Accelerated Image Processing 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.

draper,ross,bohm,rossc,chawathe@cs.colostate.edu

ABSTRACT

The Cameron project has developed a language called Single Assignment C (SA-C), and a compiler for mapping image-based applications written in SA-C to field programmable gate arrays (FPGAs). This paper tests this technology by implementing several applications in SA-C and compiling them to an Annapolis Microsystems (AMS) WildStar board with a Xilinx XV2000E FPGA. The performance of these applications on the FPGA is compared to the performance of the same applications written in assembly code or C for an 800MHz Pentium III. (Although no comparison across processors is perfect, these chips were the first of their respective classes fabricated at 0.18 microns, and are therefore of comparable ages.) We find that applications written in SA-C and compiled to FPGAs are between 8 and 800 times faster than the equivalent program run on the Pentium III.

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

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

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 primarily out 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). Logic blocks can be connected into circuits of arbitrary complexity by using the programmable wires to route the outputs of logic blocks to the input of others. FPGAs as a whole can therefore implement circuit diagrams, by mapping the gates and registers onto logic blocks. The achievable clock rate of an FPGA configuration depends on the depth of the computation in terms of logic blocks, and their relative placement, which determines the length of the wires needed to connect them.

Modern FPGAs are actually more complex than the discussion above might imply. All FPGAs have special purpose I/O blocks that communicate with external pins. Many have on-chip memory in the form of RAM blocks. Others have multipliers or even complete RISC processors in addition to general purpose logic blocks. In general, however, we will stick to the simplified view of an FPGA as a set of logic blocks connected by programmable wires, because this is the model used by the SA-C compiler.

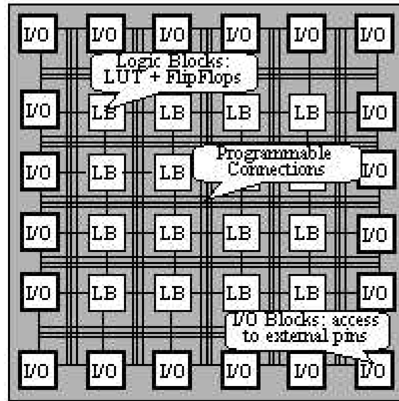

Figure 1: A conceptual illustration of an FPGA. Every logic block contains one or more LUTs, plus a bit or two of memory. The contents of the LUTs are (re)programmable, as are the grid connections. I/O blocks provide access to external pins, which usually connect to local memories.

FPGAs serve as "glue logic" between off the shelf components, and as replacements for ASICs in first generation products. Recently, however, FPGAs have become so dense and fast that they have evolved into the central processors of powerful reconfigurable computing systems [1]. A Xilinx XCV-2000E, for example, contains 38,400 logic blocks, and can operate at speeds of up to 180MHz. The logic blocks can be configured so as to exploit data, pipeline, process, or 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], and video and image compression [5].

The economics of FPGAs are fundamentally different from the economics of other parallel 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 [6].

Unfortunately, FPGAs are generally programmed in hardware design languages, such as VHDL. This excludes the vast majority of software developers who do not know circuit design. In addition, (synthesizable subsets of) hardware languages force implementers to focus on details such as synchronization and timing. Even new experimental languages such as Celoxica's Handel-C still require the programmer to consider timing information (see related work). Just as important, given that simulators are too slow to emulate more than a tiny fraction of time, debugging all too often occurs at the circuit level.

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 [7, 8],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 800MHz Pentium III. 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)[2], 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 is 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

[2] The terminology and syntax of array manipulation is borrowed from Fortran-90.

construct circuits with any precision[3]. 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.

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 using the array statement. 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 [7] or at the Cameron web site.

4) THE SA-C COMPILER

The SA-C compiler translates high-level SA-C code into host code and FPGA configurations. The underlying model is that a reconfigurable processor is available to the host for use as a co-processor. The SA-C compiler divides the source program into host code and code destined for the FPGA. Sequential statements outside of any loop are translated into C and compiled using a standard C compiler for the host. Parallel loops, which typically process images, are translated into FPGA configurations. The compiler inserts into the host C program all the code necessary for downloading the FPGA configuration, image data, and program parameters to the reconfigurable processor, and for uploading the results, as shown in Figure 3.

