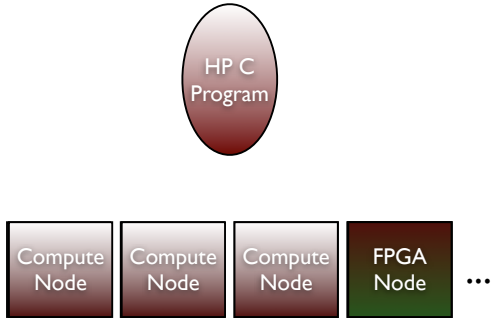
Modular Design of FPGA-Based Accelerators in C

Walid Najjar and Jason Villarreal
Computer Science & Engineering, University of California Riverside
& Jacquard Computing Inc.

UC RIVERSIDE CSE

FPGAs Potential - HW Accelerators for HPC

- **Accelerate computationally demanding applications**
  - Molecular Dynamics
  - Genetic String Matching
  - XML Query processing
- **Applications could take days to weeks to run on multiprocessor systems**
  - Specialized hardware could significantly reduce the time



Strengths of FPGAs

- **Massive amounts of parallelism available**
  - Much greater level of parallelism than any processor
- **Large pipelines can be created**
  - Streaming applications are very efficient
- **Reprogrammability**
  - Several applications may use the same FPGA
- **Available now**
  - Don't have to wait for an ASIC to be created

Problem: Programmability

- **FPGAs are programmed with low level hardware description languages**
  - Tedious and error-prone
  - Clock-level accuracy required
  - Not a common skill set for application programmers
- **Different strengths from software**
  - Cannot just put software on an FPGA and expect an improvement
    - SW good at large data structures and memory
    - HW good at large number of operations occurring simultaneously
  - Typically, an optimal HW algorithm is much different from an optimal SW algorithm
    - Temporal versus spatial domain

ROCCC

- **Riverside Optimizing Compiler for Configurable Computing**

- **Code acceleration**
  - By mapping of circuits to FPGA
  - Achieve same speed as hand-written VHDL codes

- **Hardware generated from C descriptions**
  - Improves productivity
  - Allows design and algorithm space exploration

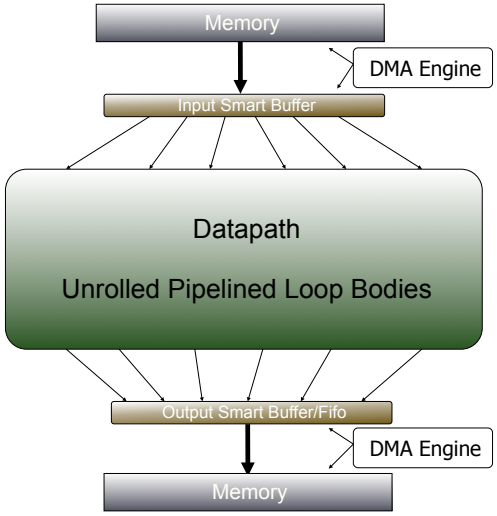
- **Keeps the user fully in control**
  - We automate only what is very well understood

ROCCC 1.0 Example

```
void begin_hw() {} ;
void end_hw() {} ;
int main()
{
  int numAtoms ;
  float p_i_x, inputArray[100], r2, t2, r2_delta, cutoff2_delta ;
  float outputArray[100] ;
  begin_hw();
  for (k = 0 ; k < numAtoms ; ++k)
  {
    t2 = p_i_x - inputArray[k] ;
    r2 = t2 * t2 + r2_delta ;
    if (r2 <= cutoff2_delta)
      outputArray[k] = 1 ;
    else
      outputArray[k] = 0 ;
  }
  end_hw() ;
  return 0 ;
}
```

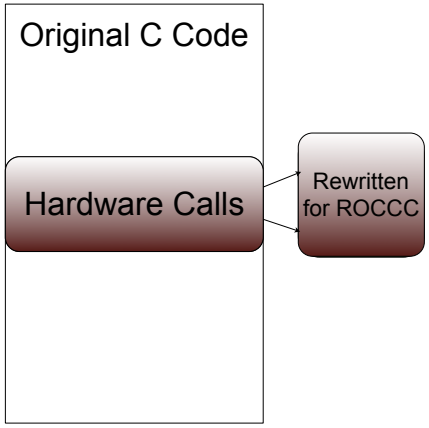
Execution Model

- **Decoupled architecture**
  - Memory accesses separate from datapath instructions
  - Memory accesses configured by the compiler
  - Parallel loop bodies
  - Smart input buffer handles data reuse



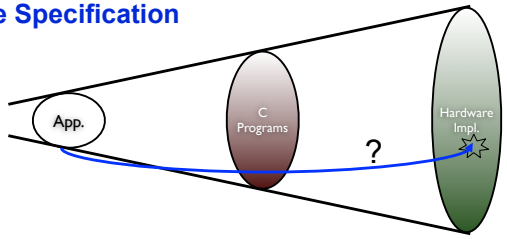
Hardware Accelerator Approach To Speedup

