Cameron: High Level Language Compilation for Reconfigurable Systems

Jeff Hammes, Bob Rinker, Wim Böhm, Walid Najjar,
Bruce Draper, Ross Beveridge
Department of Computer Science, Colorado State University

Abstract

This paper presents the Cameron Project [1]*, which aims to provide a high level, algorithmic language and optimizing compiler for the development of image processing applications on Reconfigurable Computing Systems (RCSs). SA-C, a single assignment variant of the C programming language, is designed to exploit both coarse-grain and fine-grain parallelism in image processing applications. Khoros, a software development environment commonly used for image processing, has been modified to support SA-C program development.*

SA-C supports image processing with true multidimensional arrays, and with sophisticated array access and windowing mechanisms. Reduction operators such as medians and histograms are also provided. The optimizing compiler targets RCSs, which are fine-grained parallel processors made up of Field Programmable Gate Arrays (FPGAs), memories and interconnection hardware. They can be used as inexpensive co-processors with conventional workstations or PCs. This paper discusses compiler optimizations to generate optimal FPGA code using dataflow analysis techniques applied to data dependence graphs. Initial results are presented.

Keywords: Image Processing, Single Assignment, Reconfigurable Systems, FPGAs.

1. Introduction

A common software development methodology in image processing (IP) uses a graphical programming environment where application programs are constructed by interconnecting the outputs of one primitive operator to the inputs of others. One of the most widely used graphical programming environments for image processing is Khoros$^{(TM)}$ [11], but other environments exist or are being developed for this purpose as well, including the Image Understanding Environment (IUE) [6] and CVIPtools [10]. These programming environments separate applications-level programming from low-level image processing and/or computer vision programming, and even lower level machine-dependent parallel programming. They can also distribute programs across multiple processors or assign processes to special purpose co-processors. By insulating the application programmer from low-level programming details, these environments allow domain experts to develop useful image processing applications.

This paper presents the Cameron Project, which aims to provide a high level, algorithmic language and optimizing compiler for the development of image processing algorithms on reconfigurable computing systems. It includes a language, Single Assignment C (SA-C), that is integrated into the Khoros programming environment, and that is compiled to host code and FPGA configurations. The compilation, simulation and execution path, shown in figure 1, uses Data Dependence Control Flow (DDCF) graphs as an intermediate form for optimizations. Code partitioning between host and RCS is directed by the user; the RCS code is converted to low-level dataflow graphs (DFGs) that are then translated via VHDL to FPGA codes. The system allows executable code to be generated at various stages of the compilation process for validation and simulation purposes. First, the complete DDCF graph can be translated to C and run on conventional workstations. Second, the DFGs can be simulated and animated along with host code to allow additional testing as well as collection of performance measures.

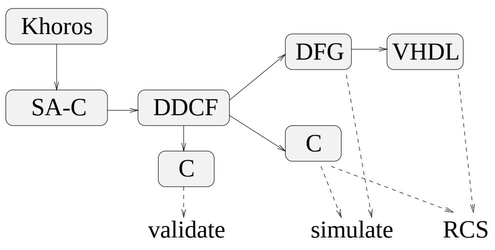

Figure 1. Compilation and Execution Path.

[1]The Cameron Project is supported by DARPA under US Air Force Research Laboratory contract F33615-98-C-1319

SA-C is a programming language based on C that has been developed at Colorado State University and integrated in the Khoros graphical programming environment by Khoral Research Inc. (KRI). The goal is to write low-level image processing algorithms in SA-C inside the Khoros software development environment and then to manipulate these programs as glyphs inside Cantata (the Khoros GUI). Glyphs written in SA-C can then be executed on parallel architectures, and in particular on Reconfigurable Computing Systems (RCSs) using Field Programmable Gate Arrays (FPGAs).

Image Processing (IP) applications feature large, regular image data structures with regular access patterns and therefore can benefit from parallel implementations. The programmable nature of FPGA-based parallel systems allows great flexibility, and promises massive fine-grain parallelism and high throughput. RCSs are therefore interesting candidates for use as special purpose IP acceleration hardware. Currently, FPGAs are difficult to use in this context because they are programmed using hardware description languages. The goal is to make FPGAs available to IP experts, as opposed to circuit designers.

