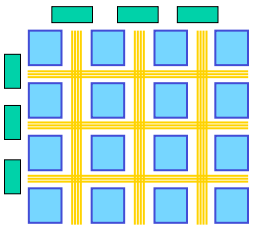
High Level, high speed FPGA programming

Wim Bohm, Bruce Draper, Ross Beveridge,
Charlie Ross, Monica Chawathe

Colorado State University

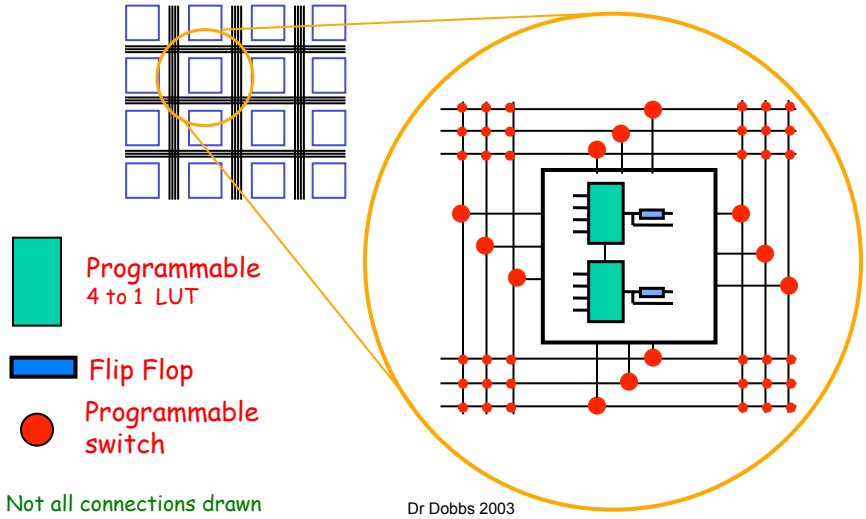
Opportunity: FPGAs

- **Reconfigurable Computing technology**
  - High speed at low power
  - Array of programmable computing cells:
    Configurable Logic Blocks (CLBs)
  - Programmable interconnect among cells
  - Perimeter: IO cells



- **Fine grain and coarse grain architectures**
  - Fine grain: FPGAs, cells are configurable logic blocks often combined with memory on the chip
    . Virtex 1000 (Xilinx Inc.)
  - Coarse grain: cells are variable size processing elements often combined with one or two microprocessors on the chip
    . Morphosys chip (U Irvine)
    . Virtex II Pro

FPGA details

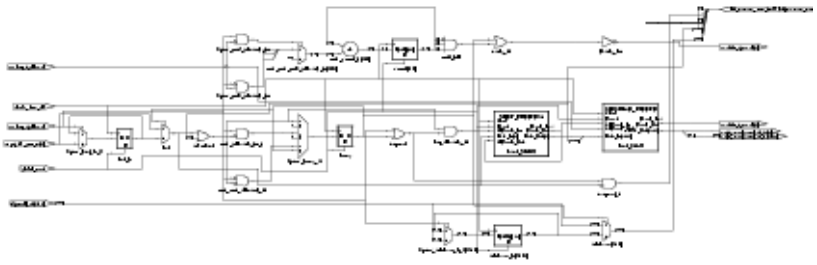


Not all connections drawn

Obstacle to reconfigurable hardware use

Circuit Level Programming Paradigm

- VHDL (timing, clock signals).
- Worse than writing time/space efficient assembly in the 1950s



Read One word from memory

Project Goals

- Objective
  - Provide a path from **algorithms** (not circuits) to **FPGA hardware**
  - Via an **algorithmic** language: SA-C an extended subset of C
    - data flow graphs as intermediate representation
    - language support for Image Processing
- Approach
  - One Step Compilation to **host** and FPGA **configuration** codes
  - Automatic generation of host-board **interface**
  - Compiler optimizations to improve traffic, circuit speed and area
  - If needed, optimizations are controlled by user **pragmas**

SA-C Image Processing Support

Data parallelism through tight coupling of loops and n-D arrays

- Loop header: structured parallel access of n-D array
  - Elements
  - Slices (lower dimensional sub-arrays)
  - Windows (same dimensional sub-arrays)
- Loop body: single assignment
  - Easily detectable fine grain parallelism
- Loop return: reduction or array construction
  - Logic/arithmetic reductions: sum, product, and, or, max, min
  - More complex reductions: median, standard deviation, histogram
  - Concatenation and tiling