---

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

For simplicity and retargetability, the model of the reconfigurable processor is kept simple. The reconfigurable processor is assumed to have local memories, one or more FPGAs, and a PCI bus connection. The FPGAs are assumed to have I/O blocks and a grid of reprogrammable logic blocks with reprogrammable connections. So far, the SA-C compiler has targeted a Annapolis Microsystems (AMS) WildForce processor with five Xilinx XC-4036 FPGAs, an AMS StarFire with a single Xilinx XVC-1000 FPGA, and an AMS WildStar with three XVC-2000E FPGAs. The results reported in this paper are with the AMS WildStar.

Every loop on the FPGA consists of a generator module, a loop body, and a collector module. The generator module implements the sliding window operation driving a SA-C loop using a two dimensional block of shift registers, one row of registers for each row of the SA-C window[4]. All SA-C generators (element, window, and vector) are implemented in this fashion. The current implementation restricts this to fixed sized windows where the window size is known at compile time.

The shifts occur on a word basis. For each loop iteration, selectors retrieve the appropriate bitfields from the block of shift registers and send them into the loop body. The loop body is divided into pipelined sections to increase the achievable clock rate. Reads from memory within the loop body are synchronized by an arbitrator, with one arbitrator per memory. Handshaking among the arbitrators and pipeline stages is used to guarantee the validity of the computation. The collector module receives loop body results and, for each loop body result, fills a word buffer, which is written to memory when full.

The SA-C compiler optimizes the loops it maps onto FPGAs. It fully unrolls loops anytime the number of iterations through the loop can be determined at compile-time. Full unrolling of loops is important when generating code for FPGAs, because it spreads the iterations in code space rather than in time. Array value propagation searches for array references with constant indices, and replaces such references with the values of the array elements. When the value is a compile time constant, this enables constant propagation. A loop can also be partially unrolled under the control of a pragma. When the result of one loops feeds another, the loops can be fused, avoiding the creation of intermediate data and enlarging the loop body.

[4] Iterations over N-dimensional arrays use N-dimensional blocks of shift registers.

Common sub-expression elimination (CSE) is a well known compiler optimization that eliminates redundancies by looking for identical sub-expressions that compute the same value. Traditional CSE could be called ``spatial CSE'' since it looks for common sub-expressions within a block of code. The SA-C compiler not only performs spatial CSE, but also Temporal CSE, looking for values computed in one loop iteration that were previously computed in another. In such cases, redundant computations are eliminated by holding values in register chains.

Some operators, such as division, can be inefficient to implement in logic gates, and are better implemented as lookup tables. A pragma indicates that a function or an array should be compiled as a lookup table. Bitwidth narrowing exploits user-defined bitwidths of variables to infer the minimal bitwidths of intermediate variables.

The optimizations, some controlled by user pragmas, transform the program into an equivalent program with a new set of inner-loops optimized for FPGAs. These inner-loops are compiled to VHDL. Commercial software is used to synthesize and place-and-route the VHDL, producing FPGA configurations. The configurations and their execution frequencies are recorded. At run-time, when a particular inner-loop is to be executed on the reconfigurable processor, the host interface module downloads the configuration and input data via DMA, sets the clock frequency, executes the inner-loop, and retrieves the data from the reconfigurable processor.

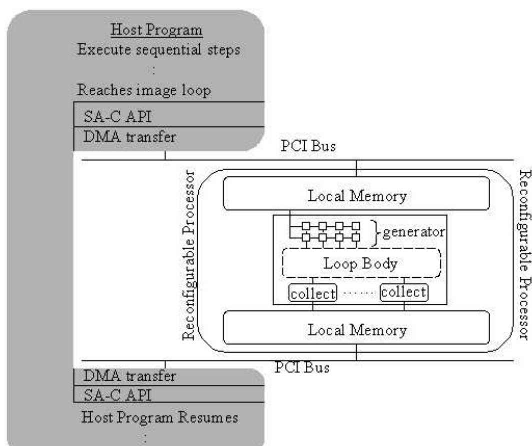

Figure 3: The SA-C compiler divides the souce program into sequential host code and loops to be executed on the FPGA. Host code is translated to C and compiled for the host. Loops are compiled for the FPGA, with loop generators, bodies and data collectors. All code for downloading and uploading data it generated by the compiler and automatically inserted into the host code.

