA-C compiler also generates host code to download the FPGA configuration, data, and parameters, to trigger the FPGA, and to upload the results.

5) IMAGE PROCESSING ON FPGAs

The SA-C language and compiler allow FPGAs to be programmed in the same way as other processors. Programs are written in a high-level language, and can be compiled, debugged, and executed from a local workstation. It so happens that for SA-C programs, the host executable off-loads the processing of loops onto an FPGA, but this is invisible. SA-C therefore makes reconfigurable processors accessible to applications programmers with no hardware expertise.

The empirical question in this paper is whether image processing tasks run faster on FPGAs than on conventional general-purpose hardware, in particular Pentiums. The tests presented below in Table 1 suggest that in general, the answer is yes. Simple image operators are faster on reconfigurable processors because of their greater capabilities for I/O between the FPGA and local memory, although this speed-up is modest (a factor of ten or less). More complex tasks result in larger speed-ups, up to a factor of 800 in one test, by exploiting the parallelism within FPGAs and the strengths of an optimizing compiler.

The reconfigurable processor used in our tests is an Annapolis Microsystems WildStar with 3 Xilinx XV-2000E FPGAs and 12 local memories. Our conventional processor is a Pentium III running at 800 MHz. The chips in both processors are of a similar age and were the first of their respective classes fabricated at 0.18 microns.

Routine	Pentium III	XV-2000E	Ratio
AddS	0.00595	0.00067	8.88
Prewitt	0.15808	0.00196	83.16
Canny	0.13500	0.00606	22.5
Wavelet	0.07708	0.00208	38.5
Dilates (8)	0.06740	0.00311	21.6
Probing	65.0	0.08	812.5

Table 1. Execution times in seconds for routines with 512x512 8-bit input images. The comparison is between a 800Mhz Pentium III and an AMS WildStar with Xilinx XV-2000E FPGAs.

5.1) The Simplest Image Operator: Scalar Addition

[3] Synplicity and the Xilinx Foundation Tools.

The simplest program we tested adds a scalar argument to every pixel in an image. For the WildStar, we wrote the function in SA-C and compiled it to a single FPGA. We compared its performance to the matching routine from the Intel Image Processing Library (IPL) running on the Pentium. In so doing, we compare the performance of compiled SA-C code on an FPGA to hand-optimized (by Intel) Pentium assembly code. As shown in Table 1, the WildStar outperforms the Pentium by a factor of 8.

Why is the WildStar faster? The clock rate of an FPGA is a function of the latency of the programmed circuit, but in general FPGAs run at lower clock rates than Pentiums. For this program, the WildStar ran at 51.7 MHz, compared to 800 MHz for the Pentium. Unfortunately for the Pentium, however, memory response times have not kept up with processor speeds, and the 512x512 source image is too large to fit in its primary cache. As a result, the Pentium is not able to read or write data at anything close to 800MHz, so its primary advantage is squandered.

FPGAs, on the other hand, are capable of parallel I/O. The WildStar gives the FPGAs 32-bit I/O channels to each of four local memories, so the FPGA can both read and write 8 8-bit pixels per cycle. This is four times the I/O bandwidth of the Pentium. Also, the operator's pixel-wise access pattern is easily identified by the SA-C compiler, which is able to optimize the I/O so that the program reads and writes almost 8 pixels per cycle.

The FPGA outperforms the Pentium on the scalar addition task by slightly more than I/O considerations alone would predict. This is because the SA-C compiler exploits both data and pipeline parallelism. On every cycle, the FPGA (1) reads eight 8-bit pixels, (2) adds eight copies of the scalar to the eight pixels read on the previous cycle (in parallel), and (3) writes back to memory the sums of the scalar with the eight pixels read on the cycle before that.

This program represents one extreme in the FPGA vs. Pentium comparison. It is a simple, pixel-based operation that fails to fully exploit the FPGA, since only 9% of the lookup tables (and 9% of flip-flops) were used. However, it demonstrates that FPGAs will outperform Pentiums on simple image operators because of their parallel I/O capabilities. The exact amount of the speed-up will depend on the number of local memories the FPGA has access to and the speeds of the memories, but in general one expects a small speed-up of less than a factor of ten.

