

# **Programmability, Performance and Productivity**

*The Emergence of FPGA Code Accelerators*

**SPL 2011 - CORDOBA, ARGENTINA**

Walid A Najjar

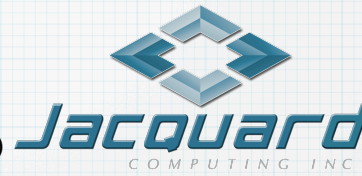
*University of California Riverside*

*Jacquard Computing Inc.*

# Credits

## Over the past six years

- ▶ Dr. Jason Villarreal (Jacquard Computing)
- ▶ Adrian Park (Jacquard Computing)
- ▶ Roby Atadero (Jacquard Computing)
- ▶ Robert Halstead (UCR)
- ▶ Dr. Zhi Guo (Huawei)
- ▶ Dr. A. Betul Buyukkurt (Google)
- ▶ John Cortes (Huawei)
- ▶ Dr. Dinesh Suresh (AMD)



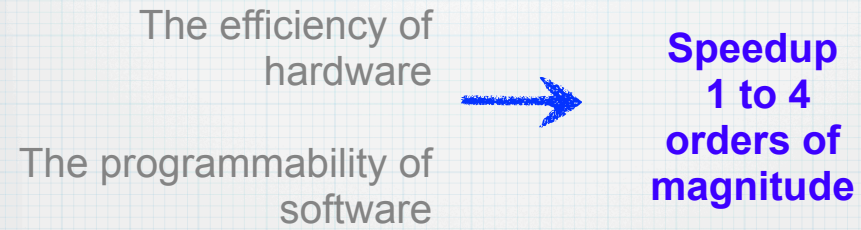
# **FPGA CODE ACCELERATION**

**opportunities & challenges**

## Opportunities: High Throughput

- Multiple studies report
  - wireline throughput on FPGA code
  - one to four orders of magnitude speed-up against software
- V/s GPUs
  - One to two orders speed-up

## why FPGAs as code accelerators?



# efficiency

- Limitations of stored program model

- software is in memory
- software controls datapath
- datapath controls data

- On FPGA

- software is the datapath

# efficiency & parallelism

\*Guo et al. in 2004 Symp. On FPGAs, February 2004

## ○ Highly repetitive computation on streamable data

- efficiency advantage of *spatial computing*:  $\sim 10x^*$
  - parallelism advantage of FPGAs:  $\sim 100x$
  - pipelining advantage:  $\sim 10x$
  - clock rate disadvantage of FPGAs:  $\sim 0.20x$
- $\rightarrow \sim 2,000x$

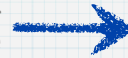
❖ **10%** would get you a paper in a top computer architecture conference!

why <sup>not</sup> FPGAs as code accelerators?

**programmability!**

HDL programming:

Unfamiliar behavioral  
languages  
Tedious low level circuit  
design, Timing sensitive



**Many  
man-months/  
application**

Complex tool-chain:

synthesis, place & route



# Main Challenge

## HLL to circuit

	Imperative HLL	Circuit
language	algorithmic, procedural time insensitive	behavioral, timed
sequencing	temporal & sequential	concurrent
memory	central, virtual	distributed, registers
limit	memory size	circuit area

# Challenge Algorithm

Stored Program

minimize number of steps  
per unit data

Circuit

minimize area used per  
unit data  
maximize unit data per  
clock cycle  
minimize clock cycle time

**Needed: complete application restructuring using spatially  
concurrent algorithms**

# Challenge

## Data size

- **Stored Program**

- fixed word/datapath size: 8, 16, 32, 64, 128, 256 ...

