Compiling and Optimizing Image Processing Algorithms for FPGAs

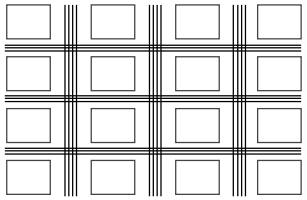
Bruce Draper, Walid Najjar*, Wim Böhm, Jeff Hammes, Bob Rinker, Charlie Ross, Monica Chawathe and José Bins

Colorado State University
**University of California Riverside*

Reconfigurable Computing Systems (RCS)

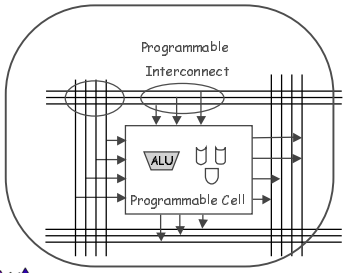
- Common Architecture
  - Array of programmable computing cells
  - Programmable interconnect among cells
  - Memory, local to the array, for data and code
- Fine-Grained (FPGA-based) Architectures
  - Direct mapping of a computation onto a circuit of (logical) gates
  - Cells: programmable look-up tables
- Coarse-Grained Architectures
  - Cells: variable size (power) processors, or just ALUs

Architecture of RCSs



General Architecture of a Reconfigurable Array

Cell Architecture Detail



Pros & Cons of RCSs

- Advantages:
  - Higher computation density then CPUs (MIPS/area)
  - More flexible than ASICs: reconfigurable
  - Large and variable level of parallelism
- Where does an RCS fit?
  - **Currently:** attached processor (I/O bus: PCI, PC-card, etc)
  - **Ideally:** co-processor (on memory bus) or as a functional unit within a CPU (share registers)
- Problems:
  - FPGAs are programmed using Hardware Description Languages (HDLs): Verilog or VHDL
  - Applications programmers do not know (or want to know) HDLs
  - RCS are not accessible where they are needed!

Cameron Project Overview

- Objective
  - Provide a path from algorithms to hardware implementation
  - Seamless compilation from an algorithmic high-level language to netlists
- Approach
  - A software-first approach
  - Extensive leverage of compiler optimization *before* mapping to hardware
- Current Focus
  - FPGA-based RCSs hosted by a PC
  - Image processing applications

Project Objectives

- Current Programmability of RCS

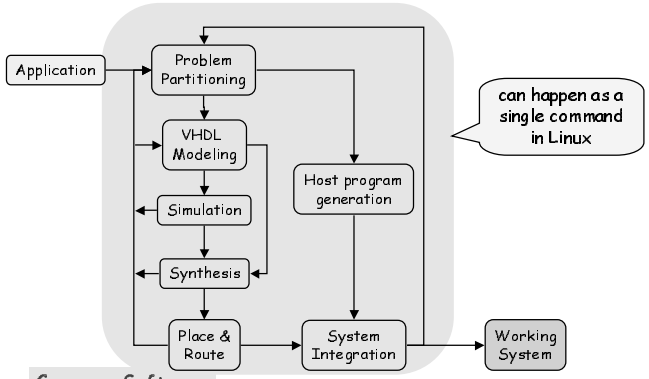
  - Image processing application programmers will never be good circuit designers
  - *The poor programmability of RCS is the most serious obstacle to a wide use of RCS by application programmers*

The primary objective of the Cameron Project is to raise the abstraction level of ACS programming from circuits to algorithms

Cameron Project Approach

- Language - Single Assignment C
  - A subset of C
  - Efficient support of IP and mapping to hardware
- Compilation to a dataflow graph representation
  - An ideal platform for optimization
  - Uncovers all levels of parallelism: bit, operation & loop
- Mapping to RCS Hardware
  - From the DFG representation to VHDL
  - Use of commercial tools for synthesis, place & route
- Real Applications
  - Level 1: 40 image processing libraries (most from Intel's IPL)
  - Level 2: several end-to-end medium to large applications

Programmability of RCS - Currently



SA-C: Single Assignment C

- Overview
  - C--: single assignment, no pointers & no recursion
  - Support for IP applications and hardware implementations
  - SA-C is implicitly parallel
    - Only true data dependencies, translate to wires on FPGA
    - No aliasing problems

```
//convolution inner loop
result[:,:] =
      for window win[rk,ck] in persrc
          {uint8 conv =
              for elem1 in win dot elem2 in kernel
              return(sum(elem1*elem2));
          }
      return(array(conv));
```