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, so long as the workstation has access to a reconfigurable processor. 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 written in SA-C and executed on FPGAs are faster than the equivalent applications on conventional general-purpose hardware, in particular Pentiums. The tests presented in Table 1 suggest that in general, the answer is yes. Simple image operators are faster on FPGAs because of their greater I/O bandwidth to 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 experiment, 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. Each FPGA has 38,400 4x1-bit lookup tables (LUTs) and 38,400 flip-flops. These are organized into 19,200 "slices", each of which contains two LUTs and two flip-flops. Slices are the XV-2000E version of the more generic term "logic block" used in Figure 1. Our conventional processor is a Pentium III running at 800 MHz with 256KBytes of primary cache. The chips in both processors are of a similar age and were the first of their respective classes fabricated at 0.18 microns.

The images used are 512x512 images of 8-bit pixels. Because all six applications are based on sliding windows where the height of the window is 13 rows or less, the primary cache on the Pentium is large enough to guarantee that the only primary cache misses occur the first time a data element is accessed. Table 2 shows the

chip resources used by the programs in the test suite. The probing application employs all three FPGAs on the WildStar; the other applications use only a single chip.

The execution times reported in this section were collected as follows. For the FPGA, the image data has already been downloaded to the local memory of the reconfigurable processor. We time how long it takes for the FPGA to read the image from memory, process it, and write the result back to the local memory of the reconfigurable processor. We do not count the time required to transfer data to/from the host to the memory of the reconfigurable processor (this data is given in Section 6 and Table 3). Equivalently, for the Pentium we time how long it takes to read the image from the local memory of the host, process it, and write it back to the local memory of the host.

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.

Routine	LUTs (%)	FFs (%)	Slices (%)
AddS	9	9	16
Prewitt	18	13	28
Canny	48	45	87
Wavelet	54	69	99
Dilates (8)	56	56	97
Probing (chip 1)	33	39	65
Probing (chip 2)	36	41	72
Probing (chip 3)	42	49	85

Table 2. FPGA resource use in percentages for the routines shown in Table 1: LUTs (4x1 look-up tables), FFs (flip-flops) and Slices (blocks of 2 LUTs and 2 FFs). A slice is "used" is any of its resources are used.

5.1) Scalar Addition

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. For the Pentium, we used the iplAddS routine from the Intel Image Processing Library (IPL, release 2.5). In so doing, we compare the performance of compiled SA-C code on an FPGA to hand-optimized Pentium assembly code. As shown in Table 1, the WildStar outperforms the Pentium by a factor of 8.

Why is the WildStar faster? For this application, the WildStar runs 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. Since each image pixel is used only once, this application on the Pentium runs at memory speed, not at cache speed.

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. 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 no cycles are wasted. On every cycle, 8 pixels are read, 8 additions are performed (on pixels read in the previous cycle) and 8 pixels are written.

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) are used. However, it demonstrates that FPGAs will outperform Pentiums on simple image operators because of their pipelining and parallel I/O capabilities.

5.2) Prewitt & Canny

The Prewitt edge operator shown in Figure 2 is more complex than simple scalar addition. Every 3x3 image window 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. The Prewitt program written in SA-C and compiled for the WildStar computes the edge magnitude response image for a 512x512 8-bit input image in 1.96 milliseconds. In comparison, the equivalent Intel IPL function[5] on the Pentium takes 158.08 milliseconds, or approximately 80 times longer (see Table 1).

Why is the Prewitt edge detector faster on an FPGA? One of the reasons, as before, is I/O, although the effect is less. As written, the Prewitt operator reads only four pixels per cycle, not eight. Also, the Pentium now gets a performance benefit from its cache, because the input image is small enough for three rows to be kept in primary cache, and therefore pixels do not have to be read from memory three times.

---

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

FPGAs do not have caches, but the SA-C compiler implements an optimization that minimizes rereads. A naïve implementation of a 3x3 window would slide the window horizontally across the image until it reaches the end of the row, and then drop down one row and repeat the process. While the shift registers in the loop generator avoid reading a pixel more than once on any given horizontal sweep, in a naïve implementation every pixel would still be 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 4). This reduces the number of input operations needed to process the image by almost a factor of three, and is an example of a general optimization technique called partial loop unrolling that is used to optimize I/O over N-dimensional arrays.

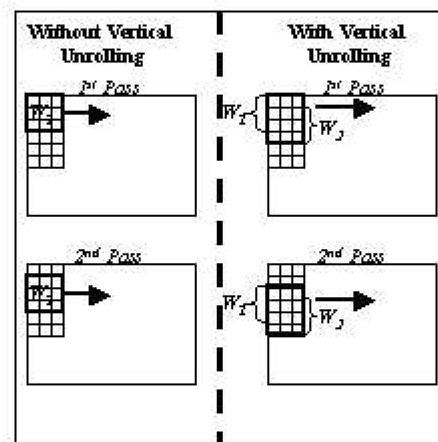

Figure 4: How partial vertical unrolling optimizes I/O. As a 3x3 image window slides across an image, each pixel is read 3 times (assuming shift registers hold values as it slides horizontally). Under partial unrolling, two or more vertical windows are computed in parallel, allowing the passes to skip 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 pipeline stages.