- **Circuit**

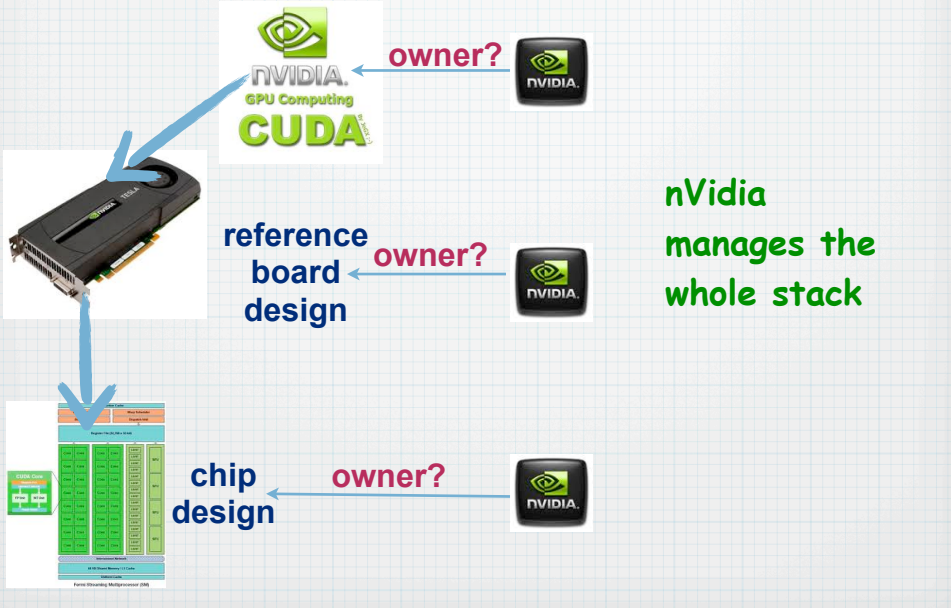
- smaller area: more parallelism & shorter clock cycle ==> **higher throughput**
- customizable data size: to fit dynamic range of data, precision

# Challenge

## Data access

	Stored Program	FPGA
cache memory	✓	X
virtual memory	✓	X
memory mapped I/O	✓	X
standard memory interface	✓	X

# Why are GPUs in HPC?



# **ROCCC 2.0**

**ROOTS & VISION**

# What is ROCCC?

- Riverside Optimizing Compiler for Configurable Circuits
- Designed for the generation of FPGA code accelerators from C
  - Not an EDA tool!
  - Focus is on loop nests and streaming data
    - extensive loop and array analysis for parallelism
- Designed to:
  - achieve same clock speed as hand-written HDL
  - keep the user in full control:
    - algorithm and design space exploration
    - only what is well understood is automated

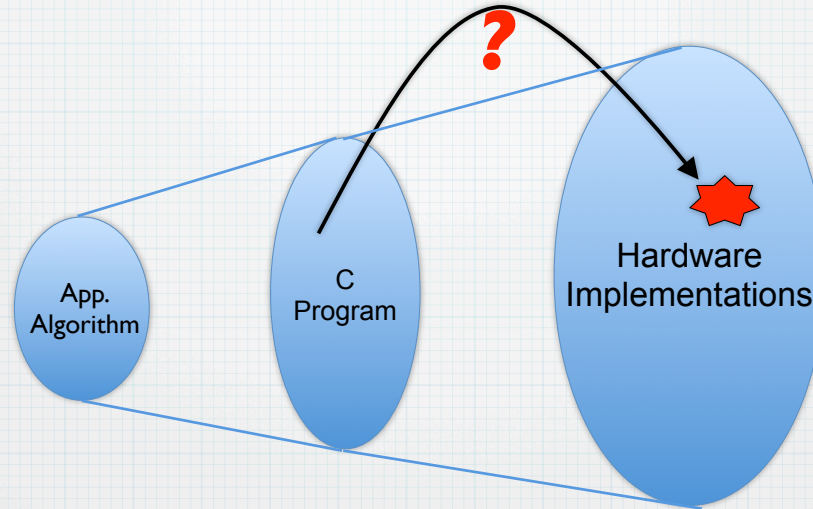
# Origins Of ROCCC 1.0


- SA-C (Single Assignment C): Colorado State
  - Language + compiler project
    - demonstrated efficient hardware based upon single assignment property
- StreamsC: LANL
  - Addition of streaming mechanisms to C, explicit parallelism
  - Now ImpulseC
- ROCCC 1.0
  - Purely top down approach
  - Limits the design space that can be explored



Why ROCCC 2.0?

# Hardware Specification Problem



 my favorite solution

Why ROCCC 2.0?

