One-step Compilation of Image Processing Applications to FPGAs [*]

A.P.W. Böhm, B. Draper, W. Najjar, J. Hammes, R. Rinker, M. Chawathe, C. Ross
Computer Science Department
Colorado State University
Ft. Collins, CO, U.S.A.

Abstract ***This paper describes a system for one-step compilation of image processing (IP) codes, written in the machine-independent, algorithmic, high-level single assignment language SA-C, to FPGA-based hardware. The SA-C compiler performs a variety of optimizations, some conventional and some specialized, before generating dataflow graphs and host code. The dataflow graphs are then compiled, via VHDL, to FPGA configuration codes. This paper introduces SA-C and describes the optimization and code generation stages in the compiler. The performance of a target acquisition prescreener (ARAGTAP), the Intel Image Processing Library, and an iterative tri-diagonal solver running on a reconfigurable system are compared to their performance on a Pentium PC with MMX.***

1 Introduction

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, e.g., real-time point tracking [9], stereo vision [40], color-based detection [10], image compression [22], and neural networks [14]. Unfortunately, the biggest obstacle to the more widespread use of reconfigurable computing systems lies in the difficulty of developing application programs. FPGAs are typically programmed using behavioral hardware description languages such as VHDL [31] and Verilog. These languages require great attention to detail such as timing issues and low level synchronization. However, application programmers are typically not trained in these hardware description languages and may be reluctant to learn them. They usually prefer a higher level, algorithmic programming language to express their applications.

[*]This work is supported by DARPA under US Air Force Research Laboratory contract F33615-98-C-1319.

The Cameron Project [21, 28] has created a high-level algorithmic language, named SA-C [20], for expressing image processing applications and compiling them to FPGAs. For a SA-C program the compiler provides one-step compilation to a ready-to-run host executable and FPGA configurations. The compiler uses dataflow graphs to perform a variety of program optimizations. While some of these transformations are conventional, others are novel and have been developed specifically for mapping to FPGAs.

After parsing and type checking, the SA-C compiler converts the program to a hierarchical graph representation called DDCF (Data Dependence and Control Flow) graphs. DDCF graphs are used in many optimizations, some general and some specific to SA-C and its target platform. After optimization, the program is converted to a combination of low-level dataflow graphs (DFGs) and host code. DFGs are then compiled to VHDL code, which is synthesized and place-and-routed to FPGAs by commercial software tools.

To aid in program development, the SA-C compiler has two other modes. In the first, the entire SA-C code is compiled to a host executable for traditional debugging. In the second, the DFGs are executed by a simulator for validation. Dataflow execution is animated, allowing the user to view the flow of data through the graph.

Figure 1 shows a high-level view of the system. The SA-C compiler can run in stand-alone mode, but it also has been integrated into the Khoros$^{(TM)}$ [25] graphical software development environment.

The rest of this paper presents an overview of the SA-C language in section 2. Compiler optimizations and pragmas are discussed in section 3. Translations to low-level dataflow graphs and then to VHDL are discussed in sections 4 and 5. The Cameron Project's test platform is described in section 6, and some applications with performance data are presented in section 7. References to related work are given in section 8, and section 9 concludes and describes future work.

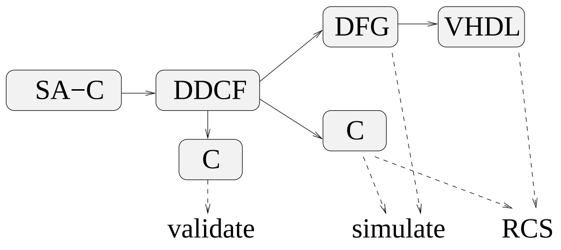

Figure 1. SA-C Compilation system.

2 The SA-C Language

The design goals of SA-C are to have a language that can express image processing applications elegantly, and to allow seamless compilation to reconfigurable hardware. Variables in SA-C are associated with wires, not with memory locations. SA-C is a single-assignment side effect free language; each variable's declaration occurs together with the expression defining its value. This avoids pointer and von Neumann memory model complications and allows for better compiler analysis and translation to DFGs. The IP applications that have been coded in SA-C transform images to images and are easily expressed using single assignment. Data types in SA-C include signed and unsigned integers and fixed point numbers, with user-specified bit widths. SA-C has multidimensional rectangular arrays whose extents can be determined either dynamically or statically. The type declaration int14 M[:,6] for example, is a declaration of a matrix M of 14-bit signed integers. The left dimension will be determined dynamically; the right dimension has been specified by the user.

The most important aspect of SA-C is its treatment of **for** loops and their close interaction with arrays. SA-C is expression oriented, so every construct including a loop returns one or more values. A loop has three parts: one or more generators, a loop body and one or more return values. The generators provide parallel array access operators that are concise and easy for the compiler to analyze. There are four kinds of loop generators: *scalar*, *array-element*, *array-slice* and *window*. The scalar generator produces a linear sequence of scalar values, similar to Fortran's **do** loop. The array-element generator extracts scalar values from a source array, one per iteration. The array-slice generator extracts lower dimensional sub-arrays (e.g. vectors out of a matrix). Finally, window generators allow rectangular sub-arrays to be extracted from a source array. All possible sub-arrays of the specified size are produced, one per iteration. Generators can be combined through **dot** and **cross** products. The **dot** product runs the generators in lock step, whereas **cross** products produce all combinations of components from the generators.

