Compiling and Optimizing Image Processing Algorithms for FPGA's

Bruce Draper, Walid Najjar, Wim Böhm, Jeff Hammes, Bob Rinker, Charlie Ross, Monica Chawathe, José Bins

Department of Computer Science
Colorado State University
Fort Collins, CO, U.S.A. 80523
draper, najjar,…@cs.colostate.edu

Abstract

This paper presents a high-level language for expressing image processing algorithms, and an optimizing compiler that targets FPGAs. The language is called SA-C, and this paper focuses on the language features that 1) support image processing, and 2) enable efficient compilation to FPGAs. It then describes the compilation process, in which SA-C algorithms are translated into non-recursive data flow graphs, which in turn are translated into VHDL. Finally, it presents performance numbers for some well-known image processing routines, written in SA-C and automatically compiled to an Annapolis Microsystems WildForce board with Xilinx 4036XL FPGAs.

1. Introduction

Over the past several years, field-programmable gate arrays (FPGAs) have evolved from simple "glue logic" circuits into the central components of reconfigurable computing systems [1]. In general, FPGAs consist of grids of reprogrammable logic blocks, connected by meshes of reprogrammable wires. Reconfigurable computing systems combine one or more FPGAs with local memories and (PCI) bus connections to create reconfigurable co-processors. The current state of the art is to accelerate the most expensive part of any computation by directly embedding it as a circuit in a reconfigurable co-processor. The greater future potential lies in including FPGAs on-chip with the main processor, giving the benefit of general-purpose acceleration without the communication bottleneck created by placing the FPGA in a co-processor.

Recently, the image processing community has become aware of the potential for massive parallelism and high computational density in FPGAs. For example, FPGAs have been used for real-time point tracking [2], stereo [3] and color-based object detection [4]. Unfortunately, the difficulty of implementing complex algorithms in circuit design languages has discouraged many computer vision researchers from exploiting FPGAs, while the intrepid few who do are repeatedly frustrated by the laborious process of modifying or combining FPGA 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 [5]) for expressing 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 an FPGA. While the resulting run-times are still somewhat less 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 do not know circuit design languages such as VHDL or Verilog.

In particular, this paper provides a brief description of SA-C, focusing on the language features that 1) support computer vision and image processing, and 2) enable efficient compilation to FPGAs. (A more complete description of SA-C can be found in [5], while a

programmer's reference is available on the web[1]). It then describes the compilation process, in which SA-C algorithms are translated first into non-recursive data flow graphs and then into VHDL. A commercial compiler then converts the VHDL into FPGA configurations. We also briefly describe the run-time system for downloading data and FPGA configurations and retrieving results.

Finally, we present performance numbers for some well-known image processing routines, written in SA-C and automatically compiled to an Annapolis Microsystems WildForce board with Xilinx 4036XL FPGAs. As part of these performance numbers we are able to demonstrate the impact of specific compiler optimizations, including loop fusion, constant propagation and stripmining, on individual image processing functions and on sequences of image processing functions.

2. SA-C

SA-C is a single-assignment variant of the C programming language developed in concert with Khoral Inc. and designed to exploit both coarse-grained (loop-level) and fine-grained (instruction-level) parallelism in image processing applications. Roughly speaking, the differences between SA-C and standard C can be grouped into three categories: 1) data types added to C to exploit arbitrary bit precision logic circuits; 2) control extensions to C designed to support image processing by providing parallel looping mechanisms and true multi-dimensional arrays; and 3) restrictions on C, designed to prevent programmers from applying von Neuman memory models that do not map well onto FPGAs. In general, it has been our experience that programmers who are familiar with C adapt quickly to SA-C.

2.1. SA-C Data Types

To go into more detail, data types in SA-C include signed and unsigned integers and fixed point numbers, with user-specified bit widths. For example, *uint8* is an 8-bit unsigned type, whereas *fix12.4* is a fixed point type with a sign bit, seven whole number bits and four fractional bits. SA-C also has *float* and *double* types that hold 32- and 64-bit values respectively, and *bool* and *bits* types that can hold non-numeric bit vectors. Complex types composed of the language's signed numeric types are also available. Since SA-C is a single-assignment language, variables are declared and given values simultaneously. For example, the statement *uint12 c = a + b;* gives *c* both its type and its value.

