Unlocking FPGAs Using High-Level Synthesis Compiler Technologies

Fernando Martinez Vallina, Henry Styles

Xilinx

Feb 22, 2015

Why are FPGAs Good

- **Scalable, highly parallel and customizable compute**
  - – 10s to 1000s of processing elements
  - – Processing elements can be fined tuned to specific work loads
  - – Bit level integer and fixed point operations

- **Custom Memory Architectures**
  - – Memories sized to the application
  - – Memory elements placed next to datapath

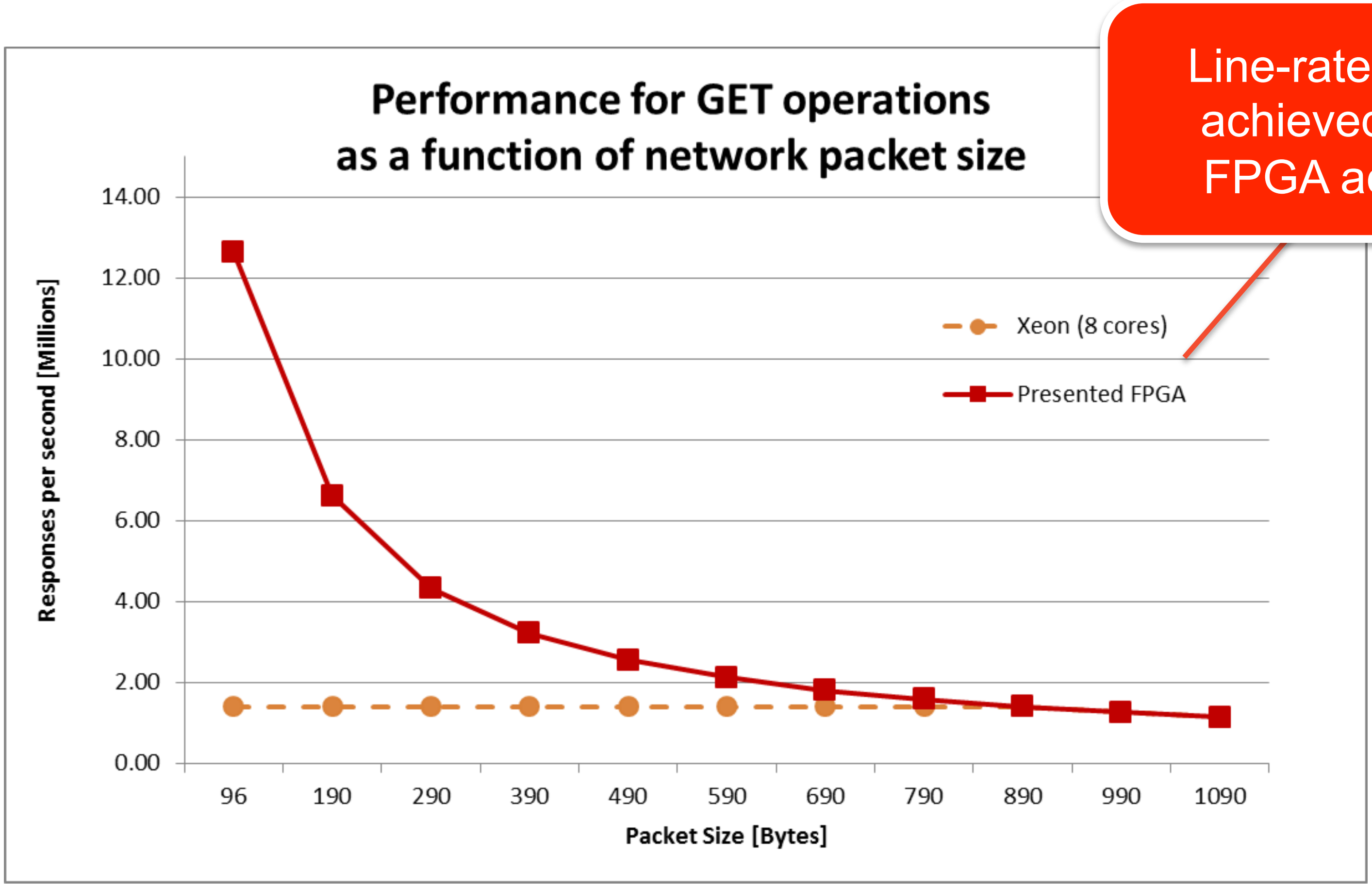
- **Custom Data Movement and I/O**
  - – More silicon area dedicated to wires and data transport than any other device
  - – Direct low latency connections between I/O pins and compute

FPGA Applications for Data Center

Application	FPGA Benefit
Network Function Virtualization Encryption	Efficient bit level operations
Dictionary compression (i.e. Deflate)	Customized memory hierarchy and efficient string matching
Search categorization	Control flow dominated, thousands of active state machines
Image classification by machine learning	Ability to customize data movement to specific working sets
Image transcoding	Fine grained parallelism with customized memory hierarchy
Genome sequencing	Thousands of parallel bit level operations
Key Value Store	Low latency I/O

FPGAs Offer Significant Speedup Potential vs CPU and GPU

Key Value Store Acceleration with FPGAs

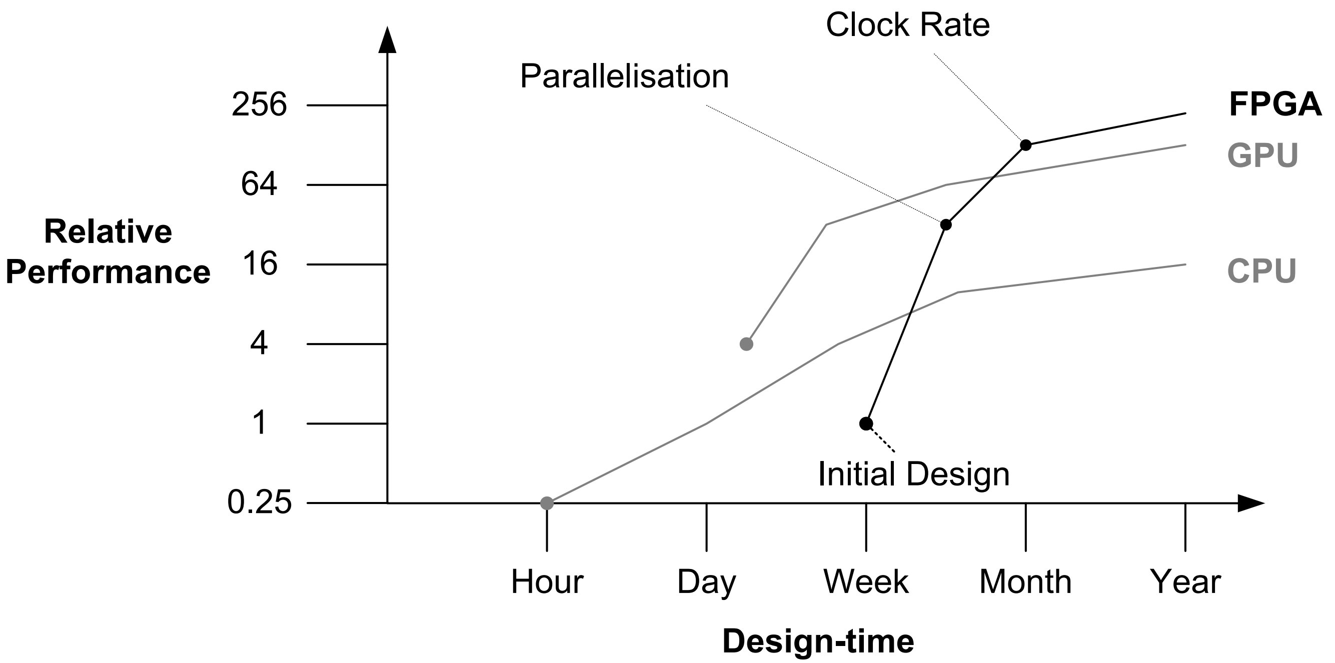


Line-rate maximum achieved by Xilinx FPGA accelerator

Up to 36x in performance/Watt demonstrated
Plus 10-100x lower latency

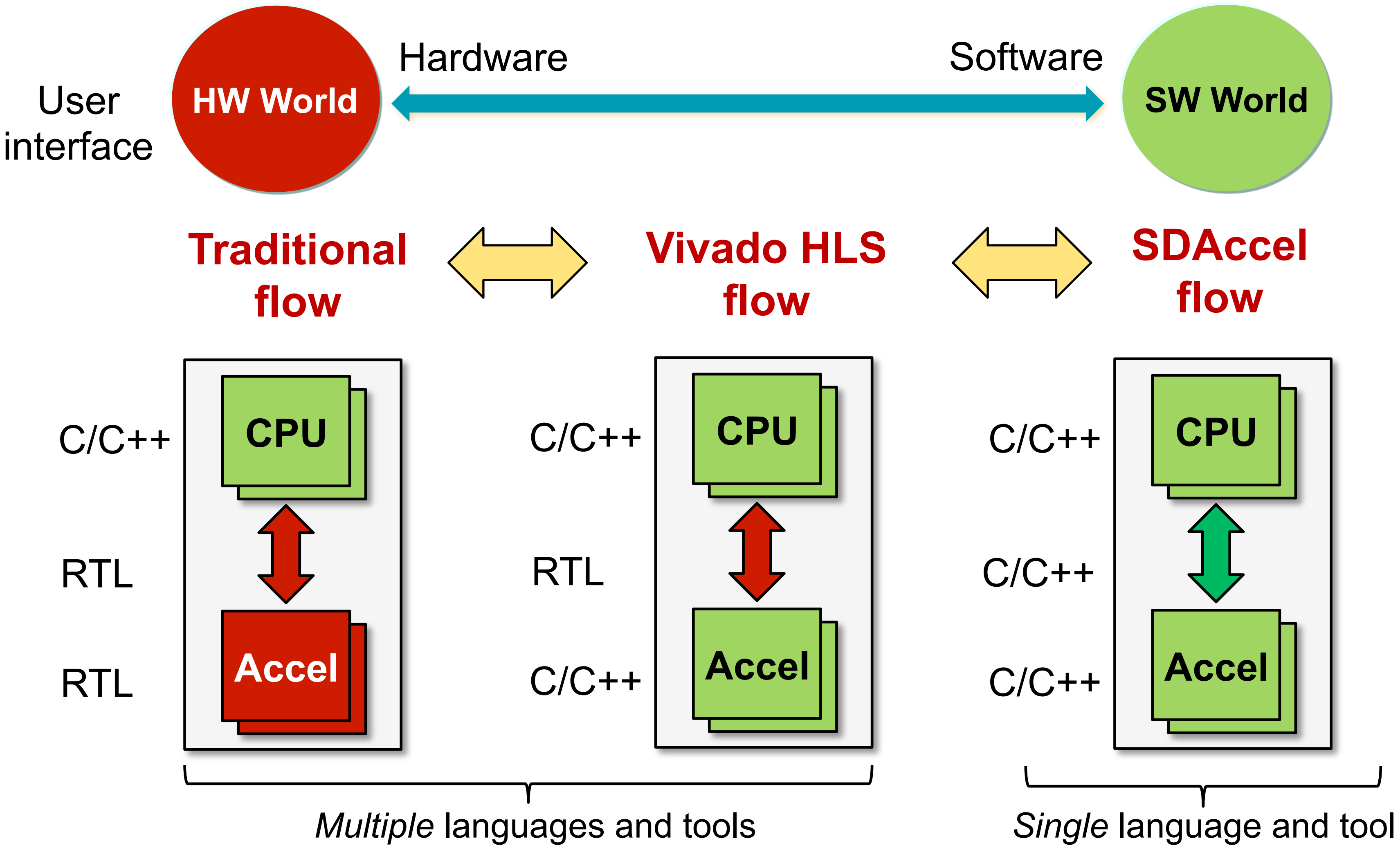
Productivity Challenges for Software

(David Thomas, Imperial College, UK)



