Int. Conf. on Vision Systems, Vancouver, July 7-8, 2001. p. 220-235.

Compiling SA-C Programs to FPGAs: Performance Results

Bruce A. Draper[1], A.P. Willem Böhm[1], Jeff Hammes[1], Walid Najjar[2], J. Ross Beveridge[1], Charlie Ross[1], Monica Chawathe[1], Mitesh Desai[1], José Bins[3]

[1] Department of Computer Science
Colorado State University
Fort Collins, CO 80523, USA
draper,bohm,…@cs.colostate.edu
[2] Department of Computer Science,
University of California at Riverside
Riverside, CA 92521-0304, USA
najjar@cs.ucr.edu
[3] Faculdade de Informática
Pontifícia Universidade Católica (RS)
Porto Alegre, RS 90619-900, Brazil
bins@inf.pucrs.br

Abstract. At the first ICVS, we presented SA-C ("sassy"), a single-assignment variant of the C programming language designed to exploit both coarse-grain and fine-grain parallelism in computer vision and image processing applications. This paper presents a new optimizing compiler that maps SA-C source code onto field programmable gate array (FPGA) configurations. The compiler allows programmers to exploit FPGAs as inexpensive and massively parallel processors by writing high-level source code rather than hardware-level circuit designs. We present several examples of simple image-based programs and the optimizations that are automatically applied to them during compilation, and compare their performance on FPGAs and Pentiums of similar ages. From this, we determine what types of applications benefit from current FPGA technology, and conclude with some speculations on the future development of FPGAs and their expanding role in computer vision systems.

1 Introduction

Over the past several years, field-programmable gate arrays (FPGAs) have evolved from simple "glue logic" circuits into the central processors of reconfigurable computing systems [1]. For readers who are not familiar with FPGAs, they are grids of

reprogrammable logic blocks, connected by reconfigurable networks of wires. For example, the Xilinx XV-1000 Virtex FPGA contains 12,288 logic blocks, each containing a four-bit lookup table (LUT) and two bits of memory. Additional circuitry allows pairs of logic blocks to be configured as 5-bit LUTs. The reconfigurable network on the Virtex is essentially a mesh, with one router for every four logic blocks. Reconfigurable computing systems combine one or more FPGAs with local memories and bus connections to create reconfigurable co-processors. An example of such a coprocessor is the Annapolis Microsystems (AMS) Starfire, which has a Xilinx XV-1000 Virtex chip, six 1 MBytes local memories and a PCI interface.

The economics of FPGAs are fundamentally different from that of other parallel architectures proposed for computer vision. Because of the comparatively small size of the computer vision market, most special-purpose vision processors are unable to keep up 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.

Recently, the computer vision and image processing communities have become aware of the potential for massive parallelism and high computational density in FPGAs. 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]. Unfortunately, to exploit FPGAs programs have to be implemented as circuits in a hardware description language such as VHDL or Verilog. This has discouraged many researchers from exploiting FPGAs. The intrepid few who do are repeatedly frustrated by the process of modifying or combining complex logic circuits.

The goal of the Cameron project at Colorado State University is to change how reconfigurable systems are programmed from a hardware-oriented circuit design paradigm to a software-oriented algorithmic one. To this end, we have developed a high-level language (called SA-C) for expressing computer vision and image processing 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. While the resulting run-times are still greater than the run-times for hand-coded logic circuits, there is a tremendous gain in programmer productivity. Moreover, because SA-C is a variant of C, it makes FPGAs accessible to the majority of programmers who are not skilled in circuit design.

This paper briefly reviews SA-C and then describes the compiler that maps SA-C programs onto FPGA configurations. We compare the run-times of SA-C programs on an AMS Starfire to equivalent programs from Intel's Image Processing Library (IPL) running on a 450 MHz Pentium II. (These machines are approximately the same age.) Based on these comparisons, we draw conclusions about what types of applications can benefit from this technology, and what future advancements in FPGA technology might make them more broadly interesting to computer vision researchers.

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 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 providing parallel looping mechanisms and true multi-dimensional arrays. These extensions make it easier for programmers to operate on sliding windows or slices of data, and also make it easier for the compiler to identify and optimize common data access patterns. 3) SA-C restricts C by outlawing pointers and recursion, and restricting variables to be single assignment. This prevents programmers from applying memory models that do not map well onto FPGAs. A full description of the SA-C language can be found in [7], while a reference manual can be found at http://www.cs.colostate.edu/cameron.

```
uint11[:,:] prewitt(uint8 Image[:,:])
  // Declare constant Prewitt masks
  int2 H[3,3] = {  {-1, 0, 1},
                   {-1, 0, 1},
                   {-1, 0, 1}} ;
  int2 V[3,3] = {  {1, 1, 1},
                   {0, 0, 0},
                   {-1, -1, -1}} ;
  // Compute mask responses and edge magnitude
  uint11 R[:,:] =
      for window W[3,3] in Image {
         int11 dx, int11 dy =  for w in W dot h in H dot v in V
                 return( sum(h*(int11)w), sum(v*(int11)w );
         uint11 edge = sqrt(dx * (int22)dx + dy * (int22)dy);
      } return( array(edge) );
} return R;
```

Fig. 1. SA-C source code for computing edge magnitudes using horizontal and vertical Prewitt masks.