SA-C has multidimensional rectangular arrays whose extents are determined dynamically during program execution (although programmer's have the option to specify the extents statically). The type declaration *int14 M[:,6]* declares a matrix *M* of 14-bit signed integers. The first dimension is determined dynamically; the second dimension is specified by the user to be six. Arrays in SA-C can be sectioned or "sliced" using a syntax similar to Fortran 90 [6]: for example, the expression *A[:,i]* returns the i^{th} column of array *A*.

2.2. Looping in SA-C

The most important part of SA-C is its treatment of *for* loops. A loop in SA-C returns one or more values (i.e., a loop is an expression), and has three parts: one or more generators, a loop body, and one or more return values. The generators provide array access expressions that are concise and easy for the compiler to analyze. Most interesting is the *window* generator, which extracts sub-arrays from a source array. Here is a median filter written in SA-C:

```
uint8 R[:,:] =
  for window W[3,3] in A {
    uint8 med = array_median (W);
  } return (array (med));
```

The *for* loop is driven by the extraction of 3x3 sub-arrays from array *A*. All possible 3x3 arrays are taken, one per loop iteration. The loop body takes the median of the sub-array, using a built-in SA-C operator. The loop returns an array of the median values, whose extents are determined by the size of *A* and the loop's generator. SA-C's generators can take windows, slices and scalar elements from source arrays in arbitrary strides, making it frequently unnecessary for source code to do any explicit array indexing whatsoever.

[1] The SA-C language reference, compiler documentation, and other Cameron project information can be found at http://www.colostate.edu/cameron.

Arrays may be padded prior to the loop if the output has to be the same size as the input.

2.3. Restrictions (compared to C)

SA-C restricts C by eliminating pointers and recursion and by enforcing single-assignment semantics. Pointers allow programmers to access memory in a manner that is difficult to analyze at compile-time. By eliminating pointers, SA-C makes data dependencies clear and allows the compiler to easily identify instruction-level parallelism. It also makes data access patterns clearer, making it easier to identify and support loop-level parallelism. Since the most common use of pointers in image processing is for rapid access into large arrays, SA-C's multidimensional arrays and parallel looping mechanisms (described above) make this use of pointers obsolete.

SA-C also eliminates recursion. FPGAs do not have a von Neuman style processor with a calling stack; instead, they are a grid of programmable logic blocks with software-controllable interconnections. As a result, programs have to be mapped to flat logic circuits. Although some recursive programs can be transformed into loops, SA-C guarantees that the process of mapping programs to circuits will not be unnecessarily complex by eliminating recursion.

SA-C's single-assignment restriction is often misunderstood, at least by readers who are not familiar with functional languages. No variable in SA-C can be assigned a value more than once. There is no restriction against reusing variable names, however, so there is nothing wrong with the statement *b = b+1;* it simply creates a new variable named *b* whose initial value is one more than the value of the previous variable named *b* (and whose type is inferred from *b*). Instead, the biggest impact from the programmer's point of view of single-assignment semantics lies in the control statements. Since variables cannot be reassigned, they cannot be conditionally assigned without risking uninitialized values. To compensate, all control statements (*if*, *for*, etc.) are expressions that return (possibly multiple) values. This allows programmers to avoid assigning values inside conditionals, for example by creating a conditional that returns either a computed value or a default value, and assigning the result of the conditional to a variable. So-called "nextified" variables make it possible to carry values from one iteration of a loop body to the next [7].

The reason for the single assignment restriction is that it establishes a one-to-one correspondence between variables in the program and wires in the resulting circuit. Since variables do not change value, they do not need to be associated with memory. Instead, every operator is a sub-circuit, and the variables it operates on are the input wires. This not only makes is easy to generate VHDL circuits from SA-C, but it makes it easier for programmers to understand this mapping and to write SA-C programs that translate into efficient circuits.