SA-C offers a powerful set of language features supporting image operations, including:

- variable-precision integer and fixed point numbers and operators
- true multidimensional arrays
- array sectioning, such as slicing vectors and submatrices out of matrices
- windowing: selecting a stream of sub-matrices of a certain size from a matrix
- loops that use slicing or windowing to drive their loop bodies
- reduction operators, such as histogram and accumulation primitives

On the other hand, SA-C bans recursion and pointer manipulation, and allows each variable to be assigned only once. These restrictions allow data dependencies to be analyzed, enabling compiler optimizations and parallel code generation.

Dataflow graphs are translated via VHDL to configuration codes to be executed on the RCS. The part of the program that runs on the host initializes the RCS by configuring it with the appropriate code, transfers input data to it, and, upon completion of the RCS code, transfers result data back. The RCS code is partitioned into data distribution, computation, and output sections. The data distribution section takes data from a local memory or input FIFO and sends it into the computation section. The computation section is either completely combinatorial, or uses lookup tables for complex operations. The output section writes data to the local memories or communicates with the output FIFOs.

This paper describes the entire compilation path using the Prewitt edge detection algorithm as an example. The next section describes Reconfigurable Systems. Section three introduces SA-C. Section four details the compilation and optimization phase based on DDCF. Section five discusses the mapping to reconfigurable hardware. Section six concludes and discusses future work.

2. Reconfigurable Systems

Reconfigurable computing systems are typically based on FPGAs, which are large arrays of programmable logic cells, organized into one or more arrays of Configurable Logic Blocks (CLBs). In most FPGAs, the perimeter has I/O cells that interface with the external pins of the chip. Multiple FPGAs can be packaged with local memory and/or co-processors on boards such as the Wildforce$^{(TM)}$, built by Annapolis Microsystems[15].

Although the clock speeds of current FPGAs are slower than RISC processor clocks, the potential massive parallelism in an FPGA makes them good candidates for many real-time image processing tasks. DeHon calculates that, even with their lower clock speeds, FPGAs may have an order of magnitude better *computational density* (the number of bit operations a device can perform per unit of area-time) as compared with RISC CPUs [3]. Petersen and Hutchings specifically consider digital signal processing tasks, and also calculate a ten-fold speed-up [7].

Reconfigurable computing presents new challenges to language designers and compiler writers. The task of programming and compiling applications currently consists of partitioning the algorithm between a host processor and reconfigurable modules, and devising ways of producing efficient FPGA configurations for each piece of code. Presently, FPGAs are programmed in hardware description languages, such as VHDL or Verilog. While such languages are suitable for chip design, they are poorly suited for the kind of algorithmic expression that takes place in applications programming.

To enable programmers to develop IP applications for reconfigurable hardware, the Cameron project adds the following to the Khoros environment:

- a parallel programming language for expressing IP applications
- an optimizing and parallelizing compiler for generating dataflow graphs
- a translator for mapping the dataflow graphs to VHDL for FPGA-based, reconfigurable systems

With these extensions, the IP application programmer will be able to exploit reconfigurable computing hardware through a high level programming interface. The remainder of this paper will discuss the development of each of these three components.

3. SA-C

SA-C[2] combines features of existing imperative and functional languages in order to create a language well suited to image processing and amenable to compiler analysis and optimization. The language is intended to exploit both coarse-grain (loop-level) and fine-grain (instruction-level) parallelism, as appropriate for the target architecture. The language should be suitable for a variety of parallel platforms such as symmetric multiprocessors, networks of workstations, and vector computers. However, the principal target architecture of interest to the Cameron Project is the RCS.

SA-C is based on C in order to be as intuitive as possible to image processing experts, most of whom program in C or C++. That said, it differs from C in some important ways. It is an *expression-oriented*, functional language. Its scalar types include signed and unsigned integers and fixed point numbers with specified bit widths. For example, the type `uint12` represents a twelve-bit unsigned integer. It has no explicit pointers, and is non-recursive. It has true multi-dimensional arrays, including array sections similar to those in Fortran 90. For example, "`int10 A[:,:]`" declares a variable-length two dimensional array of ten-bit integers, and "`int10 A[0,:]`" is its first row. It has powerful loop generators and return operators similar to those in the Sisal language[5]. It has multiple-value returns and assignments.

The elimination of pointers and recursion and the single-assignment restriction enable important compiler code optimizations. In compensation for these restrictions, there are powerful high-level constructs to create and access arrays in concise ways. Loops and arrays are at the heart of the language, and are closely interrelated. Loops have special forms designed to work with arrays, and arrays are easily created as return values of loops. The parallel `for` loop is the source of coarse-grain parallelism, and has three parts: one or more *generators*, a loop *body*, and one or more *return* values. There are three kinds of loop generators: *scalar*, *array-component* and *window*. The scalar generator produces a linear sequence of scalar values, similar to Fortran's `do` loop. The array-component and window generators extract components of arrays in various ways. Array-component or slice generators extract lower dimensional sub-arrays (e.g. vectors out of a matrix). Window generators allow a rectangular window to traverse the source array, with a rank equal to the rank of the source, but with smaller extents.

