One-step Compilation of Image Processing Applications to FPGAs

A.P.W. Bohm, B. Draper, W. Najjar, J. Hammes, R. Rinker, M. Chawathe, C. Ross

Computer Science Department
Colorado State University
Ft. Collins, CO, U.S.A.
Cameron Project Goals

- FPGA’s offer massive parallelism and high computational density platforms for image processing applications.

- Programming for FPGA’s in behavioral hardware description languages is difficult for application programmers used to working with high-level algorithmic languages.

- The goal of the Cameron project is to make FPGAs and other adaptive computer systems available to more applications programmers, by raising the abstraction level from hardware circuits to software algorithms.

- Developed **Single Assignment C (SA-C)** - a variant of C programming language and an optimizing compiler that maps high-level programs directly onto FPGAs.
Cameron Project Architecture
(little boxes and lines - if only it was that easy ^_^)

- **SA-C**: a high-level algorithmic language for expressing image processing applications and compiling them to FPGAs.

- **Intermediate Compilation**: converts the SA-C program to a hierarchical graph representation called DDCF (Data Dependence and Control Flow) graph used for optimizations.

- **Program Division**: Compile program to a combination of low-level Data Flow Graphs (DFGs) and code for host architecture.

- **Final Compilation**: three modes:
  - Compile entire program to host code for debugging.
  - Run Data Flow Graphs under simulation.
  - Translate Data Flow Graphs into VHDL for synthesis and place-and-routing to FPGA.
The SA-C Language

( just like C, but more sassy ^_^ )

- SA-C (pronounced “sassy”) is a single-assignment side effect free language; each variable's declaration occurs together with the expression defining its value and variables are associated with wires.
- The goal of SA-C is to have a language that can express image processing applications elegantly, as well as allow seamless compilation to reconfigurable hardware.
- Data types in SA-C include signed and unsigned integers and fixed-point numbers, with user-specified bit widths including 9, 12, 14, 16, 32, etc.
- When a new variable is declared with the same name as a previous variable, the previous variable goes out of scope.

```c
/*
deares a new variable x that replaces
the old variable x (the new and old
variables x may or may not have the same
data type)
*/
int12 x = x + 1;
```

- SA-C does not allow dynamic data structures or recursive function calls.
- SA-C has multidimensional rectangular arrays whose extents can be determined either dynamically or statically.

```c
/*
matrix M of 14-bit signed integers.Left
dimension is dynamically determined,
right dimension is 6 fields.
*/
int14 M[:,6];
```
SA-C for-loops

- Biggest change from C is the interaction between arrays and for-loops in SA-C. SA-C does not contain address-of (&) and dereferencing (*) operators.
- To allow array access, a new system of array and for-loop interaction is used.

- for-loops in SA-C have 3 parts:
  - A loop body.
  - One or more return values.
  - One or more generators - produce all possible sub-arrays of the specified size.
    - scalar: produces a linear sequence of scalar values.
    - array-element: extracts scalar values from a source array, one per iteration.
    - array-slice: extracts lower dimensional sub-arrays (e.g. vectors out of a matrix).
    - array-window: allow rectangular sub-arrays to be extracted from a source array.

- Loop bodies may contain “nexified” variables (using the keyword next) which carry dependencies from one loop iteration to the next.
- Loops can return “reductions” calculated in loop bodies such as sum, product, min, max, mean, median, histogram.

- This syntax allows easy coding of array processing algorithms.
- Makes some optimizations possible:
  - compiler can reliably infer patterns of array access.
  - Loops without next variables are fully parallel
Example: The Prewitt Edge Detection Algo.
(I can see my house from here! ^_^)

The Prewitt edge detection masks are one of the oldest and best understood methods of detecting edges in images. To find edges, a user convolves an image with 2 image masks, one for horizontal and one for vertical, producing two derivative images (dx & dy). The strength of the edge at any given image location is then the square root of the sum of the squares of these two derivatives.

/* Prewitt Edge Detector Code */
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);
The Data Dependency and Control Flow Graph

- The DDCF graph is the compiler’s internal representation.
- Black rectangles along the top and bottom of compound nodes represent input and output ports.

```c
/* Prewitt Edge Detector Code */
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);
} return (M);
```
Optimizations and Pragmas
(We can rebuild him. We have the technology. ^_^)

- The compiler performs a number of general optimizations as DDCF-to-DDCF transformations.
  - These include Invariant Code Motion, Function Inlining, Switch Constant Elimination, Constant Propagation and Folding, Algebraic Identities, Dead Code Elimination, and Common Sub-Expression Elimination.
- User can include function prototypes for external VHDL plug-ins.
- Compiler can also make single-assignment specific optimizations through pragmas.
  - **Size Inference**: propagate constant array and loop sizes through the dependence graph in an upward, downward, or lateral flow.
  - **Full Loop Unrolling**: unroll all inner loops with fixed size masks.
  - **Array Value Propagation (Array Elimination)**: Replace array references with constant indices with the values of the array elements. When all references to an array have constant indices, entire arrays can be eliminated.
  - **Loop Carried Array Elimination**: If an array used in a loop is of a fixed size, its data can be stored in registers rather than allocated as an array. Compiler determines this through an inductive process. 2 requirements:
    - (1) The compiler must know or be able to infer the size of the array outside the loop.
    - (2) The compiler must be able to infer that the next array size is the same.
Optimizations and Pragmas (cont.)