2.4. The Compiler Architecture

The SA-C compiler can target either a traditional processor (for debugging) or the combination of a traditional host processor with a reconfigurable coprocessor. When targeting a reconfigurable system, SA-C compiles parallelizable loops to logic circuits for the FPGA. Sequential code not inside a loop is compiled to C for the host processor; the SA-C compiler also generates C code for downloading the FPGA configuration and data to the coprocessor and for reading the results back. When compiling loops into circuits, it translates the SA-C code first into a dataflow graph and then into a logic circuits expressed in VHDL. A commercial VHDL compiler maps these circuits into FPGA configurations.

3. Data Flow Graphs

The SA-C compiler translates *for* loops and *while* loops into dataflow graphs, which can be viewed as abstract hardware circuit diagrams without timing information [7]. Nodes in a data graph are either arithmetic, low-level control (e.g. selective merge), or array access/storage nodes; edges are data paths and correspond to variables. A dataflow graph simulator is available for validation and debugging.

As an example, consider the following SA-C fragment, which convolves an image with a 3×3 (vertically-oriented) Prewitt edge detection mask:

```
int16[:,:] main(uint8 Image[:,:])
    int16 H[3,3] = {  {-1, 0, 1},
                      {-1, 0, 1},
                      {-1, 0, 1}} ;
    int16 R[:,:] =
        for window W[3,3] in Image {
          int16 iph =  for h in H dot w in W
                  return( sum(h*(int16)w) );
        } return( array(iph) );
} return R;
```

The dataflow graph for this code is shown in Figure 1. Because the inner loop can be unrolled, the SA-C compiler converts both loops of this nested structure into a single dataflow graph. The Window-Generator node near the top of the graph reads elements from a 3×3 window of *Image* at each iteration, and as this window of data flows through the graph, the required convolution with *H* is performed. Notice that the multiplications in the code have either been removed by the compiler (for multiplication by 0 or 1) or converted to negation operators (for multiplication by -1). This is the result of constant propagation and constant folding. Thus, the nine multiplies and eight additions explicit in the code at each iteration of the loop have been replaced with three negation operations and five additions.

4. The SA-C compiler

The SA-C compiler does a variety of data flow graph optimizations, some traditional and some specifically designed for optimizing SA-C programs on reconfigurable hardware. The traditional optimizations include Common Subexpression Elimination, Constant Folding, Invariant Code Motion, and Dead Code Elimination. The specialized optimizations are variants of Loop Stripmining, Array Value Propagation, Loop Fusion, Loop Unrolling, Function Inlining, Lookup Tables, Array Blocking and Loop Nextification, as well as (loop and array) Size Propagation Analysis. Some of these interact closely and are now described briefly.

Since SA-C targets FPGAs, the compiler does aggressive full loop unrolling of inner loops, creating non-iterative blocks of code that are more suitable for translation to a DFG (and eventually a logic circuit). To help identify opportunities for unrolling, the compiler propagates array sizes through the DFG graph, inferring sizes wherever possible. SA-C's close association of arrays and loops makes this possible. Since the compiler converts only bottom-level loops to dataflow graphs, full loop unrolling can allow a higher-level loop to become a bottom-level loop, allowing it to be converted to a DFG.

The SA-C compiler does Loop Stripmining, which when followed by full loop unrolling produces the effect of multidimensional partial loop unrolling. For example, a stripmine pragma can be added to the median filter:

```
uint8 R[:,:] =
    // PRAGMA (stripmine (6,4))
    for window W[3,3] in A {
      uint8 med = array_median (W);
    } return (array (med));
```

This wraps the existing loop inside a new loop with a 6x4 window generator. Loop unrolling then replaces the inner loop with eight median code bodies. The resulting loop takes 6x4 sub-arrays and computes eight 3x3 medians in parallel.

The SA-C compiler can fuse many loops that have a producer/consumer relationship. For example, consider the following code in which a median filter is followed by an edge detector:

```
uint8 R[:,:] =
    for window W[3,3] in A {
      uint8 med = array_median (W);
    } return (array (med));
  uint8 S[:,:] =
    for window W[3,3] in R {
      uint8 pix = prewitt (W);
    } return (array (pix));
```