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.

Loop generators provide a simple and concise way of processing arrays in regular patterns, often making it unnecessary to create loop nests to handle multi-dimensional arrays or to refer explicitly to the array's extents or the loop's index variables. Also, they make compiler analysis of array access patterns significantly easier. In C or Fortran, 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.

Every construct, and in particular every loop, returns one or more values. In addition to returning scalar values, 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. As an example, figure 2 shows SA-C code for the Prewitt edge detector.

```
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.

4. Compilation and Optimization

At the heart of the compiler is the "Data Dependence and Control Flow" (DDCF) graph, an intermediate form similar to the Sisal compiler's IF1 [8]. DDCF is a hierarchical dataflow graph, in which the high-level subgraphs

[2]SA-C (pronounced "sassy"), is not the only Single Assignment C. As an example, the Sac project[9], attempts to combine the imperative and functional programming styles.

have a direct correspondence to the constructs of the source language. This dataflow form exposes data dependencies, opening up a wide range of loop- and array-related optimization opportunities. The `for` loop generators produce data streams, bridging the gap between a memory-reading execution model and a data-driven stream model that more closely resembles the kind of execution that takes place on FPGAs. Similarly, the `array` loop-return operator produces array elements in storage order, making it easy to stream results back into memory in a straightforward way.

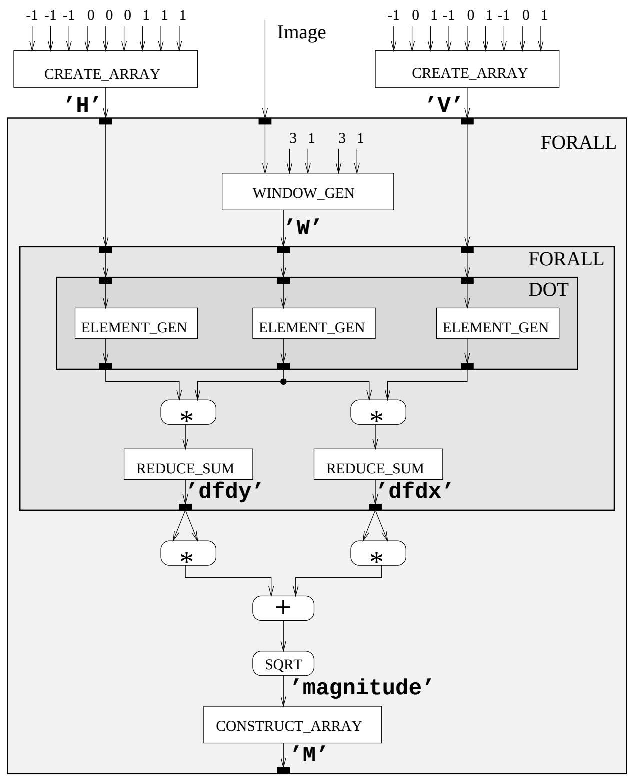

Figure 3. DDCF graph for Prewitt program.

Figure 3 shows the initial DDCF graph of the Prewitt program of figure 2. The FORALL and DOT nodes are compound, containing subgraphs. The black rectangles along the top of a compound node represent the node's input ports; the rectangles along the bottom are the 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 3×3 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. The array's shape is derived from the loop's generator; in this case, an $n \times m$ input image will produce a $(n-2) \times (m-2)$ result array.

Data flow analysis is performed on the DDCF graph. Information can travel with the flow (downward), against the flow (upward), and in certain cases sideways (see size inference below). A number of conventional optimizations [1] are performed as DDCF-to-DDCF transformations:

- **Code Motion.** Loop-invariant code is moved outside of the loop body.
- **Function Inlining.** Inlining is important for FPGA target hardware since there is no function calling mechanism in such hardware.
- **Switch Constant Elimination.** When the selection expression of a switch (or conditional) is known at compile time, the switch is replaced by the specified expression. This is often enabled by constant propagation and constant folding.
- **Constant Propagation and Folding.** Operators with constant inputs are computed at compile time.
- **Algebraic Identities.** Addition with zero, multiplication with zero or one, etc, are removed and replaced by the appropriate value.
- **Dead Code Elimination.** Dead code is eliminated. Such code may originate in the source program or be produced by the application of other optimizations.
- **Common Subexpression Elimination.** This optimization eliminates redundant expressions, thereby conserving FPGA resources.

Another set of optimizations is more specific to the application domain, to the single assignment nature of SA-C, and to the target hardware. IP operators often involve fixed size, 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. When a loop executes, it has a shape that is determined by the loop's generators, and the loop's shape in turn determines the shapes of any arrays it produces. This means that source array size information can propagate through a loop and into 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 individual generators enclosed in a dot product graph have identical shapes, size information from one generator can be used to infer sizes in the others (sideways flow).

Full Loop Unrolling of loops with small, compile time assessable numbers of iterations is very important when generating code for FPGAs, because iteration on such hardware is generally impractical and fine grain, pipelined, parallelism can be exploited by completely laying out the loop as a circuit. These small loops occur frequently as inner loops in IP codes, for example implementing convolutions with fixed size masks. This is also important for N-dimensional stripmining discussed later.

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 other optimizations may be enabled. As will be shown in the Prewitt example, this optimization may remove entire user defined (or compiler generated) arrays.

It has already been argued that it is important to know the sizes of loops and arrays at compile time so that these can be laid out as fixed circuits. **N-dimensional Stripmining** extends stripmining [13] 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 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:

```
uint8[:,:] Conv(uint8 I[:,:], uint8 M[3,3]){
  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));
}return(Res);
```

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. Since the stepping may give rise to left over fringes, much as in vectorization, the compiler generates code to compute them.

Some (combinations of) operators can be inefficient to implement directly in hardware. For example the computation of *magnitude* in Prewitt:

```
int16 magnitude = sqrt(dfdy*dfdy+dfdx*dfdx);
```

requires multiplications and square root operators. The evaluation of the whole expression can be replaced by an access to a "magnitude" **Lookup Table**. The compiler creates the lookup table by wrapping a loop around the expression, recursively compiling and executing the loop, and reading the results back in. Both the stripmining and lookup table optimizations are controlled by user pragmas.

The code motion and function inlining phases are performed once. The remaining phases are performed in multiple sweeps, since there is a cyclic effect where one optimization can enable another. The sweeps are repeated until a sweep produces no change.

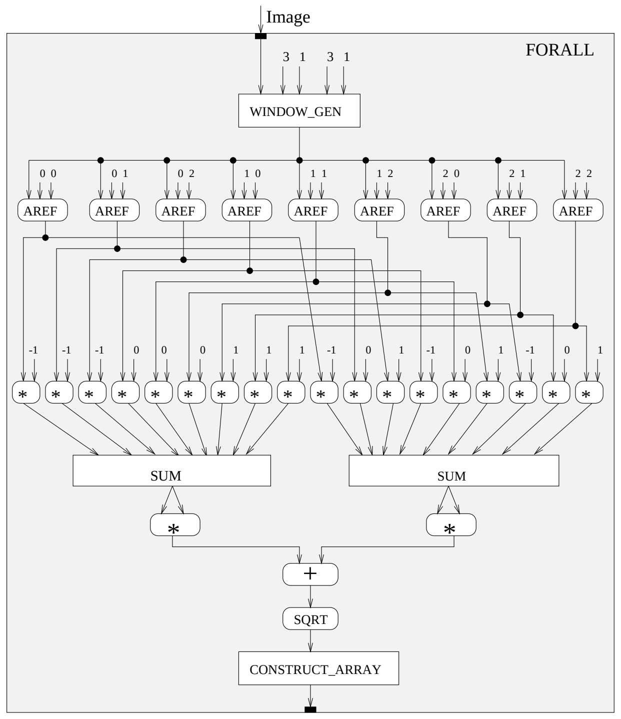

Figure 4. DDCF graph for Prewitt program after loop unrolling and array value propagation.

In the Prewitt program, array and loop size inference allows the compiler to determine that the inner loop will execute nine times, with a 3×3 shape. Since the number of loop iterations is known and is not large, the inner loop can be fully unrolled. Each of the three `ELEMENT_GEN` nodes becomes nine array reference nodes with constant indices, and the loop body becomes eighteen multiplications with two large `SUM` nodes taking in their values. The array value propagation sweep then replaces eighteen of the array reference nodes (those referencing the H and V arrays) with the values from those arrays. Figure 4 shows the state of

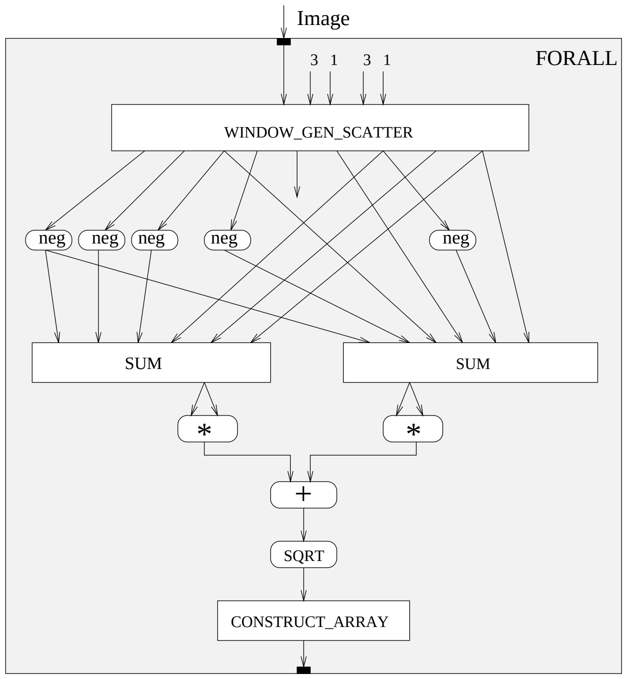

Figure 5. DFG after full optimizations.

the DDCF graph at this point. Continuing, algebraic identities allow all but two multiply nodes to be removed, since all multiplications are by either 1, 0 or −1. (Multiplies by −1 are converted to `NEG` nodes.) Finally, the window generator is converted to a form (appropriate only to windows with statically known sizes) in which each scalar value in the window is brought out as an individual output. This eliminates the remaining nine array reference nodes, and yields the graph shown in figure 5.

The optimized DDCF graph has now become a dataflow graph (DFG): it is flat, and it has operators that are suitable for generating VHDL code representing the FPGA configurations. 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. Mapping to Reconfigurable Hardware

After compiler optimizations, the program is partitioned into host and RCS code. The RCS code is translated to a dataflow graph, and the host code is compiled to C, and combined with run time routines that communicate with the RCS. The dataflow graph is translated via VHDL to FPGA configuration codes.

DFGs are a natural model for reconfigurable computing. The ordering of operations is determined by true data dependencies only, not constrained by artificial dependencies introduced by the stored data model. Dataflow expresses parallelism at all levels. Fine grain parallelism can be exploited within one processor, and loop-level parallelism can be exploited across processors. Combinational logic functions are functional by nature, and are thus easily synthesizable from dataflow graphs.