- **FPGAs provide large speed-up and power savings – *at a price!***
  - Days or weeks to get an initial version working
  - Multiple optimisation and verification cycles to get high performance

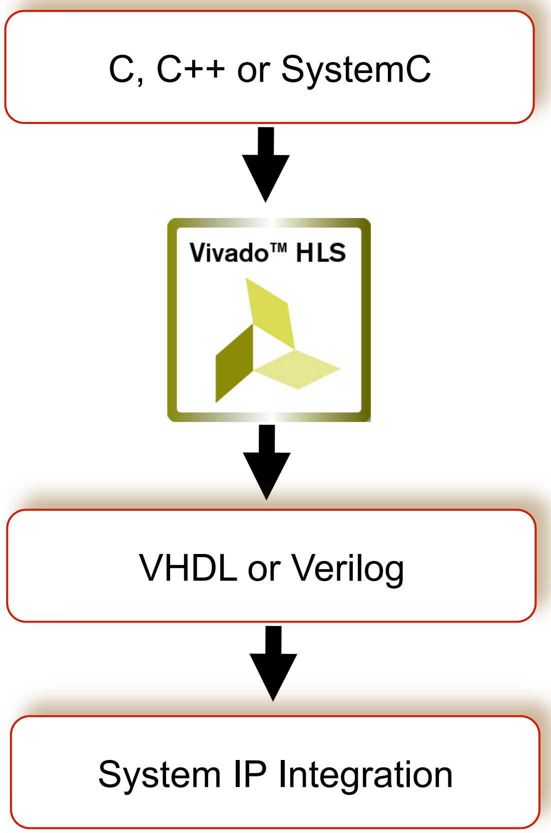
Programming Flow Evolution

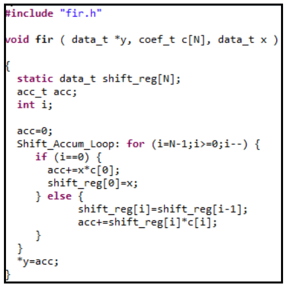


Vivado HLS

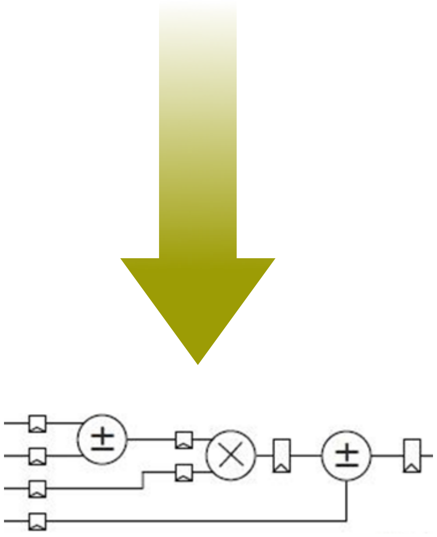
Raising the Abstraction for IP Creation

Vivado High-Level Synthesis





```
#include "fir.h"

void fir ( data_t *y, coef_t c[N], data_t x )

{
  static data_t shift_reg[N];
  acc_t acc;
  int i;

  acc=0;
  Shift_Accum_Loop: for (i=N-1;i>=0;i--) {
     if (i==0) {
       acc+=x*c[0];
       shift_reg[0]=x;
     } else {
            shift_reg[i]=shift_reg[i-1];
            acc+=shift_reg[i]*c[i];
     }
  }
  *y=acc;
}
```



Algorithmic Specification

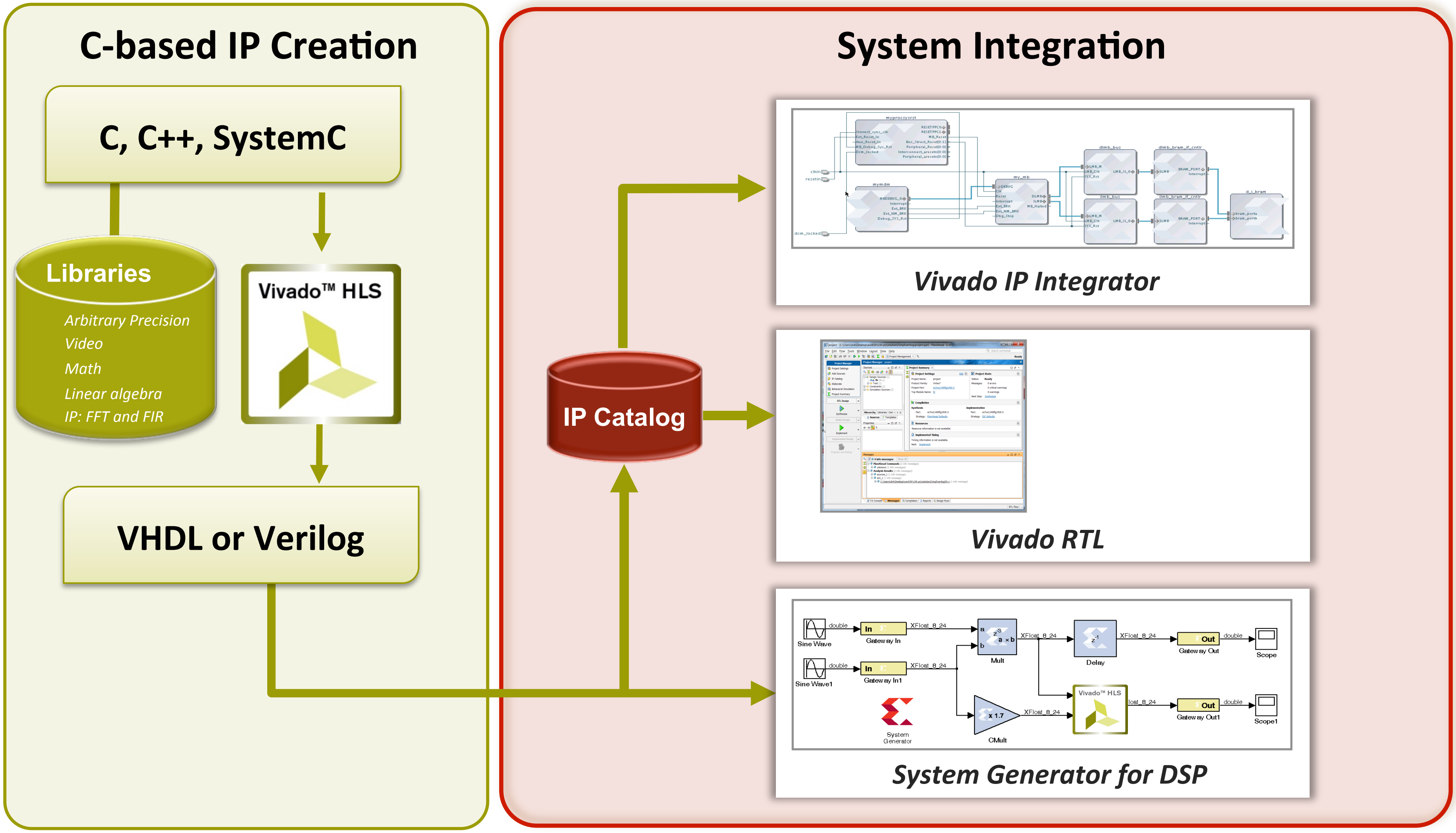
Micro Architecture Exploration

RTL Implementation

Comprehensive Integration with the Xilinx Design Environment

Accelerates Algorithmic C to RTL IP integration

Vivado HLS System IP Integration Flow



C-based IP Creation

C, C++, SystemC

Libraries

Arbitrary Precision
Video
Math
Linear algebra
IP: FFT and FIR

Vivado™ HLS

VHDL or Verilog

System Integration

IP Catalog

Vivado IP Integrator

Vivado RTL

System Generator for DSP

Vivado HLS Integrates into System Flows

Design Decisions

Decisions made by designer

- Functionality
  - As implicit state machine
- Performance
  - Latency, throughput
- Interfaces
- Storage architecture
  - Memories, registers banks etc…
- Partitioning into modules
- Design Exploration

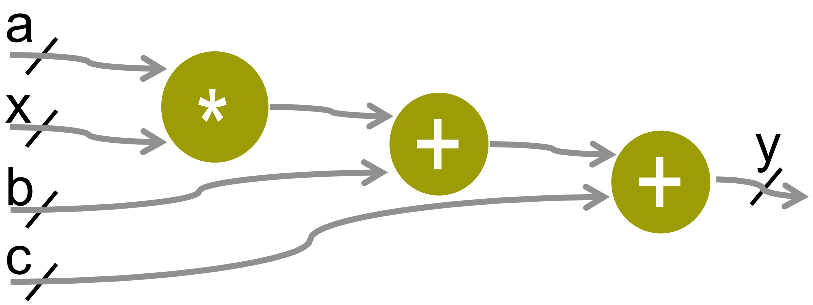
Decisions made by the tool

- State machine
  - Structure, encoding
- Pipelining
  - Pipeline registers allocation
- Scheduling
  - Memory I/O
  - Interface I/O
  - Functional operations