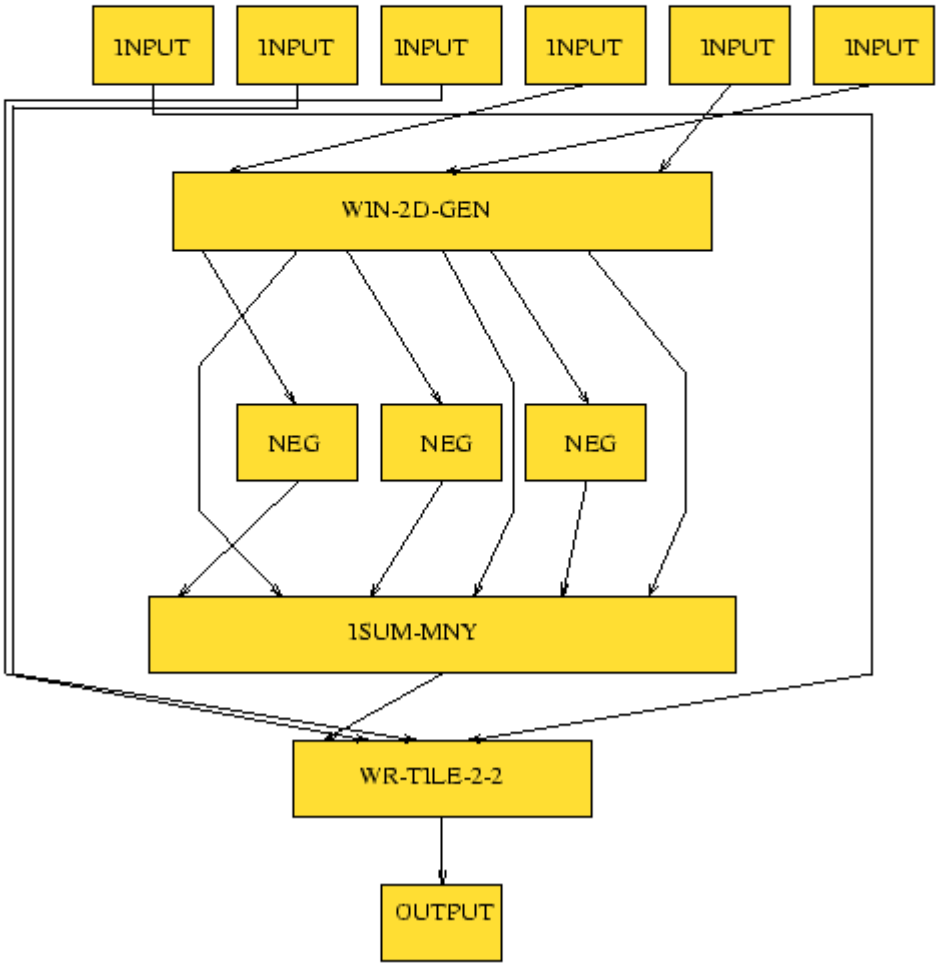

Figure 1. Dataflow graph for applying the vertical mask of the Prewitt edge detector, as compiled from the SA-C code above.

where *prewitt* is defined by the user as a separate function. The compiler will inline the function call, and fuse the loops into a single loop that runs a 5x5 window across *A*. (Note that a 5x5 window is required to provide the needed elements to compute one value of *S*.) The new loop body will have nine median code bodies, and one *prewitt* code body. The goal of fusion is primarily to reduce data communication between the coprocessor's local memory and its FPGA.

Loop fusion as described above does redundant computation of medians. If FPGA space is plentiful, this is not a problem since the medians are computed in parallel, but if space is tight, Loop Nextification will remove the redundancies in the horizontal dimension by passing the computed medians from one iteration to the next. In the example above, this reduces the loop body to three medians and one *prewitt*. If, after nextification, sufficient FPGA space is available, the fused loop can then be stripmined to reduce the number of iterations.

5. The VHDL Abstract Architecture

Unlike traditional processors, which provide a relatively small set of well-defined instructions to the user, reconfigurable processors such as FPGAs are composed of an amorphous mass of logic cells which can be interconnected in a countless number of ways. To limit the number of possibilities available to the compiler, an *abstract machine* has been defined to provide a hardware independent target model.

The data flow graph for a SA-C program has three parts, as shown in Figure 2. The first part consists of one or more data generators that read data from the coprocessor's local memory and present it in the proper sequence to the loop body. The second part is the main computational section of the program, called the inner loop body (ILB). The third part has one or more data collectors that buffer values calculated by the ILB and write them to memory.