5.2) Edge Operators

The Prewitt edge operator shown in Figure 2 is more complex than simple scalar addition. Every 3x3 window in the image is convolved with horizontal and vertical edge masks; and the edge magnitude at a pixel is the square root of the sum of the square of the convolution responses. When the Prewitt program written in SA-C is compiled for the WildStar, it computes the edge magnitude response image for a 512x512 input image in 1.96 milliseconds. In comparison, the equivalent Intel IPL function[4] on the Pentium takes 158.08 milliseconds, or approximately 80 times longer (see Table 1).

Why is the Prewitt edge detector so much faster on an FPGA? One of the reasons, as before, is I/O: the FPGA can read the input image and write the output image faster than the Pentium can. The advantage is magnified by the fact that edge detection is a window-based operator. A naïve implementation of a 3x3 window will slide the window horizontally across the image until it reaches the end of the row, at which point it will drop down one row and repeat the process. While shift registers can be used to avoid reading a pixel more than once on any given horizontal sweep, every pixel is still read three times, once for each row in the window. The SA-C compiler avoids this by partially unrolling the loop and computing eight vertical windows in parallel (see Figure 3). This reduces the number of input operations needed to process the image by almost a factor of three by exploiting the parallelism of the FPGA.

[4] iplConvolve2D with two masks (the Prewitt horizontal and vertical edge masks) and IPL_SUMSQROOT as the combination operator.

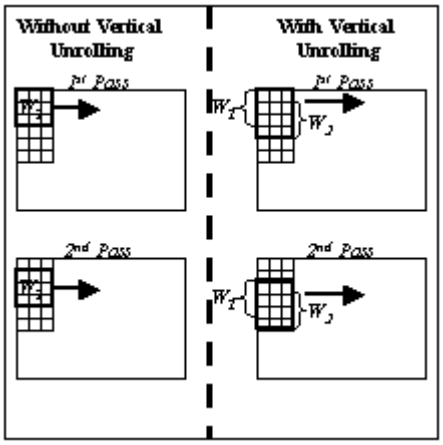

Figure 3: How partial vertical unrolling optimizes I/O. As a 3x3 image window slides across an image, each pixel is read 3 times (assuming shift registers hold values as it slides horizontally). Under partial unrolling, two or more vertical windows are computed in parallel, allowing the passes to skip rows and reducing I/O.

Of course, the parallelism of the FPGAs does more than just reduce the number of I/O cycles. We mentioned above that the FPGA computes eight image windows in parallel. It also exploits parallelism within the windows. Convolutions, in general, are ideal for data parallelism. The multiplications can be done in parallel, while the additions are implemented as trees of parallel adders. Pipeline parallelism is equally important, since square root is a complex operation that leads to a long circuit (on an FPGA) or a complex series of instructions (on a Pentium). The SA-C compiler lays out the circuit for the complete edge operator including the square root, and then inserts registers to divide it into approximately twenty stages, each of which operate in parallel.

Finally, the SA-C compiler has the advantage of being a compiler, not a library of subroutines. Thus, while the IPL convolution routine must be general enough to perform arbitrary convolutions, the SA-C implementation of Prewitt can take advantage of compile-time constants. In particular, the Prewitt edge masks are composed entirely of ones, zeroes and minus ones, so all of the multiplications in these particular convolutions can be optimized away or replaced by negation. Furthermore, the eight windows being processed in parallel contain redundant additions, the extra copies of which are removed by common subexpression elimination.

The Canny edge detector is similar to Prewitt, only more complex. It smoothes the image and convolves it with horizontal and vertical edge masks. It then computes edge orientations as well as edge magnitudes, performs non-maximal suppression in the direction of the gradient, and applies high and low thresholds to the result. (A final connected components step was not implemented; see [12] pp. 76-80.)