Because FPGA-based reconfigurable computing systems provide a high degree of flexibility in the design and programming of reconfigurable circuits, it is imperative to provide some form of structure, an *abstract architecture*, that the compiler can target. It is intended that this abstract architecture be independent of any particular type of reconfigurable hardware, yet include characteristics that can be used by any RCS architecture. It defines an overall execution model in the form of a set of computation and data movement primitives. The main components of the abstract architecture are *data transfer models, on-chip storage models,* and *operations.*

5.1. Prewitt on the AMS Wildforce Board

In order to evaluate compilation and especially code generation strategies, the dataflow graph of the Prewitt operator (see figure 5) was hand translated to VHDL and mapped onto the AMS Wildforce Board [15] in several ways. The Prewitt algorithm is important in its own right as a fundamental IP operation; it was selected for this exercise because of characteristics that make it an interesting example for reconfigurable computing:

- It is an inherently parallel operation, with each of the convolutions entirely independent of the others. The compiler can take advantage of this by creating a number of high level, coarse grain loop chunks, and can exploit data reuse by buffering and shifting data within these chunks.
- It involves the use of constant masks that allow considerable optimizations such as loop unrolling, constant folding, and algebraic simplification, before being implemented in hardware.
- It requires a squaring (multiplication) and a square root operation, which are expensive to implement in hardware.
- It involves the use of a streaming data model from host to hardware and back to host, which is a common mode of operation with reconfigurable computing, and is very natural for Image Processing.
- Finally, it is representative of a large number (perhaps even the majority) of common IP applications, involving the passing of a constant mask over a large image array.

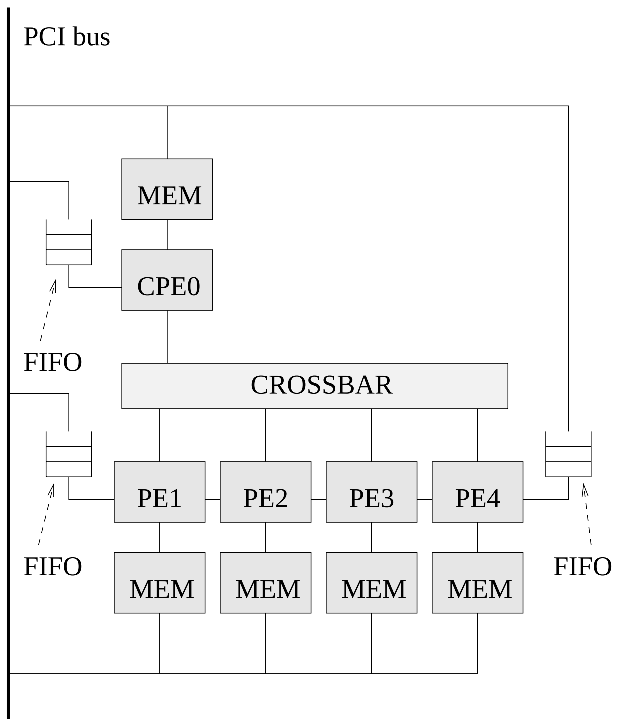

Figure 6. The AMS Wildforce board

The Wildforce-XL board is an FPGA-based reconfigurable computing engine; its architecture is shown in figure 6. It contains five user-programmable Xilinx FPGA (4036XL) chips. The first is designated CPE0 (Control Processing Element 0), while the others are designated PE1–4 (Processing Elements 1 through 4). For the Prewitt algorithm, the board is configured such that CPE0 accepts eight bit image pixels from the host and distributes these to the other PEs via the crossbar, a user-programmable 36-bit wide interconnect. PEs 1–4 are nearly identical in function – each receives data from the crossbar, performs the Prewitt calculation, and places the result where the host can retrieve it. The same configuration code is used in each. Coarse grain loop chunk parallelism occurs across the four PEs, and loop stripmining places several instantiations of the inner loop body on each PE.

The optimized inner loop body in figure 5 was implemented in VHDL. As discussed in section 4, the magnitude computation in the algorithm is implemented using a lookup table, which is stored in each PE's local memory prior to execution. This table is constrained to a reasonable size of 32×32, or 1K values, by discarding lower bits.