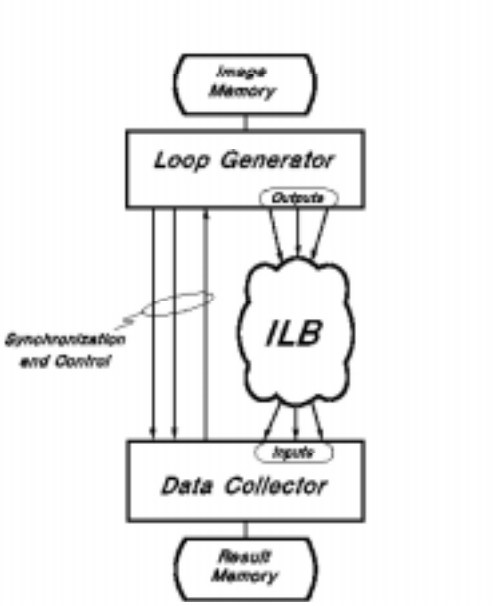

Figure 2. The three parts of a data flow graph.

Because the generators and collectors are buffered, the system as a whole works like a three stage pipeline. At any given moment, the generators may be reading new data from the memory while the ILB is processing the data from the previous time step, and the collectors are writing the results from the time step before that. Currently, the ILB is entirely combinational; all timing is handled by the data generator, with input from the collector. Future optimizations which pipeline the ILB are planned, however.

The process of translating a DFG to VHDL is divided into two main parts. The ILB is translated by traversing the dataflow graph and converting each node into VHDL. Many DFG nodes implement simple operations, such as arithmetic or logical operations; for these nodes, there is a one-to-one correspondence between DFG nodes and VHDL statements. For more complicated operations, including reduction operations like *array sum*, the translator generates a connection to a predefined VHDL component. A library of such components allow the SA-C compiler to directly access efficient hardware implementations for many complex operations.

Once the ILB has been translated into VHDL, the loop generator and collection nodes are implemented by selecting the proper VHDL components from a library, and are

parameterized with values extracted from the DFG. The interconnections between the ILB and generator/collection components are made by a top-level VHDL module, which is also generated by the translator.

The loop generator is responsible for presenting data to the ILB in the proper order and providing the signals necessary to control the operation of the result buffers in the collector. Data read from memory is placed in shift registers. On each cycle, the oldest column of data is shifted out while a new column is shifted in. These shift registers provide a "sliding window" of data that can be presented to the inputs of the ILB. Conversely, the collector accepts the ILB outputs, buffers them into words, and then writes them into the result memory.

Once the entire DFG has been converted to VHDL, the code is processed by a commercial VHDL compiler and place and route tools. These tools produce the final FPGA configuration files that are downloaded onto the reconfigurable processor and executed.

6. Image Processing Operators

To demonstrate SA-C and its optimizing compiler, we implemented a library of 22 image processing routines. The individual routines were selected not for any particular application, but to be representative of the routines in the Vector, Signal and Image Processing Library (VSIPL; http://www.vsip.org/) and the Intel Image Processing Library (IPL; http://support.intel.com/support/performancetools/libraries/). For each routine, Table 1 gives the processing rate in megapixels per second, with and without the data transfer time from the host to the reconfigurable processor. Note that when an image processing operator is applied to an image stream, data transfer can be overlapped with processing, creating the maximum sustained throughputs given in column 3. When operators are applied to isolated images, data transfer times must be included, leading to the throughputs in column 4. Table 1 also gives the size of the circuit, expressed in logic blocks, in column 5 and the clock rate of the FPGA, which is a function of the critical path length of the circuit, in column 2. All timings are relative to a 300×198 8-bit test image.

Most of the routines in Table 1 should be self-explanatory. Dyadic operators take two images as input, while monadic operators require an image and a constant. The routine labeled "window sum of diffs" is a dyadic operation that takes two images as input, computes the pixel-wise difference between the images, sums these differences and compares this sum to a threshold. It was taken from an implementation of the MP4 image compression standard.