We implemented the Canny operator in SA-C and executed it on the WildStar. The result was compared to two versions of Canny on the Pentium. The first version was written in VisualC++, using IPL routines for the convolutions. This allowed us to compare compiled SA-C on the WildStar to compiled C on the Pentium; the WildStar was 120 times faster. We then tested the hand-optimized assembly code version of Canny in Intel's OpenCV, setting the high and low thresholds equal to prevent the connected components routine from iterating. The Pentium's performance improved five fold, but the FPGA still outperformed the Pentium by a factor of 22. Table 1 shows the comparison with OpenCV.

Why is performance relatively better for Prewitt than for Canny? There are two reasons. First, the Canny operator uses fixed convolution masks, so the OpenCV Canny routine has the same opportunities for constant propagation and common subexpression elimination that SA-C has. Second, the Canny operator does not include a square root operation. Square roots can be pipelined on an FPGA but require multiple cycles on a Pentium.

5.3) Wavelet

In a test on the Cohen-Daubechies-Feauveau Wavelet [13], the WildStar beat the Pentium by a factor of 35. Here a SA-C implementation of the wavelet was compared to a C implementation provided by Honeywell as part of a reconfigurable computing benchmark [14].

5.4) The ARAGTAP Pre-screener

We also compared FPGAs and Pentiums on two military applications. The first is the ARAGTAP pre-screener [15], a morphology-based focus of attention mechanism for finding targets in SAR images. The pre-screener uses six morphological subroutines (downsample, erode, dilate, bitwise and, positive difference, and majority threshold), all of which were written in SA-C. Most of the computation in the pre-screener, however, is in a sequence of 8 erosion operators with alternating

masks, and a later sequence of 8 dilations. We therefore compared these sequences on FPGAs and Pentiums. (Just the dilate is shown in Table 1; results are similar for erosion).

The SA-C compiler fused all 8 dilations into a single pass over the image; it also applied temporal common subexpression elimination, pipelining, and other optimizations. The result was a 20 fold speed-up over the Pentium running automatically compiled (but heavily optimized) C. We had expected a greater speed-up, but were foiled by the simplicity of the dilation operator: after optimization, it reduces to a collection of max operators, and there is not enough computation per pixel to fully exploit the parallelism in the FPGA.

5.5) Probing

The second application is an ATR probing algorithm [16]. The goal of probing is to find a target in a LADAR or IR image. A target (as seen from a particular viewpoint) is represented by a set of probes, where a probe is a pair of pixels that straddle the silhouette of the target. The idea is that the difference in values between the pixels in a probe should exceed a threshold if there is a boundary between them. The match between a template and an image location is measured by the percentage of probes that straddle image boundaries.

Probe sets must be evaluated at every window position in the image. What makes this application complex is the number of probes. In our example there are approximately 35 probes per view, 81 views per target, and three targets. In total, the application defines 7,573 probes per window position. Fortunately, many of these probes are redundant in either space or time, and the SA-C optimizing compiler is able to reduce the problem to computing 400 unique probes (although the summation trees remain complex). This is still too large to fit on one FPGA, so to avoid dynamic reconfiguration we distribute the task across all three FPGAs on the WildStar. When compiled using VisualC++ for the Pentium, probing takes 65 seconds; the SA-C implementation on the WildStar run in 0.08 seconds.

For the probing algorithm, the WildStar is 800 times faster than the Pentium. This is true despite the fact that the WildStar runs (for this program) at 41.2 MHz, or approximately $1/20^{th}$ the speed of the Pentium. However, the optimizing compiler is able to reduce the total number of computations by a factor of about 19, using optimizations enabled by the predictable access patterns in SA-C. We also gain a factor of three by exploiting all three FPGA chips (something we do not do for the other applications). Most importantly, we gain a factor of approximately 400 over the Pentium in terms of fine-grained parallel operations.

Interestingly, the compiler does such a good job of exploiting the fine-grained parallelism (including pipelining), that probing is still I/O bound on the FPGA: it processes the probes so quickly (and in parallel with I/O) that the run-time is determined by the number of cycles needed to read and write the data.