# Abstractions

## oABSTRACTIONS!

- secret ingredient in the computing revolution of the past 60 years
- none such, yet, for FPGAs

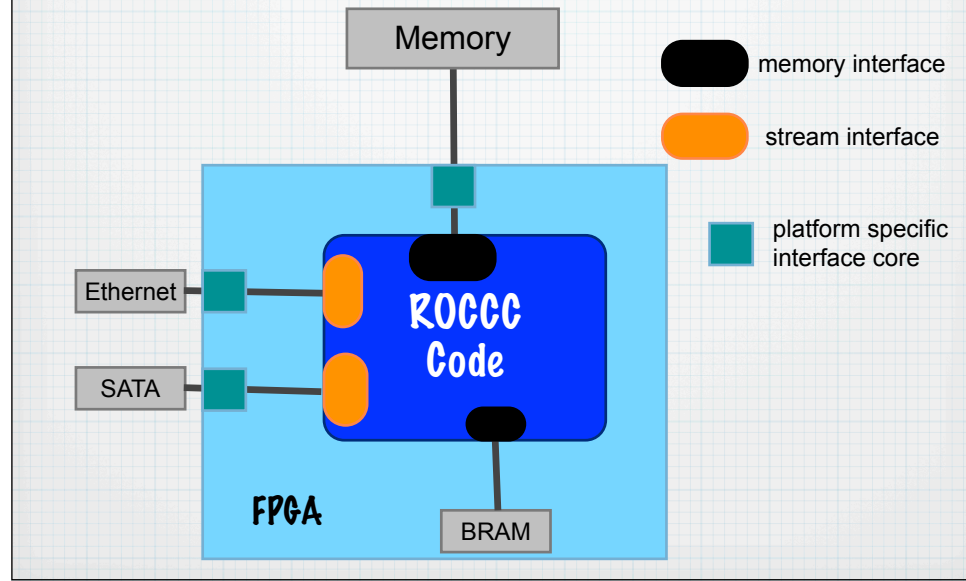
## oCompilers rely on abstractions of the hardware eco-system

- virtual memory, memory mapped I/O etc

## oROCCC 2.0: two data source abstractions

- streams & random access memory
- inferred by compiler

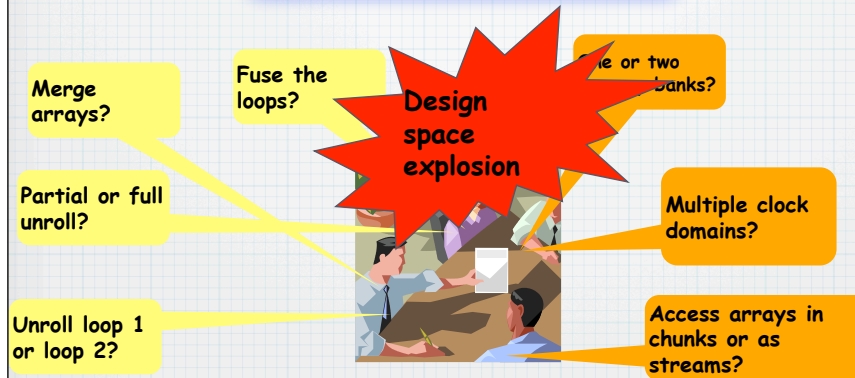
# Interface Abstractions



Why ROCCC 2.0?

# Exploration

example:  
two loops nested accessing two arrays



# ROCCC 2.0 Vision

○ Give user better control over outcome

○ Use C to build modular computing elements

- modular design
- bottom-up construction
- composable and reusable modules

○ Abstract away platform specifics

- from compiler code generation: interface to memories

○ Support efficient design space exploration

- user driven, platform aware



# **ROCCC 2.0**

IMPLEMENTATION

# ROCCC 2.0

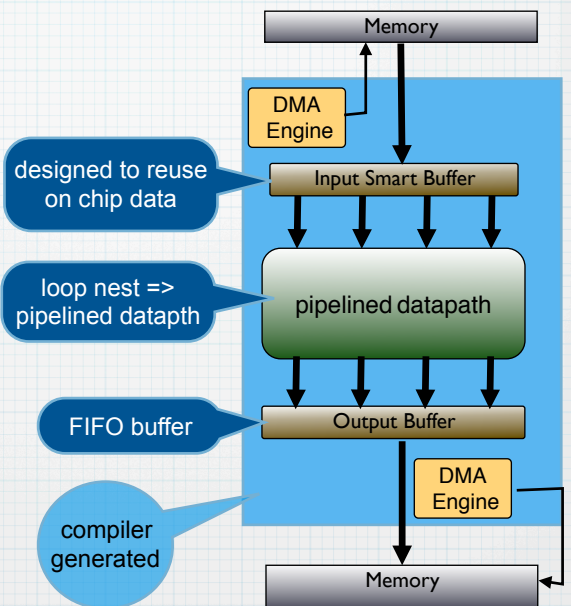
- Enable higher level programming of FPGAs
  - Transform a subset of C
  - Focus on hardware-appropriate algorithms as opposed to generic C
- Enable reusability, composability, and portability
  - Modular approach to hardware construction
  - Separate platform interface from compiler
  - Tuning of applications to particular platform accomplished through optimizations - GUI
- Generate high throughput circuits
  - Extensive optimizations
- Free, open source toolset