- **N-Dimensional Stripmining**: Divide a large loop into several smaller loops that can be run in parallel through the use of an intermediate for-loop.
  - Allows inner nested loops to be unrolled to produce a larger more parallel circuit.
  - Result arrays are concatenated together with the tile operator.

- **Lookup Tables**: Convert computationally expensive functions into lookup tables by wrapping the expression in a loop and recursively compiling, executing, and storing the results.

- **Fusion**: Combine together producer and consumer loops.
  - Reduces amount of data traffic necessary to perform calculations.
  - Eliminates the need for intermediate data storage.

- **Temporal Common Sub-Expression Elimination (CSE)**: look for values computed in one loop iteration that were already calculated in previous loop iterations. Hold these values in registers and eliminate recombinations.

- **Window Narrowing / Window Compaction**: Eliminate unused portions of a window. Move calculations on window elements to earlier iterations. Used in conjunction with TCSE.

- **Pipeline Registers**: Fusion and unrolling cause long critical path, a low clock frequency. Adding stages of pipeline registers can break up the critical path and 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.
Conversion to Data Flow Graphs (DFGs)

- Data Flow Graphs are used for mapping parts of SA-C programs onto reconfigurable hardware.
- Nodes are operators, edges are data paths.
- DFGs can be used in token driven simulation for debugging.
- The Compiler searches for loops in the optimized DDCF that can be translated into logic.
  - The SA-C Compiler attempts to convert every inner most loop into a Data Flow Graph.
  - Some inner most loops may be the product of optimizations.
  - Currently, only loops with statically sized window generators can be translated into DFGs.

![DDCF graph for Prewitt Algorithm](image1.png)  ➞  ![DFG for Prewitt after optimizations](image2.png)
From Data Flow Graphs to VHDL

- Translator produces synchronous circuits from asynchronous DFGs.
- Nodes are of 2 types:
  - Simple combinational nodes, such as arithmetic and logical operators.
    - Form the Inner Loop Bodies (ILBs) of circuits.
    - Translated as a single VHDL statement or as a pre-built VHDL component.
  - Complex generator and collector nodes - contain registers, state machines, and a clock.
    - Translated by selecting a component from a library of pre-built modules.
    - Complex nodes make up a “wrapper” around the ILB circuit.
- Generator & Reduction Nodes
  - Receives data from external memory in blocks.
  - Stores data in buffer created from a shift register matching the size of the generating window.
  - On each clock cycle, the register shifts out the last column of data and in a new column to produce a “sliding window” effect.
  - Reduction nodes are the same process in reverse.
- Compiler Output
  - Script files for commercial VHDL and Place&Route tools.
  - Parameter file specifying the size and shape of various buffer, timing information to synchronize wrapper components.
Testing System

- Annapolis Microsystems StarFire PCI
  - Single Xilinx XCV1000-BG560-4 Virtex FPGA
  - Six Local 1MB 32-bit-addressable Memories
  - Clock speeds of 25MHz to 180 MHz

- Host System
  - 266-MHz Pentium Based PC
  - StarFire connects over 66MHz PCI bus.

- Comparison System
  - 450MHz Pentium II Based PC
  - The Xilinx XV-1000 is roughly cotemporaneous with the 450MHz Pentium II.
  - Windows NT operating system
Application: Intel Image Processing Library

- The Intel Image Processing Library (IPL) contains low-level image processing operations coded for highly efficient execution on MMX processors.
- Implemented 32 image processing operations to exactly match the API’s of Intel IPL counterparts.
- Ran side by side tests of operations on 8-bit 512x512 pixel images.