- **Profile entire application**
  - Find the most computationally intensive part of the code
- **Rewrite critical region as hardware**
- **Pass Through ROCCC**
- **Replace original code with a call to hardware**



Issues Revealed by ROCCC 1.0

- **Top down compilation approach**
  - Isolate the user from the details of the target platform
  - Works with CPUs: one underlying fundamental model, von Neumann
- **Complexity of platforms**
  - A plethora of platforms with varying capabilities
    - On board memories, I/O interfaces, firmware support etc.
    - Evolving FPGA architectures, a moving target
  - User is unaware of complexities of target platform
    - Complexities are reflected in the compiler
- **Hardware Specification**

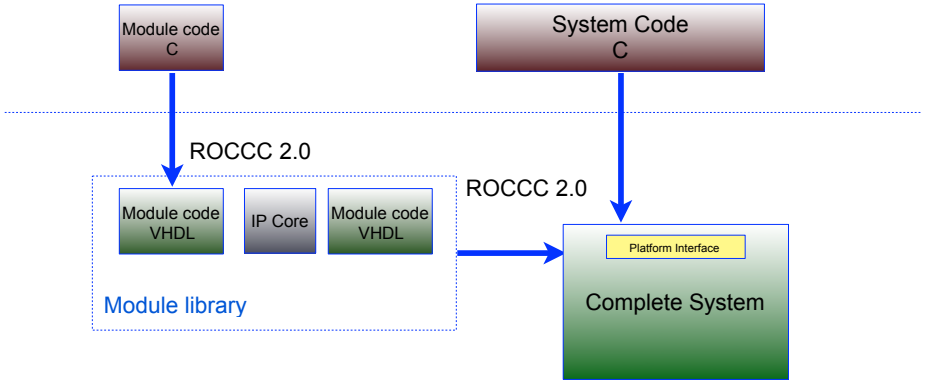


User must navigate hardware design space using compiler transformations: compiler technology is not suitable for this

Next Generation: ROCCC 2.0

- **Goals:**
  - Give more control of generated structure to designers
    - Build hardware systems in C from the bottom up
    - Description of components and interconnections using a C subset
  - Still maintain optimizations for hardware from ROCCC 1.0
    - User controlled optimizations
- **Two objectives:**
  - *Modularity* and *composability*
  - Keeping the positives of ROCCC 1.0
  - Enable hardware reuse
- **How**
  - Compile standalone C functions to HDL modules
  - Import pre-existing cores
    - IP or pre-compiled
  - Separate platform specific interfaces from algorithm codes
    - These can be other modules too
    - Multiple interfaces possible in each platform

ROCCC 2.0 Design Flow



Modules And Systems in ROCCC

- **No additional keywords or constructs added to C**
  - If you can read C, you can read and understand ROCCC code
  - Can compile and run module code in software to verify functionality
- **Module: hardware equivalent of a procedure**
  - Can exist as:
    - C code,
    - VHDL/Verilog code
    - Hardware macro (FPGA specific circuit)
  - Can be imported into other C modules or C code in any of these forms
- **System: hardware that processes streams of data**
  - Written as ROCCC 1.0 code with the addition of modules
  - Modules can be replicated and pipelined through compiler manipulations
  - Generated VHDL communicates through a platform independent ROCCC Abstraction Layer

ROCCC Abstraction Layer

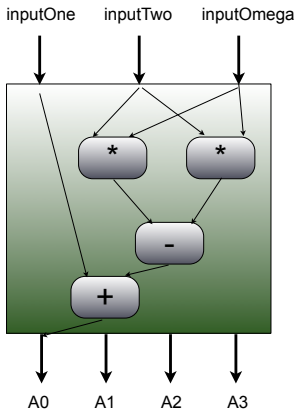
- **Platform independent hooks to connect to memories or streams**
  - Memories may be on-chip or off-chip
  - Streams may be any input device
    - Ethernet, serial, microphone in, etc
- **Each board/system can connect to these hooks through state machines**
  - User may control optimizations to specialize hardware for a certain board

Expressing a Module in C

- **Specify an interface**
  - Struct that specifies all inputs and outputs
  - All signals that can be seen outside the black box must be specified
- **Specify an implementation**
  - A function that takes and returns an instance of the struct
  - All outputs should be assigned in the function and all inputs read
  - Local variables translate into registers internal to the function