# Decoupled Execution

Every cycle:

- Input smart buffer pushes data to datapath
- Output buffer collects data

Memory can be on or off chip or board



# Modular Bottom-up Design

## ○ *Modules and Systems*

### ○ Modules:

- Self contained hardware computational blocks, expressed in C
- Bottom-up composition of modules
- Standard database (SQLite3) contains all information on modules either in C, VHDL, or netlists

### ○ Systems

- Process streams of data in critical loops
- Can use modules
- Many user controllable optimizations

# Module Code

- Interface
  - A struct that specifies inputs and outputs
- Implementation
  - A function that describes computation inside the black box
  - All outputs must be assigned

```
// Interface
typedef struct
{
    int realOne_in ;
    int imagOne_in ;
    int realTwo_in ;      input registers
    int imaTwo_in ;
    int realOmega_in ;
    int imagOmega_in ;

    int A0_out ;
    int A1_out ;          output registers
    int A2_out ;
    int A3_out ;
} FFT_t ;

// Implementation
FFT_t FFT(FFT_t f)
{
    int tmp1 ;           internal registers
    int tmp2 ;

    tmp1 = f.realOmega_in * f.realTwo_in ;
    tmp2 = f.imagOmega_in * f.imagTwo_in ;

    f.A0_out = f.realOne_in + tmp1 - tmp2 ;
    // The other outputs computations go here...
    return f ;
}
```

# Module Instantiation

- All Modules accessible as functions
  - "roccc-library.h"
  - Called as any other C function
- Standard database maintains all module information
- GUI supports automatic insertion of module instantiations

```
#include "roccc-library.h"

typedef struct
{
    int input0_in ;
    // Other inputs

    int tmp0_out ;
    // Other outputs
} FFTOneStage_t ;

FFTOneStage_t FFTOneStage(FFTOneStage_t t)
{
    FFT(t.input0_in,
        t.omega0_in,
        t.input16_in,
        t.omega1_in,
        t.input17_in,
        t.input1_in,
        t.temp0_out,
        t.temp1_out,
        t.temp2_out,
        t.temp3_out) ;

    // Others ...

    return t ;
}
```

*module instantiation*

# System Code

```
#include "roccc-library.h"

void firSystem()
{
  int A[100] ;      identified as input & output streams
  int B[100] ;

  int i ;
  int endValue ; ← actual value passed to hardware at runtime
  int myTmp ;

  for(i = 0 ; i < endValue ; ++i)
  {
    // Data reuse is detected in
    // loops by the compiler
    FIR(A[i], A[i+1], A[i+2],
        A[i+3], A[i+4], myTmp) ; ← module instantiation can be duplicated if loop is unrolled
    B[i] = myTmp ;
  }
}
```

# ROCCC Transformations

## ○ Standard

- CFG, DFG, and UD/DU
- Constant propagation/folding
- Copy propagation
- Dead and unreachable code elimination
- Scalar renaming
- Common subexpression elimination

## ○ Loop

- Unrolling
- Fusing
- Interchange
- Peeling
- Tiling
- Unswitching
- Invariant code motion

## ○ Array

- Scalar replacement
- Array renaming
- RAW and WAW elimination
- Feedback elimination
- Systolic array generation
- Sequential loop pipelined unrolling
- Temporal CSE