As an example, Figure 1 shows the SA-C code for computing edge magnitudes using horizontal and vertical Prewitt edge masks. At first glance, the code may seem unlike C, but the differences are motivated by the three criteria above, and programmers adapt quickly. The heart of the program is a pair of nested loops. The outer loop executes once for each 3x3 window in the source image. It contains the inner loop, which convolves the Prewitt masks with the image window; and also computes the edge magnitude as the square root of the sum of the squares of the mask responses. This demonstrates SA-C's looping mechanisms in terms of "for

window" and "for element" loops, as well as the ability to combine loops through dot products that run generators in lock step[1].

Many of the other syntactic differences between SA-C and C in this example concern data types. In this case, the source image has unsigned 8-bit pixels, but of course the Prewitt mask values only require two bits each (one for sign, one for value). This is significant because FPGA configurations are special-purpose circuits, and it takes less time and space to multiply an 8-bit value by a 2-bit value (producing a signed, 9-bit value) then it does to multiply two signed 16 bit values (the closest alternative in C). In this example, we have carefully specified the smallest data types possible, for demonstration purposes. This creates efficient circuits, but requires a lot of casting to set the data types of intermediate values, and is often a source of programming errors. In practice, we tend to set the input and output types of a function precisely, while using larger values for intermediate values. In most cases, the compiler can infer maximum sizes and remove unnecessary bits automatically, without the risk of introducing software bugs.

Figure 2 shows another example, this time of a dilation operator. In particular, we show a program designed to compute a gray-scale dilation using a 3x3 mask of 1s. This example is written so as to demonstrate SA-C's ability to take a "slice" of an array (i.e. part of a row or column), in this case by using `array_max(W[:,0])` to compute the maximum of the first row of a window. It also demonstrates the array operators (in this case, `array_max`) that are predefined for simple loops.

```
uint8[:,:] dilate (uint8 A[:,:]) {
  uint8 R[:,:] =
    for window W[3,3] in A {
      uint8 m0 = array_max(W[:,2]);
      uint8 m1 = array_max(W[:,1]);
      uint8 m2 = array_max(W[:,0]);
      uint8 V[3] = {m0,m1,m2};
      uint8 mx = array_max(V);
      uint8 val = array_min((uint9)(mx)+1,255);
    } return (array(val));
  } return (R);
```

Fig. 2. SA-C source code for computing a gray-scale dilation with a 3x3 mask of 1's.

[1] Loops can also be combined through cross products that generate all combinations of data elements.

3 The SA-C Compiler

3.1 Overview

The SA-C compiler translates high-level SA-C code into data flow graphs, which can be viewed as abstract hardware circuit diagrams without timing information [8]. Nodes in a data flow graph are either simple arithmetic, simple control (e.g. selective merge), or array access/storage nodes; while edges are data paths that correspond to variables. Figure 3 shows an unoptimized data flow representation of the Prewitt code from Figure 1. Note that the variables in Figure 1 turn into edges in the data flow graph; instead of memory locations as with a von Neumann processor. The operators are simple arithmetic operators that can be implemented in circuitry.

The two exceptions to the "simple operator" rule in Figure 3 are the window generator and window collector nodes. Large arrays and unbounded arrays (i.e. arrays whose sizes are not known at compile time) are stored in local memory on the reconfigurable processor, and window generators[2] are used to incrementally feed them into the FPGA. Window collectors gather output values and store them in memory as arrays (if needed). Both of these node types are themselves broken down into graphs of simple primitives, such as read word, write word, and increment address. However, because these subgraphs include registers (to hold addresses and values), they are not stateless, and therefore are not traditional data flow nodes. They must be handled specially within the compiler.

The SA-C compiler optimizes data flow graphs before generating a VHDL hardware description. Some optimizations are traditional while others were specifically designed to suit the language and reconfigurable hardware. Traditional optimizations include: Common Subexpression Elimination (CSE), Constant Folding, Invariant Code Motion, Dead Code Elimination, Function Inlining, and Loop Unrolling. The specialized optimizations include Loop Stripmining, Array Value Propagation, Loop Fusion, Lookup Tables, Temporal CSE, Window Narrowing, Size Propagation Analysis, and Pipelining. We show examples of some of these optimizations below.

Once the SA-C compiler has translated the data flow graph into VHDL, commercial compilers map the VHDL onto an FPGA configuration. The SA-C compiler also automatically generates all the necessary run-time code to execute the program on a reconfigurable processor. This includes downloading the FPGA configuration, data and parameters, and uploading results.

[2] Or slice generators, or element generators.