Except for the table lookup, the Prewitt inner loop body is a *combinational* circuit (i.e., it is asynchronous and contains no registers), and hence its output response for a given set of input values is delayed only by the propagation time of the signals through the circuit gates. The propagation delay of this loop body circuit establishes the maximum clock frequency for the entire system. By adding latches, the calculation can be pipelined to reduce the total propagation delay, allowing the clock frequency to be increased. Adding such a latch allows our Prewitt implementation to operate approximately 70% faster than the non-pipelined version.

Parallelism is achieved both across PEs and by replicating the inner loop body on each PE. By sending a vertical slice of four values (one pixel from each of four rows packed into a 32-bit word), a "stripe" of data is sent over time such that each PE can compute two Prewitt operations at one time. Thus, a total of eight operations are in progress at any one moment. Increasing the stripe width to eight rows (two 32-bit words, sent in two consecutive cycles on the crossbar) allows a total of six-Prewitt operations to be computed on each PE at a time, for a total of 24 operations. This results in a 50% throughput improvement over the two-body scheme, although the extra complexity of the interconnections reduces the maximum clock rate slightly.

Taking advantage of this parallelism requires the design of efficient transfer of data between host and RCS. Two main techniques for transferring data between the host and the Wildforce board were explored:

- **Streaming Data Input.** The host sends data to CPE0's hardware FIFO, in the order required by the calculations implemented on the PEs. This method is simple, and the streaming nature of the calculations allows the hardware to be in continuous calculation mode. However, since the host must send data in the order required within the PEs, some image pixels must be re-sent.

- **Batch Data Input.** The host sends the image array in row major order to CPE0's memory using DMA transfers. CPE0 then accesses the array in the order required by the other PEs and sends the values via the crossbar. Data needs to be sent only once, since this design moves the data ordering mechanism (the loop generators) from host to CPE0. Very fast transfer rates between host and CPE0 can be achieved, because of large block memory transfers. However, since the hardware cannot access its memory when the host is transferring data, computations and data transfer cannot overlap. The transfer time adds to the total processing time, unlike the FIFO method where pipeline parallelism can hide the transfer time.

The PEs return their results in one of two main ways. The first is a *systolic scheme*, whereby each PE sends data to its right neighbor, and PE4 (the rightmost PE) places the data into the output FIFO, where it can be retrieved by the host. This method is illustrated in figure 7a, which also shows the instantiation of two Prewitt Inner Loop Bodies (ILBs). In the other scheme each PE places its results in its local memory, where the host can retrieve it. This configuration is shown with six Prewitt ILBs in figure 7b.

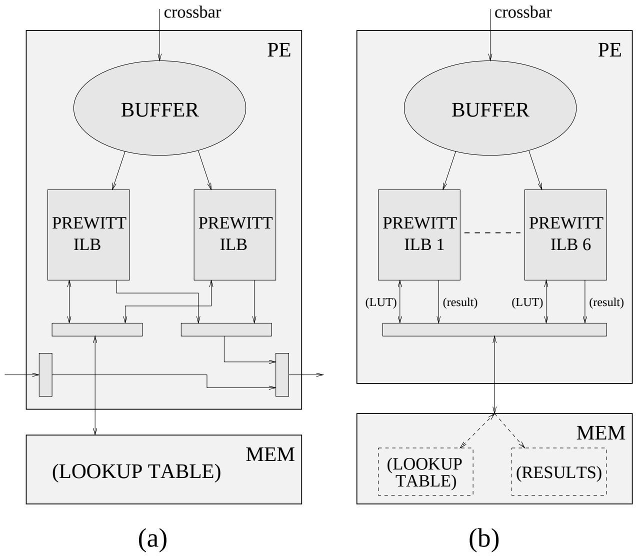

Figure 7. Two algorithm designs for Prewitt: (a) Two inner loop bodies calculated in parallel, with systolic output (b) Six inner loop bodies, output to memory.

Design	2 Prewitt CLBs	2 Prewitt Max Freq (MHz)	6 Prewitt CLBs	6 Prewitt Max Freq (MHz)
Stream	430 (33%)	11.3	1087 (84%)	10.3
Batch	420 (32%)	10.7	1051 (81%)	10.7
Pipelined	435 (33%)	17.0	1067 (83%)	16.8

