Compiling and Optimizing Image Processing Algorithms for FPGAs

Bruce Draper, Walid Najjar*, Wim Buhm, Jeff Hommes, Bob Rinker, Charlie Ross, Monica Chawathe and Jose 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

Pros & Cons of RCSs

- Advantages:
  - Higher computation density than 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

- Application
- VHD L
- Simulation
- Synthesis
- Place & Route
- System Integration
- Working System

- Problem Partitioning
- Host program generation

Can happen as a single command in Linux
### 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

```c
// convolution inner loop
result[::] =
  for window win[r,k] in psrc
    {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

### Example: Prewitt edge detection & threshold

```c
int16 H[3,3] = { -1, -1, -1, 0, 0, 0, 1, 1, 1 };  // struct level (not element level)
          - parallelism easily detectable
          - data reuse easily detectable

int16 V[3,3] = { -1, 0, 1, -1, 0, 1, -1, 0, 1 };  // struct level (not element level)
          - parallelism easily detectable
          - data reuse easily detectable

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*iph + ipv*ipv));
} return(array(SqrtSumSquare));

uint8 T[::] = for r in R return(\(r > 127 ? 255 : 0\));  % threshold
```
**Software Architecture**

- Optimizations
  - SA-C
  - DDCF
  - C
  - OBJ
- RCS code
- VHDL
- Configuration Codes
- Host code
- C
- OBJ
- OBJ
- RCS execution
- Verification
- Simulation

**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
  - Use 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**

```c
... = 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

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

```c
def 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(dx*dx+dy*dy)
- Implementation:
  - inline function calls
  - enclose function body in loop (running over all values)
  - generate DCCF 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

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

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

```c
uint8 T[3,3] = 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 {
    uint8 iph = (W[0,0]+W[1,0]+W[2,0]) - (W[0,0]+W[1,0]+W[2,0]);
    uint8 ipv = (W[0,1]+W[1,1]+W[2,1]) - (W[0,1]+W[1,1]+W[2,1]);
    uint8 mag = sqrt(iph*iph + ipv*ipv);
} return( array(mag) );

uint8 T[:,::] = for pix in R{
    uint8 t1 = 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 {
    uint8 iph = (W[0,0]+W[1,0]+W[2,0]) - (W[0,0]+W[1,0]+W[2,0]);
    uint8 ipv = (W[2,0]+W[1,0]+W[2,0]) - (W[0,0]+W[1,0]+W[2,0]);
    uint8 mag = sqrt(iph*iph + ipv*ipv);
} return( array(t) );

uint8 T[:,::] = for pix in R{
    uint8 t1 = pix>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) {
    uint8 iph = (W[0,0]+W[1,0]+W[2,0]) - (W[0,0]+W[1,0]+W[2,0]);
    uint8 ipv = (W[2,0]+W[1,0]+W[2,0]) - (W[0,0]+W[1,0]+W[2,0]);
    uint8 mag = sqrt(iph*iph + ipv*ipv);
    uint8 t1 = mag>127 ? 255 : 0;
}

uint8 T[:,::] = for window W[2,1] in Image step(2,1) {
    uint8 iph = (W[0,0]+W[1,0]+W[2,0]) - (W[0,0]+W[1,0]+W[2,0]);
    uint8 ipv = (W[1,0]+W[1,1]+W[1,2]) - (W[0,0]+W[1,0]+W[2,0]);
    uint8 mag = sqrt(iph*iph + ipv*ipv);
    uint8 t2 = mag>127 ? 255 : 0;
}

uint8 T[:,::] = (t1, t2);