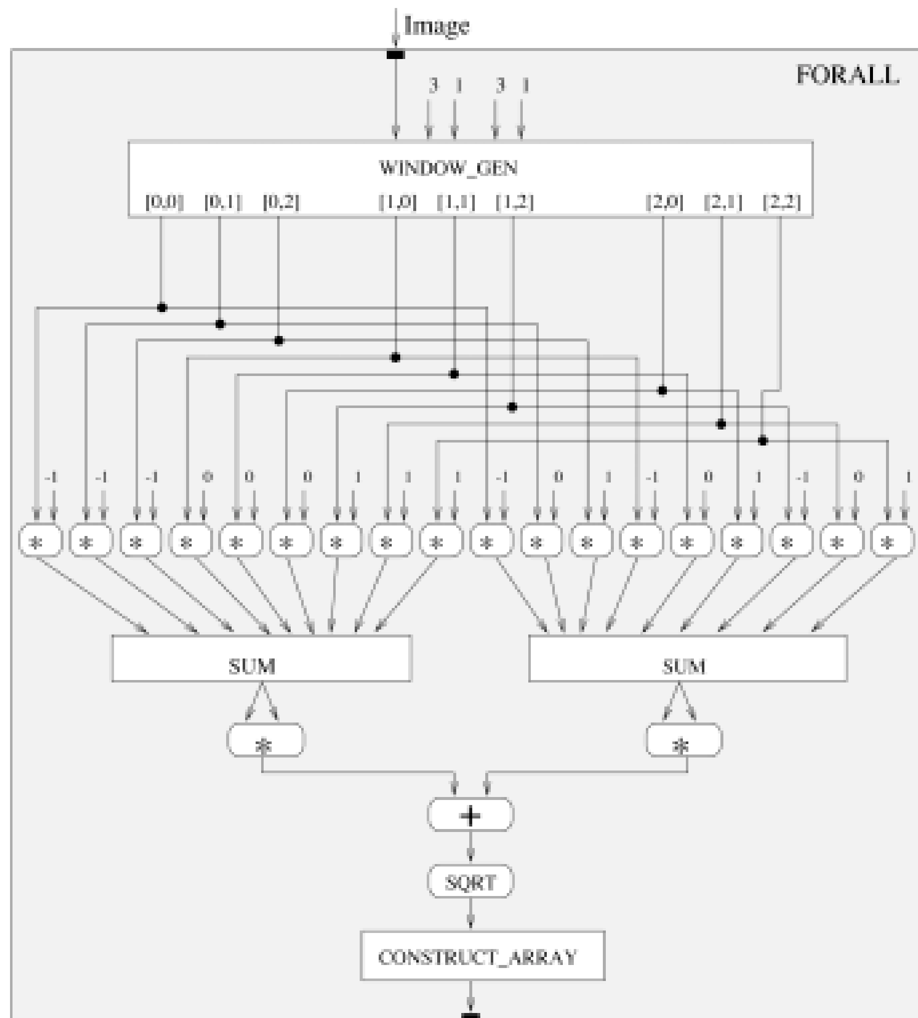

Fig. 3. Naïve mapping of the SA-C source code in Figure 1 to a data flow graph.

3.2 Example #1: Edge Magnitude

Part of the power of SA-C's compiler comes from traditional optimizations. Consider, for example, the inner loop of the Prewitt operator in Figure 1. This loop convolves a 3x3 data window with two 3x3 masks. Implemented naively (as in Figure

3), it would require 18 multiplications and 16 additions to compute the two responses. However, the SA-C compiler is aggressive about constant propagation and constant folding; in particular, it propagates constants through arrays. Six of the 18 multiplications are with the constant zero, and can be eliminated (along with the subsequent additions, since there is no reason to add zero.) Six more are multiplications with the constant 1; and can also be removed. Finally, the last six are with the constant –1; these are replaced by a negation operator. As a result, the inner loop requires no multiplications, ten additions, and five negations[3], as shown in Figure 4. These can be arranged in a tree to maximize parallelism.

Other optimizations in the SA-C compiler are specific to the language and machine model. One of these is pipelining. After the convolutions with the Prewitt masks, the program in Figure 1 still has to compute the square root of the sum of the squares of the mask responses. While the sum of the squares is relatively easy (two multiplications and an addition), square root is an expensive operation. It generates a circuit whose length is proportional to the number of bits in the argument (see [9], Chapter 21). Since there is only one square root operation per window, the size of the square root circuit is not a problem. The length of the square root circuitry is a problem, however, since the propagation delay through the circuit determines the frequency (MHz) of the FPGA. The SA-C compiler addresses this by pipelining the computation. It puts layers of registers through the square root circuitry, turning it into a multi-cycle operation as shown in Figure 5. This keeps the frequency of the circuit up, and introduces a new form of parallelism, since all stages of the pipeline are executed concurrently.

Together, these two optimizations solve a classic problem for parallel image processors. Convolution is a data-parallel operation that can be accelerated by allocating one processor per pixel. Square root is a sequential operation that can be accelerated by breaking it into stages and allocating one stage per processor. Combining a convolution and a square root is therefore problematic for many multiprocessors. FPGAs have enough logic blocks, however, to allocate a (sub)circuit to every pixel in the convolution step and every stage of the square root pipeline, combining both forms of parallelism and eliminating any bottleneck.