Table 1. PE1-4 space requirements and maximum clock frequency for several Prewitt designs

Design	Data Input (ms)	Prewitt Calc (ms)	Data Output (ms)	Total Time (ms)	Overall Rate (MP/Sec)
Stream (10 MHz)		176.0		176.0	1.48
Batch (10 MHz)	2.99	23.01	8.24	34.24	7.65
Pipelined (17 MHz)	3.13	13.1	8.60	24.83	10.5

Table 2. Measured performance of the Prewitt designs.

The performance of the different Prewitt implementations on the Wildforce board can be measured in both space (amount of FPGA circuitry used) and speed. A summary of the size, expressed as the number of CLBs and the percentage of the total space used in the FPGA, and maximum clock speed, as determined by the Xilinx Place and Route tools, for the configuration code for PE1-4 for several of the Prewitt designs are shown in Table 1. In all cases, the size and speed of PE1-4, rather than CPE0, determines the operational limits of performance for the designs. Table 2 lists the measured performance for several of the two-Prewitt designs. Times are reported for a 512×512 image size; the processing rates (in MPixels/sec) for other image sizes are similar.

At 10MHz, the streaming (FIFO) version is capable of performing the Prewitt calculation at a rate of 20 MPixels/sec for the two-Prewitts-per-PE design, and 30 MPixels/sec for the six Prewitt design. This rate requires a transfer rate of 80 MB/sec between host and hardware, a value that should fall within the capabilities of the PCI bus (which has a maximum transfer rate of 132 MB/sec). However, so far our implementation has only achieved a small fraction of this rate. Work is continuing on implementing a DMA method for transferring data to the FIFO.

For the batch input case, the measured transfer rates between host and RCS memory are much more in line with expected values, falling in the range of 60-80 MB/sec. However, with batch mode this transfer time must be added to the computation time, since the two operations cannot overlap. This results in an overall computation rate of around 7.5 MPixels/sec for the non-pipelined version, and 10.5 MPixels/sec for the pipelined case.

To put these results in perspective, a Prewitt calculator using the same algorithm as implemented on the RCS was coded in C, compiled using Microsoft Visual C++ with optimizations, and executed on a 450 MHz Pentium PC. This program achieved 2.5 MPixels/sec. It is difficult to make performance comparisons when different hardware, compilers, and environments are involved. However, these initial results are encouraging, and appear to indicate that FPGA based co-processors can be effective.

6. Conclusions and Future Work

This paper has introduced the Cameron project, in which the Khoros GUI is being enhanced to allow image processing Applications to be written that are to be executed partly on a host machine and partly on reconfigurable computing systems (RCSs) consisting of FPGAs, memories and interconnections. Glyphs with parts to be run on the RCS are written in SA-C, a restricted C with true N-dimensional arrays, parallel loops, and sophisticated array access and array creation operators. An optimizing compilation process has been introduced, which targets dataflow graphs as an intermediate form, and maps these dataflow graphs via VHDL to the RCS. To show the compilation stages the Prewitt edge detector was used, and various code generation strategies were discussed. Initial performance comparisons with a hand coded C version are encouraging.