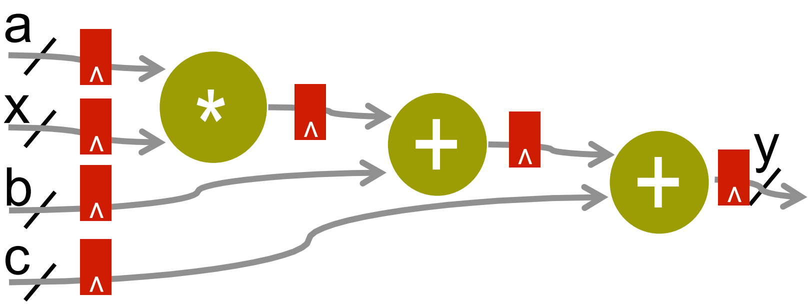
Datapath Synthesis

*Example: y = a*x + b + c;*

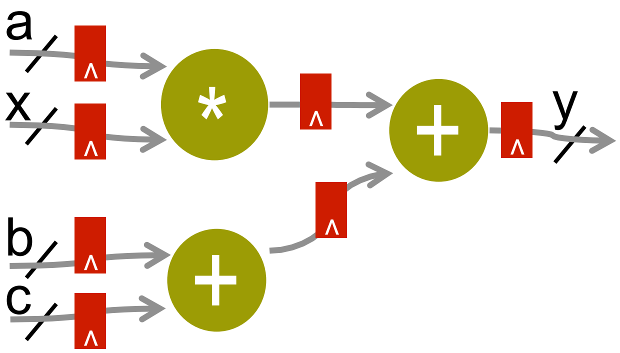
1. HLS begins by extracting a data flow graph (DFG), a functional representation



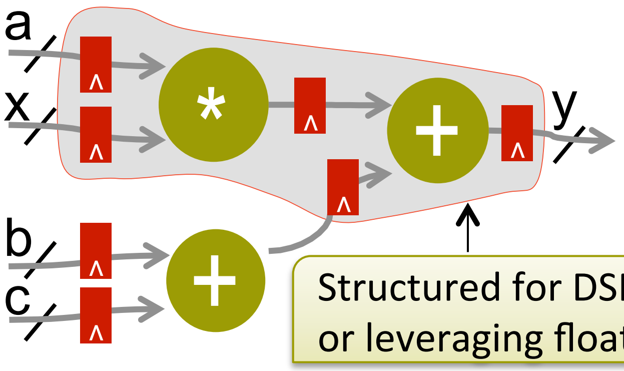
2. Accounts for target Fmax to determine minimum required pipelining (not yet the optimal implementation)



3. Expression balancing for latency reduction Restructuring to optimize fabric resources



4. Restructuring for optimized DSP48

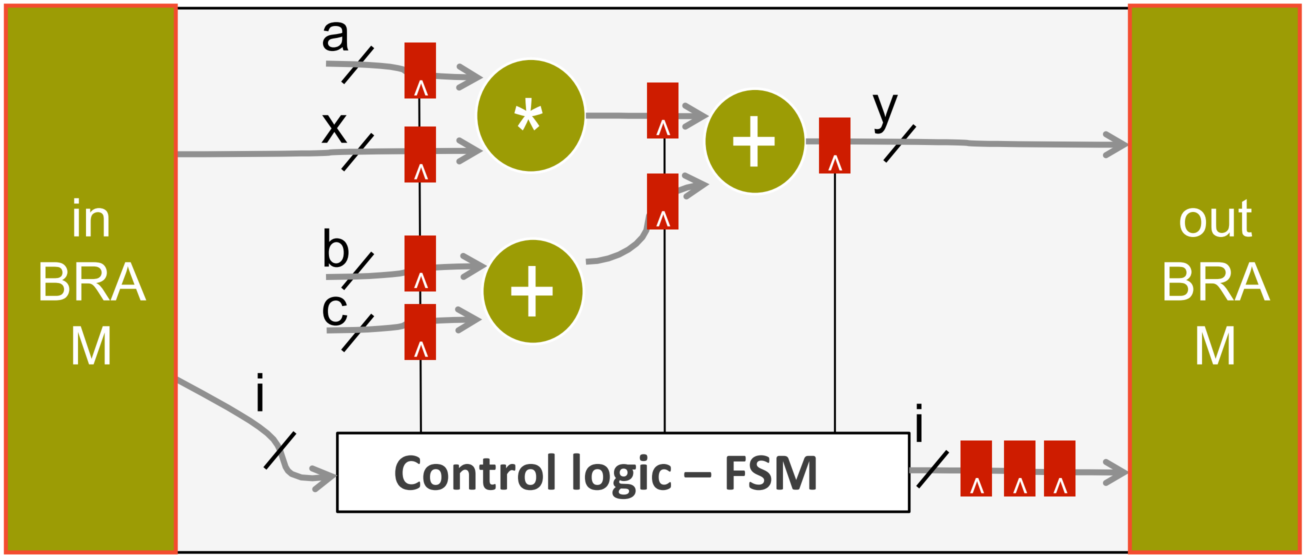


Structured for DSP inference or leveraging floating point IPs

Interface Synthesis – Completing the design…

- C function arguments become RTL interface ports

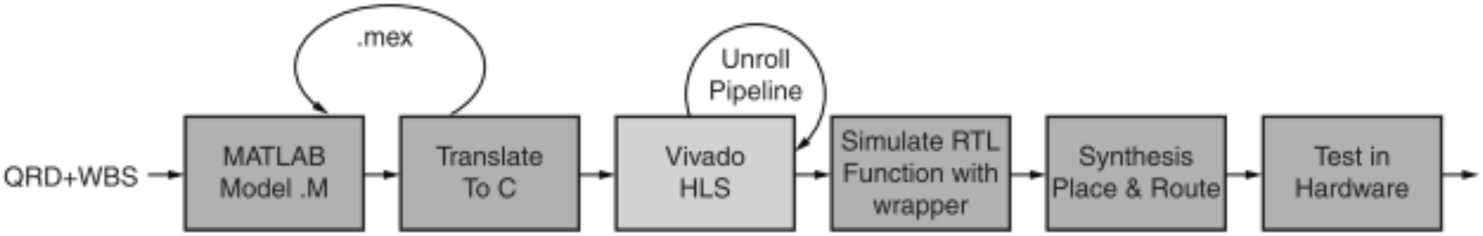
```
f(int in[20], int out[20]) {
 int a,b,c,x,y;
 for(int i = 0; i < 20; i++) {
     x = in[i]; y = a*x + b + c; out[i] = y;
 }
```

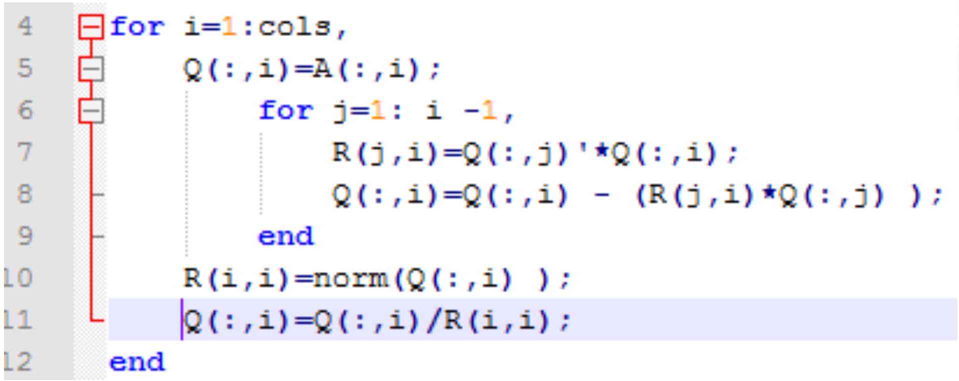


The State Machine Automatically Adapts to the Design Interface

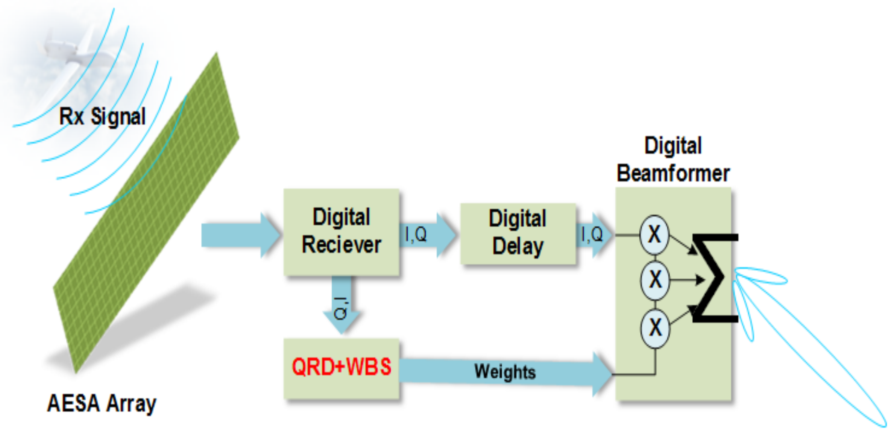
Vivado HLS Examples

MGS QRD + Weight back Substitution → STAP
This is the Hardest Floating Point Problem for Hardware



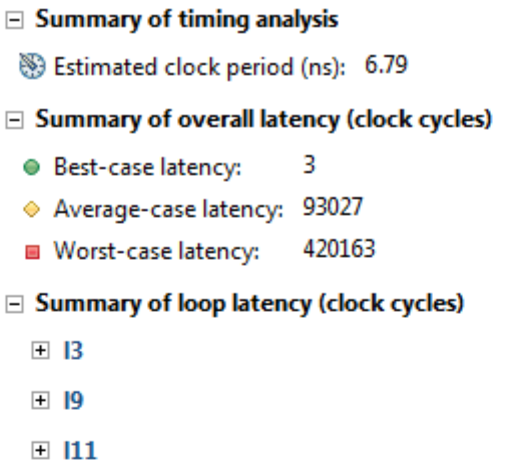


```
for i=1:cols,
    Q(:,i)=A(:,i);
        for j=1: i -1,
            R(j,i)=Q(:,j)'*Q(:,i);
            Q(:,i)=Q(:,i) - (R(j,i)*Q(:,j) );
        end
    R(i,i)=norm(Q(:,i) );
    Q(:,i)=Q(:,i)/R(i,i);
end
```



- This Real System Design took 6 Months using VHDL. Using HLS, it took 4 Hours!
- 260x Speed-Up. This does not include system integration which will be Faster!