[3] There are five negations, not six, because of common subexpression elimination.

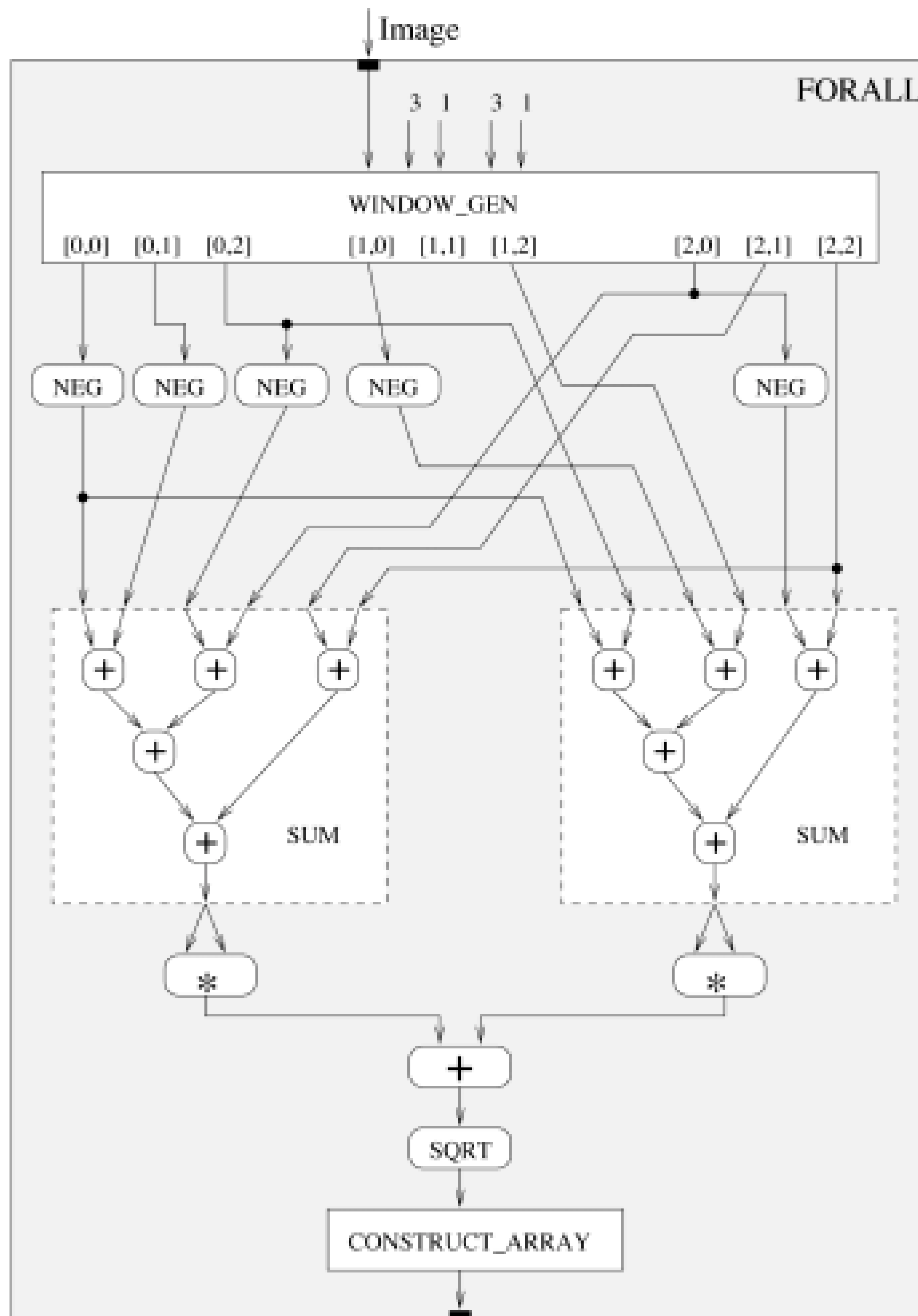

Fig. 4. Data-flow graph for source code in Figure 1, after constant propagation, constant folding, and common subexpression elimination.