6) Practical Timing Considerations

So far, we have reported only run-times. This is misleading, if the reconfigurable processor is being used as a co-processor. To run an operator on a co-processor, the image has to be downloaded to the reconfigurable systems' memory, and the results must be returned to the host processor. A typical upload or download time on a PCI bus for a 512x512 8-bit image is about 0.022 seconds. As a result, the FPGA is slower than a Pentium at adding a scalar to an image, if data communication times are taken into account. The other programs listed in Table 1 are still faster on the FPGA, although their speed-up factors are reduced.

In addition, current FPGAs cannot be reconfigured quickly. It takes about 0.14 seconds to reconfigure an XV-2000E over a PCI bus. Any function compiled to an FPGA configuration must save enough time to justify the reconfiguration cost. The simplest way to do this in practice is to select one operator sequence to accelerate per FPGA, and to pre-load the FPGAs with the appropriate configurations, thus eliminating the need for dynamic reconfiguration. However, non real time applications may find it useful to reconfigure and FPGA, so long as the benefit of each configuration is great enough.

7) Related Work

Researchers have tried to accelerate image processing on parallel computers for as long as there have been parallel computers. Some of this early work tried to map IP onto commercially available parallel processors (e.g. [17]), while other research focused on building special-purpose machines (e.g.

[18]). Unfortunately, in both cases the market did not support the architecture designs, which were eclipsed by general-purpose processors and Moore's law. More recent work has focused on so-called "vision chips" that build the sensor into the processor [19]. Another approach (advocated here) is to work at the board level and integrate existing chips – either DSPs or FPGAs -- into parallel processors that are appropriate for image processing. Focusing on FPGAs, Splash-2 [20] was the first reconfigurable processor based on commercially available FPGAs (Xilinx 4010s) and applied to image processing. The current state of the art in commercially available reconfigurable processors is represented by the AMS WildStar[5], the Nallatech Benblue[6] and the SLAAC project[7], all of which use Xilinx FPGAs. (The experimental results in this paper were computed on an AMS WildStar.) Research projects into new designs for reconfigurable computers include PipeRench [21], RAW [22] and Morphosis [23].

To exploit new hardware, researchers have to develop software libraries and/or programming languages. One of the most important software libraries is the Vector, Signal, and Image Processing Library (VSIPL)[8], proposed by a consortium of companies, universities and government laboratories as a single library to be supported by all manufacturers of image processing hardware. The Intel Image Processing Library (IPL)[9] and OpenCV[10] are similar libraries that map image processing and computer vision operators onto Pentiums. It is also possible to build graphical user interfaces (GUIs) to make using libraries easier. AMS provides such a GUI to a library of primitive operators for programming the WildStar; CHAMPION [24] uses the Khoros [25] GUI, having implemented all the primitive Khoros routines in VHDL. SA-C has also been integrated with Khoros, which can be used both to call pre-written SA-C routines or to write new ones.

One of the first programming languages designed to map image processing onto parallel hardware was Adapt [26]; C\\ [27] and C_T++ [28] are more recent languages designed for the same purpose. Other languages are less specialized and try to map arbitrary computations onto fine-grained parallel hardware; several of these projects focus on reconfigurable computing. Handel-C [29] is similar to SA-C in that it both extends C to express important hardware functionality, such as bit-widths, explicit timing parameters, and parallelism, and limits the language to exclude C features that do not lend themselves to hardware translation. It is lower-level than SA-C, however, in that programmers need to consider clock signals and other timing considerations explicitly. Streams-C [30] emphasizes streams to facilitate the expression of parallel processes. Finally, the MATCH project [31] uses MATLAB as its input language, while targeting reconfigurable processors.

8) Future Work

For many years, real-time applications on traditional processors had to be written in assembly code, because the code generated by compilers was not as efficient. We believe there is an analogous progression happening with VHDL and the SA-C compiler. At the moment, applications written directly in VHDL are more efficient (albeit more difficult to develop), but we expect future improvements to the compiler to narrow this gap.