MGS QRD+WBS RESULTS



⊟ **Summary of timing analysis**

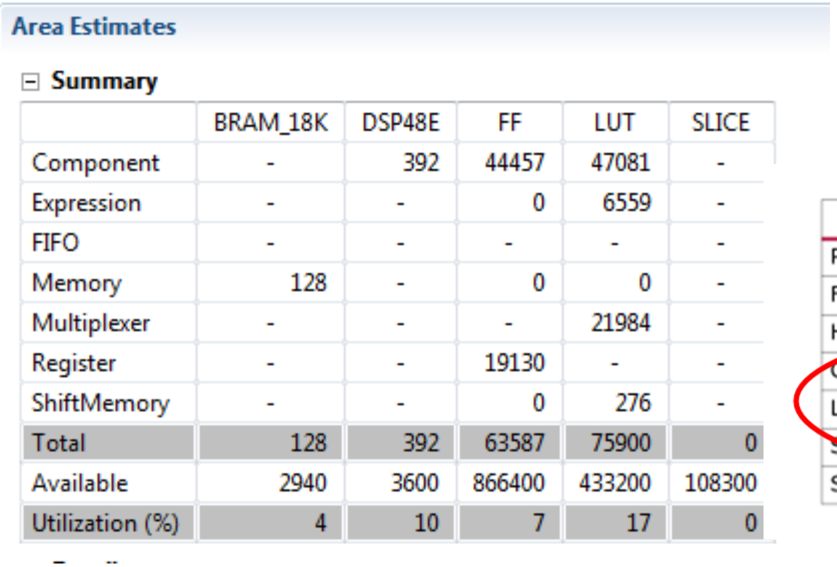
Estimated clock period (ns): 6.79

⊟ **Summary of overall latency (clock cycles)**

● Best-case latency:	3
◇ Average-case latency:	93027
■ Worst-case latency:	420163

⊟ **Summary of loop latency (clock cycles)**

- ⊞ l3
- ⊞ l9
- ⊞ l11

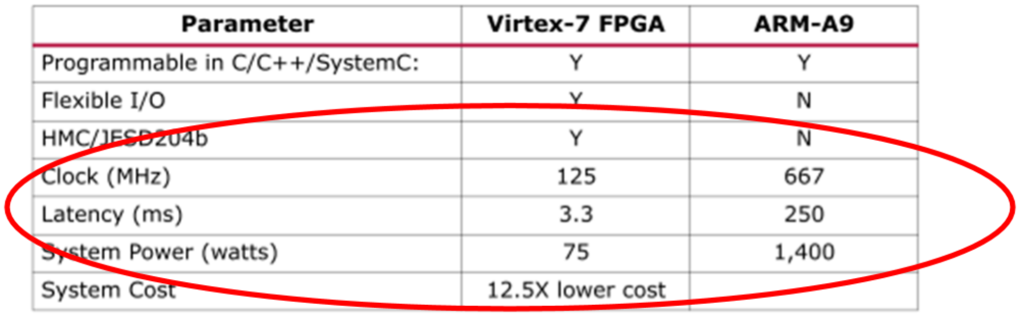


Area Estimates

⊟ **Summary**

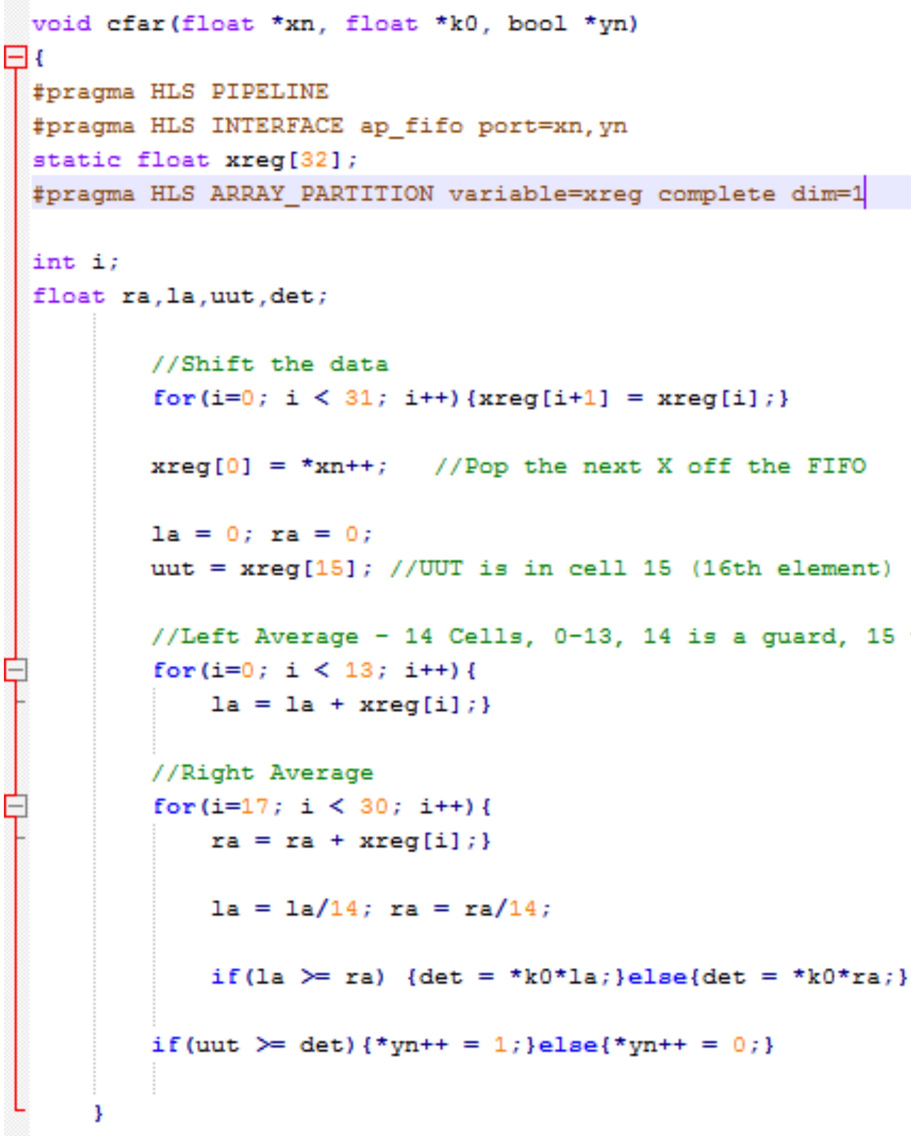
	BRAM_18K	DSP48E	FF	LUT	SLICE
Component	-	392	44457	47081	-
Expression	-	-	0	6559	-
FIFO	-	-	-	-	-
Memory	128	-	0	0	-
Multiplexer	-	-	-	21984	-
Register	-	-	19130	-	-
ShiftMemory	-	-	0	276	-
Total	128	392	63587	75900	0
Available	2940	3600	866400	433200	108300
Utilization (%)	4	10	7	17	0

- **128x64 Complex FP ~3.3ms (125 MHz)**
- **FP Matrix Inversion thanks to Vivado HLS is trivial in FPGAs**
- **If it can be done Mathematically, It can be done in HLS**
- **C/C++ easily moves to CPUs**
  - Truly Portable Libraries
  - Rapid Trade Space Exploration

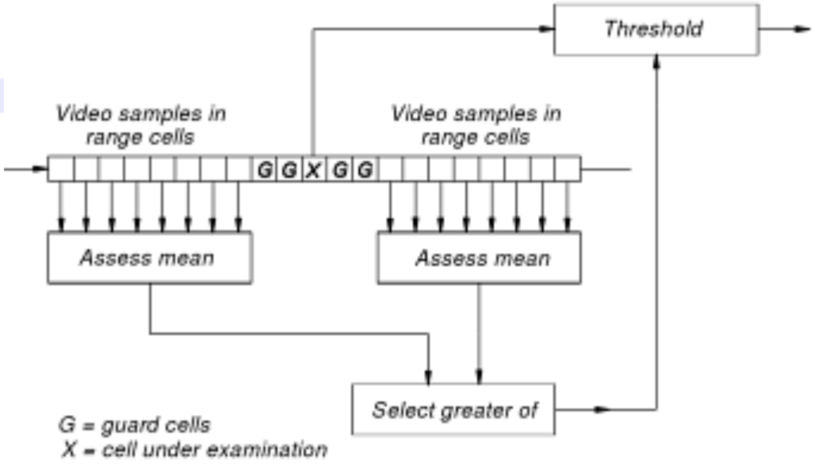


Parameter	Virtex-7 FPGA	ARM-A9
Programmable in C/C++/SystemC:	Y	Y
Flexible I/O	Y	N
HMC/JESD204b	Y	N
Clock (MHz)	125	667
Latency (ms)	3.3	250
System Power (watts)	75	1,400
System Cost	12.5X lower cost	

HLS Simple CA- CFAR
32 Cells, 14 Left, 14 Right, 2 Guard

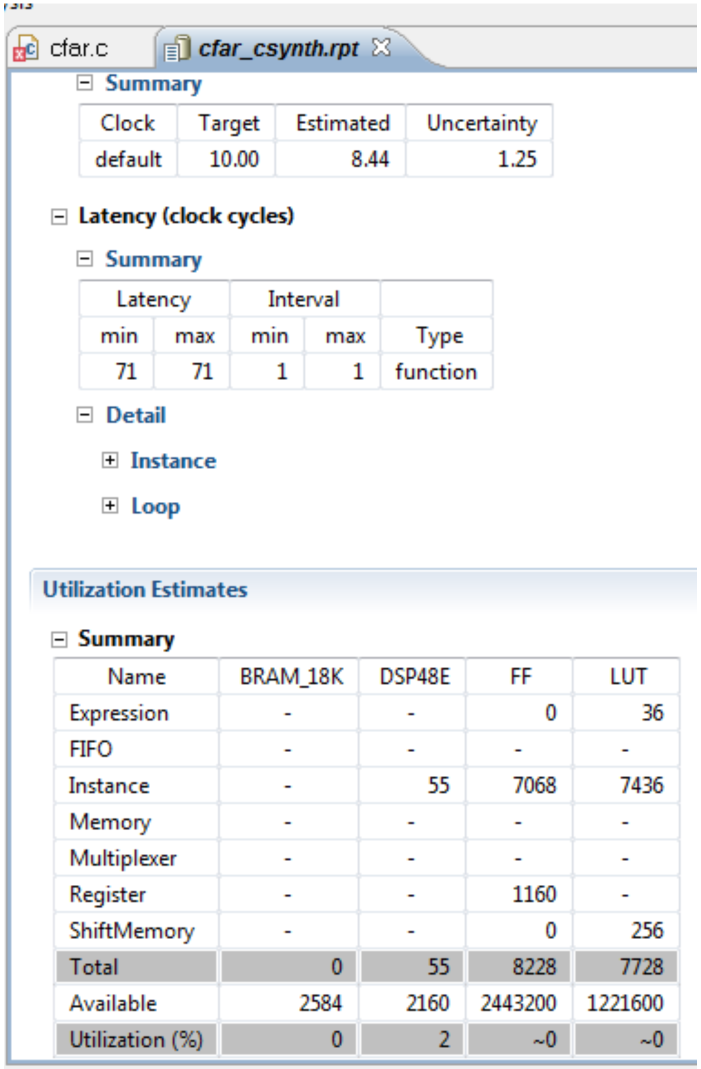


```
void cfar(float *xn, float *k0, bool *yn)
{
#pragma HLS PIPELINE
#pragma HLS INTERFACE ap_fifo port=xn,yn
static float xreg[32];
#pragma HLS ARRAY_PARTITION variable=xreg complete dim=1

int i;
float ra,la,uut,det;

        //Shift the data
        for(i=0; i < 31; i++){xreg[i+1] = xreg[i];}

        xreg[0] = *xn++;    //Pop the next X off the FIFO

        la = 0; ra = 0;
        uut = xreg[15]; //UUT is in cell 15 (16th element)

        //Left Average - 14 Cells, 0-13, 14 is a guard, 15 uut, 16 guard, 17-30
        for(i=0; i < 13; i++){
            la = la + xreg[i];}

        //Right Average
        for(i=17; i < 30; i++){
            ra = ra + xreg[i];}

            la = la/14; ra = ra/14;

            if(la >= ra) {det = *k0*la;}else{det = *k0*ra;}

        if(uut >= det){*yn++ = 1;}else{*yn++ = 0;}

    }
```