specific to hardware



# Hardware Optimizations

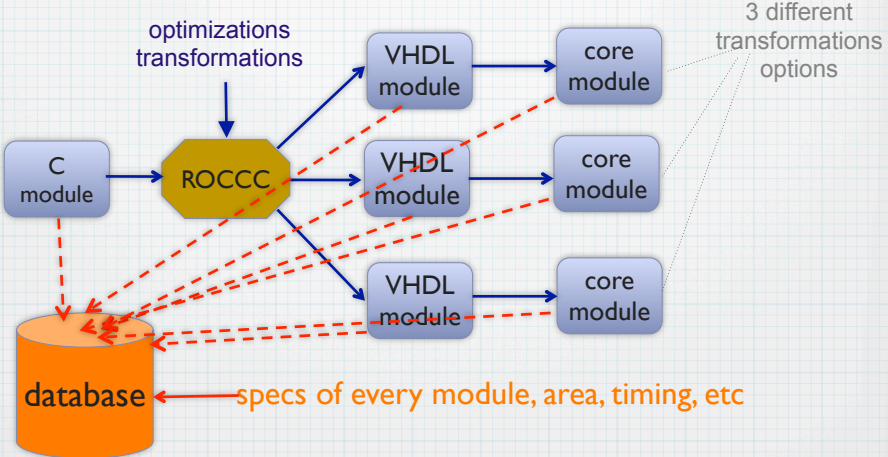
- Redundancy Specification
- Pipelining and retiming
  - Ability to specify weights for basic operations to guide the number of pipeline stages
- Smart Buffer Generation
  - Keep memory accesses that will be reused to minimize off chip requests
- Systolic array generation
  - Wavefront algorithms are coded as nested *for* loops that iterate over a two-dimensional array
  - Specific set of optimizations to transform into a systolic array
- Temporal common subexpression elimination
  - Remove common code across loop iterations
  - Extends to the removal of modules across loop iterations and addition of feedback variables

# Platform Independent Interfacing

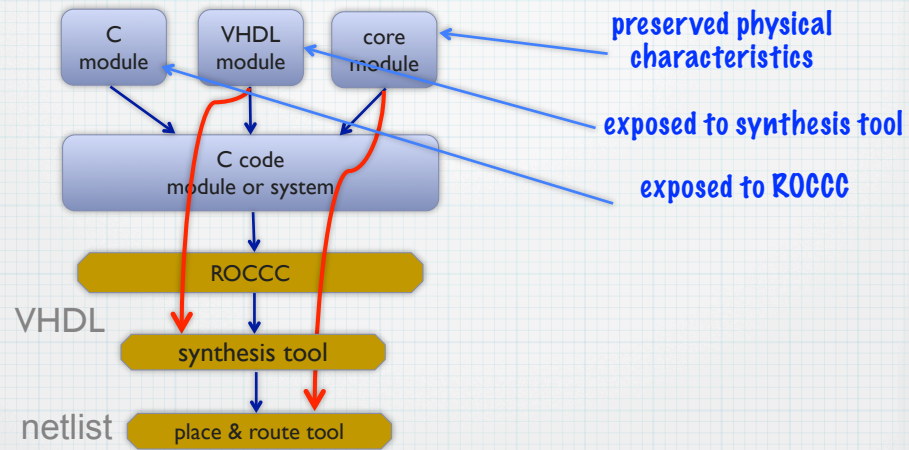
- ROCCC Generates Generic Hooks
  - Memories - address generator
  - Streams - FIFOs
- Each interface configurable by the user on a stream-by-stream basis
  - Number of outgoing memory requests
  - Number of reads/writes per clock cycle
- Optimized for maximum throughput
  - Can support reading values every clock cycle or as fast as can be fed

# ROCCC 2.0 Modules

A module, in a module, in a ... you get the idea



# Importing Modules



# ROCCC 2.0 Features

- Support for modular circuit design in C
  - Reusable modules, in C, VHDL or cores
  - Bottom-up design and top-down designs supported
  - External IP cores integrated in ROCCC designs
  - Partial compilation and partial synthesis with reuse
  - Inlining and black box instantiation supported
- Subset of C with no additional keywords
  - ROCCC-code compiled and run with a software compiler
- Support for variable bit width
  - User control over precision of arithmetic operations
- Floating point operations
  - half (16-bit), single (32-bit) or double precision (64-bit)

# ROCCC 2.0 Features

- Automatic VHDL testbench code generation
- Platform independent code generation
  - allows fast re-targeting
- User control over all transformations, optimizations
  - Automate only what is fully understood, by us and the user!
  - High-level transformations: loops and arrays
  - Low-level optimizations: DFG and circuit levels
  - Fine-grained control over low level optimizations
    - pipelining depth, fanout tree generation
    - parallelization of input and output streams