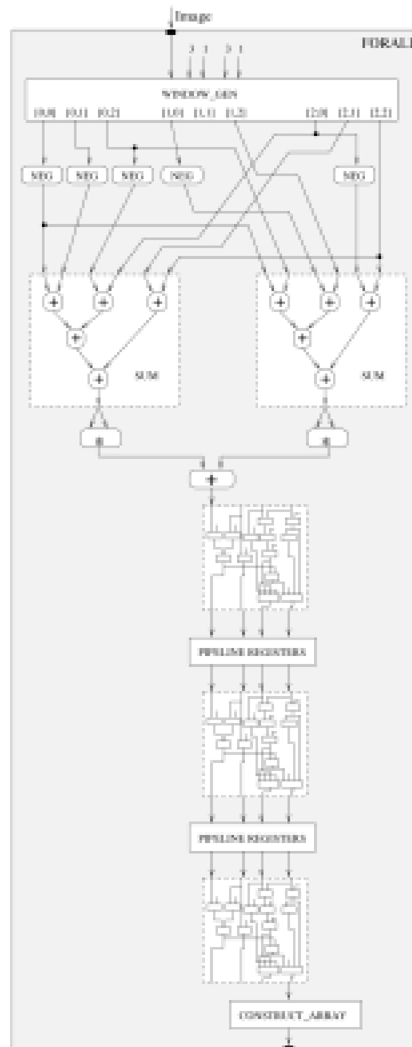

Fig. 5. Edge magnitude data flow graph after pipelining. Note that the position of the pipeline registers is symbolic; they are actually placed so as to equalize the propagation delays and thereby maximize frequencies.

3.3 Example #2: Dilation

The dilation example in Figure 2 is also optimized using language-specific techniques. Dilating a gray-scale image by a 3x3 mask of 1s requires taking the maximum of nine values in every 3x3 data window. Once again, the naïve implementa-

tion is to arrange the nine max operators in a binary tree to maximize parallelism. The code in Figure 2 was organized so as to emphasize a point, however: the computation can be structured hierarchically into columns, by first taking the max of each column, and then taking the maximum of the three column maximums. Structuring the code in this way makes it clear how the compiler can take advantage of image windowing. As the data window slides from left to right, the same column-wise operations are being repeated. The maximum of the right-most column at time step #1 (*m0* in Figure 2) is the maximum of the middle column (*m1*) at time step #2 and the maximum of the left-most column (*m2*) at time step #3. Instead of recomputing these values, the compiler can insert shift registers to store the values computed at previous time steps and reuse them. Thus, at any given time step, the program only has to take the maximum of five values: the two previously computed column maximums and the three new values in the right-most column. This results in a smaller and faster binary tree of max operators. This general optimization of reusing values computed on previous iterations is known as *Temporal Common Subexpression Elimination*, or Temporal CSE.

Temporal CSE has a second, less obvious benefit. The only pixels being accessed are now those in the right-most column; the pixels in the middle and left columns are not accessed, since their maximum was computed and stored on a previous iteration. As a result, the window size can be reduced from 3x3 to 3x1, saving registers in the window generator. In and of itself, there is only a modest savings in this step, which we call *window narrowing*. Simple procedures like dilation, however, are almost never executed alone; they are components of larger programs. Another optimization, called *loop fusion,* minimizes the number of passes over the data by fusing loops that have producer/consumer relationships. For example, if a program dilates an image twice with a 3x3 mask, it can be implemented as a single pass with a 5x5 window.

The disadvantage of loop fusion is that the size of the image window grows as more loops are fused. In the example from Figure 2, the window is 3x3, so fusing two iterations produces a 5x5 window, and fusing four iterations produces a 9x9 window. Implementing a 9x9 sliding window requires buffering 81 8-bit values, at a cost of at least 648 logic blocks. If the window size or pixel size is larger, the buffering requirements get worse. Fortunately, Temporal CSE and window narrowing reduce the window size of the dilation operator in Figure 2 to 3x1; fusing four iterations of the dilation operator therefore produces a window of only 9x1. In other words, the size of the buffers grows linearly with the number of loops fused, not quadratically.

In other programs the computation may be less regular, in the sense that the computation over the pixels in one column may be distinct from the computation in any other. In this case, it may still be possible to “push the computation to the right” by computing values as soon as the source pixels appear in the image window and then shifting the result to the left via registers. In this case, window narrowing can still be applied prior to loop fusion.

3.4 Example #3: I/O

One disadvantage of FPGAs is their limited I/O facilities. Although we can compute many things in parallel, we can only read or write 32 bits per cycle per memory (64 on some machines). As a result, very simple operations that do not do much computation per pixel will be I/O bound and perform poorly on FPGAs (see Section 4.1). This problem is exacerbated by the redundancy inherent in sliding a window across an image. Although we use shift registers to avoid reading any pixel more than once as we slide the image window from left to right, at the end of the row we have to drop the window one row and start sliding again (like the carriage return on a typewriter). As a result, if the processing window is *m by n*, each pixel is read *m* times.

An optimization that reduces this problem in both the edge magnitude and dilation examples is called vertical *stripmining*. The general idea is that instead of sliding a 3x3 window across the image, we slide a 4x3 window instead. By adding an extra row to the data window, we can compute two instances of the 3x3 operator (either edge magnitude or dilation) at once, one above the other. This adds another level of parallelism to the system. More importantly, because we are computing two windows worth of data at each step, when we reach the end of a row we can drop the data window two rows instead of one. This halves the amount of I/O by halving the number of times each pixel is read. In general, for a window of height n, if we stripmine it q times, we reduce the memory I/O by a factor of $(n+q)/(q+1)$.