In particular, the FPGA configurations generated by the SA-C compiler currently use only one clock signal. This limits the I/O ports to operate at the same speed as the computational circuit. Xilinx FPGAs, however, support multiple clocks running at different speeds, and include internal RAM blocks that can serve as data buffers. Future versions of the compiler will use two clocks, one for internal computation and one for I/O. This should double (or more) the speed of I/O bound applications.

We also plan to introduce streams into the SA-C language and compiler. This will support new FPGA boards with channels for direct sensor input, and will also make it easier to implement applications where the run-times of subroutines are strongly data dependent, for example connected components. We also plan to introduce new compiler optimizations to support trees and other complex data structures in memory, and to improve pipelining in the presence of nextified (a.k.a. loop carried) variables.

The goal of these extensions is to support stand-alone applications on FPGAs. Imagine, for example, reconfigurable processor boards with one or more FPGAs, local memories, A/D converters

[5] www.annapmicro.com
[6] www.nallatech.com
[7] www.east.isi.edu/SLAAC/
[8] www.vsipl.org
[9] www.intel.com/software/products/perflib/ipl/
[10] www.intel.com/software/products/opensource/libraries/cvfl.htm

(or digital camera ports), and internet access. Such processors could be incorporated inside a camera, and would consume very little power. A security application running on the FPGAs could then inspect images as they came from the camera, and notify users via the internet whenever something irregular occurred. The application could be as simple as motion detection or as complex as human face recognition. A single host processor could then monitor a large number of cameras/FPGAs from any location.

8) CONCLUSION

FPGAs are a class of processor with a two billion dollar per year market. As a result, they obey Moore's law, getting faster and denser at the same rate as other processors. The thesis of this paper is that most image processing applications run faster on FPGAs than on general-purpose processors, and that this will continue to be true as both types of processors become faster.

In particular, complex image processing applications do enough processing per pixel to be compute bound, rather than I/O bound. In such cases, FPGAs dramatically outperform Pentiums by factors of up to 800. Simpler image processing operators tend to be I/O bound. In these cases, FPGAs still outperform Pentiums because of their greater I/O capabilities, but by smaller margins (factors of 10 or less).

References

[1] A. DeHon, "The Density Advantage of Reconfigurable Computing," *IEEE Computer*, vol. 33, pp. 41-49, 2000.

[2] A. Benedetti and P. Perona, "Real-time 2-D Feature Detection on a Reconfigurable Computer," presented at IEEE Conference on Computer Vision and Pattern Recognition, Santa Barbara, CA, 1998.

[3] J. Woodfill and B. v. Herzen, "Real-Time Stereo Vision on the PARTS Reconfigurable Computer," presented at IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, 1997.

[4] D. Benitez and J. Cabrera, "Reactive Computer Vision System with Reconfigurable Architecture," presented at International Conference on Vision Systems, Las Palmas de Gran Canaria, 1999.

[5] R. W. Hartenstein, J. Becker, R. Kress, H. Reinig, and K. Schmidt, "A Reconfigurable Machine for Applications in Image and Video Compression," presented at Conference on Compression Technologies and Standards for Image and Video Compression, Amsterdam, 1995.

[6] J. G. Eldredge and B. L. Hutchings, "RRANN: A Hardware Implementation of the Backpropagation Algorithm Using Reconfigurable FPGAs," presented at IEEE International Conference on Neural Networks, Orlando, FL, 1994.

[7] N. Tredennick, "Moore's Law Shows No Mercy," in *Dynamic Silicon*, vol. 1: Gilder Publishing, LLC, 2001, pp. 1-8.

[8] J. P. Hammes, B. A. Draper, and A. P. W. Böhm, "Sassy: A Language and Optimizing Compiler for Image Processing on Reconfigurable Computing Systems," presented at International Conference on Vision Systems, Las Palmas de Gran Canaria, Spain, 1999.

[9] A. P. W. Böhm, J. Hammes, B. A. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar, "Mapping a Single Assignment Programming Language to Reconfigurable Systems," *Supercomputing*, vol. 21, pp. 117-130, 2002.