Threshold
Video samples in range cells
Video samples in range cells
G
G
X
G
G
Assess mean
Assess mean
Select greater of
G = guard cells
X = cell under examination

CFAR RESULTS



Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.44	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
71	71	1	1	function

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	36
FIFO	-	-	-	-
Instance	-	55	7068	7436
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	1160	-
ShiftMemory	-	-	0	256
Total	0	55	8228	7728
Available	2584	2160	2443200	1221600
Utilization (%)	0	2	~0	~0

- Took about 10 minutes to design
- Uses only 55 DSPs!
- Floating Point
- To process more Cells—
  - Just change the C code and rerun HLS
- Optimize to reduce latency/Area—
  - At expense of increased DSP48 usage
- This is just a starting point…
  - Floating point lends itself to QRD for
  - STAP
  - CFAR
  - Adaptive Beamforming

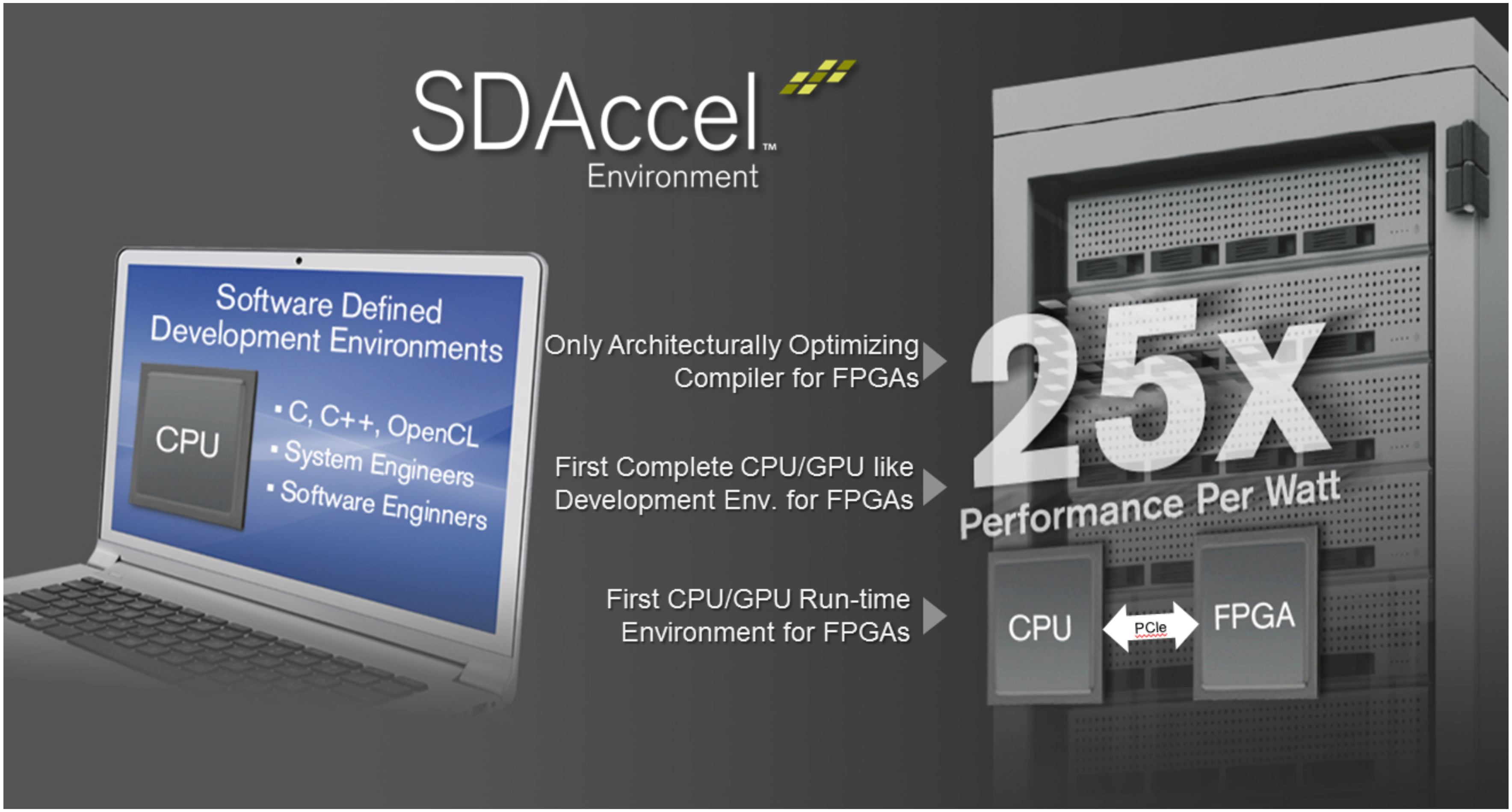
SDAccel

Software Centric View of FPGA Programming

OpenCL : Technical Goals

- Royalty Free standard for Parallel Programming
  - Shared IP zones
- Target Everything in a Heterogeneous Platform
  - CPU + GPU + ASIC accelerators + DSP + FPGA
- Cross-Vendor Functional Portability
  - No proprietary APIs + languages to learn
- Enable Architecture Specific Optimization
  - Refactor code to optimize
  - Vendor extensions

Introducing SDAccel



SDAccel™ Environment

Software Defined Development Environments

- C, C++, OpenCL
- System Engineers
- Software Enginners

Only Architecturally Optimizing Compiler for FPGAs

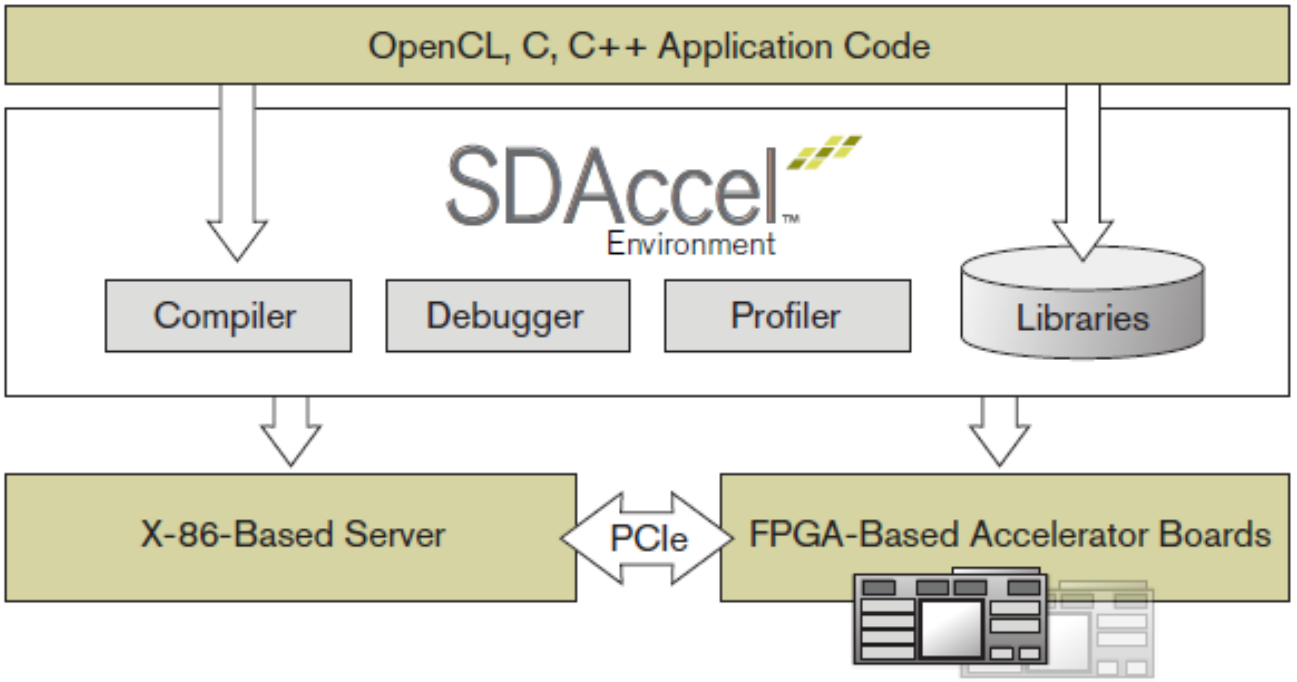
First Complete CPU/GPU like Development Env. for FPGAs

First CPU/GPU Run-time Environment for FPGAs

25x Performance Per Watt

Breakthrough Solution for Application Developers

SDAccel - CPU/GPU Development Experience on FPGAs



Libraries	Availability
OpenCL built-ins	Included
Video, DSP, Linear Algebra	Included
OpenCV BLAS	Provided by Auviz Systems