- The StarFire is 1.2 to 6 times slower than the Pentium system.
  - These are I/O bound tasks
  - Data paths on the FPGA are no wider than on the Pentium, however the FPGA runs at a much slower clock speed.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Pentium Exec.</th>
<th>RCS Exec.</th>
<th>RCS data download</th>
<th>RCS data upload</th>
<th>Frequency (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddS</td>
<td>0.081531</td>
<td>0.008355</td>
<td>0.02221</td>
<td>0.03292</td>
<td>39.5</td>
</tr>
<tr>
<td>And</td>
<td>0.003179</td>
<td>0.008492</td>
<td>0.04418</td>
<td>0.03298</td>
<td>38.9</td>
</tr>
<tr>
<td>AndS</td>
<td>0.001865</td>
<td>0.008331</td>
<td>0.02222</td>
<td>0.03275</td>
<td>39.6</td>
</tr>
<tr>
<td>Close</td>
<td>0.018069</td>
<td>0.012800</td>
<td>0.02337</td>
<td>0.03308</td>
<td>25.0</td>
</tr>
<tr>
<td>Convolve2D</td>
<td>0.006548</td>
<td>0.006624</td>
<td>0.02341</td>
<td>0.03385</td>
<td>25.1</td>
</tr>
<tr>
<td>Dilate</td>
<td>0.011578</td>
<td>0.028910</td>
<td>0.02376</td>
<td>0.03385</td>
<td>25.0</td>
</tr>
<tr>
<td>Erode</td>
<td>0.016764</td>
<td>0.028910</td>
<td>0.02375</td>
<td>0.03385</td>
<td>25.0</td>
</tr>
<tr>
<td>Gaussian3x3</td>
<td>0.005670</td>
<td>0.006637</td>
<td>0.03386</td>
<td>0.03390</td>
<td>25.1</td>
</tr>
<tr>
<td>Greater</td>
<td>0.000109</td>
<td>0.008461</td>
<td>0.04430</td>
<td>0.00413</td>
<td>39.0</td>
</tr>
<tr>
<td>GreaterS</td>
<td>0.011431</td>
<td>0.009563</td>
<td>0.02220</td>
<td>0.02238</td>
<td>34.5</td>
</tr>
<tr>
<td>LShiftS</td>
<td>0.001469</td>
<td>0.008537</td>
<td>0.02239</td>
<td>0.02256</td>
<td>38.7</td>
</tr>
<tr>
<td>Less</td>
<td>0.000074</td>
<td>0.008438</td>
<td>0.04434</td>
<td>0.00413</td>
<td>39.1</td>
</tr>
<tr>
<td>LessS</td>
<td>0.011567</td>
<td>0.008179</td>
<td>0.02176</td>
<td>0.00409</td>
<td>40.4</td>
</tr>
<tr>
<td>MaxFilter</td>
<td>0.005189</td>
<td>0.021259</td>
<td>0.02304</td>
<td>0.03341</td>
<td>28.2</td>
</tr>
<tr>
<td>MinFilter</td>
<td>0.005328</td>
<td>0.021755</td>
<td>0.02304</td>
<td>0.03342</td>
<td>27.5</td>
</tr>
<tr>
<td>Multiply</td>
<td>0.003541</td>
<td>0.009055</td>
<td>0.04322</td>
<td>0.03306</td>
<td>36.4</td>
</tr>
</tbody>
</table>

IPL Results - times are in seconds
Application: ARAGTAP Pre-Screener
( fun with acronyms ^_^ )

- ARAGTAP - large automatic target recognition (ATR) system for synthetic aperture radar (SAR) images developed by the U.S. Air Force. The ARAGTAP pre-screener is the front end of ARAGTAP. It is a focus of attention (FOA) system designed to detect possible targets in SAR images, returning regions of interest (ROIs) to be verified and identified.

- Algorithm involves down sampling, 8 grayscale dilation (in square and plus shapes), 8 grayscale erosion (in square and plus shapes), positive differencing, majority thresholding, bitwise And, percentile thresholding labeling, and label pruning.

- Each of these operations can be implemented to run on the FPGA.

- Compiler can fuse the dilate and erode loops using producer-consumer fusing.

- Use Temporal CES, Window Compaction, and Window Narrowing to produce logic driven by a 9x1 window generator.
Application: ARAGTAP Results

- Execution times on the reconfigurable system are about **ten times faster**.
- Cost of downloading the circuit configuration onto the FPGA, and of transporting the data back and forth between the RCS system and the host processor.
  - Download times swamp all other costs.
  - Only feasible to accelerate one operator per FPGA
- Gain in speed of the reconfigurable processor justifies the cost of downloading and uploading the image data
  - Total improvement is reduced to a factor of four.
Application: Matrix Solving
( why should PC’s have all the fun? ^_^ )

- Developed an iterative tri-diagonal solver program in SA-C.
- A tri-diagonal matrix system is an equation of the form $Ax = b$, where $x$ and $b$ are vectors, and $A$ is a tri-diagonal matrix.
- $A$ is necessarily square, and has non-zero entries only along its diagonal and immediately adjacent to its diagonal. The goal is to find $x$, given $A$ and $b$.

- Performance was tested on an 8x8 diagonally dominant tri-diagonal matrix.
- An equivalent C program was run on an 800MHz Pentium III.

<table>
<thead>
<tr>
<th>seconds (800 MHz P3)</th>
<th>microseconds (AMS)</th>
<th>Frequency</th>
<th>Logic Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3x10^{-6}</td>
<td>0.5x10^{-6}</td>
<td>25.2 MHz</td>
<td>35%</td>
</tr>
</tbody>
</table>

Tri-diagonal Solver Results: AMS here is the StarFire FPGA system previously described.
(microseconds is possibly (probably?) a typo)
Conclusions

(^^^)

- Performance evaluation of the language is still in its infancy, but early results are promising.

- Additional compiler optimizations are in the works:
  - More drastic replacement of DFG sections with lookup tables.
  - Addition of stream data-structures to allow processes on the FPGA to work cooperatively.

- Currently effectiveness is limited by I/O.
  - For now, SA-C is limited to applications involving large calculation times compared to I/O
Discussion