[10] J. B. Dennis, "The evolution of 'static' dataflow architecture," in *Advanced Topics in Data-Flow Computing*, J. L. Gaudiot and L. Bic, Eds.: Prentice-Hall, 1991.

[11] J. Hammes, W. Bohm, C. Ross, M. Chawathe, B. Draper, R. Rinker, and W. Najjar, "Loop Fusion and Temporal Common Subexpression Elimination in Window-based Loops," presented at 8th Reconfigurable Architectures Workshop, San Francisco, 2001.

[12] E. Trucco and A. Verri, *Introductory Techniques for 3-D Computer Vision*. Saddle River, NJ: Prentice-Hall, 1998.

[13] A. Cohen, I. Daubechies, and J. C. Feauveau, "Biorthogonal bases of compactly supported wavelets," *Communications of Pure and Applied Mathematics*, vol. 45, pp. 485-560, 1992.

[14] S. Kumar, "A Benchmark Suite for Evaluating Configurable Computing Systems - Status, Reflections, and Future Directions," presented at International SYmposium on FPGAs, Monterey, CA, 2000.

[15] S. D. Raney, A. R. Nowicki, J. N. Record, and M. E. Justice, "ARAGTAP ATR system overview," presented at Theater Missile Defense 1993 National Fire Control Symposium, Boulder, CO, 1993.

[16] J. E. Bevington, "Laser Radar ATR Algorithms: Phase III Final Report," Alliant Techsystems, Inc. May 1992.
[17] P. J. Narayanan, L. T. Chen, and L. S. Davis, "Effective Use of SIMD Parallelism in Low- and Intermediate-Level Vision," *IEEE Computer*, vol. 25, pp. 68-73, 1992.
[18] C. C. Weems, E. M. Riseman, and A. R. Hanson, "Image Understanding Architecture: Exploiting Potential Parallelism in Machine Vision," *IEEE Computer*, vol. 25, pp. 65-68, 1992.
[19] A. Moini, *Vision Chips*, vol. 526. Boston: Kluwer, 1999.
[20] D. Buell, J. Arnold, and W. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*: IEEE CS Press, 1996.
[21] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "PipeRench: A Coprocessor for Streaming Multimedia Acceleration," presented at International Symposium on Computer Architecture, 1999.
[22] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to Software: RAW Machines," *IEEE Computer*, vol. 30, pp. 86-93, 1997.
[23] G. Lu, H. Singh, M. Lee, N. Bagherzadeh, and F. Kurhadi, "The Morphosis Parallel Reconfigurable System," presented at EuroPar, 1999.
[24] S. Natarajan, B. Levine, C. Tan, D. Newport, and D. Bouldin, "Automatic Mapping of Khoros-based Applications to Adaptive Computing Systems," University of Tennessee 1999.
[25] K. Konstantinides and J. Rasure, "The Khoros Software Development Environment for Image and Signal Processing," *IEEE Transactions on Image Processing*, vol. 3, pp. 243-252, 1994.
[26] J. A. Webb, "Steps Toward Architecture-Independent Image Processing," *IEEE Computer*, vol. 25, pp. 21-31, 1992.
[27] A. Fatni, D. Houzet, and J. Basille, " The C\\ Data Parallel Language on a Shared Memory Multiprocessor.," presented at Computer Architectures for Machine Perception, Cambridge, MA, 1997.
[28] F. Bodin, H. Essafi, and M. Pic, " A Specific Compilation Scheme for Image Processing Architecture.," presented at Computer Architecture for Machine Perception, Cambridge, MA, 1997.
[29] O. H. C. Group, "The Handel Language," Oxford University 1997.
[30] M. Gokhale, "Stream Oriented FPGA Computing in Streams-C," presented at IEEE Symposium on Field-Programmable Custom Computing Machines, Nap, CA, 2000.
[31] P. Banerjee, "A MATLAB Compiler for Distributed, Hetergeneous, Reconfigurable Computing Systems," presented at IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, 2000.