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. The compiler produces both the FPGA configuration and all necessary host code to download the configuration onto the FPGA, download the data onto the recongifurable processor's local memory, trigger the FPGA, and upload the results. SA-C therefore makes reconfigurable processors accessible to applications programmers with no particular expertise in FPGAs or circuit design.

The question for applications programmers is which tasks will benefit from reconfigurable technology. To answer this question, we have compiled image processing programs of varying complexity to an Annapolis Microsystems StarFire, with a Xilinx XV-1000 Virtex FPGA and six local memories of 1 MByte each. We compare its performance to a general-purpose processor of similar age, a 450 MHz Pentium II with 128 MBytes of RAM.

4.1 Simple Image Processing Operators

Our first instinct was to write the same program in SA-C and C, and compare their performance on the Starfire and Pentium II. Unfortunately, this would not have been a fair comparison, since neither the Microsoft nor Gnu C++ compilers exploit the Pentium's MMX technology. Instead, we wrote SA-C programs corresponding to a set of simple image operators from the Intel Image Processing Library (IPL), since the IPL was written by Intel in assembly code and optimized for MMX. We then compare the performance of the Starfire to the Pentium II with MMX acceleration, as shown in Table 1. As can be seen, the Pentium outperforms the FPGA on these tasks by factors ranging from 1.2 to 5.8.

Table 1. Execution times in seconds of simple IPL routines on a 512x512 8-bit image. The comparison is between a 450Mhz Pentium II and an AMS Starfire with one Xilinx XV-1000 (Virtex) FPGA. The ratio is between the execution times (Pentium/Starfire).

Routine		Pentium II	Starfire	Ratio
1	AddS	0.00120	0.00687	0.17
2	OrS	0.00127	0.00628	0.20
3	SubtractS	0.00133	0.00617	0.22
4	Not	0.00140	0.00650	0.22
5	AndS	0.00146	0.00628	0.23
6	Or	0.00241	0.00822	0.29
7	And	0.00243	0.00822	0.30
8	Subtract	0.00253	0.00841	0.30
9	Open	0.01147	0.03493	0.33
10	Close	0.01158	0.03280	0.35
11	Add	0.00246	0.00688	0.36
12	Multiply	0.00300	0.00816	0.37
13	Dilate	0.00857	0.01441	0.59
14	Erode	0.00902	0.01463	0.62
15	Square	0.00529	0.00659	0.80
16	Gaussian	0.00538	0.00625	0.86

This result is not surprising. Although the FPGA has the ability to exploit fine grained parallelism, it operates at a much slower clock speed than the Pentium. (The clock speed of an FPGA depends on the configuration; these programs were clocked at between 30 and 40 MHz.) These simple programs are all I/O bound, and the slower clock speed of the FPGA is a major limitation when it comes to fetching data from memory.

The first eight routines in Table 1 are pixel-wise operators with very simple logic, and the Pentium is three to six times faster than the FPGA. Six of the last eight are neighborhood operators, and here the Pentium is 1.2 to 3 times faster. One reason for this relative improvement is that the SA-C compiler is able to vertically strip-mine neighborhood operations, minimizing the number of read operations, as discussed in Section 3.4. Another is that the SA-C compiler knows the neighborhood

masks at compile time, and is able to reduce the total number of operations by constant propagation and constant folding, as discussed in Section 3.2.

4.2 Application Tasks

Next we look at the edge magnitude function in Figure 1, which is more complex then the IPL routines above. Edge magnitudes can be computed using the Prewitt masks with a single function call (iplConvolve[4]) in the IPL. The execution times for the edge magnitude operator are on the top line of Table 2. This time it is the FPGA that is faster by a factor of four. This is because the slower clock rate of the FPGA is offset by its ability to do more work per cycle by exploiting parallelism.

The FPGA compares even better as we look at more complex operations. The ARAGTAP pre-screener was developed by the U.S. Air Force as the initial focus-of-attention mechanism for a SAR automatic target recognition application [10]. It includes operators for down sampling, dilation, erosion, positive differencing, majority thresholding, bitwise "and", percentile thresholding, labeling, label pruning and image creation. Most of the computation time is spent in a sequence of eight gray-scale morphological dilations, and a later sequence of eight gray-scale erosions. Both the dilations and erosions alternate between using a 3x3 mask of 1's (as in Figure 2) and a 3x3 mask with 1's in the shape of a cross and zeros at the corners.

In principle, the SA-C compiler should be able to fuse a sequence of eight dilations into a single operator that passes only once over the image. Unfortunately, a problem with our place and route mechanism prevents us from fusing all eight dilations (or all eight erosions) into a single operation. We are able to fuse four dilations into a single loop, however, and therefore divide the eight dilations into two sets of four. Table 2 shows the run-times for four dilations and four erosions on the FPGA and Pentium II. In both cases, the FPGA is fifteen times faster than the Pentium. If we go further and run the dilations, erosions, positive differencing, majority threshold and bitwise "and" on the FPGA, the run-time drops from 1.07 seconds on a Pentium to 0.041 seconds, an twentysix-fold speed-up.