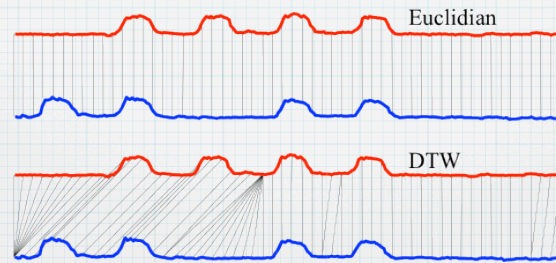
# **PERFORMANCE?**

dynamic time warping

a data mining example

# Dynamic Time Warping

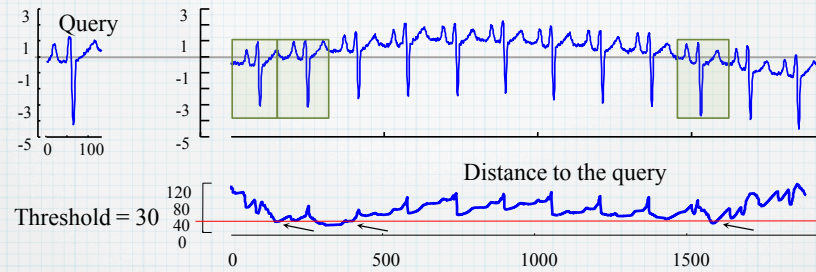
- Data mining using subsequence similarity search on time series
- Relies on dynamic programming on two strings: signal and query



E. Keogh and C.A. Ratanamahatana. Exact indexing of dynamic time warping. Knowledge and Information Systems, 7(3):358-386, 2005.

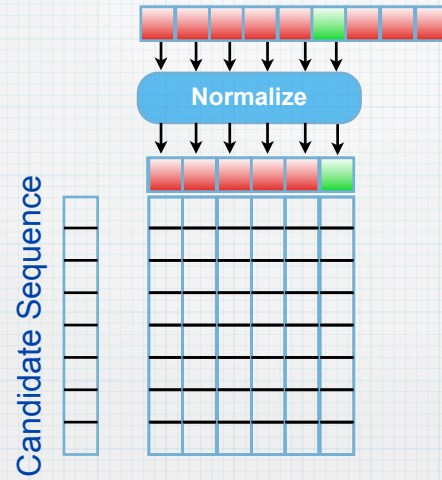


# Dynamic Time Warping

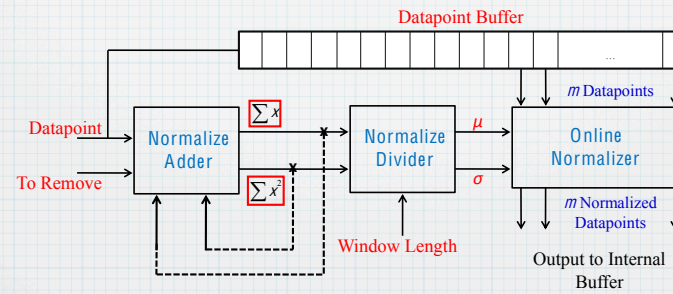
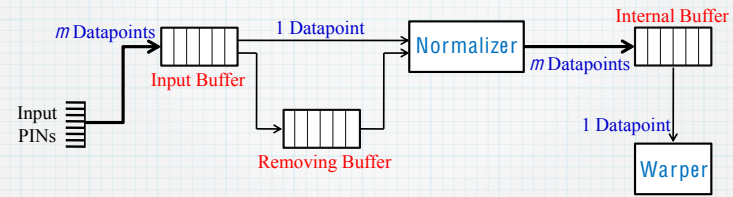


- Detect a pattern across a time series
- Normalization of data is extremely important

# DTW - Software Solution



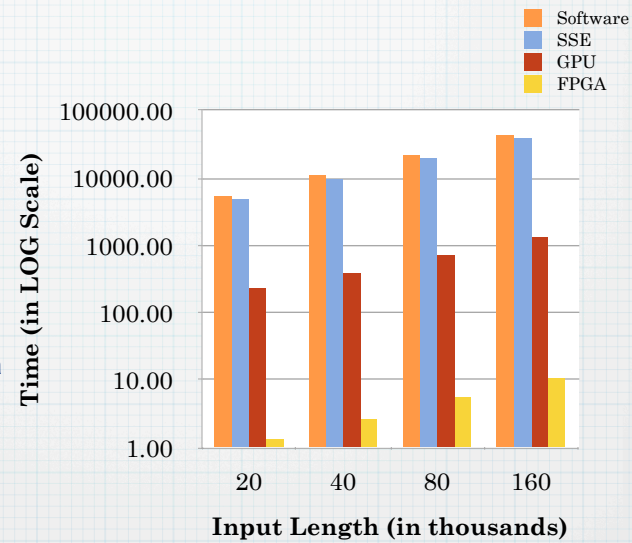
# DTW - Hardware Architecture



# DTW - Performance

## ○ Evaluation Platforms:

- Software: Intel Pentium i7 2.66 GHz, 6 GB RAM
- FPGA: Xilinx Virtex-5 LX-330
- GPU: NVIDIA Tesla T10



D. Sart, A. Mueen, W. Najjar, V. Niennattrakul, and E. Keogh. *Accelerating Dynamic Time Warping Subsequence Search with GPUs and FPGAs*, in IEEE Int. Conf. on Data Mining, Sydney, Australia, Dec. 2010.