SA-C loops provide a simple and concise way of processing arrays in regular patterns, often making it unnecessary to create nested loops to handle multi-dimensional arrays or to refer explicitly to the array's extents or the loop's index variables. They make compiler analysis of array access patterns significantly easier than in C or Fortran, where the compiler must analyze index expressions in loop nests and infer array access patterns from these expressions. In SA-C, the index generators and the array references have been unified; the compiler can reliably infer the patterns of array access.

A loop can return arrays and reductions built from values that are produced in the loop iterations, including **sum**, **product**, **min**, **max**, **mean**, and **median**. The **histogram** operator returns a histogram of loop body values in a one-dimensional array.

```
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 h in H dot w in W dot v in V
          return (sum (h*w), sum (v*w));
      int16 magnitude =
        sqrt (dfdy*dfdy+dfdx*dfdx);
      } return (array (magnitude));
  } return (M);
```

Figure 2. Prewitt Edge detector code.

Figure 2 shows SA-C code for the Prewitt edge detector, a standard IP operator. The outer **for** loop is driven by the extraction of 3x3 sub-arrays from array Image. All possible 3x3 arrays are taken, one per loop iteration. Its loop body applies two masks to the extracted window W, producing a magnitude. An array of these magnitudes is collected and returned as the program's result. The shape of the return array is derived from the shape of Image and the loop's generator. If Image were a 100x200 array, the result array M would have a shape of 98x198.

Loop carried values are allowed in SA-C using the keyword next instead of a type specifier in a loop body. This indicates that an initial value is available outside the loop, and that each iteration can use the current value to compute a next value. The following code fragment shows an initial value of an array Y computed outside a loop driven by a scalar generator, array elements used to compute next elements inside the loop, and the final version of Y being returned by the loop. In general, the size of Y may change from iteration to iteration.

```
fix16.14 Y[:] = ...
fix16.14 res[:] =
 for uint8 i in [SIZE]
  next Y =  for y in Y return(array(f(y)));
 }return(final(Y));
```

3 Optimizations and pragmas

The compiler's internal program representation is a hierarchical graph form called the "Data Dependence and Control Flow" (DDCF) graph. DDCF subgraphs correspond to source language constructs. Edges in the DDCF express data dependencies, opening up a wide range of loop- and array-related optimization opportunities.

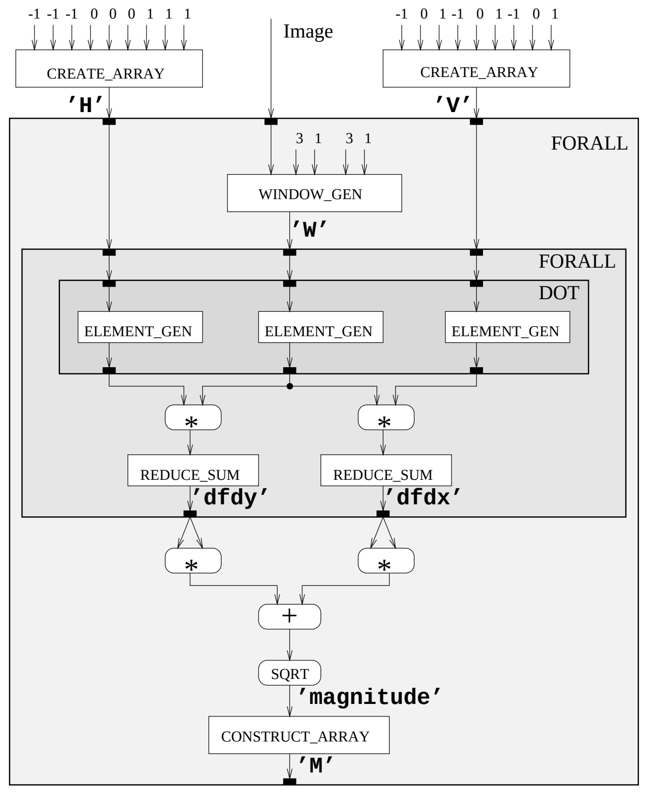

Figure 3. DDCF graph for Prewitt program.

Figure 3 shows the initial DDCF graph of the Prewitt program of figure 2. The FORALL and DOT nodes are compound, containing subgraphs. Black rectangles along the top and bottom of a compound node represent input ports and output ports. The outer FORALL has a single window generator. The WINDOW_GEN is operating on a two-dimensional image, so it requires window size and step inputs for each of the two dimensions. In this example, both dimensions are size three, with unit step sizes. The output of the WINDOW_GEN node is a 3x3 array that is passed into the inner FORALL loop. This loop has a DOT graph that runs three generators in parallel, each producing a stream of nine values from its source array. Each REDUCE_SUM node sums a stream of values to a single value. Finally, the CONSTRUCT_ARRAY node at the bottom of the outer loop takes a stream of values and builds an array with them.