SA-C – Image Processing Support

- True n-Dimensional arrays
- Typical IP access functions
  - Slices (lower dimensional sub-arrays)
  - Windows (same dimensional sub-arrays)
  - Built-in perimeter handling
- Typical IP reduction functions
  - median, standard deviation &histogram
  - concatenation and tiling
  - logic/arithmetic: sum, max, min, mean

SA-C – Support for RCS

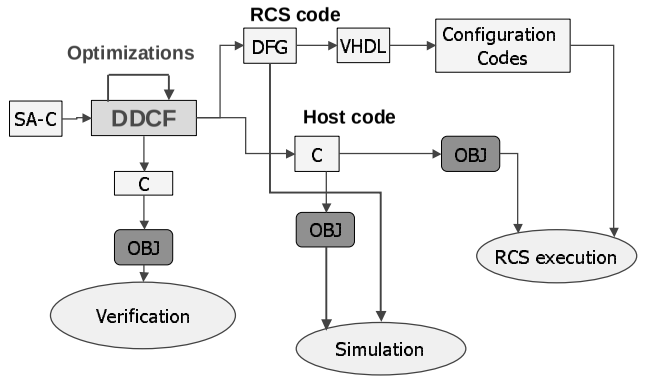
- Variable size / precision integers and fixed-points
  - uint4, fix16.4, complex int8, …
- Single assignment
  - simplifies generation of (fine-grain) parallel code
  - only true data dependencies, translate to wires on FPGA
- Strict arrays
  - arrays are always *completely* and *uniquely* defined
  - no intra-array data dependencies: improve loop optimizations
  - simplify analysis of inter array data dependence relations
  - array access and reduction functions expose parallelism
  - allow mapping to variety of parallel forms: expression, SIMD, pipeline, loop, MIMD

Example: Prewitt edge detection & threshold

```
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 R[:,:] = for window W[3,3] in Image {

                  int16 iph, int16 ipv = for h in H  dot  w in W  dot v in V
                                            return(sum(h*w), sum(v*w));
                  int16 SqrtSumSquare = sqrt(iph*iph + ipv*ipv);
               } return( array(SqrtSumSquare) );

uint8 T[:,:] = for r in R return(array(r > 127 ? 255 : 0));  %threshold
```

-structure level (not element level)
-parallelism easily detectable
-data reuse easily detectable

Software Architecture



Data Dependence & Control Flow Graphs

- Block structured data dependence graphs
  - *simple nodes* represent operators (e.g. addition)
  - *compound nodes* represent control structures (e.g.loop)
- Optimizations performed on this representation
  - currently concentrating on RCS optimizations
  - DDCF analysis determines which loops can go to RCS
    - user pragma can prevent loop from going to RCS
    - DDCF of RCS loop body mapped to data flow graph
    - multiple loop bodies can be mapped

DDCF Level Optimizations

- Conventional optimizations
  - constant folding, operator strength reduction, dead code elimination, invariant code motion, CSE
- Size inference of loops and arrays
  - uses close relationship of SA-C loops and arrays (generators, constructors)
  - information can propagate UP, DOWN and SIDEWAYS
- Full Loop Unrolling
  - follows from size inference
  - allows mapping onto circuit without cycles

Array Value Propagation

```
uint8 A[4] = {E0, E1, E2, E3}
        …  = A[2];
```

- A[2] can be replaced by E2
- E2 often constant but does not have to be
- Masks are such arrays
  - Defined by a set of expressions
- Constant indices occur in fully unrolled loops
- Often this then allows algebraic simplifications
- All this occurs in the Prewitt code

Producer Consumer Loop Fusion

```
A = for window W[3,3] in Image ...
B = for window W[5,5] in A ...
```

- Replace producer/consumer loops by one loop:
  - Eliminates the intermediate array A
  - This reduces data movement between host and reconfigurable board and/or from FPGA to local memory and back
  - Eliminates a reconfiguration step
- Can handle different window sizes & strides
- This occurs in the Prewitt plus threshold code

N-dimensional Strip Mining

- A pragma-directed optimization
- Creates a constant bound intermediate loop, which can be unrolled and mapped onto FPGA

```
      // pragma (stripmine(4,3))
         for window W[3,3] in Image
         return(array(Prewitt(W)))

 for window WT[4,3] in Image step(2,1){
  uint8 T[2,1] = for window W[3,3] in WT

return(array(Prewitt(W)));
 }return(tile(T))
```