# Details

## ○ Clock rate

- Normalization Module runs at 174.6 MHz.
- Warping Matrix runs at 240 MHz.

## ○ Area

- Normalization Module requires 13% of FPGA Area
- Warping Matrix requires 7% of FPGA Area

## ○ Throughput:

- Normalization unit generates results every clock cycle
- Warping Matrix generates a result every 128 cycles
- 8 Warping Matrix units are used

# **PERFORMANCE?**

comparing to GPUs  
on image algorithms

# Evaluation Platforms

## ○ GPU - NVidia Tesla processor

- 30 Streaming Multiprocessors (8 cores each) total of 240 cores
- Both GPGPU-Sim and measured values running on a Tesla processor
- 1, 4, 10, and 30 Streaming Multiprocessor configurations tested

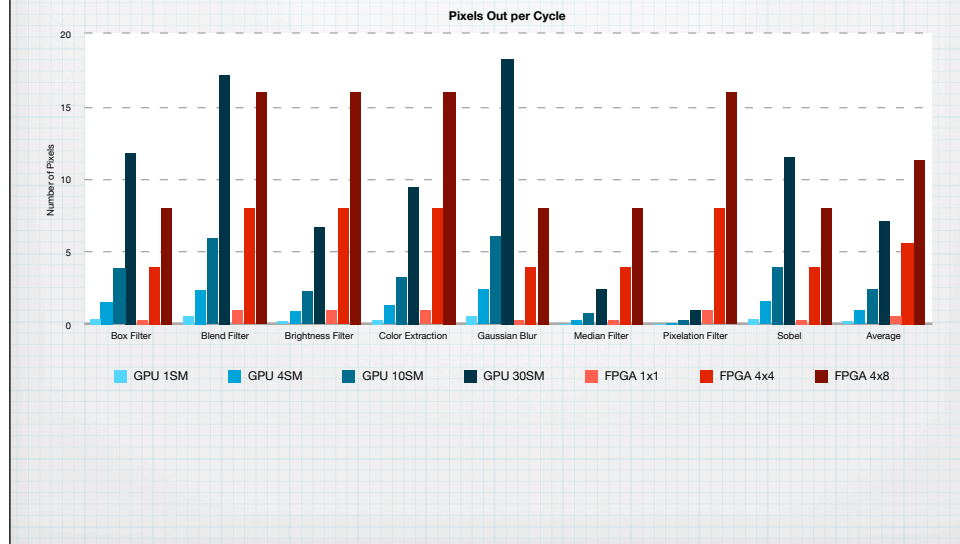
## ○ FPGA - Virtex 6 LX760

- Programmed in ROCCC compliant C
- Unrolled for different levels of parallelism
  - One loop body, 4x4 loop bodies, and 4x8 loop bodies
- All hardware for one application created through the tuning parameters of ROCCC and one source file

## ○ Benchmarks

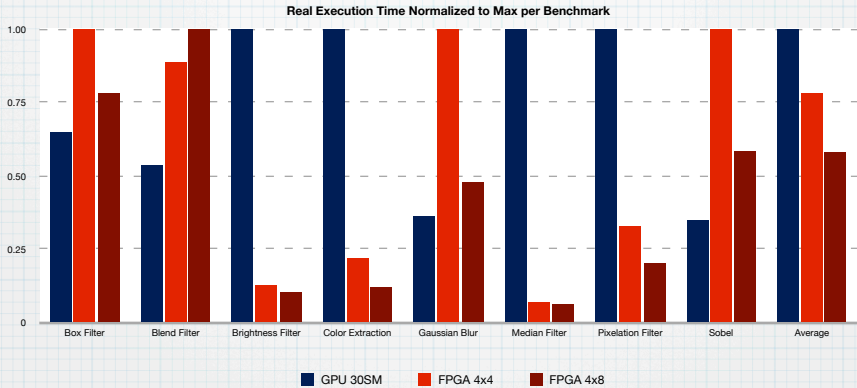
- Brightness Filter
- Color Extraction
- Box Filter
- Gaussian Blur
- Blend
- Sobel
- Median Filter
- Pixelation