Dataflow analysis is performed on the DDCF graph, and a number of conventional optimizations [3] are performed as DDCF-to-DDCF transformations, including Invariant Code Motion, Function Inlining, Switch Constant Elimination, Constant Propagation and Folding, Algebraic Identities, Dead Code Elimination and Common Subexpression Elimination. Another set of optimizations is more specific to the single assignment nature of SA-C and to the target hardware. Many of these are controlled by user pragmas.

Many IP operators involve fixed size and often constant convolution masks. A ***Size Inference*** pass propagates information about constant size loops and arrays through the dependence graph. This pass is vitally important in this compiler, since it exploits the close association between arrays and loops in the SA-C language. Source array size information can propagate through a loop to its result arrays (downward flow), and result array size information can be used to infer the sizes of source arrays (upward flow). In addition, since the language requires that the generators in a dot product have identical shapes, size information from one generator can be used to infer sizes in the others (sideways flow).

Effective size inference allows other optimizations to take place. Examples are Full Loop Unrolling and array elimination. Full Unrolling of loops is discussed next. Array elimination can occur in two contexts described below. In Array Value Propagation, when the array is fixed size and all references to elements have fixed indices, the array is broken up into its elements, i.e. storage is avoided and elements are represented as scalar values. In Loop Carried Array Elimination the array is again replaced by individual elements, but here the values are carried from one iteration to the next.

Full Unrolling of loops with small, compile time assessable numbers of iterations can be important when generating code for FPGAs, because it spreads the iterations in code space rather than in time. Small loops occur frequently as inner loops in IP codes, for example in convolutions with fixed size masks.

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. As will be shown in the Prewitt example, this optimization may remove entire user defined (or compiler generated) arrays. When all references to the array have constant indices, the array is eliminated completely.

Loop Carried Array Elimination The most efficient representation of arrays in loop bodies is to have their values reside in registers. This eliminates the need for array allocation in memory and array references causing delays. This requires that the array size be statically known. The important case is that of a loop carried array that changes values but not size during each iteration. To allocate a fixed number of registers for these arrays two requirements need to be met. 1) The compiler must be able to infer the size of the initial array computed outside the loop. 2) Given this size, the compiler must be able to infer that the next array value inside the loop is of the same size.

To infer sizes, the compiler clones the DDCF graph representing the loop, and speculates that the size of the array is as given by its context. It then performs analysis and tranformations based on this assumption. There are three possible outcomes of this process. If the size of the next array cannot be inferred, the optimization fails. If the size of the next array can be inferred, but is different from the initial size, it is shown that the array changes size dynamically, and the optimization fails again, but now for stronger reasons. If the size of the next array is equal to the initial size, the sizes of the arrays in all iterations have been proven, by induction, to be equal and the transformation is allowed, the important transformation being array elimination. This form of speculative optimization requires the suppression of global side effects, such as error messages.

N-dimensional Stripmining extends stripmining [39] and creates an intermediate loop with fixed bounds. The inner loop can be fully unrolled with respect to the newly created intermediate loop. As an example the following code shows the convolution of an image with a three by three mask:

```
uint8[:,:] Conv (uint8 I[:,:], uint8 M[3,3]) {
  uint16 Res[:,:] =
    for window W[3,3] in I {
      uint16 ip =
        for w in W dot m in M
          return (sum (w*m));
      } return (array (ip));
  } return (Res);
```

The inner loop computing ip gets unrolled. However, this unrolled loop is still rather small and it is often more efficient to execute a number of these inner loops in parallel. If the outer loop is stripmined with a user specified size of eight by eight, the compiler transforms it to the following:

```
uint16 Res[:,:] =
  for window WT[8,8] in I step (6,6) {
    uint16 ResTile[6,6] =
      for window W[3,3] in WT {
        uint16 ip =
          for w in W dot m in M
            return (sum (w*m));
        } return (array (ip) );
    } return (tile (ResTile));
```

This is important because the two nested inner loops can now be unrolled, generating a larger more parallel circuit. The **tile** loop return operator concatenates equal size N-dimensional sub-arrays into one N-dimensional array. The compiler generates code to compute left over fringes.

Some (combinations of) operators can be inefficient to implement directly in hardware. For example the computation of magnitude in Prewitt requires multiplications and square root operators. The evaluation of the whole expression can be replaced by an access to a ***Lookup Table***, which the compiler creates by wrapping a loop around the expression, recursively compiling and executing the loop, and reading the results.

The performance of many systems today, both conventional and specialized, is often limited by the time required to move data to the processing units. ***Fusion*** of producer-consumer loops is often helpful, since it reduces data traffic and may eliminate intermediate data structures. In simple cases, where arrays are processed element-by-element, this is straightforward. However, the windowing behavior of many IP operators presents a challenge. Consider the following loop pair:

```
uint8 R0[:,:] =
  for window W[2,2] in Image
      return (array (f (W)));
uint8 R1[:,:] =
  for window W[2,2] in R0
      return (array (g (W)));
```

If the Image array has extents d_0xd_1, then R0 will have extents (d_0-1)x(d_1-1) (determined by the number of 2x2 windows that can be referenced in Image.) Similarly, the extents of R1 will be (d_0-2)x(d_1-2). This means that these loops do not have the same number of iterations. Nevertheless, it is possible to fuse such a loop pair by examining their data dependencies. One element of R1 depends on a 2x2 sub-array of R0, and the four values in that sub-array together depend on a

3x3 sub-array of Image. It is possible to replace the loop pair with one new loop that uses a 3x3 window and has a loop body that computes one element of R1 from nine elements of Image. Sub-arrays of size 2x2 are extracted from the window, and each feeds a copy of the upper loop body f. The values emerging from the copies of f feed the lower loop body g. This kind of fusion can create large loop bodies since the loop body of the upper loop is replicated in the new loop. These space problems are tackled by Temporal CSE, as described next.