Table Lookup Functions

- Another pragma-directed optimization
- For functions with narrow bit-width parameters
  - e.g. "magnitude" in Prewitt: sqrt(dfdx*dfdx+dfdy*dfdy)
- Implementation:
  - inline function calls
  - enclose function body in loop (running over all values)
  - generate DDCF and recursively compile and run
  - retrieve results and form array for lookup table
  - inline the function at its call location
  - lift out of the loop using invariant code motion

Example: Prewitt + Threshold

```
int8 V[3,3] = { {-1, -1, -1}, { 0,  0,  0}, { 1,  1,  1} };

int8 H[3,3] = { {-1,  0,  1}, {-1,  0,  1}, {-1,  0,  1} };

uint8 R[:,:] = for window W[3,3] in Image {
                 int8 iph, int8 ipv = for h in H  dot  w in W  dot v in V
                                        return(sum(h*w), sum(v*w));
                 uint8 mag = sqrt(iph*iph + ipv*ipv);
               } return( array(mag) );

uint8 T[:,:] = for pix in R{
                 uint8 t = pix>127 ?255 : 0;
               }return(array(t));
```

two loops running on the reconfigurable board, the first one is activated multiple times

Prewitt + Threshold, unrolled, propagated

```
uint8 R[:,:] = for window W[3,3] in Image {
                  int8 iph =  (W[0,2]+W[1,2]+W[2,2]) -
(W[0,0]+W[1,0]+W[2,0]);
                  int8 ipv =  (W[2,0]+W[2,1]+W[2,2]) -
(W[0,0]+W[0,1]+W[0,2]);
                  uint8 mag = sqrt(iph*iph + ipv*ipv);
                 } return( array(mag) );

uint8 T[:,:] = for pix in R{
                  uint8 t = pix>127 ?255 : 0;
                }return(array(t));
```

two loops running on the reconfigurable board, both activated once

Prewitt + Threshold, loops fused

```
uint8 T[:,:] = for window W[3,3] in Image {
                  int8 iph =  (W[0,2]+W[1,2]+W[2,2]) -
(W[0,0]+W[1,0]+W[2,0]);
                  int8 ipv =  (W[2,0]+W[2,1]+W[2,2]) -
(W[0,0]+W[0,1]+W[0,2]);
                  uint8 mag = sqrt(iph*iph + ipv*ipv);
                  uint8 t = mag>127 ? 255 : 0;
                } return( array(t) );
```

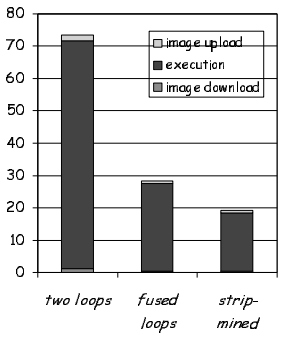
one loop running on the reconfigurable board, activated once

Prewitt + Threshold, strip-mined (4,3)

```
uint8 T[:,:] = for window W[4,3] in Image step(2,1) {
                  int8 iph1 =  (W[0,2]+W[1,2]+W[2,2]) - (W[0,0]+W[1,0]+W[2,0]);
                  int8 ipv1 =  (W[2,0]+W[2,1]+W[2,2]) - (W[0,0]+W[0,1]+W[0,2])
                  uint8 mag1 = sqrt(iph1*iph1 + ipv1*ipv1);
                  uint8 t1 = mag1>127 ? 255 : 0;

                  int8 iph2 =  (W[1,2]+W[2,2]+W[3,2]) - (W[1,0]+W[2,0]+W[3,0]);
                  int8 ipv2 =  (W[3,0]+W[3,1]+W[3,2]) - (W[1,0]+W[1,1]+W[1,2]);
                  uint8 mag2 = sqrt(iph2*iph2 + ipv2*ipv2);
                  uint8 t2 = mag2>127 ? 255 : 0;

                  uint8 t[2,1] = {{t1},{t2}};

                 } return( tile(t) );
```

one loop running on the reconfigurable board, half the iterations

Prewitt + Threshold Results



- Dominant cost is the download of FPGA configuration codes: 106 msec.
- Speed-ups without configuration code downloads
  - fusion: 2.58
  - strip mining: 3.8

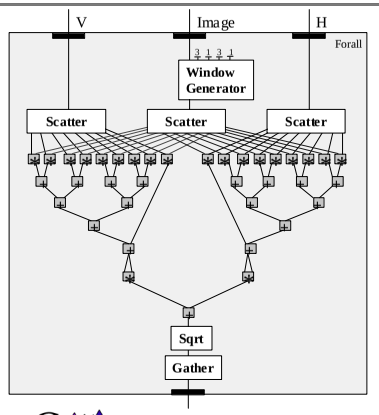
	Area (FMAPs)	Frequency (MHz)
Prewitt	1594 (62%)	2.48
Threshold	620 (24%)	10.35
Fused Loops	1347 (52%)	5.45
Strip-mined	2007 (78%)	4.53