In this experiment, we did not use the Intel IPL library to evaluate the Pentium. The IPL's dilate and erode routines cannot alternate masks, nor do they match the definitions of gray-scale dilate and erode used by ARAGTAP. Therefore we used an implementation of ARAGTAP by Khoral Research, Inc., compiled with optimizations using g++ and timed on an Intel Pentium II running Linux. As a result, the Pentium II was not exploiting its MMX technology for this evaluation.

[4] iplConvolve can apply multiple masks and return the square root of the sum of the squares of the mask responses at each pixel, as required for computing edge magnitudes.

Table 2. Run-times for complex operators. The comparison is between a 450Mhz Pentium II and an AMS Starfire with one Xilinx XV-1000 (Virtex) FPGA. The ratio is between the execution times (Pentium/Starfire).

Routine	Pentium II	Starfire	Ratio
Prewitt	0.057	0.013	4.38
Open (4)	0.20	0.00612	32.68
Close (4)	0.20	0.00610	32.79
ARAGTAP	1.07	0.04078	26.24

4.3 FLIR Probing

The most computationally intensive algorithm we have studied to date is a probing algorithm developed by the U.S. Army Night Vision Laboratory. The underlying idea is simple: a "probe" is a pair of pixels that should straddle the boundary of a target when viewed from a known position. If the difference between the two probe values in an image exceeds a threshold, the probe fires. A "probe set" is a set of probes arranged around the silhouette of a target, as shown in Figure 6. The positions of probes in a probe set are specified relative to a window, and the window slides across the image. At any image position, if the number of firing probes in a probe set exceeds a (second) threshold, the target is detected at that location.

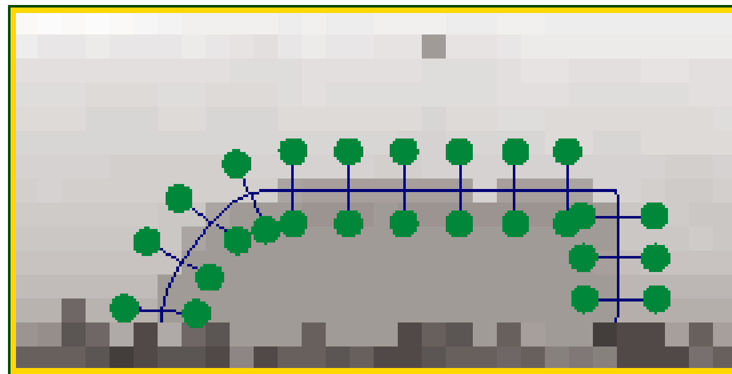

Fig. 6. A set of probes outlining the silhouette of a target. The positions of probes in a probe set is relative to a sliding image window. If at any point the number of firing probes exceeds a threshold, the target is detected at that location.

What makes this algorithm difficult is the quantity of probes and the time constraint. A typical probe set consists of twenty to twenty-five probes. Each target requires one probe set for each possible view, and the viewing hemisphere is divided into eighty-one canonical views. Multiple distinct targets may have to be recognized in real-time (1/30th of a second). Currently, the goal is to detect four targets via 324 probe sets of 20-25 probes each at every image location, 30 times a second. Doing

this on traditional hardware requires a bank of processors, with the associated space, weight and power requirements thereof.

This algorithm is well suited to SA-C and FPGAs. The basic computational model is a sliding window. Moreover, it is easy to program SA-C to evaluate N probe sets at each image position before sliding the window, which takes advantage of the parallelism inherent in FPGAs. We are currently investigating how many FPGAs will be required to compute 324 probe sets in real-time.

The probing algorithm exploits the strengths of the SA-C compiler. If two probe sets have a probe in common, common subexpression elimination (CSE) will prevent it from being computed twice. If two probe sets share the same pair of probes, CSE will also reuse the sum of the probe responses. More importantly, if one probe is a horizontally translated version of another probe, temporal common subexpression elimination (TCSE) will compute the probe response only once, and insert shift registers to reuse the result, as discussed in Section 3.3. Finally, window narrowing will compute all probe responses as soon as the data appears in the image window, reducing the number of registers needed to implement the sliding window.

Execution times for one, two, four and eight probe sets on the AMS Starfire with a single Xilinx XV-1000 FPGA are shown in Table 3. The run-times were computed on a 100x256 12-bit image. The execution times for the AMS Starfire are a bit erratic as a function of the number of probe sets, for reasons we do not understand yet. Unfortunately, we do not have a C implementation of this algorithm by a neutral party to use as a fair comparator.

Table 3: Comparison of run-times for probing on an AMS Starfire and Pentium II. The execution times are in seconds on a 100x256 12-bit image.