Common Subexpression Elimination (CSE) is an old and well known compiler optimization that eliminates redundancies by looking for identical subexpressions that compute the same value [3]. The redundancies are removed by keeping just one of the subexpressions and using its result for all the computations that need it. This could be called "spatial CSE" since it looks for common subexpressions within a block of code. The SA-C compiler performs conventional spatial CSE, but it also performs ***Temporal CSE***, looking for values computed in one loop iteration that were already computed in previous loop iterations. In such cases, the redundant computation can be eliminated by holding such values in registers so that they are available later and need not be recomputed.

Here is a simple example containing a temporal common subexpression:

```
for window W[3,2] in A {
  uint8 s0 = array_sum (W[:,0]);
  uint8 s1 = array_sum (W[:,1]);
  } return (array (s0+s1));
```

This code computes a separate sum of each of the two columns of the window, then adds the two. Notice that after the first iteration of the loop, the window slides to the right one step, and the column sum s1 in the first iteration will be the same as the column sum s0 in the next iteration. By saving s1 in a register, the compiler can eliminate one of the two column sums, nearly halving the space required for the loop body. This is the essence of Temporal CSE performed by the SA-C compiler. This optimization requires some special handling to take care of the problem of getting the registers "primed" with initial values, a level of detail that is beyond the scope of this paper.

A useful phenomenon often occurs with Temporal CSE: one or more columns in the left part of the window are unreferenced, making it possible to eliminate those columns. ***Narrowing*** the window lessens the FPGA space required to store the window's values. Figure 4 shows the dataflow graph that results after Temporal CSE and Window Narrowing have been applied to the column-sum example above. The register is produced by the TCSE step. Window narrowing then removes one column of the window.

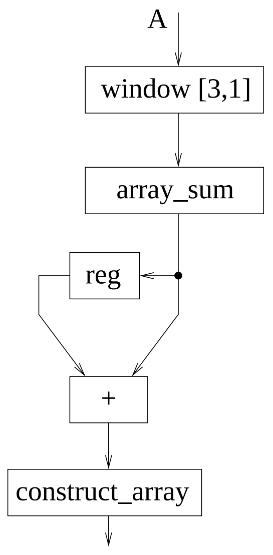

Figure 4. Effect of Temporal CSE and Window Narrowing.

Another optimization that sets the stage for window narrowing moves the computation of expressions fed by window elements into earlier iterations by moving the window references rightward, and uses a register delay chain to move the result to the correct iteration. This can remove references to left columns of the window, and allows window narrowing. This optimization trades window space for register delay chain space. Hence, whether this optimization actually provides a gain depends on the individual computation. We call this ***Window Compaction.***

In many cases the performance tradeoffs of various optimizations are not obvious; sometimes they can only be assessed empirically. The SA-C compiler allows many of its optimizations to be controlled by ***user pragmas*** in the source code. This makes it relatively painless for a user to experiment with different approaches and evaluate the space-time tradeoffs that inevitably exist.

After multiple applications of fusion and unrolling, the resulting loop often has a long critical path, resulting in a low clock frequency. Adding stages of ***pipeline registers*** can break up the critical path and thereby boost the frequency. The SA-C compiler uses propagation delay estimates, empirically gathered for each of the DFG node types, to determine the proper placement of pipeline registers. The user specifies the number of pipeline stages.

4 Conversion to DFGs

A dataflow graph (DFG) is a low-level, non-hierarchical and asynchronous program representation. DFGs are used for mapping SA-C program parts onto reconfigurable hardware. DFGs can be viewed as abstract hardware circuit diagrams without timing considerations taken into account. Nodes are operators and edges are data paths. Dataflow graphs allow token driven simulation, used by the compiler writer and applications programmer for validation and debugging.

After optimizations have been performed on a program's DDCF graph, the compiler searches for loops that meet the criteria for execution on reconfigurable hardware. The SA-C compiler attempts to translate every innermost loop to a DFG. The innermost loops the compiler finds may not be the innermost loops of the original program, as loops may have been fully unrolled or stripmined.

In the present system, not all loops can be translated to DFGs. The most important limitation is the requirement that the sizes of a loop's window generators be statically known.