one loop running on the reconfigurable board, half the iterations

Prewitt + Threshold Results

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

- Dual Objectives:
  - Executable representation for 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
  - Puts pixel values in correct order to ILB
  - Determines end-of-row condition
- **Inner Loop Body**
  - Performs computation
  - Most computations
- **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 is > 5 return s + 2 else return s - 5.

```c
void main(uint8 A[4][3],
           uint8 x)
{
    uint8 s[4][3];
    for window W[4][3] in A
    {
        uint8 s = array_sum (W);
        uint8 r = if (s>5)
                   return (s+2)
                   else return (s-5);
    }
    return [array (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 XCV1000-4
  - Crossbar controller
  - CPE0 on one side
  - PE1-PE4 on the other
  - All data paths 36 bits (except memory)

- **StarFire Board**
  - One Xilinx XC601000
  - Four memory banks: two 64-bits, two 32-bits.
**Simplified WildForce Model**

Source Image DMA from Host (PCI bus) → Memory (1 MB) → PE0 → PE1 → PE2 → Result Image DMA to Host

**Window Generator Implementation**

Memory → XBar → Data Read → Multiply (Dyadic) → Memory

**Performance Data on WildForce**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Frequency (MHz)</th>
<th>Processing (MIPS)</th>
<th>WC dim</th>
<th>CLBs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add (Dyadic)</td>
<td>8.777</td>
<td>2.43</td>
<td>3.68</td>
<td>750</td>
</tr>
<tr>
<td>Add (Monadic)</td>
<td>9.040</td>
<td>2.01</td>
<td>2.76</td>
<td>641</td>
</tr>
<tr>
<td>Convolution (5x5)</td>
<td>8.754</td>
<td>2.30</td>
<td>5.50</td>
<td>1,233</td>
</tr>
<tr>
<td>Diffusion (5x5)</td>
<td>7.915</td>
<td>1.85</td>
<td>6.21</td>
<td>1,036</td>
</tr>
<tr>
<td>Beacon (3x3)</td>
<td>7.981</td>
<td>1.85</td>
<td>7.62</td>
<td>1,036</td>
</tr>
<tr>
<td>Gaussian Filter (7x7)</td>
<td>8.084</td>
<td>3.85</td>
<td>6.59</td>
<td>315</td>
</tr>
<tr>
<td>Laplace Filter (3x3)</td>
<td>8.025</td>
<td>2.77</td>
<td>7.86</td>
<td>173</td>
</tr>
<tr>
<td>Laplace Filter (5x5)</td>
<td>8.104</td>
<td>2.52</td>
<td>5.80</td>
<td>1,274</td>
</tr>
<tr>
<td>Window Size of Data (MB)</td>
<td>10.109</td>
<td>3.11</td>
<td>2.03</td>
<td>477</td>
</tr>
<tr>
<td>Max Error (4x4)</td>
<td>7.136</td>
<td>8.60</td>
<td>7.78</td>
<td>435</td>
</tr>
<tr>
<td>Max Error (16x16)</td>
<td>12.830</td>
<td>12.8</td>
<td>10.06</td>
<td>320</td>
</tr>
<tr>
<td>Multiply (Dyadic, 16x40, int)</td>
<td>8.715</td>
<td>4.55</td>
<td>4.13</td>
<td>800</td>
</tr>
<tr>
<td>Multiply (Monadic, RAM, int)</td>
<td>8.911</td>
<td>8.65</td>
<td>6.76</td>
<td>695</td>
</tr>
<tr>
<td>Polar Magnitude</td>
<td>2.518</td>
<td>2.56</td>
<td>2.55</td>
<td>1,181</td>
</tr>
<tr>
<td>Roberts Magnitude</td>
<td>3.096</td>
<td>3.06</td>
<td>2.04</td>
<td>1,062</td>
</tr>
<tr>
<td>Select Magnitude</td>
<td>2.162</td>
<td>2.30</td>
<td>2.29</td>
<td>1,164</td>
</tr>
<tr>
<td>Square Root (Real, 8-bit)</td>
<td>6.097</td>
<td>3.68</td>
<td>5.76</td>
<td>660</td>
</tr>
<tr>
<td>Subtrac (Dyadic)</td>
<td>3.090</td>
<td>4.42</td>
<td>5.62</td>
<td>354</td>
</tr>
<tr>
<td>Subtrac (Monadic)</td>
<td>9.922</td>
<td>9.43</td>
<td>6.26</td>
<td>641</td>
</tr>
<tr>
<td>Threshold (Int, int)</td>
<td>8.275</td>
<td>8.55</td>
<td>3.84</td>
<td>602</td>
</tr>
<tr>
<td>Wavelet (3x5)</td>
<td>8.053</td>
<td>2.48</td>
<td>1.50</td>
<td>1,455</td>
</tr>
</tbody>
</table>
### Frequency on WildForce & StarFire Boards

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>W-F</th>
<th>S-F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add (Dyadic)</td>
<td>8.48</td>
<td>34.83</td>
</tr>
<tr>
<td>Add (Monadic)</td>
<td>9.04</td>
<td>37.18</td>
</tr>
<tr>
<td>Convolution (3x3)</td>
<td>9.55</td>
<td>26.24</td>
</tr>
<tr>
<td>Dilation (3x3)</td>
<td>8.91</td>
<td>25.64</td>
</tr>
<tr>
<td>Erosion (3x3)</td>
<td>8.98</td>
<td>26.45</td>
</tr>
<tr>
<td>Gaussian Filter (3x3)</td>
<td>8.68</td>
<td>32.39</td>
</tr>
<tr>
<td>Laplace Filter (3x3)</td>
<td>8.62</td>
<td>31.61</td>
</tr>
<tr>
<td>Laplace Filter (5x5)</td>
<td>8.69</td>
<td>25.43</td>
</tr>
<tr>
<td>Sum of Diffs (MPH)</td>
<td>10.11</td>
<td>35.52</td>
</tr>
<tr>
<td>UFG-DAC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Another Example - Wavelet

- Algorithm: SA-C
- XNF: 2,246 (1.96) 86.5% of 2 XL4036
- Host to host: 2 MPix/s
- UN-DAC: 37
- 270 nsec 3.7 MHz
- UN-DAC: 38

### Conclusion & Future Work

- **Implemented**
  - Single command compilation from SA-C to hardware, includes the generation of host code, data transfer code, etc.
  - Extensive compilation 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 estimates

UN-DAC: 39