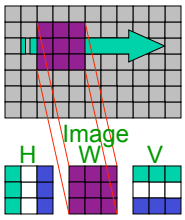
SA-C Hardware Support

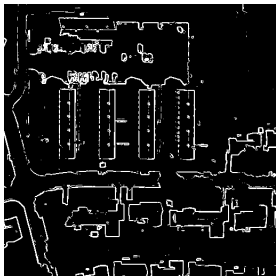
- Fine grain parallelism through Single Assignment
  - Function or Loop body is (equivalent to) a Data Flow Graph
  - Loop header fetches data from local memory and fires it into loop body
  - Loop return collects data from body and writes it to local memory
  - Automatically pipelined
- Variable bit precision
  - Integers: uint4, int5, int81
  - Fixed-points: fix16.4, fix80.30
  - Automatically narrowed
- Lookup tables (user pragma)
  - Function as a look up table
    - automatically unfolded
  - Array as a look up table

Example: Prewitt

```
int2 V[3,3] = {{-1, -1, -1},
               { 0,  0,  0},
               { 1,  1,  1}};

int2 H[3,3] = {{-1,  0,  1},
               {-1,  0,  1},
               {-1,  0,  1}};
for window W[3,3] in Image {
  int16 x, int16 y =
    for h in H  dot  w in W  dot v in  V
    return(sum(h*w), sum(v*w));
  int8 mag = sqrt(x*x + y*y);
} return( array(mag) );
```





Application performance summary

Summary of SA-C Applications

Application	WildStar (Virtex2000E)	Pentium III (800 MHZ)	Speed-up
Probing	0.08 sec	65 sec (VC++)	~800x
Prewitt Edge	.0019 sec	.1580 sec (Assm)	~80x
Canny Edge	.006 sec	.135 (Assm) .850 sec (VC++)	~20x ~120x
CDF Wavelet	.0020 sec	.0770 sec (VC++)	~35x
ARAGTAP	.0031 sec	.067 sec (VC++)	~20x
AddS (IPL)	.00067 sec	.00595 (Assm)	~8x

SA-Compilation

- Compilation: a sequence of program transformations
  - Parse and type-check: Source code to dependence graph form
  - Optimize using the dependence graph form
  - Generate time independent code:
    dependence graph to dataflow graph
  - Analyze timing behavior (handshaking, memory arbitration)
    dataflow graph to Abstract Hardware Architecture
  - Generate VHDL: Hardware Description Language
    linking the run time library

Compiler Optimizations

- Objectives
  - Eliminate unnecessary computations
  - Re-use previous computations
  - Reduce storage area on FPGA
  - Reduce number of reconfigurations
  - Exploit locality of data: reduce data traffic
  - Improve clock rate
- Standard optimizations
  - constant folding, operator strength reduction, dead code elimination, invariant code motion, common sub-expression elimination.

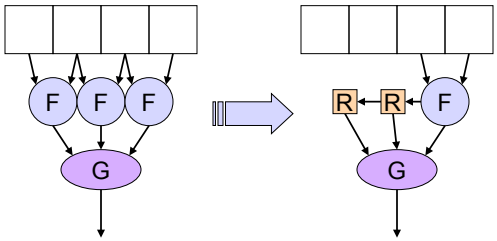
Initial optimizations

- Size inference
  Propagate constant size information of arrays and loops down up,and sideways (dot products).
- Full loop unrolling
  Replace loop with fully unrolled, replicated loop bodies. Loop and array indices become constants.
- Array Value and constant Propagation
  Array references with constant indices are replaced by the array elements, and by constants if the array is constant.
- Loop fusion
  Even of loops with different extents

Iterative (transitive closure) application of these optimizations replaces run-time execution with compile-time evaluation a lot like partial evaluation or symbolic execution

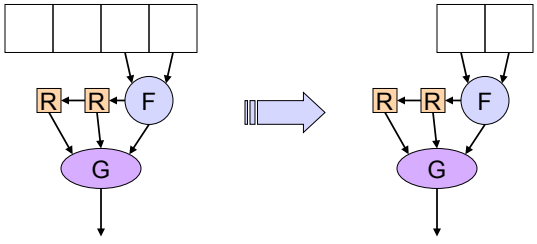
Temporal CSE

CSE eliminates redundancies by identifying spatially common sub-expressions. Temporal CSE identifies common sub-expressions between loop iterations and replaces the result by delay lines (registers). Reduces space.