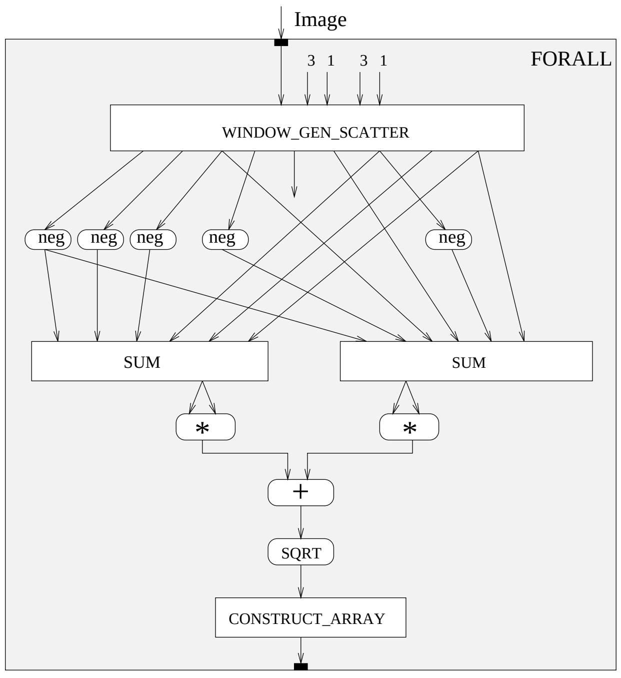

Figure 5. DFG for Prewitt after optimizations.

In the Prewitt program shown earlier, the DDCF graph is transformed to the DFG shown in figure 5. The SUM nodes can be implemented in a variety of ways, including a tree of simple additions. The window generator also allows a variety of implementations, based on the use of shift registers. The CONSTRUCT_ARRAY node streams its values out to a local memory or to the host, as appropriate.

5 Translation of DFGs to VHDL

The DFG to VHDL translator produces synchronous circuits, expressed in VHDL, from asynchronous DFGs [12]. Each DFG *node* corresponds to a VHDL operation; *edges* correspond to the input and output signals that the operation requires. The simple DFG nodes, such as the arithmetic and logical operators, are strictly combinational. Complex nodes like generator nodes and collector nodes are synchronous and contain registers or state machines that use a clock.

The translator handles the two types of nodes differently. The combinational nodes form the inner loop body (ILB) of the circuit, and are translated either into a single VHDL statement or into the instantiation of a pre-built VHDL component. Complex nodes are implemented by selecting the proper VHDL component from a library of pre-built modules; they are parameterized by information gleaned from the DFG. These clock driven nodes are connected together into a "wrapper" around the ILB, resulting in the structure shown in Figure 6. For each iteration of the loop, the generator produces data at the top of the ILB, and the ILB's output is available a short time later, the exact time being determined by the circuit propagation delay. This delay determines the maximum execution frequency of the resulting circuit.

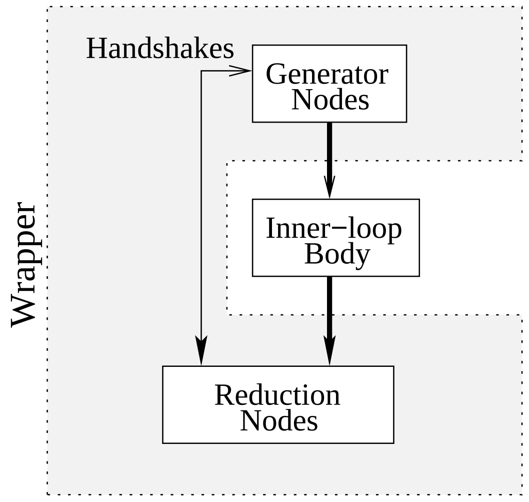

Figure 6. Abstract System Structure

A generator node retrieves the input data in blocks from the external memory in the order required by the ILB, buffers it, and presents it at the proper time to the inputs of the ILB. A state machine controls the generation of the proper memory access and buffer signals, and provides the timing for the operation of the entire circuit. In the case of a "2D window-generator" node, a 2D buffer array is instantiated, of the same size as the window specified in the generator node. This buffer is formed as a shift register; after the current window of data has been used by the ILB, the oldest column is shifted out, and a new column is shifted in, to form the next window of data. This shift register operation produces the "sliding window" effect required by the SA-C language. Other types of generators provide 1D windows, single array element, and scalars to the ILB in a similar fashion. The generator code also handles the prefetching of data from memory, so the buffers can be kept full and the ILB can be kept as busy as possible.

In a complementary fashion, the reduction nodes take the output from the ILB, buffer this output, and write completed buffers to memory. As with the input, ILB output may be a single element, several single elements, or tiles (sub-arrays.)

The translator selects the proper VHDL components for the generator and reductions and creates VHDL glue logic that "wires" the components together. A parameter file, which specifies the sizes and shapes of the various buffers, as well as any additional timing information required to synchronize the wrapper components, is also created. Finally, the translator generates the script files needed by the commercial VHDL compiler and Place-and-Route tools to synthesize the final design.

6 System Description

As stated earlier, the SA-C system can be used in three modes:

1. Compile the entire program to a host executable. This is useful in the early stages of code development, for quick compiles and functional debugging.

2. Compile the program to dataflow graphs, called by a host executable. The system has a token-driven dataflow simulator that allows validation of the dataflow graphs produced by the compiler. There is an optional "view" mode that displays the DFGs and allows the user to single-step the execution and watch the values flowing through the graph, windows stepping through input images, and creation of result images..

3. Compile the program to FPGA codes, called by a host executable. The Cameron compiler produces VHDL code [13] that is compiled and place-and-routed by commercial software tools.