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 are 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 [9] 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++ (version 6.0, fully optimized), using IPL routines to perform 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 (version 001), setting the high and low thresholds equal to prevent the connected components routine from iterating. The Pentium's performance improved five fold in comparison to the C++ version, 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 [10], 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 [11]. In this case, we attribute the speed difference to I/O differences and SA-C's temporal CSE optimization. Honeywell's algorithm makes two passes; a 5x1 mask creates two intermediate images from the source image, and a second pass with a 1x5 mask over the intermediate images creates the four final images. The SA-C algorithm, on the other hand, makes a single pass over the source image with a 5x5 mask, writing all four output images in parallel. In addition, the first and last columns of the 5x5 mask are the same. The intermediate values computed for the right column are the same as the values computed four cycles earlier for the left column. This allows for temporal subexpression elimination. (The second and fourth columns are also the same.) In addition, the SA-C compiler partially unrolls the 5x5 loop 8 times to minimize redundant read operations.

5.4) The ARAGTAP Pre-screener

We also compared FPGAs and Pentiums on two military applications. The first is the ARAGTAP pre-screener [12], 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, 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 [13]. The goal of probing is to find a target in a 12-bit per pixel 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 13x34 window. 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 explicitly distribute the task across all three FPGAs on the WildStar. This is done by writing three loops, one for each vehicle, and using a pragma to direct each loop to one FPGA. It is worth noting that methods have been proposed for automatically partitioning applications across FPGAs when the applications are specified as a task graph [14] (see also [15]).

When compiled using VisualC++ for the Pentium, probing takes 65 seconds on a 512x1024 12-bit image; the SA-C implementation on the WildStar runs in 0.08 seconds.

These times can be explained as follows. For the probing algorithm generated by the SA-C compiler, the FPGAs run at 41.1 MHz. The program is completely memory IO bound: on every clock cycle each FPGA reads one 32 bit word, containing two 12 bit pixels. As there are $(512-13+1)\times(1024)\times 13$ pixel columns to be read, the FPGAs perform $(512-13+1) \times (1024) \times (13/2) = 3{,}328{,}000$ reads. At 41.1 MHz this takes 80.8 milliseconds.

The Pentium performs $(512-13+1) \times (1024-34+1)$ window accesses. Each of these window accesses involves 7573 threshold operations. Hence the inner loop that performs one threshold operation is executed $(512-13+1) \times (1024-34+1) \times 7573 = 3{,}752{,}421{,}500$ times. The inner loop body in C is:

```
for(j=0; j<sizes[i]; j++){
  diff = ptr[set[i][j][2]*in_width+set[i][j][3]] –
         ptr[set[i][j][0]*in_width+set[i][j][1]];
  count += (diff>THRESH || diff<-THRESH);
}
```

where in_width and THRESH are constants. The VC++ compiler infers that ALL the accesses to the set array can be done by pointer increments, and generates an inner loop body of 16 instructions. (This is, by the way, much better than the 22 instructions that the gcc compiler produces at optimization setting -O6.) The total number of instructions executed in the inner loop is therefore $3{,}752{,}421{,}500 \times 16 = 60{,}038{,}744{,}000$. If one instruction were executed per cycle, this would result in an execution time of 75 seconds. As the execution time of the whole program is 65 seconds, the Pentium (a super scalar architecture) is actually executing more than one instruction per cycle!

6) Co-processor Timing Considerations

In Section 5, we reported run times that measured how long it took an FPGA to read data from local memory, process it, and write it back to local memory. We believe that these are the most relevant numbers, since the trend is toward putting RISC processors on FPGA chips, thereby eliminating the time needed to transfer data between the host and reconfigurable processor. The Virtex II Pro from Xilinx is an example of such a combined processor. There are also reconfigurable processors with direct camera connections, eliminating data download times.