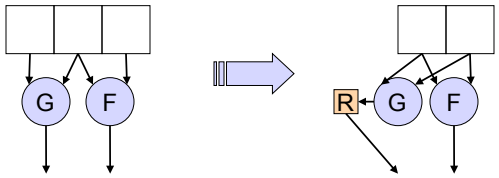
Window Narrowing

After Temporal CSE, left columns of the window may not be referenced. Narrowing the window further reduces space.



Window Compaction

Another way of setting the stage for window narrowing, by moving window references rightward and using register delay lines to move the inputs to the correct iteration.



Low level optimizations

- Array + Function Lookup Table conversion through Pragmas
  - Array Lookup conversion treats a SA-C array like a lookup table
  - Function Lookup conversion replaces an expression by a table lookup
- Bit-width narrowing
  - Exploits the user defined bit-widths of variables to minimize operator bit-widths. Used to save space.
- Pipelining
  - Estimates the propagation delay of nodes and breaks up the critical path in a user defined number of stages by inserting pipeline register bars. Used in all codes to increase frequency.

Application: Probing

A probe is a point pair in a window of an image
A probe set defines one silhouette of a vehicle
(automatically generated from a 3D model)
A vehicle is represented by 81 probe sets
(27 angles in an X,Y plane) x (3 angles in Z plane)
We have 12 bit LADAR images of three vehicles:

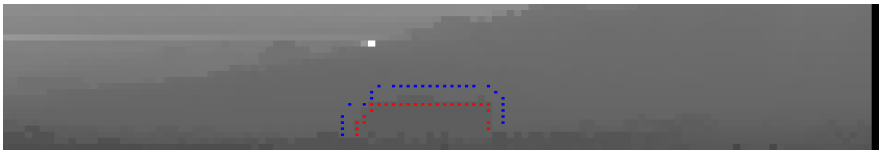
m60	Tank
m113	Armored Personnel Carrier
m901	Armored Personnel Carrier + Missile Launcher

A hit occurs when the pair straddles an edge:
one point is inside the object, the other is outside it
Probing finds the best matching probe set in each window.
The best match has largest ratio: count / probe-set-size.

Still life with m113

Color image

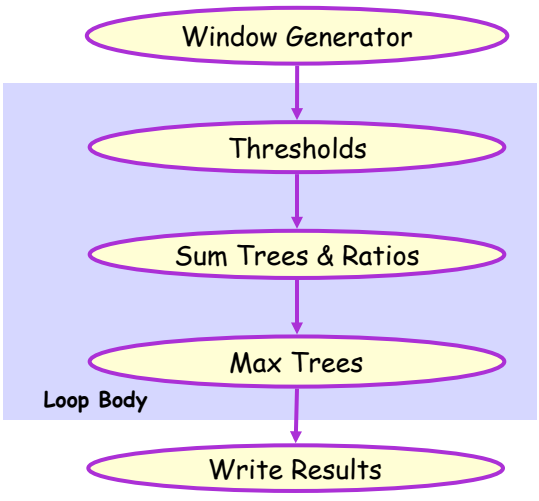
LADAR image



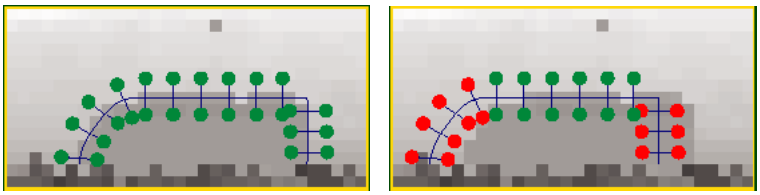
Probing code structure

```
for each window in Image
//return best score and its probe-set-index
   score, probe-set-index =
     for all probe-sets
       hit-count =
         for all probes in probe-set
         return(sum(hit))
       score = hit-count / probe-set-size
     return(max(score),probe-set-index)
return(array(score),array(probe-set-index)
```


Probing program flow



Probing: the challenge



Since every silhouette of every target needs its own probe set, probing leads to a massive number of simple operations.

In our test set, there are 243 probe sets, containing a total of 7573 probes. How do we optimize this for real-time operation on FPGAs?