In future work, the complete compilation process from Khoros to dataflow graphs to hardware implementation will become more automated. Also, additional optimizations must be designed and implemented. Important optimizations yet to come are *loop fusion* and *bit width narrowing*. Loop fusion avoids intermediate data structures, reduces the amount of control logic, and increases the size of loop bodies to be executed on the RCS. Bit width narrowing reduces circuit size.

In the Khoros environment, the user can select which parts of a program are to be run on the RCS, as long as these parts are written in SA-C. The performance of the system, in terms of host, RCS data traffic, total system execution time and FPGA program space will be monitored. Performance feedback is to be integrated in the Khoros GUI.

Apart from using commercial FPGA hardware, two other experimental hardware designs are to be targeted: Morphosys [2] and PipeRench [4]. Morphosys is a parallel array of coarse grain reconfigurable cells on a chip combined with an on-chip core processor, DMA controller, and memories for (re)configuration codes and program data. Generating code for Morphosys will be akin to generating SPMD code. PipeRench is a pipeline organized system, which allows efficient mapping of a virtual (large) pipeline onto a physical (smaller) one. Pipeline stages are programmed in a very simple C, called DIL, representing a loop body as a dataflow graph. Generating code for PipeRench will entail mapping of the SA-C compiler's dataflow graph to DIL.

References

[1] A.V. Aho, R. Sethi and J.D. Ullman. Compilers, Principles, Techniques, and Tools. Addison Wesley Publishing Company 1986.

[2] Nader Bagherzadeh, Fadi J Kurdahi, et. al. Morphosys: A Reconfigurable Architecture. *Proceedings of NATO RTO Symposium of System Concepts and Integration*, Monterey, CA, April 98.

[3] A. DeHon. Dynamically Programmable Gate Arrays: A Step Toward Increased Computational Density. *Proc of Fourth Canadian Workshop of Field-Programmable Devices*, Toronto, Canada, May 1996.

[4] Seth Copen Goldstein, Herman Schmit, et. al. PipeRench: a Coprocessor for Streaming Multimedia Acceleration. *ISCA* 1999.

[5] J. McGraw et.al., SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2, Lawrence Livermore National Laboratory, Memo M-146, Rev. 1, 1985.

[6] J. Mundy. The Image Understanding Environment Progam. *IEEE Expert,* 10(6):64-73, 1995.

[7] R. Petersen and B. Hutchings. An Assessment of the Suitability of FPGA-Based Systems for use in Digital Signal Processing. *5th Int. Workshop on Field-Programmable Logic and Applications,* Oxford, 1995.

[8] S. K. Skedzielewski, J. R. W. Glauert. IF1, an Intermediate Form for Applicative Languages. Refernce Manual, M-170, Lawrence Livermore National Laboratory, July 1985.

[9] Sven-Bodo Scholz. Single Assignment C - Functional Programming Using Imperative Style. *Functional Language Implementation Workshop. Paper 21.*

[10] S. Umbaugh. *Computer Vision and Image Processing: A Practical Approach using CVIPtools.* Prentice Hall, New Jersey, 1998.

[11] J. Rasure and S. Kubica. The KHOROS Application Development Environment. In H. I. Christenses and J. L. Crowley, editors, *Experimental Environments for Computer Vision and Image Processing*. World Scientific, New Jersey, 1994.

[12] http://www.vsipl.org/

[13] M. Wolfe. High Performance Compilers for Parallel Computing. Addison Wesley Publishing Company 1996.

[14] Xilinx. *The Programmable Logic Data Book.* Xilinx, Inc., San Jose, California, 1998.

[15] WILDFORCE^{TM} Reference Manual. Annapolis Micro Systems, Inc., 1999