The frequencies in Table 1 top out at around 9.5 MHz. As mentioned in Section 5, SA-C programs compile to circuits (configurations) with three pipelined components: the generator(s), inner loop body (ILB), and collector(s). The propagation delay of the overall circuit is the longest of the propagation delays of the three pipelined components. As a result, for the relatively simple routines in Table 1, it is often the propagation delay of the generator(s) or collector(s) that determines the maximum frequency, not the inner loop body.

In particular, clock rate for the image collector is about 9.5 MHz. (The image collector buffers pixels until it has a 32-bit value, computes the memory address, and then writes a word of data.) As a result, the only routines with a clock speed above 9.5 MHz are the *window sum of diffs* and *max value in image* routine, both of which return single values rather than images and therefore use simpler collectors. These routines are clocked at 10.1 MHz and 12.8 MHz, respectively.

The propagation delay of routines clocked below 9.5 MHz is determined by their generator(s) or inner loop body. For example, the three edge magnitude operators (*Prewitt*, *Roberts* and *Sobel*) all have clock rates at or below 3 MHz. In these cases, the inner loop body not only convolves the image window with both a horizontal and a vertical mask, but also computes the square root of the sum of the squares of the convolution results. It is this magnitude computation that accounts for three quarters of the propagation delay; manual experiments with look-up tables suggest that replacing the magnitude computation with a look-up table will significantly increase the frequency of these functions.

The maximum sustained throughput of a routine is a function both of its frequency and its I/O behavior. Many of the routines in Table 1 are I/O bound. For example, the monadic and dyadic *add* operators have similar frequencies (8.5 MHz and 9.0 MHz, respectively), but very different throughputs (4.2 megapixels/second vs. 9.0 megapixels/second). The number of cycles

Table 1. Timings for twenty-two simple image processing routines. The second column is the clock rate for the FPGA (which depends on the circuit). The third column is the processing rate, excluding data transfer time, expressed as megapixels per second. The fourth column is the processing rate including data transfer time. The fifth column is the number of configurable logic blocks (CLBs) used.

Benchmark	Frequency (MHz)	Processing (MP/s)	W/ data transfer	CLBs
Add (Dyadic)	8.477	4.23	3.94	750
Add (Monadic)	9.040	9.01	7.38	647
Convolution (3x3)	9.554	9.70	5.89	1,333
Dilation (3x3)	8.915	9.05	3.61	1,036
Erosion (3x3)	8.981	9.13	3.62	1,036
Gaussian Filter (3x3)	8.684	8.83	6.59	725
LaPlace Filter (3x3)	8.625	8.77	6.62	773
LaPlace Filter (5x5)	8.692	4.52	4.00	1,214
Window Sum of Diffs (MP4)	10.109	2.13	2.03	477
Max Filter (4x5)	9.338	9.60	7.78	835
Max Value in Image	12.830	12.8	10.6	320
Min Filter (3x4)	8.652	8.80	7.24	743
Multiply (Dyadic; 16-bit out)	8.713	4.35	4.13	800
Multiply (Monadic; 16-bit out)	8.911	8.88	6.59	686
Prewitt Magnitude	2.318	2.36	2.23	1,141
Roberts Magnitude	3.038	3.06	2.84	940
Sobel Magnitude	2.162	2.20	2.09	1,184
Square Root (fix8.4 out)	6.497	6.48	5.58	660
Subtract (Dyadic)	8.960	4.45	3.67	764
Subtract (Monadic)	9.492	9.43	6.88	641
Threshold (1-bit out)	8.577	8.55	3.84	693
Wavelet (5x5)	8.025	2.48	1.29	1,433

required is at least the maximum of the number of cycles required to read the data and the number of cycles required to process the data. (Since we are reading to and writing from separate memories, there is no read/write contention, and all of the routines in Table 1 produce less than or equal to the number of pixels they consume. As a result, these routines are input bound, not output bound.) Since the dyadic add operator has to read twice as many values as the monadic version, it has half the throughput. A similar effect happens with the monadic and dyadic *multiply* operators.