Nonetheless, the configuration tested here and shown in Figure 3 has a separate reconfigurable co-processor connected to a host via a PCI bus. To execute an operator on the co-processor, the image must be downloaded to the reconfigurable system's 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 0.019 seconds. In all of the applications except dilation and probing, the output image has more output pixels than the source image, typically doubling the time to upload the results. For probing, the image must be downloaded three times (once for each FPGA), and since it creates two result images, a total of six uploads are required. When upload and download times are included, the FPGA is slower than a Pentium at scalar addition and wavelet decomposition, as shown in Table 3. The other applications tested are faster on the FPGA, but the performance ratios are very small except for probing. This suggests that FPGAs should only be used as co-processors for very large applications, although they are almost always faster for image computation if data transfer times can be eliminated.

Routine	**Pentium III**	**XV-2000E**	**Ratio**
AddS	0.00595	0.05206	0.11
Prewitt	0.15808	0.05072	3.12
Canny	0.13500	0.06091	2.22
Wavelet	0.07708	0.09258	0.83
Dilates (8)	0.06740	0.05923	1.14
Probing	65.0	0.78	83.3

Table 3. Execution times in seconds for routines with 512x512 8-bit input images, when data transfer times between the host and reconfigurable processor are included. In all cases except dilation and probing, the bit resolution of the result is larger than the source, increasing upload times.

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.

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. [16]), while other research focused on building special-purpose machines (e.g. [17]). 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 [18]. 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 [19] 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[6], the Nallatech Benblue[7] and the SLAAC project[8], 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 [20], RAW [21] and Morphosis [22].

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)[9], 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)[10] and OpenCV[11] 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. CHAMPION [23] uses

---

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

the Khoros [24] 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 [25]; C\\ [26] and C_T++ [27] 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.

DSP-C, a programming language for Digital Signal Processors, is a sequential extension of ISO C, developed and supported by Adelante[12]. It has fixed point numbers of user specifiable sizes, and pointer arithmetic for efficiently implementing circular buffers.

Other languages have specifically targeted FPGAs. Handel-C [28], originally from Oxford University and now further developed by Celoxica, is a C derived language with explicit Occam type sequential and parallel regions. The boundaries of these regions form barriers. This provides an explicit timing model in the language. Handel-C has variables with user-defined bitwidths similar to SA-C. Streams-C [29] emphasizes streams to facilitate the expression of parallel processes. System C[13] is a C++ class library that allows system simulation and circuit design in a C++ programming environment. Finally, the MATCH project [30] uses MATLAB as its input language, while targeting reconfigurable processors.

Various vendors provide macro dataflow design packages, e.g. CoreFire from Annapolis Micro Systems, and Viva from Starbridge. In addition, embedded systems design kits provide a frame work for modules to be connected together and interfaced on an FPGA, where the modules is programmed in VHDL by the user.

---

[11] www.intel.com/software/products/opensource/libraries/cvfl.htm
[12] www.adelantetech.com
[13] www.systemc.org

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 high level algorithmic languages for FPGAs. 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 (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, and may merge in future systems on a chip.

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

[7] 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.

[8] 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.

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

[10] 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.

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

[12] 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.

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

[14] M. Kaul, R. Vemuri, S. Govindarajan, and I. E. Ouaiss, "An Automated Temporal Partitioning and Loop Fission for FGPA-based Reconfigurable Synthesis of DSP applications," presented at 36th Design Automation Conference, New Orleans, LA, 1999.

[15] R. D. Hudson, D. Lehn, J. Hess, J. Atwell, D. Moye, K. Shiring, and P. M. Athanas, "Spatio-temporal Partitioning of Computational Structures onto Configurable Computing Machines," presented at SPIE, Bellingham, WA, 1998.

[16] 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.

[17] 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.

[18] A. Moini, *Vision Chips*, vol. 526. Boston: Kluwer, 1999.

[19] D. Buell, J. Arnold, and W. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*: IEEE CS Press, 1996.

[20] 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.

[21] 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.

[22] G. Lu, H. Singh, M. Lee, N. Bagherzadeh, and F. Kurhadi, "The Morphosis Parallel Reconfigurable System," presented at EuroPar, 1999.

[23] 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.

[24] 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.

[25] J. A. Webb, "Steps Toward Architecture-Independent Image Processing," *IEEE Computer*, vol. 25, pp. 21-31, 1992.

[26] 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.

[27] 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.

[28] O. H. C. Group, "The Handel Language," Oxford University 1997.

[29] M. Gokhale, "Stream Oriented FPGA Computing in Streams-C," presented at IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, 2000.

[30] P. Banerjee, "A MATLAB Compiler for Distributed, Hetergeneous, Reconfigurable Computing Systems," presented at IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, 2000.