The SA-C run-time system has I/O formats that are compatible with standard image processing formats PBM, PGM and PPM. This means that after compilation to host and FPGA codes, the program is ready to run immediately on standard image files.

The current test platform in Cameron is the StarFire Board, produced by Annapolis Microsystems [4], which has a single XCV1000-BG560-4 Virtex FPGA made by Xilinx [41]. The board contains six local memories of one megabyte each. Each of the memories is addressable in 32 bit words; all six can be used simultaneously if needed. The StarFire board is capable of operating at frequencies from 25 MHz to 180 MHz. It communicates over the PCI bus with the host computer at 66 MHz. In our system, the board is housed in a 266-MHz Pentium-based PC.

7 Applications

This section reports the performance of SA-C codes running on the StarFire system described earlier.

7.1 Intel Image Processing Library

When comparing simple IP operators one might write corresponding SA-C and C codes and compare them on the Starfire and Pentium II. However, neither the Microsoft nor the Gnu C++ compilers exploit the Pentium's MMX technology. Instead, we compare SA-C codes to corresponding operators from the Intel Image Processing Library (IPL). The Intel IPL library consists of a large number of low-level Image Processing operations. Many of these are simple point- (pixel-) wise operations such as square, add, etc. These operations have been coded by Intel for highly efficient MMX execution. Comparing these on a 450 Mhz Pentium II (with MMX) to SA-C on the StarFire, the StarFire is 1.2 to six times slower than the Pentium. This result is not surprising. Although the FPGA has the ability to exploit fine grain parallelism, it operates at a much slower clock rate than the Pentium. These simple programs are all I/O bound, and the slower clock of the FPGA is a major limitation when it comes to fetching data from memory. However, the Prewitt edge detector written in C using IPL calls and running on the 450 Mhz Pentium II, takes 0.053 seconds as compared to 0.017 seconds when running the equivalent SA-C code on the StarFire. Thus, non I/O bound SA-C programs running on the StarFire board are competitive with their hand optimized IPL counterparts.

7.2 ARAGTAP

SA-C running on the StarFire compares even better when we look at more complex operations. The ARAGTAP pre-screener [33] was developed by the U.S. Air Force at Wright Labs as the initial focus-of-attention mechanism for a SAR automatic target recognition application. Aragtap's components include down sampling, dilation, erosion, positive differencing, majority thresholding, bitwise And, percentile thresholding labeling, label pruning, and image creation. All these components, apart from image creation, have been written in SA-C. 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. Four of these dilations are with a 3x3 mask of 1's (the shape of a square), the other four are with a 3x3 mask with 1's in the shape of a cross and zeros at the edges.

First, the compiler can fuse dilate and erode loops in groups of four. (Fusing loop sequences longer than four currently brings the frequency of the resulting FPGA configurations below 25 Mhz, the minimum frequency required by the StarFire.) The dilate and erode loops also allow temporal CSE, window compaction, and window narrowing. Combined with the fusion of four loops, this results in an FPGA code driven by a 9x1 window generator.

A straight C implementation of ARAGTAP running on the Pentium takes 1.07 seconds. Running down sampling, eight unfused dilations, and eight unfused erosions, positive differencing, majority thresholding, and bitwise And on the StarFire board and the rest of the code on the PC takes 0.096 seconds, a more than ten fold speedup over the Pentium. Fusing erode and dilate loops into groups of four brings the execution time down to 0.065 seconds: about 50% gain over the unfused FPGA execution time, and a sixteen fold speedup over the Pentium.

7.3 An Iterative Tri-Diagonal Solver

```
fix16.14 X[:] = DiB; // initial X = D_inverse.B
fix16.14 res[:] =
  for uint8 iter in [2*SIZE] {
   fix16.14 Xperim[:] =
    for uint8 i in [SIZE+2]
    return(array(i==0 || i==(SIZE+1)
                 ? (fix16.14)(0.0): X[i-1]));
   // next X = D_inverse.B - D_inverse.(L+U).X
   next X =
    for b in DiB dot l in DiL dot u in DiU
                   dot window WX[3] in Xperim
    return(array(b - (l*WX[0] + u*WX[2])));
  }return(final(X));
```

Solving systems of tri-diagonal matrices is important in many applications in e.g. earth sciences. A tri-diagonal system $Ax = b$, where $A = L + D + U$ and L, D and U are a lower diagonal, diagonal and upper diagonal matrices, respectively, can be solved iteratively: $x_n = D^{-1}b - D^{-1}(L + U)x_{n-1}$.

The SA-C code above shows this iterative process. SIZE is a compile time constant. The context of this loop allows the size of the initial value of X to be inferred to be equal to SIZE. The compiler then infers the size of next X to be SIZE also, so array elimination can occur and the loop computing next X can be fully unrolled, allowing the loop computing res to be run on the FPGA. A SIZE = 8 problem runs in 9 microseconds on the Pentium and 18 times faster, in .5 microseconds, on the StarFire.

8 Related work

Hardware and software research in reconfigurable computing is active and ongoing. Hardware projects fall into two categories – those using commercial off-the-shelf components (e.g. FPGAs), and those using custom designs.