Readily Available Boards

ALPHA DATA

Virtex-7 OpenCL Platform

- Virtex-7 690T
- Half-length, low profile
- 2x 8GB DDR3 1600Mb/s, ECC
- Gen 3 x8 PCIe
- Up to 28.7 GFlops/W
- 2x SFP+ modules

http://www.alpha-data.com/pdfs/adm-pcie-7v3.pdf

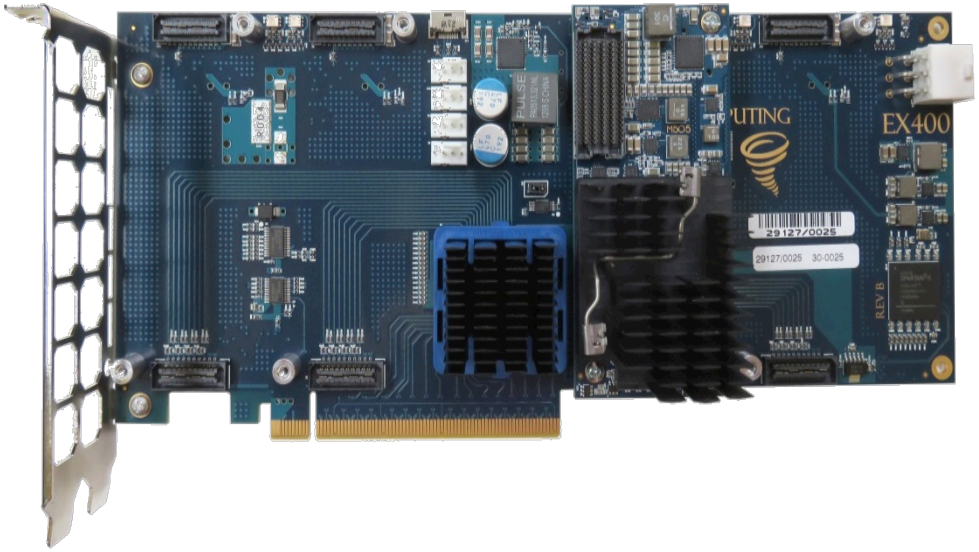
OpenCL COTS Platforms from Development to Production

picocomputing

now you can

Kintex-7 OpenCL Platform M-505-K325T

- Kintex-7 325T
- 8GB DDR3
- Gen 2 X8 PCI-e
- ISO 13485



http://picocomputing.com/products/embedded-modules/m-505-k325t-embedded-2

Scalable Customizable Acceleration

CONVEY
COMPUTER™

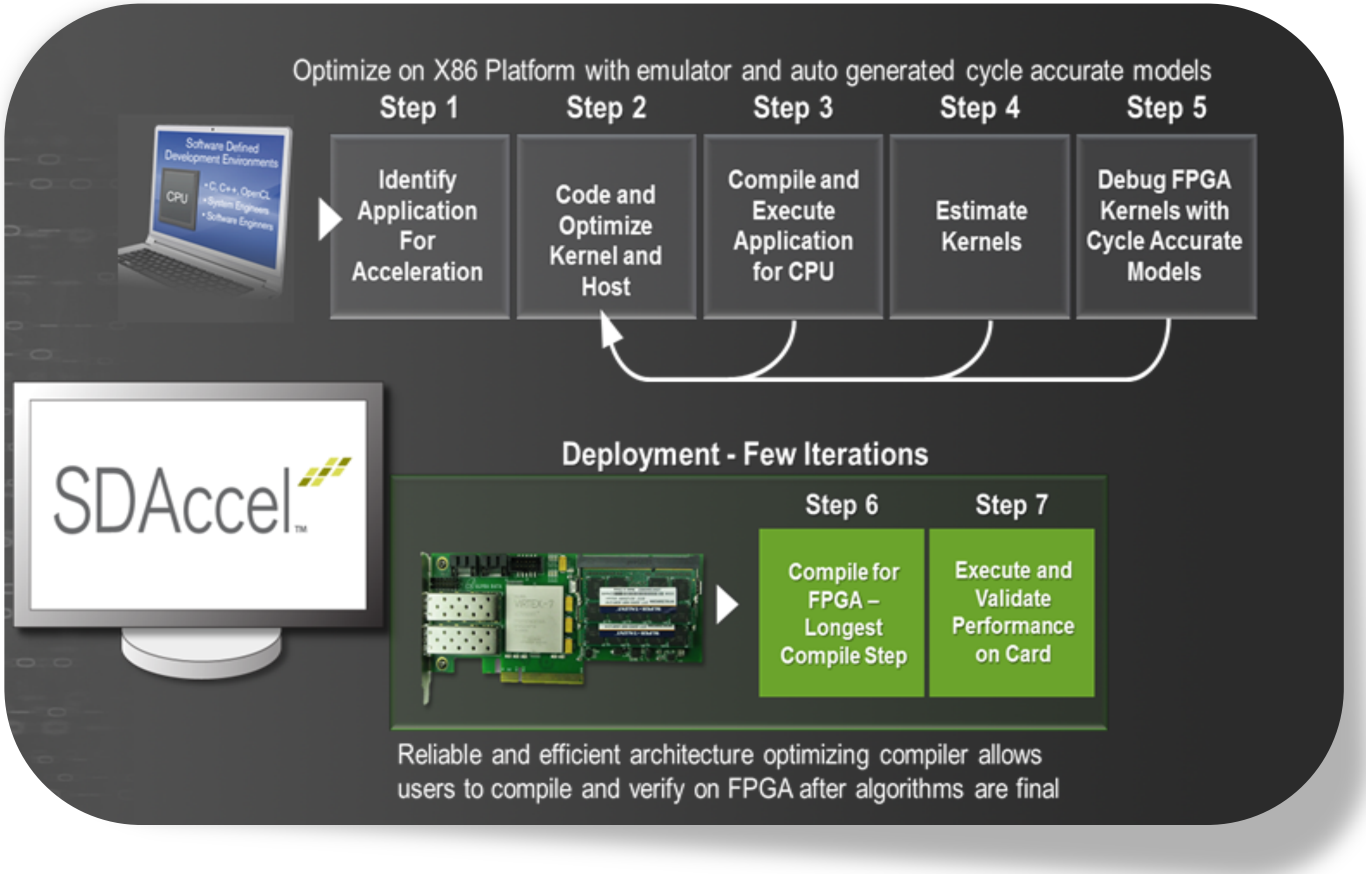
Virtex7 OpenCL Platform
Wolverine WX690/WX2000

- **WX690: V7 690T**
- **WX2000: V7 2000T**
- **16/32/64 GB DDR3**
- **Gen3 X16 PCI-e**

http://www.conveycomputer.com/products/wolverine/

Highest performance acceleration at the lowest power envelope

Accelerated OpenCL Programming and Deployment



Optimize on X86 Platform with emulator and auto generated cycle accurate models

Step 1: Identify Application For Acceleration

Step 2: Code and Optimize Kernel and Host

Step 3: Compile and Execute Application for CPU

Step 4: Estimate Kernels

Step 5: Debug FPGA Kernels with Cycle Accurate Models

Deployment - Few Iterations

Step 6: Compile for FPGA – Longest Compile Step

Step 7: Execute and Validate Performance on Card

Reliable and efficient architecture optimizing compiler allows users to compile and verify on FPGA after algorithms are final

CPU/GPU like programming environment

- ✓ X86 development and debug
- ✓ Cycle accurate simulation
- ✓ Platform acceleration

SDAccel Kernel Compiler for FPGAs

C, C++
OpenCL

Leveraging Vivado HLS for

- ✓ Efficiently Optimizes a Variety of Input Types
- ✓ Optimized Architectural Parallelizing and Pipelining
- ✓ Broadest Range of Compiler Optimization for Memory, Dataflow, & Loop Pipelining

SDAccel™ Environment

Performance and Resource Efficiency

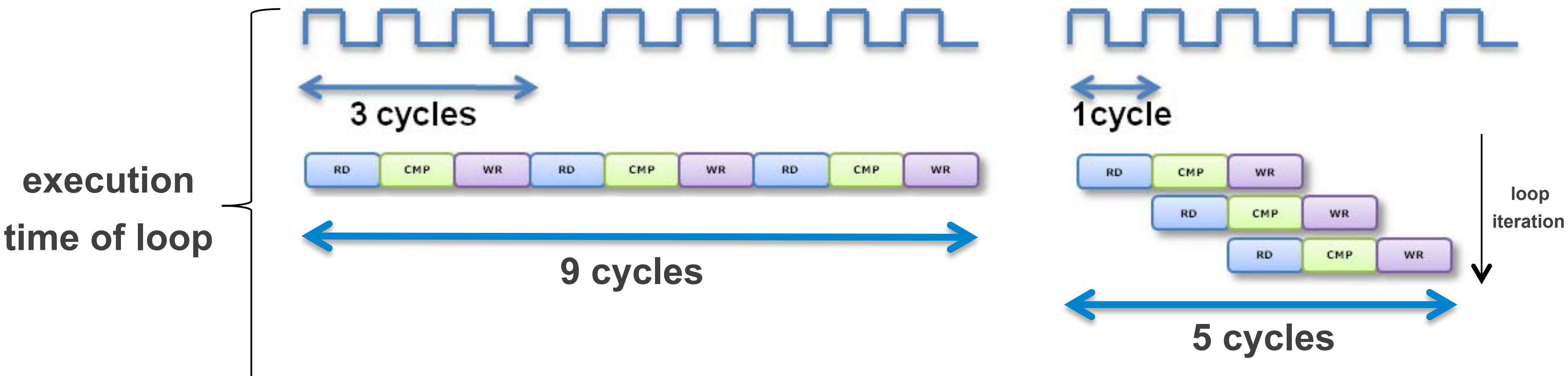
Pipelining

OpenCL C Kernel

```
kernel void
foo(...)
{
  __attribute__((xcl_pipeline_loop))
  for (int i=0; i<3; i++) {
    int idx = get_global_id(0)*3 + i;
    op_Read(idx);
    op_Compute(idx);
    op_Write(idx);
  }

}
```

C/C++ Kernel

```
void
foo(...)
{
  for (int i=0; i<3; i++) {
#pragma HLS pipeline
    int idx = get_global_id(0)*3 + i;
    op_Read(idx);
    op_Compute(idx);
    op_Write(idx);
  }

}
```



(A) Without Loop Pipelining

(B) With Loop Pipelining

Loop Unrolling

```
/* vector multiply */
kernel void
vmult(local int* a, local int* b, local int* c)
{
  int tid = get_global_id(0);

  __attribute__((unroll_loop_hint(2)))
  for (int i=0; i<4; i++) {
    ınt idx = tid*4 + i;
    a[idx] = b[idx] * c[idx];
  }
}
```

```
/* vector multiply */
kernel void
vmult(local int* a, local int* b, local int* c)
{
  int tid = get_global_id(0);

  __attribute__((unroll_loop_hint))
  for (int i=0; i<4; i++) {
    int idx = tid*4 + i;
    a[idx] = b[idx] * c[idx];
  }
}
```