The DFG Representation

- Dual Objectives:
  - Executable representation: simulation of DFGs for verification and debugging of programs
  - Compilation to VHDL
- DFG Format
  - Semantically rich: good for simulation
  - Some operators eliminated in hardware: e.g. fixed distance shifts
  - Complex nodes implemented using VHDL macros
    - Read & Distribute Window
    - Collect & Write Tile
    - Reduction (median, histogram, …)

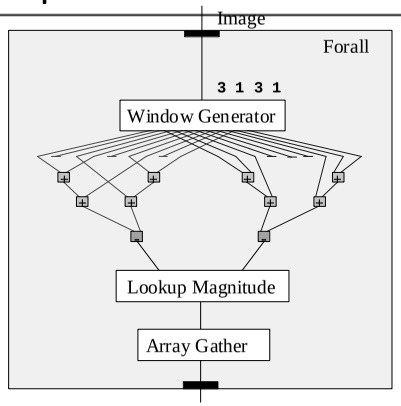
Prewitt DFG - Unoptimized

Prewitt:
inner loop unrolled



Prewitt DFG - Optimized

Prewitt:
constant-folded,
lookup table

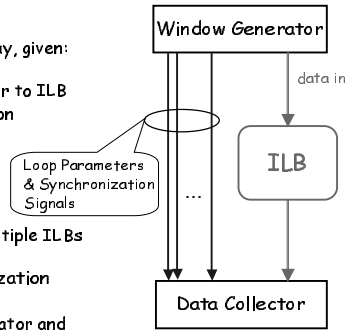


DFG to VHDL

- Abstract Architecture
  - separate input and output memories
  - array data (and other parameters) are written to input memory, host computes locations & strides
  - inputs to DFG:
    - addresses of input and output arrays
    - other parameters (array size, loop count, strides)
  - loop body driven by window or element generators
  - loop body is combinatorial (functional, stateless)
  - return values: elements or tiles written to output memory, host computes locations
  - result arrays fetched back by host

Overall Structure of a Loop

- Window Generator
  - ·extracts a window from an array, given: array size, window size & step
  - ·presents values in correct order to ILB
  - ·determines end-of-row condition
- Inner Loop Body
  - ·performs computation
  - ·mostly combinatorial
- Data Collector
  - ·collects results of single or multiple ILBs in proper order
- Loop Parameters & Synchronization Signals
  - ·coordinates operation of generator and collector



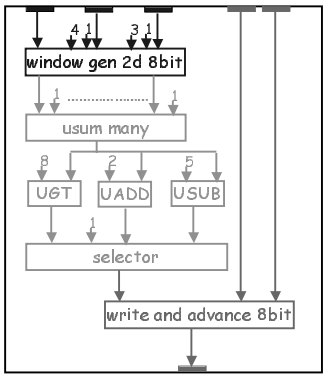
Example - SA-C Code

- Simple SA-C Code
  - Sum the pixels in a 4x3 window.
  - If the sum s > 8 return s+2 else return s-5.

```
uint8[:,:], uint8[:,:] main (uint8 A[:,:],
    uint8 x)
{
  uint8 r[:,:] =
     for window W[4,3] in A
         {
         uint8 s = array_sum (W);
         uint8 v = if (s>8)
                  return (s+2)
                else  return (s-5);
         } return (array (v));
  } return (r);
```


Example - DFG

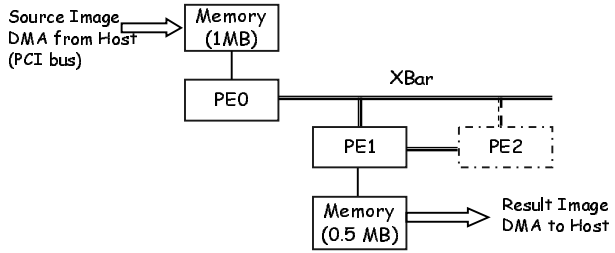
- Legend
  - Window Generator on CPE0
  - Inner Loop Body (ILB) on PE1 (PE2-4)
  - Write Memory (PE1-4)