The Splash-2 [11] is an early (circa 1991) implementation of an RCS, built from 17 Xilinx [42] 4010 FPGAs, and connected to a Sun host as a co-processor. Several different types of applications have been implemented on the Splash-2, including searching[23, 32], pattern matching[35], convolution [34] and image processing [6].

Representing the current state of the art in FPGA-based RCS systems are the AMS WildStar[5] and the SLAAC project [36]. Both utilize Xilinx Virtex [41] FPGAs, which offer over an order of magnitude more programmable logic, and provide a several-fold improvement in clock speed, compared to the earlier chips.

Several projects are developing custom hardware. The Morphosys project [27] marries an on-board RISC processor with an array of reconfigurable cells (RCs). Each RC contains an ALU, shifter, and a small register file.

The RAW Project [38] also uses an array of computing cells, called *tiles*; each tile is itself a complete processor, coupled with an intelligent network controller and a section of FPGA-like configurable logic that is part of the processor data path. The PipeRench [18] architecture consists of a series of *stripes*, each of which is a pipeline stage with an input interconnection network, a lookup-table based PE, a results register, and an output network. The system allows a virtual pipeline of any size to be mapped onto the finite physical pipeline.

On the software front, a framework called "Nimble" compiles C codes to reconfigurable targets where the reconfigurable logic is closely coupled to an embedded CPU [26]. Several other hardware projects also involve software development. The RAW project includes a significant compiler effort [2] whose goal is to create a C compiler to target the architecture. For PipeRench, a low-level language called DIL [17] has been developed for expressing an application as a series of pipeline stages mapped to stripes.

Several projects (including Cameron) focus on hardware-independent software for reconfigurable computing; the goal – still quite distant – is to make development of RCS applications as easy as for conventional processors, using commonly known languages or application environments. Several projects use C as a starting point. DEFACTO [19] uses SUIF as a front end to compile C to FPGA-based hardware. Handel-C [1] both extends the C language to express important hardware functionality, such as bit-widths, explicit timing parameters, and parallelism, and limits the language to exclude C features that do not lend themselves to hardware translation, such as random pointers. Streams-C [15, 16] does a similar thing, with particular emphasis on extensions to facilitate the expression of communication between parallel processes. SystemC [37] and Ocapi [24] provide C++ class libraries to add the functionality required of RCS programming to an existing language.

Finally, a couple of projects use higher-level application environments as input. The MATCH project [7, 8, 30] uses MATLAB as its input language – applications that have already been written for MATLAB can be compiled and committed to hardware, eliminating the need for re-coding them in another language. Similarly, CHAMPION [29] is using Khoros [25] for its input – common glyphs have been written in VHDL, so GUI-based applications can be created in Khoros and mapped to hardware.

9 Conclusions and Future Work

The Cameron Project has created a language, called SA-C, for one-step compilation of image processing applications that target FPGAs. Various optimizations, both conventional and novel, have been implemented in the SA-C compiler. These optimizations are focused on reducing execution time and/or reducing FPGA space. The system has been used to implement some simple primitive IP operations, such as the routines from the Intel IPL, as well as more comprehensive applications, such as the ARAGTAP target acquisition prescreener.

Performance evaluation of the SA-C system has just begun. As performance issues become clearer, the system will be given greater ability to evaluate various metrics including code space, memory use and time performance, and to evaluate the tradeoffs between conventional functional code and lookup tables.

More compiler optimizations are under way, including further manipulations of window generators. Some more optimizations at the Dataflow graph level are also being implemented. One optimization involves the replacement of delay-intensive portions of the DFG with lookup tables, which often trade FPGA space for a more time-efficient solution.

Also, stream data structures are being added to the SA-C language, which will allow multiple cooperating processes to be mapped onto FPGAs.

References

[1] OXFORD hardware compiler group, the Handel Language. Technical report, Oxford University, 1997.

[2] A. Agarwal, S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar, and M. Srikrishna, D.and Taylor. The RAW compiler project. In *Proc. Second SUIF Compiler Workshop*, August 1997.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[4] Annapolis Micro Systems, Inc., Annapolis, MD. *WILDFORCE Reference Manual*, 1997. www.annapmicro.com.

[5] Annapolis Micro Systems, Inc., Annapolis, MD. *STARFIRE Reference Manual*, 1999. www.annapmicro.com.

[6] P. M. Athanas and A. L. Abbott. Processing images in real time on a custom computing platform. In R. W. Hartenstein and M. Z. Servit, editors, *Field-Programmable Logic Architectures, Synthesis, and Applications*, pages 156–167. Springer-Verlag, Berlin, 1994.

[7] P. Banerjee et al. A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems. In *The 8th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2000.

[8] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Chang, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, and M. Walkden. MATCH: a MATLAB compiler for configurable computing systems. Technical Report CPDC-TR-9908-013, Center for Parallel and distributed Computing, Northwestern University, August 1999.

[9] A. Benedetti and P. Perona. Real-time 2-d feature detection on a reconfigurable computer. In *IEEE Conference on Computer Vision and Pattern Recognition*, Santa Barbara, CA, 1998.

[10] D. Benitez and J. Cabrera. Reactive computer vision system with reconfigurable architecture. In *International Conference on Vision Systems*, Las Palmas de Gran Canaria, Spain, 1999.