Probing and Optimizations

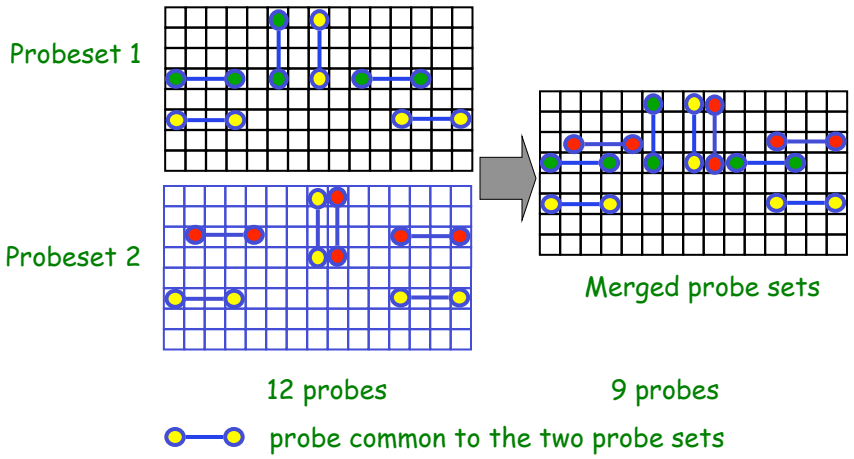
```
for each window in Image
    for all probe-sets PS
        for all probes P in PS
        compute score = (sum of hit(P)) / size(PS)
    identify  P with maximum score
```

The two inner **for** loops are fully unrolled, which turns them into a **giant** loop body (from 7573 inner loop bodies). This allows for:

- Constant folding / array value propagation
- Spatial Common Sub-expression Elimination
- Temporal Common Sub-expression Elimination
- Window Compaction

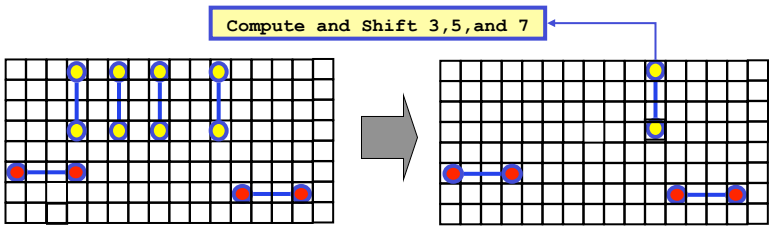
Spatial CSE in probing

Identify common probes across different probe sets and merge.



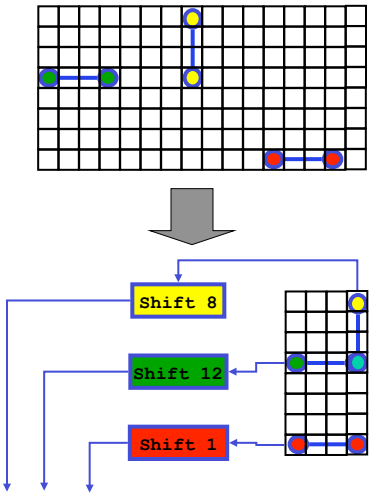
Temporal CSE in Probing

Identify probes that will be recomputed in next iterations, and replace them by delay lines of registers.



Window Compaction in Probing

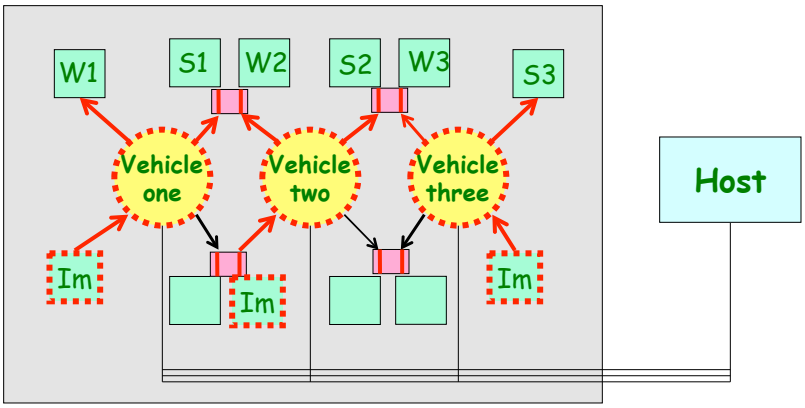
- Shifts all operations as far right as possible (earlier in time)
- Inserts 1-bit delay registers to bring result to proper temporal placement
- Sets the stage for window narrowing, removing 12 bit registers from circuit