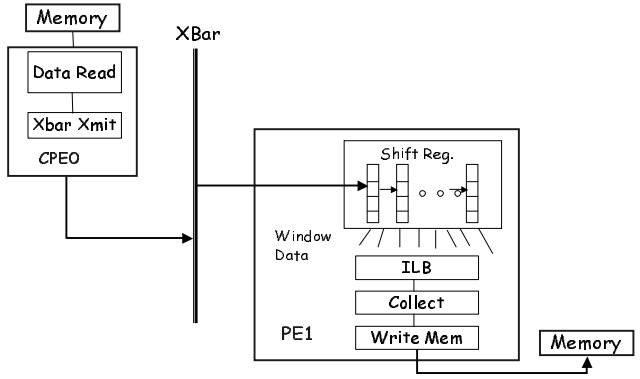
Hardware Platforms

- WildForce Board
  - Five Xilinx XV4036-3
  - Programmable crossbar
  - CPE0 on one side
  - PE1-PE4 on the other
  - All datapaths 36 bits (except memory)
- StarFire Board
  - One Xilinx XCV 1000
  - Four memory banks: two 64-bits, two 32-bits.

Simplified WildForce Model

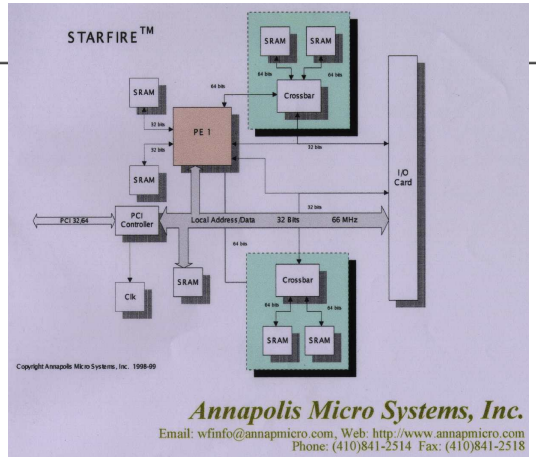


Window Generator Implementation



Performance Data on WildForce

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

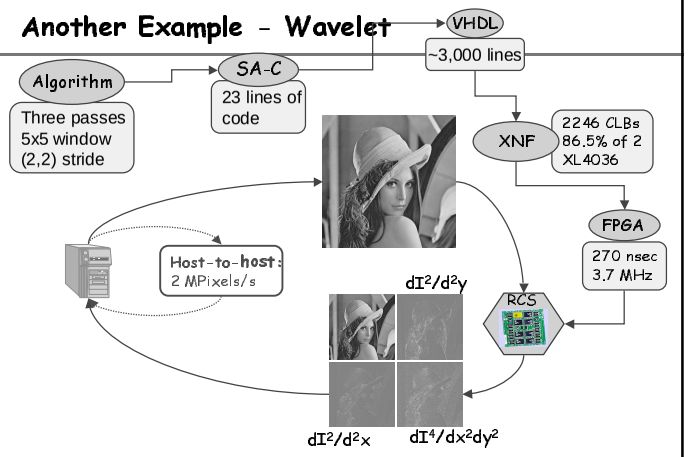


Frequency on WildForce & StarFire Boards

Benchmark	W-F	S-F
Add (Dyadic)	8.48	34.83
Add (Monadic)	9.04	37.18
Convolution (3x3)	9.55	26.24
Dilation (3x3)	8.91	25.64
Erosion (3x3)	8.98	26.45
Gaussian Filter (3x3)	8.68	32.39
LaPlace Filter (3x3)	8.62	31.61
LaPlace Filter (5x5)	8.69	25.43
Sum of Diffs (MP4)	10.11	35.52

Benchmark	W-F	S-F
Max Filter (4x5)	9.34	25.10
Max Value in Image	12.83	34.95
Min Filter (3x4)	8.65	26.89
Multiply (Dyadic; 16-bit out)	8.71	37.45
Multiply (Monadic; 16-bit out)	8.91	34.85
Subtract (Dyadic)	8.96	31.55
Subtract (Monadic)	9.49	36.53
Threshold (1-bit out)	8.58	27.74

Another Example - Wavelet



Conclusion & Future Work

- Implemented
  - Single command compilation from SA-C to hardware, includes the generation of host code, data transfer code, etc.
  - Extensive compilations under user control.
  - Over 50 low level IP operators and two large end to end applications.
- Current Work
  - Language and compiler optimization support for streaming video.
  - More end to end applications.
  - Hardware-based optimizations: pipelining, automated table look-up.
  - Compile-time area & time estimations.