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

[12] M. Chawathe. Dataflow graph to VHDL translation. Master's thesis, Colorado State University, 2000.

[13] M. Chawathe, M. Carter, C. Ross, R. Rinker, A. Patel, and W. Najjar. Dataflow graph to VHDL translation. Technical report, Colorado State University, Dept. of Computer Science, 2000.

[14] J. Eldredge and B. Hutchings. Rrann: A hardware implementation of the backpropagation algorithm using reconfigurable fpgas. In *IEEE International Conference on Neural Networks*, Orlando, FL, 1994.

[15] M. Gokhale. The Streams-C Language, 1999. `www.darpa.mil/ito/psum19999/F282-0.html`.

[16] M. Gokhale et al. Stream oriented PFGA computing in Streams-C. In *The 8th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2000.

[17] S. C. Goldstein and M. Budiu. *The DIL Language and Compiler Manual*. Carnegie Mellon University, 1999. `www.ece.cmu.edu/research/piperench/dil.ps`.

[18] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. Piperench: A coprocessor for streaming multimedia acceleration. In *Proc. Intl. Symp. on Computer Architecture (ISCA '99)*, 1999. `www.cs.cmu.edu/~mihaib/research/isca99.ps.gz`.

[19] M. Hall, P. Diniz, K. Bondalapati, H. Ziegler, P. Duncan, R. Jain, and J. Granacki. DEFACTO: A design environment for adaptive computing technology. In *Proc. 6th Reconfigurable Architectures Workshop (RAW'99)*. Springer-Verlag, 1999.

[20] J. Hammes and W. Böhm. *The SA-C Language - Version 1.0*, 1999. Document available from `www.cs.colostate.edu/cameron`.

[21] J. Hammes, R. Rinker, W. Böhm, and W. Najjar. Cameron: High level language compilation for reconfigurable systems. In *PACT'99*, Oct. 1999.

[22] R. Hartenstein et al. A reconfigurable machine for applications in image and video compression. In *Conference on Compression Technologies and Standards for Image and Video Compression*, Amsterdam, Holland, 1995.

[23] D. Hoang. Searching genetic databases on Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–192. CS Press, Loa Alamitos, CA, 1993.

[24] IMEC. Ocapi overview. `www.imec.be/ocapi/`.

[25] K. Konstantinides and J. Rasure. The Khoros software development environment for image and signal processing. In *IEEE Transactions on Image Processing*, volume 3, pages 243–252, May 1994.

[26] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *Proc. 37th Design Automation Conference*, 1999.

[27] G. Lu, H. Singh, M. Lee, N. Bagherzadeh, and F. Kurhadi. The Morphosis parallel reconfigurable system. In *Proc. of EuroPar 99*, Sept. 1999.

[28] W. Najjar. The Cameron Project. Information about the Cameron Project, including several publications, is available at the project's web site, `www.cs.colostate.edu/cameron`.

[29] S. Natarajan, B. Levine, C. Tan, D. Newport, and D. Bouldin. Automatic mapping of Khoros-based applications to adaptive computing systems. Technical report, University of Tennessee, 1999. Available from `http://microsys6.engr.utk.edu:80/~bouldin/darpa/mapld2/mapld_paper.pdf`.

[30] S. Periyayacheri, A. Nayak, A. Jones, N. Shenoy, A. Choudhary, and P. Banerjee. Library functions in reconfigurable hardware for matrix and signal processing operations in MATLAB. In *Proc. 11th IASTED Parallel and Distributed Computing and Systems Conf. (PDCS'99)*, November 1999.

[31] D. Perry. *VHDL*. McGraw-Hill, 1993.

[32] D. V. Pryor, M. R. Thistle, and N. Shirazi. Text searching on Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 172–178. CS Press, Loa Alamitos, CA, 1993.

[33] S. Raney, A. Nowicki, J. Record, and M. Justice. Aragtap atr system overview. In *Theater Missile Defense 1993 National Fire Control Symposium*, Boulder, CO, 1993.

[34] N. K. Ratha, D. T. Jain, and D. T. Rover. Convolution on Splash 2. In *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pages 204–213. CS Press, Loa Alamitos, CA, 1995.

[35] N. K. Ratha, D. T. Jain, and D. T. Rover. Fingerprint matching on Splash 2. In *Splash 2: FPGAs in a Custom Computing Machine*, pages 117–140. IEEE CS Press, 1996.

[36] B. Schott, S. Crago, C. C., J. Czarnaski, M. French, I. Hom, T. Tho, and T. Valenti. Reconfigurable architectures for systems level applications of adaptive computing. Available from `http://www.east.isi.edu/SLAAC/`.

[37] SystemC. SystemC homepage. `www.systemc.org/`.

[38] 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. *Computer*, September 1997.

[39] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.

[40] J. Woodfill and B. v. Herzen. Real-time stereo vision on the parts reconfigurable computero. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 1997.

[41] Xilinx, Inc. *Virtex 2.5V Field programmable Gate Arrays: Preliminary Product Description*, Oct. 1999. `www.xilinx.com`.

[42] Xilinx, Inc., San Jose, CA. *The Programmable Logic Databook*, 1998. `www.xilinx.com`.