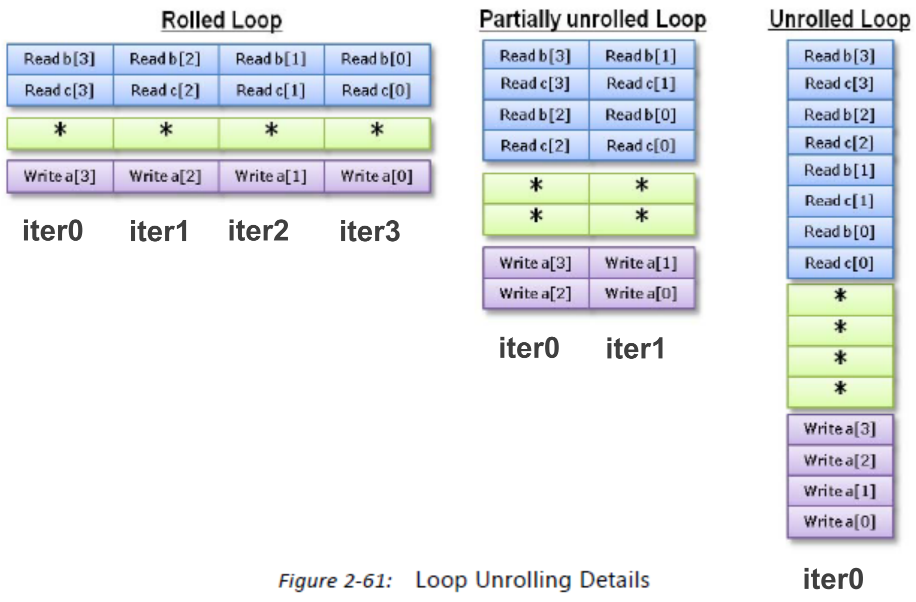


Figure 2-61: Loop Unrolling Details

- **Automatic with heuristics**
- **User directed**
  - OpenCL attribute
- **Expose more concurrency to the compiler's scheduler**
  - Similar to SIMD vectorization

Memory Specialization

OpenCL C Kernel

```
__local int buffer[16] ;
```

C/C++ Kernel

```
int buffer[16] ;
```

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

buffer

- **Buffer implemented in 1 physical memory**
- **Buffer can sustain 2 concurrent transactions**
- **Reading all values of buffer can take 8 clock cycles**

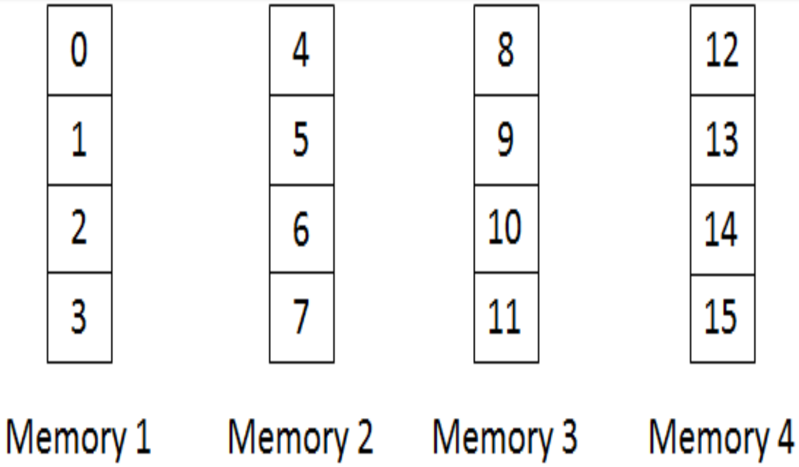
Memory Specialization

OpenCL C Kernel

```
__local int buffer[16]   __attribute__((xcl_array_partition(block,4,1)));
```

C/C++ Kernel

```
int buffer[16];
#pragma HLS ARRAY_PARTITION variable=buffer block factor=4 dim=1
```



- **Buffer implemented in 4 physical memories**
- **Buffer can sustain 8 concurrent transactions**
- **Compiler handles all address translations to physical memories**
- **Reading all values of buffer can take 2 clock cycles**

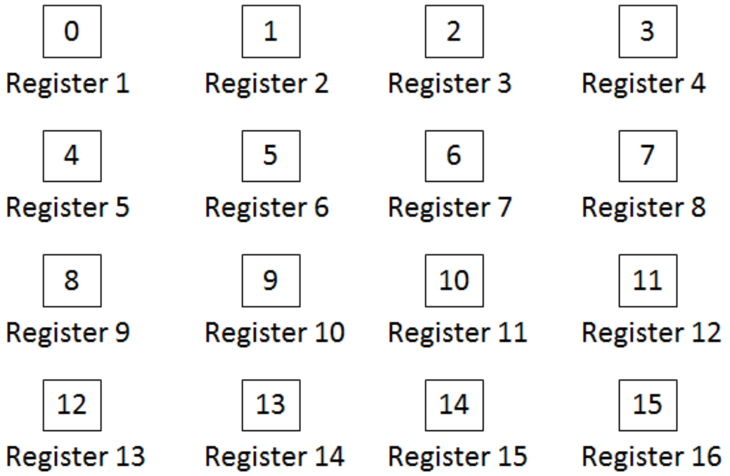
Memory Specialization

OpenCL C Kernel

```
__local int buffer[16]   __attribute__((xcl_array_partition(complete,1)));
```

C/C++ Kernel

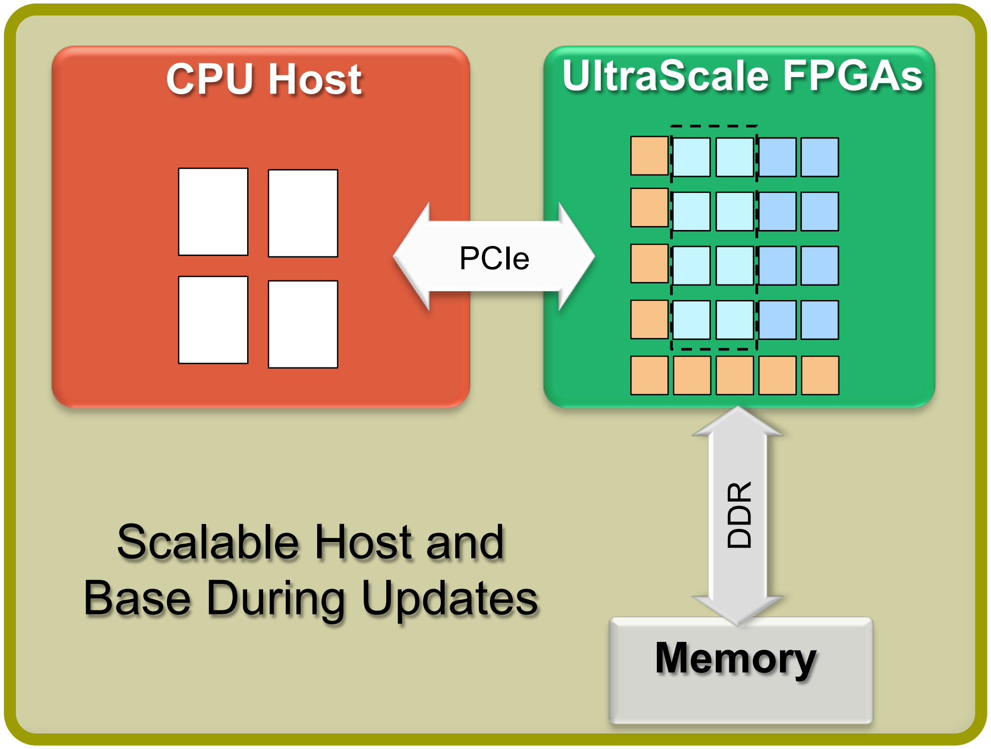
```
int buffer[16];
#pragma HLS ARRAY_PARTITION variable=buffer complete dim=1
```



0 Register 1	1 Register 2	2 Register 3	3 Register 4
4 Register 5	5 Register 6	6 Register 7	7 Register 8
8 Register 9	9 Register 10	10 Register 11	11 Register 12
12 Register 13	13 Register 14	14 Register 15	15 Register 16

- **Buffer implemented in 16 registers**
- **Buffer can sustain 16 concurrent transactions**
- **Compiler handles all address translations to physical memories**
- **Reading all values of buffer can take 1 clock cycles**

SDAccel Platform Model

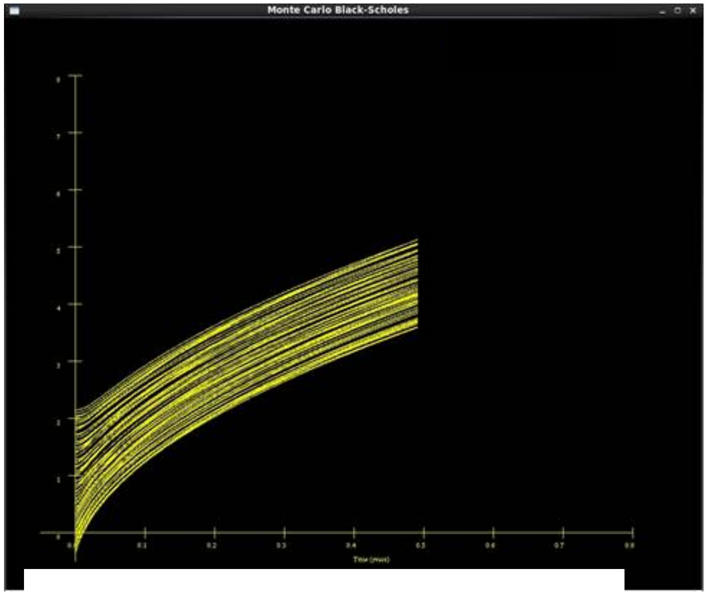


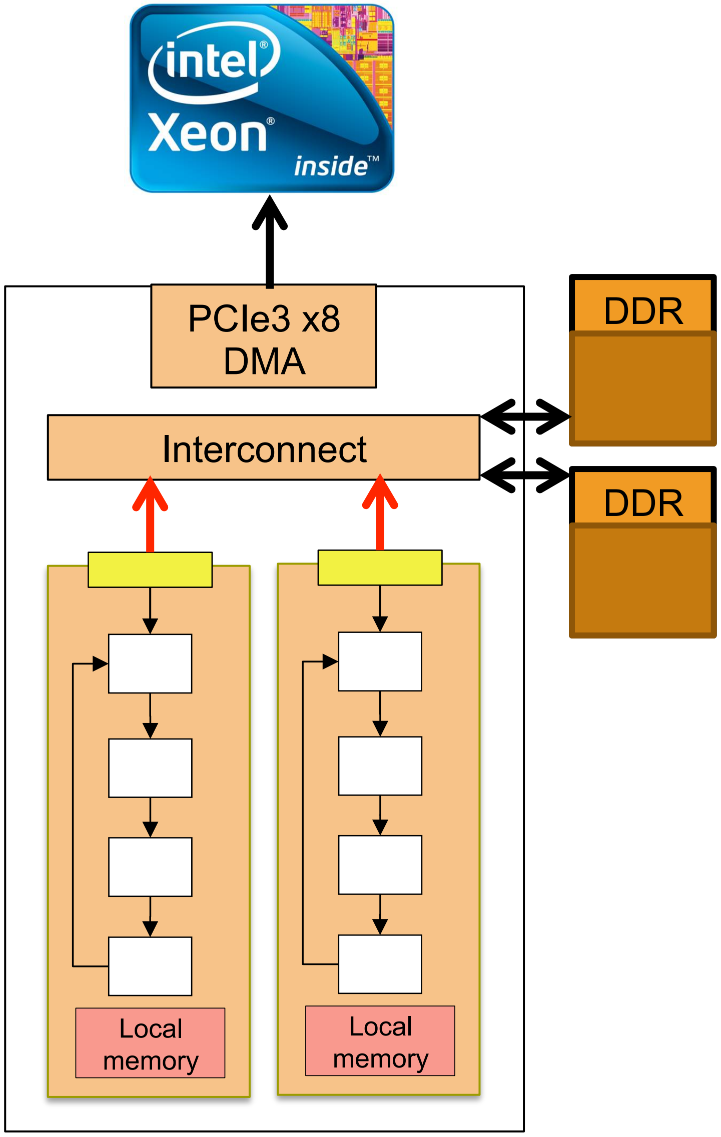
Software Runtime Experience

- ✓ On-demand loadable acceleration units
- ✓ Always on interfaces (Memory, Ethernet PCIe, Video)
- ✓ Minimizes compilation time