# Throughput Comparison





# Execution Time



■ GPU 30SM ■ FPGA 4x4 ■ FPGA 4x8

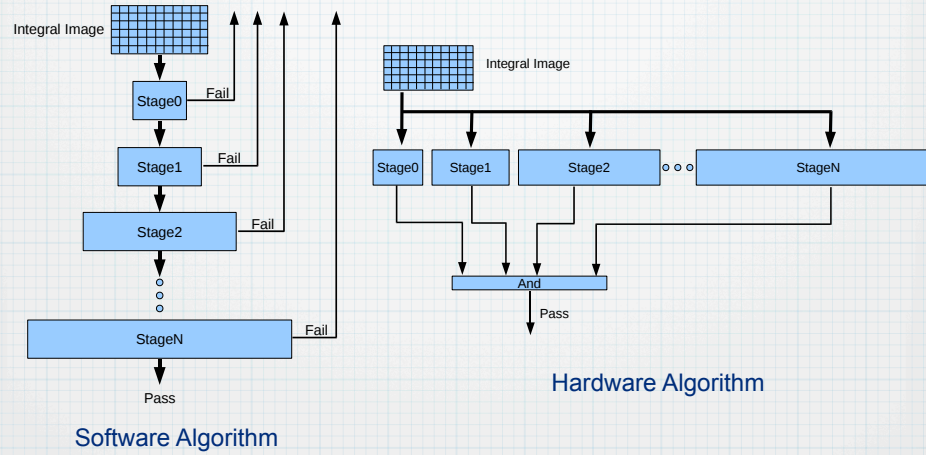
# **PRODUCTIVITY?**

face detection example

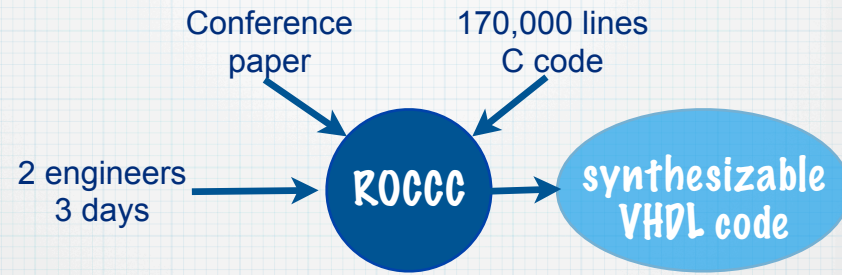
# Viola-Jones Face Detection

- Detect forward facing faces
  - Works on a sliding window over an image
  - Window must be scaled several times to detect different sized faces
- 24 classifier stages
  - Many features in each stage
  - Each feature is identical computation with different values
- Software runs at 1.5 seconds / frame (.66 frames/sec) on average
  - 600x600 sized images

# VJ - Software v/s Hardware



# Productivity



# Results

## ○ Productivity gains

- Well known, de facto standard, available in code repositories, OpenCV, 17,000 lines of C code
- Two engineers, with no prior knowledge of code or application domain (Computer Vision)
- read original paper, open the code distribution
- ROCCC ported algorithm developed in 3 days, resulting in synthesizable VHDL

## ○ Approximately 3000 features in software

- 1 feature 0.2% of the FPGA
- Approximately 510 features per FPGA, 2040 total on the HC-1

## ○ Stage optimizations

- Last 8 stages (1467 total features) very close to original algorithm
- 343 frames per second
- 520X improvement over software

# Other Applications

## ○ Short strings matching

- bioinformatics
- 90 - 200x over Bowtie

## ○ XML query matching

- whole twig matching, expressed as Xpath
- equivalent to CFG recognition
- 2 to 4 orders of magnitude over CPU and GPU
- no memory off loading

# Conclusion

- ROCCC 2.0 - A third generation C to HDL tool
  - designed for code acceleration (not general hardware design)
  - extensive compile-time optimizations and transformations
  - modular bottom-up designs with code reuse
  - code generation independent of target platform, first attempt at abstractions
- Productivity: >10x over HDL design
  - small cost in additional area, being improved right now
  - much better results on large codes (≠small kernels)
- Ever widening spectrum of applications



# Questions?

Thank you!

ROCCC 2.0

<http://www.jacquardcomputing.org>