Number of Probe Sets	Total Number of Probes	Execution (Starfire)	Percent of FPGA used
1	22	0.0015	14%
2	43	0.0039	15%
4	87	0.0016	17%
8	186	0.0043	20%

7 Conclusion

Should applications programmers rush out and adopt FPGA technology? Sadly, no (or at least not yet). To run an operator on an FPGA, 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 for a 512x512 image is about 0.019 seconds on our system. As a result, the data transfer time almost cancels the gain in execution time for the edge magnitude example. Even with data transfer times included, the FPGA is still faster than the Pentium II on the dilate and erode

examples, but the gain is reduced to a factor of 4.5. Generally it is not worth adopting new hardware technology unless the speed-up is a full order of magnitude.

Worse still, current FPGAs cannot be reconfigured quickly. It takes about 0.1 seconds to reconfigure a Virtex over a PCI bus. It is therefore not feasible to dynamically reconfigure it during an application. This is a problem, for example, when running the ARAGTAP pre-screener, where more than half of the one second improvement in execution time is negated by the need to reconfigure the FPGA six times. Applications programmers need to select one operator sequence to accelerate per reconfigurable processor and pre-load it with the appropriate FPGA configuration. In the case of the ARAGTAP pre-screener, it would be best to have two reconfigurable processors, and use one to accelerate the dilations and the other to accelerate the erosions.

As a result, FPGAs are currently well suited to only a small fraction of computer vision applications. In particular, they are suited to applications where a small speed-up (e.g. by a factor of five) matters, and where most of the processing time is spent in a single sequence of image processing operators. In addition, this sequence of operators must be able to fit on an FPGA (although FPGAs get larger all the time). Applications that meet these criteria can be accelerated by FPGAs, and should continue to be accelerated as both FPGAs and general purpose processors get faster.

The future trend in FPGA design is to build processors that 1) can be quickly reconfigured and 2) have a general-purpose RISC processor on-board. Chameleon, for example, is a new company building FPGAs that can be reconfigured in a single clock cycle (see http://www.chameleonsystems.com). This will greatly expand the application domain of FPGAs, since they will no longer be constrained to applications with a single computational bottleneck. In addition, Xilinx has announced (but not yet released) the Virtex II, which puts a RISC processor on every FPGA chip. This will eliminate the data transfer time, making FPGAs much faster than standard processors for many operations, and making it easier to interleave processing between an FPGA and a general-purpose processor. The Virtex II will also include floating point arithmetic units distributed throughout the FPGA grid, making floating point operations much more efficient. When this next generation of chips are available, FPGAs should become a standard part of every computer vision system.

The contribution of the Cameron project to this future is a language and compiler for mapping high-level algorithms to FPGA configurations. Currently, FPGAs are rarely applied even for applications that fit the criteria above; it is just too difficult to implement complex algorithms in circuit design languages. SA-C and the SA-C optimizing compiler eliminate this barrier, allowing FPGAs to be easily and quickly exploited.

8 Acknowledgements

This work was sponsored by DARPA and the U.S. Air Force Research Laboratory under contract F33615-98-C-1319.

9 References

1. DeHon, A., *The Density Advantage of Reconfigurable Computing.* IEEE Computer, 2000. **33**(4): p. 41-49.
2. Benedetti, A. and P. Perona. *Real-time 2-D Feature Detection on a Reconfigurable Computer*. in *IEEE Conference on Computer Vision and Pattern Recognition*. 1998. Santa Barbara, CA: IEEE Press.
3. Woodfill, J. and B.v. Herzen. *Real-Time Stereo Vision on the PARTS Reconfigurable Computer*. in *IEEE Symposium on Field-Programmable Custom Computing Machines*. 1997. Napa, CA: IEEE Press.
4. Benitez, D. and J. Cabrera. *Reactive Computer Vision System with Reconfigurable Architecture*. in *International Conference on Vision Systems*. 1999. Las Palmas de Gran Canaria: Springer.
5. Hartenstein, R.W., *et al. A Reconfigurable Machine for Applications in Image and Video Compression*. in *Conference on Compression Technologies and Standards for Image and Video Compression*. 1995. Amsterdam.
6. Eldredge, J.G. and B.L. Hutchings. *RRANN: A Hardware Implementation of the Backpropagation Algorithm Using Reconfigurable FPGAs*. in *IEEE International Conference on Neural Networks*. 1994. Orlando, FL.
7. Hammes, J.P., B.A. Draper, and A.P.W. Böhm. *Sassy: A Language and Optimizing Compiler for Image Processing on Reconfigurable Computing Systems*. in *International Conference on Vision Systems*. 1999. Las Palmas de Gran Canaria, Spain: Springer.
8. Dennis, J.B., *The evolution of 'static' dataflow architecture*, in *Advanced Topics in Data-Flow Computing*, J.L. Gaudiot and L. Bic, Editors. 1991, Prentice-Hall.
9. Parhami, B., *Computer Arithmetic: Algorithms and Hardware Designs*. 2000, New York: Oxford University Press.
10. Raney, S.D., *et al. ARAGTAP ATR system overview*. in *Theater Missile Defense 1993 National Fire Control Symposium*. 1993. Boulder, CO.