SDAccel Applications

Virtex-7 Monte-Carlo Options Pricing

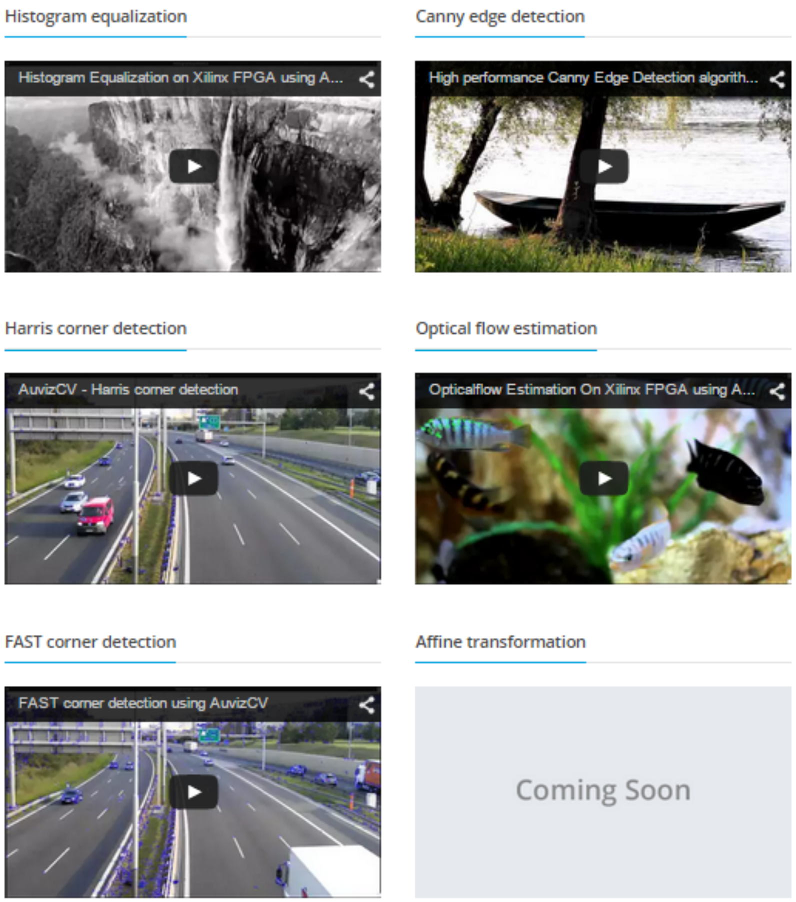




MC Black Scholes
OpenCL Compute Unit

Auviz Systems

http://auvizsystems.com/products/auvizcv/



Histogram equalization

Histogram Equalization on Xilinx FPGA using A...

Canny edge detection

High performance Canny Edge Detection algorith...

Harris corner detection

AuvizCV - Harris corner detection

Optical flow estimation

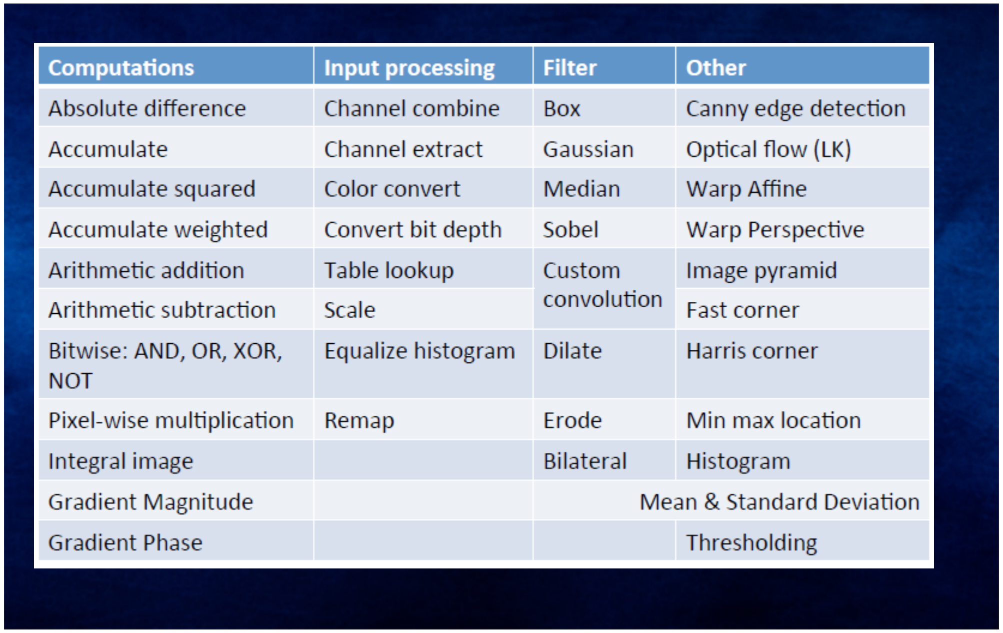
Opticalflow Estimation On Xilinx FPGA using A...

FAST corner detection

FAST corner detection using AuvizCV

Affine transformation

Coming Soon



Computations	Input processing	Filter	Other
Absolute difference	Channel combine	Box	Canny edge detection
Accumulate	Channel extract	Gaussian	Optical flow (LK)
Accumulate squared	Color convert	Median	Warp Affine
Accumulate weighted	Convert bit depth	Sobel	Warp Perspective
Arithmetic addition	Table lookup	Custom convolution	Image pyramid
Arithmetic subtraction	Scale		Fast corner
Bitwise: AND, OR, XOR, NOT	Equalize histogram	Dilate	Harris corner
Pixel-wise multiplication	Remap	Erode	Min max location
Integral image		Bilateral	Histogram
Gradient Magnitude			Mean & Standard Deviation
Gradient Phase			Thresholding

FPGA Optimized Libraries

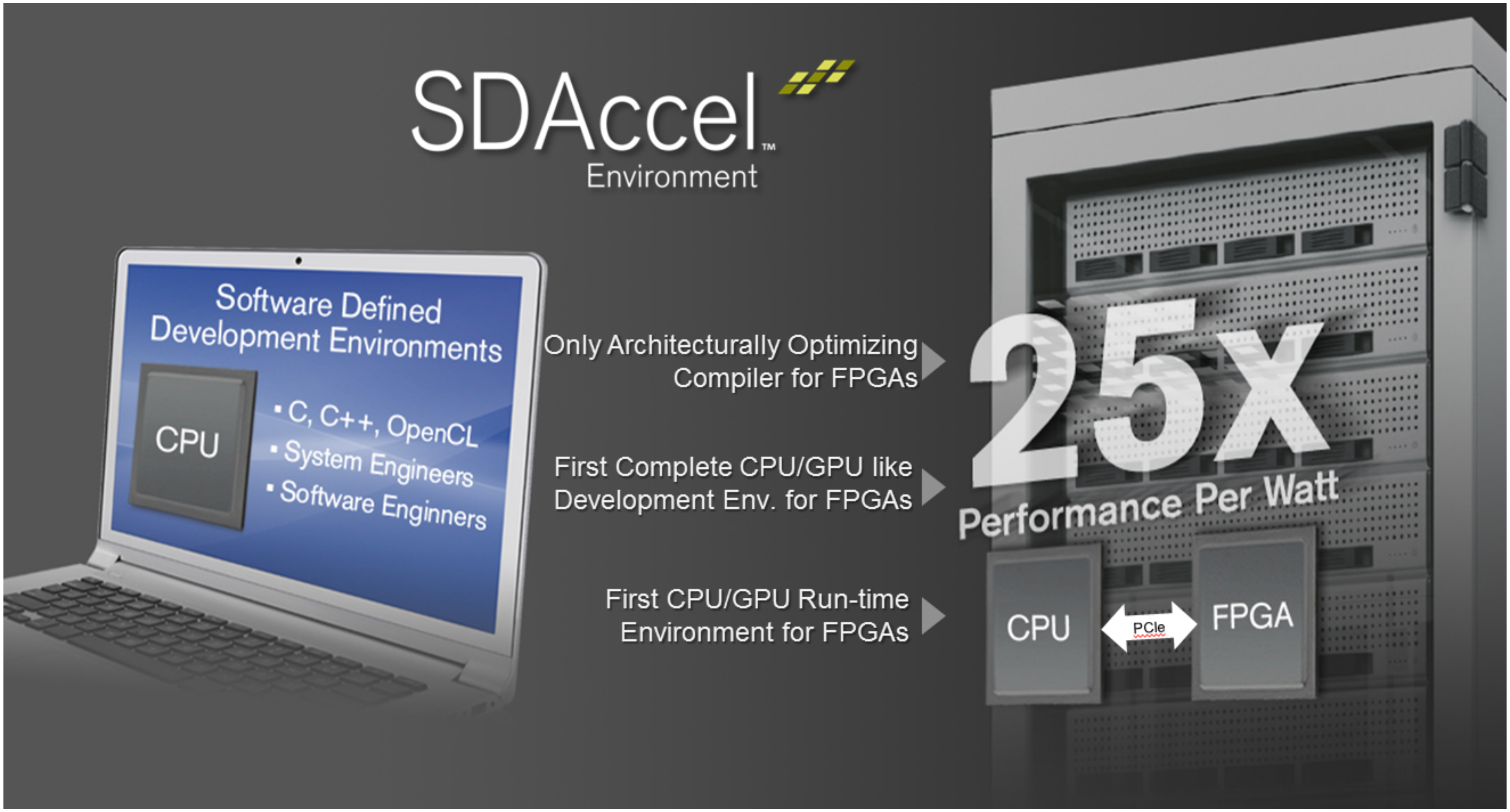
SDAccel in Action

SDAccel Performance/Watt Advantage

Application	Metric	SDAccel with AuvizCV library	Nvidia K20 with CUDA	SDAccel Advantage
HD Sobel Filter	Frames/watt	80	11	7x
HD Image Downscaling	Frames/watt	36	5	7x

Data from Auviz Systems

Start Designing with SDAccel Today



Thank You