Many of the routines are variations of each other that demonstrate the some of the automatic compiler optimizations. For example, the *convolution* routine convolves an image with an unknown mask of known size. The source code for *convolution* is shown below. (It returns an unsigned 20-bit number to minimize the probability of overflow.) Restricting the mask to a known size is necessary for SA-C to know how big a circuit to make. It also allows the inner loop, which performs the dot product of the image window with the mask, to be unrolled. As shown in Table 1, the convolution routine can convolve an 8-bit image with a 3x3 8-bit mask (producing 20-bit results) at a rate of 9.7 megapixels per second.

```
uint20 [:,:] main (uint8 image[:,:], uint8
kernel[:,:])
{ // ***** computes the gradient for the  image
*****
  uint20 res[:,:] =
     for window win[3,3] in image
        { uint20 val =
             for elem1 in win dot elem2 in kernel
             return(sum((uint20)elem1*elem2));
        }
     return(array(val));
} return (res);
```

Gaussian Filter is a convolution with a fixed mask, which allows the compiler to apply constant propagation and constant folding. These optimizations eliminate many of the arithmetic operations inside the loop and produce smaller circuits. These optimizations decrease the size of the Gaussian Filter to 725 logic blocks (compared to 1,333 for general convolution). This does not significantly increase throughput, however, because both operations are I/O bound. However, the smaller circuit size allows the Gaussian filter to be stripmined, increasing the amount of parallelism and decreasing the total number of bytes read from memory[2]. With stripmining, the throughput of the *Gaussian Filter* increases by a factor of two to 18.0 megapixels per second. (Table 1 was generated with stripmining disabled.)

Another common relation is sequencing. The morphological *open* and *close* operators are combinations of the more primitive *erode* and *dilate* operators. The *open* operator, for example, is defined as dilation followed by erosion. (The *close* operator is the reverse.) The *open* operator is therefore a candidate for loop fusion: the dilation loop can be merged with the erosion loop that follows it. As a result, *open* takes only 79% of the time required to do *dilate* followed by a separate *erode*.

Loop fusion can have an even greater impact when one loop allows another to be simplified. The *Prewitt* operator in Table 1 is more than just convolution; it convolves the image with both horizontal and vertical edge masks, and then computes the magnitude of the results. As noted earlier, three quarters of the cost of the Prewitt operator lies in the magnitude computation. When *Prewitt* is followed by a threshold operator, the loops can be fused. When Prewitt is followed by a threshold set to pass any value that is 128 or greater, only the highest order bit of the square root result is needed. Therefore, large parts of the square root circuit can be removed, creating a smaller and faster circuit. This optimization improves the throughput of *Prewitt + Threshold* by 56%.

7. Conclusion

By modifying the C programming language and creating SA-C, we have created a high-level algorithmic programming language that can be mapped in a straightforward manner onto field programmable gate arrays (FPGAs). By using data flow graphs as the intermediate representation, we are able to optimize the resulting circuits. The end result is a system that allows programmers to quickly write image processing algorithms and map them onto reconfigurable computers.

8. Future Work

Work on the optimizing SA-C compiler continues. We are currently retargeting the compiler for the Annapolis Microsystems StarFire board, with a larger and faster Xilinx Virtex XL-1000 FPGA. We are also introducing optimizations that will 1) make the collectors more efficient, 2) pipeline the inner loop bodies when appropriate, and 3) more aggressively "nextify" (temporally stripmine) the loop bodies to reduce redundant computation and minimize the size and number of shift registers needed to implement sliding windows. (See [8] for a more thorough description of this optimization in the context of probing.) We are also automating the use of look-up tables, which is currently a manual process.

Acknowledgements

This work was funded by DARPA through AFRL under contract F3361-98-C-1319. We would also like to thank Steve Jorgenson of Khoral Research, Inc. for providing feedback about SA-C, and Pankaj Patil for generating Table 1.

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

[2] Vertical stripmining reduces input by using a value more often each time it is read. For example, in a 3x3 sliding window each pixel is read three times; if the same program is stripmined into a 4x3 window, each pixel is read only twice.

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] 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.

[6] Ellis, T., J. Phillips, and T. Lahey, *Fortran 90 Programming*. 1994: Addison-Wesley.

[7] 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.

[8] Rinker, R., et al. “Compiling Image Processing Application to Reconfigurable Hardware”, in *IEEE International Conference on Application-specific Architectures and Processors*. 2000. Boston.