Low level optimizations in probing

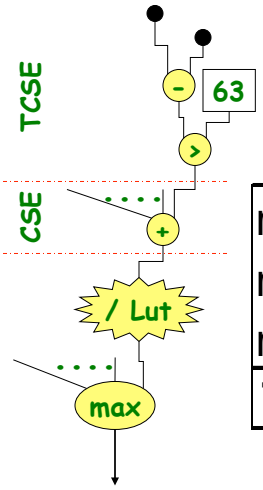
- Table lookup for ratios
  - For each Probe set size, there is a 1-D LUT
    count → rank in absolute ordering of ratios
  - Refinement: 0 for uninteresting ratios (< 60 %)
- Bit width narrowing
  - Initial hit: 1 bit
  - Each sum tree level uses minimal bit-width
- Pipelining
  - Based on automatically generated estimation tables: OP(bw1,bw2)
  - "Exhaustive" pipelining, until pipeline delay cannot be further reduced

Probe execution on WildStar



Producing 3 winner (W) and 3 score (S) images

Probing: DFG level statistics



	Unoptimized			Optimized		
	Probes	Adds	Win.	Probes	Adds	Win.
m60	2832	2751	12x34	151	1413	12x4
m113	2315	2234	11x26	106	967	11x4
m901	2426	2345	13x25	143	1196	13x4
Total	7573	7330	13x34	400	3576	13x4

Probing: 800 MHz P3 performance

Number of windows: (512-13+1)*(1024-34+1) = 495,500
Number of probes (three vehicles) = X 7,573
Number of inner loops = 3,752,421,500

- Linux, compiler gcc -O6
  - 22 instructions in inner loop = 82,553,273,000
  - 800 MHz (1 instruction / cycle) ~103 Sec
  - Actual run time ~119 Sec
- Windows, compiler MS VC++
  - 16 instructions in inner loop = 60,038,744,000
  - 800 MHz (1 instruction / cycle) ~75 Sec
  - Actual run time (super scalar) ~65 Sec

Probing: WildStar Performance

Clock speed: 41 MHz

(Almost) every clock performs a 32-bit memory read

Number of reads, 13x1 window:
(512-13+1)*(1024) windows * 13 pixels / 2 pixels per word
= 3,328,000 reads / 41 MHz
~= 80.8 Milliseconds

Real run time: 0.081 seconds
Real # cycles: 3329023 cycles

NOW WE'RE SUPERCOMPUTING !!

FPGAs 800x Faster

- ~25x fewer operations
  - Aggressive compiler optimization
- ~4000x more parallelism
  - The nature of FPGA based computation
- ~125x slower rate
  - Clock frequency ~19.5x
  - Memory bandwidth is bottleneck ~6.5x

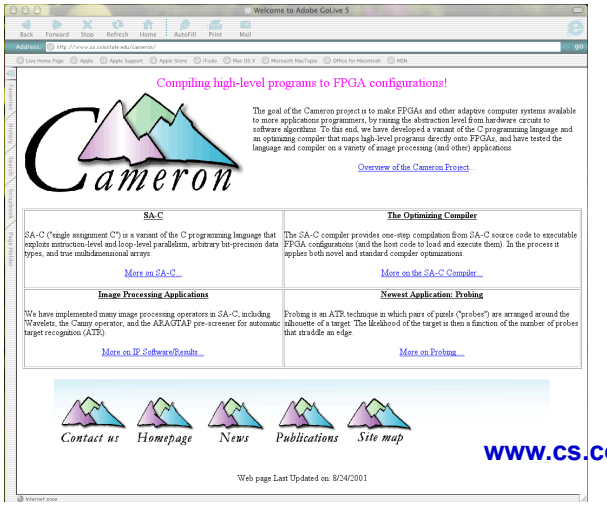
Concluding Remarks

- Trend: from Hand-written VHDL to High Level Language
  - Larger chips
    - Compactness is less critical
    - Exploiting internal parallelism is more critical
  - More complex chips
    - RISC kernels, multipliers, polymorphous components
    - More complex for human programmers
  - Productivity more important than hand tuned hardware
    - Time to market
    - Portability
    - Software quality
      - Debugging
      - Analysis

Future directions

- Embedded Net-based Applications
  - Neural Nets
    - Classifiers / Support Vector Machines
  - Security applications (monitor cameras, face recognition)
  - Network routers (payload aware)
- Language / compiler requirements
  - Stand-alone systems: no host
    - Stripped-down OS
  - Multiple processes connected by streams
  - Non-strict, random access, updateable data structures
  - New optimizations for pipelining cyclic computations

So long, and thanks for all the fish …



www.cs.colostate.edu/cameron