Implementing Image Applications on FPGAs[1]

Bruce A. Draper, J. Ross Beveridge, A.P. Willem Böhm, Charles Ross, Monica Chawathe
Department of Computer Science
Colorado State University
Fort Collins, CO 80523 U.S.A.

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

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 a grid of wires with programmable connections. Each logic block has one or more lookup tables 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 were originally developed to serve as test vehicles for hardware circuit designs, since any circuit can be converted into an FPGA configuration and tested. Recently, however, FPGAs have become so dense that they have evolved from simple test and “glue logic” circuits into the central processors of powerful reconfigurable computing systems [1]. In computer vision and image processing, FPGAs have been used for 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 that of other parallel architectures. Because of the comparatively small size of the computer vision and 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.

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 laborious process of modifying or combining FPGA circuits repeatedly frustrates the intrepid few who do.

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 algorithms, and an optimizing SA-C 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 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 provide guidelines for when FPGAs will accelerate specific applications.

2. SA-C

SA-C is a single-assignment variant 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

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

types (e.g. int12) and fixed point data types (e.g. fix12.4). This exploits the ability of FPGAs to form arbitrary bit precision logic circuits, and compensates for the high cost of floating point operations on FPGAs. 2) SA-C includes extensions to C that provide parallel looping mechanisms and true multi-dimensional arrays. These extensions make it easier to express sliding windows or slices of data, 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 reflects a model in which variables correspond to wires in a circuit rather than memory addresses.

3. 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 [7]. 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 [8].

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

4. IMAGE PROCESSING

The SA-C language and compiler allow FPGAs to be programmed in the same way as any other processor. Programs are written in a high-level language, compiled and debugged on a local workstation, and then downloaded to a reconfigurable processor. SA-C therefore makes reconfigurable processors accessible to applications programmers with no hardware expertise.

The question is thus raised, do image processing tasks run faster on FPGAs than on conventional hardware? Tests presented below suggest the answer is yes, and that the more complex the task, the better the FPGA does relative to conventional hardware. The FPGA used in our tests is an Annapolis Microsystems WildStar reconfigurable processor containing 3 Xilinx XCV-2000E FPGAs and 12 local memories. Our conventional processor is a Pentium III running at 800 MHz. Both chips are of a similar age and were the first of their respective classes fabricated at 0.18 microns.

4.1) Simple Image Operators

The simplest program in our benchmark suite simply adds a scalar to every pixel in an image. We wrote the function in SA-C, and compiled it to a single FPGA on the WildStar. 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 circuit, but in general they run at lower clock rates than Pentiums. For this program, the WildStar ran at 51.7 MHz, vs. 800 MHz for the Pentium. However, FPGAs do more work per cycle than Pentiums. In the case of scalar addition, there is very little work to do per pixel; the program is I/O bound. Because the WildStar gives every FPGA access to four local memories, however, the FPGA can both read and write 8 bytes per cycle. The SA-C compiler exploits this and the internal parallelism on an FPGA to create a three-stage pipeline. On every cycle, the FPGA (1) reads eight 8-bit pixels, (2) adds a scalar to the eight pixels read on the previous cycle, and (3) writes back to memory the pixel and scalar sums of 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. The FPGA is able to outperform the Pentium by a factor of 8 because of its greater bandwidth to memory and the ability to implement in parallel (and in a single cycle) the simple pixel-based operation. This program in no way fully exploits the FPGA, however, since only 9% of its lookup tables (and 9% of its flip-flops) were used.

The Prewitt edge operator is more complex. Every 3x3 window in the image is convolved with horizontal and vertical edge masks; the edge magnitude is the square root of the sum of the square of the responses. This program does much more work per output pixel, which SA-C exploits both by data parallelism (for the convolutions) and pipeline parallelism (for the square root). The sliding window also implies that most input pixels would be read three times in a naïve implementation. The SA-C compiler avoids this by partially unrolling the loop to compute several windows (arranged vertically) in parallel. This reduces the number of reads per pixel. The final result is that the WildStar outperforms the Pentium by a factor of 80.

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 orientation as well as edge magnitude, performs non-maximal suppression in the direction of the gradient, and then applies high and low thresholds to the result. (A final connected components step was not implemented.)

We compared the Canny operator in SA-C 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 of Canny in Intel's OpenCV, setting the high and low thresholds to be the same 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.

In a test on the Cohen-Daubechies-Feauveau Wavelet [9], 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.

4.2) Larger Applications

We also compared FPGAs and Pentiums on two military applications. The first is the ARAGTAP pre-screener [10], 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.)

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 is a 20x speed-up over the Pentium running automatically compiled (but heavily optimized) C. We had expected more, but were foiled by the simplicity of the dilation operator: it reduces to a collection of max operators, and does not have enough computation per pixel to fully exploit the FPGA.

Routine	**Pentium III**	**XCV-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 XCV-2000E FPGAs.

The second application is an ATR probing algorithm [11]. The goal of probing is to find a target in a LADAR or IR image. Targets (as seen from particular viewpoints) are represented by sets of probes, where a probe is a pair of pixels that straddle the silhouette of the target. If the target appears in the image, the difference in values between the probe pixels should exceed a threshold. The match between a template and an image window is measured by the percent of probes whose difference exceeds the threshold.

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). 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, despite running at 41.2 MHz, or about 1/20th the frequency of the Pentium. Some of the gain comes from the optimizing compiler which reduces the total number of operations by a factor of 25. Some of this is through optimizations enabled by the predictable access patterns in SA-C, and some results from the greater internal memory capacity of the FPGA, which reduces the number of reads and writes to memory or cache. The Pentium has to execute 60 billion instructions to implement probing, while the FPGA only performs about 2.4 billion operations. More importantly, the FPGAs on the WildStar execute over 4,000 operations per cycle, while the Pentium's superscalar architecture averages 1.15 instructions per cycle for this task. Interestingly, probing on FPGAs is I/O bound: the run-time is determined by the number of cycles needed to read and write the data; the computational loop body executes on only 15% (2/13) cycles. The rest of the time it is waiting for input.

4.3) Other Timing Considerations

To run an operator on an FPGA, the image has to be downloaded to local memory, and the results must be returned to the host processor. A typical upload or download time over a PCI bus for a 512x512 8-bit image is about 0.022 seconds. As a result, the FPGA is slower than a Pentium for 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 XCV-2000E over a PCI bus. Depending on the application, it may or may not be feasible to dynamically reconfigure an FPGA in midstream.

7. CONCLUSION

FPGAs are well suited to real-time image processing applications where there is the opportunity to exploit data parallelism. (It is hard to exploit the capabilities of an FPGA with pipeline and process parallelism alone.) With our compiler, it seems that almost all image operations are significantly faster on FPGAs than general purpose processors, and that this should remain true for as long as both types of processors obey Moore's law.

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," 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," IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, 1997.

[4] D. Benitez and J. Cabrera, "Reactive Computer Vision System with Recon-figurable Architecture," 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 Vid-eo Compression," Conference on Compress-ion 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," IEEE International Conference on Neural Networks, Orlando, FL, 1994.

[7] 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.

[8] J. Hammes, W. Bohm, C. Ross, M. Cha-wathe, B. Draper, R. Rinker, and W. Najjar, "Loop Fusion and Temporal Common Sub-expression Elimination in Window-based Loops," 8th Reconfigurable Architetcures Workshop, San Francisco, 2001.

[9] 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.

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

[11] J. E. Bevington, "Laser Radar ATR Algorithms: Phase III Final Report," Alliant Techsystems, Inc. May 1992.