ROCCC 2.0 C Module Example

```
typedef struct
{
  int realOne_in ;
  int imagOne_in ;
  int realTwo_in ;
  int imagTwo_in ;
  int realOmega_in ;
  int imagOmega_in ;

  int A0_out ;
  int A1_out ;
  int A2_out ;
  int A3_out ;
} FFT_t ;
FFT_t FFT(FFT_t f)
{
  f.A0_out = f.realOne_in + (f.realOmega_in * f.realTwo_in) -
             (f.imagOmega_in * f.imagTwo_in) ;
  // ....
  return f ;
}
```



Modules Are Exported Back

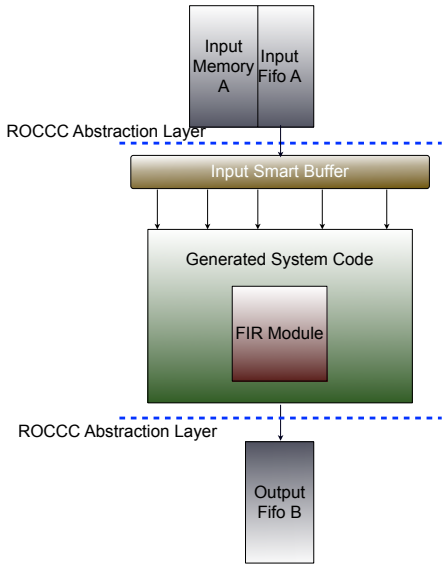
- **ROCCC maintains a database of previously compiled modules**
  - Exported back at the C level as function calls
  - Exported to hardware implementations as VHDL
- **Standard database can be interfaced and appended through SQL**
  - IP can be added through SQL queries if no C exists
    - Example: Floating point cores from Xilinx CoreGen
  - All the cores in the database can be integrated directly into the pipelines of larger systems compiled with ROCCC

ROCCC 2.0 - Using Modules as Building Blocks

```
#include “roccc-library.h”

OneStageButterfly_t OneStageButterfly(OneStageButterfly_t t)
{
  // Each FFT submodule is instantiated with a call to the exported function

  FFT(t.input0_in, t.input1_in, t.input16_in, t.input17_in, t.omega0_in, t.omega1_in,
        t.out0_out, t.out1_out, t.out2_out, t.out3_out) ;

  FFT(t.input2_in, t.input3_in, t.input14_in, t.input15_in, t.omega0_in, t.omega1_in,
        t.out4_out, t.out5_out, t.out6_out, t.out7_out) ;

  // The rest of the FFT modules...

  return t ;
}
```

Example System Code

```
void firSystem()
{
  int A[100] ;
  int B[100] ;
  int EndValue ;
  int i ;
  int myOutput ;

  for (i = 0  ; i < EndValue ; ++i)
  {
    FIR(A[i], A[i+1], A[i+2],
          A[i+3], A[i+4], myOutput) ;
    B[i] = myOutput ;
  }
}
```



Variance Filter - System Code

- **Problem:**
  - Locate moving objects over a number of frames
    - Such as satellites or asteroids
- **Procedure:**
  - For each pixel in N frames, compute the variance
  - Compare variance to threshold, and zero out anything under
  - Moving objects become detected
- **Written as ROCCC system code**
  - 270 lines of C code translated into ~17000 lines of VHDL
  - Synthesized targeting a Virtex 5 FX70T
    - 3491 Slices (7% of the FPGA)
    - 164.4 MHz clock
    - Generates 1 output per clock cycle

Steps in Compilation

- **Hi-CIRRF Transformations (SUIF)**
  - High level optimizations such as constant propagation and folding
  - Parallelizing optimizations including extensive loop unrolling
  - Identification of input and output scalars and streams
  - Exporting the module to the C level library
  - Outputting the Hi-CIRRF
- **Lo-CIRRF Transformations (LLVM)**
  - Identify floating point operations and replace with calls to hardware
  - Convert the CFG into a DFG
  - Pipeline the design (inserting copies where appropriate)
  - Generate the VHDL structure

Future ROCCC Improvements

- **Triple Modular Redundancy**
  - Available at any module level
- **Multi-FPGA communication built in through channel specification**
  - Different platforms must interface with our common interface
- **Design space exploration through different transformations**
  - Different levels of modularization
  - Systolic array generation
  - Pipelined unrolling
  - Tree based unrolling
    - Filter versus accumulation
  - All generated from the same C description and controlled through switches

Summary

- **ROCCC 2.0 opens up modular construction of hardware to C programmers**
  - No timing necessary
  - Specific hardware designs can be targeted to leverage the strengths of FPGAs
- **Hardware optimizations provide significant speedup over software**
  - Available at any level in the modular design
- **Hardware accelerators for High Performance Computing can be integrated at the C level**
  - Working software can produce working hardware

Distribution Available

- **ROCCC 2.0 - Version 0.3**
  - www.cs.ucr.edu/~roccc
  - System is open source
  - Example system and module code
    - Testbenches included
    - Tested on Linux machines
  - GUI
    - Uses QT libraries - platform independent
    - Easy integration of available modules